Derin Öğrenmeye Dalış

Release 0.17.5

Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola

Nov 29, 2022

Contents

Önsöz

Sadece birkaç yıl öncesi kadar, büyük şirket ve girişimlerde akıllı ürün ve hizmetler geliştiren ve derin öğrenme uzmanlarından oluşan birimler yoktu. Biz bu alana girdiğimizde, makine öğrenmesi günlük gazetelerde manşetlere çıkmıyordu. Ebeveynlerimizin, bırakın onu neden tıpta veya hukukta bir kariyere tercih ettiğimizi, makine öğrenmesinin ne olduğuna dair hiçbir fikri yoktu. Makine öğrenmesi endüstriyel önemi gerçek dünyada, konuşma tanıma ve bilgisayarlı görme dahil, dar uygulama alanlı ileriye dönük bir akademik disiplindi. Dahası, bu uygulamaların birçoğu o kadar çok alan bilgisi gerektiriyordu ki, makine öğrenmesi küçük bir bileşeni olan tamamen ayrı alanlar olarak kabul ediliyorlardı. O zamanlar sinir ağları, bu kitapta odaklandığımız derin öğrenme yöntemlerinin ataları, genel olarak modası geçmiş araçlar olarak görülüyordu.

Sadece son beş yılda, derin öğrenme dünyayı şaşırttı ve bilgisayarla görme, doğal dil işleme, otomatik konuşma tanıma, pekiştirici öğrenme ve biyomedikal bilişim gibi farklı alanlarda hızlı ilerleme sağladı. Ayrıca, derin öğrenmenin pek çok pratik ilgi gerektiren işteki başarısı, teorik makine öğrenmesi ve istatistikteki gelişmeleri bile hızlandırdı. Elimizdeki bu ilerlemelerle, artık kendilerini her zamankinden daha fazla otonomlukla (ve bazı şirketlerin sizi inandırdığından daha az otonomlukla) kullanan otomobiller, standart e-postaları otomatik olarak hazırlayıp insanların devasa büyüklükteki e-posta kutularından kurtulmalarını sağlayan akıllı yanıt sistemleri ve go gibi masa oyunlarında dünyanın en iyi insanlarına hükmeden yazılımlar -ki bir zamanlar onlarca yıl uzakta bir özellik olarak tahmin ediliyordu- üretebiliyoruz. Bu araçlar endüstri ve toplum üzerinde şimdiden geniş etkiler yaratıyor, filmlerin yapılma şeklini değiştiriyor, hastalıklar teşhis ediyor ve temel bilimlerde, astrofizikten biyolojiye kadar, büyüyen bir rol oynuyor.

Bu Kitap Hakkında

Bu kitap, derin öğrenmeyi ulaşılabilir yapma girişimimizi temsil eder, size *kavramları*, *bağlamı* ve *kodu* öğretir.

Kod, Matematik ve HTML'yi Bir Araya Getirme

Herhangi bir bilgi işlem teknolojisinin tam etkinliğine ulaşması için, iyi anlaşılmış, iyi belgelenmiş, olgun ve güncellenen araçlarla desteklenmesi gerekir. Anahtar fikirler açıkça damıtılmalı ve yeni uygulama geliştiricileri güncel hale getirmek için gereken işi öğrenme süresi en aza indirilmelidir. Olgun kütüphaneler ortak görevleri otomatikleştirmeli ve örnek kod uygulama geliştiricilerin ortak uygulamaları ihtiyaçlarına göre değiştirmesini ve yeni özellikler eklemesini kolaylaştırmalıdır. Dinamik web uygulamalarını örnek olarak alalım. 1990'larda başarılı veritabanı temelli web uygulamaları geliştiren, Amazon gibi, çok sayıda şirket olmasına rağmen, bu teknolojinin yaratıcı girişimcilere yardım etme potansiyeli, güçlü ve iyi belgelenmiş altyapıların geliştirilmesi sayesinde son on yılda çok daha büyük oranda gerçekleşti.

Derin öğrenmenin potansiyelini test ederken, çeşitli disiplinler bir araya geldiği için zorluklarla karşılaşabilirsiniz. Derin öğrenmeyi uygulamak aynı anda (i) belirli bir problemi belirli bir şekilde çözme motivasyonlarını (ii) belli bir modelin matematiksel formu (iii) modellerin verilere uyumu (fitting) için kullanılan optimizasyon algoritmalarını (iv) modellerimizin görünmeyen verilere genellenmesini ne zaman beklememiz gerektiğini bize söyleyen istatistiksel ilkelerini ve aslında genelleştirilmiş olduklarını doğrulamak için pratik yöntemlerini (v) modelleri verimli bir şekilde eğitmek için gerekli mühendisliği, sayısal hesaplamanın gizli tuzaklarında gezinme ve mevcut donanımdan en iyi şekilde yararlanma yöntemlerini anlamayı gerektirir. Sorunları formüle etmek için gerekli eleştirel düşünme becerilerini, onları çözmek için gereken matematiği ve bu çözümleri uygulamak için kullanılan yazılım araçlarını tek bir yerde öğretebilmek oldukça zor. Bu kitaptaki amacımız istekli uygulayıcılara hız kazandıran bütünleşik bir kaynak sunmaktır.

Bu kitap projesine başladığımızda, aşağıdaki özelliklerin hepsini bir arada barındıran hiçbir kaynak yoktu: (i) Güncel olma, (ii) modern makine öğrenmesinin tamamını geniş bir teknik derinlikle kapsama, (iii) ilgi çekici bir ders kitabından beklenen kaliteyi uygulamalı derslerde bulmayı beklediğiniz temiz çalıştırılabilir kod ile içiçe serpiştirilmiş olarak sunma.

Belirli bir derin öğrenme çerçevesinin nasıl kullanıldığını (örneğin, TensorFlow'daki matrislerle temel sayısal hesaplama) veya belirli tekniklerin nasıl uygulandığını (ör. LeNet, AlexNet, ResNets, vb. için kod parçaları) gösteren çeşitli blog yayınlarına ve GitHub depolarına dağılmış birçok kod örneği bulduk. Bu örnekler genellikle belirli bir yaklaşımı *nasıl* uygulayacağına odaklanmakta, ancak bazı algoritmik kararların *neden* verildiği tartışmasını dışlamaktaydı. Ara sıra bazı etkileşimli kaynaklar yalnızca derin öğrenmedeki belirli bir konuyu ele almak için ortaya çıkmış olsa da, örneğin Distill[1] web sitesinde veya kişisel bloglarda yayınlanan ilgi çekici blog yayınları, genellikle ilişkili koddan yoksundu. Öte yandan, derin öğrenmenin temelleri hakkında kapsamlı bir inceleme sunan (Goodfellow *et al.*, 2016) gibi birkaç derin öğrenme ders kitabı ortaya çıkmış olsa da, bu kaynaklar, kavramların kodda gerçekleştirilmesine ait açıklamalar ile okuyucuların bunları nasıl uygulayabileceğini ilişkilendirmesi konusunda bazen bilgisiz bırakıyor. Ayrıca, birçok kaynak ticari kurs sağlayıcılarının ödeme duvarlarının arkasında gizlenmiştir.

Biz yola çıkarken (i) herkesin erişimine açık olan, (ii) hakiki bir uygulamalı makine öğrenmesi bilim insanı olma yolunda başlangıç noktası sağlamak için yeterli teknik derinliği sunan, (iii) okuyuculara problemleri pratikte *nasıl* çözecebileceklerini gösteren çalıştırılabilir kodu içeren, (iv) hem topluluk hem de bizim tarafımızdan hızlı güncellemelere izin veren, (v) teknik detayların etkileşimli tartışılması ve soruların cevaplanması için bir forum[2] ile tamamlanan bir kaynak oluşturmayı hedefledik.

Bu hedefler genellikle çatışıyordu. Denklemler, teoremler ve alıntılar LaTeX'te en iyi şekilde düzenlenebilir ve yönetilebilir. Kod en iyi Python'da açıklanır. Web sayfaları için HTML ve JavaScript idealdir. Ayrıca içeriğin hem çalıştırılabilir kod, hem fiziksel bir kitap, hem indirilebilir bir PDF, hem de internette bir web sitesi olarak erişilebilir olmasını istiyoruz. Şu anda bu taleplere tam uygun hiçbir araç ve iş akışı yok, bu yüzden kendimiz bir araya getirmek zorunda kaldık. Yaklaşımımızı ayrıntılı olarak şurada açıklıyoruz Section 19.6. Kaynağı paylaşmak ve düzenlemelere izin vermek için GitHub'a, kod, denklemler ve metin karıştırmak için Jupyter not defterlerine, çoklu çıktılar oluşturmak için bir oluşturma motoru olan Sphinx'e ve forum için Discourse'a karar verdik. Sistemimiz henüz mükemmel olmasa da, bu seçenekler farklı hedefler arasında iyi bir uzlaşma sağlamaktadır. Bunun böyle bir tümleşik iş akışı kullanılarak yayınlanan ilk kitap olabileceğine inanıyoruz.

[1] http://distill.pub
[2] http://discuss.d2l.ai

Yaparak Öğrenmek

Pek çok ders kitabı, her birini ayrıntılı bir şekilde kapsayarak kavramları art arda sunar. Örneğin, Chris Bishop'un mükemmel ders kitabı (Bishop, 2006), her konuyu o kadar titizlikle öğretir ki, doğrusal regresyon konusunda bile hatırı sayılır bir çalışma gerektirir. Uzmanlar bu kitabı tam olarak bu titizliğinden dolayı severler ancak, detay seviyesinin fazlalığından ötürü kitabın kullanışlılığı yeni başlayanlar için azdır.

Bu kitapta, çoğu kavramı *tam zamanında* öğreteceğiz. Başka bir deyişle, bazı pratik sonlara ulaşmak için gerekli oldukları anda kavramları öğreneceksiniz. Başlangıçta doğrusal cebir ve olasılık gibi temelleri öğretmek için biraz zamanınızı alırken, daha özel olasılık dağılımlarına girmeden önce ilk modelinizi eğitmenin zevkini tatmanızı istiyoruz.

Temel matematiksel altyapıya hızlı giriş yapmanızı sağlayan baştaki birkaç bölüm dışında, sonraki her bölüm hem makul sayıda yeni kavramı tanıtır hem de bağımsız veri kümeleri kullanarak tek başına çalışan örnekler görmenizi sağlar. Bu durum organizasyonel bir zorluğa da yol açıyor çünkü bazı modeller mantıksal olarak tek bir not defterinde gruplandırılabilirken bazı fikirler en iyi şekilde birkaç model arka arkaya uygulanarak öğretilebiliyor. Öte yandan, *bir çalışan örnek, bir not defteri* yaklaşımını benimsememizin büyük bir avantajı var: Kodumuzu kullanarak kendi araştırma projelerinizi hızlıca başlatabilirsiniz. Sadece bir Jupyter not defterini kopyalayın ve değiştirmeye başlayın.

Çalıştırılabilir kodu gerektiğinde arka plan materyalleri ile zenginleştireceğiz. Genel olarak, araçları bütün detaylarıyla açıklamadan önce nasıl kullanıldığını göstermeyi tercih edeceğiz. Örneğin, neden yararlı olduğunu veya neden işe yaradığını tam olarak açıklamadan önce *rastgele eğim inişini (stochastic gradient descent-SGD)* doğrudan kullanacağız. Bu, okuyucunun bazı kararlarda bize güvenmesi pahasına, sorunları hızlı bir şekilde çözmek için gerekli ekipmana hızlıca ulaşmasına yardımcı olur.

Bu kitap derin öğrenme kavramlarını sıfırdan öğretecek. Bazen, derin öğrenme çerçevelerinin gelişmiş soyutlamaları ile tipik olarak kullanıcıdan gizlenen modeller hakkındaki ince detayları irdelemek istiyoruz. Özellikle temel eğitimlerde, belirli bir katmanda veya eniyileyicide (optimizer) gerçekleşen her şeyi anlamanızı istediğimizde örneğin iki versiyonunu sunacağız: Bir tanesi her şeyi sıfırdan uyguladığımız, sadece NumPy arayüzüne ve otomatik türev almaya dayananı ve diğeri ise derin öğrenme çerçevelerinin üst-düzey API'lerini (uygulama programlama arayüzleri) kullanarak kısa kodunu yazdığımız daha pratik bir örneği. Size bazı bileşenlerin nasıl çalıştığını öğrettikten sonra, üst-düzey API'leri sonraki derslerde kullanacağız.

İçerik ve Yapı

Kitap kabaca, ön hazırlıklara, derin öğrenme tekniklerine ve gerçek sistemlere ve uygulamalara odaklı ileri düzey konulara odaklanan üç bölüme ayrılabilir (Fig. 1).

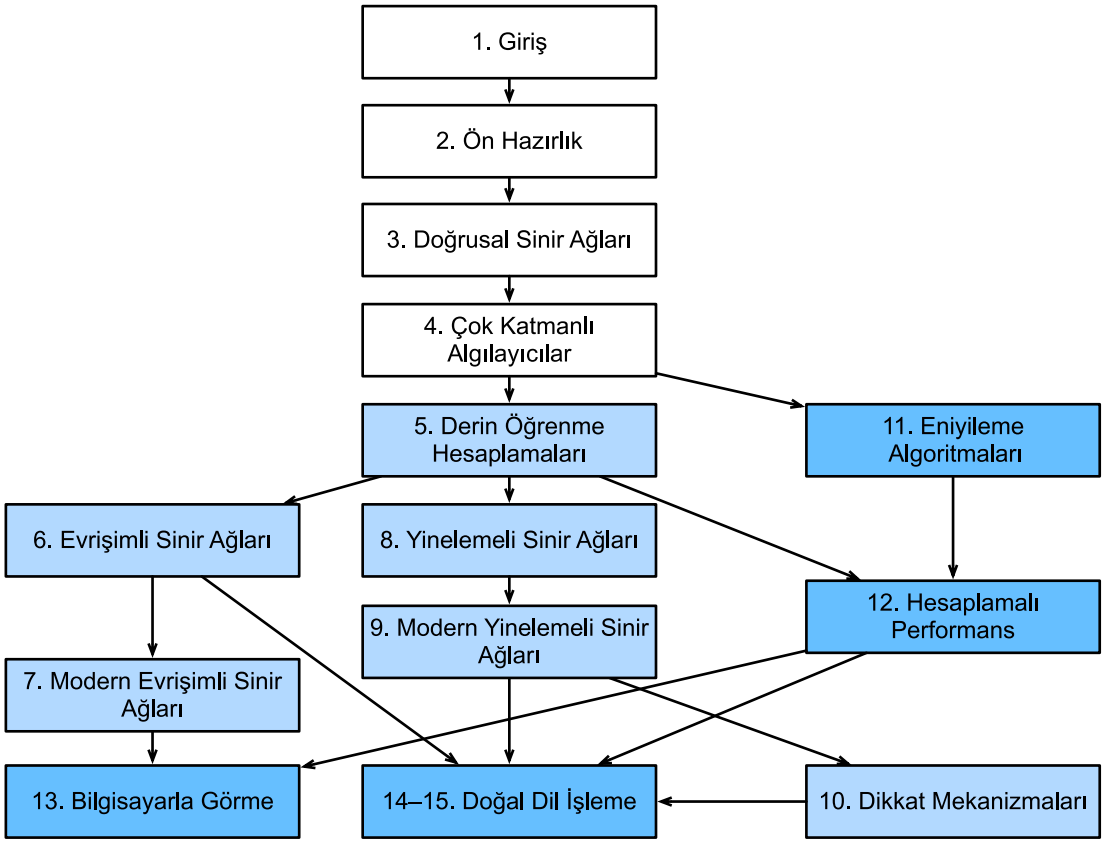


Fig. 1: Kitabın yapısı

- İlk bölüm temelleri ve ön bilgileri içerir. Chapter 1 derin öğrenmeye girişi içerir. Daha sonra, Chapter 2 içinde hızlı bir şekilde verilerin nasıl saklanacağı ve işleneceği ve temel kavramlara dayalı çeşitli sayısal işlemlerin nasıl uygulanacağı gibi derin öğrenmede gereken cebir, matematik ve olasılık önkoşullarını size sunuyoruz. Chapter 3 ve Chapter 4, bağlanım (regresyon) ve sınıflandırma, doğrusal modeller ve çok katmanlı algılayıcılar (multilayer perceptrons), aşırı öğrenme (overfitting) ve düzenlileştirme (regularization) dahil olmak üzere derin öğrenmedeki en temel kavram ve teknikleri kapsar.
- Sonraki beş bölüm modern derin öğrenme tekniklerine odaklanmaktadır. Chapter 5 derin öğrenme hesaplamalarının çeşitli temel bileşenlerini açıklar ve daha sonra daha karmaşık modeller uygulamamız için gereken zemini hazırlar. Daha sonra, Chapter 6 ve Chapter 7 içinde, çoğu modern bilgisayarla görme sisteminin omurgasını oluşturan güçlü araçlar olan evrişimli sinir ağlarını (CNN'ler) sunuyoruz. Benzer şekilde, Chapter 8 ve Chapter 9, verilerde sıralı (örneğin, zamansal) yapıdan yararlanan ve doğal dil işleme ve zaman serisi tahmini için yaygın olarak kullanılan modeller olan yinelemeli sinir ağlarını (RNN'ler) tanıtır. Chapter 10 içinde, çoğu doğal dil işleme görevi için baskın mimari olarak RNN'lerin yerini alan, dikkat (attention) mekanizmalarına dayanan nispeten yeni bir model sınıfını tanıtıyoruz. Bu bölümler, derin öğrenme uygulayıcıları tarafından yaygın olarak kullanılan en güçlü ve genel araçlar hakkında sizi bilgilendirecektir.
- Üçüncü bölüm ölçeklenebilirliği, verimliliği ve uygulamaları tartışmaktadır. İlk olarak

Chapter 11 içinde, derin öğrenme modellerini eğitmek için kullanılan birkaç yaygın eniyileme algoritmasını tartışıyoruz. Bir sonraki bölüm Chapter 12, derin öğrenme kodunuzun hesaplama performansını etkileyen birkaç anahtar faktörü inceler. Chapter 13 içinde, bilgisayarla görmede derin öğrenmenin başlıca uygulamalarını göstereceğiz. Chapter 14 ve Chapter 15 içinde de dil gösterimi modellerinin nasıl önceden eğitileceğini ve doğal dil işleme görevlerine nasıl uygulanacağını bulabilirsiniz.

Kod

Bu kitabın çoğu bölümünde yürütülebilir kod bulunur. Bazı sezgilerin en iyi şekilde deneme yanılma yoluyla, kodu küçük şekillerde değiştirerek ve sonuçları gözlemleyerek geliştirildiğine inanıyoruz. İdeal olarak, zarif bir matematiksel teorisi, istenen sonucu elde etmek için kodumuzu tam olarak nasıl değiştireceğimizi söyleyebilir. Bununla birlikte, bugün derin öğrenme uygulayıcıları, çoğu zaman, hiçbir inandırıcı teorinin kesin rehberlik sağlayamayacağı yerlere gitmek zorundadır. En iyi çabalarımıza rağmen, hem bu modelleri karakterize etmek için matematik çok zor olabileceğinden hem de bu konularda ciddi araştırmaların henüz yeni yeni hızlanmaya başlamasından dolayı, çeşitli tekniklerin etkinliği için biçimsel açıklamalar hala eksiktir. Derin öğrenme teorisi ilerledikçe, bu kitabın gelecekteki baskılarının, şu anda mevcut olanları gölgede bırakan içgörüler sağlayabileceğini umuyoruz.

Gereksiz tekrarlardan kaçınmak için, en sık içe aktarılan (import) ve atıfta bulunulan işlev ve sınıflarımızdan bazılarını `d2l` paketine yerleştirdik. Daha sonra `d2l` paketi aracılığıyla erişilecek olan bir işlev, sınıf veya içe aktarım ifadeleri koleksiyonu gibi bir kod bloğunu belirtmek için, onu `#@save` ile işaretleyeceğiz. Bu işlevlere ve sınıflara ayrıntılı bir genel bakışı Section 19.7 içinde sunuyoruz. `d2l` paketi hafiftir ve yalnızca aşağıdaki bağımlılıkları gerektirir:

```
#@save
import collections
import hashlib
import math
import os
import random
import re
import shutil
import sys
import tarfile
import time
import zipfile
from collections import defaultdict
import pandas as pd
import requests
from IPython import display
from matplotlib import pyplot as plt
from matplotlib_inline import backend_inline

d2l = sys.modules[__name__]
```

Bu kitaptaki kodun çoğu derin öğrenme araştırma topluluğu tarafından çoşkunca benimsenmiş son derece popüler bir açık kaynaklı çerçeve olan PyTorch'a dayanmaktadır. Bu kitaptaki tüm kodlar en yeni kararlı PyTorch sürümünün altında testlerden geçmiştir. Ancak, derin öğrenmenin hızla gelişmesi nedeniyle, *basılı sürümündeki* bazı kodlar PyTorch'un gelecekteki sürümlerinde düzgün çalışmayabilir. Ancak, çevrimiçi sürümü güncel tutmayı planlıyoruz. Böyle bir sorunla karşılaşırsanız, kodunuzu ve çalışma zamanı ortamınızı güncellemek için lütfen şuraya başvurun

Kurulum (page 9).

Modülleri PyTorch'tan şu şekilde içe aktarıyoruz.

```
#@save
import numpy as np
import torch
import torchvision
from PIL import Image
from torch import nn
from torch.nn import functional as F
from torch.utils import data
from torchvision import transforms
```

Hedef Kitle

Bu kitap derin öğrenme pratik tekniklerini sağlam bir şekilde kavramak isteyen öğrenciler (lisans veya lisansüstü), mühendisler ve araştırmacılar içindir. Her kavramı sıfırdan açıkladığımız için, derin öğrenme veya makine öğrenmesinde geçmiş bir birikim gerekmez. Derin öğrenme yöntemlerini tam olarak açıklamak biraz matematik ve programlama gerektirir, ancak yeter miktarda doğrusal cebir, matematik, olasılık ve Python programlama dahil bazı temel bilgilerle geldiğinizi varsayacağız. Eğer temelleri unuttuysanız, Ek'te (Apendiks), bu kitapta yer alan matematiğin çoğu hakkında bir bilgi tazeleyici sağlıyoruz. Çoğu zaman, matematiksel titizlik yerine sezgiye ve fikirlere öncelik vereceğiz.

Eğer bu temelleri bu kitabı anlamada gerekli öngereksinimlerden öteye genişletmek isterseniz, sizlere müthiş kitaplar önerebiliriz: Bela Bollobas'ın Doğrusal Analizi (Bollobás, 1999), doğrusal cebiri ve fonksiyonel analizi çok derinlemesine inceler. İstatistiğin Tamamı (Wasserman, 2013), istatistiğe harika bir giriş sağlar. Joe Blitzstein'ın kitapları[3] ve dersleri[4] olasılık ve çıkarsama üzerine pedagojik cevherlerdir. Python'u daha önce kullanmadıysanız, bu Python eğitimini[5] incelemek isteyebilirsiniz.

[3] https://www.amazon.com/Introduction-Probability-Chapman-Statistical-Science/dp/1138369918
[4] https://projects.iq.harvard.edu/stat110/home
[5] http://learnpython.org/

Forum

Bu kitapla ilgili olarak bir tartışma forumu başlattık, discuss.d2l.ai[6] adresinden ulaşabilirsiniz. Kitabın herhangi bir bölümü hakkında sorularınız olduğunda, ilgili tartışma sayfasına ait bağlantıyı her defter dosyasının sonunda bulabilirsiniz.

Teşekkürler

Hem İngilizce, hem Çince, hem de Türkçe taslaklar için yüzlerce katılımcıya kendimizi borçlu hissediyoruz. İçeriğin geliştirilmesine yardımcı oldular ve değerli geri bildirimler sundular. Özellikle, bu İngilizce taslağa katkıda bulunan herkese, onu herkes için daha iyi hale getirdikleri için teşekkür ediyoruz. GitHub kimliklerini veya isimleri (belirli bir sıra olmadan) şöyle sıralıyoruz: alxnorden, avinashingit, bowen0701, brettkoonce, Chaitanya Prakash Bapat, cryptonaut, Davide Fiocco, edgarroman, gkutiel, John Mitro, Liang Pu, Rahul Agarwal, Mohamed Ali Jamaoui, Michael (Stu) Stewart, Mike Müller, NRauschmayr, Prakhar Srivastav, sad-, sfermigier, Sheng Zha, sundeepteki, topecongiro, tpdi, vermicelli, Vishaal Kapoor, Vishwesh Ravi Shrimali, YaYaB, Yuhong Chen, Evgeniy Smirnov, lgov, Simon Corston-Oliver, Igor Dzreyev, Ha Nguyen, pmuens, Andrei Lukovenko, senorcinco, vfdev-5, dsweet, Mohammad Mahdi Rahimi, Abhishek Gupta, uwsd, DomKM, Lisa Oakley, Bowen Li, Aarush Ahuja, Prasanth Buddareddygari, brianhendee, mani2106, mtn, lkevinzc, caojilin, Lakshya, Fiete Lüer, Surbhi Vijayvargeeya, Muhyun Kim, dennismalmgren, adursun, Anirudh Dagar, liqingnz, Pedro Larroy, lgov, ati-ozgur, Jun Wu, Matthias Blume, Lin Yuan, geogunow, Josh Gardner, Maximilian Böther, Rakib Islam, Leonard Lausen, Abhinav Upadhyay, rongruosong, Steve Sedlmeyer, Ruslan Baratov, Rafael Schlatter, liusy182, Giannis Pappas, ati-ozgur, qbaza, dchoi77, Adam Gerson, Phuc Le, Mark Atwood, christabella, vn09, Haibin Lin, jjangga0214, RichyChen, noelo, hansent, Giel Dops, dvincent1337, WhiteD3vil, Peter Kulits, codypenta, joseppinilla, ahmaurya, karolszk, heytitle, Peter Goetz, rigtorp, Tiep Vu, sfilip, mlxd, Kale-ab Tessera, Sanjar Adilov, MatteoFerrara, hsneto, Katarzyna Biesialska, Gregory Bruss, Duy–Thanh Doan, paulaurel, graytowne, Duc Pham, sl7423, Jaedong Hwang, Yida Wang, cys4, clhm, Jean Kaddour, austinmw, trebeljahr, tbaums, Cuong V. Nguyen, pavelkomarov, vzlamal, NotAnotherSystem, J-Arun-Mani, jancio, eldarkurtic, the-great-shazbot, doctorcolossus, gducharme, cclauss, Daniel-Mietchen, hoonose, biagiom, abhinavsp0730, jonathanhrandall, ysraell, Nodar Okroshiashvili, UgurKap, Jiyang Kang, StevenJokes, Tomer Kaftan, liweiwp, netyster, ypandya, NishantTharani, heiligerl, SportsTHU, Hoa Nguyen, manuel-arno-korfmann-webentwicklung, aterzis-personal, nxby, Xiaoting He, Josiah Yoder, mathresearch, mzz2017, jroberayalas, iluu, ghejc, BSharmi, vkramdev, simonwardjones, LakshKD, TalNeoran, djliden, Nikhil95, Oren Barkan, guoweis, haozhu233, pratikhack, Yue Ying, tayfununal, steinsag, charleybeller, Andrew Lumsdaine, Jiekui Zhang, Deepak Pathak, Florian Donhauser, Tim Gates, Adriaan Tijsseling, Ron Medina, Gaurav Saha, Murat Semerci, Lei Mao, Levi McClenny, Joshua Broyde, jake221, jonbally, zyhazwraith, Brian Pulfer, Nick Tomasino, Lefan Zhang, Hongshen Yang, Vinney Cavallo, yuntai, Yuanxiang Zhu, amarazov, pasricha, Ben Greenawald, Shivam Upadhyay, Quanshangze Du, Biswajit Sahoo, Parthe Pandit, Ishan Kumar, HomunculusK, Lane Schwartz, varadgunjal, Jason Wiener, Armin Gholampoor, Shreshtha13, eigen-arnav, Hyeonggyu Kim, EmilyOng, Bálint Mucsányi, Chase DuBois.

Türkçe çevirisindeki katkılarından dolayı Murat Semerci, Barış Yaşin ve Emre Kaya'ya teşekkür ediyoruz.

[6] https://discuss.d2l.ai/

Amazon Web Services'e, özellikle Swami Sivasubramanian, Peter DeSantis, Adam Selipsky ve Andrew Jassy'ye bu kitabı yazma konusundaki cömert desteklerinden dolayı teşekkür ediyoruz. Yeterli zaman, kaynaklar, meslektaşlarla tartışmalar ve sürekli teşvik olmasaydı bu kitap olmazdı.

Özet

- Derin öğrenme, şimdilerde bilgisayarla görme, doğal dil işleme, otomatik konuşma tanıma da dahil olmak üzere çok çeşitli teknolojilere güç veren teknolojiyi tanıtarak desen tanımada (pattern recognition) devrim yaratmıştır.
- Derin öğrenmeyi başarılı bir şekilde uygulamak için bir problemin nasıl çözüleceğini, modellemenin matematiğini, modellerinizi verilere uyarlama algoritmalarını ve hepsini uygulamak için de mühendislik tekniklerini anlamalısınız.
- Bu kitap düzyazı, şekiller, matematik ve kod dahil olmak üzere kapsamlı bir kaynak sunar.
- Bu kitapla ilgili soruları cevaplamak için https://discuss.d2l.ai/ adresindeki forumumuzu ziyaret edin.
- Tüm not defterleri GitHub'dan indirilebilir.

Alıştırmalar

1. Bu kitabın discuss.d2l.ai[7] tartışma forumunda bir hesap açın.
2. Python'u bilgisayarınıza yükleyin.
3. Yazarın ve daha geniş bir topluluğun katılımıyla yardım arayabileceğiniz, kitabı tartışabileceğiniz ve sorularınıza cevap bulabileceğiniz bölüm altındaki forum bağlantılarını takip edin.

Tartışmalar[8]

[7] https://discuss.d2l.ai/
[8] https://discuss.d2l.ai/t/20

Kurulum

Sizi uygulamalı öğrenme deneyimine hazır hale getirmek için Python'u, Jupyter not defterlerini, ilgili kütüphaneleri ve kitabın kendisini çalıştırmada gerekli kodu koşturan bir ortam kurmanız gerekiyor.

Miniconda'yı Yükleme

Miniconda[9]'yı yükleyerek işe başlayabilirsiniz. Python 3.x sürümü gereklidir. Eğer makinenizde conda önceden kurulmuş ise, aşağıdaki adımları atlayabilirsiniz.

Miniconda'nın web sitesini ziyaret edin ve Python 3.x sürümünüze ve makine mimarinize göre sisteminiz için uygun sürümü belirleyin. Örneğin, macOS ve Python 3.x kullanıyorsanız, adı "Miniconda3" ve "MacOSX" dizelerini içeren bash betiğini indirin, indirme konumuna gidin ve kurulumu aşağıdaki gibi yürütün:

```
sh Miniconda3-py39_4.12.0-MacOSX-x86_64.sh -b
```

Python 3.x'e sahip bir Linux kullanıcısı, adı "Miniconda3" ve "Linux" dizelerini içeren dosyayı indirmeli ve indirme konumunda aşağıda yazılanları yürütmeli:

```
sh Miniconda3-latest-Linux-x86_64.sh -b
```

Ardından, doğrudan conda'yı çalıştırabilmeniz için kabuğu (shell) sıfırlayın.

```
~/miniconda3/bin/conda init
```

Şimdi mevcut kabuğunuzu kapatın ve yeniden açın. Artık aşağıdaki gibi yeni bir ortam oluşturabilirsiniz:

```
conda create --name d2l python=3.9 -y
```

[9] https://conda.io/en/latest/miniconda.html

D2L Not Defterlerini İndirme

Sonra, bu kitapta kullanılan kodu indirmeniz gerekiyor. Kodu indirmek ve açmak için herhangi bir HTML sayfasının üst kısmındaki "Not Defterleri" sekmesine tıklayabilirsiniz. Alternatif olarak, "unzip" varsa kullanabilirsiniz (yoksa "sudo apt install unzip" yazarak kurabilirsiniz):

```
mkdir d2l-tr && cd d2l-tr
curl https://tr.d2l.ai/d2l-tr.zip -o d2l-tr.zip
unzip d2l-tr.zip && rm d2l-tr.zip
```

Şimdi `d2l` ortamını etkinleştirebiliriz:

```
conda activate d2l
```

Çerçeveyi ve `d2l` Paketini Yükleme

Herhangi bir derin öğrenme çerçevesini kurmadan önce, lütfen önce makinenizde uygun GPU'ların olup olmadığını kontrol edin (standart bir dizüstü bilgisayarda ekranı destekleyen GPU'lar bizim amacımıza uygun sayılmaz). GPU'lu bir sunucuda çalışıyorsanız, ilgili kütüphanelerin GPU-dostu sürümlerinin kurulum talimatları için şu adrese ilerleyin *GPU Desteği* (page 11).

Makinenizde herhangi bir GPU yoksa, henüz endişelenmenize gerek yok. CPU'nuz, ilk birkaç bölümü tamamlamanız için fazlasıyla yeterli beygir gücü sağlar. Daha büyük modelleri koşmadan önce GPU'lara erişmek isteyeceğinizi unutmayın. CPU sürümünü kurmak için aşağıdaki komutu yürütün.

```
pip install torch torchvision
```

Bir sonraki adımımız, bu kitapta bulunan sık kullanılan işlevleri ve sınıfları kapsamak için geliştirdiğimiz `d2l` paketini kurmaktır.

```
pip install d2l==0.17.5
```

Bu kurulum adımlarını tamamladıktan sonra, Jupyter not defteri sunucusunu şu şekilde çalıştırarak başlatabiliriz:

```
jupyter notebook
```

Bu noktada, http://localhost:8888 (otomatik olarak açılmış olabilir) adresini web tarayıcınızda açabilirsiniz. Sonra kitabın her bölümü için kodu çalıştırabilirsiniz. Lütfen kitabın kodunu çalıştırmadan veya derin öğrenme çerçevesini veya `d2l` paketini güncellemeden önce çalışma zamanı ortamını etkinleştirmek için daima `conda enable d2l` komutunu çalıştırın. Ortamdan çıkmak için `conda deactivate` komutunu çalıştırın.

GPU Desteği

Derin öğrenme çerçevesi, aksi belirtilmediyse GPU desteğiyle kurulacaktır. Eğer bilgisayarınızda NVIDIA GPU'su varsa ve CUDA[10] kuruluysa, artık hazırsınız.

Alıştırmalar

1. Kitabın kodunu indirin ve çalışma zamanı ortamını yükleyin.

Tartışmalar[11]

[10] https://developer.nvidia.com/cuda-downloads
[11] https://discuss.d2l.ai/t/24

Notasyon

Bu kitap boyunca, aşağıdaki gösterim kurallarına bağlı kalacağız. Bu sembollerden bazılarının göstermelik değişken olduğunu, bazılarının ise belirli nesnelere atıfta bulunduğunu unutmayın. Genel bir kural olarak, belirsiz "a" nesnesi, sembolün bir göstermelik değişken olduğunu ve benzer şekilde biçimlendirilmiş sembollerin aynı tipteki diğer nesneleri gösterebileceğini belirtir. Örneğin, "x: bir sayıl", küçük harflerin genellikle sayıl değerleri temsil ettiği anlamına gelir.

Sayısal Nesneler

- x: skalar (sayıl)
- $\mathbf{x}$: vektör (Yöney)
- $\mathbf{X}$: matris (Dizey)
- X: bir genel tensör (Gerey)
- $\mathbf{I}$: birim dizeyi – köşegen girdileri 1 köşegen-olmayan girdileri 0 olan kare dizey
- x_i, $[\mathbf{x}]_i$: $\mathbf{x}$ dizeyinin $i.$ elemanı
- x_{ij}, $x_{i,j}$,$[\mathbf{X}]_{ij}$, $[\mathbf{X}]_{i,j}$: $\mathbf{X}$ dizeyinin $i.$ satır $j.$ sütundaki elemanı

Küme Kuramı

- $\mathcal{X}$: küme
- $\mathbb{Z}$: tam sayılar kümesi
- $\mathbb{Z}^+$: pozitif tam sayılar kümesi
- $\mathbb{R}$: gerçel sayılar kümesi
- $\mathbb{R}^n$: n boyutlu gerçel sayılı yöneyler kümesi
- $\mathbb{R}^{a\times b}$: a satır ve b sütunlu gerçek sayılı matrisler kümesi
- $|\mathcal{X}|$: $\mathcal{X}$ kümesinin kardinalitesi (eleman sayısı)
- $\mathcal{A} \cup \mathcal{B}$: $\mathcal{A}$ ve $\mathcal{B}$ kümelerinin bileşkesi
- $\mathcal{A} \cap \mathcal{B}$: $\mathcal{A}$ ve $\mathcal{B}$ kümelerinin kesişimi
- $\mathcal{A} \setminus \mathcal{B}$: $\mathcal{B}$ kümesinin $\mathcal{A}$ kümesinden çıkarılması (sadece $\mathcal{A}$ kümesinde olup $\mathcal{B}$ kümesinde olmayan elemanları içerir)

Fonksiyonlar ve Operatörler

- $f(\cdot)$: işlev (fonksiyon)
- $\log(\cdot)$: doğal logaritma
- $\log_2(\cdot)$: 2lik tabanda logaritma
- $\exp(\cdot)$: üstel fonksiyon
- $\mathbf{1}(\cdot)$: gösterge fonksiyonu, mantık argümanı doğru ise 1 ve değil ise 0
- $\mathbf{1}_{\mathcal{X}}(z)$: küme-üyeliği gösterge işlevi, eğer z elemanı $\mathcal{X}$ kümesine ait ise 1 ve değil ise 0
- $(\cdot)^\top$: bir vektörün veya matrisin devriği
- $\mathbf{X}^{-1}$: $\mathbf{X}$ matrisinin tersi
- $\odot$: Hadamard (eleman-yönlü) çarpımı
- $[\cdot, \cdot]$: bitiştirme
- $\|\cdot\|_p$: L_p büyüklüğü (Norm)
- $\|\cdot\|$: L_2 büyüklüğü (Norm)
- $\langle \mathbf{x}, \mathbf{y} \rangle$: $\mathbf{x}$ ve $\mathbf{y}$ vektörlerinin iç (nokta) çarpımı
- $\sum$: bir elemanlar topluluğu üzerinde toplam
- $\prod$: bir elemanlar topluluğu üzerinde toplam çarpımı
- $\stackrel{\text{def}}{=}$: sol tarafta bir sembol tanımlarken kullanılan eşitlik

Hesaplama (Kalkülüs)

- $\frac{dy}{dx}$: y'nin x'e göre türevi
- $\frac{\partial y}{\partial x}$: y'nin x'e göre kısmi türevi
- $\nabla_{\mathbf{x}} y$: y'nin $\mathbf{x}$'e göre eğimi (Gradyan)
- $\int_a^b f(x)\, dx$: f'in x'e göre a'dan b'ye belirli bir tümlevi (integrali)
- $\int f(x)\, dx$: f'in x'e göre belirsiz bir tümlevi

Olasılık ve Bilgi Kuramı

- X: rasgele değişken
- P: olasılık dağılımı
- $X \sim P$: X rasgele değişkeni P olasılık dağılımına sahiptir
- $P(X = x)$: X rasgele değişkeninin x değerini alma olayının olasılığı
- $P(X \mid Y)$: Y bilindiğinde X'in koşullu olasılık dağılımı
- $p(\cdot)$: P dağılımı ile ilişkili koşullu olasılık yoğunluk fonksiyonu

- $E[X]$: X rasgele değişkeninin beklentisi
- $X \perp Y$: X ve Y rasgele değişkenleri bağımsızdır
- $X \perp Y \mid Z$: X ve Y rasgele değişkenleri, Z göz önüne alındığında (verildiğinde) koşullu olarak bağımsızdır
- σ_X: X rasgele değişkeninin standart sapması
- $\text{Var}(X)$: X rasgele değişkeninin değişintisi (varyansı), σ_X^2'e eşittir
- $\text{Cov}(X, Y)$: X ve Y rasgele değişkenlerinin eşdeğişintisi (kovaryansı)
- $\rho(X, Y)$: X ve Y rasgele değişkenleri arasındaki Pearson ilinti katsayısı (korrelasyonu), $\frac{\text{Cov}(X,Y)}{\sigma_X \sigma_Y}$'ye eşittir
- $H(X)$: X rasgele değişkeninin düzensizliği (entropisi)
- $D_{\text{KL}}(P\|Q)$: Q dağılımından P dağılımına KL-Iraksaması (veya göreceli düzensizlik)

Tartışmalar[12]

[12] https://discuss.d2l.ai/t/25

1 | Giriş

Yakın zamana kadar, günlük etkileşimde bulunduğumuz hemen hemen her bilgisayar programı basit prensiplerle yazılım geliştiricileri tarafından kodlandı. Bir e-ticaret uygulaması yazmak istediğimizi varsayalım. Soruna bir beyaz tahta üzerinde birkaç saat kafa yorduktan sonra muhtemelen aşağıdaki gibi bir çözüme ulaşırız: (i) Kullanıcılar bir web tarayıcısında veya mobil uygulamada çalışan bir arabirim aracılığıyla uygulama ile etkileşimde bulunurlar, (ii) uygulamamız, her kullanıcının durumunu takip etmek ve geçmiş işlemlerin kayıtlarını tutmak için ticari düzeyde bir veritabanı motoruyla etkileşime girer ve (iii) uygulamamızın merkezindeki *iş mantığı* (uygulamanın beyni diyebiliriz), farklı senaryolarda uygulamanın nasıl davranacağını belirler.

Uygulamamızın *beynini* oluşturmak için karşılaşacağımızı tahmin ettiğimiz her senaryoyu değerlendirerek uygun kuralları belirlememiz gerekir. Bir müşteri alışveriş sepetine bir ürün eklemek için her tıkladığında, alışveriş sepeti veritabanı tablosuna bu müşterinin kimliğini istenen ürünün kimliği ile ilişkilendirerek bir kayıt ekleriz. Böyle bir programı basit prensiplerle yazabilir ve güvenle başlatabiliriz. Çok az geliştirici ilk seferde tamamen doğru yapabilse de (sorunları çözmek için birkaç deneme çalışması gerekebilir), çoğunlukla böyle bir programı basit prensiplerden yazabilir ve gerçek bir müşteri görmeden *önceden* programı güvenle başlatabiliriz. Genellikle yeni durumlarda, işlevsel ürün ve sistemleri yöneten uygulamaları tasarlama yeteneğimiz, dikkate değer bir bilişsel başarıdır. Ayrıca %100'lük oranda işe yarayan çözümler tasarlayabildiğinizde, makine öğrenmesi kullanmamalısınız.

Giderek artan sayıdaki makine öğrenmesi uzmanı için ne mutlu ki, otomatikleştirmek istediğimiz birçok görev insan yaratıcılığına bu kadar kolay boyun eğmiyor. Beyaz tahta etrafında bildiğiniz en akıllı zihinlerle toplandığınızı hayal edin, ancak bu sefer aşağıdaki sorunlardan birini ele alıyorsunuz:

- Coğrafi bilgi, uydu görüntüleri ve yakın bir zaman aralığındaki geçmiş hava koşulları göz önüne alındığında yarının hava durumunu tahmin eden bir program yazma.
- Serbest biçimli metinle ifade edilen bir soruyu alan ve onu doğru cevaplayan bir program yazma.
- Verilen bir fotoğrafın içerdiği tüm insanları her birinin etrafına çerçeve çizerek tanımlayabilen bir program yazma.
- Kullanıcılara internette gezinirken karşılaşma olasılıkları yüksek olmayan ancak keyif alabilecekleri ürünler sunan bir program yazma.

Bu durumların her birinde, seçkin programcılar bile çözümleri sıfırdan kodlayamazlar. Bunun farklı nedenleri olabilir. Bazen aradığımız program zaman içinde değişen bir deseni takip eder ve programlarımızın adapte olması gerekir. Diğer durumlarda, ilişki (pikseller ve soyut kategoriler arasında) çok karmaşık olabilir ve bilinçli anlayışımızın ötesinde binlerce veya milyonlarca hesaplama gerekebilir, ki gözlerimiz bu görevi halihâzırda zahmetsizce yönetir. *Makine öğrenmesi (MÖ)* deneyimlerden öğrenebilen güçlü tekniklerin incelenmesidir. Bir makine öğrenmesi

algoritmasının performansı, tipik gözlemsel veri veya bir çevre ile etkileşim şeklinde olur, daha fazla deneyim biriktirdikçe artar. Bunu, ne kadar deneyim kazanırsa kazansın, aynı iş mantığına göre çalışmaya devam eden geliştiricilerin kendileri öğrenip yazılımın güncellenme zamanının geldiğine karar verene kadar deterministik (gerekirci) e-ticaret platformumuzla karşılaştırın. Bu kitapta size makine öğrenmesinin temellerini öğreteceğiz ve özellikle de bilgisayarla görme, doğal dil işleme, sağlık ve genomik gibi farklı alanlarda yenilikleri yönlendiren güçlü bir teknik altyapıya, yani *derin öğrenme*ye odaklanacağız.

1.1 Motive Edici Bir Örnek

Bu kitabı yazmaya başlayabilmek için, birçok çalışan gibi, bol miktarda kahve tüketmemiz gerekiyordu. Arabaya bindik ve sürmeye başladık. Alex "Hey Siri" diye seslenerek iPhone'unun sesli asistan sistemini uyandırdı. Sonra Mu "Blue Bottle kafesine yol tarifi" komutunu verdi. Telefon komutun metnini (transkripsiyonunu) hızlı bir şekilde gösterdi. Ayrıca yol tarifini istediğimizi fark etti ve talebimizi yerine getirmek için Maps uygulamasını (app) başlattı. Başlatılınca Maps uygulaması bir dizi rota belirledi, her rotanın yanında tahmini bir varış süresi de gösterdi. Eğitim amaçlı tasarlandığımız bu hikaye, bir akıllı telefondaki günlük etkileşimlerimizin saniyeler içinde birkaç makine öğrenmesi modeliyle işbirligi yaptığını gösteriyor.

"Alexa", "OK, Google" veya "Hey Siri" gibi bir *uyandırma kelimesine* yanıt vermek için bir program yazdığınızı düşünün. Bir odadayken kendiniz bir bilgisayar ve kod editöründen başka bir şey olmadan kodlamayı deneyin Fig. 1.1.1. Böyle bir programı basit prensiplerle nasıl yazarsınız? Bir düşünün ... problem zor. Mikrofon her saniye yaklaşık 44000 örnek toplayacaktır. Her örnek, ses dalgasının genliğinin bir ölçümüdür. Hangi kural güvenilir bir şekilde, ses parçasının uyandırma sözcüğünü içerip içermediğine bağlı olarak bir ham ses parçasından emin $\{\text{evet}, \text{hayır}\}$ tahminlerine eşleme yapabilir? Cevabı bulmakta zorlanıyorsanız endişelenmeyin. Böyle bir programı nasıl sıfırdan yazacağımızı bilmiyoruz. Bu yüzden makine öğrenmesi kullanıyoruz.

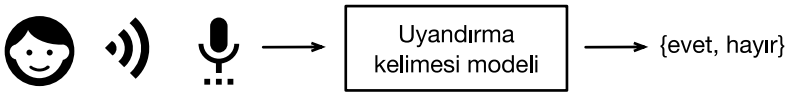


Fig. 1.1.1: Bir uyandırma kelimesi tanıma.

Olayın özünü şöyle açıklayabiliriz: Çoğu zaman, bir bilgisayara girdilerle çıktıları nasıl eşleştirebileceğini açıklayamayı bilmediğimizde bile, kendimiz bu bilişsel işi gerçekleştirebiliyoruz. Diğer bir deyişle, "Alexa" kelimesini tanımak için bir bilgisayarı nasıl programlayacağınızı bilmeseniz bile siz kendiniz "Alexa" kelimesini tanıyabilirsiniz. Bu yetenekle donanmış bizler ses örnekleri içeren büyük bir *veri kümesi* toplayabilir ve uyandırma kelimesi *içerenleri* ve *içermeyenleri* etiketleyebiliriz. Makine öğrenmesi yaklaşımında, uyandırma kelimelerini tanımak için *açıktan* bir sistem tasarlamaya çalışmayız. Bunun yerine, davranışı bir takım *parametre* ile belirlenen esnek bir program tanımlarız. Ardından, veri kümesini, ilgili görevdeki performans ölçüsüne göre, programımızın performansını artıran en iyi parametre kümesini belirlemek için kullanırız.

Parametreleri, çevirerek programın davranışını değiştirebileceğimiz düğmeler olarak düşünebilirsiniz. Parametreleri sabitlendiğinde, programa *model* diyoruz. Sadece parametreleri manipüle ederek üretebileceğimiz tüm farklı programlara (girdi-çıktı eşlemeleri) *model ailesi* denir. Ayrıca parametreleri seçmek için veri kümemizi kullanan meta (başkalaşım) programa *öğrenme algoritması* denir.

Devam etmeden ve öğrenme algoritmasını kullanmadan önce, sorunu kesin olarak tanımlamalı, girdi ve çıktıların kesin doğasını tespit etmeli ve uygun bir model ailesi seçmeliyiz. Bu durumda, modelimiz *girdi* olarak bir ses parçasını alır ve *çıktı* olarak {evet, hayır} arasında bir seçim oluşturur. Her şey plana göre giderse, modelin parçanın uyandırma kelimesini içerip içermediğine dair tahminleri genellikle doğru olacaktır.

Doğru model ailesini seçersek, o zaman model "Alexa" kelimesini her duyduğunda "evet"i seçecek düğmelerin bir ayarı olmalıdır. Uyandırma kelimesinin kesin seçimi keyfi olduğundan, muhtemelen yeterince zengin bir model ailesine ihtiyacımız olacak, öyle ki düğmelerin başka bir ayarı ile, sadece "Kayısı" kelimesini duyduktan sonra da "evet" seçilebilsin. Aynı model ailesinin "Alexa"yı tanıma ve "Kayısı"yı tanıma için uygun olması beklenir, çünkü sezgisel olarak benzer görevler gibi görünüyorlar. Bununla birlikte, temel olarak farklı girdiler veya çıktılarla uğraşmak istiyorsak, resimlerden altyazılara veya İngilizce cümlelerden Çince cümlelere eşlemek istiyorsak mesela, tamamen farklı bir model ailesine ihtiyacımız olabilir.

Tahmin edebileceğiniz gibi, tüm düğmeleri rastgele bir şekilde ayarlarsak, modelimizin "Alexa", "Kayısı" veya başka bir kelimeyi tanıması muhtemel değildir. Makine öğrenmesinde, *öğrenme*, modelimizi istenen davranışa zorlayan düğmelerin doğru ayarını keşfettiğimiz süreçtir. Başka bir deyişle, modelimizi veri ile *eğitiyoruz*. Fig. 1.1.2'de gösterildiği gibi, eğitim süreci genellikle şöyle görünür:

1. Yararlı bir şey yapamayan rastgele ilkletilen bir model ile başlayın.
2. Verilerinizin bir kısmını alın (örneğin, ses parçaları ve onlara karşılık gelen {evet, hayır} etiketleri).
3. Modelin bu örneklere göre daha az hata yapması için düğmelerin ayarlarını değiştirin.
4. Model harika olana kadar 2. ve 3. adımı tekrarlayın.

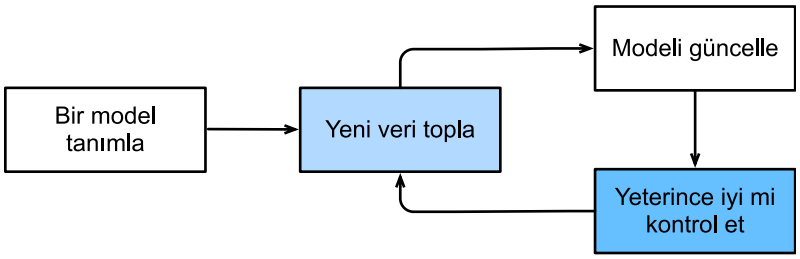


Fig. 1.1.2: Tipik bir eğitim süreci.

Özetlemek gerekirse, bir uyandırma kelimesi tanıyıcısını kodlamak yerine, büyük bir etiketli veri kümesi sunarsak uyandırma sözcüklerini tanımayı *öğrenebilen* bir program kodlarız. Bu eylemi bir programın davranışını ona bir veri kümesi sunup *veri ile programlayarak* belirleme gibi düşünebilirsiniz. Söylemek istediğimiz MÖ sistemimize birçok kedi ve köpek örneği sağlayarak bir kedi dedektörü "programlayabiliriz". Bu şekilde dedektör, sonunda, bir kedi ise çok büyük bir pozitif sayı, bir köpekse çok büyük bir negatif sayı ve emin değilse sıfıra daha yakın bir şey yaymayı öğrenir ve bu, makine öğrenmesinin neler yapabileceğinin ancak yüzeyine ışık tutar. Derin öğrenme (DÖ), ki daha sonra çok detaylı açıklayacağız, makine öğrenmesi problemlerini çözmek için mevcut birçok popüler yöntemden sadece biridir.

1.2 Temel Bileşenler

Uyandırma kelimesi örneğimizde, ses parçaları ve ikili etiketlerden oluşan bir veri kümesi tanımladık ve parçalardan sınıflandırmalara bir eşlemeyi yaklaşık olarak nasıl eğitebileceğimize dair çok ciddi olmayan bir izlenim verdik. Bu tarz bir problem, etiketleri bilinen örneklerden oluşan bir veri kümesinin verildiği ve bilinen girdilerin etiketini öngörmeye çalıştığımız, *gözetimli öğrenme* olarak adlandırılır. Bu MÖ problemi çeşitlerinden sadece bir tanesidir. Daha sonra, farklı MÖ sorunlarına derinlemesine bakacağız. İlk olarak, ne tür bir MÖ problemi olursa olsun, bizi takip edecek temel bileşenlere daha fazla ışık tutmak istiyoruz:

1. Öğrenebileceğimiz *veri*.
2. Verinin nasıl dönüştürüleceğine dair bir *model*.
3. Modelin ne kadar iyi veya kötü iş çıkardığını ölçümleyen bir *amaç* işlevi (objective function).
4. Kaybı en aza indirmede modelin parametrelerini ayarlamak için bir *algoritma*.

1.2.1 Veri

Veri bilimini veri olmadan yapamayacağınızı söylemeye gerek yok. Tam olarak veriyi neyin oluşturduğunu düşünerek yüzlerce sayfayı doldurabiliriz, ancak şimdilik pratik tarafta hata yapacağız ve bizi ilgilendiren temel özelliklere odaklanacağız. Genellikle örnekler derlemesiyle ilgileniriz. Veriyle yararlı bir şekilde çalışmak için, genellikle uygun bir sayısal temsil (gösterim) bulmamız gerekir. Her *örnek* (*veri noktası*, *veri örnekleri* veya *örneklem* olarak da adlandırılır) tipik olarak onlardan modelimizin tahminlemelerini yaptığı *öznitelikler* (veya *ortak değişkenler*) adı verilen sayısal özelliklerden oluşur. Yukarıdaki gözetimli öğrenme problemlerinde, tahmin etmeye çalıştığımız şey *etiket* (veya *hedef*) olarak atanmış özel bir özelliktir.

Eğer görüntü verileriyle çalışıyorsak, bir fotoğraf için, her bir pikselin parlaklığına karşılık gelen sıralı bir sayısal değerler listesi ile temsil edilen bir örnek oluşturabilir. 200×200 bir renkli fotoğraf, her bir uzamsal konum için kırmızı, yeşil ve mavi kanalların parlaklığına karşılık gelen $200 \times 200 \times 3 = 120000$ tane sayısal değerden oluşur. Başka bir geleneksel görevde ise yaş, yaşamsal belirtiler ve teşhisler gibi standart bir dizi özellik göz önüne alındığında, bir hastanın hayatta kalıp kalmayacağını tahmin etmeye çalışabiliriz.

Her örnek aynı sayıda sayısal değerle karakterize edildiğinde, verinin sabit uzunluklu vektörlerden oluştuğunu söylüyoruz ve vektörlerin sabit uzunluğunu verinin *boyutu* olarak tanımlıyoruz. Tahmin edebileceğiniz gibi, sabit uzunluk işleri kolaylaştıracak bir özellik olabilir. Mikroskopik görüntülerde kanseri tanımak için bir model eğitmek istersek, sabit uzunluktaki girdilere sahip olmak endişelenecek şeylerin sayısının bir tane azaldığı anlamına gelir.

Ancak, tüm veri kolayca sabit uzunluklu vektörler olarak gösterilemez. Mikroskop görüntülerinin standart ekipmanlardan gelmesini beklesek de, internetten toplanan görüntülerin aynı çözünürlük veya şekil ile ortaya çıkmasını bekleyemeyiz. Görüntüler için, hepsini standart bir boyuta kırpmayı düşünebiliriz, ancak bu strateji bizi bir yere kadar götürür. Kırpılan bölümlerde bilgi kaybetme riskiyle karşı karşıyayız. Ayrıca, metin verisi sabit uzunluklu gösterimlere daha inatçı bir şekilde direnir. Amazon, IMDB veya TripAdvisor gibi e-ticaret sitelerine bırakılan müşteri yorumlarını düşünün. Bazıları kısadır: "Berbat!", bazıları da sayfalara yayılır. Geleneksel yöntemlere göre derin öğrenmenin en büyük avantajlarından biri *değişken uzunluktaki* veriyi işlemedeki göreceli yeteneğidir.

Genel olarak, ne kadar fazla veriye sahip olursak işimiz o kadar kolay olur. Daha fazla veriye sahip olduğumuzda, daha güçlü modeller eğitebilir ve önceden tasarlanmış varsayımlara daha az

bel bağlayabiliriz. (Nispeten) küçük veriden büyük veriye düzen değişikliği, modern derin öğrenmenin başarısına önemli bir katkıda bulunur. İşin özü, derin öğrenmedeki en heyecan verici modellerin çoğu, büyük veri kümeleri olmadan çalışmaz. Bazıları küçük veri rejiminde çalışır, ancak geleneksel yaklaşımlardan daha iyi değildir.

Son olarak, çok fazla veriye sahip olmak ve onu akıllıca işlemek yeterli değildir. *Doğru* veriye ihtiyacımız vardır. Veri hatalarla doluysa veya seçilen öznitelikler hedefle ilgisizse, öğrenme başarısız olacaktır. Durum şu klişe ile iyi betimlenebilir: *Çöp girerse çöp çıkar*. Ayrıca, kötü tahmin performansı tek olası sonuç değildir. Tahminli polislik, özgeçmiş taraması ve borç verme için kullanılan risk modelleri gibi makine öğrenmesinin hassas uygulamalarında, özellikle çöp verinin olası sonuçlarına karşı dikkatli olmalıyız. Yaygın bir hata durumu bazı insan gruplarının eğitim verilerinde temsil edilmediği veri kümelerinde gerçekleşir. Gerçek hayatta, daha önce hiç koyu ten görmemiş bir cilt kanseri tanıma sistemi uyguladığınızı düşünün. Ayrıca başarısızlık, veri sadece bazı grupları az temsil ettiğinde değil, aynı zamanda toplumsal önyargıları yansıttığı zaman da meydana gelebilir. Örneğin, özgeçmişleri taramak için kullanılacak bir öngörü modeli eğitmek için geçmiş işe alım kararları kullanılıyorsa, makine öğrenme modelleri yanlışlıkla tarihi adaletsizlikleri yakalayıp onları otomatikleştirebilir. Tüm bunların veri bilimcisi aktif olarak komplo kurmadan ve hatta o farkında olmadan gerçekleşebileceğini unutmayın.

1.2.2 Modeller

Çoğu makine öğrenmesi, veriyi bir anlamda dönüştürmeyi içerir. Fotoğrafları alarak güleryüzlülük tahmin eden bir sistem kurmak isteyebiliriz. Alternatif olarak, bir dizi sensör okuması alarak normal veya anormal olup olmadıklarını tahmin etmek isteyebiliriz. *Model* ile, bir tipteki veriyi alan ve muhtemel farklı tipteki tahminleri veren hesaplama makinelerini belirtiyoruz. Özellikle veriden tahmin yapabilecek istatistiksel modellerle ilgileniyoruz. Basit modeller, basitliği uygun problemleri mükemmel bir şekilde çözebilirken, bu kitapta odaklandığımız problemler klasik yöntemlerin sınırlarını aşmaktadır. Derin öğrenme, klasik yaklaşımlardan esas olarak odaklandığı güçlü modeller kümesi ile ayrılır. Bu modeller, yukarıdan aşağıya zincirlenmiş verinin art arda dönüşümlerinden oluşur, bu nedenle adları *derin öğrenme*dir. Derin modelleri tartışırken, bazı geleneksel yöntemlere de değineceğiz.

1.2.3 Amaç Fonksiyonları

Daha önce, makine öğrenmesini deneyimden öğrenme olarak tanıttık. Burada *öğrenme* ile zamanla bazı görevlerde iyileştirmeyi kastediyoruz. Peki kim neyin bir iyileştirme oluşturduğunu söyleyecek? Modeli güncellemeyi önerdiğiniz zaman önerilen güncellemenin bir iyileştirme mi yoksa bir düşüş mü oluşturacağı konusunda görüş ayrılıkları olabileceğini tahmin edebilirsiniz.

Biçimsel bir matematiksel öğrenen makineler sistemi geliştirmek için modellerimizin ne kadar iyi (ya da kötü) olduğuna dair kurallı ölçümlere ihtiyacımız var. Makine öğrenmesinde ve daha genel olarak optimizasyonda (eniyilemede), bunları amaç fonsiyonları olarak adlandırıyoruz. Yaygın yaklaşım olarak, genellikle amaç fonksiyonları tanımlarız, böylece daha düşük değer daha iyi anlamına gelir. Bu sadece yaygın bir kanıdır. Daha yüksekken daha iyi olan herhangi bir fonksiyonu alabilir ve onu, işaretini değiştirerek niteliksel olarak özdeş ama daha düşükken daha iyi yeni bir fonksiyona dönüştürebilirsiniz. Düşük daha iyi olduğu için, bu fonksiyona bazen *yitim fonksiyonları (loss function)* denir.

Sayısal değerleri tahmin etmeye çalışırken, en yaygın amaç fonksiyonu hata karesidir, yani gerçek referans değeri ile tahmin değeri arasındaki farkın karesi. Sınıflandırma için en yaygın amaç fonksiyonu, hata oranını, yani tahminlerimizin gerçek değere uymadığı örneklerin oranını, en aza

indirmektir. Bazı hedeflerin (hata karesi gibi) optimize edilmesi kolaydır. Diğerlerinin (hata oranı gibi) türevlerinin alınamaması veya diğer başka zorluklar nedeniyle doğrudan optimize edilmesi zordur. Bu durumlarda, *vekil (surrogate) amaç* optimizasyonu yaygındır.

Tipik olarak, yitim fonksiyonu modelin parametrelerine göre tanımlanır ve veri kümesine bağlıdır. Modelimizin parametrelerinin en iyi değerlerini, eğitim için toplanan örneklerden oluşan bir kümede meydana gelen kaybı en aza indirerek öğreniriz. Bununla birlikte, eğitim verisinde iyi performans göstermemiz, görülmemiş veri üzerinde iyi performans göstereceğimizi garanti etmez. Bu nedenle, genellikle mevcut veriyi iki parçaya ayıracağız: Modelimizin ikisinin de üzerindeki performanslarını raporladığımız eğitim veri kümesi (veya eğitim kümesi, model parametrelerini bulmak için) ve test veri kümesi (veya test kümesi, değerlendirme için dışarıda tutulur). Eğitim hatasını, bir öğrencinin gerçek bir final sınava hazırlanmak için girdiği uygulama sınavlarındaki puanları gibi düşünebilirsiniz. Sonuçlar cesaret verici olsa bile, bu final sınavında başarıyı garanti etmez. Başka bir deyişle, test performansı eğitim performansından ciddi derecede sapabilir. Bir model eğitim kümesi üzerinde iyi performans gösterdiğinde, ancak bunu görünmeyen veriye genelleştiremediğinde, buna *aşırı öğrenme (overfitting)* diyoruz. Bu durumu uygulama sınavlarında başarılı olunmasına rağmen gerçek sınavda çakmaya benzetebiliriz.

1.2.4 Optimizasyon (Eniyileme) Algoritmaları

Bir kez bir veri kaynağına ve gösterime, bir modele ve iyi tanımlanmış bir amaç fonksiyonuna sahip olduktan sonra, yitim fonksiyonunu en aza indirmek için mümkün olan en iyi parametreleri arayabilme becerisine sahip bir algoritmaya ihtiyacımız var. Derin öğrenme için popüler optimizasyon algoritmaları, gradyan (eğim) inişi (gradient descent) olarak adlandırılan bir yaklaşıma bağlıdır. Kısacası, her adımda, her bir parametre için, bu parametreyi sadece küçük bir miktar bozarsanız eğitim kümesi kaybının nasıl hareket edeceğini (değişeceğini) kontrol ederler. Daha sonra parametreyi kaybı azaltabilecek yönde güncellerler.

1.3 Makine Öğrenmesi Problemleri Çeşitleri

Motive edici örneğimizdeki uyanma kelimesi problemi, makine öğrenmesinin üstesinden gelebileceği birçok problemden sadece biridir. Okuyucuyu daha fazla motive etmek ve kitap boyunca daha fazla sorun hakkında konuştuğumuzda bize ortak bir dil sağlaması için, aşağıda makine öğrenmesi sorunlarının bir örneğini listeliyoruz. Veriler, modeller ve eğitim teknikleri gibi yukarıda belirtilen kavramlarımıza sürekli olarak atıfta bulunacağız.

1.3.1 Gözetimli Öğrenme

Gözetimli öğrenme girdi öznitelikleri verildiğinde etiketleri tahmin etme görevini ele alır. Her öznitelik–etiket çiftine örnek denir. Bazen, bağlam açık olduğunda, bir girdi topluluğuna atıfta bulunmak için -karşılık gelen etiketler bilinmese bile- *örnekler* terimini kullanabiliriz. Hedefimiz, herhangi bir girdiyi bir etiket tahminiyle eşleyen bir model üretmektir.

Bu açıklamayı somut bir örneğe oturtalım; sağlık hizmetlerinde çalışıyorsak, bir hastanın kalp krizi geçirip geçirmeyeceğini tahmin etmek isteyebiliriz. “Kalp krizi var” veya “kalp krizi yok” gözlemi, bizim etiketimiz olacaktır. Girdi öznitelikleri, kalp atış hızı, diyastolik ve sistolik kan basıncı gibi hayati belirtiler olabilir.

Burada gözetim devreye girer, çünkü parametreleri seçmek için, biz (gözetimciler) modele *etiketli örnekleri* içeren bir veri kümesi sağlıyoruz, ki bu kümedeki her örnek, gerçek referans değeri etiketle eşleştirilmiştir.

Olasılıksal terimlerle, tipik olarak öznitelikler verildiğinde bir etiketinin koşullu olasılığını tahmin etmekle ilgileniyoruz. Makine öğrenmesi içindeki birçok paradigmadan sadece biri olsa da, gözetimli öğrenme, makine öğrenmesinin endüstrideki başarılı uygulamalarının çoğunu oluşturur. Kısmen, bunun nedeni, birçok önemli görevin, belirli bir mevcut veri kümesi göz önüne alındığında bilinmeyen bir şeyin olasılığını tahmin etmek gibi net bir şekilde tanımlanabilmesidir. Örneğin:

- CT (hesaplamalı tomografi - computed tomography) görüntüsü verildiğinde kanserli olma/olmama tahmini.
- İngilizce bir cümle verildiğinde doğru Fransızca çevirisinin tahmini.
- Bir hisse senedinin gelecek aydaki fiyatının bu ayın finansal raporlama verilerine dayalı tahmini.

Temel olarak "girdi öznitelikleri verildiğinde etiketlerin tahmin edilmesi" diye tanımladığımız gözetimli öğrenme çok çeşitli şekillerde olabilir, diğer hususların yanı sıra girdilerin ve çıktıların türüne, boyutuna ve sayısına bağlı olarak çok sayıda modelleme kararı gerektirebilir. Örneğin, keyfi uzunluktaki dizileri işlemek için farklı ve sabit uzunluklu vektör temsillerini işlemek için farklı modeller kullanırız. Bu kitapta bu sorunların birçoğunu derinlemesine işleyeceğiz.

Gayri resmi olarak, öğrenme süreci şöyle görünür: İlk olarak özniteliklerin bilindiği büyük bir örnek koleksiyonunu alın ve rastgele bir altküme seçin - eğer yoksa bu altkümedeki her örneğin doğru değer etiketini oluşturun. Bazen bu etiketler zaten toplanmış da olabilir (örn. Bir hasta sonraki yıl içinde öldü mü?) ve diğer zamanlarda veriyi etiketlemek için insanların yorumlarını kullanmamız gerekebilir (örn. Görüntülere kategori atama). Bu girdiler ve karşılık gelen etiketler birlikte eğitim kümesini oluştururlar. Eğitim veri kümesini gözetimli bir öğrenme algoritmasına besleriz; bu algoritma veri kümesini girdi olarak alır ve başka bir fonksiyon verir: Öğrenmiş model. Son olarak, çıktılarını karşılık gelen etiketin tahminleri olarak kullanarak önceden görülmemiş girdileri öğrenmiş modele besleyebiliriz. Tüm süreç şöyle çizilebilir Fig. 1.3.1.

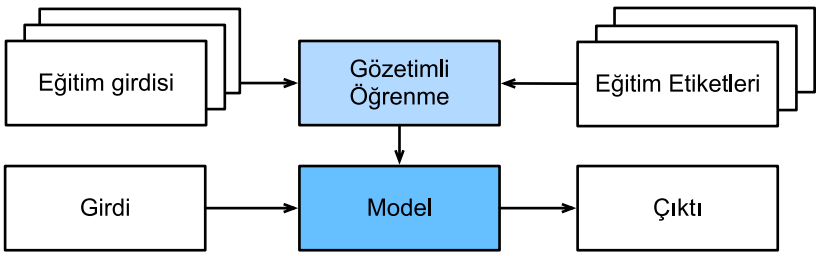


Fig. 1.3.1: Gözetimli öğrenme.

Bağlanım

Belki de kafanıza sokmak için en basit gözetimli öğrenme görevi *bağlanım*dır. Örneğin, ev satışları veritabanından toplanan bir veri kümesini düşünün. Her sıranın farklı bir eve karşılık geldiği bir tablo oluşturabiliriz ve her sütun, bir evin alanı, yatak odası sayısı, banyo sayısı ve şehir merkezine (yürüyüş) dakika uzaklığı gibi ilgili bazı özelliklere karşılık gelir. Bu veri kümesinde, her örnek belirli bir ev olacaktır ve karşılık gelen öznitelik vektörü tabloda bir satır olacaktır. New York veya San Francisco'da yaşıyorsanız ve Amazon, Google, Microsoft veya Facebook'un CEO'su değilseniz, (metrekare bilgileri, yatak odası sayısı, banyo sayısı, yürüme mesafesi) eviniz için vektör özelliği şuna benzeyebilir: $[600, 1, 1, 60]$. Ancak, Pittsburgh'da yaşıyorsanız, daha çok $[3000, 4, 3, 10]$ gibi görünebilir. Bunun gibi öznitelik vektörleri, çoğu klasik makine öğrenmesi algoritması için gereklidir.

Bir problemi *bağlanım* yapan aslında çıktılardır. Yeni bir ev için pazarda olduğunuzu varsayalım. Yukarıdaki gibi bazı öznitelikler göz önüne alındığında, bir evin adil piyasa değerini tahmin etmek isteyebilirsiniz. Etiket değeri, ki satış fiyatıdır, bir sayısal değerdir. Etiketler keyfi sayısal değerler aldığında buna bir *bağlanım* problemi diyoruz. Hedefimiz, tahminleri gerçek etiket değerlerine çok yakın olan bir model üretmektir.

Birçok pratik problem iyi tanımlanmış bağlanım problemleridir. Bir kullanıcının bir filme atayacağı puanı tahmin etmek bir bağlanım sorunu olarak düşünülebilir ve 2009'da bu özelliği gerçekleştirmek için harika bir algoritma tasarlasaydınız, 1 milyon dolarlık Netflix ödülünü[13] kazanmış olabilirdiniz. Hastanedeki hastalar için kalış süresinin öngörülmesi de bir bağlanım sorunudur. Pratik bir kural; aşağıdaki gibi, herhangi bir *ne kadar?* veya *kaç tane?* problemi bağlanım içerir:

- Bu ameliyat kaç saat sürecek?
- Önümüzdeki altı saatte bu kasabaya ne kadar yağmur düşecek?

Daha önce hiç makine öğrenmesi ile çalışmamış olsanız bile, muhtemelen gayri ihtiyari olarak bir bağlanım problemi ile çalışmışsınızdır. Örneğin, atık su giderlerinizin onarıldığını ve personelin kanalizasyon borularınızdan pisliği temizlemek için 3 saat harcadığını düşünün. Sonra size $350\$$ tutarında bir fatura gönderildi. Şimdi arkadaşınızın aynı personelini 2 saat kiraladığını ve $250\$$ fatura aldığını düşünün. Birisi size yaklaşan pislik temizleme faturasında ne kadar beklediğinizi sorarsa, bazı makul varsayımlar yapabilirsiniz; daha fazla çalışma saati daha fazla ücret maliyeti gibi. Ayrıca bir sabit ücretin olduğunu ve personelin saatlik ücret aldığını varsayabilirsiniz. Bu varsayımlar geçerliyse, bu iki veri örneği göz önüne alındığında, personelin fiyatlandırma yapısını zaten tanımlayabilirsiniz: Saat başı $100\$$ artı evinizde görünmesi için $50\$$. Eğer buraya kadar takip edebildiyseniz, doğrusal bağlanımın arkasındaki üst-seviye fikri zaten anlıyorsunuz.

Bu durumda, personelin fiyatına tam olarak uyan parametreleri üretebiliriz. Bazen bu mümkün olmayabilir; örneğin, varyansın bir kısmı iki özniteliğinizin yanı sıra birkaç diğer faktöre bağlıysa. Bu durumlarda, tahminlerimiz ile gözlenen değerler arasındaki mesafeyi en aza indiren modelleri öğrenmeye çalışacağız. Bölümlerimizin çoğunda kare hata kayıp fonksiyonunu en aza indirmeye odaklanacağız. Daha sonra göreceğimiz gibi, bu kayıp fonksiyonu, verilerimizin Gauss gürültüsü ile bozulduğu varsayımına karşılık gelir.

[13] https://en.wikipedia.org/wiki/Netflix_Prize

Sınıflandırma

Bağlanım modelleri *kaç tane?* sorusunu ele almak için mükemmel olsa da, birçok sorun bu şablona rahatça uymaz. Örneğin, bir banka mobil uygulamasına çek taraması eklemek istesin. Bu, müşterinin akıllı telefonunun kamerasıyla bir çekin fotoğrafını çekmesini içerir ve uygulamanın görüntüde görülen metni otomatik olarak anlaması gerektirir. Daha dirençli olması için elle yazılmış metni de anlaması gerekir, örneğin el yazması bir karakteri bilindik bir karaktere eşlemek gibi. Bu tür *hangisi?* sorununa *sınıflandırma* denir. Birçok teknik buraya da taşınacak olsa, bağlanım için kullanılanlardan farklı bir algoritma kümesi ile işlem görür.

*Sınıflandırma*da, modelimizin özniteliklere, mesela bir görüntüdeki piksel değerlerine, bakmasını ve ardından örneğimizin bazı ayrık seçenekler kümesi arasından hangi *kategori*ye (aslen *sınıf* olarak adlandırılırlar) ait olduğunu tahmin etmesini istiyoruz. Elle yazılmış rakamlar için, 0 ile 9 arasındaki rakamlara karşılık gelen on sınıfımız olabilir. Sınıflandırmanın en basit şekli, sadece iki sınıf olduğunda, *ikili sınıflandırma* dediğimiz bir problemdir. Örneğin, veri kümemiz hayvanların görüntülerinden oluşabilir ve *etiketlerimiz* {kedi, köpek} sınıfları olabilir. Bağlanımdayken, sayısal bir değer çıkarmak için bir *bağlanımcı* aradık, sınıflandırmada çıktısı tahminlenen sınıf ataması olan bir *sınıflandırıcı* arıyoruz.

Kitap daha teknik hale geldikçe gireceğimiz nedenlerden ötürü, yalnızca kategorik bir atama yapabilen, örneğin "kedi" veya "köpek" çıktısı, bir modeli optimize etmek zor olabilir. Bu tür durumlarda, modelimizi olasılıklar dilinde ifade etmek genellikle daha kolaydır. Bir örneğin öznitelikleri verildiğinde, modelimiz her olası sınıfa bir olasılık atar. Hayvan sınıflandırma örneğimize dönersek, ki burada sınıflar {kedi, köpek}'tir, bir sınıflandırıcı bir görüntü görebilir ve görüntünün bir kedi olma olasılığını 0.9 çıkarabilir. Bu sayıyı, sınıflandırıcının görüntünün bir kediyi gösterdiğinden %90 emin olduğunu söyleyerek yorumlayabiliriz. Öngörülen sınıf için olasılığın büyüklüğü bir çeşit belirsizlik taşır. Bu tek mevcut belirsizlik kavramı değildir ve diğerlerini de daha ileri bölümlerde tartışacağız.

İkiden fazla olası sınıfımız olduğunda, soruna *çok sınıflı sınıflandırma* diyoruz. Yaygın örnekler arasında elle yazılmış karakteri, $\{0, 1, 2, \ldots 9, a, b, c, \ldots\}$, tanıma yer alır. Bağlanım problemleriyle uğraşırken kare hata kayıp fonksiyonunu en aza indirmeye çalışırız; sınıflandırma problemleri için ortak kayıp fonksiyonu, sonraki bölümlerdeki bilgi teorisine giriş ile adını açıklığa kavuşturacağımız *çapraz düzensizlik* (entropi) diye adlandırılır.

En olası sınıfın kararınız için kullanacağınız esas sınıf olmak zorunda olmadığını unutmayın. Arka bahçenizde Fig. 1.3.2'de gösterildiği gibi güzel bir mantar bulduğunuzu varsayın.

Fig. 1.3.2: Ölüm tehlikesi — yemeyin!

Şimdi, bir sınıflandırıcı oluşturduğunuzu ve bir mantarın bir fotoğrafına göre zehirli olup olmadığını tahmin etmek için eğittiğinizi varsayın. Zehir tespit sınıflandırıcısının Fig. 1.3.2'nin zehirli olma olasılığında 0.2 sonucunu verdiğini varsayalım. Başka bir deyişle, sınıflandırıcı, mantarımızın zehirli *olmadığından* %80 emindir. Yine de, yemek için ahmak olmalısın. Çünkü lezzetli bir akşam yemeğinin belirli bir yararı, ondan ölme riski olan %20 değerine değmez. Başka bir deyişle, belirsiz riskin etkisi faydadan çok daha fazladır. Bu nedenle, zarar fonksiyonu olarak maruz kaldığımız beklenen riski hesaplamamız gerekir, yani sonucun olasılığını onunla ilişkili fayda (veya zarar) ile çarpmamız gerekir. Temel olarak, maruz kaldığımız beklenen riski kayıp fonksiyonu olarak hesaplamamız gerekir, yani sonucun olasılığını, bununla ilişkili fayda (veya zarar) ile çarpmamız gerekir. Bu durumda, mantarı yemekten kaynaklanan kayıp $0.2 \times \infty + 0.8 \times 0 = \infty$ olurken, onu çöpe atmanın kaybı $0.2 \times 0 + 0.8 \times 1 = 0.8$ olacaktır. Dikkatimiz haklıydı: Herhangi bir mantarbilimcinin bize söyleyeceği gibi, Fig. 1.3.2'deki mantar aslında zehirlidir.

Sınıflandırma sadece ikili sınıflandırmadan çok daha karmaşık hale gelebilir; çok sınıflı ve hatta çoklu etiketli. Örneğin, hiyerarşilere yönelik bazı değişik sınıflandırmalar vardır. Hiyerarşiler birçok sınıf arasında bazı ilişkilerin olduğunu varsayar. Bu yüzden tüm hatalar eşit değildir - eğer hata yapacaksak, uzak bir sınıf yerine ilgili bir sınıfa yanlış sınıflamayı tercih ederiz. Genellikle buna *hiyerarşik sınıflandırma* denir. İlk örneklerden biri, hayvanları bir hiyerarşide düzenleyen Linnaeus[14]'tır.

Hayvan sınıflandırması durumunda, bir kanişi bir schnauzer (bir tür Alman köpeği) ile karıştırmak o kadar kötü olmayabilir, ancak modelimiz bir dinozor ile bir fino köpeğini karıştırırsa büyük bir ceza ödeyecektir. Hangi hiyerarşinin alakalı olduğu, modeli nasıl kullanmayı planladığınıza bağlı olabilir. Örneğin, çıngıraklı yılanlar ve garter yılanları filogenetik ağaçta yakın olabilir, ancak bir garter yılanını bir çıngıraklı ile karıştırmak ölümcül olabilir.

Etiketleme (Tagging)

Bazı sınıflandırma sorunları, ikili veya çok sınıflı sınıflandırma ayarlarına uyar. Örneğin, kedileri köpeklerden ayırmak için normal bir ikili sınıflandırıcı eğitebiliriz. Bilgisayarla görmenin mevcut durumu göz önüne alındığında, bunu hali-hazırda araçlarla kolayca yapabiliriz. Bununla birlikte, modelimiz ne kadar doğru olursa olsun, sınıflandırıcı *Bremen Mızıkacıları*'nın, Fig. 1.3.3'teki meşhur bir Alman masalındaki dört hayvan, bir görüntüsüyle karşılaştığında kendimizi ufak bir belada bulabiliriz.

[14] https://en.wikipedia.org/wiki/Carl_Linnaeus

Fig. 1.3.3: Bir eşek, bir köpek, bir kedi ve bir horoz.

Gördüğünüz gibi, Fig. 1.3.3'te bir kedi ve bir horoz, bir köpek ve bir eşek ile arka planda bazı ağaçlar var. Nihayetinde modelimizle ne yapmak istediğimize bağlı olarak, bunu ikili bir sınıflandırma problemi olarak ele almak pek anlamlı olmayabilir. Bunun yerine, modele görüntünün bir kediyi, bir köpeği, bir eşeği *ve* bir horozu tasvir ettiğini söyleme seçeneği vermek isteyebiliriz.

Karşılıklı olarak münhasır olmayan sınıfları tahmin etmeyi öğrenme problemine *çoklu etiket sınıflandırması* denir. Otomatik etiketleme sorunları genellikle en iyi çoklu etiket sınıflandırma sorunları olarak tanımlanabilir. Kullanıcıların bir teknoloji blogundaki yayınlara uygulayabilecekleri etiketleri, örneğin "makine öğrenmesi", "teknoloji", "araçlar", "programlama dilleri", "Linux", "bulut bilişim", "AWS" gibi, düşünün. Tipik bir makalede 5–10 etiket uygulanabilir, çünkü bu kavramlar birbiriyle ilişkilidir. "Bulut bilişim" hakkındaki gönderilerin "AWS"den bahsetmesi muhtemeldir ve "makine öğrenmesi" ile ilgili gönderiler de "programlama dilleri" ile ilgili olabilir.

Ayrıca, makalelerin doğru etiketlenmesinin önemli olduğu biyomedikal literatürle uğraşırken bu tür bir sorunla uğraşmak zorundayız, çünkü bu araştırmacıların literatürde kapsamlı incelemeler yapmasına izin veriyor. (Amerikan) Ulusal Tıp Kütüphanesi'nde, bir dizi profesyonel yorumlayıcı, PubMed'de endekslenen her makaleyi, kabaca 28000 etiketlik bir koleksiyon olan MeSH'den ilgili terimlerle ilişkilendirmek için gözden geçiriyor. Bu zaman alıcı bir süreçtir ve yorumlayıcıların genellikle arşivlemesi ve etiketlemesi arasında bir yıllık bir gecikme vardır. Makine öğrenmesi burada, her makaleye uygun bir manuel (elle) incelemeye sahip oluncaya kadar geçici etiketler sağlamak için kullanılabilir. Gerçekten de, birkaç yıl boyunca, BioASQ organizasyonu tam olarak bunu yapmak için yarışmalar düzenledi[15].

[15] http://bioasq.org/

Arama

Bazen her örneği bir kovaya veya gerçek bir değere atamak istemiyoruz. Bilgi getirimi alanında, bir dizi maddeye bir sıralama uygulamak istiyoruz. Örneğin, web aramasını ele alalım. Hedef belirli bir sayfanın bir sorgu için alakalı olup olmadığını belirlemekten daha ziyade, birçok arama sonucundan hangisinin belirli bir kullanıcı için en alakalı olduğunu belirlemektir. Alakalı arama sonuçlarının sırasına gerçekten önem veriyoruz ve öğrenme algoritmamızın daha geniş bir gruptan sıralanmış alt kümeleri üretmesi gerekiyor. Başka bir deyişle, alfabeden ilk 5 harfi üretmemiz istenirse, "A B C D E" ve "C A B E D" döndürme arasında bir fark vardır. Sonuç kümesi aynı olsa bile, küme içindeki sıralama önemlidir.

Bu soruna olası bir çözüm, önce kümedeki her bir öğeye, ona karşılık gelen bir alaka puanı atamak ve daha sonra en yüksek dereceli öğeleri almaktır. PageRank[16], Google arama motorunun arkasındaki esas gizli bileşen, böyle bir puanlama sisteminin erken bir örneğiydi, ancak tuhaf tarafı gerçek sorguya bağlı değildi. Burada, ilgili öğelerin kümesini tanımlamak için basit bir alaka filtresine ve ardından sorgu terimini içeren sonuçları sıralamak için PageRank'e güveniyorlardı. Günümüzde arama motorları, sorguya bağlı alaka düzeyi puanlarını belirlemek için makine öğrenmesi ve davranışsal modeller kullanmaktadır. Sadece bu konuyla ilgili akademik konferanslar vardır.

Tavsiye Sistemleri

Tavsiye sistemleri, arama ve sıralama ile ilgili başka bir problem ailesidir. Amaç, kullanıcıya ilgili bir dizi öğeyi görüntülemek olduğu sürece benzer problemlerdir. Temel fark, tavsiye sistemleri bağlamında, sözkonusu kullanıcılara *kişiselleştirme* vurgusu yapılmasıdır. Mesela, film önerilerinde, bir bilim kurgu hayranı için sonuç sayfası ile Peter Sellers komedileri uzmanının sonuç sayfası önemli ölçüde farklılıklar gösterebilir. Perakende satış ürünleri, müzik ve haber önerileri gibi diğer öneri gruplarında da benzer sorunlar ortaya çıkar.

Bazı durumlarda, müşteriler belirli bir ürünü ne kadar sevdiklerini bildiren açık geri bildirimler sağlar (Ör. Amazon, IMDB, Goodreads, vb. üzerindeki ürün puanları ve incelemeleri). Diğer bazı durumlarda, örneğin bir müzik çalma listesindeki parçaları atlarken, kullanıcılar memnuniyetsizliği de, şarkının o anki bağlamında uygunsuz olduğunu da gösterebilecek (gizli) örtük geri bildirim sağlarlar. En basit formülasyonlarda, bu sistemler bir kullanıcı ve bir ürün göz önüne alındığında tahmini bir derecelendirme veya satın alma olasılığı gibi bir skoru tahmin etmek üzere eğitilir.

Böyle bir model göz önüne alındığında, herhangi bir müşteri için, en yüksek puanları olan ve daha sonra müşteriye önerilebilecek nesneler kümesini bulabiliriz. Üretimdeki sistemler oldukça ileri düzeydedir ve bu puanları hesaplarken ayrıntılı kullanıcı etkinliği ve öğenin özelliklerini dikkate alır. Fig. 1.3.4 imgesi, yazarın tercihlerini yakalamak için ayarlanan kişiselleştirme algoritmalarına dayanarak Amazon tarafından önerilen derin öğrenme kitaplarına bir örnektir.

[16] https://en.wikipedia.org/wiki/PageRank

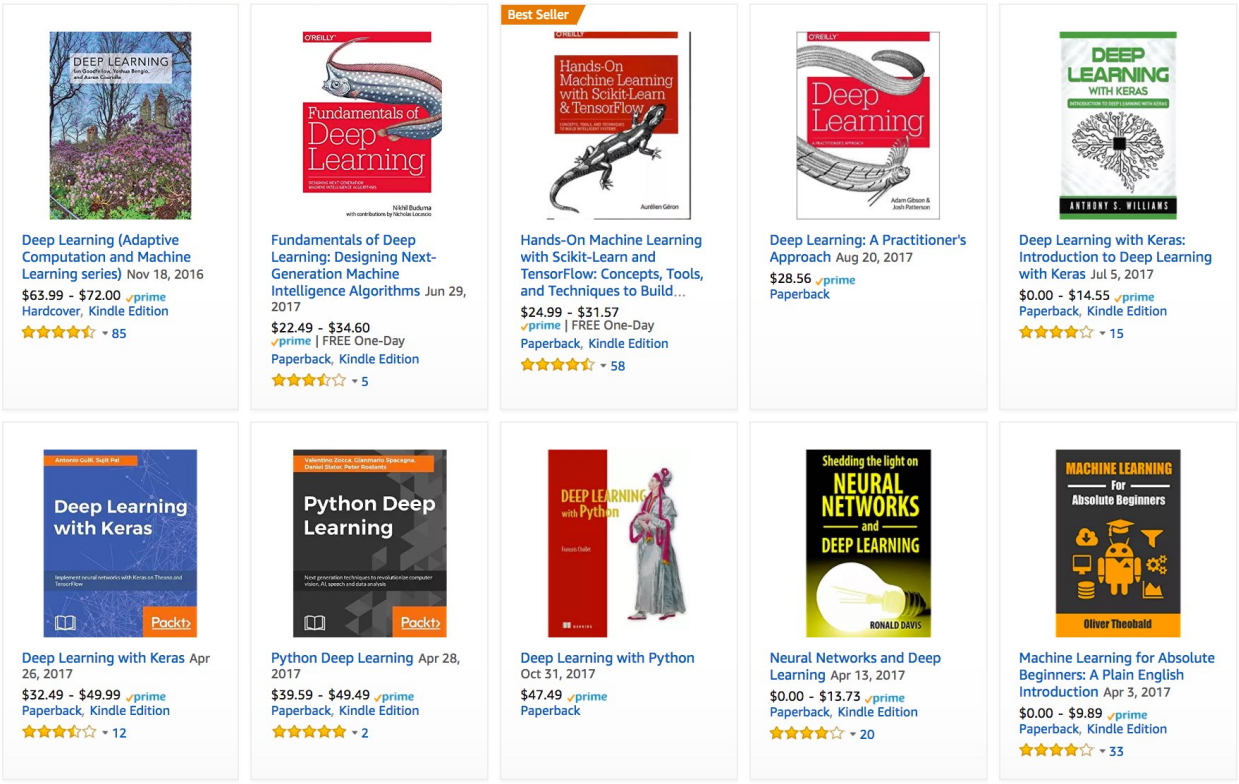


Fig. 1.3.4: Amazon tarafından önerilen derin öğrenme kitapları.

Muazzam ekonomik değerlerine rağmen, tahminci modeller üzerine saf olarak inşa edilmiş tavsiye sistemleri bazı ciddi kavramsal kusurlara maruz kalmaktadırlar. Öncelikle sadece *sansürlü geri bildirim* gözlemliyoruz. Kullanıcılar, özellikle, güçlü bir şekilde hissettikleri filmleri derecelendirir. Örneğin beş puanlı ölçeklerde, öğelerin çok sayıda beş ve bir yıldız derecelendirmesi aldığını, ancak dikkat çekici derecede az üç yıldızlı derecelendirme olduğunu fark edebilirsiniz. Ayrıca, mevcut satın alma alışkanlıkları genellikle şu anda mevcut olan tavsiye algoritmasının bir sonucudur, ancak öğrenme algoritmaları bu ayrıntıyı her zaman dikkate almazlar. Bu nedeni, bir geri bildirim döngüsünün oluşmasının mümkün olmasıdır: Bir tavsiye sistemi, daha sonra daha iyi olması için (daha büyük miktarda satın alımlar nedeniyle), alınan bir öğeyi tercihli olarak yukarı iter ve daha da sık tavsiye edilmesine neden olur. Sansür, teşvikler ve geri bildirim döngüleri ile nasıl başa çıkılacağı gibi bu tarz ilgili sorunların birçoğu önemli açık araştırma konularıdır.

Dizi Öğrenimi

Şimdiye kadar, sabit sayıda girdimiz olan ve sabit sayıda çıktı üreten sorunlara baktık. Örneğin ev fiyatlarını sabit bir dizi öznitelikten tahmin ettik: Metrekare alanları, yatak odası sayısı, banyo sayısı, şehir merkezine yürüme süresi. Ayrıca, bir görüntüyü (sabit boyutlu), sabit sayıda sınıfın hangi birine ait olduğu tahmin eden olasılıklarla eşlemeyi veya bir kullanıcı kimliği ve ürün kimliği alarak bir yıldız derecelendirmesi tahmin etmeyi tartıştık. Bu durumlarda, sabit uzunluklu girdimizi bir çıktı üretmek için modele beslediğimizde, model gördüklerini hemen unutur.

Girdilerimizin hepsi aynı boyutlara sahipse ve birbirini takip eden girdilerin birbirleriyle hiçbir ilgisi yoksa, bu iyi olabilir. Ancak video parçalarıyla nasıl başa çıkabiliriz? Bu durumda, her parça farklı sayıda çerçeveden oluşabilir. Ayrıca önceki veya sonraki çerçeveleri dikkate alırsak, her çerçevede neler olup bittiğine dair tahminimiz çok daha güçlü olabilir. Aynı şey dil için de geçerli.

Popüler bir derin öğrenme sorunu, makine çevirisidir: Bazı kaynak dilde cümleleri alma ve başka bir dilde çevirilerini tahmin etme görevidir.

Bu problemler tıpta da görülür. Yoğun bakım ünitesindeki hastaları izlemek ve önümüzdeki 24 saat içinde ölüm riskleri belli bir eşiği aşarsa, uyarıcıları tetiklemek için bir model isteyebiliriz. Bu modelin her saatteki hasta geçmişi hakkında bildiği her şeyi atmasını ve sadece en son ölçümlere dayanarak tahminlerini yapmasını kesinlikle istemeyiz.

Bu problemler makine öğrenmesinin en heyecan verici uygulamaları arasındadır ve *dizi öğrenmenin* örnekleridir. Girdilerin dizilerini almak veya çıktı dizilerini (veya her ikisini) saçmak için bir modele ihtiyaç duyarlar. Özellikle, *diziden diziye öğrenme*, girdinin ve çıktının ikisinin de değişken uzunluklu olduğu, makine çevirisi ve sözlü hitaptan metine dökme benzeri problemleri içerir. Bütün dizi dönüştürme türlerini burada düşünmek imkansız olsa da, aşağıdaki özel durumlardan bahsetmeye değerdir.

Etiketleme ve Ayrıştırma. Bu, nitelikleri olan bir metin dizisine açıklama eklemeyi içerir. Başka bir deyişle, girdi ve çıktıların sayısı aslında aynıdır. Örneğin, fiillerin ve öznelerin nerede olduğunu bilmek isteyebiliriz. Alternatif olarak, hangi kelimelerin adlandırılmış varlıklar olduğunu bilmek isteyebiliriz. Genel olarak amaç, bir açıklama almak için yapısal ve dilbilgisel varsayımlara dayalı olarak metni ayrıştırmak ve açıklama eklemektir. Bu aslında olduğundan daha karmaşıkmış gibi geliyor. Aşağıdaki çok basit bir örnek, hangi kelimelerin adlandırılmış varlıkları ifade ettiğini belirten etiketleri, bir cümleye açıklama olarak eklemeyi gösterir.

```
Tom'un Washington'da Sally ile akşam yemeği var.
Var      Var         Var  -    -     -      -
```

Otomatik Konuşma Tanıma. Konuşma tanımada, girdi dizisi bir hoparlörün ses kaydıdır (Fig. 1.3.5 içinde gösterilen) ve çıktı konuşmacının söylediklerinin metne dökümüdür. Buradaki zorluk, metinden çok daha fazla ses karesi çerçevesi olması (ses genellikle 8kHz veya 16kHz'de örneklenmiştir), yani ses ve metin arasında 1:1 karşılık olmamasıdır, çünkü binlerce sesli örnek tek bir sözlü kelimeye karşılık gelebilir. Bunlar, çıktının girdiden çok daha kısa olduğu diziden diziye öğrenme problemleridir.

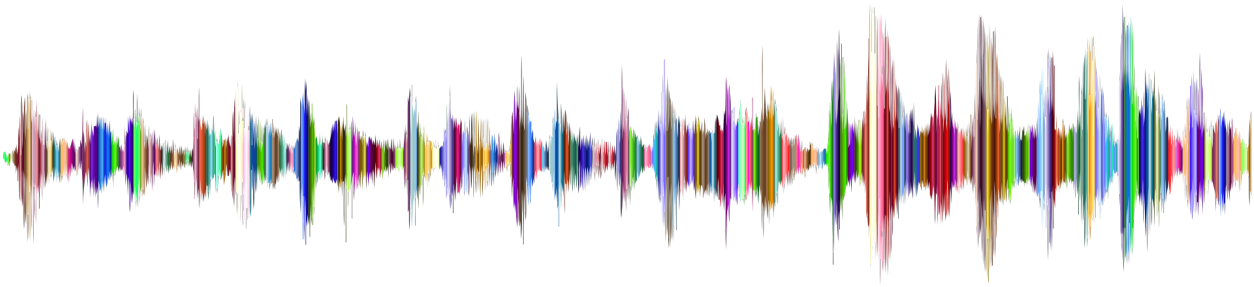


Fig. 1.3.5: Bir ses kaydından `-D-e-e-p- L-ea-r-ni-ng-`

Metinden Konuşmaya. Bu, otomatik konuşma tanımanın tersidir. Başka bir deyişle, girdi metindir ve çıktı bir ses dosyasıdır. Bu durumda, çıktı girdiden çok daha uzun olur. İnsanların kötü bir ses dosyasını tanıması kolay olsa da, bu bilgisayarlar için o kadar da bariz değildir.

Makine Çevirisi. Karşılık gelen girdi ve çıktıların aynı sırada (hizalamadan sonra) gerçekleştiği konuşma tanıma durumundan farklı olarak, makine çevirisinde, sırayı ters çevirme hayati önem taşıyabilir. Başka bir deyişle, bir diziyi diğerine dönüştürürken, ne girdi ve çıktıların sayısı ne de karşılık gelen veri örneklerinin sırası aynı kabul edilmektedir. Almanların fiilleri cümle sonuna yerleştirme eğiliminin aşağıdaki açıklayıcı örneğini düşünün.

```
Almanca:          Haben Sie sich schon dieses grossartige Lehrwerk angeschaut?
İngilizce:        Did you already check out this excellent tutorial?
Yanlış Hizalama:  Did you yourself already this excellent tutorial looked-at?
```

İlgili birçok sorun diğer öğrenme görevlerinde ortaya çıkar. Örneğin, bir kullanıcının bir Web sayfasını okuma sırasını belirlemek iki boyutlu bir düzen analizi sorunudur. Diyalog sorunları her türlü ek karmaşıklığı ortaya çıkarır, bir sonrasında ne söyleneceğini belirlemede, gerçek dünya bilgisini ve uzun zamansal mesafelerde konuşmanın önceki durumunu dikkate almayı gerektirmek gibi. Bunlar aktif bir araştırma alanlarıdır.

1.3.2 Gözetimsiz ve Kendi Kendine Gözetimli Öğrenme

Şimdiye kadarki tüm örnekler gözetimli öğrenme, yani, modeli hem öznitelikleri hem de karşılık gelen etiket değerleri içeren dev bir veri kümesi ile beslediğimiz durumlarla ilgilidir. Gözetimli öğreniciyi son derece uzmanlaşmış bir işe ve son derece bunaltıcı bir patrona sahip olmak gibi düşünebilirsiniz. Patron omzunuzun üzerinden bakar ve siz durumlardan eylemlere eşlemeyi öğrenene kadar her durumda tam olarak ne yapacağınızı söyler. Böyle bir patron için çalışmak oldukça tatsızdır. Öte yandan, bu patronu memnun etmek kolaydır. Deseni mümkün olduğunca çabuk tanır ve eylemlerini taklit edersiniz.

Tamamen zıt bir şekilde, ne yapmanızı istediğini bilmeyen bir patron için çalışmak sinir bozucu olabilir. Ancak, bir veri bilimcisi olmayı planlıyorsanız, buna alışsanız iyi olur. Patron size sadece dev bir veri dökümü verebilir ve *onunla veri bilimi yapmanızı söyleyebilir!* Bu kulağa belirsiz geliyor, çünkü öyle. Bu sorun sınıfına *gözetimsiz öğrenme* diyoruz ve sorabileceğimiz soruların türü ve sayısı yalnızca yaratıcılığımızla sınırlıdır. Daha sonraki bölümlerde denetimsiz öğrenme tekniğini ele alacağız. Şimdilik iştahınızı hafifletmek için sormak isteyebileceğiniz birkaç sorudan aşağıda bahsediyoruz:

- Verileri doğru bir şekilde özetleyen az sayıda ilk örnek (prototip) bulabilir miyiz? Bir dizi fotoğraf verildiğinde, onları manzara fotoğrafları, köpek resimleri, bebekler, kediler ve dağ zirveleri olarak gruplandırabilir miyiz? Benzer şekilde, kullanıcıların göz atma etkinlikleri koleksiyonu göz önüne alındığında, onları benzer davranışa sahip kullanıcılara ayırabilir miyiz? Bu sorun genellikle *kümeleme* olarak bilinir.
- Verilerin ilgili özelliklerini doğru bir şekilde yakalayan az sayıda parametre bulabilir miyiz? Bir topun yörüngeleri, topun hızı, çapı ve kütlesi ile oldukça iyi tanımlanmıştır. Terziler, kıyafetlerin uyması amacıyla insan vücudunun şeklini oldukça doğru bir şekilde tanımlayan az sayıda parametre geliştirmiştir. Bu problemlere *altuzay tahmini* denir. Bağımlılık doğrusal ise, buna *ana bileşen analizi* denir.
- (Keyfi olarak yapılandırılmış) Nesnelerin Öklid uzayında sembolik özelliklerinin iyi eşleştirilebileceği bir temsili var mı? Bu varlıkları ve onların ilişkilerini, "Roma" − "İtalya" + "Fransa" = "Paris" gibi, tanımlamak için kullanılabilir.
- Gözlemlediğimiz verilerin çoğunun temel nedenlerinin bir açıklaması var mı? Örneğin, konut fiyatları, kirlilik, suç, yer, eğitim ve maaşlar ile ilgili demografik verilerimiz varsa, bunların deneysel verilerine dayanarak nasıl ilişkili olduğunu bulabilir miyiz? *Nedensellik* ve *olasılıksal grafik modeller* ile ilgili alanlar bu sorunu ele almaktadır.
- Gözetimsiz öğrenmedeki bir diğer önemli ve heyecan verici gelişme, *çekişmeli üretici ağların* ortaya çıkmasıdır. Bunlar bize verileri, görüntüler ve ses gibi karmaşık yapılandırılmış verileri bile, sentezlemek için yöntemsel bir yol sunar. Temel istatistiksel mekanizmalar, gerçek ve sahte verilerin aynı olup olmadığını kontrol etmek için kullanılan testlerdir.

Denetimsiz öğrenmenin bir biçimi olarak *kendi kendine gözetimli öğrenme*, eğitimde gözetim sağlamak için etiketlenmemiş verilerden yararlanır; örneğin, verilerin saklanan bir kısmının diğer bölümleri kullanarak tahmin edilmesi gibi. Metinler için, herhangi bir etiketleme çabası olmadan büyük dökümdeki etrafındaki kelimeleri (bağlamları) kullanarak rastgele maskelenmiş kelimeleri tahmin ederek "boşlukları doldurmak" için modeller eğitebiliriz (Devlin *et al.*, 2018)! İmgeler için, aynı imgenin iki kırpılmış bölgesi arasındaki göreli konumu söylemek için modelleri eğitebiliriz (Doersch *et al.*, 2015). Bu iki kendi kendine gözetimli öğrenme örneğinde, olası sözcükleri ve göreli konumları tahmin etmeye yönelik eğitim modellerinin her ikisi de (gözetimli öğrenme) sınıflandırma görevleridir.

1.3.3 Bir Ortamla Etkileşim

Şimdiye kadar, verilerin gerçekte nereden geldiğini veya bir makine öğrenmesi modeli bir çıktı oluşturduğunda gerçekte ne olduğunu tartışmadık. Çünkü gözetimli öğrenme ve gözetimsiz öğrenme bu konuları çok karmaşık bir şekilde ele almaz. Her iki durumda da, büyük bir veri yığınını önceden alıyoruz, ardından bir daha çevre ile etkileşime girmeden desen tanıma makinelerimizi harekete geçiriyoruz. Tüm öğrenme, algoritma ortamdan koparıldıktan sonra gerçekleştiği için, buna bazen *çevrimdışı öğrenme* denir. Gözetimli öğrenme için bir ortamdan veri toplamayı düşünürsek süreç şuna benzer Fig. 1.3.6.

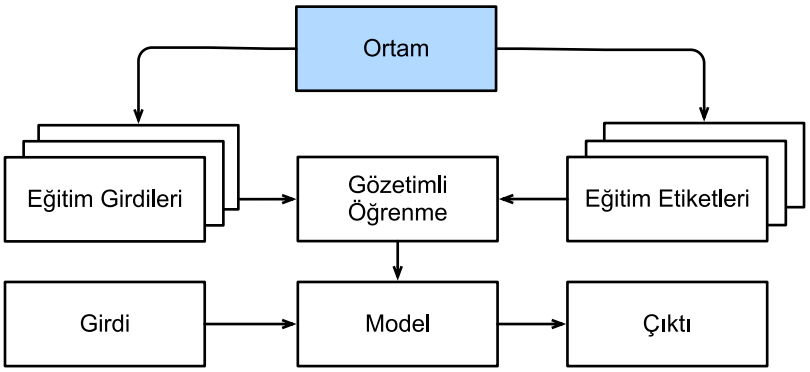


Fig. 1.3.6: Bir ortamdan gözetimli öğrenme için veri toplama.

Çevrimdışı öğrenmenin bu basitliğinin cazibesi vardır. Bunun olumlu tarafı, diğer sorunlardan herhangi bir dikkat dağılımı olmadan sadece örüntü tanıma konusu ile tek başına ilgilenebiliriz. Ancak olumsuz tarafı, formülasyonun oldukça kısıtlayıcı olmasıdır. Daha hırslıysanız ya da Asimov'un Robot Serisi'ni okuyarak büyüdüyseniz, sadece tahminler yapmakla kalmayıp, dünyada hareket edebilecek yapay zeka botları hayal edebilirsiniz. Sadece modelleri değil, akıllı *etmenleri (ajanları)* de düşünmek istiyoruz. Bu, sadece tahminler yapmakla kalmayıp, *eylemleri* seçmeyi de düşünmemiz gerektiği anlamına gelir. Dahası, öngörülerin aksine, eylemler aslında ortamı etkiler. Akıllı bir etmen eğitmek istiyorsak, eylemlerinin etmenin gelecekteki gözlemlerini nasıl etkileyebileceğini hesaba katmalıyız.

Bir ortam ile etkileşimi dikkate almak, bir dizi yeni modelleme sorusunu açar. Aşağıda birkaç örnek görebiliriz.

- Ortam önceden ne yaptığımızı hatırlıyor mu?
- Ortam bize bir konuşma tanıyıcıya metin okuyan bir kullanıcı gibi yardım etmek istiyor mu?
- Ortam bizi yenmek mi istiyor, yani spam filtreleme (spam göndericilere karşı) veya oyun oynama (rakiplere karşı) gibi rakip bir ortam mı?

- Ortam bizi umursumuyor mu?
- Ortam değişen dinamiklere sahip mi? Örneğin, gelecekteki veriler her zaman geçmiştekine benziyor mu, ya da doğal olarak veya otomatik araçlarımıza yanıt olarak zaman içinde değişiyor mu?

Eğitim ve test verileri farklı olduğunda, bu son soru *dağılım kayması* sorununu gündeme getirmektedir. Bu bir öğretim üyesi tarafından hazırlanan yazılı sınava girerken yaşadığımız bir problemdir, çünkü ödevler asistanlar tarafından oluşturulmuştur. Sonrasında, bir çevreyle etkileşimi açıkça dikkate alan bir düzen olan pekiştirmeli öğrenmeyi kısaca anlatacağız.

1.3.4 Pekiştirmeli Öğrenme

Bir ortamla etkileşime giren ve eylemler yapan bir etmen geliştirmek için makine öğrenmesini kullanmakla ilgileniyorsanız, muhtemelen *pekiştirmeli öğrenme* konusuna odaklanacaksınız. Bu, robotik, diyalog sistemleri ve hatta video oyunları için yapay zeka (YZ) geliştirme uygulamalarını içerebilir. Derin öğrenme ağlarını pekiştirmeli öğrenme problemlerine uygulayan *derin pekiştirmeli öğrenme* popülerlik kazanmıştır. Bu atılımda yalnızca görsel girdileri kullanarak Atari oyunlarında insanları yenen derin Q-ağ (Q-network) ve Go oyunu dünya şampiyonunu tahtından indiren AlphaGo programı iki önemli örnektir.

Pekiştirmeli öğrenmede, bir etmenin bir dizi zaman adımı üzerinden bir ortam ile etkileşime girdiği, çok genel bir sorun ifade edilir. Her bir zaman adımında, etmen ortamdan birtakım *gözlem* alır ve daha sonra bir mekanizma (bazen çalıştırıcı -aktüatör- olarak da adlandırılır) aracılığıyla çevreye geri iletilecek bir eylemi seçmelidir. Son olarak, etmen ortamdan bir ödül alır. Bu süreç Fig. 1.3.7 şeklinde gösterilmektedir. Etmen daha sonra bir gözlem alır ve bir sonraki eylemi seçer, vb. Bir pekiştirmeli öğrenme etmeninin davranışı bir politika tarafından yönetilir. Kısacası, bir *politika*, sadece, çevrenin gözlemlerinden eylemlere eşlenen bir fonksiyondur. Pekiştirmeli öğrenmenin amacı iyi bir politika üretmektir.

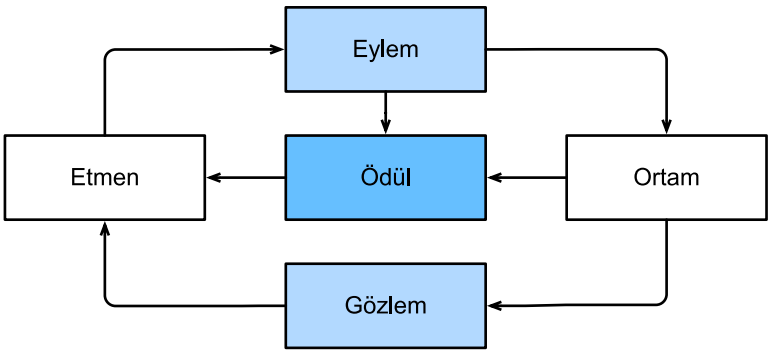


Fig. 1.3.7: Pekiştirmeli öğrenme ve çevre arasındaki etkileşim.

Pekiştirmeli öğrenme çerçevesinin genelliğini abartmak zordur. Örneğin, herhangi bir gözetimli öğrenme problemini bir pekiştirmeli öğrenme problemine dönüştürebiliriz. Diyelim ki bir sınıflandırma problemimiz var. Her sınıfa karşılık gelen bir eylem ile bir pekiştirmeli öğrenme etmeni oluşturabiliriz. Daha sonra, orijinal gözetimli öğrenme probleminin yitim fonksiyonuna tamamen eşit bir ödül veren bir ortam yaratabiliriz.

Bununla birlikte, pekiştirmeli öğrenme, gözetimli öğrenmenin yapamadığı birçok sorunu da ele alabilir. Örneğin, gözetimli öğrenmede her zaman eğitim girdisinin doğru etiketle ilişkilendirilmesini bekleriz. Ancak pekiştirmeli öğrenmede, her gözlem için çevrenin bize en uygun

eylemi söylediğini varsaymıyoruz. Genel olarak, sadece bir ödül alırız. Dahası, ortam bize hangi eylemlerin ödüle yol açtığını bile söylemeyebilir.

Örneğin satranç oyununu düşünün. Tek gerçek ödül sinyali, oyunun sonunda ya kazandığımızda 1, ya da kaybettiğimizde -1 diye gelir. Bu yüzden pekiştirmeli öğreniciler kredi atama problemi ile ilgilenmelidir: Bir sonuç için hangi eylemlerin beğeni toplayacağını veya suçlanacağını belirleme. Aynı şey 11 Ekim'de terfi alan bir çalışan için de geçerli. Bu terfi büyük olasılıkla bir önceki yılda itibaren çok sayıda iyi seçilmiş eylemi yansıtmaktadır. Gelecekte daha fazla terfi almak için zaman boyunca hangi eylemlerin terfiye yol açtığını bulmak gerekir.

Pekiştirmeli öğreniciler kısmi gözlenebilirlik sorunuyla da uğraşmak zorunda kalabilirler. Yani, mevcut gözlem size mevcut durumunuz hakkında her şeyi söylemeyebilir. Diyelim ki bir temizlik robotu kendini bir evdeki birçok aynı dolaptan birinde sıkışmış buldu. Robotun kesin yerini (ve dolayısıyla durumunu) bulmak, dolaba girmeden önce önceki gözlemlerini dikkate almayı gerektirebilir.

Son olarak, herhangi bir noktada, pekiştirmeli öğreniciler iyi bir politika biliyor olabilir, ancak etmenin hiç denemediği daha iyi birçok politika olabilir. Pekiştirmeli öğrenici ya sürekli olarak politika olarak şu anda bilinen en iyi stratejiyi *sömürmeyi* veya stratejiler alanını *keşfetmeyi*, yani potansiyel bilgi karşılığında kısa vadede ödülden vazgeçmeyi, seçmelidir.

Genel pekiştirmeli öğrenme sorunu çok genel bir düzenlemedir. Eylemler sonraki gözlemleri etkiler. Yalnızca seçilen eylemlere denk gelen ödüller gözlemlenir. Ortam tamamen veya kısmen gözlemlenebilir. Tüm bu karmaşıklığı bir kerede hesaplamak araştırmacılardan çok fazla şey beklemek olabilir. Dahası, her pratik sorun bu karmaşıklığın tamamını sergilemez. Sonuç olarak, araştırmacılar pekiştirmeli öğrenme sorunlarının bir dizi özel vakasını incelemişlerdir.

Ortam tam olarak gözlemlendiğinde, pekiştirmeli öğrenme sorununa *Markov Karar Süreci* (MKS) diyoruz. Durum önceki eylemlere bağlı olmadığında, probleme *bağlamsal bir kollu kumar makinesi sorunu* diyoruz. Durum yoksa, sadece başlangıçta ödülleri bilinmeyen bir dizi kullanılabilir eylem varsa, bu problem klasik *çok kollu kumar makinesi problemidir*.

1.4 Kökenler

Şu ana dek makine öğrenmesinin çözebileceği sorunların küçük bir alt grubunu inceledik. Çok çeşitli makine öğrenmesi sorunları için derin öğrenme, bunları çözmede güçlü araçlar sunar. Birçok derin öğrenme yöntemi yeni buluşlar olmasına rağmen, veri ve sinir ağları (birçok derin öğrenme modelinin adı) ile programlamanın temel fikri yüzyıllardır çalışılmıştır. Aslında, insanlar uzun süredir verileri analiz etme ve gelecekteki sonuçları tahmin etme arzusunu taşıyorlar ve doğa bilimlerinin çoğunun kökleri buna dayanıyor. Örneğin, Bernoulli dağılımı Jacob Bernoulli (1655–1705)[17] ile isimlendirildi ve Gaussian dağılımı Carl Friedrich Gauss (1777–1855)[18] tarafından keşfedildi. Örneğin, bugün hala sigorta hesaplamalarından tıbbi teşhislere kadar sayısız problem için kullanılan en düşük kareler ortalaması algoritmasını icat etti. Bu araçlar doğa bilimlerinde deneysel bir yaklaşıma yol açmıştır — örneğin, Ohm'un bir dirençteki akım ve voltajla ilgili yasası doğrusal bir modelle mükemmel bir şekilde tanımlanır.

Orta çağlarda bile, matematikçilerin tahminlerde keskin bir sezgileri vardı. Örneğin, Jacob Köbel (1460–1533)[19]'ün geometri kitabı ortalama ayak uzunluğunu elde etmek için 16 erkek yetişkinin ayak uzunluğunu ortalamayı göstermektedir.

[17] https://en.wikipedia.org/wiki/Jacob_Bernoulli
[18] https://en.wikipedia.org/wiki/Carl_Friedrich_Gauss
[19] https://www.maa.org/press/periodicals/convergence/mathematical-treasures-jacob-kobels-geometry

Fig. 1.4.1: Ayak uzunluğunu tahmin etme.

Fig. 1.4.1 bu tahmincinin nasıl çalıştığını gösterir. 16 yetişkin erkekten kiliseden ayrılırken bir hizada dizilmeleri istendi. Daha sonra toplam uzunlukları günümüzdeki 1 ayak (foot) birimine ilişkin bir tahmin elde etmek için 16'ya bölündü. Bu "algoritma" daha sonra biçimsiz ayaklarla başa çıkmak için de düzenlendi - sırasıyla en kısa ve en uzun ayakları olan 2 adam gönderildi, sadece geri kalanların ortalaması alındı. Bu, kırpılmış ortalama tahmininin en eski örneklerinden biridir.

İstatistik gerçekte verilerin toplanması ve kullanılabilirliği ile başladı. Dev isimlerden biri Ronald Fisher (1890–1962)[20], istatistik teorisine ve aynı zamanda onun genetikteki uygulamalarına önemli katkıda bulundu. Algoritmalarının çoğu (doğrusal ayırtaç analizi gibi) ve formülü (Fisher bilgi matrisi gibi) günümüzde hala sık kullanılmaktadır. Aslında 1936'da kullanıma açtığı İris veri kümesi bile, hala bazen makine öğrenmesi algoritmalarını göstermek için kullanılıyor. O aynı zamanda, veri biliminin ahlaki olarak şüpheli kullanımının, endüstride ve doğa bilimlerinde verimli kullanımı kadar uzun ve kalıcı bir geçmişi olduğunu hatırlatan bir öjeni (doğum ile kalıtımsal olarak istenen özelliklere sahip bireylerin üremesine çalışan bilim dalı) savunucusuydu.

Makine öğrenmesi için ikinci bir etki, Claude Shannon, (1916–2001)[21] aracılığıyla bilgi teorisi ve Alan Turing (1912–1954)[22] aracılığıyla hesaplama teorisinden geldi. Turing, ünlü makalesinde, *Computing Machinery and Intelligence* (Turing, 1950), "Makineler düşünebilir mi?" diye sordu. Turing testi olarak tanımladığı testte, bir insan değerlendiricinin metin etkileşimlerine dayanarak cevapların bir makineden mi ve bir insan mı geldiğini arasında ayırt etmesinin zor olması durumunda, bir makine *akıllı* kabul edilebilir.

Nörobilim ve psikolojide de başka bir etki bulunabilir. Sonuçta, insanlar açıkça akıllı davranış

[20] https://en.wikipedia.org/wiki/Ronald_Fisher
[21] https://en.wikipedia.org/wiki/Claude_Shannon
[22] https://en.wikipedia.org/wiki/Alan_Turing

sergilerler. Bu nedenle, birinin sadece bu beceriyi açıklayıp tersine mühendislik yapıp yapamayacağını sormak mantıklıdır. Bu şekilde esinlenen en eski algoritmalardan biri Donald Hebb (1904–1985)[23] tarafından formüle edildi. Çığır açan *Davranış Örgütlenmesi* (Hebb and Hebb, 1949) adlı kitabında, nöronların pozitif pekiştirme ile öğrendiklerini ileri sürdü. Bu Hebbian öğrenme kuralı olarak bilindi. Bu Rosenblatt'ın algılayıcı öğrenme algoritmasının ilk örneğidir ve bugün derin öğrenmeyi destekleyen birçok rasgele eğim inişi (stochastic gradient descent) algoritmasının temellerini atmıştır: Sinir ağındaki parametrelerin iyi ayarlarını elde etmek için arzu edilen davranışı güçlendir ve istenmeyen davranışı zayıflat.

Sinir ağlarına adını veren şey biyolojik ilhamdır. Yüzyılı aşkın bir süre (Alexander Bain, 1873 ve James Sherrington, 1890 modellerine kadar geri gider) araştırmacılar, etkileşen nöron ağlarına benzeyen hesaplama devreleri oluşturmaya çalıştılar. Zamanla, biyolojinin yorumu daha az gerçek hale geldi, ancak isim yapıştı. Özünde, bugün çoğu ağda bulunabilecek birkaç temel ilke yatmaktadır:

- Genellikle *katmanlar* olarak adlandırılan doğrusal ve doğrusal olmayan işlem birimlerinin değişimi.
- Tüm ağdaki parametreleri bir kerede ayarlamak için zincir kuralının (*geri yayma (backpropagation)* olarak da bilinir) kullanımı.

İlk hızlı ilerlemeden sonra, sinir ağlarındaki araştırmalar 1995'ten 2005'e kadar yavaşladı. Bunun temel iki nedeni vardır. İlk olarak bir ağın eğitiminin hesaplama maliyeti çok pahalıdır. Rasgele erişim belleği (RAM) geçen yüzyılın sonunda bol miktarda bulunurken, hesaplama gücü azdı. İkincisi, veri kümeleri nispeten küçüktü. Aslında, Fisher'in 1932'deki Iris veri kümesi algoritmaların etkinliğini test etmek için popüler bir araçtı. MNIST veri kümesi, 60000 el yazısı rakam ile devasa sayılırdı.

Veri ve hesaplama kıtlığı göz önüne alındığında, çekirdek (kernel) yöntemleri, karar ağaçları ve grafik modeller gibi güçlü istatistiksel araçlar deneysel olarak daha üstün oldu. Sinir ağlarından farklı olarak, eğitim için haftalar gerektirmediler ve güçlü teorik garantilerle öngörülebilir sonuçlar verdiler.

1.5 Derin Öğrenmeye Giden Yol

Bunların çoğu, yüz milyonlarca kullanıcıya çevrimiçi hizmet veren şirketlerin gelişi, ucuz ve yüksek kaliteli sensörlerin yayılması, ucuz veri depolama (Kryder yasası) ve ucuz hesaplama (Moore yasası) maliyeti ve özellikle bilgisayar oyunları için tasarlanan GPU'ların kullanımı ile değişti. Aniden, hesaplamaya elverişli görünmeyen algoritmalar ve modeller bariz hale geldi (veya tam tersi). Bu en iyi şekilde Chapter 1.5 içinde gösterilmektedir.

[23] https://en.wikipedia.org/wiki/Donald_O._Hebb

On Yıl	Ver i Küm esi	Bel lek	San iye de kay an vir gül lü (Fl oat ing) say ı hes apl ama sı
1970	100 (İr is)	1 KB	100 KF (In tel 808 0)
1980	1 K (Bo sto n'd aki ev fiy atl arı)	100 KB	1 MF (In tel 801 86)
1990	10 K (op tik kar akt er tan ıma)	10 MB	10 MF (In tel 804 86)
2000	10 M (we b say fal arı)	100 MB	1 GF (In tel Cor e)
2010	10 G (re kla m)	1 GB	1 TF (Nv idi a C20 50)
2020	1 T (so sya l ağ)	100 GB	1 PF (Nv idi a DGX -2)

Table: Veri kümesi ve bilgisayar belleği ve hesaplama gücü

Rasgele erişim belleğinin veri büyümesine ayak uyduramadığı açıktır. Aynı zamanda, hesaplama gücündeki artış mevcut verilerinkinden daha fazladır. Bu, istatistiksel işlemlerin bellekte daha verimli hale dönüşmesi (bu genellikle doğrusal olmayan özellikler ekleyerek elde edilir) ve aynı zamanda, artan bir hesaplama bütçesi nedeniyle bu parametreleri optimize etmek için daha fazla zaman harcanması gerektiği anlamına gelir. Sonuç olarak, makine öğrenmesi ve istatistikteki ilgi noktası (genelleştirilmiş) doğrusal modellerden ve çekirdek yöntemlerinden derin ağlara taşındı. Bu aynı zamanda derin öğrenmenin dayanak noktalarının, çok katmanlı algılayıcılar (McCulloch and Pitts, 1943), evrişimli sinir ağları (LeCun *et al.*, 1998), uzun-ömürlü kısa dönem bellek (Hochreiter and Schmidhuber, 1997) ve Q-Öğrenme (Watkins and Dayan, 1992) gibi, oldukça uzun bir süre nispeten uykuda kaldıktan sonra, esasen "yeniden keşfedilme"sindeki birçok nedenden biridir.

İstatistiksel modeller, uygulamalar ve algoritmalardaki son gelişmeler bazen Kambriyen (Cambrian) patlamasına benzetildi: Türlerin evriminde hızlı bir ilerleme anı. Gerçekten de, en son teknoloji, sadece, onlarca yıllık algoritmaların mevcut kaynaklara uygulanmasının bir sonucu değildir. Aşağıdaki liste, araştırmacıların son on yılda muazzam bir ilerleme kaydetmesine yardımcı olan fikirlerin sadece yüzeyine ışık tutmaktadir.

- *Hattan düşürme* (dropout) gibi kapasite kontrolüne yönelik yeni yöntemler (Srivastava *et al.*, 2014), aşırı öğrenme tehlikesini azaltmaya yardımcı oldu. Bu, ağ boyunca gürültü zerk edilerek (enjeksiyon) sağlandı, (Bishop, 1995), eğitim amaçlı ağırlıkları rastgele değişkenlerle değiştirdi.
- Dikkat mekanizmaları, yüzyılı aşkın bir süredir istatistiği rahatsız eden ikinci bir sorunu çözdü: Öğrenilebilir parametre sayısını artırmadan bir sistemin belleğini ve karmaşıklığını nasıl artırabiliriz. Araştırmacılar sadece öğrenilebilir bir işaretçi yapısı olarak görülebilecek zarif bir çözüm buldu (Bahdanau *et al.*, 2014). Bir cümlenin tamamını hatırlamak yerine, örneğin, sabit boyutlu bir gösterimdeki makine çevirisi için, depolanması gereken tek şey, çeviri işleminin ara durumunu gösteren bir işaretçiydi. Bu, modelin artık yeni bir dizi oluşturulmadan önce tüm diziyi hatırlamasını gerektirmediğinden, uzun diziler için önemli ölçüde artırılmış doğruluğa izin verdi.
- Çok aşamalı tasarımlar, örneğin, bellek ağları (MemNets) (Sukhbaatar *et al.*, 2015) ve sinir programcısı-tercüman (Neural Programmer-Interpreter) (Reed and De Freitas, 2015) aracılığıyla, istatistiksel modelcilerin tekrarlamalı yaklaşımlar ile akıl yürütme tanımlamasına izin verdi. Bu araçlar, bir işlemcinin bir hesaplama için belleği değiştirmesine benzer şekilde derin ağın dahili bir durumunun tekrar tekrar değiştirilmesine izin verir; böylece bir akıl yürütme zincirinde sonraki adımlar gerçekleştirilebilir.

- Bir başka önemli gelişme de çekişmeli üretici ağların icadıdır (Goodfellow *et al.*, 2014). Geleneksel olarak, yoğunluk tahmini için istatistiksel yöntemler ve üretici modeller, uygun olasılık dağılımlarını ve bunlardan örnekleme için (genellikle yaklaşık) algoritmaları bulmaya odaklanmıştır. Sonuç olarak, bu algoritmalar büyük ölçüde istatistiksel modellerin doğasında var olan esneklik eksikliği ile sınırlıydı. Çekişmeli üretici ağlardaki en önemli yenilik, örnekleyiciyi türevlenebilir parametrelere sahip rastgele bir algoritma ile değiştirmekti. Bunlar daha sonra, ayrımcının (aslen ikili-örneklem testidir) sahte verileri gerçek verilerden ayırt edemeyeceği şekilde ayarlanır. Veri üretmek için rasgele algoritmalar kullanma yeteneği sayesinde yoğunluk tahminini çok çeşitli tekniklere açmıştır. Dörtnala Zebralar (Zhu *et al.*, 2017) ve sahte ünlü yüzleri (Karras *et al.*, 2017) örnekleri bu ilerlemenin kanıtıdır. Amatör karalamacılar bile, bir sahnenin düzeninin nasıl göründüğünü açıklayan eskizlere dayanan fotogerçekçi görüntüler üretebilir (Park *et al.*, 2019).
- Çoğu durumda, tek bir GPU eğitim için mevcut olan büyük miktarda veriyi işlemede yetersizdir. Son on yılda, paralel ve dağıtılmış eğitim algoritmaları oluşturma yeteneği önemli ölçüde gelişmiştir. Ölçeklenebilir algoritmaların tasarlanmasındaki temel zorluklardan biri, derin öğrenme optimizasyonunun ana öğesinin, rasgele eğim inişinin, işlenecek verilerin nispeten küçük minigruplarına (minibatch) dayanmasıdır. Aynı zamanda, küçük gruplar GPU'ların verimliliğini sınırlar. Bu nedenle, 1024 GPU'nun eğitimindeki minigrup büyüklüğü, örneğin toplu iş başına 32 resim diyelim, toplam yaklaşık 32000 resim anlamına gelir. Son çalışmalarda, önce Li (Li, 2017) ve ardından (You *et al.*, 2017) ve (Jia *et al.*, 2018) boyutu 64000 gözleme yükselterek, ImageNet veri kümesinde ResNet50 modeli için eğitim süresini 7 dakikadan daha az bir süreye azalttılar. Karşılaştırma için — başlangıçta eğitim süreleri günlere göre ölçülürdü.
- Hesaplamayı paralel hale getirme yeteneği, en azından simülasyon (benzetim) bir seçenek olduğunda, pekiştirmeli öğrenmedeki ilerlemeye oldukça önemli bir katkıda bulunmuştur. Bu önemli ilerlemeler Go, Atari oyunları, Starcraft ve fizik simülasyonlarında (örn. MuJoCo kullanarak) insanüstü performans elde eden bilgisayarlara yol açtı. AlphaGo'da bunun nasıl yapılacağına ilişkin açıklama için bakınız (Silver *et al.*, 2016). Özetle, pek çok (durum, eylem, ödül) üçlük mevcutsa, yani birbirleriyle nasıl ilişkilendiklerini öğrenmek için birçok şeyi denemek mümkün olduğunda pekiştirmeli öğrenme en iyi sonucu verir. Benzetim böyle bir yol sağlar.
- Derin öğrenme çerçeveleri fikirlerin yayılmasında önemli bir rol oynamıştır. Kapsamlı modellemeyi kolaylaştıran ilk nesil çerçeveler: Caffe[24], Torch[25] ve Theano[26]. Bu araçlar kullanılarak birçok yeni ufuklar açan makale yazılmıştır. Şimdiye kadar yerlerini TensorFlow[27] ve onu da genellikle yüksek düzey API aracılığıyla kullanan Keras[28], CNTK[29], Caffe 2[30] ve Apache MXNet[31] aldı. Üçüncü nesil araçlara, yani derin öğrenme için zorunlu araçlara, modelleri tanımlamak için Python NumPy'ye benzer bir sözdizimi kullanan Chainer[32] öncülük etti. Bu fikir hem PyTorch[33], hem Gluon API[34] MXNet ve Jax[35] tarafından benimsenmiştir.

[24] https://github.com/BVLC/caffe
[25] https://github.com/torch
[26] https://github.com/Theano/Theano
[27] https://github.com/tensorflow/tensorflow
[28] https://github.com/keras-team/keras
[29] https://github.com/Microsoft/CNTK
[30] https://github.com/caffe2/caffe2
[31] https://github.com/apache/incubator-mxnet
[32] https://github.com/chainer/chainer
[33] https://github.com/pytorch/pytorch
[34] https://github.com/apache/incubator-mxnet
[35] https://github.com/google/jax

Daha iyi araçlar üreten sistem araştırmacıları ve daha iyi sinir ağlar inşa eden istatistiksel modelciler arasındaki işbölümü, işleri basitleştirdi. Örneğin, doğrusal bir lojistik regresyon modelinin eğitilmesi, 2014 yılında Carnegie Mellon Üniversitesi'nde makine öğrenmesi yeni doktora öğrencilerine bir ödev problemi vermeğe değer bariz olmayan bir problemdi. Şimdilerde, sıkı bir programcı kavrayışını içine katarak, bu görev 10'dan az kod satırı ile gerçekleştirilebilir.

1.6 Başarı Öyküleri

YZ, aksi takdirde başarılması zor olacak sonuçları dağıtmada uzun bir geçmişe sahiptir. Örneğin, postaları optik karakter tanıma kullanılarak sıralama sistemi 1990'lardan beri kullanılmaktadır. Bu, aslında, ünlü MNIST el yazısı rakam veri kümelerinin kaynağıdır. Aynı şey banka mevduatları için çek okumada ve başvuru sahiplerinin kredi değerliliğini için puanlanmada da geçerlidir. Finansal işlemler otomatik olarak sahtekarlığa karşı kontrol edilir. Bu, PayPal, Stripe, AliPay, WeChat, Apple, Visa ve MasterCard gibi birçok e-ticaret ödeme sisteminin omurgasını oluşturur. Bilgisayar satranç programları onlarca yıldır rekabetçidir. Makine öğrenmesi, internette arama, öneri, kişiselleştirme ve sıralamayı besler. Başka bir deyişle, makine öğrenmesi, çoğu zaman gözden gizli olsa da, yaygındır.

Sadece son zamanlarda YZ, çoğunlukla daha önce zorlu olarak kabul edilen ve doğrudan tüketicileri ilgilendiren sorunlara çözümlerinden dolayı ilgi odağı olmuştur. Bu ilerlemelerin birçoğu derin öğrenmeli ile ilişkilidir.

- Apple'ın Siri, Amazon'un Alexa ve Google'ın Asistanı gibi akıllı asistanlar sözlü soruları makul bir doğrulukla cevaplayabilir. Bu, ışık anahtarlarını (engelliler için bir nimet) açmadan, berber randevuları ayarlamaya ve telefon destek iletişim diyalogu sunmaya kadar önemli görevleri içerir. Bu muhtemelen YZ'nın hayatlarımızı etkilediğinin en belirgin işaretidir.
- Dijital asistanların önemli bir bileşeni, konuşmayı doğru bir şekilde tanıma yeteneğidir. Yavaş yavaş bu tür sistemlerin doğruluğu, belirli uygulamalar için insan paritesine ulaştığı noktaya kadar artmıştır (Xiong *et al.*, 2018).
- Nesne tanıma da aynı şekilde uzun bir yol kat etti. Bir resimdeki nesneyi tahmin etmek 2010 yılında oldukça zor bir işti. ImageNet karşılaştırmalı değerlendirmesinde NEC Laboratuarından ve Illinois at Urbana-Champaign üniversitesinden araştırmacılar ilk 5'te %28'lik bir hata oranına ulaştı (Lin *et al.*, 2010). 2017 itibariyle bu hata oranını %2.25'e düştü (Hu *et al.*, 2018). Benzer şekilde, kuşları tanımlamada veya cilt kanserini teşhis etmede çarpıcı sonuçlar elde edilmiştir.
- Oyunlar eskiden insan zekasının kalesi idi. TD-Gammon'dan itibaren, tavla oynamak için zamansal fark pekiştirmeli öğrenme kullanan bir programdır, algoritmik ve hesaplamalı ilerleme birçok çeşitli uygulamada kullanılan algoritmalara yol açmıştır. Tavlanın aksine, satranç çok daha karmaşık bir durum uzayına ve bir dizi eyleme sahiptir. DeepBlue, Garry Kasparov'u, büyük paralelleştirme, özel amaçlı donanım ve oyun ağacında verimli arama kullanarak yendi (Campbell *et al.*, 2002). Büyük durum uzayı nedeniyle Go hala daha zordur. AlphaGo, 2015 yılında insan paritesine ulaştı, derin öğrenmeyi Monte Carlo ağaç örneklemesi ile birlikte kullandı (Silver *et al.*, 2016). Poker'deki zorluk, durum uzayının geniş olması ve tam olarak gözlenememesidir (rakiplerin kartlarını bilmiyoruz). Libratus, etkin bir şekilde yapılandırılmış stratejiler kullanarak Poker'deki insan performansını aştı (Brown and Sandholm, 2017). Bu, oyunlardaki etkileyici ilerlemeyi ve gelişmiş algoritmaların oyunlarda önemli bir rol oynadığını göstermektedir.

- Yapay zekadaki ilerlemenin bir başka göstergesi, kendi kendine giden (otonom - özerk) otomobil ve kamyonların ortaya çıkışıdır. Tam özerkliğe henüz tam olarak ulaşılamamasına rağmen mükemmel bir ilerleme kaydedildi; Tesla, NVIDIA ve Waymo gibi şirketlerle en azından kısmi özerklik ile ürün teslimatına olanak sağlıyor. Tam özerkliği bu kadar zorlaştıran şey, uygun sürüşün, kuralları algılama, akıl yürütme ve kuralları bir sisteme dahil etme yeteneğini gerektirmesidir. Günümüzde, derin öğrenme bu sorunlardan öncelikle bilgisayarla görme ile ilgili alanında kullanılmaktadır. Geri kalanları mühendisler tarafından yoğun bir şekilde ince ayarlamaya çekilir.

Yine, yukarıdaki liste, makine öğrenmesinin pratik uygulamalarının etkilediği yüzeylere çok az ışık tutmaktadır. Örneğin, robotik, lojistik, hesaplamalı biyoloji, parçacık fiziği ve astronomi, en etkileyici son gelişmelerinden bazılarını en azından kısmen makine öğrenmesine borçludur. Makine öğrenmesi böylece mühendisler ve bilim insanları için her yerde mevcut bir araç haline geliyor.

YZ ile ilgili teknik olmayan makalelerde, YZ kıyameti veya YZ tekilliği sorunu sıkça gündeme gelmiştir. Korku, bir şekilde makine öğrenmesi sistemlerinin, insanların geçimini doğrudan etkileyen şeyler hakkında programcılarından (ve sahiplerinden) bağımsız bir şekilde duyarlı (akıllı) olacağına ve karar vereceğinedir. Bir dereceye kadar, YZ zaten insanların yaşamlarını şimdiden etkiliyor: Kredibilite otomatik olarak değerlendiriliyor, otomatik pilotlar çoğunlukla taşıtları yönlendiriyor, kefalet vermeye istatistiksel veri girdisi kullanarak karar veriliyor. Daha anlamsızcası, Alexa'dan kahve makinesini açmasını isteyebiliriz.

Neyse ki, insan yaratıcılarını manipüle etmeye (veya kahvelerini yakmaya) hazır, duyarlı bir YZ sisteminden çok uzaktayız. İlk olarak, YZ sistemleri belirli, hedefe yönelik bir şekilde tasarlanır, eğitilir ve devreye alınır. Davranışları genel zeka yanılsamasını verebilse de, tasarımın altında yatan kuralların, sezgisel ve istatistiksel modellerin birleşimleridirler. İkincisi, şu anda, *genel yapay zeka* için, kendilerini geliştirebilen, kendileriyle ilgili akıl yürüten ve genel görevleri çözmeye çalışırken kendi mimarilerini değiştirebilen, genişletebilen ve geliştirebilen araçlar yoktur.

Çok daha acil bir endişe YZ'nın günlük yaşamımızda nasıl kullanıldığıdır. Kamyon şoförleri ve mağaza asistanları tarafından yerine getirilen birçok önemli görevin otomatikleştirilebileceği ve otomatikleştirileceği muhtemeldir. Çiftlik robotları büyük olasılıkla organik tarım maliyetini düşürecek, ayrıca hasat işlemlerini de otomatikleştirecek. Sanayi devriminin bu aşamasının toplumun büyük kesimlerinde derin sonuçları olabilir, çünkü kamyon şoförleri ve mağaza asistanları birçok ülkedeki en yaygın işlerden ikisidir. Ayrıca, istatistiksel modeller, dikkatsizce uygulandığında ırk, cinsiyet veya yaş yanlılığına yol açabilir ve sonuç kararları otomatik hale getirildiklerinde usul adaleti konusunda makul endişeler doğurabilir. Bu algoritmaların dikkatle kullanılmasını sağlamak önemlidir. Bugün bildiklerimizle, bu bize, kötü niyetli bir süper zekanın insanlığı yok etme potansiyelinden çok daha acil bir endişe gibi geliyor.

1.7 Özellikler

Şimdiye kadar, hem yapay zekanın bir dalı hem de yapay zekaya bir yaklaşım olan makine öğrenmesinden geniş olarak bahsettik. Derin öğrenme, makine öğreniminin bir alt kümesi olmasına rağmen, baş döndürücü algoritmalar ve uygulamalar kümesi, derin öğrenmenin bileşenlerinin özellikle ne olabileceğini değerlendirmeyi zorlaştırıyor. Bu, neredeyse her bileşen değiştirilebilir olduğundan, pizza için gerekli malzemeleri belirlemeye çalışmak kadar zordur.

Tanımladığımız gibi, makine öğrenmesi, konuşma tanımada sesi metne dönüştürme benzeri işlerde, girdiler ve çıktılar arasındaki dönüşümleri öğrenmek için verileri kullanabilir. Bunu yaparken, bu tür temsilleri çıktıya dönüştürmek için genellikle verileri algoritmalara uygun bir şek-

ilde temsil etmek gerekir. *Derin öğrenme*, tam olarak modellerinin birçok dönüşüm *katmanını* öğrenmesi anlamında *derindir*, ki burada her katman, temsili bir düzeyde sunar. Örneğin, girdiye yakın katmanlar, verilerin düşük seviyeli ayrıntılarını temsil edebilirken, sınıflandırma çıktısına daha yakın katmanlar, ayrım için kullanılan daha soyut kavramları temsil edebilir. *Temsil öğrenimi* temsilin kendisini bulmayı amaçladığından, derin öğrenme çok seviyeli temsil öğrenimi olarak adlandırılabilir.

Ham ses sinyalinden öğrenme, görüntülerin ham piksel değerleri veya keyfi uzunluktaki cümleler ile yabancı dillerdeki karşılıkları arasında eşleme gibi şimdiye kadar tartıştığımız sorunlar, derin öğrenmenin üstünleşip ve geleneksel makine öğrenmesinin zayıf kaldığı sorunlardır. Bu, çok katmanlı modellerin, önceki araçların yapamadığı şekilde düşük seviyeli algısal verileri ele alma yeteneğine sahip olduğu ortaya çıkardı. Muhtemelen derin öğrenme yöntemlerindeki en önemli ortak nokta *uçtan uca eğitimin* kullanılmasıdır. Yani, ayrı ayrı ayarlanmış bileşenlere dayalı bir sistem kurmak yerine, sistem kurulur ve ardından performansları birlikte ayarlanır. Örneğin, bilgisayarla görmede bilim adamları, *öznitelik mühendisliği* sürecini makine öğrenmesi modelleri oluşturma sürecinden ayırırlardı. Canny kenar detektörü (Canny, 1987) ve Lowe'un SIFT öznitelik çıkarıcısı (Lowe, 2004), görüntüleri öznitelik vektörlerine eşleme algoritmaları olarak on yıldan fazla bir süre üstünlük sağladı. Geçmiş günlerde, makine öğrenmesini bu sorunlara uygulamanın en önemli kısmı, verileri sığ modellere uygun bir forma dönüştürmek için manuel olarak tasarlanmış yöntemler bulmaktan ibaretti. Ne yazık ki, bir algoritma tarafından otomatik olarak gerçekleştirilen milyonlarca seçeneğin tutarlı değerlendirmesiyle karşılaştırıldığında, insanların ustalıkla başarabilecekleri çok az şey vardır. Derin öğrenme egemenlik sağladığında, bu öznitelik çıkarıcıların yerini otomatik olarak ayarlanmış filtreler aldı ve daha üstün doğruluk oranı sağladı.

Bu nedenle, derin öğrenmenin önemli bir avantajı, yalnızca geleneksel öğrenme işlem hatlarının sonundaki sığ modellerin değil, aynı zamanda emek yoğun öznitelik mühendisliği sürecinin de yerini almasıdır. Ayrıca, derin öğrenme, alana özgü ön işlemenin çoğunu değiştirerek, bilgisayarla görme, konuşma tanıma, doğal dil işleme, tıbbi bilişim ve diğer uygulama alanlarını daha öncesinde birbirinden ayıran sınırların çoğunu ortadan kaldırarak, çeşitli sorunların üstesinden gelmek için birleşik bir araç kümesi sunar.

Uçtan uca eğitimin ötesinde, parametrik istatistiksel tanımlamalardan tamamen parametrik olmayan modellere geçiş yaşıyoruz. Veriler kıt olduğunda, faydalı modeller elde etmek için gerçeklikle ilgili basitleştirilmiş varsayımlara güvenmek gerekir. Veri bol olduğunda, bunun yerine gerçekliğe daha doğru uyan parametrik olmayan modeller kullanılabilir. Bu, bir dereceye kadar, fiziğin bilgisayarların mevcudiyeti ile önceki yüzyılın ortalarında yaşadığı ilerlemeyi yansıtıyor. Elektronların nasıl davrandığına dair parametrik yaklaşımları elle çözmek yerine, artık ilgili kısmi türevli denklemlerin sayısal benzetimlerine başvurabiliriz. Bu, çoğu zaman açıklanabilirlik pahasına da olsa, çok daha doğru modellere yol açmıştır.

Önceki çalışmalardan bir diğer fark da, optimal olmayan çözümlerin kabul edilmesi, dışbükey olmayan doğrusal olmayan optimizasyon problemleriyle uğraşılması ve bir şeyleri kanıtlamadan önce denemeye istekli olmasıdır. İstatistiksel problemlerle uğraşmada yeni keşfedilen bu deneysellik, hızlı bir yetenek akışı ile birleştiğinde, birçok durumda on yıllardır var olan araçları değiştirme ve yeniden icat etme pahasına da olsa pratik algoritmaların hızlı ilerlemesine yol açmıştır.

Sonuç olarak, derin öğrenme topluluğu, akademik ve kurumsal sınırları aşarak araçları paylaşmaktan, birçok mükemmel kütüphaneyi, istatistiksel modelleri ve eğitimli ağları açık kaynak olarak yayınlamaktan gurur duyuyor. Bu kitabı oluşturan defterlerin dağıtım ve kullanım için ücretsiz olarak temin edilmesi bu anlayıştandır. Herkesin derin öğrenme hakkında bilgi edinmesinde erişim engellerini azaltmak için çok çalıştık ve okuyucularımızın bundan faydalanacağını umuyoruz.

1.8 Özet

- Makine öğrenmesi, belirli görevlerde performansı artırmak için bilgisayar sistemlerinin *deneyiminden* (genellikle veri) nasıl yararlanabileceğini inceler. İstatistik, veri madenciliği ve optimizasyon fikirlerini birleştirir. Genellikle, YZ çözümlerinin uygulanmasında bir araç olarak kullanılır.
- Bir makine öğrenmesi sınıfı olarak, temsili öğrenme, verileri temsil etmek için uygun yolu otomatik olarak nasıl bulacağınıza odaklanır. Derin öğrenme, birçok dönüşüm katmanını öğrenerek çok kademeli temsili öğrenmedir.
- Derin öğrenme, yalnızca geleneksel makine öğrenmesi ardışık düzenlerinin sonundaki sığ modellerin değil, aynı zamanda emek yoğun öznitelik mühendisliği sürecinin de yerini alır.
- Derin öğrenmedeki yakın zaman ilerlemenin çoğu, ucuz sensörler ve internetteki ölçekli uygulamalardan kaynaklanan çok miktarda veri ve çoğunlukla GPU'lar aracılığıyla hesaplamadaki önemli ilerleme ile tetiklenmiştir.
- Tüm sistem optimizasyonu, yüksek performans elde etmede önemli bir ana bileşendir. Etkili derin öğrenme çerçevelerinin mevcudiyeti, bunun tasarımını ve uygulamasını önemli ölçüde kolaylaştırmıştır.

1.9 Alıştırmalar

1. Şu anda yazdığınız kodun hangi bölümleri "öğrenilebilir", yani kodunuzda yapılan tasarım seçimleri öğrenilerek ve otomatik olarak belirlenerek geliştirilebilir? Kodunuzda sezgisel (heuristic) tasarım seçenekleri var mı?
2. Hangi karşılaştığınız sorunlarda nasıl çözüleceğine dair birçok örnek var, ancak bunları otomatikleştirmenin belirli bir yolu yok? Bunlar derin öğrenmeyi kullanmaya aday olabilirler.
3. YZ gelişimini yeni bir sanayi devrimi olarak görürsek algoritmalar ve veriler arasındaki ilişki nedir? Buharlı motorlara ve kömüre benzer mi? Temel fark nedir?
4. Uçtan uca eğitim yaklaşımını, Fig. 1.1.2, fizik, mühendislik ve ekonometri gibi başka nerede uygulayabilirsiniz?

Tartışmalar[36]

[36] https://discuss.d2l.ai/t/22

2 | Ön Hazırlık

Derin öğrenmeye başlamak için birkaç temel beceri geliştirmemiz gerekecek. Tüm makine öğrenmesi verilerden bilgi çıkarmakla ilgilidir. Bu nedenle, verileri depolamak, oynama yapmak (manipule etmek) ve ön işlemek için pratik becerileri öğrenerek başlayacağız.

Dahası, makine öğrenmesi tipik olarak satırların örneklere ve sütunların niteliklere karşılık geldiği tablolar olarak düşünebileceğimiz büyük veri kümeleriyle çalışmayı gerektirir. Doğrusal cebir, tablo verileriyle çalışmak için bize bir dizi güçlü teknik sunar. Boyumuzu aşan sulara çok fazla girmeyeceğiz, daha ziyade dizey (matris) işlemlerinin temeline ve bunların uygulanmasına odaklanacağız.

Ek olarak, derin öğrenme tamamen eniyileme (optimizasyon) ile ilgilidir. Bazı parametrelere sahip bir modelimiz var ve verilerimize *en uygun olanları* bulmak istiyoruz. Bir algoritmanın her adımında her bir parametreyi hangi şekilde hareket ettireceğini karar vermek için, burada kısaca bahsedeceğiz, bir miktar hesaplama (kalkülüs) gerekir. Neyse ki, `autograd` paketi bizim için otomatik olarak türevleri hesaplar; bunu daha sonra işleyeceğiz.

Dahası, makine öğrenmesi tahminlerde bulunmakla ilgilidir: Gözlemlediğimiz bilgiler göz önüne alındığında, bazı bilinmeyen özelliklerin olası değeri nedir? Belirsizlik altında titizlikle çıkarsama yapabilmek için olasılık dilini hatırlamamız gerekecek.

Hakikatinde, asli kaynaklar bu kitabın ötesinde birçok açıklama ve örnek sunmaktadır. Bölümü bitirken size gerekli bilgiler için kaynaklara nasıl bakacağınızı göstereceğiz.

Bu kitap, derin öğrenmeyi doğru bir şekilde anlamak için gerekli olan matematiksel içeriği en azda tutmuştur. Ancak, bu, bu kitabın matematik içermediği anlamına gelmez. Bu nedenle, bu bölüm, herhangi bir kişinin, kitabın matematiksel içeriğinin en azından *çoğunu* anlayabilmesi için temel ve sık kullanılan matematiğe hızlı bir giriş yapmasını sağlar. Matematiksel içeriğin *tümünü* anlamak istiyorsanız, matematik üzerine çevrimiçi ek[37]i derinlemesine gözden geçirmek yeterli olacaktır.

2.1 Veri ile Oynama Yapmak

Bir şeylerin yapılabilmesi için, verileri depolamak ve oynama yapmak (manipule etmek) için bazı yollar bulmamız gerekiyor. Genellikle verilerle ilgili iki önemli şey vardır: (i) Bunları elde etmek; ve (ii) bilgisayar içine girdikten sonra bunları işlemek. Veri depolamanın bir yolu olmadan onu elde etmenin bir anlamı yok, bu yüzden önce sentetik (yapay) verilerle oynayarak ellerimizi kirletelim. Başlamak için, *gerey (tensör)* olarak da adlandırılan n boyutlu diziyi tanıtalım.

Python'da en çok kullanılan bilimsel hesaplama paketi olan NumPy ile çalıştıysanız, bu bölümü tanıdık bulacaksınız. Hangi çerçeveyi kullanırsanız kullanın, *tensör sınıfı* (MXNet'teki `ndarray`,

[37] https://tr.d2l.ai/chapter_appendix-mathematics-for-deep-learning/index.html

hem PyTorch hem de TensorFlow'daki `Tensor`), fazladan birkaç vurucu özellik ile NumPy'nin `ndarray`'ına benzer. İlk olarak, GPU hesaplama hızlandırmayı iyi desteklerken, NumPy sadece CPU hesaplamasını destekler. İkincisi, tensör sınıfı otomatik türev almayı destekler. Bu özellikler, tensör sınıfını derin öğrenme için uygun hale getirir. Kitap boyunca, tensörler dediğimizde, aksi belirtilmedikçe tensör sınıfının örneklerinden bahsediyoruz.

2.1.1 Başlangıç

Bu bölümde, sizi ayaklandırıp koşturmayı, kitapta ilerledikçe üstüne koyarak geliştireceğiniz temel matematik ve sayısal hesaplama araçlarıyla donatmayı amaçlıyoruz. Bazı matematiksel kavramları veya kütüphane işlevlerini içselleştirmede zorlanıyorsanız, endişelenmeyin. Aşağıdaki bölümlerde bu konular pratik örnekler bağlamında tekrar ele alınacak ve yerine oturacak. Öte yandan, zaten biraz bilgi birikiminiz varsa ve matematiksel içeriğin daha derinlerine inmek istiyorsanız, bu bölümü atlayabilirsiniz.

Başlamak için `torch`u içe aktarıyoruz. PyTorch olarak adlandırılsa da, `pytorch` yerine `torch`u içe aktarmamız gerektiğini unutmayın.

```
import torch
```

Bir tensör, (muhtemelen çok boyutlu) bir sayısal değerler dizisini temsil eder. Bir eksende, tensöre *vektör* denir. İki eksende, tensöre *matris* denir. İkiden fazla ekseni olan tensörlerin özel matematik isimleri yoktur. $k > 2$ eksende, tensörlerin özel adları yoktur, bu nesnelere k. *dereceli tensörler* deriz.

PyTorch, değerlerle önceden doldurulmuş yeni tensörler oluşturmak için çeşitli işlevler sağlar. Örneğin, `arange(n)`'yi çağırarak, 0'dan başlayarak (dahil) ve `n` ile biten (dahil değil) eşit aralıklı değerlerden oluşan bir vektör oluşturabiliriz. Varsayılan olarak, aralık boyutu 1'dir. Aksi belirtilmedikçe, yeni tensörler ana bellekte depolanır ve CPU tabanlı hesaplama için atanır.

```
x = torch.arange(12, dtype=torch.float32)
x
```

```
tensor([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11.])
```

Bir tensörün *şekli*ne (her eksen boyunca uzunluk) `shape` özelliğini inceleyerek erişebiliriz.

```
x.shape
```

```
torch.Size([12])
```

Bir tensördeki toplam eleman sayısını, yani tüm şekil elemanlarının çarpımını bilmek istiyorsak, boyutunu inceleyebiliriz. Burada bir vektörle uğraştığımız için, `shape` (şeklinin) tek elemanı, vektör boyutu ile aynıdır.

```
x.numel()
```

```
12
```

Eleman sayısını veya değerlerini değiştirmeden bir tensörün şeklini değiştirmek için `reshape` işlevini çağırabiliriz. Örneğin, `x` tensörümüzü, (12,) şekilli bir satır vektöründen (3, 4) şekilli bir matrise dönüştürebiliriz . Bu yeni tensör tam olarak aynı değerleri içerir, ancak onları 3 satır ve 4 sütun olarak düzenlenmiş bir matris olarak görür. Yinelemek gerekirse, şekil değişmiş olsa da, elemanlar değişmemiştir. Boyutun yeniden şekillendirilme ile değiştirilmediğine dikkat edin.

```
X = x.reshape(3, 4)
X
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]])
```

Her boyutu manuel olarak belirterek yeniden şekillendirmeye gerek yoktur. Hedef şeklimiz (yükseklik, genişlik) şekilli bir matrisse, o zaman genişliği bilirsek, yükseklik üstü kapalı olarak verilmiş olur. Neden bölmeyi kendimiz yapmak zorunda olalım ki? Yukarıdaki örnekte, 3 satırlı bir matris elde etmek için, hem 3 satır hem de 4 sütun olması gerektiğini belirttik. Neyse ki, tensörler, bir boyut eksik geri kalanlar verildiğinde, kalan bir boyutu otomatik olarak çıkarabilir. Bu özelliği, tensörlerin otomatik olarak çıkarımını istediğimiz boyuta `-1` yerleştirerek çağırıyoruz. Bizim durumumuzda, `x.reshape(3, 4)` olarak çağırmak yerine, eşit biçimde `x.reshape(-1, 4)` veya `x.reshape(3, -1)` olarak çağırabilirdik.

Tipik olarak, matrislerimizin sıfırlar, birler, diğer bazı sabitler veya belirli bir dağılımdan rastgele örneklenmiş sayılarla başlatılmasını isteriz. Tüm elemanları 0 olarak ayarlanmış ve (2, 3, 4) şeklindeki bir tensörü temsil eden bir tensörü aşağıdaki şekilde oluşturabiliriz:

```
torch.zeros((2, 3, 4))
```

```
tensor([[[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]],

        [[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]]])
```

Benzer şekilde, her bir eleman 1'e ayarlanmış şekilde tensörler oluşturabiliriz:

```
torch.ones((2, 3, 4))
```

```
tensor([[[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]],

        [[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]]])
```

Genellikle, bir tensördeki her eleman için değerleri bir olasılık dağılımından rastgele örneklemek isteriz. Örneğin, bir sinir ağında parametre görevi görecek dizileri oluşturduğumuzda, değerlerini genellikle rastgele ilkletiriz. Aşağıdaki kod parçası (3, 4) şekilli bir tensör oluşturur. Elemanlarının

her biri ortalaması 0 ve standart sapması 1 olan standart Gauss (normal) dağılımından rastgele örneklenir.

```
torch.randn(3, 4)
```

```
tensor([[-0.5191,  0.0136,  0.9964, -0.6709],
        [ 0.5979, -1.0787,  1.1654,  1.6462],
        [-1.3329,  1.4749, -1.6688,  0.3651]])
```

Sayısal değerleri içeren bir Python listesi (veya liste listesi) sağlayarak istenen tensördeki her eleman için kesin değerleri de belirleyebiliriz. Burada, en dıştaki liste 0. eksene, içteki liste ise 1. eksene karşılık gelir.

```
torch.tensor([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

```
tensor([[2, 1, 4, 3],
        [1, 2, 3, 4],
        [4, 3, 2, 1]])
```

2.1.2 İşlemler

Bu kitap yazılım mühendisliği ile ilgili değildir. İlgi alanlarımız basitçe dizilerden/dizilere veri okumak ve yazmakla sınırlı değildir. Bu diziler üzerinde matematiksel işlemler yapmak istiyoruz. En basit ve en kullanışlı işlemlerden bazıları *eleman yönlü (elementwise)* işlemlerdir. Bunlar bir dizinin her elemanına standart bir sayıl işlem uygular. İki diziyi girdi olarak alan işlevler için, eleman yönlü işlemler iki diziden karşılık gelen her bir eleman çiftine standart bir ikili operatör uygular. Sayıldan sayıla eşleşen herhangi bir fonksiyondan eleman yönlü bir fonksiyon oluşturabiliriz.

Matematiksel gösterimde, böyle bir *tekli* skaler işlemi (bir girdi alarak) $f : \mathbb{R} \rightarrow \mathbb{R}$ imzasıyla ifade ederiz. Bu, işlevin herhangi bir gerçel sayıdan ($\mathbb{R}$) diğerine eşlendiği anlamına gelir. Benzer şekilde, $f : \mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$ imzası ile bir *ikili* skaler operatörü (iki gerçel girdi alarak ve bir çıktı verir) belirtiriz. *Aynı şekilli* iki $\mathbf{u}$ ve $\mathbf{v}$ vektörü ve f ikili operatörü verildiğinde, tüm $i$ler için $c_i \leftarrow f(u_i, v_i)$ diye ayarlayarak $\mathbf{c} = F(\mathbf{u}, \mathbf{v})$ vektörünü üretebiliriz; burada c_i, u_i ve v_i, $\mathbf{c}, \mathbf{u}$ ve $\mathbf{v}$ vektörlerinin $i.$ elemanlarıdır. Burada, skaler fonksiyonu eleman yönlü bir vektör işlemini *yükselterek* vektör değerli $F : \mathbb{R}^d, \mathbb{R}^d \rightarrow \mathbb{R}^d$ ürettik.

Ortak standart aritmetik operatörler (`+`, `-`, `*`, `/` ve `**`), rastgele şekile sahip herhangi bir benzer şekilli tensörler için eleman yönlü işlemlere *yükseltilmiştir*. Aynı şekle sahip herhangi iki tensör üzerinde eleman yönlü işlemleri çağırabiliriz. Aşağıdaki örnekte, 5 öğeli bir grubu formüle etmek için virgül kullanıyoruz, her öğe eleman yönlü bir işlemin sonucudur.

İşlemler

Genel standart aritmatik işlemler (`+`, `-`, `*`, `/`, ve `**`) eleman yönlü işlemlere *yükseltilmiştir*.

```
x = torch.tensor([1.0, 2, 4, 8])
y = torch.tensor([2, 2, 2, 2])
x + y, x - y, x * y, x / y, x ** y  # ** işlemi kuvvet almadır.
```

```
(tensor([ 3.,  4.,  6., 10.]),
 tensor([-1.,  0.,  2.,  6.]),
 tensor([ 2.,  4.,  8., 16.]),
 tensor([0.5000, 1.0000, 2.0000, 4.0000]),
 tensor([ 1.,  4., 16., 64.]))
```

Kuvvet alma gibi tekli operatörler de dahil olmak üzere, çok daha fazla işlem eleman yönlü olarak uygulanabilir.

```
torch.exp(x)
```

```
tensor([2.7183e+00, 7.3891e+00, 5.4598e+01, 2.9810e+03])
```

Eleman yönlü hesaplamalara ek olarak, vektör iç çarpımı ve matris çarpımı dahil olmak üzere doğrusal cebir işlemleri de gerçekleştirebiliriz. Doğrusal cebirin önemli parçalarını (varsayılmış hiçbir ön bilgi olmadan) Section 2.3 içinde açıklayacağız.

Ayrıca birden fazla tensörü bir araya getirip daha büyük bir tensör oluşturmak için uçtan uca *istifleyebiliriz*. Sadece tensörlerin bir listesini vermeli ve sisteme hangi eksende birleştireceklerini söylemeliyiz. Aşağıdaki örnek, satırlar (eksen 0, şeklin ilk öğesi) ile sütunlar (eksen 1, şeklin ikinci öğesi) boyunca iki matrisi birleştirdiğimizde ne olacağını gösterir. İlk çıktı tensörünün eksen-0 uzunluğunun (6) iki girdi tensörünün eksen-0 uzunluklarının $(3+3)$ toplamı olduğunu görebiliriz; ikinci çıktı tensörünün eksen-1 uzunluğu (8) iki girdi tensörünün eksen-1 uzunluklarının $(4+4)$ toplamıdır.

```
X = torch.arange(12, dtype=torch.float32).reshape((3,4))
Y = torch.tensor([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
torch.cat((X, Y), dim=0), torch.cat((X, Y), dim=1)
```

```
(tensor([[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [ 2.,  1.,  4.,  3.],
         [ 1.,  2.,  3.,  4.],
         [ 4.,  3.,  2.,  1.]]),
 tensor([[ 0.,  1.,  2.,  3.,  2.,  1.,  4.,  3.],
         [ 4.,  5.,  6.,  7.,  1.,  2.,  3.,  4.],
         [ 8.,  9., 10., 11.,  4.,  3.,  2.,  1.]]))
```

Bazen, *mantıksal ifadeler* aracılığıyla bir ikili tensör oluşturmak isteriz. Örnek olarak `X == Y`'yi ele alalım. Her konum için, eğer `X` ve `Y` bu konumda eşitse, yeni tensördeki karşılık gelen girdi 1 değerini alır, yani mantıksal ifade `X == Y` o konumda doğrudur; aksi halde o pozisyon 0 değerini alır.

```
X == Y
```

```
tensor([[False,  True, False,  True],
        [False, False, False, False],
        [False, False, False, False]])
```

Tensördeki tüm elemanların toplanması, sadece bir elemanlı bir tensör verir.

```
X.sum()
```

```
tensor(66.)
```

2.1.3 Yayma Mekanizması

Yukarıdaki bölümde, aynı şekle sahip iki tensör üzerinde eleman yönlü işlemlerin nasıl yapıldığını gördük. Belli koşullar altında, şekiller farklı olsa bile, *yayma mekanizmasını* çağırarak yine de eleman yönlü işlemler gerçekleştirebiliriz. Bu mekanizma şu şekilde çalışır: İlk olarak, bir veya her iki diziyi elemanları uygun şekilde kopyalayarak genişletin, böylece bu dönüşümden sonra iki tensör aynı şekle sahip olur. İkincisi, sonuç dizileri üzerinde eleman yönlü işlemleri gerçekleştirin.

Çoğu durumda, bir dizinin başlangıçta yalnızca 1 uzunluğuna sahip olduğu bir eksen boyunca yayın yaparız, aşağıdaki gibi:

```
a = torch.arange(3).reshape((3, 1))
b = torch.arange(2).reshape((1, 2))
a, b
```

```
(tensor([[0],
         [1],
         [2]]),
 tensor([[0, 1]]))
```

a ve b sırasıyla 3×1 ve 1×2 matrisler olduğundan, onları toplamak istiyorsak şekilleri uyuşmaz. Her iki matrisin girdilerini aşağıdaki gibi daha büyük bir 3×2 matrisine *yayınlıyoruz*: Her ikisini de eleman yönlü eklemeden önce a matrisi için sütunlar çoğaltılır ve b matrisi için satırlar çoğaltılır.

```
a + b
```

```
tensor([[0, 1],
        [1, 2],
        [2, 3]])
```

2.1.4 İndeksleme ve Dilimleme

Diğer tüm Python dizilerinde olduğu gibi, bir tensördeki öğelere indeksle erişilebilir. Herhangi bir Python dizisinde olduğu gibi, ilk öğenin dizini 0'dır ve aralıklar ilk öğeyi içerecek ancak son öğeden *öncesi* eklenecek şekilde belirtilir. Standart Python listelerinde olduğu gibi, öğelere, negatif endeksler kullanarak listenin sonuna göreceli konumlarına göre erişebiliriz.

Böylece, `[-1]` son elemanı ve `[1:3]` ikinci ve üçüncü elemanları aşağıdaki gibi seçer:

```
X[-1], X[1:3]
```

```
(tensor([ 8.,  9., 10., 11.]),
 tensor([[ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.]]))
```

Okumanın ötesinde, indeksleri belirterek matrisin elemanlarını da yazabiliriz.

```
X[1, 2] = 9
X
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  9.,  7.],
        [ 8.,  9., 10., 11.]])
```

Birden fazla öğeye aynı değeri atamak istiyorsak, hepsini indeksleriz ve sonra da değer atarız. Örneğin, `[0:2, :]` birinci ve ikinci satırlara erişir, burada `:` Eksen 1 (sütun) boyunca tüm elemanları alır. Biz burada matrisler için indekslemeyi tartışırken, anlatılanlar açıkça vektörler ve 2'den fazla boyuttaki tensörler için de geçerlidir.

```
X[0:2, :] = 12
X
```

```
tensor([[12., 12., 12., 12.],
        [12., 12., 12., 12.],
        [ 8.,  9., 10., 11.]])
```

2.1.5 Belleği Kaydetme

Koşulan işlemler, sonuçların tutulması için yeni bellek ayrılmasına neden olabilir. Örneğin, `Y = X + Y` yazarsak, `Y`'yi göstermek için kullanılan tensörden vazgeçer ve bunun yerine yeni ayrılan bellekteki `Y`'yi işaret ederiz. Aşağıdaki örnekte, bunu, bize bellekteki referans edilen nesnenin tam adresini veren Python'un `id()` fonksiyonu ile gösteriyoruz. `Y = Y + X` komutunu çalıştırdıktan sonra, `id(Y)` ifadesinin farklı bir yeri gösterdiğini göreceğiz. Bunun nedeni, Python'un önce `Y + X` değerini hesaplayarak sonuç için yeni bellek ayırması ve ardından `Y`'yi bellekteki bu yeni konuma işaret etmesidir.

```
onceki = id(Y)
Y = Y + X
id(Y) == onceki
```

```
False
```

Bu iki nedenden dolayı istenmez olabilir. Birincisi, her zaman gereksiz yere bellek ayırmaya çalışmak istemiyoruz. Makine öğrenmesinde yüzlerce megabayt parametreye sahip olabilir ve hepsini saniyede birkaç kez güncelleyebiliriz. Genellikle, bu güncellemeleri *yerinde* yapmak isteyeceğiz. İkinci olarak, birden çok değişkenden aynı parametrelere işaret edebiliriz. Yerinde güncelleme yapmazsak, diğer referanslar hala eski bellek konumuna işaret eder ve bu da kodumuzun bazı bölümlerinin yanlışlıkla eski parametrelere atıfta bulunmasını olası kılar.

Neyse ki, yerinde işlemler yapmak kolaydır. Bir işlemin sonucunu daha önce ayrılmış bir diziye dilim gösterimi ile atayabiliriz, örneğin, `Y[:] = <ifade>`. Bu kavramı göstermek için, önce başka bir `Y` ile aynı şekle sahip yeni bir `Z` matrisi yaratıyoruz ve bir blok 0 girdisi tahsis etmek üzere `zeros_like`'i kullanıyoruz.

```
Z = torch.zeros_like(Y)
print('id(Z):', id(Z))
Z[:] = X + Y
print('id(Z):', id(Z))
```

```
id(Z): 139926018572912
id(Z): 139926018572912
```

`X` değeri sonraki hesaplamalarda yeniden kullanılmazsa, işlemin bellek yükünü azaltmak için `X[:] = X + Y` veya `X += Y` kullanabiliriz.

```
onceki = id(X)
X += Y
id(X) == onceki
```

```
True
```

2.1.6 Diğer Python Nesnelerine Dönüştürme

NumPy tensörüne (`ndarray`) dönüştürmek veya tam tersi kolaydır. Torch Tensörü ve numpy dizisi, temeldeki bellek konumlarını paylaşacak ve yerinde bir işlemle birini değiştirmek diğerini de değiştirecektir.

```
A = X.numpy()
B = torch.from_numpy(A)
type(A), type(B)
```

```
(numpy.ndarray, torch.Tensor)
```

1-boyutlu tensörünü bir Python skalerine (sayılına) dönüştürmek için `item` işlevini veya Python'un yerleşik işlevlerini çağırabiliriz.

```
a = torch.tensor([3.5])
a, a.item(), float(a), int(a)
```

```
(tensor([3.5000]), 3.5, 3.5, 3)
```

2.1.7 Özet

- Derin öğrenme için veri depolamada ve oynama yapmada ana arayüz tensördür (n-boyutlu dizi). Temel matematik işlemleri, yayınlama, indeksleme, dilimleme, bellek tasarrufu ve diğer Python nesnelerine dönüştürme gibi çeşitli işlevler sağlar.

2.1.8 Alıştırmalar

1. Bu bölümdeki kodu çalıştırın. Bu bölümdeki `X == Y` koşullu ifadesini, `X < Y` veya `X > Y` olarak değiştirin ve sonra ne tür bir tensör alabileceğinizi görün.
2. Yayın mekanizmasındaki öğeye göre çalışan iki tensörü, diğer şekillerle, örneğin 3 boyutlu tensörler ile değiştirin. Sonuç beklendiği gibi mi?

Tartışmalar[38]

2.2 Veri Ön İşleme

Şimdiye kadar, zaten tensörlerde saklanan verilerde oynama yapmak için çeşitli teknikleri tanıştırdık. Gerçek dünyadaki problemleri çözmede derin öğrenme uygularken, genellikle tensör biçiminde güzel hazırlanmış veriler kullanmak yerine ham verileri ön işleyerek başlarız. Python'daki popüler veri analitik araçları arasında `pandas` paketi yaygın olarak kullanılmaktadır. Python'un geniş ekosistemindeki diğer birçok uzatma paketi gibi, `pandas` da tensörler ile birlikte çalışabilir. Bu nedenle, ham verileri `pandas` ile ön işleme ve bunları tensör formatına dönüştürme adımlarını kısaca ele alacağız. Daha sonraki bölümlerde daha çok veri ön işleme tekniğini ele alacağız.

2.2.1 Veri Kümesini Okuma

Örnek olarak, `../data/house_tiny.csv` isimli csv (virgülle ayrılmış değerler) dosyasında saklanan yapay bir veri kümesi yaratarak başlıyoruz. Diğer formatlarda saklanan veriler de benzer şekilde işlenebilir.

Aşağıda veri kümesini satır satır bir csv dosyasına yazıyoruz.

```
import os

os.makedirs(os.path.join('..', 'data'), exist_ok=True)
veri_dosyasi = os.path.join('..', 'data', 'house_tiny.csv')
with open(veri_dosyasi, 'w') as f:
    f.write('NumRooms,Alley,Price\n')  # Sütun adları
    f.write('NA,Pave,127500\n')  # Her satır bir örnek temsil eder
    f.write('2,NA,106000\n')
    f.write('4,NA,178100\n')
    f.write('NA,NA,140000\n')
```

[38] https://discuss.d2l.ai/t/27

Oluşturulan csv dosyasından ham veri kümesini yüklemek için pandas paketini içe aktarıyoruz ve read_csv işlevini çağırıyoruz. Bu veri kümesinde dört satır ve üç sütun bulunur; burada her satırda oda sayısı ("NumRooms"), sokak tipi ("Alley") ve evin fiyatı ("Price") açıklanır.

```
# Eğer pandas kurulu ise alt satırdaki yorumu kaldır
# !pip install pandas
import pandas as pd

veri = pd.read_csv(veri_dosyasi)
print(veri)
```

```
   NumRooms Alley   Price
0       NaN  Pave  127500
1       2.0   NaN  106000
2       4.0   NaN  178100
3       NaN   NaN  140000
```

2.2.2 Eksik Verileri İşleme

"NaN" girdilerinin eksik değerler olduğuna dikkat edin. Eksik verilerle başa çıkmak için, tipik yöntemler arasında *itham etme* ve *silme* yer alır; burada itham etme eksik değerleri ikame edenlerle değiştirir, silme ise eksik değerleri yok sayar. Burada ithamı ele alacağız.

Tamsayı-konuma dayalı indeksleme (iloc) ile, veriyi girdilere ve ciktilara (çıktılar) böldük, burada girdi ilk iki sütuna, çıktı ise son sütuna denk gelir. Girdilerdeki eksik sayısal değerler için, "NaN" girdilerini aynı sütunun ortalama değeri ile değiştiriyoruz.

```
girdiler, ciktilar = veri.iloc[:, 0:2], veri.iloc[:, 2]
girdiler = girdiler.fillna(girdiler.mean())
print(girdiler)
```

```
   NumRooms Alley
0       3.0  Pave
1       2.0   NaN
2       4.0   NaN
3       3.0   NaN
```

Girdilerdeki kategorik veya ayrık değerler için NaN'i bir kategori olarak kabul ediyoruz. "Alley" sütunu yalnızca "Pave" ve "NaN" olmak üzere iki tür kategorik değer aldığından, pandas bu sütunu otomatik olarak "Alley_Pave" ve "Alley_nan" sütunlarına dönüştürebilir. Sokak tipi "Pave" olan bir satır "Alley_Pave" ve "Alley_nan" değerlerini 1 ve 0 olarak tayin eder. Sokak türü eksik olan bir satır, değerlerini 0 ve 1 olarak tayin eder.

```
girdiler = pd.get_dummies(girdiler, dummy_na=True)
print(girdiler)
```

```
   NumRooms  Alley_Pave  Alley_nan
0       3.0           1          0
1       2.0           0          1
```

(continues on next page)

(continued from previous page)

```
2       4.0           0          1
3       3.0           0          1
```

2.2.3 Tensör Formatına Dönüştürme

Artık `girdiler`deki ve `ciktilar`daki tüm girdi değerleri sayısal olduğundan, bunlar tensör formatına dönüştürülebilir. Veriler bu formatta olduğunda, daha önce tanıttığımız tensör işlevleri ile daha fazla oynama yapılabilir Section 2.1.

```
import torch

X, y = torch.tensor(girdiler.values), torch.tensor(ciktilar.values)
X, y
```

```
(tensor([[3., 1., 0.],
         [2., 0., 1.],
         [4., 0., 1.],
         [3., 0., 1.]], dtype=torch.float64),
 tensor([127500, 106000, 178100, 140000]))
```

2.2.4 Özet

- Python'un geniş ekosistemindeki diğer birçok eklenti paketi gibi, `pandas` da tensörler ile birlikte çalışabilir.
- İtham etme ve silme, eksik verilerle baş etmek için kullanılabilir.

2.2.5 Alıştırmalar

Daha fazla satır ve sütun içeren bir ham veri kümesi oluşturun.

1. En fazla eksik değerlere sahip sütunu silin.
2. Ön işlenmiş veri kümesini tensör formatına dönüştürün.

Tartışmalar[39]

2.3 Doğrusal Cebir

Şimdi verileri saklayabileceğinize ve oynama yapabileceğinize göre, bu kitapta yer alan modellerin çoğunu anlamanız ve uygulamanızda gereken temel doğrusal cebir bilgilerini kısaca gözden geçirelim. Aşağıda, doğrusal cebirdeki temel matematiksel nesneleri, aritmetiği ve işlemleri tanıtarak, bunların her birini matematiksel gösterim ve koddaki ilgili uygulama ile ifade ediyoruz.

[39] https://discuss.d2l.ai/t/29

2.3.1 Sayıllar

Doğrusal cebir veya makine öğrenmesi üzerine hiç çalışmadıysanız, matematikle ilgili geçmiş deneyiminiz muhtemelen bir seferde bir sayı düşünmekten ibaretti. Ayrıca, bir çek defterini dengelediyseniz veya hatta bir restoranda akşam yemeği için ödeme yaptıysanız, bir çift sayıyı toplama ve çarpma gibi temel şeyleri nasıl yapacağınızı zaten biliyorsunuzdur. Örneğin, Palo Alto'daki sıcaklık 52 Fahrenheit derecedir. Usul olarak, sadece bir sayısal miktar içeren değerlere *sayıl (skaler)* diyoruz. Bu değeri Celsius'a (metrik sistemin daha anlamlı sıcaklık ölçeği) dönüştürmek istiyorsanız, f'i 52 olarak kurup $c = \frac{5}{9}(f - 32)$ ifadesini hesaplarsınız. Bu denklemde —5, 9 ve 32— terimlerinin her biri skaler değerlerdir. c ve f göstermelik ifadelerine (placeholders) *değişkenler* denir ve bilinmeyen skaler değerleri temsil ederler.

Bu kitapta, skaler değişkenlerin normal küçük harflerle (ör. x, y ve z) gösterildiği matematiksel gösterimi kabul ediyoruz. Tüm (sürekli) *gerçel değerli* skalerlerin alanını $\mathbb{R}$ ile belirtiyoruz. Amaca uygun olarak, tam olarak *uzayın* ne olduğunu titizlikle tanımlayacağız, ancak şimdilik sadece $x \in \mathbb{R}$ ifadesinin x değerinin gerçel değerli olduğunu söylemenin usula uygun bir yolu olduğunu unutmayın. $\in$ sembolü "içinde" olarak telaffuz edilebilir ve sadece bir kümeye üyeliği belirtir. Benzer şekilde, x ve y'nin değerlerinin yalnızca 0 veya 1 olabilen rakamlar olduğunu belirtmek için $x, y \in \{0, 1\}$ yazabiliriz.

Skaler, sadece bir elemente sahip bir tensör ile temsil edilir. Sıradaki kod parçasında, iki skalere değer atıyoruz ve onlarla toplama, çarpma, bölme ve üs alma gibi bazı tanıdık aritmetik işlemleri gerçekleştiriyoruz.

```
import torch

x = torch.tensor(3.0)
y = torch.tensor(2.0)

x + y, x * y, x / y, x**y
```

```
(tensor(5.), tensor(6.), tensor(1.5000), tensor(9.))
```

2.3.2 Vektörler (Yöneyler)

Bir vektörü basitçe skaler değerlerin bir listesi olarak düşünebilirsiniz. Bu değerlere vektörün *elemanları* (*giriş değerleri* veya *bileşenleri*) diyoruz. Vektörlerimiz veri kümemizdeki örnekleri temsil ettiğinde, değerleri gerçek dünyadaki önemini korur. Örneğin, bir kredinin temerrüde düşme (ödenmeme) riskini tahmin etmek için bir model geliştiriyorsak, her başvuru sahibini, gelirine, istihdam süresine, önceki temerrüt sayısına ve diğer faktörlere karşılık gelen bileşenleri olan bir vektörle ilişkilendirebiliriz. Hastanedeki hastaların potansiyel olarak yaşayabilecekleri kalp krizi riskini araştırıyor olsaydık, her hastayı bileşenleri en güncel hayati belirtilerini, kolesterol seviyelerini, günlük egzersiz dakikalarını vb. içeren bir vektörle temsil edebilirdik. Matematiksel gösterimlerde, genellikle vektörleri kalın, küçük harfler (örneğin, $\mathbf{x}$, $\mathbf{y}$ ve $\mathbf{z}$) olarak göstereceğiz.

Vektörlerle tek boyutlu tensörler aracılığıyla çalışırız. Genelde tensörler, makinenizin bellek sınırlarına tabi olarak keyfi uzunluklara sahip olabilir.

```
x = torch.arange(4)
x
```

```
tensor([0, 1, 2, 3])
```

Bir vektörün herhangi bir öğesini bir altindis kullanarak tanımlayabiliriz. Örneğin, $\mathbf{x}$'in $i.$ elemanını x_i ile ifade edebiliriz. x_i öğesinin bir skaler olduğuna dikkat edin, bu nedenle ondan bahsederken fontta kalın yazı tipi kullanmıyoruz. Genel literatür sütun vektörlerini vektörlerin varsayılan yönü olarak kabul eder, bu kitap da öyle kabullenir. Matematikte, $\mathbf{x}$ vektörü şu şekilde yazılabilir:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \tag{2.3.1}$$

burada $x_1, \ldots, x_n$ vektörün elemanlarıdır. Kod olarak, herhangi bir öğeye onu tensöre indeksleyerek erişiriz.

```
x[3]
```

```
tensor(3)
```

Uzunluk, Boyutluluk ve Şekil

Bazı kavramları tekrar gözden geçirelim Section 2.1. Bir vektör sadece bir sayı dizisidir. Ayrıca her dizinin bir uzunluğu olduğu gibi, her vektör de bir uzunluğa sahiptir. Matematiksel gösterimde, $\mathbf{x}$ vektörünün n gerçel değerli skalerlerden oluştuğunu söylemek istiyorsak, bunu $\mathbf{x} \in \mathbb{R}^n$ olarak ifade edebiliriz. Bir vektörün uzunluğuna genel olarak vektörün *boyutu* denir.

Sıradan bir Python dizisinde olduğu gibi, Python'un yerleşik (built-in) `len()` işlevini çağırarak bir tensörün uzunluğuna erişebiliriz.

```
len(x)
```

```
4
```

Bir tensör bir vektörü (tam olarak bir ekseni ile) temsil ettiğinde, uzunluğuna `.shape` özelliği ile de erişebiliriz. Şekil, tensörün her ekseni boyunca uzunluğu (boyutsallığı) listeleyen bir gruptur. Sadece bir ekseni olan tensörler için, şeklin sadece bir elemanı vardır.

```
x.shape
```

```
torch.Size([4])
```

"Boyut" kelimesinin bu bağlamlarda aşırı yüklenme eğiliminde olduğunu ve bunun da insanları şaşırtma yöneliminde olduğunu unutmayın. Açıklığa kavuşturmak için, bir *vektörün* veya *ekseninin* boyutluluğunu onun uzunluğuna atıfta bulunmak için kullanırız; yani bir vektörün veya eksenin eleman sayısı. Halbuki, bir tensörün boyutluluğunu, bir tensörün sahip olduğu eksen sayısını ifade etmek için kullanırız. Bu anlamda, bir tensörün bazı eksenlerinin boyutluluğu, bu eksenin uzunluğu olacaktır.

2.3.3 Matrisler

Vektörler, skalerleri sıfırdan birinci dereceye kadar genelleştirirken, matrisler de vektörleri birinci dereceden ikinci dereceye genelleştirir. Genellikle kalın, büyük harflerle (örn., **X**, **Y**, and **Z**) göstereceğimiz matrisler, kodda iki eksenli tensörler olarak temsil edilir.

Matematiksel gösterimde, **A** matrisinin gerçel değerli skaler m satır ve n sütundan oluştuğunu ifade etmek için $\mathbf{A} \in \mathbb{R}^{m \times n}$'i kullanırız. Görsel olarak, herhangi bir $\mathbf{A} \in \mathbb{R}^{m \times n}$ matrisini a_{ij} öğesinin $i.$ satıra ve $j.$ sütuna ait olduğu bir tablo olarak gösterebiliriz:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}. \tag{2.3.2}$$

Herhangi bir $\mathbf{A} \in \mathbb{R}^{m \times n}$ için, **A** (m, n) veya $m \times n$ şeklindedir. Özellikle, bir matris aynı sayıda satır ve sütuna sahip olduğunda, şekli bir kareye dönüşür; dolayısıyla buna *kare matris* denir.

Bir tensörü örneği yaratırken, en sevdiğimiz işlevlerden herhangi birini çağırıp m ve n iki bileşeninden oluşan bir şekil belirterek $m \times n$ matrisi oluşturabiliriz.

```
A = torch.arange(20).reshape(5, 4)
A
```

```
tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11],
        [12, 13, 14, 15],
        [16, 17, 18, 19]])
```

Satır (i) ve sütun (j) indekslerini belirterek **A** matrisinin a_{ij} skaler öğesine erişebiliriz, $[\mathbf{A}]_{ij}$ gibi. Bir **A** matrisinin skaler elemanları verilmediğinde, örneğin (2.3.2) gibi, basitçe **A** matrisinin küçük harfli altindislisi a_{ij}'yi kullanarak $[\mathbf{A}]_{ij}$'ye atıfta bulunuruz. Gösterimi basit tutarken indeksleri ayırmak için virgüller yalnızca gerekli olduğunda eklenir, örneğin $a_{2,3j}$ ve $[\mathbf{A}]_{2i-1,3}$ gibi.

Bazen eksenleri ters çevirmek isteriz. Bir matrisin satırlarını ve sütunlarını değiştirdiğimizde çıkan sonuç matrisine *devrik (transpose)* adı verilir. Usul olarak, bir **A**'nin devriğini $\mathbf{A}^\top$ ile gösteririz ve eğer $\mathbf{B} = \mathbf{A}^\top$ ise herhangi bir i and j için $b_{ij} = a_{ji}$'dir. Bu nedenle, (2.3.2)'deki **A**'nin devriği bir $n \times m$ matristir:

$$\mathbf{A}^\top = \begin{bmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{bmatrix}. \tag{2.3.3}$$

Şimdi kodda bir matrisin devriğine erişiyoruz.

```
A.T
```

```
tensor([[ 0,  4,  8, 12, 16],
        [ 1,  5,  9, 13, 17],
        [ 2,  6, 10, 14, 18],
        [ 3,  7, 11, 15, 19]])
```

Kare matrisin özel bir türü olarak, bir *simetrik matris* $\mathbf{A}$, devriğine eşittir: $\mathbf{A} = \mathbf{A}^\top$. Burada simetrik bir matrisi `B` diye tanımlıyoruz.

```
B = torch.tensor([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
B
```

```
tensor([[1, 2, 3],
        [2, 0, 4],
        [3, 4, 5]])
```

Şimdi `B`yi kendi devriğiyle karşılaştıralım.

```
B == B.T
```

```
tensor([[True, True, True],
        [True, True, True],
        [True, True, True]])
```

Matrisler yararlı veri yapılarıdır: Farklı değişim (varyasyon) kiplerine (modalite) sahip verileri düzenlememize izin verirler. Örneğin, matrisimizdeki satırlar farklı evlere (veri örneklerine) karşılık gelirken, sütunlar farklı özelliklere karşılık gelebilir. Daha önce elektronik tablo yazılımı kullandıysanız veya şurayı okuduysanız Section 2.2, bu size tanıdık gelecektir. Bu nedenle, tek bir vektörün varsayılan yönü bir sütun vektörü olmasına rağmen, bir tablo veri kümesini temsil eden bir matriste, her veri örneğini matristeki bir satır vektörü olarak ele almak daha gelenekseldir. Ve sonraki bölümlerde göreceğimiz gibi, bu düzen ortak derin öğrenme tatbikatlarını mümkün kılacaktır. Örneğin, bir tensörün en dış ekseni boyunca, veri örneklerinin minigruplarına veya minigrup yoksa yalnızca veri örneklerine erişebilir veya bir bir sayabiliriz.

2.3.4 Tensörler

Vektörlerin skalerleri genellemesi ve matrislerin vektörleri genellemesi gibi, daha fazla eksenli veri yapıları oluşturabiliriz. Tensörler (bu alt bölümdeki "tensörler" cebirsel nesnelere atıfta bulunur) bize rastgele sayıda ekseni olan n-boyutlu dizileri tanımlamanın genel bir yolunu verir. Vektörler, örneğin, birinci dereceden tensörlerdir ve matrisler ikinci dereceden tensörlerdir. Tensörler, özel bir yazı tipinin büyük harfleriyle (ör. `X`, `Y` ve `Z`) ve indeksleme mekanizmalarıyla (ör. X_{ijk} ve $[\mathsf{X}]_{1,2i-1,3}$), matrislerinkine benzer gösterilir.

Renk kanallarını (kırmızı, yeşil ve mavi) istiflemek için yükseklik, genişlik ve bir *kanal* eksenine karşılık gelen 3 eksene sahip n-boyutlu dizi olarak gelen imgelerle çalışmaya başladığımızda tensörler daha önemli hale gelecektir. Şimdilik, daha yüksek dereceli tensörleri atlayacağız ve temellere odaklanacağız.

```
X = torch.arange(24).reshape(2, 3, 4)
X
```

```
tensor([[[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]],
```

(continues on next page)

(continued from previous page)

```
        [[12, 13, 14, 15],
         [16, 17, 18, 19],
         [20, 21, 22, 23]]])
```

2.3.5 Tensör Aritmetiğinin Temel Özellikleri

Skalerler, vektörler, matrisler ve rasgele sayıda eksenli tensörler (bu alt bölümdeki "tensörler" cebirsel nesnelere atıfta bulunur), çoğu zaman kullanışlı olan bazı güzel özelliklere sahiptir. Örneğin, bir eleman yönlü işlemin tanımından, herhangi bir eleman yönlü tekli işlemin işlenen nesnenin şeklini değiştirmediğini fark etmiş olabilirsiniz. Benzer şekilde, aynı şekle sahip herhangi bir iki tensör göz önüne alındığında, herhangi bir ikili elemanlı işlemin sonucu, gene aynı şekle sahip bir tensör olacaktır. Örneğin, aynı şekle sahip iki matris toplama, bu iki matrisin üzerinde eleman yönlü toplama gerçekleştirir.

```
A = torch.arange(20, dtype=torch.float32).reshape(5, 4)
B = A.clone()  # `A`nın kopyasını yeni bellek tahsis ederek `B`ye atayın
A, A + B
```

```
(tensor([[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [12., 13., 14., 15.],
         [16., 17., 18., 19.]]),
 tensor([[ 0.,  2.,  4.,  6.],
         [ 8., 10., 12., 14.],
         [16., 18., 20., 22.],
         [24., 26., 28., 30.],
         [32., 34., 36., 38.]]))
```

Özellikle, iki matrisin eleman yönlü çarpımına *Hadamard çarpımı* (matematik gösterimi $\odot$) denir. i. satır ve j. sütununun öğesi b_{ij} olan $\mathbf{B} \in \mathbb{R}^{m \times n}$ matrisini düşünün. $\mathbf{A}$ ((2.3.2)'da tanımlanmıştır) ve $\mathbf{B}$ matrislerinin Hadamard çarpımı:

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \dots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{22}b_{22} & \dots & a_{2n}b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \dots & a_{mn}b_{mn} \end{bmatrix}. \tag{2.3.4}$$

```
A * B
```

```
tensor([[  0.,   1.,   4.,   9.],
        [ 16.,  25.,  36.,  49.],
        [ 64.,  81., 100., 121.],
        [144., 169., 196., 225.],
        [256., 289., 324., 361.]])
```

Bir tensörün skaler ile çarpılması veya toplanması, işlenen tensörün her elemanını skaler ile toplayacağından veya çarpacağından tensörün şeklini de değiştirmez.

```
a = 2
X = torch.arange(24).reshape(2, 3, 4)
a + X, (a * X).shape
```

```
(tensor([[[ 2,  3,  4,  5],
          [ 6,  7,  8,  9],
          [10, 11, 12, 13]],

         [[14, 15, 16, 17],
          [18, 19, 20, 21],
          [22, 23, 24, 25]]]),
 torch.Size([2, 3, 4]))
```

2.3.6 İndirgeme

Keyfi tensörlerle gerçekleştirebileceğimiz faydalı işlemlerden biri elemanlarının toplamını hesaplamaktır. Matematiksel gösterimde, $\sum$ sembolünü kullanarak toplamları ifade ederiz. d uzunluğa sahip $\mathbf{x}$ vektöründeki öğelerin toplamını ifade etmek için $\sum_{i=1}^{d} x_i$ yazarız. Kodda, toplamı hesaplamak için sadece ilgili işlevi çağırabiliriz.

```
x = torch.arange(4, dtype=torch.float32)
x, x.sum()
```

```
(tensor([0., 1., 2., 3.]), tensor(6.))
```

Rasgele şekilli tensörlerin elamanları üzerindeki toplamları ifade edebiliriz. Örneğin, $m \times n$ matris $\mathbf{A}$ öğelerinin toplamı $\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}$ diye yazılabilir.

```
A.shape, A.sum()
```

```
(torch.Size([5, 4]), tensor(190.))
```

Varsayılan olarak, toplamı hesaplama işlevini çağırmak, tüm eksenleri boyunca bir tensörü skalere *indirger*. Toplama yoluyla tensörün indirgendiği eksenleri de belirtebiliriz. Örnek olarak matrisleri alın. Tüm satırların öğelerini toplayarak satır boyutunu (eksen 0) indirgemek için, işlevi çağırırken `axis=0` değerini belirtiriz. Girdi matrisi, çıktı vektörü oluşturmak için eksen 0 boyunca indirgendiğinden, girdinin eksen 0 boyutu, çıktının şeklinde kaybolur.

```
A_sum_axis0 = A.sum(axis=0)
A_sum_axis0, A_sum_axis0.shape
```

```
(tensor([40., 45., 50., 55.]), torch.Size([4]))
```

`axis=1` olarak belirtmek, tüm sütunların öğelerini toplayarak sütun boyutunu (eksen 1) azaltacaktır. Böylece, girdinin eksen 1 boyutu, çıktının şeklinde kaybolur.

```
A_sum_axis1 = A.sum(axis=1)
A_sum_axis1, A_sum_axis1.shape
```

```
(tensor([ 6., 22., 38., 54., 70.]), torch.Size([5]))
```

Bir matrisin toplama yoluyla hem satırlar hem de sütunlar boyunca indirgenmesi, matrisin tüm öğelerinin toplanmasıyla eşdeğerdir.

```
A.sum(axis=[0, 1])  # `A.sum()` ile aynı
```

```
tensor(190.)
```

İlgili bir miktar da *ortalama*dır. Ortalamayı, toplamı toplam eleman sayısına bölerek hesaplıyoruz. Kod olarak, keyfi şekildeki tensörlerdeki ortalamanın hesaplanmasında ilgili işlevi çağırabiliriz.

```
A.mean(), A.sum() / A.numel()
```

```
(tensor(9.5000), tensor(9.5000))
```

Benzer şekilde, ortalama hesaplama fonksiyonu, belirtilen eksenler boyunca bir tensörü de indirgeyebilir.

```
A.mean(axis=0), A.sum(axis=0) / A.shape[0]
```

```
(tensor([ 8.,  9., 10., 11.]), tensor([ 8.,  9., 10., 11.]))
```

İndirgemesiz Toplama

Gene de, bazen toplamı veya ortalamayı hesaplamak için işlevi çağırırken eksen sayısını değiştirmeden tutmak yararlı olabilir.

```
sum_A = A.sum(axis=1, keepdims=True)
sum_A
```

```
tensor([[ 6.],
        [22.],
        [38.],
        [54.],
        [70.]])
```

Örneğin, `sum_A` her satırı topladıktan sonra hala iki eksenini koruduğundan,`A`'yı yayınlayarak `sum_A` ile bölebiliriz.

```
A / sum_A
```

```
tensor([[0.0000, 0.1667, 0.3333, 0.5000],
        [0.1818, 0.2273, 0.2727, 0.3182],
        [0.2105, 0.2368, 0.2632, 0.2895],
        [0.2222, 0.2407, 0.2593, 0.2778],
        [0.2286, 0.2429, 0.2571, 0.2714]])
```

Bir eksen boyunca `A`'nın öğelerinin biriktirilmiş (kümülatif) toplamını hesaplamak istiyorsak, `axis=0` diyelim (satır satır), `cumsum` işlevini çağırabiliriz. Bu işlev girdi tensörünü herhangi bir eksen boyunca indirgemez.

```
A.cumsum(axis=0)
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  6.,  8., 10.],
        [12., 15., 18., 21.],
        [24., 28., 32., 36.],
        [40., 45., 50., 55.]])
```

2.3.7 Nokta Çarpımları

Şimdiye kadar sadece eleman yönlü işlemler, toplamalar ve ortalamalar gerçekleştirdik. Ayrıca tüm yapabileceğimiz bu olsaydı, doğrusal cebir muhtemelen kendi bölümünü hak etmeyecekti. Bununla birlikte, en temel işlemlerden biri iç çarpımdır. İki vektör $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ verildiğinde, *iç çarpımları* $\mathbf{x}^\top \mathbf{y}$ (veya $\langle \mathbf{x}, \mathbf{y} \rangle$), aynı konumdaki öğelerin çarpımlarının toplamıdır: $\mathbf{x}^\top \mathbf{y} = \sum_{i=1}^{d} x_i y_i$.

```
y = torch.ones(4, dtype = torch.float32)
x, y, torch.dot(x, y)
```

```
(tensor([0., 1., 2., 3.]), tensor([1., 1., 1., 1.]), tensor(6.))
```

İki vektörün nokta çarpımlarını, eleman yönlü bir çarpma ve ardından bir toplam gerçekleştirerek eşit şekilde ifade edebileceğimizi unutmayın:

```
torch.sum(x * y)
```

```
tensor(6.)
```

Nokta çarpımları çok çeşitli bağlamlarda kullanışlıdır. Örneğin, $\mathbf{x} \in \mathbb{R}^d$ vektörü ve $\mathbf{w} \in \mathbb{R}^d$ ile belirtilen bir ağırlık kümesi verildiğinde, $\mathbf{x}$ içindeki değerlerin $\mathbf{w}$ ağırlıklarına göre ağırlıklı toplamı $\mathbf{x}^\top \mathbf{w}$ nokta çarpımı olarak ifade edilebilir. Ağırlıklar negatif olmadığında ve bire (örn., $\left(\sum_{i=1}^{d} w_i = 1\right)$) toplandığında, nokta çarpımı *ağırlıklı ortalama*yı ifade eder. İki vektörü birim uzunluğa sahip olacak şekilde normalleştirdikten sonra, nokta çarpımlar arasındaki açının kosinüsünü ifade eder. *Uzunluk* kavramını bu bölümün ilerleyen kısımlarında usüle uygun tanıtacağız.

2.3.8 Matris-Vektör Çarpımları

Artık nokta çarpımlarını nasıl hesaplayacağımızı bildiğimize göre, *matris-vektör çarpımları*nı anlamaya başlayabiliriz. $\mathbf{A} \in \mathbb{R}^{m \times n}$ matrisini ve $\mathbf{x} \in \mathbb{R}^n$ vektörünü sırasıyla tanımladık ve (2.3.2) ve (2.3.1)'de görselleştirdik. $\mathbf{A}$ matrisini satır vektörleriyle görselleştirerek başlayalım.

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix}, \tag{2.3.5}$$

burada her $\mathbf{a}_i^\top \in \mathbb{R}^n$, $\mathbf{A}$ matrisinin $i.$ satırını temsil eden bir satır vektörüdür. Matris-vektör çarpımı $\mathbf{Ax}$, basitçe $i.$ elemanı $\mathbf{a}_i^\top \mathbf{x}$ iç çarpımı olan m uzunluğunda bir sütun vektörüdür.

$$\mathbf{Ax} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{x} \\ \mathbf{a}_2^\top \mathbf{x} \\ \vdots \\ \mathbf{a}_m^\top \mathbf{x} \end{bmatrix}. \tag{2.3.6}$$

$\mathbf{A} \in \mathbb{R}^{m \times n}$ matrisi ile çarpmayı vektörleri $\mathbb{R}^n$'den $\mathbb{R}^m$'e yansıtan bir dönüşüm olarak düşünebiliriz. Bu dönüşümler oldukça faydalı oldu. Örneğin, döndürmeleri bir kare matrisle çarpma olarak gösterebiliriz. Sonraki bölümlerde göreceğimiz gibi, bir önceki katmanın değerleri göz önüne alındığında, bir sinir ağındaki her bir katman hesaplanırken gereken en yoğun hesaplamaları tanımlamak için matris-vektör çarpımlarını da kullanabiliriz.

Matris-vektör çarpımlarını tensörlerle kodda ifade ederken, `mv` işlevini kullanırız. `A` matrisi ve `x` vektörü ile `torch.mv(A, x)` dediğimizde matris-vektör çarpımı gerçekleştirilir. `A` sütun boyutunun (eksen 1 boyunca uzunluğu) `x` boyutuyla (uzunluğu) aynı olması gerektiğini unutmayın.

```
A.shape, x.shape, torch.mv(A, x)
```

```
(torch.Size([5, 4]), torch.Size([4]), tensor([ 14.,  38.,  62.,  86., 110.]))
```

2.3.9 Matris-Matris Çarpımı

Eğer nokta ve matris-vektör çarpımlarını anladıysanız, *matris-matris çarpımı* açık olmalıdır.

$\mathbf{A} \in \mathbb{R}^{n \times k}$ ve $\mathbf{B} \in \mathbb{R}^{k \times m}$ diye iki matrisimizin olduğunu diyelim:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k1} & b_{k2} & \cdots & b_{km} \end{bmatrix}. \tag{2.3.7}$$

$\mathbf{A}$ matrisinin $i.$ satırını temsil eden satır vektörünü $\mathbf{a}_i^\top$ ile belirtelim ve $\mathbf{B}$ matrisinin $j.$ sütunu da $\mathbf{b}_j$ olsun. $\mathbf{C} = \mathbf{AB}$ matris çarpımını üretmek için $\mathbf{A}$'yı satır vektörleri ve $\mathbf{B}$'yi sütun vektörleri ile düşünmek en kolay yoldur:

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_n^\top \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_m \end{bmatrix}. \tag{2.3.8}$$

Daha sonra, $\mathbf{C} \in \mathbb{R}^{n \times m}$ matris çarpımı her bir c_{ij} öğesi $\mathbf{a}_i^\top \mathbf{b}_j$ nokta çarpımı hesaplanarak üretilir:

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_n^\top \end{bmatrix} \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_m \end{bmatrix} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{b}_1 & \mathbf{a}_1^\top \mathbf{b}_2 & \cdots & \mathbf{a}_1^\top \mathbf{b}_m \\ \mathbf{a}_2^\top \mathbf{b}_1 & \mathbf{a}_2^\top \mathbf{b}_2 & \cdots & \mathbf{a}_2^\top \mathbf{b}_m \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}_n^\top \mathbf{b}_1 & \mathbf{a}_n^\top \mathbf{b}_2 & \cdots & \mathbf{a}_n^\top \mathbf{b}_m \end{bmatrix}. \tag{2.3.9}$$

Matris-matris çarpımı **AB**'yi sadece m tane matris-vektör çarpımı gerçekleştirmek ve sonuçları $n \times m$ matrisi oluşturmak için birleştirmek olarak düşünebiliriz. Aşağıdaki kod parçasında, `A` ve `B` üzerinde matris çarpımı yapıyoruz. Burada `A`, 5 satır ve 4 sütunlu bir matristir ve `B` 4 satır ve 3 sütunlu bir matristir. Çarpma işleminden sonra 5 satır ve 3 sütun içeren bir matris elde ederiz.

```
B = torch.ones(4, 3)
torch.mm(A, B)
```

```
tensor([[ 6.,  6.,  6.],
        [22., 22., 22.],
        [38., 38., 38.],
        [54., 54., 54.],
        [70., 70., 70.]])
```

Matris-matris çarpımı basitçe *matris çarpımı* olarak adlandırılabilir ve Hadamard çarpımı ile karıştırılmamalıdır.

2.3.10 Normlar (Büyüklükler)

Doğrusal cebirde en kullanışlı operatörlerden bazıları *normlardır*. Gayri resmi olarak, bir vektörün normu bize bir vektörün ne kadar *büyük* olduğunu söyler. Burada ele alınan *ebat* kavramı boyutsallık değil, bileşenlerin büyüklüğü ile ilgilidir.

Doğrusal cebirde, bir vektör normu, bir vektörü bir skalere eşleştiren ve bir avuç dolusu özelliği karşılayan bir f fonksiyonudur. Herhangi bir $\mathbf{x}$ vektörü verildiğinde, ilk özellik, bir vektörün tüm elemanlarını sabit bir α çarpanına göre ölçeklendirirsek, normunun da aynı sabit çarpanın *mutlak değerine* göre ölçeklendiğini söyler:

$$f(\alpha \mathbf{x}) = |\alpha| f(\mathbf{x}). \tag{2.3.10}$$

İkinci özellik, tanıdık üçgen eşitsizliğidir:

$$f(\mathbf{x} + \mathbf{y}) \leq f(\mathbf{x}) + f(\mathbf{y}). \tag{2.3.11}$$

Üçüncü özellik, normun negatif olmaması gerektiğini söyler:

$$f(\mathbf{x}) \geq 0. \tag{2.3.12}$$

Çoğu bağlamda herhangi bir şey için en küçük *ebat* 0 olduğu için bu mantıklıdır. Nihai özellik, en küçük normun sadece ve sadece tüm sıfırlardan oluşan bir vektör tarafından elde edilmesini gerektirir.

$$\forall i, [\mathbf{x}]_i = 0 \Leftrightarrow f(\mathbf{x}) = 0. \tag{2.3.13}$$

Normların mesafe ölçülerine çok benzediğini fark edebilirsiniz. Ayrıca eğer ilkokuldan Öklid mesafesini hatırlarsanız (Pisagor teoremini düşünün), o zaman negatif olmama ve üçgen eşitsizlik kavramları zihininizde bir zil çalabilir. Aslında Öklid mesafesi bir normdur: Özellikle L_2 normudur. n boyutlu vektör, $\mathbf{x}$, içindeki öğelerin $x_1, \ldots, x_n$ olduğunu varsayalım. $\mathbf{x}$'in L_2 *normu*, vektör öğelerinin karelerinin toplamının karekökü dür:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}, \tag{2.3.14}$$

burada 2 altindisi genellikle L_2 normlarında atlanır, yani, $\|\mathbf{x}\|$, $\|\mathbf{x}\|_2$ ile eşdeğerdir. Kodda, bir vektörün L_2 normunu aşağıdaki gibi hesaplayabiliriz.

```
u = torch.tensor([3.0, -4.0])
torch.norm(u)
```

```
tensor(5.)
```

Derin öğrenmede, kare L_2 normuyla daha sık çalışırız. Ayrıca, vektör öğelerinin mutlak değerlerinin toplamı olarak ifade edilen L_1 *normu* ile de sık karşılaşacaksınız:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i| \,. \tag{2.3.15}$$

L_2 normuna kıyasla, sıradışı (aykırı) değerlerden daha az etkilenir. L_1 normunu hesaplamak için, elemanların toplamı üzerinde mutlak değer fonksiyonunu oluştururuz.

```
torch.abs(u).sum()
```

```
tensor(7.)
```

Hem L_2 normu hem de L_1 normu, daha genel L_p *normu*nun özel durumlarıdır:

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p}. \tag{2.3.16}$$

L_2 vektör normlarına benzer bir şekilde, $\mathbf{X} \in \mathbb{R}^{m \times n}$ matrisinin *Frobenius normu*, matris elemanlarının karelerin toplamının kare köküdür:

$$\|\mathbf{X}\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij}^2}. \tag{2.3.17}$$

Frobenius normu, vektör normlarının tüm özelliklerini karşılar. Matris şeklindeki bir vektörün bir L_2 normu gibi davranır. Aşağıdaki işlevi çağırmak, bir matrisin Frobenius normunu hesaplar.

```
torch.norm(torch.ones((4, 9)))
```

```
tensor(6.)
```

Kendimizi aşmak istemesek de, şimdiden bu kavramların neden faydalı olduğuna dair bazı sezgiler ekleyebiliriz. Derin öğrenmede, genellikle optimizasyon (eniyileme) sorunlarını çözmeye çalışıyoruz: Gözlenen verilere atanan olasılığı *en üst düzeye çıkar*; tahminler ve gerçek-doğru gözlemler arasındaki mesafeyi *en aza indir*. Benzer öğeler arasındaki mesafe en aza indirilecek ve benzer olmayan öğeler arasındaki mesafe en üst düzeye çıkarılacak şekilde öğelere (kelimeler, ürünler veya haber makaleleri gibi) vektör ifadeleri ata. Çoğu zaman, amaç fonksiyonları, ki belki de derin öğrenme algoritmalarının (verilerin yanı sıra) en önemli bileşenleridir, normlar cinsinden ifade edilir.

2.3.11 Doğrusal Cebir Hakkında Daha Fazlası

Sadece bu bölümde, modern derin öğrenmenin dikkate değer bir bölümünü anlamak için ihtiyaç duyacağınız tüm doğrusal cebiri öğrettik. Doğrusal cebirde çok daha fazlası vardır ve daha fazla matematik makine öğrenmesi için yararlıdır. Örneğin, matrisler faktörlere ayrıştırılabilir ve bu ayrışmalar gerçek dünya veri kümelerinde düşük boyutlu yapıları ortaya çıkarabilir. Veri kümelerindeki yapıyı keşfetmek ve tahmin problemlerini çözmek için matris ayrıştırmalarına ve onların yüksek dereceli tensörlere genellemelerini kullanmaya odaklanan koca makine öğrenmesi alt alanları vardır. Ancak bu kitap derin öğrenmeye odaklanmaktadır. Gerçek veri kümelerinde faydalı makine öğrenmesi modelleri uygulayarak ellerinizi kirlettikten sonra daha fazla matematik öğrenmeye çok daha meyilli olacağınıza inanıyoruz. Bu nedenle, daha sonra daha fazla matematiği öne sürme hakkımızı saklı tutarken, bu bölümü burada toparlayacağız.

Doğrusal cebir hakkında daha fazla bilgi edinmek istiyorsanız, şunlardan birine başvurabilirsiniz: Doğrusal cebir işlemleri üzerine çevrimiçi ek[40] veya diğer mükemmel kaynaklar (Kolter, 2008, Petersen *et al.*, 2008, Strang, 1993).

2.3.12 Özet

- Skalerler, vektörler, matrisler ve tensörler doğrusal cebirdeki temel matematiksel nesnelerdir.
- Vektörler skaleri genelleştirir ve matrisler vektörleri genelleştirir.
- Skalerler, vektörler, matrisler ve tensörler sırasıyla sıfır, bir, iki ve rastgele sayıda eksene sahiptir.
- Bir tensör, belirtilen eksenler boyunca `toplam` ve `ortalama` ile indirgenebilir.
- İki matrisin eleman yönlü olarak çarpılmasına Hadamard çarpımı denir. Matris çarpımından farklıdır.
- Derin öğrenmede, genellikle L_1 normu, L_2 normu ve Frobenius normu gibi normlarla çalışırız.
- Skalerler, vektörler, matrisler ve tensörler üzerinde çeşitli işlemler gerçekleştirebiliriz.

[40] https://tr.d2l.ai/chapter_appendix-mathematics-for-deep-learning/geometry-linear-algebraic-ops.html

2.3.13 Alıştırmalar

1. **A**'nın devrik bir matrisinin devrik işleminin **A**, yani $(\mathbf{A}^\top)^\top = \mathbf{A}$ olduğunu kanıtlayın.
2. **A** ve **B** matrisleri verildiğinde, devriklerin toplamının bir toplamın devriğine eşit olduğunu gösterin: $\mathbf{A}^\top + \mathbf{B}^\top = (\mathbf{A} + \mathbf{B})^\top$.
3. Herhangi bir kare matris **A** verildiğinde, $\mathbf{A} + \mathbf{A}^\top$ her zaman simetrik midir? Neden?
4. Bu bölümde (2, 3, 4) şeklinde `X` tensörünü tanımladık. `len(X)` çıktısı nedir?
5. Rasgele şekilli bir tensör `X` için, `len(X)` her zaman belirli bir `X` ekseninin uzunluğuna karşılık gelir mi? Bu eksen nedir?
6. `A / A.sum(axis=1)` komutunu çalıştırın ve ne olduğuna bakın. Sebebini analiz edebilir misiniz?
7. Manhattan'da iki nokta arasında seyahat ederken, koordinatlar açısından, yani cadde ve sokak cinsinden, kat etmeniz gereken mesafe nedir? Çaprazlama seyahat edebilir misiniz?
8. (2, 3, 4) şekilli bir tensörü düşünün. 0, 1 ve 2 ekseni boyunca toplam çıktılarının şekilleri nelerdir?
9. 3 veya daha fazla eksenli bir tensörü `linalg.norm` fonksiyonuna besleyin ve çıktısını gözlemleyin. Bu işlev keyfi şekilli tansörler için ne hesaplar?

Tartışmalar[41]

2.4 Hesaplama (Kalkülüs)

Bir çokgenin alanını bulmak, eski Yunanlıların bir çokgeni üçgenlere böldüğü ve alanlarını topladığı en az 2.500 yıl öncesine kadar gizemli kalmıştı. Bir daire gibi kavisli şekillerin alanını bulmak için, eski Yunanlılar bu şekillerin içine çokgenler yerleştirdiler. Fig. 2.4.1'de gösterildiği gibi, eşit uzunlukta daha fazla kenarı olan çizili bir çokgen daireye bayağı yaklaşır. Bu işlem *tüketme yöntemi* olarak da bilinir.

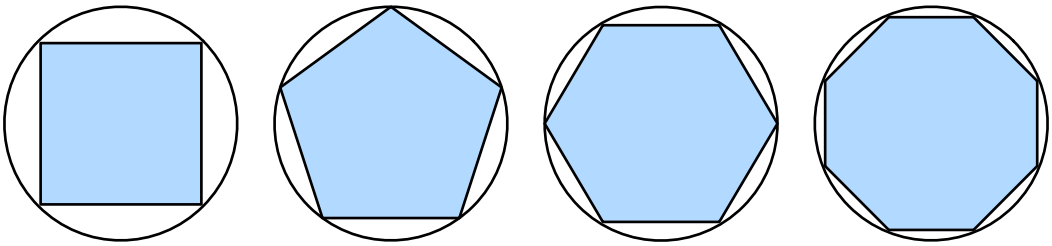


Fig. 2.4.1: Tüketme yöntemiyle bir dairenin alanını bulun.

Aslında, tüketme yöntemi *integral hesabının* (şurada açıklanacaktır Section 18.5) kaynaklandığı yerdir. 2.000 yıldan fazla bir müddetten sonra, diğer kalkülüs alanı, *diferansiyel (türevsel) kalkülüs* icat edildi. Diferansiyel kalkülüsün en kritik uygulamaları arasındaki optimizasyon problemleri bir şeyin nasıl *en iyi* şekilde yapılacağına kafa yorar. Section 2.3.10'de tartışıldığı gibi, bu tür sorunlar derin öğrenmede her yerde bulunur.

[41] https://discuss.d2l.ai/t/31

Derin öğrenmede, modelleri daha fazla veri gördükçe daha iyi ve daha iyi olmaları için arka arkaya güncelleyerek *eğitiyoruz*. Genellikle, daha iyi olmak, "modelimiz ne kadar *kötü*?" sorusuna cevap veren bir skor olan *kayıp (yitim) fonksiyonunu* en aza indirmek anlamına gelir. Bu soru göründüğünden daha zekicedir. Sonuçta, gerçekten önemsediğimiz, daha önce hiç görmediğimiz veriler üzerinde iyi performans gösteren bir model üretmektir. Ancak modeli yalnızca gerçekten görebildiğimiz verilere uydurabiliriz. Böylece modellerin uydurulması görevini iki temel kaygıya ayırabiliriz: (i) *Eniyileme*: Modellerimizi gözlemlenen verilere uydurma süreci; (ii) *Genelleme*: Geçerliliği onları eğitmek için kullanılan kesin veri örnekleri kümesinin ötesine geçen modellerin nasıl üretileceğinde bize rehberlik eden matematiksel ilkelerin ve uygulayıcılarının bilgeliği.

Daha sonraki bölümlerde optimizasyon problemlerini ve yöntemlerini anlamanıza yardımcı olmak için burada, derin öğrenmede yaygın olarak kullanılan diferansiyel hesaplama hakkında bir tutam bilgi veriyoruz.

2.4.1 Türev ve Türev Alma

Hemen hemen tüm derin öğrenme optimizasyon algoritmalarında önemli bir adım olan türevlerin hesaplanmasını ele alarak başlıyoruz. Derin öğrenmede, tipik olarak modelimizin parametrelerine göre türevi alınabilen kayıp fonksiyonlarını seçeriz. Basitçe ifade etmek gerekirse, bu, her parametre için, o parametreyi sonsuz derecede küçük bir miktarda *arttırırsak* veya *azaltırsak* kaybın ne kadar hızlı artacağını veya azalacağını belirleyebileceğimiz anlamına gelir.

Girdi ve çıktıların her ikisi de skaler olan $f : \mathbb{R} \rightarrow \mathbb{R}$ fonksiyonumuz olduğunu varsayalım. f'nin *türevi* şöyle tanımlanır:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}, \tag{2.4.1}$$

eğer bu limit varsa. $f'(a)$ varsa, f'nin a'da *türevlenebilir* olduğu söylenir. f, bir aralığın her sayısında türevlenebilirse, o zaman bu fonksiyon bu aralıkta türevlenebilir. $f'(x)$'in (2.4.1)'deki türevini $f(x)$'in x'e göre *anlık* değişim oranı olarak yorumlayabiliriz. Sözde anlık değişim oranı, x cinsinden h 0'a yaklaşırken değişimini temel alır.

Türevleri açıklamayı için bir örnekle deneyelim. $u = f(x) = 3x^2 - 4x$ tanımlayın.

```
%matplotlib inline
import numpy as np
from matplotlib_inline import backend_inline
from d2l import torch as d2l


def f(x):
    return 3 * x ** 2 - 4 * x
```

$x = 1$ diye ayarlayıp h değerinin 0 değerine yaklaşmasına izin verince $\frac{f(x+h) - f(x)}{h}$ (2.4.1)'in sayısal sonucu 2'ye yaklaşır. Bu deney matematiksel bir kanıt olmasa da, daha sonra u' türevinin $x = 1$ olduğunda 2 olduğunu göreceğiz.

```
def numerical_lim(f, x, h):
    return (f(x + h) - f(x)) / h

h = 0.1
for i in range(5):
```

(continues on next page)

(continued from previous page)

```
    print(f'h={h:.5f}, numerik limit={numerical_lim(f, 1, h):.5f}')
    h *= 0.1
```

```
h=0.10000, numerik limit=2.30000
h=0.01000, numerik limit=2.03000
h=0.00100, numerik limit=2.00300
h=0.00010, numerik limit=2.00030
h=0.00001, numerik limit=2.00003
```

Kendimizi türevler için birkaç eşdeğer gösterimle tanıştıralım. $y = f(x)$ verildiğinde, x ve y sırasıyla f işlevinin bağımsız ve bağımlı değişkenleridir. Aşağıdaki ifadeler eşdeğerdir:

$$f'(x) = y' = \frac{dy}{dx} = \frac{df}{dx} = \frac{d}{dx}f(x) = Df(x) = D_x f(x), \tag{2.4.2}$$

burada $\frac{d}{dx}$ ve D sembolleri *türev alma* işlevini gösteren *türev alma* operatörleridir. Yaygın işlevlerin türevini alma için aşağıdaki kuralları kullanabiliriz:

- $DC = 0$ (C bir sabit),
- $Dx^n = nx^{n-1}$ (*üs kuralı*, n bir gerçel sayı),
- $De^x = e^x$,
- $D\ln(x) = 1/x$.

Yukarıdaki yaygın işlevler gibi birkaç basit işlevden oluşan bir işlevin türevini alırken için aşağıdaki kurallar bizim için kullanışlı olabilir. f ve g işlevlerinin ikisinin de türevlenebilir ve C'nin sabit olduğunu varsayalım, elimizde *sabit çarpım kuralı*,

$$\frac{d}{dx}[Cf(x)] = C\frac{d}{dx}f(x), \tag{2.4.3}$$

toplam kuralı

$$\frac{d}{dx}[f(x) + g(x)] = \frac{d}{dx}f(x) + \frac{d}{dx}g(x), \tag{2.4.4}$$

çarpım kuralı

$$\frac{d}{dx}[f(x)g(x)] = f(x)\frac{d}{dx}[g(x)] + g(x)\frac{d}{dx}[f(x)], \tag{2.4.5}$$

ve *bölme kuralı* vardır.

$$\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x)\frac{d}{dx}[f(x)] - f(x)\frac{d}{dx}[g(x)]}{[g(x)]^2}. \tag{2.4.6}$$

Şimdi $u' = f'(x) = 3\frac{d}{dx}x^2 - 4\frac{d}{dx}x = 6x - 4$'ı bulmak için yukarıdaki kurallardan birkaçını uygulayabiliriz. Bu nedenle, $x = 1$ atadığımız da, $u' = 2$ değerine sahibiz: Sayısal sonucun 2'ye yaklaştığı, bu bölümdeki önceki denememiz tarafından desteklenmektedir. Bu türev aynı zamanda $u = f(x)$ eğrisine $x = 1$'deki teğet doğrusunun eğimidir.

Türevlerin bu tür yorumunu görselleştirmek için Python'da popüler bir çizim kütüphanesi olan `matplotlib`'i kullanacağız. `matplotlib` tarafından üretilen şekillerin özelliklerini yapılandırmak

için birkaç işlev tanımlamamız gerekir. Aşağıdaki `use_svg_display` işlevi, daha keskin görüntülü svg şekilleri çıktısı almak için `matplotlib` paketini özelleştirir. `#@save` yorumunun, aşağıdaki işlev, sınıf veya ifadelerin `d2l` paketine kaydedildiği ve böylece daha sonra yeniden tanımlanmadan doğrudan çağrılabilecekleri (örneğin, `d2l.use_svg_display()`) özel bir terim olduğuna dikkat edin.

```
def use_svg_display():  #@save
    """Jupyter içinde şekli göstermek için svg formatı kullan"""
    backend_inline.set_matplotlib_formats('svg')
```

Şekil boyutlarını belirtmek için `set_figsize` fonksiyonunu tanımlarız. Burada doğrudan `d2l.plt`'yi kullandığımıza dikkat edin, çünkü içe aktarma komutu, `from matplotlib import pyplot as plt`, önsöz bölümündeki `d2l` paketine kaydedilmek üzere işaretlenmişti.

```
def set_figsize(figsize=(3.5, 2.5)):  #@save
    """Şekil ebatını matplotlib için ayarla"""
    use_svg_display()
    d2l.plt.rcParams['figure.figsize'] = figsize
```

Aşağıdaki `set_axes` işlevi, `matplotlib` tarafından üretilen şekillerin eksenlerinin özelliklerini ayarlar.

```
#@save
def set_axes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend):
    """Eksenleri matplotlib için ayarla."""
    axes.set_xlabel(xlabel)
    axes.set_ylabel(ylabel)
    axes.set_xscale(xscale)
    axes.set_yscale(yscale)
    axes.set_xlim(xlim)
    axes.set_ylim(ylim)
    if legend:
        axes.legend(legend)
    axes.grid()
```

Şekil biçimlendirmeleri için bu üç işlevle, kitap boyunca birçok eğriyi görselleştirmemiz gerekeceğinden, çoklu eğrileri kısa ve öz olarak çizmek için `plot` işlevini tanımlıyoruz.

```
#@save
def plot(X, Y=None, xlabel=None, ylabel=None, legend=None, xlim=None,
         ylim=None, xscale='linear', yscale='linear',
         fmts=('-', 'm--', 'g-.', 'r:'), figsize=(3.5, 2.5), axes=None):
    """Veri noktalarını çiz."""
    if legend is None:
        legend = []

    set_figsize(figsize)
    axes = axes if axes else d2l.plt.gca()

    # `X` (tensor veya liste) 1 eksenli ise True değeri döndür
    def has_one_axis(X):
        return (hasattr(X, "ndim") and X.ndim == 1 or isinstance(X, list)
                and not hasattr(X[0], "__len__"))
```

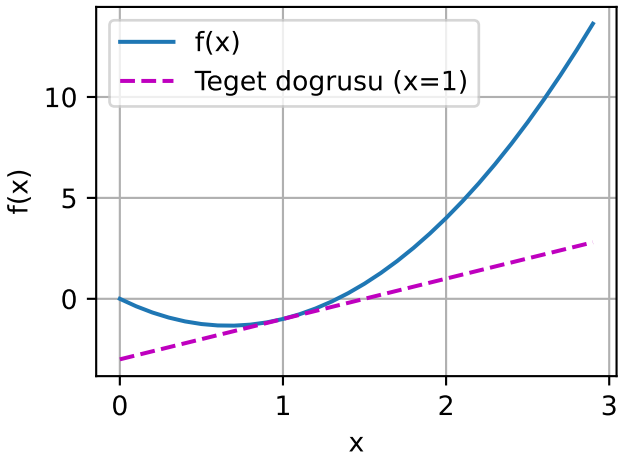
(continues on next page)

(continued from previous page)

```
    if has_one_axis(X):
        X = [X]
    if Y is None:
        X, Y = [[]] * len(X), X
    elif has_one_axis(Y):
        Y = [Y]
    if len(X) != len(Y):
        X = X * len(Y)
    axes.cla()
    for x, y, fmt in zip(X, Y, fmts):
        if len(x):
            axes.plot(x, y, fmt)
        else:
            axes.plot(y, fmt)
    set_axes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend)
```

Şimdi $u = f(x)$ fonksiyonunu ve $y = 2x - 3$ teğet doğrusunu $x = 1$'de çizebiliriz, burada 2 katsayısı teğet doğrusunun eğimidir.

```
x = np.arange(0, 3, 0.1)
plot(x, [f(x), 2 * x - 3], 'x', 'f(x)', legend=['f(x)', 'Teget dogrusu (x=1)'])
```



2.4.2 Kısmi Türevler

Şimdiye kadar sadece tek değişkenli fonksiyonların türevinin alınması ile uğraştık. Derin öğrenmede, işlevler genellikle *birçok* değişkene bağlıdır. Bu nedenle, türev alma fikirlerini bu *çok değişkenli* fonksiyonlara genişletmemiz gerekiyor.

$y = f(x_1, x_2, \ldots, x_n)$, n değişkenli bir fonksiyon olsun. y'nin $i.$ parametresi x_i'ye göre *kısmi türevi* şöyledir:

$$\frac{\partial y}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, \ldots, x_{i-1}, x_i + h, x_{i+1}, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{h}. \tag{2.4.7}$$

$\frac{\partial y}{\partial x_i}$'ı hesaplarken, $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$'ı sabitler olarak kabul eder ve y'nin x_i'ye göre türevini

hesaplayabiliriz. Kısmi türevlerin gösterimi için aşağıdakiler eşdeğerdir:

$$\frac{\partial y}{\partial x_i} = \frac{\partial f}{\partial x_i} = f_{x_i} = f_i = D_i f = D_{x_i} f. \tag{2.4.8}$$

2.4.3 Gradyanlar (Eğimler)

Fonksiyonun *gradyan* vektörünü elde etmek için çok değişkenli bir fonksiyonun tüm değişkenlerine göre kısmi türevlerini art arda bitiştirebiliriz. $f : \mathbb{R}^n \to \mathbb{R}$ işlevinin girdisinin n boyutlu bir vektör, $\mathbf{x} = [x_1, x_2, \ldots, x_n]^\top$ olduğunu varsayalım ve çıktı bir skalerdir. $\mathbf{x}$'e göre $f(\mathbf{x})$ fonksiyonunun gradyanı, n tane kısmi türevli bir vektördür:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_n}\right]^\top, \tag{2.4.9}$$

belirsizlik olmadığında, $\nabla_{\mathbf{x}} f(\mathbf{x})$ genellikle $\nabla f(\mathbf{x})$ ile değiştirilir.

$\mathbf{x}$ bir n boyutlu vektör olsun, aşağıdaki kurallar genellikle çok değişkenli fonksiyonların türevini alırken kullanılır:

- Her $\mathbf{A} \in \mathbb{R}^{m \times n}$ için, $\nabla_{\mathbf{x}} \mathbf{A}\mathbf{x} = \mathbf{A}^\top$,
- Her $\mathbf{A} \in \mathbb{R}^{n \times m}$ için, $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A} = \mathbf{A}$,
- Her $\mathbf{A} \in \mathbb{R}^{n \times n}$ için, $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A}\mathbf{x} = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$,
- $\nabla_{\mathbf{x}} \|\mathbf{x}\|^2 = \nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{x} = 2\mathbf{x}$.

Benzer şekilde, herhangi bir $\mathbf{X}$ matrisi için, $\nabla_{\mathbf{X}} \|\mathbf{X}\|_F^2 = 2\mathbf{X}$ olur. Daha sonra göreceğimiz gibi, gradyanlar derin öğrenmede optimizasyon algoritmaları tasarlamak için kullanışlıdır.

2.4.4 Zincir kuralı

Bununla birlikte, bu tür gradyanları bulmak zor olabilir. Bunun nedeni, derin öğrenmedeki çok değişkenli işlevlerin genellikle *bileşik* olmasıdır, bu nedenle bu işlevlerin türevleri için yukarıda belirtilen kuralların hiçbirini uygulamayabiliriz. Neyse ki, *zincir kuralı* bileşik işlevlerin türevlerini almamızı sağlar.

Önce tek değişkenli fonksiyonları ele alalım. $y = f(u)$ ve $u = g(x)$ işlevlerinin her ikisinin de türevlenebilir olduğunu varsayalım, bu durumda zincir kuralı şunu belirtir:

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}. \tag{2.4.10}$$

Şimdi dikkatimizi, fonksiyonların keyfi sayıda değişkene sahip olduğu daha genel bir senaryoya çevirelim. y türevlenebilir fonksiyonunun $u_1, u_2, \ldots, u_m$ değişkenlerine sahip olduğunu varsayalım, burada her türevlenebilir fonksiyon u_i, $x_1, x_2, \ldots, x_n$ değişkenlerine sahiptir. y değerinin $x_1, x_2, \ldots, x_n$'nin bir işlevi olduğuna dikkat edin. Sonra bütün $i = 1, 2, \ldots, n$ için, zincir kuralı şunu gösterir:

$$\frac{dy}{dx_i} = \frac{dy}{du_1}\frac{du_1}{dx_i} + \frac{dy}{du_2}\frac{du_2}{dx_i} + \cdots + \frac{dy}{du_m}\frac{du_m}{dx_i} \tag{2.4.11}$$

2.4.5 Özet

- Diferansiyel kalkülüs ve integral kalkülüs, ilki derin öğrenmede her yerde bulunan optimizasyon problemlerine uygulanabilen iki analiz dalıdır.
- Bir türev, bir fonksiyonun değişkenine göre anlık değişim hızı olarak yorumlanabilir. Aynı zamanda fonksiyonun eğrisine teğet doğrusunun eğimidir.
- Gradyan, bileşenleri çok değişkenli bir fonksiyonun tüm değişkenlerine göre kısmi türevleri olan bir vektördür.
- Zincir kuralı, bileşik fonksiyonların türevlerini almamızı sağlar.

2.4.6 Alıştırmalar

1. $y = f(x) = x^3 - \frac{1}{x}$ fonksiyonunu ve $x = 1$ olduğunda teğet doğrusunu çizin.
2. $f(\mathbf{x}) = 3x_1^2 + 5e^{x_2}$ fonksiyonunun gradyanını bulun.
3. $f(\mathbf{x}) = \|\mathbf{x}\|_2$ fonksiyonunun gradyanı nedir?
4. $u = f(x, y, z)$ ve $x = x(a, b)$, $y = y(a, b)$ ve $z = z(a, b)$ olduğu durum için zincir kuralını yazabilir misiniz?

Tartışmalar[42]

2.5 Otomatik Türev Alma

Section 2.4 içinde açıkladığımız gibi, türev alma neredeyse tüm derin öğrenme optimizasyon algoritmalarında çok önemli bir adımdır. Bu türevleri almak için gerekli hesaplamalar basittir ve sadece biraz temel kalkülüs gerektirirken, karmaşık modeller için güncellemeleri elle yapmak bir sancılı olabilir (ve genellikle hataya açık olabilir).

Derin öğrenme çerçeveleri, türevleri otomatik olarak hesaplayarak, yani *otomatik türev alma* yoluyla bu çalışmayı hızlandırır. Uygulamada, tasarladığımız modele dayalı olarak sistem, çıktıyı üretmek için hangi verinin hangi işlemlerle birleştirildiğini izleyen bir *hesaplama grafiği (çizgesi)* oluşturur. Otomatik türev alma, sistemin daha sonra gradyanları geri yaymasını (backpropagation) sağlar. Burada *geri yayma*, her bir parametreye göre kısmi türevleri doldurarak, hesaplama grafiğini izlemek anlamına gelir.

2.5.1 Basit Bir Örnek

Bir oyuncak örnek olarak, $y = 2\mathbf{x}^\top\mathbf{x}$ fonksiyonunun $\mathbf{x}$ sütun vektörüne göre türevini almakla ilgilendiğimizi söyleyelim. Başlamak için, `x` değişkenini oluşturalım ve ona bir başlangıç değeri atayalım.

```
import torch

x = torch.arange(4.0)
x
```

[42] https://discuss.d2l.ai/t/33

```
tensor([0., 1., 2., 3.])
```

y'nin $\mathbf{x}$'e göre gradyanını hesaplamadan önce, onu saklayabileceğimiz bir yere ihtiyacımız var. Bir parametreye göre her türev aldığımızda yeni bir bellek ayırmamamız önemlidir çünkü aynı parametreleri binlerce kez veya milyonlarca kez güncelleyeceğiz ve belleğimiz hızla tükenebilir. Skaler değerli bir fonksiyonun bir $\mathbf{x}$ vektörüne göre gradyanının kendisinin vektör değerli olduğuna ve $\mathbf{x}$ ile aynı şekle sahip olduğuna dikkat edin.

```
x.requires_grad_(True)  # `x = torch.arange(4.0, requires_grad=True)` ile aynıdır
x.grad  # None varsayılan değerdir
```

Şimdi y'yi hesaplayalım.

```
y = 2 * torch.dot(x, x)
y
```

```
tensor(28., grad_fn=<MulBackward0>)
```

`x`, 4 uzunluklu bir vektör olduğu için, `x` ve `x`'in iç çarpımı gerçekleştirilir ve `y`'ye atadığımız skaler çıktı elde edilir. Daha sonra, geri yayma için işlevi çağırarak ve gradyanı yazdırarak `x`'in her bir bileşenine göre `y` gradyanını otomatik olarak hesaplayabiliriz.

```
y.backward()
x.grad
```

```
tensor([ 0.,  4.,  8., 12.])
```

$y = 2\mathbf{x}^\top\mathbf{x}$ fonksiyonunun $\mathbf{x}$'e göre gradyanı $4\mathbf{x}$ olmalıdır. İstenilen gradyanın doğru hesaplandığını hızlıca doğrulayalım.

```
x.grad == 4 * x
```

```
tensor([True, True, True, True])
```

Şimdi başka bir `x` fonksiyonunu hesaplayalım.

```
# PyTorch, gradyanı varsayılan olarak biriktirir, önceki değerleri temizlememiz gerekir.
x.grad.zero_()
y = x.sum()
y.backward()
x.grad
```

```
tensor([1., 1., 1., 1.])
```

2.5.2 Skaler Olmayan Değişkenler için Geriye Dönüş

Teknik olarak, `y` skaler olmadığında, `y` vektörünün `x` vektörüne göre türevinin en doğal yorumu bir matristir. Daha yüksek kademeli ve daha yüksek boyutlu `y` ve `x` için, türevin sonucu yüksek kademeli bir tensör olabilir.

Bununla birlikte, bu daha egzotik nesneler gelişmiş makine öğrenmesinde (derin öğrenmedekiler dahil) ortaya çıkarken, daha sıklıkla bir vektör üzerinde geriye doğru dönük çağırdığımızda, bir *grup* eğitim örneğinde kayıp fonksiyonlarının her bir bileşeni için türevlerini hesaplamaya çalışıyoruz. Burada amacımız, türev matrisini hesaplamak değil, gruptaki her örnek için ayrı ayrı hesaplanan kısmi türevlerin toplamını hesaplamaktır.

```
# Skaler olmayan üzerinde `geridönüş` çağırmak, farklılaştırılmış işlevin kendine göre
# `self` gradyanını belirten bir `gradyan` argümanının iletilmesini gerektirir. Bizim␣
↪durumumuzda,
# sadece kısmi türevleri toplamak istiyoruz, bu nedenle birlerden oluşan gradyanı iletmek␣
↪uygundur.
x.grad.zero_()
y = x * x
# y.backward(torch.ones(len(x))) aşağısı ile aynıdır
y.sum().backward()
x.grad
```

```
tensor([0., 2., 4., 6.])
```

2.5.3 Hesaplamanın Ayrılması

Bazen, bazı hesaplamaları kaydedilen hesaplama grafiğinin dışına taşımak isteriz. Örneğin, `y`'nin `x`'in bir fonksiyonu olarak hesaplandığını ve daha sonra `z`'nin hem `y`'nin hem de `x`'in bir fonksiyonu olarak hesaplandığını varsayalım. Şimdi, `z`'nin gradyanını `x`'e göre hesaplamak istediğimizi, ancak nedense `y`'yi bir sabit olarak kabul etmek istediğimizi ve `x`'in `y` hesaplandıktan sonra oynadığı rolü hesaba kattığımızı hayal edin.

Burada, `y` ile aynı değere sahip yeni bir `u` değişkeni döndürmek için `y`'yi ayırabiliriz, ancak bu `y`'nin hesaplama grafiğinde nasıl hesaplandığına dair tüm bilgileri yok sayar. Başka bir deyişle, gradyan `u`'dan `x`'e geriye doğru akmayacaktır. Bu nedenle, aşağıdaki geri yayma işlevi, `z = x * x * x`'in `x`'e göre kısmi türevi hesaplamak yerine, `z = u * x`'in `x`'e göre kısmi türevini `u` sabitmiş gibi davranarak hesaplar.

```
x.grad.zero_()
y = x * x
u = y.detach()
z = u * x

z.sum().backward()
x.grad == u
```

```
tensor([True, True, True, True])
```

`y` hesaplaması kaydedildiğinden, `y = x * x`'nin `x`'e göre türevi olan `2 * x`'i elde etmek için `y` üzerinden geri yaymayı başlatabiliriz.

```
x.grad.zero_()
y.sum().backward()
x.grad == 2 * x
```

```
tensor([True, True, True, True])
```

2.5.4 Python Kontrol Akışının Gradyanını Hesaplama

Otomatik türev almayı kullanmanın bir yararı, bir Python kontrol akışı labirentinden (örneğin, koşullu ifadeler, döngüler ve rastgele fonksiyon çağrıları) geçen bir fonksiyondan hesaplama çizgesi oluşturup, elde edilen değişkenin gradyanını yine de hesaplayabilmemizdir. Aşağıdaki kod parçacığında, `while` döngüsünün yineleme sayısının ve `if` ifadesinin değerlendirmesinin `a` girdisinin değerine bağlı olduğuna dikkat edin.

```
def f(a):
    b = a * 2
    while b.norm() < 1000:
        b = b * 2
    if b.sum() > 0:
        c = b
    else:
        c = 100 * b
    return c
```

Hadi gradyanı hesaplayalım.

```
a = torch.randn(size=(), requires_grad=True)
d = f(a)
d.backward()
```

Şimdi yukarıda tanımlanan `f` fonksiyonunu analiz edebiliriz. Fonksiyonun `a` girdisinde parçalı doğrusal olduğuna dikkat edin. Diğer bir deyişle, herhangi bir `a` için, `k` değerinin `a` girdisine bağlı olduğu `f(a) = k * a` olacak şekilde sabit bir `k` vardır. Sonuç olarak `d / a`, gradyanın doğru olduğunu doğrulamamıza izin verir.

```
a.grad == d / a
```

```
tensor(False)
```

2.5.5 Özet

- Derin öğrenme çerçeveleri türevlerin hesaplanmasını otomatikleştirebilir. Kullanmak için, önce kısmi türevleri bulmak istediğimiz değişkenlere gradyanlar ekleriz. Daha sonra hedef değerimizin hesaplamasını kaydeder, geri yayma için işlevi uygularız ve elde edilen gradyanlara erişiriz.

2.5.6 Alıştırmalar

1. İkinci türevi hesaplamak neden birinci türeve göre çok daha pahalıdır?
2. Geri yayma için işlevi çalıştırdıktan sonra, hemen tekrar çalıştırın ve ne olduğunu görün.
3. d'nin a'ya göre türevini hesapladığımız kontrol akışı örneğinde, `a` değişkenini rastgele bir vektör veya matris olarak değiştirirsek ne olur. Bu noktada, `f(a)` hesaplamasının sonucu artık skaler değildir. Sonuca ne olur? Bunu nasıl analiz ederiz?
4. Kontrol akışının gradyanını bulmak için yeniden bir örnek tasarlayın. Sonucu çalıştırın ve analiz edin.
5. $f(x) = \sin(x)$ olsun. $f(x)$ ve $\frac{df(x)}{dx}$ grafiklerini çizin, buradaki ikinci terim $f'(x) = \cos(x)$ kullanılmadan hesaplansın.

Tartışmalar[43]

2.6 Olasılık

Bir şekilde, makine öğrenmesi tamamen tahminlerde bulunmakla ilgilidir. Klinik geçmişi göz önüne alındığında, bir hastanın önümüzdeki yıl kalp krizi geçirme *olasılığını* tahmin etmek isteyebiliriz. Anormallik tespitinde, bir uçağın jet motorundan bir dizi okumanın uçak normal çalışıyor olsaydı ne kadar *muhtemel* olacağını değerlendirmek isteyebiliriz. Pekiştirmeli öğrenmede, bir etmenin bir ortamda akıllıca hareket etmesini istiyoruz. Bu, mevcut eylemlerin her birinin altında yüksek bir ödül alma olasılığını düşünmemiz gerektiği anlamına gelir. Ayrıca tavsiye sistemleri oluşturduğumuzda, olasılık hakkında da düşünmemiz gerekir. Örneğin, *varsayımsal olarak* büyük bir çevrimiçi kitapçı için çalıştığımızı söyleyelim. Belirli bir kullanıcının belirli bir kitabı satın alma olasılığını tahmin etmek isteyebiliriz. Bunun için olasılık dilini kullanmamız gerekiyor. Birçok ders, anadal, tez, kariyer ve hatta bölüm olasılığa ayrılmıştır. Doğal olarak, bu bölümdeki amacımız konunun tamamını öğretmek değildir. Bunun yerine sizi ayaklarınızın üzerine kaldırmayı, ilk derin öğrenme modellerinizi oluşturmaya başlayabileceğiniz kadarını öğretmeyi ve dilerseniz konuyu kendi başınıza keşfetmeye başlayabilmeniz için bir tutam bilgi vermeyi umuyoruz.

Daha önceki bölümlerde, tam olarak ne olduklarını açıklamadan veya somut bir örnek vermeden olasılıkları zaten çağırmıştık. Şimdi ilk vakayı ele alarak daha ciddileşelim: Fotoğraflardan kedi ve köpekleri ayırmak. Bu basit gelebilir ama aslında zorlu bir görevdir. Başlangıç olarak, sorunun zorluğu imgenin çözünürlüğüne bağlı olabilir.

[43] https://discuss.d2l.ai/t/35

Fig. 2.6.1: Farklı çözünürlükteki imgeler (10×10, 20×20, 40×40, 80×80, and 160×160 piksel).

Gösterildiği gibi Fig. 2.6.1, insanlar için kedileri ve köpekleri 160×16 piksel çözünürlükte tanımak kolayken, 40×40 pikselde zorlayıcı ve 10×10 pikselde imkansıza yakın hale geliyor. Başka bir deyişle, kedi ve köpekleri büyük bir mesafeden (ve dolayısıyla düşük çözünürlükten) ayırma yeteneğimiz bilgisiz (cahilce) tahminlere yaklaşabilir. Olasılık, bize kesinlik seviyemiz hakkında resmi (kurallı) bir mantık yürütme yöntemi verir. İmgenin bir kediyi gösterdiğinden tamamen eminsek, karşılık gelen y etiketinin "kedi" olma *olasılığının*, $P(y =$ "kedi"$)$, 1'e eşit olduğunu söyleriz. $y =$ "kedi" veya $y =$ "köpek" olduğunu önerecek hiçbir kanıtımız yoksa, iki olasılığın eşit derecede *muhtemelen* olduğunu $P(y =$ "kedi"$) = P(y =$ "köpek"$) = 0.5$ diye ifade ederek söyleyebiliriz. Makul derecede emin olsaydık, ancak imgenin bir kediyi gösterdiğinden kesin emin olamasaydık, $0.5 < P(y =$ "kedi"$) < 1$ bir olasılık atayabilirdik.

Şimdi ikinci bir durumu düşünün: Bazı hava durumu izleme verilerini göz önüne alarak yarın Taipei'de yağmur yağma olasılığını tahmin etmek istiyoruz. Yaz mevsimindeyse, yağmur 0.5 olasılıkla gelebilir.

Her iki durumda da, bir miktar ilgi değerimiz var. Her iki durumda da sonuç hakkında emin değiliz. Ancak iki durum arasında temel bir fark var. Bu ilk durumda, imge aslında ya bir köpektir ya da bir kedidir ve hangisi olduğunu bilmiyoruz. İkinci durumda, eğer bu tür şeylere inanıyorsanız (ve çoğu fizikçi bunu yapıyor), sonuç aslında rastgele bir olay olabilir. Dolayısıyla olasılık, kesinlik seviyemiz hakkında akıl yürütmek için esnek bir dildir ve geniş bir bağlam kümesinde etkili bir şekilde uygulanabilir.

2.6.1 Temel Olasılık Kuramı

Bir zar attığımızı ve başka bir rakam yerine 1 rakamını görme şansının ne olduğunu bilmek istediğimizi düşünelim. Eğer zar adilse, altı sonucun tümü $\{1, \ldots, 6\}$ eşit derecede meydana gelir ve bu nedenle altı durumdan birinde 1 görürüz. Resmi olarak 1'in $\frac{1}{6}$ olasılıkla oluştuğunu belirtiyoruz.

Bir fabrikadan aldığımız gerçek bir zar için bu oranları bilmeyebiliriz ve hileli olup olmadığını kontrol etmemiz gerekir. Zarı araştırmanın tek yolu, onu birçok kez atmak ve sonuçları kaydetmektir. Zarın her atımında, $\{1, \ldots, 6\}$'den bir değer gözlemleyeceğiz. Bu sonuçlar göz önüne

alındığında, her bir sonucu gözlemleme olasılığını araştırmak istiyoruz.

Doğal bir yaklaşım her değer için, o değerin bireysel sayımını almak ve bunu toplam atış sayısına bölmektir. Bu bize belirli bir *olayın* olasılığının *tahminini* verir. *Büyük sayılar yasası* bize, atışların sayısı arttıkça bu tahminin gerçek temel olasılığa gittikçe yaklaşacağını söyler. Burada neler olup bittiğinin ayrıntılarına girmeden önce bunu deneyelim.

Başlamak için gerekli paketleri içeri aktaralım.

```
%matplotlib inline
import torch
from torch.distributions import multinomial
from d2l import torch as d2l
```

Sonra, zarı atabilmeyi isteyeceğiz. İstatistikte bu olasılık dağılımlarından örnek alma sürecini *örnekleme* olarak adlandırıyoruz. Olasılıkları bir dizi ayrık seçime atayan dağılıma *katlıterimli (multinomial) dağılım* denir. Daha sonra *dağılımın* daha resmi bir tanımını vereceğiz, ancak üst seviyede, bunu sadece olaylara olasılıkların atanması olarak düşünün.

Tek bir örneklem çekmek için, basitçe bir olasılık vektörü aktarırız. Çıktı aynı uzunlukta başka bir vektördür: i indisindeki değeri, örnekleme sonucunun i'ye karşılık gelme sayısıdır.

```
fair_probs = torch.ones([6]) / 6
multinomial.Multinomial(1, fair_probs).sample()
```

```
tensor([0., 0., 0., 1., 0., 0.])
```

Örnekleyiciyi birkaç kez çalıştırırsanız, her seferinde rastgele değerler çıkardığını göreceksiniz. Bir zarın adilliğini tahmin ederken olduğu gibi, genellikle aynı dağılımdan birçok örneklem oluşturmak isteriz. Bunu bir Python `for` döngüsüyle yapmak dayanılmaz derecede yavaş olacaktır, bu nedenle kullandığımız işlev, aynı anda birden fazla örneklem çekmeyi destekler ve arzu edebileceğimiz herhangi bir şekile sahip bağımsız örneklemler dizisi döndürür.

```
multinomial.Multinomial(10, fair_probs).sample()
```

```
tensor([2., 0., 3., 2., 0., 3.])
```

Artık zar atışlarını nasıl örnekleyeceğimizi bildiğimize göre, 1000 atış benzetimi yaparız. Daha sonra, 1000 atışın her birinden sonra, her bir sayının kaç kez atıldığını inceleyebilir ve sayabiliriz. Özellikle belirtirsek, göreceli frekansı gerçek olasılığın tahmini olarak hesaplarız.

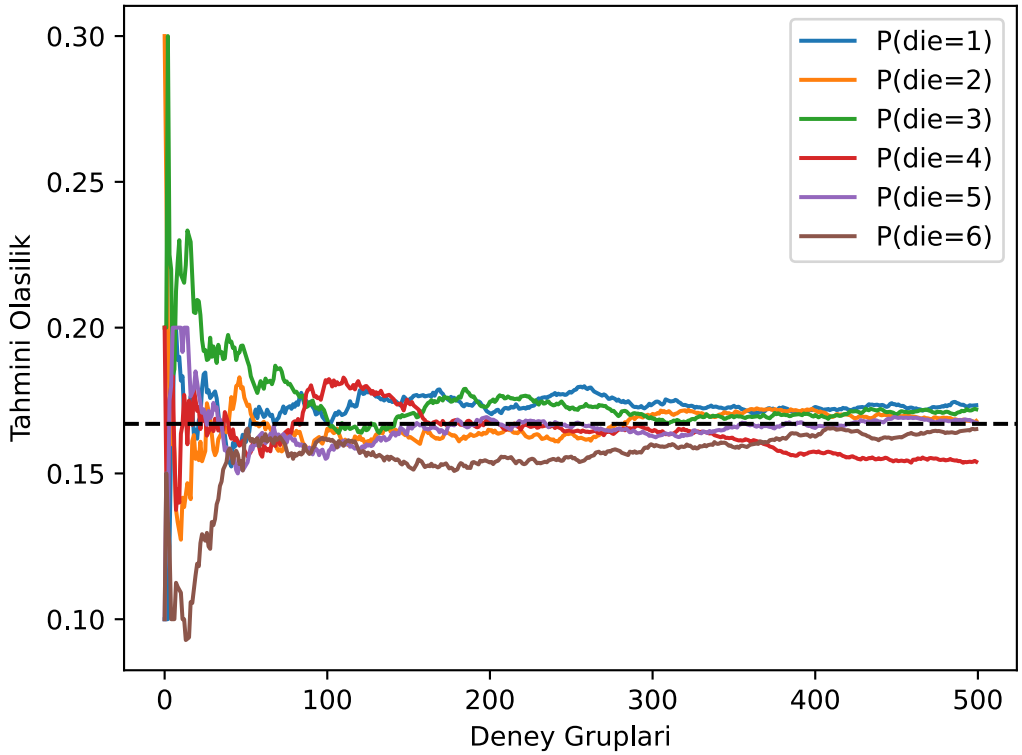
```
# Bölme için sonuçları 32-bitlik virgüllü kazan sayı olarak depolarız
counts = multinomial.Multinomial(1000, fair_probs).sample()
counts / 1000  # Relative frequency as the estimate
```

```
tensor([0.1590, 0.1660, 0.1910, 0.1610, 0.1640, 0.1590])
```

Verileri adil bir zardan oluşturduğumuz için, her sonucun gerçek olasılığının $\frac{1}{6}$ olduğunu, yani kabaca 0.167 olduğunu biliyoruz, bu nedenle yukarıdaki çıktı tahminleri iyi görünüyor.

Ayrıca bu olasılıkların zaman içinde gerçek olasılığa doğru nasıl yakınsadığını da görselleştirebiliyoruz. Her grubun 10 örnek çektiği 500 adet deney yapalım.

```
counts = multinomial.Multinomial(10, fair_probs).sample((500,))
cum_counts = counts.cumsum(dim=0)
estimates = cum_counts / cum_counts.sum(dim=1, keepdims=True)

d2l.set_figsize((6, 4.5))
for i in range(6):
    d2l.plt.plot(estimates[:, i].numpy(),
                 label=("P(die=" + str(i + 1) + ")"))
d2l.plt.axhline(y=0.167, color='black', linestyle='dashed')
d2l.plt.gca().set_xlabel('Deney Gruplari')
d2l.plt.gca().set_ylabel('Tahmini Olasilik')
d2l.plt.legend();
```



Her katı eğri, zarın altı değerinden birine karşılık gelir ve her deney grubundan sonra değerlendirildiğinde zarın bu değerleri göstermesinde tahmin ettiğimiz olasılığı verir. Kesikli siyah çizgi, gerçek temel olasılığı verir. Daha fazla deney yaparak daha fazla veri elde ettikçe, 6 katı eğri gerçek olasılığa doğru yaklaşıyor.

Olasılık Teorisinin Aksiyomları

Bir zarın atışları ile uğraşırken, $\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$ kümesine *örnek uzay* veya *sonuç uzayı* diyoruz, burada her eleman bir *sonuçtur*. Bir *olay*, belirli bir örnek uzaydan alınan bir dizi sonuçtur. Örneğin, "5 görmek" ($\{5\}$) ve "tek sayı görmek" ($\{1, 3, 5\}$) zar atmanın geçerli olaylarıdır. Rastgele bir denemenin sonucu $\mathcal{A}$ olayındaysa, $\mathcal{A}$ olayının gerçekleştiğine dikkat edin. Yani, bir zar atıldıktan sonra 3 nokta üstte gelirse, $3 \in \{1, 3, 5\}$ olduğundan, "tek bir sayı görme" olayı gerçekleşti diyebiliriz.

Biçimsel olarak, *olasılık*, bir kümeyi gerçek bir değere eşleyen bir işlev olarak düşünülebilir. $P(\mathcal{A})$ olarak gösterilen, verilen $\mathcal{S}$ örnek uzayında bir $\mathcal{A}$ olayının olasılığı aşağıdaki özellikleri karşılar:

- Herhangi bir $\mathcal{A}$ olayı için, olasılığı asla negatif değildir, yani, $P(\mathcal{A}) \geq 0$;
- Tüm örnek alanın olasılığı 1'dir, yani $P(\mathcal{S}) = 1$;
- Birbirini dışlayan sayılabilir herhangi bir olay dizisi için ($\mathcal{A}_i \cap \mathcal{A}_j = \emptyset$ bütün $i \neq j$) için, herhangi bir şeyin olma olasılığı kendi olasılıklarının toplamına eşittir, yani, $P(\bigcup_{i=1}^{\infty} \mathcal{A}_i) = \sum_{i=1}^{\infty} P(\mathcal{A}_i)$.

Bunlar aynı zamanda 1933'te Kolmogorov tarafından önerilen olasılık teorisinin aksiyomlarıdır. Bu aksiyom sistemi sayesinde, rastlantısallıkla ilgili herhangi bir felsefi tartışmayı önleyebiliriz; bunun yerine matematiksel bir dille titiz bir şekilde akıl yürütebiliriz. Örneğin, $\mathcal{A}_1$ olayının tüm örnek uzay olmasına ve $\mathcal{A}_i = \emptyset$'a bütün $i > 1$ için izin vererek, $P(\emptyset) = 0$, yani imkansız bir olayın olasılığı 0'dır.

Rastgele Değişkenler

Bir zar atma rastgele deneyimizde, *rastgele değişken* kavramını tanıttık. Rastgele bir değişken hemen hemen herhangi bir miktar olabilir ve deterministik (belirlenimci) değildir. Değişken rastgele bir deneyde bir dizi olasılık arasından bir değer alabilir. Değeri bir zar atmanın $\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$ örnek uzayında olan X rastgele değişkenini düşünün. "Bir 5 görme" olayını $\{X = 5\}$ veya $X = 5$ ve olasılığını $P(\{X = 5\})$ veya $P(X = 5)$ diye belirtiriz. $P(X = a)$ ile, X rastgele değişkeni ile X'in alabileceği değerler (örneğin, a) arasında bir ayrım yaparız. Bununla birlikte, bu tür bilgiçlik, hantal bir gösterimle sonuçlanır. Kısa bir gösterim için, bir yandan, $P(X)$'i, X rasgele değişkeni üzerindeki *dağılım* olarak gösterebiliriz: Dağılım bize X'in herhangi bir değeri alma olasılığını söyler. Öte yandan, rastgele bir değişkenin a değerini alma olasılığını belirtmek için $P(a)$ yazabiliriz. Olasılık teorisindeki bir olay, örnek uzaydan bir küme sonuç olduğu için, rastgele bir değişkenin alması için bir dizi değer belirleyebiliriz. Örneğin, $P(1 \leq X \leq 3)$, $\{1 \leq X \leq 3\}$ olayının olasılığını belirtir, yani $\{X = 1, 2, \text{veya} 3\}$ anlamına gelir. Aynı şekilde, $P(1 \leq X \leq 3)$, X rasgele değişkeninin $\{1, 2, 3\}$'ten bir değer alabilme olasılığını temsil eder.

Bir zarın yüzleri gibi *kesikli* rastgele değişkenler ile bir kişinin ağırlığı ve boyu gibi *sürekli* olanlar arasında ince bir fark olduğunu unutmayın. İki kişinin tam olarak aynı boyda olup olmadığını sormanın pek bir anlamı yok. Yeterince hassas ölçümler alırsak, gezegendeki hiçbir insanın aynı boyda olmadığını göreceksiniz. Aslında, yeterince ince bir ölçüm yaparsak, uyandığınızda ve uyuduğunuzda da boyunuz aynı olmayacaktır. Dolayısıyla, birinin 1.80139278291028719210196740527486202 metre boyunda olma olasılığını sormanın hiçbir amacı yoktur. Dünya insan nüfusu göz önüne alındığında, olasılık neredeyse 0'dır. Bu durumda, birinin boyunun belirli bir aralıkta, örneğin 1.79 ile 1.81 metre arasında olup olmadığını sormak daha mantıklıdır. Bu durumlarda, bir değeri *yoğunluk* olarak görme olasılığımızı ölçüyoruz. Tam olarak 1.80 metrelik boyun olasılığı yoktur, ancak yoğunluğu sıfır değildir. Herhangi iki farklı boy arasındaki aralıkta sıfır olmayan bir olasılığa sahibiz. Bu bölümün geri kalanında, olasılığı ayrık uzayda ele alıyoruz. Sürekli rastgele değişkenler üzerindeki olasılık için, şuraya başvurabilirsiniz Section 18.6.

2.6.2 Çoklu Rastgele Değişkenlerle Başa Çıkma

Çok sık olarak, bir seferde birden fazla rastgele değişkeni dikkate almak isteyeceğiz. Örneğin, hastalıklar ile belirtiler arasındaki ilişkiyi modellemek isteyebiliriz. Bir hastalık ve bir belirti verildiğinde, örneğin "grip" ve "öksürük", bir olasılıkla bir hastada ortaya çıkabilirler veya çıkmayabilirler. Her ikisinin de olasılığının sıfıra yakın olacağını umarken, bu olasılıkları ve bunların birbirleriyle olan ilişkilerini tahmin etmek isteyebiliriz, böylece daha iyi tıbbi bakım sağlamak için çıkarımlarımızı uygulayabiliriz.

Daha karmaşık bir örnek olarak, imgeler milyonlarca piksel, dolayısıyla milyonlarca rastgele değişken içerir. Ve çoğu durumda imgeler, imgedeki nesneleri tanımlayan bir etiketle birlikte gelir. Etiketi rastgele bir değişken olarak da düşünebiliriz. Tüm meta (üst) verileri konum, zaman, diyafram, odak uzaklığı, ISO, odak mesafesi ve kamera türü gibi, rastgele değişkenler olarak bile düşünebiliriz. Bunların hepsi birlikte oluşan rastgele değişkenlerdir. Birden çok rastgele değişkenle uğraştığımızda, ilgilendiğimiz birkaç miktar vardır.

Bileşik Olasılık

İlki, *bileşik olasılık* $P(A = a, B = b)$ olarak adlandırılır. Herhangi a ve b değerleri verildiğinde, bileşik olasılık şu cevabı vermemizi sağlar: $A = a$ ve $B = b$ olaylarının aynı anda olma olasılığı nedir? Tüm a ve b değerleri için, $P(A = a, B = b) \leq P(A = a)$ olduğuna dikkat edin. Durum böyle olmalıdır, çünkü $A = a$ ve $B = b$ olması için $A = a$ olması gerekir *ve* $B = b$ de gerçekleşmelidir (ve bunun tersi de geçerlidir). Bu nedenle, $A = a$ ve $B = b$, tek tek $A = a$ veya $B = b$ değerinden daha büyük olamaz.

Koşullu Olasılık

Bu bizi ilginç bir orana getiriyor: $0 \leq \frac{P(A=a,B=b)}{P(A=a)} \leq 1$. Bu oranı bir *koşullu olasılık* olarak adlandırıyoruz ve bunu $P(B = b \mid A = a)$ ile gösteriyoruz: $A = a$ olması koşuluyla, $B = b$ olasılığıdır.

Bayes Kuramı (Teoremi)

Koşullu olasılıkların tanımını kullanarak, istatistikteki en kullanışlı ve ünlü denklemlerden birini türetebiliriz: *Bayes teoremi*. Aşağıdaki gibidir. Yapısı gereği, $P(A, B) = P(B \mid A)P(A)$ şeklindeki *çarpma kuralına* sahibiz. Simetriye göre, bu aynı zamanda $P(A, B) = P(A \mid B)P(B)$ için de geçerlidir. $P(B) > 0$ olduğunu varsayalım. Koşullu değişkenlerden birini çözerek şunu elde ederiz:

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}. \tag{2.6.1}$$

Burada, $P(A, B)$'nin *bileşik dağılım* ve $P(A \mid B)$'nin *koşullu dağılım* olduğu, daha sıkıştırılmış gösterimi kullandığımıza dikkat edin. Bu tür dağılımlar belirli $A = a, B = b$ değerleri için hesaplanabilir.

Marjinalleştirme

Bayes teoremi, bir şeyi diğerinden çıkarmak istiyorsak, neden ve sonuç mesela, çok kullanışlıdır, ancak bu bölümde daha sonra göreceğimiz gibi, yalnızca özellikleri ters yönde biliyoruz. Bunun işe yaraması için ihtiyacımız olan önemli bir işlem, *marjinalleştirme*dir. $P(A, B)$'den $P(B)$ belirleme işlemidir. B olasılığının, tüm olası A seçeneklerini hesaba katma ve bunların hepsinde bileşik olasılıkları bir araya toplama olduğunu görebiliriz:

$$P(B) = \sum_A P(A, B), \tag{2.6.2}$$

bu aynı zamanda *toplam kuralı* olarak da bilinir. Tümleştirmenin bir sonucu olan olasılık veya dağılım, *marjinal (tümsel) olasılık* veya *marjinal (tümsel) dağılım* olarak adlandırılır.

Bağımsızlık

Kontrol edilmesi gereken diğer bir yararlı özellik, *bağımlılık* ve *bağımsızlık*tır. İki rastgele değişken olan A ve B'nin birbirinden bağımsızlığı, A olayının ortaya çıkmasının, B olayının oluşumu hakkında herhangi bir bilgi vermediği anlamına gelir. Bu durumda $P(B \mid A) = P(B)$'dir. İstatistikçiler bunu genellikle $A \perp B$ olarak ifade ederler. Bayes teoreminden, bunu aynı zamanda $P(A \mid B) = P(A)$ olduğunu da izler. Diğer tüm durumlarda A ve B'ye bağımlı diyoruz. Örneğin, bir zarın iki ardışık atışı bağımsızdır. Aksine, bir ışık anahtarının konumu ve odadaki parlaklık değildir (her zaman kırılmış bir ampulümüz, elektrik kesintisi veya kırık bir anahtarımız olabileceğinden, tam olarak belirlenimci (deterministik) değildirler).

$P(A \mid B) = \frac{P(A,B)}{P(B)} = P(A)$ ve $P(A, B) = P(A)P(B)$ eşit olduğundan, iki rastgele değişken ancak ve ancak bileşik dağılımları kendi bireysel dağılımlarının çarpımına eşit ise bağımsızdır. Benzer şekilde, iki rastgele değişken A ve B başka bir rasgele değişken C verildiğinde, ancak ve ancak $P(A, B \mid C) = P(A \mid C)P(B \mid C)$ ise *koşullu olarak bağımsızdır*. Bu, $A \perp B \mid C$ olarak ifade edilir.

Uygulama

Becerilerimizi test edelim. Bir doktorun bir hastaya HIV testi uyguladığını varsayalım. Bu test oldukça doğrudur ve yalnızca %1 olasılıkla hasta sağlıklı olduğu halde hasta olarak bildirme hatası yapar. Dahası, eğer hastada gerçekten varsa HIV'i asla tespit etmemezlik yapmaz. Teşhisi belirtmek için D_1 (pozitifse 1 ve negatifse 0) ve HIV durumunu belirtmek için H (pozitifse 1 ve negatifse 0) kullanırız. Section 2.6.2 bu tür koşullu olasılıkları listeler.

Koşullu olasılık	$H = 1$	$H = 0$
$P(D_1 = 1 \mid H)$	1	0.01
$P(D_1 = 0 \mid H)$	0	0.99

Table: $P(D_1 \mid H)$ koşullu olasılığı

Koşullu olasılığın, olasılık gibi 1'e toplanması gerektiğinden, sütun toplamlarının hepsinin 1 olduğuna dikkat edin (ancak satır toplamları değildir). Test pozitif çıkarsa hastanın HIV'li olma olasılığını hesaplayalım, yani $P(H = 1 \mid D_1 = 1)$. Açıkçası bu, yanlış alarmların sayısını etkilediği için hastalığın ne kadar yaygın olduğuna bağlı olacaktır. Nüfusun oldukça sağlıklı olduğunu varsayalım, örneğin $P(H = 1) = 0.0015$. Bayes teoremini uygulamak için, marjinalleştirmeyi ve

çarpım kuralını uygulamalıyız.

$$\begin{aligned}&P(D_1 = 1)\\ =&P(D_1 = 1, H = 0) + P(D_1 = 1, H = 1)\\ =&P(D_1 = 1 \mid H = 0)P(H = 0) + P(D_1 = 1 \mid H = 1)P(H = 1)\\ =&0.011485.\end{aligned} \tag{2.6.3}$$

Böylece bunu elde ederiz,

$$\begin{aligned}&P(H = 1 \mid D_1 = 1)\\ =&\frac{P(D_1 = 1 \mid H = 1)P(H = 1)}{P(D_1 = 1)}.\\ =&0.1306\end{aligned} \tag{2.6.4}$$

Diğer bir deyişle, çok doğru bir test kullanmasına rağmen, hastanın gerçekten HIVli olma şansı yalnızca %13.06'dır. Gördüğümüz gibi, olasılık sezgilere ters olabilir.

Böylesine korkunç bir haber alan hasta ne yapmalıdır? Muhtemelen hasta, netlik elde etmek için hekimden başka bir test yapmasını isteyecektir. İkinci testin farklı özellikleri vardır ve şu şekilde gösterildiği gibi birincisi kadar iyi değildir Section 2.6.2.

Koşullu olasılık	$H = 1$	$H = 0$
$P(D_2 = 1 \mid H)$	0.98	0.03
$P(D_2 = 0 \mid H)$	0.02	0.97

Table: $P(D_2 \mid H)$ koşullu olasılığı

Maalesef ikinci test de pozitif çıkıyor. Koşullu bağımsızlığı varsayıp Bayes teoremini çağırarak gerekli olasılıkları bulalım:

$$\begin{aligned}&P(D_1 = 1, D_2 = 1 \mid H = 0)\\ =&P(D_1 = 1 \mid H = 0)P(D_2 = 1 \mid H = 0)\\ =&0.0003,\end{aligned} \tag{2.6.5}$$

$$\begin{aligned}&P(D_1 = 1, D_2 = 1 \mid H = 1)\\ =&P(D_1 = 1 \mid H = 1)P(D_2 = 1 \mid H = 1)\\ =&0.98.\end{aligned} \tag{2.6.6}$$

Şimdi marjinalleştirme ve çarpım kuralını uygulayabiliriz:

$$\begin{aligned}&P(D_1 = 1, D_2 = 1)\\ =&P(D_1 = 1, D_2 = 1, H = 0) + P(D_1 = 1, D_2 = 1, H = 1)\\ =&P(D_1 = 1, D_2 = 1 \mid H = 0)P(H = 0) + P(D_1 = 1, D_2 = 1 \mid H = 1)P(H = 1)\\ =&0.00176955.\end{aligned} \tag{2.6.7}$$

En sonunda, her iki pozitif testin de verildiği hastanın HIV'li olma olasılığı

$$\begin{aligned}&P(H = 1 \mid D_1 = 1, D_2 = 1)\\ =&\frac{P(D_1 = 1, D_2 = 1 \mid H = 1)P(H = 1)}{P(D_1 = 1, D_2 = 1)}\\ =&0.8307.\end{aligned} \tag{2.6.8}$$

Böylece ikinci test, her şeyin yolunda olmadığına dair çok daha yüksek bir güven kazanmamızı sağladı. İkinci test, birincisinden çok daha az doğru olmasına rağmen, tahminimizi önemli ölçüde iyileştirdi.

2.6.3 Beklenti ve Varyans (Değişinti)

Olasılık dağılımlarının temel özelliklerini özetlemek için bazı ölçülere ihtiyacımız var. X rastgele değişkeninin *beklentisi* (veya ortalaması) şu şekilde belirtilir:

$$E[X] = \sum_x xP(X = x). \tag{2.6.9}$$

$f(x)$ fonksiyonunun girdisi P dağılımından farklı x değerleriyle elde edilen rastgele bir değişken olduğunda, $f(x)$ beklentisi şu şekilde hesaplanır:

$$E_{x\sim P}[f(x)] = \sum_x f(x)P(x). \tag{2.6.10}$$

Çoğu durumda X rastgele değişkeninin beklentisinden ne kadar saptığını ölçmek isteriz. Bu, varyans (değişinti) ile ölçülebilir

$$\text{Var}[X] = E\left[(X - E[X])^2\right] = E[X^2] - E[X]^2. \tag{2.6.11}$$

Varyansın karekökü *standart sapma* denir. Rastgele değişkenin bir fonksiyonunun varyansı, fonksiyonun beklentisinden ne kadar saptığını ölçer, çünkü rastgele değişkenin farklı değerleri x, dağılımından örneklenir:

$$\text{Var}[f(x)] = E\left[\left(f(x) - E[f(x)]\right)^2\right]. \tag{2.6.12}$$

2.6.4 Özet

- Olasılık dağılımlarından örnekleme yapabiliriz.
- Bileşik dağılım, koşullu dağılım, Bayes teoremi, marjinalleştirme ve bağımsızlık varsayımlarını kullanarak birden çok rastgele değişkeni analiz edebiliriz.
- Beklenti ve varyans, olasılık dağılımlarının temel özelliklerini özetlemek için yararlı ölçüler sunar.

2.6.5 Alıştırmalar

1. Her grubun $n = 10$ örnek çektiği $m = 500$ deney grubu yürüttük. m ve n değerlerini değiştirin. Deneysel sonuçları gözlemleyin ve analiz edin.
2. $P(\mathcal{A})$ ve $P(\mathcal{B})$ olasılığına sahip iki olay verildiğinde, $P(\mathcal{A} \cup \mathcal{B})$ ve $P(\mathcal{A} \cap \mathcal{B})$ için üst ve alt sınırları hesaplayın (İpucu: Durumu bir Venn Şeması[44] kullanarak görüntüleyin.)
3. A, B ve C gibi rastgele değişkenlerden oluşan bir dizimiz olduğunu varsayalım, burada B yalnızca A'ya bağlıdır ve C yalnızca B'ye bağlıdır, $P(A, B, C)$ birleşik olasılığı basitleştirebilir misiniz? (İpucu: Bu bir Markov Zinciri[45]'dir.)
4. Section 2.6.2 içinde, ilk test daha doğrudur. Hem birinci hem de ikinci testleri yapmak yerine neden ilk testi iki kez yapmıyorsunuz?

Tartışmalar[46]

[44] https://en.wikipedia.org/wiki/Venn_diagram
[45] https://en.wikipedia.org/wiki/Markov_chain
[46] https://discuss.d2l.ai/t/37

2.7 Belgeler (Dökümantasyon)

Bu kitabın uzunluğundaki kısıtlamalar nedeniyle, her bir PyTorch işlevini ve sınıfını tanıtamayız (ve muhtemelen bizim yapmamızı siz de istemezsiniz). API (Application Programming Interface - Uygulama Programlama Arayüzü) belgeleri ve ek öğreticiler (tutorial) ve örnekler kitabın ötesinde pek çok belge sağlar. Bu bölümde size PyTorch API'sini keşfetmeniz için biraz rehberlik sunuyoruz.

2.7.1 Bir Modüldeki Tüm İşlevleri ve Sınıfları Bulma

Bir modülde hangi fonksiyonların ve sınıfların çağrılabileceğini bilmek için `dir` fonksiyonunu çağırırız. Örneğin, rastgele sayılar oluşturmak için modüldeki tüm özellikleri sorgulayabiliriz:

```
import torch

print(dir(torch.distributions))
```

```
['AbsTransform', 'AffineTransform', 'Bernoulli', 'Beta', 'Binomial', 'CatTransform',
↪'Categorical', 'Cauchy', 'Chi2', 'ComposeTransform', 'ContinuousBernoulli',
↪'CorrCholeskyTransform', 'CumulativeDistributionTransform', 'Dirichlet', 'Distribution',
↪'ExpTransform', 'Exponential', 'ExponentialFamily', 'FisherSnedecor', 'Gamma', 'Geometric',
↪ 'Gumbel', 'HalfCauchy', 'HalfNormal', 'Independent', 'IndependentTransform', 'Kumaraswamy
↪', 'LKJCholesky', 'Laplace', 'LogNormal', 'LogisticNormal', 'LowRankMultivariateNormal',
↪'LowerCholeskyTransform', 'MixtureSameFamily', 'Multinomial', 'MultivariateNormal',
↪'NegativeBinomial', 'Normal', 'OneHotCategorical', 'OneHotCategoricalStraightThrough',
↪'Pareto', 'Poisson', 'PowerTransform', 'RelaxedBernoulli', 'RelaxedOneHotCategorical',
↪'ReshapeTransform', 'SigmoidTransform', 'SoftmaxTransform', 'SoftplusTransform',
↪'StackTransform', 'StickBreakingTransform', 'StudentT', 'TanhTransform', 'Transform',
↪'TransformedDistribution', 'Uniform', 'VonMises', 'Weibull', 'Wishart', '__all__', '__
↪builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__',
↪'__path__', '__spec__', 'bernoulli', 'beta', 'biject_to', 'binomial', 'categorical',
↪'cauchy', 'chi2', 'constraint_registry', 'constraints', 'continuous_bernoulli', 'dirichlet
↪', 'distribution', 'exp_family', 'exponential', 'fishersnedecor', 'gamma', 'geometric',
↪'gumbel', 'half_cauchy', 'half_normal', 'identity_transform', 'independent', 'kl', 'kl_
↪divergence', 'kumaraswamy', 'laplace', 'lkj_cholesky', 'log_normal', 'logistic_normal',
↪'lowrank_multivariate_normal', 'mixture_same_family', 'multinomial', 'multivariate_normal',
↪ 'negative_binomial', 'normal', 'one_hot_categorical', 'pareto', 'poisson', 'register_kl',
↪'relaxed_bernoulli', 'relaxed_categorical', 'studentT', 'transform_to', 'transformed_
↪distribution', 'transforms', 'uniform', 'utils', 'von_mises', 'weibull', 'wishart']
```

Genel olarak, `__` (Python'daki özel nesneler) ile başlayan ve biten işlevleri veya tek bir `_` ile başlayan işlevleri (genellikle dahili işlevler) yok sayabiliriz. Kalan işlev veya özellik adlarına bağlı olarak bu modülün tekdüze dağılım (`uniform`), normal dağılım (`normal`) ve çok terimli dağılımdan (`multinomial`) örnekleme dahil, bu modülün rastgele sayılar oluşturmak için çeşitli yöntemler sunduğunu tahmin edebiliriz.

2.7.2 Belli İşlevlerin ve Sınıfların Kullanımını Bulma

Belirli bir işlevin veya sınıfın nasıl kullanılacağına ilişkin daha özel talimatlar için `help` (yardım) işlevini çağırabiliriz. Örnek olarak, tensörlerin `ones` işlevi için kullanım talimatlarını inceleyelim.

```
help(torch.ones)
```

```
Help on built-in function ones in module torch:

ones(...)
    ones(*size, *, out=None, dtype=None, layout=torch.strided, device=None, requires_
↪grad=False) -> Tensor

    Returns a tensor filled with the scalar value 1, with the shape defined
    by the variable argument size.

    Args:
        size (int...): a sequence of integers defining the shape of the output␣
↪tensor.
            Can be a variable number of arguments or a collection like a list or␣
↪tuple.

    Keyword arguments:
        out (Tensor, optional): the output tensor.
        dtype (torch.dtype, optional): the desired data type of returned tensor.
            Default: if None, uses a global default (see torch.set_default_tensor_
↪type()).
        layout (torch.layout, optional): the desired layout of returned Tensor.
            Default: torch.strided.
        device (torch.device, optional): the desired device of returned tensor.
            Default: if None, uses the current device for the default tensor type
            (see torch.set_default_tensor_type()). device will be the CPU
            for CPU tensor types and the current CUDA device for CUDA tensor types.
        requires_grad (bool, optional): If autograd should record operations on the
            returned tensor. Default: False.

    Example::

        >>> torch.ones(2, 3)
        tensor([[ 1.,  1.,  1.],
                [ 1.,  1.,  1.]])

        >>> torch.ones(5)
        tensor([ 1.,  1.,  1.,  1.,  1.])
```

Dökümantasyondan, `ones` işlevinin belirtilen şekle sahip yeni bir tensör oluşturduğunu ve tüm öğeleri 1 değerine ayarladığını görebiliriz. Mümkün oldukça, yorumunuzu onaylamak için hızlı bir test yapmalısınız:

```
torch.ones(4)
```

```
tensor([1., 1., 1., 1.])
```

Jupyter not defterinde, belgeyi başka bir pencerede görüntülemek için ? kullanabiliriz. Örneğin, `list?`, `help(list)` ile neredeyse aynı olan içerik üretecek ve onu yeni bir tarayıcı penceresinde görüntüleyecektir. Ek olarak, `list??` gibi iki soru işareti kullanırsak, işlevi uygulayan Python kodu da görüntülenecektir.

2.7.3 Özet

- Resmi belgeler, bu kitabın dışında pek çok açıklama ve örnek sağlar.
- Jupyter not defterlerinde `dir` ve `help` işlevlerini veya ? ve ?? işlevlerini çağırarak bir API'nin kullanımına ilişkin belgelere bakabiliriz.

2.7.4 Alıştırmalar

1. Derin öğrenme çerçevesindeki herhangi bir işlev veya sınıf için belgelere (dökümantasyon) bakın. Belgeleri çerçevenin resmi web sitesinde de bulabilir misiniz?

Tartışmalar[47]

[47] https://discuss.d2l.ai/t/39

3 | Doğrusal Sinir Ağları

Derin sinir ağlarının ayrıntılarına girmeden önce, sinir ağı eğitiminin temellerini ele almamız gerekiyor. Bu bölümde tüm eğitim sürecini ele alacağız; basit sinir ağı mimarilerinin tanımlanması, verilerin işlenmesi, bir kayıp işlevinin belirtilmesi ve modelin eğitilmesi dahil. İşleri daha kolay kavrayabilmek için en basit kavramlarla başlıyoruz. Neyse ki, doğrusal ve eşiksiz en büyük işlevli (softmax) bağlanım (regresyon) gibi klasik istatistiksel öğrenme teknikleri *doğrusal* sinir ağları olarak kullanılabilir. Bu klasik algoritmalardan başlayarak, size kitabın geri kalanındaki daha karmaşık tekniklerin temelini oluşturan temel bilgileri aktaracağız.

3.1 Doğrusal Bağlanım (Regresyon)

Regresyon (Bağlanım), bir veya daha fazla bağımsız değişken ile bağımlı bir değişken arasındaki ilişkiyi modellemeye yönelik bir grup yöntemi ifade eder. Doğa bilimleri ve sosyal bilimlerde, regresyonun amacı çoğunlukla girdiler ve çıktılar arasındaki ilişkiyi *karakterize etmektir*. Öte yandan, makine öğrenmesi çoğunlukla *tahminle* ilgilidir.

Regresyon problemleri sayısal bir değeri tahmin etmek istediğimiz zaman ortaya çıkar. Yaygın örnekler arasında fiyatları tahmin etmek (evlerin, hisse senetlerinin vb.), kalış süresini tahmin etmek (hastanedeki hastalar için), talep öngörmek (perakende satışlar için) ve sayısız başkaları sayılabilir. Her tahmin problemi klasik bir regresyon problemi değildir. Sonraki bölümlerde, amacın bir dizi kategori arasından üyeliği tahmin etmek olduğu sınıflandırma problemlerini tanıtacağız.

3.1.1 Doğrusal Regresyonun Temel Öğeleri

Doğrusal regresyon, regresyon için standart araçlar arasında hem en basiti hem de en popüleri olabilir. Tarihi 19. yüzyılın başlarına kadar uzanan doğrusal regresyon birkaç basit varsayımdan doğmaktadır. İlk olarak, $\mathbf{x}$ bağımsız değişkenleri ve y bağımlı değişkeni arasındaki ilişkinin doğrusal olduğunu, yani y'nin gözlemlerdeki gürültüde göz önüne alınarak $\mathbf{x}$ içindeki öğelerin ağırlıklı toplamı olarak ifade edilebileceğini varsayıyoruz. İkinci olarak, herhangi bir gürültünün iyi davrandığını varsayıyoruz (bir Gauss dağılımı takip ettiklerini).

Yaklaşımı motive etmek için işleyen bir örnekle başlayalım. Evlerin fiyatlarını (dolar cinsinden) alanlarına (metre kare cinsinden) ve yaşlarına (yıl olarak) göre tahmin etmek istediğimizi varsayalım. Gerçekten ev fiyatlarını tahmin etmede bir model geliştirirken, her evin satış fiyatını, alanını ve yaşını bildiğimiz satışlardan oluşan bir veri kumesinin elimizde olması gerekir. Makine öğrenmesi terminolojisinde, veri kümesine *eğitim veri kümesi* veya *eğitim kümesi* denir, ve her satır (burada bir satışa karşılık gelen veriler) bir *örnek* (veya *veri noktası*, *veri örneği*, *örneklem*) olarak adlandırılır. Tahmin etmeye çalıştığımız (fiyat) şeye *etiket* (veya *hedef*) denir. Tahminlerin dayandığı bağımsız değişkenler (yaş ve alan), *öznitelikler* (veya *ortak değişkenler*) olarak adlandırılır.

Tipik olarak, veri kümemizdeki örneklerin sayısını belirtmek için n'yi kullanacağız. Veri örneklerini i ile indeksliyoruz, her girdiyi $\mathbf{x}^{(i)} = [x_1^{(i)}, x_2^{(i)}]^\top$ ve karşılık gelen etiketi $y^{(i)}$ olarak gösteriyoruz.

Doğrusal Model

Doğrusallık varsayımı sadece hedefin (fiyat) özniteliklerin ağırlıklı toplamı (alan ve yaş) olarak ifade edilebileceğini söylüyor:

$$\text{fiyat} = w_{\text{alan}} \cdot \text{alan} + w_{\text{yaş}} \cdot \text{yaş} + b. \tag{3.1.1}$$

(3.1.1)'de, w_{alan} ve $w_{\text{yaş}}$, *ağırlıklar* ve b *ek girdi (bias)* (aynı zamanda *offset* veya *kesim noktası*) olarak adlandırılır. Ağırlıklar, her özniteliğin tahminimiz üzerindeki etkisini belirler ve ek girdi, tüm öznitelikler 0 değerini aldığında tahmin edilen fiyatın hangi değeri alması gerektiğini söyler. Sıfır alana sahip veya tam olarak sıfır yaşında olan hiçbir ev göremeyecek olsak da, hala ek girdiye ihtiyacımız var, yoksa modelimizin ifade gücünü sınırlayacağız. Kesin olarak konuşursak, (3.1.1) ifadesi, ağırlıklı toplam yoluyla özniteliklerin bir *doğrusal dönüşümü* ile karakterize edilen ve eklenen ek girdi sayesinde bir *öteleme (translation)* ile birleştirilen girdi, özniteliklerinin bir *afin (affine) dönüşümüdür*.

Bir veri kümesi verildiğinde, amacımız $\mathbf{w}$ ağırlıklarını ve b ek girdisini, ortalamada, modelimizce yapılan tahminlerin veride gözlemlenen gerçek fiyatlara en iyi uyacak şekilde seçmektir. Tahmin çıktıları girdi özniteliklerinin afin dönüşümü ile belirlenen modeller *doğrusal modellerdir*, ki burada afin dönüşümü seçilen ağırlıklar ve ek girdi belirler.

Sadece birkaç özniteliğe sahip veri kümelerine odaklanmanın yaygın olduğu disiplinlerde, modellerin böyle açıkça uzun biçimli ifade edilmesi yaygındır. Makine öğrenmesinde, genellikle yüksek boyutlu veri kümeleriyle çalışırız, bu nedenle doğrusal cebir gösterimini kullanmak daha uygundur. Girdilerimiz d öznitelikten oluştuğunda, $\hat{y}$ tahminimizi (genel olarak "şapka" sembolü tahminleri gösterir) şu şekilde ifade ederiz:

$$\hat{y} = w_1 x_1 + ... + w_d x_d + b. \tag{3.1.2}$$

Tüm öznitelikleri $\mathbf{x} \in \mathbb{R}^d$ vektöründe ve tüm ağırlıkları $\mathbf{w} \in \mathbb{R}^d$ vektöründe toplayarak modelimizi bir iç çarpım kullanarak sıkıştırılmış biçimde ifade edebiliriz:

$$\hat{y} = \mathbf{w}^\top \mathbf{x} + b. \tag{3.1.3}$$

(3.1.3)'de, $\mathbf{x}$ vektörü tek bir veri örneğinin özniteliklerine karşılık gelir. n örnekli veri kümemizin tümünün özniteliklerine *tasarım matrisi* $\mathbf{X} \in \mathbb{R}^{n \times d}$ aracılığıyla atıfta bulunmayı genellikle uygun bulacağız. Burada, $\mathbf{X}$ her örnek için bir satır ve her özellik için bir sütun içerir.

$\mathbf{X}$ özniteliklerinden oluşan bir koleksiyon için, tahminler, $\hat{\mathbf{y}} \in \mathbb{R}^n$, matris-vektör çarpımı ile ifade edilebilir:

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b, \tag{3.1.4}$$

burada yayma (bkz Section 2.1.3) toplama esnasında uygulanır. $\mathbf{X}$ eğitim veri kümesinin öznitelikleri ve karşılık gelen (bilinen) $\mathbf{y}$ etiketleri verildiğinde, doğrusal regresyonun amacı $\mathbf{w}$ ağırlık vektörünü ve b ek girdi terimini bulmaktır, öyle ki $\mathbf{X}$ ile aynı dağılımdan örneklenmiş yeni bir örneğin öznitelikleri verildiğinde, yeni veri örneğinin etiketi (ortalamada) en düşük hata ile tahmin edilecektir.

x verildiğinde y tahmini için en iyi modelin doğrusal olduğuna inansak bile, n örnekten oluşan bir gerçek dünya veri kümesinde $y^{(i)}$'nin tüm $1 \leq i \leq n$ için $\mathbf{w}^\top \mathbf{x}^{(i)} + b$'ye tam olarak eşit olmasını beklemiyoruz. Örneğin, **X** özelliklerini ve **y** etiketlerini gözlemlemek için kullandığımız araçlar ne olursa olsun az miktarda ölçüm hatası yapabilir. Dolayısıyla, temeldeki ilişkinin doğrusal olduğundan emin olsak bile, bu tür hataları hesaba katmak için bir gürültü terimi dahil edeceğiz.

En iyi *parametreleri* (veya *model parametrelerini*) **w** ve b'yi aramaya başlamadan önce, iki şeye daha ihtiyacımız var: (i) Belirli bir model için bir kalite ölçütü; ve (ii) kalitesini iyileştirmek için modelin güncellenmesine yönelik bir yordam (prosedür).

Kayıp İşlevi

Modelimizi veri ile nasıl *oturtacağımızı* düşünmeye başlamadan önce, bir *uygunluk* ölçüsü belirlememiz gerekir. *Kayıp işlevi*, hedefin *gerçek* ve *tahmini* değeri arasındaki mesafeyi ölçer. Kayıp, genellikle, daha küçük değerlerin daha iyi olduğu ve mükemmel tahminlerin 0 kayba neden olduğu negatif olmayan bir sayı olacaktır. Regresyon problemlerinde en popüler kayıp fonksiyonu, hata karesidir. Bir i örneğine ilişkin tahminimiz $\hat{y}^{(i)}$ ve buna karşılık gelen doğru etiket $y^{(i)}$ olduğunda, hata karesi şu şekilde verilir:

$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2}\left(\hat{y}^{(i)} - y^{(i)}\right)^2. \tag{3.1.5}$$

$\frac{1}{2}$ sabiti gerçekte bir fark yaratmaz, ancak gösterim olarak uygun olduğunu ispatlayacak ve kaybın türevini aldığımızda dengelenip kaybolacak. Eğitim veri kümesi bize verildiğinden, kontrolümüz dışında, sadece deneysel hata model parametrelerinin bir fonksiyonudur. İşleri daha somut hale getirmek için, aşağıdaki örnekte gösterildiği gibi tek boyutlu bir durum için bir regresyon problemi çizdiğimizi düşünün Fig. 3.1.1.

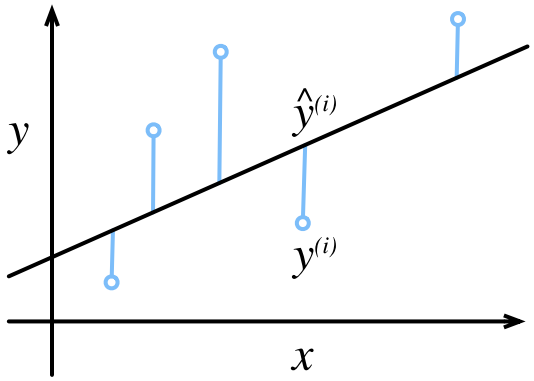


Fig. 3.1.1: Verilere doğrusal bir model oturtmak.

$\hat{y}^{(i)}$ tahminleri ile $y^{(i)}$ gözlemleri arasındaki büyük farkların ikinci dereceden bağımlılık nedeniyle daha da büyük kayıba neden olduğuna dikkat edin. n örnekli veri kümesinin tamamında bir modelin kalitesini ölçmek için, eğitim kümesindeki kayıpların ortalamasını alıyoruz (veya eşdeğer bir şekilde topluyoruz).

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} l^{(i)}(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2. \tag{3.1.6}$$

Modeli eğitirken, tüm eğitim örneklerinde toplam kaybı en aza indiren parametreleri ($\mathbf{w}^*, b^*$) bulmak istiyoruz:

$$\mathbf{w}^*, b^* = \underset{\mathbf{w},b}{\operatorname{argmin}}\ L(\mathbf{w}, b). \tag{3.1.7}$$

Analitik Çözüm

Doğrusal regresyon, alışılmadık derecede basit bir optimizasyon problemi haline gelir. Bu kitapta karşılaşacağımız diğer modellerin çoğunun aksine, doğrusal regresyon, basit bir formül uygulanarak analitik olarak çözülebilir. Başlangıç olarak, tüm olanlardan oluşan tasarım matrisine bir sütun ekleyerek b ek girdisini $\mathbf{w}$ parametresine dahil edebiliriz. Öyleyse tahmin problemimiz $\|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2$'yi en aza indirmektir. Kayıp yüzeyinde sadece bir kritik nokta vardır ve bütün alandaki minimum kayba denk gelir. $\mathbf{w}$'ye göre kaybın türevini almak ve sıfıra eşitlemek, analitik (kapalı form) çözümü verir:

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}. \tag{3.1.8}$$

Doğrusal regresyon gibi basit problemler analitik çözümleri sunarken, her zaman bu kadar talihli olmazsınız. Analitik çözümler güzel matematiksel analize izin verse de, analitik bir çözümün gerekliliği o kadar kısıtlayıcıdır ki tüm derin öğrenme dışarıda kalır.

Minigrup Rasgele Gradyan (Eğim) İnişi

Modelleri analitik olarak çözümleyemediğimiz durumlarda bile, yine de pratikte etkili bir şekilde modelleri eğitebileceğimiz ortaya çıkıyor. Dahası, birçok görev için, optimize edilmesi zor olan bu modeller o kadar iyi çalışırlar ki, onların nasıl eğiteceklerini bulma zahmetine değecektir.

Neredeyse tüm derin öğrenme modellerini optimize etmek için kullanılan ve bu kitap boyunca değineceğimiz temel teknik, kayıp fonksiyonunu kademeli olarak düşüren yönde parametreleri güncelleyerek hatayı yinelemeli olarak azaltmaktan ibarettir. Bu algoritmaya *gradyan iniş* denir.

Gradyan inişinin en saf uygulaması, veri kümesindeki her bir örnekten hesaplanan kayıpların ortalaması olan kayıp fonksiyonunun türevini almaktan oluşur. Pratikte bu çok yavaş olabilir: Tek bir güncelleme yapmadan önce tüm veri kümesinin üzerinden geçmeliyiz. Bu nedenle, güncellemeyi her hesaplamamız gerektiğinde rastgele bir minigrup örnekten örneklemeyi *minigrup rasgele gradyan inişi* adı verilen bir yöntem ile deneyeceğiz.

Her bir yinelemede, ilk olarak sabit sayıda eğitim örneğinden oluşan bir minigrubu, $\mathcal{B}$, rasgele örnekliyoruz. Daha sonra minigruptaki ortalama kaybın türevini (gradyan) model parametrelerine göre hesaplıyoruz. Son olarak, gradyanı önceden belirlenmiş pozitif bir değerle, η, çarpıyoruz ve ortaya çıkan terimi mevcut parametre değerlerinden çıkarıyoruz.

Güncellemeyi matematiksel olarak şu şekilde ifade edebiliriz (∂ kısmi türevi belirtir):

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{(\mathbf{w},b)} l^{(i)}(\mathbf{w}, b). \tag{3.1.9}$$

Özetlemek gerekirse, algoritmanın adımları şöyledir: (i) Model parametrelerinin değerlerini tipik olarak rastgele olarak başlatıyoruz; (ii) verilerden yinelemeli olarak rastgele minigruplar örnekleyerek parametreleri negatif gradyan yönünde güncelliyoruz. İkinci dereceden kayıplar

ve afin dönüşümler için, bunu açıkça şu şekilde yazabiliriz:

$$\begin{aligned}\mathbf{w} &\leftarrow \mathbf{w} - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}}\partial_{\mathbf{w}} l^{(i)}(\mathbf{w}, b) = \mathbf{w} - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}}\mathbf{x}^{(i)}\left(\mathbf{w}^\top\mathbf{x}^{(i)} + b - y^{(i)}\right),\\ b &\leftarrow b - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}}\partial_b l^{(i)}(\mathbf{w}, b) = b - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}}\left(\mathbf{w}^\top\mathbf{x}^{(i)} + b - y^{(i)}\right).\end{aligned} \tag{3.1.10}$$

$\mathbf{w}$ ve $\mathbf{x}$'in (3.1.10) içinde vektörler olduğuna dikkat edin. Burada, daha zarif vektör gösterimi, matematiği, $w_1, w_2, \ldots, w_d$ gibi, katsayılarla ifade etmekten çok daha okunaklı hale getirir. Küme niceliği (kardinalite), $|\mathcal{B}|$, her bir minigruptaki örneklerin sayısını (*grup boyutu*) ve η *öğrenme oranını* gösterir. Burada grup boyutu ve öğrenme oranı değerlerinin manuel olarak önceden belirlendiğini ve tipik olarak model eğitimi yoluyla öğrenilmediğini vurguluyoruz. Ayarlanabilir ancak eğitim döngüsünde güncellenmeyen bu parametrelere *hiper parametreler* denir. *Hiper parametre ayarı*, hiper parametrelerin seçildiği süreçtir ve genellikle onları eğitim döngüsününde ayrı bir *geçerleme veri kümesinde* (veya *geçerleme kümesinde*) elde edilen değerlendirilme sonuçlarına göre ayarlamamızı gerektirir.

Önceden belirlenmiş sayıda yineleme kadar eğitimden sonra (veya başka bazı durdurma kriterleri karşılanana kadar), $\hat{\mathbf{w}}, \hat{b}$ olarak belirtilen tahmini model parametrelerini kaydediyoruz. Fonksiyonumuz gerçekten doğrusal ve gürültüsüz olsa bile, bu parametreler kaybın kesin minimum değerleri olmayacaktır, çünkü algoritma yavaşça en küçük değerlere doğru yaklaşsa da sonlu bir adımda tam olarak başaramayacaktır.

Doğrusal regresyon, tüm etki alanı üzerinde yalnızca bir minimumun olduğu bir öğrenme problemi haline gelir. Halbuki, derin ağlar gibi daha karmaşık modeller için kayıp yüzeyleri birçok minimum içerir. Neyse ki, henüz tam olarak anlaşılmayan nedenlerden dolayı, derin öğrenme uygulayıcıları nadiren *eğitim kümelerinde* kaybı en aza indirecek parametreleri bulmakta zorlanırlar. Daha zorlu görev, daha önce görmediğimiz verilerde düşük kayıp elde edecek parametreler bulmaktır; bu, *genelleme* denen bir zorluktur. Kitap boyunca bu konulara döneceğiz.

Öğrenilen Modelle Tahmin Yapma

Öğrenilen doğrusal regresyon modeli $\hat{\mathbf{w}}^\top\mathbf{x} + \hat{b}$ göz önüne alındığında, artık (eğitim verilerinde yer almayan) yeni bir evin alanı, x_1, ve yaşı, x_2, verildiğinde fiyatını tahmin edebiliriz. Öznitelikler verilince hedeflerin tahmin edilmesine genellikle *tahmin* veya *çıkarım* denir.

Tahmine bağlı kalmaya çalışacağız çünkü bu adımı *çıkarım* olarak adlandırmak, derin öğrenmede standart bir jargon olarak ortaya çıkmasına rağmen, bir şekilde yanlış bir isimdir. İstatistiklerde, *çıkarım* daha çok bir veri kümesine dayanarak parametreleri tahmin etmeyi ifade eder. Terminolojinin bu kötüye kullanımı, derin öğrenme uygulayıcıları istatistikçilerle konuşurken yaygın bir kafa karışıklığı kaynağıdır.

3.1.2 Hız için Vektörleştirme

Modellerimizi eğitirken, tipik olarak tüm minigrup örneklerini aynı anda işlemek isteriz. Bunu verimli bir şekilde yapmak, Python'da maliyetli döngüler yazmak yerine, hesaplamaları vektörleştirmemizi ve hızlı doğrusal cebir kütüphanelerinden yararlanmamızı gerektirir.

```
%matplotlib inline
import math
```

(continues on next page)

(continued from previous page)

```
import time
import numpy as np
import torch
from d2l import torch as d2l
```

Bunun neden bu kadar önemli olduğunu göstermek için, vektörleri eklemek için iki yöntem düşünebiliriz. Başlangıç olarak, bütün girdileri 1 olan 10000 boyutlu iki vektör tanımlıyoruz. Bir yöntemde vektörler üzerinde bir Python for-döngüsü ile döngü yapacağız. Diğer yöntemde, tek bir + çağrısına güveneceğiz.

```
n = 10000
a = torch.ones(n)
b = torch.ones(n)
```

Bu kitapta çalışma süresini sık sık karşılaştıracağımızdan bir zamanlayıcı tanımlayalım.

```
class Timer:  #@save
    """Birden fazla koşma zamanını kaydedin."""
    def __init__(self):
        self.times = []
        self.start()

    def start(self):
        """Zamanlayıcıyı başlatın."""
        self.tik = time.time()

    def stop(self):
        """Zamanlayıcı durdurun ve zamanı listeye kaydedin."""
        self.times.append(time.time() - self.tik)
        return self.times[-1]

    def avg(self):
        """Ortalama zamanı döndürün."""
        return sum(self.times) / len(self.times)

    def sum(self):
        """Toplam zamanı döndürün."""
        return sum(self.times)

    def cumsum(self):
        """Biriktirilmiş zamanı döndürün."""
        return np.array(self.times).cumsum().tolist()
```

Artık iş yüklerini karşılaştırabiliriz. İlk olarak, bir for döngüsü kullanarak her seferinde bir koordinat gibi onları topluyoruz.

```
c = torch.zeros(n)
timer = Timer()
for i in range(n):
    c[i] = a[i] + b[i]
f'{timer.stop():.5f} sn'
```

```
'0.10208 sn'
```

Alternatif olarak, eleman yönlü toplamı hesaplamak için yeniden yüklenen + operatörüne güveniyoruz.

```
timer.start()
d = a + b
f'{timer.stop():.5f} sn'
```

```
'0.00027 sn'
```

Muhtemelen ikinci yöntemin birincisinden çok daha hızlı olduğunu fark etmişsinizdir. Vektörleştirme kodu genellikle büyük ölçeklerde hız artışları sağlar. Dahası, matematiğin çoğunun sorumluluğunu kütüphaneye yüklüyoruz ve kendimizin bu kadar çok hesaplamayı yazmamıza gerek yok, bu da hata olasılığını azaltıyor.

3.1.3 Normal Dağılım ve Kare Kayıp

Sadece yukarıdaki bilgileri kullanarak ellerinizi şimdiden kirletebilecekken, aşağıda gürültünün dağılımı ile ilgili varsayımlar yoluyla kare kayıp amaç fonsiyonunu daha biçimsel (formal) olarak motive edici hale getirebiliriz.

Doğrusal regresyon, 1795'te normal (*Gauss* olarak da adlandırılır) dağılımını da keşfeden Gauss tarafından icat edildi. Görünüşe göre normal dağılım ve doğrusal regresyon arasındaki bağlantı, ortak ebeveynlikten daha derindir. Belleğinizi yenilemek için, ortalaması μ ve varyansı σ^2 (standart sapma σ) olan normal bir dağılımın olasılık yoğunluğu şu şekilde verilir:

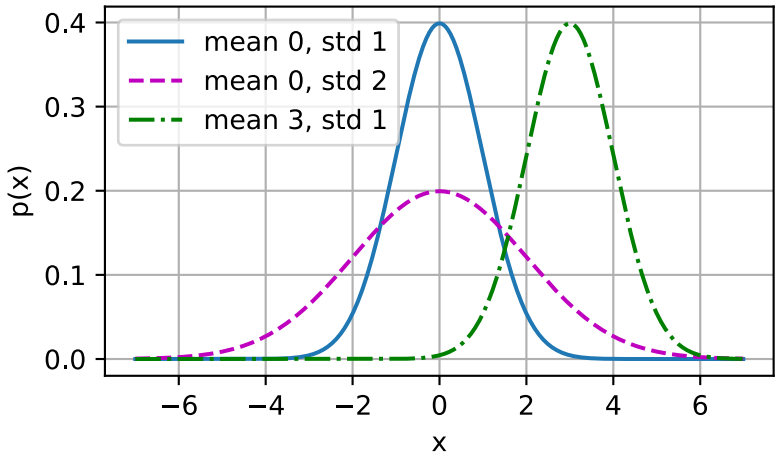
$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right). \tag{3.1.11}$$

Aşağıda normal dağılımı hesaplamak için bir Python işlevi tanımlıyoruz.

```
def normal(x, mu, sigma):
    p = 1 / math.sqrt(2 * math.pi * sigma**2)
    return p * np.exp(-0.5 / sigma**2 * (x - mu)**2)
```

Artık normal dağılımları görselleştirebiliriz.

```
# Görselleştirme için gene numpy kullanın
x = np.arange(-7, 7, 0.01)

# Ortalama ve standart sapma çifti
params = [(0, 1), (0, 2), (3, 1)]
d2l.plot(x, [normal(x, mu, sigma) for mu, sigma in params], xlabel='x',
         ylabel='p(x)', figsize=(4.5, 2.5),
         legend=[f'mean {mu}, std {sigma}' for mu, sigma in params])
```



Gördüğümüz gibi, ortalamanın değiştirilmesi x ekseni boyunca bir kaymaya karşılık gelir ve varyansı arttırmak dağılımı yayarak tepe noktasını düşürür.

Ortalama hata karesi kayıp fonksiyonlu (veya basitçe kare kaybı) doğrusal regresyonu motive etmenin bir yolu, gözlemlerin, gürültünün aşağıdaki gibi normal dağıldığı gürültülü gözlemlerden kaynaklandığını biçimsel olarak varsaymaktır:

$$y = \mathbf{w}^\top \mathbf{x} + b + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, \sigma^2). \tag{3.1.12}$$

Böylece, belirli bir y'yi belirli bir $\mathbf{x}$ için görme *olabilirliğini* şu şekilde yazabiliriz:

$$P(y \mid \mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y - \mathbf{w}^\top \mathbf{x} - b)^2\right). \tag{3.1.13}$$

Şimdi, maksimum olabilirlik ilkesine göre, $\mathbf{w}$ ve b parametrelerinin en iyi değerleri, tüm veri kümesinin *olabilirliğini* en üst düzeye çıkaranlardır:

$$P(\mathbf{y} \mid \mathbf{X}) = \prod_{i=1}^{n} p(y^{(i)}|\mathbf{x}^{(i)}). \tag{3.1.14}$$

Maksimum olabilirlik ilkesine göre seçilen tahminciler, *maksimum olabilirlik tahmincileri* olarak adlandırılır. Birçok üstel fonksiyonun çarpımını maksimize etmek zor görünse de, bunun yerine olabilirliğin logaritmasını maksimize ederek, amaç fonksiyonunu değiştirmeden işleri önemli ölçüde basitleştirebiliriz. Tarihsel nedenlerden dolayı, eniyilemeler daha çok azamileştirmekten (maksimizasyon) ziyade asgarileştirme (minimizasyon) olarak ifade edilir. Dolayısıyla, hiçbir şeyi değiştirmeden *negatif log-olabilirlik*, $-\log P(\mathbf{y} \mid \mathbf{X})$, değerini en aza indirebiliriz. Matematik üzerinde biraz çalışmak bize şunu verir:

$$-\log P(\mathbf{y} \mid \mathbf{X}) = \sum_{i=1}^{n} \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}\left(y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b\right)^2. \tag{3.1.15}$$

Şimdi σ'nın bir sabit olduğu varsayımına da ihtiyacımız var. Böylece ilk terimi göz ardı edebiliriz çünkü bu $\mathbf{w}$ veya b'ye bağlı değildir. Şimdi ikinci terim, çarpımsal sabit $\frac{1}{\sigma^2}$ dışında, daha önce tanıtılan hata karesi kaybıyla aynıdır. Neyse ki, çözüm σ'ya bağlı değildir. Ortalama hata karesini en aza indirmenin, eklenen Gauss gürültüsü varsayımı altında doğrusal bir modelin maksimum olabilirlik tahminine eşdeğer olduğu sonucu çıkar.

3.1.4 Doğrusal Regresyondan Derin Ağlara

Şimdiye kadar sadece doğrusal modellerden bahsettik. Sinir ağları çok daha zengin bir model ailesini kapsarken, doğrusal modeli sinir ağları dilinde ifade ederek bir sinir ağı gibi düşünmeye başlayabiliriz. Başlangıç için, nesneleri bir "katman" gösteriminde yeniden yazalım.

Yapay Sinir Ağı Şeması

Derin öğrenme uygulayıcıları, modellerinde neler olduğunu görselleştirmek için şemalar çizmeyi severler. Fig. 3.1.2 içinde, doğrusal regresyon modelimizi bir sinir ağı olarak tasvir ediyoruz. Bu diyagramların, her bir girdinin çıktıya nasıl bağlandığı gibi bağlantı desenlerini vurguladığına, ancak ağırlıkların veya ek girdilerin aldığı değerleri vurgulamadığına dikkat edin.

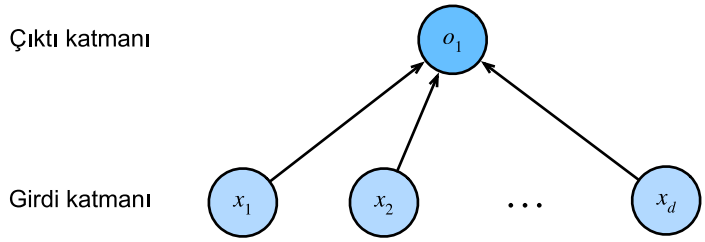


Fig. 3.1.2: Doğrusal regresyon, tek katmanlı bir sinir ağıdır.

Fig. 3.1.2 içinde gösterilen sinir ağı için, girdiler $x_1, \ldots, x_d$'dir, dolayısıyla girdi katmanındaki *girdi sayısı* (veya *öznitelik boyutu*) d'dir. Ağın Fig. 3.1.2 içindeki çıktısı o_1'dir, dolayısıyla çıktı katmanındaki *çıktı sayısı* 1'dir. Girdi değerlerinin hepsinin *verildiğini* ve sadece tek bir *hesaplanmış* nöron (sinir hücresi) olduğuna dikkat edin. Hesaplamanın nerede gerçekleştiğine odaklanarak, geleneksel olarak katmanları sayarken girdi katmanını dikkate almayız. Yani Fig. 3.1.2 içindeki sinir ağı için *katman sayısı* 1'dir. Doğrusal regresyon modellerini sadece tek bir yapay nörondan oluşan sinir ağları veya tek katmanlı sinir ağları olarak düşünebiliriz.

Doğrusal regresyon için, her girdi her çıktıya bağlı olduğundan (bu durumda yalnızca bir çıktı vardır), bu dönüşümü (Fig. 3.1.2 içindeki çıktı katmanı) *tam bağ(lantı)lı katman* veya *yoğun katman* olarak kabul edebiliriz. Bir sonraki bölümde bu tür katmanlardan oluşan ağlar hakkında daha çok konuşacağız.

Biyoloji

Doğrusal regresyon (1795'te icat edildi) hesaplamalı sinirbilimden önce geldiğinden, doğrusal regresyonu bir sinir ağı gibi tanımlamak kronolojik hata olarak görünebilir. Sibernetikçiler/nörofizyologlar, Warren McCulloch ve Walter Pitts, yapay sinir hücresi modelleri geliştirmeye başladığında doğrusal modellerin neden doğal bir başlangıç noktası olduğunu anlamak için, bir biyolojik sinir hücresinin karikatürize resmini düşünün Fig. 3.1.3: *Dendritlerden* (girdi terminalleri), *çekirdekten* (CPU), *aksondan* (çıktı teli) ve *akson terminallerinde* (çıktı terminalleri) oluşur ve *sinapslar* aracılığıyla diğer nöronlara bağlantı sağlar.

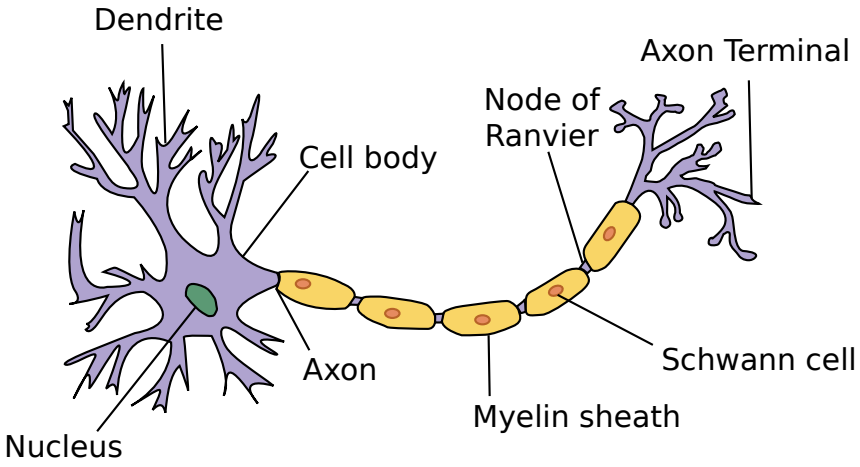


Fig. 3.1.3: Gerçek nöron.

Dendritlerde diğer nöronlardan (veya retina gibi çevresel sensörlerden) gelen bilgiyi, x_i, alınır. Özellikle, bu bilgi, girdilerin etkisini belirleyen *sinaptik ağırlıklar*, w_i, ile ağırlıklandırılır (örneğin, $x_i w_i$ çarpımı aracılığıyla etkinleştirme veya engelleme). Birden fazla kaynaktan gelen ağırlıklı girdiler, çekirdekte ağırlıklı bir toplam $y = \sum_i x_i w_i + b$ olarak biriktirilir ve bu bilgi daha sonra, tipik olarak bazı doğrusal olmayan işlemlerden, $\sigma(y)$ gibi, sonra y aksonunda daha fazla işlenmek üzere gönderilir. Oradan ya hedefine (örneğin bir kas) ulaşır ya da dendritleri yoluyla başka bir nörona beslenir.

Kuşkusuz, birçok birimin, tek başına herhangi bir nöronun ifade edebileceğinden çok daha ilginç ve karmaşık davranışlar üretmek için doğru bağlanabilirlik ve doğru öğrenme algoritmasıyla bir araya getirilebileceği yönündeki bu tür üst düzey fikir, varlığını gerçek biyolojik sinir sistemleri çalışmamıza borçludur.

Aynı zamanda, günümüzde derin öğrenmedeki çoğu araştırma, sinirbilimden çok az ilham alıyor. Stuart Russell ve Peter Norvig'in klasik YZ metin kitaplarında *Yapay Zeka: Modern Bir Yaklaşım (Artificial Intelligence: A Modern Approach)* (Russell and Norvig, 2016), görüyoruz ki, uçaklar kuşlardan *esinlenmiş* olsa da ornitolojinin birkaç yüzyıldır havacılık yeniliklerinin ana itici gücü olmamıştır. Aynı şekilde, bugünlerde derin öğrenmedeki ilham eşit veya daha büyük ölçüde matematik, istatistik ve bilgisayar bilimlerinden geliyor.

3.1.5 Özet

- Bir makine öğrenmesi modelindeki temel bileşenler eğitim verileri, bir kayıp işlevi, bir optimizasyon algoritması ve oldukça açık bir şekilde modelin kendisidir.
- Vektörleştirme, her şeyi daha iyi (çoğunlukla matematik) ve daha hızlı (çoğunlukla kod) yapar.
- Bir amaç işlevini en aza indirmek ve maksimum olabilirlik tahminini gerçekleştirmek aynı anlama gelebilir.
- Doğrusal regresyon modelleri de sinir ağlarıdır.

3.1.6 Alıştırmalar

1. $x_1, \ldots, x_n \in \mathbb{R}$ verisine sahip olduğumuzu varsayalım. Amacımız, $\sum_i (x_i - b)^2$ en aza indirilecek şekilde sabit bir b bulmaktır.
    1. b'nin optimum değeri için analitik bir çözüm bulunuz.
    2. Bu problem ve çözümü normal dağılımla nasıl ilişkilidir?
2. Hata kareli doğrusal regresyon için optimizasyon probleminin analitik çözümünü türetiniz. İşleri basitleştirmek için, problemden b ek girdisini çıkarabilirsiniz (bunu ilkeli bir şekilde, hepsini içeren $\mathbf{X}$'e bir sütun ekleyerek yapabiliriz).
    1. Optimizasyon problemini matris ve vektör gösteriminde yazınız (tüm verileri tek bir matris ve tüm hedef değerleri tek bir vektör olarak ele alınız).
    2. w'ye göre kaybın gradyanını hesaplayınız.
    3. Gradyanı sıfıra eşitleyerek ve matris denklemini çözerek analitik çözümü bulunuz.
    4. Bu ne zaman rasgele gradyan iniş kullanmaktan daha iyi olabilir? Bu yöntem ne zaman bozulabilir?
3. Eklenen gürültü ϵ'u etkin gürültü modelinin üstel dağılım olduğunu varsayın. Yani, $p(\epsilon) = \frac{1}{2} \exp(-|\epsilon|)$'dur.
    1. Verilerin negatif log-olabilirliğini $-\log P(\mathbf{y} \mid \mathbf{X})$ modeli altında yazınız.
    2. Kapalı form çözümü bulabilir misiniz?
    3. Bu sorunu çözmek için bir rasgele gradyan iniş algoritması önerin. Ne yanlış gidebilir (İpucu: Parametreleri güncellemeye devam ederken durağan noktanın yakınında ne olur)? Bunu düzeltebilir misiniz?

Tartışmalar[48]

3.2 Sıfırdan Doğrusal Regresyon Uygulaması Yaratma

Artık doğrusal regresyonun arkasındaki temel fikirleri anladığınıza göre, işe ellerimizi daldırarak kodda uygulamaya başlayabiliriz. Bu bölümde, veri komut işleme hattı (pipeline), model, kayıp fonksiyonu ve minigrup rasgele gradyan iniş eniyileyici dahil olmak üzere tüm yöntemi sıfırdan uygulayacağız. Modern derin öğrenme çerçeveleri neredeyse tüm bu çalışmayı otomatikleştirebilirken, bir şeyleri sıfırdan uygulamak, ne yaptığınızı gerçekten bildiğinizden emin olmanın tek yoludur. Dahası, modelleri ihtiyacımıza göre özelleştirirken, kendi katmanlarımızı veya kayıp işlevlerimizi tanımlama zamanı geldiğinde, kaputun altında işlerin nasıl ilerlediğini anlamak kullanışlı olacaktır. Bu bölümde, sadece tensörlere ve otomatik türev almaya güveneceğiz. Daha sonra, derin öğrenme çerçevelerinin çekici ek özelliklerden yararlanarak daha kısa bir uygulama sunacağız.

```
%matplotlib inline
import random
import torch
from d2l import torch as d2l
```

[48] https://discuss.d2l.ai/t/258

3.2.1 Veri Kümesini Oluşturma

İşleri basitleştirmek için, gürültülü doğrusal bir model için yapay bir veri kümesi oluşturacağız. Görevimiz, veri kümemizde bulunan sonlu örnek kümesini kullanarak bu modelin parametrelerini elde etmek olacaktır. Verileri düşük boyutlu tutacağız, böylece kolayca görselleştirebiliriz. Aşağıdaki kod parçacığında, her biri standart bir normal dağılımdan örneklenmiş 2 öznitelikten oluşan 1000 örnekli bir veri kümesi oluşturuyoruz. Böylece, sentetik veri kümemiz matris $\mathbf{X} \in \mathbb{R}^{1000 \times 2}$ olacaktır.

Veri kümemizi oluşturan gerçek parametreler $\mathbf{w} = [2, -3.4]^\top$ ve $b = 4.2$ olacaktır ve sentetik etiketlerimiz aşağıdaki ϵ gürültü terimli doğrusal modele göre atanacaktır:

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b + \epsilon. \tag{3.2.1}$$

ϵ'u öznitelikler ve etiketlerdeki olası ölçüm hatalarını yakalıyor diye düşünebilirsiniz. Standart varsayımların geçerli olduğunu ve böylece ϵ değerinin ortalaması 0 olan normal bir dağılıma uyduğunu varsayacağız. Problemimizi kolaylaştırmak için, standart sapmasını 0.01 olarak ayarlayacağız. Aşağıdaki kod, sentetik veri kümemizi üretir.

```
def synthetic_data(w, b, num_examples):  #@save
    """Veri yaratma, y = Xw + b + gürültü."""
    X = torch.normal(0, 1, (num_examples, len(w)))
    y = torch.matmul(X, w) + b
    y += torch.normal(0, 0.01, y.shape)
    return X, y.reshape((-1, 1))
```

```
true_w = torch.tensor([2, -3.4])
true_b = 4.2
features, labels = synthetic_data(true_w, true_b, 1000)
```

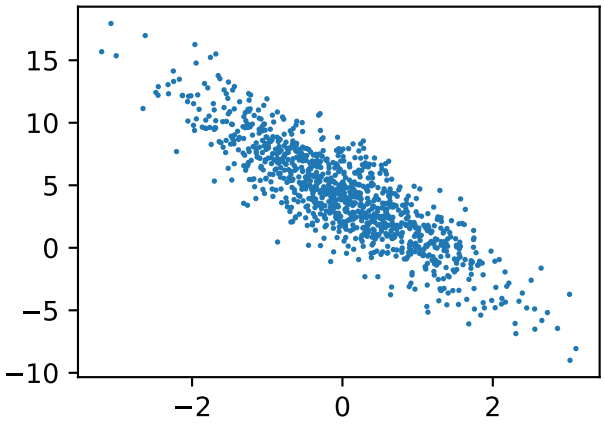
`features`'deki (öznitelikleri tutan değişken) her satırın 2 boyutlu bir veri örneğinden oluştuğuna ve `labels`'deki (etiketleri tutan değişken) her satırın 1 boyutlu bir etiket değerinden (bir skaler) oluştuğuna dikkat edin.

```
print('oznitelikler:', features[0],'\netiket:', labels[0])
```

```
oznitelikler: tensor([0.1546, 2.3925])
etiket: tensor([-3.6205])
```

İkinci öznitelik `features[:, 1]` ve `labels` kullanılarak bir dağılım grafiği oluşturup ikisi arasındaki doğrusal korelasyonu net bir şekilde gözlemleyebiliriz.

```
d2l.set_figsize()
# İki nokta üstüste sadece gösterim amaçlıdır
d2l.plt.scatter(features[:, 1].detach().numpy(), labels.detach().numpy(), 1);
```



3.2.2 Veri Kümesini Okuma

Model eğitimlerinin, veri kümesi üzerinde birden çok geçiş yapmaktan, her seferde bir minigrup örnek almaktan ve bunları modelimizi güncellemek için kullanmaktan oluştuğunu hatırlayın. Bu süreç, makine öğrenmesi algoritmalarını eğitmek için çok temel olduğundan, veri kümesini karıştırmak ve ona minigruplar halinde erişmek için bir yardımcı işlev tanımlamaya değer.

Aşağıdaki kodda, bu işlevselliğin olası bir uygulamasını göstermek için `data_iter` işlevini tanımlıyoruz. Fonksiyon, bir grup boyutunu, bir öznitelik matrisini ve bir etiket vektörünü alarak `batch_size` boyutundaki minigrupları verir. Her bir minigrup, bir dizi öznitelik ve etiketten oluşur.

```
def data_iter(batch_size, features, labels):
    num_examples = len(features)
    indices = list(range(num_examples))
    # Örnekler belirli bir sıra gözetmeksizin rastgele okunur
    random.shuffle(indices)
    for i in range(0, num_examples, batch_size):
        batch_indices = torch.tensor(
            indices[i: min(i + batch_size, num_examples)])
        yield features[batch_indices], labels[batch_indices]
```

Genel olarak, paralelleştirme işlemlerinde mükemmel olan GPU donanımından yararlanmak için makul boyutta minigruplar kullanmak istediğimizi unutmayın. Her örnek, modellerimiz üzerinden paralel olarak beslenebildiği ve her örnek için kayıp fonksiyonunun gradyanı da paralel olarak alınabildiğinden, GPU'lar, yüzlerce örneği yalnızca tek bir örneği işlemek için gerekebileceğinden çok daha az kısa sürede işlememize izin verir.

Biraz sezgi oluşturmak için, ilk olarak küçük bir grup veri örneği okuyup yazdıralım. Her minigruptaki özniteliklerin şekli bize hem minigrup boyutunu hem de girdi özniteliklerinin sayısını söyler. Aynı şekilde, minigrubumuzun etiketleri de `batch_size` ile verilen şekle sahip olacaktır.

```
batch_size = 10

for X, y in data_iter(batch_size, features, labels):
    print(X, '\n', y)
    break
```

```
tensor([[ 1.4869,  1.5494],
        [-0.8690, -0.0986],
        [ 1.5624,  0.0090],
        [-0.4172,  0.9719],
        [ 0.3500, -0.1156],
        [-0.3433,  2.4511],
        [-1.1783, -1.2873],
        [-1.1068, -0.5236],
        [ 0.8041,  0.2262],
        [-2.2314,  0.0541]])
 tensor([[ 1.9146],
        [ 2.7896],
        [ 7.2889],
        [ 0.0441],
        [ 5.3029],
        [-4.8102],
        [ 6.2084],
        [ 3.7547],
        [ 5.0364],
        [-0.4410]])
```

Yinelemeyi çalıştırırken, tüm veri kümesi tükenene kadar art arda farklı minigruplar elde ederiz (bunu deneyin). Yukarıda uygulanan yineleme, eğitici amaçlar için iyi olsa da, gerçek problemlerde bizim başımızı belaya sokacak şekilde verimsizdir. Örneğin, tüm verileri belleğe yüklememizi ve çok sayıda rastgele bellek erişimi gerçekleştirmemizi gerektirir. Derin öğrenme çerçevesinde uygulanan yerleşik yineleyiciler önemli ölçüde daha verimlidir ve hem dosyalarda depolanan verilerle hem de veri akışları aracılığıyla beslenen verilerle ilgilenebilirler.

3.2.3 Model Parametrelerini İlkleme

Modelimizin parametrelerini minigrup rasgele gradyan inişiyle optimize etmeye başlamadan önce, ilk olarak bazı parametrelere ihtiyacımız var. Aşağıdaki kodda, ağırlıkları, ortalaması 0 ve standart sapması 0.01 olan normal bir dağılımdan rasgele sayılar örnekleyerek ve ek girdiyi 0 olarak ayarlayarak ilkliyoruz.

```
w = torch.normal(0, 0.01, size=(2,1), requires_grad=True)
b = torch.zeros(1, requires_grad=True)
```

Parametrelerimizi ilkledikten sonra, bir sonraki görevimiz, verilerimize yeterince iyi uyum sağlayana kadar onları güncellemektir. Her güncelleme, parametrelere göre kayıp fonksiyonumuzun gradyanını almayı gerektirir. Gradyan verildiğinde, her parametreyi kaybı azaltabilecek yönde güncelleyebiliriz.

Hiç kimse gradyanları açıkça hesaplamak istemediğinden (bu sıkıcı ve hataya açıktır), gradyanı hesaplamak için Section 2.5 içinde tanıtıldığı gibi otomatik türev almayı kullanırız.

3.2.4 Modeli Tanımlama

Daha sonra, modelimizi, onun girdileri ve parametreleri çıktıları ile ilişkilendirerek tanımlamalıyız. Doğrusal modelin çıktısını hesaplamak için, $\mathbf{X}$ girdi özniteliklerinin ve $\mathbf{w}$ model ağırlıklarının matris vektör nokta çarpımını alıp her bir örneğe b ek girdisini eklediğimizi hatırlayın. Aşağıda $\mathbf{Xw}$ bir vektör ve b bir skalerdir. Yayma mekanizmasının şurada açıklandığı anımsayalım Section 2.1.3. Bir vektör ve bir skaleri topladığımızda, skaler vektörün her bileşenine eklenir.

```
def linreg(X, w, b):  #@save
    """Doğrusal regresyon modeli."""
    return torch.matmul(X, w) + b
```

3.2.5 Kayıp Fonksiyonunu Tanımlama

Modelimizi güncellemek, kayıp fonksiyonumuzun gradyanını almayı gerektirdiğinden, önce kayıp fonksiyonunu tanımlamalıyız. Burada kare kayıp fonksiyonunu şurada açıklandığı, Section 3.1, gibi kullanacağız . Uygulamada, `y` gerçek değerini tahmin edilen değer `y_hat` şekline dönüştürmemiz gerekir. Aşağıdaki işlev tarafından döndürülen sonuç da `y_hat` ile aynı şekle sahip olacaktır.

```
def squared_loss(y_hat, y):  #@save
    """Kare kayıp."""
    return (y_hat - y.reshape(y_hat.shape)) ** 2 / 2
```

3.2.6 Optimizasyon Algoritmasını Tanımlama

Section 3.1 içinde tartıştığımız gibi, doğrusal regresyon kapalı biçim bir çözüme sahiptir. Ancak, bu doğrusal regresyon hakkında bir kitap değil: Derin öğrenme hakkında bir kitap. Bu kitabın tanıttığı diğer modellerin hiçbiri analitik olarak çözülemediğinden, bu fırsatı minigrup rasgele gradyan inişinin ilk çalışan örneğini tanıtmak için kullanacağız.

Her adımda, veri kümemizden rastgele alınan bir minigrup kullanarak, parametrelerimize göre kaybın gradyanını tahmin edeceğiz. Daha sonra kayıpları azaltabilecek yönde parametrelerimizi güncelleyeceğiz. Aşağıdaki kod, bir küme parametre, bir öğrenme oranı ve bir grup boyutu verildiğinde minigrup rasgele gradyan iniş güncellemesini uygular. Güncelleme adımının boyutu, öğrenme oranı `lr` tarafından belirlenir. Kaybımız, örneklerin minigrubu üzerinden bir toplam olarak hesaplandığından, adım boyutumuzu grup boyutuna (`batch_size`) göre normalleştiririz, böylece tipik bir adım boyutunun büyüklüğü, grup boyutu seçimimize büyük ölçüde bağlı olmaz.

```
def sgd(params, lr, batch_size):  #@save
    """Minigrup rasgele gradyan inişi."""
    with torch.no_grad():
        for param in params:
            param -= lr * param.grad / batch_size
            param.grad.zero_()
```

3.2.7 Eğitim

Artık tüm parçaları yerine koyduğumuza göre, ana eğitim döngüsünü uygulamaya hazırız. Bu kodu anlamanız çok önemlidir çünkü derin öğrenmede kariyeriniz boyunca neredeyse aynı eğitim döngülerini tekrar tekrar göreceksiniz.

Her yinelemede, bir minigrup eğitim örneği alacağız ve bir dizi tahmin elde etmek için bunları modelimizden geçireceğiz. Kaybı hesapladıktan sonra, her parametreye göre gradyanları depolayarak ağ üzerinden geriye doğru geçişi başlatırız. Son olarak, model parametrelerini güncellemek için optimizasyon algoritması sgd'yi çağıracağız.

Özetle, aşağıdaki döngüyü uygulayacağız:

- $(\mathbf{w}, b)$ parametrelerini ilkletin.
- Tamamlanana kadar tekrarlayın
  - Gradyanı hesaplayın: $\mathbf{g} \leftarrow \partial_{(\mathbf{w},b)} \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} l(\mathbf{x}^{(i)}, y^{(i)}, \mathbf{w}, b)$
  - Parametreleri güncelleyin: $(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta \mathbf{g}$

Her bir *dönemde (epoch)*, eğitim veri kümesindeki her örnekten geçtikten sonra (örneklerin sayısının grup boyutuna bölünebildiği varsayılarak) tüm veri kümesini (`data_iter` işlevini kullanarak) yineleyeceğiz. Dönemlerin sayısı, `num_epochs`, ve öğrenme hızı, `lr`, burada sırasıyla 3 ve 0.03 olarak belirlediğimiz hiper parametrelerdir. Ne yazık ki, hiper parametrelerin belirlenmesi zordur ve deneme yanılma yoluyla bazı ayarlamalar gerektirir. Bu ayrıntıları şimdilik atlıyoruz, ancak daha sonra Chapter 11 içinde tekrarlayacağız.

```
lr = 0.03
num_epochs = 3
net = linreg
loss = squared_loss
```

```
for epoch in range(num_epochs):
    for X, y in data_iter(batch_size, features, labels):
        l = loss(net(X, w, b), y)  # `X` ve `y`'deki minigrup kaybı
        # [`w`, `b`]'e göre `l` üzerindeki gradyanı hesaplayın
        l.sum().backward()
        sgd([w, b], lr, batch_size)  # Parametreleri gradyanlarına göre güncelle
    with torch.no_grad():
        train_l = loss(net(features, w, b), labels)
        print(f'epoch {epoch + 1}, loss {float(train_l.mean()):f}')
```

```
epoch 1, loss 0.035539
epoch 2, loss 0.000133
epoch 3, loss 0.000050
```

Bu durumda, veri kümemizi kendimiz sentezlediğimiz için, gerçek parametrelerin ne olduğunu tam olarak biliyoruz. Böylece eğitimdeki başarımızı, gerçek parametreleri eğitim döngümüz aracılığıyla öğrendiklerimizle karşılaştırarak değerlendirebiliriz. Gerçekten de birbirlerine çok yakın oldukları ortaya çıkıyor.

```
print(f'w tahminindeki hata: {true_w - w.reshape(true_w.shape)}')
print(f'b tahminindeki hata: {true_b - b}')
```

```
w tahminindeki hata: tensor([0.0002, 0.0002], grad_fn=<SubBackward0>)
b tahminindeki hata: tensor([-4.7684e-07], grad_fn=<RsubBackward1>)
```

Parametreleri mükemmel bir şekilde elde ettiğimizi kabullenmememiz gerektiğini unutmayın. Bununla birlikte, makine öğrenmesinde, genellikle temeldeki gerçek parametreleri elde etmek ile daha az ilgileniriz ve yüksek derecede doğru tahminlere yol açan parametrelerle daha çok ilgileniriz. Neyse ki, zorlu optimizasyon problemlerinde bile, rasgele gradyan inişi, kısmen derin ağlar için, oldukça doğru tahmine götüren parametrelerin birçok konfigürasyonunun mevcut olmasından dolayı, genellikle dikkate değer ölçüde iyi çözümler bulabilir.

3.2.8 Özet

- Derin bir ağın, katmanları veya süslü optimize edicileri tanımlamaya gerek kalmadan sadece tensörler ve otomatik türev alma kullanarak nasıl sıfırdan uygulanabileceğini ve optimize edilebileceğini gördük.
- Bu bölüm sadece mümkün olanın yüzeyine ışık tutar. İlerideki bölümlerde, yeni tanıttığımız kavramlara dayalı yeni modelleri açıklayacak ve bunları daha kısaca nasıl uygulayacağımızı öğreneceğiz.

3.2.9 Alıştırmalar

1. Ağırlıkları sıfıra ilkleseydik ne olurdu? Algoritma yine de çalışır mıydı?
2. Voltaj ve akım arasında bir model bulmaya çalışan Georg Simon Ohm[49] olduğunuzu varsayın. Modelinizin parametrelerini öğrenmek için otomatik türev almayı kullanabilir misiniz?
3. Spektral enerji yoğunluğunu kullanarak bir nesnenin sıcaklığını belirlemek için Planck Yasası[50]'nı kullanabilir misiniz?
4. İkinci türevleri hesaplamak isterseniz karşılaşabileceğiniz sorunlar nelerdir? Onları nasıl düzeltirsiniz?
5. `squared_loss` işlevinde `reshape` işlevi neden gereklidir?
6. Kayıp işlevi değerinin ne kadar hızlı düştüğünü bulmak için farklı öğrenme oranları kullanarak deney yapın.
7. Örneklerin sayısı parti boyutuna bölünemezse, `data_iter` işlevinin davranışına ne olur?

Tartışmalar[51]

[49] https://en.wikipedia.org/wiki/Georg_Ohm
[50] https://en.wikipedia.org/wiki/Planck%27s_law
[51] https://discuss.d2l.ai/t/43

3.3 Doğrusal Regresyonunun Kısa Uygulaması

Son birkaç yıldır derin öğrenmeye olan geniş ve yoğun ilgi, gradyan tabanlı öğrenme algoritmalarını uygulamanın tekrarlayan iş yükünü otomatikleştirmek için şirketler, akademisyenler ve amatör geliştiricilere çeşitli olgun açık kaynak çerçeveleri geliştirmeleri için ilham verdi. Section 3.2 içinde, biz sadece (i) veri depolama ve doğrusal cebir için tensörlere; ve (ii) gradyanları hesaplamak için otomatik türev almaya güvendik. Pratikte, veri yineleyiciler, kayıp işlevleri, optimize ediciler ve sinir ağı katmanları çok yaygın olduğu için, modern kütüphaneler bu bileşenleri bizim için de uygular.

Bu bölümde, derin öğrenme çerçevelerinin üst düzey API'lerini kullanarak Section 3.2 içindeki doğrusal regresyon modelini kısaca nasıl uygulayacağınızı göstereceğiz.

3.3.1 Veri Kümesini Oluşturma

Başlamak için, şuradaki aynı veri kümesini oluşturacağız: Section 3.2.

```
import numpy as np
import torch
from torch.utils import data
from d2l import torch as d2l
```

```
true_w = torch.tensor([2, -3.4])
true_b = 4.2
features, labels = d2l.synthetic_data(true_w, true_b, 1000)
```

3.3.2 Veri Kümesini Okuma

Kendi yineleyicimizi döndürmek yerine, verileri okumak için bir çerçevedeki mevcut API'yi çağırabiliriz. `features` (öznitelikleri tutan değişken) ve `labels`'i (etiketleri tutan değişken) bağımsız değişken olarak iletiriz ve bir veri yineleyici nesnesi başlatırken `batch_size` (grup boyutu) belirtiriz. Ayrıca, mantıksal veri tipi (boolean) değeri `is_train`, veri yineleyici nesnesinin her bir dönemdeki (epoch) verileri karıştırmasını isteyip istemediğimizi gösterir (veri kümesinden geçer).

```
def load_array(data_arrays, batch_size, is_train=True):  #@save
    """Bir PyTorch veri yineleyici oluşturun."""
    dataset = data.TensorDataset(*data_arrays)
    return data.DataLoader(dataset, batch_size, shuffle=is_train)
```

```
batch_size = 10
data_iter = load_array((features, labels), batch_size)
```

Şimdi, `data_iteri`, `data_iter` işlevini Section 3.2 içinde çağırdığımız şekilde kullanabiliriz. Çalıştığını doğrulamak için, örneklerin ilk minigrubunu okuyabilir ve yazdırabiliriz. Section 3.2 içindeki ile karşılaştırıldığında, burada bir Python yineleyici oluşturmak için `iter` kullanıyoruz ve yineleyiciden ilk öğeyi elde etmek için `next`'i kullanıyoruz.

```
next(iter(data_iter))
```

```
[tensor([[-0.7894, -0.4236],
         [-0.1704, -1.6098],
         [-1.3614,  1.1222],
         [-0.3556,  0.7254],
         [-1.0136, -0.1632],
         [ 1.0528,  1.5194],
         [ 2.1396, -2.2619],
         [-0.2321, -0.7793],
         [ 1.4198, -1.2292],
         [ 1.3337, -1.1340]]),
 tensor([[ 4.0635],
         [ 9.3159],
         [-2.3210],
         [ 1.0009],
         [ 2.7228],
         [ 1.1503],
         [16.1817],
         [ 6.3618],
         [11.2123],
         [10.7317]])]
```

3.3.3 Modeli Tanımlama

Doğrusal regresyonu Section 3.2 içinde sıfırdan uyguladığımızda, model parametrelerimizi açık bir şekilde tanımladık ve temel doğrusal cebir işlemlerini kullanarak çıktı üretmek için hesaplamalarımızı kodladık. Bunu nasıl yapacağınızı *bilmelisiniz*. Ancak modelleriniz daha karmaşık hale geldiğinde ve bunu neredeyse her gün yapmanız gerektiğinde, bu yardım için memnun olacaksınız. Durum, kendi blogunuzu sıfırdan kodlamaya benzer. Bunu bir veya iki kez yapmak ödüllendirici ve öğreticidir, ancak bir bloga her ihtiyaç duyduğunuzda tekerleği yeniden icat etmek için bir ay harcarsanız kötü bir web geliştiricisi olursunuz.

Standart işlemler için, uygulamaya (kodlamaya) odaklanmak yerine özellikle modeli oluşturmak için kullanılan katmanlara odaklanmamızı sağlayan bir çerçevenin önceden tanımlanmış katmanlarını kullanabiliriz. Önce, `Sequential` (ardışık, sıralı) sınıfının bir örneğini ifade edecek `net` (ağ) model değişkenini tanımlayacağız. `Sequential` sınıfı, birbirine zincirlenecek birkaç katman için bir kapsayıcı (container) tanımlar. Girdi verileri verildiğinde, `Sequential` bir örnek, bunu birinci katmandan geçirir, ardından onun çıktısını ikinci katmanın girdisi olarak geçirir ve böyle devam eder. Aşağıdaki örnekte, modelimiz yalnızca bir katmandan oluşuyor, bu nedenle gerçekten `Sequential` örneğe ihtiyacımız yok. Ancak, gelecekteki modellerimizin neredeyse tamamı birden fazla katman içereceği için, sizi en standart iş akışına alıştırmak için yine de kullanacağız.

Tek katmanlı bir ağın mimarisini şurada gösterildiği gibi hatırlayın: Fig. 3.1.2. Katmanın *tamamen bağlı* olduğu söylenir, çünkü girdilerinin her biri, bir matris-vektör çarpımı yoluyla çıktılarının her birine bağlanır.

PyTorch'ta, tam bağlantılı katman `Linear` sınıfında tanımlanır. `nn.Linear`'e iki bağımsız değişken aktardığımıza dikkat edin. Birincisi, 2 olan girdi öznitelik boyutunu belirtir ve ikincisi, tek bir skaler olan ve dolayısıyla 1 olan çıktı öznitelik boyutudur.

```
# `nn` sinir ağları için kısaltmadır
from torch import nn

net = nn.Sequential(nn.Linear(2, 1))
```

3.3.4 Model Parametrelerini İlkleme

net'i kullanmadan önce, doğrusal regresyon modelindeki ağırlıklar ve ek girdi gibi model parametrelerini ilkletmemiz gerekir. Derin öğrenme çerçeveleri genellikle parametreleri ilklemek için önceden tanımlanmış bir yola sahiptir. Burada, her ağırlık parametresinin, ortalama 0 ve standart sapma 0.01 ile normal bir dağılımdan rastgele örneklenmesi gerektiğini belirtiyoruz. Ek girdi parametresi sıfır olarak ilklenecektir.

nn.Linear oluştururken girdi ve çıktı boyutlarını belirttik, şimdi, onların ilk değerlerini belirtmek için parametrelere doğrudan erişebiliriz. İlk olarak ağdaki ilk katmanı net[0] ile buluruz ve ardından parametrelere erişmek için weight.data ve bias.data yöntemlerini kullanırız. Daha sonra, parametre değerlerinin üzerine yazmak için normal_ ve fill_ değiştirme yöntemlerini kullanırız.

```
net[0].weight.data.normal_(0, 0.01)
net[0].bias.data.fill_(0)
```

```
tensor([0.])
```

3.3.5 Kayıp Fonksiyonunu Tanımlama

MSELoss sınıfı, ortalama hata karesini hesaplar ((3.1.5) içindeki $1/2$ çarpanı olmadan). Varsayılan olarak, örneklerdeki ortalama kaybı döndürür.

```
loss = nn.MSELoss()
```

3.3.6 Optimizasyon Algoritmasını Tanımlama

Minigrup rasgele gradyan inişi, sinir ağlarını optimize etmek için standart bir araçtır ve bu nedenle PyTorch, bunu Trainer sınıfı aracılığıyla bu algoritmadaki bir dizi varyasyonla birlikte destekler. SGD'den örnek yarattığımızda, optimize edilecek parametreleri (net.collect_params() aracılığıyla net modelimizden elde edilebilir) ve optimizasyon algoritmamızın gerektirdiği hiper parametreleri bir sözlük ile (dictionary veri yapısı) belirteceğiz. Minigrup rasgele gradyan inişi, burada 0.03 olarak ayarlanan lr (öğrenme oranı) değerini ayarlamamızı gerektirir.

```
trainer = torch.optim.SGD(net.parameters(), lr=0.03)
```

3.3.7 Eğitim

Modelimizi derin öğrenme çerçevesinin yüksek seviyeli API'leri aracılığıyla ifade etmenin nispeten birkaç satır kod gerektirdiğini fark etmiş olabilirsiniz. Parametreleri ayrı ayrı tahsis etmemiz, kayıp fonksiyonumuzu tanımlamamız veya minigrup rasgele gradyan inişini uygulamamız gerekmedi. Çok daha karmaşık modellerle çalışmaya başladığımızda, üst düzey API'lerin avantajları önemli ölçüde artacaktır. Bununla birlikte, tüm temel parçaları bir kez yerine getirdiğimizde, eğitim döngüsünün kendisi, her şeyi sıfırdan uygularken yaptığımıza çarpıcı bir şekilde benzer.

Hafızanızı yenilemek için: Bazı dönemler (epoch) için, veri kümesinin (`train_data`) üzerinden eksiksiz bir geçiş yapacağız ve yinelemeli olarak bir minigrup girdiyi ve ilgili referans gerçek değer etiketleri alacağız. Her minigrup için aşağıdaki ritüeli uyguluyoruz:

- `net(X)`'i çağırarak tahminler oluşturun ve `l` kaybını (ileriye doğru yayma) hesaplayın.
- Geri yaymayı çalıştırarak gradyanları hesaplayın.
- Optimize edicimizi çağırarak model parametrelerini güncelleyin.

İyi bir önlem olarak, her dönemden sonra kaybı hesaplıyor ve ilerlemeyi izlemek için yazdırıyoruz.

```
num_epochs = 3
for epoch in range(num_epochs):
    for X, y in data_iter:
        l = loss(net(X) ,y)
        trainer.zero_grad()
        l.backward()
        trainer.step()
    l = loss(net(features), labels)
    print(f'donem {epoch + 1}, kayip {l:f}')
```

```
donem 1, kayip 0.000174
donem 2, kayip 0.000093
donem 3, kayip 0.000093
```

Aşağıda, sonlu veri üzerinde eğitimle öğrenilen model parametrelerini ve veri kümemizi oluşturan gerçek parametreleri karşılaştırıyoruz. Parametrelere erişmek için önce ihtiyacımız olan katmana `net`'ten erişiyoruz ve sonra bu katmanın ağırlıklarına ve ek girdilerine erişiyoruz. Sıfırdan uygulamamızda olduğu gibi, tahmin edilen parametrelerimizin gerçek referans değerlere yakın olduğuna dikkat edin.

```
w = net[0].weight.data
print('w tahmin hatasi:', true_w - w.reshape(true_w.shape))
b = net[0].bias.data
print('b tahmin hatasi:', true_b - b)
```

```
w tahmin hatasi: tensor([ 2.4557e-05, -2.3890e-04])
b tahmin hatasi: tensor([2.3842e-06])
```

3.3.8 Özet

- PyTorch'un üst düzey API'lerini kullanarak modelleri çok daha kısa bir şekilde uygulayabiliriz.
- PyTorch'ta, `data` modülü veri işleme için araçlar sağlar, `nn` modülü çok sayıda sinir ağı katmanını ve genel kayıp işlevlerini tanımlar.
- Değerlerini _ ile biten yöntemlerle değiştirerek parametreleri ilkleyebiliriz.

3.3.9 Alıştırmalar

1. Eğer `nn.MSELoss()`ı `nn.MSELoss(reduction='sum')` ile değiştirirsek, kodun aynı şekilde davranması için öğrenme oranını nasıl değiştirebiliriz? Neden?
2. Hangi kayıp işlevlerinin ve ilkleme yöntemlerinin sağlandığını görmek için PyTorch belgelerini inceleyin. Kaybı Huber kaybıyla yer değiştirin.
3. `net[0].weight`'in gradyanına nasıl erişirsiniz?

Tartışmalar[52]

3.4 Eşiksiz En Büyük İşlev Bağlanımı (Softmaks Regresyon)

Section 3.1 içinde, doğrusal bağlanımı tanıttık, Section 3.2 içinde sıfırdan uygulamalar üzerinde çalıştık ve ağır işi yapmak için Section 3.3 içinde derin öğrenme çerçevesinin yüksek seviyeli API'lerini tekrar kullandık.

Bağlanım, *ne kadar?* veya *kaç?* sorularını yanıtlamak istediğimizde ulaştığımız araçtır. Bir evin kaç dolara satılacağını (fiyatı) veya bir beyzbol takımının kazanabileceği galibiyet sayısını veya bir hastanın taburcu edilmeden önce hastanede kalacağı gün sayısını tahmin etmek istiyorsanız, o zaman muhtemelen bir regresyon modeli arıyorsunuz.

Uygulamada, daha çok *sınıflandırma* ile ilgileniyoruz. "Ne kadar" değil, "hangisi" sorusunu sorarak:

- Bu e-posta spam klasörüne mi yoksa gelen kutusuna mı ait?
- Bu müşterinin bir abonelik hizmetine *kaydolma* mı veya *kaydolmama* mı olasılığı daha yüksek?
- Bu resim bir eşeği, köpeği, kediyi veya horozu tasvir ediyor mu?
- Aston'ın bir sonraki izleyeceği film hangisi?

Konuşma dilinde, makine öğrenmesi uygulayıcıları iki incelikle farklı sorunu tanımlamak için *sınıflandırma* kelimesini aşırı yüklüyorlar: (i) Yalnızca örneklerin kategorilere (sınıflara) zor olarak atanmasıyla ilgilendiğimiz konular ve (ii) yumuşak atamalar yapmak istediğimiz yerler, yani her bir kategorinin geçerli olma olasılığını değerlendirme. Ayrım, kısmen bulanıklaşma eğilimindedir, çünkü çoğu zaman, yalnızca zor görevleri önemsediğimizde bile, yine de yumuşak atamalar yapan modeller kullanıyoruz.

[52] https://discuss.d2l.ai/t/45

3.4.1 Sınıflandırma Problemi

Ayaklarımızı ısındırmak için basit bir imge sınıflandırma problemiyle başlayalım. Buradaki her girdi 2×2 gri tonlamalı bir imgeden oluşur. Her piksel değerini tek bir skaler ile temsil edebiliriz ve bu da bize dört öznitelik, x_1, x_2, x_3, x_4, verir. Ayrıca, her imgenin "kedi", "tavuk" ve "köpek" kategorilerinden birine ait olduğunu varsayalım.

Daha sonra, etiketleri nasıl temsil edeceğimizi seçmeliyiz. İki bariz seçeneğimiz var. Belki de en doğal dürtü, tam sayıların sırasıyla $\{\text{köpek}, \text{kedi}, \text{tavuk}\}$ temsil eden $\{1, 2, 3\}$ içinden y'yi seçmek olacaktır. Bu, bu tür bilgileri bir bilgisayarda *saklamanın* harika bir yoludur. Kategoriler arasında doğal bir sıralama varsa, örneğin $\{\text{bebek}, \text{yürümeye başlayan çocuk}, \text{ergen}, \text{genç yetişkin}, \text{yetişkin}, \text{yaşlı}\}$ tahmin etmeye çalışıyor olsaydık, bu sorunu bağlanım olarak kabul etmek ve etiketleri bu biçimde tutmak mantıklı bile olabilirdi.

Ancak genel sınıflandırma sorunları, sınıflar arasında doğal sıralamalarla gelmez. Neyse ki, istatistikçiler uzun zaman önce kategorik verileri göstermenin basit bir yolunu keşfettiler: *Bire bir kodlama*. Bire bir kodlama, kategorilerimiz kadar bileşen içeren bir vektördür. Belirli bir örneğin kategorisine karşılık gelen bileşen 1'e ve diğer tüm bileşenler 0'a ayarlanmıştır. Bizim durumumuzda, y etiketi üç boyutlu bir vektör olacaktır: $(1, 0, 0)$ - "kedi", $(0, 1, 0)$ - "tavuk" ve $(0, 0, 1)$ - "köpek":

$$y \in \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}. \tag{3.4.1}$$

3.4.2 Ağ Mimarisi

Tüm olası sınıflarla ilişkili koşullu olasılıkları tahmin etmek için, sınıf başına bir tane olmak üzere birden çok çıktıya sahip bir modele ihtiyacımız var. Doğrusal modellerle sınıflandırmayı ifade etmek için, çıktılarımız kadar sayıda afin (affine) fonksiyona ihtiyacımız olacak. Her çıktı kendi afin işlevine karşılık gelecektir. Bizim durumumuzda, 4 özniteliğimiz ve 3 olası çıktı kategorimiz olduğundan, ağırlıkları temsil etmek için 12 sayıla (indeksli w) ve ek girdileri temsil etmek için 3 sayıla (indeksli b) ihtiyacımız olacak. Her girdi için şu üç *logiti*, o_1, o_2 ve o_3, hesaplıyoruz:

$$\begin{aligned} o_1 &= x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1, \\ o_2 &= x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2, \\ o_3 &= x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3. \end{aligned} \tag{3.4.2}$$

Bu hesaplamayı Fig. 3.4.1 içinde gösterilen sinir ağı diyagramı ile tasvir edebiliriz. Doğrusal regresyonda olduğu gibi, softmaks regresyon da tek katmanlı bir sinir ağıdır. Ayrıca her çıktının, o_1, o_2 ve o_3, hesaplanması, tüm girdilere, x_1, x_2, x_3 ve x_4, bağlı olduğundan, eşiksiz en büyük işlev bağlanımının (softmaks regresyonunun) çıktı katmanı da tamamen bağlı katman olarak tanımlanır.

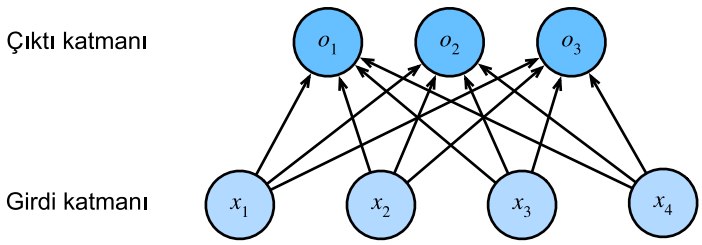


Fig. 3.4.1: Eşiksiz en büyük işlev bağlanımı bir tek katmanlı sinir ağıdır.

Modeli daha öz bir şekilde ifade etmek için doğrusal cebir gösterimini kullanabiliriz. Vektör biçiminde, hem matematik hem de kod yazmak için daha uygun bir biçim olan $\mathbf{o} = \mathbf{W}\mathbf{x} + \mathbf{b}$'ye ulaşıyoruz. Tüm ağırlıklarımızı bir 3×4 matriste topladığımızı ve belirli bir $\mathbf{x}$ veri örneğinin öznitelikleri için, çıktılarımızın bizim öznitelikler girdimiz ile ağırlıklarımızın bir matris-vektör çarpımı artı ek girdilerimiz $\mathbf{b}$ olarak verildiğini unutmayın.

3.4.3 Tam Bağlantı Katmanların Parametrelendirme Maliyeti

Sonraki bölümlerde göreceğimiz gibi, derin öğrenmede tam bağlı katmanlar her yerde bulunur. Ancak, adından da anlaşılacağı gibi, tam bağlı katmanlar, potansiyel olarak birçok öğrenilebilir parametreyle *tamamen* bağlantılıdır. Özellikle, d girdileri ve q çıktıları olan herhangi bir tam bağlı katman için, parametreleştirme maliyeti $\mathcal{O}(dq)$'dır ve bu pratikte aşırı derecede yüksek olabilir. Neyse ki, d girdilerini q çıktılarına dönüştürmenin bu maliyeti $\mathcal{O}(\frac{dq}{n})$'a düşürülebilir, burada n hiper parametresi bizim tarafımızdan gerçek dünya uygulamalarında, parametre tasarrufu ve model etkinliği arasındaki dengeyi sağlamak için esnek bir şekilde belirtilebilir (Zhang *et al.*, 2021).

3.4.4 Eşiksiz En Büyük İşlev İşlemi

Burada ele alacağımız ana yaklaşım, modelimizin çıktılarını olasılıklar olarak yorumlamaktır. Gözlemlenen verilerin olabilirliğini en üst düzeye çıkaran olasılıkları üretmek için parametrelerimizi optimize edeceğiz (eniyileceğiz). Ardından, tahminler üretmek için bir eşik belirleyeceğiz, örneğin maksimum tahmin edilen olasılığa sahip etiketi seçeceğiz.

Biçimsel olarak ifade edersek, herhangi bir $\hat{y}_j$ çıktısının belirli bir öğenin j sınıfına ait olma olasılığı olarak yorumlanmasını istiyoruz. Sonra en büyük çıktı değerine sahip sınıfı tahminimiz $\operatorname{argmax}_j y_j$ olarak seçebiliriz. Örneğin, $\hat{y}_1$, $\hat{y}_2$ ve $\hat{y}_3$ sırasıyla 0.1, 0.8 ve 0.1 ise, o zaman (örneğimizde) "tavuğu" temsil eden kategori 2'yi tahmin ederiz.

Logit o'yu doğrudan ilgilendiğimiz çıktılarımız olarak yorumlamamızı önermek isteyebilirsiniz. Bununla birlikte, doğrusal katmanın çıktısının doğrudan bir olasılık olarak yorumlanmasında bazı sorunlar vardır. Bir yandan, hiçbir şey bu sayıların toplamını 1'e sınırlamıyor. Diğer yandan, girdilere bağlı olarak negatif değerler alabilirler. Bunlar, Section 2.6 içinde sunulan temel olasılık aksiyomlarını ihlal ediyor.

Çıktılarımızı olasılıklar olarak yorumlamak için, (yeni verilerde bile) bunların negatif olmayacağını ve toplamlarının 1 olacağını garanti etmeliyiz. Dahası, modeli gerçeğe uygun olasılıkları tahmin etmeye teşvik eden bir eğitim amaç fonksiyonuna ihtiyacımız var. Bir sınıflandırıcı tüm örneklerden 0.5 çıktısını verdiğinde, bu örneklerin yarısının gerçekte tahmin edilen sınıfa ait olacağını umuyoruz. Bu, *kalibrasyon* adı verilen bir özelliktir.

1959'da sosyal bilimci R. Duncan Luce tarafından *seçim modelleri* bağlamında icat edilen *eşiksiz en büyük işlevi (softmaks)* tam olarak bunu yapar. Logitlerimizi negatif olmayacak ve toplamı 1 olacak şekilde dönüştürmek için, modelin türevlenebilir kalmasını gerekliyken, önce her logitin üssünü alırız (negatif olmamasını sağlar) ve sonra toplamlarına böleriz (toplamlarının 1 olmasını sağlar):

$$\hat{\mathbf{y}} = \operatorname{softmax}(\mathbf{o}) \quad \text{öyle ki} \quad \hat{y}_j = \frac{\exp(o_j)}{\sum_k \exp(o_k)}. \tag{3.4.3}$$

Tüm j için $\hat{y}_1 + \hat{y}_2 + \hat{y}_3 = 1$'i, $0 \leq \hat{y}_j \leq 1$ ile görmek kolaydır. Dolayısıyla, $\hat{\mathbf{y}}$, eleman değerleri uygun şekilde yorumlanabilen uygun bir olasılık dağılımıdır. Softmaks işleminin, her sınıfa

atanan olasılıkları belirleyen basit softmaks-öncesi değerler olan **o** logitleri arasındaki sıralamayı değiştirmediğini unutmayın. Bu nedenle, tahmin sırasında yine en olası sınıfı seçebiliriz:

$$\operatorname*{argmax}_j \hat{y}_j = \operatorname*{argmax}_j o_j. \tag{3.4.4}$$

Softmaks doğrusal olmayan bir fonksiyon olmasına rağmen, softmaks regresyonunun çıktıları hala girdi özniteliklerinin afin dönüşümü ile *belirlenir*; dolayısıyla, softmaks regresyon doğrusal bir modeldir.

3.4.5 Minigruplar için Vektörleştirme

Hesaplama verimliliğini artırmak ve GPU'lardan yararlanmak için, genellikle veri minigrupları için vektör hesaplamaları yapıyoruz. Öznitelik boyutsallığı (girdi sayısı) d ve parti boyutu n içeren bir minigrup $\mathbf{X}$ verildiğini varsayalım. Üstelik, çıktıda q kategorimizin olduğunu varsayalım. Sonra minigrup öznitelikleri $\mathbf{X}$, $\mathbb{R}^{n\times d}$ içinde, ağırlıkları $\mathbf{W} \in \mathbb{R}^{d\times q}$ ve ek girdiyi, $\mathbf{b} \in \mathbb{R}^{1\times q}$ olarak gösterir.

$$\begin{aligned} \mathbf{O} &= \mathbf{X}\mathbf{W} + \mathbf{b}, \\ \hat{\mathbf{Y}} &= \operatorname{softmax}(\mathbf{O}). \end{aligned} \tag{3.4.5}$$

Bu, baskın işlemi, her seferinde bir örnek işlersek yürüteceğimiz matris-vektör çarpımlarına karşı $\mathbf{X}\mathbf{W}$ matris-matris çarpımını, hızlandırır. $\mathbf{X}$ içindeki her satır bir veri örneğini temsil ettiğinden, softmaks işleminin kendisi *satır bazında* hesaplanabilir: Her $\mathbf{O}$ satırı için, tüm girdilerin üssünü alın ve sonra bunları toplayarak normalleştirin. (3.4.5) içindeki $\mathbf{X}\mathbf{W} + \mathbf{b}$ toplamı sırasında yayınlamayı tetikleriz, hem minigrup logitleri $\mathbf{O}$ hem de çıktı olasılıkları $\hat{\mathbf{Y}}$, $n \times q$ matrislerdir.

3.4.6 Kayıp (Yitim) İşlevi

Sonrasında, tahmin edilen olasılıklarımızın kalitesini ölçmek için bir kayıp fonksiyonuna ihtiyacımız var. Doğrusal regresyonda ortalama hata karesi amaç fonksiyonu için olasılıksal bir gerekçelendirme sağlarken karşılaştığımız kavramla aynı olan maksimum olabilirlik tahminine güveneceğiz (Section 3.1.3).

Log-Olabilirlik

Softmaks işlevi bize $\hat{\mathbf{y}}$ vektörünü verir, bunu herhangi bir $\mathbf{x}$ girdisi verildiğinde her bir sınıfın tahmini koşullu olasılıkları olarak yorumlayabiliriz, ör. $\hat{y}_1 = P(y = \text{kedi} \mid \mathbf{x})$. $\{\mathbf{X}, \mathbf{Y}\}$ veri kümesinin tamamınında, i indekslenen $\mathbf{x}^{(i)}$ örneğini ve ilgili bire-bir etiket vektörü $\mathbf{y}^{(i)}$'yi içeren n örneğe sahip olduğunu varsayalım. Öznitelikleri göz önüne alındığında, gerçek sınıfların modelimize göre ne kadar olası olduğunu kontrol ederek tahminleri gerçekle karşılaştırabiliriz:

$$P(\mathbf{Y} \mid \mathbf{X}) = \prod_{i=1}^{n} P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}). \tag{3.4.6}$$

Maksimum olabilirlik tahminine göre, negatif log olabilirliği en aza indirmeye eşdeğer olan $P(\mathbf{Y} \mid \mathbf{X})$'i maksimize ediyoruz:

$$-\log P(\mathbf{Y} \mid \mathbf{X}) = \sum_{i=1}^{n} -\log P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}) = \sum_{i=1}^{n} l(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}), \tag{3.4.7}$$

y ve q sınıflarının üzerindeki $\hat{\mathbf{y}}$ model tahmini etiket çifti için, kayıp fonksiyonu l'dir.

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^{q} y_j \log \hat{y}_j. \tag{3.4.8}$$

Daha sonra açıklanacak nedenlerden ötürü, (3.4.8) içindeki kayıp işlevi genellikle *çapraz entropi kaybı* olarak adlandırılır. **y**, q uzunluğunda bire bir vektör olduğundan, j tüm koordinatlarının toplamı, bir terim hariç tümü için kaybolur. Tüm $\hat{y}_j$'ler tahmini olasılıklar olduğundan, logaritmaları hiçbir zaman 0'dan büyük olmaz. Sonuç olarak, gerçek etiketi *kesinlik* ile doğru bir şekilde tahmin edersek, yani gerçek etiket **y** için tahmini olasılık $P(\mathbf{y} \mid \mathbf{x}) = 1$ ise, kayıp fonksiyonu daha fazla küçültülemez. Bunun genellikle imkansız olduğunu unutmayın. Örneğin, veri kümesinde etiket gürültüsü olabilir (bazı örnekler yanlış etiketlenmiş olabilir). Girdi öznitelikleri her örneği mükemmel bir şekilde sınıflandırmak için yeterince bilgilendirici olmadığında da mümkün olmayabilir.

Softmaks ve Türevleri

Softmaks ve karşılık gelen kayıp fonksiyonu çok yaygın olduğundan, nasıl hesaplandığını biraz daha iyi anlamaya değerdir. (3.4.3) içindeki kayıp tanımına, (3.4.8) eklersek ve softmaks tanımını kullanırsak:

$$\begin{aligned} l(\mathbf{y}, \hat{\mathbf{y}}) &= -\sum_{j=1}^{q} y_j \log \frac{\exp(o_j)}{\sum_{k=1}^{q} \exp(o_k)} \\ &= \sum_{j=1}^{q} y_j \log \sum_{k=1}^{q} \exp(o_k) - \sum_{j=1}^{q} y_j o_j \\ &= \log \sum_{k=1}^{q} \exp(o_k) - \sum_{j=1}^{q} y_j o_j. \end{aligned} \tag{3.4.9}$$

Neler olduğunu biraz daha iyi anlamak için, herhangi bir logit o_j ile ilgili türevi düşünün. Böylece şunu görürüz:

$$\partial_{o_j} l(\mathbf{y}, \hat{\mathbf{y}}) = \frac{\exp(o_j)}{\sum_{k=1}^{q} \exp(o_k)} - y_j = \text{softmax}(\mathbf{o})_j - y_j. \tag{3.4.10}$$

Başka bir deyişle, türev, softmax işlemiyle ifade edildiği gibi modelimiz tarafından atanan olasılık ile bire-bir etiket vektöründeki öğeler tarafından ifade edildiği gibi gerçekte ne olduğu arasındaki farktır. Bu anlamda, regresyonda gördüğümüze çok benzerdir; gradyan y gözlemi ile $\hat{y}$ tahmini arasındaki farktır. Bu bir tesadüf değil. Herhangi bir üstel aile (bkz. dağılımlar üzerine çevrimiçi ek[53]) modelinde, log-olabilirlik gradyanları tam olarak bu terim tarafından verilmektedir. Bu gerçek, gradyanları hesaplamayı pratikte kolaylaştırır.

[53] https://tr.d2l.ai/chapter_appendix-mathematics-for-deep-learning/distributions.html

Çapraz Entropi Kaybı

Şimdi, sadece tek bir sonucu değil, sonuçlara göre bütün bir dağılımı gözlemlediğimiz durumu düşünün. **y** etiketi için önceki ile aynı gösterimi kullanabiliriz. Tek fark, $(0, 0, 1)$ gibi sadece ikilik girdiler içeren bir vektör yerine, artık genel bir olasılık vektörüne sahibiz, örneğin $(0.1, 0.2, 0.7)$. Daha önce (3.4.8) içinde l kaybını tanımlamak için kullandığımız matematik hala iyi çalışıyor, sadece yorumu biraz daha genel. Burada etiketler üzerindeki dağılım için beklenen kayıp değeridir. Bu kayıp, *çapraz entropi kaybı* olarak adlandırılır ve sınıflandırma problemlerinde en sık kullanılan kayıplardan biridir. Bilgi teorisinin sadece temellerini tanıtarak ismin gizemini çözebiliriz. Bilgi teorisinin daha fazla detayını anlamak isterseniz, bilgi teorisi üzerine çevrimiçi ek[54]'e başvurabilirsiniz.

3.4.7 Bilgi Teorisinin Temelleri

Bilgi teorisi, bilgiyi (veri olarak da bilinir) mümkün olduğunca kısa bir biçimde kodlama, kod çözme, iletme ve kullanma sorunuyla ilgilenir.

Entropi

Bilgi teorisindeki ana fikir, verilerdeki bilgi içeriğini ölçmektir. Bu miktar, verileri sıkıştırma yeteneğimize sıkı bir sınır koyar. Bilgi teorisinde, bu miktar bir P dağılımının *entropisi* olarak adlandırılır ve aşağıdaki denklem ile ifade edilir:

$$H[P] = \sum_j -P(j) \log P(j). \tag{3.4.11}$$

Bilgi teorisinin temel teoremlerinden biri, P dağılımından rastgele çekilen verileri kodlamak için, en az $H[P]$ "nats"a ihtiyacımız olduğunu belirtir. "Nat" nedir diye merak ediyorsanız, 2 tabanlı bir kod yerine e tabanlı bir kod kullanıldığında bu bit ile eşdeğerdir, bir nat $\frac{1}{\log(2)} \approx 1,44$ bittir.

Şaşırtıcılık

Sıkıştırmanın tahminle ne ilgisi olduğunu merak ediyor olabilirsiniz. Sıkıştırmak istediğimiz bir veri akışımız olduğunu hayal edin. Bir sonraki belirteci (token) tahmin etmek bizim için her zaman kolaysa, bu verinin sıkıştırılması da kolaydır! Akıştaki her belirtecin her zaman aynı değeri aldığı uç örneği ele alalım. Bu çok sıkıcı bir veri akışı! Ayrıca sadece sıkıcı değil, aynı zamanda tahmin etmesi de kolay. Her zaman aynı olduklarından, akışın içeriğini iletmek için herhangi bir bilgi iletmemiz gerekmez. Tahmin etmesi kolay, sıkıştırması kolay.

Halbuki, her olayı tam olarak tahmin edemezsek, o zaman bazen şaşırabiliriz. Bir olaya daha düşük bir olasılık atadığımızda sürprizimiz daha da büyük olur. Claude Shannon (öznel) bir olasılık, $P(j)$, atamış olan bir j olayı gözlemlemenin *şaşırtıcı* olduğunu göstermek için $\log \frac{1}{P(j)} = -\log P(j)$'yi hesapladı. (3.4.11) içinde tanımlanan entropi, veri oluşturma süreciyle gerçekten eşleşen doğru olasılıklar atandığında *beklenen şaşırtıcılıktır*.

[54] https://tr.d2l.ai/chapter_appendix-mathematics-for-deep-learning/information-theory.html

Öyleyse, entropi gerçek olasılığı bilen birinin yaşadığı sürpriz seviyesiyse, merak ediyor olabilirsiniz, çapraz entropi nedir? $H(P, Q)$ olarak gösterilen P ile Q *arasındaki* çapraz entropi, öznel olasılıkları Q olan bir gözlemcinin gerçekte P olasılıklarına göre oluşturulan verileri gördükten sonra beklenen şaşkınlığıdır. Olası en düşük çapraz entropi $P = Q$ olduğunda elde edilir. Bu durumda, P ile Q arasındaki çapraz entropi $H(P, P) = H(P)$ şeklindedir.

Kısacası, çapraz entropi sınıflandırma amaç fonksiyonunu iki şekilde düşünebiliriz: (i) Gözlemlenen verilerin olabilirliğini maksimize etmek ve (ii) etiketleri iletmek için gerekli olan şaşkınlığımızı (ve dolayısıyla bit sayısını) en aza indirmek.

3.4.8 Model Tahmini ve Değerlendirme

Softmaks regresyon modelini eğittikten sonra, herhangi bir örnek öznitelik verildiğinde, her bir çıktı sınıfının olasılığını tahmin edebiliriz. Normalde, en yüksek tahmin edilen olasılığa sahip sınıfı çıktı sınıfı olarak kullanırız. Gerçek sınıfla (etiket ile) tutarlıysa tahmin doğrudur. Sonraki deney bölümünde, modelin performansını değerlendirmek için *doğruluk oranını* (kısaca doğruluk) kullanacağız. Bu, doğru tahmin sayısı ile toplam tahmin sayısı arasındaki orana eşittir.

3.4.9 Özet

- Softmaks işlemi bir vektör alır ve onu olasılıklara eşler.
- Softmaks regresyonu, sınıflandırma problemleri için geçerlidir. Softmaks işlemi çıktı sınıfının olasılık dağılımını kullanır.
- Çapraz entropi, iki olasılık dağılımı arasındaki farkın iyi bir ölçüsüdür. Modelimize verilen verileri kodlamak için gereken bit sayısını ölçer.

3.4.10 Alıştırmalar

1. Üstel aileler ile softmaks arasındaki bağlantıyı biraz daha derinlemesine inceleyebiliriz.
    1. Softmaks için çapraz entropi kaybının, $l(\mathbf{y}, \hat{\mathbf{y}})$, ikinci türevini hesaplayın.
    2. softmax($\mathbf{o}$) tarafından verilen dağılımın varyansını hesaplayın ve yukarıda hesaplanan ikinci türevle eşleştiğini gösterin.
2. Eşit olasılıkla ortaya çıkan üç sınıfımız olduğunu varsayın, yani olasılık vektörü $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.
    1. Bunun için bir ikilik kod tasarlamaya çalışırsak ne sorun olur?
    2. Daha iyi bir kod tasarlayabilir misiniz? İpucu: İki bağımsız gözlemi kodlamaya çalışırsak ne olur? Ya n tane gözlemi birlikte kodlarsak?
3. Softmaks, yukarıda anlatılan eşleme için yanlış bir isimdir (ancak derin öğrenmede herkes bunu kullanır). Gerçek softmaks, RealSoftMax$(a, b) = \log(\exp(a) + \exp(b))$ olarak tanımlanır.
    1. RealSoftMax$(a, b) > \max(a, b)$ olduğunu kanıtlayın.
    2. Bunun $\lambda > 0$ olması koşuluyla λ^{-1}RealSoftMax$(\lambda a, \lambda b)$ için geçerli olduğunu kanıtlayın.

3. $\lambda \to \infty$ için $\lambda^{-1}\text{RealSoftMax}(\lambda a, \lambda b) \to \max(a, b)$ olduğunu gösterin.
4. Soft-min neye benzer?
5. Bunu ikiden fazla sayı için genişletin.

Tartışmalar[55]

3.5 İmge Sınıflandırma Veri Kümesi

İmge sınıflandırması için yaygın olarak kullanılan veri kümelerinden biri MNIST veri kümesidir (LeCun *et al.*, 1998). Bir kıyaslama veri kümesi olarak iyi bir çalışma gerçekleştirmiş olsa da, günümüz standartlarına göre basit modeller bile %95'in üzerinde sınıflandırma doğruluğu elde ettiğinden daha güçlü modeller ile daha zayıf olanları ayırt etmek için uygun değildir. Bugün, MNIST bir kıyaslama ölçütü olmaktan çok makullük (sanity) kontrolü işlevi görüyor. Biraz daha ileriye gitmek için, önümüzdeki bölümlerdeki tartışmamızı niteliksel olarak benzer, ancak nispeten karmaşık olan 2017'de piyasaya sürülen Fashion-MNIST veri kümesine odaklayacağız (Xiao *et al.*, 2017).

```
%matplotlib inline
import torch
import torchvision
from torch.utils import data
from torchvision import transforms
from d2l import torch as d2l

d2l.use_svg_display()
```

3.5.1 Veri Kümesini Okuma

Fashion-MNIST veri kümesini çerçevemizdeki yerleşik işlevler aracılığıyla indirebilir ve belleğe okuyabiliriz.

```
# 'ToTensor', imge verilerini PIL türünden 32 bit kayan virgüllü sayı tensörlerine
# dönüştürür. Tüm sayıları 255'e böler, böylece tüm piksel değerleri 0 ile 1 arasında olur.
trans = transforms.ToTensor()
mnist_train = torchvision.datasets.FashionMNIST(
    root="../data", train=True, transform=trans, download=True)
mnist_test = torchvision.datasets.FashionMNIST(
    root="../data", train=False, transform=trans, download=True)
```

Fashion-MNIST, her biri eğitim veri kümesinde 6000 görsel ve test veri kümesinde 1000 görsel ile temsil edilen 10 kategorideki görsellerden oluşur. Eğitim için değil, model performansını değerlendirmek için bir *test veri kümesi* (veya *test kümesi*) kullanılır.

```
len(mnist_train), len(mnist_test)
```

```
(60000, 10000)
```

[55] https://discuss.d2l.ai/t/46

Her girdi imgesinin yüksekliği ve genişliği 28 pikseldir. Veri kümesinin, kanal sayısı 1 olan gri tonlamalı görsellerden oluştuğuna dikkat edin. Kısaca, bu kitapta yüksekliği h genişliği w piksel olan herhangi bir imgenin şekli $h \times w$ veya (h, w)'dir.

```
mnist_train[0][0].shape
```

```
torch.Size([1, 28, 28])
```

Fashion-MNIST'teki görseller şu kategorilerle ilişkilidir: Tişört, pantolon, kazak, elbise, ceket, sandalet, gömlek, spor ayakkabı, çanta ve ayak bileği hizası bot. Aşağıdaki işlev, sayısal etiket indeksleri ve metindeki adları arasında dönüştürme yapar.

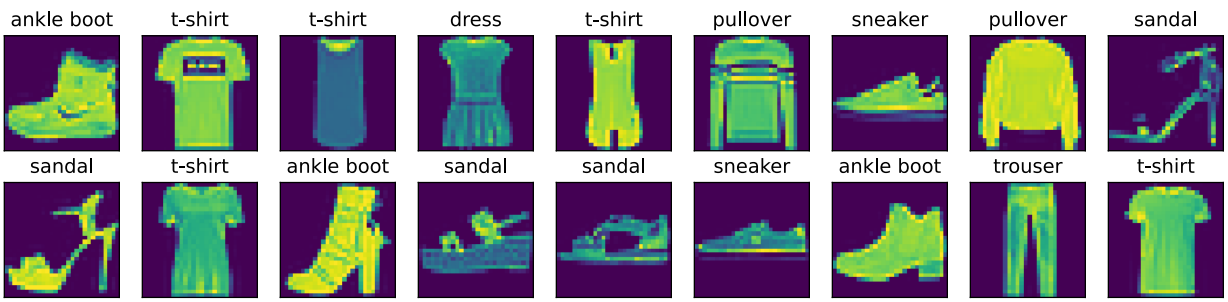
```
def get_fashion_mnist_labels(labels):  #@save
    """Fashion-MNIST veri kümesi için metin etiketleri döndürün."""
    text_labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
                   'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
    return [text_labels[int(i)] for i in labels]
```

Şimdi bu örnekleri görselleştirmek için bir işlev oluşturabiliriz.

```
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):  #@save
    """Görsellerin bir listesini çizin"""
    figsize = (num_cols * scale, num_rows * scale)
    _, axes = d2l.plt.subplots(num_rows, num_cols, figsize=figsize)
    axes = axes.flatten()
    for i, (ax, img) in enumerate(zip(axes, imgs)):
        if torch.is_tensor(img):
            # Tensor Image
            ax.imshow(img.numpy())
        else:
            # PIL Image
            ax.imshow(img)
        ax.axes.get_xaxis().set_visible(False)
        ax.axes.get_yaxis().set_visible(False)
        if titles:
            ax.set_title(titles[i])
    return axes
```

Eğitim veri kümesindeki ilk birkaç örnek için görseller ve bunlara karşılık gelen etiketler (metin olarak) aşağıdadır.

```
X, y = next(iter(data.DataLoader(mnist_train, batch_size=18)))
show_images(X.reshape(18, 28, 28), 2, 9, titles=get_fashion_mnist_labels(y));
```



3.5.2 Minigrup Okuma

Eğitim ve test kümelerinden okurken hayatımızı kolaylaştırmak için sıfırdan bir tane oluşturmak yerine yerleşik veri yineleyiciyi kullanıyoruz. Her yinelemede, bir yineleyicinin her seferinde grup (`batch_size`) boyutundaki bir veri minigrubunu okuduğunu hatırlayın. Ayrıca eğitim verisi yineleyicisi için örnekleri rastgele karıştırıyoruz.

```
batch_size = 256

def get_dataloader_workers():  #@save
    """Verileri okumak için 4 işlem kullanın."""
    return 4

train_iter = data.DataLoader(mnist_train, batch_size, shuffle=True,
                             num_workers=get_dataloader_workers())
```

Eğitim verilerini okurken geçen süreye bakalım.

```
timer = d2l.Timer()
for X, y in train_iter:
    continue
f'{timer.stop():.2f} sec'
```

```
'2.25 sec'
```

3.5.3 Her Şeyi Bir Araya Getirme

Şimdi Fashion-MNIST veri kümesini alan ve okuyan `load_data_fashion_mnist` fonksiyonunu tanımlıyoruz. Hem eğitim kümesi hem de geçerleme kümesi için veri yineleyicileri döndürür. Ek olarak, imgeleri başka bir şekle yeniden boyutlandırmak için isteğe bağlı bir argüman kabul eder.

```
def load_data_fashion_mnist(batch_size, resize=None):  #@save
    """Fashion-MNIST veri kümesini indirin ve ardından belleğe yükleyin."""
    trans = [transforms.ToTensor()]
    if resize:
        trans.insert(0, transforms.Resize(resize))
    trans = transforms.Compose(trans)
    mnist_train = torchvision.datasets.FashionMNIST(
        root="../data", train=True, transform=trans, download=True)
    mnist_test = torchvision.datasets.FashionMNIST(
        root="../data", train=False, transform=trans, download=True)
    return (data.DataLoader(mnist_train, batch_size, shuffle=True,
                            num_workers=get_dataloader_workers()),
            data.DataLoader(mnist_test, batch_size, shuffle=False,
                            num_workers=get_dataloader_workers()))
```

Aşağıda, `resize` bağımsız değişkenini belirterek `load_data_fashion_mnist` işlevinin görseli yeniden boyutlandırma özelliğini test ediyoruz.

```
train_iter, test_iter = load_data_fashion_mnist(32, resize=64)
for X, y in train_iter:
    print(X.shape, X.dtype, y.shape, y.dtype)
    break
```

```
torch.Size([32, 1, 64, 64]) torch.float32 torch.Size([32]) torch.int64
```

Artık ilerleyen bölümlerde Fashion-MNIST veri kümesiyle çalışmaya hazırız.

3.5.4 Özet

- Fashion-MNIST, 10 kategoriyi temsil eden resimlerden oluşan bir giyim sınıflandırma veri kümesidir. Bu veri kümesini, çeşitli sınıflandırma algoritmalarını değerlendirmek için sonraki bölümlerde kullanacağız.
- Yüksekliği h genişliği w piksel olan herhangi bir imgenin şeklini $h \times w$ veya (h, w) olarak saklarız.
- Veri yineleyiciler, verimli performans için önemli bir bileşendir. Eğitim döngünüzü yavaşlatmaktan kaçınmak için yüksek performanslı hesaplamalardan yararlanan iyi uygulanmış veri yineleyicilerine güvenin.

3.5.5 Alıştırmalar

1. `batch_size` değerini (örneğin 1'e) düşürmek okuma performansını etkiler mi?
2. Veri yineleyici performansı önemlidir. Mevcut uygulamanın yeterince hızlı olduğunu düşünüyor musunuz? İyileştirmek için çeşitli seçenekleri keşfediniz.
3. Çerçevenin çevrimiçi API belgelerine bakın. Başka hangi veri kümeleri mevcuttur?

Tartışmalar[56]

3.6 Sıfırdan Softmaks Regresyon Uygulaması Yaratma

Sıfırdan lineer regresyon uyguladığımız gibi, softmax regresyonunun da benzer şekilde temel olduğuna ve kanlı ayrıntılarını ve nasıl kendinizin uygulanacağını bilmeniz gerektiğine inanıyoruz. Section 3.5 içinde yeni eklenen Fashion-MNIST veri kümesiyle çalışacağız, grup boyutu 256 olan bir veri yineleyicisi kuracağız.

```
import torch
from IPython import display
from d2l import torch as d2l
```

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

[56] https://discuss.d2l.ai/t/49

3.6.1 Model Parametrelerini İlkletme

Doğrusal regresyon örneğimizde olduğu gibi, buradaki her örnek sabit uzunlukta bir vektörle temsil edilecektir. Ham veri kümesindeki her örnek 28×28'lik görseldir. Bu bölümde, her bir görseli 784 uzunluğundaki vektörler olarak ele alarak düzleştireceğiz. Gelecekte, görsellerdeki uzamsal yapıyı kullanmak için daha karmaşık stratejilerden bahsedeceğiz, ancak şimdilik her piksel konumunu yalnızca başka bir öznitelik olarak ele alıyoruz.

Softmaks regresyonunda, sınıflar kadar çıktıya sahip olduğumuzu hatırlayın. Veri kümemiz 10 sınıf içerdiğinden, ağımızın çıktı boyutu 10 olacaktır. Sonuç olarak, ağırlıklarımız 784×10 matrisi ve ek girdiler 1×10 satır vektörü oluşturacaktır. Doğrusal regresyonda olduğu gibi, `W` ağırlıklarımızı Gauss gürültüsüyle ve ek girdilerimizi ilk değerini 0 alacak şekilde başlatacağız.

```
num_inputs = 784
num_outputs = 10

W = torch.normal(0, 0.01, size=(num_inputs, num_outputs), requires_grad=True)
b = torch.zeros(num_outputs, requires_grad=True)
```

3.6.2 Softmaks İşlemini Tanımlama

Softmaks regresyon modelini uygulamadan önce, toplam operatörünün bir tensörde belirli boyutlar boyunca nasıl çalıştığını Section 2.3.6 ve Section 2.3.6 içinde anlatıldığı gibi kısaca gözden geçirelim. Bir `X` matrisi verildiğinde, tüm öğeleri (varsayılan olarak) veya yalnızca aynı eksendeki öğeleri toplayabiliriz, örneğin aynı sütun (eksen 0) veya aynı satır (eksen 1) üzerinden. `X` (2, 3) şeklinde bir tensör ise ve sütunları toplarsak, sonucun (3,) şeklinde bir vektör olacağını unutmayın. Toplam operatörünü çağırırken, üzerinde topladığımız boyutu daraltmak yerine esas tensördeki eksen sayısını korumayı belirtebiliriz. Bu, (1, 3) şeklinde iki boyutlu bir tensörle sonuçlanacaktır.

```
X = torch.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
X.sum(0, keepdim=True), X.sum(1, keepdim=True)
```

```
(tensor([[5., 7., 9.]]),
 tensor([[ 6.],
         [15.]]))
```

Artık softmaks işlemini uygulamaya hazırız. Softmaks'ın üç adımdan oluştuğunu hatırlayın: (i) Her terimin üssünü alıyoruz (`exp` kullanarak); (ii) her bir örnek için normalleştirme sabitini elde etmek için her satırı topluyoruz (toplu işte (batch) örnek başına bir satırımız vardır); (iii) her satırı normalleştirme sabitine bölerek sonucun toplamının 1 olmasını sağlıyoruz. Koda bakmadan önce, bunun bir denklem olarak nasıl ifade edildiğini hatırlayalım:

$$\text{softmax}(\mathbf{X})_{ij} = \frac{\exp(\mathbf{X}_{ij})}{\sum_k \exp(\mathbf{X}_{ik})}. \tag{3.6.1}$$

Payda veya normalleştirme sabiti bazen *bölmeleme fonksiyonu* olarak da adlandırılır (ve logaritmasına log-bölmeleme fonksiyonu denir). Bu ismin kökenleri, ilgili bir denklemin bir parçacıklar topluluğu üzerindeki dağılımı modellediği istatistiksel fizik[57]'tedir.

[57] https://en.wikipedia.org/wiki/Partition_function_(statistical_mechanics)

```
def softmax(X):
    X_exp = torch.exp(X)
    partition = X_exp.sum(1, keepdim=True)
    return X_exp / partition  # Burada yayin mekanizmasi uygulanir
```

Gördüğünüz gibi, herhangi bir rastgele girdi için, her bir öğeyi negatif olmayan bir sayıya dönüştürüyoruz. Ayrıca, olasılık belirtmek için gerektiği gibi, her satırın toplamı 1'dir.

```
X = torch.normal(0, 1, (2, 5))
X_prob = softmax(X)
X_prob, X_prob.sum(1)
```

```
(tensor([[0.0155, 0.2640, 0.5381, 0.1014, 0.0810],
         [0.7024, 0.0739, 0.0257, 0.0696, 0.1284]]),
 tensor([1.0000, 1.0000]))
```

Bu matematiksel olarak doğru görünse de, uygulamamızda biraz özensiz davrandık çünkü matrisin büyük veya çok küçük öğeleri nedeniyle sayısal taşma (overflow) veya küçümenliğe (underflow) karşı önlem alamadık.

3.6.3 Modeli Tanımlama

Artık softmaks işlemini tanımladığımıza göre, softmaks regresyon modelini uygulayabiliriz. Aşağıdaki kod, girdinin ağ üzerinden çıktıya nasıl eşlendiğini tanımlar. Verileri modelimizden geçirmeden önce, `reshape` işlevini kullanarak toplu işteki (batch) her esas görseli bir vektör halinde düzleştirdiğimize dikkat edin.

```
def net(X):
    return softmax(torch.matmul(X.reshape((-1, W.shape[0])), W) + b)
```

3.6.4 Kayıp Fonksiyonunu Tanımlama

Daha sonra, Section 3.4 içinde tanıtıldığı gibi, çapraz entropi kaybı işlevini uygulamamız gerekir. Bu, tüm derin öğrenmede en yaygın kayıp işlevi olabilir, çünkü şu anda sınıflandırma sorunları regresyon sorunlarından çok daha fazladır.

Çapraz entropinin, gerçek etikete atanan tahmin edilen olasılığın negatif log-olabilirliğini aldığını hatırlayın. Bir Python for-döngüsü (verimsiz olma eğilimindedir) ile tahminler üzerinde yinelemek yerine, tüm öğeleri tek bir operatörle seçebiliriz. Aşağıda, 3 sınıf üzerinde tahmin edilen olasılıkların 2 örneğini ve bunlara karşılık gelen `y` etiketleriyle `y_hat` örnek verilerini oluşturuyoruz. `y` ile, ilk örnekte birinci sınıfın doğru tahmin olduğunu biliyoruz ve ikinci örnekte üçüncü sınıf temel referans değerdir. `y`'yi `y_hat` içindeki olasılıkların indeksleri olarak kullanarak, ilk örnekte birinci sınıfın olasılığını ve ikinci örnekte üçüncü sınıfın olasılığını seçiyoruz.

Aşağıda, 3 sınıf üzerinden tahmin edilen olasılıkların 2 örneğini içeren bir oyuncak verisi `y_hat`'i oluşturuyoruz. Ardından birinci örnekte birinci sınıfın olasılığını ve ikinci örnekte üçüncü sınıfın olasılığını seçiyoruz.

```
y = torch.tensor([0, 2])
y_hat = torch.tensor([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
y_hat[[0, 1], y]
```

```
tensor([0.1000, 0.5000])
```

Artık, çapraz entropi kaybı işlevini tek bir kod satırı ile verimli bir şekilde uygulayabiliriz.

```
def cross_entropy(y_hat, y):
    return - torch.log(y_hat[range(len(y_hat)), y])

cross_entropy(y_hat, y)
```

```
tensor([2.3026, 0.6931])
```

3.6.5 Sınıflandırma Doğruluğu

Tahmin edilen olasılık dağılımı `y_hat` göz önüne alındığında, genellikle kesin bir tahmin vermemiz gerektiğinde tahmin edilen en yüksek olasılığa sahip sınıfı seçeriz. Aslında, birçok uygulama bir seçim yapmamızı gerektirir. Gmail, bir e-postayı "Birincil", "Sosyal", "Güncellemeler" veya "Forumlar" olarak sınıflandırmalıdır. Olasılıkları dahili olarak tahmin edebilir, ancak günün sonunda sınıflar arasından birini seçmesi gerekir.

Tahminler `y` etiket sınıfıyla tutarlı olduğunda doğrudur. Sınıflandırma doğruluğu, doğru olan tüm tahminlerin oranıdır. Doğruluğu doğrudan optimize etmek zor olabilse de (türevleri alınamaz), genellikle en çok önemsediğimiz performans ölçütüdür ve sınıflandırıcıları eğitirken neredeyse her zaman onu rapor edeceğiz.

Doğruluğu hesaplamak için aşağıdakileri yapıyoruz. İlk olarak, `y_hat` bir matris ise, ikinci boyutun her sınıf için tahmin puanlarını sakladığını varsayıyoruz. Her satırdaki en büyük girdi için dizine göre tahmin edilen sınıfı elde ederken `argmax` kullanırız. Ardından, tahmin edilen sınıfı gerçek referans değer `y` ile karşılaştırırız. Eşitlik operatörü == veri türlerine duyarlı olduğundan, `y_hat` veri türünü `y` ile eşleşecek şekilde dönüştürürüz. Sonuç, 0 (yanlış) ve 1 (doğru) girişlerini içeren bir tensördür. Toplamlarını almak doğru tahminlerin sayısını verir.

```
def accuracy(y_hat, y):  #@save
    """Doğru tahminlerin sayısını hesaplayın."""
    if len(y_hat.shape) > 1 and y_hat.shape[1] > 1:
        y_hat = y_hat.argmax(axis=1)
    cmp = y_hat.type(y.dtype) == y
    return float(cmp.type(y.dtype).sum())
```

Önceden tanımlanan `y_hat` ve `y` değişkenlerini sırasıyla tahmin edilen olasılık dağılımları ve etiketler olarak kullanmaya devam edeceğiz. İlk örneğin tahmin sınıfının 2 olduğunu görebiliriz (satırın en büyük öğesi dizin 2 ile 0.6'dır), bu gerçek etiket, 0 ile tutarsızdır. İkinci örneğin tahmin sınıfı 2'dir (satırın en büyük öğesi 2 endeksi ile 0.5'tir) ve bu gerçek etiket 2 ile tutarlıdır. Bu nedenle, bu iki örnek için sınıflandırma doğruluk oranı 0.5'tir.

```
accuracy(y_hat, y) / len(y)
```

```
0.5
```

Benzer şekilde, veri yineleyici `data_iter` aracılığıyla erişilen bir veri kümesindeki herhangi bir `net` modelinin doğruluğunu hesaplayabiliriz.

```
def evaluate_accuracy(net, data_iter):  #@save
    """Bir veri kumesinde bir modelin doğruluğunu hesaplayın."""
    if isinstance(net, torch.nn.Module):
        net.eval()  # Modeli değerlendirme moduna kurun
    metric = Accumulator(2)  # Doğru tahmin sayısı, tahmin sayısı

    with torch.no_grad():
        for X, y in data_iter:
            metric.add(accuracy(net(X), y), y.numel())
    return metric[0] / metric[1]
```

Burada, `Accumulator`, birden çok değişken üzerindeki toplamları biriktirmek için bir yardımcı sınıftır. Yukarıdaki `evaluate_accuracy` işlevinde, `Accumulator` örneğinde sırasıyla hem doğru tahminlerin sayısını hem de tahminlerin sayısını depolamak için 2 değişken oluştururuz. Veri kümesini yineledikçe her ikisi de zaman içinde birikecektir.

```
class Accumulator:  #@save
    """`n` değişken üzerinden toplamları biriktirmek için"""
    def __init__(self, n):
        self.data = [0.0] * n

    def add(self, *args):
        self.data = [a + float(b) for a, b in zip(self.data, args)]

    def reset(self):
        self.data = [0.0] * len(self.data)

    def __getitem__(self, idx):
        return self.data[idx]
```

`net` modelini rastgele ağırlıklarla başlattığımız için, bu modelin doğruluğu rastgele tahmin etmeye yakın olmalıdır, yani 10 sınıf için 0.1 gibi.

```
evaluate_accuracy(net, test_iter)
```

```
0.0979
```

3.6.6 Eğitim

Softmaks regresyonu için eğitim döngüsü, Section 3.2 içindeki doğrusal regresyon uygulamamızı okursanız, çarpıcı bir şekilde tanıdık gelebilir. Burada uygulamayı yeniden kullanılabilir hale getirmek için yeniden düzenliyoruz. İlk olarak, bir dönemi (epoch) eğitmek için bir işlev tanımlıyoruz. `updater`'in, grup boyutunu bağımsız değişken olarak kabul eden, model parametrelerini güncellemek için genel bir işlev olduğuna dikkat edin. `d2l.sgd` işlevinin bir sarmalayıcısı (wrapper) veya bir çerçevenin yerleşik optimizasyon işlevi olabilir.

```
def train_epoch_ch3(net, train_iter, loss, updater):  #@save
    """Bölüm 3'te tanımlanan eğitim döngüsü."""
    # Modeli eğitim moduna kurun
    if isinstance(net, torch.nn.Module):
        net.train()
    # Eğitim kaybı toplamı, eğitim doğruluğu toplamı, örnek sayısı
    metric = Accumulator(3)
    for X, y in train_iter:
        # Gradyanları hesaplayın ve parametreleri güncelleyin
        y_hat = net(X)
        l = loss(y_hat, y)
        if isinstance(updater, torch.optim.Optimizer):
            # PyTorch yerleşik optimize edicisini ve kayıp kriterini kullanma
            updater.zero_grad()
            l.mean().backward()
            updater.step()
        else:
            # Özel olarak oluşturulmuş optimize edicisini ve kayıp ölçütünü kullanma
            l.sum().backward()
            updater(X.shape[0])
        metric.add(float(l.sum()), accuracy(y_hat, y), y.numel())
    # Eğitim kaybını ve doğruluğunu döndür
    return metric[0] / metric[2], metric[1] / metric[2]
```

Eğitim işlevinin uygulamasını göstermeden önce, verileri animasyonda (canlandırma) çizen bir yardımcı program sınıfı tanımlıyoruz. Yine kitabın geri kalanında kodu basitleştirmeyi amaçlamaktadır.

```
class Animator:  #@save
    """Animasyonda veri çizdirme"""
    def __init__(self, xlabel=None, ylabel=None, legend=None, xlim=None,
                 ylim=None, xscale='linear', yscale='linear',
                 fmts=('-', 'm--', 'g-.', 'r:'), nrows=1, ncols=1,
                 figsize=(3.5, 2.5)):
        # Çoklu çizgileri artarak çizdir
        if legend is None:
            legend = []
        d2l.use_svg_display()
        self.fig, self.axes = d2l.plt.subplots(nrows, ncols, figsize=figsize)
        if nrows * ncols == 1:
            self.axes = [self.axes, ]
        # Argumanları elde tutmak için bir lambda işlevi kullan
        self.config_axes = lambda: d2l.set_axes(
            self.axes[0], xlabel, ylabel, xlim, ylim, xscale, yscale, legend)
        self.X, self.Y, self.fmts = None, None, fmts
```

(continues on next page)

(continued from previous page)

```
    def add(self, x, y):
        # Çoklu veri noktalarını şekile ekleyin
        if not hasattr(y, "__len__"):
            y = [y]
        n = len(y)
        if not hasattr(x, "__len__"):
            x = [x] * n
        if not self.X:
            self.X = [[] for _ in range(n)]
        if not self.Y:
            self.Y = [[] for _ in range(n)]
        for i, (a, b) in enumerate(zip(x, y)):
            if a is not None and b is not None:
                self.X[i].append(a)
                self.Y[i].append(b)
        self.axes[0].cla()
        for x, y, fmt in zip(self.X, self.Y, self.fmts):
            self.axes[0].plot(x, y, fmt)
        self.config_axes()
        display.display(self.fig)
        display.clear_output(wait=True)
```

Aşağıdaki eğitim işlevi daha sonra, `num_epochs` ile belirtilen birden çok dönem için `train_iter` aracılığıyla erişilen bir eğitim veri kümesinde bir `net` modeli eğitir. Her dönemin sonunda model, `test_iter` aracılığıyla erişilen bir test veri kümesinde değerlendirilir. Eğitimin ilerlemesini görselleştirmek için `Animator` sınıfından yararlanacağız.

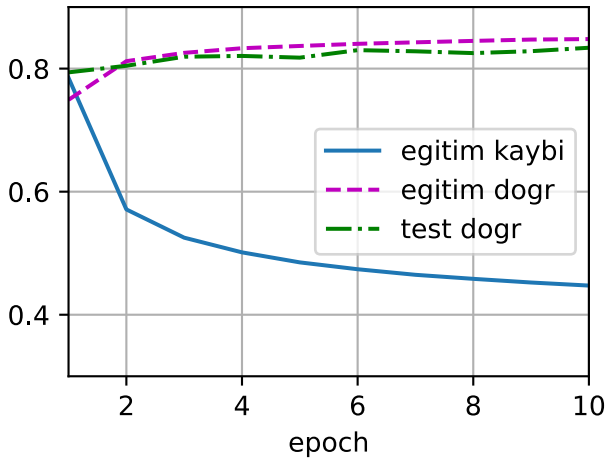
```
def train_ch3(net, train_iter, test_iter, loss, num_epochs, updater):  #@save
    """Bir modeli eğitin (Bölüm 3'te tanımlanmıştır)."""
    animator = Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0.3, 0.9],
                        legend=['egitim kaybi', 'egitim dogr', 'test dogr'])
    for epoch in range(num_epochs):
        train_metrics = train_epoch_ch3(net, train_iter, loss, updater)
        test_acc = evaluate_accuracy(net, test_iter)
        animator.add(epoch + 1, train_metrics + (test_acc,))
    train_loss, train_acc = train_metrics
    assert train_loss < 0.5, train_loss
    assert train_acc <= 1 and train_acc > 0.7, train_acc
    assert test_acc <= 1 and test_acc > 0.7, test_acc
```

Sıfırdan bir uygulama olarak, modelin kayıp fonksiyonunu 0.1 öğrenme oranıyla optimize ederek Section 3.2 içinde tanımlanan minigrup rasgele gradyan inişini kullanıyoruz.

```
lr = 0.1

def updater(batch_size):
    return d2l.sgd([W, b], lr, batch_size)
```

Şimdi modeli 10 dönem ile eğitiyoruz. Hem dönem sayısının (`num_epochs`) hem de öğrenme oranının (`lr`) ayarlanabilir hiper parametreler olduğuna dikkat edin. Değerlerini değiştirerek modelin sınıflandırma doğruluğunu artırabiliriz.

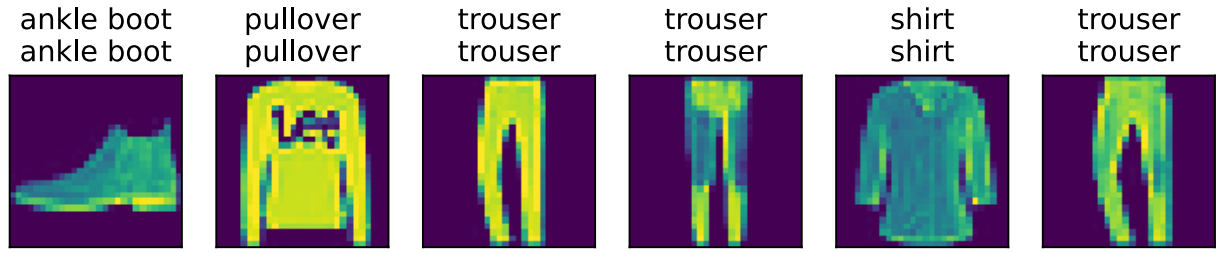
```
num_epochs = 10
train_ch3(net, train_iter, test_iter, cross_entropy, num_epochs, updater)
```



3.6.7 Tahminleme

Artık eğitim tamamlandı, modelimiz bazı imgeleri sınıflandırmaya hazır. Bir dizi resim verildiğinde, bunların gerçek etiketlerini (metin çıktısının ilk satırı) ve modelden gelen tahminleri (metin çıktısının ikinci satırı) karşılaştıracağız.

```
def predict_ch3(net, test_iter, n=6):  #@save
    """Etiketleri tahmin etme (Bölüm 3'te tanımlanmıştır)."""
    for X, y in test_iter:
        break
    trues = d2l.get_fashion_mnist_labels(y)
    preds = d2l.get_fashion_mnist_labels(net(X).argmax(axis=1))
    titles = [true +'\n' + pred for true, pred in zip(trues, preds)]
    d2l.show_images(
        X[0:n].reshape((n, 28, 28)), 1, n, titles=titles[0:n])

predict_ch3(net, test_iter)
```



3.6.8 Özet

- Softmaks regresyonu ile çok sınıflı sınıflandırma için modeller eğitebiliriz.
- Softmaks regresyonunun eğitim döngüsü doğrusal regresyondakine çok benzer: Verileri alın ve okuyun, modelleri ve kayıp fonksiyonlarını tanımlayın, ardından optimizasyon algoritmalarını kullanarak modelleri eğitin. Yakında öğreneceğiniz gibi, en yaygın derin öğrenme modellerinin benzer eğitim yordamları vardır.

3.6.9 Alıştırmalar

1. Bu bölümde, softmaks işlemini matematiksel tanımına dayalı olarak doğrudan uyguladık. Bu hangi sorunlara neden olabilir? İpucu: $\exp(50)$'nin boyutunu hesaplamaya çalışın.
2. Bu bölümdeki `cross_entropy` işlevi, çapraz entropi kaybı işlevinin tanımına göre uygulandı. Bu uygulamadaki sorun ne olabilir? İpucu: Logaritmanın etki alanını düşünün.
3. Yukarıdaki iki sorunu çözmek için düşünebileceğiniz çözümler nelerdir?
4. En olası etiketi iade etmek her zaman iyi bir fikir midir? Örneğin, bunu tıbbi teşhis için yapar mıydınız?
5. Bazı özniteliklere dayanarak sonraki kelimeyi tahmin etmek için softmaks regresyonunu kullanmak istediğimizi varsayalım. Geniş bir kelime dağarcığı kullanımından ortaya çıkabilecek problemler nelerdir?

Tartışmalar[58]

3.7 Softmaks Regresyonunun Kısa Uygulaması

Section 3.3 içindeki derin öğrenme çerçevelerinin yüksek seviyeli API'leri doğrusal regresyon uygulamasını çok daha kolay hale getirdi, onu sınıflandırma modellerini uygulamada benzer şekilde (veya muhtemelen daha fazla) uygun bulacağız. Fashion-MNIST veri kümesine bağlı kalalım ve iş boyutunu Section 3.6 gibi 256'da tutalım.

```
import torch
from torch import nn
from d2l import torch as d2l
```

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

[58] https://discuss.d2l.ai/t/51

3.7.1 Model Parametrelerini İlkleme

Section 3.4 içinde bahsedildiği gibi, softmaks regresyonunun çıktı katmanı tam bağlı bir katmandır. Bu nedenle, modelimizi uygulamak için, `Sequential`'a 10 çıktılı tam bağlı bir katman eklememiz yeterlidir. Yine burada, `Sequential` gerçekten gerekli değildir, ancak derin modelleri uygularken her yerde bulunacağından bu alışkanlığı oluşturalım. Yine, ağırlıkları sıfır ortalama ve 0.01 standart sapma ile rastgele ilkliyoruz.

```
# PyTorch, girdileri dolaylı olarak yeniden şekillendirmez. Bu yüzden,
# ağımızdaki doğrusal katmandan önceki girdileri yeniden şekillendirmek için
# düzleştirilmiş katmanı tanımlarız.
net = nn.Sequential(nn.Flatten(), nn.Linear(784, 10))

def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, std=0.01)

net.apply(init_weights);
```

3.7.2 Softmaks Uygulamasına Yeniden Bakış

Önceki örneğimiz Section 3.6 içinde, modelimizin çıktısını hesapladık ve sonra bu çıktıyı çapraz entropi kaybıyla çalıştırdık. Matematiksel olarak bu, yapılacak son derece makul bir şeydir. Bununla birlikte, hesaplama açısından, üs alma, sayısal kararlılık sorunlarının bir kaynağı olabilir.

Softmaks fonksiyonunun $\hat{y}_j = \frac{\exp(o_j)}{\sum_k \exp(o_k)}$'yi hesapladığını hatırlayın; burada $\hat{y}_j$ tahmin edilen olasılık dağılımı $\hat{\mathbf{y}}$'nin $j.$ öğesidir ve o_j, $\mathbf{o}$ logitlerinin $j.$ öğesidir. o_k değerlerinden bazıları çok büyükse (yani çok pozitifse), o zaman $\exp(o_k)$ belirli veri türleri için sahip olabileceğimiz en büyük sayıdan daha büyük olabilir (yani *taşar*). Bu, paydayı (ve/veya payı) `inf` (sonsuz) yapar ve o zaman $\hat{y}_j$ için 0, `inf` veya `nan` (sayı değil) ile karşılaşırız. Bu durumlarda çapraz entropi için iyi tanımlanmış bir dönüş değeri elde edemeyiz.

Bunu aşmanın bir yolu, softmaks hesaplamasına geçmeden önce ilk olarak $\max(o_k)$'yı tüm o_k'dan çıkarmaktır. Burada her bir o_k'nın sabit faktörle kaydırılmasının softmaksın dönüş değerini değiştirmediğini görebilirsiniz.

$$\begin{aligned} \hat{y}_j &= \frac{\exp(o_j - \max(o_k))\exp(\max(o_k))}{\sum_k \exp(o_k - \max(o_k))\exp(\max(o_k))} \\ &= \frac{\exp(o_j - \max(o_k))}{\sum_k \exp(o_k - \max(o_k))}. \end{aligned} \tag{3.7.1}$$

Çıkarma ve normalleştirme adımından sonra, bazı $o_j - \max(o_k)$ büyük negatif değerlere sahip olabilir ve bu nedenle karşılık gelen $\exp(o_j - \max(o_k))$ sıfıra yakın değerler alacaktır. Bunlar, sonlu kesinlik (yani, *küçümenlik*) nedeniyle sıfıra yuvarlanabilir, $\hat{y}_j$ sıfır yapar ve $\log(\hat{y}_j)$ için bize `-inf` verir. Geri yaymada yolun birkaç adım aşağısında, kendimizi korkutucu `nan` sonuçlarıyla karşı karşıya bulabiliriz.

Neyse ki, üstel fonksiyonları hesaplasak bile, nihayetinde onların loglarını (çapraz entropi kaybını hesaplarken) almayı planladığımız gerçeğiyle kurtulduk. Bu iki softmaks ve çapraz entropi operatörünü bir araya getirerek, aksi takdirde geri yayma sırasında başımıza bela olabilecek sayısal kararlılık sorunlarından kaçabiliriz. Aşağıdaki denklemde gösterildiği gibi, $\exp(o_j - \max(o_k))$'yi

hesaplamaktan kaçınırız ve bunun yerine $\log(\exp(\cdot))$ içini iptal ederek doğrudan $o_j - \max(o_k)$ kullanabiliriz:

$$\begin{aligned}\log\left(\hat{y}_j\right) &= \log\left(\frac{\exp(o_j - \max(o_k))}{\sum_k \exp(o_k - \max(o_k))}\right) \\ &= \log\left(\exp(o_j - \max(o_k))\right) - \log\left(\sum_k \exp(o_k - \max(o_k))\right) \\ &= o_j - \max(o_k) - \log\left(\sum_k \exp(o_k - \max(o_k))\right).\end{aligned} \tag{3.7.2}$$

Modelimizin çıktı olasılıklarını değerlendirmek istememiz durumunda, geleneksel softmaks işlevini el altında tutmak isteyeceğiz. Ancak softmaks olasılıklarını yeni kayıp fonksiyonumuza geçirmek yerine, akıllılık yapıp "LogSumExp numarası"[59] kullanarak logitleri geçireceğiz, bütün softmaks ve logaritma değerlerini çapraz entropi kaybında hesaplayacağız.

```
loss = nn.CrossEntropyLoss(reduction='none')
```

3.7.3 Optimizasyon Algoritması

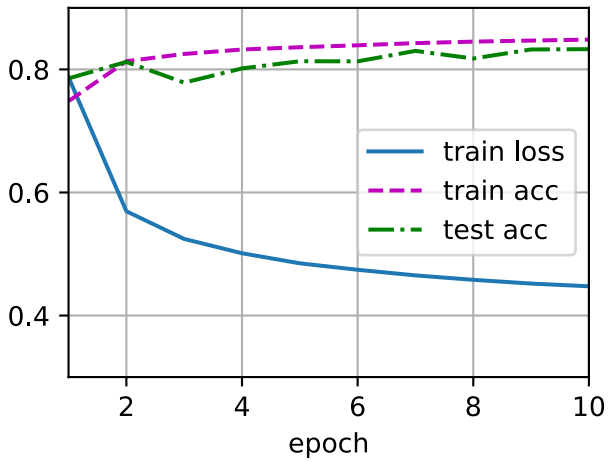
Burada, optimizasyon algoritması olarak 0.1 öğrenme oranıyla minigrup rasgele gradyan inişini kullanıyoruz. Bunun doğrusal regresyon örneğinde uyguladığımızla aynı olduğuna ve optimize edicilerin genel uygulanabilirliğini gösterdiğine dikkat edin.

```
trainer = torch.optim.SGD(net.parameters(), lr=0.1)
```

3.7.4 Eğitim

Ardından modeli eğitirken Section 3.6 içinde tanımlanan eğitim işlevini çağırıyoruz.

```
num_epochs = 10
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



[59] https://en.wikipedia.org/wiki/LogSumExp

Daha önce olduğu gibi, bu algoritma, bu sefer öncekinden daha az kod satırı ile, iyi bir doğruluk sağlayan bir çözüme yakınsıyor.

3.7.5 Özet

- Yüksek seviyeli API'leri kullanarak softmaks regresyonunu çok daha öz uygulayabiliriz.
- Hesaplama bakış açısından, softmaks regresyonunun uygulanmasının karmaşıklıkları vardır. Pek çok durumda, bir derin öğrenme çerçevesinin sayısal kararlılığı sağlamak için bu çok iyi bilinen hilelerin ötesinde ek önlemler aldığını ve bizi pratikte tüm modellerimizi sıfırdan kodlamaya çalıştığımızda karşılaşacağımız daha da fazla tuzaktan kurtardığını unutmayın.

3.7.6 Alıştırmalar

1. Sonuçların ne olduğunu görmek için grup boyutu, dönem sayısı ve öğrenme oranı gibi hiper parametreleri ayarlamayı deneyin.
2. Eğitim için dönem sayısını artırın. Test doğruluğu neden bir süre sonra düşüyor olabilir? Bunu nasıl düzeltebiliriz?

Tartışmalar[60]

[60] https://discuss.d2l.ai/t/53

4 | Çok Katmanlı Algılayıcılar

Bu bölümde, ilk gerçek *derin* ağınızı tanıtacağız. En basit derin ağlara çok katmanlı algılayıcılar denir ve bunlar, her biri aşağıdaki katmandakilere (girdi aldıkları) ve yukarıdakilere (sırayla etkiledikleri) tam bağlı olan birden fazla nöron (sinir hücresi, burada hesaplama ünitesi) katmanından oluşur. Yüksek kapasiteli modelleri eğittiğimizde, aşırı eğitme (overfitting) riskiyle karşı karşıyayız. Bu nedenle, aşırı eğitme, eksik eğitme (underfitting) ve model seçimi kavramlarıyla ilk titiz karşılaşmanızı sağlamamız gerekecek. Bu sorunlarla mücadele etmenize yardımcı olmak için, ağırlık sönümü (weight decay) ve hattan düşme (dropout) gibi düzenlileştirme tekniklerini tanıtacağız. Derin ağları başarılı bir şekilde eğitmenin anahtarı olan sayısal kararlılık ve parametre ilkleme gibi ilgili sorunları da tartışacağız. Baştan sona, size sadece kavramları değil, aynı zamanda derin ağları kullanma pratiğini de sağlam bir şekilde kavratmayı amaçlıyoruz. Bu bölümün sonunda, şimdiye kadar sunduğumuz şeyleri gerçek bir vakaya uyguluyoruz: Ev fiyatı tahmini. Modellerimizin hesaplama performansı, ölçeklenebilirliği ve verimliliği ile ilgili konuları sonraki bölümlerde değerlendiriyoruz.

4.1 Çok Katmanlı Algılayıcılar

Chapter 3 içinde, softmaks regresyonunu (Section 3.4), algoritmayı sıfırdan uygulayarak (Section 3.6) ve yüksek seviyeli API'leri (Section 3.7) kullanarak ve düşük çözünürlüklü görsellerden 10 giysi kategorisini tanımak için sınıflandırıcıları eğiterek tanıttık. Yol boyunca, verileri nasıl karıştıracağımızı, çıktılarımızı geçerli bir olasılık dağılımına nasıl zorlayacağımızı, uygun bir kayıp fonksiyonunu nasıl uygulayacağımızı ve modelimizin parametrelerine göre onu nasıl en aza indireceğimizi öğrendik. Şimdi bu mekaniği basit doğrusal modeller bağlamında öğrendiğimize göre, bu kitabın öncelikli olarak ilgilendiği nispeten zengin modeller sınıfı olan derin sinir ağlarını keşfetmeye başlayabiliriz.

4.1.1 Gizli Katmanlar

Section 3.1.1 içinde, bir ek girdi eklenen doğrusal bir dönüşüm olan afin dönüşümünü tanımladık. Başlangıç olarak, Fig. 3.4.1 içinde gösterilen softmaks regresyon örneğimize karşılık gelen model mimarisini hatırlayalım. Bu model, girdilerimizi tek bir afin dönüşüm ve ardından bir softmaks işlemi aracılığıyla doğrudan çıktılarımıza eşledi. Etiketlerimiz gerçekten bir afin dönüşüm yoluyla girdi verilerimizle ilişkili olsaydı, bu yaklaşım yeterli olurdu. Ancak afin dönüşümlerdeki doğrusallık *güçlü* bir varsayımdır.

Doğrusal Modeller Ters Gidebilir

Örneğin, doğrusallık *daha zayıf monotonluk* varsayımını ifade eder: Özniteliğimizdeki herhangi bir artışın ya modelimizin çıktısında her zaman bir artışa (karşılık gelen ağırlık pozitifse) ya da modelimizin çıktısında her zaman bir düşüşe neden olmasını gerektirmesi (karşılık gelen ağırlık negatifse). Genelde bu mantıklı gelir. Örneğin, bir bireyin bir krediyi geri ödeyip ödemeyeceğini tahmin etmeye çalışıyor olsaydık, makul olarak ve her şeyi eşit tutarak, daha yüksek gelire sahip bir başvuru sahibinin, daha düşük gelirli bir başvuru sahibine göre geri ödeme olasılığının her zaman daha yüksek olacağını hayal edebilirdik. Monoton olsa da, bu ilişki muhtemelen geri ödeme olasılığıyla doğrusal olarak ilişkili değildir. Gelirdeki 0'dan 50 bine bir artış, geri ödeme olabilirliğinde kapsamında gelirdeki 1 milyondan 1.05 milyona artıştan daha büyük bir artmaya karşılık gelir. Bunu halletmenin bir yolu, verilerimizi, örneğin özniteliğimiz olan gelirin logaritmasını kullanarak doğrusallığın daha makul hale geleceği şekilde, önceden işlemek olabilir.

Monotonluğu ihlal eden örnekleri kolayca bulabileceğimizi unutmayın. Örneğin, vücut ısısına bağlı olarak ölüm olasılığını tahmin etmek istediğimizi varsayalım. Vücut ısısı 37°C'nin (98.6°F) üzerinde olan kişiler için, daha yüksek sıcaklıklar daha büyük riski gösterir. Bununla birlikte, vücut ısısı 37°C'nin altında olan kişiler için, daha yüksek sıcaklıklar daha düşük riski gösterir! Bu durumda da sorunu akıllıca bir ön işlemle çözebiliriz. Yani 37°C'den uzaklığı özniteliğimiz olarak kullanabiliriz.

Peki ya kedi ve köpeklerin imgelerini sınıflandırmaya ne dersiniz? (13, 17) konumundaki pikselin yoğunluğunu artırmak, imgenin bir köpeği tasvir etme olabilirliğini her zaman artırmalı mı (yoksa her zaman azaltmalı mı)? Doğrusal bir modele güvenmek, kedileri ve köpekleri ayırt etmek için tek gerekliliğin tek tek piksellerin parlaklığını değerlendirmek olduğu şeklindeki örtülü varsayıma karşılık gelir. Bu yaklaşım, bir imgenin tersine çevrilmesinin kategoriyi koruduğu bir dünyada başarısızlığa mahkumdur.

Yine de, burada doğrusallığın aşikar saçmalığına rağmen, önceki örneklerimizle karşılaştırıldığında, sorunu basit bir ön işleme düzeltmesiyle çözebileceğimiz daha az açıktır. Bunun nedeni, herhangi bir pikselin anlamının karmaşık yollarla bağlamına (çevreleyen piksellerin değerlerine) bağlı olmasıdır. Verilerimizin özniteliklerimiz arasındaki ilgili etkileşimleri hesaba katan bir temsili olabilir, bunun üzerine doğrusal bir model uygun olabilir, ancak biz bunu elle nasıl hesaplayacağımızı bilmiyoruz. Derin sinir ağlarıyla, hem gizli katmanlar aracılığıyla bir gösterimi hem de bu gösterim üzerinde işlem yapan doğrusal bir tahminciyi birlikte öğrenmek için gözlemsel verileri kullandık.

Gizli Katmanları Birleştirme

Doğrusal modellerin bu sınırlamalarının üstesinden gelebiliriz ve bir veya daha fazla gizli katman ekleyerek daha genel işlev sınıflarıyla başa çıkabiliriz. Bunu yapmanın en kolay yolu, tam bağlı birçok katmanı birbirinin üzerine yığmaktır. Her katman, biz çıktılar üretene kadar üstündeki katmana beslenir. İlk $L-1$ katmanı temsilimiz ve son katmanı da doğrusal tahmincimiz olarak düşünebiliriz. Bu mimariye genellikle *çok katmanlı algılayıcı (multilayer perceptron)* denir ve genellikle *MLP* olarak kısaltılır. Aşağıda, bir MLP'yi şematik olarak tasvir ediyoruz (Fig. 4.1.1).

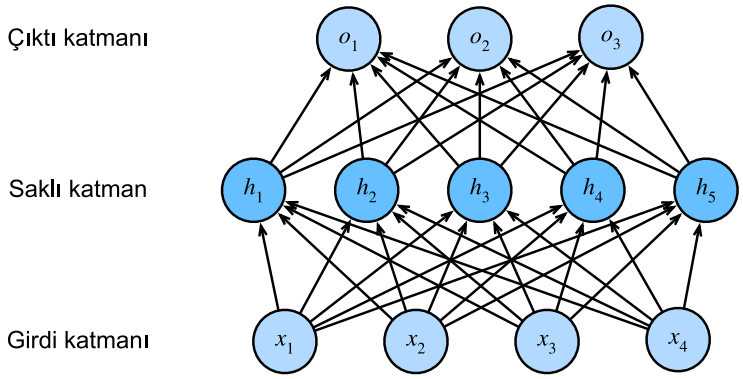


Fig. 4.1.1: 5 gizli birimli bir gizli katmana sahip bir MLP

Bu MLP'nin 4 girdisi, 3 çıktısı vardır ve gizli katmanı 5 gizli birim içerir. Girdi katmanı herhangi bir hesaplama içermediğinden, bu ağ ile çıktıların üretilmesi hem gizli hem de çıktı katmanları için hesaplamaların gerçekleştirilmesini gerektirir; dolayısıyla, bu MLP'deki katman sayısı 2'dir. Bu katmanların her ikisinin de tam bağlı olduğuna dikkat edin. Her girdi, gizli katmandaki her nöronu etkiler ve bunların her biri de çıktı katmanındaki her nöronu etkiler. Ancak, Section 3.4.3 içinde önerildiği gibi, tam bağlı katmanlara sahip MLP'lerin parametreleştirme maliyeti girdi veya çıktı boyutunu değiştirmeden bile parametre tasarrufu ve model etkinliği arasındaki ödünleşimi motive edebilecek şekilde yüksek olabilir (Zhang *et al.*, 2021).

Doğrusaldan Doğrusal Olmayana

Daha önce olduğu gibi, $\mathbf{X} \in \mathbb{R}^{n \times d}$ matrisiyle, her örneğin d girdisine (öznitelikler) sahip olduğu n örneklerden oluşan bir minigrubu gösteriyoruz. Gizli katmanı h gizli birimlere sahip olan tek gizli katmanlı bir MLP için, gizli katmanın çıktılarını, ki bunlar *gizli gösterimler*dir, $\mathbf{H} \in \mathbb{R}^{n \times h}$ ile belirtelim. Matematikte ve kodlamada, $\mathbf{H}$ aynı zamanda *gizli katman değişkeni* veya *gizli değişken* olarak da bilinir. Gizli ve çıktı katmanlarının her ikisi de tam bağlı olduğundan, $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h}$ ve $\mathbf{b}^{(1)} \in \mathbb{R}^{1 \times h}$ içinde sırasıyla gizli katman ağırlıkları ve ek girdileri, $\mathbf{W}^{(2)} \in \mathbb{R}^{h \times q}$ ve $\mathbf{b}^{(2)} \in \mathbb{R}^{1 \times q}$ içinde de çıktı katmanı ağırlıkları ve ek girdileri var. Biçimsel olarak, tek gizli katmanlı MLP'nin $\mathbf{O} \in \mathbb{R}^{n \times q}$ çıktılarını şu şekilde hesaplarız:

$$\begin{aligned} \mathbf{H} &= \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}, \\ \mathbf{O} &= \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}. \end{aligned} \tag{4.1.1}$$

Gizli katmanı ekledikten sonra, modelimizin artık ek parametre kümelerini izlememizi ve güncellememizi gerektirdiğini unutmayın. Peki karşılığında ne kazandık? Yukarıda tanımlanan modelde *sorunlarımız için hiçbir şey kazanmadığımızı* öğrenmek sizi şaşırtabilir! Nedeni açık. Yukarıdaki gizli birimler, girdilerin bir afin işlevi tarafından verilir ve çıktılar (softmaks-öncesi), gizli birimlerin yalnızca bir afin işlevidir. Bir afin fonksiyonun afin fonksiyonu, kendi başına bir afin fonksiyonudur. Dahası, doğrusal modelimiz zaten herhangi bir afin işlevi temsil edebiliyordu.

Ağırlıkların herhangi bir değeri için gizli katmanı daraltarak parametrelerle eşdeğer bir tek katmanlı model oluşturabileceğimizi kanıtlayarak eşdeğerliği biçimsel olarak görebiliriz, $\mathbf{W} = \mathbf{W}^{(1)}\mathbf{W}^{(2)}$ and $\mathbf{b} = \mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$:

$$\mathbf{O} = (\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W} + \mathbf{b}. \tag{4.1.2}$$

Çok katmanlı mimarilerin potansiyelini gerçekleştirmek için, bir anahtar bileşene daha ihtiyacımız var: Afin dönüşümün ardından her gizli birime uygulanacak doğrusal olmayan σ *etkin-*

leştirme (aktivasyon) fonksiyonu. Etkinleştirme fonksiyonlarının çıktılarına (örn. $\sigma(\cdot)$) etkinleştirmeler denir. Genel olarak, etkinleştirme işlevleri yürürlükte olduğunda, MLP'mizi doğrusal bir modele indirgemek artık mümkün değildir:

$$\begin{aligned}\mathbf{H} &= \sigma(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}),\\ \mathbf{O} &= \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}.\end{aligned} \tag{4.1.3}$$

$\mathbf{X}$ içindeki her satır, minigruptaki bir örneğe karşılık geldiğinden, birazcık gösterimi kötüye kullanarak, σ doğrusal olmama durumunu kendi girdilerine satır yönlü, yani her seferinde bir örnek olarak uygulanacak şekilde tanımlarız. Softmaks için gösterimi, aynı şekilde, numref:`subsec_softmax_vectorization` içindeki gibi bir satırsal işlemi belirtmek için kullandığımıza dikkat edin. Çoğu zaman, bu bölümde olduğu gibi, gizli katmanlara uyguladığımız etkinleştirme işlevleri yalnızca satır bazında değil, bazen eleman yönlüdür. Bu, katmanın doğrusal bölümünü hesapladıktan sonra, diğer gizli birimler tarafından alınan değerlere bakmadan her bir etkinleştirmeyi hesaplayabileceğimiz anlamına gelir. Bu, çoğu etkinleştirme işlevi için geçerlidir.

Daha genel MLP'ler oluşturmak için, bu tür gizli katmanı yığmaya devam edebiliriz, örneğin, $\mathbf{H}^{(1)} = \sigma_1(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})$ ve $\mathbf{H}^{(2)} = \sigma_2(\mathbf{H}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)})$, birbiri ardına, her zamankinden daha kuvvetli ifade edici modeller üretiyor.

Evrensel Yaklaşımlar

MLP'ler, girdilerin her birinin değerine bağlı olan gizli nöronları aracılığıyla girdilerimiz arasındaki karmaşık etkileşimleri yakalayabilir. Örneğin, bir çift girdi üzerinde temel mantık işlemleri gibi rastgele hesaplamalar için kolayca gizli düğümler tasarlayabiliriz. Dahası, etkinleştirme fonksiyonunun belli seçimleri için MLP'lerin evrensel yaklaşımcılar olduğu yaygın olarak bilinmektedir. Yeterli düğüm (muhtemelen saçma bir şekilde çok) ve doğru ağırlık kümesi verilen tek gizli katmanlı bir ağla bile, aslında o işlevi öğrenmek zor olan kısım olsa da, herhangi bir işlevi modelleyebiliriz. Sinir ağınızı biraz C programlama dili gibi düşünebilirsiniz. Dil, diğer herhangi bir modern dil gibi, herhangi bir hesaplanabilir programı ifade etme yeteneğine sahiptir. Ama aslında sizin şartnamenizi karşılayan bir program bulmak zor kısımdır.

Dahası, tek gizli katmanlı bir ağın *herhangi bir işlevi öğrenebilmesi*, tüm problemlerinizi tek gizli katmanlı ağlarla çözmeye çalışmanız gerektiği anlamına gelmez. Aslında, daha derin (daha genişe kıyasla) ağları kullanarak birçok işlevi çok daha öz bir şekilde tahmin edebiliriz. Sonraki bölümlerde daha sıkı tartışmalara gireceğiz.

4.1.2 Etkinleştirme Fonksiyonları

Etkinleştirme fonksiyonları, ağırlıklı toplamı hesaplayarak ve buna ek girdi ekleyerek bir nöronun aktive edilip edilmeyeceğine karar verir. Bunlar girdi sinyallerini çıktılara dönüştürmek için türevlenebilir operatörlerdir ve çoğu doğrusal olmayanlık ekler. Etkinleştirme fonksiyonları derin öğrenme için temel olduğundan, bazı genel etkinleştirme fonksiyonlarını kısaca inceleyelim.

```
%matplotlib inline
import torch
from d2l import torch as d2l
```

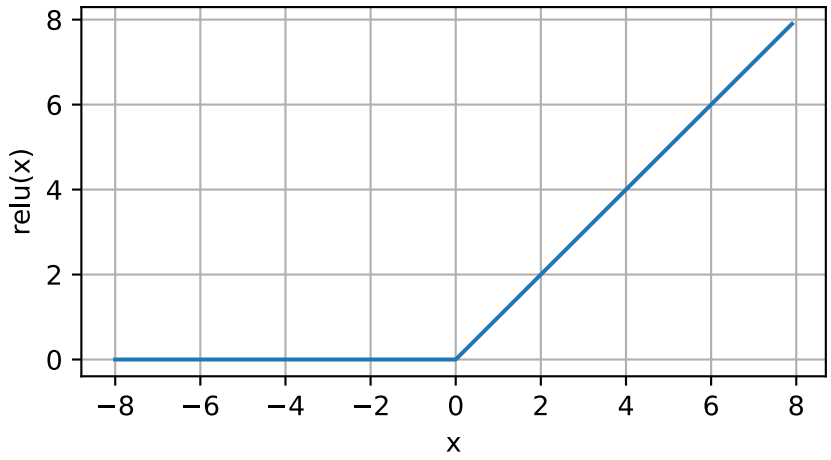
ReLU İşlevi

Hem uygulamanın basitliği hem de çeşitli tahminci görevlerdeki iyi performansı nedeniyle en popüler seçenek, *düzeltilmiş doğrusal birimdir* (*ReLU*). ReLU, çok basit doğrusal olmayan bir dönüşüm sağlar. x öğesi verildiğinde, işlev o öğenin ve 0'ın maksimum değeri olarak tanımlanır:

$$\mathrm{ReLU}(x) = \max(x, 0). \tag{4.1.4}$$

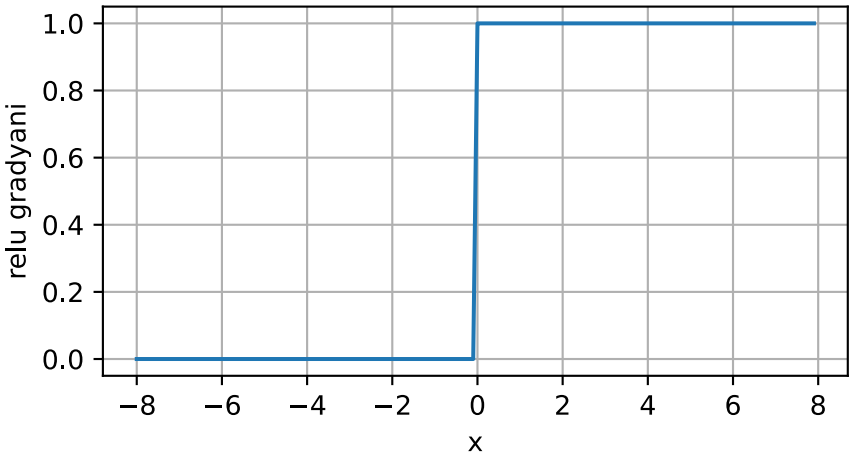
ReLU işlevi tabiri caizse yalnızca pozitif öğeleri tutar ve karşılık gelen etkinleştirmeleri 0'a ayarlayarak tüm negatif öğeleri atar. Biraz önsezi kazanmak için bu işlevi çizebiliriz. Gördüğünüz gibi, aktivasyon fonksiyonu parçalı doğrusaldır.

```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.relu(x)
d2l.plot(x.detach(), y.detach(), 'x', 'relu(x)', figsize=(5, 2.5))
```



Girdi negatif olduğunda, ReLU fonksiyonunun türevi 0'dır ve girdi pozitif olduğunda ReLU fonksiyonunun türevi 1'dir. Girdi tam olarak 0'a eşit değer aldığında ReLU fonksiyonunun türevlenemeyeceğine dikkat edin. Bu durumlarda, sol taraftaki türevi varsayılan olarak kullanırız ve girdi 0 olduğunda türevin 0 olduğunu söyleriz. Bundan sıyırabiliriz çünkü girdi aslında sıfır olmayabilir. Eski bir özdeyiş vardır, eğer ince sınır koşulları önemliyse, muhtemelen mühendislik değil (*gerçek*) matematik yapıyoruz. Bu geleneksel bilgelik burada geçerli olabilir. Aşağıda ReLU fonksiyonunun türevini çiziyoruz.

```
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'relu gradyani', figsize=(5, 2.5))
```



ReLU kullanmanın nedeni, türevlerinin özellikle makul davranmasıdır: Ya kaybolurlar ya da sadece argümanın geçmesine izin verirler. Bu, optimizasyonun daha iyi davranmasını sağlar ve sinir ağlarının önceki sürümlerinin belalısı bilindik gradyanların kaybolması sorununu hafifletir (bu konu hakkında daha sonra tartışacağız).

Parametreleştirilmiş ReLU (*pReLU*) işlevi dahil olmak üzere ReLU işlevinin birçok çeşidi olduğunu unutmayın (He *et al.*, 2015). Bu sürüm, ReLU'ya doğrusal bir terim ekler, bu nedenle, argüman negatif olsa bile biraz bilgi yine de geçer:

$$\text{pReLU}(x) = \max(0, x) + \alpha \min(0, x). \tag{4.1.5}$$

Sigmoid İşlevi

Sigmoid işlevi, değerleri $\mathbb{R}$ alanında bulunan girdileri (0, 1) aralığındaki çıktılara dönüştürür. Bu nedenle, sigmoid genellikle *sıkıştırma işlevi* olarak adlandırılır: (-sonsuz, sonsuz) aralığındaki herhangi bir girdiyi (0, 1) aralığındaki bir değere sıkıştırır:

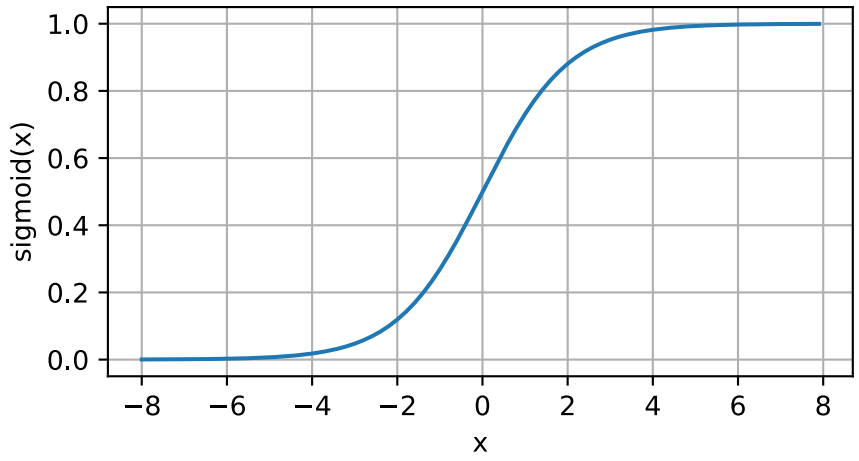
$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}. \tag{4.1.6}$$

İlk sinir ağlarında, bilim adamları ya *ateşleyen* ya da *ateşlemeyen* biyolojik nöronları modellemekle ilgileniyorlardı. Böylece, yapay nöronun mucitleri McCulloch ve Pitts'e kadar uzanan bu alanın öncüleri, eşikleme birimlerine odaklandılar. Bir eşikleme aktivasyonu, girdisi herhangi bir eşiğin altında olduğunda 0 değerini ve girdi eşiği aştığında 1 değerini alır.

Dikkat gradyan tabanlı öğrenmeye kaydırıldığında, sigmoid işlevi doğal bir seçimdi çünkü bir eşikleme birimine yumuşak, türevlenebilir bir yaklaşımdır. Sigmoidler, çıktıları ikili sınıflandırma problemleri için olasılıklar olarak yorumlamak istediğimizde (sigmoidi softmaksın özel bir durumu olarak düşünebilirsiniz), çıktı birimlerinde etkinleştirme fonksiyonları olarak hala yaygın olarak kullanılmaktadır. Bununla birlikte, çoğu kullanımda gizli katmanlarda sigmoid çoğunlukla daha basit ve daha kolay eğitilebilir ReLU ile değiştirilmiştir. Yinelemeli sinir ağlarıyla ilgili bölümlerde, zaman içinde bilgi akışını kontrol etmek için sigmoid birimlerinden yararlanan mimarilere değineceğiz.

Aşağıda sigmoid fonksiyonunu çiziyoruz. Girdi 0'a yakın olduğunda, sigmoid fonksiyonunun doğrusal bir dönüşüme yaklaştığını unutmayın.

```
y = torch.sigmoid(x)
d2l.plot(x.detach(), y.detach(), 'x', 'sigmoid(x)', figsize=(5, 2.5))
```

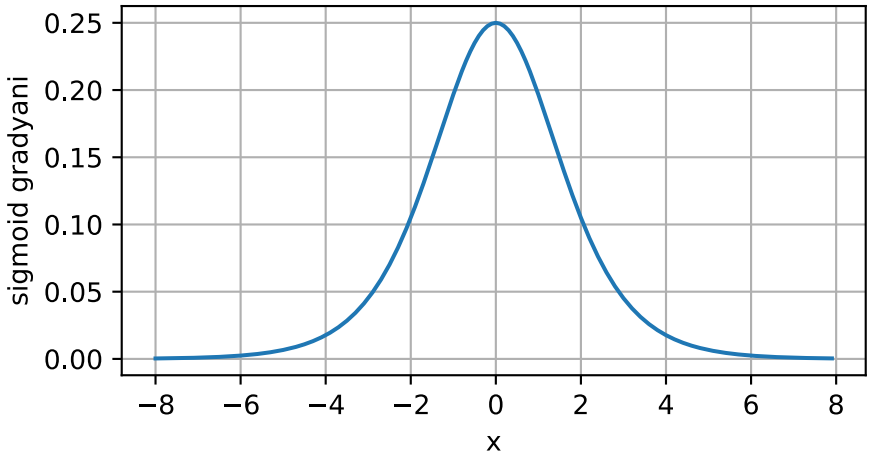


Sigmoid fonksiyonunun türevi aşağıdaki denklemde verilmiştir:

$$\frac{d}{dx}\,\text{sigmoid}(x) = \frac{\exp(-x)}{(1+\exp(-x))^2} = \text{sigmoid}(x)\,(1-\text{sigmoid}(x))\,. \tag{4.1.7}$$

Sigmoid fonksiyonunun türevi aşağıda çizilmiştir. Girdi 0 olduğunda, sigmoid fonksiyonunun türevinin maksimum 0.25'e ulaştığını fark edin. Girdi her iki yönde de 0'dan uzaklaştıkça, türev 0'a yaklaşır.

```
# Clear out previous gradients
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'sigmoid gradyani', figsize=(5, 2.5))
```



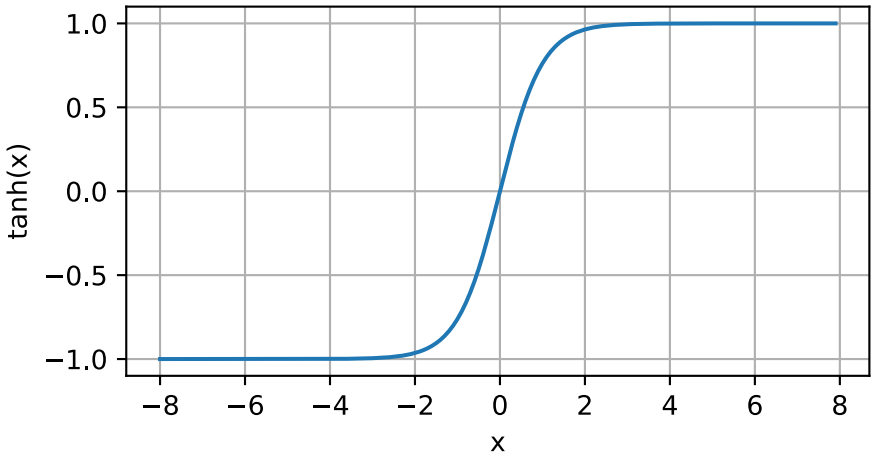
Tanh İşlevi

Sigmoid işlevi gibi, tanh (hiperbolik tanjant) işlevi de girdilerini sıkıştırarak -1 ile 1 aralığındaki öğelere dönüştürür:

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}. \tag{4.1.8}$$

Tanh fonksiyonunu aşağıda çiziyoruz. Girdi 0'a yaklaştıkça tanh fonksiyonunun doğrusal bir dönüşüme yaklaştığına dikkat edin. Fonksiyonun şekli sigmoid fonksiyonuna benzer olmasına rağmen, tanh fonksiyonu koordinat sisteminin orijinine göre nokta simetrisi sergiler.

```
y = torch.tanh(x)
d2l.plot(x.detach(), y.detach(), 'x', 'tanh(x)', figsize=(5, 2.5))
```

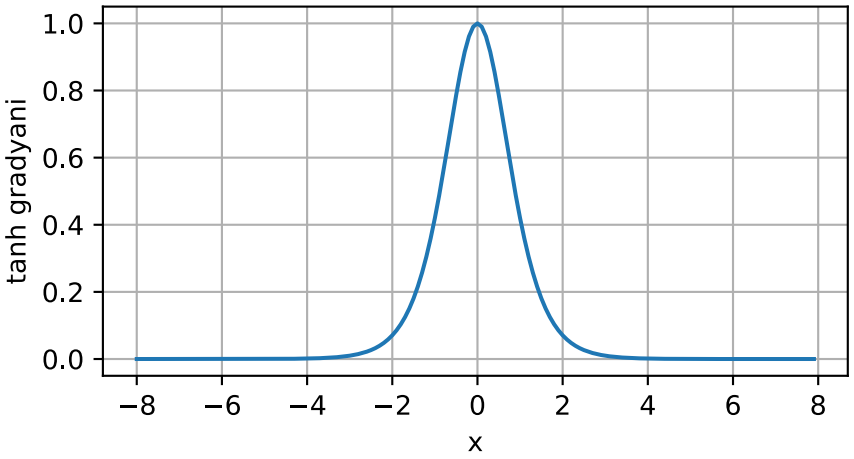


Tanh fonksiyonunun türevi şöyledir:

$$\frac{d}{dx} \tanh(x) = 1 - \tanh^2(x). \tag{4.1.9}$$

Tanh fonksiyonunun türevi aşağıda çizilmiştir. Girdi 0'a yaklaştıkça, tanh fonksiyonunun türevi maksimum 1'e yaklaşır. Sigmoid fonksiyonunda gördüğümüz gibi, girdi her iki yönde de 0'dan uzaklaştıkça, tanh fonksiyonunun türevi 0'a yaklaşır.

```
# Clear out previous gradients.
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'tanh gradyani', figsize=(5, 2.5))
```



Özetle, artık etkileyici çok katmanlı sinir ağı mimarileri oluşturmak için doğrusal olmayanlıkları nasıl birleştireceğimizi biliyoruz. Bir ek bilgi olarak, bilginiz sizi zaten 1990 civarında bir uygulayıcıya benzer bir araç setinin yönetebilmenizi sağlamaktadir. Bazı yönlerden, 1990'larda çalışan herhangi birine göre bir avantajınız var, çünkü sadece birkaç satır kod kullanarak hızlı modeller oluşturmak için güçlü açık kaynaklı derin öğrenme çerçevelerinden yararlanabilirsiniz. Daha önce, bu ağları eğitmek, araştırmacıların binlerce C ve Fortran satırını kodlamasını gerektiriyordu.

4.1.3 Özet

- MLP, çıktı ve girdi katmanları arasına bir veya birden fazla tam bağlı gizli katman ekler ve gizli katmanın çıktısını bir etkinleştirme işlevi aracılığıyla dönüştürür.
- Yaygın olarak kullanılan etkinleştirme işlevleri arasında ReLU işlevi, sigmoid işlevi ve tanh işlevi bulunur.

4.1.4 Alıştırmalar

1. pReLU etkinleştirme fonksiyonunun türevini hesaplayın.
2. Yalnızca ReLU (veya pReLU) kullanan bir MLP'nin sürekli bir parçalı doğrusal fonksiyon oluşturduğunu gösterin.
3. $\tanh(x) + 1 = 2 \,\text{sigmoid}(2x)$ olduğunu gösterin.
4. Bir seferde bir minigruba uygulanan bir doğrusal olmayanlık durumumuz olduğunu varsayalım. Bunun ne tür sorunlara neden olmasını beklersiniz?

Tartışmalar[61]

[61] https://discuss.d2l.ai/t/91

4.2 Çok Katmanlı Algılayıcıların Sıfırdan Uygulanması

Artık çok katmanlı algılayıcıları (MLP'ler) matematiksel olarak nitelendirdiğimize göre, birini kendimiz uygulamaya çalışalım. Softmax regresyonu (Section 3.6) ile elde ettiğimiz önceki sonuçlarla karşılaştırmak için Fashion-MNIST imge sınıflandırma veri kümesi (Section 3.5) ile çalışmaya devam edeceğiz.

```
import torch
from torch import nn
from d2l import torch as d2l
```

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

4.2.1 Model Parametrelerini İlkleme

Fashion-MNIST'in 10 sınıf içerdiğini ve her imgenin $28 \times 28 = 784$ gri tonlamalı piksel değerleri ızgarasından oluştuğunu hatırlayın. Yine şimdilik pikseller arasındaki uzamsal yapıyı göz ardı edeceğiz, bu nedenle bunu 784 girdi özniteliği ve 10 sınıf içeren basit bir sınıflandırma veri kümesi olarak düşünebiliriz. Başlarken, bir gizli katman ve 256 gizli birim içeren bir MLP uygulayacağız. Bu miktarların ikisini de hiper parametreler olarak kabul edebileceğimizi unutmayın. Tipik olarak, belleğin donanımda öyle tahsis edildiğinden ve adreslendiğinden hesaplama açısından verimli olma eğiliminde olan 2'nin katlarında katman genişliklerini seçiyoruz.

Yine, parametrelerimizi birkaç tensörle temsil edeceğiz. *Her katman* için, bir ağırlık matrisini ve bir ek girdi vektörünü izlememiz gerektiğini unutmayın. Her zaman olduğu gibi, bu parametrelere göre kaybın gradyanları için bellek ayırıyoruz.

```
num_inputs, num_outputs, num_hiddens = 784, 10, 256

W1 = nn.Parameter(torch.randn(
    num_inputs, num_hiddens, requires_grad=True) * 0.01)
b1 = nn.Parameter(torch.zeros(num_hiddens, requires_grad=True))
W2 = nn.Parameter(torch.randn(
    num_hiddens, num_outputs, requires_grad=True) * 0.01)
b2 = nn.Parameter(torch.zeros(num_outputs, requires_grad=True))

params = [W1, b1, W2, b2]
```

4.2.2 Etkinleştirme Fonksiyonu

Her şeyin nasıl çalıştığını bildiğimizden emin olmak için, ReLU aktivasyonunu yerleşik `relu` işlevini doğrudan çağırmak yerine maksimum işlevi kullanarak kendimiz uygulayacağız.

```
def relu(X):
    a = torch.zeros_like(X)
    return torch.max(X, a)
```

4.2.3 Model

Uzamsal yapıyı göz ardı ettiğimiz için, her iki boyutlu imgeyi `num_inputs` uzunluğuna sahip düz bir vektör halinde (`reshape`) yeniden şekillendiriyoruz. Son olarak, modelimizi sadece birkaç satır kodla uyguluyoruz.

```
def net(X):
    X = X.reshape((-1, num_inputs))
    H = relu(X@W1 + b1)  # Burada '@' matris carpimini temsil eder
    return (H@W2 + b2)
```

4.2.4 Kayıp İşlevi

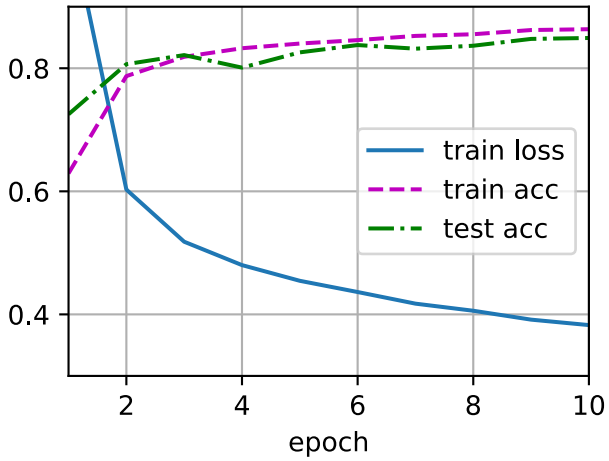
Sayısal kararlılığı sağlamak için ve softmaks işlevini zaten sıfırdan uygularken (Section 3.6), softmaks ve çapraz entropi kaybını hesaplamak için yüksek seviyeli API'lerden birleşik işlevi kullanıyoruz. Bu karmaşıklıkla ilgili önceki tartışmamızı Section 3.7.2 içinden hatırlayın. İlgili okuru, uygulama ayrıntıları hakkındaki bilgilerini derinleştirmek için kayıp işlevi kaynak kodunu incelemeye teşvik ediyoruz.

```
loss = nn.CrossEntropyLoss(reduction='none')
```

4.2.5 Eğitim

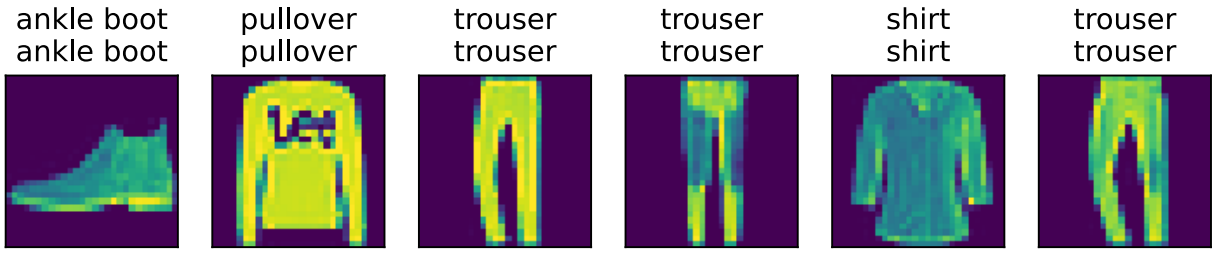
Neyse ki, MLP'ler için eğitim döngüsü softmax bağlanımıyla tamamen aynıdır. Tekrar `d2l` paketini kullanarak, `train_ch3` fonksiyonunu çağırıyoruz (bkz. Section 3.6), dönem sayısını 10 ve öğrenme oranını 0.1 olarak ayarlıyoruz.

```
num_epochs, lr = 10, 0.1
updater = torch.optim.SGD(params, lr=lr)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, updater)
```



Öğrenilen modeli değerlendirmek için onu bazı test verisine uyguluyoruz.

```
d2l.predict_ch3(net, test_iter)
```



4.2.6 Özet

- Manuel olarak yapıldığında bile basit bir MLP uygulamanın kolay olduğunu gördük.
- Bununla birlikte, çok sayıda katmanla, MLP'leri sıfırdan uygulamak yine de karmaşık olabilir (örneğin, modelimizin parametrelerini adlandırmak ve takip etmek).

4.2.7 Alıştırmalar

1. Hiper parametre `num_hiddens` değerini değiştirin ve bu hiper parametrenin sonuçlarınızı nasıl etkilediğini görün. Diğerlerini sabit tutarak bu hiper parametrenin en iyi değerini belirleyiniz.
2. Sonuçları nasıl etkilediğini görmek için ek bir gizli katman eklemeyi deneyiniz.
3. Öğrenme oranını değiştirmek sonuçlarınızı nasıl değiştirir? Model mimarisini ve diğer hiper parametreleri (dönem sayısı dahil) sabitlersek, hangi öğrenme oranı size en iyi sonuçları verir?
4. Tüm hiper parametreleri (öğrenme oranı, dönem sayısı, gizli katman sayısı, katman başına gizli birim sayısı) birlikte optimize ederek elde edebileceğiniz en iyi sonuç nedir?
5. Birden fazla hiper parametre ile uğraşmanın neden çok daha zor olduğunu açıklayınız.
6. Birden fazla hiper parametre üzerinde bir arama yapılandırmak için düşünebileceğiniz en akıllı strateji nedir?

Tartışmalar[62]

4.3 Çok Katmanlı Algılayıcıların Kısa Uygulaması

Tahmin edebileceğiniz gibi, yüksek seviye API'lere güvenerek, MLP'leri daha da kısaca uygulayabiliriz.

```
import torch
from torch import nn
from d2l import torch as d2l
```

[62] https://discuss.d2l.ai/t/93

4.3.1 Model

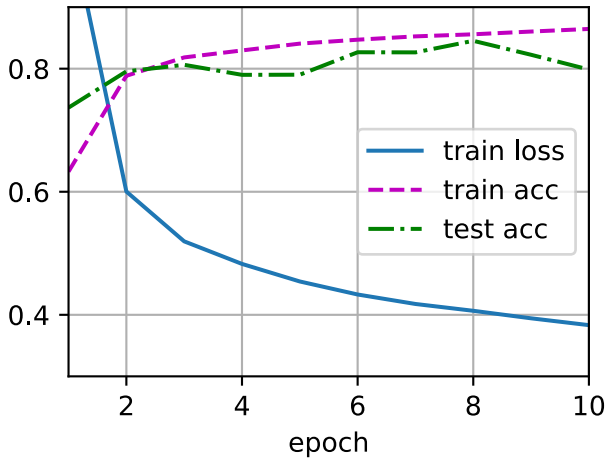
Kısa softmaks bağlanım uygulamamızla karşılaştırıldığında (Section 3.7), tek fark *iki* tam bağlı katman eklememizdir (önceden *bir* tane ekledik). İlki, 256 gizli birim içeren ve ReLU etkinleştirme fonksiyonunu uygulayan gizli katmanımızdır. İkincisi, çıktı katmanımızdır.

```
net = nn.Sequential(nn.Flatten(),
                    nn.Linear(784, 256),
                    nn.ReLU(),
                    nn.Linear(256, 10))

def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, std=0.01)

net.apply(init_weights);
```

Eğitim döngüsü, softmaks bağlanımını uyguladığımız zamanki ile tamamen aynıdır. Bu modülerlik, model mimarisiyle ilgili konuları dikey düşünmelerden ayırmamızı sağlar.

```
batch_size, lr, num_epochs = 256, 0.1, 10
loss = nn.CrossEntropyLoss(reduction='none')
trainer = torch.optim.SGD(net.parameters(), lr=lr)
```

```
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



4.3.2 Özet

- Yüksek seviye API'leri kullanarak MLP'leri çok daha kısaca uygulayabiliriz.
- Aynı sınıflandırma problemi için, bir MLP'nin uygulanması, etkinleştirme fonksiyonlarına sahip ek gizli katmanlar haricinde softmaks bağlanımının uygulanmasıyla aynıdır.

4.3.3 Alıştırmalar

1. Farklı sayıda gizli katman eklemeyi deneyiniz (öğrenme oranını da değiştirebilirsiniz). Hangi ayar en iyi sonucu verir?
2. Farklı etkinleştirme işlevlerini deneyin. Hangisi en iyi çalışır?
3. Ağırlıkları ilkletmek için farklı tertipler deneyiniz. En iyi hangi yöntem işe yarar?

Tartışmalar[63]

4.4 Model Seçimi, Eksik Öğrenme ve Aşırı Öğrenme

Makine öğrenmesi bilimcileri olarak amacımız *desenleri* keşfetmektir. Ancak, verilerimizi ezberlemeden, gerçekten *genel* bir model keşfettiğimizden nasıl emin olabiliriz? Örneğin, hastaları kişilik bölünmesi durumlarına bağlayan genetik belirteçler arasında desenler aramak istediğimizi hayal edin, etiketler de {kişilik bölünmesi, hafif bilişsel bozukluk, sağlıklı} kümesinden çekilsin. Her kişinin genleri onları benzersiz bir şekilde tanımladığından (tek yumurta kardeşleri göz ardı ederek), tüm veri kümesini ezberlemek mümkündür.

Modelimizin *"Bu Bob! Onu hatırlıyorum! Kişilik bölünmesi var!"* demesini istemeyiz. Nedeni basit. Modeli ileride uyguladığımızda, modelimizin daha önce hiç görmediği hastalarla karşılaşacağız. Bizim tahminlerimiz yalnızca modelimiz gerçekten *genel* bir desen keşfetmişse yararlı olacaktır.

Daha biçimsel bir şekilde özetlersek, hedefimiz eğitim kümemizin alındığı temel popülasyondaki düzenlilikleri yakalayan desenleri keşfetmektir. Bu çabada başarılı olursak, daha önce hiç karşılaşmadığımız bireyler için bile riski başarıyla değerlendirebiliriz. Bu problem—desenlerin nasıl keşfedileceği *genelleştirmek*—makine öğrenmesinin temel problemidir.

Tehlike şu ki, modelleri eğitirken, sadece küçük bir veri örneklemine erişiyoruz. En büyük herkese açık görsel veri kümeleri yaklaşık bir milyon imge içermektedir. Daha sıklıkla, yalnızca binlerce veya on binlerce veri örneğinden öğrenmemiz gerekir. Büyük bir hastane sisteminde yüzbinlerce tıbbi kayda erişebiliriz. Sonlu örneklemlerle çalışırken, daha fazla veri topladığımızda tutmayacağı ortaya çıkan aşikar ilişkilendirmeler keşfetme riskiyle karşı karşıyayız.

Eğitim verilerimizi altta yatan dağılıma uyduğundan daha yakına uydurma olgusuna *aşırı öğrenme* ve aşırı öğrenmeyle mücadele için kullanılan tekniklere *düzenlileştirme* denir. Önceki bölümlerde, Fashion-MNIST veri kümesinde deney yaparken bu etkiyi gözlemlemiş olabilirsiniz. Deney sırasında model yapısını veya hiper parametreleri değiştirseydiniz, yeterli sayıda nöron, katman ve eğitim dönemiyle modelin, test verilerinde doğruluk kötüleşse bile, eğitim kümesinde en sonunda mükemmel doğruluğa ulaşabileceğini fark etmiş olabilirsiniz.

[63] https://discuss.d2l.ai/t/95

4.4.1 Eğitim Hatası ve Genelleme Hatası

Bu olguyu daha biçimsel olarak tartışmak için, eğitim hatası ve genelleme hatası arasında ayrım yapmamız gerekir. *Eğitim hatası*, eğitim veri kümesinde hesaplanan modelimizin hatasıdır, *genelleme hatası* ise eğer onu esas örneklemimiz ile aynı temel veri dağılımından alınan sonsuz bir ek veri örneği akışına uygularsak modelimizin göstereceği hatanın beklentisidir.

Sorunsal olarak, genelleme hatasını tam olarak hesaplayamayız. Bunun nedeni, sonsuz veri akışının hayali bir nesne olmasıdır. Uygulamada, modelimizi eğitim kümemizden rastgele seçilmiş veri örneklerinden oluşan bağımsız bir test kümesine uygulayarak genelleme hatasını *tahmin etmeliyiz*.

Aşağıdaki üç düşünce deneyi bu durumu daha iyi açıklamaya yardımcı olacaktır. Final sınavına hazırlanmaya çalışan bir üniversite öğrencisini düşünün. Çalışkan bir öğrenci, önceki yılların sınavlarını kullanarak iyi pratik yapmaya ve yeteneklerini test etmeye çalışacaktır. Bununla birlikte, geçmiş sınavlarda başarılı olmak, gerekli olduğunda başarılı olacağının garantisi değildir. Örneğin, öğrenci sınav sorularının cevaplarını ezberleyerek hazırlanmaya çalışabilir. Bu, öğrencinin birçok şeyi ezberlemesini gerektirir. Hatta geçmiş sınavların cevaplarını da mükemmel bir şekilde hatırlayabilir. Başka bir öğrenci, belirli cevapları vermenin nedenlerini anlamaya çalışarak hazırlanabilir. Çoğu durumda, ikinci öğrenci çok daha iyisini yapacaktır.

Benzer şekilde, soruları yanıtlamak için sadece bir arama tablosu kullanan bir model düşünün. İzin verilen girdiler kümesi ayrık ve makul ölçüde küçükse, o zaman belki *birçok* eğitim örneğini gördükten sonra, bu yaklaşım iyi sonuç verecektir. Yine de bu modelin, daha önce hiç görmediği örneklerle karşılaştığında rastgele tahmin etmekten daha iyisini yapma yeteneği yoktur. Gerçekte, girdi uzayları, akla gelebilecek her girdiye karşılık gelen yanıtları ezberlemek için çok büyüktür. Örneğin, 28×28'lik siyah beyaz imgeleri düşünün. Her piksel 256 gri tonlama değerlerinden birini alabiliyorsa, 256^{784} olası imge vardır. Bu, evrendeki atomlardan çok daha fazla sayıda, düşük çözünürlüklü gri tonlamalı küçük boyutlu imge olduğu anlamına gelir. Bu tür verilerle karşılaşsak bile, arama tablosunu asla saklayamayız.

Son olarak, yazı tura atmaların sonuçlarını (sınıf 0: tura, sınıf 1: yazı) mevcut olabilecek bazı bağlamsal özniteliklere göre sınıflandırmaya çalışma problemini düşünün. Paranın hilesiz olduğunu varsayalım. Hangi algoritmayı bulursak bulalım, genelleme hatası her zaman $\frac{1}{2}$ olacaktır. Bununla birlikte, çoğu algoritma için, herhangi bir özniteliğimiz olmasa bile, çekiliş şansına bağlı olarak eğitim hatamızın önemli ölçüde daha düşük olmasını beklemeliyiz! {0, 1, 1, 1, 0, 1} veri kümesini düşünün. Öznitelikiz algoritmamız, her zaman sınırlı örneklemimizden *1* olarak görünen *çoğunluk sınıfını* tahmin etmeye başvuracaktır. Bu durumda, sınıf 1'i $\frac{1}{3}$ hata ile her zaman tahmin eden model, bizim genelleme hatamızdan önemli ölçüde daha iyi olacaktır. Veri miktarını artırdıkça, turaların oranının $\frac{1}{2}$'den sapma olasılığı önemli ölçüde azalır ve eğitim hatamız genelleme hatasıyla eşleşir.

İstatistiksel Öğrenme Teorisi

Genelleme, makine öğrenmesindeki temel sorun olduğundan, birçok matematikçi ve teorisyenin hayatlarını bu olguyu tanımlamak için biçimsel teoriler geliştirmeye adadığını öğrenince şaşırmayabilirsiniz. Glivenko ve Cantelli eponymous teoremlerinde[64], eğitim hatasının genelleme hatasına yakınsadığı oranı türetmiştir. Bir dizi ufuk açıcı makalede, Vapnik ve Chervonenkis[65] bu teoriyi daha genel işlev sınıflarına genişletti. Bu çalışma istatistiksel öğrenme teorisinin temellerini attı.

[64] https://en.wikipedia.org/wiki/Glivenko%E2%80%93Cantelli_theorem
[65] https://en.wikipedia.org/wiki/Vapnik%E2%80%93Chervonenkis_theory

Şimdiye kadar ele aldığımız ve bu kitabın çoğunda bağlı kalacağımız standart gözetimli öğrenme ortamında, hem eğitim verilerinin hem de test verilerinin *bağımsız* olarak *özdeş* dağılımlardan alındığını varsayıyoruz. Bu yaygın olarak *iid varsayımı* diye adlandırılır, verilerimizi örnekleyen işlemin belleğe sahip olmadığı anlamına gelir. Diğer bir ifade ile, çekilen ikinci örnek ile üçüncü örnek, çekilen ikinci ve iki milyonuncu örnekten daha fazla ilişkili değildir.

İyi bir makine öğrenmesi bilimcisi olmak eleştirel düşünmeyi gerektirir ve şimdiden bu varsayımda boşluklar açıyor olmalısınız, varsayımın başarısız olduğu yaygın durumlar ortaya çıkar. Ya UCSF Tıp Merkezi'ndeki hastalardan toplanan verilerle bir ölüm riski tahmincisi eğitirsek ve bunu Massachusetts General Hospital'daki hastalara uygularsak? Bu dağılımlar kesinlikle özdeş değildir. Dahası, örneklemler zamanla ilişkilendirilebilir. Ya Tweetlerin konularını sınıflandırıyorsak? Haber döngüsü, tartışılan konularda herhangi bir bağımsızlık varsayımını ihlal ederek zamansal bağımlılıklar yaratacaktır.

Bazen iid (böd - bağımsız ve özdeşçe dağılmış) varsayımından küçük ihlallerle uzaklaşabiliriz ve modellerimiz oldukça iyi çalışmaya devam edecektir. Sonuçta, neredeyse her gerçek dünya uygulaması, böd (iid) varsayımın en azından bazı küçük ihlallerini içerir ama yine de yüz tanıma, konuşma tanıma ve dil çevirisi benzeri uygulamalar için bir çok yararlı araca sahibiz.

Diğer ihlallerin sorun yaratacağı kesindir. Örneğin, bir yüz tanıma sistemini sadece üniversite öğrencileri ile eğitmeyi denediğimizi ve sonra onu bir huzurevi popülasyonunda yaşlılığı izlemede bir araç olarak kullanmak istediğimizi düşünün. Üniversite öğrencileri yaşlılardan oldukça farklı görünme eğiliminde olduklarından, bunun iyi sonuç vermesi olası değildir.

Sonraki bölümlerde, böd (iid) varsayım ihlallerinden kaynaklanan sorunları tartışacağız. Şimdilik, böd (iid) varsayımı sayesinde bile genellemeyi anlamak zorlu bir problemdir. Dahası, derin sinir ağlarının neden bu kadar iyi genelleştirdiğini açıklayabilecek kesin teorik temellerin aydınlatılması öğrenme teorisindeki en büyük zihinlerin canını sıkmaya devam ediyor.

Modellerimizi eğittiğimizde, eğitim verilerine mümkün olduğu kadar uyan bir işlev aramaya çalışırız. İşlev, gerçek ilişkilendirmeler kadar kolay sahte desenleri yakalayabilecek kadar esnekse, görünmeyen verileri iyi genelleyen bir model üretmeden *çok iyi* performans gösterebilir. Bu tam olarak kaçınmak veya en azından kontrol etmek istediğimiz şeydir. Derin öğrenmedeki tekniklerin çoğu, aşırı öğrenmeye karşı korumayı amaçlayan sezgisel yöntemler ve hilelerdir.

Model Karmaşıklığı

Basit modellere ve bol veriye sahip olduğumuzda, genelleme hatasının eğitim hatasına benzemesini bekleriz. Daha karmaşık modellerle ve daha az örnekle çalıştığımızda, eğitim hatasının azalmasını, ancak genelleme açığının artmasını bekliyoruz. Model karmaşıklığını tam olarak oluşturan şey karmaşık bir konudur. Bir modelin iyi bir genelleme yapıp yapmayacağını birçok etken belirler. Örneğin, daha fazla parametresi olan bir model daha karmaşık kabul edilebilir. Parametrelerini daha geniş bir değer aralığından alabilen bir model daha karmaşık olabilir. Genellikle sinir ağlarında, daha fazla eğitim yinelemesi yapan bir modeli daha karmaşık olarak ve *erken durdurma*ya (daha az eğitim yinelemesi) maruz kalan bir modeli daha az karmaşık olarak düşünürüz.

Aşırı farklı model sınıflarının üyeleri arasındaki karmaşıklığı karşılaştırmak zor olabilir (örneğin, karar ağaçlarına karşı sinir ağları). Şimdilik, basit bir pratik kural oldukça kullanışlıdır: Gelişigüzel gerçekleri kolayca açıklayabilen bir modeli istatistikçiler karmaşık olarak görürken, yalnızca sınırlı bir ifade gücüne sahip olan ancak yine de verileri iyi açıklamayı başaran model muhtemelen gerçeğe daha yakındır. Felsefede bu, Popper'ın bilimsel bir teorinin yanlışlanabilirlik kriteriyle yakından ilgilidir: Bir teori, verilere uyuyorsa ve onu çürütebilecek belirli testler

varsa iyidir. Bu önemlidir, çünkü tüm istatistiksel tahminler *olay sonrası*dır (post hoc), yani gerçekleri gözlemledikten sonra tahmin ederiz, dolayısıyla ilgili yanılgılara karşı savunmasız kalırız. Şimdilik felsefeyi bir kenara bırakıp daha somut meselelere bağlı kalacağız.

Bu bölümde, size biraz sezgi vermek için, bir model sınıfının genelleştirilebilirliğini etkileme eğiliminde olan birkaç etmene odaklanacağız:

1. Ayarlanabilir parametrelerin sayısı. Bazen *serbestlik derecesi* olarak adlandırılan ayarlanabilir parametrelerin sayısı büyük olduğunda, modeller aşırı öğrenmeye daha duyarlı olma eğilimindedir.
2. Parametrelerin aldığı değerler. Ağırlıklar daha geniş bir değer aralığı alabildiğinde, modeller aşırı öğrenmeye daha duyarlı olabilir.
3. Eğitim örneklerinin sayısı. Modeliniz basit olsa bile, yalnızca bir veya iki örnek içeren bir veri kümesinde aşırı öğrenmek çok kolaydır. Ancak milyonlarca örnekli bir veri kümesini aşırı öğrenmek, son derece esnek bir model gerektirir.

4.4.2 Model Seçimi

Makine öğrenmesinde, genellikle birkaç aday modeli değerlendirdikten sonra son modelimizi seçeriz. Bu işleme *model seçimi* denir. Bazen karşılaştırmaya konu olan modeller doğaları gereği temelden farklıdır (örneğin, karar ağaçları ve doğrusal modeller). Diğer zamanlarda, farklı hiper parametre ayarlarıyla eğitilmiş aynı model sınıfının üyelerini karşılaştırıyoruz.

Örneğin, MLP'de, modelleri farklı sayıda gizli katman, farklı sayıda gizli birim ve her gizli katmana uygulanan çeşitli etkinleştirme işlevleri seçenekleriyle karşılaştırma yapmak isteyebiliriz. Aday modellerimiz arasında en iyisini belirlemek için, genellikle bir geçerleme veri kümesi kullanırız.

Geçerleme Veri Kümesi

Prensip olarak, tüm hiper parametrelerimizi seçinceye kadar test kümemize dokunmamalıyız. Model seçim sürecinde test verilerini kullanacak olursak, test verilerini aşırı öğrenme riski vardır. O zaman ciddi bir belaya gireriz. Eğitim verilerimizi aşırı öğrenirsek, test verileri üzerinden değerlendirme her zaman bizi dürüst tutmak için oradadır. Ama test verilerini aşırı öğrenirsek, nasıl bilebiliriz?

Bu nedenle, model seçimi için asla test verilerine güvenmemeliyiz. Yine, model seçimi için yalnızca eğitim verilerine güvenemeyiz çünkü modeli eğitmek için kullandığımız verilerdeki genelleme hatasını tahmin edemiyoruz.

Pratik uygulamalarda resim bulanıklaşır. İdeal olarak, en iyi modeli değerlendirmek veya az sayıda modeli birbiriyle karşılaştırmak için test verilerine yalnızca bir kez dokunsak da, gerçek dünya test verileri nadiren tek bir kullanımdan sonra atılır. Her deney turu için nadiren yeni bir test kümesi alabiliriz.

Bu sorunu ele almaya yönelik yaygın uygulama, verilerimizi eğitim ve test veri kümelerine bir de *geçerleme veri kümesi (ya da geçerleme kümesi)* ekleyerek üçe bölmektir. Sonuç, geçerleme ve test verileri arasındaki sınırların endişe verici derecede bulanık olduğu belirsiz bir uygulamadır. Açıkça aksi belirtilmedikçe, bu kitaptaki deneylerde, gerçek test kümeleri olmadan, gerçekte haklı olarak eğitim verileri ve geçerleme verileri olarak adlandırılması gerekenlerle çalışıyoruz. Bu nedenle, bu kitaptaki her deneyde bildirilen doğruluk, gerçekte geçerleme doğruluğudur ve gerçek bir test kümesi doğruluğu değildir.

K-Kat Çapraz Geçerleme

Eğitim verisi kıt olduğunda, uygun bir geçerleme kümesi oluşturmak için yeterli veriyi tutmaya bile imkanımız olmayabilir. Bu soruna popüler bir çözüm, *K-kat çapraz geçerleme* kullanmaktır. Burada, esas eğitim verileri K tane çakışmayan alt kümeye bölünmüştür. Ardından, model eğitimi ve geçerleme, her seferinde $K-1$ tane alt küme üzerinde eğitim ve farklı bir alt kümede (bu turda eğitim için kullanılmayan) geçerleme olmak üzere, K kez yürütülür. Son olarak, eğitim ve geçerleme hataları K deneyden elde edilen sonuçların ortalaması alınarak tahmin edilir.

4.4.3 Eksik Öğrenme mi veya Aşırı Öğrenme mi?

Eğitim ve geçerleme hatalarını karşılaştırdığımızda, iki genel duruma dikkat etmek istiyoruz. Birincisi, eğitim hatamızın ve geçerleme hatamızın hem önemli hem de aralarında küçük bir boşluk olduğu durumlara dikkat etmek istiyoruz. Model eğitim hatasını azaltamıyorsa, bu, modelimizin modellemeye çalıştığımız deseni yakalamak için çok basit (yani, yeterince ifade edici değil) olduğu anlamına gelebilir. Dahası, eğitim ve geçerleme hatalarımız arasındaki *genelleme boşluğu* küçük olduğundan, daha karmaşık bir modelle kurtulabileceğimize inanmak için nedenimiz var. Bu olgu, *eksik öğrenme* olarak bilinir.

Öte yandan, yukarıda tartıştığımız gibi, eğitim hatamızın geçerleme hatamızdan önemli ölçüde düşük olduğu ve ciddi şekilde *aşırı öğrenme* gösterdiği durumlara dikkat etmek istiyoruz. Aşırı öğrenmenin her zaman kötü bir şey olmadığını unutmayın. Özellikle derin öğrenmede, en iyi tahminci modellerin çoğu zaman eğitim verilerinde harici tutulan verilerden çok daha iyi performans gösterdiği iyi bilinmektedir. Sonuç olarak, genellikle eğitim ve geçerleme hataları arasındaki boşluktan daha ziyade geçerleme hatasını önemsiyoruz.

Aşırı mı yoksa yetersiz mi öğrendiğimiz, hem modelimizin karmaşıklığına hem de mevcut eğitim veri kümelerinin boyutuna bağlı olabilir, bu iki konuyu aşağıda tartışıyoruz.

Model Karmaşıklığı

Aşırı öğrenme ve model karmaşıklığı hakkında bazı klasik sezgileri göstermek için, polinomların kullanıldığı bir örnek veriyoruz. Tek bir x özniteliği ve buna karşılık gelen gerçel değerli bir y etiketinden oluşan eğitim verileri verildiğinde, d dereceli polinomunu bulmaya çalışıyoruz.

$$\hat{y} = \sum_{i=0}^{d} x^i w_i \tag{4.4.1}$$

Amacımız y etiketlerini tahmin etmek. Bu, özniteliklerimizin x'in kuvvetleri tarafından, modelin ağırlıklarının w_i tarafından ve tüm x için $x^0 = 1$ olduğu ek girdinin w_0 tarafından verildiği doğrusal bir regresyon problemidir. Bu sadece doğrusal bir regresyon problemi olduğundan, kayıp fonksiyonumuz olarak hata karesini kullanabiliriz.

Daha yüksek dereceden bir polinom fonksiyonu, daha düşük dereceli bir polinom fonksiyonundan daha karmaşıktır, çünkü yüksek dereceli polinom daha fazla parametreye sahiptir ve model fonksiyonunun seçim aralığı daha geniştir. Eğitim veri kümesini sabitlersek, yüksek dereceli polinom fonksiyonları, düşük dereceli polinomlara göre her zaman daha düşük (en kötü durumda, eşit) eğitim hatası elde etmelidir. Aslında, veri örneklerinin her biri farklı x değerlerine sahip olduğunda, veri örneklerinin sayısına eşit dereceye sahip bir polinom fonksiyonu, eğitim kümesine mükemmel şekilde oturabilir. Polinom derecesi ile eksik ve aşırı öğrenme arasındaki ilişkiyi Fig. 4.4.1 şeklinde görselleştiriyoruz.

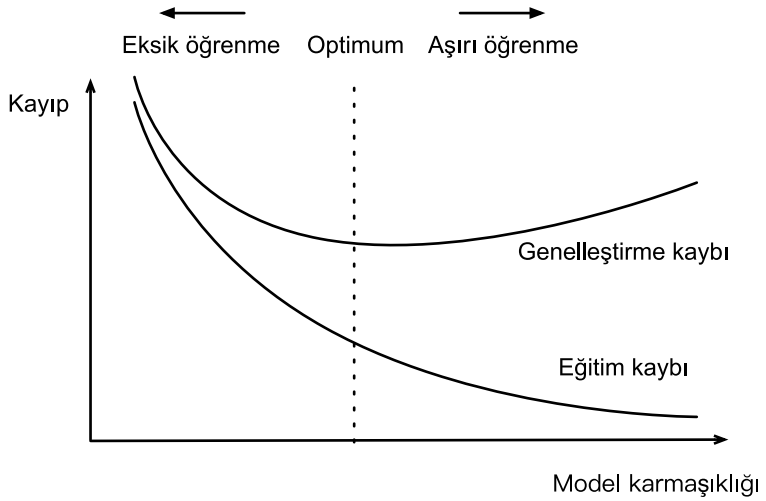


Fig. 4.4.1: Eksik ve aşırı öğrenmenin model karmaşıklığına etkisi

Veri Kümesinin Boyutu

Akılda tutulması gereken diğer büyük husus, veri kümesinin boyutudur. Modelimizi sabitlediğimizde, eğitim veri kümesinde ne kadar az numuneye sahip olursak, aşırı öğrenme ile karşılaşma olasılığımız o kadar (ve daha şiddetli) olacaktır. Eğitim verisi miktarını artırdıkça, genelleme hatası tipik olarak azalır. Dahası, genel olarak, daha fazla veri asla zarar vermez. Sabit bir görev ve veri dağılımı için, genellikle model karmaşıklığı ve veri kümesi boyutu arasında bir ilişki vardır. Daha fazla veri verildiğinde, karlı bir şekilde daha karmaşık bir model öğrenmeye teşebbüs edebiliriz. Yeterli veri olmadığında, daha basit modellerin alt edilmesi daha zor olabilir. Birçok görev için, derin öğrenme yalnızca binlerce eğitim örneği mevcut olduğunda doğrusal modellerden daha iyi performans gösterir. Kısmen, derin öğrenme mevcut başarısını, İnternet şirketleri, ucuz depolama, bağlı cihazlar ve ekonominin geniş dijitalleşmesi nedeniyle mevcut büyük veri kümelerinin bolluğuna borçludur.

4.4.4 Polinom Regresyon

Polinomları verilere oturturken artık bu kavramları etkileşimli olarak keşfedebiliriz.

```
import math
import numpy as np
import torch
from torch import nn
from d2l import torch as d2l
```

Veri Kümesini Üretme

Önce verilere ihtiyacımız var. x verildiğinde, eğitim ve test verilerinde ilgili etiketleri oluşturmak için aşağıdaki kübik polinomu kullanacağız:

$$y = 5 + 1.2x - 3.4\frac{x^2}{2!} + 5.6\frac{x^3}{3!} + \epsilon \text{ öyle ki } \epsilon \sim \mathcal{N}(0, 0.1^2). \tag{4.4.2}$$

Gürültü terimi ϵ, ortalaması 0 ve standart sapması 0.1 olan normal bir dağılıma uyar. Eniyileme için, genellikle çok büyük gradyan değerlerinden veya kayıplardan kaçınırız. Bu nedenden *öznitelikler* x^i'dan $\frac{x^i}{i!}$'ya ölçeklendirilir. Bu bizim geniş i kuvvetleri için çok geniş değerlerden kaçınmamıza izin verir. Eğitim ve test kümelerinin her biri için 100 örnek sentezleyeceğiz.

```
max_degree = 20  # Polinomun maksimum derecesi
n_train, n_test = 100, 100  # Eğitim ve test veri kumesi boyutları
true_w = np.zeros(max_degree)  # Boş alan tahsisi
true_w[0:4] = np.array([5, 1.2, -3.4, 5.6])

features = np.random.normal(size=(n_train + n_test, 1))
np.random.shuffle(features)
poly_features = np.power(features, np.arange(max_degree).reshape(1, -1))
for i in range(max_degree):
    poly_features[:, i] /= math.gamma(i + 1)  # `gamma(n)` = (n-1)!
# `labels`'in şekli: (`n_train` + `n_test`,)
labels = np.dot(poly_features, true_w)
labels += np.random.normal(scale=0.1, size=labels.shape)
```

Yine, `poly_features` içinde depolanan tek terimler, $\Gamma(n) = (n-1)!$ olmak üzere, gamma işlevi tarafından yeniden ölçeklendirilir. Oluşturulan veri kümesinden ilk 2 örneğe bir göz atın. 1 değeri teknik olarak bir özniteliktir, yani ek girdiye karşılık gelen sabit özniteliktir.

```
# NumPy ndarray'lardan tensorlere cevirme
true_w, features, poly_features, labels = [torch.tensor(x, dtype=
    torch.float32) for x in [true_w, features, poly_features, labels]]
```

```
features[:2], poly_features[:2, :], labels[:2]
```

```
(tensor([[-1.4127],
         [-0.8767]]),
 tensor([[ 1.0000e+00, -1.4127e+00,  9.9782e-01, -4.6987e-01,  1.6594e-01,
          -4.6884e-02,  1.1039e-02, -2.2277e-03,  3.9338e-04, -6.1747e-05,
           8.7228e-06, -1.1202e-06,  1.3188e-07, -1.4331e-08,  1.4460e-09,
          -1.3619e-10,  1.2024e-11, -9.9918e-13,  7.8418e-14, -5.8305e-15],
         [ 1.0000e+00, -8.7669e-01,  3.8430e-01, -1.1230e-01,  2.4614e-02,
          -4.3158e-03,  6.3060e-04, -7.8977e-05,  8.6549e-06, -8.4307e-07,
           7.3912e-08, -5.8907e-09,  4.3036e-10, -2.9023e-11,  1.8174e-12,
          -1.0622e-13,  5.8203e-15, -3.0015e-16,  1.4619e-17, -6.7455e-19]]),
 tensor([-2.5969,  1.7821]))
```

Model Eğitimi ve Testi

Önce belirli bir veri kümesindeki kaybı değerlendirmek için bir fonksiyon uygulayalım.

```
def evaluate_loss(net, data_iter, loss):  #@save
    """Verilen veri kümesindeki bir modelin kaybını değerlendirin."""
    metric = d2l.Accumulator(2)  # Kayıpların toplamı, örnek sayısı
    for X, y in data_iter:
        out = net(X)
        y = y.reshape(out.shape)
        l = loss(out, y)
        metric.add(l.sum(), l.numel())
    return metric[0] / metric[1]
```

Şimdi eğitim işlevini tanımlayın.

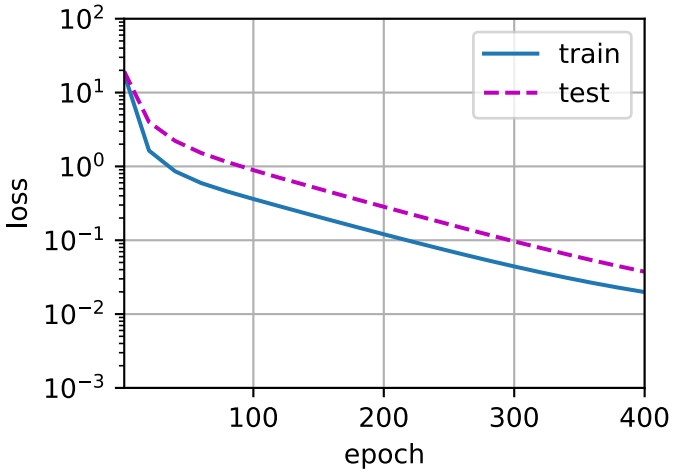
```
def train(train_features, test_features, train_labels, test_labels,
          num_epochs=400):
    loss = nn.MSELoss(reduction='none')
    input_shape = train_features.shape[-1]
    # Polinomdaki öznitelikllerde zaten sağladığımız için ek girdiyi yok sayın
    net = nn.Sequential(nn.Linear(input_shape, 1, bias=False))
    batch_size = min(10, train_labels.shape[0])
    train_iter = d2l.load_array((train_features, train_labels.reshape(-1,1)),
                                batch_size)
    test_iter = d2l.load_array((test_features, test_labels.reshape(-1,1)),
                               batch_size, is_train=False)
    trainer = torch.optim.SGD(net.parameters(), lr=0.01)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss', yscale='log',
                            xlim=[1, num_epochs], ylim=[1e-3, 1e2],
                            legend=['train', 'test'])
    for epoch in range(num_epochs):
        d2l.train_epoch_ch3(net, train_iter, loss, trainer)
        if epoch == 0 or (epoch + 1) % 20 == 0:
            animator.add(epoch + 1, (evaluate_loss(net, train_iter, loss),
                                     evaluate_loss(net, test_iter, loss)))
    print('weight:', net[0].weight.data.numpy())
```

Üçüncü Dereceden Polinom Fonksiyon Uydurma (Normal)

İlk olarak, veri oluşturma işlevi ile aynı dereceye sahip üçüncü dereceden bir polinom işlevini kullanarak başlayacağız. Sonuçlar bu modelin eğitim ve test kaybının, birlikte, etkili biçimde düşürülebildiği göstermektedir. Eğitilen model parametreleri de $w = [5, 1.2, -3.4, 5.6]$ gerçek değerlerine yakındır.

```
# Polinom özniteliklerden ilk dört boyutu alın, örn., 1, x, x^2/2!, x^3/3!
train(poly_features[:n_train, :4], poly_features[n_train:, :4],
      labels[:n_train], labels[n_train:])
```

```
weight: [[ 4.9413424  1.3797615 -3.2177389  5.0117817]]
```

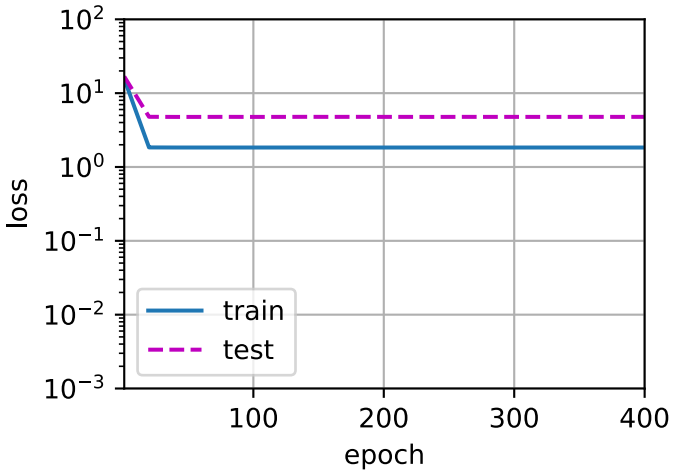


Doğrusal Fonksiyon Uydurma (Eksik Öğrenme)

Doğrusal fonksiyon oturtmaya başka bir bakış atalım. Erken dönemlerdeki düşüşten sonra, bu modelin eğitim kaybını daha da düşürmek zorlaşır. Son dönem yinelemesi tamamlandıktan sonra, eğitim kaybı hala yüksektir. Doğrusal olmayan desenleri (buradaki üçüncü dereceden polinom işlevi gibi) uydurmak için kullanıldığında doğrusal modeller eksik öğrenme gösterme eğilimindedir.

```
# Polinom özniteliklerden ilk iki boyutu alın, örn., 1, x.
train(poly_features[:n_train, :2], poly_features[n_train:, :2],
      labels[:n_train], labels[n_train:])
```

```
weight: [[3.9534883 2.506134 ]]
```

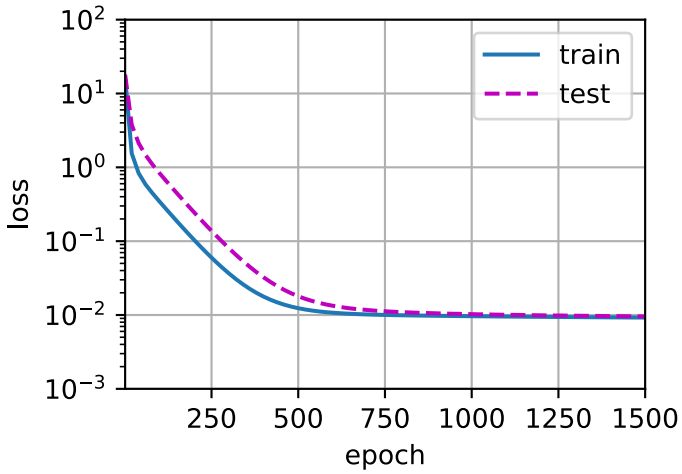


Yüksek Dereceli Polinom İşlevi Uydurmak (Aşırı Öğrenme)

Şimdi modeli çok yüksek dereceli bir polinom kullanarak eğitmeye çalışalım. Burada, daha yüksek dereceli katsayıların sıfıra yakın değerlere sahip olması gerektiğini öğrenmek için yeterli veri yoktur. Sonuç olarak, aşırı karmaşık modelimiz o kadar duyarlıdır ki eğitim verisindeki gürültüden etkilenir. Eğitim kaybı etkili bir şekilde azaltılabilse de, test kaybı hala çok daha yüksektir. Karmaşık modelin verilere fazla uyduğunu gösterir.

```
# Polinom oznitelikleri̇rden butun boyutlarini alin.
train(poly_features[:n_train, :], poly_features[n_train:, :],
      labels[:n_train], labels[n_train:], num_epochs=1500)
```

```
weight: [[ 4.9981842e+00  1.2709697e+00 -3.4211504e+00  5.1789103e+00
   3.2724470e-02  1.3359066e+00  1.3480692e-01  6.5162614e-02
   2.1408103e-01  1.8083531e-02  1.8239583e-01  1.9697568e-01
  -1.1200300e-02 -3.2266791e-03  1.8600665e-01  1.8732022e-02
   6.6375777e-02 -1.4465892e-01  1.7397648e-01 -1.3597389e-02]]
```



Daha ileriki bölümlerde, aşırı öğrenme problemlerini ve bunlarla başa çıkma yöntemlerini, örneğin ağırlık sönümü ve hattan düşürme gibi, tartışmaya devam edeceğiz.

4.4.5 Özet

- Genelleme hatası eğitim hatasına dayalı olarak tahmin edilemediğinden, basitçe eğitim hatasını en aza indirmek, genelleme hatasında bir azalma anlamına gelmeyecektir. Makine öğrenmesi modellerinin, genelleme hatasını en aza indirgerken aşırı öğrenmeye karşı koruma sağlamak için dikkatli olması gerekir.
- Model seçimi için bir geçerleme kümesi, çok serbest kullanılmaması koşuluyla, kullanılabilir.
- Eksik öğrenme, modelin eğitim hatasını azaltamadığı anlamına gelir. Model eğitim hatası geçerleme kümesi hatasından çok daha düşük olduğu zaman da aşırı öğrenme vardır.
- Uygun şekilde karmaşık bir model seçmeli ve yetersiz eğitim örneklemleri kullanmaktan kaçınmalıyız.

4.4.6 Alıştırmalar

1. Polinom regresyon problemini tam olarak çözebilir misiniz? İpucu: Doğrusal cebir kullanın.
2. Polinomlarda model seçimini düşünün:
    1. Model karmaşıklığına (polinom derecesi) karşı eğitim kaybını çizin. Ne gözlemliyorsunuz? Eğitim kaybını 0'a indirgemek için kaçıncı dereceden polinoma ihtiyacınız vardır?
    2. Bu durumda test kaybını çizin.
    3. Aynı şekli veri miktarının bir fonksiyonu olarak oluşturun.
3. x^i polinom özniteliklerinin normalleştirmesini, $1/i!$, düşürürseniz ne olur? Bunu başka bir şekilde düzeltebilir misiniz?
4. Sıfır genelleme hatası görmeyi bekleyebilir misiniz?

Tartışmalar[66]

4.5 Ağırlık Sönümü

Artık aşırı öğrenme sorununu tanımladığımıza göre, modelleri düzenlileştirmek için bazı standart tekniklerle tanışabiliriz. Dışarı çıkıp daha fazla eğitim verisi toplayarak aşırı öğrenmeyi her zaman azaltabileceğimizi hatırlayın. Bu maliyetli, zaman alıcı veya tamamen kontrolümüz dışında olabilir ve kısa vadede koşturmayı imkansız hale getirebilir. Şimdilik, kaynaklarımızın izin verdiği kadar yüksek kaliteli veriye sahip olduğumuzu ve düzenlileştirmek tekniklerine odaklandığımızı varsayabiliriz.

Polinom bağlanım örneğimizde (Section 4.4), modelimizin kapasitesini sadece yerleştirilmiş polinomun derecesini ayarlayarak sınırlayabileceğimizi hatırlayın. Gerçekten de, özniteliklerin sayısını sınırlamak, aşırı öğrenme ile mücadele için popüler bir tekniktir. Bununla birlikte, öznitelikleri bir kenara atmak, iş için çok patavatsız bir araç olabilir. Polinom bağlanım örneğine bağlı kalarak, yüksek boyutlu girdilerde neler olabileceğini düşünün. Polinomların çok değişkenli verilere doğal uzantıları, basitçe değişkenlerin güçlerinin çarpımı, *tek terimli* olarak adlandırılır. Bir tek terimliğin derecesi, güçlerin toplamıdır. Örneğin, $x_1^2 x_2$ ve $x_3 x_5^2$'nin her ikisi de 3. dereceden tek terimlidir.

d derecesine sahip terimlerin sayısının, d büyüdükçe hızla arttığını unutmayın. k tane değişken verildiğinde, d derecesindeki tek terimli sayıların sayısı (örn., d'den çok seçmeli k) $\binom{k-1+d}{k-1}$ olur. Mesela 2'den 3'e kadar olan küçük derece değişiklikler bile modelimizin karmaşıklığını önemli ölçüde artırır. Bu nedenle, işlev karmaşıklığını ayarlamak için genellikle daha ince ayarlı bir araca ihtiyaç duyarız.

[66] https://discuss.d2l.ai/t/97

4.5.1 Normlar ve Ağırlık Sönümü

Hem L_2 hem de L_1 normunu daha önce Section 2.3.10 ünitesinde tanıtmıştık, ikisi de genel L_p normunun özel durumlarıdır. *Ağırlık sönümü* (genellikle L_2 düzenlileştirme olarak adlandırılır), parametrik makine öğrenmesi modellerini düzenlemek için en yaygın kullanılan teknik olabilir. Teknik, tüm f işlevleri arasında, $f = 0$ işlevinin (tüm girdilere 0 değerini atayarak) bir anlamda *en basit* olduğu ve sıfırdan uzaklığına göre bir fonksiyonun karmaşıklığını ölçebileceğimiz temel sezgisiyle motive edilir. Fakat bir fonksiyon ile sıfır arasındaki mesafeyi ne kadar kesinlikle ölçmeliyiz? Tek bir doğru cevap yok. Aslında, fonksiyonel analiz bölümleri ve Banach uzayları teorisi de dahil olmak üzere matematiğin tüm dalları, bu sorunu yanıtlamaya adanmıştır.

Basit bir yorumlama, bir $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$ doğrusal fonksiyonunun karmaşıklığını, ağırlık vektörünün bir normu ile ölçmek olabilir, örneğin, $\|\mathbf{w}\|^2$ gibi. Küçük bir ağırlık vektörü sağlamanın en yaygın yöntemi, onun normunu, kaybın en aza indirilmesi problemine bir ceza terimi olarak eklemektir. Böylece orijinal amaç fonksiyonumuzu, *eğitim etiketlerindeki tahmin kaybını en aza indirir*, yeni bir amaç fonksiyonu ile değiştiriyor, *tahmin kaybı ile ceza teriminin toplamını en aza indiriyoruz*. Şimdi, ağırlık vektörümüz çok büyürse, öğrenme algoritmamız eğitim hatasını en aza indirmek yerine $\|\mathbf{w}\|^2$ ağırlık normunu en aza indirmeye odaklanabilir. Bu tam olarak istediğimiz şey. Bu şeyleri kodda örneklendirmek için, Section 3.1 içindeki önceki doğrusal regresyon örneğimizi canlandıralım. Kaybımız şöyle verilir:

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2. \tag{4.5.1}$$

$\mathbf{x}^{(i)}$'in öznitelikler, $y^{(i)}$'nin her i veri örneği için etiket ve $(\mathbf{w}, b)$ değerlerinin sırasıyla ağırlık ve ek girdi parametreleri olduğunu hatırlayın. Ağırlık vektörünün büyüklüğünü cezalandırmak için, bir şekilde $\|\mathbf{w}\|^2$'yi kayıp fonksiyonuna eklemeliyiz, ancak model bu yeni ilave ceza ile standart kaybı nasıl bir ödünleşmeye sokmalıdır? Pratikte, bu ödünleşme, geçerleme verilerini kullanarak öğrendiğimiz negatif olmayan bir hiper parametre, yani *düzenlileştirme sabiti* λ ile karakterize ediyoruz:

$$L(\mathbf{w}, b) + \frac{\lambda}{2}\|\mathbf{w}\|^2, \tag{4.5.2}$$

$\lambda = 0$ için, esas kayıp fonksiyonumuzu elde ediyoruz. $\lambda > 0$ için, $\|\mathbf{w}\|$'in boyutunu kısıtlıyoruz. Kural olarak 2'ye böleriz: İkinci dereceden bir fonksiyonun türevini aldığımızda, 2 ve $1/2$ birbirini götürür ve güncelleme ifadesinin güzel ve basit görünmesini sağlar. Dikkatli okuyucular, neden standart normla (yani Öklid mesafesi) değil de kare normla çalıştığımızı merak edebilirler. Bunu hesaplama kolaylığı için yapıyoruz. L_2 normunun karesini alarak, karekökü kaldırıyoruz ve ağırlık vektörünün her bir bileşeninin karelerinin toplamını bakıyoruz. Bu, cezanın türevini hesaplamayı kolaylaştırır: Türevlerin toplamı, toplamın türevine eşittir.

Dahası, neden ilk olarak L2 normuyla çalıştığımızı ve örneğin L1 normuyla çalışmadığımızı sorabilirsiniz. Aslında, diğer seçenekler geçerli ve istatistiksel bakımdan popülerdir. L_2-regresyonlu doğrusal modeller klasik *sırt regresyon* algoritmasını oluştururken, L_1-regresyonlu doğrusal regresyon benzer şekilde istatistikte temel bir modeldir, ki popüler olarak *kement regresyon* diye bilinir.

L_2 normuyla çalışmanın bir nedeni, ağırlık vektörünün büyük bileşenlerine daha büyük cezalar verilmesidir. Bu, öğrenme algoritmamızı, ağırlığı daha fazla sayıda özniteliğe eşit olarak dağıtan modellere doğru yönlendirir. Uygulamada, bu onları tek bir değişkendeki ölçüm hatasına karşı daha gürbüz hale getirebilir. Aksine, L_1 cezaları, diğer ağırlıkları sıfıra yaklaştırarak temizler ve ağırlıkları küçük bir öznitelik kümesine yoğunlaştıran modellere yol açar. Buna *öznitelik seçme* denir ve başka nedenlerden dolayı arzu edilebilir.

(3.1.10) içindeki gösterimi kullanırsak, L_2 ile düzenlileştirilmiş regresyon için rasgele gradyan iniş güncellemeleri aşağıdaki gibidir:

$$\mathbf{w} \leftarrow (1 - \eta\lambda)\,\mathbf{w} - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}}\mathbf{x}^{(i)}\left(\mathbf{w}^\top\mathbf{x}^{(i)} + b - y^{(i)}\right). \tag{4.5.3}$$

Daha önce olduğu gibi, **w**'yi tahminimizin gözlemden farklı olduğu miktara göre güncelliyoruz. Bununla birlikte, **w**'nin boyutunu sıfıra doğru küçültürüz. Bu nedenle, bu yönteme bazen "ağırlık sönümü" adı verilir: Yalnızca ceza terimi verildiğinde, optimizasyon algoritmamız, eğitimin her adımında ağırlığı *söndürür*. Öznitelik seçiminin tersine, ağırlık sönümü bize bir fonksiyonun karmaşıklığını ayarlamak için süregelen bir mekanizma sunar. Daha küçük λ değerleri, daha az kısıtlanmış **w**'ye karşılık gelirken, daha büyük λ değerleri **w**'yi daha önemli ölçüde kısıtlar.

Karşılık gelen ek girdi cezasını, b^2'yi, dahil edip etmememiz, uygulamalar arasında değişebilir ve hatta bir sinir ağının katmanları arasında değişebilir. Genellikle, bir ağın çıktı katmanının ek girdi terimini düzenli hale getirmeyiz.

4.5.2 Yüksek Boyutlu Doğrusal Regresyon

Basit bir sentetik örnekle ağırlık sönümlenmesinin faydalarını gösterebiliriz. İlk önce, daha önce olduğu gibi biraz veri oluşturuyoruz:

```
%matplotlib inline
import torch
from torch import nn
from d2l import torch as d2l
```

İlk olarak, önceki gibi biraz veri üretiyoruz

$$y = 0.05 + \sum_{i=1}^{d} 0.01x_i + \epsilon \text{ öyleki } \epsilon \sim \mathcal{N}(0, 0.01^2). \tag{4.5.4}$$

Etiketimizi, sıfır ortalama ve 0.01 standart sapma ile Gauss gürültüsü ile bozulmuş girdilerimizin doğrusal bir fonksiyonu olarak seçiyoruz. Aşırı eğitimin etkilerini belirgin hale getirmek için problemimizin boyutunu $d = 200$'e çıkarabilir ve sadece 20 örnek içeren küçük bir eğitim kümesi ile çalışabiliriz.

```
n_train, n_test, num_inputs, batch_size = 20, 100, 200, 5
true_w, true_b = torch.ones((num_inputs, 1)) * 0.01, 0.05
train_data = d2l.synthetic_data(true_w, true_b, n_train)
train_iter = d2l.load_array(train_data, batch_size)
test_data = d2l.synthetic_data(true_w, true_b, n_test)
test_iter = d2l.load_array(test_data, batch_size, is_train=False)
```

4.5.3 Sıfırdan Uygulama

Aşağıda, orijinal hedef işlevine basitçe kare L_2 cezasını ekleyerek, ağırlık sönümünü sıfırdan uygulayacağız.

Model Parametrelerini İlkletme

İlk olarak, model parametrelerimizi rastgele ilkletmek için bir fonksiyon tanımlayacağız.

```
def init_params():
    w = torch.normal(0, 1, size=(num_inputs, 1), requires_grad=True)
    b = torch.zeros(1, requires_grad=True)
    return [w, b]
```

L_2 Norm Cezasının Tanımlanması

Belki de bu cezayı uygulamanın en uygun yolu, tüm terimlerin karesini almak ve bunları toplamaktır.

```
def l2_penalty(w):
    return torch.sum(w.pow(2)) / 2
```

Eğitim Döngülerini Tanımlama

Aşağıdaki kod, eğitim kümesine bir model uyarlar ve onu test kümesinde değerlendirir. Doğrusal ağ ve kare kayıp Chapter 3 bölümünden bu yana değişmedi, bu yüzden onları sadece `d2l.linreg` ve `d2l.squared_loss` yoluyla içe aktaracağız. Buradaki tek değişiklik, kaybımızın artık ceza terimi içermesidir.

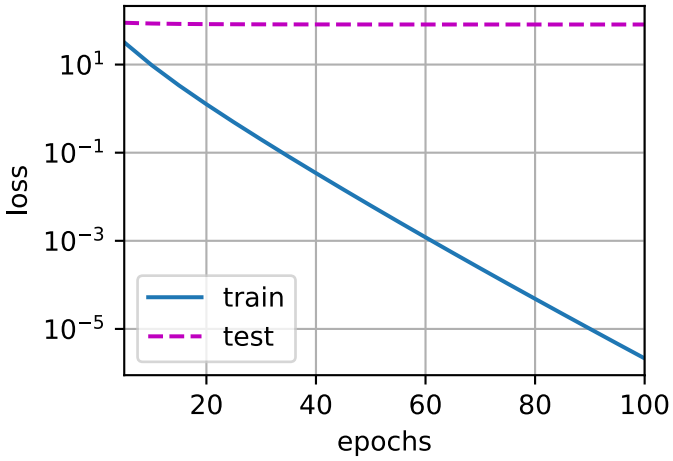
```
def train(lambd):
    w, b = init_params()
    net, loss = lambda X: d2l.linreg(X, w, b), d2l.squared_loss
    num_epochs, lr = 100, 0.003
    animator = d2l.Animator(xlabel='epochs', ylabel='loss', yscale='log',
                            xlim=[5, num_epochs], legend=['train', 'test'])
    for epoch in range(num_epochs):
        for X, y in train_iter:
            # L2 normu ceza terimi eklendi ve yayma,`l2_penalty(w)`'yi
            # uzunluğu `batch_size` olan bir vektör yapıyor.
            l = loss(net(X), y) + lambd * l2_penalty(w)
            l.sum().backward()
            d2l.sgd([w, b], lr, batch_size)
        if (epoch + 1) % 5 == 0:
            animator.add(epoch + 1, (d2l.evaluate_loss(net, train_iter, loss),
                                     d2l.evaluate_loss(net, test_iter, loss)))
    print('w\'nin L2 normu:', torch.norm(w).item())
```

Düzenleştirmesiz Eğitim

Şimdi bu kodu `lambd = 0` ile çalıştırarak ağırlık sönümünü devre dışı bırakıyoruz. Kötü bir şekilde fazla öğrendiğimizi, eğitim hatasını azalttığımızı ancak test hatasını azaltmadığımızı unutmayın—aşırı öğrenmenin bir ders kitabı vakası.

```
train(lambd=0)
```

```
w'nin L2 normu: 13.958930969238281
```

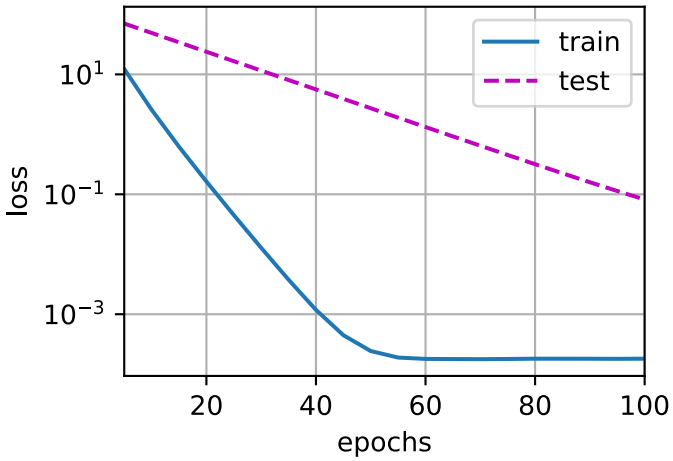


Ağırlık Sönümünü Kullanma

Aşağıda, önemli ölçüde ağırlık sönümü ile çalışıyoruz. Eğitim hatasının arttığını ancak test hatasının azaldığını unutmayın. Düzenlileştirme ile beklediğimiz etki tam da budur.

```
train(lambd=3)
```

```
w'nin L2 normu: 0.3902004361152649
```



4.5.4 Kısa Uygulama

Ağırlık sönümü sinir ağı optimizasyonunda her yerde mevcut olduğu için, derin öğrenme çerçevesi, herhangi bir kayıp fonksiyonuyla birlikte kolay kullanım için ağırlık sönümü optimizasyon algoritmasını kendisine kaynaştırarak bunu özellikle kullanışlı hale getirir. Dahası, bu kaynaştırma, herhangi bir ek hesaplama yükü olmaksızın, uygulama marifetlerinin algoritmaya ağırlık sönümü eklemesine izin vererek hesaplama avantajı sağlar. Güncellemenin ağırlık sönümü kısmı yalnızca her bir parametrenin mevcut değerine bağlı olduğundan, optimize edicinin herhalükarda her parametreye bir kez dokunması gerekir.

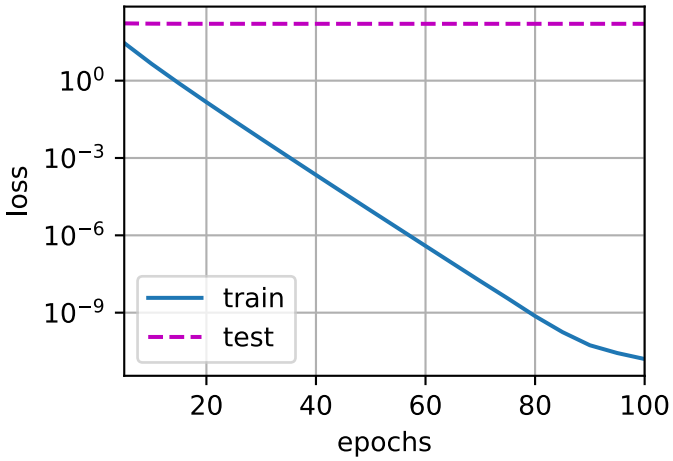
Aşağıdaki kodda, optimize edicimizi başlatırken ağırlık sönümü hiper parametresini doğrudan `weight_decay` aracılığıyla belirtiyoruz. PyTorch varsayılan olarak hem ağırlıkları hem de ek girdileri aynı anda azaltır. Burada ağırlık için yalnızca `weight_decay`'i ayarlıyoruz, böylece ek girdi parametresi b sönmeyecektir.

```
def train_concise(wd):
    net = nn.Sequential(nn.Linear(num_inputs, 1))
    for param in net.parameters():
        param.data.normal_()
    loss = nn.MSELoss(reduction='none')
    num_epochs, lr = 100, 0.003
    # Ek girdi parametresi sönümlenmedi
    trainer = torch.optim.SGD([
        {"params":net[0].weight,'weight_decay': wd},
        {"params":net[0].bias}], lr=lr)
    animator = d2l.Animator(xlabel='epochs', ylabel='loss', yscale='log',
                            xlim=[5, num_epochs], legend=['train', 'test'])
    for epoch in range(num_epochs):
        for X, y in train_iter:
            trainer.zero_grad()
            l = loss(net(X), y)
            l.mean().backward()
            trainer.step()
        if (epoch + 1) % 5 == 0:
            animator.add(epoch + 1, (d2l.evaluate_loss(net, train_iter, loss),
                                     d2l.evaluate_loss(net, test_iter, loss)))
    print(' w\'nin L2 normu:', net[0].weight.norm().item())
```

Grafikler, ağırlık sönümünü sıfırdan uyguladığımızdakilerle aynı görünüyor. Bununla birlikte, önemli ölçüde daha hızlı çalışırlar ve uygulanması daha kolaydır, bu fayda daha büyük problemler için daha belirgin hale gelecektir.

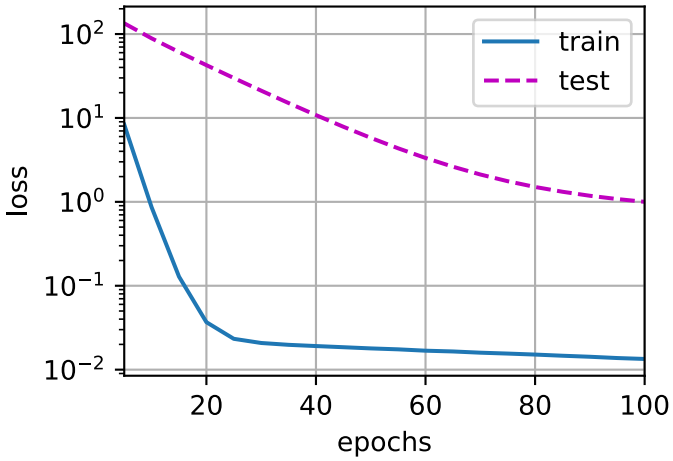
```
train_concise(0)
```

```
w'nin L2 normu: 13.966211318969727
```




```
train_concise(3)
```

```
w'nin L2 normu: 0.44215071201324463
```



Şimdiye kadar, basit bir doğrusal işlevi neyin oluşturduğuna dair yalnızca bir fikre değindik. Dahası, basit doğrusal olmayan bir işlevi neyin oluşturduğu daha da karmaşık bir soru olabilir. Örneğin, çekirdek Hilbert uzayını çoğaltma (Reproducing Kernel Hilbert Spaces - RKHS)[67], doğrusal olmayan bir bağlamda doğrusal fonksiyonlar için tanınmış araçları uygulamaya izin verir. Ne yazık ki, RKHS tabanlı algoritmalar büyük, yüksek boyutlu verilere vasat ölçeklenme eğilimindedir. Bu kitapta, derin bir ağın tüm katmanlarına ağırlık sönümü uygulamanın basit sezgisel yöntemini varsayılan olarak alacağız.

[67] https://en.wikipedia.org/wiki/Reproducing_kernel_Hilbert_space

4.5.5 Özet

- Düzenlileştirme, aşırı öğrenme ile başa çıkmak için yaygın bir yöntemdir. Öğrenilen modelin karmaşıklığını azaltmak için eğitim kümesindeki kayıp işlevine bir ceza terimi ekler.
- Modeli basit tutmak için belirli bir seçenek, L_2 ceza kullanarak ağırlık sönümlemektir. Bu, öğrenme algoritmasının güncelleme adımlarında ağırlık sönümüne yol açar.
- Ağırlık sönümü işlevi, derin öğrenme çerçevelerinden optimize edicilerde sağlanır.
- Farklı parametre kümeleri, aynı eğitim döngüsü içinde farklı güncelleme davranışlarına sahip olabilir.

4.5.6 Alıştırmalar

1. Bu bölümdeki tahmin probleminde λ değeri ile deney yapınız. Eğitim ve test doğruluğunu λ işlevinin bir işlevi olarak çizin. Ne gözlemliyorsunuz?
2. En uygun λ değerini bulmak için bir geçerleme kümesi kullanın. Gerçekten optimal değer bu mudur? Bu önemli mi?
3. $\|\mathbf{w}\|^2$ yerine ceza seçimi olarak $\sum_i |w_i|$ kullansaydık (L_1 düzenlileştirme) güncelleme denklemleri nasıl görünürdü?
4. $\|\mathbf{w}\|^2 = \mathbf{w}^\top \mathbf{w}$ olduğunu biliyoruz. Matrisler için benzer bir denklem bulabilir misiniz (matematikçiler buna Frobenius normu[68] diyorlar)?
5. Eğitim hatası ile genelleme hatası arasındaki ilişkiyi gözden geçirin. Ağırlık sönümü, artan eğitim ve uygun karmaşıklıkta bir modelin kullanılmasına ek olarak, aşırı öğrenme ile başa çıkmak için başka hangi yolları düşünebilirsiniz?
6. Bayesçi istatistikte, $P(w \mid x) \propto P(x \mid w)P(w)$ aracılığıyla bir sonsal olasılığın ve önsel olasılığın çarpımını kullanırız. $P(w)$'yi düzenlileştirme ile nasıl tanımlayabilirsiniz?

Tartışmalar[69]

4.6 Hattan Düş(ür)me

Section 4.5 içinde, ağırlıkların L_2 normunu cezalandırarak istatistiksel modelleri düzenlileştirmek için klasik yaklaşımı tanıttık. Olasılıksal terimlerle, ağırlıkların ortalaması sıfır bir Gauss dağılımından değerler aldığına dair önsel bir inancı varsaydığımızı iddia ederek bu tekniği haklı gösterebiliriz. Daha sezgisel olarak, modeli, az sayıdaki muhtemel sahte ilişkilere çok fazla bağlı yapmak yerine, ağırlıklarını birçok özniteliğe dağıtmaya teşvik ettiğimizi iddia edebiliriz.

[68] https://en.wikipedia.org/wiki/Matrix_norm#Frobenius_norm
[69] https://discuss.d2l.ai/t/99

4.6.1 Aşırı Öğrenmeye Tekrar Bakış

Örneklerden daha fazla öznitelikle karşı karşıya kalan doğrusal modeller, aşırı öğrenme eğilimindedir. Ancak özniteliklerden daha fazla örnek verildiğinde, genellikle doğrusal modellere aşırı öğrenmeme için güvenebiliriz. Ne yazık ki, doğrusal modellerin genelleştirildiği güvenilirliğin bir bedeli vardır. Naif olarak uygulanan doğrusal modeller, öznitelikler arasındaki etkileşimleri hesaba katmaz. Doğrusal bir model, her öznitelik için bağlamı yok sayarak pozitif veya negatif bir ağırlık atamalıdır.

Geleneksel metinlerde, genelleştirilebilirlik ve esneklik arasındaki bu temel gerilim, *yanlılık-değişinti ödünleşmesi (bias-variance tradeoff)* olarak tanımlanır. Doğrusal modeller yüksek yanlılığa sahiptir: Yalnızca küçük bir işlev sınıfını temsil edebilirler. Ancak bu modeller düşük varyansa sahiptirler: Verilerin farklı rastgele örneklemlerinde benzer sonuçlar verirler.

Derin sinir ağları, yanlılık-değişinti spektrumunun diğer ucunda bulunur. Doğrusal modellerin aksine, sinir ağları her özelliğe ayrı ayrı bakmakla sınırlı kalmazlar. Öznitelik grupları arasındaki etkileşimleri öğrenebilirler. Örneğin, bir e-postada birlikte görünen "Nijerya" ve "Western Union"nin yaramaz postayı (spam) gösterdiğini ancak ayrı ayrıyken göstermediklerini çıkarabilirler.

Özniteliklerden çok daha fazla örneğimiz olsa bile, derin sinir ağları gereğinden aşırı öğrenebilirler. 2017'de bir grup araştırmacı, rastgele etiketlenmiş imgeler üzerinde derin ağları eğiterek sinir ağlarının aşırı esnekliğini gösterdi. Girdileri çıktılara bağlayan herhangi bir gerçek model olmamasına rağmen, rasgele gradyan inişi tarafından optimize edilen sinir ağının eğitim kümesindeki her imgeyi mükemmel şekilde etiketleyebileceğini buldular. Bunun ne anlama geldiğini bir düşünün. Etiketler rastgele atanırsa ve 10 sınıf varsa, hiçbir sınıflandırıcı harici tutulan verilerinde %10'den daha iyi doğruluk sağlayamaz. Buradaki genelleme açığı %90 gibi büyük bir oran. Modellerimiz bu seviye aşırı öğrenebilecek kadar ifade edici ise, o halde ne zaman aşırı öğrenmemelerini beklemeliyiz?

Derin ağların kafa karıştırıcı genelleme özelliklerinin matematiksel temelleri, açık araştırma soruları olmaya devam ediyor ve teori yönelimli okuyucuyu bu konuda daha derine inmeye teşvik ediyoruz. Şimdilik, derin ağların genelleştirilmesini deneysel olarak iyileştirme eğiliminde olan pratik araçların araştırmasına dönüyoruz.

4.6.2 Düzensizliğe Gürbüzlük

İyi bir tahminci modelden ne beklediğimizi kısaca düşünelim. Görünmeyen veriler üzerinde iyi başarım göstermelerini istiyoruz. Klasik genelleme teorisi, eğitim ve test performansı arasındaki aralığı kapatmak için basit bir modeli hedeflememiz gerektiğini öne sürer. Sadelik, az sayıda boyut şeklinde olabilir. Section 4.4 içinde doğrusal modellerin tek terimli temel fonksiyonlarını tartışırken bunu araştırdık. Ek olarak, ağırlık sönümünü (L_2 düzenlileştirmesi) Section 4.5 içinde tartışırken gördüğümüz gibi, parametrelerin (ters) normu, basitliğin de bir kullanışlı ölçüsünü temsil eder. Bir başka kullanışlı basitlik kavramı, pürüzsüzlüktür, yani işlevin girdilerindeki küçük değişikliklere hassas olmaması gerektiğidir. Örneğin, imgeleri sınıflandırdığımızda, piksellere bazı rastgele gürültü eklemenin çoğunlukla zararsız olmasını bekleriz.

1995'te Christopher Bishop, girdi gürültüsü ile eğitimin Tikhonov düzenlemesine eşdeğer olduğunu kanıtladığında bu fikri formüle döktü (Bishop, 1995). Bu çalışma, bir fonksiyonun düzgün (ve dolayısıyla basit) olması gerekliliği ile girdideki karışıklıklara dirençli olması gerekliliği arasında açık bir matematiksel bağlantı kurdu.

Ardından 2014 yılında Srivastava ve ark. (Srivastava *et al.*, 2014), Bishop'un fikrinin bir ağın iç katmanlarına da nasıl uygulanacağı konusunda akıllıca bir fikir geliştirdi. Yani, eğitim sırasında sonraki katmanı hesaplamadan önce ağın her katmanına gürültü yerleştirmeyi önerdiler. Birçok katman içeren derin bir ağı eğitirken, gürültü yerleştirmenin sadece girdi-çıktı eşlemesinde pürüzsüzlüğü zorladığını fark ettiler.

Hattan düşürme olarak adlandırılan fikirleri, ileri yayılma sırasında her bir iç katmanı hesaplarken gürültü yerleştirmeyi içerir ve sinir ağlarını eğitmek için standart bir teknik haline gelmiştir. Bu yönteme *hattan düşürme* deniyor çünkü eğitim sırasında bazı nöronları tam anlamıyla *hattan düşürüyoruz*. Standart hattan düşürme eğitim boyunca, her yinelemede, sonraki katmanı hesaplamadan önce, her katmandaki düğümlerin bir kısmının sıfırlanmasından oluşur.

Açık olmak gerekirse, Bishop'la bağlantı kurarak kendi hikayemizi dayatıyoruz. Hattan düşürme ilgili orijinal makale, cinsel üreme ile şaşırtıcı bir benzetme yaparak bir önsezi sunuyor. Yazarlar, sinir ağının aşırı öğrenmesinin, her bir katmanın önceki katmandaki belirli bir etkinleştirme modeline dayandığı ve bu koşulu *birlikte-uyarlama* olarak adlandırdığı bir durumla karakterize edildiğini savunuyorlar. Onlar, tıpkı cinsel üremenin birlikte uyarlanmış genleri parçaladığının iddia edildiği gibi, hattan düşürmenin birlikte uyarlamayı bozduğunu iddia ediyorlar.

O halde asıl zorluk bu gürültünün nasıl yerleştirileceğidir. Bir fikir, gürültüyü *tarafsız* bir şekilde yerleştirmektir, böylece her katmanın beklenen değeri—diğerlerini sabitlerken—gürültü eksikken alacağı değere eşittir.

Bishop çalışmasında, doğrusal bir modelin girdilerine Gauss gürültüsünü ekledi. Her eğitim yinelemesinde, $\mathbf{x}$ girdisine ortalaması sıfır $\epsilon \sim \mathcal{N}(0, \sigma^2)$ bir dağılımdan örneklenen gürültü ekleyerek dürtülmüş bir $\mathbf{x}' = \mathbf{x} + \epsilon$ noktası elde etti. Beklenti değeri, $E[\mathbf{x}'] = \mathbf{x}$'dir.

Standart hattan düşürme düzenlileştirmesinde, bir kısmı tutulan (hattan düşürülmeyen) düğümlere göre normalleştirerek her katmanı yansızlaştırır. Diğer bir deyişle, *hattan düşürme olasılığı* p ile, her bir ara h etkinleştirmesi aşağıdaki gibi rastgele bir h' değişkeni ile değiştirilir:

$$h' = \begin{cases} 0 & \text{with probability } p \\ \frac{h}{1-p} & \text{otherwise} \end{cases} \tag{4.6.1}$$

Tasarım gereği, beklenti değişmeden kalır, yani $E[h'] = h$.

4.6.3 Pratikte Hattan Düşürme

Fig. 4.1.1 içindeki bir gizli katmanlı ve 5 gizli birimli MLP'yi hatırlayın. Hattan düşürmeyi gizli bir katmana uyguladığımızda, her gizli birimi p olasılığı ile sıfırlarsak, sonuç, orijinal nöronların yalnızca bir alt kümesini içeren bir ağ olarak görülebilir. Fig. 4.6.1 içinde, h_2 ve h_5 kaldırılmıştır. Sonuç olarak, çıktıların hesaplaması artık h_2 veya h_5 değerlerine bağlı değildir ve bunların ilgili gradyanları da geri yayma gerçekleştirilirken kaybolur. Bu şekilde, çıktı katmanının hesaplanması, $h_1, \ldots, h_5$ öğelerinin herhangi birine aşırı derecede bağımlı olamaz.

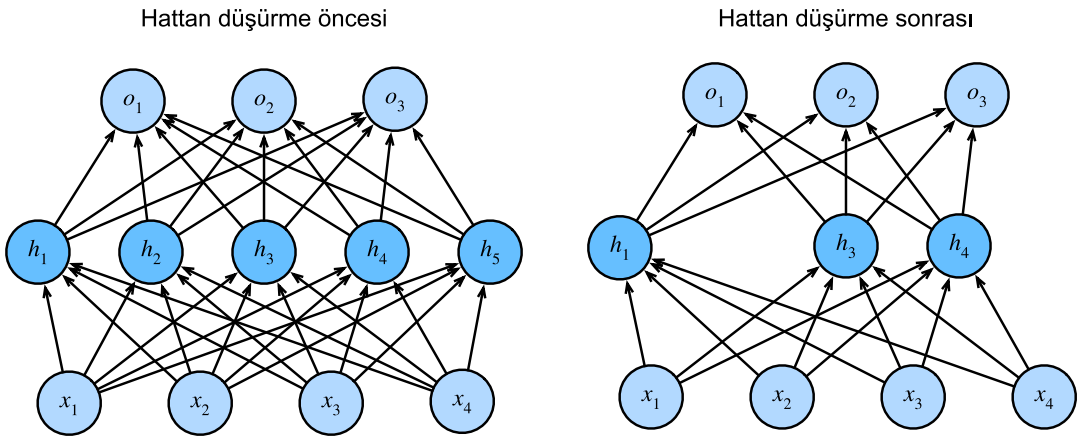


Fig. 4.6.1: Hattan düşürme öncesi ve sonrası MLP.

Tipik olarak, test sırasında hattan düşürmeyi devre dışı bırakırız. Eğitimli bir model ve yeni bir örnek verildiğinde, herhangi bir düğümü çıkarmıyoruz ve bu nedenle normalleştirmemize gerek yok. Bununla birlikte, bazı istisnalar vardır: Bazı araştırmacılar, sinir ağı tahminlerinin *belirsizliğini* tahmin etmek için sezgisel olarak test zamanında hattan düşürmeyi kullanır. Tahminler birçok farklı hattan düşürme maskesinde uyuşuyorsa, ağın daha güvenli olduğunu söyleyebiliriz.

4.6.4 Sıfırdan Uygulama

Hattan düşürme işlevini tek bir katman için uygulamak için, katmanımızın boyutları olduğu için Bernoulli (ikili) rastgele değişkenden olabildiğince çok örneklem almalıyız, burada rastgele değişken $1-p$ olasılıkla 1 (tut), p olasılıkla 0 (düşür) değerini alır. Bunu gerçekleştirmenin kolay bir yolu, ilk olarak $U[0,1]$ tekdüze dağılımından örnekler almaktır. Daha sonra, p'den büyük olan düğümlere karşılık gelen örnekleri tutarak geri kalanı hattan düşürebiliriz.

Aşağıdaki kodda, `X` tensör girdisindeki öğeleri `hattan düşürme` olasılığı ile hattan düşüren ve kalanı yukarıda açıklandığı gibi yeniden ölçeklendiren bir `dropout_layer` işlevi uyguluyoruz; kurtulanları `1.0-dropout` ile böleriz.

```
import torch
from torch import nn
from d2l import torch as d2l


def dropout_layer(X, dropout):
    assert 0 <= dropout <= 1
    # Bu durumda tüm elemanlar düşürülür
    if dropout == 1:
        return torch.zeros_like(X)
    # Bu durumda tüm elemanlar tutulur
    if dropout == 0:
        return X
    mask = (torch.rand(X.shape) > dropout).float()
    return mask * X / (1.0 - dropout)
```

`dropout_layer` fonksiyonunu birkaç örnek üzerinde test edebiliriz. Aşağıdaki kod satırlarında, `X` girdimizi sırasıyla 0, 0.5 ve 1 olasılıklarla hattan düşürme işleminden geçiriyoruz.

```
X= torch.arange(16, dtype = torch.float32).reshape((2, 8))
print(X)
print(dropout_layer(X, 0.))
print(dropout_layer(X, 0.5))
print(dropout_layer(X, 1.))
```

```
tensor([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11., 12., 13., 14., 15.]])
tensor([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11., 12., 13., 14., 15.]])
tensor([[ 0.,  2.,  0.,  0.,  0., 10.,  0., 14.],
        [ 0., 18., 20.,  0., 24., 26.,  0.,  0.]])
tensor([[0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0.]])
```

Model Parametrelerini Tanımlama

Yine, Section 3.5 içinde tanıtılan Fashion-MNIST veri kümesiyle çalışıyoruz. Her biri 256 çıktı içeren iki gizli katmana sahip bir MLP tanımlıyoruz.

```
num_inputs, num_outputs, num_hiddens1, num_hiddens2 = 784, 10, 256, 256
```

Modeli Tanımlama

Aşağıdaki model, her bir gizli katmanın çıktısına (etkinleştirme işlevini takiben) hattan düşürme uygular. Her katman için ayrı ayrı hattan düşürme olasılıkları belirleyebiliriz. Yaygın bir eğilim, girdi katmanına yakınken daha düşük bir hattan düşürme olasılığı ayarlamaktır. Aşağıda, birinci ve ikinci gizli katmanlar için onları sırasıyla 0.2 ve 0.5 olarak ayarladık. Hattan düşürmenin yalnızca eğitim sırasında etkin olmasını sağlıyoruz.

```
dropout1, dropout2 = 0.2, 0.5

class Net(nn.Module):
    def __init__(self, num_inputs, num_outputs, num_hiddens1, num_hiddens2,
                 is_training = True):
        super(Net, self).__init__()
        self.num_inputs = num_inputs
        self.training = is_training
        self.lin1 = nn.Linear(num_inputs, num_hiddens1)
        self.lin2 = nn.Linear(num_hiddens1, num_hiddens2)
        self.lin3 = nn.Linear(num_hiddens2, num_outputs)
        self.relu = nn.ReLU()

    def forward(self, X):
        H1 = self.relu(self.lin1(X.reshape((-1, self.num_inputs))))
        # Hattan düşürmeyi yalnızca modeli eğitirken kullan
        if self.training == True:
            #  İlk tamamen bağlı katmandan sonra bir hattan düşürme katmanı ekle
            H1 = dropout_layer(H1, dropout1)
        H2 = self.relu(self.lin2(H1))
```

(continues on next page)

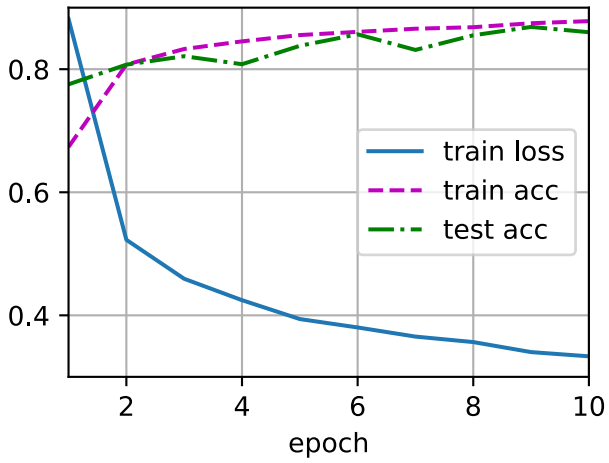
(continued from previous page)

```
        if self.training == True:
            # İkinci tamamen bağlı katmandan sonra bir hattan düşürme katmanı ekle
            H2 = dropout_layer(H2, dropout2)
        out = self.lin3(H2)
        return out


net = Net(num_inputs, num_outputs, num_hiddens1, num_hiddens2)
```

Eğitim and Test Etme

Bu, daha önce açıklanan MLP eğitimine ve testine benzer.

```
num_epochs, lr, batch_size = 10, 0.5, 256
loss = nn.CrossEntropyLoss(reduction='none')
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
trainer = torch.optim.SGD(net.parameters(), lr=lr)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



4.6.5 Kısa Uygulama

Üst düzey API'lerle, tek yapmamız gereken, her tam bağlı katmandan sonra bir `Dropout` katmanı eklemek ve hattan düşürme olasılığını kurucusuna tek argüman olarak aktarmaktır. Eğitim sırasında, `Dropout` katmanı, belirtilen hattan düşürme olasılığına göre önceki katmanın çıktılarını (veya eşdeğer olarak sonraki katmana olan girdileri) rasgele düşürür. Eğitim modunda olmadığında, `Dropout` katmanı verileri test sırasında basitçe iletir.

```
net = nn.Sequential(nn.Flatten(),
        nn.Linear(784, 256),
        nn.ReLU(),
        # İlk tamamen bağlı katmandan sonra bir hattan düşürme katmanı ekle
        nn.Dropout(dropout1),
        nn.Linear(256, 256),
```

(continues on next page)

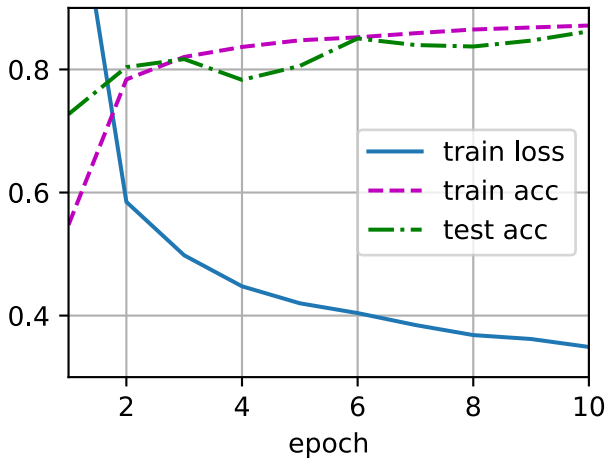
(continued from previous page)

```
        nn.ReLU(),
        # İkinci tamamen bağlı katmandan sonra bir hattan düşürme katmanı ekle
        nn.Dropout(dropout2),
        nn.Linear(256, 10))

def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, std=0.01)

net.apply(init_weights)
```

```
Sequential(
  (0): Flatten(start_dim=1, end_dim=-1)
  (1): Linear(in_features=784, out_features=256, bias=True)
  (2): ReLU()
  (3): Dropout(p=0.2, inplace=False)
  (4): Linear(in_features=256, out_features=256, bias=True)
  (5): ReLU()
  (6): Dropout(p=0.5, inplace=False)
  (7): Linear(in_features=256, out_features=10, bias=True)
)
```

Sonra, modeli eğitir ve test ederiz.

```
trainer = torch.optim.SGD(net.parameters(), lr=lr)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



4.6.6 Özet

- Boyutların sayısını ve ağırlık vektörünün boyutunu kontrol etmenin ötesinde, hattan düşürme, aşırı öğrenmeyi önlemek için başka bir araçtır. Genellikle birlikte kullanılırlar.
- Hattan düşürme, h etkinleştirmesini, beklentisi h değerine sahip rastgele bir değişken ile değiştirir.
- Hattan düşürme yalnızca eğitim sırasında kullanılır.

4.6.7 Alıştırmalar

1. İlk ve ikinci katmanlar için hattan düşürme olasılıklarını değiştirirseniz ne olur? Özellikle, her iki katman için olanları yer değiştirirseniz ne olur? Bu soruları yanıtlamak için bir deney tasarlayınız, bulgularınızı nicel olarak açıklayınız ve nitel çıkarımları özetleyiniz.
2. Dönem sayısını arttırınız ve hattan düşürme kullanılırken elde edilen sonuçları kullanmadığınız zamanlarda elde ettikleriniz ile karşılaştırınız.
3. Hattan düşürme uygulandığında ve uygulanmadığında her bir gizli katmandaki etkinleştirmelerin varyansı nedir? Her iki model için de bu miktarın zaman içinde nasıl değiştiğini gösteren bir grafik çiziniz.
4. Hattan düşürme neden tipik olarak test zamanında kullanılmaz?
5. Bu bölümdeki modeli örnek olarak kullanarak, hattan düşürme ve ağırlık sönümünün etkilerini karşılaştırınız. Hattan düşürme ve ağırlık sönümü aynı anda kullanıldığında ne olur? Sonuçlar katkı sağlıyor mu? Azalan getiri mi (ya da daha kötüsü mü) var? Birbirlerini iptal ediyorlar mı?
6. Hattan düşürmeyi, etkinleştirmeler yerine ağırlık matrisinin bireysel ağırlıklarına uygularsak ne olur?
7. Her katmanda rastgele gürültü yerleştirmek için standart hattan düşürme tekniğinden farklı başka bir teknik bulunuz. Fashion-MNIST veri kümesinde (sabit bir mimari için) hattan düşürmeden daha iyi performans gösteren bir yöntem geliştirebilir misiniz?

Tartışmalar[70]

4.7 İleri Yayma, Geri Yayma ve Hesaplamalı Çizge

Şimdiye kadar modellerimizi minigrup rasgele gradyan inişi ile eğittik. Bununla birlikte, algoritmayı uyguladığımızda, yalnızca model aracılığıyla *ileri yayma* ile ilgili hesaplamalar hakkında endişelendik. Gradyanları hesaplama zamanı geldiğinde, derin öğrenme çerçevesi tarafından sağlanan geri yayma fonksiyonunu çalıştırdık.

Gradyanların otomatik olarak hesaplanması (otomatik türev alma), derin öğrenme algoritmalarının uygulanmasını büyük ölçüde basitleştirir. Otomatik türev almadan önce, karmaşık modellerde yapılan küçük değişiklikler bile karmaşık türevlerin elle yeniden hesaplanmasını gerektiriyordu. Şaşırtıcı bir şekilde, akademik makaleler güncelleme kurallarını türetmek için çok sayıda sayfa ayırmak zorunda kalırdı. İlginç kısımlara odaklanabilmemiz için otomatik türeve

[70] https://discuss.d2l.ai/t/101

güvenmeye devam etmemiz gerekse de, sığ bir derin öğrenme anlayışının ötesine geçmek istiyorsanız, bu gradyanların kaputun altında nasıl hesaplandığını bilmelisiniz.

Bu bölümde, *geriye doğru yayma*nın (daha yaygın olarak *geri yayma* olarak adlandırılır) ayrıntılarına derinlemesine dalacağız. Hem teknikler hem de uygulamaları hakkında bazı bilgiler vermek için birtakım temel matematik ve hesaplama çizgelerine güveniyoruz. Başlangıç olarak, açıklamamızı ağırlık sönümlü (L_2 düzenlileştirme), bir gizli katmanlı MLP'ye odaklıyoruz.

4.7.1 İleri Yayma

İleri yayma (veya *ileri iletme*), girdi katmanından çıktı katmanına sırayla bir sinir ağı için ara değişkenlerin (çıktılar dahil) hesaplanması ve depolanması anlamına gelir. Artık tek bir gizli katmana sahip bir sinir ağı mekaniği üzerinde adım adım çalışacağız. Bu sıkıcı görünebilir ama funk virtüözü James Brown'un ebedi sözleriyle, "patron olmanın bedelini ödemelisiniz".

Kolaylık olması açısından, girdi örneğinin $\mathbf{x} \in \mathbb{R}^d$ olduğunu ve gizli katmanımızın bir ek girdi terimi içermediğini varsayalım. İşte ara değişkenimiz:

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}, \tag{4.7.1}$$

$\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ gizli katmanın ağırlık parametresidir. $\mathbf{z} \in \mathbb{R}^h$ ara değişkenini ϕ etkinleştirme fonksiyonu üzerinden çalıştırdıktan sonra, h uzunluğundaki gizli etkinleştirme vektörümüzü elde ederiz,

$$\mathbf{h} = \phi(\mathbf{z}). \tag{4.7.2}$$

$\mathbf{h}$ gizli değişkeni de bir ara değişkendir. Çıktı katmanının parametrelerinin yalnızca $\mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$ ağırlığına sahip olduğunu varsayarsak, q uzunluğunda vektörel bir çıktı katmanı değişkeni elde edebiliriz:

$$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}. \tag{4.7.3}$$

Kayıp fonksiyonunun l ve örnek etiketin y olduğunu varsayarsak, tek bir veri örneği için kayıp terimini hesaplayabiliriz,

$$L = l(\mathbf{o}, y). \tag{4.7.4}$$

L_2 düzenlileştirmenin tanımına göre, λ hiper parametresi verildiğinde, düzenlileştirme terimi şöyledir:

$$s = \frac{\lambda}{2} \left(\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2 \right), \tag{4.7.5}$$

burada matrisin Frobenius normu, matris bir vektöre düzleştirildikten sonra uygulanan L_2 normudur. Son olarak, modelin belirli bir veri örneğine göre düzenlileştirilmiş kaybı şudur:

$$J = L + s. \tag{4.7.6}$$

Aşağıdaki tartışmada J'ye *amaç fonksiyonu* olarak atıfta bulunacağız.

4.7.2 İleri Yaymanın Hesaplamalı Çizgesi

Hesaplamalı çizgeleri çizmek, hesaplamadaki operatörlerin ve değişkenlerin bağımlılıklarını görselleştirmemize yardımcı olur. Fig. 4.7.1, yukarıda açıklanan basit ağ ile ilişkili çizgeyi içerir, öyleki kareler değişkenleri ve daireler işlemleri temsil eder. Sol alt köşe girdiyi, sağ üst köşesi çıktıyı belirtir. Okların yönlerinin (veri akışını gösteren) esasen sağa ve yukarıya doğru olduğuna dikkat edin.

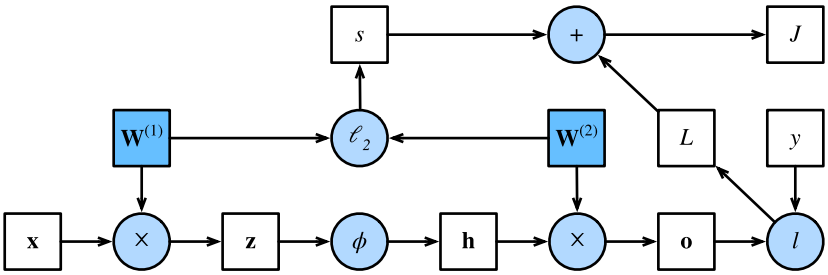


Fig. 4.7.1: İleri yaymanın hesaplamalı çizgesi

4.7.3 Geri Yayma

Geri yayma, sinir ağı parametrelerinin gradyanını hesaplama yöntemini ifade eder. Kısacası yöntem, analizden *zincir kuralına* göre ağı çıktıdan girdi katmanına ters sırada dolaşır. Algoritma, gradyanı bazı parametrelere göre hesaplarken gerekli olan tüm ara değişkenleri (kısmi türevler) depolar. $\mathsf{Y} = f(\mathsf{X})$ ve $\mathsf{Z} = g(\mathsf{Y})$, fonksiyonlarımız olduğunu $\mathsf{X}, \mathsf{Y}, \mathsf{Z}$ girdi ve çıktılarının rastgele şekilli tensörler olduğunu varsayalım. Zincir kuralını kullanarak, Z'nin X'e göre türevini hesaplayabiliriz.

$$\frac{\partial \mathsf{Z}}{\partial \mathsf{X}} = \text{prod}\left(\frac{\partial \mathsf{Z}}{\partial \mathsf{Y}}, \frac{\partial \mathsf{Y}}{\partial \mathsf{X}}\right). \tag{4.7.7}$$

Burada, aktarma ve girdi konumlarını değiştirme gibi gerekli işlemler gerçekleştirildikten sonra argümanlarını çarpmak için prod operatörünü kullanıyoruz. Vektörler için bu basittir: Bu basitçe matris-matris çarpımıdır. Daha yüksek boyutlu tensörler için uygun muadili kullanırız. prod operatörü tüm gösterim ek yükünü gizler.

Hesaplamalı çizgesi Fig. 4.7.1 içinde gösterilen bir gizli katmana sahip basit ağın parametrelerinin $\mathbf{W}^{(1)}$ ve $\mathbf{W}^{(2)}$ olduğunu hatırlayalım. Geri yaymanın amacı, $\partial J/\partial \mathbf{W}^{(1)}$ ve $\partial J/\partial \mathbf{W}^{(2)}$ gradyanlarını hesaplamaktır. Bunu başarmak için, zincir kuralını uygularız ve sırayla her bir ara değişken ve parametrenin gradyanını hesaplarız. Hesaplamalı çizgenin sonucuyla başlamamız ve parametrelere doğru yolumuza devam etmemiz gerektiğinden, hesaplamaların sırası ileri yaymada gerçekleştirilenlere göre tersine çevrilir. İlk adım $J = L + s$ amaç fonksiyonunun gradyanlarını L kayıp terimi ve s düzenlileştirme terimine göre hesaplamaktır.

$$\frac{\partial J}{\partial L} = 1 \text{ ve } \frac{\partial J}{\partial s} = 1. \tag{4.7.8}$$

Ardından, zincir kuralı ile $\mathbf{o}$ çıktı katmanının değişkenine göre amaç fonksiyonunun gradyanını hesaplıyoruz:

$$\frac{\partial J}{\partial \mathbf{o}} = \text{prod}\left(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}}\right) = \frac{\partial L}{\partial \mathbf{o}} \in \mathbb{R}^q. \tag{4.7.9}$$

Ardından, her iki parametreye göre düzenlileştirme teriminin gradyanlarını hesaplıyoruz:

$$\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)} \text{ and } \frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}. \tag{4.7.10}$$

Şimdi, çıktı katmanına en yakın model parametrelerinin gradyanını, $\partial J/\partial \mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$, hesaplayabiliriz. Zincir kuralını kullanalım:

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}}\right) = \frac{\partial J}{\partial \mathbf{o}}\mathbf{h}^\top + \lambda \mathbf{W}^{(2)}. \tag{4.7.11}$$

$\mathbf{W}^{(1)}$'e göre gradyanı elde etmek için, çıktı katmanı boyunca gizli katmana geri yaymaya devam etmemiz gerekir. Gizli katmanın $\partial J/\partial \mathbf{h} \in \mathbb{R}^h$ çıktılarına göre gradyan şu şekilde verilir:

$$\frac{\partial J}{\partial \mathbf{h}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}}\right) = {\mathbf{W}^{(2)}}^\top \frac{\partial J}{\partial \mathbf{o}}. \tag{4.7.12}$$

ϕ etkinleştirme fonksiyonu eleman yönlü uygulandığından, $\mathbf{z}$ ara değişkeninin $\partial J/\partial \mathbf{z} \in \mathbb{R}^h$ gradyanını hesaplamak $\odot$ ile gösterdiğimiz eleman yönlü çarpma operatörünü kullanmayı gerektirir:

$$\frac{\partial J}{\partial \mathbf{z}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}}\right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'\left(\mathbf{z}\right). \tag{4.7.13}$$

Son olarak, girdi katmanına en yakın model parametrelerinin, $\partial J/\partial \mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$, gradyanını elde edebiliriz. Zincir kuralına göre hesaplarsak,

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}}\right) = \frac{\partial J}{\partial \mathbf{z}}\mathbf{x}^\top + \lambda \mathbf{W}^{(1)}. \tag{4.7.14}$$

4.7.4 Sinir Ağları Eğitimi

Sinir ağlarını eğitirken, ileri ve geri yayma birbirine bağlıdır. Özellikle, ileri yayma için, hesaplamalı çizgeyi bağımlılıklar yönünde gezeriz ve yoldaki tüm değişkenleri hesaplarız. Bunlar daha sonra grafikteki hesaplama sırasının tersine çevrildiği geri yayma için kullanılır.

Kafanızda canlandırmak için yukarıda belirtilen basit ağı örnek olarak alın. Bir yandan, ileri yayma sırasında (4.7.5) düzenlileştirme teriminin hesaplanması $\mathbf{W}^{(1)}$ ve $\mathbf{W}^{(2)}$ model parametrelerinin mevcut değerlerine bağlıdır. En son yinelemede geri yaymaya göre optimizasyon algoritması tarafından verilirler. Öte yandan, geri yayma sırasında (4.7.11) parametresi için gradyan hesaplaması, ileri yayma tarafından verilen gizli $\mathbf{h}$ değişkeninin mevcut değerine bağlıdır.

Bu nedenle, sinir ağlarını eğitirken, model parametreleri ilklendikten sonra, ileri yaymayı geri yayma ile değiştiririz, model parametrelerini geri yayma tarafından verilen gradyanları kullanarak güncelleriz. Geri yaymanın, tekrarlanan hesaplamaları önlemek için ileriye yaymadan depolanan ara değerleri yeniden kullandığını unutmayın. Sonuçlardan biri, geri yayma tamamlanana kadar ara değerleri korumamız gerektiğidir. Bu aynı zamanda, eğitimin basit tahminden önemli ölçüde daha fazla bellek gerektirmesinin nedenlerinden biridir. Ayrıca, bu tür ara değerlerin boyutu, ağ katmanlarının sayısı ve iş boyutu ile kabaca orantılıdır. Bu nedenle, daha büyük toplu iş boyutlarını kullanarak daha derin ağlar eğitmek, kolaylıkla *yetersiz bellek* hatalarına yol açar.

4.7.5 Özet

- İleri yayma, ara değişkenleri sırayla hesaplar ve sinir ağı tarafından tanımlanan hesaplamalı çizge içinde depolar. Girdiden çıktı katmanına doğru ilerler.
- Geri yayma, sinir ağı içindeki ara değişkenlerin ve parametrelerin gradyanlarını ters sırayla sırayla hesaplar ve saklar.
- Derin öğrenme modellerini eğitirken, ileri yayma ve geri yayma birbirine bağlıdır.
- Eğitim, tahminlemeden önemli ölçüde daha fazla bellek gerektirir.

4.7.6 Alıştırma

1. **X**'in bazı sayıl (skaler) fonksiyonu f'nin girdilerinin $n \times m$ matrisler olduğunu varsayın. **X**'e göre f'nin gradyanının boyutsallığı nedir?
2. Bu bölümde açıklanan modelin gizli katmanına bir ek girdi ekleyiniz (düzenlileştirme teriminde ek girdiyi katmanız gerekmez).
    1. İlgili hesaplamalı çizgeyi çizin.
    2. İleri ve geri yayma denklemlerini türetiniz.
3. Bu bölümde açıklanan modeldeki eğitim ve tahmin için bellek ayak izini hesaplayın.
4. İkinci türevleri hesaplamak istediğinizi varsayın. Hesaplamalı çizgeye ne olur? Hesaplamanın ne kadar sürmesini bekliyorsunuz?
5. Hesaplamalı çizgenin GPU'nuz için çok büyük olduğunu varsayın.
    1. Birden fazla GPU'ya bölebilir misiniz?
    2. Daha küçük bir minigrup üzerinde eğitime göre avantajları ve dezavantajları nelerdir?

Tartışmalar[71]

4.8 Sayısal Kararlılık ve İlkleme

Buraya kadar, uyguladığımız her model, parametrelerini önceden belirlenmiş bazı dağılımlara göre ilklememizi gerektirdi. Şimdiye kadar, bu seçimlerin nasıl yapıldığının ayrıntılarını gözden ardı ederek ilkleme düzenini doğal kabul ettik. Bu seçimlerin özellikle önemli olmadığı izlenimini bile almış olabilirsiniz. Aksine, ilkleme düzeninin seçimi sinir ağı öğrenmesinde önemli bir rol oynar ve sayısal kararlılığı korumak için çok önemli olabilir. Dahası, bu seçimler doğrusal olmayan etkinleştirme fonksiyonunun seçimi ile ilginç şekillerde bağlanabilir. Hangi işlevi seçtiğimiz ve parametreleri nasıl ilklettiğimiz, optimizasyon algoritmamızın ne kadar hızlı yakınsadığını belirleyebilir. Buradaki kötü seçimler, eğitim sırasında patlayan veya kaybolan gradyanlarla karşılaşmamıza neden olabilir. Bu bölümde, bu konuları daha ayrıntılı olarak inceliyoruz ve derin öğrenmedeki kariyeriniz boyunca yararlı bulacağınız bazı yararlı buluşsal yöntemleri tartışıyoruz.

[71] https://discuss.d2l.ai/t/102

4.8.1 Kaybolan ve Patlayan Gradyanlar

L katmanlı $\mathbf{x}$ girdili ve $\mathbf{o}$ çıktılı derin bir ağ düşünün. Bir f_l dönüşümü ile tanımlanan $\mathbf{W}^{(l)}$ ağırlıklarıyla parametreleştirilen her bir katman l ile, ki gizli değişkenleri $\mathbf{h}^{(l)}$'dir ($\mathbf{h}^{(0)} = \mathbf{x}$), ağımız şu şekilde ifade edilebilir:

$$\mathbf{h}^{(l)} = f_l(\mathbf{h}^{(l-1)}) \text{ ve böylece } \mathbf{o} = f_L \circ \ldots \circ f_1(\mathbf{x}). \tag{4.8.1}$$

Tüm gizli değişkenler ve girdiler vektör ise, $\mathbf{o}$ gradyanını herhangi bir $\mathbf{W}^{(l)}$ parametre kümesine göre aşağıdaki gibi yazabiliriz:

$$\partial_{\mathbf{W}^{(l)}} \mathbf{o} = \underbrace{\partial_{\mathbf{h}^{(L-1)}} \mathbf{h}^{(L)}}_{\mathbf{M}^{(L)} \stackrel{\text{def}}{=}} \cdot \ldots \cdot \underbrace{\partial_{\mathbf{h}^{(l)}} \mathbf{h}^{(l+1)}}_{\mathbf{M}^{(l+1)} \stackrel{\text{def}}{=}} \underbrace{\partial_{\mathbf{W}^{(l)}} \mathbf{h}^{(l)}}_{\mathbf{v}^{(l)} \stackrel{\text{def}}{=}}. \tag{4.8.2}$$

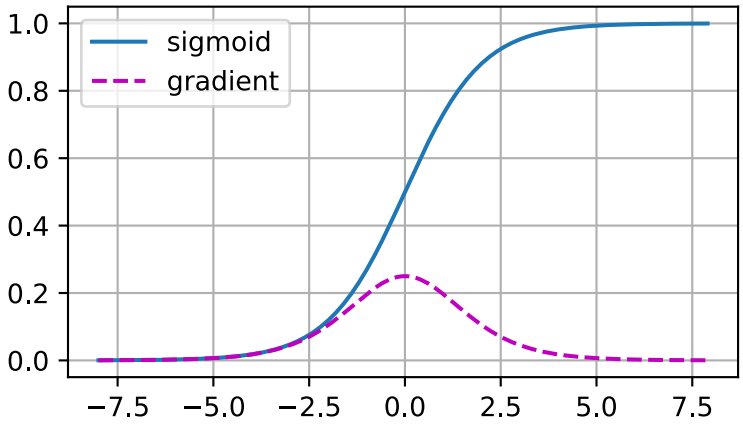
Başka bir deyişle, bu gradyan $L - l$ matrisleri $\mathbf{M}^{(L)} \cdot \ldots \cdot \mathbf{M}^{(l+1)}$ ve $\mathbf{v}^{(l)}$ gradyan vektörünün çarpımıdır. Bu nedenle, çok fazla olasılığı bir araya getirirken sıklıkla ortaya çıkan aynı sayısal küçümenlik sorunlarına duyarlıyız. Olasılıklarla uğraşırken, yaygın bir hile, logaritma-uzayına geçmektir, yani basıncı mantisten sayısal temsilin üssüne kaydırmaktır. Ne yazık ki, yukarıdaki sorunumuz daha ciddidir: Başlangıçta $\mathbf{M}^{(l)}$ matrisleri çok çeşitli özdeğerlere sahip olabilir. Küçük veya büyük olabilirler ve çarpımları *çok büyük* veya *çok küçük* olabilir.

Kararsız gradyanların yarattığı riskler sayısal temsilin ötesine geçer. Tahmin edilemeyen büyüklükteki gradyanlar, optimizasyon algoritmalarımızın kararlılığını da tehdit eder: Ya (i) aşırı büyük olan, modelimizi yok eden (*patlayan gradyan* problemi); veya (ii) aşırı derecede küçük (*kaybolan gradyan* problemi), parametreler neredeyse hiç hareket etmediği için öğrenmeyi imkansız kılan güncellemeler.

Kaybolan Gradyanlar

Kaybolan gradyan sorununa neden olan sık karşılaşılan bir zor durum, her katmanın doğrusal işlemlerinin ardından eklenen σ etkinleştirme işlevinin seçimidir. Tarihsel olarak, sigmoid işlevi $1/(1 + \exp(-x))$ (bkz. Section 4.1), bir eşikleme işlevine benzediği için popülerdi. Öncül yapay sinir ağları biyolojik sinir ağlarından ilham aldığından, ya *tamamen* ateşleyen ya da *hiç* ateşlemeyen (biyolojik nöronlar gibi) nöronlar fikri çekici görünüyordu. Neden yok olan gradyanlara neden olabileceğini görmek için sigmoide daha yakından bakalım.

```
%matplotlib inline
import torch
from d2l import torch as d2l

x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.sigmoid(x)
y.backward(torch.ones_like(x))

d2l.plot(x.detach().numpy(), [y.detach().numpy(), x.grad.numpy()],
         legend=['sigmoid', 'gradient'], figsize=(4.5, 2.5))
```



Gördüğünüz gibi sigmoidin gradyanı, girdileri hem büyük hem de küçük olduklarında kaybolur. Dahası, birçok katmanda geri yayma yaparken, birçok sigmoidin girdilerinin sıfıra yakın olduğu altın bölgede (Goldilocks) olmadıkça, tüm çarpımın gradyanları kaybolabilir. Ağımız birçok katmana sahip olduğunda, dikkatli olmadıkça, gradyan muhtemelen herhangi bir katmanda kesilecektir. Gerçekten de, bu problem derin ağ eğitiminda başa bela oluyordu. Sonuç olarak, daha kararlı (ancak sinirsel olarak daha az makul olan) ReLU'lar, uygulayıcıların varsayılan seçimi olarak öne çıktı.

Patlayan Gradyanlar

Tersi problem de, gradyanlar patladığında, benzer şekilde can sıkıcı olabilir. Bunu biraz daha iyi açıklamak için, 100 tane Gauss'luk rasgele matrisler çekip bunları bir başlangıç matrisiyle çarpıyoruz. Seçtiğimiz ölçek için ($\sigma^2 = 1$ varyans seçimi ile), matris çarpımı patlar. Bu, derin bir ağın ilklenmesi nedeniyle gerçekleştiğinde, yakınsama için bir gradyan iniş optimize ediciyi alma şansımız yoktur.

```
M = torch.normal(0, 1, size=(4,4))
print('Tek bir matris \n',M)
for i in range(100):
    M = torch.mm(M,torch.normal(0, 1, size=(4, 4)))

print('100 matrisi carptiktan sonra\n',M)
```

```
Tek bir matris
 tensor([[ 0.3836, -0.5918,  1.4536,  0.4872],
        [-2.4418,  0.1294, -1.6191, -1.0533],
        [-1.9755, -0.4855,  0.6013, -0.7963],
        [ 0.0786,  2.7733,  1.0219, -0.6463]])
100 matrisi carptiktan sonra
 tensor([[-1.6250e+23,  5.7504e+23,  1.3138e+23,  9.8486e+23],
        [ 2.9379e+23, -1.0396e+24, -2.3753e+23, -1.7805e+24],
        [ 7.3199e+22, -2.5902e+23, -5.9181e+22, -4.4363e+23],
        [-1.1024e+23,  3.9009e+23,  8.9127e+22,  6.6810e+23]])
```

Bakışımı (Simetriyi) Kırma

Sinir ağı tasarımındaki bir başka sorun, parametrelendirilmesinde bulunan simetridir. Bir gizli katman ve iki birim içeren basit bir MLP'miz olduğunu varsayalım. Bu durumda, ilk katmanın $\mathbf{W}^{(1)}$ ağırlıklarını ve aynı şekilde aynı işlevi elde etmek için çıktı katmanının ağırlıklarını devrişebiliriz (permütasyon). İlk gizli birimi veya ikinci gizli birimi türev alma arasında bir fark yoktur. Başka bir deyişle, her katmanın gizli birimleri arasında devrişim bakışımımız var.

Bu teorik bir can sıkıntısından daha fazlasıdır. Yukarıda bahsedilen, iki gizli birimi olan tekkatmanlı MLP'yi düşünün. Örnek olarak, çıktı katmanının iki gizli birimi yalnızca bir çıktı birimine dönüştürdüğünü varsayalım. Bir sabit c için gizli katmanın tüm parametrelerini $\mathbf{W}^{(1)} = c$ olarak ilkletirsek ne olacağını hayal edin. Bu durumda, ileriye doğru yayma sırasında gizli birim aynı girdileri ve parametreleri alarak aynı aktivasyonu üretir, ki o da çıktı birimine beslenir. Geri yayma sırasında, çıktı biriminin $\mathbf{W}^{(1)}$ parametrelerine göre türevini almak, öğelerinin tümü aynı değeri alan bir gradyan verir. Bu nedenle, gradyan tabanlı yinelemeden sonra (örneğin, minigrup rasgele gradyan inişi), $\mathbf{W}^{(1)}$ öğesinin tüm öğeleri hala aynı değeri alır.

Bu yinelemeler, *bakışımı asla kendi başına bozmaz* ve ağın ifade gücünü asla gerçekleyemeyebiliriz. Gizli katman, yalnızca tek bir birimi varmış gibi davranacaktır. Minigrup rasgele gradyan inişi bu bakışımı bozmasa da, hattan düşürme düzenlileştirmesinin bozacağını unutmayın!

4.8.2 Parametre İlkleme

Yukarıda belirtilen sorunları ele almanın—ya da en azından hafifletmenin—bir yolu dikkatli ilkletmektir. Optimizasyon sırasında ek özen ve uygun düzenlileştirme, kararlılığı daha da artırabilir.

Varsayılan İlkletme

Önceki bölümlerde, örneğin Section 3.3 içindeki gibi, ağırlıklarımızın değerlerini ilkletmek için normal dağılım kullandık. İlkleme yöntemini belirtmezsek, çerçeve, pratikte genellikle orta düzey problem boyutları için iyi çalışan varsayılan bir rastgele ilkleme yöntemi kullanacaktır.

Xavier İlkletmesi

Bir çıktının (örneğin, bir gizli değişken) o_i tam bağlı bir katman *doğrusal olmayanlar* için ölçek dağılımına bakalım. Bu katman için n_{in} tane x_j girdisi ve bunlarla ilişkili ağırlıkları w_{ij} ile, çıktı şu şekilde verilir:

$$o_i = \sum_{j=1}^{n_{\text{in}}} w_{ij} x_j. \tag{4.8.3}$$

w_{ij} ağırlıklarının tümü aynı dağılımdan bağımsız olarak çekilir. Dahası, bu dağılımın sıfır ortalamaya ve σ^2 varyansına sahip olduğunu varsayalım. Bu, dağılımın Gaussian olması gerektiği anlamına gelmez, sadece ortalama ve varyansın var olması gerektiği anlamına gelir. Şimdilik, x_j katmanındaki girdilerin de sıfır ortalamaya ve γ^2 varyansına sahip olduğunu ve w_{ij}'den ve birbirinden bağımsız olduklarını varsayalım. Bu durumda, o_i'nin ortalamasını ve varyansını şu şek-

ilde hesaplayabiliriz:

$$\begin{aligned}
E[o_i] &= \sum_{j=1}^{n_{\text{in}}} E[w_{ij} x_j] \\
&= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}] E[x_j] \\
&= 0, \\
\text{Var}[o_i] &= E[o_i^2] - (E[o_i])^2 \\
&= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}^2 x_j^2] - 0 \\
&= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}^2] E[x_j^2] \\
&= n_{\text{in}} \sigma^2 \gamma^2.
\end{aligned} \tag{4.8.4}$$

Varyansı sabit tutmanın bir yolu, $n_{\text{in}}\sigma^2 = 1$ diye ayarlamaktır. Şimdi geri yaymayı düşünün. Orada, çıktıya yakın katmanlardan yayılmakta olan gradyanlarla da olsa benzer bir sorunla karşı karşıyayız. İleri yaymayla aynı mantığı kullanarak, gradyanların varyansının $n_{\text{out}}\sigma^2 = 1$, ki n_{out} burada bu katmanın çıktı sayısıdır, olmadıkça patlayabileceğini görüyoruz. Bu bizi bir ikilemde bırakıyor: Her iki koşulu aynı anda karşılayamayız. Bunun yerine, sadece şunlara uymaya çalışırız:

$$\frac{1}{2}(n_{\text{in}} + n_{\text{out}})\sigma^2 = 1 \text{ veya eşdeğeri } \sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}. \tag{4.8.5}$$

Bu, yaratıcılarının ilk yazarının adını taşıyan, artık standart ve pratik olarak faydalı *Xavier ilkletmesi*nin altında yatan mantıktır (Glorot and Bengio, 2010). Tipik olarak, Xavier ilkletmesi sıfır ortalama ve $\sigma^2 = \frac{2}{n_{\text{in}}+n_{\text{out}}}$ varyansına sahip bir Gauss dağılımından ağırlıkları örnekler. Aynı zamanda, tekdüze bir dağılımdan ağırlıkları örneklerken, Xavier'in sezgisini varyansı seçecek şekilde uyarlayabiliriz. $U(-a, a)$ tekdüze dağılımının $\frac{a^2}{3}$ varyansına sahip olduğuna dikkat edin. $\frac{a^2}{3}$'ı σ^2'daki koşulumuza eklemek, şu ilkletme önerisini verir:

$$U\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right). \tag{4.8.6}$$

Yukarıdaki matematiksel akıl yürütmede doğrusal olmayanların var olmadığı varsayımı, sinir ağlarında kolayca ihlal edilebilse de, Xavier ilkletme yönteminin pratikte iyi çalıştığı ortaya çıkıyor.

Daha Fazlası

Yukarıdaki mantık, parametre ilklendirmesine yönelik modern yaklaşımların yüzeyine ışık tutar. Bir derin öğrenme çerçevesi genellikle bir düzineden fazla farklı buluşsal yöntem uygular. Dahası, parametre ilkletme, derin öğrenmede temel araştırma için sıcak bir alan olmaya devam ediyor. Bunlar arasında bağlı (paylaşılan) parametreler, süper çözünürlük, dizi modelleri ve diğer durumlar için özelleştirilmiş buluşsal yöntemler bulunur. Örneğin, Xiao ve ark. dikkatle tasarlanmış bir ilkletme yöntemi kullanarak mimari hileler olmadan 10000 katmanlı sinir ağlarını eğitme olasılığını gösterdi (Xiao *et al.*, 2018).

Konu ilginizi çekiyorsa, bu modülün önerdiklerine derinlemesine dalmanızı, her buluşsal yöntemi öneren ve analiz eden çalışmaları okumanızı ve ardından konuyla ilgili en son yayınlarda gezinmenizi öneririz. Belki de zekice bir fikre rastlarsınız, hatta icat edersiniz ve derin öğrenme çerçevelerinde bir uygulamaya katkıda bulunursunuz.

4.8.3 Özet

- Kaybolan ve patlayan gradyanlar, derin ağlarda yaygın sorunlardır. Gradyanların ve parametrelerin iyi kontrol altında kalmasını sağlamak için parametre ilklemeye büyük özen gösterilmesi gerekir.
- İlk gradyanların ne çok büyük ne de çok küçük olmasını sağlamak için ilkleme buluşsal yöntemlerine ihtiyaç vardır.
- ReLU etkinleştirme fonksiyonları, kaybolan gradyan problemini azaltır. Bu yakınsamayı hızlandırabilir.
- Rastgele ilkleme, optimizasyondan önce bakışımın bozulmasını sağlamak için bir araçtır.
- Xavier ilkletme, her katman için herhangi bir çıktının varyansının girdi sayısından etkilenmediğini ve herhangi bir gradyanın varyansının çıktı sayısından etkilenmediğini öne sürer.

4.8.4 Alıştırmalar

1. Bir sinir ağının, bir MLP'nin, katmanlarında yer değiştirme bakışımının yanısıra kırılma gerektiren bakışım sergileyebileceği başka durumlar tasarlayabilir misiniz?
2. Doğrusal regresyonda veya softmaks regresyonunda tüm ağırlık parametrelerini aynı değere ilkleyebilir miyiz?
3. İki matrisin çarpımının özdeğerlerindeki analitik sınırlara bakınız. Bu, gradyanların iyi şartlandırılmasının sağlanması konusunda size ne anlatıyor?
4. Bazı terimlerin ıraksadığını biliyorsak, bunu sonradan düzeltebilir miyiz? İlham almak için katmanlı uyarlanabilir oran ölçekleme makalesine bakınız (You *et al.*, 2017).

Tartışmalar[72]

4.9 Ortam ve Dağılım Kayması

Önceki bölümlerde, modelleri çeşitli veri kümelerine oturtarak bir dizi uygulamalı makine öğrenmesi üzerinde çalıştık. Yine de, verilerin ilk etapta nereden geldiğini veya modellerimizin çıktılarıyla sonuçta yapmayı planladığımız şeyi düşünmeyi asla bırakmadık. Sıklıkla, verilere sahip olan makine öğrenmesi geliştiricileri, bu temel sorunları dikkate alıp üstlerinde duraklamadan modeller geliştirmek için acele ederler.

Başarısız olan birçok makine öğrenmesi konuşlandırılması bu örüntü ile köklendirilebilir. Bazen modeller, test kümesi doğruluğu ile ölçüldüğünde harika bir performans sergiliyor gibi görünebilir, ancak veri dağılımı aniden değiştiğinde, ürün olarak konuşlandırılınca felaket bir şekilde başarısız olur. Daha sinsi bir şekilde, bazen bir modelin konuşlandırılması, veri dağılımını bozan katalizör olabilir. Örneğin, bir krediyi kimin geri ödeyeceğini tahmin etmek için bir model

[72] https://discuss.d2l.ai/t/104

eğittiğimizi ve başvuranın ayakkabı seçiminin temerrüt (geri ödememe) riskiyle ilişkili olduğunu tespit ettiğimizi varsayalım (Oxfords türü ayakkabılar geri ödemeyi gösterir, spor ayakkabılar temerrüdü gösterir). Daha sonra Oxfords giyen tüm başvuru sahiplerine kredi verme ve spor ayakkabı giyen tüm başvuru sahiplerini reddetme eğiliminde olabiliriz.

Bu durumda, örüntü tanımadan karar vermeye yanlış düşünülmüş bir sıçrayışımız ve ortamı eleştirel bir şekilde dikkate almadaki başarısızlığımız feci sonuçlar doğurabilir. Başlangıç olarak, ayakkabı seçimlerine dayalı kararlar almaya başlar başlamaz, müşteriler farkına varır ve davranışlarını değiştirirdi. Çok geçmeden, tüm başvuru sahipleri, kredi güvenirliliğinde herhangi bir tesadüfi gelişme olmaksızın Oxfords giyeceklerdi. Bunu sindirmek için bir dakikanızı ayırın çünkü makine öğrenmesinin pek çok uygulamasında benzer sorunlar bolca bulunur: Modele dayalı kararlarımızı ortama sunarak modeli bozabiliriz.

Bu konuları tek bir bölümde tam olarak ele alamasak da, burada bazı ortak endişeleri ortaya çıkarmayı ve bu durumları erken tespit etmek, hasarı azaltmak ve makine öğrenmesini sorumlu bir şekilde kullanmak için gereken eleştirel düşünmeyi teşvik etmeyi amaçlıyoruz. Çözümlerden bazıları basittir ("doğru" veriyi istemek), bazıları teknik olarak zordur (pekiştirmeli öğrenme sistemi uygulamak) ve diğerleri, istatistiksel tahmin alanının tamamen dışına çıkmamızı ve algoritmaların etik uygulanması gibi konularla ilgili zor felsefi sorularla boğuşmamızı gerektirir.

4.9.1 Dağılım Kayması Türleri

Başlangıç olarak, veri dağılımlarının değişebileceği çeşitli yolları ve model performansını kurtarmak için neler yapılabileceğini göz önünde bulundurarak pasif tahminleme ayarına sadık kalıyoruz. Klasik bir kurulumda, eğitim verilerimizin herhangi bir $p_S(\mathbf{x}, y)$ dağılımından örneklendiğini, ancak etiketsiz test verilerimizin farklı bir $p_T(\mathbf{x}, y)$ dağılımından çekildiğini varsayalım. Şimdiden ciddi bir gerçekle yüzleşmeliyiz. p_S ve p_T'nin birbiriyle nasıl ilişkili olduğuna dair herhangi bir varsayım olmadığından, gürbüz bir sınıflandırıcı öğrenmek imkansızdır.

Köpekler ve kediler arasında ayrım yapmak istediğimiz bir ikili sınıflandırma problemini düşünün. Dağılım keyfi şekillerde kayabiliyorsa, kurulumumuz girdiler üzerinden dağılımın sabit kaldığı patolojik bir duruma izin verir: $p_S(\mathbf{x}) = p_T(\mathbf{x})$ ancak etiketlerin tümü ters çevrilmiştir $p_S(y|\mathbf{x}) = 1 - p_T(y|\mathbf{x})$. Başka bir deyişle, eğer Tanrı aniden gelecekte tüm "kedilerin" artık köpek olduğuna ve daha önce "köpek" dediğimiz şeyin artık kedi olduğuna karar verebilirse—$p(\mathbf{x})$ girdilerinin dağılımında herhangi bir değişiklik olmaksızın, bu kurulumu dağılımın hiç değişmediği bir kurulumdan ayırt edemeyiz.

Neyse ki, verilerimizin gelecekte nasıl değişebileceğine dair bazı kısıtlı varsayımlar altında, ilkeli algoritmalar kaymayı algılayabilir ve hatta bazen anında kendilerini uyarlayarak orijinal sınıflandırıcının doğruluğunu iyileştirebilir.

Ortak Değişken Kayması

Dağılım kayması kategorileri arasında, ortak değişken kayması en yaygın olarak çalışılmışı olabilir. Burada, girdilerin dağılımının zamanla değişebileceğini varsayıyoruz, etiketleme fonksiyonu, yani koşullu dağılım $P(y \mid \mathbf{x})$ değişmez. İstatistikçiler buna *ortak değişken kayması* diyorlar çünkü problem ortak değişkenlerin (öznitelikler) dağılımındaki bir kayma nedeniyle ortaya çıkıyor. Bazen nedenselliğe başvurmadan dağılım kayması hakkında akıl yürütebiliyor olsak da, ortak değişken kaymasının $\mathbf{x}$'in y'ye neden olduğuna inandığımız durumlarda çağrılacak doğal varsayım olduğuna dikkat ediyoruz.

Kedileri ve köpekleri ayırt etmenin zorluğunu düşünün. Eğitim verilerimiz Fig. 4.9.1 içindeki türden resimlerden oluşabilir:

Fig. 4.9.1: Kedi ve köpek ayrımında eğitim kümesi.

Test zamanında Fig. 4.9.2 içindeki resimleri sınıflandırmamız istenebilir:

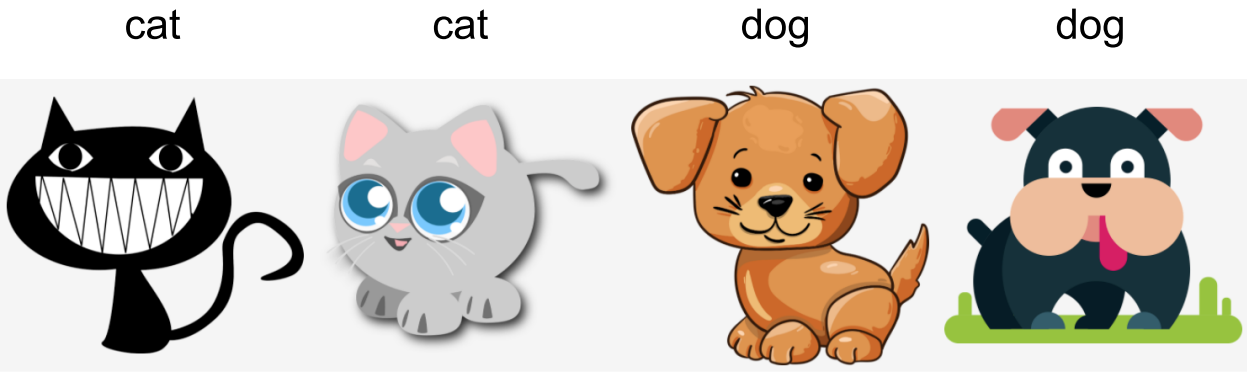


Fig. 4.9.2: Kedi ve köpek ayrımında test kümesi.

Eğitim kümesi fotoğraflardan oluşurken, test kümesi sadece çizimlerden oluşuyor. Test kümesinden büyük ölçüde farklı özelliklere sahip bir veri kümesi üzerinde eğitim, yeni etki alanına nasıl adapte olacağına dair tutarlı bir planın olmaması sorununu yaratabilir.

Etiket Kayması

Etiket kayması ters problemi tanımlar. Burada, etiketin marjinali $P(y)$'nin değişebileceğini varsayıyoruz ancak sınıf koşullu dağılım $P(\mathbf{x} \mid y)$ etki alanları arasında sabit kalır. Etiket kayması, y'nin $\mathbf{x}$'e neden olduğuna inandığımızda yaptığımız makul bir varsayımdır. Örneğin, tanıların göreceli yaygınlığı zamanla değişse bile, semptomları (veya diğer belirtileri) verilen tanıları tahmin etmek isteyebiliriz. Etiket kayması burada uygun varsayımdır çünkü hastalıklar semptomlara neden olur. Bazı yozlaşmış durumlarda, etiket kayması ve ortak değişken kayma varsayımları aynı anda geçerli olabilir. Örneğin, etiket belirlenimci olduğunda, y $\mathbf{x}$'e neden olsa bile, ortak değişken kayma varsayımı karşılanacaktır. İlginç bir şekilde, bu durumlarda, etiket kayması varsayımından kaynaklanan yöntemlerle çalışmak genellikle avantajlıdır. Bunun nedeni, derin öğrenmede yüksek boyutlu olma eğiliminde olan girdiye benzeyen nesnelerin aksine, bu yöntemlerin etiketlere benzeyen (genellikle düşük boyutlu) nesnelerde oynama yapmak eğiliminde olmasıdır.

Kavram Kayması

Ayrıca etiketlerin tanımları değiştiğinde ortaya çıkan ilgili *kavram kayması* sorunuyla da karşılaşabiliriz. Kulağa garip geliyor—bir *kedi* bir *kedi*dir, değil mi? Ancak, diğer kategoriler zaman içinde kullanımda değişikliklere tabidir. Akıl hastalığı için tanı kriterleri, modaya uygun olanlar ve iş unvanları önemli miktarda kavram kaymasına tabidir. Amerika Birleşik Devletleri çevresinde dolaşırsak, verilerimizin kaynağını coğrafyaya göre değiştirirsek, *meşrubat* adlarının dağılımıyla ilgili olarak, Fig. 4.9.3 içinde gösterildiği gibi önemli bir kavram kayması bulacağımız ortaya çıkar.

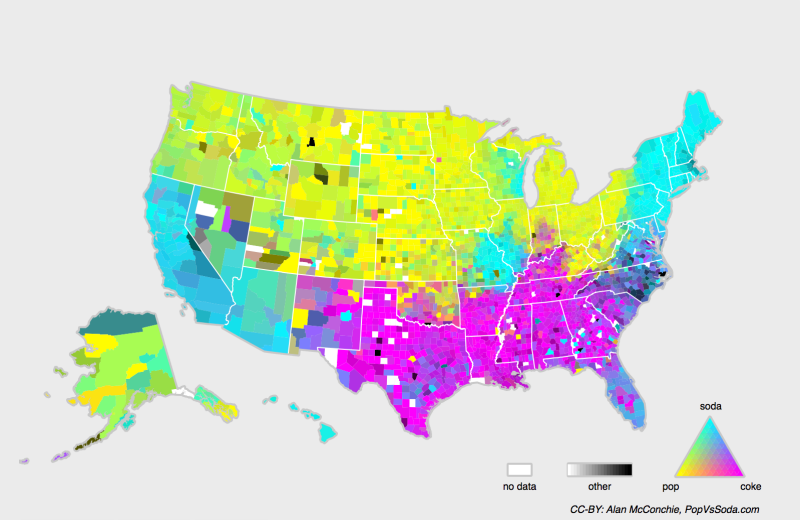


Fig. 4.9.3: Amerika Birleşik Devletleri'nde meşrubat isimlerinde kavram değişikliği.

Bir makine çeviri sistemi kuracak olsaydık, $P(y \mid \mathbf{x})$ dağılımı konumumuza bağlı olarak farklı olabilirdi. Bu sorunu tespit etmek zor olabilir. Değişimin yalnızca kademeli olarak gerçekleştiği bilgiden hem zamansal hem de coğrafi anlamda yararlanmayı umabiliriz.

Dağılım Kayması Örnekleri

Biçimselliğe ve algoritmalara girmeden önce, ortak değişken veya kavram kaymasının bariz olmayabileceği bazı somut durumları tartışabiliriz.

Tıbbi Teşhis

Kanseri tespit etmek için bir algoritma tasarlamak istediğinizi hayal edin. Sağlıklı ve hasta insanlardan veri topluyorsunuz ve algoritmanızı geliştiriyorsunuz. İyi çalışıyor, size yüksek doğruluk sağlıyor ve tıbbi teşhis alanında başarılı bir kariyere hazır olduğunuz sonucuna varıyorsunuz. *O kadar da hızlı değil.*

Eğitim verilerini ortaya çıkaranlar ile gerçek hayatta karşılaşacağınız dağılımlar önemli ölçüde farklılık gösterebilir. Bu, bizden bazılarının (biz yazarlar) yıllar önce çalıştığı talihsiz bir girişimin başına geldi. Çoğunlukla yaşlı erkekleri etkileyen bir hastalık için bir kan testi geliştiriyorlardı ve hastalardan topladıkları kan örneklerini kullanarak bu hastalığı incelemeyi umuyorlardı. Bununla birlikte, sağlıklı erkeklerden kan örnekleri almak, sistemdeki mevcut hastalardan almaktan çok daha zordur. Girişim, bununla başa çıkmak için, bir üniversite kampüsündeki öğrencilerden testlerini geliştirmede sağlıklı kontrol grubu olmaları amacıyla kan bağışı istedi. Ardından,

onlara hastalığı tespit etmek için bir sınıflandırıcı oluşturmalarına yardım edip edemeyeceğimiz soruldu.

Onlara açıkladığımız gibi, sağlıklı ve hasta grupları neredeyse mükemmel bir doğrulukla ayırt etmek gerçekten kolay olurdu. Çünkü, bunun nedeni, deneklerin yaş, hormon seviyeleri, fiziksel aktivite, diyet, alkol tüketimi ve hastalıkla ilgisi olmayan daha birçok faktör açısından farklılık göstermesidir. Gerçek hastalarda durum böyle değildi. Örneklem prosedürleri nedeniyle, aşırı ortak değişken kayması ile karşılaşmayı bekleyebiliriz. Dahası, bu durumun geleneksel yöntemlerle düzeltilmesi pek olası değildir. Kısacası, önemli miktarda para israf ettiler.

Kendi Kendine Süren Arabalar

Bir şirketin sürücüsüz otomobiller geliştirmek için makine öğrenmesinden yararlanmak istediğini varsayalım. Buradaki temel bileşenlerden biri yol kenarı detektörüdür. Gerçek açıklamalı verilerin elde edilmesi pahalı olduğu için, bir oyun oluşturma motorundan gelen sentetik verileri ek eğitim verileri olarak kullanma (zekice ve şüpheli) fikirleri vardı. Bu, işleme motorundan alınan "test verileri" üzerinde gerçekten iyi çalıştı. Ne yazık ki, gerçek bir arabanın içinde tam bir felaketti. Görünüşe göre oluşturulan yol kenarı çok basit bir dokuya sahipti. Daha da önemlisi, *tüm* yol kenarı *aynı* dokuya sahipti ve yol kenarı dedektörü bu "özniteliği" çok çabuk öğrendi.

ABD Ordusu ormandaki tankları ilk defa tespit etmeye çalıştıklarında da benzer bir şey oldu. Ormanın tanksız hava fotoğraflarını çektiler, ardından tankları ormana sürdüler ve bir dizi fotoğraf daha çektiler. Sınıflandırıcının *mükemmel* çalıştığı görüldü. Ne yazık ki, o sadece gölgeli ağaçları gölgesiz ağaçlardan nasıl ayırt edeceğini öğrenmişti—ilk fotoğraf kümesi sabah erken, ikinci küme öğlen çekilmişti.

Durağan Olmayan Dağılımlar

Dağılım yavaş değiştiğinde ve model yeterince güncellenmediğinde çok daha hassas bir durum ortaya çıkar (*durağan olmayan dağılımlar* da diye de bilinir). Bazı tipik durumlar aşağıdadır:

- Bir hesaplamalı reklamcılık modelini eğitiyor ve ardından onu sık sık güncellemekte başarısız oluyoruz (örneğin, iPad adı verilen belirsiz yeni bir cihazın henüz piyasaya sürüldüğünü dahil etmeyi unuttuk).
- Bir yaramaz posta filtresi oluşturuyoruz. Şimdiye kadar gördüğümüz tüm yaramaz postaları tespit etmede iyi çalışıyor. Ancak daha sonra, yaramaz posta gönderenler akıllanıyor ve daha önce gördüğümüz hiçbir şeye benzemeyen yeni mesajlar oluşturuyorlar.
- Ürün öneri sistemi oluşturuyoruz. Kış boyunca işe yarıyor, ancak Noel'den sonra da Noel Baba şapkalarını önermeye devam ediyor.

Birkaç Benzer Tecrübe

- Yüz dedektörü yapıyoruz. Tüm kıyaslamalarda iyi çalışıyor. Ne yazık ki test verilerinde başarısız oluyor—zorlayıcı örnekler, yüzün tüm resmi doldurduğu yakın çekimlerdir (eğitim kümesinde böyle bir veri yoktu).
- ABD pazarı için bir web arama motoru oluşturuyoruz ve bunu Birleşik Krallık'ta kullanmak istiyoruz.

- Büyük bir sınıf kümesinin her birinin veri kümesinde eşit olarak temsil edildiği büyük bir veri kümesi derleyerek bir imge sınıflandırıcı eğitiyoruz, örneğin 1000 kategori var ve her biri 1000 görüntü ile temsil ediliyor. Ardından sistemi, fotoğrafların gerçek etiket dağılımının kesinlikle tekdüze olmadığı gerçek dünyada konuşlandırıyoruz.

4.9.2 Dağılım Kaymasını Düzeltme

Daha önce tartıştığımız gibi, $P(\mathbf{x}, y)$ eğitim ve test dağılımlarının farklı olduğu birçok durum vardır. Bazı durumlarda şanslıyız ve modeller ortak değişken, etiket veya kavram kaymasına rağmen çalışıyor. Diğer durumlarda, kaymalarla başa çıkmak için ilkeli stratejiler kullanarak daha iyisini yapabiliriz. Bu bölümün geri kalanı önemli ölçüde daha teknik hale geliyor. Sabırsız okuyucu bir sonraki bölüme geçebilir çünkü bu içerik sonraki kavramlar için ön koşul değildir.

Deneysel Risk ve Risk

Önce model eğitimi sırasında tam olarak ne olduğunu düşünelim: Eğitim verilerinin öznitelikleri ve ilişkili etiketleri $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ üzerinde yineleniriz ve her minigruptan sonra f modelinin parametrelerini güncelleriz. Basit olması için düzenlileştirmeyi düşünmüyoruz, böylece eğitimdeki kaybı büyük ölçüde en aza indiriyoruz:

$$\underset{f}{\text{minimize}} \frac{1}{n} \sum_{i=1}^{n} l(f(\mathbf{x}_i), y_i), \tag{4.9.1}$$

burada l, ilişkili y_i etiketi ile verilen $f(\mathbf{x}_i)$ tahmininin "ne kadar kötü" olduğunu ölçen kayıp işlevidir. İstatistikçiler (4.9.1) içindeki bu terimi *deneysel risk* olarak adlandırır. *Deneysel risk*, $p(\mathbf{x}, y)$ gerçek dağılımından elde edilen tüm veri popülasyonu üzerindeki kaybın beklentisi olan *riske* yaklaştıran eğitim verileri üzerindeki ortalama kayıptır:

$$E_{p(\mathbf{x},y)}[l(f(\mathbf{x}), y)] = \int \int l(f(\mathbf{x}), y) p(\mathbf{x}, y) \, d\mathbf{x} dy. \tag{4.9.2}$$

Bununla birlikte, pratikte tipik olarak tüm veri popülasyonunu elde edemeyiz. Bu nedenle, (4.9.1) içindeki deneysel riski en aza indiren *deneysel risk minimizasyonu*, riski yaklaşık olarak en aza indirmeyi uman makine öğrenmesi için pratik bir stratejidir.

Ortak Değişken Kaymasını Düzeltme

Verileri $(\mathbf{x}_i, y_i)$ olarak etiketlediğimiz $P(y \mid \mathbf{x})$ bağımlılığını tahmin etmek istediğimizi varsayalım. Ne yazık ki, $\mathbf{x}_i$ gözlemleri, *hedef dağılımı* $p(\mathbf{x})$ yerine bazı *kaynak dağılımı* $q(\mathbf{x})$'den alınmıştır. Neyse ki, bağımlılık varsayımı koşullu dağılımın değişmediği anlamına gelir: $p(y \mid \mathbf{x}) = q(y \mid \mathbf{x})$. $q(\mathbf{x})$ kaynak dağılımı "yanlış" ise, riskte aşağıdaki basit özdeşlik kullanarak bunu düzeltebiliriz:

$$\int \int l(f(\mathbf{x}), y) p(y \mid \mathbf{x}) p(\mathbf{x}) \, d\mathbf{x} dy = \int \int l(f(\mathbf{x}), y) q(y \mid \mathbf{x}) q(\mathbf{x}) \frac{p(\mathbf{x})}{q(\mathbf{x})} \, d\mathbf{x} dy. \tag{4.9.3}$$

Başka bir deyişle, her bir veri örneğini, doğru dağılımdan alınmış olma olasılığının yanlış dağılımdan alınmış olma olasılığına oranıyla yeniden ağırlıklandırmamız gerekir:

$$\beta_i \stackrel{\text{def}}{=} \frac{p(\mathbf{x}_i)}{q(\mathbf{x}_i)}. \tag{4.9.4}$$

Her veri örneği $(\mathbf{x}_i, y_i)$ için β_i ağırlığını ekleyerek modelimizi *ağırlıklı deneysel risk minimizasyonu* kullanarak eğitebiliriz:

$$\underset{f}{\text{minimize}} \frac{1}{n} \sum_{i=1}^{n} \beta_i l(f(\mathbf{x}_i), y_i). \tag{4.9.5}$$

Ne yazık ki, bu oranı bilmiyoruz, bu yüzden faydalı bir şey yapmadan önce onu tahmin etmemiz gerekiyor. Beklenti işlemini bir minimum norm veya bir maksimum entropi ilkesi kullanarak doğrudan yeniden ayar etmeye çalışan bazı süslü operatör-teorik yaklaşımlar dahil olmak üzere birçok yöntem mevcuttur. Bu tür herhangi bir yaklaşım için, her iki dağılımdan da alınan örneklere ihtiyacımız olduğuna dikkat edin: "Doğru" p, örneğin, test verilerine erişim yoluyla elde edilen ve q eğitim kümesini oluşturmak için kullanılan (sonraki bariz mevcuttur). Ancak, yalnızca $\mathbf{x} \sim p(\mathbf{x})$ özniteliklerine ihtiyacımız olduğunu unutmayın; $y \sim p(y)$ etiketlerine erişmemize gerek yok.

Bu durumda, neredeyse orijinali kadar iyi sonuçlar verecek çok etkili bir yaklaşım vardır: İkili sınıflandırmada softmaks regresyonun özel bir durumu olan (bkz. Section 3.4) lojistik regresyon. Tahmini olasılık oranlarını hesaplamak için gereken tek şey budur. $p(\mathbf{x})$'den alınan veriler ile $q(\mathbf{x})$'den alınan verileri ayırt etmek için bir sınıflandırıcı öğreniyoruz. İki dağılım arasında ayrım yapmak imkansızsa, bu, ilişkili örneklerin iki dağılımdan birinden gelme olasılığının eşit olduğu anlamına gelir. Öte yandan, iyi ayırt edilebilen herhangi bir örnek, buna göre önemli ölçüde yüksek veya düşük ağırlıklı olmalıdır.

Basit olması açısından, her iki dağılımdan da eşit sayıda örneğe sahip olduğumuzu ve sırasıyla $p(\mathbf{x})$ ve $q(\mathbf{x})$ diye gösterildiklerini varsayalım. Şimdi, p'den alınan veriler için 1 ve q'dan alınan veriler için -1 olan z etiketlerini belirtelim. Daha sonra, karışık bir veri kümesindeki olasılık şu şekilde verilir:

$$P(z = 1 \mid \mathbf{x}) = \frac{p(\mathbf{x})}{p(\mathbf{x}) + q(\mathbf{x})} \text{ ve bu yüzden } \frac{P(z = 1 \mid \mathbf{x})}{P(z = -1 \mid \mathbf{x})} = \frac{p(\mathbf{x})}{q(\mathbf{x})}. \tag{4.9.6}$$

Bu nedenle, lojistik regresyon yaklaşımını kullanırsak, öyleki $P(z = 1 \mid \mathbf{x}) = \frac{1}{1+\exp(-h(\mathbf{x}))}$ (h parametreli bir fonksiyondur), aşağıdaki sonuca varırız:

$$\beta_i = \frac{1/(1 + \exp(-h(\mathbf{x}_i)))}{\exp(-h(\mathbf{x}_i))/(1 + \exp(-h(\mathbf{x}_i)))} = \exp(h(\mathbf{x}_i)). \tag{4.9.7}$$

Sonuç olarak, iki sorunu çözmemiz gerekiyor: İlk olarak her iki dağılımdan alınan verileri ayırt etme ve ardından terimleri β_i ile ağırlıklandırdığımız (4.9.5) içindeki ağırlıklı deneysel risk minimizasyon problemi.

Artık bir düzeltme algoritması tanımlamaya hazırız. $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ eğitim kümemiz ve etiketlenmemiş bir $\{\mathbf{u}_1, \ldots, \mathbf{u}_m\}$ test kümemiz olduğunu varsayalım. Ortak değişken kaydırma için, tüm $1 \leq i \leq n$ için $\mathbf{x}_i$'nin bir kaynak dağılımından ve tüm $1 \leq i \leq m$ için $\mathbf{u}_i$'in hedef dağılımdan çekildiğini varsayıyoruz. İşte ortak değişken kaymasını düzeltmek için prototipik bir algoritma:

1. Bir ikili sınıflandırma eğitim kümesi oluşturun: $\{(\mathbf{x}_1, -1), \ldots, (\mathbf{x}_n, -1), (\mathbf{u}_1, 1), \ldots, (\mathbf{u}_m, 1)\}$.
2. h fonksiyonunu elde etmek için lojistik regresyon kullanarak bir ikili sınıflandırıcı eğitin.
3. $\beta_i = \exp(h(\mathbf{x}_i))$ veya daha iyisi $\beta_i = \min(\exp(h(\mathbf{x}_i)), c)$, c herhangi bir sabittir, kullanarak eğitim verilerini ağırlıklandırın.
4. $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ üzerinde eğitim için (4.9.5) içindeki β_i ağırlıklarını kullanın.

Yukarıdaki algoritmanın önemli bir varsayıma dayandığını unutmayın. Bu düzenin çalışması için, hedef (örn. test zamanı) dağılımındaki her veri örneğinin eğitim zamanında meydana gelme olasılığının sıfır olmayan bir şekilde olması gerekir. $p(\mathbf{x}) > 0$ ama $q(\mathbf{x}) = 0$ olan bir nokta bulursak, buna karşılık gelen önem ağırlığı sonsuz olmalıdır.

Etiket Kaymasını Düzeltme

k kategorili bir sınıflandırma göreviyle uğraştığımızı varsayalım. Section 4.9.2 içindeki aynı gösterimi kullanarak, q ve p sırasıyla kaynak dağılımı (ör. eğitim zamanı) ve hedef dağılımıdır (ör. test zamanı). Etiketlerin dağılımının zaman içinde değiştiğini varsayın: $q(y) \neq p(y)$, ancak sınıf-koşullu dağılım aynı kalır: $q(\mathbf{x} \mid y) = p(\mathbf{x} \mid y)$. $q(y)$ kaynak dağılımı "yanlış" ise, bunu (4.9.2) içinde tanımlanan riskte aşağıdaki özdeşliğe göre düzeltebiliriz:

$$\int\int l(f(\mathbf{x}), y)p(\mathbf{x} \mid y)p(y) \; d\mathbf{x}dy = \int\int l(f(\mathbf{x}), y)q(\mathbf{x} \mid y)q(y)\frac{p(y)}{q(y)} \; d\mathbf{x}dy. \tag{4.9.8}$$

Burada önem ağırlıklarımız etiket olabilirlik oranlarına karşılık gelecektir.

$$\beta_i \stackrel{\text{def}}{=} \frac{p(y_i)}{q(y_i)}. \tag{4.9.9}$$

Etiket kayması ile ilgili güzel bir şey, kaynak dağılımı üzerinde oldukça iyi bir modelimiz varsa, ortam boyutuyla hiç uğraşmadan bu ağırlıkların tutarlı tahminlerini elde edebilmemizdir. Derin öğrenmede girdiler, imgeler gibi yüksek boyutlu nesneler olma eğilimindeyken, etiketler genellikle kategoriler gibi daha basit nesnelerdir.

Hedef etiket dağılımını tahmin etmek için, önce makul ölçüde iyi olan kullanıma hazır mevcut sınıflandırıcımızı (tipik olarak eğitim verileri üzerinde eğitilmiştir) alıp geçerleme kümesini kullanarak (o da eğitim dağılımından) hata matrisini hesaplıyoruz. *Hata matrisi*, $\mathbf{C}$, basitçe bir $k \times k$ matrisidir, burada her sütun etiket sınıfına (temel doğru) ve her satır, modelimizce tahmin edilen sınıfa karşılık gelir. Her bir hücrenin değeri c_{ij}, geçerleme kümesinde gerçek etiketin j olduğu ve modelimizin i tahmin ettiği toplam tahminlerin oranıdır.

Şimdi, hedef verilerdeki hata matrisini doğrudan hesaplayamayız, çünkü karmaşık bir gerçek zamanlı açıklama veri işleme hattına yatırım yapmazsak, gerçek hayatta gördüğümüz örneklerin etiketlerini göremeyiz. Ancak yapabileceğimiz şey, birlikte test zamanında tüm model tahminlerimizin ortalamasıdır ve ortalama model çıktılarını $\mu(\hat{\mathbf{y}}) \in \mathbb{R}^k$ verir, ki $i.$ öğesi $\mu(\hat{y}_i)$, modelimizin i tahmin ettiği test kümesindeki toplam tahminlerin oranıdır.

Bazı ılımlı koşullar altında — eğer sınıflandırıcımız ilk etapta makul ölçüde doğruysa ve hedef veriler yalnızca daha önce gördüğümüz kategorileri içeriyorsa ve ilk etapta etiket kayması varsayımı geçerliyse (en güçlü varsayım), o zaman basit bir doğrusal sistemi çözerek test kümesi etiket dağılımını tahmin edebiliriz.

$$\mathbf{C}p(\mathbf{y}) = \mu(\hat{\mathbf{y}}), \tag{4.9.10}$$

çünkü bir tahmin olarak $\sum_{j=1}^{k} c_{ij}p(y_j) = \mu(\hat{y}_i)$ tüm $1 \leq i \leq k$ için geçerlidir, burada $p(y_j)$, k boyutlu etiket dağılım vektörü $p(\mathbf{y})$'nin $j.$ öğesidir. Sınıflandırıcımız başlangıç için yeterince doğruysa, o zaman $\mathbf{C}$ hata matrisi tersine çevrilebilir ve $p(\mathbf{y}) = \mathbf{C}^{-1}\mu(\hat{\mathbf{y}})$ çözümünü elde ederiz.

Kaynak verilerdeki etiketleri gözlemlediğimiz için $q(y)$ dağılımını tahmin etmek kolaydır. Ardından, y_i etiketli herhangi bir eğitim örneği i için, β_i ağırlığını hesaplamak için tahmini $p(y_i)/q(y_i)$ oranını alabilir ve bunu (4.9.5) içindeki ağırlıklı deneysel risk minimizasyonuna bağlayabiliriz.

Kavram Kayması Düzeltmesi

Kavram kaymasını ilkeli bir şekilde düzeltmek çok daha zordur. Örneğin, sorunun birdenbire kedileri köpeklerden ayırt etmekten beyaz siyah hayvanları ayırmaya dönüştüğü bir durumda, yeni etiketler toplamadan ve sıfırdan eğitmeden çok daha iyisini yapabileceğimizi varsaymak mantıksız olacaktır. Neyse ki pratikte bu tür aşırı değişimler nadirdir. Bunun yerine, genellikle olan şey, görevin yavaş yavaş değişmeye devam etmesidir. İşleri daha somut hale getirmek için işte bazı örnekler:

- Hesaplamalı reklamcılıkta yeni ürünler piyasaya sürülür, eski ürünler daha az popüler hale gelir. Bu, reklamlar üzerindeki dağılımın ve popülerliğinin kademeli olarak değiştiği ve herhangi bir tıklama oranı tahmincisinin bununla birlikte kademeli olarak değişmesi gerektiği anlamına gelir.
- Trafik kamerası lensleri, çevresel aşınma nedeniyle kademeli olarak bozulur ve görüntü kalitesini aşamalı olarak etkiler.
- Haber içeriği kademeli olarak değişir (yani, haberlerin çoğu değişmeden kalır, ancak yeni hikayeler ortaya çıkar).

Bu gibi durumlarda, ağları verilerdeki değişime adapte etmek için eğitim ağlarında kullandığımız yaklaşımı kullanabiliriz. Başka bir deyişle, mevcut ağ ağırlıklarını kullanıyoruz ve sıfırdan eğitim yerine yeni verilerle birkaç güncelleme adımı gerçekleştiriyoruz.

4.9.3 Öğrenme Sorunlarının Sınıflandırması

Dağılımlardaki değişikliklerle nasıl başa çıkılacağı hakkında bilgi sahibi olarak, şimdi makine öğrenmesi problem formülasyonunun diğer bazı yönlerini ele alabiliriz.

Toplu Öğrenme

Toplu öğrenmede, bir model, $f(\mathbf{x})$, eğitmek için kullandığımız $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ eğitim özniteliklerine ve etiketlerine erişimimiz var. Daha sonra, aynı dağılımdan çekilen yeni verileri $(\mathbf{x}, y)$ değerlendirmek için bu modeli konuşlandırabiliriz. Bu, burada tartıştığımız sorunların herhangi biri için varsayılan varsayımdır. Örneğin, birçok kedi ve köpek resmine dayalı bir kedi dedektörü eğitebiliriz. Onu eğittikten sonra, sadece kedilerin içeri girmesine izin veren akıllı bir kedi kapısı bilgisayarla görme sisteminin parçası olarak göndeririz. Bu daha sonra bir müşterinin evine kurulur ve bir daha asla güncellenmez (aşırı durumlar hariç).

Çevrimiçi Öğrenme

Şimdi $(\mathbf{x}_i, y_i)$ verisinin her seferinde bir örnek olarak geldiğini hayal edin. Daha belirleyici olarak, önce $\mathbf{x}_i$'i gözlemlediğimizi, ardından bir $f(\mathbf{x}_i)$ tahmini bulmamız gerektiğini ve yalnızca bunu yaptığımızda y_i'yi gözlemlediğimizi ve bununla bizim kararımıza göre bir ödül veya bir ceza aldığımızı varsayalım. Birçok gerçek sorun bu kategoriye girer. Örneğin, yarınki hisse senedi fiyatını tahmin etmemiz gerekir, bu, bu tahmine dayalı olarak işlem yapmamızı sağlar ve günün sonunda tahminimizin kâr elde etmemize izin verip vermediğini öğreniriz. Başka bir deyişle, *çevrimiçi öğrenme*de, yeni gözlemlerle modelimizi sürekli iyileştirdiğimiz aşağıdaki döngüye sahibiz.

$$\text{model } f_t \longrightarrow \text{veri } \mathbf{x}_t \longrightarrow \text{tahmin } f_t(\mathbf{x}_t) \longrightarrow \text{gözlem } y_t \longrightarrow \text{kayıp } l(y_t, f_t(\mathbf{x}_t)) \longrightarrow \text{model } f_{t+1} \tag{4.9.11}$$

Kollu Kumar Makinesi

Kollu kumar makinesi yukarıdaki problemin özel bir durumudur. Çoğu öğrenme probleminde parametrelerini öğrenmek istediğimiz yerde (örneğin derin bir ağ) sürekli değerlerle parametrize edilmiş bir f fonksiyonumuz varken, bir *kollu kumar makinesi* probleminde çekebileceğimiz sınırlı sayıda kolumuz var, yani, sonlu yapabileceğimiz eylem sayısı. Bu basit problem için optimallik açısından daha güçlü teorik garantilerin elde edilebilmesi çok şaşırtıcı değildir. Onu temel olarak listeliyoruz çünkü bu problem genellikle (kafa karıştırıcı bir şekilde) farklı bir öğrenim ortamı gibi ele alınır.

Kontrol

Çoğu durumda ortam ne yaptığımızı hatırlar. Mutlaka düşmanca bir şekilde değil, ancak sadece hatırlayacak ve yanıt daha önce ne olduğuna bağlı olacaktır. Örneğin, bir kahve kaynatıcı kontrolörü, kaynatıcıyı önceden ısıtıp ısıtmadığına bağlı olarak farklı sıcaklıklar gözlemleyecektir. PID (orantılı integral türev) kontrolcü algoritmaları burada popüler bir seçimdir. Benzer şekilde, bir kullanıcının bir haber sitesindeki davranışı, ona daha önce ne gösterdiğimize bağlı olacaktır (örneğin, çoğu haberi yalnızca bir kez okuyacaktır). Bu tür birçok algoritma, kararlarının daha az rastgele görünmesini sağlamak gibi hareket ettikleri ortamın bir modelini oluşturur. Son zamanlarda, kontrol teorisi (örneğin, PID varyantları), daha iyi çözme ve geri çatma kalitesi elde etmek ve oluşturulan metnin çeşitliliğini ve oluşturulan görüntülerin geri çatma kalitesini iyileştirmek için hiper parametreleri otomatik olarak ayarlamak için de kullanıldı (Shao *et al.*, 2020).

Pekiştirmeli Öğrenme

Hafızalı bir ortamın daha genel durumu olarak ortamın bizimle işbirliği yapmaya çalıştığı durumlarla karşılaşabiliriz (özellikle sıfır toplamlı olmayan oyunlar için işbirlikçi oyunlar) veya çevrenin kazanmaya çalışacağı diğerler durumlarla. Satranç, Go, Tavla veya StarCraft pekiştirmeli öğrenme vakalardan bazılarıdır. Aynı şekilde, otonom arabalar için iyi bir kontrolör inşa etmek isteyebiliriz. Diğer arabaların otonom arabanın sürüş tarzına önemsiz şekillerde tepki vermesi muhtemeldir; örneğin, ondan kaçınmaya çalışmak, bir kazaya neden olmamaya çalışmak ve onunla işbirliği yapmaya çalışmak.

Ortamı Düşünmek

Yukarıdaki farklı durumlar arasındaki önemli bir ayrım, sabit bir ortam durumunda baştan sona işe yaramış olabilecek aynı stratejinin, ortam uyum sağlayabildiğinde baştan sona çalışmayabileceğidir. Örneğin, bir tüccar tarafından keşfedilen bir borsada kar fırsatı, onu kullanmaya başladığında muhtemelen ortadan kalkacaktır. Ortamın değiştiği hız ve tarz, büyük ölçüde uygulayabileceğimiz algoritma türlerini belirler. Örneğin, nesnelerin yalnızca yavaş değişebileceğini bilirsek, herhangi bir tahmini de yalnızca yavaşça değişmeye zorlayabiliriz. Ortamın aniden değişebileceğini bilirsek, ancak çok seyrek olarak, buna izin verebiliriz. Bu tür bilgiler, hevesli veri bilimcilerinin kavram kayması, yani çözmeye çalıştığı problem zamanla değişmesi, ile başa çıkması için çok önemlidir.

4.9.4 Makine Öğrenmesinde Adillik, Hesap Verebilirlik ve Şeffaflık

Son olarak, makine öğrenmesi sistemlerini devreye aldığınızda, yalnızca bir tahmine dayalı modeli optimize etmediğinizi, genellikle kararları (kısmen veya tamamen) otomatikleştirmek için kullanılacak bir araç sağladığınızı hatırlamak önemlidir. Bu teknik sistemler, ortaya çıkan kararlara tabi bireylerin yaşamlarını etkileyebilir. Tahminleri değerlendirmekten kararlara sıçrama, yalnızca yeni teknik soruları değil, aynı zamanda dikkatle değerlendirilmesi gereken bir dizi etik soruyu da gündeme getirir. Tıbbi bir teşhis sistemi kuruyorsak, hangi topluluklar için işe yarayıp hangilerinin işe yaramayacağını bilmemiz gerekir. Bir nüfus altgrubunun refahına yönelik öngörülebilir riskleri gözden kaçırmak, daha düşük düzeyde bakım vermemize neden olabilir. Dahası, karar verme sistemlerini düşündüğümüzde geri adım atmalı ve teknolojimizi nasıl değerlendirdiğimizi yeniden düşünmeliyiz. Bu kapsam değişikliğinin diğer sonuçlarının yanı sıra, *doğruluğun* nadiren doğru ölçü olduğunu göreceğiz. Örneğin, tahminleri eyleme dönüştürürken, genellikle çeşitli şekillerde hataların olası maliyet hassaslığını hesaba katmak isteriz. Bir imgeyi yanlış sınıflandırmanın bir yolu ırkçı bir aldatmaca olarak algılanabilirken, farklı bir kategoriye yanlış sınıflandırma zararsızsa, o zaman eşiklerimizi buna göre ayarlamak ve karar verme protokolünü tasarlarken toplumsal değerleri hesaba katmak isteyebiliriz. Ayrıca tahmin sistemlerinin nasıl geri bildirim döngülerine yol açabileceği konusunda dikkatli olmak istiyoruz. Örneğin, devriye görevlilerini suç oranı yüksek olan alanlara tahsis eden tahmine dayalı polisiye sistemleri düşünün. Endişe verici bir modelin nasıl ortaya çıkacağını kolayca görebiliriz:

1. Suçun daha fazla olduğu mahallelerde daha fazla devriye gezer.
2. Sonuç olarak, bu mahallelerde daha fazla suç keşfedilir ve gelecekteki yinelemeler için mevcut eğitim verilerine eklenir.
3. Daha fazla pozitif örneklere maruz kalan model, bu mahallelerde daha fazla suç öngörür.
4. Bir sonraki yinelemede, güncellenmiş model aynı mahalleyi daha da yoğun bir şekilde hedef alır ve daha fazla suç keşfedilmesine neden olur vb.

Çoğunlukla, bir modelin tahminlerinin eğitim verileriyle birleştirildiği çeşitli mekanizmalar, modelleme sürecinde hesaba katılmaz. Bu, araştırmacıların *kaçak geri bildirim döngüleri* dediği şeye yol açabilir. Ek olarak, ilk etapta doğru sorunu ele alıp almadığımıza dikkat etmek istiyoruz. Tahmine dayalı algoritmalar artık bilginin yayılmasına aracılık etmede büyük bir rol oynuyor. Bir bireyin karşılaşacağı haberler, *Beğendikleri (Liked)* Facebook sayfalarına göre mi belirlenmeli? Bunlar, makine öğrenmesindeki bir kariyerde karşılaşabileceğiniz birçok baskın etik ikilemden sadece birkaçı.

4.9.5 Özet

- Çoğu durumda eğitim ve test kümeleri aynı dağılımdan gelmez. Buna dağılım kayması denir.
- Risk, gerçek dağılımlarından elde edilen tüm veri popülasyonu üzerindeki kayıp beklentisidir. Ancak, bu popülasyonun tamamı genellikle mevcut değildir. Deneysel risk, riski yaklaşık olarak tahmin etmek için eğitim verileri üzerinden ortalama bir kayıptır. Uygulamada, deneysel risk minimizasyonu gerçekleştiriyoruz.
- İlgili varsayımlar altında, ortak değişken ve etiket kayması tespit edilebilir ve test zamanında düzeltilebilir. Bu yanlılığın hesaba katılmaması, test zamanında sorunlu hale gelebilir.
- Bazı durumlarda, ortam otomatik eylemleri hatırlayabilir ve şaşırtıcı şekillerde yanıt verebilir. Modeller oluştururken bu olasılığı hesaba katmalı ve canlı sistemleri izlemeye devam

etmeliyiz, modellerimizin ve ortamın beklenmedik şekillerde dolaşması olasılığına açık olmalıyız.

4.9.6 Alıştırmalar

1. Bir arama motorunun davranışını değiştirdiğimizde ne olabilir? Kullanıcılar ne yapabilir? Peki ya reklamverenler?
2. Bir ortak değişken kayması detektörü uygulayınız. İpucu: Bir sınıflandırıcı oluşturunuz.
3. Bir ortak değişken kayması düzelticisi uygulayınız.
4. Dağılım kaymasının yanı sıra, deneysel riskin riske yaklaşmasını başka ne etkileyebilir?

Tartışmalar[73]

4.10 Kaggle'da Ev Fiyatlarını Tahmin Etme

Artık derin ağlar oluşturmak ve eğitmek için bazı temel araçları sunduğumuza ve bunları ağırlık sönümü ve hattan düşürme gibi tekniklerle düzenlileştirdiğimize göre, tüm bu bilgileri bir Kaggle yarışmasına katılarak uygulamaya koymaya hazırız. Ev fiyat tahmini yarışması başlangıç için harika bir yerdir. Veriler oldukça geneldir ve özel modeller gerektirebilecek (ses veya video gibi) acayip yapılar sergilememektedir. Bart de Cock tarafından 2011 yılında toplanan bu veri kümesi, (De Cock, 2011), 2006–2010 döneminden itibaren Ames, IA'daki ev fiyatlarını kapsamaktadır. Harrison ve Rubinfeld'in (1978) ünlü Boston konut veri kümesi[74]'nden önemli ölçüde daha büyüktür ve hem fazla örneğe, hem de daha fazla özniteliğe sahiptir.

Bu bölümde, veri ön işleme, model tasarımı ve hiper parametre seçimi ayrıntılarında size yol göstereceğiz. Uygulamalı bir yaklaşımla, bir veri bilimcisi olarak kariyerinizde size rehberlik edecek bazı sezgiler kazanacağınızı umuyoruz.

4.10.1 Veri Kümelerini İndirme ve Önbelleğe Alma

Kitap boyunca, indirilen çeşitli veri kümeleri üzerinde modelleri eğitecek ve test edeceğiz. Burada, veri indirmeyi kolaylaştırmak için çeşitli yardımcı fonksiyonlar gerçekleştiriyoruz. İlk olarak, bir dizeyi (veri kümesinin *adını*) hem veri kümesini bulmak için URL'yi hem de dosyanın bütünlüğünü doğrulamak için kullanacağımız SHA-1'i anahtarı içeren bir çokuzluya eşleyen bir `DATA_HUB` sözlüğü tutuyoruz. Tüm bu tür veri kümeleri, adresi `DATA_URL`'ye atanan sitede barındırılmaktadır.

```
import hashlib
import os
import tarfile
import zipfile
import requests

#@save
DATA_HUB = dict()
DATA_URL = 'http://d2l-data.s3-accelerate.amazonaws.com/'
```

[73] https://discuss.d2l.ai/t/105
[74] https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.names

Aşağıdaki `download` işlevi, bir veri kümesini indirir, yerel bir dizinde (varsayılan olarak `../data`) önbelleğe alır ve indirilen dosyanın adını döndürür. Bu veri kümesine karşılık gelen bir dosya önbellek dizininde zaten mevcutsa ve SHA-1'i `DATA_HUB`'da depolananla eşleşiyorsa, kodumuz internetinizi gereksiz indirmelerle tıkamamak için önbelleğe alınmış dosyayı kullanacaktır.

```
def download(name, cache_dir=os.path.join('..', 'data')):  #@save
    """DATA_HUB'a eklenen bir dosyayı indir, yerel dosya adını döndür."""
    assert name in DATA_HUB, f"{name} does not exist in {DATA_HUB}."
    url, sha1_hash = DATA_HUB[name]
    os.makedirs(cache_dir, exist_ok=True)
    fname = os.path.join(cache_dir, url.split('/')[-1])
    if os.path.exists(fname):
        sha1 = hashlib.sha1()
        with open(fname, 'rb') as f:
            while True:
                data = f.read(1048576)
                if not data:
                    break
                sha1.update(data)
        if sha1.hexdigest() == sha1_hash:
            return fname  # Hit cache
    print(f'Downloading {fname} from {url}...')
    r = requests.get(url, stream=True, verify=True)
    with open(fname, 'wb') as f:
        f.write(r.content)
    return fname
```

Ayrıca iki ek fayda işlevini de gerçekleştiriyoruz: Biri bir zip veya tar dosyasını indirip çıkarmak, diğeri ise bu kitapta kullanılan veri kümelerinin tümünü `DATA_HUB`'dan önbellek dizinine indirmek.

```
def download_extract(name, folder=None):  #@save
    """Bir zip/tar dosyası indir ve aç."""
    fname = download(name)
    base_dir = os.path.dirname(fname)
    data_dir, ext = os.path.splitext(fname)
    if ext == '.zip':
        fp = zipfile.ZipFile(fname, 'r')
    elif ext in ('.tar', '.gz'):
        fp = tarfile.open(fname, 'r')
    else:
        assert False, 'Only zip/tar files can be extracted.'
    fp.extractall(base_dir)
    return os.path.join(base_dir, folder) if folder else data_dir

def download_all():  #@save
    """DATA_HUB içindeki tüm dosyaları indir."""
    for name in DATA_HUB:
        download(name)
```

4.10.2 Kaggle

Kaggle[75], makine öğrenmesi yarışmalarına ev sahipliği yapan popüler bir platformdur. Her yarışma bir veri kümesine odaklanır ve çoğu, kazanan çözümlere ödüller sunan paydaşlar tarafından desteklenir. Platform, kullanıcıların forumlar ve paylaşılan kodlar aracılığıyla etkileşime girmesine yardımcı olarak hem işbirliğini hem de rekabeti teşvik eder. Liderlik tahtası takibi çoğu zaman kontrolden çıkarken, araştırmacılar temel sorular sormaktan ziyade ön işleme adımlarına yakın olarak odaklanırken, bir platformun nesnelliğinde rakip yaklaşımlar arasında doğrudan nicel karşılaştırmaları ve böylece neyin işe yarayıp yaramadığını herkes öğrenebileceği kod paylaşımını kolaylaştıran muazzam bir değer vardır. Bir Kaggle yarışmasına katılmak istiyorsanız, önce bir hesap açmanız gerekir (bakınız Fig. 4.10.1).

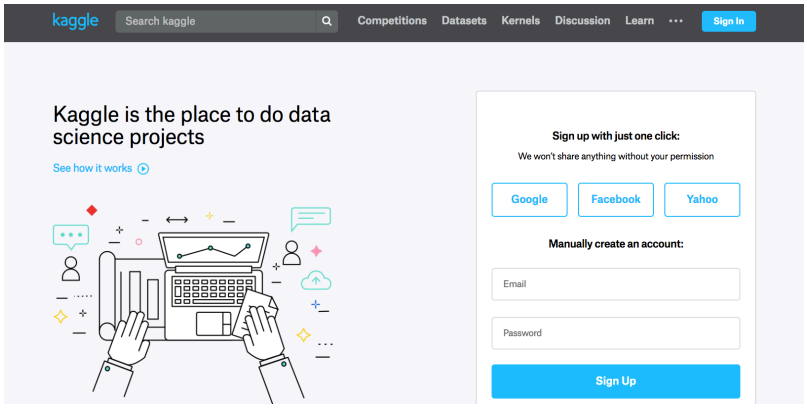


Fig. 4.10.1: Kaggle web sitesi

Ev fiyatları tahminleme yarışması sayfasında, Fig. 4.10.2 içinde gösterildiği gibi, veri kümesini bulabilir ("Data" sekmesinin altında), tahminleri gönderebilir ve sıralamanıza bakabilirsiniz. URL tam buradadır:

https://www.kaggle.com/c/house-prices-advanced-regression-techniques

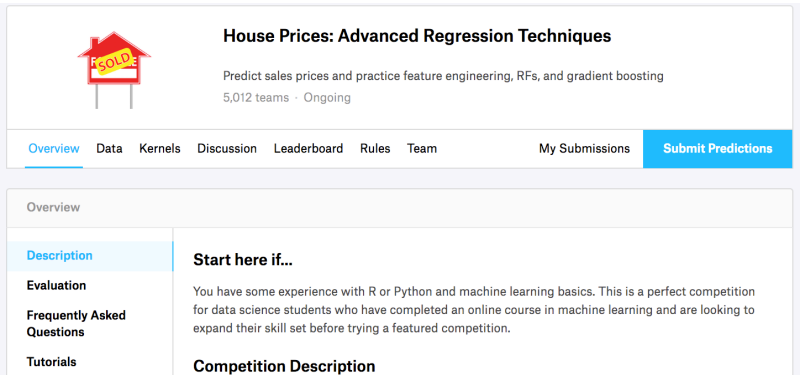


Fig. 4.10.2: Ev fiyatları tahminleme yarışması sayfası

[75] https://www.kaggle.com

4.10.3 Veri Kümesine Erişim ve Okuma

Yarışma verilerinin eğitim ve test kümelerine ayrıldığını unutmayın. Her kayıt, evin mülk değerini ve sokak tipi, yapım yılı, çatı tipi, bodrum durumu vb. öznitelikleri içerir. Öznitelikler çeşitli veri türlerinden oluşur. Örneğin, yapım yılı bir tamsayı ile, çatı tipi ayrı kategorik atamalarla ve diğer öznitelikler kayan virgüllü sayılarla temsil edilir. Böylece burada gerçeklik işleri zorlaştırmaya başlar: Bazı örnekler için, bazı veriler tamamen eksiktir ve eksik değer sadece "na" olarak işaretlenmiştir. Her evin fiyatı sadece eğitim kümesi için dahildir (sonuçta bu bir yarışmadır). Bir geçerleme kümesi oluşturmak için eğitim kümesini bölümlere ayırmak isteyeceğiz, ancak modellerimizi yalnızca tahminleri Kaggle'a yükledikten sonra resmi test kümesinde değerlendirebiliriz. Fig. 4.10.2 içindeki yarışma sekmesindeki "Data" sekmesi, verileri indirmek için bağlantılar içerir.

Başlamak için, verileri Section 2.2 içinde tanıttığımız pandas'ı kullanarak okuyup işleyeceğiz. Bu nedenle devam etmeden önce pandas'ı kurduğunuza emin olmak isteyeceksiniz. Neyse ki, Jupyter'de okuyorsanız, pandas'ı not defterinden bile çıkmadan kurabiliriz.

```
# Pandas kurulu değilse, lütfen aşağıdaki satırın yorumunu kaldırın:
# !pip install pandas

%matplotlib inline
import numpy as np
import pandas as pd
import torch
from torch import nn
from d2l import torch as d2l
```

Kolaylık olması için, yukarıda tanımladığımız komut dosyasını kullanarak Kaggle konut veri kümesini indirebilir ve önbelleğe alabiliriz.

```
DATA_HUB['kaggle_house_train'] = (  #@save
    DATA_URL + 'kaggle_house_pred_train.csv',
    '585e9cc93e70b39160e7921475f9bcd7d31219ce')

DATA_HUB['kaggle_house_test'] = (  #@save
    DATA_URL + 'kaggle_house_pred_test.csv',
    'fa19780a7b011d9b009e8bff8e99922a8ee2eb90')
```

Sırasıyla eğitim ve test verilerini içeren iki csv dosyasını yüklemek için pandas'ı kullanıyoruz.

```
train_data = pd.read_csv(download('kaggle_house_train'))
test_data = pd.read_csv(download('kaggle_house_test'))
```

Eğitim veri kümesi 1460 tane örnek, 80 tane öznitelik ve 1 tane etiket içerirken, test verileri 1459 tane örnek ve 80 tane öznitelik içerir.

```
print(train_data.shape)
print(test_data.shape)
```

```
(1460, 81)
(1459, 80)
```

İlk dört örnekten etiketin (SalePrice - SatışFiyatı) yanı sıra ilk dört ve son iki özniteliğe bir göz atalım.

```
print(train_data.iloc[0:4, [0, 1, 2, 3, -3, -2, -1]])
```

```
   Id  MSSubClass MSZoning  LotFrontage SaleType SaleCondition  SalePrice
0   1          60       RL         65.0       WD        Normal     208500
1   2          20       RL         80.0       WD        Normal     181500
2   3          60       RL         68.0       WD        Normal     223500
3   4          70       RL         60.0       WD       Abnorml     140000
```

Her örnekte ilk özniteliğin kimlik numarası olduğunu görebiliriz. Bu, modelin her eğitim örneğini tanımlamasına yardımcı olur. Bu uygun olsa da, tahmin amaçlı herhangi bir bilgi taşımaz. Bu nedenle, verileri modele beslemeden önce veri kümesinden kaldırıyoruz.

```
all_features = pd.concat((train_data.iloc[:, 1:-1], test_data.iloc[:, 1:]))
```

4.10.4 Veri Ön İşleme

Yukarıda belirtildiği gibi, çok çeşitli veri türlerine sahibiz. Modellemeye başlamadan önce verileri ön işlememiz gerekecek. Sayısal özniteliklerle başlayalım. İlk olarak, tüm eksik değerleri karşılık gelen özniteliğin ortalaması ile değiştiren bir buluşsal yöntem uygularız. Ardından, tüm öznitelikleri ortak bir ölçeğe koymak için, onları sıfır ortalamaya ve birim varyansa yeniden ölçeklendirerek standardize ederiz:

$$x \leftarrow \frac{x - \mu}{\sigma}, \tag{4.10.1}$$

burada μ ve σ sırasıyla ortalamayı ve standard sapmayı ifade eder. Bunun özniteliği (değişkenimizi) sıfır ortalamaya ve birim varyansına sahip olacak şekilde dönüştürdüğünü doğrulamak için, $E[\frac{x-\mu}{\sigma}] = \frac{\mu-\mu}{\sigma} = 0$ ve $E[(x-\mu)^2] = (\sigma^2+\mu^2)-2\mu^2+\mu^2 = \sigma^2$ olduğuna dikkat edin. Sezgisel olarak, verileri iki nedenden dolayı standartleştiriyoruz. Birincisi, optimizasyon için uygun oluyor. İkinci olarak, hangi özniteliklerin ilgili olacağını *önsel* bilmediğimiz için, bir özniteliğe atanan katsayıları diğerlerinden daha fazla cezalandırmak istemiyoruz.

```
# Test verilerine erişilemiyorsa, eğitim verilerinden
# ortalama ve standart sapma hesaplanabilir
numeric_features = all_features.dtypes[all_features.dtypes != 'object'].index
all_features[numeric_features] = all_features[numeric_features].apply(
    lambda x: (x - x.mean()) / (x.std()))
# # Verileri standartlaştırdıktan sonra her şey yok olur,
# dolayısıyla eksik değerleri 0 olarak ayarlayabiliriz.
all_features[numeric_features] = all_features[numeric_features].fillna(0)
```

Daha sonra ayrık değerlerle ilgileniyoruz. Bu, "MSZoning" gibi değişkenleri içerir. Onları daha önce çok sınıflı etiketleri vektörlere dönüştürdüğümüz gibi (bkz. Section 3.4.1) bire bir kodlama ile değiştiriyoruz. Örneğin, "MSZoning", "RL" ve "RM" değerlerini varsayar. "MSZoning" özniteliğini kaldırılarak, "MSZoning_RL" ve "MSZoning_RM" olmak üzere iki yeni gösterge özniteliğini 0 veya 1 değerleriyle oluşturulur. Bire bir kodlamaya göre, "MSZoning"'in orijinal değeri "RL" ise, "MSZoning_RL" 1'dir ve "MSZoning_RM" 0'dır. pandas paketi bunu bizim için otomatik olarak yapar.

```
# # `Dummy_na=True`, "na"yı (eksik değer) geçerli bir öznitelik değeri olarak
# kabul eder ve bunun için bir gösterge özniteliği oluşturur
all_features = pd.get_dummies(all_features, dummy_na=True)
all_features.shape
```

```
(2919, 331)
```

Bu dönüşümün özelliklerin sayısını 79'dan 331'e çıkardığını görebilirsiniz. Son olarak, `values` özelliğiyle, NumPy formatını pandas veri formatından çıkarabilir ve eğitim için tensör temsiline dönüştürebiliriz.

```
n_train = train_data.shape[0]
train_features = torch.tensor(all_features[:n_train].values, dtype=torch.float32)
test_features = torch.tensor(all_features[n_train:].values, dtype=torch.float32)
train_labels = torch.tensor(
    train_data.SalePrice.values.reshape(-1, 1), dtype=torch.float32)
```

4.10.5 Eğitim

Başlarken, hata karesi kullanan doğrusal bir model eğitiyoruz. Şaşırtıcı olmayan bir şekilde, doğrusal modelimiz rekabeti kazanan bir teslime yol açmayacaktır, ancak verilerde anlamlı bilgi olup olmadığını görmek için bir makulluk kontrolü sağlar. Burada rastgele tahmin etmekten daha iyisini yapamazsak, o zaman bir veri işleme hatasına sahip olma ihtimalimiz yüksek olabilir. Eğer işler yolunda giderse, doğrusal model bize basit modelin en iyi rapor edilen modellere ne kadar yaklaştığı konusunda biraz önsezi vererek daha süslü modellerden ne kadar kazanç beklememiz gerektiğine dair bir fikir verir.

```
loss = nn.MSELoss()
in_features = train_features.shape[1]

def get_net():
    net = nn.Sequential(nn.Linear(in_features,1))
    return net
```

Konut fiyatlarında, hisse senedi fiyatlarında olduğu gibi, göreceli miktarları mutlak miktarlardan daha fazla önemsiyoruz. Bu nedenle, $\frac{y-\hat{y}}{y}$ göreceli hatasını $y - \hat{y}$ mutlak hatasından daha çok önemsiyoruz. Örneğin, tipik bir evin değerinin 125000 Amerikan Doları olduğu Rural Ohio'daki bir evin fiyatını tahmin ederken tahminimiz 100000 USD yanlışsa, muhtemelen korkunç bir iş yapıyoruz demektir. Öte yandan, Los Altos Hills, California'da bu miktarda hata yaparsak, bu şaşırtıcı derecede doğru bir tahmini temsil edebilir (burada, ortalama ev fiyatı 4 milyon Amerikan Dolarını aşar).

Bu sorunu çözmenin bir yolu, fiyat tahminlerinin logaritmasındaki tutarsızlığı ölçmektir. Aslında, bu aynı zamanda yarışma tarafından gönderimlerin kalitesini değerlendirmek için kullanılan resmi hata ölçüsüdür. Sonuçta, küçük bir δ için $|\log y - \log \hat{y}| \leq \delta$ değerini $e^{-\delta} \leq \frac{\hat{y}}{y} \leq e^{\delta}$'e çevirir. Bu da tahmini fiyat logaritması ile etiket fiyat logaritması arasında aşağıdaki ortalama kare hata kare kökü kaybına yol açar:

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(\log y_i - \log \hat{y}_i)^2}. \tag{4.10.2}$$

```
def log_rmse(net, features, labels):
    # Logaritma alındığında değeri daha da sabitlemek için
    # 1'den küçük değeri 1 olarak ayarlayın
    clipped_preds = torch.clamp(net(features), 1, float('inf'))
    rmse = torch.sqrt(loss(torch.log(clipped_preds),
                           torch.log(labels)))
    return rmse.item()
```

Önceki bölümlerden farklı olarak, eğitim işlevlerimiz Adam optimize edicisine dayanacaktır (daha sonra daha ayrıntılı olarak açıklayacağız). Bu eniyileyicinin ana cazibesi hiper parametre optimizasyonu için sınırsız kaynaklar verildiğinde daha iyisini yapmamasına (ve bazen daha kötüsünü yapmasına) rağmen, insanların başlangıç öğrenme oranına önemli ölçüde daha az duyarlı olduğunu bulma eğiliminde olmasıdır.

```
def train(net, train_features, train_labels, test_features, test_labels,
          num_epochs, learning_rate, weight_decay, batch_size):
    train_ls, test_ls = [], []
    train_iter = d2l.load_array((train_features, train_labels), batch_size)
    # Burada Adam optimizasyon algoritması kullanılıyor
    optimizer = torch.optim.Adam(net.parameters(),
                                 lr = learning_rate,
                                 weight_decay = weight_decay)
    for epoch in range(num_epochs):
        for X, y in train_iter:
            optimizer.zero_grad()
            l = loss(net(X), y)
            l.backward()
            optimizer.step()
        train_ls.append(log_rmse(net, train_features, train_labels))
        if test_labels is not None:
            test_ls.append(log_rmse(net, test_features, test_labels))
    return train_ls, test_ls
```

4.10.6 K-Kat Çapraz-Geçerleme

Model seçimiyle nasıl başa çıkılacağını tartıştığımız bölümde K-kat çapraz geçerlemeyi tanıttığımızı hatırlayabilirsiniz (Section 4.4). Bunu, model tasarımını seçmek ve hiper parametreleri ayarlamak için iyi bir şekilde kullanacağız. Öncelikle, K-kat çapraz geçerleme prosedüründe verilerin i. katını döndüren bir işleve ihtiyacımız var. i. parçayı geçerleme verisi olarak dilimleyerek ve geri kalanını eğitim verisi olarak döndürerek ilerler. Bunun veri işlemenin en verimli yolu olmadığını ve veri kümemiz çok daha büyük olsaydı kesinlikle çok daha akıllıca bir şey yapacağımızı unutmayın. Ancak bu ek karmaşıklık, kodumuzu gereksiz yere allak bullak edebilir, bu nedenle sorunumuzun basitliğinden dolayı burada güvenle atlayabiliriz.

```
def get_k_fold_data(k, i, X, y):
    assert k > 1
    fold_size = X.shape[0] // k
    X_train, y_train = None, None
    for j in range(k):
        idx = slice(j * fold_size, (j + 1) * fold_size)
        X_part, y_part = X[idx, :], y[idx]
```

(continues on next page)

(continued from previous page)

```
        if j == i:
            X_valid, y_valid = X_part, y_part
        elif X_train is None:
            X_train, y_train = X_part, y_part
        else:
            X_train = torch.cat([X_train, X_part], 0)
            y_train = torch.cat([y_train, y_part], 0)
    return X_train, y_train, X_valid, y_valid
```

Eğitim ve geçerleme hatası ortalamaları, K-kat çapraz geçerleme ile K kez eğitimimizde döndürülür.

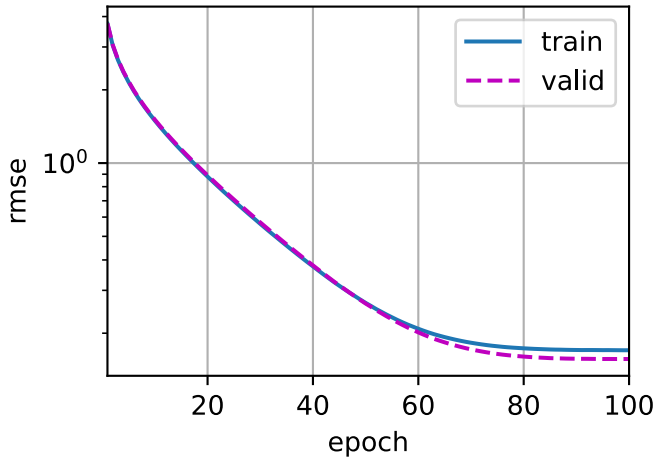
```
def k_fold(k, X_train, y_train, num_epochs, learning_rate, weight_decay,
           batch_size):
    train_l_sum, valid_l_sum = 0, 0
    for i in range(k):
        data = get_k_fold_data(k, i, X_train, y_train)
        net = get_net()
        train_ls, valid_ls = train(net, *data, num_epochs, learning_rate,
                                   weight_decay, batch_size)
        train_l_sum += train_ls[-1]
        valid_l_sum += valid_ls[-1]
        if i == 0:
            d2l.plot(list(range(1, num_epochs + 1)), [train_ls, valid_ls],
                     xlabel='epoch', ylabel='rmse', xlim=[1, num_epochs],
                     legend=['train', 'valid'], yscale='log')
        print(f'fold {i + 1}, train log rmse {float(train_ls[-1]):f}, '
              f'valid log rmse {float(valid_ls[-1]):f}')
    return train_l_sum / k, valid_l_sum / k
```

4.10.7 Model Seçimi

Bu örnekte, ayarlanmamış bir hiper parametre kümesi seçiyoruz ve modeli geliştirmek için okuyucuya bırakıyoruz. İyi bir seçim bulmak, kişinin kaç değişkeni optimize ettiğine bağlı olarak zaman alabilir. Yeterince büyük bir veri kümesi ve normal hiper parametreler ile K-kat çapraz geçerleme, çoklu testlere karşı makul ölçüde dirençli olma eğilimindedir. Bununla birlikte, mantıksız bir şekilde çok sayıda seçeneği denersek, sadece şanslı olabiliriz ve geçerleme performansımızın artık gerçek hatayı temsil etmediğini görebiliriz.

```
k, num_epochs, lr, weight_decay, batch_size = 5, 100, 5, 0, 64
train_l, valid_l = k_fold(k, train_features, train_labels, num_epochs, lr,
                          weight_decay, batch_size)
print(f'{k}-fold validation: avg train log rmse: {float(train_l):f}, '
      f'avg valid log rmse: {float(valid_l):f}')
```

```
fold 1, train log rmse 0.170226, valid log rmse 0.156672
fold 2, train log rmse 0.162236, valid log rmse 0.190424
fold 3, train log rmse 0.163491, valid log rmse 0.168458
fold 4, train log rmse 0.168297, valid log rmse 0.154856
fold 5, train log rmse 0.163502, valid log rmse 0.182911
5-fold validation: avg train log rmse: 0.165550, avg valid log rmse: 0.170664
```



K-kat çapraz geçerlemedeki hata sayısı önemli ölçüde daha yüksek olsa bile, bir dizi hiper parametre için eğitim hatası sayısının bazen çok düşük olabileceğine dikkat edin. Bu bizim aşırı öğrendiğimizi gösterir. Eğitim boyunca her iki sayıyı da izlemek isteyeceksiniz. Daha az aşırı öğrenme, verilerimizin daha güçlü bir modeli destekleyebileceğini gösteremez. Aşırı öğrenme, düzenlileştirme tekniklerini dahil ederek kazanç sağlayabileceğimizi gösterebilir.

4.10.8 Tahminleri Kaggle'da Teslim Etme

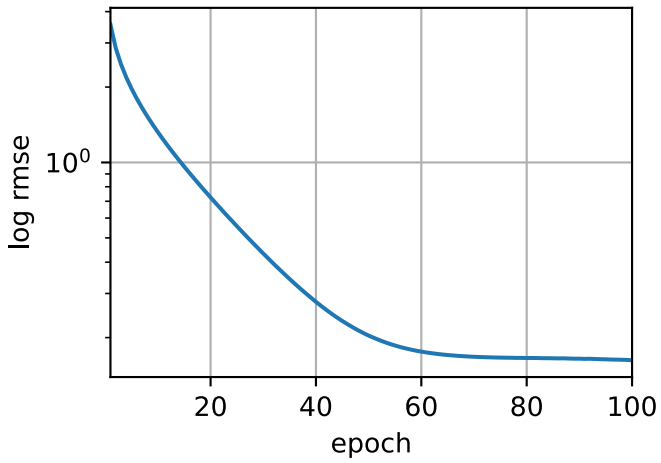
Artık iyi bir hiper parametre seçiminin ne olması gerektiğini bildiğimize göre, onu eğitmek için tüm verileri de kullanabiliriz (çapraz geçerleme dilimlerinde kullanılan verinin yalnızca $1 - 1/K$'ı yerine). Bu şekilde elde ettiğimiz model daha sonra test kümesine uygulanabilir. Tahminlerin bir csv dosyasına kaydedilmesi, sonuçların Kaggle'a yüklenmesini kolaylaştıracaktır.

```
def train_and_pred(train_features, test_features, train_labels, test_data,
                   num_epochs, lr, weight_decay, batch_size):
    net = get_net()
    train_ls, _ = train(net, train_features, train_labels, None, None,
                        num_epochs, lr, weight_decay, batch_size)
    d2l.plot(np.arange(1, num_epochs + 1), [train_ls], xlabel='epoch',
             ylabel='log rmse', xlim=[1, num_epochs], yscale='log')
    print(f'train log rmse {float(train_ls[-1]):f}')
    # Ağı test kümesine uygula
    preds = net(test_features).detach().numpy()
    # Kaggle'a dış aktarmak için yeniden biçimlendirin
    test_data['SalePrice'] = pd.Series(preds.reshape(1, -1)[0])
    submission = pd.concat([test_data['Id'], test_data['SalePrice']], axis=1)
    submission.to_csv('submission.csv', index=False)
```

Güzel bir makulluk kontrolü, test kümesindeki tahminlerin K-kat çapraz geçerleme sürecindekilere benzeyip benzemediğini görmektir. Yaparsa, Kaggle'a yükleme zamanı gelmiştir. Aşağıdaki kod, `submission.csv` adlı bir dosya oluşturacaktır.

```
train_and_pred(train_features, test_features, train_labels, test_data,
               num_epochs, lr, weight_decay, batch_size)
```

```
train log rmse 0.162397
```



Sonra, Fig. 4.10.3 içinde gösterildiği gibi, tahminlerimizi Kaggle'a gönderebilir ve test kümesindeki gerçek ev fiyatları (etiketler) ile nasıl karşılaştırıldıklarını görebiliriz. Adımlar oldukça basittir:

- Kaggle web sitesinde oturum açın ve ev fiyatı tahmin yarışması sayfasını ziyaret edin.
- "Tahminleri Teslim Et" ("Submit Predictions") veya "Geç Teslimat" ("Late Submission") düğmesine tıklayın (bu yazı esnasında düğme sağda yer almaktadır).
- Sayfanın alt kısmındaki kesik çizgili kutudaki "Teslimat Dosyasını Yükle" ("Upload Submission File") düğmesini tıklayın ve yüklemek istediğiniz tahmin dosyasını seçin.
- Sonuçlarınızı görmek için sayfanın altındaki "Teslim Et" ("Make Submission") düğmesine tıklayın.

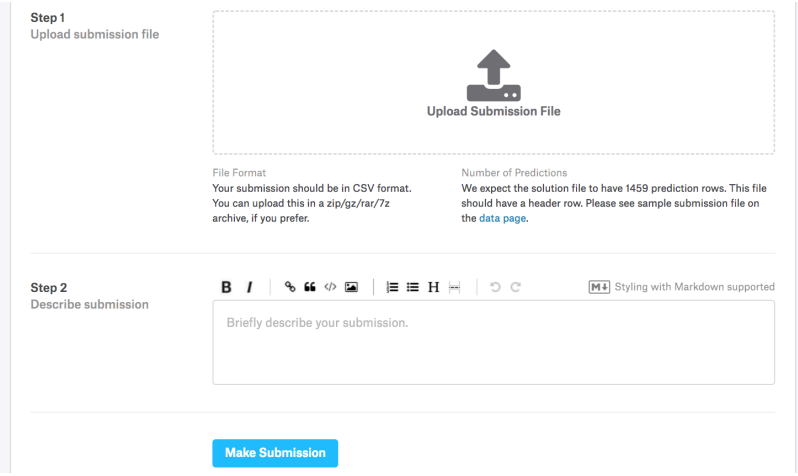


Fig. 4.10.3: Kaggle'a veriyi gönderme

4.10.9 Özet

- Gerçek veriler genellikle farklı veri türlerinin bir karışımını içerir ve önceden işlenmesi gerekir.
- Gerçek değerli verileri sıfır ortalamaya ve birim varyansına yeniden ölçeklemek iyi bir varsayılandır. Eksik değerleri ortalamalarıyla değiştirmek de öyle.
- Kategorik öznitelikleri gösterge özniteliklere dönüştürmek, onları bire bir vektörler gibi ele almamızı sağlar.
- Modeli seçmek ve hiper parametreleri ayarlamak için K-kat çapraz geçerleme kullanabiliriz.
- Logaritmalar göreceli hatalar için faydalıdır.

4.10.10 Alıştırmalar

1. Bu bölümdeki tahminlerinizi Kaggle'a gönderin. Tahminleriniz ne kadar iyi?
2. Fiyatın logaritmasını doğrudan en aza indirerek modelinizi geliştirebilir misiniz? Fiyat yerine fiyatın logaritmasını tahmin etmeye çalışırsanız ne olur?
3. Eksik değerleri ortalamalarıyla değiştirmek her zaman iyi bir fikir midir? İpucu: Değerlerin rastgele eksik olmadığı bir durum oluşturabilir misiniz?
4. K-kat çapraz geçerleme yoluyla hiper parametreleri ayarlayarak Kaggle'daki puanı iyileştiriniz.
5. Modeli geliştirerek puanı iyileştirin (örn. katmanlar, ağırlık sönümü ve hattan düşürme).
6. Bu bölümde yaptığımız gibi sayısal sürekli öznitelikleri standartlaştırmazsak ne olur?

Tartışmalar[76]

[76] https://discuss.d2l.ai/t/107

5 | Derin Öğrenme Hesaplamaları

Devasa veri kümeleri ve güçlü donanımın yanı sıra, harika yazılım araçları, derin öğrenmenin hızlı ilerlemesinde yadsınamaz bir rol oynamıştır. 2007'de yayınlanan çığır açan Theano kütüphanesinden başlayarak, esnek açık kaynaklı araçlar, araştırmacıların modelleri hızlı bir şekilde prototip haline getirmelerine, standart bileşenleri geri dönüştürürken tekrarlanan çalışmalardan kaçınmalarına ve aynı zamanda alt seviyede değişiklikler yapma yeteneğini sürdürmelerine olanak tanıdı. Zamanla, derin öğrenme kütüphaneleri giderek daha kaba soyutlamalar sunmak için evrimleşti. Yarı iletken tasarımcıların transistörlerin belirlenmesinden mantıksal devrelere kod yazmaya geçmesi gibi, sinir ağları araştırmacıları da tek tek yapay nöronların davranışları hakkında düşünmekten ağları tüm katmanlar açısından kavramaya geçtiler ve şimdi genellikle zihindeki çok daha kaba *bloklara* sahip mimariler tasarladılar.

Şimdiye kadar, bazı temel makine öğrenmesi kavramlarını tanıtmaktan tamamen işlevsel derin öğrenme modellerine yükseliyoruz. Son bölümde, bir MLP'nin her bileşenini sıfırdan uyguladık ve hatta aynı modelleri zahmetsizce kullanıma sunmak için yüksek düzey API'lerden nasıl yararlanılacağını gösterdik. Sizi bu kadar mesafeye bu kadar hızlı ulaştırmak için, kütüphaneleri *çağırdık*, ancak *nasıl çalıştıkları* ile ilgili daha gelişmiş ayrıntıları atladık. Bu bölümde, derin öğrenme hesaplamasının temel bileşenlerine daha derinlemesine, yani model oluşturma, parametre erişimi ve ilkleme, ısmarlama (özel kesim) katmanlar ve bloklar tasarlama, modelleri diskten okuma, diske yazma ve çarpıcı bir hızlandırma elde etmek için GPU'lardan yararlanmaya bakacağız. Bu içgörüler, sizi *son kullanıcıdan güçlü kullanıcıya* taşıyarak, olgun bir derin öğrenme kütüphanesinin faydalarından yararlanmak için gereken araçları sağlarken, kendi icat ettikleriniz de dahil olmak üzere daha karmaşık modelleri uygulama esnekliğini korur! Bu bölüm herhangi bir yeni model veya veri kümesi tanıtmasa da, takip eden gelişmiş modelleme bölümleri büyük ölçüde bu tekniklere dayanmaktadır.

5.1 Katmanlar ve Bloklar

Sinir ağlarını ilk tanıttığımızda, tek çıktılı doğrusal modellere odaklandık. Burada tüm model sadece tek bir nörondan oluşuyor. Tek bir nöronun (i) bazı girdiler aldığını, (ii) karşılık gelen skaler bir çıktı ürettiğini ve (iii) ilgili bazı amaç işlevlerini optimize etmek için güncellenebilen bir dizi ilişkili parametreye sahip olduğunu unutmayın. Sonra, birden çok çıktıya sahip ağları düşünmeye başladığımızda, tüm bir nöron katmanını karakterize etmek için vektörleştirilmiş aritmetikten yararlandık. Tıpkı bireysel nöronlar gibi, katmanlar (i) bir dizi girdi alır, (ii) karşılık gelen çıktıları üretir ve (iii) bir dizi ayarlanabilir parametre ile açıklanır. Softmaks bağlanımı üzerinde çalıştığımızda, tek bir katmanın kendisi modeldi. Bununla birlikte, daha sonra MLP'leri devreye soktuğumuzda bile, modelin bu aynı temel yapıyı koruduğunu düşünebiliriz.

İlginç bir şekilde, MLPler için hem tüm model hem de kurucu katmanları bu yapıyı paylaşır. Tam model ham girdileri (öznitelikleri) alır, çıktılar (tahminler) üretir ve parametrelere (tüm kurucu

katmanlardan birleşik parametreler) sahiptir. Benzer şekilde, her bir katman girdileri alır (önceki katman tarafından sağlanır) çıktılar (sonraki katmana girdiler) üretir ve sonraki katmandan geriye doğru akan sinyale göre güncellenen bir dizi ayarlanabilir parametreye sahiptir.

Nöronların, katmanların ve modellerin bize işimiz için yeterince soyutlama sağladığını düşünseniz de, genellikle tek bir katmandan daha büyük ancak tüm modelden daha küçük olan bileşenler hakkında konuşmayı uygun bulduğumuz ortaya çıkıyor. Örneğin, bilgisayarla görmede aşırı popüler olan ResNet-152 mimarisi, yüzlerce katmana sahiptir. Bu katmanlar, *katman gruplarının* tekrar eden desenlerinden oluşur. Böyle bir ağın her seferinde bir katman olarak uygulanması sıkıcı bir hal alabilir. Bu endişe sadece varsayımsal değildir—bu tür tasarım modelleri pratikte yaygındır. Yukarıda bahsedilen ResNet mimarisi, hem tanıma hem de tespit için 2015 ImageNet ve COCO bilgisayarla görme yarışmalarını kazandı (He *et al.*, 2016) ve birçok görme görevi için bir ilk tatbik edilen mimari olmaya devam ediyor. Katmanların çeşitli yinelenen desenlerde düzenlendiği benzer mimariler artık doğal dil işleme ve konuşma dahil olmak üzere diğer alanlarda da her yerde mevcuttur.

Bu karmaşık ağları uygulamak için, bir sinir ağı *bloğu* kavramını sunuyoruz. Bir blok, tek bir katmanı, birden çok katmandan oluşan bir bileşeni veya tüm modelin kendisini tanımlayabilir! Blok soyutlamayla çalışmanın bir yararı, bunların genellikle yinelemeli olarak daha büyük yapay nesnelerle birleştirilebilmeleridir. Bu, Fig. 5.1.1 içinde gösterilmiştir. İsteğe bağlı olarak rastgele karmaşıklıkta bloklar oluşturmak için kod tanımlayarak, şaşırtıcı derecede sıkıştırılmış kod yazabilir ve yine de karmaşık sinir ağlarını uygulayabiliriz.

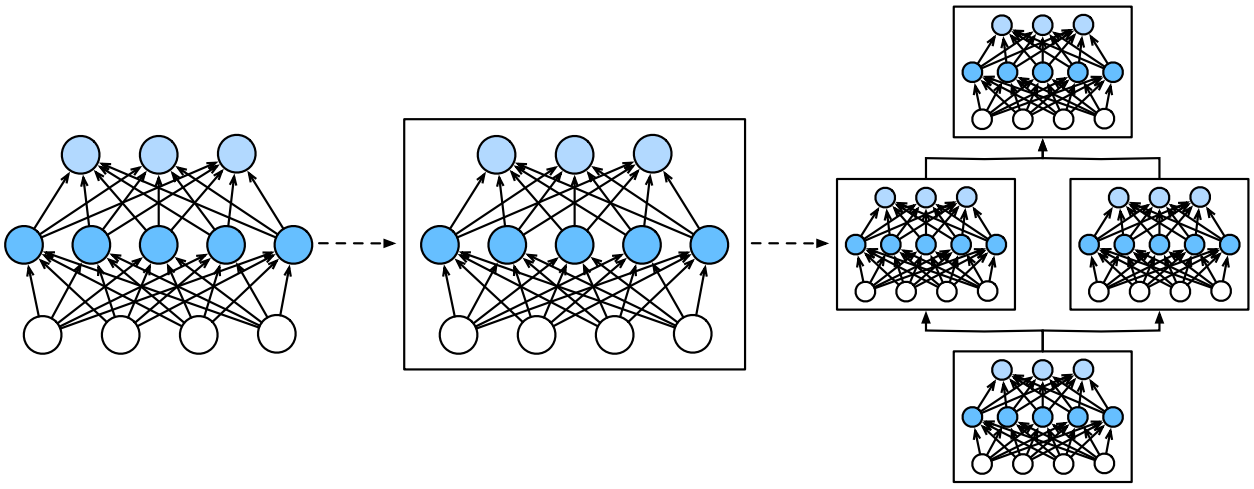


Fig. 5.1.1: Çoklu katmanlar bloklara birleştirilir, daha geniş modelleri oluşturan tekrar eden desenler oluştururlar

Programlama açısından, bir blok *sınıf* ile temsil edilir. Onun herhangi bir alt sınıfı, girdisini çıktıya dönüştüren ve gerekli parametreleri depolayan bir ileri yayma yöntemi tanımlamalıdır. Bazı blokların herhangi bir parametre gerektirmediğini unutmayın. Son olarak, gradyanları hesaplamak için bir blok geriye yayma yöntemine sahip olmalıdır. Neyse ki, kendi bloğumuzu tanımlarken otomatik türev almanın sağladığı bazı perde arkası sihir sayesinde (Section 2.5 içinde tanıtıldı), sadece parametreler ve ileri yayma işlevi hakkında endişelenmemiz gerekir.

Başlangıç olarak MLPleri uygulamak için kullandığımız kodları yeniden gözden geçiriyoruz (Section 4.3). Aşağıdaki kod, 256 birim ve ReLU etkinleştirmesine sahip tam bağlı bir gizli katmana sahip bir ağ oluşturur ve ardından 10 birimle (etkinleştirme işlevi yok) tam bağlı çıktı katmanı gelir.

```
import torch
from torch import nn
from torch.nn import functional as F

net = nn.Sequential(nn.Linear(20, 256), nn.ReLU(), nn.Linear(256, 10))

X = torch.rand(2, 20)
net(X)
```

```
tensor([[-0.2314,  0.0426, -0.1038, -0.0008,  0.1609, -0.0306,  0.0023, -0.0257,
         -0.1857, -0.0899],
        [-0.1886,  0.0906, -0.1492, -0.0039,  0.1726,  0.0048, -0.0374, -0.0620,
         -0.2537, -0.0323]], grad_fn=<AddmmBackward0>)
```

Bu örnekte, modelimizi bir `nn.Sequential` örneğini oluşturarak, katmanları bağımsız değişken olarak iletilmeleri gereken sırayla oluşturduk. Kısaca, `nn.Sequential`, PyTorch'ta bir blok sunan özel bir sınıf `Module` türünü tanımlar. Kurucu `Module`'lerin sıralı bir listesini tutar. Tam bağlı iki katmanın her birinin, kendisi de `Module`'ün bir alt sınıfı olan `Linear` sınıfının bir örneği olduğuna dikkat edin. İleri yayma (`forward`) işlevi de oldukça basittir: Listedeki her bloğu birbirine zincirleyerek her birinin çıktısını bir sonrakine girdi olarak iletir. Şimdiye kadar, çıktıları elde etmek için modellerimizi `net(X)` yapısı aracılığıyla çağırdığımızı unutmayın. Bu aslında `net.__call__(X)` için kısa yoldur.

5.1.1 Özel Yapım Blok

Bir bloğun nasıl çalıştığına dair sezgiyi geliştirmenin belki de en kolay yolu, bloğu kendimiz uygulamaktır. Kendi özel yapım bloğumuzu uygulamadan önce, her bloğun sağlaması gereken temel işlevleri kısaca özetliyoruz:

1. Girdi verilerini ileri yayma yöntemine bağımsız değişkenler olarak alın.
2. İleri yayma işlevi kullanarak bir değer döndürüp bir çıktı oluşturun. Çıktının girdiden farklı bir şekle sahip olabileceğini unutmayın. Örneğin, yukarıdaki modelimizdeki ilk tam bağlı katman 20 boyutlu bir girdi alır, ancak 256 boyutunda bir çıktı verir.
3. Girdiye göre çıktısının gradyanını hesaplayın, ki bu da geriye yayma yöntemiyle erişilebilir. Genellikle bu otomatik olarak gerçekleşir.
4. İleri yaymayı hesaplamayı yürütmek için gerekli olan bu parametreleri saklayın ve bunlara erişim sağlayın.
5. Model parametrelerini gerektiği gibi ilkletin.

Aşağıdaki kod parçacığında, 256 gizli birime sahip bir gizli katman ve 10 boyutlu bir çıktı katmanı ile bir MLP'ye karşılık gelen bir bloğu sıfırdan kodladık. Aşağıdaki `MLP` sınıfının bir bloğu temsil eden sınıftan kalıtsal çoğaltıldığını unutmayın. Yalnızca kendi kurucumuzu (Python'daki '**init**' işlevi) ve ileri yayma işlevini sağlayarak, büyük ölçüde ana sınıfın işlevlerine güveneceğiz.

```
class MLP(nn.Module):
    # Model parametreleriyle bir katman tanımlayın.
    # Burada tam bağlı iki katman tanımlıyoruz.
    def __init__(self):
```

(continues on next page)

(continued from previous page)

```
        # Gerekli ilklemeyi gerçekleştirmek için `MLP` üst sınıfının `Module`
        # kurucusunu çağırın. Bu şekilde, sınıf örneği yaratma sırasında model parametreleri,
        # `params` (daha sonra açıklanacak) gibi diğer fonksiyon argümanları da␣
↪belirtilebilir.
        super().__init__()
        self.hidden = nn.Linear(20, 256)  # Gizli katman
        self.out = nn.Linear(256, 10)  # Çıktı katmanı

    # Modelin ileri yaymasını tanımla, yani `X` girdisine dayalı
    # olarak gerekli model çıktısının nasıl döndürüleceğini tanımla.
    def forward(self, X):
        # Burada, nn.function modülünde tanımlanan ReLU'nun fonksiyonel
        # versiyonunu kullandığımıza dikkat edin.
        return self.out(F.relu(self.hidden(X)))
```

İlk olarak ileri yayma işlevine odaklanalım. Girdi olarak `X`'i aldığını, gizli gösterimi etkinleştirme işlevi uygulanmış olarak hesapladığını ve logitlerini çıktı verdiğini unutmayın. Bu `MLP` uygulamasında, her iki katman da örnek değişkenlerdir. Bunun neden makul olduğunu anlamak için, iki MLP'yi (`net1` ve `net2`) somutlaştırdığınızı ve bunları farklı veriler üzerinde eğittiğinizi hayal edin. Doğal olarak, iki farklı öğrenilmiş modeli temsil etmelerini bekleriz.

MLP'nin katmanlarını kurucuda ilkliyoruz ve daha sonra bu katmanları her bir ileri yayma işlevi çağrısında çağırıyoruz. Birkaç önemli ayrıntıya dikkat edin. İlk olarak, özelleştirilmiş `__init__` işlevimiz, `super().__init__()` aracılığıyla üst sınıfın `__init__` işlevini çağırır ve bizi çoğu bloğa uygulanabilen standart şablon kodunu yeniden biçimlendirme zahmetinden kurtarır. Daha sonra, tam bağlı iki katmanımızı `self.hidden` ve `self.out`'a atayarak somutlaştırıyoruz. Yeni bir operatör uygulamadığımız sürece, geriye yayma işlevi veya parametre ilkleme konusunda endişelenmemize gerek olmadığını unutmayın. Sistem bu işlevleri otomatik olarak üretecektir. Bunu deneyelim.

```
net = MLP()
net(X)
```

```
tensor([[ 0.0986,  0.0048,  0.0926,  0.0814, -0.0960, -0.0417, -0.0494, -0.0037,
          0.1997, -0.1000],
        [ 0.0230, -0.0899,  0.0527,  0.1209, -0.0111, -0.1575, -0.0703, -0.0153,
          0.0516, -0.1078]], grad_fn=<AddmmBackward0>)
```

Blok soyutlamanın önemli bir özelliği çok yönlülüğüdür. Katmanlar (tam bağlı katman sınıfı gibi), tüm modeller (yukarıdaki `MLP` sınıfı gibi) veya ara karmaşıklığın çeşitli bileşenlerini oluşturmak için bir bloğu alt sınıflara ayırabiliriz. Bu çok yönlülüğü sonraki bölümlerde, mesela evrişimli sinir ağlarını, ele alırken kullanıyoruz.

5.1.2 Dizili Blok

Artık `Sequential` (dizili) sınıfının nasıl çalıştığına daha yakından bakabiliriz. `Sequential`'nin diğer blokları birbirine zincirleme bağlamak için tasarlandığını hatırlayın. Kendi basitleştirilmiş `MySequential`'ımızı oluşturmak için, sadece iki anahtar işlev tanımlamamız gerekir: 1. Blokları birer birer listeye eklemek için bir işlev. 2. Bir girdiyi blok zincirinden, eklendikleri sırayla iletmek için bir ileri yayma işlevi.

Aşağıdaki `MySequential` sınıfı aynı işlevselliği varsayılan `Sequential` sınıfıyla sunar:

```
class MySequential(nn.Module):
    def __init__(self, *args):
        super().__init__()
        for idx, module in enumerate(args):
            # Burada, `module` , `mMdule` alt sınıfının bir örneğidir.
            # Bunu, `Module` sınıfının `_modules` üye değişkenine kaydederiz
            # ve türü OrderedDict'tir.
            self._modules[str(idx)] = module

    def forward(self, X):
        # OrderedDict, üyelerin eklendikleri sırayla işletileceğini garanti eder.
        for block in self._modules.values():
            X = block(X)
        return X
```

`__init__` yönteminde, her modülü sıralı `_modules` sözlüğüne tek tek ekliyoruz. Neden her `Module`'ün bir `_modules` özelliğine sahip olduğunu ve sadece bir Python listesi tanımlamak yerine bunu neden kullandığımızı merak edebilirsiniz. Kısacası, `_modules`'ün başlıca avantajı, modülün parametre ilklemesi sırasında, sistemin, parametreleri de ilklemesi gereken alt modülleri bulmak için `_modules` sözlüğüne bakmayı bilmesidir.

`MySequential`'ımızın ileri yayma işlevi çağrıldığında, eklenen her blok eklendikleri sırayla yürütülür. Artık `MySequential` sınıfımızı kullanarak bir MLP'yi yeniden uygulayabiliriz.

```
net = MySequential(nn.Linear(20, 256), nn.ReLU(), nn.Linear(256, 10))
net(X)
```

```
tensor([[ 0.1398,  0.1564, -0.0464, -0.0877,  0.2311,  0.1336, -0.0970, -0.0086,
          0.0315, -0.0298],
        [ 0.1132,  0.0785, -0.1431,  0.0261,  0.2176, -0.0221, -0.1219,  0.0359,
          0.0496, -0.0944]], grad_fn=<AddmmBackward0>)
```

Bu `MySequential` kullanımının, daha önce `Sequential` sınıfı için yazdığımız kodla aynı olduğuna dikkat edin (Section 4.3 içinde açıklandığı gibi).

5.1.3 İleri Yayma İşlevinde Kodu Yürütme

`Sequential` sınıfı, model yapımını kolaylaştırarak, kendi sınıfımızı tanımlamak zorunda kalmadan yeni mimarileri bir araya getirmemize olanak tanır. Bununla birlikte, tüm mimariler basit papatya zinciri değildir. Daha fazla esneklik gerektiğinde, kendi bloklarımızı tanımlamak isteyeceğiz. Örneğin, Python'un kontrol akışını ileri yayma işlevi ile yürütmek isteyebiliriz. Dahası, önceden tanımlanmış sinir ağı katmanlarına güvenmek yerine, keyfi matematiksel işlemler gerçekleştirmek isteyebiliriz.

Şimdiye kadar ağlarımızdaki tüm işlemlerin ağımızın etkinleştirmelerine ve parametrelerine göre hareket ettiğini fark etmiş olabilirsiniz. Ancak bazen, ne önceki katmanların sonucu ne de güncellenebilir parametrelerin sonucu olmayan terimleri dahil etmek isteyebiliriz. Bunlara *sabit parametreler* diyoruz. Örneğin, $f(\mathbf{x}, \mathbf{w}) = c \cdot \mathbf{w}^\top \mathbf{x}$ işlevini hesaplayan bir katman istediğimizi varsayalım, burada $\mathbf{x}$ girdi, $\mathbf{w}$ bizim parametremiz ve c optimizasyon sırasında güncellenmeyen belirli bir sabittir. Bu yüzden `FixedHiddenMLP` sınıfını aşağıdaki gibi uyguluyoruz.

```
class FixedHiddenMLP(nn.Module):
    def __init__(self):
        super().__init__()
        # Gradyanları hesaplamayan ve bu nedenle eğitim sırasında sabit kalan
        # rastgele ağırlık parametreleri
        self.rand_weight = torch.rand((20, 20), requires_grad=False)
        self.linear = nn.Linear(20, 20)

    def forward(self, X):
        X = self.linear(X)
        # Oluşturulan sabit parametrelerin yanı sıra `relu` ve `mm`işlevlerini kullanın
        X = F.relu(torch.mm(X, self.rand_weight) + 1)
        # Tam bağlı katmanı yeniden kullanın. Bu, parametreleri tamamen bağlı iki
        # katmanla paylaşmaya eşdeğerdir.
        X = self.linear(X)
        # Kontrol akışı
        while X.abs().sum() > 1:
            X /= 2
        return X.sum()
```

Bu `FixedHiddenMLP` modelinde, ağırlıkları (`self.rand_weight`) örneklemede rastgele ilkletilen ve daha sonra sabit olan gizli bir katman uygularız. Bu ağırlık bir model parametresi değildir ve bu nedenle asla geri yayma ile güncellenmez. Ağ daha sonra bu "sabit" katmanın çıktısını tam bağlı bir katmandan geçirir.

Çıktıyı döndürmeden önce, modelimizin olağandışı bir şey yaptığını unutmayın. Bir while-döngüsü çalıştırdık, L_1 normunun 1'den büyük olması koşulunu test ettik ve çıktı vektörümüzü bu koşulu karşılayana kadar 2'ye böldük. Son olarak, `X`'deki girdilerin toplamını döndürdük. Bildiğimiz kadarıyla hiçbir standart sinir ağı bu işlemi gerçekleştirmez. Bu özel işlemin herhangi bir gerçek dünya sorununda yararlı olmayabileceğini unutmayın. Amacımız, yalnızca rastgele kodu sinir ağı hesaplamalarınızın akışına nasıl tümleştirebileceğinizi göstermektir.

```
net = FixedHiddenMLP()
net(X)
```

```
tensor(0.0956, grad_fn=<SumBackward0>)
```

Blokları bir araya getirmenin çeşitli yollarını karıştırıp eşleştirebiliriz. Aşağıdaki örnekte, blokları birtakım yaratıcı yollarla iç içe yerleştiriyoruz.

```
class NestMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(nn.Linear(20, 64), nn.ReLU(),
                                 nn.Linear(64, 32), nn.ReLU())
        self.linear = nn.Linear(32, 16)

    def forward(self, X):
        return self.linear(self.net(X))

chimera = nn.Sequential(NestMLP(), nn.Linear(16, 20), FixedHiddenMLP())
chimera(X)
```

```
tensor(0.1101, grad_fn=<SumBackward0>)
```

5.1.4 Verimlilik

Hevesli okuyucular, bu işlemlerin bazılarının verimliliği konusunda endişe duymaya başlayabilir. Ne de olsa, yüksek performanslı bir derin öğrenme kitaplığı olması gereken yerde çok sayıda sözlük araması, kod koşturma ve birçok başka Pythonik şey var. Python'un genel yorumlayıcı kilidi[77] sorunları iyi bilinmektedir. Derin öğrenme bağlamında, son derece hızlı GPU'larımızın, başka bir işi çalıştırmadan önce cılız bir CPU'nun Python kodunu çalıştırmasını beklemesi gerekebileceğinden endişe duyabiliriz.

5.1.5 Özet

- Katmanlar bloklardır.
- Birçok katman bir blok içerebilir.
- Bir blok birçok blok içerebilir.
- Bir blok kod içerebilir.
- Bloklar, parametre ilkleme ve geri yayma dahil olmak üzere birçok temel yürütme işlerini halleder.
- Katmanların ve blokların dizili birleştirmeleri `Sequential` blok tarafından gerçekleştirilir.

[77] https://wiki.python.org/moin/GlobalInterpreterLock

5.1.6 Alıştırmalar

1. Blokları bir Python listesinde saklamak için `MySequential`'ı değiştirirseniz ne tür sorunlar ortaya çıkacaktır?
2. Bağımsız değişken olarak iki blok alan bir blok uygulayın, örneğin `net1` ve `net2` ve ileri yaymada her iki ağın birleştirilmiş çıktısını döndürsün. Buna paralel blok da denir.
3. Aynı ağın birden çok örneğini birleştirmek istediğinizi varsayın. Aynı bloğun birden çok örneğini oluşturan ve ondan daha büyük bir ağ oluşturan bir fabrika işlevi uygulayın.

Tartışmalar[78]

5.2 Parametre Yönetimi

Bir mimari seçip hiper parametrelerimizi belirledikten sonra, hedefimiz yitim işlevimizi en aza indiren parametre değerlerini bulmak olduğu eğitim döngüsüne geçiyoruz. Eğitimden sonra, gelecekteki tahminlerde bulunmak için bu parametrelere ihtiyacımız olacak. Ek olarak, bazen parametreleri başka bir bağlamda yeniden kullanmak, modelimizi başka bir yazılımda yürütülebilmesi için diske kaydetmek veya bilimsel anlayış kazanma umuduyla incelemek için ayıklamak isteyeceğiz.

Çoğu zaman, ağır işlerin üstesinden gelmek için derin öğrenme çerçevelerine dayanarak, parametrelerin nasıl beyan edildiğine ve değiştirildiğine dair işin esas ayrıntıları görmezden gelebileceğiz. Bununla birlikte, standart katmanlara sahip yığılmış mimarilerden uzaklaştığımızda, bazen parametreleri bildirme ve onların üstünde oynama yabani otlarına girmemizi gerekecektir. Bu bölümde aşağıdakileri ele alıyoruz:

- Hata ayıklama, teşhis ve görselleştirmeler için parametrelere erişim.
- Parametre ilkletme.
- Parametreleri farklı model bileşenleri arasında paylaşma.

Tek bir gizli katmana sahip bir MLP'ye odaklanarak başlıyoruz.

```
import torch
from torch import nn

net = nn.Sequential(nn.Linear(4, 8), nn.ReLU(), nn.Linear(8, 1))
X = torch.rand(size=(2, 4))
net(X)
```

```
tensor([[-0.4043],
        [-0.3974]], grad_fn=<AddmmBackward0>)
```

[78] https://discuss.d2l.ai/t/55

5.2.1 Parametre Erişimi

Zaten bildiğiniz modellerden parametrelere nasıl erişileceğiyle başlayalım. Bir model `Sequential` sınıfı aracılığıyla tanımlandığında, ilk olarak herhangi bir katmana, bir listeymiş gibi modelde üzerinden indeksleme ile erişebiliriz. Her katmanın parametreleri, özelliklerinde uygun bir şekilde bulunur. Tam bağlı ikinci katmanın parametrelerini aşağıdaki gibi irdeleyebiliriz.

```
print(net[2].state_dict())
```

```
OrderedDict([('weight', tensor([[ 0.0882, -0.0851, -0.0998, -0.0520,  0.1348,  0.3360, -0.
↪1585, -0.1034]])), ('bias', tensor([-0.3121]))])
```

Çıktı bize birkaç önemli şey gösteriyor. İlk olarak, bu tam bağlı katman, sırasıyla o katmanın ağırlıklarına ve ek girdilerine karşılık gelen iki parametre içerir. Her ikisi de tek hassas basamaklı kayan virgüllü sayı (float32) olarak saklanır. Parametrelerin adlarının, yüzlerce katman içeren bir ağda bile, her katmanın parametrelerini benzersiz şekilde tanımlamamıza izin verdiğini unutmayın.

Hedeflenen Parametreler

Her parametrenin, parametre sınıfının bir örneği olarak temsil edildiğine dikkat edin. Parametrelerle yararlı herhangi bir şey yapmak için önce altta yatan sayısal değerlere erişmemiz gerekir. Bunu yapmanın birkaç yolu var. Bazıları daha basitken diğerleri daha geneldir. Aşağıdaki kod, bir parametre sınıfı örneği döndüren ikinci sinir ağı katmanından ek girdiyi dışarı çıkarır ve dahası bu parametrenin değerine erişim sağlar.

```
print(type(net[2].bias))
print(net[2].bias)
print(net[2].bias.data)
```

```
<class 'torch.nn.parameter.Parameter'>
Parameter containing:
tensor([-0.3121], requires_grad=True)
tensor([-0.3121])
```

Parametreler; değer, gradyanlar ve ek bilgiler içeren karmaşık nesnelerdir. Bu nedenle değerleri açıkça talep etmemiz gerekiyor.

Değere ek olarak, her parametre gradyana erişmemize de izin verir. Henüz bu ağ için geri yaymayı çağırmadığımız için, ağ ilk durumundadır.

```
net[2].weight.grad == None
```

```
True
```

Bütün Parametrelere Bir Kerede Erişim

Tüm parametrelerde işlem yapmamız gerektiğinde, bunlara tek tek erişmek bıktırıcı olabilir. Her bir alt bloğun parametrelerini dışarı çıkarmayı tüm ağaç boyunca tekrarlamamız gerekeceğinden, daha karmaşık bloklarla (örneğin, iç içe geçmiş bloklarla) çalıştığımızda durum özellikle kullanışsızca büyüyebilir. Aşağıda, ilk tam bağlı katmanın parametrelerine erişmeye karşılık tüm katmanlara erişmeyi gösteriyoruz.

```
print(*[(name, param.shape) for name, param in net[0].named_parameters()])
print(*[(name, param.shape) for name, param in net.named_parameters()])
```

```
('weight', torch.Size([8, 4])) ('bias', torch.Size([8]))
('0.weight', torch.Size([8, 4])) ('0.bias', torch.Size([8])) ('2.weight', torch.Size([1,␣
↪8])) ('2.bias', torch.Size([1]))
```

Bu bize ağın parametrelerine erişmenin aşağıdaki gibi başka bir yolunu sağlar:

```
net.state_dict()['2.bias'].data
```

```
tensor([-0.3121])
```

İçiçe Bloklardan Parametreleri Toplama

Birden çok bloğu iç içe yerleştirirsek, parametre adlandırma kurallarının nasıl çalıştığını görelim. Bunun için önce blok üreten bir fonksiyon tanımlıyoruz (tabiri caizse bir blok fabrikası) ve sonra bunları daha büyük bloklar içinde birleştiriyoruz.

```
def block1():
    return nn.Sequential(nn.Linear(4, 8), nn.ReLU(),
                         nn.Linear(8, 4), nn.ReLU())

def block2():
    net = nn.Sequential()
    for i in range(4):
        # Burada iç içe konuyor
        net.add_module(f'block {i}', block1())
    return net

rgnet = nn.Sequential(block2(), nn.Linear(4, 1))
rgnet(X)
```

```
tensor([[0.3051],
        [0.3051]], grad_fn=<AddmmBackward0>)
```

Ağı tasarladığımıza göre, şimdi nasıl düzenlendiğini görelim.

```
print(rgnet)
```

```
Sequential(
  (0): Sequential(
    (block 0): Sequential(
      (0): Linear(in_features=4, out_features=8, bias=True)
      (1): ReLU()
      (2): Linear(in_features=8, out_features=4, bias=True)
      (3): ReLU()
    )
    (block 1): Sequential(
      (0): Linear(in_features=4, out_features=8, bias=True)
      (1): ReLU()
      (2): Linear(in_features=8, out_features=4, bias=True)
      (3): ReLU()
    )
    (block 2): Sequential(
      (0): Linear(in_features=4, out_features=8, bias=True)
      (1): ReLU()
      (2): Linear(in_features=8, out_features=4, bias=True)
      (3): ReLU()
    )
    (block 3): Sequential(
      (0): Linear(in_features=4, out_features=8, bias=True)
      (1): ReLU()
      (2): Linear(in_features=8, out_features=4, bias=True)
      (3): ReLU()
    )
  )
  (1): Linear(in_features=4, out_features=1, bias=True)
)
```

Katmanlar hiyerarşik olarak iç içe olduğundan, iç içe geçmiş listeler aracılığıyla dizinlenmiş gibi bunlara da erişebiliriz. Örneğin, birinci ana bloğa, içindeki ikinci alt bloğa ve bunun içindeki ilk katmanın ek girdisine aşağıdaki şekilde erişebiliriz.

```
rgnet[0][1][0].bias.data
```

```
tensor([-0.2715,  0.3849,  0.4326,  0.4178,  0.4517, -0.2524,  0.2472, -0.1414])
```

5.2.2 Parametre İlkleme

Artık parametrelere nasıl erişeceğimizi bildiğimize göre, onları nasıl doğru şekilde ilkleteceğimize bakalım. Uygun ilkleme ihtiyacını Section 4.8 içinde tartıştık. Derin öğrenme çerçevesi katmanlarına varsayılan rastgele ilklemeler sağlar. Bununla birlikte, ağırlıklarımızı öteki farklı protokollere göre ilkletmek istiyoruz. Çerçeve, en sık kullanılan protokolleri sağlar ve ayrıca özelleştirilmiş bir ilkletici oluşturmaya izin verir.

Varsayılan olarak PyTorch, girdi ve çıktı boyutuna göre hesaplanan bir aralıktan çekerek ağırlık ve ek girdi matrislerini tekdüze olarak başlatır. PyTorch'un `nn.init` modülü önceden ayarlı çeşitli ilkleme yöntemleri sağlar.

Yerleşik İlkletme

Yerleşik ilkleticileri çağırarak ilkleyelim. Aşağıdaki kod, tüm ağırlık parametrelerini standart sapması 0.01 olan Gauss rastgele değişkenler olarak ilkletirken ek girdi parametreleri de 0 olarak atanır.

```
def init_normal(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, mean=0, std=0.01)
        nn.init.zeros_(m.bias)
net.apply(init_normal)
net[0].weight.data[0], net[0].bias.data[0]
```

```
(tensor([ 0.0006, -0.0043,  0.0077, -0.0008]), tensor(0.))
```

Ayrıca tüm parametreleri belirli bir sabit değere ilkleyebiliriz (örneğin, 1 gibi).

```
def init_constant(m):
    if type(m) == nn.Linear:
        nn.init.constant_(m.weight, 1)
        nn.init.zeros_(m.bias)
net.apply(init_constant)
net[0].weight.data[0], net[0].bias.data[0]
```

```
(tensor([1., 1., 1., 1.]), tensor(0.))
```

Ayrıca belirli bloklar için farklı ilkleyiciler uygulayabiliriz. Örneğin, aşağıda ilk katmanı Xavier ilkleyicisi ile ilkliyoruz ve ikinci katmanı sabit 42 değeri ile ilkliyoruz.

```
def xavier(m):
    if type(m) == nn.Linear:
        nn.init.xavier_uniform_(m.weight)
def init_42(m):
    if type(m) == nn.Linear:
        nn.init.constant_(m.weight, 42)

net[0].apply(xavier)
net[2].apply(init_42)
print(net[0].weight.data[0])
print(net[2].weight.data)
```

```
tensor([-0.0983,  0.5310, -0.2538,  0.6591])
tensor([[42., 42., 42., 42., 42., 42., 42., 42.]])
```

Özelleştirilmiş İlkleme

Bazen, ihtiyaç duyduğumuz ilkletme yöntemleri derin öğrenme çerçevesi tarafından sağlanmayabilir. Aşağıdaki örnekte, herhangi bir ağırlık parametresi w için aşağıdaki garip dağılımı kullanarak bir ilkleyici tanımlıyoruz:

$$w \sim \begin{cases} U(5, 10) & \text{olasılık değeri } \frac{1}{4} \\ 0 & \text{olasılık değeri } \frac{1}{2} \\ U(-10, -5) & \text{olasılık değeri } \frac{1}{4} \end{cases} \tag{5.2.1}$$

Yine, `net`'e uygulamak için bir `my_init` işlevi uyguluyoruz.

```
def my_init(m):
    if type(m) == nn.Linear:
        print("Init", *[(name, param.shape)
                        for name, param in m.named_parameters()][0])
        nn.init.uniform_(m.weight, -10, 10)
        m.weight.data *= m.weight.data.abs() >= 5

net.apply(my_init)
net[0].weight[:2]
```

```
Init weight torch.Size([8, 4])
Init weight torch.Size([1, 8])
```

```
tensor([[ 0.0000, -8.7999,  9.7935,  8.7815],
        [-0.0000,  0.0000, -0.0000,  5.8831]], grad_fn=<SliceBackward0>)
```

Her zaman parametreleri doğrudan ayarlama seçeneğimiz olduğunu unutmayın.

```
net[0].weight.data[:] += 1
net[0].weight.data[0, 0] = 42
net[0].weight.data[0]
```

```
tensor([42.0000, -7.7999, 10.7935,  9.7815])
```

5.2.3 Bağlı Parametreler

Genellikle, parametreleri birden çok katmanda paylaşmak isteriz. Bunu biraz daha zekice bir şekilde nasıl yapacağımızı görelim. Aşağıda yoğun (dense) bir katman ayırıyoruz ve ardından onun parametrelerini de özellikle başka bir katmanınkileri ayarlamak için kullanıyoruz.

```
# Parametrelerine atıfta bulunabilmemiz için paylaşılan katmana bir ad
# vermemiz gerekiyor
shared = nn.Linear(8, 8)
net = nn.Sequential(nn.Linear(4, 8), nn.ReLU(),
                    shared, nn.ReLU(),
                    shared, nn.ReLU(),
                    nn.Linear(8, 1))
```

(continues on next page)

(continued from previous page)

```
net(X)
# Parametrelerin aynı olup olmadığını kontrol edin
print(net[2].weight.data[0] == net[4].weight.data[0])
net[2].weight.data[0, 0] = 100
# Aynı değere sahip olmak yerine aslında aynı nesne olduklarından emin olun
print(net[2].weight.data[0] == net[4].weight.data[0])
```

```
tensor([True, True, True, True, True, True, True, True])
tensor([True, True, True, True, True, True, True, True])
```

Bu örnek, ikinci ve üçüncü katmanın parametrelerinin birbirine bağlı olduğunu göstermektedir. Sadece eşit değiller, tamamen aynı tensörle temsil ediliyorlar. Bu yüzden parametrelerden birini değiştirirsek diğeri de değişir. Merak edebilirsiniz, parametreler bağlı olduğunda gradyanlara ne olur? Model parametreleri gradyanlar içerdiğinden, ikinci ve üçüncü gizli katmanların gradyanları geri yayma sırasında birbiriyle toplanır.

5.2.4 Özet

- Model parametrelerine erişmek, ilklemek ve onları bağlamak için birkaç farklı yol var.
- Özelleştirilmiş ilkleme kullanabiliriz.

5.2.5 Alıştırmalar

1. Section 5.1 içinde tanımlanan `FancyMLP` modelini kullanınız ve çeşitli katmanların parametrelerine erişiniz.
2. Farklı ilkleyicileri keşfetmek için ilkleme modülü dökümanına bakınız.
3. Paylaşılan bir parametre katmanı içeren bir MLP oluşturunuz ve onu eğitiniz. Eğitim sürecinde, her katmanın model parametrelerini ve gradyanlarını gözlemleyiniz.
4. Parametreleri paylaşmak neden iyi bir fikirdir?

Tartışmalar[79]

5.3 Özelleştirilmiş Katmanlar

Derin öğrenmenin başarısının ardındaki faktörlerden biri, çok çeşitli görevlere uygun mimariler tasarlamak için yaratıcı yollarla oluşturulabilen çok çeşitli katmanların mevcut olmasıdır. Örneğin, araştırmacılar özellikle imgelerle baş etmek, metin, dizili veriler üzerinde döngü yapmak ve dinamik programlama yapmak için katmanlar icat ettiler. Er ya da geç, derin öğrenme çerçevesinde henüz var olmayan bir katmanla karşılaşacaksınız (veya onu icat edeceksiniz). Özel bir katman oluşturmanız gerekebilir. Bu bölümde size bunu nasıl yapacağınızı gösteriyoruz.

[79] https://discuss.d2l.ai/t/57

5.3.1 Paramatresiz Katmanlar

Başlangıç olarak, kendi parametresi olmayan özelleştirilmiş bir katman oluşturalım. Bloğu tanıttığımızı bölümü hatırlarsanız, Section 5.1 içindeki, burası size tanıdık gelecektir. Aşağıdaki `CenteredLayer` sınıfı, girdiden ortalamayı çıkarır. Bunu inşa etmek için, temel katman sınıfından kalıtımla üretmemiz ve ileri yayma işlevini uygulamamız gerekir.

```
import torch
from torch import nn
from torch.nn import functional as F


class CenteredLayer(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, X):
        return X - X.mean()
```

Katmanımızın amaçlandığı gibi çalıştığını, ona biraz veri besleyerek doğrulayalım.

```
layer = CenteredLayer()
layer(torch.FloatTensor([1, 2, 3, 4, 5]))
```

```
tensor([-2., -1.,  0.,  1.,  2.])
```

Artık katmanımızı daha karmaşık modeller oluşturmada bir bileşen olarak kullanabiliriz.

```
net = nn.Sequential(nn.Linear(8, 128), CenteredLayer())
```

Ekstra bir makulluk kontrolü olarak, ağa rastgele veri gönderebilir ve ortalamanın gerçekte 0 olup olmadığına bakabiliriz. Kayan virgüllü sayılarla uğraştığımız için, nicemlemeden dolayı çok küçük sıfır olmayan bir sayı görebiliriz.

```
Y = net(torch.rand(4, 8))
Y.mean()
```

```
tensor(-7.4506e-09, grad_fn=<MeanBackward0>)
```

5.3.2 Parametreli Katmanlar

Artık basit katmanları nasıl tanımlayacağımızı bildiğimize göre, eğitim yoluyla ayarlanabilen parametrelerle katmanları tanımlamaya geçelim. Bazı temel idari işlevleri sağlayan parametreler oluşturmak için yerleşik işlevleri kullanabiliriz. Özellikle erişim, ilkleme, paylaşma, modeli kaydetme ve yükleme parametrelerini yönetirler. Bu şekilde, diğer faydaların yanı sıra, her özel katman için özel serileştirme (serialization) rutinleri yazmamız gerekmeyecek.

Şimdi tam bağlı katman sürümümüzü uygulayalım. Bu katmanın iki parametreye ihtiyaç duyduğunu hatırlayınız, biri ağırlığı ve diğeri ek girdiyi temsil etmek için. Bu uygulamada, varsayılan

olarak ReLU etkinleştirmesini kullanıyoruz. Bu katman, sırasıyla girdilerin ve çıktıların sayısını gösteren `in_units` ve `units` girdi argümanlarının girilmesini gerektirir.

```
class MyLinear(nn.Module):
    def __init__(self, in_units, units):
        super().__init__()
        self.weight = nn.Parameter(torch.randn(in_units, units))
        self.bias = nn.Parameter(torch.randn(units,))
    def forward(self, X):
        linear = torch.matmul(X, self.weight.data) + self.bias.data
        return F.relu(linear)
```

Daha sonra, `MyLinear` sınıfını ilkliyoruz ve model parametrelerine erişiyoruz.

```
linear = MyLinear(5, 3)
linear.weight
```

```
Parameter containing:
tensor([[-0.4603,  0.5533,  0.0135],
        [-1.0431, -1.7814,  0.5278],
        [-0.9134,  0.1051, -0.5784],
        [ 1.8552,  0.6963,  0.8525],
        [-0.5034,  1.6521, -1.1436]], requires_grad=True)
```

Özel kesim katmanları kullanarak doğrudan ileri yayma hesaplamaları yapabiliriz.

```
linear(torch.rand(2, 5))
```

```
tensor([[0.6806, 1.0990, 1.5497],
        [0.0000, 0.0619, 0.8226]])
```

Özelleştirilmiş kesim katmanlar kullanarak da modeller oluşturabiliriz. Bir kere ona sahip olduğumuzda, onu tıpkı yerleşik tam bağlı katman gibi kullanabiliriz.

```
net = nn.Sequential(MyLinear(64, 8), MyLinear(8, 1))
net(torch.rand(2, 64))
```

```
tensor([[9.2351],
        [7.0112]])
```

5.3.3 Özet

- Temel katman sınıfı üzerinden özel kesim katmanlar tasarlayabiliriz. Bu, kütüphanedeki mevcut katmanlardan farklı davranan yeni esnek katmanlar tanımlamamıza olanak tanır.
- Tanımlandıktan sonra, özel kesim katmanlar keyfi bağlamlarda ve mimarilerde çağrılabilir.
- Katmanlar, yerleşik işlevler aracılığıyla yaratılabilen yerel parametrelere sahip olabilirler.

5.3.4 Alıştırmalar

1. Bir girdi alan ve bir tensör indirgemesi hesaplayan bir katman tasarlayınız, yani $y_k = \sum_{i,j} W_{ijk} x_i x_j$ döndürsün.
2. Verilerin Fourier katsayılarının ilk baştaki yarısını döndüren bir katman tasarlayınız.

Tartışmalar[80]

5.4 Dosya Okuma/Yazma

Şu ana kadar verilerin nasıl işleneceğini ve derin öğrenme modellerinin nasıl oluşturulacağını, eğitileceğini ve test edileceğini tartıştık. Bununla birlikte, bir noktada, öğrenilmiş modellerden yeterince mutlu olacağımızı, öyle ki sonuçları daha sonra çeşitli bağlamlarda kullanmak için saklamak isteyeceğimiz umuyoruz (belki de konuşlandırmada tahminlerde bulunmak için). Ek olarak, uzun bir eğitim sürecini çalıştırırken, sunucumuzun güç kablosuna takılırsak birkaç günlük hesaplamayı kaybetmememiz için en iyi mesleki uygulama, periyodik olarak ara sonuçları kaydetmektir (kontrol noktası belirleme). Bu nedenle, hem bireysel ağırlık vektörlerini hem de tüm modelleri nasıl yükleyeceğimizi ve depolayacağımızı öğrenmenin zamanı geldi. Bu bölüm her iki sorunu da irdelemektedir.

5.4.1 Tensörleri Yükleme ve Kaydetme

Tek tensörler için, sırasıyla okumak ve yazmak için `load` ve `save` işlevlerini doğrudan çağırabiliriz. Her iki işleve de bir ad girmemiz gerekir ve `save` işlevi için kaydedilecek değişkeni de girdi olarak girmemiz gerekir.

```
import torch
from torch import nn
from torch.nn import functional as F

x = torch.arange(4)
torch.save(x, 'x-file')
```

Artık verileri kayıtlı dosyadan belleğe geri okuyabiliriz.

```
x2 = torch.load('x-file')
x2
```

```
tensor([0, 1, 2, 3])
```

Tensörlerin bir listesini kaydedebilir ve bunları belleğe geri okuyabiliriz.

```
y = torch.zeros(4)
torch.save([x, y],'x-files')
x2, y2 = torch.load('x-files')
(x2, y2)
```

[80] https://discuss.d2l.ai/t/59

```
(tensor([0, 1, 2, 3]), tensor([0., 0., 0., 0.]))
```

Dizgilerden (string) tensörlere eşleşen bir sözlük bile yazabilir ve okuyabiliriz. Bu, bir modeldeki tüm ağırlıkları okumak veya yazmak istediğimizde kullanışlıdır.

```
mydict = {'x': x, 'y': y}
torch.save(mydict, 'mydict')
mydict2 = torch.load('mydict')
mydict2
```

```
{'x': tensor([0, 1, 2, 3]), 'y': tensor([0., 0., 0., 0.])}
```

5.4.2 Model Parametrelerini Yükleme ve Kayıt Etme

Bireysel ağırlık vektörlerini (veya diğer tensörleri) kaydetmek faydalıdır, ancak tüm modeli kaydetmek (ve daha sonra yüklemek) istiyorsak bu çok bezdirici hale gelir. Sonuçta, her yere dağıtılmış yüzlerce parametre grubumuz olabilir. Bu nedenle derin öğrenme çerçevesi, tüm ağları yüklemek ve kaydetmek için yerleşik işlevsellikler sağlar. Farkında olunması gereken önemli bir ayrıntı, bunun tüm modeli değil, model *parametrelerini* kaydetmesidir. Örneğin, 3 katmanlı bir MLP'ye sahipsek, mimariyi ayrıca belirtmemiz gerekir. Bunun nedeni, modellerin kendilerinin keyfi kodlar içerebilmeleri, dolayısıyla doğal olarak serileştirilememeleridir. Bu nedenle, bir modeli eski haline getirmek için, kod içinde mimariyi oluşturmamız ve ardından parametreleri diskten yüklememiz gerekir. Tanıdık MLP'mizle başlayalım.

```
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(20, 256)
        self.output = nn.Linear(256, 10)

    def forward(self, x):
        return self.output(F.relu(self.hidden(x)))

net = MLP()
X = torch.randn(size=(2, 20))
Y = net(X)
```

Daha sonra modelin parametrelerini `mlp.params` adıyla bir dosya olarak kaydediyoruz.

```
torch.save(net.state_dict(), 'mlp.params')
```

Modeli geri yüklemek için, orijinal MLP modelinin bir klonunu örnekliyoruz. Model parametrelerini rastgele ilklemek yerine, dosyada saklanan parametreleri doğrudan okuyoruz.

```
clone = MLP()
clone.load_state_dict(torch.load('mlp.params'))
clone.eval()
```

```
MLP(
  (hidden): Linear(in_features=20, out_features=256, bias=True)
  (output): Linear(in_features=256, out_features=10, bias=True)
)
```

Her iki örnek de aynı model parametrelerine sahip olduğundan, aynı `X` girdisinin hesaplamalı sonucu aynı olmalıdır. Bunu doğrulayalım.

```
Y_clone = clone(X)
Y_clone == Y
```

```
tensor([[True, True, True, True, True, True, True, True, True, True],
        [True, True, True, True, True, True, True, True, True, True]])
```

5.4.3 Özet

- `save` ve `load` fonksiyonları, tensör nesneler için dosya yazma/okuma gerçekleştirmek için kullanılabilir.
- Parametre sözlüğü aracılığıyla bir ağ için tüm parametre kümelerini kaydedebilir ve yükleyebiliriz.
- Mimarinin kaydedilmesi parametreler yerine kodda yapılmalıdır.

5.4.4 Alıştırmalar

1. Eğitilmiş modelleri farklı bir cihaza konuşlandırmaya gerek olmasa bile, model parametrelerini saklamanın pratik faydaları nelerdir?
2. Bir ağın yalnızca belli bölümlerini farklı bir mimariye sahip bir ağa dahil edilecek şekilde yeniden kullanmak istediğimizi varsayalım. Yeni bir ağdaki eski bir ağdan ilk iki katmanı nasıl kullanmaya başlarsınız?
3. Ağ mimarisini ve parametrelerini kaydetmeye ne dersiniz? Mimariye ne gibi kısıtlamalar uygularsınız?

Tartışmalar[81]

5.5 GPU (Grafik İşleme Birimi)

Chapter 1.5 için de, son yirmi yılda hesaplamanın hızlı büyümesini tartıştık. Özetle, GPU performansı 2000 yılından bu yana her on yılda bir 1000 kat artmıştır. Bu büyük fırsatlar sunarken aynı zamanda bu tür bir performansın sağlanması için önemli bir gereksinim olduğunu da göstermektedir.

Bu bölümde, araştırmanız için bu hesaplamalı performanstan nasıl yararlanılacağını tartışmaya başlıyoruz. Öncelikle tek GPU'ları kullanarak ve daha sonra, birden çok GPU ve birden çok sunucuyu (birden çok GPU ile) nasıl kullanacağınızı tartışacağız.

[81] https://discuss.d2l.ai/t/61

Özellikle, hesaplamalar için tek bir NVIDIA GPU'nun nasıl kullanılacağını tartışacağız. İlk olarak kurulu en az bir NVIDIA GPU'nuz olduğundan emin olun. Ardından, NVIDIA sürücüsünü ve CUDA'yı[82] indirin ve uygun yere kurmak için istemleri takip edin. Bu hazırlıklar tamamlandıktan sonra, `nvidia-smi`, komutu grafik kartı bilgilerini görüntülemek için kullanılabilir.

```
!nvidia-smi
```

```
Mon Oct 17 22:48:03 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.106.00   Driver Version: 460.106.00   CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  Off  | 00000000:00:1B.0 Off |                    0 |
| N/A   39C    P0    50W / 300W |   1760MiB / 16160MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  Tesla V100-SXM2...  Off  | 00000000:00:1C.0 Off |                    0 |
| N/A   38C    P0    50W / 300W |   1542MiB / 16160MiB |     30%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   2  Tesla V100-SXM2...  Off  | 00000000:00:1D.0 Off |                    0 |
| N/A   38C    P0    43W / 300W |      3MiB / 16160MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   3  Tesla V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   36C    P0    53W / 300W |   1432MiB / 16160MiB |     22%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    1   N/A  N/A     32127      C   ...l-tr-release-0/bin/python     1539MiB |
|    3   N/A  N/A     32127      C   ...l-tr-release-0/bin/python     1429MiB |
+-----------------------------------------------------------------------------+
```

PyTorch'ta her dizilimin bir aygıtı vardır, biz onu genellikle bağlam olarak adlandırırız. Şimdiye kadar, varsayılan olarak, tüm değişkenler ve ilişkili hesaplama CPU'ya atanmıştır. Tipik olarak, diğer bağlamlar çeşitli GPU'lar olabilir. İşleri birden çok sunucuya dağıttığımızda işler arap saçına dönebilir. Dizileri bağlamlara akıllıca atayarak, cihazlar arasında veri aktarımında harcanan zamanı en aza indirebiliriz. Örneğin, GPU'lu bir sunucuda sinir ağlarını eğitirken, genellikle modelin parametrelerinin GPU'da kalmasını tercih ederiz.

Ardından, PyTorch'un GPU sürümünün kurulu olduğunu onaylamamız gerekiyor. PyTorch'un bir CPU sürümü zaten kuruluysa, önce onu kaldırmamız gerekir. Örneğin, `pip uninstall torch` komutunu kullanın, ardından CUDA sürümünüze göre ilgili PyTorch sürümünü kurun. CUDA 10.0'un kurulu olduğunu varsayarsak, CUDA 10.0'u destekleyen PyTorch sürümünü `pip install torch-cu100` aracılığıyla kurabilirsiniz.

[82] https://developer.nvidia.com/cuda-downloads

Bu bölümdeki programları çalıştırmak için en az iki GPU'ya ihtiyacınız var. Bunun çoğu masaüstü bilgisayar için abartılı olabileceğini, ancak bulutta (ör. AWS EC2 çoklu-GPU bulut sunucularını kullanarak) kolayca kullanılabileceğini unutmayın. Hemen hemen diğer bütün bölümler birden fazla GPU *gerektirmez*. Bunun burdaki kullanım nedeni, basitçe verilerin farklı cihazlar arasında nasıl aktığını göstermektir.

5.5.1 Hesaplama Cihazları

Depolama ve hesaplama için CPU ve GPU gibi cihazları belirtebiliriz. Varsayılan olarak, ana bellekte tensörler oluşturulur ve ardından bunu hesaplamak için CPU'yu kullanır.

PyTorch'ta CPU ve GPU, `torch.device('cpu')` ve `torch.device('cuda')` ile gösterilebilir. `cpu` aygıtının tüm fiziksel CPU'lar ve bellek anlamına geldiğine dikkat edilmelidir. Bu, PyTorch'un hesaplamalarının tüm CPU çekirdeklerini kullanmaya çalışacağı anlamına gelir. Bununla birlikte, bir `gpu` cihazı yalnızca bir kartı ve ona denk gelen belleği temsil eder. Birden çok GPU varsa, i. GPU (i 0'dan başlar) temsil etmek için `torch.device(f'cuda:{i}')`yi kullanırız. Ayrıca `gpu:0` ve `gpu` eşdeğerdir.

```
import torch
from torch import nn

torch.device('cpu'), torch.device('cuda'), torch.device('cuda:1')
```

```
(device(type='cpu'), device(type='cuda'), device(type='cuda', index=1))
```

Mevcut GPU adetini sorgulayabiliriz.

```
torch.cuda.device_count()
```

```
2
```

Şimdi, istenen GPU'lar var olmasa bile kodları çalıştırmamıza izin veren iki kullanışlı işlev tanımlıyoruz.

```
def try_gpu(i=0):  #@save
    """Varsa gpu(i) döndürün, aksi takdirde cpu() döndürün."""
    if torch.cuda.device_count() >= i + 1:
        return torch.device(f'cuda:{i}')
    return torch.device('cpu')

def try_all_gpus():  #@save
    """Mevcut tüm GPU'ları veya GPU yoksa [cpu()] döndürün."""
    devices = [torch.device(f'cuda:{i}')
             for i in range(torch.cuda.device_count())]
    return devices if devices else [torch.device('cpu')]

try_gpu(), try_gpu(10), try_all_gpus()
```

```
(device(type='cuda', index=0),
 device(type='cpu'),
 [device(type='cuda', index=0), device(type='cuda', index=1)])
```

5.5.2 Tensörler and GPUlar

Varsayılan olarak, CPU'da tensörler oluşturulur. Tensörün bulunduğu cihazı sorgulayabiliriz.

```
x = torch.tensor([1, 2, 3])
x.device
```

```
device(type='cpu')
```

Birden çok terimle çalışmak istediğimizde, aynı bağlamda olmaları gerektiğine dikkat etmemiz önemlidir. Örneğin, iki tensörü toplarsak, her iki argümanın da aynı cihazda olduğundan emin olmamız gerekir—aksi takdirde çerçeve, sonucu nerede saklayacağını ve hatta hesaplamayı nerede gerçekleştireceğine nasıl karar vereceğini bilemez.

GPU'da Depolama

GPU'da bir tensör depolamanın birkaç yolu vardır. Örneğin, bir tensör oluştururken bir depolama cihazı belirleyebiliriz. Sonra, ilk gpu'da tensör değişkeni `X`'i oluşturuyoruz. Bir GPU'da oluşturulan tensör yalnızca o GPU'nun belleğini harcar. GPU bellek kullanımını görüntülemek için `nvidia-smi` komutunu kullanabiliriz. Genel olarak, GPU bellek sınırını aşan veriler oluşturmadığımızdan emin olmamız gerekir.

```
X = torch.ones(2, 3, device=try_gpu())
X
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.]], device='cuda:0')
```

En az iki GPU'ya sahip olduğunuzu varsayarak, aşağıdaki kod ikinci GPU'da keyfi bir tensor oluşturacaktır.

```
Y = torch.rand(2, 3, device=try_gpu(1))
Y
```

```
tensor([[0.5575, 0.9225, 0.4448],
        [0.2045, 0.3821, 0.9360]], device='cuda:1')
```

Kopyalama

`X + Y`'yi hesaplamak istiyorsak, bu işlemi nerede gerçekleştireceğimize karar vermemiz gerekir. Örneğin Fig. 5.5.1 içinde gösterildiği gibi, `X`'i ikinci GPU'ya aktarabilir ve işlemi orada gerçekleştirebiliriz. Sadece `X` ve `Y`'yi *toplamayın*, çünkü bu bir istisnayla sonuçlanacaktır. Koşma zamanı motoru ne yapacağını bilemez: Veriyi aynı cihazda bulamaz ve başarısız olur. Çünkü `Y` ikinci GPU'da bulunur, ikisini toplamadan önce `X`'i taşımamız gerekir.

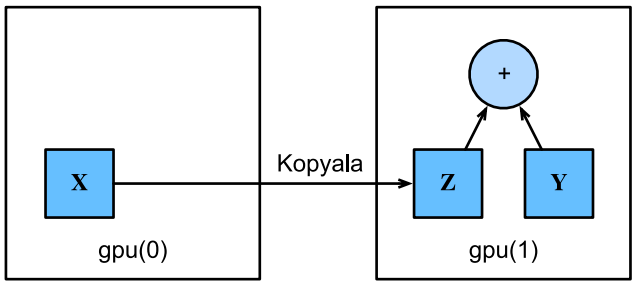


Fig. 5.5.1: İşlemi yapmadan önce verileri aynı cihaza kopyalayın.

```
Z = X.cuda(1)
print(X)
print(Z)
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.]], device='cuda:0')
tensor([[1., 1., 1.],
        [1., 1., 1.]], device='cuda:1')
```

Artık veriler aynı GPU'da olduğuna göre (hem `Z` hem de `Y`), onları toplayabiliriz.

```
Y + Z
```

```
tensor([[1.5575, 1.9225, 1.4448],
        [1.2045, 1.3821, 1.9360]], device='cuda:1')
```

`Z` değişkeninizin halihazırda ikinci GPU'nuzda var olduğunu hayal edin. Gene de `Z.cuda(1)` diye çağırırsak ne olur? Kopyalamak ve yeni bellek ayırmak yerine `Z`'yi döndürür.

```
Z.cuda(1) is Z
```

```
True
```

İnsanlar hızlı olmalarını beklediklerinden makine öğrenmesi için GPU'ları kullanıyorlar. Ancak değişkenlerin cihazlar arasında aktarılması yavaştır. Bu yüzden, yapmanıza izin vermeden önce yavaş bir şey yapmak istediğinizden %100 emin olmanızı istiyoruz. Derin öğrenme çerçevesi kopyayı çökmeden otomatik olarak yaptıysa, yavaş çalışan bir kod yazdığınızı fark etmeyebilirsiniz.

Ayrıca, cihazlar (CPU, GPU'lar ve diğer makineler) arasında veri aktarımı, hesaplamadan çok daha yavaş bir şeydir. Ayrıca, daha fazla işlem ile ilerlemeden önce verilerin gönderilmesini (veya daha doğrusu alınmasını) beklememiz gerektiğinden bu paralelleştirmeyi çok daha zor hale getirir. Bu nedenle kopyalama işlemlerine büyük özen gösterilmelidir. Genel bir kural olarak, birçok küçük işlem, tek bir büyük işlemden çok daha kötüdür. Dahası, bir seferde birkaç işlem, koda serpiştirilmiş birçok tek işlemden çok daha iyidir, ne yaptığınızı biliyorsanız o ayrı. Bu durumda, bir aygıtın bir şey yapmadan önce bir diğerini beklemesi gerektiğinde bu tür işlemler onu engelleyebilir. Bu biraz, kahvenizi telefonla ön sipariş vermek ve siz istediğinizde hazır olduğunu öğrenmek yerine sırada bekleyerek sipariş etmek gibidir.

Son olarak, tensörleri yazdırdığımızda veya tensörleri NumPy formatına dönüştürdüğümüzde, veri ana bellekte değilse, çerçeve onu önce ana belleğe kopyalayacak ve bu da ek iletim yüküne neden olacaktır. Daha da kötüsü, şimdi Python'un her şeyi tamamlanmasını beklemesine neden olan o korkunç global yorumlayıcı kilidine tabidir.

5.5.3 Sinir Ağları ve GPUlar

Benzer şekilde, bir sinir ağı modeli cihazları belirtebilir. Aşağıdaki kod, model parametrelerini GPU'ya yerleştirir.

```
net = nn.Sequential(nn.Linear(3, 1))
net = net.to(device=try_gpu())
```

Aşağıdaki bölümlerde GPU'larda modellerin nasıl çalıştırılacağına dair daha birçok örnek göreceğiz, çünkü onlar hesaplama açısından biraz daha yoğun hale gelecekler.

Girdi, GPU'da bir tensör olduğunda, model sonucu aynı GPU'da hesaplayacaktır.

```
net(X)
```

```
tensor([[0.7531],
        [0.7531]], device='cuda:0', grad_fn=<AddmmBackward0>)
```

Model parametrelerinin aynı GPU'da depolandığını doğrulayalım.

```
net[0].weight.data.device
```

```
device(type='cuda', index=0)
```

Kısacası tüm veriler ve parametreler aynı cihazda olduğu sürece modelleri verimli bir şekilde öğrenebiliriz. Sonraki bölümlerde bu tür birkaç örnek göreceğiz.

5.5.4 Özet

- Depolama ve hesaplama için CPU veya GPU gibi cihazları belirleyebiliriz. Varsayılan olarak, veriler ana bellekte oluşturulur ve ardından hesaplamalar için CPU kullanılır.
- Derin öğrenme çerçevesi, hesaplama için tüm girdi verilerinin aynı cihazda olmasını gerektirir, ister CPU ister aynı GPU olsun.
- Verileri dikkatsizce taşıyarak önemli bir performans kaybına uğrayabilirsiniz. Tipik bir hata şudur: GPU'daki her minigrup için kaybı hesaplamak ve bunu komut satırında kullanıcıya geri bildirmek (veya bir NumPy `ndarray`'de kaydetmek), bu tüm GPU'ları durduran global yorumlayıcı kilidini tetikleyecektir. GPU içinde kayıt tutmak için bellek ayırmak ve yalnızca daha büyük kayıtları taşımak çok daha iyidir.

5.5.5 Alıştırmalar

1. Büyük matrislerin çarpımı gibi daha büyük bir hesaplama görevi deneyiniz ve CPU ile GPU arasındaki hız farkını görünüz. Az miktarda hesaplama içeren bir göreve ne olur?
2. GPU'daki model parametrelerini nasıl okuyup yazmalıyız?
3. 100×100'lük 1000 matris-matris çarpımını hesaplamak için gereken süreyi ölçünüz ve her seferde bir sonucun çıktı matrisinin Frobenius normunu günlüğe kaydetmeye karşılık GPU'da günlük tutma ve yalnızca son sonucu aktarmayı kıyaslayınız.
4. İki GPU'da iki matris-matris çarpımını aynı anda gerçekleştirme ile tek bir GPU'da sıralı gerçekleştirmenin ne kadar zaman aldığını ölçerek karşılaştırınız. İpucu: Neredeyse doğrusal bir ölçekleme görmelisiniz.

Tartışmalar[83]

[83] https://discuss.d2l.ai/t/63

6 | Evrişimli Sinir Ağları

Önceki bölümlerde, her örnek iki boyutlu bir piksel ızgarasından oluşan imge verileriyle karşılaştık. Siyah-beyaz veya renkli imgeleri kullanıp kullanmadığımıza bağlı olarak, her piksel konumu sırasıyla bir veya birden çok sayısal değerle ilişkilendirilebilir. Şimdiye kadar, bu zengin yapıyla başa çıkma yolumuz çok da tatmin edici değildi. Her imgenin mekansal yapısını tek boyutlu vektörlere düzleştirerek onları tam bağlı bir MLP'ye besledik. Bu ağlar özniteliklerin sıralamasından etkilenmez olduğundan, piksellerin konumsal yapısına karşılık gelen bir sırayı koruyup korumadığımıza veya MLP'nin parametrelerini oturtmadan önce tasarım matrisimizin sütunlarını devşirmemize (permute) bakılmaksızın benzer sonuçlar elde edebiliriz. Tercihen, imge verilerinden öğrenmeye yönelik etkili modeller oluşturmak için yakındaki piksellerin tipik olarak birbiriyle ilişkili olduğu ön bilgimizden yararlanırız.

Bu bölümde, tam olarak bu amaç için tasarlanmış güçlü bir sinir ağları ailesi olan *evrişimli sinir ağları (convolutional neural networks)* (CNN) tanıtılmaktadır. CNN tabanlı mimariler artık bilgisayarla görme alanında her yerde bulunmaktadırlar ve o kadar baskın hale gelmişlerdir ki, bugün bu yaklaşım üzerinden inşa etmeden bir kimsenin ticari bir uygulama geliştirmesi veya imge tanıma, nesne algılama veya anlamsal bölünme (semantic segmentation) ile ilgili bir yarışmaya katılması zordur.

Bilinen adlarıyla modern CNN'ler, tasarımlarını biyolojiden, grup teorisiden ve aşırı olmayan miktarda deneysel müdahaleden alınan ilhamlara borçludur. Doğru modellere ulaşmada örneklem verimliliğine ek olarak, CNN'ler hem tam bağlı mimarilere göre daha az parametre gerektirdiklerinden, hem de GPU çekirdeklerinde evrişimlerin paralelleştirilmesi kolay olduğundan hesaplama açısından verimli olma eğilimindedirler. Sonuç olarak, uygulayıcılar genellikle CNN'leri ne zaman mümkün olursa uygularlar ve böylece yinelemeli sinir ağlarının geleneksel olarak kullanıldığı ses, metin ve zaman serisi analizi gibi tek boyutlu bir dizi yapısına sahip görevlerde bile giderek daha güvenilir rakipler olarak ortaya çıkmışlardır. CNN'lerin bazı zekice uyarlamaları onları çizge yapılı verileri ve tavsiye sistemleri de taşır hale getirdi.

İlk olarak, tüm evrişimli ağların omurgasını oluşturan temel işlemleri irdeleyeceğiz. Bunlar, evrişimli katmanlarının kendisini, dolgu (padding) ve uzun adım (stride) gibi esaslı ince detayları, komşu konumsal bölgeler boyunca bilgi toplamak için kullanılan ortaklama (pooling) katmanlarını, her katmanda birden fazla kanal kullanımını ve modern mimarilerin yapısının dikkatli bir şekilde tartışılmasını içerir. Bu bölümü, modern derin öğrenmenin doğuşundan çok önce başarıyla konuşlandırılan ilk evrişimli ağ olan LeNet'in tam bir çalışan örneği ile sonlandıracağız. Bir sonraki bölümde, tasarımları modern uygulayıcılar tarafından yaygın olarak kullanılan tekniklerin çoğunu temsil eden bazı popüler ve nispeten yeni CNN mimarilerinin tam uygulamalarına dalacağız.

6.1 Tam Bağlı Katmanlardan Evrişimlere

Bugüne kadar, buraya kadar tartıştığımız modeller, çizelge halindeki (tabular) veriyle uğraşırken uygun seçenekler olmaya devam ediyorlar. Çizelge hali ile, verinin, özniteliklere karşılık gelen sütunlardan ve örneklere arşılık gelen satırlardan oluştuğunu kastediyoruz. Çizelge veriyle, aradığımız desenlerin öznitelikler arasında etkileşimler içerebileceğini tahmin edebiliriz, ancak özniteliklerin nasıl etkileşime girdiğine ilişkin herhangi bir *önsel* yapıyı varsayamayız.

Bazen, zanaatkar (işini bilen) mimarilerin yapımına gerçekten rehberlik etmede bilgi eksikliğimiz olur. Bu gibi durumlarda, bir MLP yapabileceğimizin en iyisi olabilir. Bununla birlikte, yüksek boyutlu algısal veriler için, bu tür yapı içermeyen ağlar kullanışsız hale gelebilir.

Mesela, kedileri köpeklerden ayırma örneğimize dönelim. Veri toplamada kapsamlı bir iş yaptık, bir megapiksel fotoğraflardan açıklamalı bir veri kümesi topladık diyelim. Bu, ağa giren her girdinin bir milyon boyuta sahip olduğu anlamına gelir. Section 3.4.3 içindeki tam bağlı katmanların parametreleştirme maliyeti konusundaki tartışmalarımıza göre, bin gizli boyuta saldırgan bir azaltma bile, $10^6 \times 10^3 = 10^9$ parametre ile karakterize edilen tam bağlı bir katman gerektirecektir. Çok sayıda GPU'muz, dağıtılmış eniyileme için bir çaremiz ve olağanüstü bir sabrımız olmadıkça, bu ağın parametrelerini öğrenmek mümkün olmayabilir.

Dikkatli bir okuyucu, bu argümana bir megapiksel çözünürlüğün gerekli olmayabileceği temelinde itiraz edebilir. Bununla birlikte, yüz bin piksel ile mücadele edebilsek de, 1000 birim büyüklüğündeki gizli katmanımız, imgelerin iyi temsillerini öğrenmek için gereken gizli birimlerin sayısını oldukça hafife alır, bu nedenle pratik bir sistem hala milyarlarca parametre gerektirecektir. Dahası, bir sınıflandırıcıyı bu kadar çok parametre oturtarak öğrenmek muazzam bir veri kümesini toplamayı gerektirebilir. Ayrıca, bugün hem insanlar hem de bilgisayarlar kedileri köpeklerden oldukça iyi ayırt edebiliyor, bu da görünüşte bu sezgilerle çelişiyor. Bunun nedeni, imgelerin insanlar ve makine öğrenmesi modelleri tarafından istismar edilebilecek zengin bir yapı sergilemesidir. Evrişimli sinir ağları (CNN), makine öğrenmesinin doğal imgelerdeki bilinen yapıların bazılarını kullanmak için benimsediği yaratıcı bir yoldur.

6.1.1 Değişmezlik

İmgedeki bir nesneyi tespit etmek istediğinizi düşünün. Nesneleri tanımak için kullandığımız yöntemin görüntüdeki nesnenin kesin konumu ile aşırı derecede ilgili olmaması akla yatkın görünüyor. İdeal olarak, sistemimiz bu bilgiyi kullanmalıdır. Domuzlar genellikle uçmaz ve uçaklar genellikle yüzmez. Yine de, imgenin en üstünde görünen bir domuzu gene de fark etmeliyiz. Burada "Waldo Nerede" isimli çocuk oyunundan biraz ilham alabiliriz (Fig. 6.1.1 içinde tasvir edilmiştir). Oyun düzensiz sahneler ile dolu bir dizi faaliyetten oluşur. Waldo her birinde bir yerlerden ortaya çıkıyor, tipik olarakta alışılmadık bir yerde gizleniyor. Okuyucunun amacı onu bulmaktır. Karakteristik kıyafetine rağmen, dikkat dağıtıcı çok sayıda unsur nedeniyle şaşırtıcı derecede zor olabilir. Ancak, *Waldo'nun neye benzediği*, Waldo'nun nerede bulunduğuna bağlı değildir. İmgeyi her yama için bir puan atayabilen bir Waldo dedektörü ile tarayabiliriz ve bu da yamanın Waldo içerme olabilirliliğini gösterir. CNN'ler bu *konumsal değişmezlik* fikrini sistemleştirerek, daha az parametre ile yararlı gösterimleri öğrenmek için kullanırlar.

Fig. 6.1.1: "Waldo Nerede" oyundan bir resim.

Bilgisayarla görmede uygun bir sinir ağı mimarisi tasarımımıza rehberlik etmesi için birkaç arzulanan şeyi numaralandırarak bu sezgileri daha somut hale getirebiliriz:

1. En öncü katmanlarda, ağımız imgede nerede göründüğüne bakılmaksızın aynı yamaya benzer şekilde yanıt vermelidir. Bu ilke *çeviri değişmezliği* olarak adlandırılır.
2. Ağın en öncü katmanları, uzak bölgelerdeki imgenin içeriğine bakılmaksızın yerel bölgelere odaklanmalıdır. Bu, *yerellik* ilkesidir. Sonuç olarak, bu yerel temsiller tüm imge düzeyinde tahminler yapmak için toplanabilir.

Bunun matematik ile nasıl ifade edildiğini görelim.

6.1.2 MLP'yi Kısıtlama

Başlangıç olarak, iki boyutlu $\mathbf{X}$ imgeler ile ve onların doğrudan gizli $\mathbf{H}$ gösterimlerinin matematikte benzer ebatta matrisler ve kodda iki boyutlu tensörler olarak temsil edildiği bir MLP düşünebiliriz. Bu aklımızda yer etsin. Şimdi sadece girdileri değil, aynı zamanda gizli temsilleri de konumsal yapıya sahip olarak tasavvur ediyoruz.

$[\mathbf{X}]_{i,j}$ ve $[\mathbf{H}]_{i,j}$'nin, sırasıyla girdi imgesinde ve gizli gösterimde (i, j) pikselini gösterdiğini varsayalım. Sonuç olarak, gizli birimlerin her birinin girdi piksellerinin her birinden girdi almasını sağlamak için ağırlık matrislerini kullanmaktan (daha önce MLP'lerde yaptığımız gibi) parametrelerimizi dördüncü dereceden ağırlık tensörleri W olarak temsil etmeye geçeceğiz. $\mathbf{U}$'nun ek girdiler içerdiğini varsayalım, tam bağlı katmanı biçimsel olarak şöyle ifade edebiliriz:

$$\begin{aligned}[\mathbf{H}]_{i,j} &= [\mathbf{U}]_{i,j} + \sum_k \sum_l [\mathsf{W}]_{i,j,k,l} [\mathbf{X}]_{k,l} \\ &= [\mathbf{U}]_{i,j} + \sum_a \sum_b [\mathsf{V}]_{i,j,a,b} [\mathbf{X}]_{i+a,j+b}.\end{aligned}, \tag{6.1.1}$$

W'ten V'ye olan geçiş, her iki dördüncü mertebeden katsayılar arasında bire bir örtüşme olduğundan, şimdilik tamamen gösteriş içindir. Biz sadece (k, l) altindisleri yeniden $k = i + a$ ve $l = j + b$ diye dizinliyoruz. Başka bir deyişle, $[\mathsf{V}]_{i,j,a,b} = [\mathsf{W}]_{i,j,i+a,j+b}$'yi kurduk. a ve b indisleri, tüm imgeyi

kapsayan hem pozitif hem de negatif basit ek değerlerin üzerinden çalışır. $[\mathbf{H}]_{i,j}$ gizli gösterimindeki herhangi bir (i, j) konumunun değerini, x'deki, (i, j) civarında ortalanmış ve $[\mathsf{V}]_{i,j,a,b}$ ile ağırlıklandırılmış pikseller üzerinden toplayarak hesaplarız.

Çeviri Değişmezliği

Şimdi yukarıda bahsedilen ilk ilkeyi çağıralım: Çeviri değişmezliği. Bu, $\mathbf{X}$ girdisindeki bir kaymanın, gizli gösterimde $\mathbf{H}$'de bir kaymaya yol açması gerektiği anlamına gelir. Bu sadece V ve $\mathbf{U}$ aslında (i, j)'ye bağımlı değilse mümkündür, yani $[\mathsf{V}]_{i,j,a,b} = [\mathbf{V}]_{a,b}$ ve $\mathbf{U}$ bir sabit, diyelim u ise. Sonuç olarak, $\mathbf{H}$ tanımını basitleştirebiliriz:

$$[\mathbf{H}]_{i,j} = u + \sum_a \sum_b [\mathbf{V}]_{a,b} [\mathbf{X}]_{i+a,j+b}. \tag{6.1.2}$$

Bu bir *evrişim!* $[\mathbf{H}]_{i,j}$ değerini elde etmek için (i, j)'nin $[\mathbf{V}]_{a,b}$ katsayıları ile $(i + a, j + b)$ adresindeki pikselleri etkin bir şekilde ağırlıklandırıyoruz. $[\mathbf{V}]_{a,b}$'ün, artık imge içindeki konuma bağlı olmadığından, $[\mathsf{V}]_{i,j,a,b}$'den çok daha az katsayıya ihtiyaç duyduğunu unutmayın. Önemli bir ilerleme kaydettik!

Yerellik

Şimdi ikinci prensibi anımsıyalım: Yerellik. Yukarıda motive edildiği gibi, $[\mathbf{H}]_{i,j}$'te neler olup bittiğini değerlendirken ilgili bilgileri toplamak için (i, j) konumundan çok uzaklara bakmak zorunda olmamamız gerektiğine inanıyoruz. Bu, bazı $|a| > \Delta$ veya $|b| > \Delta$ aralığı dışında, $[\mathbf{V}]_{a,b} = 0$ diye ayarlamanız gerektiği anlamına gelir. Eşdeğer olarak, $[\mathbf{H}]_{i,j}$ olarak yeniden yazabiliriz

$$[\mathbf{H}]_{i,j} = u + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [\mathbf{V}]_{a,b} [\mathbf{X}]_{i+a,j+b}. \tag{6.1.3}$$

Özetle, (6.1.3) denkleminin bir *evrişimli katman* olduğuna dikkat edin. *Evrişimli sinir ağları* (CNN), evrişimli katmanlar içeren özel bir sinir ağları ailesidir. Derin öğrenme araştırma topluluğunda $\mathbf{V}$, bir *evrişim çekirdeği*, bir *filtre* veya genellikle öğrenilebilir parametreler olan katmanın *ağırlıkları* olarak adlandırılır. Yerel bölge küçük olduğunda, tam bağlı bir ağ ile karşılaştırıldığında fark çarpıcı olabilir. Daha önce, bir imge işleme ağındaki tek bir katmanı temsil etmek için milyarlarca parametre gerekirken şimdi genellikle girdilerin veya gizli temsillerin boyutsallığını değiştirmeden sadece birkaç yüz taneye ihtiyacımız var. Parametrelerdeki bu ciddi azalma için ödenen bedel, özniteliklerimizin artık çeviri değişmez olması ve katmanımızın her gizli etkinleştirmenin değerine karar verirken yalnızca yerel bilgileri içerebilmesidir. Tüm öğrenme tümevarımsal yanlılığın uygulamaya koymasına bağlıdır. Bu yanlılık gerçekle aynı yönde olduğunda, görünmeyen verilere iyi genelleyen örneklem verimli modeller elde ederiz. Ancak elbette, bu yanlılık gerçekle aynı yönde değilse, örneğin, imgelerin çeviri değişmez olmadığı ortaya çıkarsa, modellerimizin eğitim verilerimize uyması için bile debelenmesi gerekir.

6.1.3 Evrişimler

Daha da ilerlemeden önce, yukarıdaki işlemin neden bir evrişim (birlikte evrimleşme) olarak adlandırıldığını kısaca gözden geçirmeliyiz. Matematikte, iki fonksiyon arasındaki *evrişim*, mesela $f, g : \mathbb{R}^d \to \mathbb{R}$ için, şöyle tanımlanır:

$$(f * g)(\mathbf{x}) = \int f(\mathbf{z}) g(\mathbf{x} - \mathbf{z}) d\mathbf{z}. \tag{6.1.4}$$

Yani, bir işlev "çevrilmiş" ve $\mathbf{x}$ ile kaydırılmış olduğunda f ve g arasındaki örtüşmeyi ölçüyoruz. Ayrık nesnelere sahip olduğumuzda, integral bir toplama dönüşür. Örneğin, $\mathbb{Z}$ üzerinde çalışan indisi olan kare toplanabilir sonsuz boyutlu vektörler kümesinden vektörler için aşağıdaki tanımı elde ederiz:

$$(f * g)(i) = \sum_a f(a) g(i - a). \tag{6.1.5}$$

İki boyutlu tensörler için, sırasıyla f için (a, b) ve g için $(i - a, j - b)$ indisleri ile karşılık gelen bir toplama işlemine sahibiz:

$$(f * g)(i, j) = \sum_a \sum_b f(a, b) g(i - a, j - b). \tag{6.1.6}$$

Bu, (6.1.3) denklemine benzer, ama büyük bir farkla. $(i+a, j+b)$ kullanmak yerine farkı kullanıyoruz. Yine de, bu ayrımın çoğunlukla gösterişsel olduğunu unutmayın, çünkü (6.1.3) ve (6.1.6) arasındaki gösterimi her zaman eşleştirebiliriz. (6.1.3) denklemindeki orijinal tanımımız, daha doğru bir şekilde *çapraz korelasyonu* tanımlıyor. Aşağıdaki bölümde buna geri döneceğiz.

6.1.4 "Waldo Nerede"ye Tekrar Bakış

Waldo dedektörümüze dönersek, bunun neye benzediğini görelim. Evrişimli tabaka, belirli bir boyuttaki pencereleri seçer ve Fig. 6.1.2 içinde gösterildiği gibi V'ye göre yoğunlukları ağırlıklandırır. Bir model öğrenmeyi hedefleyebiliriz, böylece "Waldoluk" en yüksek nerede olursa olsun, gizli katman temsillerinde bir tepe değer bulmalıyız.

Fig. 6.1.2: Waldo'yu bul.

Bu yaklaşımla ilgili tek bir sorun var. Şimdiye kadar, imgelerin 3 kanaldan oluştuğunu görmezden geldik: Kırmızı, yeşil ve mavi. Gerçekte, imgeler iki boyutlu nesneler değil, yükseklik, genişlik ve kanal ile karakterize edilen üçüncü dereceden tensörlerdir, örn. $1024 \times 1024 \times 3$ şekilli pikseller. Bu eksenlerin ilk ikisi konumsal ilişkileri ilgilendirirken, üçüncüsü her piksel konumuna çok boyutlu bir temsil atama olarak kabul edilebilir. Böylece, X'i $[\mathsf{X}]_{i,j,k}$ olarak indisleriz. Evrişimli filtre buna göre uyarlanmak zorundadır. $[\mathbf{V}]_{a,b}$ yerine artık $[\mathsf{V}]_{a,b,c}$ var.

Dahası, girdimiz üçüncü mertebeden bir tensörden oluştuğu gibi, gizli temsillerimizi de benzer şekilde üçüncü mertebeden tensörler H olarak formüle etmenin iyi bir fikir olduğu ortaya çıkıyor. Başka bir deyişle, her uzaysal konuma karşılık gelen tek bir gizli gösterime sahip olmaktan ziyade, her uzaysal konuma karşılık gelen gizli temsillerin tam vektörünü istiyoruz. Gizli temsilleri, birbirinin üzerine yığılmış bir dizi iki boyutlu ızgaralar topluluğu olarak düşünebiliriz. Girdilerde olduğu gibi, bunlara bazen *kanallar* denir. Bunlara bazen *öznitelik eşlemeleri* de denir, çünkü her biri sonraki katmana uzamlaştırılmış bir öğrenilmiş öznitelikler kümesi sağlar. Sezgisel olarak, girdilere daha yakın olan alt katmanlarda, bazı kanalların kenarları tanımak için uzmanlaşabileceğini, diğerlerinin de dokuları tanıyabileceğini tasavvur edebilirsiniz.

Hem girdide (X) hem de gizli temsillerde (H) birden fazla kanalı desteklemek için V'ye dördüncü bir koordinat ekleyebiliriz: $[\mathsf{V}]_{a,b,c,d}$. Sahip olduğumuz her şeyi bir araya koyarsak:

$$[\mathsf{H}]_{i,j,d} = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_{c} [\mathsf{V}]_{a,b,c,d} [\mathsf{X}]_{i+a,j+b,c}, \tag{6.1.7}$$

burada d, H gizli temsillerinde çıktı kanalları dizinler. Sonraki evrişimli tabaka, girdi olarak üçüncü mertebeden bir tensör olan H'i almaya devam edecektir. Daha genel olarak, (6.1.7), birden fazla kanal için bir evrişimli tabakanın tanımıdır; burada V, katmanın bir çekirdeği veya filtresidir.

Halen ele almamız gereken birçok işlem var. Örneğin, tüm gizli temsilleri tek bir çıktıda nasıl birleştireceğimizi bulmamız gerekiyor, örn. resimde *herhangi bir yerde* Waldo var mı? Ayrıca işleri verimli bir şekilde nasıl hesaplayacağımıza, birden fazla katmanı nasıl birleştireceğimize, uygun etkinleştirme işlevlerine ve pratikte etkili ağlar oluşturmak için makul tasarım seçimlerinin nasıl yapılacağına karar vermeliyiz. Bölümün geri kalanında bu sorunlara eğiliyoruz.

6.1.5 Özet

- İmgelerdeki çeviri değişmezliği, bir imgenin tüm yamalarının aynı şekilde işleneceğini ima eder.
- Yerellik, karşılık gelen gizli temsilleri hesaplamak için yalnızca küçük bir piksel komşuluğunun kullanılacağı anlamına gelir.
- İmge işlemede, evrişimli katmanlar genellikle tam bağlı katmanlardan çok daha az parametre gerektirir.
- CNN'ler, evrişimli katmanlar içeren özel bir sinir ağları ailesidir.
- Girdi ve çıktıdaki kanallar, modelimizin bir imgenin her uzaysal konumda birden çok yönünü yakalamasına olanak tanır.

6.1.6 Alıştırmalar

1. Evrişim çekirdeğinin boyutu $\Delta = 0$ olduğunu varsayalım. Bu durumda, evrişim çekirdeğinin her kanal kümesi için bağımsız olarak bir MLP uyguladığını gösterin.
2. Çeviri değişmezliği neden iyi bir fikir olmayabilir?
3. Bir imgenin sınırındaki piksel konumlarına karşılık gelen gizli temsillerine nasıl muamele edileceğine karar verirken hangi sorunlarla baş etmeliyiz?
4. Ses için benzer bir evrişimli katman tanımlayın.
5. Evrişimli katmanların metin verileri için de geçerli olabileceğini düşünüyor musunuz? Neden ya da neden olmasın?
6. Bunu kanıtlayın: $f * g = g * f$.

Tartışmalar[84]

6.2 İmgeler için Evrişimler

Artık evrişimli katmanların teoride nasıl çalıştığını anladığımıza göre, uygulamada nasıl çalıştıklarını görmeye hazırız. İmge verilerindeki yapıyı keşfetmek için etkili mimariler olarak evrişimli sinir ağlarının motivasyonunu temel alarak imgeler üzerinde çalışan örneğimize bağlı kalıyoruz.

6.2.1 Çapraz Korelasyon İşlemi

Kesin olarak konuşursak, evrişimli katmanların yanlış adlandırıldığını hatırlayın, çünkü ifade ettikleri işlemler daha doğru bir şekilde çapraz korelasyon olarak tanımlanabilir. Section 6.1 içindeki evrişimli tabaka açıklamalarına göre, böyle bir katmanda, bir girdi tensör ve bir çekirdek tensör çapraz korelasyon işlemi yoluyla bir çıktı tensörü üretmek için birleştirilir.

Şimdilik kanalları yok sayalım ve bunun iki boyutlu veri ve gizli temsillerle nasıl çalıştığını görelim. Fig. 6.2.1 içinde, girdi, yüksekliği 3 ve genişliği 3 olan iki boyutlu bir tensördür. Tensörün şeklini 3×3 veya $(3, 3)$ diye belirtiyoruz. Çekirdeğin yüksekliğinin ve genişliğinin her ikisi de 2'dir. *Çekirdek penceresinin* (veya *evrişim penceresi*) şekli çekirdeğin yüksekliği ve genişliği ile verilir (burada 2×2'dir).

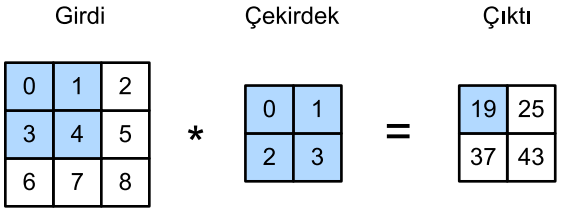


Fig. 6.2.1: İki boyutlu çapraz korelasyon işlemi. Gölgeli kısımlar, çıktı hesaplaması için kullanılan girdi ve çekirdek tensör elemanlarının yanı sıra ilk çıktı elemanıdır: $0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$.

İki boyutlu çapraz korelasyon işleminde, girdi tensörünün sol üst köşesinde konumlandırılmış evrişim penceresi ile başlar ve hem soldan sağa hem de yukarıdan aşağıya doğru, girdi tensörü

[84] https://discuss.d2l.ai/t/64

boyunca kaydırırız. Evrişim penceresi belirli bir konuma kaydırıldığında, bu pencerede bulunan girdi alt-tensör ve çekirdek tensör eleman yönlü olarak çarpılır ve elde edilen tensör tek bir sayıl (skaler) değer oluşturacak şekilde toplanır. Bu sonuç, çıktı tensörünün ilgili konumdaki değerini verir. Burada, çıktı tensörünün yüksekliği ve genişliği 2'dir ve dört eleman iki boyutlu çapraz korelasyon işleminden türetilmiştir:

$$\begin{aligned}0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19,\\ 1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 = 25,\\ 3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 = 37,\\ 4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 = 43.\end{aligned} \tag{6.2.1}$$

Her eksen boyunca, çıktı boyutunun girdi boyutundan biraz daha küçük olduğunu unutmayın. Çekirdeğin genişliği ve yüksekliği birden büyük olduğundan, çekirdeğin tamamen imge içine sığdığı konumlar için çapraz korelasyonu düzgün bir şekilde hesaplayabiliriz, çıktı boyutu $n_h \times n_w$ eksi evrişim çekirdeğinin boyutu $k_h \times k_w$ ile verilir.

$$(n_h - k_h + 1) \times (n_w - k_w + 1). \tag{6.2.2}$$

Evrişim çekirdeğini imge boyunca "kaydırmak" için yeterli alana ihtiyacımız olduğu durum budur. Daha sonra, imgeyi sınırının etrafında sıfırlarla doldurarak boyutun değişmeden nasıl tutulacağını göreceğiz, böylece çekirdeği kaydırmak için yeterli alanımız olacak. Daha sonra, bir girdi tensör `X` ve bir çekirdek tensör `K` kabul eden ve bir çıktı tensör `Y` döndüren `corr2d` işlevinde bu işlemi uyguluyoruz.

```
import torch
from torch import nn
from d2l import torch as d2l
```

```
def corr2d(X, K):  #@save
    """2 boyutlu çapraz korelasyonu hesapla."""
    h, w = K.shape
    Y = torch.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j] = (X[i:i + h, j:j + w] * K).sum()
    return Y
```

İki boyutlu çapraz korelasyon işleminin yukarıdaki uygulamasının çıktısını doğrulamak için Fig. 6.2.1 içinden girdi tensörünü `X`'i ve çekirdek tensörünü `K`'yı inşa edebiliriz.

```
X = torch.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
K = torch.tensor([[0.0, 1.0], [2.0, 3.0]])
corr2d(X, K)
```

```
tensor([[19., 25.],
        [37., 43.]])
```

6.2.2 Evrişimli Katmanlar

Bir evrişimli katman, girdi ve çekirdeği çapraz-ilişkilendirir (cross-correlate) ve çıktı üretmek için bir skaler ek girdi ekler. Bir evrişimli tabakanın iki parametresi çekirdek ve skaler ek girdidir. Modelleri evrişimli katmanlara göre eğitirken, tam bağlı bir katmanda olduğu gibi, çekirdekleri genelde rastgele olarak ilkleriz.

Yukarıda tanımlanan `corr2d` işlevine dayanan iki boyutlu bir evrişimli katmanı uygulamaya hazırız. `__init__` kurucu işlevinde, iki model parametresi olarak `weight` ve `bias`'i beyan ederiz. İleri yayma işlevi `corr2d` işlevini çağırır ve ek girdiyi ekler.

```
class Conv2D(nn.Module):
    def __init__(self, kernel_size):
        super().__init__()
        self.weight = nn.Parameter(torch.rand(kernel_size))
        self.bias = nn.Parameter(torch.zeros(1))

    def forward(self, x):
        return corr2d(x, self.weight) + self.bias
```

$h \times w$ evrişiminde veya $h \times w$ evrişim çekirdeğinde, evrişim çekirdeğinin yüksekliği ve genişliği sırasıyla h ve w'dir. Ayrıca $h \times w$ evrişim çekirdeğine sahip bir evrişimli tabakaya kısaca $h \times w$ evrişim tabaka diye atıfta bulunuyoruz.

6.2.3 İmgelerde Nesne Kenarını Algılama

Evrişimli bir katmanın basit bir uygulamasını ayrıştırmak için biraz zaman ayıralım: Piksel değişiminin yerini bularak bir imgedeki nesnenin kenarını tespit etme. İlk olarak, 6×8 piksellik bir "imge" oluşturuyoruz. Orta dört sütun siyah (0) ve geri kalanı beyaz (1) olsun.

```
X = torch.ones((6, 8))
X[:, 2:6] = 0
X
```

```
tensor([[1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.]])
```

Daha sonra, 1 yüksekliğinde ve 2 genişliğinde bir çekirdek `K` inşa ediyoruz. Girdi ile çapraz korelasyon işlemini gerçekleştirdiğimizde, yatay olarak bitişik elemanlar aynıysa, çıktı 0'dır. Aksi takdirde, çıktı sıfır değildir.

```
K = torch.tensor([[1.0, -1.0]])
```

`X` (girdimiz) ve `K` (çekirdeğimiz) argümanlarıyla çapraz korelasyon işlemini gerçekleştirmeye hazırız. Gördüğünüz gibi, beyazdan siyaha kenar için 1 ve siyahtan beyaza kenar için -1 tespit ediyoruz. Diğer tüm çıktılar 0 değerini alır.

```
Y = corr2d(X, K)
Y
```

```
tensor([[ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.]])
```

Artık çekirdeği devrik imgeye uygulayabiliriz. Beklendiği gibi, yok oluyor. Çekirdek `K` yalnızca dikey kenarları algılar.

```
corr2d(X.t(), K)
```

```
tensor([[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]])
```

6.2.4 Bir Çekirdeği Öğrenme

Sonlu farklar `[1, -1]` ile bir kenar dedektörü tasarlamak, aradığımız şeyin tam olarak ne olduğunu biliyorsak temiz olur. Ancak, daha büyük çekirdeklere baktığımızda ve ardışık evrişim katmanlarını göz önünde bulundurduğumuzda, her filtrenin manuel olarak ne yapması gerektiğini tam olarak belirtmek imkansız olabilir.

Şimdi `X`'ten `Y`'yi oluşturan çekirdeği yalnızca girdi-çıktı çiftlerine bakarak öğrenip öğrenemeyeceğimizi görelim. Önce bir evrişimli tabaka oluşturup çekirdeğini rastgele bir tensör olarak ilkletiriz. Daha sonra, her yinelemede, `Y`'yi evrişimli tabakanın çıktısıyla karşılaştırmak için kare hatayı kullanacağız. Daha sonra çekirdeği güncellemek için gradyanı hesaplayabiliriz. Basitlik uğruna, aşağıda iki boyutlu evrişimli katmanlar için yerleşik sınıfı kullanıyoruz ve ek girdiyi görmezden geliyoruz.

```
# 1 çıktı kanallı ve (1, 2) şekilli çekirdekli iki boyutlu bir evrişim katmanı
# oluşturun. Basitlik adına, burada ek girdiyi görmezden geliyoruz.
conv2d = nn.Conv2d(1,1, kernel_size=(1, 2), bias=False)

# İki boyutlu evrişimli katman, (örnek, kanal, yükseklik, genişlik) biçiminde
# dört boyutlu girdi ve çıktı kullanır; burada toplu iş boyutunun (gruptaki örnek sayısı)
# ve kanal sayısının her ikisi de 1'dir.
X = X.reshape((1, 1, 6, 8))
Y = Y.reshape((1, 1, 6, 7))
lr = 3e-2  # Öğrenme oranı

for i in range(10):
```

(continues on next page)

(continued from previous page)

```
    Y_hat = conv2d(X)
    l = (Y_hat - Y) ** 2
    conv2d.zero_grad()
    l.sum().backward()
    # Çekirdeği güncelle
    conv2d.weight.data[:] -= lr * conv2d.weight.grad
    if (i + 1) % 2 == 0:
        print(f'donem {i + 1}, loss {l.sum():.3f}')
```

```
donem 2, loss 10.132
donem 4, loss 1.769
donem 6, loss 0.325
donem 8, loss 0.066
donem 10, loss 0.016
```

Hatanın 10 yinelemeden sonra küçük bir değere düştüğünü fark ediniz. Şimdi öğrendiğimiz çekirdek tensörüne bir göz atacağız.

```
conv2d.weight.data.reshape((1, 2))
```

```
tensor([[ 0.9744, -0.9929]])
```

Gerçekten de, öğrenilen çekirdek tensör, daha önce tanımladığımız çekirdek tensörüne `K`'e oldukça yakındır.

6.2.5 Çapraz Korelasyon ve Evrişim

Çapraz korelasyon ve evrişim işlemleri arasındaki ilişkilendirmeler için Section 6.1 içindeki gözlemlerimizi hatırlayın. Burada iki boyutlu evrişimli katmanları düşünmeye devam edelim. Bu tür katmanlar çapraz korelasyon yerine (6.1.6) içinde tanımlandığı gibi tam evrişim işlemlerini gerçekleştirirse ne olur? Tam *evrişim* işleminin çıktısını elde etmek için, iki boyutlu çekirdek tensörünü hem yatay hem de dikey olarak çevirmemiz ve daha sonra girdi tensörüyle *çapraz korelasyon* işlemini gerçekleştirmemiz gerekir.

Çekirdekler derin öğrenmede verilerden öğrenildiğinden, bu tür katmanlar tam evrişim işlemlerini veya çapraz korelasyon işlemlerini gerçekleştirse de, evrişimli katmanların çıktılarının etkilenmeden kalması dikkat çekicidir.

Bunu göstermek için, bir evrişimli katmanın *çapraz korelasyon* gerçekleştirdiğini ve çekirdeği Fig. 6.2.1 içinde öğrendiğini varsayalım, burada $\mathbf{K}$ matris olarak ifade ediliyor. Bu katman yerine tam *evrişim* gerçekleştirdiğinde, diğer koşulların değişmeden kaldığını varsayarsak, $\mathbf{K}'$ hem yatay hem de dikey olarak çevrildikten sonra $\mathbf{K}$ ile aynı olacaktır. Yani, evrişimli tabaka Fig. 6.2.1 ve $\mathbf{K}'$'deki girdi için tam *evrişim* gerçekleştirdiğinde, Fig. 6.2.1 içinde gösterilen aynı çıktı (girdi ve $\mathbf{K}$'nin çapraz korelasyon) elde edilecektir.

Derin öğrenme yazınındaki standart terminolojiye uygun olarak, çapraz korelasyon işlemine bir evrişim olarak atıfta bulunmaya devam edeceğiz, katı bir şekilde konuşursak da, biraz farklı olsa da. Ayrıca, bir katman temsilini veya bir evrişim çekirdeğini temsil eden herhangi bir tensörün girdisini (veya bileşenini) ifade etmek için *eleman (öğe)* terimini kullanıyoruz.

6.2.6 Öznitelik Eşleme ve Alım Alanı (Receptive Field)

Section 6.1.4 içinde açıklandığı gibi, Fig. 6.2.1 içindeki evrişimli katman çıktısı bazen *öznitelik eşleme (feature mapping)* olarak adlandırılır, çünkü uzamsal boyutlarda (örn. genişlik ve yükseklik) sonraki katmana öğrenilmiş temsiller (öznitelikler) olarak kabul edilebilir. CNN'lerde, herhangi tabakanın herhangi bir elemanı x için, *alım alanı*, ileri yayma sırasında x'nin hesaplanmasını etkileyebilecek tüm elemanları (önceki katmanlardan) ifade eder. Alım alanın girdinin gerçek boyutundan daha büyük olabileceğini unutmayın.

Alım alanını açıklamak için Fig. 6.2.1 içeriğini kullanmaya devam edelim. 2×2 evrişim çekirdeği göz önüne alındığında, gölgeli çıktı elemanının alım alanı (19 değeri) girdinin gölgeli kısmındaki dört öğedir. Şimdi 2×2 çıktısını $\mathbf{Y}$ olarak ifade edelim ve $\mathbf{Y}$'yi girdi alıp tek z eleman çıktısı veren 2×2 evrişimli tabakalı daha derin bir CNN düşünelim. Bu durumda, $\mathbf{Y}$'deki z'nin alım alanı $\mathbf{Y}$'nin dört öğesini içerirken, girdideki alım alanı dokuz girdi elemanını içerir. Böylece, bir öznitelik eşlemedeki herhangi bir elemanın daha geniş bir alan üzerindeki girdi özelliklerini algılamak için daha büyük bir alım alanına ihtiyacı olduğunda, daha derin bir ağ kurabiliriz.

6.2.7 Özet

- İki boyutlu bir evrişimli tabakanın çekirdek hesaplaması, iki boyutlu bir çapraz korelasyon işlemidir. En basit haliyle, bu iki boyutlu girdi verisi ve çekirdek üzerinde çapraz korelasyon işlemi gerçekleştirir ve sonra bir ek girdi ekler.
- İmgelerdeki kenarları tespit etmek için bir çekirdek tasarlayabiliriz.
- Çekirdeğin parametrelerini verilerden öğrenebiliriz.
- Verilerden öğrenilen çekirdekler ile, evrişimli katmanların çıktıları, bu tür katmanlarda gerçekleştirilen işlemlerden bağımsız kalarak (tam evrişim veya çapraz korelasyon) etkilenmez.
- Bir öznitelik eşlemedeki herhangi bir öğe, girdideki daha geniş öznitelikleri algılamak için daha büyük bir alım alanına ihtiyaç duyduğunda, daha derin bir ağ düşünülebilir.

6.2.8 Alıştırmalar

1. Çapraz kenarlı bir `X` imgesi oluşturun.
    1. Çekirdek `K`'yi bu bölümde uygularsanız ne olur?
    2. `X` devrik olursa ne olur?
    3. `K` devrik olursa ne olur?
2. Oluşturduğumuz `Conv2D` sınıfının gradyanını otomatik olarak bulmaya çalıştığınızda ne tür bir hata mesajı görürsunuz?
3. Girdi ve çekirdek tensörlerini değiştirerek çapraz korelasyon işlemini nasıl bir matris çarpımı olarak temsil edebilirsiniz?
4. Bazı çekirdekleri manuel olarak tasarlayın.
    1. İkinci türev için bir çekirdeğin biçimi nedir?
    2. İntegral için bir çekirdek nedir?

3. d derece türevi elde etmek için bir çekirdeğin minimum boyutu nedir?

Tartışmalar[85]

6.3 Dolgu ve Uzun Adımlar

Önceki Fig. 6.2.1 örneğinde, girdimizin hem yüksekliği hem de genişliği 3 idi ve evrişim çekirdeğimizin hem yüksekliği hem de genişliği 2 idi, bu da 2×2 boyutlu bir çıktı gösterimi sağladı. Section 6.2 içinde genelleştirdiğimiz gibi, girdinin şeklinin $n_h \times n_w$ olduğunu ve evrişim çekirdeğinin şeklinin $k_h \times k_w$ olduğunu varsayarsak, çıktının şekli $(n_h - k_h + 1) \times (n_w - k_w + 1)$ olacaktır. Bu nedenle, evrişimli tabakanın çıktı şekli, girdinin şekli ve evrişim çekirdeğinin şekli ile belirlenir.

Bazı durumlarda, çıktının boyutunu etkileyen dolgu ve uzun adımlı evrişimler dahil olmak üzere teknikleri dahil ediyoruz.Motivasyon olarak, çekirdeklerin genellikle 1'den büyük genişliğe ve yüksekliğe sahip olduğundan, birçok ardışık evrişim uyguladıktan sonra, girdimizden önemli ölçüde daha küçük çıktılar elde etme eğiliminde olduğumuzu unutmayın. 240×240 piksel imgeyle başlarsak, 10 tane 5×5 evrişim katmanı imgeyi 200×200 piksele indirir, imgenin $\%30$'unu keserek atar ve orijinal imgenin sınırları hakkındaki ilginç bilgileri yok eder. *Dolgu*, bu sorunu ele almada en popüler araçtır.

Diğer durumlarda, örneğin orijinal girdi çözünürlüğünün kullanışsız olduğunu görürsek, boyutsallığı büyük ölçüde azaltmak isteyebiliriz. *Uzun adımlı evrişimler*, bu örneklerde yardımcı olabilecek popüler bir tekniktir.

6.3.1 Dolgu

Yukarıda açıklandığı gibi, evrişimli katmanları uygularken zor bir sorun, imgemizin çevresindeki pikselleri kaybetme eğiliminde olmamızdır. Tipik olarak küçük çekirdekler kullandığımızdan, herhangi bir evrişim için, yalnızca birkaç piksel kaybedebiliriz, ancak birçok ardışık evrişimli katman uyguladığımız için bu kayıplar toplanarak artacaktır. Bu soruna basit bir çözüm, girdi imgemizin sınırına ekstra dolgu pikselleri eklemek, böylece imgenin etkin boyutunu arttırmaktır. Tipik olarak, ekstra piksellerin değerlerini sıfıra ayarlarız. Fig. 6.3.1 içinde, 3×3'lük girdiyi doldurarak boyutunu 5×5'e yükseltiyoruz. Karşılık gelen çıktı daha sonra bir 4×4 matrisine yükselir. Gölgeli kısımlar, çıktı hesaplamasında kullanılan girdi ve çekirdek tensör elemanlarının yanı sıra ilk çıktı elemanıdır: $0 \times 0 + 0 \times 1 + 0 \times 2 + 0 \times 3 = 0$.

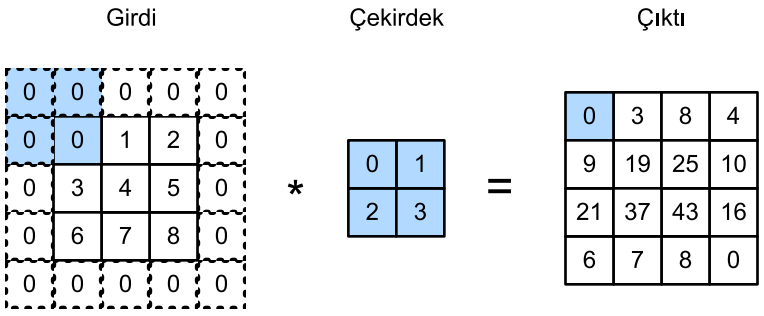


Fig. 6.3.1: Dolgu ile iki boyutlu çapraz korelasyon.

[85] https://discuss.d2l.ai/t/66

Genel olarak, toplam p_h satırlı dolgu (kabaca yarısı üstte ve altta yarısı) ve toplam p_w sütunlu dolgu (kabaca yarısı solda ve sağda yarısı) eklersek, çıktının şekli şöyle olur:

$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1). \tag{6.3.1}$$

Bu, çıktının yüksekliğinin ve genişliğinin sırasıyla p_h ve p_w artacağı anlamına gelir.

Birçok durumda, girdiye ve çıktıya aynı yüksekliği ve genişliği vermek için $p_h = k_h - 1$ ve $p_w = k_w - 1$'i ayarlamak isteyeceğiz. Bu, ağ oluştururken her katmanın çıktı şeklini tahmin etmeyi kolaylaştıracaktır. k_h'nın burada tek sayı olduğunu varsayarsak, yüksekliğin her iki tarafında $p_h/2$ satır dolgu olacak. Eğer k_h çift ise, bir olasılık girdinin üstünde $\lfloor p_h/2 \rfloor$ ve altında $\lceil p_h/2 \rceil$ satır dolgu olmasıdır. Genişliğin her iki tarafını da aynı şekilde dolduracağız.

CNN'ler genellikle 1, 3, 5 veya 7 gibi tek yükseklik ve genişlik değerlerine sahip evrişim çekirdeklerini kullanır. Tek sayı çekirdek boyutlarını seçmek, üstte ve altta aynı sayıda satır ve solda ve sağda aynı sayıda sütun ile doldururken uzamsal boyutsallığı koruyabilmemiz avantajına sahiptir.

Dahası, bu boyutsallığı hassas bir şekilde korumak için tek sayı boyutlu çekirdekler ve dolgu kullanma uygulaması rutinsel faydalar sağlar. Herhangi bir iki boyutlu tensör `X` için, çekirdeğin boyutu tek olduğunda ve her iki taraftaki dolgu satırları ve sütunlarının sayısı aynı olduğunda, girdiyle aynı yüksekliğe ve genişliğe sahip bir çıktı ürettiğini, `Y[i, j]` çıktısının girdi ve `X[i, j]` ile ortalanmış pencereli evrişim çekirdeğinin çapraz korelasyonu ile hesaplandığını biliyoruz.

Aşağıdaki örnekte, yüksekliği ve genişliği 3 olan iki boyutlu bir evrişimli tabaka oluşturuyoruz ve her tarafa 1 piksel dolgu uyguluyoruz. Yüksekliği ve genişliği 8 olan bir girdi göz önüne alındığında, çıktının yüksekliğinin ve genişliğinin de 8 olduğunu buluruz.

```
import torch
from torch import nn


# Kolaylık sağlamak için evrişimli katmanı hesaplamada bir fonksiyon tanımladık.
# Bu işlev, evrişimli katman ağırlıklarını başlatır
# ve girdi ve çıktıda karşılık gelen boyutsallık yükseltmelerini
# ve azaltmalarını gerçekleştirir
def comp_conv2d(conv2d, X):
    # Burada (1, 1) grup boyutunun ve kanal sayısının her
    # ikisinin de 1 olduğunu gösterir.
    X = X.reshape((1, 1) + X.shape)
    Y = conv2d(X)
    # Bizi ilgilendirmeyen ilk iki boyutu hariç tutun: Örnekler
    # ve kanallar
    return Y.reshape(Y.shape[2:])
# Burada her iki tarafta 1 satır veya sütunun doldurulduğunu unutmayın,
# bu nedenle toplam 2 satır veya sütun eklidir.
conv2d = nn.Conv2d(1, 1, kernel_size=3, padding=1)
X = torch.rand(size=(8, 8))
comp_conv2d(conv2d, X).shape
```

```
torch.Size([8, 8])
```

Evrişim çekirdeğinin yüksekliği ve genişliği farklı olduğunda, yükseklik ve genişlik için farklı dolgu sayıları ayarlayarak çıktı ve girdinin aynı yükseklik ve genişliğe sahip olmasını sağlayabiliriz.

```
# Burada yüksekliği 5 ve genişliği 3 olan bir evrişim çekirdeği
# kullanıyoruz. Yüksekliğin ve genişliğin her iki tarafındaki
# dolgu sayıları sırasıyla 2 ve 1'dir.
conv2d = nn.Conv2d(1, 1, kernel_size=(5, 3), padding=(2, 1))
comp_conv2d(conv2d, X).shape
```

```
torch.Size([8, 8])
```

6.3.2 Uzun Adım

Çapraz korelasyonu hesaplarken, girdi tensörünün sol üst köşesinden evrişim penceresi ile başlar ve ardından hem aşağı hem de sağa doğru tüm konumların üzerinden kaydırırız. Önceki örneklerde, varsayılan olarak bir öğeyi aynı anda kaydırırız. Ancak, bazen, ya hesaplama verimliliği için ya da örnek seyreltmek (downsampling) istediğimiz için, penceremizi aynı anda birden fazla öğe üzerinden hareket ettirerek ara konumları atlıyoruz.

Kayma başına geçilen satır ve sütun sayısını *uzun adım (stride)* diye adlandırırız. Şimdiye kadar, hem yükseklik hem de genişlik için 1'lik adımlar kullandık. Bazen, daha büyük bir adım kullanmak isteyebiliriz. Fig. 6.3.2, dikey 3 ve yatay 2 adımlı iki boyutlu bir çapraz korelasyon işlemi gösterir. Gölgeli kısımlar çıktı elemanlarının yanı sıra çıktı hesaplaması için kullanılan girdi ve çekirdek tensör elemanlarıdır: $0 \times 0 + 0 \times 1 + 1 \times 2 + 2 \times 3 = 8$, $0 \times 0 + 6 \times 1 + 0 \times 2 + 0 \times 3 = 6$. İlk sütunun ikinci elemanı çıkıldığında, evrişim penceresinin üç sıra aşağı kaydığını görebiliriz. İlk satırın ikinci öğesi işlenince evrişim penceresi sağa iki sütun kaydırılıyor. Evrişim penceresi girdide sağa iki sütun kaymaya devam ettiğinde, girdi öğesi pencereyi dolduramayacağı için (başka bir dolgu sütunu eklemediğimiz sürece) çıktı yoktur.

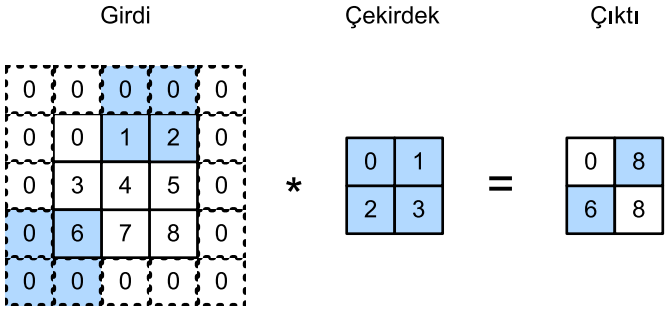


Fig. 6.3.2: Yükseklik ve genişlik için sırasıyla 3'lük ve 2'lik uzun adımlarla çapraz korelasyon.

Genel olarak, yükseklik için s_h adım ve genişlik için s_w adım olduğunda, çıktının şekli:

$$\lfloor (n_h - k_h + p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w)/s_w \rfloor. \tag{6.3.2}$$

$p_h = k_h - 1$ ve $p_w = k_w - 1$'i ayarlarsak, çıktı şekli $\lfloor (n_h + s_h - 1)/s_h \rfloor \times \lfloor (n_w + s_w - 1)/s_w \rfloor$'e sadeleşir. Bir adım daha ileri gidersek, eğer girdi yüksekliği ve genişliği, yükseklik ve genişlik üzerindeki uzun adımlarla bölünebilirse, çıktının şekli $(n_h/s_h) \times (n_w/s_w)$ olacaktır.

Aşağıda, hem yükseklik hem de genişlik üzerindeki uzun adımları 2'ye ayarladık, böylece girdinin yüksekliğini ve genişliğini yarıya indirdik.

```
conv2d = nn.Conv2d(1, 1, kernel_size=3, padding=1, stride=2)
comp_conv2d(conv2d, X).shape
```

```
torch.Size([4, 4])
```

Daha sonra, biraz daha karmaşık bir örneğe bakacağız.

```
conv2d = nn.Conv2d(1, 1, kernel_size=(3, 5), padding=(0, 1), stride=(3, 4))
comp_conv2d(conv2d, X).shape
```

```
torch.Size([2, 2])
```

Kısa olması amacıyla, girdi yüksekliğinin ve genişliğinin her iki tarafındaki dolgu sayısı sırasıyla p_h ve p_w olduğunda, dolguya (p_h, p_w) diyoruz. Özellikle, $p_h = p_w = p$ olduğunda, dolgu p'dir. Yükseklik ve genişlik üzerindeki uzun adımlar sırasıyla s_h ve s_w olduğunda, uzun adıma (s_h, s_w) diyoruz. Özellikle, $s_h = s_w = s$ olduğunda, uzun adım s'dir. Varsayılan olarak, dolgu 0 ve uzun adım 1'dir. Uygulamada, nadiren homojen olmayan adımlar veya dolgu kullanırız, yani genellikle $p_h = p_w$ ve $s_h = s_w$'dir.

6.3.3 Özet

- Dolgu, çıktının yüksekliğini ve genişliğini artırabilir. Bu, genellikle çıktıya girdi ile aynı yükseklik ve genişlik vermek için kullanılır.
- Uzun adım, çıktının çözünürlüğünü azaltabilir, örneğin çıktının yüksekliğini ve genişliğini, girdinin yüksekliğinin ve genişliğinin yalnızca $1/n$'sine düşürür (n, 1'den büyük bir tamsayıdır).
- Dolgu ve uzun adım, verilerin boyutsallığını etkin bir şekilde ayarlamak için kullanılabilir.

6.3.4 Alıştırmalar

1. Bu bölümdeki son örnek için, deneysel sonuçla tutarlı olup olmadığını görmek amacıyla çıktı şeklini matematik kullanarak hesaplayın.
2. Bu bölümdeki deneylerde diğer dolgu ve uzun adım kombinasyonlarını deneyin.
3. Ses sinyalleri için, 2'lik bir uzun adım neye karşılık gelir?
4. 1'den büyük bir uzun adımın hesaplamalı faydaları nelerdir?

Tartışmalar[86]

[86] https://discuss.d2l.ai/t/68

6.4 Çoklu Girdi ve Çoklu Çıktı Kanalları

Her imgeyi Section 6.1.4 içinde birden fazla kanal (örneğin, renkli görüntüler kırmızı, yeşil ve mavi miktarını belirtmek için standart RGB kanallarına sahiptir) için evrişimli katmanlar tarif etmişken, şimdiye kadar, tüm sayısal örneklerimizi sadece tek bir girdi ve tek bir çıktı kanalı kullanarak basitleştirdik. Bu, girdilerimizi, evrişim çekirdeklerini ve çıktılarımızı iki boyutlu tensörler olarak düşünmemizi sağladı.

Bu karışıma kanal eklediğimizde, girdilerimiz ve gizli temsillerimiz üç boyutlu tensörler haline gelir. Örneğin, her RGB girdi imgesi $3 \times h \times w$ şeklindedir. 3 uzunluklu bu eksene, *kanal* boyutu olarak atıfta bulunuyoruz. Bu bölümde, birden fazla girdi ve birden fazla çıktı kanalı içeren evrişim çekirdeklerine daha derin bir bakış atacağız.

6.4.1 Çoklu Girdi Kanalları

Girdi verileri birden çok kanal içerdiğinde, girdi verileriyle aynı sayıda girdi kanalı içeren bir evrişim çekirdeği oluşturmamız gerekir, böylece girdi verisi ile çapraz korelasyon gerçekleştirebilir. Girdi verisi için kanal sayısının c_i olduğunu varsayarsak, evrişim çekirdeğinin girdi kanallarının sayısının da c_i olması gerekir. Eğer evrişim çekirdeğimizin pencere şekli $k_h \times k_w$ ise, o zaman $c_i = 1$ olduğunda, evrişim çekirdeğimizi $k_h \times k_w$ şeklindeki iki boyutlu bir tensör olarak düşünebiliriz.

Ancak, $c_i > 1$ olduğunda, *her* girdi kanalı için $k_h \times k_w$ şeklindeki bir tensör içeren bir çekirdeğe ihtiyacımız var. Bu c_i tensörlerin birleştirilmesi, $c_i \times k_h \times k_w$ şeklindeki bir evrişim çekirdeği verir. Girdi ve evrişim çekirdeğinin her biri c_i kanallara sahip olduğundan, her kanal için iki boyutlu tensör girdinin ve iki boyutlu tensör evrişim çekirdeğinin üzerinde çapraz korelasyon işlemi gerçekleştirebiliriz, c_i sonuçlarını da birlikte toplayarak (kanallar üzerinde toplanarak) iki boyutlu tensör elde ederiz. Bu, çok kanallı girdi ve çok girdi kanallı evrişim çekirdeği arasındaki iki boyutlu çapraz korelasyonun sonucudur.

Fig. 6.4.1 içinde, iki girdi kanalı ile iki boyutlu çapraz korelasyon örneğini gösteriyoruz. Gölgeli kısımlar, çıktı hesaplaması için kullanılan girdi ve çekirdek tensör elemanlarının yanı sıra ilk çıktı öğesidir: $(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56$.

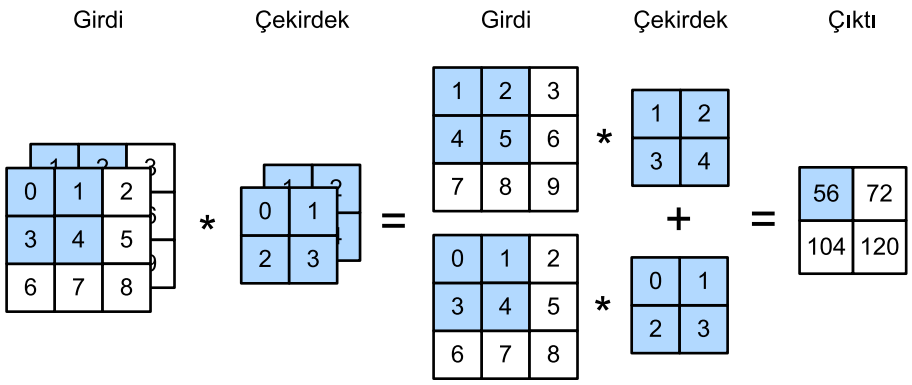


Fig. 6.4.1: 2 girdi kanalı ile çapraz korelasyon hesaplaması.

Burada neler olduğunu gerçekten anladığımızdan emin olmak için birden fazla girdi kanalı ile çapraz korelasyon işlemlerini kendimiz uygulayabiliriz. Yaptığımız tek şeyin kanal başına bir çapraz korelasyon işlemi gerçekleştirmek ve ardından sonuçları toplamak olduğuna dikkat edin.

```
import torch
from d2l import torch as d2l
```

```
def corr2d_multi_in(X, K):
    # İlk olarak, `X` ve `K`'nin 0. boyutunu (kanal boyutu) yineleyin.
    # Ardından, onları toplayın.
    return sum(d2l.corr2d(x, k) for x, k in zip(X, K))
```

Çapraz korelasyon işleminin çıktısını doğrulamak için Fig. 6.4.1 içindeki değerlere karşılık gelen `X` girdi tensörünü ve `K` çekirdek tensörünü inşa edebiliriz.

```
X = torch.tensor([[[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]],
                  [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]])
K = torch.tensor([[[0.0, 1.0], [2.0, 3.0]], [[1.0, 2.0], [3.0, 4.0]]])

corr2d_multi_in(X, K)
```

```
tensor([[ 56.,  72.],
        [104., 120.]])
```

6.4.2 Çoklu Çıktı Kanalları

Girdi kanallarının sayısı ne olursa olsun, şimdiye kadar hep elimizde bir çıktı kanalı kaldı. Bununla birlikte, Section 6.1.4 içinde tartıştığımız gibi, her katmanda birden fazla kanalın olması gerekli olduğu ortaya çıkıyor. En popüler sinir ağı mimarilerinde, sinir ağında daha yükseğe çıktıkça kanal boyutunu artırıyoruz, tipik olarak uzamsal çözünürlüğü daha büyük bir *kanal derinliği* için takas ederek örnek seyreltme yapıyoruz. Sezgisel olarak, her kanalı bazı farklı öznitelik kümesine yanıt veriyor gibi düşünebilirsiniz. Gerçeklik, bu sezginin en saf yorumlarından biraz daha karmaşıktır, çünkü temsiller bağımsız olarak öğrenilmemiştir, ancak ortaklaşa yararlı olmak için eniyilenmişlerdir. Bu nedenle, tek bir kanalın bir kenar dedektörünü öğrenmesi değil, kanal uzayındaki bazı yönlerin kenarları algılamaya karşılık gelmesi olabilir.

c_i ile c_o sırasıyla girdi ve çıktı kanallarının sayısını belirtsin ve k_h ile k_w çekirdeğin yüksekliği ve genişliği olsun. Birden fazla kanal içeren bir çıktı elde etmek amacıyla, *her* çıktı kanalı için $c_i \times k_h \times k_w$ şeklinde bir çekirdek tensör oluşturabiliriz. Onları çıktı kanalı boyutunda birleştiririz, böylece evrişim çekirdeğinin şekli $c_o \times c_i \times k_h \times k_w$ olur. Çapraz korelasyon işlemlerinde, her çıktı kanalındaki sonuç, çıktı kanalına karşılık gelen evrişim çekirdeğinden hesaplanır ve girdi tensöründeki tüm kanallardan girdi alır.

Aşağıda gösterildiği gibi birden fazla kanalın çıktısını hesaplamak için bir çapraz korelasyon fonksiyonu uyguluyoruz.

```
def corr2d_multi_in_out(X, K):
    # `K`'nın 0. boyutunu yineleyin ve her seferinde `X`
    # girdisiyle çapraz korelasyon işlemleri gerçekleştirin.
    # Tüm sonuçlar birlikte istiflenir.
    return torch.stack([corr2d_multi_in(X, k) for k in K], 0)
```

`K` çekirdek tensörünü `K+1` (K'deki her eleman için artı bir) ve `K+2` ile birleştirerek 3 çıktı kanalı içeren bir evrişim çekirdeği inşa ediyoruz.

```
K = torch.stack((K, K + 1, K + 2), 0)
K.shape
```

```
torch.Size([3, 2, 2, 2])
```

Aşağıda, `K` çekirdek tensör ile `X` girdi tensöründe çapraz korelasyon işlemleri gerçekleştiriyoruz. Şimdi çıktı 3 kanal içeriyor. İlk kanalın sonucu, önceki `X` girdi tensörünün ve çoklu girdi kanallı, tek çıktı kanallı çekirdeğin sonucu ile tutarlıdır.

```
corr2d_multi_in_out(X, K)
```

```
tensor([[[ 56.,  72.],
         [104., 120.]],

        [[ 76., 100.],
         [148., 172.]],

        [[ 96., 128.],
         [192., 224.]]])
```

6.4.3 1×1 Evrişimli Katman

İlk başta, bir 1×1 evrişimi, yani $k_h = k_w = 1$, çok mantıklı görünmüyor. Sonuçta, bir evrişim bitişik pikselleri ilişkilendirir. Bir 1×1 evrişimi besbelli öyle yapmıyor. Bununla birlikte, bazen karmaşık derin ağların tasarımlarına dahil olan popüler işlemlerdir. Aslında ne yaptığını biraz ayrıntılı olarak görelim.

En küçük pencere kullanıldığında, 1×1 evrişim, daha büyük evrişimli katmanların yükseklik ve genişlik boyutlarındaki bitişik elemanlar arasındaki etkileşimlerden oluşan desenleri tanıma yeteneğini kaybeder. 1×1 evrişiminin tek hesaplaması kanal boyutunda gerçekleşir.

Fig. 6.4.2, 3 girdi kanalı ve 2 çıktı kanalı ile 1×1 evrişim çekirdeği kullanılarak elde edilen çapraz korelasyon hesaplamalarını gösterir. Girdi ve çıktıların aynı yükseklik ve genişliğe sahip olduğunu unutmayın. Çıktıdaki her öğe, girdi görüntüsünün *aynı konumundaki* öğelerinin doğrusal bir kombinasyonundan türetilir. 1×1 evrişimli katmanın, c_i'ye karşılık gelen girdi değerlerini c_o çıktı değerlerine dönüştürmek için her piksel konumunda uygulanan tam bağlı bir katman oluşturduğunu düşünebilirsiniz. Bu hala bir evrişimli katman olduğundan, ağırlıklar piksel konumu boyunca bağlıdır. Böylece 1×1 evrişimsel tabaka $c_o \times c_i$ ağırlık gerektirir (artı ek girdi).

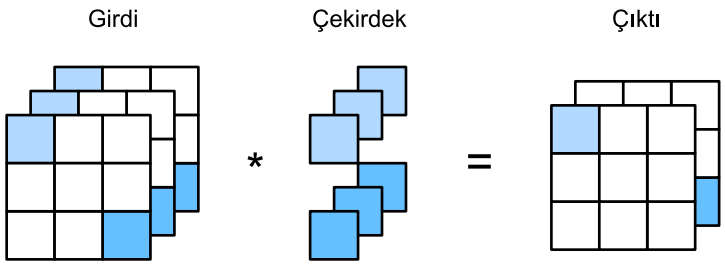


Fig. 6.4.2: Çapraz korelasyon hesaplaması, 3 girdi kanalı ve 2 çıktı kanalı olan 1×1 evrişim çekirdeğini kullanıyor. Girdi ve çıktı aynı yüksekliğe ve genişliğe sahiptir.

Bunun pratikte işe yarayıp yaramadığını kontrol edelim: Tam bağlı bir tabaka kullanarak 1×1 evrişimini uyguluyoruz. Düşünmemiz gereken tek şey, matris çarpımından önce ve sonra veri şekline bazı ayarlamalar yapmamız gerektiğidir.

```
def corr2d_multi_in_out_1x1(X, K):
    c_i, h, w = X.shape
    c_o = K.shape[0]
    X = X.reshape((c_i, h * w))
    K = K.reshape((c_o, c_i))
    # Tam bağlı katmanda matris çarpımı
    Y = torch.matmul(K, X)
    return Y.reshape((c_o, h, w))
```

1×1 evrişim gerçekleştirirken, yukarıdaki işlev daha önce uygulanan çapraz korelasyon fonksiyonu `corr2d_multi_in_out` ile eşdeğerdir. Bunu örnek verilerle kontrol edelim.

```
X = torch.normal(0, 1, (3, 3, 3))
K = torch.normal(0, 1, (2, 3, 1, 1))
```

```
Y1 = corr2d_multi_in_out_1x1(X, K)
Y2 = corr2d_multi_in_out(X, K)
assert float(torch.abs(Y1 - Y2).sum()) < 1e-6
```

6.4.4 Özet

- Evrişimli tabakanın model parametrelerini genişletmek için birden fazla kanal kullanılabilir.
- 1×1 evrişimli katman, piksel başına uygulandığında, tam bağlı katmana eşdeğerdir.
- 1×1 evrişimli katman genellikle ağ katmanları arasındaki kanal sayısını ayarlamak ve model karmaşıklığını kontrol etmek için kullanılır.

6.4.5 Alıştırmalar

1. Sırasıyla k_1 ve k_2 boyutunda iki evrişim çekirdeği olduğunu varsayalım (aralarında hiçbir doğrusal olmayanlık yoktur).
    1. İşlem sonucunun tek bir evrişim ile ifade edilebileceğini kanıtlayın.
    2. Eşdeğer tek evrişimin boyutsallığı nedir?
    3. Bunun tersi doğru mudur?
2. $c_i \times h \times w$ şeklinde bir girdi ve $c_o \times c_i \times k_h \times k_w$ şeklindeki bir evrişim çekirdeği, (p_h, p_w) dolgu ve (s_h, s_w) uzun adım varsayalım.
    1. İleri yayma için hesaplama maliyeti (çarpma ve eklemeler) nedir?
    2. Bellek ayak izi nedir?
    3. Geriye doğru hesaplama için bellek ayak izi nedir?
    4. Geri yayma için hesaplama maliyeti nedir?

3. c_i girdi kanallarının sayısını ve c_o çıktı kanallarının sayısını iki katına çıkarırsak, hesaplamaların sayısı hangi faktörle artar? Dolguyu ikiye katlarsak ne olur?
4. Bir evrişim çekirdeğinin yüksekliği ve genişliği $k_h = k_w = 1$ ise, ileri yaymanın hesaplama karmaşıklığı nedir?
5. Bu bölümün son örneğindeki değişkenler, `Y1` ve `Y2`, tamamen aynı mı? Neden?
6. Evrişim penceresi 1×1 olmadığında matris çarpımı kullanarak evrişimleri nasıl uygularsınız?

Tartışmalar[87]

6.5 Ortaklama

Çoğu zaman, imgeleri işledikçe, gizli temsillerimizin konumsal çözünürlüğünü yavaş yavaş azaltmak istiyoruz, böylece ağda ne kadar yükseğe çıkarsak, her gizli düğümün hassas olduğu alım alanı (girdide) o kadar büyük olur.

Genellikle esas görevimiz bize imge hakkında küresel bir soru sormaktadır, örn. *bir kedi içeriyor mu?* Bu nedenle tipik olarak son katmanımızın birimleri tüm girdiye karşı hassas olmalıdır. Bilgiyi kademeli olarak toplayarak, daha kaba eşlemeler üreterek, en sonunda küresel bir gösterimi öğrenme amacını gerçekleştiriyoruz ve bunu yaparken evrişimli ara katmanlardaki işlemlerin tüm avantajlarını tutuyoruz.

Dahası, kenarlar gibi (Section 6.2 içinde tartışıldığına benzer) alt seviye öznitelikleri tespit ederken, genellikle temsillerimizin yer değiştirmelerden etkilenmez olmasını isteriz. Örneğin, siyah beyaz arasında keskin gösterimli bir `X` imgesini alıp tüm imgeyi bir pikselle sağa kaydırırsak, yani `Z[i, j] = X[i, j + 1]`, yeni imgenin çıktısı çok farklı olabilir. Kenar bir piksel ile kaydırılmış olacaktır. Gerçekte, nesneler neredeyse hiç bir zaman aynı yerde olmaz. Aslında, bir tripod ve sabit bir nesneyle bile, deklanşörün hareketi nedeniyle kameranın titreşimi her şeyi bir piksel kaydırabilir (üst düzey kameralar bu sorunu gidermek için özel özelliklerle donatılmıştır).

Bu bölümde, evrişimli katmanların konuma duyarlılığını azaltmak ve gösterimleri uzaysal örnek seyreltmek gibi ikili amaçlara hizmet eden *ortaklama katmanları* tanıtılmaktadır.

6.5.1 Maksimum Ortaklama ve Ortalama Ortaklama

Ortaklama işlemcileri, evrişimli katmanlar gibi, sabit şekilli pencerenin (bazen *ortaklama penceresi* olarak da bilinir) geçtiği her konum için tek bir çıktı hesaplayarak, uzun adımına göre girdideki tüm bölgelere kaydırılan sabit şekilli bir pencereden oluşur. Bununla birlikte, evrişimli katmandaki girdi ve çekirdeklerin çapraz korelasyon hesaplamasının aksine, ortaklama katmanı hiçbir parametre içermez (*çekirdek* yoktur). Bunun yerine, ortaklama uygulayıcıları gerekircidir (deterministik) ve genellikle ortaklama penceresindeki öğelerin maksimum veya ortalama değerini hesaplar. Bu işlemler sırasıyla *maksimum ortaklama* (*kısaca ortaklama*) ve *ortalama ortaklama* olarak adlandırılır.

Her iki durumda da, çapraz korelasyon uygulayıcısında olduğu gibi, ortaklama penceresinin girdi tensörünün sol üstünden başlayarak girdi tensörünün soldan sağa ve yukarıdan aşağıya doğru kayması olarak düşünebiliriz. Ortaklama penceresinin vurduğu her konumda, maksimum veya

[87] https://discuss.d2l.ai/t/70

ortalama ortaklamanın kullanılmasına bağlı olarak, pencerede girdi alt tensörünün maksimum veya ortalama değerini hesaplar.

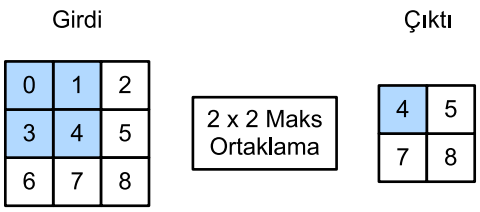


Fig. 6.5.1: 2×2 şeklinde bir ortaklama penceresi ile maksimum ortaklama. Gölgeli kısımlar, ilk çıktı elemanı ve çıktı hesaplaması için kullanılan girdi tensör elemanlarıdır: $\max(0, 1, 3, 4) = 4$.

Fig. 6.5.1 içindeki çıktı tensör 2'lik yüksekliğe ve 2'lik genişliğe sahiptir. Dört öğe, her ortaklama penceresindeki maksimum değerden türetilir:

$$\begin{aligned}\max(0, 1, 3, 4) &= 4,\\ \max(1, 2, 4, 5) &= 5,\\ \max(3, 4, 6, 7) &= 7,\\ \max(4, 5, 7, 8) &= 8.\end{aligned} \tag{6.5.1}$$

Ortaklama penceresi şeklindeki $p \times q$ ortaklama katmanına $p \times q$ ortaklama katmanı denir. Ortaklama işlemi $p \times q$ ortaklama olarak adlandırılır.

Bu bölümün başında belirtilen nesne kenarı algılama örneğine dönelim. Şimdi 2×2 maksimum ortaklama için girdi olarak evrişimli tabakanın çıktısını kullanacağız. Evrişimli katman girdisini `X` ve ortaklama katmanı çıktısını `Y` olarak düzenleyelim. `X[i, j]` ve `X[i, j + 1]` veya `X[i, j + 1]` ve `X[i, j + 2]` değerleri farklı olsa da olmasa da, ortaklama katmanı her zaman `Y[i, j] = 1` çıktısını verir. Yani, 2×2 maksimum ortaklama katmanını kullanarak, evrişimli katman tarafından tanınan desenin yükseklik veya genişlik olarak birden fazla eleman yine de hareket edip etmediğini tespit edebiliriz.

Aşağıdaki kodda, `pool2d` işlevinde ortaklama katmanının ileri yaymasını uyguluyoruz. Bu işlev Section 6.2 içindeki `corr2d` işlevine benzer. Ancak, burada çekirdeğimiz yok, çıktıyı girdideki her bölgenin maksimumu veya ortalaması olarak hesaplıyoruz.

```
import torch
from torch import nn
from d2l import torch as d2l
```

```
def pool2d(X, pool_size, mode='max'):
    p_h, p_w = pool_size
    Y = torch.zeros((X.shape[0] - p_h + 1, X.shape[1] - p_w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            if mode == 'max':
                Y[i, j] = X[i: i + p_h, j: j + p_w].max()
            elif mode == 'avg':
                Y[i, j] = X[i: i + p_h, j: j + p_w].mean()
    return Y
```

İki boyutlu maksimum ortaklama tabakasının çıktısını doğrulamak için Fig. 6.5.1 içinde girdi tensörü `X`'i inşa ediyoruz.

```
X = torch.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
pool2d(X, (2, 2))
```

```
tensor([[4., 5.],
        [7., 8.]])
```

Ayrıca, ortalama ortaklama katmanıyla da deney yapalım.

```
pool2d(X, (2, 2), 'avg')
```

```
tensor([[2., 3.],
        [5., 6.]])
```

6.5.2 Dolgu ve Uzun Adım

Evrişimli katmanlarda olduğu gibi, ortaklama katmanları da çıktının şeklini değiştirebilir. Ayrıca daha önce olduğu gibi, girdiyi dolgulayarak ve uzun adımı ayarlayarak istenen çıktı şeklini elde etmek için işlemi değiştirebiliriz. Derin öğrenme çerçevesinden yerleşik iki boyutlu maksimum ortaklama katmanı aracılığıyla ortaklama katmanlarında dolgu ve uzun adımların kullanımını gösterebiliriz. İlk olarak dört boyutlu şekle sahip bir `X` girdi tensörü inşa ediyoruz, burada örneklerin sayısı (iş boyutu) ve kanalların sayısının her ikisi de 1'dir.

```
X = torch.arange(16, dtype=torch.float32).reshape((1, 1, 4, 4))
X
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]]]])
```

Varsayılan olarak, çerçevenin yerleşik sınıfındaki örnekteki uzun adım ve ortaklama penceresi aynı şekle sahiptir. Aşağıda, `(3, 3)` şeklindeki bir ortaklama penceresi kullanıyoruz, bu nedenle varsayılan olarak `(3, 3)`'lük bir adım şekli alıyoruz.

```
pool2d = nn.MaxPool2d(3)
pool2d(X)
```

```
tensor([[[[10.]]]])
```

Uzun adım ve dolgu manuel olarak belirtilebilir.

```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

```
tensor([[[[ 5.,  7.],
          [13., 15.]]]])
```

Tabii ki, keyfi bir dikdörtgen ortaklama penceresi belirleyebilir ve sırasıyla yükseklik ve genişlik için dolguyu ve uzun adımı belirtebiliriz.

```
pool2d = nn.MaxPool2d((2, 3), stride=(2, 3), padding=(0, 1))
pool2d(X)
```

```
tensor([[[[ 5.,  7.],
          [13., 15.]]]])
```

6.5.3 Çoklu Kanal

Çok kanallı girdi verilerini işlerken, ortaklama katmanı, girdileri bir evrişimli katmanda olduğu gibi kanallar üzerinden toplamak yerine her girdi kanalını ayrı ayrı ortaklar. Bu, ortaklama katmanının çıktı kanallarının sayısının girdi kanalı sayısıyla aynı olduğu anlamına gelir. Aşağıda, 2 kanallı bir girdi oluşturmak için kanal boyutundaki `X` ve `X + 1` tensörleri birleştiriyoruz.

```
X = torch.cat((X, X + 1), 1)
X
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]],

         [[ 1.,  2.,  3.,  4.],
          [ 5.,  6.,  7.,  8.],
          [ 9., 10., 11., 12.],
          [13., 14., 15., 16.]]]])
```

Gördüğümüz gibi, çıktı kanallarının sayısı ortaklamadan sonra hala 2'dir.

```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

```
tensor([[[[ 5.,  7.],
          [13., 15.]],

         [[ 6.,  8.],
          [14., 16.]]]])
```

6.5.4 Özet

- Ortaklama penceresinde girdi öğelerini alarak, maksimum ortaklama işlemi çıktı olarak maksimum değeri atar ve ortalama ortaklama işlemi ortalama değeri çıktı olarak atar.
- Bir ortaklama tabakasının en önemli avantajlarından biri, evrişimli tabakanın konumuna aşırı duyarlılığını hafifletmektir.
- Ortaklama katmanı için dolgu ve uzun adım belirtebiliriz.
- Uzamsal boyutları (örn. genişlik ve yükseklik) azaltmak için 1'den büyük bir uzun adımla birlikte maksimum ortaklama kullanılabilir.
- Ortaklama katmanının çıktı kanalı sayısı, girdi kanallarının sayısıyla aynıdır.

6.5.5 Alıştırmalar

1. Bir evrişim tabakasının özel bir durumu olarak ortalama ortaklama uygulayabilir misiniz? Eğer öyleyse, yapınız.
2. Bir evrişim tabakasının özel bir durumu olarak maksimum ortaklama uygulayabilir misiniz? Eğer öyleyse, yapınız.
3. Ortaklama katmanının hesaplama maliyeti nedir? Ortaklama katmanına girdi boyutunun $c \times h \times w$ olduğunu, ortaklama penceresinin $p_h \times p_w$ bir şekle sahip, (p_h, p_w) dolgulu ve (s_h, s_w) uzun adımlı olduğunu varsayalım.
4. Neden maksimum ortaklama ile ortalama ortaklamanın farklı çalışmasını beklersiniz?
5. Ayrı bir minimum ortaklama katmanına ihtiyacımız var mıdır? Onu başka bir işlemle değiştirebilir misiniz?
6. Ortalama ve maksimum ortaklama arasında düşünebileceğiniz başka bir işlem var mıdır (İpucu: Softmaks'i anımsayın)? Neden o kadar popüler olmayacaktır?

Tartışmalar[88]

6.6 Evrişimli Sinir Ağları (LeNet)

Artık tam fonksiyonlu bir CNN kurmak için gerekli tüm malzemelere sahibiz. İmge verileriyle daha önceki karşılaşmamızda, Fashion-MNIST veri kümesindeki giyim resimlerine bir softmaks bağlanım modeli (Section 3.6) ve bir MLP modeli (Section 4.2) uyguladık. Bu tür verileri softmaks bağlanıma ve MLP'lere uygun hale getirmek için, önce 28×28 matrisinden her imgeyi sabit uzunlukta 784 boyutlu bir vektöre düzleştirdik ve daha sonra bunları tam bağlı katmanlarla işledik. Artık evrişimli katmanlar üzerinde bir kavrayışımız olduğuna göre, imgelerimizdeki konumsal yapıyı koruyabiliriz. Tam bağlı katmanları evrişimsel katmanlarla değiştirmenin ek bir avantajı olarak, çok daha az parametre gerektiren daha eli sıkı modellerin keyfini çıkaracağız.

Bu bölümde, ilk yayınlanan CNN'ler arasından bilgisayarla görme görevlerinde performansından dolayı dikkat çeken *LeNet*'i tanıtacağız. Model, ((LeCun *et al.*, 1998)) imgelerdeki el yazısı rakamlarını tanımak amacıyla AT&T Bell Labs'te araştırmacı olan Yann LeCun tarafından tanıtıldı (ve adlandırıldı). Bu çalışma, bu teknolojiyi geliştiren on yıllık bir araştırmanın doruk noktalarını

[88] https://discuss.d2l.ai/t/72

temsil ediyordu. 1989'da LeCun, CNN'leri geri yayma yoluyla başarılı bir şekilde eğiten ilk çalışmayı yayınladı.

LeNet, destek vektörü makinelerinin performansıyla eşleşen olağanüstü sonuçlar elde etti, daha sonra gözetimli öğrenmede baskın bir yaklaşım oldu. LeNet en sonunda ATM makinelerinde mevduat işlemede rakamları tanımak için uyarlanmıştır. Günümüze kadar, bazı ATM'ler hala Yann ve meslektaşı Leon Bottou'nun 1990'larda yazdığı kodu çalıştırıyor!

6.6.1 LeNet

Yüksek düzeyde, LeNet (LeNet-5) iki parçadan oluşur: (i) İki evrişimli katmandan oluşan bir evrişimli kodlayıcı; ve (ii) üç tam bağlı katmandan oluşan yoğun bir blok: Mimarisi Fig. 6.6.1 içinde özetlenmiştir.

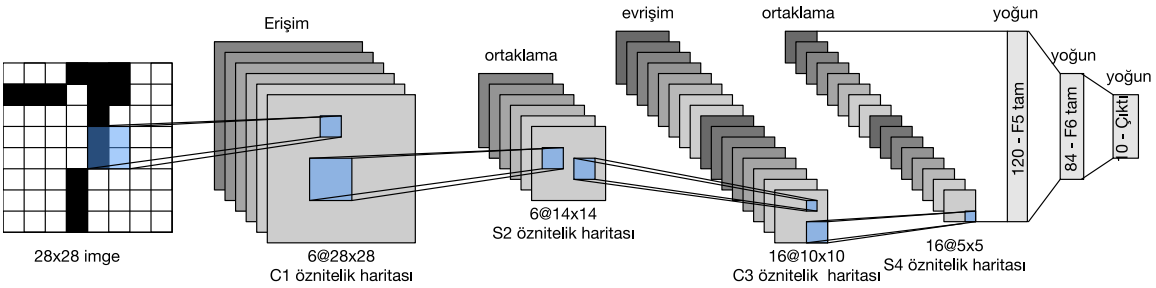


Fig. 6.6.1: LeNet'te veri akışı. Girdi el yazısı bir rakamdır, çıktı ise 10 olası sonucun üzerinden bir olasılıktır.

Her bir evrişimli bloktaki temel birimler, bir evrişimli tabaka, bir sigmoid etkinleştirme fonksiyonu ve sonrasında bir ortalama ortaklama işlemidir. ReLU'lar ve maksimum ortaklama daha iyi çalışırken, bu keşifler henüz 1990'larda yapılmamıştı. Her bir evrişimli katman bir 5×5 çekirdeği ve sigmoid etkinleştirme işlevi kullanır. Bu katmanlar, konumsal olarak düzenlenmiş girdileri bir dizi iki boyutlu öznitelik eşlemelerine eşler ve genellikle kanal sayısını arttırır. İlk evrişimli tabaka 6 tane çıktı kanalına, ikincisi ise 16 taneye sahiptir. Her 2×2 ortaklama işlemi (2'lik uzun adım), uzamsal örnek seyreltme yoluyla boyutsallığı 4 kat düşürür. Evrişimli blok tarafından verilen şekle sahip bir çıktı yayar (toplu iş boyutu, kanal sayısı, yükseklik, genişlik).

Evrişimli bloktan yoğun bloğa çıktıyı geçirmek için, minigruptaki her örneği düzleştirmeliyiz. Başka bir deyişle, bu dört boyutlu girdiyi alıp tam bağlı katmanlar tarafından beklenen iki boyutlu girdiye dönüştürüyoruz: Bir hatırlatma olarak, arzu ettiğimiz iki boyutlu gösterim, minigrup örneklerini indekslemek için ilk boyutu kullanır ve ikincisini örneği temsil eden düz vektörü vermek için kullanır. LeNet'in yoğun bloğu sırasıyla 120, 84 ve 10 çıktılı üç tam bağlı katman içerir. Hala sınıflandırma gerçekleştirdiğimiz için, 10 boyutlu çıktı katmanı olası çıktı sınıflarının sayısına karşılık gelir.

LeNet'in içinde neler olup bittiğini gerçekten anladığınız noktaya gelirken, umarım aşağıdaki kod parçacığı sizi modern derin öğrenme çerçeveleri ile bu tür modellerin uygulanmasının son derece basit olduğuna ikna edecektir. Sadece `Sequential` bloğunu örneklendirmeniz ve uygun katmanları birbirine bağlamamız gerekiyor.

```
import torch
from torch import nn
from d2l import torch as d2l

net = nn.Sequential(
    nn.Conv2d(1, 6, kernel_size=5, padding=2), nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2),
    nn.Conv2d(6, 16, kernel_size=5), nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2),
    nn.Flatten(),
    nn.Linear(16 * 5 * 5, 120), nn.Sigmoid(),
    nn.Linear(120, 84), nn.Sigmoid(),
    nn.Linear(84, 10))
```

Orijinal modelde biraz özgürce davrandık, son kattaki Gauss etkinleştirmesini kaldırdık. Bunun dışında, bu ağ orijinal LeNet-5 mimarisiyle eşleşir.

Tek kanallı (siyah beyaz) 28×28 imgesini ağ üzerinden geçirerek ve çıktı şeklini her katmanda yazdırarak, işlemlerinin Fig. 6.6.2 içinde gösterilen gibi beklediğimiz şeyle hizaladığından emin olmak için modeli inceleyebiliriz.

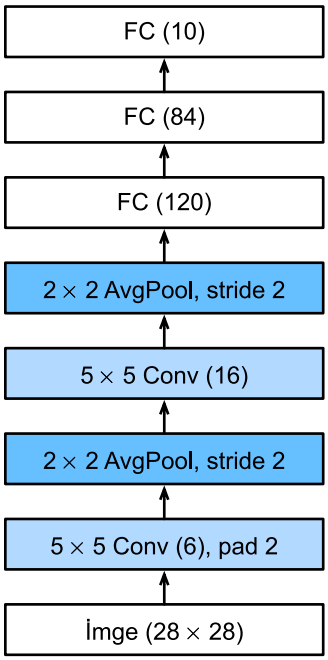


Fig. 6.6.2: LeNet-5'in sıkıştırılmış gösterimi.

```
X = torch.rand(size=(1, 1, 28, 28), dtype=torch.float32)
for layer in net:
    X = layer(X)
    print(layer.__class__.__name__,'output shape: \t',X.shape)
```

```
Conv2d output shape:         torch.Size([1, 6, 28, 28])
Sigmoid output shape:        torch.Size([1, 6, 28, 28])
AvgPool2d output shape:      torch.Size([1, 6, 14, 14])
```

(continues on next page)

(continued from previous page)

```
Conv2d output shape:         torch.Size([1, 16, 10, 10])
Sigmoid output shape:        torch.Size([1, 16, 10, 10])
AvgPool2d output shape:      torch.Size([1, 16, 5, 5])
Flatten output shape:        torch.Size([1, 400])
Linear output shape:         torch.Size([1, 120])
Sigmoid output shape:        torch.Size([1, 120])
Linear output shape:         torch.Size([1, 84])
Sigmoid output shape:        torch.Size([1, 84])
Linear output shape:         torch.Size([1, 10])
```

Evrişimli blok boyunca her katmanda (önceki katmanla karşılaştırıldığında) gösterim yüksekliğinin ve genişliğinin azaltıldığını unutmayın. İlk evrişimli katman, 5×5 çekirdeğinin kullanılmasından kaynaklanan yükseklik ve genişlik azalmasını telafi etmek için 2 piksellik dolgu kullanır. Buna karşılık, ikinci evrişimli tabaka dolgudan vazgeçer ve böylece yüksekliğin ve genişliğin her ikisi de 4'er piksel azaltılır. Katman yığınında yukarı çıktığımızda, kanalların sayısı, ilk evrişimli tabakadan sonra girdideki 1'den 6'ya ve ikinci evrişimli tabakadan sonra 16'ya yükselir. Bununla birlikte, her bir ortaklama katmanı yüksekliği ve genişliği yarıya indirir. Son olarak, her tam bağlı katman boyutsallığı azaltır ve nihai olarak boyutu sınıf sayısıyla eşleşen bir çıktı yayar.

6.6.2 Eğitim

Modeli uyguladığımıza göre, LeNet'in Fashion-MNIST üzerinde nasıl çalışacağını görmek için bir deney yapalım.

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=batch_size)
```

CNN'lerde daha az parametre olsa da, hesaplamalar benzer derin MLP'lerden daha külfetli olabilir çünkü her parametre daha fazla çarpmada kullanılır. GPU'ya erişiminiz varsa, bu işlemi hızlandırmak için harekete geçirmenin tam zamanı olabilir.

Değerlendirme için Section 3.6 içinde tarif ettiğimiz `evaluate_accuracy` işlevinde hafif bir değişiklik yapmamız gerekiyor. Bütün veri kümesi ana bellekte olduğundan, modelin veri kümesiyle hesaplama yapabilmesi için GPU'yu kullanmadan önce veriyi GPU belleğine kopyalamamız gerekir.

```
def evaluate_accuracy_gpu(net, data_iter, device=None): #@save
    """GPU kullanarak bir veri kümesindeki bir modelin doğruluğunu hesapla."""
    if isinstance(net, nn.Module):
        net.eval()  # Modeli değerlendirme moduna ayarlayın
        if not device:
            device = next(iter(net.parameters())).device
   # Doğru tahmin sayısı, tahminlerin sayısı
    metric = d2l.Accumulator(2)

    with torch.no_grad():
        for X, y in data_iter:
            if isinstance(X, list):
                # BERT ince ayarı için gerekli (daha sonra ele alınacaktır)
                X = [x.to(device) for x in X]
            else:
```

(continues on next page)

(continued from previous page)

```
                X = X.to(device)
            y = y.to(device)
            metric.add(d2l.accuracy(net(X), y), y.numel())
    return metric[0] / metric[1]
```

Ayrıca GPU'larla başa çıkmak için eğitim fonksiyonumuzu güncellememiz gerekiyor. Section 3.6 içinde tanımlanan `train_epoch_ch3`'in aksine, şimdi ileri ve geri yayma yapmadan önce her bir veri minigrubunu belirlenen cihazımıza (GPU olması beklenir) taşımamız gerekiyor.

Eğitim fonksiyonu `train_ch6`, Section 3.6 içinde tanımlanan `train_ch3`'ya da benzer. Birçok ileriye doğru ilerleyen katmanlardan oluşan ağları uygulayacağımızdan, öncelikle üst düzey API'lere güveneceğiz. Aşağıdaki eğitim işlevi, girdi olarak üst düzey API'lerden oluşturulan bir modeli varsayar ve buna göre eniyilenir. Section 4.8.2 içinde tanıtıldığı gibi Xavier ilklemede `device` argümanı ile belirtilen cihazdaki model parametrelerini ilkliyoruz. Tıpkı MLP'lerde olduğu gibi, kayıp fonksiyonumuzu çapraz entropi ve minigrup rasgele eğim inişi yoluyla en aza indiriyoruz. Her bir dönemin çalışması on saniye sürdüğünden, eğitim kaybını daha sık görselleştiriyoruz.

```
#@save
def train_ch6(net, train_iter, test_iter, num_epochs, lr, device):
    """GPU ile bir modeli eğitin (Bölüm 6'da tanımlanmıştır)."""
    def init_weights(m):
        if type(m) == nn.Linear or type(m) == nn.Conv2d:
            nn.init.xavier_uniform_(m.weight)
    net.apply(init_weights)
    print('training on', device)
    net.to(device)
    optimizer = torch.optim.SGD(net.parameters(), lr=lr)
    loss = nn.CrossEntropyLoss()
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                            legend=['train loss', 'train acc', 'test acc'])
    timer, num_batches = d2l.Timer(), len(train_iter)
    for epoch in range(num_epochs):
        # Eğitim kaybı toplamı, eğitim doğruluğu toplamı, örnek sayısı
        metric = d2l.Accumulator(3)
        net.train()
        for i, (X, y) in enumerate(train_iter):
            timer.start()
            optimizer.zero_grad()
            X, y = X.to(device), y.to(device)
            y_hat = net(X)
            l = loss(y_hat, y)
            l.backward()
            optimizer.step()
            with torch.no_grad():
                metric.add(l * X.shape[0], d2l.accuracy(y_hat, y), X.shape[0])
            timer.stop()
            train_l = metric[0] / metric[2]
            train_acc = metric[1] / metric[2]
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (train_l, train_acc, None))
        test_acc = evaluate_accuracy_gpu(net, test_iter)
```

(continues on next page)

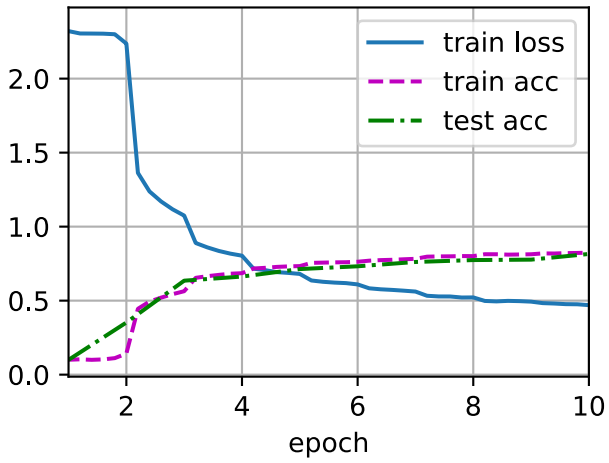
(continued from previous page)

```
        animator.add(epoch + 1, (None, None, test_acc))
    print(f'loss {train_l:.3f}, train acc {train_acc:.3f}, '
          f'test acc {test_acc:.3f}')
    print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec '
          f'on {str(device)}')
```

Şimdi LeNet-5 modelini eğitip değerlendirelim.

```
lr, num_epochs = 0.9, 10
train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.470, train acc 0.824, test acc 0.815
82076.9 examples/sec on cuda:0
```



6.6.3 Özet

- CNN, evrişimli katmanları kullanan bir ağdır.
- Bir CNN'de, evrişimlerinin arasına, doğrusal olmayanları işlemleri ve (genellikle) ortaklama işlemlerini koyuyoruz.
- Bir CNN'de, evrişimli katmanlar tipik olarak, kanal sayısını arttırırken gösterimlerin mekansal çözünürlüğünü yavaş yavaş azaltacak şekilde düzenlenir.
- Geleneksel CNN'lerde, evrişimli bloklar tarafından kodlanan temsiller, çıktı yayılmadan önce bir veya daha fazla tam bağlı katman tarafından işlenir.
- LeNet, tartışmasız, böyle bir ağın ilk başarılı konuşlandırması oldu.

6.6.4 Alıştırmalar

1. Ortalama ortaklama ile maksimum ortaklamayı değiştirin. Ne olur?
2. Doğruluğunu artırmak için LeNet'e dayalı daha karmaşık bir ağ oluşturmaya çalışın.
    1. Evrişim penceresi boyutunu ayarlayın.
    2. Çıktı kanallarının sayısını ayarlayın.
    3. Etkinleştirme işlevini ayarlayın (örneğin, ReLU).
    4. Evrişim katmanlarının sayısını ayarlayın.
    5. Tam bağlı katmanların sayısını ayarlayın.
    6. Öğrenme oranlarını ve diğer eğitim ayrıntılarını ayarlayın (örneğin, ilkleme ve dönem sayısı).
3. Özgün MNIST veri kümesi üzerinde geliştirdiğiniz ağı deneyin.
4. Farklı girdiler için LeNet'in birinci ve ikinci katmanının etkinleştirmelerini gösterin (ör. kazak ve paltolar gibi).

Tartışmalar[89]

[89] https://discuss.d2l.ai/t/74

7 | Modern Evrişimli Sinir Ağları

Artık CNN'leri biraraya bağlama temellerini anladığımıza göre, sizlerle modern CNN mimarilerinde bir tur atacağız. Bu ünitede, her bölüm bir noktada (veya şu anda) birçok araştırma projesinde ve konuşlandırılmış sistemlerde inşanın temel modeli olan önemli bir CNN mimarisine karşılık gelmektedir. Bu ağların her biri kısaca baskın bir mimariydi ve çoğu, 2010 yılından bu yana bilgisayarla görmede gözetimli öğrenmenin ilerlemesinin göstergesi olarak hizmet eden ImageNet yarışmasında kazananlar veya ikincilerdi.

Bu modeller, büyük ölçekli bir görme yarışmasında geleneksel bilgisayarla görme yöntemlerini yenmek için konuşlandırılan ilk büyük ölçekli ağ olan AlexNet'i içerir; bir dizi tekrarlayan eleman bloklarını kullanan VGG ağı; girdileri tüm sinir ağlarını yamalı olarak evriştiren ağ içindeki ağ (NiN); paralel birleştirmelere sahip ağları kullanan GoogLeNet; bilgisayarla görmede en popüler kullanıma hazır mimari olmaya devam eden artık ağlar (residual networks - ResNet) ve hesaplanması külfetli ancak bazı alanlarda en yüksek başarı performanslarını ortaya koyan yoğun bağlı ağlar (DenseNet).

Derin sinir ağları fikri oldukça basit olsa da (bir grup katmanı bir araya getirin), performans mimarilere ve hiper parametre seçeneklerine bağlı olarak aşırı farklılık gösterebilir. Bu bölümde açıklanan sinir ağları sezginin, birkaç matematiksel içgörünün ve bir sürü deneme yanılmanın ürünüdür. Bu modelleri tarihsel sırayla sunuyoruz, kısmen de tarih duygusunu aktarmak için, böylece alanın nereye gittiği ile ilgili kendi sezgilerinizi oluşturabilir ve belki de kendi mimarilerinizi geliştirebilirsiniz. Örneğin, bu bölümde açıklanan toptan normalleştirme ve artık bağlantılar, derin modellerin eğitimi ve tasarlanması için iki popüler fikir sunmuştur.

7.1 Derin Evrişimli Sinir Ağları (AlexNet)

CNN'ler LeNet'in ortaya çıkışını takiben bilgisayarla görme ve makine öğrenmesi topluluklarında iyi bilinse de, alanda hemen baskın olmadılar. LeNet, ilk küçük veri kümelerinde iyi sonuçlar elde etse de, CNN'leri daha büyük, daha gerçekçi veri kümeleri üzerinde eğitmenin başarımı ve uygulanabilirliği henüz belirlenmemişti. Aslında, 1990'ların başları ile 2012 yılının dönüm noktası sonuçları ortasındaki ara dönemin büyük bir bölümünde, sinir ağları genellikle destek vektör makineleri gibi diğer makine öğrenmesi yöntemleri tarafından aşıldı.

Bilgisayarla görme için, bu karşılaştırma belki de adil değildir. Bu, evrişimli ağlara girdiler ham veya hafif işlenmiş (örneğin, merkeze çekme yoluyla) piksel değerlerinden oluşsa da, uygulayıcılar asla ham pikselleri geleneksel modellere beslemezlerdi. Bunun yerine, tipik bilgisayarla görmedeki işlem hatları, manuel tasarlanmış öznitelik çıkarma işlem hatlarından oluşuyordu. *Öznitelikleri öğrenmek*tense, öznitelikler *hazırlanırdı*. İlerlemenin çoğu, öznitelikler için daha akıllı fikirlere sahip olunmasından gelirdi ve öğrenme algoritması genellikle sonradan düşünülürdü.

Bazı sinir ağı hızlandırıcıları 1990'larda da mevcut olmasına rağmen, çok sayıda parametreyle derin çok kanallı, çok katmanlı CNN'ler yapmak için henüz yeterince güçlü değillerdi. Dahası, veri kümeleri hala nispeten küçüktü. Bu engellere ek olarak, parametre ilkleme sezgisel yöntemleri, rasgele eğim inişin akıllı biçimleri, daralmayan etkinleştirme fonksiyonları ve etkili düzenlileştirme teknikleri de dahil olmak üzere sinir ağlarını eğitmek için anahtar püf noktaları hala eksikti.

Böylece, *uçtan uça* (pikselden sınıflandırmaya) sistemleri eğitmekten ziyade, klasik işlem hatları daha çok şöyle görünüyordu:

1. İlginç bir veri kümesi elde edin. İlk günlerde, bu veri kümeleri pahalı sensörlere ihtiyaç duyuyordu (o zamanlarda 1 megapiksel imgeler son teknolojiydi).
2. Veri kümesini, optik, geometri, diğer analitik araçlar ve bazen şanslı lisansüstü öğrencilerin rastlantısal keşiflerine dayanan elle hazırlanmış özniteliklerle ön işleyin.
3. Verileri SIFT (ölçek-değişmez öznitelik dönüşümü) (Lowe, 2004), SURF (hızlandırılmış gürbüz öznitelikler) (Bay *et al.*, 2006) veya herhangi bir sayıda diğer elle ayarlanmış işlem hatları gibi standart öznitelik çıkarıcılar kümesine besleyin.
4. Bir sınıflandırıcı eğitmek için ortaya çıkan temsilleri en sevdiğiniz sınıflandırıcıya (muhtemelen doğrusal bir model veya çekirdek yöntemi) boşaltın.

Makine öğrenmesi araştırmacılarıyla konuştuysanız, makine öğrenmesinin hem önemli hem de güzel olduğuna inanırlardı. Zarif teoriler çeşitli sınıflandırıcıların özelliklerini kanıtladı. Makine öğrenmesi alanı gelişen, titiz ve son derece yararlıydı. Ancak, bir bilgisayarla görme araştırmacısı ile konuştuysanız, çok farklı bir hikaye duyardınız. İmge tanımanın kirli gerçeğini size söylerlerdi; öğrenme algoritmaları değil, öznitelikler ilerlemeyi yönlendirir. Bilgisayarla görme araştırmacıları haklı olarak, biraz daha büyük veya daha temiz bir veri kümesinin veya biraz geliştirilmiş öznitelik çıkarma işlem hattının nihai doğruluk oranı için herhangi bir öğrenme algoritmasından çok daha önemli olduğuna inanıyordu.

7.1.1 Temsilleri Öğrenme

Vaziyeti ortaya dökmenin bir diğer yolu da işlem hattının en önemli kısmının temsil olmasıdır. 2012 yılına kadar temsil mekanik olarak hesaplanırdı. Aslında, yeni bir öznitelik fonksiyonu kümesi işleme, sonuçları iyileştirme ve yöntemi yazma belirgin bir makale türüydü. SIFT (Lowe, 2004), SURF (Bay *et al.*, 2006), HOG (yönlendirilmiş gradyan histogramları) (Dalal and Triggs, 2005), görsel sözcük torbası[90] ve benzeri öznitelik çıkarıcılar ortamın hakimiydi.

Yann LeCun, Geoff Hinton, Yoshua Bengio, Andrew Ng, Shun-ichi Amari ve Juergen Schmidhuber de dahil olmak üzere bir başka araştırmacı grubunun farklı planları vardı. Onlar özniteliklerin kendilerinin öğrenilmesi gerektiğine inanıyordu. Dahası, makul derecede karmaşık olması için, özniteliklerin hiyerarşik olarak, her biri öğrenilebilir parametrelere sahip birden çok ortak öğrenilen katmanla oluşması gerektiğine inanıyorlardı. İmge durumunda, en alt katmanlar kenarları, renkleri ve dokuları algılayabilir. Gerçekten de, Alex Krizhevsky, Ilya Sutskever ve Geoff Hinton, yeni bir CNN biçimini, *AlexNet*i, önerdi ve o 2012 ImageNet yarışmasında mükemmel bir performans elde etti. AlexNet, atılım yapan ImageNet sınıflandırma makalesinin (Krizhevsky *et al.*, 2012) ilk yazarı olan Alex Krizhevsky'nin adını aldı.

İlginçtir ki, ağın en düşük katmanlarında, model bazı geleneksel filtrelere benzeyen öznitelik çıkarıcıları öğrendi. Şekil Fig. 7.1.1, AlexNet (Krizhevsky *et al.*, 2012) makalesinden tekrar üretilmiştir ve alt düzey imge tanımlayıcılarını göstermektedir.

[90] https://en.wikipedia.org/wiki/Bag-of-words_model_in_computer_vision

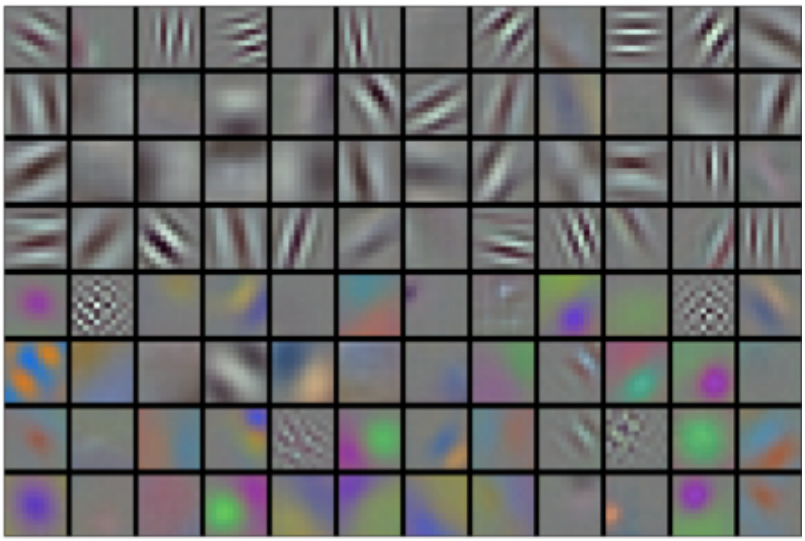


Fig. 7.1.1: AlexNet'in ilk katmanındaki öğrenilmiş imgeler

Ağdaki daha üst katmanlar, gözler, burunlar, çimen çizgileri vb. gibi daha büyük yapıları temsil edecek şekilde bu temsillerin üzerine inşa edilebilir. Daha üst katmanlar bile insanlar, uçaklar, köpekler veya frizbi gibi tüm nesneleri temsil edebilir. Nihayetinde, son gizli durum, farklı kategorilere ait verilerin kolayca ayrılabilmesi için içeriğini özetleyen imgenin sıkıştırılmış bir temsilini öğrenir.

Çok katmanlı CNN'ler için nihai atılım 2012'de gelirken, çekirdek bir grup araştırmacı, uzun yıllar görsel verilerin hiyerarşik temsillerini öğrenmeye çalışarak kendilerini bu fikre adamıştı. 2012'deki nihai atılım iki temel faktöre bağlanabilir.

Eksik Malzeme: Veri

Birçok katmana sahip derin modeller, dışbükey optimizasyonlara (örneğin doğrusal ve çekirdek yöntemleri) dayalı geleneksel yöntemlerden önemli ölçüde daha iyi performans gösterdikleri düzene girmek için büyük miktarda veri gerektirirler. Bununla birlikte, bilgisayarların sınırlı depolama kapasitesi, sensörlerin göreceli pahalı fiyatları ve 1990'lardaki nispeten daha kısıtlı araştırma bütçeleri göz önüne alındığında, çoğu araştırma küçük veri kümelerine dayanıyordu. Çok sayıda makale, UCI veri kümelerinin koleksiyonuna değiniyordi, bunların birçoğu düşük çözünürlükte doğal olmayan ortamlarda çekilmiş sadece yüzlerce veya (birkaç) binlerce imge içeriyordu.

2009'da ImageNet veri kümesi yayımlandı ve araştırmacıları 1000 farklı nesne kategorisinden her biri farklı 1000 eleman içeren toplamda 1 milyon örnek ile modeller öğrenmeye zorladı. Bu veri kümesini tanıtan Fei-Fei Li liderliğindeki araştırmacılar, her kategori için büyük aday kümelerini ön filtrelemek için Google Image Search kullandı ve her imgenin ilgili kategoriye ait olup olmadığını onaylatmak için Amazon Mechanical Turk kitle kaynak kullanımı işlem hattını kullandı. Bu eşi görülmemiş bir ölçekti. ImageNet yarışması olarak adlandırılan ilgili rekabet, bilgisayarla görme ve makine öğrenmesi araştırmalarını ileriye sürdü, araştırmacıları hangi modellerin daha önce akademisyenlerin düşündüklerinden daha büyük bir ölçekte en iyi performans gösterdiğini belirlemeye zorladı.

Eksik Malzeme: Donanım

Derin öğrenme modelleri, bilgi işlem döngülerinin doymak bilmeyen tüketicileridir. Eğitim yüzlerce dönem alabilir ve her yineleme, veriyi hesaplamalı olarak pahalı doğrusal cebir işlemlerine sahip birçok katmanından geçirmeyi gerektirir. Bu, 1990'ların ve 2000'lerin başında, daha verimli bir şekilde optimize edilmiş dışbükey amaç fonksiyonlarına dayanan basit algoritmaların tercih edilmesinin başlıca nedenlerinden biridir.

Grafik işleme birimleri (GPU'lar) derin öğrenmeyi mümkün kılmada bir oyun değiştirici olduğu kanıtladı. Bu yongalar uzun zamandan öncesinden bilgisayar oyunlarında faydalanmak için grafik işlemeyi hızlandırmak amacıyla geliştirilmişti. Özellikle, birçok bilgisayar grafik görevinde gerekli olan yüksek verim 4×4 matris vektör çarpımları için optimize edilmişlerdir. Neyse ki, buradaki matematik, evrişimli katmanları hesaplamak için gerekli olana çarpıcı bir şekilde benzer. Bu süre zarfında NVIDIA ve ATI, GPU'ları genel hesaplama işlemleri için optimize etmeye ve bunları *genel amaçlı GPU'lar* (GPGPU) olarak pazarlamaya başlamıştı.

Biraz sezgi sağlamak için, modern bir mikroişlemcinin (CPU) çekirdeklerini göz önünde bulundurun. Çekirdeklerin her biri, yüksek bir saat frekansı ve gösterişli büyük önbellekleri (birkaç megabayta kadar L3) ile oldukça güçlü çalışır. Her çekirdek, dallanma tahmincileri, derin bir işlem hattı ve çok çeşitli programları çalıştırmasını sağlayan diğer çekici ek özellikleri ile birlikte birçok çeşitli talimatı uygulamak için çok uygundur. Bununla birlikte, bu belirgin güç aynı zamanda Aşil topuğudur: Genel amaçlı çekirdekler inşa etmek çok pahalıdır. Çok sayıda yonga alanı, karmaşık bir destek yapısı (bellek arabirimleri, çekirdekler arasında önbelleğe alma mantığı, yüksek hızlı ara bağlantılar vb.) gerektirirler ve herhangi bir özel görevde nispeten kötüdürler. Modern dizüstü bilgisayarlar 4 adede kadar çekirdeğe sahiptir ve hatta üst düzey sunucular bile 64 çekirdeği nadiren aşmaktadır, çünkü uygun maliyetli değildir.

Karşılaştırma olarak, GPU'lar $100 \sim 1000$ arasında küçük işleme elemanlarından oluşur (ayrıntılar NVIDIA, ATI, ARM ve diğer yonga satıcıları arasında biraz farklılık gösterir), genellikle daha büyük gruplar halinde gruplandırılır (NVIDIA bunları bükülme (warp) olarak adlandırır). Her çekirdek nispeten zayıfken, bazen 1GHz altı saat frekansında bile olsa, GPU'ların büyüklük mertebesini CPU'lardan daha hızlı hale getiren bu tür çekirdeklerin toplam sayısıdır. Örneğin, NVIDIA'nın yeni Volta nesli, özel talimatlar için çip başına 120 TFlop (ve daha genel amaçlı olanlar için 24 TFlop'a kadar) sunarken, CPU'ların kayan virgüllü sayı performansı bugüne kadar 1 TFlop'u aşmadı. Bunun mümkün olmasının nedeni aslında oldukça basittir: Birincisi, güç tüketimi saat frekansı ile *dört kat* büyümeye eğilimlidir. Bu nedenle, 4 kat daha hızlı çalışan bir CPU çekirdeğinin güç bütçesi ile (tipik bir sayı), $1/4$ hızında 16 GPU çekirdeğini kullanabilirsiniz, bu da performansın $16 \times 1/4 = 4$ katını verir. Ayrıca, GPU çekirdekleri çok daha basittir (aslında, uzun bir süre için genel amaçlı kod *yürütebilir* bile değillerdi), bu da onları daha verimli kılar. Son olarak, derin öğrenmede birçok işlem yüksek bellek bant genişliği gerektirir. Yine, GPU'lar burada en az 10 kat daha geniş olan veriyolları ile parlıyor.

2012'ye geri dönelim. Alex Krizhevsky ve Ilya Sutskever GPU donanımı üzerinde çalışabilecek derin bir CNN uyguladığında büyük bir atılım oldu. CNN'lerdeki hesaplamalı darboğazların, evrişimlerin ve matris çarpımlarının, tümünün donanımda paralelleştirilebilecek işlemler olduğunu fark ettiler. 3GB belleğe sahip iki NVIDIA GTX 580s kullanarak hızlı evrişimler uyguladılar. cuda-convnet[91] kodu o kadar iyiydi ki birkaç yıl endüstri standardı oldu ve derin öğrenmenin patladığı ilk birkaç yılın itici gücü oldu.

[91] https://code.google.com/archive/p/cuda-convnet/

7.1.2 AlexNet

8 katmanlı CNN kullanan AlexNet, 2012 ImageNet Büyük Ölçekli Görsel Tanıma Yarışması'nı olağanüstü derecede büyük bir farkla kazandı. Bu ağ, ilk kez, öğrenme yoluyla elde edilen özniteliklerin el ile tasarlanmış öznitelikleri aşabildiğini ve bilgisayarla görmede önceki alışılmış kalıpları kırabileceğini gösterdi.

Fig. 7.1.2 içinde gösterildiği gibi AlexNet ve LeNet'in mimarileri çok benzerdir. Modelin iki küçük GPU'ya uyması için 2012'de gerekli olan tasarım tuhaflıklarından bazılarını kaldırarak AlexNet'in biraz aerodinamik bir versiyonunu sunduğumuza dikkat edin.

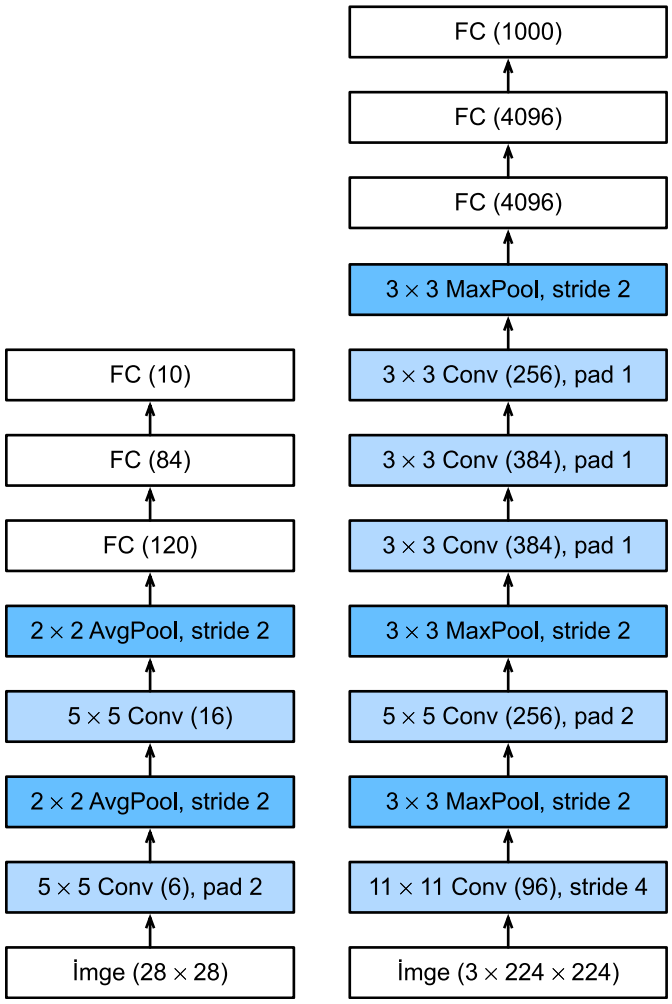


Fig. 7.1.2: LeNet'ten (sol) AlexNet'e (sağ).

AlexNet ve LeNet'in tasarım felsefeleri çok benzer, ancak önemli farklılıklar da vardır. İlk olarak, AlexNet nispeten küçük LeNet5'ten çok daha derindir. AlexNet sekiz katmandan oluşur: Beş evrişimli katman, iki tam bağlı gizli katman ve bir tam bağlı çıktı katmanı. İkincisi, AlexNet etkinleştirme fonksiyonu olarak sigmoid yerine ReLU'yu kullandı. Aşağıda ayrıntıları daha derin inceleyelim.

Mimari

AlexNet'in ilk katmanında, evrişim penceresinin şekli 11×11'dir. ImageNet'teki çoğu imge MNIST imgelerinden on kat daha yüksek ve daha geniş olduğundan, ImageNet verilerindeki nesneler daha fazla piksel kaplama eğilimindedir. Sonuç olarak, nesneyi anlamak için daha büyük bir evrişim penceresi gereklidir. İkinci kattaki evrişim pencere şekli 5×5'e, ardından 3×3'e indirgenir. Buna ek olarak, birinci, ikinci ve beşinci kıvrımsal katmanlardan sonra, ağ 3×3 pencere şekli ve 2'lik bir uzun adım ile maksimum ortaklama katmanları ekler. Ayrıca, AlexNet'in LeNet'ten on kat daha fazla evrişim kanalı vardır.

Son evrişimli tabakadan sonra 4096 çıktılı iki tam bağlı katman vardır. Bu iki büyük tam bağlı katman, yaklaşık 1 GB'lık model parametresi üretir. İlk GPU'lardaki sınırlı bellek nedeniyle, orijinal AlexNet çift veri akışı tasarımı kullandı, böylece iki GPU'larının her biri modelin yalnızca yarısını depolamaktan ve hesaplamaktan sorumlu olabilirdi. Neyse ki, GPU belleği artık nispeten bol, bu yüzden nadiren GPU'lar arasında modelleri parçalamamız gerekiyor (AlexNet modeli versiyonumuz bu açıdan orijinal makaleden sapıyor).

Etkinleştirme İşlevleri

Ayrıca, AlexNet sigmoid etkinleştirme işlevini daha basit bir ReLU etkinleştirme fonksiyonu ile değiştirdi. Bir yandan, ReLU etkinleştirme işlevinin hesaplanması daha kolaydır. Örneğin, sigmoid etkinleştirme işlevinde bulunan üst alma işlemi yoktur. Öte yandan, ReLU etkinleştirme işlevi, farklı parametre ilkleme yöntemleri kullanıldığında model eğitimini kolaylaştırır. Bunun nedeni, sigmoid etkinleştirme işlevinin çıktısı 0 veya 1'e çok yakın olduğunda, bu bölgelerin gradyanının neredeyse 0 olmasıdır, böylece geri yayma model parametrelerinin bazılarını güncelleştirmeye devam edemez. Buna karşılık, pozitif aralıktaki ReLU etkinleştirme işlevinin gradyanı her zaman 1'dir. Bu nedenle, model parametreleri düzgün ilklenmezse, sigmoid işlevi pozitif aralıkta neredeyse 0 gradyan elde edebilir, böylece model etkili bir şekilde eğitilemez.

Kapasite Kontrolü ve Ön İşleme

AlexNet, tam bağlı katmanın model karmaşıklığını hattan düşürme (dropout) ile (Section 4.6) kontrol ederken, LeNet sadece ağırlık sönümlemesini kullanır. Verileri daha da zenginleştirme için AlexNet'in eğitim döngüsü, ters çevirme, kırpma ve renk değişiklikleri gibi çok fazla imge zenginleştirme ekledi. Bu, modeli daha gürbüz hale getirir ve daha büyük örneklem boyutu aşırı öğrenmeyi etkili bir şekilde azaltır. Veri zenginleştirme işlemlerini Section 13.1 içinde daha ayrıntılı olarak tartışacağız.

```
import torch
from torch import nn
from d2l import torch as d2l

net = nn.Sequential(
    # Burada nesneleri yakalamak için daha büyük bir 11 x 11'lik pencere kullanıyoruz.
    # Aynı zamanda, çıktının yüksekliğini ve genişliğini büyük ölçüde azaltmak
    # için 4'lük bir uzun adım kullanıyoruz. Burada, çıktı kanallarının sayısı
    # LeNet'tekinden çok daha fazladır.
    nn.Conv2d(1, 96, kernel_size=11, stride=4, padding=1), nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2),
    # Evrişim penceresini küçült, girdi ve çıktı boyunca tutarlı yükseklik
```

(continues on next page)

(continued from previous page)

```
    # ve genişlik için dolguyu 2'ye ayarlayın ve çıktı kanallarının sayısını artırın
    nn.Conv2d(96, 256, kernel_size=5, padding=2), nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2),
    # Ardışık üç evrişim katmanı ve daha küçük bir evrişim penceresi kullan.
    # Son evrişim katmanı dışında, çıktı kanallarının sayısı daha da artırılır.
    # Ortaklama katmanları, ilk iki evrişim katmanından sonra girdinin
    # yüksekliğini ve genişliğini azaltmak için kullanılmaz.
    nn.Conv2d(256, 384, kernel_size=3, padding=1), nn.ReLU(),
    nn.Conv2d(384, 384, kernel_size=3, padding=1), nn.ReLU(),
    nn.Conv2d(384, 256, kernel_size=3, padding=1), nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2),
    nn.Flatten(),
    # Burada, tam bağlı katmanın çıktı sayısı, LeNet'tekinden birkaç kat
    # daha fazladır. Aşırı öğrenmeyi azaltmak için hattan düşürme katmanını kullan
    nn.Linear(6400, 4096), nn.ReLU(),
    nn.Dropout(p=0.5),
    nn.Linear(4096, 4096), nn.ReLU(),
    nn.Dropout(p=0.5),
    # Çıktı katmanı. Fashion-MNIST kullandığımız için sınıf sayısı
    # makaledeki gibi 1000 yerine 10'dur.
    nn.Linear(4096, 10))
```

Her katmanın çıktı şeklini gözlemlemek için 224 yüksekliğinde ve genişliğinde tek kanallı bir veri örneği oluşturuyoruz. Fig. 7.1.2 içindeki AlexNet mimarisine uyuyor.

```
X = torch.randn(1, 1, 224, 224)
for layer in net:
    X=layer(X)
    print(layer.__class__.__name__,'cikli sekli:\t',X.shape)
```

```
Conv2d cikli sekli:  torch.Size([1, 96, 54, 54])
ReLU cikli sekli:    torch.Size([1, 96, 54, 54])
MaxPool2d cikli sekli:       torch.Size([1, 96, 26, 26])
Conv2d cikli sekli:  torch.Size([1, 256, 26, 26])
ReLU cikli sekli:    torch.Size([1, 256, 26, 26])
MaxPool2d cikli sekli:       torch.Size([1, 256, 12, 12])
Conv2d cikli sekli:  torch.Size([1, 384, 12, 12])
ReLU cikli sekli:    torch.Size([1, 384, 12, 12])
Conv2d cikli sekli:  torch.Size([1, 384, 12, 12])
ReLU cikli sekli:    torch.Size([1, 384, 12, 12])
Conv2d cikli sekli:  torch.Size([1, 256, 12, 12])
ReLU cikli sekli:    torch.Size([1, 256, 12, 12])
MaxPool2d cikli sekli:       torch.Size([1, 256, 5, 5])
Flatten cikli sekli:         torch.Size([1, 6400])
Linear cikli sekli:  torch.Size([1, 4096])
ReLU cikli sekli:    torch.Size([1, 4096])
Dropout cikli sekli:         torch.Size([1, 4096])
Linear cikli sekli:  torch.Size([1, 4096])
ReLU cikli sekli:    torch.Size([1, 4096])
Dropout cikli sekli:         torch.Size([1, 4096])
Linear cikli sekli:  torch.Size([1, 10])
```

7.1.3 Veri Kümesini Okuma

AlexNet makalede ImageNet üzerinde eğitilmiş olsa da, bir ImageNet modelinin yakınsaması modern bir GPU'da bile saatler veya günler sürebileceğinden, burada Fashion-MNIST veri kümesini kullanıyoruz. AlexNet'in doğrudan Fashion-MNIST üzerine uygulanmasıyla ilgili sorunlardan biri, imgelerinin ImageNet imgelerinden daha düşük çözünürlüğe (28×28 piksel) sahip olmasıdır. İşleri yürütebilmek için onları 224×224'ye yükseltiyoruz (genellikle akıllı bir uygulama değil, ama burada AlexNet mimarisine sadık olmak için yapıyoruz). Bu yeniden boyutlandırmayı `resize` bağımsız değişkeni ile `d2l.load_data_fashion_mnist` işlevinde gerçekleştiriyoruz.

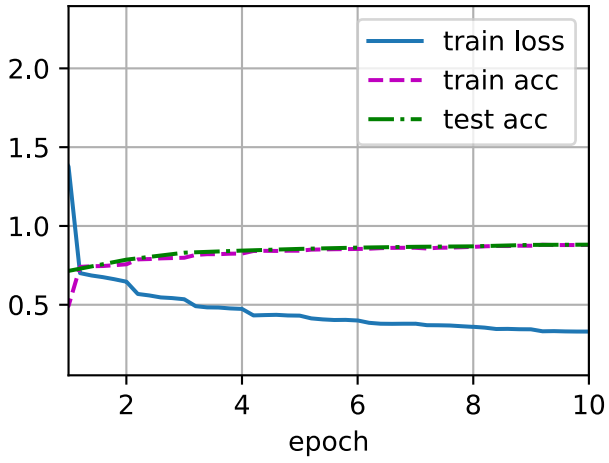
```
batch_size = 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
```

7.1.4 Eğitim

Şimdi AlexNet'i eğitmeye başlayabiliriz. Section 6.6 içindeki LeNet ile karşılaştırıldığında, buradaki ana değişiklik, daha derin ve daha geniş ağ, daha yüksek imge çözünürlüğü ve daha maliyetli evrişimler nedeniyle daha küçük bir öğrenme hızı ve çok daha yavaş eğitim kullanılmasıdır.

```
lr, num_epochs = 0.01, 10
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.330, train acc 0.879, test acc 0.881
4092.1 examples/sec on cuda:0
```



7.1.5 Özet

- AlexNet, LeNet'e benzer bir yapıya sahiptir, ancak büyük ölçekli ImageNet veri kümesine uyacak şekilde daha fazla evrişimli katman ve daha büyük bir parametre uzayı kullanır.
- Bugün AlexNet çok daha etkili mimariler tarafından aşıldı, ancak sığ ağlardan günümüzde kullanılan derin ağlara doğru önemli bir adımdır.
- AlexNet'in uygulamasının LeNet'ten sadece birkaç satır daha fazlası var gibi görünse de, akademik cemiyetin bu kavramsal değişimi benimsemesi ve onun mükemmel deneysel sonuçlarından yararlanması uzun yıllar aldı. Bu aynı zamanda verimli hesaplama araçlarının eksikliğinden kaynaklanıyordu.
- Hattan düşürme, ReLU ve ön işleme, bilgisayarla görme görevlerinde mükemmel performans elde etmek için diğer önemli adımlardı.

7.1.6 Alıştırmalar

1. Dönemlerin sayısını artırmayı deneyin. LeNet ile karşılaştırıldığında, sonuçlar nasıl farklıdır? Neden?
2. AlexNet Fashion-MNIST veri kümesi için çok karmaşık olabilir.
    1. Doğruluğun önemli ölçüde düşmediğinden emin olurken, eğitimi daha hızlı yapmak için modeli basitleştirmeyi deneyin.
    2. Doğrudan 28×28 imgelerde çalışan daha iyi bir model tasarlayın.
3. Minigrup boyutunu değiştirin ve doğruluk ve GPU belleğindeki değişiklikleri gözlemleyin.
4. AlexNet'in hesaplama performansını analiz edin.
    1. AlexNet'in bellek ayak izinin baskın kısım nedir?
    2. AlexNet'te hesaplama için baskın kısım nedir?
    3. Sonuçları hesaplarken bellek bant genişliğinin etkisi ne olabilir?
5. LeNet-5'e hattan düşürme ve ReLU uygulayın. İyileşiyor mu? Ön işlemenin etkisi ne olabilir?

Tartışmalar[92]

7.2 Blokları Kullanan Ağlar (VGG)

AlexNet derin CNN'lerin iyi sonuçlar elde edebileceğine dair deneysel kanıtlar sunarken, yeni ağlar tasarlamada sonraki araştırmacılara rehberlik etmek için genel bir şablon sağlamadı. Aşağıdaki bölümlerde, derin ağları tasarlamak için yaygın olarak kullanılan çeşitli sezgisel kavramları tanıtacağız.

Bu alandaki ilerleme, mühendislerin transistörleri yerleştirilmede mantıksal elemanlardan mantık bloklarına geçtiği yonga tasarımına benzemektedir. Benzer şekilde, sinir ağı mimarilerinin tasarımı, araştırmacıların bireysel nöron tabanlı düşünmekten tam katmanlara ve şimdi de katmanların kalıplarını tekrarlayan bloklara geçerek daha soyut bir hale gelmişti.

[92] https://discuss.d2l.ai/t/76

Blokları kullanma fikri ilk olarak Oxford Üniversitesi'ndeki Görsel Geometri Grubu[93]'nun (Visual Geometry Group - VGG), kendi adını taşıyan *VGG* ağında ortaya çıkmıştır. Bu tekrarlanan yapıları, döngüler ve alt programlar kullanarak herhangi bir modern derin öğrenme çerçevesi ile kodda uygulamak kolaydır.

7.2.1 VGG Blokları

Klasik CNN'lerin temel yapı taşı aşağıdakilerin bir dizisidir: (i) Çözünürlüğü korumak için dolgulu bir evrişimli katman, (ii) ReLU gibi bir doğrusal olmayan işlev, (iii) Maksimum ortaklama katmanı gibi bir ortaklama katmanı. Bir VGG bloğu, uzamsal örnek seyreltme için bir maksimum ortaklama katmanı izleyen bir evrişimli katman dizisinden oluşur. Orijinal VGG makalesinde (Simonyan and Zisserman, 2014), yazarlar 3×3 çekirdeklerle evrişim (yüksekliği ve genişliği aynı tutarak) ve 2 birim uzun adımlı 2×2 maksimum ortaklama (her bloktan sonra çözünürlüğü yarıya indirerek) kullandılar. Aşağıdaki kodda, bir VGG bloğu uygulamak için `vgg_block` adlı bir işlev tanımlıyoruz.

İşlev, `num_convs` evrişim katmanlarının sayısına, `in_channels` girdi kanallarının sayısına ve `out_channels` çıktı kanallarının sayısına karşılık gelen üç argüman alır.

```
import torch
from torch import nn
from d2l import torch as d2l


def vgg_block(num_convs, in_channels, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(nn.Conv2d(in_channels, out_channels,
                                kernel_size=3, padding=1))
        layers.append(nn.ReLU())
        in_channels = out_channels
    layers.append(nn.MaxPool2d(kernel_size=2,stride=2))
    return nn.Sequential(*layers)
```

[93] http://www.robots.ox.ac.uk/~vgg/

7.2.2 VGG Ağı

AlexNet ve LeNet gibi, VGG ağı iki kısma bölünebilir: Birincisi çoğunlukla evrişim ve ortaklama katmanlarından ve ikincisi tam bağlı katmanlardan oluşur. Bu Fig. 7.2.1 içinde tasvir edilmiştir.

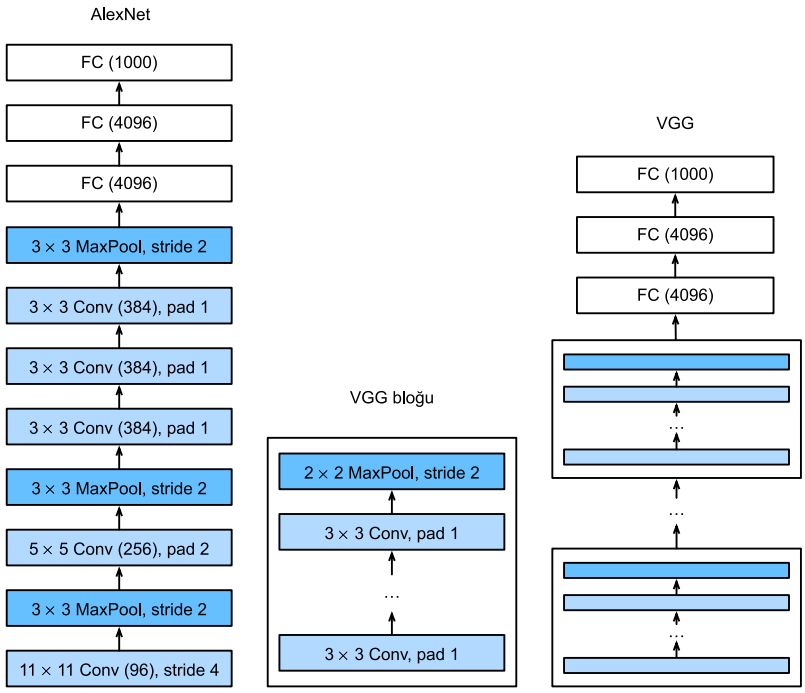


Fig. 7.2.1: AlexNet'ten yapı bloklarından tasarlanmış VGG'ye

Ağın evrişimli kısmı, Fig. 7.2.1 içindeki gösterilen (`vgg_block` işlevinde de tanımlanmıştır) birkaç VGG bloğunu arka arkaya bağlar. Aşağıdaki `conv_arch` değişken, her biri iki değer içeren bir çokuzlu (blok başına bir) listesinden oluşur: Evrişimli katmanların sayısı ve çıktı kanallarının sayısı, ki tam olarak `vgg_block` işlevini çağırmak için gerekli argümanlardır. VGG ağının tam bağlı kısmı AlexNet'te kapsananla aynıdır.

Orijinal VGG ağında, ilk ikisinin her birinin bir evrişimli tabakaya sahip olduğu ve sonraki üçünün her birinin iki evrişimli katman içerdiği 5 evrişimli blok vardı. İlk blokta 64 çıktı kanalı vardır ve sonraki her blok, bu sayı 512'ye ulaşıncaya kadar çıktı kanalı sayısını iki katına çıkarır. Bu ağ 8 evrişimli katman ve 3 tam bağlı katman kullandığından, genellikle VGG-11 olarak adlandırılır.

```
conv_arch = ((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))
```

Aşağıdaki kodda VGG-11'i uygulanıyor. Bu, `conv_arch` üzerinde bir -for- döngüsü yürütme gibi basit bir konudur.

```
def vgg(conv_arch):
    conv_blks = []
    in_channels = 1
    # Evrişimli kısım
    for (num_convs, out_channels) in conv_arch:
        conv_blks.append(vgg_block(num_convs, in_channels, out_channels))
        in_channels = out_channels
```

(continues on next page)

(continued from previous page)

```
    return nn.Sequential(
        *conv_blks, nn.Flatten(),
        # Tam bağlı kısım
        nn.Linear(out_channels * 7 * 7, 4096), nn.ReLU(), nn.Dropout(0.5),
        nn.Linear(4096, 4096), nn.ReLU(), nn.Dropout(0.5),
        nn.Linear(4096, 10))

net = vgg(conv_arch)
```

Daha sonra, her katmanın çıktı şeklini gözlemlemek için 224'lik yüksekliğe ve genişliğe sahip tek kanallı bir veri örneği oluşturacağız.

```
X = torch.randn(size=(1, 1, 224, 224))
for blk in net:
    X = blk(X)
    print(blk.__class__.__name__,'output shape:\t',X.shape)
```

```
Sequential output shape:     torch.Size([1, 64, 112, 112])
Sequential output shape:     torch.Size([1, 128, 56, 56])
Sequential output shape:     torch.Size([1, 256, 28, 28])
Sequential output shape:     torch.Size([1, 512, 14, 14])
Sequential output shape:     torch.Size([1, 512, 7, 7])
Flatten output shape:        torch.Size([1, 25088])
Linear output shape:         torch.Size([1, 4096])
ReLU output shape:   torch.Size([1, 4096])
Dropout output shape:        torch.Size([1, 4096])
Linear output shape:         torch.Size([1, 4096])
ReLU output shape:   torch.Size([1, 4096])
Dropout output shape:        torch.Size([1, 4096])
Linear output shape:         torch.Size([1, 10])
```

Gördüğünüz gibi, her blokta yüksekliği ve genişliği yarıya indiriyoruz, nihayet ağın tam bağlı kısmı tarafından işleme için temsilleri düzleştirmeden önce 7'lik bir yüksekliğe ve genişliğe ulaşıyoruz.

7.2.3 Eğitim

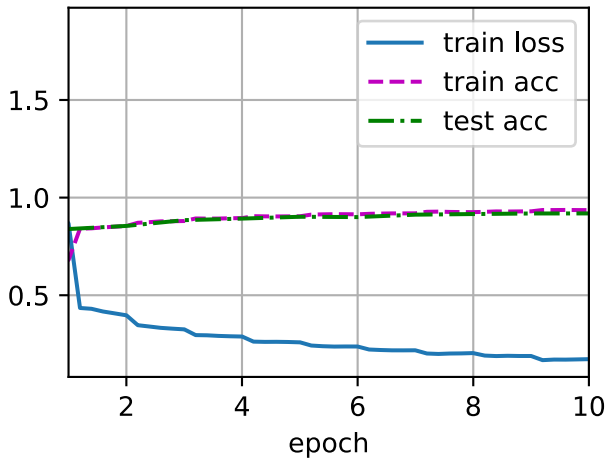
VGG-11 AlexNet'ten hesaplamalı olarak daha ağır olduğundan, daha az sayıda kanala sahip bir ağ oluşturuyoruz. Bu, Fashion-MNIST üzerinde eğitim için fazlasıyla yeterlidir.

```
ratio = 4
small_conv_arch = [(pair[0], pair[1] // ratio) for pair in conv_arch]
net = vgg(small_conv_arch)
```

Biraz daha büyük bir öğrenme hızı kullanmanın haricinde, model eğitim süreci Section 7.1 içindeki AlexNet'e benzerdir.

```
lr, num_epochs, batch_size = 0.05, 10, 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.172, train acc 0.936, test acc 0.919
2570.7 examples/sec on cuda:0
```



7.2.4 Özet

- VGG-11 yeniden kullanılabilir evrişimli bloklar kullanarak bir ağ oluşturur. Farklı VGG modelleri, her bloktaki evrişimli katman ve çıktı kanallarının sayısındaki farklılıklarla tanımlanabilir.
- Blokların kullanımı, ağ tanımının çok sıkıştırılmış temsillerine yol açar. Karmaşık ağların verimli tasarımını sağlar.
- Simonyan ve Ziserman, VGG makalelerinde çeşitli mimarilerle deneyler yaptılar. Özellikle, derin ve dar evrişimlerin (yani 3×3) birkaç katmanının daha az sayıda geniş evrişimli katmandan daha etkili olduğunu buldular.

7.2.5 Alıştırmalar

1. Katmanların boyutlarını yazdırırken 11 yerine sadece 8 sonuç gördük. Kalan 3 katman bilgisi nereye gitti?
2. AlexNet ile karşılaştırıldığında, VGG hesaplama açısından çok daha yavaştır ve ayrıca daha fazla GPU belleğine ihtiyaç duyar. Bunun nedenlerini analiz edin.
3. Fashion-MNIST içindeki imgelerin yüksekliğini ve genişliğini 224'ten 96'ya değiştirmeyi deneyin. Bunun deneyler üzerinde ne etkisi olur?
4. VGG-16 veya VGG-19 gibi diğer yaygın modelleri oluşturmak için VGG makalesindeki (Simonyan and Zisserman, 2014) Tablo 1'e bakın.

Tartışmalar[94]

[94] https://discuss.d2l.ai/t/78

7.3 Ağ İçinde Ağ (Network in Network - NiN)

LeNet, AlexNet ve VGG ortak bir tasarım deseni paylaşır: Öznitelikleri bir dizi evrişim ve ortaklama katmanları aracılığıyla *mekansal* yapıdan faydalanarak çıkar ve daha sonra tam bağlı katmanlar aracılığıyla temsilleri sonradan işle. AlexNet ve VGG tarafından LeNet üzerindeki iyileştirmeler, esas olarak bu sonraki ağların bu iki modülü nasıl genişlettiği ve derinleştirdiği konusunda yatmaktadır. Alternatif olarak, tam bağlı katmanları süreç içinde daha önce kullanmayı hayal edebiliriz. Bununla birlikte, yoğun katmanların dikkatsiz kullanımı, temsilin mekansal yapısını tamamen görmezden gelebilir; *ağ içindeki ağ* (*NiN*) blokları burada bir alternatif sunuyor. Çok basit bir anlayışa dayalı olarak önerildiler: Her piksel için kanallarda ayrı ayrı bir MLP kullanmak (Lin *et al.*, 2013).

7.3.1 NiN Blokları

Evrişimli katmanların girdi ve çıktılarının, örneğe, kanala, yüksekliğe ve genişliğe karşılık gelen eksenlere sahip dört boyutlu tensörlerden oluştuğunu hatırlayın. Ayrıca, tam bağlı katmanların girdi ve çıktılarının tipik olarak örneğe ve özniteliğe karşılık gelen iki boyutlu tensörler olduğunu hatırlayın. NiN'in arkasındaki fikir, her piksel konumuna (her yükseklik ve genişlik için) tam bağlı bir katman uygulamaktır. Ağırlıkları her mekansal konum boyunca bağlarsak, bunu bir 1×1 evrişimli katman (Section 6.4 içinde açıklandığı gibi) veya her piksel konumunda bağımsız olarak hareket eden tam bağlı bir katman olarak düşünebiliriz. Bunu görmenin bir başka yolu da mekansal boyuttaki her elemanın (yükseklik ve genişlik) bir örneğe eşdeğer ve bir kanalın bir özniteliğe eşdeğer olduğunu düşünmektir.

Fig. 7.3.1, VGG ve NiN arasındaki ana yapısal farklılıkları ve bloklarını göstermektedir. NiN bloğu, bir evrişimli katmandan ve ardından ReLU etkinleştirmeleri ile piksel başına tam bağlı katmanlar olarak hareket eden iki 1×1 evrişimli katmandan oluşur. İlk katmanın evrişim penceresi şekli genellikle kullanıcı tarafından ayarlanır. Sonraki pencere şekilleri 1×1'e sabitlenir.

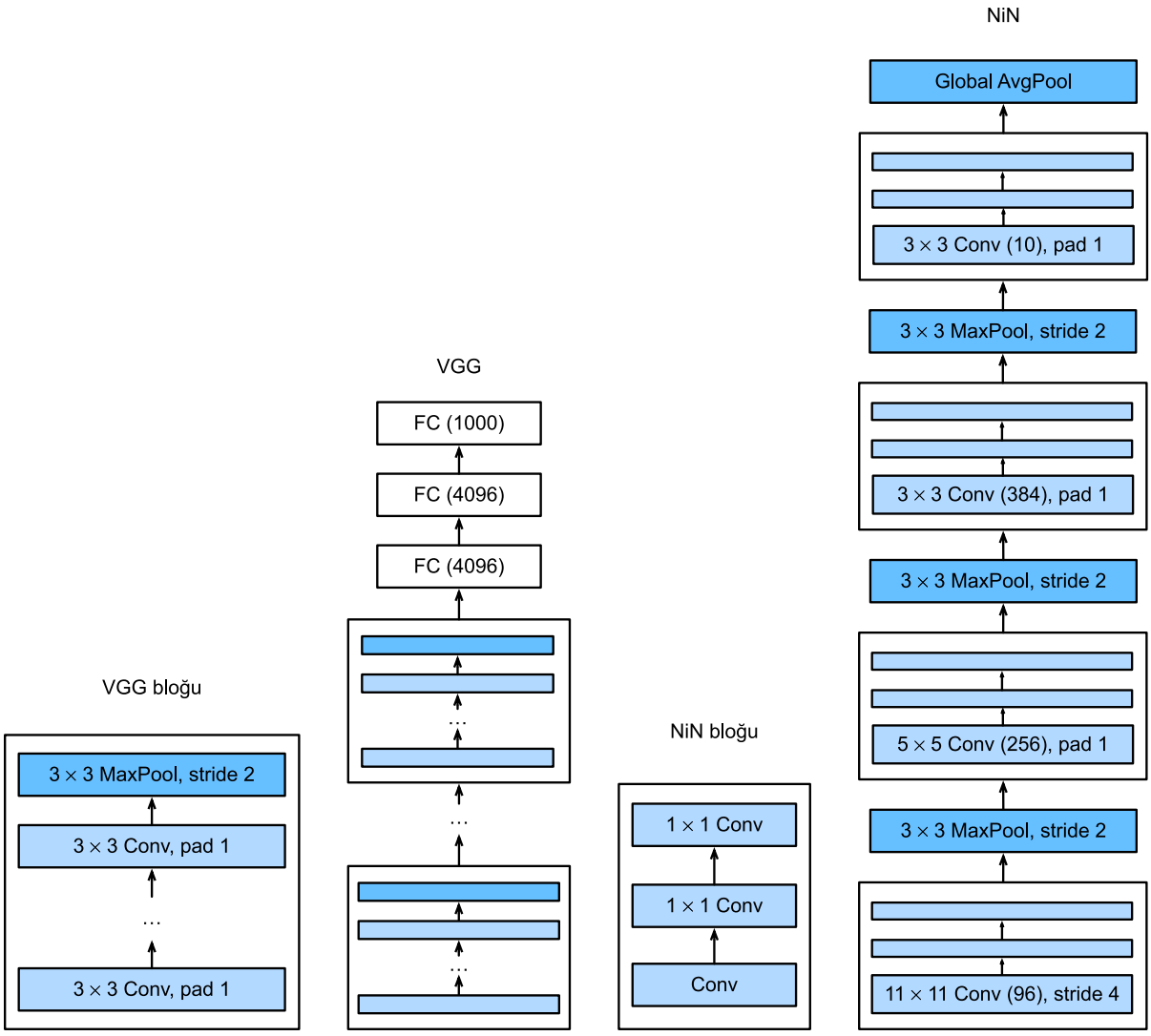


Fig. 7.3.1: VGG and NiN mimarilerinin ve bloklarının karşılaştırılması

```
import torch
from torch import nn
from d2l import torch as d2l


def nin_block(in_channels, out_channels, kernel_size, strides, padding):
    return nn.Sequential(
        nn.Conv2d(in_channels, out_channels, kernel_size, strides, padding),
        nn.ReLU(),
        nn.Conv2d(out_channels, out_channels, kernel_size=1), nn.ReLU(),
        nn.Conv2d(out_channels, out_channels, kernel_size=1), nn.ReLU())
```

7.3.2 NiN Modeli

Orijinal NiN ağı, AlexNet'ten kısa bir süre sonra önerildi ve açıkçası ondan biraz ilham almıştır. NiN, 11×11, 5×5 ve 3×3 pencere şekilleri ile evrişimli katmanlar kullanır ve karşılık gelen çıktı kanalı sayıları AlexNet'teki ile aynıdır. Her NiN bloğunu, 2'lik bir adım ve 3×3'lük bir pencere şekli ile bir maksimum ortaklama katmanı izler.

NiN ve AlexNet arasındaki önemli bir fark, NiN'in tam bağlı katmanlardan kaçınmasıdır. Bunun yerine NiN, etiket sınıflarının sayısına eşit sayıda çıktı kanalı içeren bir NiN bloğu kullanır ve onun ardından gelen *global* ortalama ortaklama katmanı ile bir logit vektörü oluşturur. NiN'in tasarımının bir avantajı, gerekli model parametrelerinin sayısını önemli ölçüde azaltmasıdır. Bununla birlikte, pratikte, bu tasarım bazen artan model eğitim süresi gerektirir.

```
net = nn.Sequential(
    nin_block(1, 96, kernel_size=11, strides=4, padding=0),
    nn.MaxPool2d(3, stride=2),
    nin_block(96, 256, kernel_size=5, strides=1, padding=2),
    nn.MaxPool2d(3, stride=2),
    nin_block(256, 384, kernel_size=3, strides=1, padding=1),
    nn.MaxPool2d(3, stride=2),
    nn.Dropout(0.5),
    # 10 etiket sınıfı var
    nin_block(384, 10, kernel_size=3, strides=1, padding=1),
    nn.AdaptiveAvgPool2d((1, 1)),
    # Dört boyutlu çıktıyı (toplu iş boyutu, 10)
    # şeklinde iki boyutlu çıktıya dönüştür
    nn.Flatten())
```

Her bloğun çıktı şeklini görmek için bir veri örneği oluşturuyoruz.

```
X = torch.rand(size=(1, 1, 224, 224))
for layer in net:
    X = layer(X)
    print(layer.__class__.__name__,'cikti sekli:\t', X.shape)
```

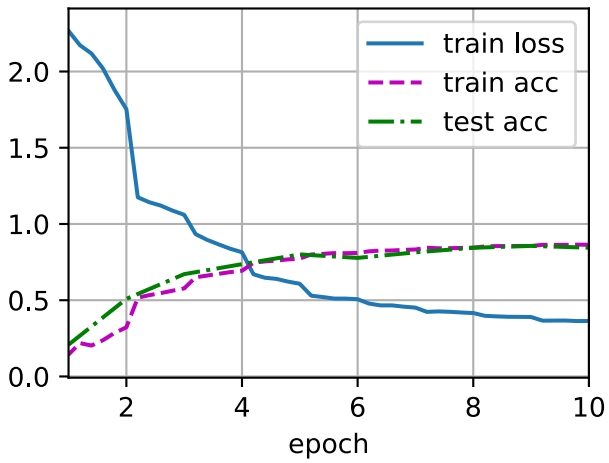
```
Sequential cikti sekli:      torch.Size([1, 96, 54, 54])
MaxPool2d cikti sekli:       torch.Size([1, 96, 26, 26])
Sequential cikti sekli:      torch.Size([1, 256, 26, 26])
MaxPool2d cikti sekli:       torch.Size([1, 256, 12, 12])
Sequential cikti sekli:      torch.Size([1, 384, 12, 12])
MaxPool2d cikti sekli:       torch.Size([1, 384, 5, 5])
Dropout cikti sekli:         torch.Size([1, 384, 5, 5])
Sequential cikti sekli:      torch.Size([1, 10, 5, 5])
AdaptiveAvgPool2d cikti sekli:       torch.Size([1, 10, 1, 1])
Flatten cikti sekli:         torch.Size([1, 10])
```

7.3.3 Eğitim

Daha önce olduğu gibi modeli eğitmek için Fashion-MNIST'i kullanıyoruz. NiN'in eğitimi AlexNet ve VGG'ninkine benzerdir.

```
lr, num_epochs, batch_size = 0.1, 10, 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.364, train acc 0.864, test acc 0.845
3180.3 examples/sec on cuda:0
```



7.3.4 Özet

- NiN, evrişimli bir tabaka ve birden fazla 1×1 evrişimli katmanlardan oluşan bloklar kullanır. Bu, piksel başına daha fazla doğrusal olmayan işleve izin vermek için evrişimli yığın içinde kullanılabilir.
- NiN, tam bağlı katmanları kaldırır ve onları kanal sayısını istenen çıktı sayısına indirdikten sonra (örneğin, Fashion-MNIST için 10) küresel ortalama ortaklama ile yer değiştirir (yani, tüm konumlar üzerinden toplar).
- Tam bağlı katmanların çıkarılması aşırı öğrenmeyi azaltır. NiN önemli ölçüde daha az parametreye sahiptir.
- NiN tasarımı, müteakip birçok CNN tasarımını etkiledi.

7.3.5 Alıştırmalar

1. Sınıflandırma doğruluğunu artırmak için hiper parametreleri ayarlayın.
2. NiN bloğunda neden iki 1×1 evrişimli katman var? Bunlardan birini çıkarın, deneysel olguları gözlemleyin ve çözümleyin.
3. NiN için kaynak kullanımını hesaplayın.
    1. Parametrelerin sayısı nedir?
    2. Hesaplama miktarı nedir?
    3. Eğitim sırasında ihtiyaç duyulan bellek miktarı nedir?
    4. Tahmin sırasında gereken bellek miktarı nedir?
4. $384 \times 5 \times 5$ gösterimini bir adımda $10 \times 5 \times 5$ gösterimine indirgeme ile ilgili olası sorunlar nelerdir?

Tartışmalar[95]

7.4 Paralel Bitiştirmeli Ağlar (GoogLeNet)

2014'te, *GoogLeNet* ImageNet Yarışması'nı kazandı ve NiN'in güçlü yanlarını ve (Szegedy *et al.*, 2015)'ün tekrarlanan bloklarının faydalarını birleştiren bir yapı önerdi. Makalenin odak noktası, hangi büyüklükteki evrişim çekirdeklerinin en iyi olduğu sorusunu ele almaktı. Sonuçta, önceki popüler ağlar 1×1 gibi küçük ve 11×11 kadar büyük seçimler kullandı. Bu makaledeki bir öngörü, bazen çeşitli boyutlarda çekirdeklerin bir kombinasyonunu kullanmanın avantajlı olabileceğiydi. Bu bölümde, orijinal modelin biraz basitleştirilmiş bir versiyonunu sunarak GoogLeNet'i tanıtacağız: Eğitimi kararlı hale getirmek için eklenen ancak artık daha iyi eğitim algoritmaları ile gereksiz olan birkaç geçici özelliği atlıyoruz.

7.4.1 Başlangıç (Inception) Blokları

GoogLeNet'teki temel evrişimli bloğa, viral bir mizah unsuru (meme) başlatan *Başlangıç (Inception)* ("Daha derine gitmemiz gerekiyor") filminden bir alıntı nedeniyle adlandırılmış bir *başlangıç bloğu* denir.

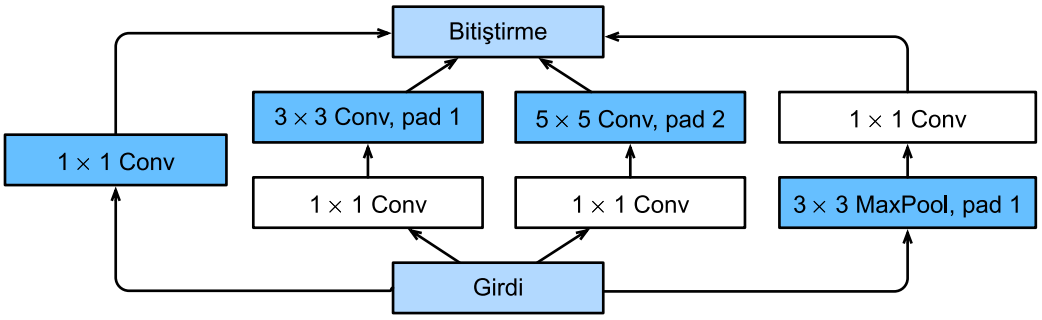


Fig. 7.4.1: Başlangıç bloğu yapısı.

[95] https://discuss.d2l.ai/t/80

Fig. 7.4.1 içinde gösterildiği gibi, başlangıç bloğu dört paralel yoldan oluşur. İlk üç yol, farklı uzamsal boyutlardan bilgi ayıklamak için 1×1, 3×3 ve 5×5 pencere boyutlarına sahip evrişimli katmanlar kullanır. Orta iki yol, kanalların sayısını azaltmak ve modelin karmaşıklığını azaltmak için girdi üzerinde bir 1×1 evrişim gerçekleştirir. Dördüncü yol, 3×3 maksimum ortaklama katmanı kullanır ve onu ardından kanal sayısını değiştiren 1×1 evrişimli katman izler. Dört yol, girdi ve çıktıya aynı yüksekliği ve genişliği vermek için uygun dolguyu kullanır. Son olarak, her yol boyunca çıktılar kanal boyutu boyunca bitiştirilir ve bloğun çıktısını oluşturur. Başlangıç bloğunun yaygın olarak ayarlanan hiper parametreleri, katman başına çıktı kanallarının sayısıdır.

```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l


class Inception(nn.Module):
    # `c1`--`c4` her yoldaki çıktı kanallarının sayısıdır
    def __init__(self, in_channels, c1, c2, c3, c4, **kwargs):
        super(Inception, self).__init__(**kwargs)
        # Yol 1, tek bir 1 x 1 evrişimli katmandır
        self.p1_1 = nn.Conv2d(in_channels, c1, kernel_size=1)
        # Yol 2, 1 x 1 evrişimli katmanı ve ardından
        # 3 x 3 evrişimli katmandır
        self.p2_1 = nn.Conv2d(in_channels, c2[0], kernel_size=1)
        self.p2_2 = nn.Conv2d(c2[0], c2[1], kernel_size=3, padding=1)
        # Yol 3, 1 x 1 evrişimli katmanı ve ardından
        # 5 x 5 evrişimli katmandır
        self.p3_1 = nn.Conv2d(in_channels, c3[0], kernel_size=1)
        self.p3_2 = nn.Conv2d(c3[0], c3[1], kernel_size=5, padding=2)
        # Yol 4, 3 x 3 maksimum ortaklama katmanı ve
        # ardından 1 x 1 evrişimli katmandır
        self.p4_1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
        self.p4_2 = nn.Conv2d(in_channels, c4, kernel_size=1)

    def forward(self, x):
        p1 = F.relu(self.p1_1(x))
        p2 = F.relu(self.p2_2(F.relu(self.p2_1(x))))
        p3 = F.relu(self.p3_2(F.relu(self.p3_1(x))))
        p4 = F.relu(self.p4_2(self.p4_1(x)))
        # Concatenate the outputs on the channel dimension
        return torch.cat((p1, p2, p3, p4), dim=1)
```

Bu ağın neden bu kadar iyi çalıştığına dair sezgi kazanmak için filtrelerin kombinasyonunu göz önünde bulundurun. İmgeyi çeşitli filtre boyutlarında tarıyorlar. Bu, farklı boyutlardaki ayrıntıların farklı boyutlardaki filtrelerle verimli bir şekilde tanınabileceği anlamına gelir. Aynı zamanda, farklı filtreler için farklı miktarlarda parametre tahsis edebiliriz.

7.4.2 GoogLeNet Modeli

Fig. 7.4.2 içinde gösterildiği gibi, GoogLeNet tahminlerini oluşturmak için toplam 9 başlangıç bloğu ve global ortalama ortaklamadan oluşan bir yığın kullanır. Başlangıç blokları arasındaki maksimum ortaklama boyutsallığı azaltır. İlk modül AlexNet ve LeNet'e benzer. Blokların yığını VGG'den devralınır ve küresel ortalama ortaklama ile sondaki tam bağlı katman yığınından kaçınır.

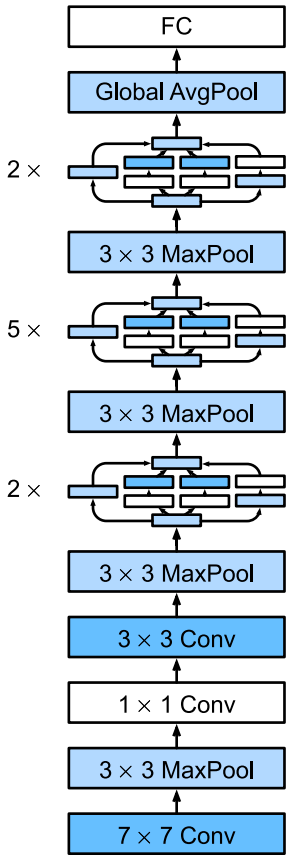


Fig. 7.4.2: GoogLeNet mimarisi.

Artık GoogLeNet'i parça parça uygulayabiliriz. İlk modül 64 kanallı bir 7×7 evrişimli katman kullanır.

```
b1 = nn.Sequential(nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),
                   nn.ReLU(),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

İkinci modül iki evrişimli katman kullanır: Birincisi, 64 kanallı 1×1 evrişimli tabaka, daha sonra kanal sayısını üçe katlayan bir 3×3 evrişimli tabaka. Bu, başlangıç bloğundaki ikinci yola karşılık gelir.

```
b2 = nn.Sequential(nn.Conv2d(64, 64, kernel_size=1),
                   nn.ReLU(),
                   nn.Conv2d(64, 192, kernel_size=3, padding=1),
                   nn.ReLU(),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

Üçüncü modül, iki bütün başlangıç bloğunu seri olarak bağlar. İlk başlangıç bloğunun çıktı kanallarının sayısı $64+128+32+32 = 256$'dır ve dört yol arasındaki çıktı kanalı oranı $64 : 128 : 32 : 32 = 2 : 4 : 1 : 1$'dir. İkinci ve üçüncü yollar önce girdi kanallarının sayısını sırasıyla $96/192 = 1/2$ ve $16/192 = 1/12$'ye düşürür ve daha sonra ikinci evrişimli tabakayı bağlar. İkinci başlangıç bloğunun çıktı kanallarının sayısı $128 + 192 + 96 + 64 = 480$'e yükseltilir ve dört yol arasındaki çıktı kanalı oranı $128 : 192 : 96 : 64 = 4 : 6 : 3 : 2$'dir. İkinci ve üçüncü yollar önce girdi kanalı sayısını sırasıyla $128/256 = 1/2$ ve $32/256 = 1/8$'e düşürür.

```
b3 = nn.Sequential(Inception(192, 64, (96, 128), (16, 32), 32),
                   Inception(256, 128, (128, 192), (32, 96), 64),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

Dördüncü modül daha karmaşıktır. Beş başlangıç bloğunu seri olarak bağlar ve sırasıyla $192 + 208 + 48 + 64 = 512$, $160 + 224 + 64 + 64 = 512$, $128 + 256 + 64 + 64 = 512$, $112 + 288 + 64 + 64 = 528$ ve $256 + 320 + 128 + 128 = 832$ çıktı kanalına sahiptir. Bu yollara atanan kanalların sayısı üçüncü modüldekine benzerdir: En fazla kanal sayısına sahip 3×3 evrişimli tabaka ile ikinci yol, onu izleyen sadece 1×1 evrişimli tabaka ile ilk yol, 5×5 evrişimli tabaka ile üçüncü yol ve 3×3 maksimum biriktirme katmanı ile dördüncü yol. İkinci ve üçüncü yollar orana göre önce kanal sayısını azaltacaktır. Bu oranlar farklı başlangıç bloklarında biraz farklıdır.

```
b4 = nn.Sequential(Inception(480, 192, (96, 208), (16, 48), 64),
                   Inception(512, 160, (112, 224), (24, 64), 64),
                   Inception(512, 128, (128, 256), (24, 64), 64),
                   Inception(512, 112, (144, 288), (32, 64), 64),
                   Inception(528, 256, (160, 320), (32, 128), 128),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

Beşinci modül, $256 + 320 + 128 + 128 = 832$ ve $384 + 384 + 128 + 128 = 1024$ çıktı kanallarına sahip iki başlangıç bloğu içerir. Her yola atanan kanal sayısı, üçüncü ve dördüncü modüllerdeki kanallarla aynıdır, ancak belirli değerlerde farklılık gösterir. Beşinci bloğu çıktı katmanının takip edildiğine dikkat edilmelidir. Bu blok, her kanalın yüksekliğini ve genişliğini NiN'de olduğu gibi 1'e değiştirmek için küresel ortalama ortaklama katmanını kullanır. Son olarak, çıktıyı iki boyutlu bir diziye dönüştürüyoruz ve ardından çıktı sayısı etiket sınıflarının sayısı olan tam bağlı bir katman oluşturuyoruz.

```
b5 = nn.Sequential(Inception(832, 256, (160, 320), (32, 128), 128),
                   Inception(832, 384, (192, 384), (48, 128), 128),
                   nn.AdaptiveAvgPool2d((1,1)),
                   nn.Flatten())

net = nn.Sequential(b1, b2, b3, b4, b5, nn.Linear(1024, 10))
```

GoogLeNet modeli hesaplama açısından karmaşıktır, bu nedenle VGG'deki gibi kanal sayısını değiştirmek kolay değildir. Fashion-MNIST üzerinde makul bir eğitim süresine sahip olmak için girdi yüksekliğini ve genişliğini 224'ten 96'ya düşürüyoruz. Bu, hesaplamayı basitleştirir. Çeşitli modüller arasındaki çıktı şeklindeki değişiklikler aşağıda gösterilmiştir.

```
X = torch.rand(size=(1, 1, 96, 96))
for layer in net:
    X = layer(X)
    print(layer.__class__.__name__,'output shape:\t', X.shape)
```

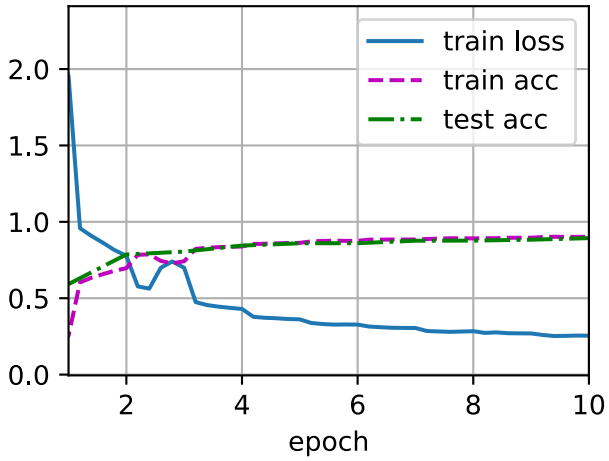
```
Sequential output shape:     torch.Size([1, 64, 24, 24])
Sequential output shape:     torch.Size([1, 192, 12, 12])
Sequential output shape:     torch.Size([1, 480, 6, 6])
Sequential output shape:     torch.Size([1, 832, 3, 3])
Sequential output shape:     torch.Size([1, 1024])
Linear output shape:         torch.Size([1, 10])
```

7.4.3 Eğitim

Daha önce olduğu gibi modelimizi Fashion-MNIST veri kümesini kullanarak eğitiyoruz. Eğitim prosedürünü çağırmadan önce veriyi 96×96 piksel çözünürlüğüne dönüştürüyoruz.

```
lr, num_epochs, batch_size = 0.1, 10, 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.255, train acc 0.902, test acc 0.893
3579.0 examples/sec on cuda:0
```



7.4.4 Özet

- Başlangıç bloğu, dört yolu olan bir alt ağa eşdeğerdir. Farklı pencere şekillerinin ve maksimum biriktirme katmanlarının evrişimli katmanları aracılığıyla bilgileri paralel olarak ayıklar. 1×1 evrişim piksel başına seviyesinde kanal boyutsallığını azaltır. Maksimum biriktirme çözünürlüğü azaltır.
- GoogLeNet, çok iyi tasarlanmış başlangıç bloklarını seri olarak diğer katmanlarla bağlar. Başlangıç bloğunda atanan kanal sayısının oranı, ImageNet veri kümesi üzerinde çok sayıda deney yoluyla elde edilmiştir.
- GoogLeNet ve onun başarılı sürümleri, ImageNet'teki en verimli modellerden biriydi ve daha düşük hesaplama karmaşıklığı ile benzer test doğruluğunu sağladı.

7.4.5 Alıştırmalar

1. GoogLeNet'in birkaç yinelemesi vardır. Uygulamayı ve çalıştırmayı deneyin. Bazıları aşağıda verilmiştir:
   - Section 7.5 içinde de daha sonra açıklandığı gibi bir toplu normalleştirme katmanı (Ioffe and Szegedy, 2015) ekleyin.
   - (Szegedy *et al.*, 2016) Başlangıç bloğu için ayarlamalar yapın.
   - Model düzenlileştirme için etiket yumuşatma kullanın (Szegedy *et al.*, 2016).
   - Section 7.6 içinde de daha sonra açıklandığı gibi artık bağlantı (Szegedy *et al.*, 2017) dahil edin.
2. GoogLeNet'in çalışması için minimum imge boyutu nedir?
3. AlexNet, VGG ve NiN model parametre boyutlarını GoogLeNet ile karşılaştırın. Son iki ağ mimarisi, model parametre boyutunu önemli ölçüde nasıl azaltır?

Tartışmalar[96]

7.5 Toplu Normalleştirme

Derin sinir ağlarını eğitmek zordur. Üstelik makul bir süre içinde yakınsamalarını sağlamak çetrefilli olabilir. Bu bölümde, derin ağların (Ioffe and Szegedy, 2015) yakınsamasını sürekli olarak hızlandıran popüler ve etkili bir teknik olan *toplu normalleştirme*yi tanıtıyoruz. Daha sonra Section 7.6 içinde kapsanan artık bloklarla birlikte toplu normalleştirme, uygulayıcıların 100'den fazla katmanlı ağları rutin olarak eğitmelerini mümkün kılmıştır.

7.5.1 Derin Ağları Eğitme

Toplu normalleştirmeyi anlamak için, özellikle makine öğrenmesi modellerini ve sinir ağlarını eğitirken ortaya çıkan birkaç pratik zorluğu gözden geçirelim.

İlk olarak, veri ön işleme ile ilgili seçimler genellikle nihai sonuçlarda muazzam bir fark yaratmaktadır. Ev fiyatlarını tahmin etmek için MLP'i uygulamamızı hatırlayın (Section 4.10). Gerçek verilerle çalışırken ilk adımımız, girdi özniteliklerimizin her birinin sıfır ortalaması ve bir varyansı olması için standartlaştırmaktı. Sezgisel olarak, bu standartleştirme eniyileyicilerimizle uyumlu olur, çünkü parametreleri benzer ölçekte bir *önsel dağılıma* yerleştirir.

İkincisi, tipik bir MLP veya CNN için, eğittiğimiz gibi, ara katmanlardaki değişkenler (örneğin, MLP'deki afin dönüşüm çıktıları), geniş çapta değişen büyüklüklere sahip değerler alabilir: Model parametrelerinde hem girdiden çıktıya kadar olan katmanlar boyunca, hem de aynı katmandaki birimler arasında ve zaman içinde yaptığımız güncellemeler nedeniyle. Toplu normalleştirmenin mucitleri, bu tür değişkenlerin dağılımındaki bu kaymanın ağın yakınsamasını engelleyebileceğini gayri resmi olarak önerdi. Sezgisel olarak, bir katmanın değişken değerleri başka bir katmandakilerin 100 katı kadarsa, bunun öğrenme oranlarında telafi edici ayarlamaları gerektirebileceğini varsayıyoruz.

Üçüncü olarak, daha derin ağlar karmaşıktır ve aşırı öğrenmeye yatkındır. Bu, düzenlileştirmenin daha kritik hale geldiği anlamına gelir.

[96] https://discuss.d2l.ai/t/82

Toplu normalleştirme bireysel katmanlara uygulanır (isteğe bağlı olarak, hepsi için) ve şu şekilde çalışır: Her eğitim yinelemesinde, ilk olarak, ortalamalarını çıkararak ve standart sapmaları ile bölerek (toplu normalleştirme), ki her ikisi de mevcut minigrup istatistiklerinden tahmin edilir, girdileri normalleştiririz. Daha sonra, bir ölçek katsayısı ve bir ölçek ofseti (sabit kaydırma değeri) uyguluyoruz. Tam olarak, *toplu normalleştirme* adı *toplu* istatistiklerine dayanan bu *normalleştirme*den kaynaklanmaktadır.

Boyutu 1 olan minigruplarla toplu normalleştirmeyi uygulamaya çalışırsak, hiçbir şey öğrenemeyeceğimizi unutmayın. Bunun nedeni, ortalamayı çıkardıktan sonra, her gizli birimin 0 değerini alacak olmasıdır! Tahmin edebileceğiniz gibi, ki bu nedenle toplu normalleştirmeye koca bir bölümü ayırıyoruz, yeterince büyük minigruplarla, yaklaşım etkili ve istikrarlı olduğunu kanıtlıyor. Buradan alınması gereken esas, toplu normalleştirmeyi uygularken, grup boyutunun seçiminin toplu normalleştirmesiz durumda olduğundan daha önemli olabileceğidir.

Biçimsel olarak, bir minigrup olan $\mathcal{B}$'ye dahil olan $\mathbf{x} \in \mathcal{B}$, toplu normalleştirmeye (BN) girdi olursa, toplu normalleştirme $\mathbf{x}$ aşağıdaki ifadeye dönüşür:

$$\mathrm{BN}(\mathbf{x}) = \boldsymbol{\gamma} \odot \frac{\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}}}{\hat{\boldsymbol{\sigma}}_{\mathcal{B}}} + \boldsymbol{\beta}. \tag{7.5.1}$$

(7.5.1) denkleminde, $\hat{\boldsymbol{\mu}}_{\mathcal{B}}$ örneklem ortalaması ve $\hat{\boldsymbol{\sigma}}_{\mathcal{B}}$ minigrup $\mathcal{B}$'nın örneklem standart sapmasıdır. Standartlaştırma uygulandıktan sonra, ortaya çıkan minigrup sıfır ortalama ve birim varyansa sahiptir. Birim varyans seçimi (diğer bazı sihirli sayılara karşı) keyfi bir seçim olduğundan, genel olarak eleman-yönlü *ölçek parametresi* $\boldsymbol{\gamma}$'yı ve *kayma parametresi* $\boldsymbol{\beta}$'yı dahil ederiz ve onlar $\mathbf{x}$ ile aynı şekle sahiptirler. $\boldsymbol{\gamma}$ ve $\boldsymbol{\beta}$'in diğer model parametreleriyle birlikte öğrenilmesi gereken parametreler olduğunu unutmayın.

Sonuç olarak, ara katmanlar için değişken büyüklükleri eğitim sırasında ıraksamaz, çünkü toplu normalleştirme bunları belirli bir ortalama ve boyuta ($\hat{\boldsymbol{\mu}}_{\mathcal{B}}$ ve $\hat{\boldsymbol{\sigma}}_{\mathcal{B}}$ üzerinden) aktif olarak ortalar ve yeniden ölçeklendirir. Uygulayıcının sezgi veya bilgeliğinin bir parçası, toplu normalleştirmenin daha saldırgan öğrenme oranlarına izin vermesi gibi görünmesidir.

Resmi olarak, (7.5.1) denklemindeki $\hat{\boldsymbol{\mu}}_{\mathcal{B}}$ ve $\hat{\boldsymbol{\sigma}}_{\mathcal{B}}$'yi aşağıdaki gibi hesaplıyoruz:

$$\begin{aligned} \hat{\boldsymbol{\mu}}_{\mathcal{B}} &= \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} \mathbf{x}, \\ \hat{\boldsymbol{\sigma}}_{\mathcal{B}}^2 &= \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} (\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}})^2 + \epsilon. \end{aligned} \tag{7.5.2}$$

Deneysel varyans tahmininin kaybolabileceği durumlarda bile sıfıra bölmeyi denemediğimizden emin olmak için varyans tahminine küçük bir sabit $\epsilon > 0$ eklediğimizi unutmayın. $\hat{\boldsymbol{\mu}}_{\mathcal{B}}$ ve $\hat{\boldsymbol{\sigma}}_{\mathcal{B}}$ tahminleri, gürültülü ortalama ve varyans tahminleri kullanarak ölçekleme sorununa karşı koymaktadır. Bu gürültücülüğün bir sorun olması gerektiğini düşünebilirsiniz. Anlaşılacağı gibi, bu aslında faydalıdır.

Bunun derin öğrenmede yinelenen bir tema olduğu ortaya çıkıyor. Teorik olarak henüz iyi ortaya çıkarılamayan nedenlerden dolayı, eniyilemedeki çeşitli gürültü kaynakları genellikle daha hızlı eğitime ve daha az aşırı öğrenmeye neden olur: Bu varyasyon bir düzenlileştirme biçimi olarak kendini gösteriyor. Bazı ön araştırmalarda, (Teye *et al.*, 2018) ve (Luo *et al.*, 2018), toplu normalleşmenin özelliklerini sırasıyla Bayesian önselleri ve cezaları ile ilişkilendiriyor. Özellikle, bu durum $50 \sim 100$ aralığındaki orta boy minigrup boyutları için toplu normalleştirmenin neden en iyi şekilde çalıştığı bulmacasına biraz ışık tutuyor.

Eğitilmiş bir modeli sabitlerken, ortalama ve varyansı tahmin etmek için tüm veri kümesini kullanmayı tercih edeceğimizi düşünebilirsiniz. Eğitim tamamlandıktan sonra, dahil olduğu gruba

bağlı olarak neden aynı imgenin farklı şekilde sınıflandırılmasını isteyelim? Eğitim sırasında, modelimizi her güncellediğimizde tüm veri örnekleri için ara değişkenler değiştiği için bu mutlak hesaplama uygulanabilir değildir. Bununla birlikte, model eğitildikten sonra, her katmanın değişkenlerinin ortalamalarını ve varyanslarını tüm veri kümesine dayalı olarak hesaplayabiliriz. Aslında bu, toplu normalleştirme kullanan modeller için standart bir uygulamadır ve böylece toplu normalleştirme katmanları *eğitim modu* (minigrup istatistiklerine göre normalleştirme) ve *tahmin modunda* (veri kümesi istatistiklerine göre normalleştirme) farklı şekilde çalışır.

Şimdi toplu normalleştirmenin pratikte nasıl çalıştığına bir göz atmaya hazırız.

7.5.2 Toplu Normalleştirme Katmanları

Tam bağlı katmanlar ve evrişimli katmanlar için toplu normalleştirme uygulamaları biraz farklıdır. Her iki durumu aşağıda tartışıyoruz. Toplu normalleştirme ve diğer katmanlar arasındaki önemli bir farkın olduğunu hatırlayın; toplu normalleştirme bir seferinde bir minigrup üzerinde çalıştığı için, diğer katmanlarına ilerlerken daha önce yaptığımız gibi toplu iş boyutunu göz ardı edemeyiz.

Tam Bağlı Katmanlar

Tam bağlı katmanlara toplu normalleştirme uygularken, orijinal makale, toplu normalleştirmeyi afin dönüşümden sonra ve doğrusal olmayan etkinleştirme işlevinden önce ekler (daha sonraki uygulamalar etkinleştirme işlevlerinden hemen sonra toplu normalleştirme ekleyebilir) (Ioffe and Szegedy, 2015). $\mathbf{x}$ ile tam bağlı katmana girdisi, $\mathbf{W}\mathbf{x} + \mathbf{b}$ afin dönüşümü (ağırlık parametresi $\mathbf{W}$ ve ek girdi parametresi $\mathbf{b}$ ile) ve ϕ ile etkinleştirme fonksiyonunu ifade ederse, tam bağlı bir katman çıktısının, $\mathbf{h}$, toplu normalleştirme etkinleştirilmiş hesaplanmasını aşağıdaki gibi ifade edebiliriz:

$$\mathbf{h} = \phi(\mathrm{BN}(\mathbf{W}\mathbf{x} + \mathbf{b})). \tag{7.5.3}$$

Ortalama ve varyansın dönüşümün uygulandığı *aynı* minigrup üzerinde hesaplandığını hatırlayın.

Evrişimli Katmanlar

Benzer şekilde, evrişimli katmanlarla, evrişimden sonra ve doğrusal olmayan etkinleştirme işlevinden önce toplu normalleştirme uygulayabiliriz. Evrişim birden fazla çıktı kanalı olduğunda, bu kanalların çıktılarının *her biri* için toplu normalleştirme gerçekleştirmemiz gerekir ve her kanalın kendi ölçek ve kayma parametreleri vardır, bunların her ikisi de skalerdir. Minigruplarımızın m örnekleri içerdiğini ve her kanal için evrişim çıktısının p yüksekliği ve q genişliği olduğunu varsayalım. Evrişimli katmanlar için, çıktı kanalı başına $m \cdot p \cdot q$ eleman üzerinde toplu normalleştirmeyi aynı anda gerçekleştiriyoruz. Böylece, ortalama ve varyansı hesaplarken tüm mekansal konumlar üzerinde değerleri biraraya getiririz ve sonuç olarak her mekansal konumdaki değeri normalleştirmek için belirli bir kanal içinde aynı ortalama ve varyansı uygularız.

Tahmin Sırasında Toplu Normalleştirme

Daha önce de belirttiğimiz gibi, toplu normalleştirme genellikle eğitim modunda ve tahmin modunda farklı davranır. İlk olarak, her birinin minigruplardan tahmin edilmesinden kaynaklanan örneklem ortalamasındaki ve örneklem varyansındaki gürültü, modeli eğittikten sonra artık arzu edilen birşey değildir. İkincisi, iş başına toplu normalleştirme istatistiklerini hesaplama lüksüne sahip olmayabiliriz. Örneğin, modelimizi bir seferde bir öngörü yapmak için uygulamamız gerekebilir.

Genellikle, eğitimden sonra, değişken istatistiklerin kararlı tahminlerini hesaplamak ve daha sonra bunları tahmin zamanında sabitlemek için veri kümesinin tamamını kullanırız. Sonuç olarak, toplu normalleştirme, eğitim sırasında ve test zamanında farklı davranır. Hattan düşürmenin de bu özelliği sergilediğini hatırlayın.

7.5.3 Sıfırdan Uygulama

Aşağıda, tensörlerle sıfırdan bir toplu normalleştirme tabakası uyguluyoruz.

```
import torch
from torch import nn
from d2l import torch as d2l


def batch_norm(X, gamma, beta, moving_mean, moving_var, eps, momentum):
    # Mevcut modun eğitim modu mu yoksa tahmin modu mu olduğunu
    # belirlemek için `is_grad_enabled`'ı kullanın
    if not torch.is_grad_enabled():
        # Tahmin modu ise, hareketli ortalama ile elde edilen ortalama
        # ve varyansı doğrudan kullan
        X_hat = (X - moving_mean) / torch.sqrt(moving_var + eps)
    else:
        assert len(X.shape) in (2, 4)
        if len(X.shape) == 2:
            # Tam bağlı bir katman kullanırken, öznitelik
            # boyutundaki ortalamayı ve varyansı hesapla
            mean = X.mean(dim=0)
            var = ((X - mean) ** 2).mean(dim=0)
        else:
            # İki boyutlu bir evrişim katmanı kullanırken, kanal
            # boyutundaki (axis=1) ortalamayı ve varyansı hesaplayın.
            # Burada, yayın işleminin daha sonra gerçekleştirilebilmesi
            # için `X`'in şeklini korumamız gerekiyor.
            mean = X.mean(dim=(0, 2, 3), keepdim=True)
            var = ((X - mean) ** 2).mean(dim=(0, 2, 3), keepdim=True)
        # Eğitim modunda, standardizasyon için mevcut ortalama ve
        # varyans kullanılır
        X_hat = (X - mean) / torch.sqrt(var + eps)
        # Hareketli ortalamayı kullanarak ortalamayı ve varyansı güncelle
        moving_mean = momentum * moving_mean + (1.0 - momentum) * mean
        moving_var = momentum * moving_var + (1.0 - momentum) * var
    Y = gamma * X_hat + beta  # Ölçek ve kaydırma
    return Y, moving_mean.data, moving_var.data
```

Artık uygun bir `BatchNorm` katmanı oluşturabiliriz. Katmanımız `gamma` ölçeği ve `beta` kayması için uygun parametreleri koruyacaktır, bunların her ikisi de eğitim sırasında güncellenecektir. Ayrıca,

katmanımız modelin tahmini sırasında sonraki kullanım için ortalamaların ve varyansların hareketli ortalamalarını koruyacaktır.

Algoritmik ayrıntıları bir kenara bırakırsak, katmanın uygulanmasının altında yatan tasarım desenine dikkat edin. Tipik olarak, matematiği ayrı bir işlevde tanımlarız, varsayalım `batch_norm`. Daha sonra bu işlevselliği, verileri doğru cihazın bağlamına taşıma, gerekli değişkenleri tahsis etme ve ilkleme, hareketli ortalamaları takip etme (ortalama ve varyans için) vb. gibi çoğunlukla kayıt tutma konularını ele alan özel bir katmana kaynaştırıyoruz. Bu model, matematiğin basmakalıp koddan temiz bir şekilde ayrılmasını sağlar. Ayrıca, kolaylık sağlamak için burada girdi şeklini otomatik olarak çıkarma konusunda endişelenmediğimizi, bu nedenle özniteliklerin sayısını belirtmemiz gerektiğini unutmayın. Kaygılanmayın, derin öğrenme çerçevesindeki üst düzey toplu normalleştirme API'leri bunu bizim için halledecek; bunu daha sonra göreceğiz.

```
class BatchNorm(nn.Module):
    # `num_features`: Tam bağlı bir katman için çıktıların sayısı veya
    # evrişimli bir katman için çıktı kanallarının sayısı.
    # `num_dims`: Tam bağlı katman için 2 ve evrişimli katman için 4
    def __init__(self, num_features, num_dims):
        super().__init__()
        if num_dims == 2:
            shape = (1, num_features)
        else:
            shape = (1, num_features, 1, 1)
        # Ölçek parametresi ve kaydırma parametresi (model parametreleri)
        # sırasıyla 1 ve 0 olarak ilklenir
        self.gamma = nn.Parameter(torch.ones(shape))
        self.beta = nn.Parameter(torch.zeros(shape))
        # Model parametresi olmayan değişkenler 0 ve 1 olarak ilklenir.
        self.moving_mean = torch.zeros(shape)
        self.moving_var = torch.ones(shape)

    def forward(self, X):
        # Ana bellekte `X` yoksa, `moving_mean` ve `moving_var`'ı `X`in
        # bulunduğu cihaza kopyalayın
        if self.moving_mean.device != X.device:
            self.moving_mean = self.moving_mean.to(X.device)
            self.moving_var = self.moving_var.to(X.device)
        # Güncellenen `moving_mean` ve `moving_var`'ı kaydedin
        Y, self.moving_mean, self.moving_var = batch_norm(
            X, self.gamma, self.beta, self.moving_mean,
            self.moving_var, eps=1e-5, momentum=0.9)
        return Y
```

7.5.4 LeNet'te Toplu Normalleştirme Uygulaması

Bağlamda `BatchNorm`'un nasıl uygulanacağını görmek için, aşağıda geleneksel bir LeNet modeline (Section 6.6) uyguluyoruz. Toplu normalleştirmenin, evrişimli katmanlardan veya tam bağlı katmanlardan sonra, ancak karşılık gelen etkinleştirme işlevlerinden önce uygulandığını hatırlayın.

```
net = nn.Sequential(
    nn.Conv2d(1, 6, kernel_size=5), BatchNorm(6, num_dims=4), nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2),
    nn.Conv2d(6, 16, kernel_size=5), BatchNorm(16, num_dims=4), nn.Sigmoid(),
```

(continues on next page)

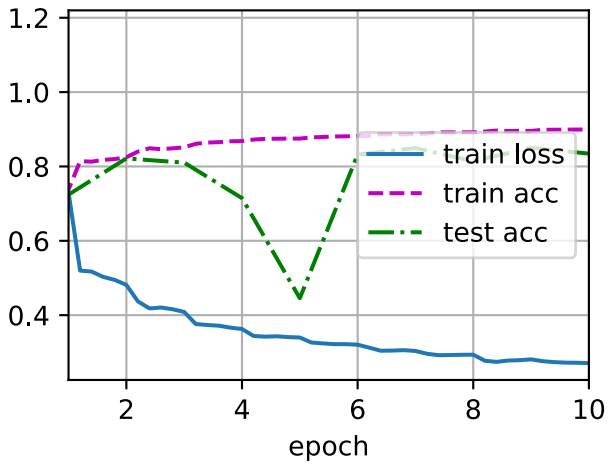
(continued from previous page)

```
    nn.AvgPool2d(kernel_size=2, stride=2), nn.Flatten(),
    nn.Linear(16*4*4, 120), BatchNorm(120, num_dims=2), nn.Sigmoid(),
    nn.Linear(120, 84), BatchNorm(84, num_dims=2), nn.Sigmoid(),
    nn.Linear(84, 10))
```

Daha önce olduğu gibi, ağımızı Fashion-MNIST veri kümesi üzerinde eğiteceğiz. Bu kod, LeNet'i ilk eğittiğimizdeki ile neredeyse aynıdır (Section 6.6). Temel fark, daha büyük öğrenme oranıdır.

```
lr, num_epochs, batch_size = 1.0, 10, 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.271, train acc 0.899, test acc 0.835
35526.6 examples/sec on cuda:0
```



`gamma` ölçek parametresine ve ilk toplu normalleştirme katmanından öğrenilen `beta` kayma parametresine bir göz atalım.

```
net[1].gamma.reshape((-1,)), net[1].beta.reshape((-1,))
```

```
(tensor([4.3948, 1.8766, 3.0088, 2.3228, 2.5096, 2.3995], device='cuda:0',
        grad_fn=<ReshapeAliasBackward0>),
 tensor([ 2.3678,  0.9310, -1.4606, -1.4570, -3.0370,  2.3806], device='cuda:0',
        grad_fn=<ReshapeAliasBackward0>))
```

7.5.5 Kısa Uygulama

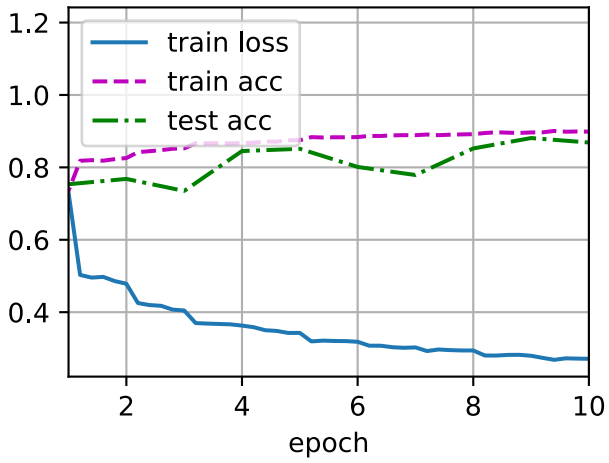
Kendimizi tanımladığımız `BatchNorm` sınıfıyla karşılaştırıldığında, doğrudan derin öğrenme çerçevesinden üst seviye API'lerde tanımlanan `BatchNorm` sınıfını kullanabiliriz. Kod, yukarıdaki uygulamamız ile hemen hemen aynı görünüyor.

```
net = nn.Sequential(
    nn.Conv2d(1, 6, kernel_size=5), nn.BatchNorm2d(6), nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2),
    nn.Conv2d(6, 16, kernel_size=5), nn.BatchNorm2d(16), nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2), nn.Flatten(),
    nn.Linear(256, 120), nn.BatchNorm1d(120), nn.Sigmoid(),
    nn.Linear(120, 84), nn.BatchNorm1d(84), nn.Sigmoid(),
    nn.Linear(84, 10))
```

Aşağıda, modelimizi eğitmek için aynı hiper parametreleri kullanıyoruz. Özel uygulamamız Python tarafından yorumlanılır iken, her zamanki gibi üst seviye API sürümünün kodu C++ veya CUDA için derlendiğinden, çok daha hızlı çalıştığını unutmayın.

```
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.271, train acc 0.899, test acc 0.869
60437.2 examples/sec on cuda:0
```



7.5.6 Tartışma

Sezgisel olarak, toplu normalleştirmenin eniyileme alanını daha pürüzsüz hale getirdiği düşünülmektedir. Bununla birlikte, derin modelleri eğitirken gözlemlediğimiz olgular için kurgusal sezgiler ve gerçek açıklamalar arasında ayrım yaparken dikkatli olmalıyız. İlk etapta daha basit derin sinir ağlarının (MLP'ler ve geleneksel CNN'ler) neden genelleştirildiğini bile bilmediğimizi hatırlayın. Hattan düşürme ve ağırlık sönümü ile bile, görünmeyen verilere genelleme kabiliyetleri geleneksel öğrenme-kuramsal genelleme garantileri ile açıklanamayacak kadar esnek kalırlar.

Toplu normalleştirmeyi öneren orijinal çalışmada, yazarlar, güçlü ve kullanışlı bir araç tanıtmanın yanı sıra, neden çalıştığına dair bir açıklama sundular: *Dahili eşdeğişken kaymasını* (internal

covariate shift) azaltmak. Muhtemelen *dahili eşdeğişken kayması* ile yazarlar, yukarıda ifade edilen sezgiye benzer bir şey ifade ettiler - değişken değerlerinin dağılımının eğitim boyunca değiştiği düşüncesi. Bununla birlikte, bu açıklamayla ilgili iki mesele vardı: i) Bu değişim, *eşdeğişken kaymas*ından çok farklıdır, bir yanlış isimlendirmeye yol açar. ii) Açıklama az belirtilmiş bir sezgi sunar ancak *neden tam olarak bu teknik çalışır* sorusunu titiz bir açıklama isteyen açık bir soru olarak bırakır. Bu kitap boyunca, uygulayıcıların derin sinir ağlarının gelişimine rehberlik etmek için kullandıkları sezgileri aktarmayı amaçlıyoruz. Bununla birlikte, bu rehber sezgileri yerleşik bilimsel gerçeklerden ayırmanın önemli olduğuna inanıyoruz. Sonunda, bu materyale hakim olduğunuzda ve kendi araştırma makalelerinizi yazmaya başladığınızda, teknik iddialar ve önseziler arasında konumunuzu bulmak için net olmak isteyeceksiniz.

Toplu normalleştirmenin başarısını takiben, başarısının *dahili eşdeğişken kayması* açısından açıklanması, teknik yazındaki tartışmalarda ve makine öğrenmesi araştırmasının nasıl sunulacağı konusunda daha geniş bir söylemde defalarca ortaya çıkmıştır. 2017 NeurIPS konferansında Zaman Testi Ödülü'nü kabul ederken verdiği unutulmaz bir konuşmada Ali Rahimi, modern derin öğrenme pratiğini simyaya benzeten bir argümanda odak noktası olarak *dahili eşdeğişken kaymas*ını kullandı. Daha sonra, örnek makine öğrenmesindeki sıkıntılı eğilimleri özetleyen bir konum kağıdında (position paper) ayrıntılı olarak yeniden gözden geçirildi (Lipton and Steinhardt, 2018). Diğer yazarlar toplu normalleştirmenin başarısı için alternatif açıklamalar önerdiler, bazıları toplu normalleştirmenin başarısının bazı yönlerden orijinal makalede (Santurkar *et al.*, 2018) iddia edilenlerin tersi olan davranışları sergilemesinden geldiğini iddia ettiler.

*Dahili eşdeğişken kaymas*ının, teknik makine öğrenmesi yazınında benzer şekilde her yıl yapılan belirsiz iddiaların herhangi birinden daha fazla eleştiriye layık olmadığını not ediyoruz. Muhtemelen, bu tartışmaların odak noktası olarak tınlamasını (rezonans), hedef kitledeki geniş tanınabilirliğine borçludur. Toplu normalleştirme, neredeyse tüm konuşlandırılmış imge sınıflandırıcılarında uygulanan vazgeçilmez bir yöntem olduğunu kanıtlamıştır, tekniği tanıtan makaleye on binlerce atıf kazandırmıştır.

7.5.7 Özet

- Model eğitimi sırasında, toplu normalleştirme, minigrubun ortalama ve standart sapmasını kullanarak sinir ağının ara çıktısını sürekli olarak ayarlar, böylece sinir ağı boyunca her katmandaki ara çıktının değerleri daha kararlı olur.
- Tam bağlı katmanlar ve evrişimli katmanlar için toplu normalleştirme yöntemleri biraz farklıdır.
- Bir hattan düşürme katmanı gibi, toplu normalleştirme katmanları da eğitim modunda ve tahmin modunda farklı hesaplama sonuçlarına sahiptir.
- Toplu normalleştirmenin, öncelikle düzenlileştirme olmak üzere birçok yararlı yan etkisi vardır. Öte yandan, orijinal motivasyondaki dahili eşdeğişken kaymasını azaltma geçerli bir açıklama gibi görünmüyor.

7.5.8 Alıştırmalar

1. Toplu normalleştirmeden önce ek girdi parametresini tam bağlı katmandan veya evrişimli katmandan kaldırabilir miyiz? Neden?
2. Toplu normalleştirme olan ve olmayan LeNet için öğrenme oranlarını karşılaştırın.
    1. Eğitim ve test doğruluğundaki artışı çizin.
    2. Öğrenme oranını ne kadar büyük yapabilirsiniz?
3. Her katmanda toplu normalleştirmeye ihtiyacımız var mı? Deneyle gözlemleyebilir misiniz?
4. Toplu normalleştirme ile hattan düşürmeyi yer değiştirebilir misiniz? Davranış nasıl değişir?
5. `beta` ve `gamma` parametrelerini sabitleyin, sonuçları gözlemleyin ve analiz edin.
6. Toplu normalleştirmenin diğer uygulamalarını görmek amacıyla üst seviye API'lerden `BatchNorm` için çevrimiçi belgelerini gözden geçirin.
7. Araştırma fikirleri: Uygulayabileceğiniz diğer normalleştirme dönüşümleri düşünün? Olasılık integral dönüşümü uygulayabilir misiniz? Tam kerteli kovaryans tahminine ne dersiniz?

Tartışmalar[97]

7.6 Artık Ağlar (ResNet)

Giderek daha derin ağlar tasarlarken, katman eklemenin ağın karmaşıklığını ve ifade gücünü nasıl artırabileceğini anlamak zorunluluk haline gelmektedir. Daha da önemlisi, katmanlar ekleyerek ağları sadece farklı olmaktan çok daha açıklayıcı hale getiren ağları tasarlama yeteneğidir. Biraz ilerleme sağlamak için bir parça matematiğe ihtiyacımız var.

7.6.1 Fonksiyon Sınıfları

Belirli bir ağ mimarisinin (öğrenme hızları ve diğer hiper parametre ayarları ile birlikte) ulaşabileceği işlevlerin sınıfı olan $\mathcal{F}$'i düşünün. Yani, $f \in \mathcal{F}$'in tümü için uygun bir veri kümesi üzerinde eğitim yoluyla elde edilebilecek bazı parametreler (örneğin, ağırlıklar ve ek girdiler) vardır. f^*'in gerçekten bulmak istediğimiz "gerçek" fonksiyon olduğunu varsayalım. Eğer $\mathcal{F}$'te ise, iyi durumdayız ancak tipik olarak bu kadar şanslı olmayacağız. Bunun yerine, $\mathcal{F}$ içinde en iyi aday olan herhangi bir $f^*_\mathcal{F}$ bulmaya çalışacağız. Örneğin, $\mathbf{X}$ öznitelikleri ve $\mathbf{y}$ etiketleri olan bir veri kümesi göz önüne alındığında, aşağıdaki eniyileme sorununu çözerek aday bulmayı deneyebiliriz:

$$f^*_\mathcal{F} \stackrel{\text{def}}{=} \operatorname*{argmin}_f L(\mathbf{X}, \mathbf{y}, f) \text{ öyle ki } f \in \mathcal{F}. \tag{7.6.1}$$

Sadece farklı ve daha güçlü bir $\mathcal{F}'$ mimarisi tasarlarsak daha iyi bir sonuca ulaşacağımızı varsaymak mantıklıdır. Başka bir deyişle, $f^*_{\mathcal{F}'}$'nin $f^*_\mathcal{F}$'dan "daha iyi" olmasını bekleriz. Ancak, eğer $\mathcal{F} \not\subseteq \mathcal{F}'$ ise, bunun gerçekleşmesi gerektiğinin garantisi yoktur. Aslında, $f^*_{\mathcal{F}'}$ daha kötü olabilir. Fig. 7.6.1 içinde gösterildiği gibi, iç içe olmayan işlev sınıfları için, daha büyük bir işlev sınıfı her zaman "gerçek" işlevi f^*'ye yakınlaşmaz. Örneğin, Fig. 7.6.1 solunda $\mathcal{F}_3$'ün f^*'a $\mathcal{F}_1$'den daha yakın

[97] https://discuss.d2l.ai/t/84

olmasına rağmen, $\mathcal{F}_6$ uzaklaşıyor ve karmaşıklığın daha da arttırılmasının f^*'ye mesafeyi azaltabileceğinin garantisi yoktur. İç içe işlev sınıfları ile, $\mathcal{F}_1 \subseteq \ldots \subseteq \mathcal{F}_6$, burada Fig. 7.6.1 sağında gösteriliyor, yukarıda bahsedilen iç içe olmayan işlev sınıfları sorunundan kaçınabilirsiniz.

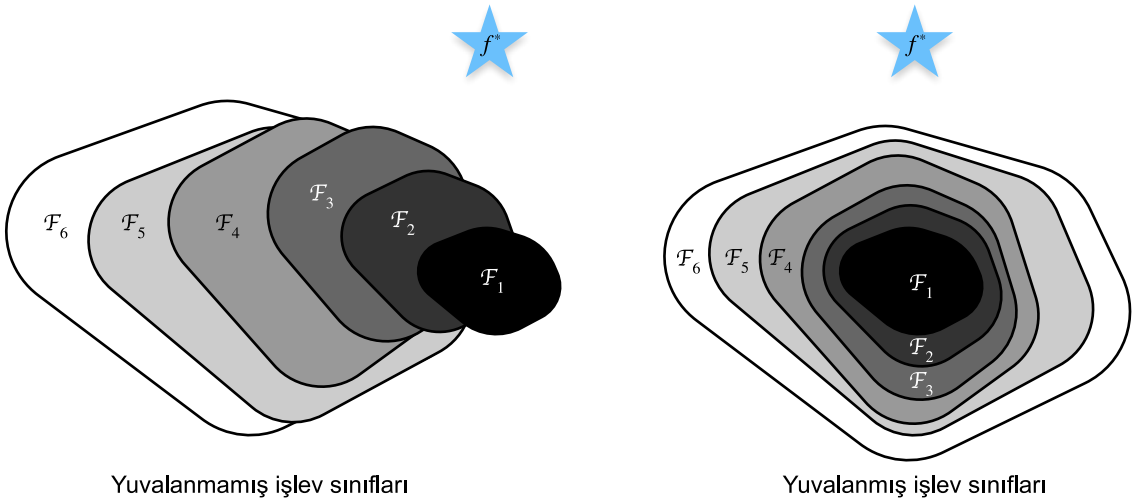


Fig. 7.6.1: İç içe olmayan işlev sınıfları için, daha büyük (alanla gösteriliyor) işlev sınıfı, "doğruluk" işlevine (f^*) yaklaşmayı garanti etmez. Bu, yuvalanmış işlev sınıflarında gerçekleşmez.

Böylece, daha büyük fonksiyon sınıfları daha küçük olanları içeriyorsa, onları artırmanın ağın açıklayıcı gücünü kesinlikle arttırdığını garanti ederiz. Derin sinir ağları için, yeni eklenen katmanı bir birim fonksiyonu $f(\mathbf{x}) = \mathbf{x}$ olarak eğitebilirsek, yeni model orijinal model kadar etkili olacaktır. Yeni model, eğitim veri kümesine uyacak şekilde daha iyi bir çözüm elde edebileceğinden, eklenen katman eğitim hatalarını azaltmayı kolaylaştırabilir.

Bu çok derin bilgisayarla görme modelleri üzerinde çalışırken He ve diğerlerinin (He *et al.*, 2016) üstünde düşündüğü sorudur. Önerilen *artık ağ*ın (*ResNet*) özünde, her ek katmanın kendi elemanlarından biri olarak birim işlevini daha kolaylıkla içermesi gerektiği fikri vardır. Bu fikirler oldukça bilgeydi, fakat şaşırtıcı derecede basit bir çözüme ulaştılar; *artık blok*. ResNet, 2015 yılında ImageNet Büyük Ölçekli Görsel Tanıma Yarışmasını kazandı. Tasarımın derin sinir ağlarının nasıl kurulacağı üzerinde derin bir etkisi oldu.

7.6.2 Artık Blokları

Fig. 7.6.2 içinde gösterildiği gibi, bir sinir ağının yerel bir bölümüne odaklanalım. Girdiyi $\mathbf{x}$ ile belirtelim. Öğrenerek elde etmek istediğimiz altta yatan eşlemenin üstteki etkinleştirme işlevine girdi olarak kullanılacak $f(\mathbf{x})$ olduğunu varsayıyoruz. Fig. 7.6.2 solunda, kesik çizgili kutudaki kısım doğrudan $f(\mathbf{x})$ eşlemesini öğrenmelidir. Sağda, kesik çizgili kutudaki bölümün, artık bloğun adını belirleyen *artık eşleme* $f(\mathbf{x}) - \mathbf{x}$'i öğrenmesi gerekir. Birim eşlemesi $f(\mathbf{x}) = \mathbf{x}$ istenen altta yatan eşleme ise, artık eşlemeyi öğrenmek daha kolaydır: Sadece kesik çizgili kutudaki üst ağırlık katmanının ağırlıklarını ve ek girdilerini (örn. tam bağlı katman ve evrişimli katman) sıfıra itmemiz gerekir. Fig. 7.6.2 içindeki sağdaki şekil, ResNet'in *artık bloğu*nu göstermektedir; burada $\mathbf{x}$ katman girdisini toplama işlemine taşıyan düz çizgiye *artık bağlantı* (veya *kısayol bağlantısı*) denir. Artık bloklarla girdiler, katmanlar arasında kalan bağlantılar üzerinden daha hızlı yayılabilir.

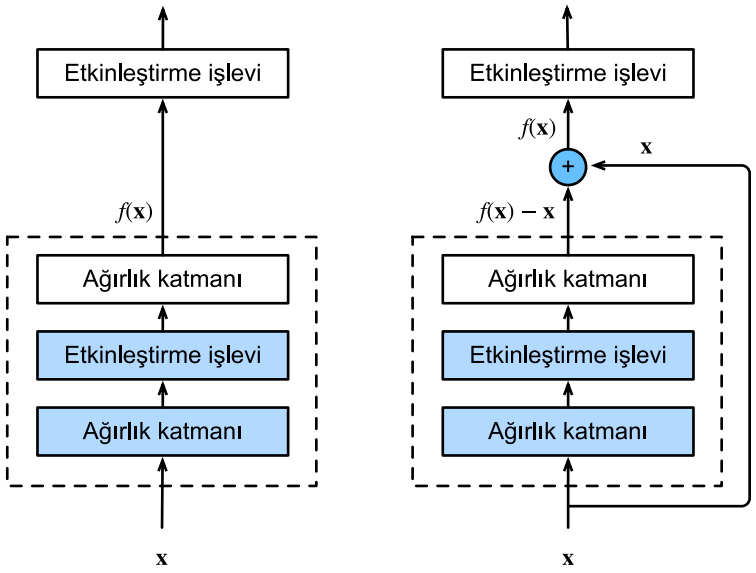


Fig. 7.6.2: Normal bir blok (solda) ve bir artık blok (sağda).

ResNet, VGG'nin 3×3 evrişimli katman tam tasarımını izler. Artık blok, aynı sayıda çıktı kanalına sahip iki 3×3 evrişimli katmana sahiptir. Her bir evrişimli katmanı, bir toplu normalleştirme katmanı ve bir ReLU etkinleştirme işlevi izler. Ardından, bu iki evrişim işlemini atlayıp girdiyi doğrudan son ReLU etkinleştirme işlevinden önce ekleriz. Bu tür bir tasarım, iki evrişimli katmanın çıktısının girdiyle aynı şekle sahip olmasını gerektirir, ancak böylece birlikte toplanabilirler. Kanal sayısını değiştirmek istiyorsak, girdiyi toplama işlemi için istenen şekle dönüştürürken ek bir 1×1 evrişimli katman koymamız gerekir. Aşağıdaki koda bir göz atalım.

```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l


class Residual(nn.Module):  #@save
    """ResNet'in artık bloğu."""
    def __init__(self, input_channels, num_channels,
                 use_1x1conv=False, strides=1):
        super().__init__()
        self.conv1 = nn.Conv2d(input_channels, num_channels,
                               kernel_size=3, padding=1, stride=strides)
        self.conv2 = nn.Conv2d(num_channels, num_channels,
                               kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.Conv2d(input_channels, num_channels,
                                   kernel_size=1, stride=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm2d(num_channels)
        self.bn2 = nn.BatchNorm2d(num_channels)

    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
```

(continues on next page)

(continued from previous page)

```
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        Y += X
        return F.relu(Y)
```

Bu kod iki tür ağ oluşturur: Biri, `use_1x1conv=False`'te ReLU doğrusal olmayanlığını uygulamadan önce çıktı ile girdiyi topladığımız ve diğeri toplamadan önce 1×1 evrişim vasıtasıyla kanalları ve çözünürlüğü ayarladığımız. Fig. 7.6.3 içinde bunu görebiliriz:

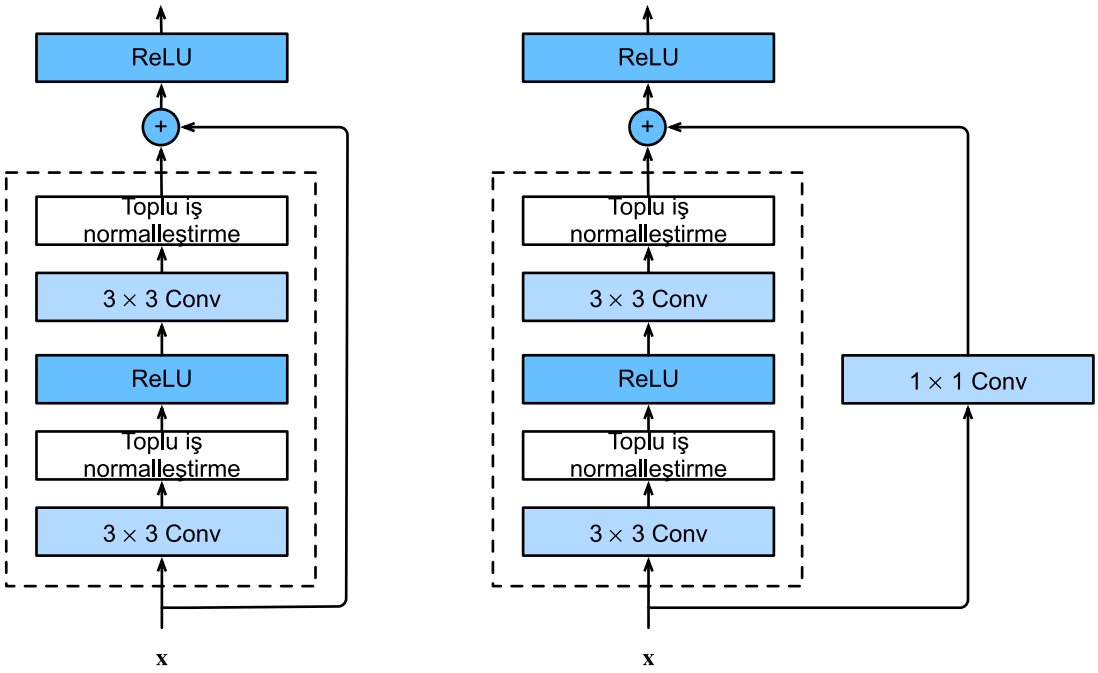


Fig. 7.6.3: 1×1 evrişim içeren ve içermeyen ResNet bloğu.

Şimdi girdinin ve çıktının aynı şekle sahip olduğu bir duruma bakalım.

```
blk = Residual(3,3)
X = torch.rand(4, 3, 6, 6)
Y = blk(X)
Y.shape
```

```
torch.Size([4, 3, 6, 6])
```

Ayrıca çıktı kanallarının sayısını artırırken çıktı yüksekliğini ve genişliğini yarıya indirme seçeneğine de sahibiz.

```
blk = Residual(3,6, use_1x1conv=True, strides=2)
blk(X).shape
```

```
torch.Size([4, 6, 3, 3])
```

7.6.3 ResNet Modeli

ResNet'in ilk iki katmanı, daha önce tarif ettiğimiz GoogLeNet'inkiyle aynıdır: 64 çıktı kanallı ve 2'lik bir uzun adımlı 7×7 evrişimli katmanını 2'lik bir uzun adımlı 3×3 maksimum ortaklama katmanı takip eder. Fark, ResNet'teki her evrişimli katmandan sonra eklenen toplu normalleştirme katmanıdır.

```
b1 = nn.Sequential(nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),
                   nn.BatchNorm2d(64), nn.ReLU(),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

GoogLeNet, başlangıç bloklarından oluşan dört modül kullanır. Bununla birlikte, ResNet, her biri aynı sayıda çıktı kanalına sahip içinde birkaç artık blok kullanan artık bloklardan oluşan dört modül kullanır. İlk modüldeki kanal sayısı, girdi kanallarının sayısı ile aynıdır. 2'lik uzun adımlı maksimum bir ortaklama katmanı kullanıldığından, yüksekliği ve genişliği azaltmak gerekli değildir. Sonraki modüllerin her biri için ilk artık blokta, kanal sayısı önceki modülünkine kıyasla iki katına çıkarılır ve yükseklik ve genişlik yarıya indirilir.

Şimdi, bu modülü uyguluyoruz. İlk modülde özel işlemenin gerçekleştirildiğine dikkat edin.

```
def resnet_block(input_channels, num_channels, num_residuals,
                 first_block=False):
    blk = []
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.append(Residual(input_channels, num_channels,
                                use_1x1conv=True, strides=2))
        else:
            blk.append(Residual(num_channels, num_channels))
    return blk
```

Ardından, tüm modülleri ResNet'e ekliyoruz. Burada, her modül için iki artık blok kullanılır.

```
b2 = nn.Sequential(*resnet_block(64, 64, 2, first_block=True))
b3 = nn.Sequential(*resnet_block(64, 128, 2))
b4 = nn.Sequential(*resnet_block(128, 256, 2))
b5 = nn.Sequential(*resnet_block(256, 512, 2))
```

Son olarak, tıpkı GoogLeNet gibi global bir ortalama ortaklama katmanı ekliyoruz ve ardına tam bağlı çıktı katmanı koyuyoruz.

```
net = nn.Sequential(b1, b2, b3, b4, b5,
                    nn.AdaptiveAvgPool2d((1,1)),
                    nn.Flatten(), nn.Linear(512, 10))
```

Her modülde 4 evrişimli katman vardır (1×1 evrişimli katman hariç). İlk 7×7 evrişimli katman ve son tam bağlı tabaka ile birlikte toplamda 18 katman vardır. Bu nedenle, bu model yaygın olarak ResNet-18 olarak bilinir. Modülde farklı sayıda kanal ve artık blokları yapılandırarak, daha derin 152 katmanlı ResNet-152 gibi farklı ResNet modelleri oluşturabiliriz. ResNet'in ana mimarisi GoogLeNet'e benzer olsa da, ResNet'in yapısı daha basit ve değiştirmesi daha kolaydır. Tüm bu faktörler ResNet'in hızlı ve yaygın kullanımı ile sonuçlandı. Fig. 7.6.4 içinde bütün ResNet-18'i görselleştiriyoruz.

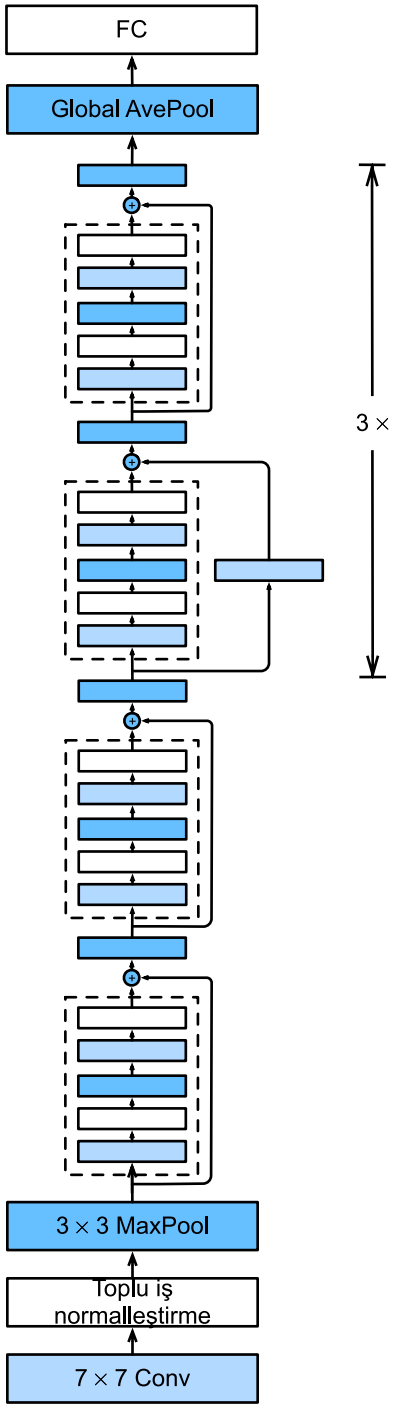


Fig. 7.6.4: The ResNet-18 architecture.

ResNet'i eğitmeden önce, girdi şeklinin ResNet'teki farklı modüllerde nasıl değiştiğini gözlemleyelim. Önceki tüm mimarilerinde olduğu gibi, çözünürlük azalırken, kanal sayısı küresel ortalama ortaklama katmanının tüm öznitelikleri topladığı noktaya kadar artar.

```
X = torch.rand(size=(1, 1, 224, 224))
for layer in net:
    X = layer(X)
    print(layer.__class__.__name__,'output shape:\t', X.shape)
```

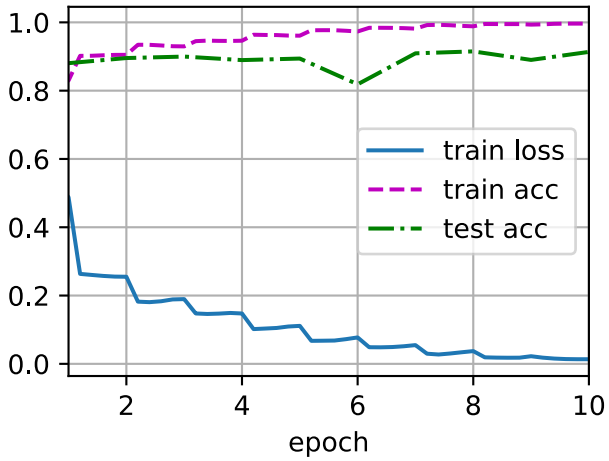
```
Sequential output shape:     torch.Size([1, 64, 56, 56])
Sequential output shape:     torch.Size([1, 64, 56, 56])
Sequential output shape:     torch.Size([1, 128, 28, 28])
Sequential output shape:     torch.Size([1, 256, 14, 14])
Sequential output shape:     torch.Size([1, 512, 7, 7])
AdaptiveAvgPool2d output shape:      torch.Size([1, 512, 1, 1])
Flatten output shape:        torch.Size([1, 512])
Linear output shape:         torch.Size([1, 10])
```

7.6.4 Eğitim

ResNet'i Fashion-MNIST veri kümesi üzerinde eğitiyoruz, tıpkı öncekiler gibi.

```
lr, num_epochs, batch_size = 0.05, 10, 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.014, train acc 0.996, test acc 0.913
4672.4 examples/sec on cuda:0
```



7.6.5 Özet

- İç içe işlev sınıfları arzu edilir. Derin sinir ağlarında bir birim fonksiyonu olarak ek bir katman öğrenmek (bu aşırı bir durum olsa da) kolaylaştırılmalıdır.
- Artık eşleme, ağırlık katmanındaki parametreleri sıfıra iterek birim işlevini daha kolay öğrenebilir.
- Artık blokları ile etkili bir derin sinir ağı eğitebiliriz. Girdiler, katmanlar arasındaki artık bağlantılar üzerinden daha hızlı yayılabilir.
- ResNet, hem evrişimli hem de sıralı içerikli, müteakip derin sinir ağlarının tasarımı üzerinde büyük bir etkiye sahip oldu.

7.6.6 Alıştırmalar

1. Fig. 7.4.1 içindeki başlangıç bloğu ile artık blok arasındaki ana farklılıklar nelerdir? Başlangıç bloğundaki bazı yolları kaldırırsak, birbirleriyle nasıl ilişkili olurlar?
2. Farklı sürümleri uygulamak için ResNet makalesindeki (He *et al.*, 2016) Tablo 1'e bakın.
3. Daha derin ağlar için ResNet, model karmaşıklığını azaltmada bir "darboğaz" mimarisi sunar. Uygulamaya çalışın.
4. ResNet'in sonraki sürümlerinde yazarlar "evrişim, toplu normalleştirme ve etkinleştirme" yapısını "toplu normalleştirme, etkinleştirme ve evrişim" yapısına değiştirdiler. Bu gelişmeyi kendiniz uygulayın. Ayrıntılar için (He *et al.*, 2016)'teki Şekil 1'e bakın.
5. İşlev sınıfları iç içe olsa bile neden işlevlerin karmaşıklığını sınırsız artıramıyoruz?

Tartışmalar[98]

7.7 Yoğun Bağlı Ağlar (DenseNet)

ResNet, derin ağlardaki işlevlerin nasıl parametrize edileceği görüşünü önemli ölçüde değiştirdi. *DenseNet* (yoğun evrişimli ağ) bir dereceye kadar bunun (Huang *et al.*, 2017) mantıksal uzantısıdır. Ona nasıl ulaşacağımızı anlamak için, matematikte küçük bir gezinti yapalım.

7.7.1 ResNet'ten DenseNet'e

Fonksiyonlar için Taylor açılımını hatırlayın. $x = 0$ noktası için şu şekilde yazılabilir:

$$f(x) = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \frac{f'''(0)}{3!}x^3 + \ldots. \tag{7.7.1}$$

Önemli nokta, bir işlevi giderek daha yüksek dereceli terimlerine ayırmasıdır. Benzer bir davranış ile, ResNet fonksiyonları aşağıdaki gibi parçalar

$$f(\mathbf{x}) = \mathbf{x} + g(\mathbf{x}). \tag{7.7.2}$$

Yani, ResNet f'i basit bir doğrusal terime ve daha karmaşık doğrusal olmayan bir terime ayırır. İki terimin ötesinde bilgi özümsemek (zorunlu olarak eklemek değil) istersek ne olur? Bir çözüm DenseNet (Huang *et al.*, 2017) oldu.

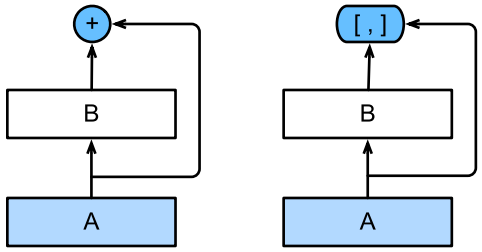


Fig. 7.7.1: Katmanlar arası bağlantılarda ResNet (sol) ve DenseNet (sağ) arasındaki temel fark: Toplama ve bitiştirme kullanımı.

[98] https://discuss.d2l.ai/t/86

Fig. 7.7.1 içinde gösterildiği gibi, ResNet ve DenseNet arasındaki temel fark, ikinci durumda çıktıların toplanmaktan ziyade *bitiştirilmesidir* ($[,]$ ile gösterilir). Sonuç olarak, giderek daha karmaşık bir işlev dizisini uyguladıktan sonra $\mathbf{x}$'ten değerlerine bir eşleme gerçekleştiriyoruz:

$$\mathbf{x} \to [\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})]), f_3([\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})])]), \ldots] . \tag{7.7.3}$$

Sonunda, tüm bu işlevler tekrar öznitelik sayısını azaltmak için MLP'de birleştirilir. Uygulama açısından bu oldukça basittir: Terimleri toplamak yerine, bunları bitiştiririz. DenseNet'in adı, değişkenler arasındaki bağımlılık grafiğinin oldukça yoğunlaştığı gerçeğinden kaynaklanmaktadır. Böyle bir zincirin son katmanı, önceki tüm katmanlara yoğun bir şekilde bağlanır. Yoğun bağlantılar Fig. 7.7.2 içinde gösterilmiştir.

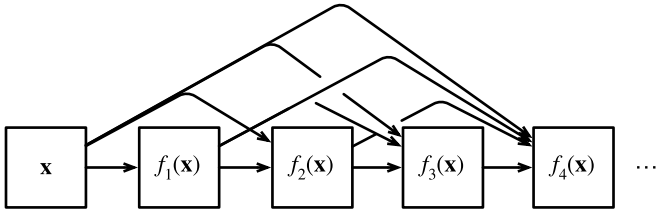


Fig. 7.7.2: DenseNet'teki yoğun bağlantılar.

DenseNet oluşturan ana bileşenler *yoğun bloklar* ve *geçiş katmanları*dır. Birincisi, girdilerin ve çıktıların nasıl bitiştirildiğini tanımlarken, ikincisi kanal sayısını kontrol eder, böylece çok büyümezlerdir.

7.7.2 Yoğun Bloklar

DenseNet, ResNet'in değiştirilmiş "toplu normalleştirme, etkinleştirme ve evrişim" yapısını kullanır (bkz. Section 7.6 içindeki alıştırma). İlk olarak, bu evrişim blok yapısını uyguluyoruz.

```
import torch
from torch import nn
from d2l import torch as d2l


def conv_block(input_channels, num_channels):
    return nn.Sequential(
        nn.BatchNorm2d(input_channels), nn.ReLU(),
        nn.Conv2d(input_channels, num_channels, kernel_size=3, padding=1))
```

Yoğun blok, her biri aynı sayıda çıktı kanalı kullanan birden fazla evrişim bloğundan oluşur. Bununla birlikte, ileri yaymada, kanal boyutundaki her evrişim bloğunun girdi ve çıktısını bitiştiririz.

```
class DenseBlock(nn.Module):
    def __init__(self, num_convs, input_channels, num_channels):
        super(DenseBlock, self).__init__()
        layer = []
        for i in range(num_convs):
            layer.append(conv_block(
```

(continues on next page)

(continued from previous page)

```
                num_channels * i + input_channels, num_channels))
        self.net = nn.Sequential(*layer)

    def forward(self, X):
        for blk in self.net:
            Y = blk(X)
            # Her bloğun girdi ve çıktısını kanal boyutunda birleştirin
            X = torch.cat((X, Y), dim=1)
        return X
```

Aşağıdaki örnekte, 10 çıktı kanalından oluşan 2 evrişim bloğu içeren bir `DenseBlock` örneği tanımlıyoruz. 3 kanallı bir girdi kullanırken, $3 + 2 \times 10 = 23$ kanallı bir çıktı alacağız. Evrişim blok kanallarının sayısı, girdi kanallarının sayısına göre çıktı kanallarının sayısındaki büyümeyi kontrol eder. Bu aynı zamanda *büyüme oranı* olarak da adlandırılır.

```
blk = DenseBlock(2, 3, 10)
X = torch.randn(4, 3, 8, 8)
Y = blk(X)
Y.shape
```

```
torch.Size([4, 23, 8, 8])
```

7.7.3 Geçiş Katmanları

Her yoğun blok kanal sayısını artıracağından, çok fazla sayıda eklemek aşırı karmaşık bir modele yol açacaktır. Modelin karmaşıklığını kontrol etmek için bir *geçiş katmanı* kullanılır. 1×1 evrişimli katmanı kullanarak kanal sayısını azaltılır ve ortalama ortaklama tabakasının yüksekliğini ve genişliğini 2'lik bir uzun adımla yarıya indirilir ve modelin karmaşıklığını daha da azaltılır.

```
def transition_block(input_channels, num_channels):
    return nn.Sequential(
        nn.BatchNorm2d(input_channels), nn.ReLU(),
        nn.Conv2d(input_channels, num_channels, kernel_size=1),
        nn.AvgPool2d(kernel_size=2, stride=2))
```

Önceki örnekte yoğun bloğun çıktısına 10 kanallı bir geçiş katmanı uygulayın. Bu, çıktı kanallarının sayısını 10'a düşürür ve yüksekliği ve genişliği yarıya indirir.

```
blk = transition_block(23, 10)
blk(Y).shape
```

```
torch.Size([4, 10, 4, 4])
```

7.7.4 DenseNet Modeli

Şimdi, bir DenseNet modeli inşa edeceğiz. DenseNet, ilk olarak ResNet'te olduğu gibi aynı tek evrişimli katmanı ve maksimum ortaklama katmanını kullanır.

```
b1 = nn.Sequential(
    nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),
    nn.BatchNorm2d(64), nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

Daha sonra, ResNet'in kullandığı artık bloklardan oluşan dört modüle benzer, DenseNet de dört yoğun blok kullanır. ResNet'e benzer şekilde, her yoğun blokta kullanılan evrişimli katmanların sayısını da ayarlayabiliriz. Burada, Section 7.6 içindeki ResNet-18 modeliyle uyumlu olarak katmanların sayısını 4'e ayarladık. Ayrıca, yoğun bloktaki evrişimli katmanlar için kanal sayısını (yani büyüme hızı) 32'ye ayarladık, böylece her yoğun bloğa 128 kanal eklenecektir.

ResNet'te, her modül arasındaki yükseklik ve genişlik, 2'lik uzun adımlı bir artık bloğu ile azaltılır. Burada, yükseklik ve genişliği yarıya indirip kanal sayısını yarılayarak geçiş katmanını kullanıyoruz.

```
# `num_channels`: mevcut kanal sayısı
num_channels, growth_rate = 64, 32
num_convs_in_dense_blocks = [4, 4, 4, 4]
blks = []
for i, num_convs in enumerate(num_convs_in_dense_blocks):
    blks.append(DenseBlock(num_convs, num_channels, growth_rate))
    # Bu, önceki yoğun bloktaki çıktı kanallarının sayısıdır.
    num_channels += num_convs * growth_rate
    # Yoğun bloklar arasına kanal sayısını yarıya indiren bir geçiş katmanı
    # eklenir
    if i != len(num_convs_in_dense_blocks) - 1:
        blks.append(transition_block(num_channels, num_channels // 2))
        num_channels = num_channels // 2
```

ResNet'e benzer şekilde, çıktıyı üretmek için global bir ortaklama katmanı ile tam bağlı bir katman bağlanır.

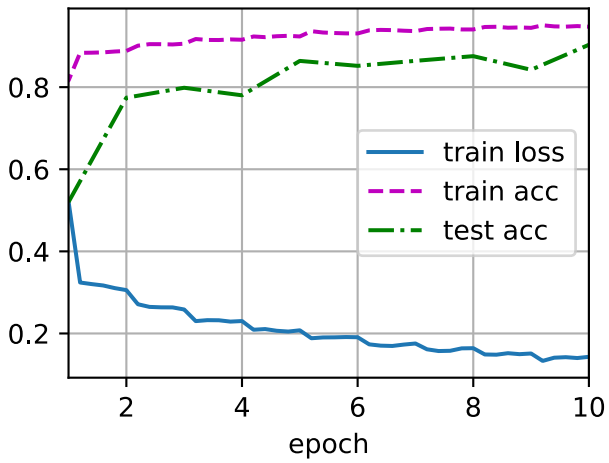
```
net = nn.Sequential(
    b1, *blks,
    nn.BatchNorm2d(num_channels), nn.ReLU(),
    nn.AdaptiveAvgPool2d((1, 1)),
    nn.Flatten(),
    nn.Linear(num_channels, 10))
```

7.7.5 Eğitim

Burada daha derin bir ağ kullandığımızdan, bu bölümde, hesaplamaları basitleştirmek için girdi yüksekliğini ve genişliğini 224'ten 96'ya düşüreceğiz.

```
lr, num_epochs, batch_size = 0.1, 10, 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.143, train acc 0.947, test acc 0.903
5535.7 examples/sec on cuda:0
```



7.7.6 Özet

- DenseNet, girdilerin ve çıktıların birlikte toplandığı ResNet'in aksine, çapraz katman bağlantıları bağlanımda, kanal boyutundaki girdi ve çıktıları bitiştirir.
- DenseNet'i oluşturan ana bileşenler yoğun bloklar ve geçiş katmanlarıdır.
- Kanal sayısını tekrar küçülten geçiş katmanları ekleyerek ağı oluştururken boyutsallığı kontrol altında tutmamız gerekir.

7.7.7 Alıştırmalar

1. Neden geçiş katmanında maksimum ortaklama yerine ortalama ortaklama kullanıyoruz?
2. DenseNet makalesinde belirtilen avantajlardan biri, model parametrelerinin ResNet'ten daha küçük olmasıdır. Bu neden burada geçerlidir?
3. DenseNet'in eleştirildiği bir sorun, yüksek bellek tüketimidir.
    1. Bu gerçekten doğru mudur? Gerçek GPU bellek tüketimini görmek için girdi şeklini 224×224 olarak değiştirmeye çalışın.
    2. Bellek tüketimini azaltmanın alternatif bir yolunu düşünebiliyor musunuz? Çerçeveyi nasıl değiştirmeniz gerekecektir?

4. DenseNet makalesi (Huang *et al.*, 2017) Tablo 1'de sunulan çeşitli DenseNet sürümlerini uygulayın.
5. DenseNet fikrini uygulayarak MLP tabanlı bir model tasarlayın. Section 4.10 içindeki konut fiyatı tahmini çalışmasına uygulayın.

Tartışmalar[99]

[99] https://discuss.d2l.ai/t/88

8 | Yinelemeli Sinir Ağları

Şimdiye kadar iki tür veriyle karşılaştık: Tablo verileri ve imge verileri. İkincisi için, içlerindeki düzenlilikten yararlanmak için özel katmanlar tasarladık. Başka bir deyişle, bir imgedeki piksellerin yerini değiştirirsek, analog TV zamanlarındaki gibi bir test deseninin arka planına çok benzeyen bir şeyin içeriğinden bahsetmek çok daha zor olurdu.

En önemlisi, şimdiye kadar, verilerimizin hepsinin bir dağılımdan çekildiğini ve tüm örneklerin bağımsız ve özdeşçe dağıtıldığını (i.i.d.) varsaydık. Ne yazık ki, bu çoğu veri için doğru değildir. Örneğin, bu paragraftaki sözcükler sırayla yazılmıştır ve rastgele yer değiştirilirlerse anlamları açığa çıkarmak oldukça zor olur. Aynı şekilde, bir videoda imge çerçeveleri, bir konuşmadaki ses sinyali ve bir web sitesinde gezinme davranışı dizili sırayı takip eder. Bu nedenle, bu tür veriler için özelleşmiş modellerin bunları tanımlamada daha iyi olacağını varsaymak mantıklıdır.

Başka bir sorun, yalnızca bir girdi olarak bir diziyi almakla kalmayıp, diziye devam etmemiz beklenebileceği gerçeğinden kaynaklanmaktadır. Örneğin, görevimiz $2, 4, 6, 8, 10, \ldots$ serisine devam etmek olabilir. Bu, zaman serisi analizinde, borsayı, hastanın ateş değeri eğrisini veya yarış arabası için gereken ivmeyi tahmin etme de oldukça yaygındır. Yine bu tür verileri işleyebilecek modellere sahip olmak istiyoruz.

Kısacası, CNN'ler uzamsal bilgileri verimli bir şekilde işleyebilirken, *yinelemeli sinir ağları* (RNN) dizili bilgileri daha iyi işleyecek şekilde tasarlanmıştır. RNN'ler, mevcut çıktıları belirlemek için geçmiş bilgileri, mevcut girdiler ile birlikte depolayan durum değişkenlerini kullanır.

Yinelemeli ağları kullanan örneklerin çoğu metin verilerine dayanmaktadır. Bu nedenle, bu bölümde dil modellerini vurgulayacağız. Dizi verilerinin daha resmi bir incelemesinden sonra, metin verilerinin ön işlenmesi için pratik teknikler sunacağız. Daha sonra, bir dil modelinin temel kavramlarını tartışıyoruz ve bu tartışmayı RNN'lerin tasarımına ilham kaynağı olarak kullanıyoruz. Sonunda, bu tür ağları eğitirken karşılaşılabilecek sorunları keşfetmek için RNN'ler için gradyan hesaplama yöntemini açıklıyoruz.

8.1 Dizi Modelleri

Netflix'te film izlediğinizi düşünün. İyi bir Netflix kullanıcısı olarak, filmlerin her birini dürüstçe değerlendirmeye karar veriyorsunuz. Sonuçta, iyi bir film iyi bir filmdir, ve siz onlardan daha fazla izlemek istiyorsunuz, değil mi? Görünüşe göre, bu işler o kadar basit değil. İnsanların filmler hakkındaki görüşleri zamanla oldukça önemli ölçüde değişebilir. Aslında, psikologların bazı etkiler için koydukları isimler bile vardır:

- Başkasının fikrine dayalı olarak "çapalama" vardır. Örneğin, Oscar ödüllerinden sonra, ilgili filmin reytingleri hala film aynı olmasına rağmen yükselir. Bu etki, ödül unutulana kadar birkaç ay boyunca devam eder. Bu etkinin yarım puandan fazla reyting yükselttiği gösterilmiştir (Wu *et al.*, 2017).

- İnsanların yeni normal olarak geliştirilmiş veya kötüleşmiş bir durumu kabul etmek için hızla adapte olduğu *zevksel uyarlama* vardır. Örneğin, birçok iyi film izledikten sonra, bir sonraki filmin eşit derecede iyi veya daha iyi olması beklentisi yüksektir. Bu nedenle, harika filmler izlendikten sonra ortalama bir film bile kötü sayılabilir.
- Mevsimsellik vardır. Çok az izleyici Ağustos ayında Noel Baba filmi izlemek ister.
- Bazı durumlarda, filmler yapımdaki yönetmenlerin veya aktörlerin yanlış davranışlarından dolayı sevilmeyen hale gelir.
- Bazı filmler kült filmlere dönüşmüştür, çünkü neredeyse komik bir şekilde kötüydüler. *Outer Space* ve *Troll 2* bu nedenle yüksek derecede şöhret elde etti.

Kısacası, film reytinglerinde sabitlik dışında her şey vardır. Böylece, zamansal dinamikleri kullanmak daha doğru film önerilerine yol açtı (Koren, 2009). Tabii ki, dizi verileri sadece film derecelendirmeleri ile ilgili değildir. Aşağıda daha fazla örnek görebilirsiniz.

- Birçok kullanıcı, uygulamaları açtıkları zamana göre son derece özel davranışlara sahiptir. Örneğin, sosyal medya uygulamaları kullanımı öğrencilerde okul zamanından sonra çok daha yaygındır. Borsa alım satım uygulamaları piyasalar açık olduğunda daha yaygın olarak kullanılır.
- Yarının hisse senedi fiyatlarını tahmin etmek, dün kaçırdığımız bir hisse senedi fiyatının boşluklarını doldurmaktan çok daha zordur, her ne kadar ikisi de bir sayıyı tahmin etme meselesi olsa da. Sonuçta, öngörü, geçmişe görüden çok daha zordur. İstatistiklerde, ilki (bilinen gözlemlerin ötesinde tahmin) *dışdeğerleme (extrapolation)* olarak adlandırılırken, ikincisi (mevcut gözlemler arasındaki tahmin) *aradeğerleme (interpolation)* olarak adlandırılır.
- Müzik, konuşma, metin ve videolar doğaları gereği dizilidir. Eğer onlarda yer değişimine verseydik, çok az anlam ifade ederlerdi. *Köpek adamı ısırdı* manşeti, kelimelerin aynı olmasına rağmen, *adam köpeği ısırdı* manşetinden çok daha az şaşırtıcıdır.
- Depremler son derece ilişkilidir, yani büyük bir deprem sonrası güçlü deprem olmayan zamana kıyasla çok daha fazla, çok daha küçük artçı şoklar vardır. Aslında depremler uzamsal olarak ilişkilidir, yani artçı şoklar genellikle kısa bir süre içinde ve yakın bir yerde meydana gelir.
- Twitter kavgalarında, dans düzenlerinde ve tartışmalarda görülebileceği gibi, insanlar birbirleri ile ardışık bir şekilde etkileşime girerler.

8.1.1 İstatistiksel Araçlar

Dizi verileriyle uğraşmak için istatistiksel araçlara ve yeni derin sinir ağı mimarilerine ihtiyacımız var. İşleri basit tutmak için örnek olarak Fig. 8.1.1 içinde gösterilen hisse senedi fiyatını (FTSE 100 endeksi) kullanalım.

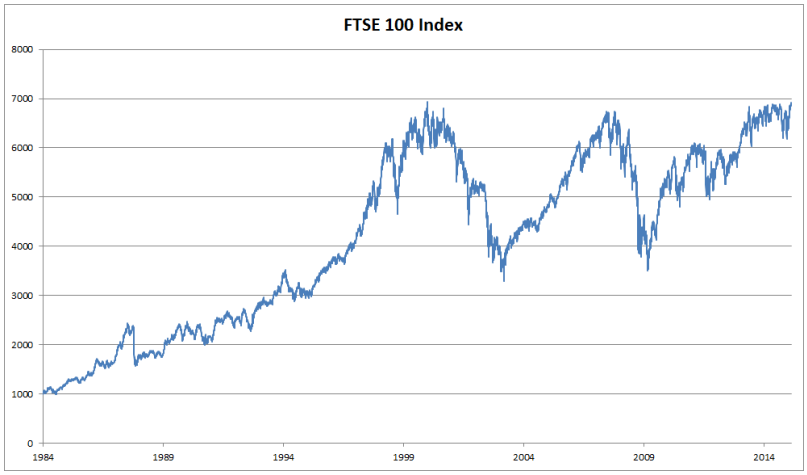


Fig. 8.1.1: Yaklaşık 30 yıldan fazla sürenin FTSE 100 endeksi.

Fiyatları x_t ile gösterelim, yani *zaman adım* $t \in \mathbb{Z}^+$'de x_t fiyatını gözlemliyoruz. Bu kitaptaki diziler için t'nin genellikle ayrık olacağını ve tamsayılara veya alt kümesine göre değişeceğini unutmayın. t. günde borsada iyi kazanmak isteyen bir borsa simsarının x_t üzerinden tahmin ettiğini varsayalım:

$$x_t \sim P(x_t \mid x_{t-1}, \ldots, x_1). \tag{8.1.1}$$

Özbağlanımlı Modeller

Bunu başarmak için, simsarımız Section 3.3 içinde eğittiğimiz gibi bir regresyon modelini kullanabilir. Sadece bir büyük sorun var: Girdilerimizin adedi, $x_{t-1}, \ldots, x_1$, t'ye bağlı olarak değişir. Yani, karşılaştığımız veri miktarı ile sayı artar ve bunu hesaplamalı olarak işlenebilir hale getirmek için bir yaklaşıma ihtiyacımız vardır. Bu bölümde konuların çoğu $P(x_t \mid x_{t-1}, \ldots, x_1)$'nin verimli bir şekilde nasıl tahmin edileceği etrafında dönecektir. Kısacası, aşağıdaki gibi iki stratejiye indirgeniyor.

İlk olarak, potansiyel olarak oldukça uzun dizinin $x_{t-1}, \ldots, x_1$ gerçekten gerekli olmadığını varsayalım. Bu durumda kendimizi τ uzunluğunda bir süre ile memnun edebilir ve sadece $x_{t-1}, \ldots, x_{t-\tau}$ gözlemlerini kullanabiliriz. İlk faydası, artık argüman sayısının en azından $t > \tau$ için her zaman aynı olmasıdır. Bu, yukarıda belirtildiği gibi derin bir ağı eğitmemizi sağlar. Bu tür modeller, kelimenin tam anlamıyla kendileri üzerinde bağlanım gerçekleştirdikleri için *özbağlanımlı modeller* olarak adlandırılacaktır.

Fig. 8.1.2 içinde gösterilen ikinci strateji, geçmiş gözlemlerin h_t'sının bir özetini tutmak ve aynı zamanda $\hat{x}_t$'in tahmine ek olarak h_t'yı güncellemektir. Bu, bizi $\hat{x}_t = P(x_t \mid h_t)$ ile x_t'i tahmin eden ve dahası $h_t = g(h_{t-1}, x_{t-1})$ formunu güncelleyen modellere yönlendirir. h_t asla gözlenmediğinden, bu modellere *saklı özbağlanımlı modeller* de denir.

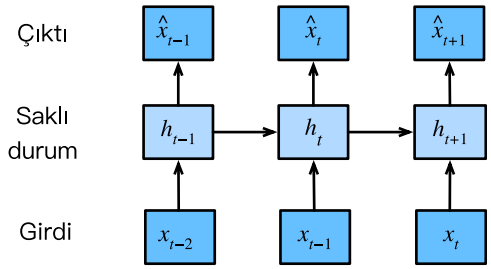


Fig. 8.1.2: Saklı özbağlanımlı model.

Her iki durumda da eğitim verilerinin nasıl oluşturulacağına dair açık bir soru ortaya çıkıyor. Kullanıcı tipik olarak tarihsel gözlemleri kullanarak, şu ana kadar verilen gözlemlere dayanarak bir sonraki gözlemi tahmin eder. Açıkça zamanın hareketsiz kalmasını beklemeyiz. Bununla birlikte, genel varsayım, x_t'in özgül değerlerinin değişebileceği halde, en azından dizinin dinamiklerinin değişmeyeceği yönündedir. Bu mantıklı, çünkü yeni dinamikler adı gibi yeni ve bu nedenle şimdiye kadar sahip olduğumuz verileri kullanarak tahmin edilemezler. İstatistikçiler değişmeyen dinamiklere durağan derler. Ne olursa olsun, böylece aşağıdaki ifade aracılığıyla tüm diziyi tahminleyebiliriz:

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{t-1}, \ldots, x_1). \tag{8.1.2}$$

Sürekli sayılar yerine kelimeler gibi ayrık nesnelerle uğraştığımızda da yukarıdaki hususların hala geçerli olduğunu unutmayın. Tek fark, böyle bir durumda $P(x_t \mid x_{t-1}, \ldots, x_1)$'yi tahmin etmek için bir regresyon modeli yerine bir sınıflandırıcı kullanmamız gerektiğidir.

Markov Modelleri

Özbağlanımlı bir modelde x_t'yi tahmin etmek için $x_{t-1}, \ldots, x_1$ yerine sadece $x_{t-1}, \ldots, x_{t-\tau}$ kullandığımız yaklaşımı hatırlayın. Bu yaklaşım doğru olduğunda, dizinin bir *Markov koşulu* karşıladığını söyleriz. Özellikle, eğer $\tau = 1$ ise, bir *birinci dereceden Markov modeli*miz vardır ve $P(x)$ şöyle ifade edilir:

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{t-1}) \text{ where } P(x_1 \mid x_0) = P(x_1). \tag{8.1.3}$$

Bu tür modeller özellikle x_t yalnızca ayrı bir değer varsayıldığında iyi çalışır, çünkü bu durumda dinamik programlama zincir boyunca değerleri tam olarak hesaplamak için kullanılabilir. Örneğin, $P(x_{t+1} \mid x_{t-1})$'in verimli bir şekilde hesaplanmasını sağlayabiliriz:

$$\begin{aligned} P(x_{t+1} \mid x_{t-1}) &= \frac{\sum_{x_t} P(x_{t+1}, x_t, x_{t-1})}{P(x_{t-1})} \\ &= \frac{\sum_{x_t} P(x_{t+1} \mid x_t, x_{t-1}) P(x_t, x_{t-1})}{P(x_{t-1})} \\ &= \sum_{x_t} P(x_{t+1} \mid x_t) P(x_t \mid x_{t-1}) \end{aligned} \tag{8.1.4}$$

Sadece geçmiş gözlemlerin çok kısa bir tarihini hesaba katmamız gerektiği gerçeğini kullanabiliriz: $P(x_{t+1} \mid x_t, x_{t-1}) = P(x_{t+1} \mid x_t)$. Dinamik programlama detaylarına girmek bu bölümün kapsamı dışındadır. Kontrol ve pekiştirmeli öğrenme algoritmaları bu tür araçları kapsamlı olarak kullanır.

Nedensellik

Prensipte, $P(x_1, \ldots, x_T)$'nin ters sırada açılmasında yanlış bir şey yoktur. Sonuçta, koşullandırma ile her zaman aşağıdaki gibi yazabiliriz:

$$P(x_1, \ldots, x_T) = \prod_{t=T}^{1} P(x_t \mid x_{t+1}, \ldots, x_T). \tag{8.1.5}$$

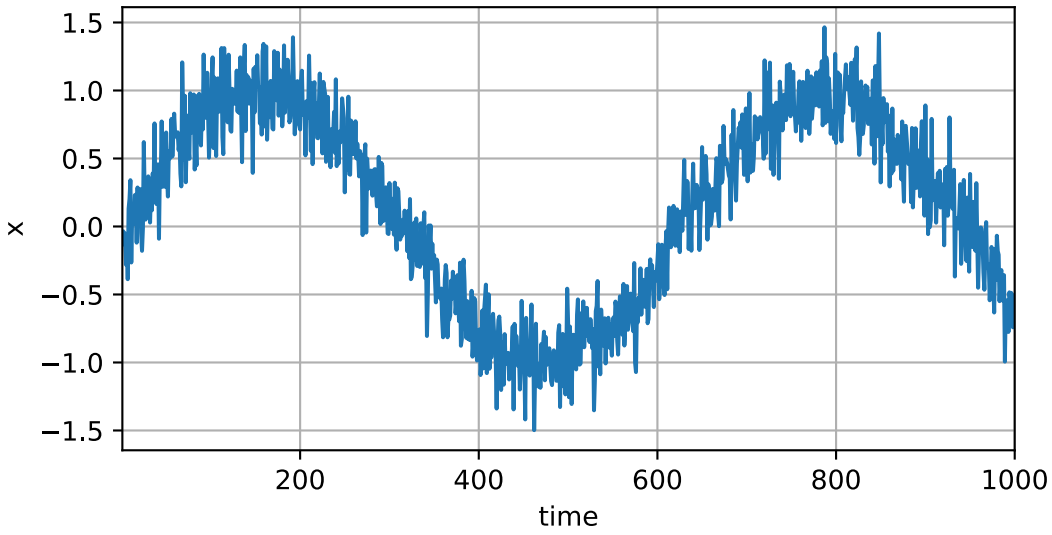
Aslında, eğer bir Markov modelimiz varsa, bir ters koşullu olasılık dağılımı da elde edebiliriz. Bununla birlikte, birçok durumda, veriler için doğal bir yön vardır, yani zaman içinde ileriye giderler. Gelecekteki olayların geçmişi etkilemeyeceği açıktır. Bu nedenle, eğer x_t'i değiştirirsek, x_{t+1}'nin ilerlemesine ne olacağını etkileyebiliriz ama tersi değil. Yani, x_t'i değiştirirsek, geçmiş olayların üzerindeki dağılım değişmez. Sonuç olarak, $P(x_t \mid x_{t+1})$'den ziyade $P(x_{t+1} \mid x_t)$'yi açıklamak daha kolay olmalıdır. Örneğin, bazı durumlarda ϵ gürültüsü için $x_{t+1} = f(x_t) + \epsilon$'yi bulabildiğimiz gösterilmiştir, oysa tersi doğru değildir (Hoyer *et al.*, 2009). Bu harika bir haber, çünkü tipik olarak tahmin etmekle ilgilendiğimiz ileri yöndür. Peters ve diğerlerinin kitabı bu konuda daha fazla açıklama yapmaktadır (Peters *et al.*, 2017). Bizse sadece yüzeyine bakıyoruz.

8.1.2 Eğitim

Bu kadar çok istatistiksel aracı inceledikten sonra, bunu pratikte de deneyelim. Biraz veri üreterek başlıyoruz. İşleri basit tutmak için, $1, 2, \ldots, 1000$ adımları için biraz gürültülü bir sinüs fonksiyonu kullanarak dizi verimizi üretiyoruz.

```
%matplotlib inline
import torch
from torch import nn
from d2l import torch as d2l
```

```
T = 1000  # Toplamda 1000 nokta oluşturun
time = torch.arange(1, T + 1, dtype=torch.float32)
x = torch.sin(0.01 * time) + torch.normal(0, 0.2, (T,))
d2l.plot(time, [x], 'time', 'x', xlim=[1, 1000], figsize=(6, 3))
```



Ardından, böyle bir diziyi modelimizin üzerinde eğitebileceği özniteliklere ve etiketlere dönüştürmemiz gerekiyor. τ gömme boyutuna dayanarak, verileri $y_t = x_t$ ve $\mathbf{x}_t = [x_{t-\tau}, \ldots, x_{t-1}]$ çiftleri halinde eşleriz. Dikkatli okuyucu, bunun bize τ adet daha az veri örneği verdiğini fark etmiş olabilir, çünkü ilk τ için yeterli geçmişe sahip değiliz. Basit bir düzeltme, özellikle dizi uzunsa, bu ilk birkaç terimi atmaktır. Alternatif olarak diziye sıfırlarla dolgu yapabiliriz. Burada eğitim için sadece ilk 600 öznitelik-etiket çiftini kullanıyoruz.

```
tau = 4
features = torch.zeros((T - tau, tau))
for i in range(tau):
    features[:, i] = x[i: T - tau + i]
labels = x[tau:].reshape((-1, 1))
```

```
batch_size, n_train = 16, 600
# Eğitim için yalnızca ilk `n_train` tane örnek kullanılır
train_iter = d2l.load_array((features[:n_train], labels[:n_train]),
                            batch_size, is_train=True)
```

Burada mimariyi oldukça basit tutuyoruz: Sadece iki tam bağlı katmanlı bir MLP, ReLU etkinleştirmesi ve kare kaybı.

```
# Ağın ağırlıklarını ilkleme işlevi
def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.xavier_uniform_(m.weight)

# Basit  bir MLP
def get_net():
    net = nn.Sequential(nn.Linear(4, 10),
                        nn.ReLU(),
                        nn.Linear(10, 1))
    net.apply(init_weights)
    return net

# Not: `MSELoss` karesel hatayı 1/2 çarpanı olmadan hesaplar
loss = nn.MSELoss(reduction='none')
```

Şimdi modeli eğitmeye hazırız. Aşağıdaki kod esas olarak Section 3.3 gibi, önceki bölümlerdeki eğitim döngüsüyle aynıdır. Bu yüzden, çok fazla ayrıntıya girmeyeceğiz.

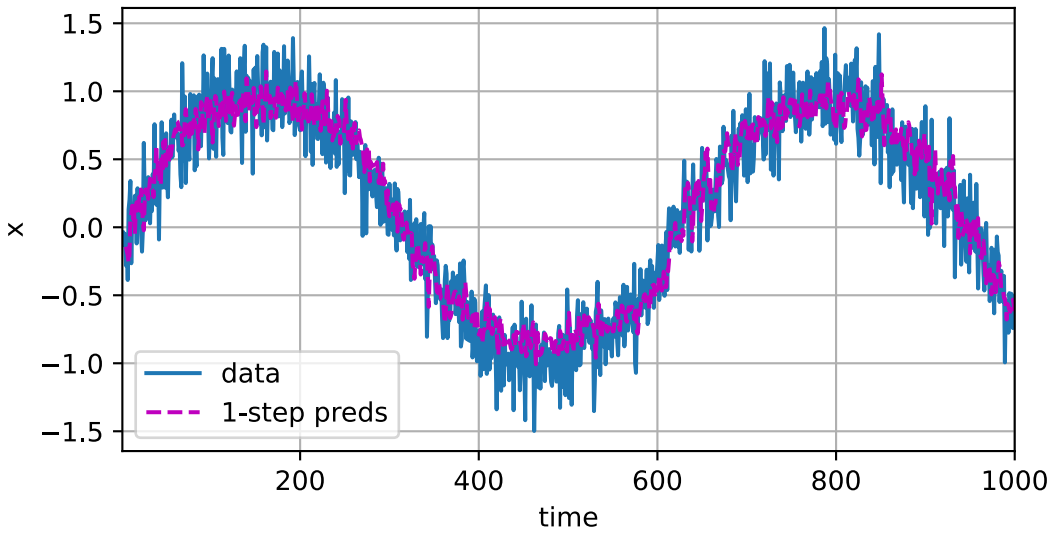
```
def train(net, train_iter, loss, epochs, lr):
    trainer = torch.optim.Adam(net.parameters(), lr)
    for epoch in range(epochs):
        for X, y in train_iter:
            trainer.zero_grad()
            l = loss(net(X), y)
            l.sum().backward()
            trainer.step()
        print(f'epoch {epoch + 1}, '
              f'loss: {d2l.evaluate_loss(net, train_iter, loss):f}')

net = get_net()
train(net, train_iter, loss, 5, 0.01)
```

```
epoch 1, loss: 0.066216
epoch 2, loss: 0.058680
epoch 3, loss: 0.057771
epoch 4, loss: 0.055117
epoch 5, loss: 0.054249
```

8.1.3 Tahmin

Eğitim kaybı küçük olduğu için modelimizin iyi çalışmasını bekleriz. Bunun pratikte ne anlama geldiğini görelim. Kontrol edilmesi gereken ilk şey, modelin sadece bir sonraki adımında ne kadar iyi tahmin edebildiğidir, yani *bir adım sonrasını tahmin*.

```
onestep_preds = net(features)
d2l.plot([time, time[tau:]], [x.detach().numpy(), onestep_preds.detach().numpy()], 'time',
         'x', legend=['data', '1-step preds'], xlim=[1, 1000], figsize=(6, 3))
```



Bir adım sonrası tahminler, beklediğimiz gibi güzel görünüyor. 604 (`n_train + tau`) gözlemin ötesinde bile tahminler hala güvenilir görünüyor. Bununla birlikte, burada küçük bir sorun var: Sıra verilerini yalnızca 604 zaman adımına kadar gözlemlersek, gelecekteki tüm bir adım önde gelen tahminler için girdileri almayı umut edemeyiz. Bunun yerine, her seferinde bir adım ileriye doğru ilerlemeliyiz:

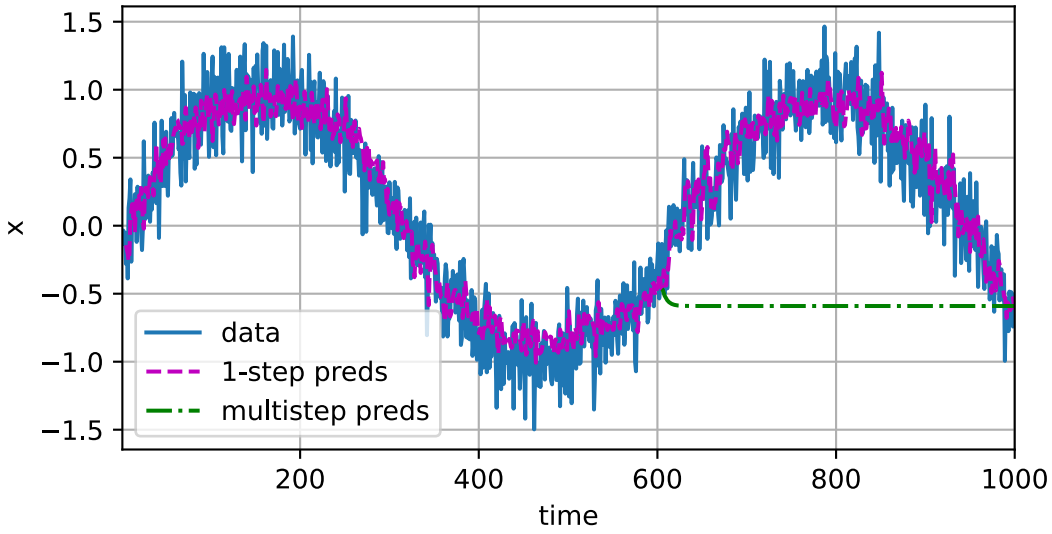
$$\begin{aligned}
\hat{x}_{605} &= f(x_{601}, x_{602}, x_{603}, x_{604}),\\
\hat{x}_{606} &= f(x_{602}, x_{603}, x_{604}, \hat{x}_{605}),\\
\hat{x}_{607} &= f(x_{603}, x_{604}, \hat{x}_{605}, \hat{x}_{606}),\\
\hat{x}_{608} &= f(x_{604}, \hat{x}_{605}, \hat{x}_{606}, \hat{x}_{607}),\\
\hat{x}_{609} &= f(\hat{x}_{605}, \hat{x}_{606}, \hat{x}_{607}, \hat{x}_{608}),\\
&\ldots
\end{aligned} \tag{8.1.6}$$

Genel olarak, x_t'ye kadar gözlenen bir dizi için $t+k$ zaman adımında tahmin edilen çıktı $\hat{x}_{t+k}$, *k-adım tahmin* olarak adlandırılır. x_{604}'ya kadar gözlemlediğimizden, k-adım tahminimiz $\hat{x}_{604+k}$'dir.

Başka bir deyişle, çok ileriki tahminler için kendi tahminlerimizi kullanmak zorunda kalacağız. Bakalım ne kadar iyi gidecek.

```
multistep_preds = torch.zeros(T)
multistep_preds[: n_train + tau] = x[: n_train + tau]
for i in range(n_train + tau, T):
    multistep_preds[i] = net(
        multistep_preds[i - tau:i].reshape((1, -1)))
```

```
d2l.plot([time, time[tau:], time[n_train + tau:]],
         [x.detach().numpy(), onestep_preds.detach().numpy(),
          multistep_preds[n_train + tau:].detach().numpy()], 'time',
         'x', legend=['data', '1-step preds', 'multistep preds'],
         xlim=[1, 1000], figsize=(6, 3))
```



Yukarıdaki örnekte görüldüğü gibi, tam bir felaket. Tahminler birkaç tahminden sonra oldukça hızlı bir şekilde bir sabite sönümlenir. Peki algoritma neden bu kadar kötü çalıştı? Bu sonuç hataların birikmesinden kaynaklanmaktadır. 1. adımdan sonra hatamızın $\epsilon_1 = \bar{\epsilon}$ olduğunu varsayalım. Şimdi 2. adımda *girdi* ϵ_1 tarafından dürtülüyor ve bazı c sabiti için $\epsilon_2 = \bar{\epsilon} + c\epsilon_1$ formunda hatalar görüyoruz. Hata, gerçek gözlemlerden oldukça hızlı bir şekilde uzaklaşabilir. Bu yaygın bir olgudur. Örneğin, önümüzdeki 24 saat için hava tahminleri oldukça doğru olma eğilimindedir ama bunun ötesinde doğruluk hızla azalır. Bu bölümde ve sonrasında bunun iyileştirilmesi için olası yöntemleri tartışacağız.

$k = 1, 4, 16, 64$ için tüm dizideki tahminleri hesaplayarak k adım ilerideki tahminlerdeki zorluklara daha yakından bir göz atalım.

```
max_steps = 64
```

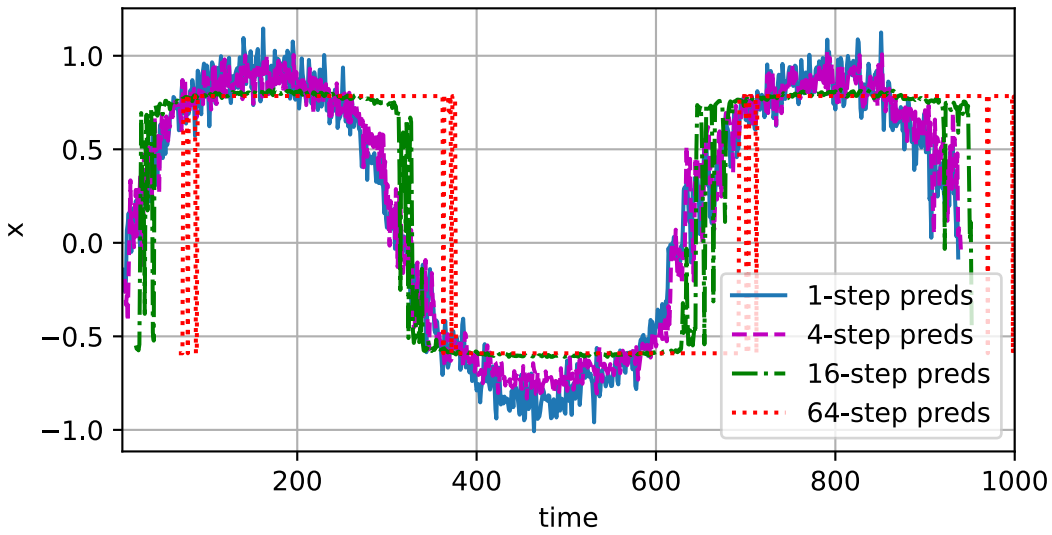
```
features = torch.zeros((T - tau - max_steps + 1, tau + max_steps))
# Sütun `i` (`i` < `tau`), `i + 1` ile `i + T - tau - max_steps + 1`
# arasındaki zaman adımları için `x`'ten gözlemlerdir.
for i in range(tau):
```

(continues on next page)

(continued from previous page)

```
    features[:, i] = x[i: i + T - tau - max_steps + 1]

# Sütun `i` (`i` >= `tau`), 'i + 1' ile `i + T - tau - max_steps + 1`
# arasındaki zaman adımları için (`i - tau + 1`)-adım ileriki tahminlerdir.
for i in range(tau, tau + max_steps):
    features[:, i] = net(features[:, i - tau:i]).reshape(-1)
```

```
steps = (1, 4, 16, 64)
d2l.plot([time[tau + i - 1: T - max_steps + i] for i in steps],
         [features[:, tau + i - 1].detach().numpy() for i in steps], 'time', 'x',
         legend=[f'{i}-step preds' for i in steps], xlim=[5, 1000],
         figsize=(6, 3))
```



Bu, gelecekte daha da ileriyi doğru tahmin etmeye çalıştıkça, tahminin kalitesinin nasıl değiştiğini açıkça göstermektedir. 4 adım ilerideki tahminler hala iyi görünse de, bunun ötesinde herhangi birşey neredeyse işe yaramaz.

8.1.4 Özet

- Aradeğerleme ve dışdeğerleme arasında zorluk bakımında oldukça fark vardır. Sonuç olarak, bir diziniz varsa, eğitim sırasında verilerin zamansal sırasına daima saygı gösterin, yani gelecekteki veriler ile asla eğitmeyin.
- Dizi modelleri tahmin için özel istatistiksel araçlar gerektirir. İki popüler seçenek özbağlanımlı modeller ve saklı-değişken özbağlanımlı modellerdir.
- Nedensel modellerde (örn. ileriye akan zaman), ileri yönün tahmin edilmesi genellikle ters yönden çok daha kolaydır.
- t'ye kadar gözlenen bir dizi için, $t + k$ zaman adımındaki tahmin edilen çıktı *k-adım tahmin* olur. k'yı artırarak daha da ileriki zaman için tahmin ettiğimizde, hatalar birikir ve tahminin kalitesi genellikle dramatik bir şekilde bozulur.

8.1.5 Alıştırmalar

1. Bu bölümün deneyindeki modeli geliştirin.
    1. Geçmiş 4 gözlemden daha fazlasını mı dahil ediyor musunuz? Gerçekten kaç taneye ihtiyacınız var?
    2. Gürültü olmasaydı kaç tane geçmiş gözleme ihtiyacınız olurdu? İpucu: sin ve cos'u diferansiyel denklem olarak yazabilirsiniz.
    3. Toplam öznitelik sayısını sabit tutarken eski gözlemleri de dahil edebilir misiniz? Bu doğruluğu artırır mı? Neden?
    4. Sinir ağı mimarisini değiştirin ve performansını değerlendirin.
2. Bir yatırımcı satın almak için iyi bir yatırım bulmak istiyor. Hangisinin iyi olacağına karar vermek için geçmiş dönemlere bakıyor. Bu stratejide ne yanlış gidebilir ki?
3. Nedensellik metinler için de geçerli midir? Peki ne dereceye kadar?
4. Verinin dinamiğini yakalamak için gizli bir özbağlanımlı modelin ne zaman gerekli olabileceğine dair bir örnek verin.

Tartışmalar[100]

8.2 Metin Ön İşleme

Dizi verileri için istatistiksel araçları ve tahmin zorluklarını inceledik ve değerlendirdik. Bu veriler birçok şekil alabilir. Özellikle, kitabın birçok bölümünde odaklanacağımız gibi, metinler dizi verilerinin en popüler örneklerindendir. Örneğin, bir makale basitçe bir sözcük dizisi veya hatta bir karakter dizisi olarak görülebilir. Dizi verileriyle gelecekteki deneylerimizi kolaylaştırmak amacıyla, bu bölümü metin için ortak ön işleme adımlarını açıklamaya adayacağız. Genellikle, aşağıdaki adımlar vardır:

1. Metni dizgi (string) olarak belleğe yükleyin.
2. Dizgileri andıçlara (token) ayırın (örn. kelimeler ve karakterler).
3. Bölünmüş andıçları sayısal indekslerle eşlemek için bir kelime tablosu oluşturun.
4. Metni sayısal indekslerin dizilerine dönüştürün, böylece modeller tarafından kolayca işlenebilirler.

```
import collections
import re
from d2l import torch as d2l
```

[100] https://discuss.d2l.ai/t/114

8.2.1 Veri Kümesini Okuma

Başlamak için H. G. Wells'in *Zaman Makinesi*[101]'nden metin yüklüyoruz. Bu 30000 kelimenin biraz üzerinde oldukça küçük bir külliyat, ama göstermek istediğimiz şey için gayet iyi. Daha gerçekçi belge koleksiyonları milyarlarca kelime içerir. Aşağıdaki işlev, veri kümesini her satırın bir dizgi olduğu metin satırları listesine okur. Basitlik için, burada noktalama işaretlerini ve büyük harfleri görmezden geliyoruz.

```
#@save
d2l.DATA_HUB['time_machine'] = (d2l.DATA_URL + 'timemachine.txt',
                                '090b5e7e70c295757f55df93cb0a180b9691891a')

def read_time_machine():  #@save
    """Zaman makinesi veri kümesini bir metin satırı listesine yükleyin."""
    with open(d2l.download('time_machine'), 'r') as f:
        lines = f.readlines()
    return [re.sub('[^A-Za-z]+', ' ', line).strip().lower() for line in lines]

lines = read_time_machine()
print(f'# text lines: {len(lines)}')
print(lines[0])
print(lines[10])
```

```
# text lines: 3221
the time machine by h g wells
twinkled and his usually pale face was flushed and animated the
```

8.2.2 Andıçlama

Aşağıdaki `tokenize` işlevi, girdi olarak bir liste (`lines`) alır ve burada her eleman bir metin dizisidir (örneğin, bir metin satırı). Her metin dizisi bir andıç listesine bölünür. *Andıç* metindeki temel birimdir. Sonunda, her andıcın bir dizgi olduğu andıç listelerinin bir listesi döndürülür.

```
def tokenize(lines, token='word'):  #@save
    """Metin satırlarını kelime veya karakter belirteçlerine ayırın."""
    if token == 'word':
        return [line.split() for line in lines]
    elif token == 'char':
        return [list(line) for line in lines]
    else:
        print('ERROR: unknown token type: ' + token)

tokens = tokenize(lines)
for i in range(11):
    print(tokens[i])
```

```
['the', 'time', 'machine', 'by', 'h', 'g', 'wells']
[]
[]
```

(continues on next page)

[101] http://www.gutenberg.org/ebooks/35

(continued from previous page)

```
[]
[]
['i']
[]
[]
['the', 'time', 'traveller', 'for', 'so', 'it', 'will', 'be', 'convenient', 'to', 'speak',
↪'of', 'him']
['was', 'expounding', 'a', 'recondite', 'matter', 'to', 'us', 'his', 'grey', 'eyes', 'shone',
↪ 'and']
['twinkled', 'and', 'his', 'usually', 'pale', 'face', 'was', 'flushed', 'and', 'animated',
↪'the']
```

8.2.3 Kelime Dağarcığı

Andıcın dizgi tipi, sayısal girdiler alan modeller tarafından kullanılmak için elverişsizdir. Şimdi dizgi andıçlarını 0'dan başlayan sayısal indekslere eşlemek için, genellikle *kelime dağarcığı* olarak adlandırılan bir sözlük oluşturalım. Bunu yapmak için, önce eğitim kümesindeki tüm belgelerdeki benzersiz andıçları, yani bir *külliyat*ı, sayarız ve daha sonra her benzersiz andıca frekansına göre sayısal bir indeks atarız. Genellikle nadiren ortaya çıkan andıçlar karmaşıklığı azaltmak için kaldırılır. Külliyat içinde bulunmayan veya kaldırılan herhangi bir andıç, bilinmeyen özel bir andıç "<unk>" olarak eşleştirilir. İsteğe bağlı olarak, dolgu için "<pad>", bir dizinin başlangıcını sunmak için "<bos>" ve bir dizinin sonu için "<eos>" gibi ayrılmış andıçların bir listesini ekleriz.

```
class Vocab:  #@save
    """Metin için kelime hazinesi."""
    def __init__(self, tokens=None, min_freq=0, reserved_tokens=None):
        if tokens is None:
            tokens = []
        if reserved_tokens is None:
            reserved_tokens = []
        # Frekanslara göre sırala
        counter = count_corpus(tokens)
        self._token_freqs = sorted(counter.items(), key=lambda x: x[1],
                                   reverse=True)
        # Bilinmeyen andıcın indeksi 0'dır
        self.idx_to_token = ['<unk>'] + reserved_tokens
        self.token_to_idx = {token: idx
                             for idx, token in enumerate(self.idx_to_token)}
        for token, freq in self._token_freqs:
            if freq < min_freq:
                break
            if token not in self.token_to_idx:
                self.idx_to_token.append(token)
                self.token_to_idx[token] = len(self.idx_to_token) - 1

    def __len__(self):
        return len(self.idx_to_token)

    def __getitem__(self, tokens):
        if not isinstance(tokens, (list, tuple)):
            return self.token_to_idx.get(tokens, self.unk)
        return [self.__getitem__(token) for token in tokens]
```

(continues on next page)

(continued from previous page)

```
    def to_tokens(self, indices):
        if not isinstance(indices, (list, tuple)):
            return self.idx_to_token[indices]
        return [self.idx_to_token[index] for index in indices]

    @property
    def unk(self):  # Bilinmeyen andıç için dizin
        return 0

    @property
    def token_freqs(self):  # Bilinmeyen andıç için dizin
        return self._token_freqs

def count_corpus(tokens):  #@save
    """Count token frequencies."""
    # Burada `tokens` bir 1B liste veya 2B listedir
    if len(tokens) == 0 or isinstance(tokens[0], list):
        # Bir belirteç listelerinin listesini belirteç listesine düzleştirin
        tokens = [token for line in tokens for token in line]
    return collections.Counter(tokens)
```

Zaman makinesi veri kümesini külliyat olarak kullanarak bir kelime dağarcığı oluşturuyoruz. Daha sonra ilk birkaç sık andıcı dizinleriyle yazdırıyoruz.

```
vocab = Vocab(tokens)
print(list(vocab.token_to_idx.items())[:10])
```

```
[('<unk>', 0), ('the', 1), ('i', 2), ('and', 3), ('of', 4), ('a', 5), ('to', 6), ('was', 7),␣
↪('in', 8), ('that', 9)]
```

Şimdi her metin satırını sayısal indekslerin bir listesine dönüştürebiliriz.

```
for i in [0, 10]:
    print('words:', tokens[i])
    print('indices:', vocab[tokens[i]])
```

```
words: ['the', 'time', 'machine', 'by', 'h', 'g', 'wells']
indices: [1, 19, 50, 40, 2183, 2184, 400]
words: ['twinkled', 'and', 'his', 'usually', 'pale', 'face', 'was', 'flushed', 'and',
↪'animated', 'the']
indices: [2186, 3, 25, 1044, 362, 113, 7, 1421, 3, 1045, 1]
```

8.2.4 Her Şeyi Bir Araya Koymak

Yukarıdaki işlevleri kullanarak, `corpus`'u (külliyat), andıç indekslerinin bir listesidir, ve `vocab`'ı döndüren `load_corpus_time_machine` işlevine her şeyi paketliyoruz. Burada yaptığımız değişiklikler şunlardır: (i) Metni daha sonraki bölümlerdeki eğitimi basitleştirmek için kelimelere değil, karakterlere andıçlıyoruz; (ii) `corpus` tek bir listedir, andıç listelerinin listesi değildir, çünkü zaman makinesi veri kümesindeki her metin satırı mutlaka bir cümle veya paragraf değildir.

```
def load_corpus_time_machine(max_tokens=-1):  #@save
    """Andıç indislerini ve zaman makinesi veri kümesinin kelime dağarcığını döndür."""
    lines = read_time_machine()
    tokens = tokenize(lines, 'char')
    vocab = Vocab(tokens)
    # Zaman makinesi veri kümesindeki her metin satırı mutlaka bir cümle veya
    # paragraf olmadığından, tüm metin satırlarını tek bir listede düzleştirin
    corpus = [vocab[token] for line in tokens for token in line]
    if max_tokens > 0:
        corpus = corpus[:max_tokens]
    return corpus, vocab

corpus, vocab = load_corpus_time_machine()
len(corpus), len(vocab)
```

```
(170580, 28)
```

8.2.5 Özet

- Metin, dizi verilerinin önemli bir biçimidir.
- Metni ön işlemek için genellikle metni andıçlara böleriz, andıç dizgilerini sayısal indekslere eşlemek için bir kelime dağarcığı oluştururuz ve modellerin işlenmesi için metin verilerini andıç dizinlerine dönüştürürüz.

8.2.6 Alıştırmalar

1. Andıçlama önemli bir ön işleme adımıdır. Farklı diller için değişiktir. Metni andıçlamak için yaygın olarak kullanılan üç yöntem daha bulmaya çalışın.
2. Bu bölümün deneyinde, metni sözcüklere andıçlayın ve `Vocab` örneğinin `min_freq` argümanlarını değiştirin. Bu, kelime dağarcığı boyutunu nasıl etkiler?

Tartışmalar[102]

[102] https://discuss.d2l.ai/t/115

8.3 Dil Modelleri ve Veri Kümesi

Section 8.2 içinde, metin verilerini andıçlara nasıl eşleyeceğimizi görüyoruz; burada bu andıçlar, sözcükler veya karakterler gibi ayrık gözlemler dizisi olarak görülebiliyor. T uzunluğunda bir metin dizisinde andıçların sırayla $x_1, x_2, \ldots, x_T$ olduğunu varsayalım. Daha sonra, metin dizisinde, x_t ($1 \leq t \leq T$), t adımındaki gözlem veya etiket olarak kabul edilebilir. Böyle bir metin dizisi göz önüne alındığında, *dil modelinin amacı* dizinin bileşik olasılığını tahmin etmektir

$$P(x_1, x_2, \ldots, x_T). \tag{8.3.1}$$

Dil modelleri inanılmaz derecede kullanışlıdır. Örneğin, ideal bir dil modeli, sadece bir seferde bir andıç çekerek tek başına doğal metin oluşturabilir: $x_t \sim P(x_t \mid x_{t-1}, \ldots, x_1)$. Daktilo kullanan maymunun aksine, böyle bir modelden çıkan tüm metinler doğal dil, örneğin İngilizce metin olarak geçer. Ayrıca, sadece metni önceki diyalog parçalarına koşullandırarak anlamlı bir diyalog oluşturmak için yeterli olacaktır. Açıkçası, böyle bir sistemi tasarlamaktan çok uzaktayız, çünkü sadece dilbilgisi olarak mantıklı içerik üretmek yerine metni *anlamak* gerekir.

Bununla birlikte, dil modelleri sınırlı biçimlerde bile mükemmel hizmet vermektedir. Örneğin, "konuşmayı tanımak" ve "konuşma tınmak" ifadeleri çok benzerdir. Bu konuşma tanımada belirsizliğe neden olabilir, ki ikinci çeviriyi tuhaf görüp reddeden bir dil modeli aracılığıyla kolayca çözülebilir. Aynı şekilde, bir belge özetleme algoritmasında "köpek adamı ısırdı" ifadesinin "adam köpeği ısırdı" ifadesinden çok daha sık ya da "Ben büyükanne yemek istiyorum" oldukça rahatsız edici bir ifade iken, "Ben yemek istiyorum, büyükanne" cümlesinin çok daha anlamlı olduğunu bilmek önemlidir.

8.3.1 Dil Modeli Öğrenme

Bariz soru, bir belgeyi, hatta bir dizi andıcı nasıl modellememiz gerektiğidir. Metin verilerini kelime düzeyinde andıçladığımızı varsayalım. Section 8.1 içindeki dizi modellerine uyguladığımız analize başvuruda bulunabiliriz. Temel olasılık kurallarını uygulayarak başlayalım:

$$P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, \ldots, x_{t-1}). \tag{8.3.2}$$

Örneğin, dört kelime içeren bir metin dizisinin olasılığı şu şekilde verilecektir:

$$\begin{aligned} P(\text{derin}, \text{öğrenme}, \text{çok}, \text{eğlencelidir}) =& P(\text{derin})P(\text{öğrenme} \mid \text{derin})P(\text{çok}) \mid \text{derin}, \text{öğrenme}) \\ & P(\text{eğlencelidir} \mid \text{derin}, \text{öğrenme}, \text{çok}) \end{aligned} \tag{8.3.3}$$

Dil modelini hesaplamak için, kelimelerin olasılığını ve bir kelimenin önceki birkaç kelimeye koşullu olasılığını hesaplamamız gerekir. Bu olasılıklar esasen dil modeli parametreleridir.

Burada, eğitim veri kümelerinin tüm Vikipedi girdileri, Gutenberg Projesi[103] ve Web'de yayınlanan tüm metinler gibi büyük bir metin külliyatı olduğunu varsayıyoruz. Kelimelerin olasılığı, eğitim veri kümesindeki belirli bir kelimenin göreceli kelime frekansından hesaplanabilir. Örneğin, $\hat{P}(\text{deep})$ tahmini, "derin" kelimesiyle başlayan herhangi bir cümlenin olasılığı olarak hesaplanabilir. Biraz daha az doğru bir yaklaşım, "derin" kelimesinin tüm oluşlarını saymak ve onu külliyat içindeki toplam kelime sayısına bölmek olacaktır. Bu, özellikle sık kullanılan kelimeler için

[103] https://en.wikipedia.org/wiki/Project_Gutenberg

oldukça iyi çalışır. Devam edersek, tahmin etmeyi deneyebiliriz:

$$\hat{P}(\text{öğrenme} \mid \text{derin}) = \frac{n(\text{derin, öğrenme})}{n(\text{derin})}, \tag{8.3.4}$$

burada $n(x)$ ve $n(x, x')$, sırasıyla tekli ve ardışık kelime çiftlerinin oluşlarının sayısıdır. Ne yazık ki, bir kelime çiftinin olasılığını tahmin etmek biraz daha zordur, çünkü "derin öğrenme" oluşları çok daha az sıklıktadır. Özellikle, bazı olağandışı kelime birleşimleri için, doğru tahminler elde ederken yeterli oluş bulmak zor olabilir. İşler üç kelimelik birleşimler ve ötesi için daha da kötüsü bir hal alır. Veri kümemizde muhtemelen göremeyeceğimiz birçok makul üç kelimelik birleşim olacaktır. Bu tür sözcük birleşimlerine sıfır olmayan sayım atamak için bazı çözümler sağlamadığımız sürece, bunları bir dil modelinde kullanamayacağız. Veri kümesi küçükse veya kelimeler çok nadirse, bunlardan bir tanesini bile bulamayabiliriz.

Genel bir strateji, bir çeşit *Laplace düzleştirme*si uygulamaktır. Çözüm, tüm sayımlara küçük bir sabit eklemektir. n ile eğitim kümesindeki toplam kelime sayısını ve m ile benzersiz kelimelerin sayısını gösterelim. Bu çözüm, teklilerde yardımcı olur, örn.

$$\begin{aligned}
\hat{P}(x) &= \frac{n(x) + \epsilon_1/m}{n + \epsilon_1}, \\
\hat{P}(x' \mid x) &= \frac{n(x, x') + \epsilon_2 \hat{P}(x')}{n(x) + \epsilon_2}, \\
\hat{P}(x'' \mid x, x') &= \frac{n(x, x', x'') + \epsilon_3 \hat{P}(x'')}{n(x, x') + \epsilon_3}.
\end{aligned} \tag{8.3.5}$$

Burada ϵ_1, ϵ_2 ve ϵ_3 hiper parametrelerdir. Örnek olarak ϵ_1'yi alın: $\epsilon_1 = 0$ olduğunda, düzleştirme uygulanmaz; ϵ_1 pozitif sonsuzluğa yaklaştığında, $\hat{P}(x)$ tekdüze $1/m$ olasılığına yaklaşır. Yukarıdakiler, diğer tekniklerin başarabileceklerinin oldukça ilkel bir türüdür (Wood *et al.*, 2011).

Ne yazık ki, bunun gibi modeller aşağıdaki nedenlerden dolayı oldukça hızlı bir şekilde hantallaşır. İlk olarak, tüm sayımları saklamamız gerekir. İkincisi, kelimelerin anlamı tamamen görmezden gelinir. Örneğin, "kedi" ve "pisi" ilgili bağlamlarda ortaya çıkmalıdır. Bu tür modelleri ek bağlamlara ayarlamak oldukça zordur, oysa, derin öğrenme tabanlı dil modelleri bunu dikkate almak için çok uygundur. Son olarak, uzun kelime dizilerinin sıradışı olması neredeyse kesindir, bu nedenle sadece daha önce görülen kelime dizilerinin sıklığını sayan bir model orada kötü başarım göstermeye mahkumdur.

8.3.2 Markov Modeller ve n-gram

Derin öğrenmeyi içeren çözümleri tartışmadan önce, daha fazla terime ve kavrama ihtiyacımız var. Section 8.1 içindeki Markov Modelleri hakkındaki tartışmamızı hatırlayın. Bunu dil modellemesine uygulayalım. Eğer $P(x_{t+1} \mid x_t, \ldots, x_1) = P(x_{t+1} \mid x_t)$ ise, diziler üzerinden bir dağılım birinci mertebeden Markov özelliğini karşılar. Daha yüksek mertebeler daha uzun bağımlılıklara karşılık gelir. Bu, bir diziyi modellemek için uygulayabileceğimiz birtakım yaklaşımlara yol açar:

$$\begin{aligned}
P(x_1, x_2, x_3, x_4) &= P(x_1)P(x_2)P(x_3)P(x_4), \\
P(x_1, x_2, x_3, x_4) &= P(x_1)P(x_2 \mid x_1)P(x_3 \mid x_2)P(x_4 \mid x_3), \\
P(x_1, x_2, x_3, x_4) &= P(x_1)P(x_2 \mid x_1)P(x_3 \mid x_1, x_2)P(x_4 \mid x_2, x_3).
\end{aligned} \tag{8.3.6}$$

Bir, iki ve üç değişken içeren olasılık formülleri genellikle sırasıyla *unigram*, *bigram* ve *trigram* modelleri olarak adlandırılır. Aşağıda, daha iyi modellerin nasıl tasarlanacağını öğreneceğiz.

8.3.3 Doğal Dil İstatistikleri

Bunun gerçek veriler üzerinde nasıl çalıştığını görelim. Section 8.2 içinde tanıtılan zaman makinesi veri kümesine dayanan bir kelime dağarcığı oluşturuyoruz ve en sık kullanılan 10 kelimeyi basıyoruz.

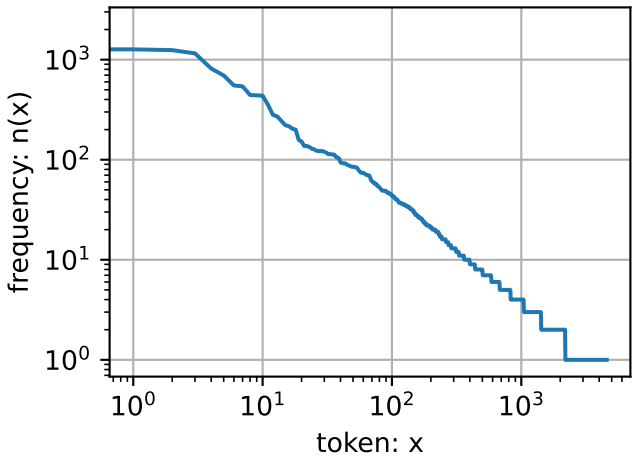
```
import random
import torch
from d2l import torch as d2l
```

```
tokens = d2l.tokenize(d2l.read_time_machine())
# Her metin satırı mutlaka bir cümle veya paragraf olmadığı için tüm metin
# satırlarını bitiştiririz
corpus = [token for line in tokens for token in line]
vocab = d2l.Vocab(corpus)
vocab.token_freqs[:10]
```

```
[('the', 2261),
 ('i', 1267),
 ('and', 1245),
 ('of', 1155),
 ('a', 816),
 ('to', 695),
 ('was', 552),
 ('in', 541),
 ('that', 443),
 ('my', 440)]
```

Gördüğümüz gibi, en popüler kelimelere bakmak aslında oldukça sıkıcı. Genellikle *duraklama kelimeleri* olarak adlandırılır ve böylece filtrelenir. Yine de, hala anlam taşırlar ve biz de onları kullanacağız. Ayrıca, kelime frekansının oldukça hızlı bir şekilde sönümlendiği oldukça açıktır. 10. en sık kullanılan kelime, en popüler olanının $1/5$'nden daha az yaygındır. Daha iyi bir fikir elde etmek için, kelime frekansının şeklini çizdiriyoruz.

```
freqs = [freq for token, freq in vocab.token_freqs]
d2l.plot(freqs, xlabel='token: x', ylabel='frequency: n(x)',
         xscale='log', yscale='log')
```



Burada oldukça temel bir şey üzerindeyiz: Kelime frekansı iyi tanımlanmış bir şekilde hızla sönümleniyor. İlk birkaç kelime istisna olarak ele alındıktan sonra, kalan tüm kelimeler log-log figür üzerinde kabaca düz bir çizgi izler. Bu, kelimelerin en sık i. kelimesinin n_i frekansının aşağıdaki gibi olduğunu belirten *Zipf yasası*nı tatmin ettiği anlamına gelir:

$$n_i \propto \frac{1}{i^\alpha}, \tag{8.3.7}$$

ki o da aşağıdaki ifadeye eşdeğerdir

$$\log n_i = -\alpha \log i + c, \tag{8.3.8}$$

burada α, dağılımı karakterize eden üstür ve c bir sabittir. İstatistiklerine saymaya ve düzleştirmeye göre kelimeleri modellemek istiyorsak, zaten burada ara vermeliyiz. Sonuçta, nadir kelimeler olarak da bilinen kuyruk sıklığına önemli ölçüde saparak fazla değer vereceğiz. Peki ya diğer kelime birleşimleri, örneğin bigramlar, trigramlar ve ötesi? Bigram frekansının unigram frekansı ile aynı şekilde davranıp davranmadığını görelim.

```
bigram_tokens = [pair for pair in zip(corpus[:-1], corpus[1:])]
bigram_vocab = d2l.Vocab(bigram_tokens)
bigram_vocab.token_freqs[:10]
```

```
[(('of', 'the'), 309),
 (('in', 'the'), 169),
 (('i', 'had'), 130),
 (('i', 'was'), 112),
 (('and', 'the'), 109),
 (('the', 'time'), 102),
 (('it', 'was'), 99),
 (('to', 'the'), 85),
 (('as', 'i'), 78),
 (('of', 'a'), 73)]
```

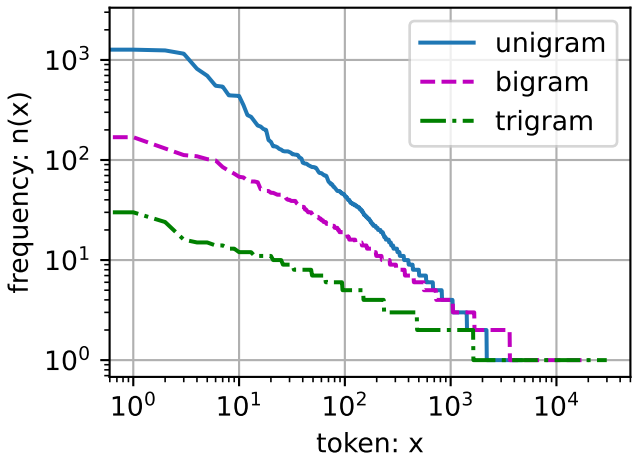
Burada dikkat çeken bir şey var. En sık görülen on kelime çiftinden dokuzunun üyeleri duraklama kelimelerinden oluşur ve yalnızca bir tanesi gerçek kitapla ("zaman") ilgilidir. Ayrıca, trigram frekansının aynı şekilde davranıp davranmadığını görelim.

```
trigram_tokens = [triple for triple in zip(
    corpus[:-2], corpus[1:-1], corpus[2:])]
trigram_vocab = d2l.Vocab(trigram_tokens)
trigram_vocab.token_freqs[:10]
```

```
[(('the', 'time', 'traveller'), 59),
 (('the', 'time', 'machine'), 30),
 (('the', 'medical', 'man'), 24),
 (('it', 'seemed', 'to'), 16),
 (('it', 'was', 'a'), 15),
 (('here', 'and', 'there'), 15),
 (('seemed', 'to', 'me'), 14),
 (('i', 'did', 'not'), 14),
 (('i', 'saw', 'the'), 13),
 (('i', 'began', 'to'), 13)]
```

Son olarak, bu üç model arasında andıç frekansını görselleştirmemize izin verin: Unigramlar, bigramlar ve trigramlar.

```
bigram_freqs = [freq for token, freq in bigram_vocab.token_freqs]
trigram_freqs = [freq for token, freq in trigram_vocab.token_freqs]
d2l.plot([freqs, bigram_freqs, trigram_freqs], xlabel='token: x',
         ylabel='frequency: n(x)', xscale='log', yscale='log',
         legend=['unigram', 'bigram', 'trigram'])
```



Bu şekil birtakım nedenlerden dolayı oldukça heyecan verici. Birincisi, unigram kelimelerin ötesinde, kelime dizileri dizi uzunluğuna bağlı olarak (8.3.7) içinde α'da daha küçük bir üs ile de olsa Zipf yasasını takip ediyor gibi görünmektedir. İkincisi, farklı n-gram sayısı o kadar büyük değildir. Bu bize dilde çok fazla yapı olduğuna dair umut veriyor. Üçüncü olarak, birçok n-gram çok nadiren ortaya çıkar, bu da Laplace düzleştirmeyi dil modellemesi için uygunsuz hale getirir. Bunun yerine, derin öğrenme tabanlı modeller kullanacağız.

8.3.4 Uzun Dizi Verilerini Okuma

Dizi verileri doğası gereği dizili olduğundan, işleme konusunu ele almamız gerekiyor. Bunu Section 8.1 içinde oldukça geçici bir şekilde yaptık. Diziler, modeller tarafından tek seferde işlenemeyecek kadar uzun olduğunda, bu tür dizileri okumak için bölmek isteyebiliriz. Şimdi genel stratejileri tanımlayalım. Modeli tanıtmadan önce, ağın önceden tanımlanmış uzunlukta, bir seferde n zaman adımı mesela, bir minigrup dizisini işlediği bir dil modelini eğitmek için bir sinir ağı kullanacağımızı varsayalım. Şimdi asıl soru, özniteliklerin ve etiketlerin minigruplarının rastgele nasıl okunacağıdır.

Başlarken, bir metin dizisi keyfi uzunlukta olabileceğinden, *Zaman Makinesi* kitabının tamamı gibi, bu kadar uzun bir diziyi aynı sayıda zaman adımlı altdizilere parçalayabiliriz. Sinir ağımızı eğitirken, bu tür altdizilerden bir minigrup modele beslenecektir. Ağın bir seferde n zaman adımlı bir altdizisini işlediğini varsayalım. Fig. 8.3.1, orijinal bir metin dizisinden altdizi elde etmenin tüm farklı yollarını gösterir; burada $n = 5$ ve her seferinde bir andıç bir karaktere karşılık gelir. Başlangıç pozisyonunu gösteren keyfi bir bağıl konum (offset) seçebileceğimizden oldukça özgür olduğumuzu unutmayın.

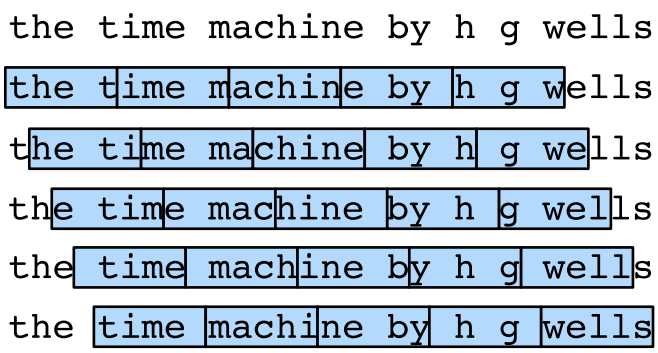


Fig. 8.3.1: Farklı bağıl konumlar, metni bölerken farklı altdizilere yol açar.

Bu nedenle, Fig. 8.3.1 içinde gösterilenden hangisini seçmeliyiz? Aslında, hepsi eşit derecede iyidir. Ancak, sadece bir bağıl konum seçersek, ağımızı eğitmek için olası tüm altdizilerin sınırlı kapsamı vardır. Bu nedenle, hem *kapsama* hem de *rasgelelik* elde etmek için bir diziyi bölümlerken rastgele bir bağıl konum ile başlayabiliriz. Aşağıda, bunu hem *rastgele örnekleme* hem de *sıralı bölümleme* stratejileri için nasıl gerçekleştireceğimizi açıklıyoruz.

Rastgele Örnekleme

Rastgele örneklemede, her örnek, orijinal uzun dizide keyfi olarak yakalanan bir altdizidir. Yineleme sırasında iki bitişik rasgele minigruptaki altdiziler mutlaka orijinal dizide bitişik olmak zorunda değildir. Dil modellemesi için hedef, şimdiye kadar gördüğümüz andıçlara dayanan bir sonraki andıcı tahmin etmektir, bu nedenle etiketler bir andıç kaydırılmış orijinal dizidir.

Aşağıdaki kod, her seferinde verilerden bir rasgele minigrup oluşturur. Burada, `batch_size` bağımsız değişkeni her minigruptaki altdizi örneklerinin sayısını belirtir ve `num_steps` her altdizideki zaman adımlarının önceden tanımlanmış sayısıdır.

```
def seq_data_iter_random(corpus, batch_size, num_steps):  #@save
    """Rastgele örnekleme kullanarak küçük bir dizi alt dizi oluşturun."""
    # Bir diziyi bölmek için rastgele bir kayma ile başlayın
    # (`num_steps - 1` dahil)
    corpus = corpus[random.randint(0, num_steps - 1):]
    # Etiketleri hesaba katmamız gerektiğinden 1 çıkarın
    num_subseqs = (len(corpus) - 1) // num_steps
    # `num_steps` uzunluğundaki alt diziler için başlangıç indeksleri
    initial_indices = list(range(0, num_subseqs * num_steps, num_steps))
    # Rastgele örneklemede, yineleme sırasında iki bitişik rastgele
    # minigruptan gelen alt diziler, orijinal dizide mutlaka bitişik değildir
    random.shuffle(initial_indices)

    def data(pos):
        # `pos`'dan başlayarak `num_steps` uzunluğunda bir dizi döndür
        return corpus[pos: pos + num_steps]

    num_batches = num_subseqs // batch_size
    for i in range(0, batch_size * num_batches, batch_size):
        # Burada, `initial_indices`, alt diziler için rastgele başlangıç
        #  dizinlerini içerir
```

(continues on next page)

(continued from previous page)

```
        initial_indices_per_batch = initial_indices[i: i + batch_size]
        X = [data(j) for j in initial_indices_per_batch]
        Y = [data(j + 1) for j in initial_indices_per_batch]
        yield torch.tensor(X), torch.tensor(Y)
```

Elimiz ile 0'dan 34'e kadar bir dizi oluşturalım. Biz grup boyutunun ve zaman adımlarının sayısının sırasıyla 2 ve 5 olduğunu varsayalım. Bu, $\lfloor (35 - 1)/5 \rfloor = 6$ öznitelik-etiket altdizi çiftleri üretebileceğimiz anlamına gelir. 2 minigrup boyutu ile sadece 3 minigrup elde ediyoruz.

```
my_seq = list(range(35))
for X, Y in seq_data_iter_random(my_seq, batch_size=2, num_steps=5):
    print('X: ', X, '\nY:', Y)
```

```
X:  tensor([[ 7,  8,  9, 10, 11],
        [27, 28, 29, 30, 31]])
Y: tensor([[ 8,  9, 10, 11, 12],
        [28, 29, 30, 31, 32]])
X:  tensor([[17, 18, 19, 20, 21],
        [22, 23, 24, 25, 26]])
Y: tensor([[18, 19, 20, 21, 22],
        [23, 24, 25, 26, 27]])
X:  tensor([[12, 13, 14, 15, 16],
        [ 2,  3,  4,  5,  6]])
Y: tensor([[13, 14, 15, 16, 17],
        [ 3,  4,  5,  6,  7]])
```

Sıralı Bölümleme

Orijinal dizinin rastgele örneklemesine ek olarak, yineleme sırasında iki bitişik minigrubun altdizilerinin orijinal dizide bitişik olmasını da sağlayabiliriz. Bu strateji, minigruplar üzerinde yineleme yaparken bölünmüş altdizilerin sırasını korur, dolayısıyla sıralı bölümleme olarak adlandırılır.

```
def seq_data_iter_sequential(corpus, batch_size, num_steps):  #@save
    """Sıralı bölümlemeyi kullanarak küçük bir alt dizi dizisi oluşturun."""
    # Bir diziyi bölmek için rastgele bir bağıl konum ile başlayın
    offset = random.randint(0, num_steps)
    num_tokens = ((len(corpus) - offset - 1) // batch_size) * batch_size
    Xs = torch.tensor(corpus[offset: offset + num_tokens])
    Ys = torch.tensor(corpus[offset + 1: offset + 1 + num_tokens])
    Xs, Ys = Xs.reshape(batch_size, -1), Ys.reshape(batch_size, -1)
    num_batches = Xs.shape[1] // num_steps
    for i in range(0, num_steps * num_batches, num_steps):
        X = Xs[:, i: i + num_steps]
        Y = Ys[:, i: i + num_steps]
        yield X, Y
```

Aynı ayarları kullanarak, sıralı bölümleme tarafından okunan her altdizi minigrubu için `X` özniteliklerini ve `Y` etiketleri yazdıracağız. Yineleme sırasında iki bitişik minigrubun altdizilerinin orijinal dizisi üzerinde gerçekten bitişik olduğunu unutmayın.

```
for X, Y in seq_data_iter_sequential(my_seq, batch_size=2, num_steps=5):
    print('X: ', X, '\nY:', Y)
```

```
X:  tensor([[ 4,  5,  6,  7,  8],
        [19, 20, 21, 22, 23]])
Y: tensor([[ 5,  6,  7,  8,  9],
        [20, 21, 22, 23, 24]])
X:  tensor([[ 9, 10, 11, 12, 13],
        [24, 25, 26, 27, 28]])
Y: tensor([[10, 11, 12, 13, 14],
        [25, 26, 27, 28, 29]])
X:  tensor([[14, 15, 16, 17, 18],
        [29, 30, 31, 32, 33]])
Y: tensor([[15, 16, 17, 18, 19],
        [30, 31, 32, 33, 34]])
```

Şimdi yukarıdaki iki örnekleme işlevini bir sınıf ile sarmalıyoruz, böylece daha sonra bir veri yineleyici olarak kullanabiliriz.

```
class SeqDataLoader:  #@save
    """Sıra verilerini yüklemek için bir yineleyici."""
    def __init__(self, batch_size, num_steps, use_random_iter, max_tokens):
        if use_random_iter:
            self.data_iter_fn = d2l.seq_data_iter_random
        else:
            self.data_iter_fn = d2l.seq_data_iter_sequential
        self.corpus, self.vocab = d2l.load_corpus_time_machine(max_tokens)
        self.batch_size, self.num_steps = batch_size, num_steps

    def __iter__(self):
        return self.data_iter_fn(self.corpus, self.batch_size, self.num_steps)
```

Son olarak, hem veri yineleyiciyi hem de kelime dağarcığını döndüren bir `load_data_time_machine` işlevi tanımlarız, böylece Section 3.5 içinde tanımlanan `d2l.load_data_fashion_mnist` gibi `load_data` öneki ile diğer işlevlerle benzer şekilde kullanabiliriz.

```
def load_data_time_machine(batch_size, num_steps,  #@save
                           use_random_iter=False, max_tokens=10000):
    """Zaman makinesi veri kümesinin yineleyicisini ve sözcük dağarcığını döndür."""
    data_iter = SeqDataLoader(
        batch_size, num_steps, use_random_iter, max_tokens)
    return data_iter, data_iter.vocab
```

8.3.5 Özet

- Dil modelleri doğal dil işlemenin anahtarıdır.
- n-gram, bağımlılığı budayarak uzun dizilerle başa çıkmak için uygun bir model sağlar.
- Uzun diziler, çok nadiren olma veya neredeyse hiç olmama problemden muzdariptir.
- Zipf yasası sadece unigram değil, aynı zamanda diğer n-gramlar için de kelime dağılımını yönetir.
- Çok fazla yapı var, ancak Laplace düzleştirmesi yoluyla nadir kelime birleşimleri ile verimli bir şekilde başa çıkmak için yeterli sıklıkta kullanım yok.
- Uzun dizileri okumak için ana seçenekler rastgele örnekleme ve sıralı bölümlemedir. İkincisi, yineleme sırasında iki bitişik minigruptaki altdizilerin orijinal dizide de bitişik olmasını sağlar.

8.3.6 Alıştırmalar

1. Eğitim veri kümesinde 100000 kelime olduğunu varsayalım. Dört-gramın ne kadar kelime frekansı ve çok kelimeli bitişik frekansı depolaması gerekiyor?
2. Bir diyaloğu nasıl modellersiniz?
3. Unigramlar, bigramlar ve trigramlar için Zipf yasasının üssünü tahmin edin.
4. Uzun dizi verilerini okumak için başka hangi yöntemleri düşünebilirsiniz?
5. Uzun dizileri okumak için kullandığımız rastgele bağıl konumu düşünün.
    1. Rastgele bir bağıl konum olması neden iyi bir fikir?
    2. Belgedeki diziler üzerinde gerçekten mükemmel bir şekilde tekdüze bir dağılıma yol açar mı?
    3. İşleri daha tekdüze hale getirmek için ne yapmalısınız?
6. Eğer bir dizi örneğinin tam bir cümle olmasını istiyorsak, bu minigrup örneklemede ne tür bir sorun ortaya çıkarır? Bu sorunu nasıl çözebiliriz?

Tartışmalar[104]

8.4 Yinelemeli Sinir Ağları

Section 8.3 içinde n-gramlık modelleri tanıttık; x_t kelimesinin t zaman adımındaki koşullu olasılığı sadece önceki $n-1$ kelimeye bağlıdır. Eğer x_t üzerinde $t-(n-1)$ zaman adımından daha önceki kelimelerin olası etkisini dahil etmek istiyorsanız, n'yi artırmanız gerekir. Bununla birlikte, model parametrelerinin sayısı da katlanarak artacaktır, çünkü $\mathcal{V}$ kelime dağarcığı kümesi için $|\mathcal{V}|^n$ tane değer depolamamız gerekir. Bu nedenle, $P(x_t \mid x_{t-1}, \ldots, x_{t-n+1})$'yı modellemek yerine bir saklı değişken modeli kullanılmak tercih edilir:

$$P(x_t \mid x_{t-1}, \ldots, x_1) \approx P(x_t \mid h_{t-1}), \tag{8.4.1}$$

[104] https://discuss.d2l.ai/t/118

burada h_{t-1}, $t-1$ adıma kadar dizi bilgisi depolayan bir *gizli durum*dur (gizli değişken olarak da bilinir). Genel olarak, herhangi bir t zaman adımındaki gizli durum, şimdiki girdi x_t ve önceki gizli durum h_{t-1} temel alınarak hesaplanabilir:

$$h_t = f(x_t, h_{t-1}). \tag{8.4.2}$$

(8.4.2) içindeki yeterince güçlü bir f işlevi için, saklı değişken modeli bir yaklaşım değildir. Sonuçta, h_t şimdiye kadar gözlemlediği tüm verileri saklayabilir. Ancak, potansiyel olarak hem hesaplamayı hem de depolamayı pahalı hale getirebilir.

Chapter 4 içinde gizli birimli gizli katmanları tartıştığımızı anımsayın. Gizli katmanların ve gizli durumların iki çok farklı kavramı ifade etmeleri önemlidir. Gizli katmanlar, açıklandığı gibi, girdiden çıktıya giden yolda gözden gizlenen katmanlardır. Gizli durumlar teknik olarak belirli bir adımda yaptığımız işleme *girdiler*dir ve yalnızca önceki zaman adımlarındaki verilere bakarak hesaplanabilirler.

Yinelemeli sinir ağları (RNN) gizli durumlara sahip sinir ağlarıdır. RNN modelini tanıtmadan önce, ilk olarak Section 4.1 içinde tanıtılan MLP modelini anımsayalım.

8.4.1 Gizli Durumu Olmayan Sinir Ağları

Tek bir gizli katmana sahip bir MLP'ye bir göz atalım. Gizli katmanın etkinleştirme işlevinin ϕ olduğunu varsayalım. n küme boyutlu ve d girdili bir minigrup örneklemi $\mathbf{X} \in \mathbb{R}^{n \times d}$ göz önüne alındığında, gizli katmanın çıktısı $\mathbf{H} \in \mathbb{R}^{n \times h}$ şöyle hesaplanır:

$$\mathbf{H} = \phi(\mathbf{X}\mathbf{W}_{xh} + \mathbf{b}_h). \tag{8.4.3}$$

(8.4.3)'te, $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$ ağırlık parametresine, $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ ek girdi parametresine ve gizli katman için h gizli birim adedine sahibiz. Böylece, toplama esnasında yayınlama (bkz. Section 2.1.3) uygulanır. Ardından, çıktı katmanının girdisi olarak $\mathbf{H}$ gizli değişkeni kullanılır. Çıktı katmanı şöyle gösterilir,

$$\mathbf{O} = \mathbf{H}\mathbf{W}_{hq} + \mathbf{b}_q, \tag{8.4.4}$$

burada $\mathbf{O} \in \mathbb{R}^{n \times q}$ çıktı değişkeni, $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$ ağırlık parametresi ve $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ çıktı katmanının ek girdi parametresidir. Eğer bir sınıflandırma problemi ise, çıktı kategorilerinin olasılık dağılımını hesaplamak için softmaks($\mathbf{O}$)'i kullanabiliriz.

Bu, Section 8.1 içinde daha önce çözdüğümüz bağlanım problemine tamamen benzer, dolayısıyla ayrıntıları atlıyoruz. Öznitelik-etiket çiftlerini rastgele seçebileceğimizi ve ağımızın parametrelerini otomatik türev alma ve rasgele eğim inişi yoluyla öğrenebileceğimizi söylemek yeterli.

8.4.2 Gizli Durumlu Yinelemeli Sinir Ağları

Gizli durumlarımız olduğunda işler tamamen farklıdır. Yapıyı biraz daha ayrıntılı olarak inceleyelim.

t zaman adımında girdileri $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ olan bir minigrubumuz olduğunu varsayalım. Başka bir deyişle, n dizi örneklerinden oluşan bir minigrup için, $\mathbf{X}_t$'nin her satırı, dizinin t adımındaki bir örneğine karşılık gelir. Ardından, $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ ile t zaman adımındaki gizli değişkeni belirtelim. MLP'den farklı olarak, burada gizli değişkeni $\mathbf{H}_{t-1}$'yi önceki zaman adımından kaydediyoruz ve şimdiki zaman adımında önceki zaman adımının gizli değişkeni nasıl kullanılacağını açıklamak

için $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ yeni ağırlık parametresini tanıtıyoruz. Özellikle, şimdiki zaman adımının gizli değişkeninin hesaplanması, önceki zaman adımının gizli değişkeni ile birlikte şimdiki zaman adımının girdisi tarafından belirlenir:

$$\mathbf{H}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h). \tag{8.4.5}$$

(8.4.3) ile karşılaştırıldığında, (8.4.5) bir terim daha, $\mathbf{H}_{t-1}\mathbf{W}_{hh}$, ekler ve böylece (8.4.2)'den bir örnek oluşturur. Bitişik zaman adımlarındaki $\mathbf{H}_t$ ve $\mathbf{H}_{t-1}$ gizli değişkenlerinin arasındaki ilişkiden, bu değişkenlerin dizinin tarihsel bilgilerini şu anki zaman adımına kadar yakaladığını ve sakladığını biliyoruz; tıpkı sinir ağının şimdiki zaman adımının durumu veya hafızası gibi. Bu nedenle, böyle bir gizli değişken *gizli durum* olarak adlandırılır. Gizli durum şu anki zaman adımında önceki zaman adımının aynı tanımını kullandığından, (8.4.5)'nın hesaplanması *yinelemedir*. Bu nedenle, yinelemeli hesaplamalara dayalı gizli durumlara sahip sinir ağları *yinelemeli sinir ağları*dır. RNN'lerde (8.4.5)'ün hesaplanmasını gerçekleştiren katmanlar *yinelemeli katmanlar* olarak adlandırılır.

RNN oluşturmak için birçok farklı yol vardır. (8.4.5) içinde tanımlanan gizli bir duruma sahip RNN'ler çok yaygındır. Zaman adımı t için çıktı katmanının çıktısı MLP'deki hesaplamaya benzer:

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q. \tag{8.4.6}$$

RNN parametreleri gizli katmanın $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$, $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ ağırlıkları ve $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ ek girdisi ile birlikte çıktı katmanının $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$ ağırlıklarını ve $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ ek girdilerini içerir. Farklı zaman adımlarında bile, RNN'lerin her zaman bu model parametrelerini kullandığını belirtmek gerekir. Bu nedenle, bir RNN parametrelendirmenin maliyeti zaman adım sayısı arttıkça büyümez.

Fig. 8.4.1, bitişik üç zaman adımında bir RNN'nin hesaplama mantığını göstermektedir. Herhangi bir t zaman adımında, gizli durumun hesaplanması şu şekilde düşünülebilir: (i) t şimdiki zaman adımındaki $\mathbf{X}_t$ girdisi ile önceki $t-1$ zaman adımındaki $\mathbf{H}_{t-1}$ gizli durumu bitiştirme; (ii) bitiştirme sonucunu ϕ etkinleştirme fonksiyonlu tam bağlı bir katmana besleme. Bu şekilde tam bağlı bir katmanın çıktısı, t şimdiki zaman adımı $\mathbf{H}_t$ gizli durumudur. Bu durumda, model parametrelerinin hepsi (8.4.5)'teki $\mathbf{W}_{xh}$ ve $\mathbf{W}_{hh}$'ün bitiştirilmesi ve $\mathbf{b}_h$ ek girdisidir. Şimdiki t zaman adımının $\mathbf{H}_t$ gizli durumu, sonraki $t+1$ adımının $\mathbf{H}_{t+1}$ gizli durumunun hesaplanmasına katılacaktır. Dahası, $\mathbf{H}_t$, t şimdiki zaman adımının $\mathbf{O}_t$ çıktısını hesaplamak için tam bağlı çıktı katmanına da beslenir.

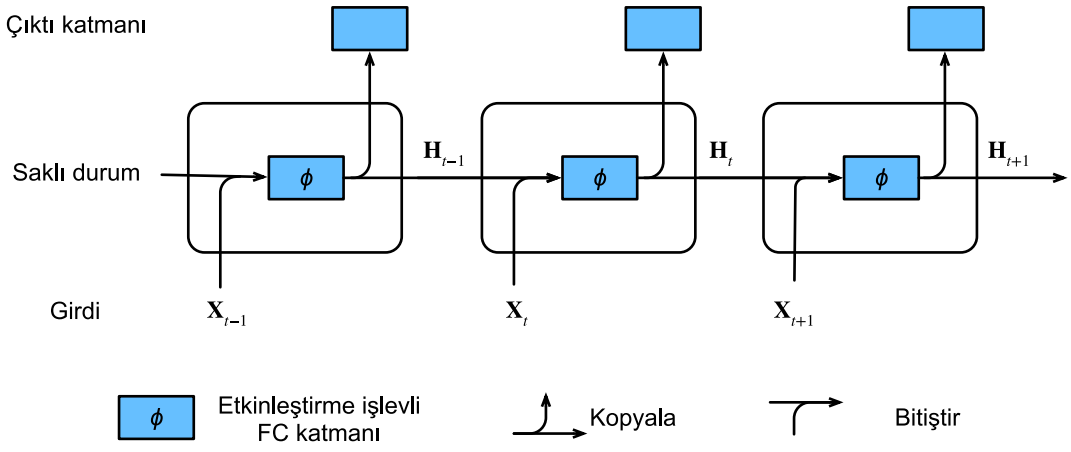


Fig. 8.4.1: Gizli duruma sahip bir RNN.

Gizli durum $\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh}$ hesaplamasının $\mathbf{X}_t$ ve $\mathbf{H}_{t-1}$ ile $\mathbf{W}_{xh}$ ve $\mathbf{W}_{hh}$ bitişik matrislerinin çarpmasına eşdeğer olduğunu belirttik. Bu matematiksel olarak kanıtlanmış olsa da, aşağıda kısaca

bunu göstermek için basit bir kod parçacığı kullanıyoruz. Başlangıç olarak, şekilleri (3, 1), (1, 4), (3, 4) ve (4, 4) olan `X`, `W_xh`, `H` ve `W_hh` matrislerini tanımlıyoruz. Sırasıyla `X` ile `W_xh`'yi ve `H` ile `W_hh`'yi çarpıyoruz ve daha sonra bu iki çarpımı toplayarak (3, 4) şekilli bir matris elde ederiz.

```
import torch
from d2l import torch as d2l
```

```
X, W_xh = torch.normal(0, 1, (3, 1)), torch.normal(0, 1, (1, 4))
H, W_hh = torch.normal(0, 1, (3, 4)), torch.normal(0, 1, (4, 4))
torch.matmul(X, W_xh) + torch.matmul(H, W_hh)
```

```
tensor([[ 0.3322,  1.6809,  0.7737,  2.3295],
        [ 4.1095,  4.0970, -2.6380,  4.0622],
        [ 3.0638, -3.0190,  1.5716,  0.1028]])
```

Şimdi `X` ve `H` matrislerini sütunlar boyunca (eksen 1) ve `W_xh` ve `W_hh` matrislerini satırlar boyunca (eksen 0) bitiştiririz. Bu iki bitiştirme, sırasıyla (3, 5) ve (5, 4) şekilli matrisler ile sonuçlanır. Bu iki bitiştirilmiş matrisi çarparak, yukarıdaki gibi (3, 4) şekilli aynı çıktı matrisini elde ederiz.

```
torch.matmul(torch.cat((X, H), 1), torch.cat((W_xh, W_hh), 0))
```

```
tensor([[ 0.3322,  1.6809,  0.7737,  2.3295],
        [ 4.1095,  4.0970, -2.6380,  4.0622],
        [ 3.0638, -3.0190,  1.5716,  0.1028]])
```

8.4.3 RNN Tabanlı Karakter Düzeyinde Dil Modelleri

Section 8.3 içindeki dil modellemesi için şimdiki ve geçmiş andıçlara dayanarak bir sonraki simgeyi tahmin etmeyi amaçladığımızı hatırlayın, böylece orijinal diziyi etiketler olarak bir andıç kaydırıyoruz. Bengio ve ark. önce dil modelleme için bir sinir ağı kullanmayı önerdi (Bengio *et al.*, 2003). Aşağıda, bir dil modeli oluşturmak için RNN'lerin nasıl kullanılabileceğini gösteriyoruz. Minigrup boyutu bir olsun ve metnin sırası "makine" (machine) olsun. Sonraki bölümlerdeki eğitimi basitleştirmek için, metni sözcükler yerine karakterler haline getiririz ve *karakter düzeyinde bir dil modeli*ni göz önünde bulundururuz. Fig. 8.4.2, karakter düzeyinde dil modellemesi için bir RNN aracılığıyla şimdiki ve önceki karakterlere dayanarak bir sonraki karakterin nasıl tahmin edileceğini gösterir.

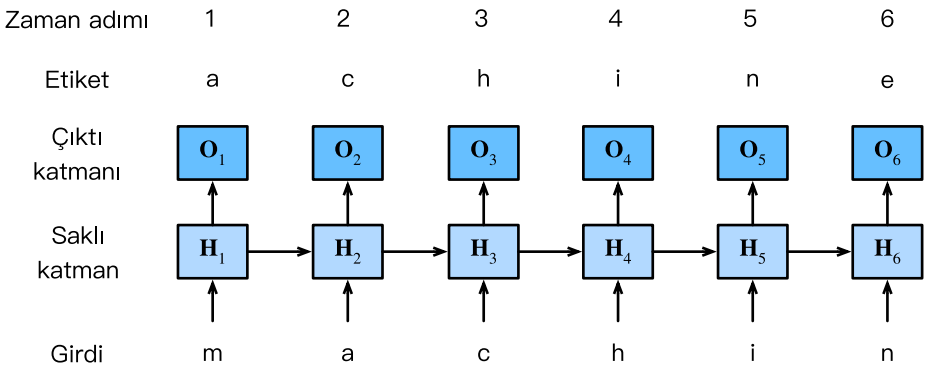


Fig. 8.4.2: RNN'ye dayalı karakter düzeyinde bir dil modeli. Girdi ve etiket dizileri sırasıyla "machin" ve "achine" dir.

Eğitim işlemi sırasında, çıktı katmanından çıktıda her zaman adım için bir softmaks işlemi çalıştırırız ve daha sonra model çıktısı ile etiket arasındaki hatayı hesaplamak için çapraz entropi kaybını kullanırız. Gizli katmandaki gizli durumun yinelemeli hesaplanması nedeniyle, Fig. 8.4.2, $\mathbf{O}_3$'teki 3. zaman adımının çıktısı, "m", "a" ve "c" metin dizisi ile belirlenir. Eğitim verilerindeki dizinin bir sonraki karakteri "h" olduğu için, 3. zaman adımının kaybı, "m", "a", "c""ye göre oluşturulan bir sonraki karakterin olasılık dağılımına ve bu zaman adımının "h" etiketine bağlı olacaktır.

Uygulamada, her andıç bir d boyutlu vektör ile temsil edilir ve grup boyutu olarak $n > 1$ kullanırız. Bu nedenle, t'deki $\mathbf{X}_t$ girdisi Section 8.4.2 içinde tartıştığımız gibi $n \times d$ şekilli bir matris olacaktır.

8.4.4 Şaşkınlık

Son olarak, sonraki bölümlerde RNN tabanlı modellerimizi değerlendirmek için kullanılacak dil modelinin kalitesini nasıl ölçeceğimizi tartışalım. Bir yol, metnin ne kadar şaşırtıcı olduğunu kontrol etmektir. İyi bir dil modeli, daha sonra ne göreceğimizi yüksek hassasiyetli andıçlarla tahmin edebilir. Farklı dil modelleri tarafından önerilen "Yağmur yağıyor" ifadesinin aşağıdaki devamlarını göz önünde bulundurun:

1. "Dışarıda yağmur yağıyor"
2. "Muz ağacı yağıyor"
3. "Piouw yağıyor; kcj pwepoiut"

Kalite açısından, örnek 1 açıkça en iyisidir. Sözcükler mantıklı ve mantıksal olarak da tutarlı. Hangi kelimenin anlamsal olarak takip ettiğini tam olarak doğru bir şekilde yansıtmayabilir ("San Francisco'da" ve "kışın" mükemmel şekilde makul uzantıları olurdu), ama model hangi kelimenin takip edebileceğini yakalayabilir. Örnek 2, mantıksız bir uzantı ürettiğinden oldukça kötüdür. Yine de, model en azından kelimelerin nasıl yazılacağını ve kelimeler arasındaki korelasyon derecesini öğrendi. Son olarak, örnek 3, veriye düzgün şekilde uymayan kötü eğitilmiş bir modeli gösterir.

Dizinin olabilirliğini hesaplayarak modelin kalitesini ölçebiliriz. Ne yazık ki bu, anlaşılması ve karşılaştırılması zor bir sayıdır. Sonuçta, daha kısa dizilerin daha uzun olanlara göre gerçekleşme olasılığı daha yüksektir, bu nedenle model Tolstoy'un şaheseri *Savaş ve Barış*'ı değerlendirirken kaçınılmaz olarak Saint-Exupery'nin "Küçük Prens" romanından çok daha küçük bir olabilirlik üretecektir. Eksik olan bir ortalama eşdeğeridir.

Bilgi teorisi burada işe yarar. Softmaks bağlanımını (Section 3.4.7) tanıttığımızda entropiyi, sürprizi ve çapraz entropiyi tanımladık ve bilgi teorisi üzerine çevrimiçi ek[105]'de bilgi teorisi daha fazla tartışılmaktadır. Metni sıkıştırmak istiyorsak, geçerli andıç kümesi verilince bir sonraki andıcı tahmin etmeyi sorabiliriz. Daha iyi bir dil modeli, bizim bir sonraki andıcı daha doğru tahmin etmemizi sağlamalıdır. Böylece, diziyi sıkıştırmak için daha az bit harcamamıza izin vermelidir. Böylece, bir dizinin tüm n andıçları üzerinden ortalamasıyla çapraz entropi kaybıyla ölçebiliriz:

$$\frac{1}{n}\sum_{t=1}^{n} -\log P(x_t \mid x_{t-1}, \ldots, x_1), \tag{8.4.7}$$

Burada P bir dil modeli tarafından verilir ve x_t, diziden t adımında gözlenen gerçek andıçtır. Bu, farklı uzunluklardaki belgelerdeki performansları karşılaştırılabilir hale getirir. Tarihsel nedenlerden dolayı, doğal dil işlemedeki bilim adamları *şaşkınlık (perplexity)* adı verilen bir ölçüm kullanmayı tercih ederler. Kısacası, (8.4.7)'ün üssüdür:

$$\exp\left(-\frac{1}{n}\sum_{t=1}^{n} \log P(x_t \mid x_{t-1}, \ldots, x_1)\right). \tag{8.4.8}$$

Şaşkınlık, en iyi hangi andıcı seçeceğimize karar verirken sahip olduğumuz gerçek seçeneklerin sayısının harmonik ortalaması olarak anlaşılabilir. Birkaç vakaya bakalım:

- En iyi senaryoda, model her zaman etiket andıcı olasılığını 1 olarak mükemmel şekilde tahmin eder. Bu durumda modelin şaşkınlığı 1'dir.
- En kötü senaryoda, model her zaman etiket andıcı olasılığını 0 olarak öngörür. Bu durumda şaşkınlık pozitif sonsuzdur.
- Referans olarak model, sözcük dağarcığının tüm kullanılabilir andıçları üzerinde tekdüze bir dağılım öngörür. Bu durumda, şaşkınlık, kelime dağarcığının benzersiz andıçlarının sayısına eşittir. Aslında, diziyi herhangi bir sıkıştırma olmadan saklarsak, kodlamak için yapabileceğimiz en iyi şey bu olurdu. Bu nedenle, bu, herhangi bir yararlı modelin yenmesi gereken bariz bir üst sınır sağlar.

Aşağıdaki bölümlerde, karakter düzeyi dil modelleri için RNN'leri uygulayacağız ve bu modelleri değerlendirmek için şaşkınlığı kullanacağız.

8.4.5 Özet

- Gizli durumlar için yinelemeli hesaplama kullanan bir sinir ağı, yinelemeli bir sinir ağı (RNN) olarak adlandırılır.
- Bir RNN'nin gizli durumu, dizinin şimdiki zaman adımına kadarki geçmiş bilgilerini tutabilir.
- Zaman adımlarının sayısı arttıkça RNN model parametrelerinin sayısı artmaz.
- Bir RNN kullanarak karakter düzeyinde dil modelleri oluşturabiliriz.
- Dil modellerinin kalitesini değerlendirmek için şaşkınlığı kullanabiliriz.

[105] https://d2l.ai/chapter_appendix-mathematics-for-deep-learning/information-theory.html

8.4.6 Alıştırmalar

1. Bir metin dizisindeki bir sonraki karakteri tahmin etmek için bir RNN kullanırsak, bir çıktı için gerekli boyut nedir?
2. RNN'ler bir zaman adımındaki andıcın metin dizisindeki önceki tüm andıçlara dayalı koşullu olasılığını nasıl ifade edebilir?
3. Uzun bir dizide geri yayarsak gradyana ne olur?
4. Bu bölümde açıklanan dil modeliyle ilgili sorunlar nelerdir?

Tartışmalar[106]

8.5 Yinelemeli Sinir Ağlarının Sıfırdan Uygulanması

Bu bölümde, Section 8.4 içindeki tanımlamalarımıza göre, karakter düzeyinde bir dil modeli için sıfırdan bir RNN uygulayacağız. Bu tarz bir model H. G. Wells'in *Zaman Makinesi* ile eğitilecek. Daha önce olduğu gibi, önce Section 8.3 içinde tanıtılan veri kümesini okuyarak başlıyoruz.

```
%matplotlib inline
import math
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

```
batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

8.5.1 Bire Bir Kodlama

Her andıcın `train_iter`'te sayısal bir indeks olarak temsil edildiğini hatırlayın. Bu indeksleri doğrudan sinir ağına beslemek öğrenmeyi zorlaştırabilir. Genellikle her andıcı daha açıklayıcı bir öznitelik vektörü olarak temsil ediyoruz. En kolay gösterim, Section 3.4.1 içinde tanıtılan *bire bir kodlama*dır (one-hot coding).

Özetle, her bir indeksi farklı bir birim vektörüne eşleriz: Kelime dağarcığındaki farklı andıçların sayısının N (`len(vocab)`) olduğunu ve andıç indekslerinin 0 ile $N-1$ arasında değiştiğini varsayalım. Bir andıç indeksi i tamsayısı ise, o zaman N uzunluğunda tümü 0 olan bir vektör oluştururuz ve i konumundaki elemanı 1'e ayarlarız. Bu vektör, orijinal andıcın bire bir kodlama vektörüdür. İndeksleri 0 ve 2 olan bire bir kodlama vektörleri aşağıda gösterilmiştir.

```
F.one_hot(torch.tensor([0, 2]), len(vocab))
```

```
tensor([[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0],
        [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0]])
```

[106] https://discuss.d2l.ai/t/1050

Her seferinde örnek aldığımız minigrubun şeklidir (grup boyutu, zaman adımlarının sayısı). `one_hot` işlevi, böyle bir minigrubu, son boyutun kelime dağarcığı uzunluğuna eşit olduğu üç boyutlu bir tensöre dönüştürür (`len(vocab)`). Girdiyi sıklıkla devirir (transpose), böylece (zaman adımlarının sayısı, grup boyutu, kelime dağarcığı uzunluğu) şeklinde bir çıktı elde edeceğiz. Bu, bir minigrubun gizli durumlarını zaman adımlarıyla güncellerken en dıştaki boyutta daha rahat döngü yapmamızı sağlayacaktır.

```
X = torch.arange(10).reshape((2, 5))
F.one_hot(X.T, 28).shape
```

```
torch.Size([5, 2, 28])
```

8.5.2 Model Parametrelerini İlkleme

Ardından, RNN modeli için model parametrelerini ilkliyoruz. Gizli birimlerin sayısı `num_hiddens` ayarlanabilir bir hiper parametredir. Dil modellerini eğitirken, girdiler ve çıktılar aynı kelime dağarcığındandır. Bu nedenle, kelime dağarcığı uzunluğuna eşit olan aynı boyuta sahiptirler.

```
def get_params(vocab_size, num_hiddens, device):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return torch.randn(size=shape, device=device) * 0.01

    # Gizli katman parametreleri
    W_xh = normal((num_inputs, num_hiddens))
    W_hh = normal((num_hiddens, num_hiddens))
    b_h = torch.zeros(num_hiddens, device=device)
    # Çıktı katmanı parametreleri
    W_hq = normal((num_hiddens, num_outputs))
    b_q = torch.zeros(num_outputs, device=device)
    # Gradyanları iliştir
    params = [W_xh, W_hh, b_h, W_hq, b_q]
    for param in params:
        param.requires_grad_(True)
    return params
```

8.5.3 RNN Modeli

Bir RNN modeli tanımlamak için, ilk olarak ilkleme esnasında gizli durumu döndürmek için bir `init_rnn_state` işlevi gerekir. Şekli (grup boyutu, gizli birimlerin sayısı) olan 0 ile doldurulmuş bir tensör döndürür. Çokuzlu (tuple) kullanmak, daha sonraki bölümlerde karşılaşacağımız gibi gizli durumun birden çok değişken içerdiği halleri işlemeyi kolaylaştırır.

```
def init_rnn_state(batch_size, num_hiddens, device):
    return (torch.zeros((batch_size, num_hiddens), device=device), )
```

Aşağıdaki `rnn` işlevi, gizli durumu ve çıktıyı bir zaman adımında nasıl hesaplayacağınızı tanımlar. RNN modelinin `inputs` değişkeninin en dış boyutunda döngü yaptığını, böylece minigrubun gizli durumları `H`'nin her zaman adımında güncellediğini unutmayın. Ayrıca, burada etkinleştirme

fonksiyonu olarak tanh işlevi kullanılır. Section 4.1 içinde açıklandığı gibi, tanh işlevinin ortalama değeri, elemanlar gerçel sayılar üzerinde eşit olarak dağıtıldığında 0'dır.

```
def rnn(inputs, state, params):
    # `inputs`'un şekli: (`num_steps`, `batch_size`, `vocab_size`)
    W_xh, W_hh, b_h, W_hq, b_q = params
    H, = state
    outputs = []
    # `X`'in şekli: (`batch_size`, `vocab_size`)
    for X in inputs:
        H = torch.tanh(torch.mm(X, W_xh) + torch.mm(H, W_hh) + b_h)
        Y = torch.mm(H, W_hq) + b_q
        outputs.append(Y)
    return torch.cat(outputs, dim=0), (H,)
```

Gerekli tüm işlevler tanımlandıktan sonra, bu işlevleri sarmalamak ve sıfırdan uygulanan bir RNN modelinin parametreleri depolamak için bir sınıf oluşturuyoruz.

```
class RNNModelScratch: #@save
    """Sıfırdan uygulanan bir RNN modeli."""
    def __init__(self, vocab_size, num_hiddens, device,
                 get_params, init_state, forward_fn):
        self.vocab_size, self.num_hiddens = vocab_size, num_hiddens
        self.params = get_params(vocab_size, num_hiddens, device)
        self.init_state, self.forward_fn = init_state, forward_fn

    def __call__(self, X, state):
        X = F.one_hot(X.T, self.vocab_size).type(torch.float32)
        return self.forward_fn(X, state, self.params)

    def begin_state(self, batch_size, device):
        return self.init_state(batch_size, self.num_hiddens, device)
```

Çıktıların doğru şekillere sahip olup olmadığını kontrol edelim, örn. gizli durumun boyutunun değişmeden kalmasını sağlamak için.

```
num_hiddens = 512
net = RNNModelScratch(len(vocab), num_hiddens, d2l.try_gpu(), get_params,
                      init_rnn_state, rnn)
state = net.begin_state(X.shape[0], d2l.try_gpu())
Y, new_state = net(X.to(d2l.try_gpu()), state)
Y.shape, len(new_state), new_state[0].shape
```

```
(torch.Size([10, 28]), 1, torch.Size([2, 512]))
```

Çıktı şekli (zaman sayısı $\times$ grup boyutu, kelime dağarcığı uzunluğu) iken, gizli durum şeklinin aynı kaldığını, yani (grup boyutu, gizli birimlerin sayısı) olduğunu görebiliriz.

8.5.4 Tahmin

Önce kullanıcı tarafından sağlanan `prefix`'i (önek), ki birkaç karakter içeren bir dizgidir, takip eden yeni karakterler oluşturmak için tahmin işlevi tanımlayalım. `prefix`'teki bu başlangıç karakterleri arasında döngü yaparken, herhangi bir çıktı oluşturmadan gizli durumu bir sonraki adımına geçirmeye devam ediyoruz. Buna, modelin kendisini güncellediği (örn. gizli durumu güncellediği) ancak tahminlerde bulunmadığı *ısınma* dönemi denir. Isınma döneminden sonra, gizli durum genellikle başlangıçtaki ilkleme değerinden daha iyidir. Artık tahmin edilen karakterleri oluşturup yayarız.

```
def predict_ch8(prefix, num_preds, net, vocab, device):  #@save
    """`prefix`'i takip eden yeni karakterler üret."""
    state = net.begin_state(batch_size=1, device=device)
    outputs = [vocab[prefix[0]]]
    get_input = lambda: torch.tensor([outputs[-1]], device=device).reshape((1, 1))
    for y in prefix[1:]:  # Warm-up period
        _, state = net(get_input(), state)
        outputs.append(vocab[y])
    for _ in range(num_preds):  # Predict `num_preds` steps
        y, state = net(get_input(), state)
        outputs.append(int(y.argmax(dim=1).reshape(1)))
    return ''.join([vocab.idx_to_token[i] for i in outputs])
```

Şimdi `predict_ch8` işlevini test edebiliriz. Öneki `time traveller` olarak belirtiyoruz ve 10 ek karakter üretiyoruz. Ağı eğitmediğimiz göz önüne alındığında, saçma tahminler üretecektir.

```
predict_ch8('time traveller ', 10, net, vocab, d2l.try_gpu())
```

```
'time traveller lfbrpfbehs'
```

8.5.5 Gradyan Kırpma

T uzunluğunda bir dizi için, bir yinelemede bu T zaman adımlarının üzerindeki gradyanları hesaplarız, bu da geri yayma sırasında $\mathcal{O}(T)$ uzunluğunda bir matris çarpımları zincirine neden olur. :numref:`sec_numerical_stability içinde belirtildiği gibi, bu da sayısal kararsızlığa neden olabilir, örneğin T büyük olduğunda gradyanlar patlayabilir veya kaybolabilir. Bu nedenle, RNN modelleri genellikle eğitimi kararlı tutmak için ekstra yardıma ihtiyaç duyar.

Genel olarak, bir eniyileme problemini çözerken, model parametresi için güncelleme adımları atıyoruz, mesela $\mathbf{x}$ vektör formunda, bir minigrup üzerinden negatif gradyan $\mathbf{g}$ yönünde. Örneğin, öğrenme oranı $\eta > 0$ ise, bir yinelemede $\mathbf{x}$'i $\mathbf{x} - \eta\mathbf{g}$ olarak güncelleriz. f amaç fonksiyonunun iyi davrandığını varsayalım, diyelim ki, L sabiti ile *Lipschitz sürekli* olsun. Yani, herhangi bir $\mathbf{x}$ ve $\mathbf{y}$ için:

$$|f(\mathbf{x}) - f(\mathbf{y})| \leq L\|\mathbf{x} - \mathbf{y}\|. \tag{8.5.1}$$

Bu durumda, parametre vektörünü $\eta\mathbf{g}$ ile güncellediğimizi varsayarsak, o zaman

$$|f(\mathbf{x}) - f(\mathbf{x} - \eta\mathbf{g})| \leq L\eta\|\mathbf{g}\|, \tag{8.5.2}$$

demektir ki $L\eta\|\mathbf{g}\|$'den daha büyük bir değişiklik gözlemlemeyeceğiz. Bu hem bir lanet hem de bir nimettir. Lanet tarafında, ilerleme kaydetme hızını sınırlar; oysa nimet tarafında, yanlış yönde hareket edersek işlerin ne ölçüde ters gidebileceğini sınırlar.

Bazen gradyanlar oldukça büyük olabilir ve eniyileme algoritması yakınsamada başarısız olabilir. Öğrenme hızı η'yı azaltarak bunu ele alabiliriz. Ama ya sadece nadiren büyük gradyanlar alırsak? Bu durumda böyle bir yaklaşım tamamen yersiz görünebilir. Popüler bir alternatif, $\mathbf{g}$ gradyanını belirli bir yarıçaptaki bir küreye geri yansıtmaktır, diyelim ki θ için:

$$\mathbf{g} \leftarrow \min\left(1, \frac{\theta}{\|\mathbf{g}\|}\right)\mathbf{g}. \tag{8.5.3}$$

Bunu yaparak, gradyanın normunun θ'yı asla geçmediğini ve güncellenen gradyanın $\mathbf{g}$'nin orijinal yönüyle tamamen hizalandığını biliyoruz. Ayrıca, herhangi bir minigrubun (ve içindeki herhangi bir örneklemin) parametre vektörü üzerinde uygulayabileceği etkiyi sınırlaması gibi arzu edilen yan etkiye sahiptir. Bu, modele belirli bir derecede gürbüzlük kazandırır. Gradyan kırpma gradyan patlaması için hızlı bir düzeltme sağlar. Sorunu tamamen çözmese de, onu hafifletmede kullanılan birçok teknikten biridir.

Aşağıda, sıfırdan uygulanan veya üst düzey API'ler tarafından oluşturulan bir modelin gradyanlarını kırpmak için bir işlev tanımlıyoruz. Ayrıca, tüm model parametrelerinin üzerinden gradyan normunu hesapladığımızı unutmayın.

```
def grad_clipping(net, theta):  #@save
    """Gradyanı kırp."""
    if isinstance(net, nn.Module):
        params = [p for p in net.parameters() if p.requires_grad]
    else:
        params = net.params
    norm = torch.sqrt(sum(torch.sum((p.grad ** 2)) for p in params))
    if norm > theta:
        for param in params:
            param.grad[:] *= theta / norm
```

8.5.6 Eğitim

Modeli eğitmeden önce, modeli bir dönemde eğitmek için bir işlev tanımlayalım. Section 3.6 içindeki model eğitimimizden üç yerde farklılık gösterir:

1. Dizili veriler için farklı örnekleme yöntemleri (rastgele örnekleme ve sıralı bölümleme), gizli durumların ilklenmesinde farklılıklara neden olacaktır.
2. Model parametrelerini güncellemeden önce gradyanları kırpıyoruz. Bu, eğitim süreci sırasında bir noktada gradyanlar patladığında bile modelin ıraksamamasını sağlar.
3. Modeli değerlendirmek için şaşkınlığı kullanıyoruz. Section 8.4.4 içinde tartışıldığı gibi, bu farklı uzunluktaki dizilerin karşılaştırılabilir olmasını sağlar.

Özellikle, sıralı bölümleme kullanıldığında, gizli durumu yalnızca her dönemin başında ilkleriz. Sonraki minigruptaki i. altdizi örneği şimdiki i. altdizi örneğine bitişik olduğundan, şimdiki minigrup sonundaki gizli durum sonraki minigrup başında gizli durumu ilklemek için kullanılır. Bu şekilde, gizli durumda saklanan dizinin tarihsel bilgileri bir dönem içinde bitişik altdizilere aktarılabilir. Ancak, herhangi bir noktada gizli durumun hesaplanması, aynı dönemdeki tüm önceki minigruplara bağlıdır, ki bu da gradyan hesaplamasını zorlaştırır. Hesaplama maliyetini azaltmak için, herhangi bir minigrup işleminden önce gradyanı koparıp ayırırız, böylece gizli durumun gradyan hesaplaması her zaman minigrup içindeki zaman adımlarıyla sınırlı kalır.

Rastgele örneklemeyi kullanırken, her örnek rastgele bir konumla örneklendiğinden, her yineleme için gizli durumu yeniden ilklememiz gerekir. Section 3.6 içindeki `train_epoch_ch3`

işlevi gibi, `updater` da model parametrelerini güncellemek için genel bir işlevdir. Sıfırdan uygulanan `d2l.sgd` işlevi veya derin bir öğrenme çerçevesindeki yerleşik eniyileme işlevi olabilir.

```
#@save
def train_epoch_ch8(net, train_iter, loss, updater, device, use_random_iter):
    """Bir modeli bir dönem içinde eğitin (Bölüm 8'de tanımlanmıştır)."""
    state, timer = None, d2l.Timer()
    metric = d2l.Accumulator(2)  # Eğitim kaybı toplamı, andıç sayısı
    for X, Y in train_iter:
        if state is None or use_random_iter:
            # İlk yineleme olduğunda veya rastgele örnekleme kullanıldığında
            # `state`'i (durumu) ilklet
            state = net.begin_state(batch_size=X.shape[0], device=device)
        else:
            if isinstance(net, nn.Module) and not isinstance(state, tuple):
                # `state` (durum) `nn.GRU` için bir tensördür
                state.detach_()
            else:
                # `state`, `nn.LSTM` ve özel sıfırdan uygulamamız için
                # bir tensör grubudur
                for s in state:
                    s.detach_()
        y = Y.T.reshape(-1)
        X, y = X.to(device), y.to(device)
        y_hat, state = net(X, state)
        l = loss(y_hat, y.long()).mean()
        if isinstance(updater, torch.optim.Optimizer):
            updater.zero_grad()
            l.backward()
            grad_clipping(net, 1)
            updater.step()
        else:
            l.backward()
            grad_clipping(net, 1)
            # `mean` işlevi çağrıldığından dolayı
            updater(batch_size=1)
        metric.add(l * y.numel(), y.numel())
    return math.exp(metric[0] / metric[1]), metric[1] / timer.stop()
```

Eğitim fonksiyonu sıfırdan veya üst düzey API'ler kullanılarak uygulanan bir RNN modelini destekler.

```
#@save
def train_ch8(net, train_iter, vocab, lr, num_epochs, device,
              use_random_iter=False):
    """Bir modeli eğitin (Bölüm 8'de tanımlanmıştır)."""
    loss = nn.CrossEntropyLoss()
    animator = d2l.Animator(xlabel='epoch', ylabel='perplexity',
                            legend=['train'], xlim=[10, num_epochs])
    # İlkleyin
    if isinstance(net, nn.Module):
        updater = torch.optim.SGD(net.parameters(), lr)
    else:
        updater = lambda batch_size: d2l.sgd(net.params, lr, batch_size)
    predict = lambda prefix: predict_ch8(prefix, 50, net, vocab, device)
```

(continues on next page)

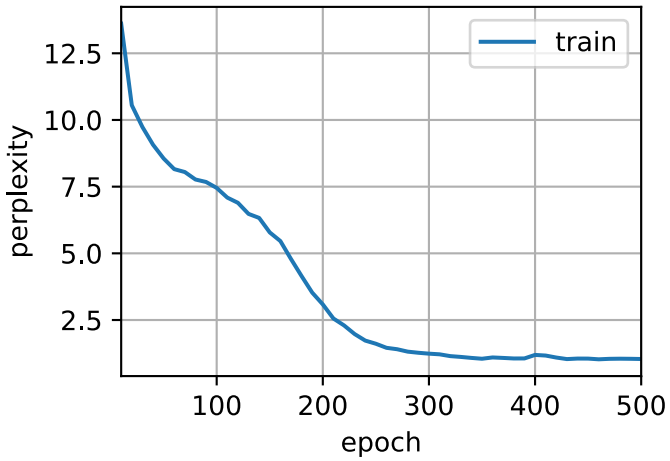
(continued from previous page)

```
    # Eğit ve tahminle
    for epoch in range(num_epochs):
        ppl, speed = train_epoch_ch8(
            net, train_iter, loss, updater, device, use_random_iter)
        if (epoch + 1) % 10 == 0:
            print(predict('time traveller'))
            animator.add(epoch + 1, [ppl])
    print(f'perplexity {ppl:.1f}, {speed:.1f} tokens/sec on {str(device)}')
    print(predict('time traveller'))
    print(predict('traveller'))
```

Şimdi RNN modelini eğitebiliriz. Veri kümesinde yalnızca 10000 andıç kullandığımızdan, modelin daha iyi yakınsaması için daha fazla döneme ihtiyacı var.

```
num_epochs, lr = 500, 1
train_ch8(net, train_iter, vocab, lr, num_epochs, d2l.try_gpu())
```

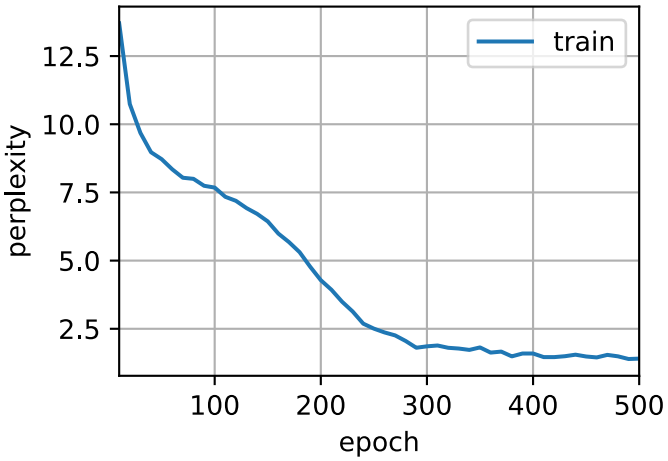
```
perplexity 1.0, 68806.3 tokens/sec on cuda:0
time travelleryou can show black is white by argument said filby
travelleryou can show black is white by argument said filby
```



Son olarak, rastgele örnekleme yöntemini kullanmanın sonuçlarını kontrol edelim.

```
net = RNNModelScratch(len(vocab), num_hiddens, d2l.try_gpu(), get_params,
                      init_rnn_state, rnn)
train_ch8(net, train_iter, vocab, lr, num_epochs, d2l.try_gpu(),
          use_random_iter=True)
```

```
perplexity 1.4, 52954.3 tokens/sec on cuda:0
time travellerit s against reason said filbywhar gethematical pi
travellerit s against reason said filbywhar gethematical pi
```



Yukarıdaki RNN modelini sıfırdan uygulamak öğretici olsa da, pek de uygun değildir. Bir sonraki bölümde, RNN modelinin nasıl geliştirileceğini göreceğiz, örneğin nasıl daha kolay uygulanmasını ve daha hızlı çalışmasını sağlarız.

8.5.7 Özet

- Kullanıcı tarafından sağlanan metin önekini takip eden metin üretmek için RNN tabanlı karakter düzeyinde bir dil modeli eğitebiliriz.
- Basit bir RNN dil modeli girdi kodlama, RNN modelleme ve çıktı üretme içerir.
- RNN modellerinin eğitim için durum ilklenmesi gerekir, ancak rasgele örnekleme ve sıralı bölümleme farklı yollar kullanır.
- Sıralı bölümleme kullanırken, hesaplama maliyetini azaltmak için gradyanı koparıp ayırmamız gerekir.
- Isınma süresi, herhangi bir tahmin yapmadan önce bir modelin kendisini güncellemesine (örneğin, ilkleme değerinden daha iyi bir gizli durum elde etmesine) olanak tanır.
- Gradyan kırpma gradyan patlamasını önler, ancak kaybolan gradyanları düzeltemez.

8.5.8 Alıştırmalar

1. Bire bir kodlamanın, her nesne için farklı bir gömme seçmeye eşdeğer olduğunu gösterin.
2. Şaşkınlığı iyileştirmek için hiper parametreleri (örn. dönemlerin sayısı, gizli birimlerin sayısı, bir minigruptaki zaman adımlarının sayısı ve öğrenme oranı) ayarlayın.
    - Ne kadar aşağıya düşebilirsiniz?
    - Bire bir kodlamayı öğrenilebilir gömmelerle değiştirin. Bu daha iyi bir performansa yol açar mı?
    - H. G. Wells'in, örn. *Dünyalar Savaşı*[107] gibi, diğer kitaplarıyla ne kadar iyi çalışacaktır?
3. En olası sonraki karakteri seçmek yerine örnekleme kullanarak tahmin işlevini değiştirin.

[107] http://www.gutenberg.org/ebooks/36

- Ne olur?
- Modeli, örneğin $\alpha > 1$ için $q(x_t \mid x_{t-1}, \ldots, x_1) \propto P(x_t \mid x_{t-1}, \ldots, x_1)^\alpha$'ten örnekleme yaparak daha olası çıktılara doğru yanlı yapın.

4. Gradyan kırpmadan bu bölümdeki kodu çalıştırın. Ne olur?
5. Sıralı bölümlemeyi gizli durumları hesaplama çizgesinden ayırmayacak şekilde değiştirin. Çalışma süresi değişiyor mu? Şaşkınlığa ne olur?
6. Bu bölümde kullanılan etkinleştirme işlevini ReLU ile değiştirin ve bu bölümdeki deneyleri tekrarlayın. Hala gradyan kırpmaya ihtiyacımız var mı? Neden?

Tartışmalar[108]

8.6 Yinelemeli Sinir Ağlarının Kısa Uygulaması

Section 8.5 RNN'lerin nasıl uygulandığını görmek için öğretici olsa da, bu uygun veya hızlı değildir. Bu bölümde, derin öğrenme çerçevesinin üst düzey API'leri tarafından sağlanan işlevleri kullanarak aynı dil modelinin nasıl daha verimli bir şekilde uygulanacağı gösterilecektir. Daha önce olduğu gibi zaman makinesi veri kümesini okuyarak başlıyoruz.

```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

8.6.1 Modelin Tanımlanması

Üst seviye API'ler, yinelemeli sinir ağlarının uygulamalarını sağlar. Yinelemeli sinir ağı tabakası `rnn_layer`'i tek bir gizli katman ve 256 gizli birimle inşa ediyoruz. Aslında, birden fazla katmana sahip olmanın ne anlama geldiğini henüz tartışmadık; onu Section 9.3 içinde göreceğiz. Şimdilik, birden çok katmanın, bir sonraki RNN katmanı için girdi olarak kullanılan bir RNN katmanının çıktısı anlamına geldiğini söylemek yeterlidir.

```
num_hiddens = 256
rnn_layer = nn.RNN(len(vocab), num_hiddens)
```

Şekli (gizli katman sayısı, grup boyutu, gizli birim sayısı) olan bir tensörü gizli durumu ilklemek için kullanırız.

```
state = torch.zeros((1, batch_size, num_hiddens))
state.shape
```

```
torch.Size([1, 32, 256])
```

[108] https://discuss.d2l.ai/t/486

Gizli bir durum ve bir girdi ile, çıktıyı güncellenmiş gizli durumla hesaplayabiliriz. `rnn_layer`'in "çıktı"'nın (`Y`) çıktı katmanlarının hesaplanmasını *içermediği* vurgulanmalıdır: *Her bir* zaman adımındaki gizli durumu ifade eder ve sonraki çıktı katmanına girdi olarak kullanılabilir.

```
X = torch.rand(size=(num_steps, batch_size, len(vocab)))
Y, state_new = rnn_layer(X, state)
Y.shape, state_new.shape
```

```
(torch.Size([35, 32, 256]), torch.Size([1, 32, 256]))
```

Section 8.5 içindekine benzer şekilde, tam bir RNN modeli için bir `RNNModel` sınıfı tanımlarız. `rnn_layer`'in yalnızca gizli yinelemeli katmanları içerdiğini unutmayın, ayrı bir çıktı katmanı oluşturmamız gerekir.

```
#@save
class RNNModel(nn.Module):
    """RNN modeli."""
    def __init__(self, rnn_layer, vocab_size, **kwargs):
        super(RNNModel, self).__init__(**kwargs)
        self.rnn = rnn_layer
        self.vocab_size = vocab_size
        self.num_hiddens = self.rnn.hidden_size
        # RNN çift yönlü ise (daha sonra tanıtılacaktır),
        # `num_directions` 2 olmalı, yoksa 1 olmalıdır.
        if not self.rnn.bidirectional:
            self.num_directions = 1
            self.linear = nn.Linear(self.num_hiddens, self.vocab_size)
        else:
            self.num_directions = 2
            self.linear = nn.Linear(self.num_hiddens * 2, self.vocab_size)

    def forward(self, inputs, state):
        X = F.one_hot(inputs.T.long(), self.vocab_size)
        X = X.to(torch.float32)
        Y, state = self.rnn(X, state)
        # Tam bağlı katman önce `Y`'nin şeklini
        # (`num_steps` * `batch_size`, `num_hiddens`) olarak değiştirir.
        # Çıktı (`num_steps` * `batch_size`, `vocab_size`) şeklindedir.
        output = self.linear(Y.reshape((-1, Y.shape[-1])))
        return output, state

    def begin_state(self, device, batch_size=1):
        if not isinstance(self.rnn, nn.LSTM):
            # `nn.GRU` gizli durum olarak bir tensör alır
            return  torch.zeros((self.num_directions * self.rnn.num_layers,
                                 batch_size, self.num_hiddens),
                                device=device)
        else:
            # `nn.LSTM` bir gizli durum çokuzu alır
            return (torch.zeros((
                self.num_directions * self.rnn.num_layers,
                batch_size, self.num_hiddens), device=device),
                    torch.zeros((
                        self.num_directions * self.rnn.num_layers,
                        batch_size, self.num_hiddens), device=device))
```

8.6.2 Eğitim ve Tahmin

Modeli eğitmeden önce, rastgele ağırlıklara sahip bir modelle bir tahmin yapalım.

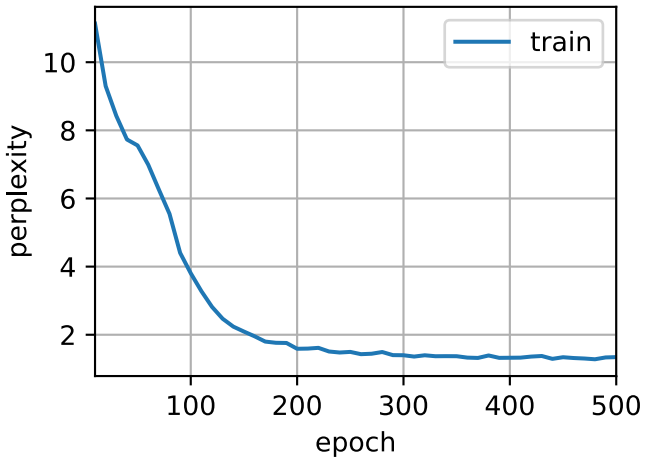
```
device = d2l.try_gpu()
net = RNNModel(rnn_layer, vocab_size=len(vocab))
net = net.to(device)
d2l.predict_ch8('time traveller', 10, net, vocab, device)
```

```
'time travellerxxxxxxxxxx'
```

Oldukça açık, bu model hiç çalışmıyor. Ardından, Section 8.5 içinde tanımlanan aynı hiper parametrelerle `train_ch8`'i çağırdık ve modelimizi üst düzey API'lerle eğitiyoruz.

```
num_epochs, lr = 500, 1
d2l.train_ch8(net, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.3, 290440.5 tokens/sec on cuda:0
time traveller aptereave but lothe rak alyscthat arsthes than a
travellerithe fore wescantwo snoweicalsiothroughi new yores
```



Son bölümle karşılaştırıldığında, bu model, kodun derin öğrenme çerçevesinin üst düzey API'leriyle daha iyi hale getirilmesinden dolayı daha kısa bir süre içinde olsa da, benzer bir şaşkınlığa ulaşmaktadır.

8.6.3 Özet

- Derin öğrenme çerçevesinin üst düzey API'leri RNN katmanının bir uygulanmasını sağlar.
- Üst düzey API'lerin RNN katmanı, çıktının çıktı katmanı hesaplamasını içermediği bir çıktı ve güncellenmiş bir gizli durum döndürür.
- Üst düzey API'lerin kullanılması, sıfırdan uygulama yaratmaktan daha hızlı RNN eğitimine yol açar.

8.6.4 Alıştırmalar

1. RNN modelini üst düzey API'leri kullanarak aşırı eğitebilir misiniz?
2. RNN modelinde gizli katman sayısını artırırsanız ne olur? Modelin çalışmasını sağlayabilecek misiniz?
3. Bir RNN kullanarak Section 8.1 içindeki özbağlanımlı modelini uygulayın.

Tartışmalar[109]

8.7 Zamanda Geri Yayma

Şimdiye kadar defalarca *patlayan gradyanlar, kaybolan gradyanlar,* ve RNN'ler için *gradyan ayırma* ihtiyacı gibi şeyler ima ettik. Örneğin, Section 8.5 içinde dizi üzerinde `detach` işlevini çağırdık. Hızlı bir model inşa edebilmek ve nasıl çalıştığını görmek amacıyla, bunların hiçbiri gerçekten tam olarak açıklanmadı. Bu bölümde, dizi modelleri için geri yaymanın ayrıntılarını ve matematiğin neden (ve nasıl) çalıştığını biraz daha derinlemesine inceleyeceğiz.

RNN'leri ilk uyguladığımızda gradyan patlamasının bazı etkileriyle karşılaştık (Section 8.5). Özellikle, alıştırmaları çözdüyseniz, doğru yakınsamayı sağlamak için gradyan kırpmanın hayati önem taşıdığını görürsünüz. Bu sorunun daha iyi anlaşılmasını sağlamak için, bu bölümde gradyanların dizi modellerinde nasıl hesaplandığını incelenecektir. Nasıl çalıştığına dair kavramsal olarak yeni bir şey olmadığını unutmayın. Sonuçta, biz hala sadece gradyanları hesaplamak için zincir kuralını uyguluyoruz. Bununla birlikte, geri yayma (Section 4.7) tekrar gözden geçirmeye değerdir.

Section 4.7 içinde MLP'lerde ileri ve geri yaymayı ve hesaplama çizgelerini tanımladık. Bir RNN'de ileriye yayma nispeten basittir. *Zamanda geri yayma* aslında RNN'lerde geri yaymanın belirli bir uygulamasıdır (Werbos, 1990). Model değişkenleri ve parametreleri arasındaki bağımlılıkları elde etmek için bir RNN'nin hesaplama çizgesini bir kerede bir adım genişletmemizi gerektirir. Ardından, zincir kuralına bağlı olarak, gradyanları hesaplamak ve depolamak için geri yayma uygularız. Diziler oldukça uzun olabileceğinden, bağımlılık oldukça uzun olabilir. Örneğin, 1000 karakterlik bir dizi için, ilk andıç nihai konumdaki andıç üzerinde potansiyel olarak önemli bir etkiye sahip olabilir. Bu gerçekten hesaplamalı olarak mümkün değildir (çok uzun sürer ve çok fazla bellek gerektirir) ve bizim bu çok zor gradyana ulaşmadan önce 1000'den fazla matrisi çarpamıza gerek duyar. Bu, hesaplamalı ve istatistiksel belirsizliklerle dolu bir süreçtir. Aşağıda neler olduğunu ve bunu pratikte nasıl ele alacağımızı aydınlatacağız.

8.7.1 RNN'lerde Gradyanların Çözümlemesi

Bir RNN'nin nasıl çalıştığını anlatan basitleştirilmiş bir modelle başlıyoruz. Bu model, gizli durumun özellikleri ve nasıl güncellendiği hakkındaki ayrıntıları görmezden gelir. Buradaki matematiksel gösterim, skalerleri, vektörleri ve matrisleri eskiden olduğu gibi açıkça ayırt etmez. Bu ayrıntılar, analiz için önemsizdir ve öbür türlü yalnızca bu alt bölümdeki gösterimi karıştırmaya hizmet edecekti.

Bu basitleştirilmiş modelde, h_t'yi gizli durum, x_t'yi girdi ve o_t'yi t'deki çıktı olarak gösteriyoruz. Section 8.4.2 içindeki tartışmalarımızı hatırlayın, girdi ve gizli durum, gizli katmandaki bir ağırlık değişkeni ile çarpılacak şekilde bitiştirilebilir. Böylece, sırasıyla gizli katmanın ve çıktı katmanının ağırlıklarını belirtmek için w_h ve w_o'yi kullanırız. Sonuç olarak, her zaman adımındaki

[109] https://discuss.d2l.ai/t/1053

gizli durumlar ve çıktılar aşağıdaki gibi açıklanabilir:

$$\begin{aligned} h_t &= f(x_t, h_{t-1}, w_h), \\ o_t &= g(h_t, w_o), \end{aligned} \tag{8.7.1}$$

burada f ve g, sırasıyla gizli katmanının ve çıktı katmanının dönüşümleridir. Bu nedenle, yinelemeli hesaplama yoluyla birbirine bağlı $\{\ldots, (x_{t-1}, h_{t-1}, o_{t-1}), (x_t, h_t, o_t), \ldots\}$ değerler zincirine sahibiz. İleri yayma oldukça basittir. İhtiyacımız olan tek şey (x_t, h_t, o_t) üçlüleri arasında bir seferde bir zaman adımı atarak döngü yapmaktır. Çıktı o_t ve istenen etiket y_t arasındaki tutarsızlık daha sonra tüm T zaman adımlarında amaç işlevi tarafından değerlendirilir

$$L(x_1, \ldots, x_T, y_1, \ldots, y_T, w_h, w_o) = \frac{1}{T}\sum_{t=1}^{T} l(y_t, o_t). \tag{8.7.2}$$

Geri yayma için, özellikle L amaç fonksiyonun w_h parametreleri ile ilgili olarak gradyanları hesaplarken işler biraz daha zorlaşır. Belirleyici olmak gerekirse, zincir kuralına göre,

$$\begin{aligned} \frac{\partial L}{\partial w_h} &= \frac{1}{T}\sum_{t=1}^{T} \frac{\partial l(y_t, o_t)}{\partial w_h} \\ &= \frac{1}{T}\sum_{t=1}^{T} \frac{\partial l(y_t, o_t)}{\partial o_t} \frac{\partial g(h_t, w_o)}{\partial h_t} \frac{\partial h_t}{\partial w_h}. \end{aligned} \tag{8.7.3}$$

Çarpımın (8.7.3) içindeki birinci ve ikinci faktörlerinin hesaplanması kolaydır. h_t'da w_h parametresinin etkisini yeniden hesaplamamız gerektiğinden, üçüncü faktör $\partial h_t/\partial w_h$'de işler zorlaşır. (8.7.1) içindeki yinelemeli hesaplamaya göre, h_t h_{t-1} ve w_h'ye bağlıdır, burada h_{t-1}'in hesaplanması da w_h'ye bağlıdır. Böylece, zincir kuralı aşağıdaki çıkarsamaya varır:

$$\frac{\partial h_t}{\partial w_h} = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h} + \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial w_h}. \tag{8.7.4}$$

Yukarıdaki gradyanı türetmek için, $t = 1, 2, \ldots$ için $a_0 = 0$ ve $a_t = b_t + c_t a_{t-1}$ koşullarını sağlayan $\{a_t\}, \{b_t\}, \{c_t\}$ üç dizisine sahip olduğumuzu varsayalım. Sonra $t \geq 1$ için, aşağıdaki ifadeyi göstermek kolaydır:

$$a_t = b_t + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} c_j \right) b_i. \tag{8.7.5}$$

a_t, b_t ve c_t'nin yerlerine aşağıdaki ifadeleri koyarsak

$$\begin{aligned} a_t &= \frac{\partial h_t}{\partial w_h}, \\ b_t &= \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h}, \\ c_t &= \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial h_{t-1}}, \end{aligned} \tag{8.7.6}$$

(8.7.4) içindeki gradyan hesaplama $a_t = b_t + c_t a_{t-1}$'yı sağlar. Böylece, (8.7.5) içindeki, (8.7.4) yinelemeli hesaplamasını kaldırabiliriz.

$$\frac{\partial h_t}{\partial w_h} = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h} + \sum_{i=1}^{t-1} \left(\prod_{j=i+1}^{t} \frac{\partial f(x_j, h_{j-1}, w_h)}{\partial h_{j-1}} \right) \frac{\partial f(x_i, h_{i-1}, w_h)}{\partial w_h}. \tag{8.7.7}$$

$\partial h_t/\partial w_h$'i yinelemeli olarak hesaplamak için zincir kuralını kullanabilsek de, t büyük olduğunda bu zincir çok uzun sürebilir. Bu sorunla başa çıkmak için bazı stratejileri tartışalım.

Tam Hesaplama

Açıkçası, (8.7.7) içindeki tam toplamı hesaplayabiliriz. Fakat, bu çok yavaştır ve gradyanlar patlayabilir, çünkü ilkleme koşullarındaki narin değişiklikler sonucu potansiyel olarak çok etkileyebilir. Yani, ilk koşullardaki minimum değişikliklerin sonuçta orantısız değişikliklere yol açtığı kelebek etkisine benzer şeyler görebiliriz. Bu aslında tahmin etmek istediğimiz model açısından oldukça istenmeyen bir durumdur. Sonuçta, iyi genelleyen gürbüz tahminciler arıyoruz. Bu nedenle bu strateji pratikte neredeyse hiç kullanılmaz.

Zaman Adımlarını Kesme

Alternatif olarak, τ adımdan sonra (8.7.7) içindeki toplamı kesebiliriz. Bu aslında şimdiye kadar tartıştığımız şey, örneğin Section 8.5 içindeki gradyanları ayırdığımız zaman gibi. Bu, toplamı $\partial h_{t-\tau}/\partial w_h$'de sonlandırarak, gerçek gradyanın *yaklaşık değerine* götürür. Pratikte bu oldukça iyi çalışır. Genellikle zaman boyunca kesilmiş geri yayma olarak adlandırılır (Jaeger, 2002). Bunun sonuçlarından biri, modelin uzun vadeli sonuçlardan ziyade kısa vadeli etkilere odaklanmasıdır. Bu aslında *arzu edilendir*, çünkü tahminleri daha basit ve daha kararlı modellere yöneltir.

Rastgele Kesme

Son olarak, $\partial h_t/\partial w_h$'yi, beklentiye göre doğru olan ancak diziyi kesen rastgele bir değişkenle değiştirebiliriz. Bu, önceden tanımlanmış $0 \leq \pi_t \leq 1$ olan bir ξ_t dizisi kullanılarak elde edilir, burada $P(\xi_t = 0) = 1 - \pi_t$ ve $P(\xi_t = \pi_t^{-1}) = \pi_t$, dolayısıyla $E[\xi_t] = 1$'dir. Bunu (8.7.4) içindeki $\partial h_t/\partial w_h$'i değiştirmek için kullanırız.

$$z_t = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h} + \xi_t \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial w_h}. \tag{8.7.8}$$

Bu ξ_t'nin tanımından gelir; $E[z_t] = \partial h_t/\partial w_h$. Ne zaman $\xi_t = 0$ olursa, yinelemeli hesaplama o t zaman adımında sona erer. Bu, uzun dizilerin nadir ancak uygun şekilde fazla ağırlıklandırılmış olduğu çeşitli uzunluklardaki dizilerin ağırlıklı bir toplamına yol açar. Bu fikir Tallec ve Ollivier (Tallec and Ollivier, 2017) tarafından önerilmiştir.

Stratejilerin Karşılaştırılması

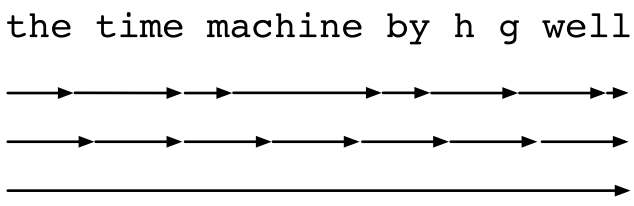


Fig. 8.7.1: RNN'lerde gradyanları hesaplama stratejilerinin karşılaştırılması. Yukarıdan aşağıya: Rastgele kesme, düzenli kesme ve tam hesaplama.

Fig. 8.7.1, RNN'ler için zamanda geri yayma kullanan *Zaman Makinesi* kitabının üç stratejideki ilk birkaç karakterini analiz ederek göstermektedir:

- İlk satır, metni farklı uzunluklardaki bölümlere ayıran rasgele kesmedir.

- İkinci satır, metni aynı uzunlukta altdizilere kıran düzenli kesimdir. Bu RNN deneylerinde yaptığımız şeydir.
- Üçüncü satır, hesaplaması mümkün olmayan bir ifadeye yol açan zamanda tam geri yaymadır.

Ne yazık ki, teoride çekici iken, rasgele kesme, büyük olasılıkla bir dizi faktöre bağlı olarak düzenli kesmeden çok daha iyi çalışmaz. Birincisi, bir gözlemin geçmişe birkaç geri yayma adımından sonraki etkisi, pratikteki bağımlılıkları yakalamak için oldukça yeterlidir. İkincisi, artan varyans, gradyanın daha fazla adımla daha doğru olduğu gerçeğine karşı yarışır. Üçüncüsü, aslında sadece kısa bir etkileşim aralığına sahip modeller istiyoruz. Bu nedenle, zamanda düzenli kesilmiş geri yayma, arzu edilebilecek hafif bir düzenlileştirici etkiye sahiptir.

8.7.2 Ayrıntılı Zamanda Geri Yayma

Genel prensibi tartıştıktan sonra, zamanda geriye yaymayı ayrıntılı olarak ele alalım. Section 8.7.1 içindeki analizden farklı olarak, aşağıda, amaç fonksiyonun gradyanlarının tüm ayrıştırılmış model parametrelerine göre nasıl hesaplanacağını göstereceğiz. İşleri basit tutmak için, gizli katmandaki etkinleştirme işlevi olarak birim eşlemelerini kullanan ek girdi parametresiz bir RNN'yi göz önünde bulunduruyoruz ($\phi(x) = x$). Zaman adımı t için, tek örnek girdinin ve etiketin sırasıyla $\mathbf{x}_t \in \mathbb{R}^d$ ve y_t olduğunu varsayalım. Gizli durum $\mathbf{h}_t \in \mathbb{R}^h$ ve çıktı $\mathbf{o}_t \in \mathbb{R}^q$ aşağıdaki gibi hesaplanır:

$$\begin{aligned}\mathbf{h}_t &= \mathbf{W}_{hx}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1},\\ \mathbf{o}_t &= \mathbf{W}_{qh}\mathbf{h}_t,\end{aligned} \tag{8.7.9}$$

burada $\mathbf{W}_{hx} \in \mathbb{R}^{h\times d}$, $\mathbf{W}_{hh} \in \mathbb{R}^{h\times h}$ ve $\mathbf{W}_{qh} \in \mathbb{R}^{q\times h}$ ağırlık parametreleridir. $l(\mathbf{o}_t, y_t)$ ile belirtilen t zaman adımındaki kayıp olsun. Bizim amaç fonksiyonumuz, yani dizinin başından T zaman adımı üzerinden kayıp şöyle hesaplanır:

$$L = \frac{1}{T}\sum_{t=1}^{T} l(\mathbf{o}_t, y_t). \tag{8.7.10}$$

RNN'nin hesaplanması sırasında model değişkenleri ve parametreleri arasındaki bağımlılıkları görselleştirmek için, model için Fig. 8.7.2 içinde gösterildiği gibi bir hesaplama çizgesi çizebiliriz. Örneğin, 3. zaman adımındaki, $\mathbf{h}_3$ gizli durumlarının hesaplanması model parametreleri $\mathbf{W}_{hx}$ ve $\mathbf{W}_{hh}$'ye, son zaman adımındaki gizli durum $\mathbf{h}_2$'ye ve şimdiki zaman adımının girdisi $\mathbf{x}_3$'e bağlıdır.

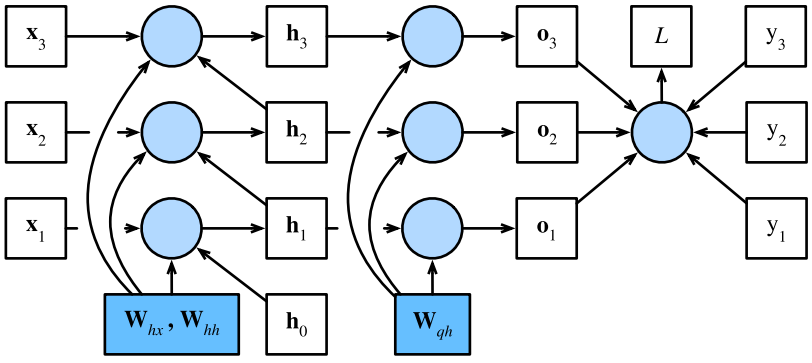


Fig. 8.7.2: Üç zaman adımlı bir RNN modeli için bağımlılıkları gösteren hesaplamalı çizge. Kutular değişkenleri (gölgeli olmayan) veya parametreleri (gölgeli) ve daireler işlemleri temsil eder.

Az önce belirtildiği gibi, Fig. 8.7.2 içindeki model parametreleri $\mathbf{W}_{hx}$, $\mathbf{W}_{hh}$ ve $\mathbf{W}_{qh}$'dır. Genel olarak, bu modelin eğitimi $\partial L/\partial \mathbf{W}_{hx}$, $\partial L/\partial \mathbf{W}_{hh}$ ve $\partial L/\partial \mathbf{W}_{qh}$ parametrelerine göre gradyan

hesaplama gerektirir. Fig. 8.7.2 içindeki bağımlılıklara göre, sırayla gradyanları hesaplamak ve depolamak için okların ters yönünde ilerleyebiliriz. Zincir kuralında farklı şekillerdeki matrislerin, vektörlerin ve skalerlerin çarpımını esnek bir şekilde ifade etmek için Section 4.7 içinde açıklandığı gibi prod işlemini kullanmaya devam ediyoruz.

Her şeyden önce, amaç işlevinin türevini herhangi bir t zaman adımındaki model çıktısına göre almak oldukça basittir:

$$\frac{\partial L}{\partial \mathbf{o}_t} = \frac{\partial l(\mathbf{o}_t, y_t)}{T \cdot \partial \mathbf{o}_t} \in \mathbb{R}^q. \tag{8.7.11}$$

Şimdi, amaç fonksiyonun gradyanını çıktı katmanındaki $\mathbf{W}_{qh}$ parametresine göre hesaplayabiliriz: $\partial L/\partial \mathbf{W}_{qh} \in \mathbb{R}^{q \times h}$. Fig. 8.7.2 temel alınarak amaç fonksiyonu L, $\mathbf{o}_1, \ldots, \mathbf{o}_T$ üzerinden $\mathbf{W}_{qh}$'e bağlıdır. Zincir kuralını kullanırsak şu sonuca ulaşırız,

$$\frac{\partial L}{\partial \mathbf{W}_{qh}} = \sum_{t=1}^{T} \text{prod}\left(\frac{\partial L}{\partial \mathbf{o}_t}, \frac{\partial \mathbf{o}_t}{\partial \mathbf{W}_{qh}}\right) = \sum_{t=1}^{T} \frac{\partial L}{\partial \mathbf{o}_t} \mathbf{h}_t^\top, \tag{8.7.12}$$

burada $\partial L/\partial \mathbf{o}_t$ (8.7.11) içindeki gibi hesaplanır.

Daha sonra, Fig. 8.7.2 içinde gösterildiği gibi, T son zaman adımındaki amaç işlevi L gizli durum $\mathbf{h}_T$'ye yalnızca $\mathbf{o}_T$ üzerinden bağlıdır. Bu nedenle, zincir kuralını kullanarak $\partial L/\partial \mathbf{h}_T \in \mathbb{R}^h$'i kolayca bulabiliriz:

$$\frac{\partial L}{\partial \mathbf{h}_T} = \text{prod}\left(\frac{\partial L}{\partial \mathbf{o}_T}, \frac{\partial \mathbf{o}_T}{\partial \mathbf{h}_T}\right) = \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_T}. \tag{8.7.13}$$

$t < T$ adımı için daha karmaşık hale gelir, burada L amaç işlevi $\mathbf{h}_t$'ye $\mathbf{h}_{t+1}$ ve $\mathbf{o}_t$ üzerinden bağlıdır. Zincir kuralına göre, herhangi bir $t < T$ zamanında, gizli durumunun gradyanı, $\partial L/\partial \mathbf{h}_t \in \mathbb{R}^h$, yinelemeli hesaplanabilir:

$$\frac{\partial L}{\partial \mathbf{h}_t} = \text{prod}\left(\frac{\partial L}{\partial \mathbf{h}_{t+1}}, \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t}\right) + \text{prod}\left(\frac{\partial L}{\partial \mathbf{o}_t}, \frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t}\right) = \mathbf{W}_{hh}^\top \frac{\partial L}{\partial \mathbf{h}_{t+1}} + \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_t}. \tag{8.7.14}$$

Analiz için, herhangi bir zaman adım $1 \leq t \leq T$ için yinelemeli hesaplamayı genişletirsek, şu ifadeye ulaşırız:

$$\frac{\partial L}{\partial \mathbf{h}_t} = \sum_{i=t}^{T} \left(\mathbf{W}_{hh}^\top\right)^{T-i} \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_{T+t-i}}. \tag{8.7.15}$$

(8.7.15) denkleminden bu basit doğrusal örneğin uzun dizi modellerinin bazı temel problemlerini zaten sergilediğini görebiliyoruz: $\mathbf{W}_{hh}^\top$'nın potansiyel olarak çok büyük kuvvetlerini içerir. İçinde, 1'den küçük özdeğerler kaybolur ve 1'den büyük özdeğerler ıraksar. Bu sayısal olarak kararsızdır, bu da kendini kaybolan ve patlayan gradyanlar şeklinde gösterir. Bunu ele almanın bir yolu, Section 8.7.1 içinde tartışıldığı gibi, zaman adımlarını hesaplama açısından uygun bir boyutta kesmektir. Pratikte, bu kesme, belirli bir sayıda zaman adımından sonra gradyanı koparıp ayırarak gerçekleştirilir. Daha sonra uzun ömürlü kısa-dönem belleği gibi daha gelişmiş dizi modellerinin bunu daha da hafifletebileceğini göreceğiz.

Son olarak Fig. 8.7.2, L amaç fonksiyonunun gizli katmandaki $\mathbf{W}_{hx}$ ve $\mathbf{W}_{hh}$ model parametrelerine $\mathbf{h}_1, \ldots, \mathbf{h}_T$ vasıtasıyla bağlı olduğunu gösterir. Bu tür parametrelerin $\partial L/\partial \mathbf{W}_{hx} \in \mathbb{R}^{h \times d}$ ve $\partial L/\partial \mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$'ye göre gradyanları hesaplamak için, zincir kuralını uygularız:

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{W}_{hx}} &= \sum_{t=1}^{T} \text{prod}\left(\frac{\partial L}{\partial \mathbf{h}_t}, \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hx}}\right) = \sum_{t=1}^{T} \frac{\partial L}{\partial \mathbf{h}_t} \mathbf{x}_t^\top,\\ \frac{\partial L}{\partial \mathbf{W}_{hh}} &= \sum_{t=1}^{T} \text{prod}\left(\frac{\partial L}{\partial \mathbf{h}_t}, \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hh}}\right) = \sum_{t=1}^{T} \frac{\partial L}{\partial \mathbf{h}_t} \mathbf{h}_{t-1}^\top,\end{aligned} \tag{8.7.16}$$

burada (8.7.13) ve (8.7.14) ile yinelemeli hesaplanan $\partial L/\partial \mathbf{h}_t$ sayısal kararlılığı etkileyen anahtar değerdir.

Zamanda geri yayma, RNN'lerde geri yayma uygulanması olduğundan, Section 4.7 içinde açıkladığımız gibi, RNN'leri eğitmek zamanda geri yayma ile ileriye doğru yaymayı değiştirir. Dahası, zamanda geri yayma yukarıdaki gradyanları hesaplar ve sırayla depolar. Özellikle, depolanan ara değerler yinelemeli hesaplamaları önlemek için yeniden kullanılır, örneğin $\partial L/\partial \mathbf{W}_{hx}$ ve $\partial L/\partial \mathbf{W}_{hh}$ hesaplamalarında kullanılacak $\partial L/\partial \mathbf{h}_t$'yi depolamak gibi.

8.7.3 Özet

- Zamanda geriye yayma, sadece geri yaymanın gizli bir duruma sahip dizi modellerindeki bir uygulamasıdır.
- Hesaplama kolaylığı ve sayısal kararlılık için kesme gereklidir, örneğin düzenli kesme ve rasgele kesme gibi.
- Matrislerin yüksek kuvvetleri ıraksayan veya kaybolan özdeğerlere yol açabilir. Bu, patlayan veya kaybolan gradyanlar şeklinde kendini gösterir.
- Verimli hesaplama için ara değerler zamanda geri yayma sırasında önbelleğe alınır.

8.7.4 Alıştırmalar

1. λ_i özdeğerleri $\mathbf{v}_i$ $(i = 1, \ldots, n)$ özvektörlerine karşılık gelen bir simetrik matrisimiz $\mathbf{M} \in \mathbb{R}^{n \times n}$ olduğunu varsayalım. Genelleme kaybı olmadan, $|\lambda_i| \geq |\lambda_{i+1}|$ diye sıralanmış olduğunu varsayalım.
2. λ_i^k'nin $\mathbf{M}^k$'nin özdeğerleri olduğunu gösterin.
3. Yüksek olasılıkla $\mathbf{x} \in \mathbb{R}^n$ rastgele vektörü için $\mathbf{M}^k\mathbf{x}$ özvektörünün $\mathbf{M}$'deki $\mathbf{v}_1$ özvektörüyle hizalanmış olacağını kanıtlayın. Bu ifadeyi formüle dökün.
4. Yukarıdaki sonuç RNN'lerdeki gradyanlar için ne anlama geliyor?
5. Gradyan kırpmanın yanı sıra, yinelemeli sinir ağlarında gradyan patlaması ile başa çıkmak için başka yöntemler düşünebiliyor musunuz?

Tartışmalar[110]

[110] https://discuss.d2l.ai/t/334

9 | Modern Yinelemeli Sinir Ağları

Dizi verileri daha iyi işleyebilen RNN'nin temellerini tanıttık. Tanışmak amacıyla, metin verilerine RNN tabanlı dil modelleri uyguladık. Ancak, bu tür teknikler günümüzdeki çok çeşitli dizi öğrenme problemleriyle karşı karşıya kaldıklarında uygulayıcılar için yeterli olmayabilir.

Örneğin, pratikte önemli bir konu RNN'lerin sayısal dengesizliğidir. Gradyan kırpma gibi uygulama püf noktalarını uygulamış olsak da, bu sorun daha gelişmiş dizi modelleriyle daha da hafifletilebilir. Özellikle, geçitli RNN'ler pratikte çok daha yaygındır. Bu tür yaygın olarak kullanılan ağlardan ikisini, yani *geçitli yinelemeli birimler (gated recurrent units)* (GRU'lar) ve *uzun ömürlü kısa-dönem belleği (long short-term memory)* (LSTM) sunarak başlayacağız. Ayrıca, şimdiye kadar tartışılan tek yönlü gizli katmanlı RNN mimarisini genişleteceğiz. Derin mimarileri birden fazla gizli katmanla tanımlayacağız ve iki yönlü tasarımı hem ileri hem de geri yinelemeli hesaplamalarla tartışacağız. Bu tür genişlemeler, modern yinelemeli ağlarda sıklıkla benimsenir. Bu RNN türlerini açıklarken, Chapter 8 içinde tanıtılan aynı dil modelleme problemini dikkate almaya devam ediyoruz.

Aslında, dil modellemesi, dizi öğrenmenin yapabileceklerinin sadece küçük bir kısmını ortaya koymaktadır. Otomatik konuşma tanıma, metinden konuşmaya dönüştürme ve makine çevirisi gibi çeşitli dizi öğrenme problemlerinde hem girdiler hem de çıktılar keyfi uzunluktaki dizilerdir. Bu tür verilere nasıl uyarlanacağını açıklamak için makine çevirisini örnek olarak ele alacağız ve dizi üretimi için ışın aramasını ve kodlayıcı-kodçözücü mimarisine dayalı RNN'leri tanıtacağız.

9.1 Geçitli Yinelemeli Birimler (GRU)

Section 8.7 içinde, gradyanların RNN'lerde nasıl hesaplandığını tartıştık. Özellikle matrislerin uzun çarpımlarının kaybolan veya patlayan gradyanlara yol açabileceğini gördük. Bu tür gradyan sıradışılıklarının pratikte ne anlama geldiğini hakkında kısaca düşünelim:

- Gelecekteki tüm gözlemleri tahmin etmek için erken bir gözlemin son derece önemli olduğu bir durumla karşılaşabiliriz. Biraz karmaşık bir durumu düşünün; ilk gözlem bir sağlama toplamı (checksum) içerir ve hedef de sağlama toplamının dizinin sonunda doğru olup olmadığını fark etmektir. Bu durumda, ilk andıcın etkisi hayati önem taşır. Hayati önem taşıyan erken bilgileri bir *bellek hücresinde* depolamak için bazı mekanizmalara sahip olmak isteriz. Böyle bir mekanizma olmadan, bu gözlemlere çok büyük bir gradyan atamak zorunda kalacağız, çünkü sonraki tüm gözlemleri etkilerler.
- Bazı andıçların uygun gözlem taşımadığı durumlarla karşılaşabiliriz. Örneğin, bir web sayfasını ayrıştırırken, sayfada iletilen duygunun değerlendirilmesi amacıyla alakasız olan yardımcı HTML kodu olabilir. Gizli durum temsilinde bu tür andıçları *atlamak* için birtakım mekanizmaya sahip olmak isteriz.

- Bir dizinin parçaları arasında mantıksal bir kırılma olduğu durumlarla karşılaşabiliriz. Örneğin, bir kitaptaki bölümler arasında bir geçiş veya menkul kıymetler piyasasında hisse değerleri arasında bir geçiş olabilir. Bu durumda iç durum temsilimizi *sıfırlamak* için bir araca sahip olmak güzel olurdu.

Bunu ele almak için bir dizi yöntem önerilmiştir. İlk öncülerden biri Section 9.2 içinde tartışacağımız uzun ömürlü kısa-dönem belleğidir (Hochreiter and Schmidhuber, 1997). Geçitli yinelemeli birim (GRU) (Cho *et al.*, 2014), genellikle benzer performans sunan ve hesaplanmanın önemli ölçüde daha hızlı olduğu biraz daha elverişli bir türdür (Chung *et al.*, 2014). Sadeliğinden dolayı, GRU ile başlayalım.

9.1.1 Geçitli Gizli Durum

Sıradan RNN ve GRU'lar arasındaki anahtar ayrım, ikincisinin gizli durumu geçitlemeyi desteklemesidir. Bu, gizli bir durumun *güncellenmesi* gerektiği zamanlara ve ayrıca *sıfırlanması* gerektiği zamanlara yönelik özel mekanizmalarımız olduğu anlamına gelir. Bu mekanizmalar öğrenilir ve yukarıda listelenen kaygıları ele alır. Örneğin, ilk andıç büyük önem taşıyorsa, ilk gözlemden sonra gizli durumu güncellememeyi öğreneceğiz. Aynı şekilde, ilgisiz geçici gözlemleri atlamayı öğreneceğiz. Son olarak, gerektiğinde gizli durumu sıfırlamayı öğreneceğiz. Bunları aşağıda ayrıntılı olarak tartışıyoruz.

Sıfırlama Geçidi ve Güncelleme Geçidi

Tanışmamız gereken ilk kavramlar, *sıfırlama geçidi* ve *güncelleme geçidi*dir. Onları $(0, 1)$'de girdileri olan vektörler olacak şekilde tasarlıyoruz, böylece dışbükey bileşimleri gerçekleştirebiliriz. Örneğin, bir sıfırlama geçidi, önceki durumun ne kadarını hala hatırlamak isteyebileceğimizi kontrol etmemizi sağlar. Aynı şekilde, bir güncelleme geçidi yeni durumun ne kadarının eski durumun bir kopyası olacağını kontrol etmemizi sağlayacaktır.

Bu geçitleri işleyerek başlıyoruz. Fig. 9.1.1, mevcut zaman adımının girdiyi ve önceki zaman adımının gizli durumu göz önüne alındığında, bir GRU'daki hem sıfırlama hem de güncelleme geçitleri için girdileri göstermektedir. İki geçidin çıktıları, sigmoid etkinleştirme işlevine sahip iki tam bağlı katman tarafından verilir.

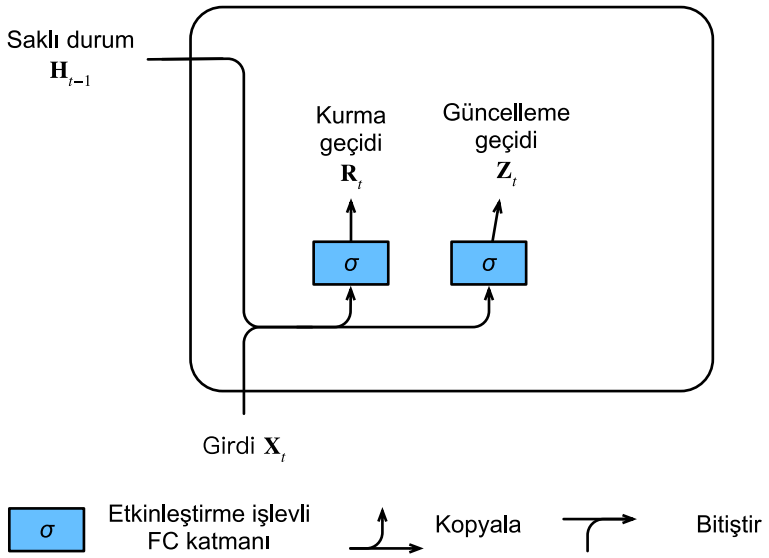


Fig. 9.1.1: Bir GRU modelinde sıfırlama ve güncelleme geçitlerini hesaplama.

Matematiksel olarak, belirli bir zaman adımı t için, girdinin bir minigrubun $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (örnek sayısı: n, girdi sayısı: d) ve önceki zaman adımının gizli durumunun $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$ (gizli birimlerin sayısı: h) olduğunu varsayalım. O zaman, sıfırlama geçidi $\mathbf{R}_t \in \mathbb{R}^{n \times h}$ ve güncelleştirme geçidi $\mathbf{Z}_t \in \mathbb{R}^{n \times h}$ aşağıdaki gibi hesaplanır:

$$\begin{aligned}\mathbf{R}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xr} + \mathbf{H}_{t-1} \mathbf{W}_{hr} + \mathbf{b}_r),\\ \mathbf{Z}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xz} + \mathbf{H}_{t-1} \mathbf{W}_{hz} + \mathbf{b}_z),\end{aligned} \tag{9.1.1}$$

burada $\mathbf{W}_{xr}, \mathbf{W}_{xz} \in \mathbb{R}^{d \times h}$ ve $\mathbf{W}_{hr}, \mathbf{W}_{hz} \in \mathbb{R}^{h \times h}$ ağırlık parametreleridir ve $\mathbf{b}_r, \mathbf{b}_z \in \mathbb{R}^{1 \times h}$ ek girdilerdir. Yayınlamanın (bkz. Section 2.1.3) toplama sırasında tetiklendiğini unutmayın. Girdi değerlerini $(0, 1)$ aralığına dönüştürmek için sigmoid işlevleri (Section 4.1 içinde tanıtıldığı gibi) kullanırız.

Aday Gizli Durum

Ardından, $\mathbf{R}_t$'i sıfırlama geçidini (8.4.5) denklemindeki normal gizli durum güncelleme mekanizmasıyla tümleştirelim. Bizi t zaman adımında aşağıdaki *aday gizli durum* $\tilde{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$'ye yönlendirir:

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1}) \mathbf{W}_{hh} + \mathbf{b}_h), \tag{9.1.2}$$

burada $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$ ve $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ ağırlık parametreleridir, $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ ek girdi ve $\odot$ sembolü Hadamard (eleman yönlü) çarpım işlemidir. Burada, aday gizli durumdaki değerlerin $(-1, 1)$ aralığında kalmasını sağlamak için tanh şeklinde bir doğrusal olmayan bir işlev kullanıyoruz.

Sonuç, hala güncelleme geçidinin eylemini dahil etmemiz gerektiğinden bir *adaydır*. (8.4.5) ile karşılaştırıldığında, önceki durumların etkisi $\mathbf{R}_t$ ve $\mathbf{H}_{t-1}$'nin (9.1.2) denkleminde eleman yönlü çarpımı ile azaltılabilir. Sıfırlama geçidi $\mathbf{R}_t$ girdileri 1'e yakın olduğunda, (8.4.5) denkleminde olduğu gibi sıradan bir RNN elde ederiz. Sıfırlama geçidi $\mathbf{R}_t$'nın 0'a yakın olan tüm girdileri için, aday gizli durum, girdisi $\mathbf{X}_t$ olan bir MLP'nin sonucudur. Önceden var olan herhangi bir gizli durum böylece *sıfırlanarak* varsayılanlara döner.

Fig. 9.1.2 sıfırlama geçidini uyguladıktan sonraki hesaplama akışını gösterir.

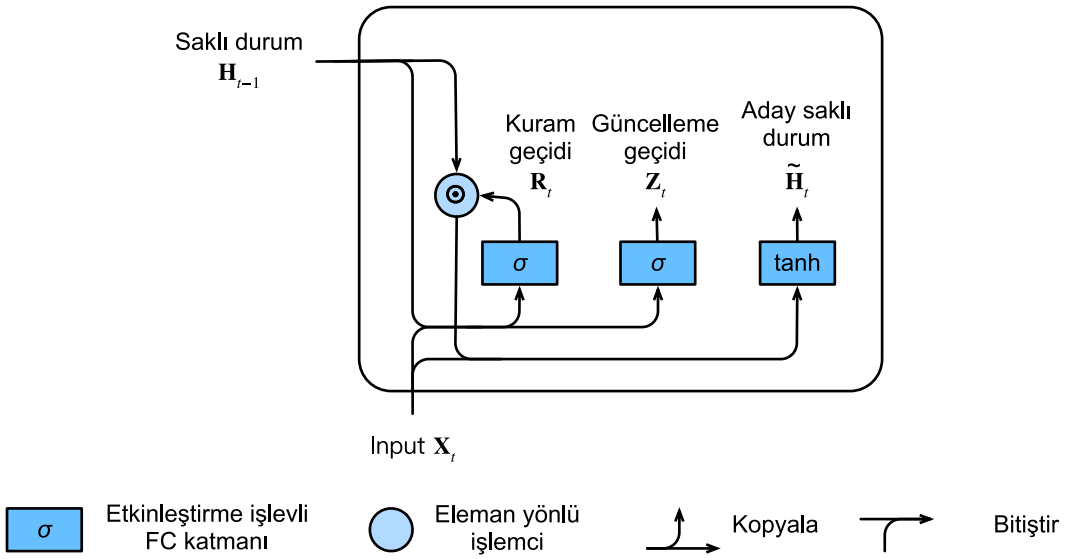


Fig. 9.1.2: GRU'de aday gizli durumu hesaplama.

Gizli Durum

Son olarak, güncelleme geçidi $\mathbf{Z}_t$'nin etkisini dahil etmemiz gerekiyor. Bu, yeni gizli durum $\mathbf{H}_t \in \mathbb{R}^{n \times h}$'in sadece eski durum $\mathbf{H}_{t-1}$'in ve yeni aday durum $\tilde{\mathbf{H}}_t$'nin ne kadar kullanıldığını belirler. $\mathbf{Z}_t$ güncelleme geçidi bu amaçla kullanılabilir, basitçe hem $\mathbf{H}_{t-1}$ hem de $\tilde{\mathbf{H}}_t$ arasındaki eleman yönlü dışbükey birleşimler alarak kullanılabilir. Bu da, GRU için son güncelleştirme denklemine yol açar:

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t. \tag{9.1.3}$$

Güncelleme geçidi $\mathbf{Z}_t$ 1'e yakın olduğunda, sadece eski durumu koruruz. Bu durumda $\mathbf{X}_t$'den gelen bilgiler esasen göz ardı edilir, bağlılık zincirinde t zaman adımı etkin bir şekilde atlanır. Buna karşılık, $\mathbf{Z}_t$ 0'a yakın olduğunda, yeni gizli durum $\mathbf{H}_t$ aday gizli durum $\tilde{\mathbf{H}}_t$'ye yaklaşır. Bu tasarımlar, RNN'lerdeki kaybolan gradyan sorunuyla başa çıkmamıza ve büyük zaman adım mesafeleri olan diziler için bağlılıkları daha iyi yakalamamıza yardımcı olabilirler. Örneğin, güncelleme geçidi tüm bir alt dizinin tüm zaman adımları için 1'e yakınsa, başlangıç zamanındaki eski gizli durum alt sıranın uzunluğuna bakılmaksızın kolayca korunur ve sonuna kadar geçirilir.

Fig. 9.1.3 güncelleme geçidi harekete geçtikten sonra hesaplama akışını gösterir.

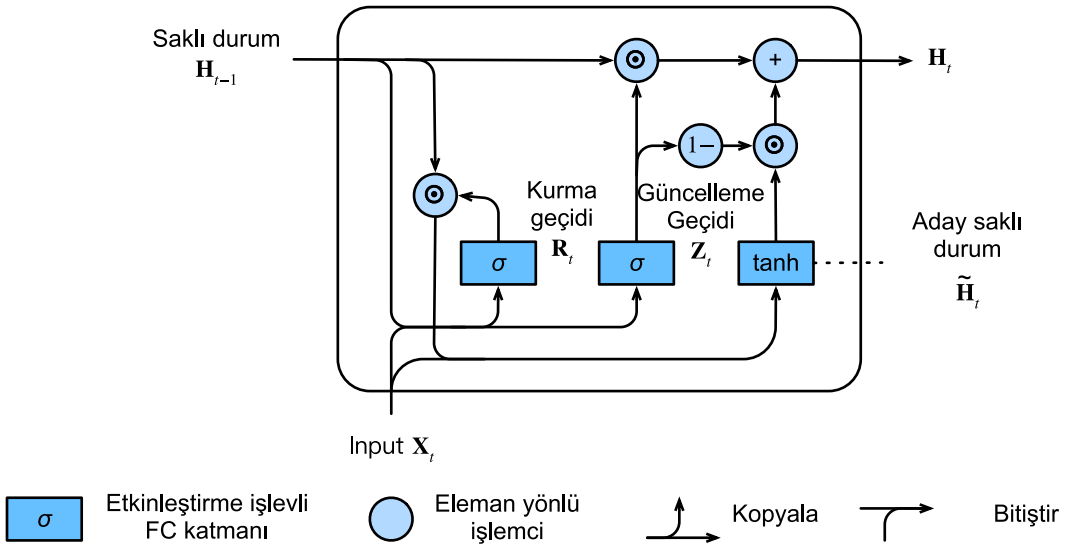


Fig. 9.1.3: GRU modelinde gizli durumu hesaplama.

Özetle, GRU'lar aşağıdaki iki ayırt edici özelliğe sahiptir:

- Sıfırlama geçitleri dizilerdeki kısa vadeli bağlılıkları yakalamaya yardımcı olur.
- Güncelleme geçitleri dizilerdeki uzun vadeli bağlılıkları yakalamaya yardımcı olur.

9.1.2 Sıfırdan Uygulama

GRU modelini daha iyi anlamak için sıfırdan uygulayalım. Section 8.5 içinde kullandığımız zaman makinesi veri kümesini okuyarak başlıyoruz. Veri kümesini okuma kodu aşağıda verilmiştir.

```
import torch
from torch import nn
from d2l import torch as d2l

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

Model Parametrelerini İlkleme

Bir sonraki adım model parametrelerini ilklemektir. Ağırlıkları standart sapması 0.01 olan bir Gauss dağılımından çekiyoruz ve ek girdiyi 0'a ayarlıyoruz. Hiper parametre `num_hiddens`, gizli birimlerin sayısını tanımlar. Güncelleme geçidi, sıfırlama geçidi, aday gizli durumu ve çıktı katmanı ile ilgili tüm ağırlıkları ve ek girdileri ilkleyeceğiz.

```
def get_params(vocab_size, num_hiddens, device):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return torch.randn(size=shape, device=device)*0.01
```

(continues on next page)

(continued from previous page)

```
    def three():
        return (normal((num_inputs, num_hiddens)),
                normal((num_hiddens, num_hiddens)),
                torch.zeros(num_hiddens, device=device))

    W_xz, W_hz, b_z = three()  # Geçit parametrelerini güncelle
    W_xr, W_hr, b_r = three()  # Geçit parametrelerini sıfırla
    W_xh, W_hh, b_h = three()  # Aday gizli durum parametreleri
    # Çıktı katmanı parametreleri
    W_hq = normal((num_hiddens, num_outputs))
    b_q = torch.zeros(num_outputs, device=device)
    # Gradyanları iliştir
    params = [W_xz, W_hz, b_z, W_xr, W_hr, b_r, W_xh, W_hh, b_h, W_hq, b_q]
    for param in params:
        param.requires_grad_(True)
    return params
```

Modelin Tanımlanması

Şimdi gizli durum ilkleme işlevini tanımlayacağız `init_gru_state`. Section 8.5 içinde tanımlanan `init_rnn_state` işlevi gibi, bu işlev, değerleri sıfırlar olan (toplu boyut, gizli birim sayısı) şekline sahip bir tensör döndürür.

```
def init_gru_state(batch_size, num_hiddens, device):
    return (torch.zeros((batch_size, num_hiddens), device=device), )
```

Şimdi GRU modelini tanımlamaya hazırız. Yapısı, güncelleme denklemlerinin daha karmaşık olması dışında, temel RNN hücresinin yapısı ile aynıdır.

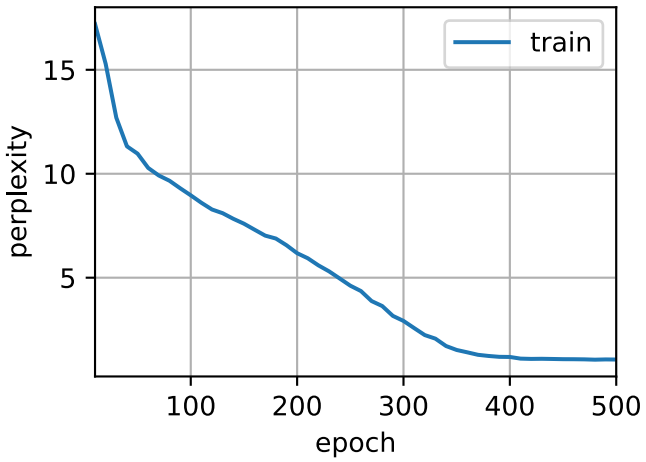
```
def gru(inputs, state, params):
    W_xz, W_hz, b_z, W_xr, W_hr, b_r, W_xh, W_hh, b_h, W_hq, b_q = params
    H, = state
    outputs = []
    for X in inputs:
        Z = torch.sigmoid((X @ W_xz) + (H @ W_hz) + b_z)
        R = torch.sigmoid((X @ W_xr) + (H @ W_hr) + b_r)
        H_tilda = torch.tanh((X @ W_xh) + ((R * H) @ W_hh) + b_h)
        H = Z * H + (1 - Z) * H_tilda
        Y = H @ W_hq + b_q
        outputs.append(Y)
    return torch.cat(outputs, dim=0), (H,)
```

Eğitme ve Tahmin Etme

Eğitim ve tahmin, tam olarak Section 8.5 içindekiyle aynı şekilde çalışır. Eğitimden sonra, sırasıyla sağlanan "zaman yolcusu" ve "yolcusu" ön eklerini takip eden eğitim kümesindeki şaşkınlığı ve tahmin edilen diziyi yazdırırız.

```
vocab_size, num_hiddens, device = len(vocab), 256, d2l.try_gpu()
num_epochs, lr = 500, 1
model = d2l.RNNModelScratch(len(vocab), num_hiddens, device, get_params,
                            init_gru_state, gru)
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.1, 24069.9 tokens/sec on cuda:0
time traveller for so it will be convenient to speak of himwas e
traveller aboutter new yarkmust of the ithed heldrea seamo
```

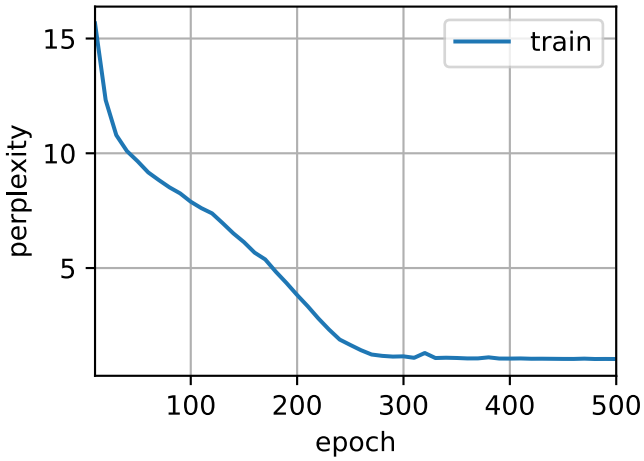


9.1.3 Kısa Uygulama

Üst düzey API'lerde, doğrudan bir GPU modelini oluşturabiliriz. Bu, yukarıda açıkça yaptığımız tüm yapılandırma ayrıntılarını gizler. Kod, daha önce yazdığımız birçok ayrıntı için Python yerine derlenmiş operatörleri kullandığı için önemli ölçüde daha hızlıdır.

```
num_inputs = vocab_size
gru_layer = nn.GRU(num_inputs, num_hiddens)
model = d2l.RNNModel(gru_layer, len(vocab))
model = model.to(device)
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.0, 325398.8 tokens/sec on cuda:0
time travelleryou can show black is white by argument said filby
travelleryou can show black is white by argument said filby
```



9.1.4 Özet

- Geçitli RNN'ler, büyük zaman adım mesafeleri olan diziler için bağlılıkları daha iyi yakalayabilir.
- Sıfırlama geçitleri dizilerdeki kısa vadeli bağlılıkları yakalamaya yardımcı olur.
- Güncelleme geçitleri dizilerdeki uzun vadeli bağlılıkları yakalamaya yardımcı olur.
- GRU'lar, sıfırlama geçidi açıldığında en uç durum olarak temel RNN'leri içerir. Ayrıca güncelleme geçidini açarak alt dizileri atlayabilirler.

9.1.5 Alıştırmalar

1. Sadece t' zaman adımını girdi olarak kullanarak $t > t'$ zaman adımlarındaki çıktıyı tahmin etmek için istediğimizi varsayalım. Her zaman adımında sıfırlama ve güncelleme geçitleri için en iyi değerler nelerdir?
2. Hiper parametreleri ayarlayın ve çalışma süresi, şaşkınlık ve çıktı dizisi üzerindeki etkilerini analiz edin.
3. `rnn.RNN` ve `rnn.GRU` uygulamaları için çalışma zamanı, şaşkınlık ve çıktı dizgilerini birbirleriyle karşılaştırın.
4. Yalnızca GRU'nun parçalarını, örneğin yalnızca bir sıfırlama geçidi veya yalnızca bir güncelleme geçidi, uygularsanız ne olur?

Tartışmalar[111]

[111] https://discuss.d2l.ai/t/1056

9.2 Uzun Ömürlü Kısa-Dönem Belleği (LSTM)

Gizli değişkenli modellerde uzun vadeli bilgi koruma ve kısa vadeli girdi atlama sorunu uzun zamandır var olmuştur. Bunu ele almak için öncü yaklaşımlardan biri uzun ömürlü kısa-dönem belleği (LSTM) (Hochreiter and Schmidhuber, 1997) oldu. GRU'nun birçok özelliklerini paylaşıyor. İlginçtir ki, LSTM'ler GRU'lardan biraz daha karmaşık bir tasarıma sahiptir, ancak GRU'lardan neredeyse yirmi yıl önce ortaya konmuştur.

9.2.1 Geçitli Bellek Hücresi

Muhtemelen LSTM'nin tasarımı bir bilgisayarın mantık kapılarından esinlenilmiştir. LSTM, gizli durumla aynı şekle sahip bir *bellek hücresi* (veya kısaca *hücre*) tanıtır (bazı çalışmalar, bellek hücresini gizli durumun özel bir türü olarak görür), ki ek bilgileri kaydetmek için tasarlanmıştır. Hafıza hücresini kontrol etmek için birkaç geçide ihtiyacımız vardır. Hücreden girdileri okumak için bir geçit gerekiyor. Biz buna *çıktı geçidi* olarak atıfta bulunacağız. Hücreye veri ne zaman okunacağına karar vermek için ikinci bir geçit gereklidir. Bunu *girdi geçidi* olarak adlandırıyoruz. Son olarak, *unutma geçidi* tarafından yönetilen hücrenin içeriğini sıfırlamak için bir mekanizmaya ihtiyacımız var. Böyle bir tasarımın motivasyonu GRU'larla aynıdır, yani özel bir mekanizma aracılığıyla gizli durumdaki girdileri ne zaman hatırlayacağınıza ve ne zaman gözardı edeceğinize karar verebilmek. Bunun pratikte nasıl çalıştığını görelim.

Girdi Geçidi, Unutma Geçidi ve Çıktı Geçidi

Tıpkı GRU'larda olduğu gibi, LSTM geçitlerine beslenen veriler, Fig. 9.2.1 içinde gösterildiği gibi, geçerli zaman adımındaki girdi ve önceki zaman adımının gizli durumudur. Girdi, unutma ve çıktı geçitleri değerlerini hesaplamak için sigmoid etkinleştirme fonksiyonuna sahip üç tam bağlı katman tarafından işlenir. Sonuç olarak, üç geçidin değerleri $(0, 1)$ aralığındadır.

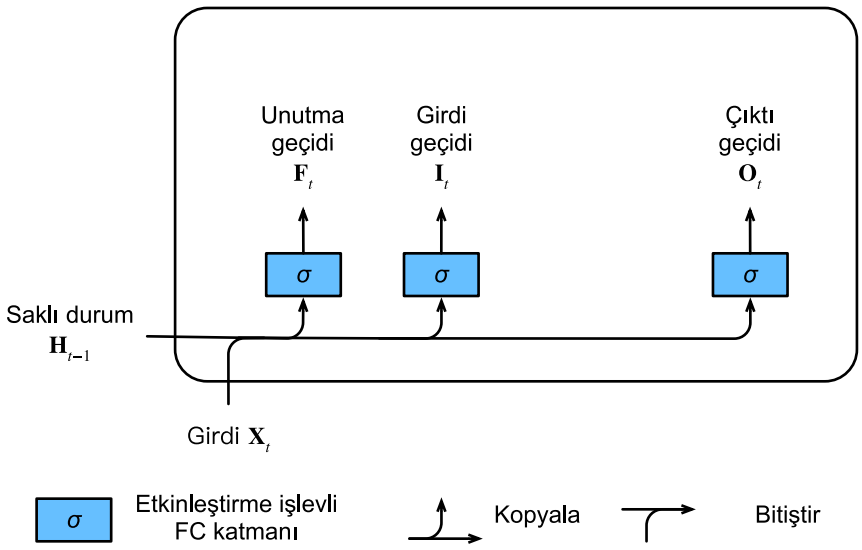


Fig. 9.2.1: LSTM modelinin girdi, unutma ve çıktı geçitleri değerlerininin hesaplanması

Matematiksel olarak, h tane gizli birim, n tane toplu iş ve d tane girdi olduğunu varsayalım. Böylece, girdi $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ ve önceki zaman adımının gizli durumu $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$'dir. Buna göre, t

zaman adımındaki geçitler şu şekilde tanımlanır: Girdi geçidi $\mathbf{I}_t \in \mathbb{R}^{n \times h}$, unutma geçidi $\mathbf{F}_t \in \mathbb{R}^{n \times h}$ ve çıktı geçidi $\mathbf{O}_t \in \mathbb{R}^{n \times h}$'dır. Bunlar aşağıdaki gibi hesaplanır:

$$\begin{aligned}\mathbf{I}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i),\\ \mathbf{F}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f),\\ \mathbf{O}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o),\end{aligned} \tag{9.2.1}$$

burada $\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo} \in \mathbb{R}^{d \times h}$ ve $\mathbf{W}_{hi}, \mathbf{W}_{hf}, \mathbf{W}_{ho} \in \mathbb{R}^{h \times h}$ ağırlık parametreleridir ve $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^{1 \times h}$ ek girdi parametreleridir.

Aday Bellek Hücresi

Sonraki adımda hafıza hücresini tasarlıyoruz. Çeşitli geçitlerin eylemini henüz belirtmediğimizden, öncelikle *aday* bellek hücresi $\tilde{\mathbf{C}}_t \in \mathbb{R}^{n \times h}$'i tanıtıyoruz. Hesaplamalar, yukarıda açıklanan üç geçittekine benzer, ancak etkinleştirme fonksiyonu olarak $(-1, 1)$'de bir değer aralığına sahip bir tanh işlevini kullanır. Bu, t zaman adımında aşağıdaki denkleme yol açar:

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xc} + \mathbf{H}_{t-1} \mathbf{W}_{hc} + \mathbf{b}_c), \tag{9.2.2}$$

burada $\mathbf{W}_{xc} \in \mathbb{R}^{d \times h}$ ve $\mathbf{W}_{hc} \in \mathbb{R}^{h \times h}$ ağırlık parametreleridir ve $\mathbf{b}_c \in \mathbb{R}^{1 \times h}$ bir ek girdi parametresidir.

Aday bellek hücresinin hızlı bir gösterimi Fig. 9.2.2 içinde verilmiştir.

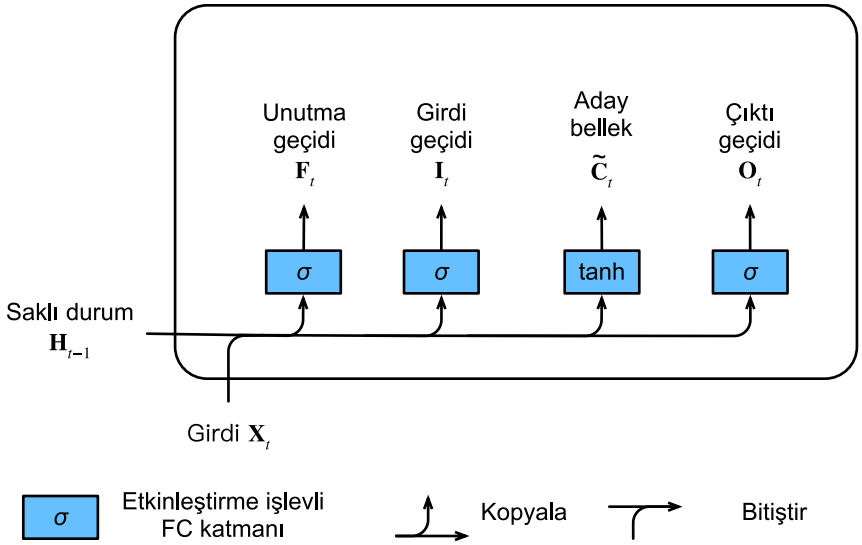


Fig. 9.2.2: LSTM modelinde aday bellek hücresini hesaplama.

Bellek Hücresi

GRU'larda, girdiyi ve unutmayı (veya atlamayı) yönetecek bir mekanizmamız vardır. Benzer şekilde, LSTM'lerde bu tür amaçlar için iki özel geçidimiz var: $\mathbf{I}_t$ girdi geçidi $\tilde{\mathbf{C}}_t$ aracılığıyla yeni verileri ne kadar hesaba kattığımızı ve $\mathbf{F}_t$, eski bellek hücresi içeriğinin $\mathbf{C}_{t-1} \in \mathbb{R}^{n \times h}$'nin ne kadarını tuttuğumuzu kontrol eder. Daha önce olduğu gibi aynı noktasal çarpım hilesini kullanarak, aşağıdaki güncelleme denklemine ulaşırız:

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t. \tag{9.2.3}$$

Unutma geçidi her zaman yaklaşık 1 ise ve girdi geçidi her zaman yaklaşık 0 ise, geçmiş bellek hücreleri $\mathbf{C}_{t-1}$ zamanla kaydedilir ve geçerli zaman adımına geçirilir. Bu tasarım, kaybolan gradyan sorununu hafifletmek ve diziler içindeki uzun menzilli bağlılıkları daha iyi yakalamak için sunuldu.

Böylece Fig. 9.2.3 içindeki akış şemasına ulaşırız.

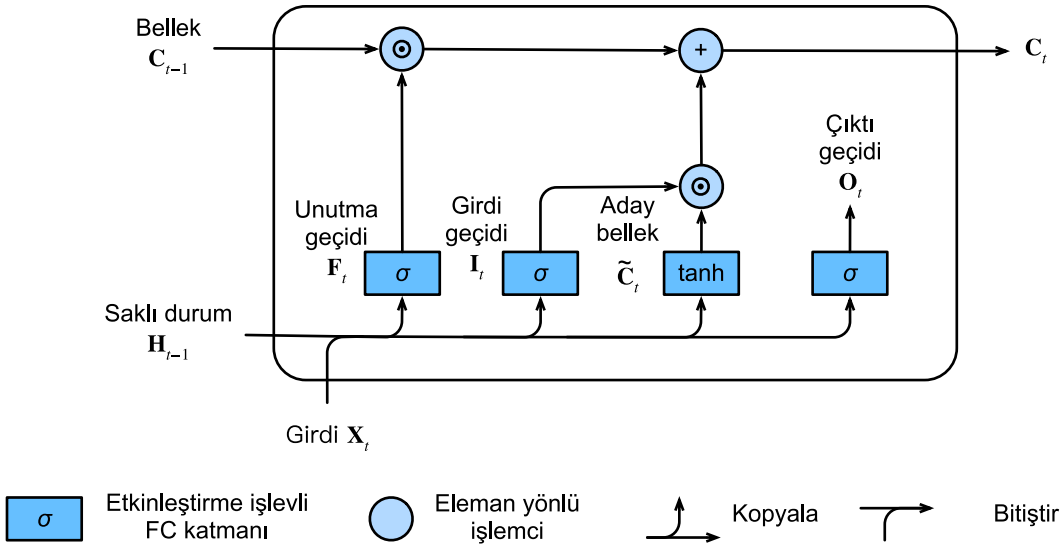


Fig. 9.2.3: LSTM modelinde bellek hücresini hesaplama.

Gizli Durum

Son olarak, $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ gizli durumunu nasıl hesaplayacağımızı tanımlamamız gerekiyor. Çıktı geçidinin devreye girdiği yer burasıdır. Basitçe LSTM'de, bellek hücresinin tanh'li geçitli versiyonudur. Bu, $\mathbf{H}_t$ değerlerinin her zaman $(-1, 1)$ aralığında olmasını sağlar.

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t). \tag{9.2.4}$$

Çıktı geçidi 1'e yaklaştığında, tüm bellek bilgilerini etkin bir şekilde tahminciye aktarırız, oysa 0'a yakın çıktı geçidi için tüm bilgileri yalnızca bellek hücresinde saklarız ve daha fazla işlem yapmayız.

Fig. 9.2.4 içinde veri akışının grafiksel bir gösterimi vardır.

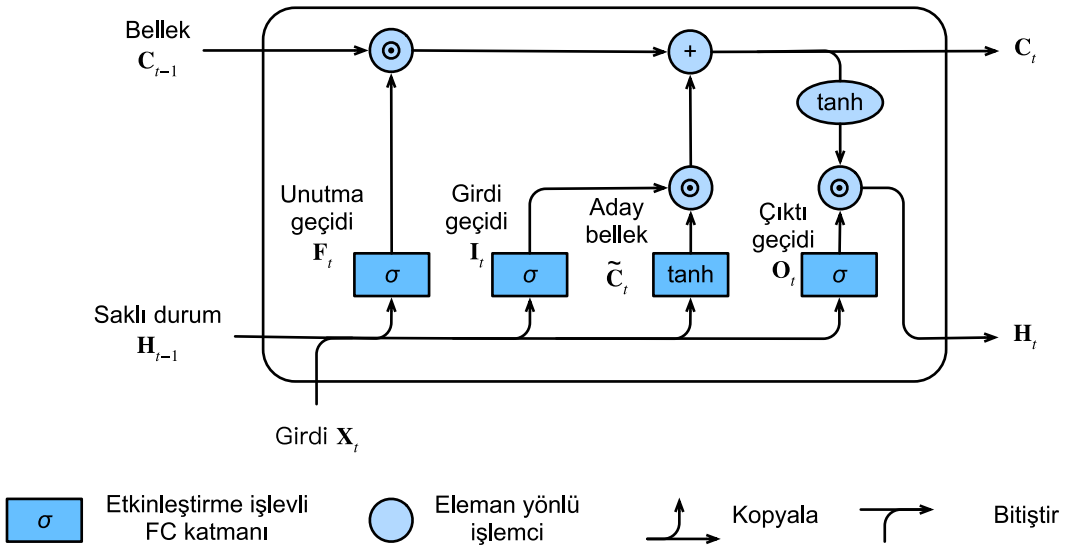


Fig. 9.2.4: Computing the hidden state in an LSTM model.

9.2.2 Sıfırdan Uygulama

Şimdi sıfırdan bir LSTM uygulayalım. Section 8.5 içindeki deneylerle aynı şekilde, önce zaman makinesi veri kümesini yükleriz.

```
import torch
from torch import nn
from d2l import torch as d2l

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

Model Parametrelerini İlkleme

Daha sonra model parametrelerini tanımlamamız ve ilklememiz gerekiyor. Daha önce olduğu gibi, hiper parametre `num_hiddens` gizli birimlerin sayısını tanımlar. 0.01 standart sapmalı Gauss dağılımı ile ağırlıkları ilkleriz ve ek girdileri 0'a ayarlarız.

```
def get_lstm_params(vocab_size, num_hiddens, device):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return torch.randn(size=shape, device=device)*0.01

    def three():
        return (normal((num_inputs, num_hiddens)),
                normal((num_hiddens, num_hiddens)),
                torch.zeros(num_hiddens, device=device))

    W_xi, W_hi, b_i = three()  # Girdi geçidi parametreleri
    W_xf, W_hf, b_f = three()  # Unutma geçidi parametreleri
    W_xo, W_ho, b_o = three()  # Çıktı geçidi parametreleri
```

(continues on next page)

(continued from previous page)

```
    W_xc, W_hc, b_c = three()  # Aday bellek hücresi parametreleri
    # Çıktı katmanı parametreleri
    W_hq = normal((num_hiddens, num_outputs))
    b_q = torch.zeros(num_outputs, device=device)
    # Gradyanları iliştir
    params = [W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc,
              b_c, W_hq, b_q]
    for param in params:
        param.requires_grad_(True)
    return params
```

Modeli Tanımlama

İlkleme işlevinde, LSTM'nin gizli durumunun değeri 0 ve şekli (toplu iş boyutu, gizli birimlerin sayısı) olan bir *ek* bellek hücresi döndürmesi gerekir. Böylece aşağıdaki durum ilklemesini elde ederiz.

```
def init_lstm_state(batch_size, num_hiddens, device):
    return (torch.zeros((batch_size, num_hiddens), device=device),
            torch.zeros((batch_size, num_hiddens), device=device))
```

Gerçek model, daha önce tartıştığımız gibi tanımlanmıştır: Üç geçit ve bir yardımcı bellek hücresi sağlar. Çıktı katmanına yalnızca gizli durumun iletildiğini unutmayın. Bellek hücresi $\mathbf{C}_t$ doğrudan çıktı hesaplamasına katılmaz.

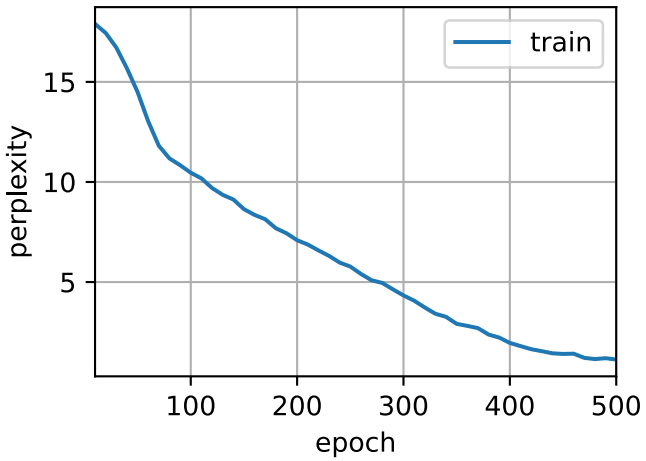
```
def lstm(inputs, state, params):
    [W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc, b_c,
     W_hq, b_q] = params
    (H, C) = state
    outputs = []
    for X in inputs:
        I = torch.sigmoid((X @ W_xi) + (H @ W_hi) + b_i)
        F = torch.sigmoid((X @ W_xf) + (H @ W_hf) + b_f)
        O = torch.sigmoid((X @ W_xo) + (H @ W_ho) + b_o)
        C_tilda = torch.tanh((X @ W_xc) + (H @ W_hc) + b_c)
        C = F * C + I * C_tilda
        H = O * torch.tanh(C)
        Y = (H @ W_hq) + b_q
        outputs.append(Y)
    return torch.cat(outputs, dim=0), (H, C)
```

Eğitim ve Tahmin Etme

Section 8.5 içinde tanıtılan `RNNModelScratch` sınıfını başlatarak Section 9.1 içinde yaptığımız gibi bir LSTM'yi eğitmeye başlayalım.

```
vocab_size, num_hiddens, device = len(vocab), 256, d2l.try_gpu()
num_epochs, lr = 500, 1
model = d2l.RNNModelScratch(len(vocab), num_hiddens, device, get_lstm_params,
                            init_lstm_state, lstm)
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.1, 20063.0 tokens/sec on cuda:0
time traveller for so it will be convenient to speak of himwas e
traveller curdictean the troug this betwy three mathematica
```

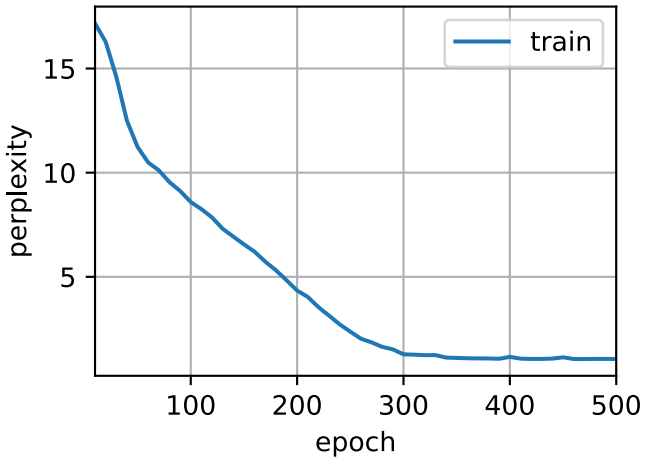


9.2.3 Kısa Uygulama

Üst düzey API'leri kullanarak doğrudan bir `LSTM` modeli oluşturabiliriz. Bu, yukarıda açıkça yaptığımız tüm yapılandırma ayrıntılarını gizler. Daha önce ayrıntılı olarak yazdığımız birçok detay yerine Python'un derlenmiş operatörlerini kullandığından kod önemli ölçüde daha hızlıdır.

```
num_inputs = vocab_size
lstm_layer = nn.LSTM(num_inputs, num_hiddens)
model = d2l.RNNModel(lstm_layer, len(vocab))
model = model.to(device)
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.1, 254131.5 tokens/sec on cuda:0
time travelleryou can show black is white by argument said filby
travelleryou can show black is white by argument said filby
```



LSTM'ler, apaçık olmayan durum kontrolüne sahip ilk örnek saklı değişken özbağlanımlı modeldir. Birçok türevi yıllar içinde önerilmiştir, örn. birden fazla katman, artık bağlantılar, farklı düzenlileştirme türleri. Bununla birlikte, LSTM'leri ve diğer dizi modellerini (GRU'lar gibi) eğitmek dizinin uzun menzilli bağlılığı nedeniyle oldukça maliyetlidir. Daha sonra bazı durumlarda kullanılabilen dönüştürücüler (transformers) gibi diğer seçenek modeller ile karşılaşacağız.

9.2.4 Özet

- LSTM'lerin üç tip geçit vardır: Girdi, unutma ve bilgi akışını kontrol eden çıktı geçitleri.
- Gizli katman çıktısı LSTM gizli durumu ve bellek hücresini içerir. Çıktı katmanına yalnızca gizli durum iletilir. Hafıza hücresi tamamen içseldir.
- LSTM'ler kaybolan ve patlayan gradyanları hafifletebilir.

9.2.5 Alıştırmalar

1. Hiper parametreleri ayarlayın ve çalışma süresi, şaşkınlık ve çıktı dizisi üzerindeki etkilerini analiz edin.
2. Karakter dizilerinin aksine doğru kelimeler üretmek için modeli nasıl değiştirmeniz gerekir?
3. Belirli bir gizli boyuttaki GRU'lar, LSTM'ler ve normal RNN'ler için hesaplama maliyetini karşılaştırın. Eğitim ve çıkarsama maliyetine özel dikkat gösterin.
4. Aday hafıza hücresi, tanh fonksiyonunu kullanarak değer aralığının -1 ile 1 arasında olmasını sağladığından, çıktı değeri aralığının -1 ile 1 arasında olmasını sağlamak için gizli durumun neden tekrar tanh fonksiyonunu kullanması gerekiyor?
5. Karakter sırası tahmini yerine zaman serisi tahmini için bir LSTM modeli uygulayın.

Tartışmalar[112]

[112] https://discuss.d2l.ai/t/1057

9.3 Derin Yinelemeli Sinir Ağları

Şimdiye kadar, tek bir tek yönlü gizli katmanlı RNN'leri tartıştık. İçinde gizli değişkenlerin ve gözlemlerin nasıl etkileşime girdiğinin özgül fonksiyonel formu oldukça keyfidir. Farklı etkileşim türlerini modellemek için yeterli esnekliğe sahip olduğumuz sürece bu büyük bir sorun değildir. Bununla birlikte, tek bir katman ile, bu oldukça zor olabilir. Doğrusal modellerde, daha fazla katman ekleyerek bu sorunu düzelttik. RNN'ler içinde bu biraz daha zor, çünkü öncelikle ek doğrusal olmayanlığın nasıl ve nerede ekleneceğine karar vermemiz gerekiyor.

Aslında, birden fazla RNN katmanını üst üste yığabiliriz. Bu, birkaç basit katmanın kombinasyonu sayesinde esnek bir mekanizma ile sonuçlanır. Özellikle, veriler yığının farklı düzeylerinde alakalı olabilir. Örneğin, finansal piyasa koşulları (hisse alım ve satım piyasası) ile ilgili üst düzey verileri elimizde mevcut tutmak isteyebiliriz, ancak daha düşük bir seviyede yalnızca kısa vadeli zamansal dinamikleri kaydederiz.

Yukarıdaki soyut tartışmaların ötesinde, Fig. 9.3.1 şeklini inceleyerek ilgilendiğimiz model ailesini anlamak muhtemelen en kolay yoldur. L adet gizli katmanlı derin bir RNN'yı göstermektedir. Her gizli durum, hem geçerli katmanın bir sonraki zaman adımına hem de bir sonraki katmanın şu anki zaman adımına sürekli olarak iletilir.

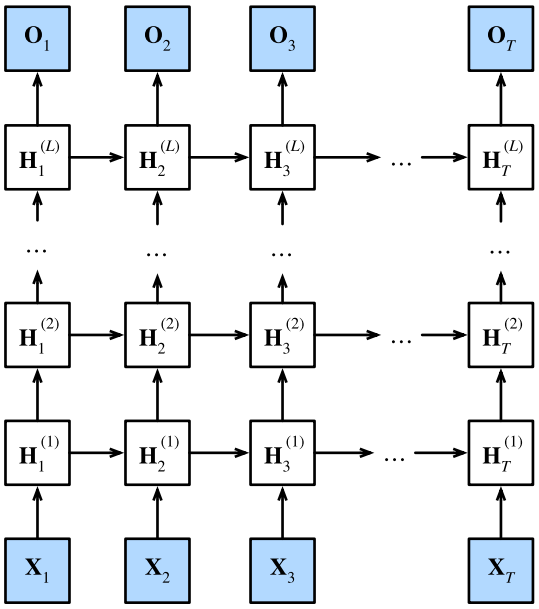


Fig. 9.3.1: Derin RNN mimarisi.

9.3.1 Fonksiyonel Bağlılıklar

Fig. 9.3.1 içinde tasvir edilen L gizli katmanın derin mimarisi içindeki işlevsel bağımlıkları formülize edebiliriz. Aşağıdaki tartışmamız öncelikle sıradan RNN modeline odaklanmaktadır, ancak diğer dizi modelleri için de geçerlidir.

Bir t zaman adımında $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ minigrup girdisine sahip olduğumuzu varsayalım (örnek sayısı: n, her örnekte girdi sayısı: d). Aynı zamanda adımda, $l.$ gizli katmanın ($l = 1, \ldots, L$), $\mathbf{H}_t^{(l)} \in \mathbb{R}^{n \times h}$ gizli durumunun (gizli birimlerin sayısı: h) ve $\mathbf{O}_t \in \mathbb{R}^{n \times q}$ çıktı katmanı değişkeninin (çıktı sayısı: q) olduğunu varsayalım. $\mathbf{H}_t^{(0)} = \mathbf{X}_t$, ϕ_l etkinleştirme işlevini kullanan $l.$ gizli katmanın gizli durumu

aşağıdaki gibi ifade edilir:

$$\mathbf{H}_t^{(l)} = \phi_l(\mathbf{H}_t^{(l-1)}\mathbf{W}_{xh}^{(l)} + \mathbf{H}_{t-1}^{(l)}\mathbf{W}_{hh}^{(l)} + \mathbf{b}_h^{(l)}), \tag{9.3.1}$$

burada $\mathbf{W}_{xh}^{(l)} \in \mathbb{R}^{h\times h}$ ve $\mathbf{W}_{hh}^{(l)} \in \mathbb{R}^{h\times h}$ ağırlıkları $\mathbf{b}_h^{(l)} \in \mathbb{R}^{1\times h}$ ek girdisi ile birlikte $l.$ gizli katmanın model parametreleridir.

Sonunda, çıktı katmanının hesaplanması yalnızca son $L.$ gizli katmanın gizli durumuna dayanır:

$$\mathbf{O}_t = \mathbf{H}_t^{(L)}\mathbf{W}_{hq} + \mathbf{b}_q, \tag{9.3.2}$$

burada ağırlık $\mathbf{W}_{hq} \in \mathbb{R}^{h\times q}$ ve ek girdi $\mathbf{b}_q \in \mathbb{R}^{1\times q}$, çıktı katmanının model parametreleridir.

MLP'lerde olduğu gibi, L gizli katmanların sayısı ve h gizli birimlerin sayısı hiper parametrelerdir. Başka bir deyişle, bunlar bizim tarafımızdan ayarlanabilir veya belirtilebilir. Buna ek olarak, (9.3.1) denklemindeki gizli durum hesaplamalarını GRU veya LSTM ile değiştirerek bir derin geçitli RNN elde edebiliriz.

9.3.2 Kısa Uygulama

Neyse ki, bir RNN'nin birden fazla katmanını uygulamak için gerekli olan lojistik detayların çoğu, üst düzey API'lerde kolayca mevcuttur. İşleri basit tutmak için sadece bu tür yerleşik işlevleri kullanarak uygulamayı gösteriyoruz. Örnek olarak bir LSTM modelini ele alalım. Kod, daha önce Section 9.2 içinde kullandığımıza çok benzer. Aslında, tek fark, tek bir katmanlı varsayılanı seçmek yerine açıkça katman sayısını belirtmemizdir. Her zamanki gibi, veri kümesini yükleyerek başlıyoruz.

```
import torch
from torch import nn
from d2l import torch as d2l

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

Hiper parametre seçimi gibi mimari kararlar Section 9.2 içindekine çok benzer. Farklı andıçlara sahip olduğumuz için aynı sayıda girdi ve çıktı seçiyoruz, yani `vocab_size`. Gizli birimlerin sayısı hala 256'dır. Tek fark, şimdi apaçık olmayan gizli katmanların sayısını `num_layers` değerini belirterek seçmemizdir.

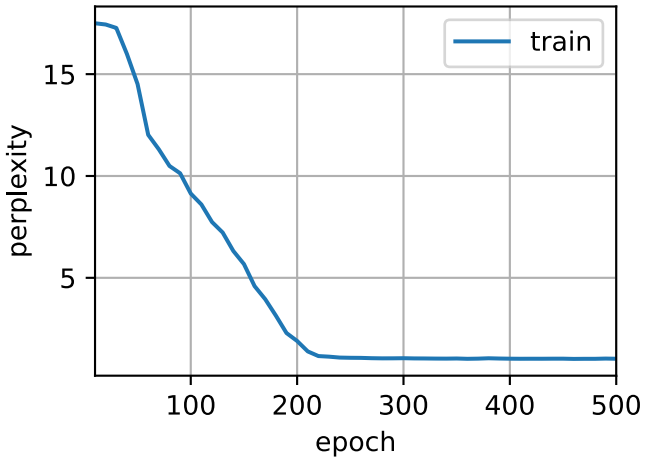
```
vocab_size, num_hiddens, num_layers = len(vocab), 256, 2
num_inputs = vocab_size
device = d2l.try_gpu()
lstm_layer = nn.LSTM(num_inputs, num_hiddens, num_layers)
model = d2l.RNNModel(lstm_layer, len(vocab))
model = model.to(device)
```

9.3.3 Eğitim ve Tahmin

Şu andan itibaren LSTM modeli ile iki katman oluşturuyoruz, bu oldukça karmaşık mimari eğitimi önemli ölçüde yavaşlatıyor.

```
num_epochs, lr = 500, 2
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.0, 215250.2 tokens/sec on cuda:0
time travelleryou can show black is white by argument said filby
traveller with a slight accession ofcheerfulness really thi
```



9.3.4 Özet

- Derin RNN'lerde, gizli durum bilgileri şu anki katmanın sonraki zaman adımına ve sonraki katmanın şu anki zaman adımına geçirilir.
- LSTM'ler, GRU'lar veya sıradan RNN gibi derin RNN'lerin birçok farklı seçenekleri vardır. Bu modellerin tümü, derin öğrenme çerçevesinin üst düzey API'lerinin bir parçası olarak mevcuttur.
- Modellerin ilklenmesi dikkat gerektirir. Genel olarak, derin RNN'ler doğru yakınsamayı sağlamak için önemli miktarda iş gerektirir (öğrenme hızı ve kırpma gibi).

9.3.5 Alıştırmalar

1. Section 8.5 içinde tartıştığımız tek katmanlı uygulamayı kullanarak sıfırdan iki katmanlı bir RNN uygulamaya çalışın.
2. LSTM'yi GRU ile değiştirin ve doğruluk ve eğitim hızını karşılaştırın.
3. Eğitim verilerini birden fazla kitap içerecek şekilde artırın. Şaşkınlık ölçeğinde ne kadar düşüğe inebilirsin?
4. Metni modellerken farklı yazarların kaynaklarını birleştirmek ister misiniz? Bu neden iyi bir fikir? Ne ters gidebilir ki?

9.4 Çift Yönlü Yinelemeli Sinir Ağları

Dizi öğrenmede, şu ana kadar amacımızın, şimdiye kadar gördüklerimizle bir sonraki çıktıyı modellemek olduğunu varsaydık; örneğin, bir zaman serisi veya bir dil modeli bağlamında. Bu tipik bir senaryo olsa da, karşılaşabileceğimiz tek senaryo değil. Sorunu göstermek için, bir metin dizisinde boşluk doldurmada aşağıdaki üç görevi gözünüzde canlandırın:

- Ben ___.
- ___ açım.
- ___ açım ve yarım kuzu yiyebilirim.

Mevcut bilgi miktarına bağlı olarak boşlukları "mutluyum", "yarı" ve "çok" gibi çok farklı kelimelerle doldurabiliriz. Açıkça ifadenin sonu (varsa) hangi kelimenin alınacağı hakkında önemli bilgiler taşır. Bundan yararlanamayan bir dizi modeli, ilgili görevlerde kötü performans gösterecektir. Örneğin, adlandırılmış varlık tanımada iyi iş yapmak için (örneğin, "Yeşil""in "Bay Yeşil""i ya da rengi ifade edip etmediğini tanımak) daha uzun menzilli bağlam aynı derecede hayati önem taşır. Problemi ele alırken biraz ilham almak için olasılıksal çizge modellerde biraz dolanalım.

9.4.1 Gizli Markov Modellerinde Dinamik Programlama

Bu alt bölüm, dinamik programlama problemini göstermeyi amaçlar. Belirli teknik detaylar derin öğrenme modellerini anlamak için önemli değildir, ancak neden derin öğrenmeyi kullanabileceğimizi ve neden belirli mimarileri seçebileceğimizi anlamaya yardımcı olurlar.

Problemi olasılıksal çizge modelleri kullanarak çözmek istiyorsak, mesela aşağıdaki gibi gizli bir değişken modeli tasarlayabiliriz. Herhangi bir t zaman adımında, $P(x_t \mid h_t)$ olasılığında x_t aracılığıyla gözlenen salınımı yöneten bazı gizli h_t değişkenimiz olduğunu varsayalım. Dahası, herhangi bir $h_t \to h_{t+1}$ geçişi, bir durum geçiş olasılığı $P(h_{t+1} \mid h_t)$ ile verilir. Bu olasılıksal çizge modeli Fig. 9.4.1 şeklinde olduğu gibi bir *saklı Markov modeli*dir.

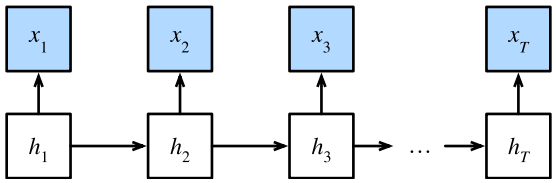


Fig. 9.4.1: Saklı Markov Modeli.

Böylece, T gözlemlerinin bir dizisi için gözlemlenen ve gizli durumlar üzerinde aşağıdaki bileşik olasılık dağılımına sahibiz:

$$P(x_1, \ldots, x_T, h_1, \ldots, h_T) = \prod_{t=1}^{T} P(h_t \mid h_{t-1}) P(x_t \mid h_t), \text{ öyleki } P(h_1 \mid h_0) = P(h_1). \tag{9.4.1}$$

[113] https://discuss.d2l.ai/t/1058

Şimdi bazı x_j'ler hariç tüm x_i'leri gözlemlediğimizi varsayalım ve amacımız $P(x_j \mid x_{-j})$'yı hesaplamaktır ve burada $x_{-j} = (x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_T)$'dir. $P(x_j \mid x_{-j})$'te saklı bir değişken olmadığından, $h_1, \ldots, h_T$ için olası tüm seçenek kombinasyonlarını toplamayı düşünürüz. Herhangi bir h_i'nin k farklı değerlere (sonlu sayıda durum) sahip olması durumunda, bu, k^T terimi toplamamız gerektiği anlamına gelir; bu da genellikle imkansız bir işlemdir! Neyse ki bunun için şık bir çözüm var: *Dinamik programlama*.

Nasıl çalıştığını görmek için, sırayla $h_1, \ldots, h_T$ saklı değişkenleri üzerinde toplamayı düşünün. (9.4.1) denklemine göre, şu ifadeye varırız:

$$\begin{aligned}
&P(x_1, \ldots, x_T) \\
&= \sum_{h_1, \ldots, h_T} P(x_1, \ldots, x_T, h_1, \ldots, h_T) \\
&= \sum_{h_1, \ldots, h_T} \prod_{t=1}^{T} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \\
&= \sum_{h_2, \ldots, h_T} \underbrace{\left[\sum_{h_1} P(h_1) P(x_1 \mid h_1) P(h_2 \mid h_1)\right]}_{\pi_2(h_2) \stackrel{\text{def}}{=}} P(x_2 \mid h_2) \prod_{t=3}^{T} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \\
&= \sum_{h_3, \ldots, h_T} \underbrace{\left[\sum_{h_2} \pi_2(h_2) P(x_2 \mid h_2) P(h_3 \mid h_2)\right]}_{\pi_3(h_3) \stackrel{\text{def}}{=}} P(x_3 \mid h_3) \prod_{t=4}^{T} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \\
&= \ldots \\
&= \sum_{h_T} \pi_T(h_T) P(x_T \mid h_T).
\end{aligned} \tag{9.4.2}$$

Genel bir *ileriye özyinelemeye* sahip oluruz:

$$\pi_{t+1}(h_{t+1}) = \sum_{h_t} \pi_t(h_t) P(x_t \mid h_t) P(h_{t+1} \mid h_t). \tag{9.4.3}$$

Özyineleme $\pi_1(h_1) = P(h_1)$ olarak ilklenir. Soyut terimlerle bu $\pi_{t+1} = f(\pi_t, x_t)$ olarak yazılabilir, burada f bir öğrenilebilir işlevdir. Bu, RNN bağlamında şimdiye kadar tartıştığımız saklı değişken modellerindeki güncelleme denklemine çok benziyor!

İleriye özyinelemeye tamamen benzer şekilde, geriye özyineleme ile aynı saklı değişken kümesi

üzerinden toplayabiliriz. Böylece şu ifadeye varırız:

$$
\begin{aligned}
& P(x_1, \ldots, x_T) \\
&= \sum_{h_1, \ldots, h_T} P(x_1, \ldots, x_T, h_1, \ldots, h_T) \\
&= \sum_{h_1, \ldots, h_T} \prod_{t=1}^{T-1} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \cdot P(h_T \mid h_{T-1}) P(x_T \mid h_T) \\
&= \sum_{h_1, \ldots, h_{T-1}} \prod_{t=1}^{T-1} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \cdot \underbrace{\left[\sum_{h_T} P(h_T \mid h_{T-1}) P(x_T \mid h_T)\right]}_{\rho_{T-1}(h_{T-1}) \stackrel{\text{def}}{=}} \\
&= \sum_{h_1, \ldots, h_{T-2}} \prod_{t=1}^{T-2} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \cdot \underbrace{\left[\sum_{h_{T-1}} P(h_{T-1} \mid h_{T-2}) P(x_{T-1} \mid h_{T-1}) \rho_{T-1}(h_{T-1})\right]}_{\rho_{T-2}(h_{T-2}) \stackrel{\text{def}}{=}} \\
&= \ldots \\
&= \sum_{h_1} P(h_1) P(x_1 \mid h_1) \rho_1(h_1).
\end{aligned}
\tag{9.4.4}
$$

Böylece *geriye özyineleme* olarak yazabiliriz:

$$\rho_{t-1}(h_{t-1}) = \sum_{h_t} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \rho_t(h_t), \tag{9.4.5}$$

$\rho_T(h_T) = 1$ olarak ilkleriz. Hem ileriye hem de geriye özyinelemeler T saklı değişkenlerini üstel zaman yerine bütün $(h_1, \ldots, h_T)$ değerler için $\mathcal{O}(kT)$ (doğrusal) zaman içinde toplamamızı sağlar. Bu, çizgesel modellerle olasılık çıkarımının en büyük avantajlarından biridir. Aynı zamanda genel mesaj geçişi algoritmasının çok özel bir örneğidir (Aji and McEliece, 2000). Hem ileriye hem de geri özyinelemeleri birleştirerek, şöyle bir hesaplamaya ulaşabiliriz:

$$P(x_j \mid x_{-j}) \propto \sum_{h_j} \pi_j(h_j) \rho_j(h_j) P(x_j \mid h_j). \tag{9.4.6}$$

Soyut terimlerle geriye özyinelemenin $\rho_{t-1} = g(\rho_t, x_t)$ olarak yazılabileceğini unutmayın; burada g öğrenilebilir bir işlevdir. Yine, bu aşırı şekilde bir güncelleme denklemi gibi görünüyor, sadece bizim RNN'de şimdiye kadar gördüğümüzün aksine geriye doğru çalışıyor. Gerçekten de, gizli Markov modelleri, mevcut olduğunda gelecekteki verileri bilmekten fayda sağlar. Sinyal işleme bilim insanları, gelecekteki gözlemleri bilme ve bilmemenin arasında aradeğerleme (interpolation) ve dışdeğerleme (extrapolation) diye ayrım yapar. Daha fazla ayrıntı için dizili Monte Carlo algoritmaları kitabının tanıtım bölümüne bakın (Doucet *et al.*, 2001).

9.4.2 Çift Yönlü Model

Eğer RNN'lerde gizli Markov modellerinde olana benzer bir ileri-bakış yeteneği sunan bir mekanizmaya sahip olmak istiyorsak, şimdiye kadar gördüğümüz RNN tasarımını değiştirmeliyiz. Neyse ki, bu kavramsal olarak kolaydır. Bir RNN'yi sadece ilk andıçtan başlayarak ileri modda çalıştırmak yerine, son andıçtan arkadan öne çalışan başka bir tane daha başlatırız. *Çift yönlü RNN*, bu tür bilgileri daha esnek bir şekilde işlemek için bilgileri geriye doğru ileten gizli bir katman ekler. Fig. 9.4.2, tek bir gizli katmanlı çift yönlü RNN mimarisini göstermektedir.

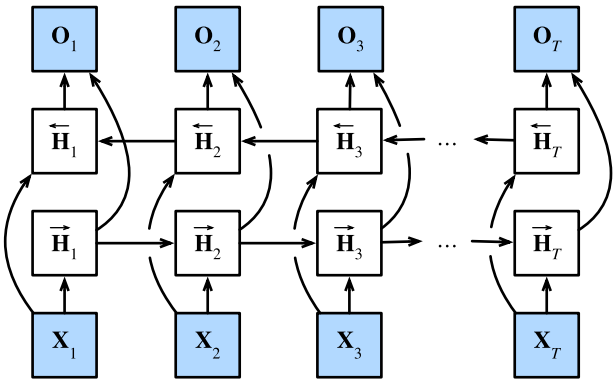


Fig. 9.4.2: Çift yönlü RNN mimarisi.

Aslında, bu, gizli Markov modellerinin dinamik programlanmasındaki ileriye ve geriye özyinelemelerden çok farklı değildir. Ana ayrım, önceki durumda bu denklemlerin belirli bir istatistiksel anlamı olmasıdır. Artık bu kadar kolay erişilebilir yorumlardan yoksunlar ve biz onlara genel ve öğrenilebilir işlevler olarak davranabiliriz. Bu geçiş, modern derin ağların tasarımına rehberlik eden ilkelerin çoğunu özetliyor: Önce, klasik istatistiksel modellerin fonksiyonel bağımlılıklarının türünü kullanın ve daha sonra bunları genel bir biçimde parameterize edin.

Tanım

Çift yönlü RNN'ler (Schuster and Paliwal, 1997) çalışmasında tanıtıldı. Çeşitli mimarilerin ayrıntılı bir tartışması için de (Graves and Schmidhuber, 2005) çalışmasında bakınız. Böyle bir ağın özelliklerine bakalım.

Herhangi bir t zaman adımı için, bir minigrup girdisi $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (örnek sayısı: n, her örnekteki girdi sayısı: d) verildiğinde gizli katman etkinleştirme işlevinin ϕ olduğunu varsayalım. Çift yönlü mimaride, bu zaman adımı için ileriye ve geriye doğru gizli durumların sırasıyla $\overrightarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ ve $\overleftarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ olduğunu varsayıyoruz, burada h gizli birimlerin sayısıdır. İleriye ve geriye doğru gizli durum güncelleştirmeleri aşağıdaki gibidir:

$$\begin{aligned}\overrightarrow{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(f)} + \overrightarrow{\mathbf{H}}_{t-1} \mathbf{W}_{hh}^{(f)} + \mathbf{b}_h^{(f)}),\\ \overleftarrow{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1} \mathbf{W}_{hh}^{(b)} + \mathbf{b}_h^{(b)}),\end{aligned} \tag{9.4.7}$$

burada $\mathbf{W}_{xh}^{(f)} \in \mathbb{R}^{d \times h}, \mathbf{W}_{hh}^{(f)} \in \mathbb{R}^{h \times h}, \mathbf{W}_{xh}^{(b)} \in \mathbb{R}^{d \times h}$, and $\mathbf{W}_{hh}^{(b)} \in \mathbb{R}^{h \times h}$ ağırlıkları ve $\mathbf{b}_h^{(f)} \in \mathbb{R}^{1 \times h}$ and $\mathbf{b}_h^{(b)} \in \mathbb{R}^{1 \times h}$ ek girdileri tüm model parametreleridir.

Daha sonra, $\mathbf{H}_t \in \mathbb{R}^{n \times 2h}$ gizli durumunu çıktı katmanına beslemek için $\overrightarrow{\mathbf{H}}_t$ ve $\overleftarrow{\mathbf{H}}_t$, ileriye ve geriye gizli durumlarını birleştiririz. Birden fazla gizli katman içeren derin çift yönlü RNN'lerde, bu tür bilgiler sonraki çift yönlü katmana *girdi* olarak aktarılır. Son olarak, çıktı katmanı $\mathbf{O}_t \in \mathbb{R}^{n \times q}$ çıktısını hesaplar (çıktı sayısı: q):

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q. \tag{9.4.8}$$

Burada, $\mathbf{W}_{hq} \in \mathbb{R}^{2h \times q}$ ağırlık matrisi ve $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ ek girdisi, çıktı katmanının model parametreleridir. Aslında, iki yön farklı sayıda gizli birime sahip olabilir.

Hesaplama Maliyeti ve Uygulamalar

Çift yönlü RNN'nin temel özelliklerinden biri, dizinin her iki ucundan gelen bilgilerin çıktıyı tahmin etmek için kullanıldığıdır. Yani, mevcut olanı tahmin etmek için hem gelecekteki hem de geçmişteki gözlemlerden gelen bilgileri kullanırız. Bir sonraki andıç tahmininde bu istediğimiz şey değildir. Sonuçta, sonraki andıcı tahmin ederken bir sonraki andıcı bilme lüksüne sahip değiliz. Dolayısıyla, çift yönlü bir RNN kullansaydık, çok iyi bir doğruluk elde edemezdik: Eğitim sırasında şimdiki zamanı tahmin etmek için geçmişteki ve gelecekteki verilere sahibiz. Test süresi boyunca ise elimizde sadece geçmişteki veri var ve dolayısıyla düşük doğruluğumuz olur. Bunu aşağıda bir deneyde göstereceğiz.

Yaraya tuz ekler gibi üstelik çift yönlü RNN'ler de son derece yavaştır. Bunun başlıca nedeni ileri yaymanın iki yönlü katmanlarda hem ileri hem de geri özyinelemeler gerektirmesi ve geri yaymanın ileri yayma sonuçlarına bağlı olmasıdır. Bu nedenle, gradyanlar çok uzun bir bağlılık zincirine sahip olacaktır.

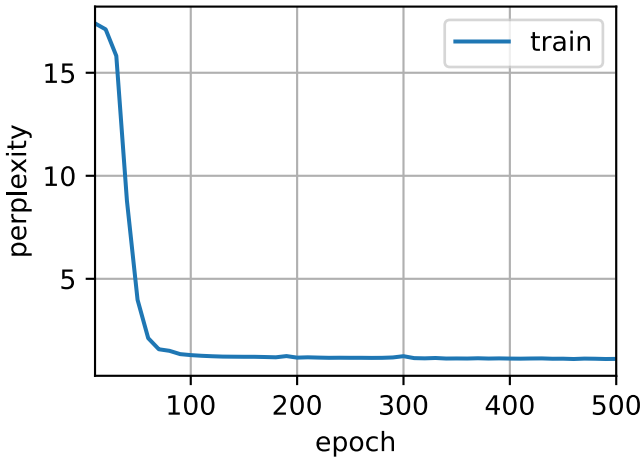
Pratikte çift yönlü katmanlar çok az kullanılır ve yalnızca eksik kelimeleri doldurma, andıçlara açıklama ekleme (örneğin, adlandırılmış nesne tanıma için) ve dizileri toptan kodlayarak diziyi veri işleme hattında bir adım işlemek gibi (örneğin, makine çevirisi için) dar bir uygulama kümesi için kullanılır. Section 14.7 ve Section 15.2 içinde, metin dizilerini kodlamak için çift yönlü RNN'lerin nasıl kullanılacağını tanıtacağız.

9.4.3 Yanlış Bir Uygulama İçin Çift Yönlü RNN Eğitmek

İki yönlü RNN'lerin geçmişteki ve gelecekteki verileri kullanmalarına gerçeğine ilişkin tüm ikazları görmezden gelirsek ve basitçe dil modellerine uygularsak, kabul edilebilir bir şaşkınlık değeriyle tahminler elde edebiliriz. Bununla birlikte, modelin gelecekteki andıçlarını tahmin etme yeteneği, aşağıdaki deneyin gösterdiği gibi ciddi bir şekilde zarar sokulmuştur. Makul şaşkınlığa rağmen, birçok yinelemeden sonra bile anlamsız ifadeler üretir. Aşağıdaki kodu, yanlış bağlamda kullanmaya karşı uyarıcı bir örnek olarak ekliyoruz.

```
import torch
from torch import nn
from d2l import torch as d2l

# Veriyi yükle
batch_size, num_steps, device = 32, 35, d2l.try_gpu()
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
# Çift yönlü LSTM modelini `bidirectional=True` olarak ayarlayarak tanımlayın.
vocab_size, num_hiddens, num_layers = len(vocab), 256, 2
num_inputs = vocab_size
lstm_layer = nn.LSTM(num_inputs, num_hiddens, num_layers, bidirectional=True)
model = d2l.RNNModel(lstm_layer, len(vocab))
model = model.to(device)
# Modeli eğitin
num_epochs, lr = 500, 1
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.1, 125783.5 tokens/sec on cuda:0
time travellererererererererererererererererererererererererer
travellererererererererererererererererererererererererer
```



Çıktı, yukarıda açıklanan nedenlerden dolayı bariz şekilde tatmin edici değildir. İki yönlü RNN'lerin daha etkili kullanımları hakkında bir tartışma için lütfen Section 15.2 içindeki duygusallık analizi uygulamasına bakın.

9.4.4 Özet

- İki yönlü RNN'lerde, her zaman adımı için gizli durum, şu anki zaman adımından önceki ve sonraki veriler tarafından eş zamanlı olarak belirlenir.
- Çift yönlü RNN'ler olasılıksal çizge modellerdeki ileri-geri algoritması ile çarpıcı bir benzerlik taşır.
- Çift yönlü RNN'ler çoğunlukla dizi kodlaması ve çift yönlü bağlam verilen gözlemlerin tahmini için yararlıdır.
- Çift yönlü RNN'lerin uzun gradyan zincirleri nedeniyle eğitilmesi maliyetlidir.

9.4.5 Alıştırmalar

1. Farklı yönler farklı sayıda gizli birim kullanıyorsa, $\mathbf{H}_t$'nin şekli nasıl değişecektir?
2. Birden fazla gizli katmanlı çift yönlü bir RNN tasarlayın.
3. Çok anlamlılık doğal dillerde yaygındır. Örneğin, "banka" kelimesinin "nakit yatırmak için bankaya gittim" ve "oturmak için banka doğru gittim" bağlamlarında farklı anlamları vardır. Bir bağlam dizisi ve bir kelime, bağlamda kelimenin vektör temsilini döndüren bir sinir ağı modelini nasıl tasarlayabiliriz? Çok anlamlılığın üstesinden gelmek için hangi tür sinir mimarileri tercih edilir?

Tartışmalar[114]

[114] https://discuss.d2l.ai/t/1059

9.5 Makine Çevirisi ve Veri Kümesi

Doğal dil işlemenin anahtarı olan dil modellerini tasarlamak için RNN kullandık. Diğer bir amiral gemisi kıyaslaması, girdi dizilerini çıktı dizilerine dönüştüren *dizi dönüştürme* modelleri için merkezi bir problem düzlemi olan *makine çevirisi*dir. Çeşitli modern yapay zeka uygulamalarında önemli bir rol oynayan dizi dönüştürme modelleri, bu bölümün geri kalanının ve sonraki bölümün, Chapter 10, odağını oluşturacaktır. Bu amaçla, bu bölüm makine çevirisi sorununu ve daha sonra kullanılacak veri kümesini anlatır.

Makine çevirisi bir dizinin bir dilden diğerine otomatik çevirisidir. Aslında, bu alan, özellikle II. Dünya Savaşı'nda dil kodlarını kırmak için bilgisayarların kullanılması göz önüne alınarak, sayısal bilgisayarların icat edilmesinin kısa bir süre sonrasından 1940'lara kadar uzanabilir. Onlarca yıldır, bu alanda, istatistiksel yaklaşımlar, (Brown *et al.*, 1990, Brown *et al.*, 1988), sinir ağlarını kullanarak uçtan uca öğrenmenin yükselmesinin öncesine kadar baskın olmuştur. İkincisi, çeviri modeli ve dil modeli gibi bileşenlerde istatistiksel analiz içeren *istatistiksel makine çevirisinden* ayırt edilmesi için genellikle *sinirsel makine çevirisi* olarak adlandırılır.

Uçtan uca öğrenmeyi vurgulayan bu kitap, sinirsel makine çevirisi yöntemlerine odaklanacaktır. Külliyatı tek bir dil olan Section 8.3 içindeki dil modeli problemimizden farklı olarak, makine çevirisi veri kümeleri sırasıyla kaynak dilde ve hedef dilde bulunan metin dizileri çiftlerinden oluşmaktadır. Bu nedenle, dil modelleme için ön işleme rutinini yeniden kullanmak yerine, makine çevirisi veri kümelerini ön işlemek için farklı bir yol gerekir. Aşağıda, önceden işlenmiş verilerin eğitim için minigruplara nasıl yükleneceğini gösteriyoruz.

```
import os
import torch
from d2l import torch as d2l
```

9.5.1 Veri Kümesini İndirme ve Ön işleme

Başlangıç olarak, Tatoeba Projesi'nden iki dilli cümle çiftleri[115]'nden oluşan bir İngiliz-Fransız veri kümesini indiriyoruz. Veri kümedeki her satır, bir sekmeyle ayrılmış İngilizce metin dizisi ve çevrilmiş Fransızca metin dizisi çiftidir. Her metin dizisinin sadece bir cümle veya birden çok cümleden oluşan bir paragraf olabileceğini unutmayın. İngilizce'nin Fransızca'ya çevrildiği bu makine çevirisi probleminde, İngilizce *kaynak dil*, Fransızca ise *hedef dil*dir.

```
#@save
d2l.DATA_HUB['fra-eng'] = (d2l.DATA_URL + 'fra-eng.zip',
                           '94646ad1522d915e7b0f9296181140edcf86a4f5')

#@save
def read_data_nmt():
    """İnglizce-Fransızca veri kümesini yükle."""
    data_dir = d2l.download_extract('fra-eng')
    with open(os.path.join(data_dir, 'fra.txt'), 'r') as f:
        return f.read()

raw_text = read_data_nmt()
print(raw_text[:75])
```

[115] http://www.manythings.org/anki/

```
Go. Va !
Hi. Salut !
Run!        Cours !
Run!        Courez !
Who?        Qui ?
Wow!        Ça alors !
```

Veri kümesini indirdikten sonra, ham metin verileri için birkaç ön işleme adımı ile devam ediyoruz. Örneğin, aralıksız boşluğu boşlukla değiştirir, büyük harfleri küçük harflere dönüştürür ve sözcüklerle noktalama işaretleri arasına boşluk ekleriz.

```
#@save
def preprocess_nmt(text):
    """İngilizce-Fransızca veri kümesini ön işle."""
    def no_space(char, prev_char):
        return char in set(',.!?') and prev_char != ' '

    # Bölünmez boşluğu boşlukla değiştirin ve büyük harfleri küçük
    # harflere dönüştürün
    text = text.replace('\u202f', ' ').replace('\xa0', ' ').lower()
    # Sözcükler ve noktalama işaretleri arasına boşluk ekleyin
    out = [' ' + char if i > 0 and no_space(char, text[i - 1]) else char
           for i, char in enumerate(text)]
    return ''.join(out)

text = preprocess_nmt(raw_text)
print(text[:80])
```

```
go .        va !
hi .        salut !
run !       cours !
run !       courez !
who ?       qui ?
wow !       ça alors !
```

9.5.2 Andıçlama

Section 8.3 içindeki karakter düzeyinde andıçlara ayırmaktan farklı olarak, makine çevirisi için burada kelime düzeyinde andıçlamayı tercih ediyoruz (son teknoloji modeller daha gelişmiş andıçlama teknikleri kullanabilir). Aşağıdaki `tokenize_nmt` işlevi, her andıç bir sözcük veya noktalama işareti olduğu ilk `num_examples` tane metin dizisi çiftini andıçlar. Bu işlev, andıç listelerinden oluşan iki liste döndürür: `source` (kaynak) ve `target` (hedef). Özellikle, `source[i]` kaynak dilde (İngilizce burada) i. metin dizisinden andıçların bir listesidir ve `target[i]` hedef dildekileri (Fransızca burada) içerir.

```
#@save
def tokenize_nmt(text, num_examples=None):
    """İngilizce-Fransızca veri kümesini andıçla."""
    source, target = [], []
    for i, line in enumerate(text.split('\n')):
        if num_examples and i > num_examples:
```

(continues on next page)

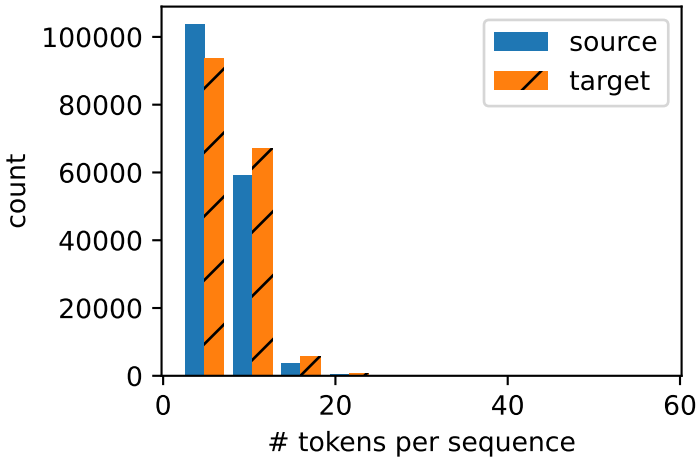
(continued from previous page)

```
            break
        parts = line.split('\t')
        if len(parts) == 2:
            source.append(parts[0].split(' '))
            target.append(parts[1].split(' '))
    return source, target

source, target = tokenize_nmt(text)
source[:6], target[:6]
```

```
([['go', '.'],
  ['hi', '.'],
  ['run', '!'],
  ['run', '!'],
  ['who', '?'],
  ['wow', '!']],
 [['va', '!'],
  ['salut', '!'],
  ['cours', '!'],
  ['courez', '!'],
  ['qui', '?'],
  ['ça', 'alors', '!']])
```

Metin dizisi başına andıç sayısının histogramını çizelim. Bu basit İngilizce-Fransız veri kümesinde, metin dizilerinin çoğunun 20'den az andıcı vardır.

```
#@save
def show_list_len_pair_hist(legend, xlabel, ylabel, xlist, ylist):
    """Liste uzunluğu çiftleri için histogramı çizin."""
    d2l.set_figsize()
    _, _, patches = d2l.plt.hist(
        [[len(l) for l in xlist], [len(l) for l in ylist]])
    d2l.plt.xlabel(xlabel)
    d2l.plt.ylabel(ylabel)
    for patch in patches[1].patches:
        patch.set_hatch('/')
    d2l.plt.legend(legend)

show_list_len_pair_hist(['source', 'target'], '# tokens per sequence',
                        'count', source, target);
```



9.5.3 Kelime Dağarcığı

Makine çeviri veri kümesi dil çiftlerinden oluştuğundan, hem kaynak dil hem de hedef dil için ayrı ayrı iki kelime hazinesi oluşturabiliriz. Kelime düzeyinde andıçlamada, kelime dağarcığı boyutu, karakter düzeyinde andıç kullanandan önemli ölçüde daha büyük olacaktır. Bunu hafifletmek için, burada 2 defadan az görünen seyrek andıçları aynı bilinmeyen (“<unk>”) andıcı ile ifade ediyoruz. Bunun yanı sıra, minigruplarda dizileri aynı uzunlukta dolgulamak için (“<pad>”) ve dizilerin başlangıcını işaretlemek için (“<bos>”) veya sonunu işaretlemek için (“<eos>”) gibi ek özel andıçlar belirtiyoruz. Bu tür özel andıçlar, doğal dil işleme görevlerinde yaygın olarak kullanılır.

```
src_vocab = d2l.Vocab(source, min_freq=2,
                      reserved_tokens=['<pad>', '<bos>', '<eos>'])
len(src_vocab)
```

```
10012
```

9.5.4 Veri Kümesini Okuma

Dil modellemesinde, her dizi örneğinin, bir cümlenin bir kesimine veya birden fazla cümle üzerine bir yayılan sabit bir uzunluğa sahip olduğunu hatırlayın. Bu Section 8.3 içindeki `num_steps` (zaman adımları veya andıç sayısı) bağımsız değişkeni tarafından belirtilmiştir. Makine çevirisinde, her örnek, her metin dizisinin farklı uzunluklara sahip olabileceği bir kaynak ve hedef metin dizisi çiftidir.

Hesaplamada verimlilik için, yine de bir minigrup metin dizisini *kırkma (truncation)* ve *dolgu* ile işleyebiliriz. Aynı minigruptaki her dizinin aynı `num_steps` uzunluğunda olması gerektiğini varsayalım. Bir metin dizisi `num_steps` andıçtan daha azsa, uzunluğu `num_steps`'e ulaşana kadar özel “<pad>” andıcını sonuna eklemeye devam edeceğiz. Aksi takdirde, metin sırasını yalnızca ilk `num_steps` andıcını alıp geri kalanını atarak keseceğiz. Bu şekilde, her metin dizisi aynı şekle sahip minigruplar olarak yüklenebileceği aynı uzunluğa sahip olacaktır.

Aşağıdaki `truncate_pad` işlevi metin dizilerini daha önce açıklandığı gibi keser veya dolgular.

```
#@save
def truncate_pad(line, num_steps, padding_token):
    """Dizileri dolgula veya kırk"""
    if len(line) > num_steps:
        return line[:num_steps]  # Truncate
    return line + [padding_token] * (num_steps - len(line))  # Pad

truncate_pad(src_vocab[source[0]], 10, src_vocab['<pad>'])
```

```
[47, 4, 1, 1, 1, 1, 1, 1, 1, 1]
```

Şimdi, metin dizilerini eğitimde minigruplara dönüştürmek için bir işlev tanımlıyoruz. Dizinin sonunu belirtmek için her dizinin sonuna özel "<eos>" andıcını ekliyoruz. Bir model bir diziyi her andıç sonrası bir andıç oluşturarak tahmin ettiğinde, modelin "<eos>" andıcını oluşturması çıktı dizisini tamamlandığını ifade edebilir. Ayrıca, dolgu andıçlarını hariç tutarak her metin dizisinin uzunluğunu da kaydediyoruz. Bu bilgi, daha sonra ele alacağımız bazı modellerde gerekli olacaktır.

```
#@save
def build_array_nmt(lines, vocab, num_steps):
    """Makine çevirisinin metin dizilerini minigruplara dönüştürün."""
    lines = [vocab[l] for l in lines]
    lines = [l + [vocab['<eos>']] for l in lines]
    array = torch.tensor([truncate_pad(
        l, num_steps, vocab['<pad>']) for l in lines])
    valid_len = (array != vocab['<pad>']).type(torch.int32).sum(1)
    return array, valid_len
```

9.5.5 Her Şeyi Bir Araya Koyma

Son olarak, veri yineleyiciyi hem kaynak dil hem de hedef dil için kelime dağarcıkları ile birlikte döndüren `load_data_nmt` işlevini tanımlıyoruz.

```
#@save
def load_data_nmt(batch_size, num_steps, num_examples=600):
    """Çeviri veri kümesinin yineleyicisini ve sözcük dağarcıklarını döndür."""
    text = preprocess_nmt(read_data_nmt())
    source, target = tokenize_nmt(text, num_examples)
    src_vocab = d2l.Vocab(source, min_freq=2,
                          reserved_tokens=['<pad>', '<bos>', '<eos>'])
    tgt_vocab = d2l.Vocab(target, min_freq=2,
                          reserved_tokens=['<pad>', '<bos>', '<eos>'])
    src_array, src_valid_len = build_array_nmt(source, src_vocab, num_steps)
    tgt_array, tgt_valid_len = build_array_nmt(target, tgt_vocab, num_steps)
    data_arrays = (src_array, src_valid_len, tgt_array, tgt_valid_len)
    data_iter = d2l.load_array(data_arrays, batch_size)
    return data_iter, src_vocab, tgt_vocab
```

İngilizce-Fransız veri kümesinden ilk minigrubu okuyalım.

```
train_iter, src_vocab, tgt_vocab = load_data_nmt(batch_size=2, num_steps=8)
for X, X_valid_len, Y, Y_valid_len in train_iter:
    print('X:', X.type(torch.int32))
    print('valid lengths for X:', X_valid_len)
    print('Y:', Y.type(torch.int32))
    print('valid lengths for Y:', Y_valid_len)
    break
```

```
X: tensor([[ 9, 21,  4,  3,  1,  1,  1,  1],
        [ 7, 59,  4,  3,  1,  1,  1,  1]], dtype=torch.int32)
valid lengths for X: tensor([4, 4])
Y: tensor([[97,  4,  3,  1,  1,  1,  1,  1],
        [38,  7,  0,  4,  3,  1,  1,  1]], dtype=torch.int32)
valid lengths for Y: tensor([3, 5])
```

9.5.6 Özet

- Makine çevirisi, bir dizinin bir dilden diğerine otomatik çevirisini ifade eder.
- Kelime düzeyinde andıçlama kullanırsak, kelime dağarcığının boyutu, karakter düzeyinde andıçlama kullanmaya göre önemli ölçüde daha büyük olacaktır. Bunu hafifletmek için, seyrek kullanılan andıçları aynı bilinmeyen andıç olarak ifade alabiliriz.
- Metin dizilerini kırkabilir ve dolgulayabiliriz, böylece hepsi minigruplarda yüklenirken aynı uzunluğa sahip olurlar.

9.5.7 Alıştırmalar

1. `load_data_nmt` işlevindeki `num_examples` değişkeninin farklı değerlerini deneyin. Bu, kaynak ve hedef dillerin kelime dağarcığı boyutlarını nasıl etkiler?
2. Çince ve Japonca gibi bazı dillerde metin, kelime sınır göstergelerine (örn., boşluk) sahip değildir. Sözcük düzeyinde andıçlama bu gibi durumlar için hala iyi bir fikir midir? Neden?

Tartışmalar[116]

9.6 Kodlayıcı-Kodçözücü Mimarisi

Section 9.5 içinde tartıştığımız gibi, makine çevirisi, girdinin ve çıktının ikisinin de değişken uzunlukta diziler olduğu dizi dönüştürme modelleri için önemli bir problem sahasıdır. Bu tür girdi ve çıktıları işlemek için iki ana bileşenli bir mimari tasarlayabiliriz. İlk bileşen bir *kodlayıcı*dır (encoder): Girdi olarak değişken uzunlukta bir diziyi alır ve sabit bir şekle sahip bir duruma dönüştürür. İkinci bileşen bir *kodçözücü*dür (decoder): Sabit şekilli bir kodlanmış durumu değişken uzunlukta bir diziye eşler. Bu, Fig. 9.6.1 şeklinde tasvir edildiği gibi *kodlayıcı-kodçözücü* mimarisi olarak adlandırılır.

[116] https://discuss.d2l.ai/t/1060

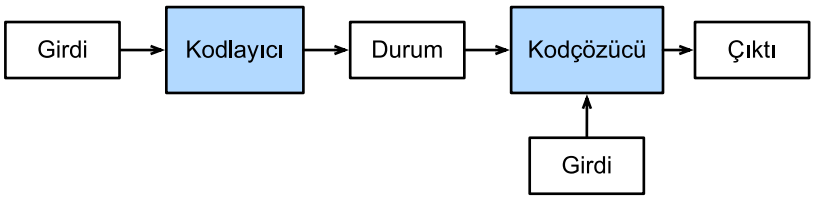


Fig. 9.6.1: Kodlayıcı-kodçözücü mimarisi.

Örnek olarak İngilizce'den Fransızca'ya makine çevirisini ele alalım. İngilizce bir girdi dizisi göz önüne alındığımızda, örneğin "They", "are", "watching","."", (Onlar izliyorlar.) gibi, kodlayıcı-kodçözücü mimarisi önce değişken uzunluklu girdiyi bir duruma kodlar, sonra da durumu her seferinde bir andıç işleyerek çevrilmiş dizi çıktısını oluşturmak üzere çözer: "Ils", "regardent", ".". Kodlayıcı-kodçözücü mimarisi, sonraki bölümlerdeki farklı dizi dönüştürme modellerinin temelini oluşturduğundan, bu bölüm bu mimariyi daha sonra uygulanacak bir arayüze dönüştürecektir.

9.6.1 Kodlayıcı

Kodlayıcı arayüzünde, kodlayıcının `X` girdisi olarak değişken uzunlukta dizileri aldığını belirtiyoruz. Uygulaması, bu temel `Encoder` sınıfından kalıtımla türetilmiş model tarafından sağlanacaktır.

```
from torch import nn


#@save
class Encoder(nn.Module):
    """Kodlayıcı-kod çözücü mimarisi için temel kodlayıcı arabirimi."""
    def __init__(self, **kwargs):
        super(Encoder, self).__init__(**kwargs)

    def forward(self, X, *args):
        raise NotImplementedError
```

9.6.2 Kodçözücü

Aşağıdaki kodçözücü arayüzünde, kodlayıcı çıktısını (`enc_outputs`) kodlanmış duruma dönüştürmek için ek bir `init_state` işlevi ekliyoruz. Bu adımın Section 9.5.4 içinde açıklanan girdinin geçerli uzunluğu gibi ek girdiler gerektirebileceğini unutmayın. Her seferinde bir andıç yaratarak değişken uzunlukta bir dizi oluşturmak için, her zaman kodçözücü bir girdiyi (örneğin, önceki zaman adımında oluşturulan andıç) ve kodlanmış durumu şu anki zaman adımındaki bir çıktı andıcına eşler.

```
#@save
class Decoder(nn.Module):
    """Kodlayıcı-kod çözücü mimarisi için temel kodçözücü arabirimi."""
    def __init__(self, **kwargs):
        super(Decoder, self).__init__(**kwargs)
```

(continues on next page)

(continued from previous page)

```
    def init_state(self, enc_outputs, *args):
        raise NotImplementedError

    def forward(self, X, state):
        raise NotImplementedError
```

9.6.3 Kodlayıcıyı ve Kodçözücüyü Bir Araya Koyma

Sonunda, kodlayıcı-kodçözücü mimarisi, isteğe bağlı olarak ekstra argümanlarla beraber, hem bir kodlayıcı hem de bir kodçözücü içeriyor. İleri yaymada, kodlayıcının çıktısı kodlanmış durumu üretmek için kullanılır ve bu durum daha sonrasında kodçözücü tarafından girdilerinden biri olarak kullanılacaktır.

```
#@save
class EncoderDecoder(nn.Module):
    """Kodlayıcı-kod çözücü mimarisi için temel sınıf."""
    def __init__(self, encoder, decoder, **kwargs):
        super(EncoderDecoder, self).__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder

    def forward(self, enc_X, dec_X, *args):
        enc_outputs = self.encoder(enc_X, *args)
        dec_state = self.decoder.init_state(enc_outputs, *args)
        return self.decoder(dec_X, dec_state)
```

Kodlayıcı-kodçözücü mimarisindeki "durum" terimi, muhtemelen durumlara sahip sinir ağlarını kullanarak bu mimariyi uygulamak için size ilham vermiştir. Bir sonraki bölümde, bu kodlayıcı-kodçözücü mimarisine dayanan dizi dönüştürme modellerini tasarlamak için RNN'lerin nasıl uygulanacağını göreceğiz.

9.6.4 Özet

- Kodlayıcı-kodçözücü mimarisi, hem değişken uzunlukta diziler olan girdi ve çıktıları işleyebilir, bu nedenle makine çevirisi gibi dizi dönüştürme problemleri için uygundur.
- Kodlayıcı, girdi olarak değişken uzunlukta bir diziyi alır ve sabit bir şekle sahip bir duruma dönüştürür.
- Kodçözücü, sabit şekilli bir kodlanmış durumu değişken uzunlukta bir diziye eşler.

9.6.5 Alıştırmalar

1. Kodlayıcı-kodçözücü mimarisini uygulamak için sinir ağları kullandığımızı varsayalım. Kodlayıcı ve kodçözücü aynı tür sinir ağı olmak zorunda mıdır?
2. Makine çevirisinin yanı sıra, kodlayıcı-kodçözücü mimarisinin uygulanabileceği başka bir uygulama düşünebiliyor musunuz?

Tartışmalar[117]

9.7 Diziden Diziye Öğrenme

Section 9.5 içinde gördüğümüz gibi, makine çevirisinde hem girdi hem de çıktı değişken uzunlukta dizilerdir. Bu tür problemleri çözmek için Section 9.6 içinde genel bir kodlayıcı-kodçözücü mimarisi tasarladık. Bu bölümde, bu mimarinin kodlayıcısını ve kodçözücüsünü tasarlamak için iki RNN kullanacağız ve makine çevirisi (Cho *et al.*, 2014, Sutskever *et al.*, 2014) için *diziden diziye* öğrenmeyi uygulayacağız.

Kodlayıcı-kodçözücü mimarisinin tasarım ilkesini takiben, RNN kodlayıcı girdi olarak değişken uzunlukta bir diziyi alabilir ve bir sabit şekilli gizli duruma dönüştürebilir. Başka bir deyişle, girdi (kaynak) dizisinin bilgileri RNN kodlayıcısının gizli durumunda *kodlanmış* olur. Çıktı dizisini andıç andıç oluşturmak için, ayrı bir RNN kodçözücü, girdi dizisinin kodlanmış bilgileriyle birlikte, hangi andıçların görüldüğünü (dil modellemesinde olduğu gibi) veya oluşturulduğuna bağlı olarak bir sonraki andıcı tahmin edebilir. Fig. 9.7.1, makine çevirisinde diziden diziye öğrenme için iki RNN'nin nasıl kullanılacağını gösterir.

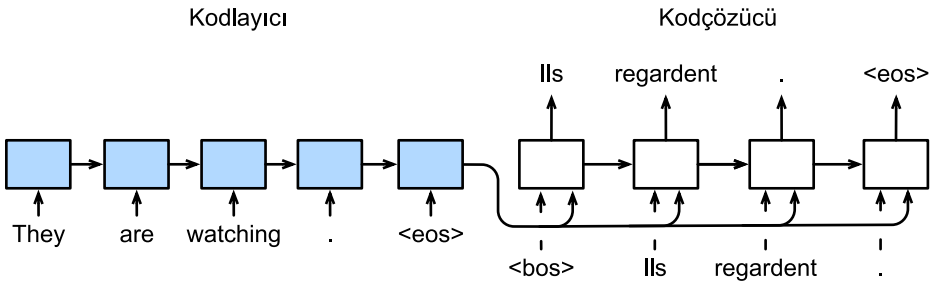


Fig. 9.7.1: Bir RNN kodlayıcı ve bir RNN kodçözücü ile diziden diziye öğrenme.

Fig. 9.7.1 şeklinde, özel "<eos>" andıcı dizinin sonunu işaretler. Model, bu andıç oluşturulduktan sonra tahminlerde bulunmayı durdurabilir. RNN kodçözücüsünün ilk zaman adımında, iki özel tasarım kararı vardır. İlk olarak, özel dizi başlangıç andıcı, "<bos>", bir girdidir. İkincisi, RNN kodlayıcısının son gizli durumu, kodçözücünün gizli durumunu ilklemek için kullanılır. (Sutskever *et al.*, 2014) çalışmasındaki gibi tasarımlarda, kodlanmış girdi dizisi bilgilerinin çıktı (hedef) dizisini oluşturmak için kodçözücüsüne beslenmesi tam olarak da budur. (Cho *et al.*, 2014) gibi diğer bazı tasarımlarda, kodlayıcının son gizli durumu, Fig. 9.7.1 şeklinde gösterildiği gibi her adımda girdilerin bir parçası olarak kodçözücüye beslenir. Section 8.3 içindeki dil modellerinin eğitimine benzer şekilde, etiketlerin bir andıç ile kaydırılmış orijinal çıktı dizisi olmasına izin verebiliriz: "<bos>", "Ils", "regardent", "." $\rightarrow$ "Ils", "regardent","."<eos>".

Aşağıda, Fig. 9.7.1 tasarımını daha ayrıntılı olarak açıklayacağız. Bu modeli Section 9.5 içinde tanıtılan İngilizce-Fransız veri kümesinden makine çevirisi için eğiteceğiz.

[117] https://discuss.d2l.ai/t/1061

```
import collections
import math
import torch
from torch import nn
from d2l import torch as d2l
```

9.7.1 Kodlayıcı

Teknik olarak konuşursak, kodlayıcı değişken uzunluktaki bir girdi dizisini sabit şekilli *bağlam değişkeni* **c**'ye dönüştürür ve bu bağlam değişkeninde girdi dizisinin bilgilerini kodlar. Fig. 9.7.1 şeklinde gösterildiği gibi, kodlayıcıyı tasarlamak için bir RNN kullanabiliriz.

Bir dizi örneği düşünelim (toplu küme boyutu: 1). Girdi dizimizin $x_1, \ldots, x_T$ olduğunu varsayalım, öyle ki x_t girdi metin dizisindeki t. andıç olsun. t zaman adımında, RNN x_t için girdi öznitelik vektörü $\mathbf{x}_t$'yi ve önceki zaman adımından gizli durum $\mathbf{h}_{t-1}$'yi şu anki gizli durum $\mathbf{h}_t$'ye dönüştürür. RNN'nin yinelemeli tabakasının dönüşümünü ifade etmek için f işlevini kullanabiliriz:

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}). \tag{9.7.1}$$

Genel olarak, kodlayıcı, gizli durumları her zaman adımında özelleştirilmiş bir q işlevi aracılığıyla bağlam değişkenine dönüştürür:

$$\mathbf{c} = q(\mathbf{h}_1, \ldots, \mathbf{h}_T). \tag{9.7.2}$$

Örneğin, Fig. 9.7.1 şeklinde olduğu gibi $q(\mathbf{h}_1, \ldots, \mathbf{h}_T) = \mathbf{h}_T$'yi seçerken, bağlam değişkeni yalnızca son zaman adımındaki girdi dizisinin gizli durumu $\mathbf{h}_T$'dir.

Şimdiye kadar kodlayıcıyı tasarlamak için tek yönlü bir RNN kullandık, burada gizli bir durum yalnızca gizli durumun önceki ve o anki zaman adımındaki girdi altdizisine bağlıdır. Ayrıca çift yönlü RNN'leri kullanarak kodlayıcılar da oluşturabiliriz. Bu durumda, tüm dizinin bilgilerini kodlayan gizli durum, zaman adımından önceki ve sonraki altdiziye (geçerli zaman adımındaki girdi dahil) bağlıdır.

Şimdi RNN kodlayıcısını uygulamaya başlayalım. Girdi dizisindeki her andıç için öznitelik vektörünü elde ederken bir *gömme katmanı* kullandığımıza dikkat edin. Bir gömme katmanın ağırlığı, satır sayısı girdi kelime dağarcığının boyutuna (`vocab_size`) ve sütun sayısı öznitelik vektörünün boyutuna eşit olan bir matristir (`embed_size`). Herhangi bir girdi andıcı dizini i için gömme katmanı, öznitelik vektörünü döndürmek üzere ağırlık matrisinin i. satırını (0'dan başlayarak) getirir. Ayrıca, burada kodlayıcıyı uygulamak için çok katmanlı bir GRU seçiyoruz.

```
#@save
class Seq2SeqEncoder(d2l.Encoder):
    """Diziden diziye öğrenme için RNN kodlayıcı."""
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0, **kwargs):
        super(Seq2SeqEncoder, self).__init__(**kwargs)
        # Gömme katmanı
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = nn.GRU(embed_size, num_hiddens, num_layers,
                          dropout=dropout)

    def forward(self, X, *args):
```

(continues on next page)

(continued from previous page)

```
        # `X` çıktısının şekli: (`batch_size`, `num_steps`, `embed_size`)
        X = self.embedding(X)
        # RNN modellerinde ilk eksen zaman adımlarına karşılık gelir
        X = X.permute(1, 0, 2)
        # Durumdan bahsedilmediğinde varsayılan olarak sıfırdır.
        output, state = self.rnn(X)
        # `output` (çıktı) şekli: (`num_steps`, `batch_size`, `num_hiddens`)
        # `state[0]` (durum) şekli: (`num_layers`, `batch_size`, `num_hiddens`)
        return output, state
```

Yinelemeli katmanların döndürülen değişkenleri Section 8.6 içinde açıklanmıştı. Yukarıdaki kodlayıcı uygulamasını göstermek için somut bir örnek kullanalım. Aşağıda, gizli birimlerin sayısı 16 olan iki katmanlı bir GRU kodlayıcısı oluşturuyoruz. `X` dizi girdilerinin bir minigrubu göz önüne alındığında (grup boyutu: 4, zaman adımı sayısı: 7), son katmanın gizli durumları (kodlayıcının yinelemeli katmanları tarafından döndürülen `output`) şekli (zaman adımlarının sayısı, grup boyutu, gizli birimlerin sayısı) olan tensörlerdir.

```
encoder = Seq2SeqEncoder(vocab_size=10, embed_size=8, num_hiddens=16,
                         num_layers=2)
encoder.eval()
X = torch.zeros((4, 7), dtype=torch.long)
output, state = encoder(X)
output.shape
```

```
torch.Size([7, 4, 16])
```

Burada bir GRU kullanıldığından, son zaman adımındaki çok katmanlı gizli durumlar (gizli katmanların sayısı, grup boyutu, gizli birim sayısı) şeklindedir. Bir LSTM kullanılıyorsa, bellek hücresi bilgileri de `state`'te yer alır.

```
state.shape
```

```
torch.Size([2, 4, 16])
```

9.7.2 Kodçözücü

Az önce de belirttiğimiz gibi, kodlayıcının çıktısının $\mathbf{c}$ bağlam değişkeni $x_1, \ldots, x_T$ tüm girdi dizisini kodlar. Eğitim veri kümesinden $y_1, y_2, \ldots, y_{T'}$ çıktı dizisi göz önüne alındığında, her zaman adım t' için (sembol, girdi dizilerinin veya kodlayıcıların t zaman adımından farklıdır), kodçözücü çıktısının olasılığı $y_{t'}$, önceki çıktı altdizisi $y_1, \ldots, y_{t'-1}$ ve $\mathbf{c}$ bağlam değişkeni üzerinde koşulludur, yani, $P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \mathbf{c})$.

Bu koşullu olasılığı diziler üzerinde modellemek için, kodçözücü olarak başka bir RNN kullanabiliriz. Herhangi bir t' zaman adımındaki çıktı dizisinde, RNN önceki zaman adımından $y_{t'-1}$ çıktı dizisini ve $\mathbf{c}$ bağlam değişkenini girdi olarak alır, sonra onları ve önceki gizli durumu $\mathbf{s}_{t'-1}$ ile beraber şu anki zaman adımındaki gizli durum $\mathbf{s}_{t'}$'ye dönüştürür. Sonuç olarak, kodçözücünün gizli katmanının dönüşümünü ifade etmek için g işlevini kullanabiliriz:

$$\mathbf{s}_{t'} = g(y_{t'-1}, \mathbf{c}, \mathbf{s}_{t'-1}). \tag{9.7.3}$$

Kodçözücünün gizli durumunu elde ettikten sonra, t' adımındaki çıktı için koşullu olasılık dağılımını, $P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \mathbf{c})$'yi hesaplamak için bir çıktı katmanını ve softmaks işlemini kullanabiliriz.

Fig. 9.7.1 şeklini takiben, kodçözücüyü aşağıdaki gibi uygularken, kodçözücünün gizli durumunu ilklemek için kodlayıcının son zaman adımındaki gizli durumu doğrudan kullanırız. Bu, RNN kodlayıcı ve RNN kodçözücüsünün aynı sayıda katman ve gizli birimlere sahip olmasını gerektirir. Kodlanmış girdi dizisi bilgilerini daha da dahil etmek için, bağlam değişkeni kodçözücü girdisiyle her zaman adımda bitiştirilir. Çıktı andıcının olasılık dağılımını tahmin etmek için, RNN kodçözücüsünün son katmanındaki gizli durumunu dönüştüren tam bağlı bir katman kullanılır.

```
class Seq2SeqDecoder(d2l.Decoder):
    """Diziden diziye öğrenme için RNN kodçözücüsü."""
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0, **kwargs):
        super(Seq2SeqDecoder, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = nn.GRU(embed_size + num_hiddens, num_hiddens, num_layers,
                          dropout=dropout)
        self.dense = nn.Linear(num_hiddens, vocab_size)

    def init_state(self, enc_outputs, *args):
        return enc_outputs[1]

    def forward(self, X, state):
        # `X` çıktısı şekli: (`num_steps`, `batch_size`, `embed_size`)
        X = self.embedding(X).permute(1, 0, 2)
        # `context`'ı yayınlayın, böylece `X` ile aynı `num_steps` değerine
        # sahip olur
        context = state[-1].repeat(X.shape[0], 1, 1)
        X_and_context = torch.cat((X, context), 2)
        output, state = self.rnn(X_and_context, state)
        output = self.dense(output).permute(1, 0, 2)
        # `output` (çıktı) şekli: (`batch_size`, `num_steps`, `vocab_size`)
        # `state` şekli: (`num_layers`, `batch_size`, `num_hiddens`)
        return output, state
```

Uygulanan kodçözücüyü göstermek için, aşağıda belirtilen kodlayıcıdan aynı hiper parametrelerle ilkliyoruz. Gördüğümüz gibi, kodçözücünün çıktı şekli (küme boyutu, zaman adımlarının sayısı, kelime dağarcığı boyutu) olur, burada tensörün son boyutu tahmin edilen andıç dağılımını tutar.

```
decoder = Seq2SeqDecoder(vocab_size=10, embed_size=8, num_hiddens=16,
                         num_layers=2)
decoder.eval()
state = decoder.init_state(encoder(X))
output, state = decoder(X, state)
output.shape, state.shape
```

```
(torch.Size([4, 7, 10]), torch.Size([2, 4, 16]))
```

Özetlemek gerekirse, yukarıdaki RNN kodlayıcı-kodçözücü modelindeki katmanlar Fig. 9.7.2 içinde gösterilmektedir.

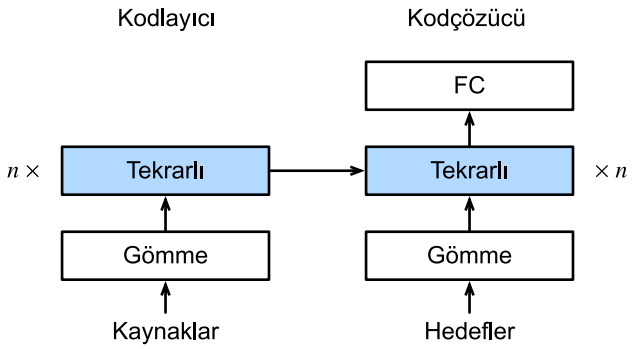


Fig. 9.7.2: Bir RNN kodlayıcı-kodçözücü modelindeki katmanlar.

9.7.3 Kayıp Fonksiyonu

Her adımda, kodçözücü çıktı andıçları için bir olasılık dağılımı öngörür. Dil modellemesine benzer şekilde, dağılımı elde etmek ve eniyilemek için çapraz entropi kaybını hesaplarken softmaks uygulayabiliriz. Section 9.5 içindeki özel dolgu andıçlarının dizilerin sonuna eklendiğini hatırlayın, böylece değişen uzunluklardaki dizilerin aynı şekildeki minigruplara verimli bir şekilde yüklenebilmesini sağlanır. Bununla birlikte, dolgu andıçlarının tahminlenmesi kayıp hesaplamalarında harici tutulmalıdır.

Bu amaçla, alakasız girdileri sıfır değerleriyle maskelemek için aşağıdaki `sequence_mask` işlevini kullanabiliriz, böylece daha sonra alakasız tahminlerin sıfır ile çarpımı sıfıra eşit olur. Örneğin, dolgu andıçları hariç iki dizinin geçerli uzunluğu sırasıyla bir ve iki ise, ilk bir ve ilk iki girdiden sonra kalan girdiler sıfırlara çekilmiş olur.

```
#@save
def sequence_mask(X, valid_len, value=0):
    """Dizilerdeki alakasız girdileri maskele."""
    maxlen = X.size(1)
    mask = torch.arange((maxlen), dtype=torch.float32,
                        device=X.device)[None, :] < valid_len[:, None]
    X[~mask] = value
    return X

X = torch.tensor([[1, 2, 3], [4, 5, 6]])
sequence_mask(X, torch.tensor([1, 2]))
```

```
tensor([[1, 0, 0],
        [4, 5, 0]])
```

Son birkaç eksendeki tüm girdileri de maskeleyebiliriz. İsterseniz, bu tür girdileri sıfır olmayan bir değerle değiştirmeyi bile belirtebilirsiniz.

```
X = torch.ones(2, 3, 4)
sequence_mask(X, torch.tensor([1, 2]), value=-1)
```

```
tensor([[[ 1.,  1.,  1.,  1.],
         [-1., -1., -1., -1.],
```

(continues on next page)

(continued from previous page)

```
         [-1., -1., -1., -1.]],

        [[ 1.,  1.,  1.,  1.],
         [ 1.,  1.,  1.,  1.],
         [-1., -1., -1., -1.]]])
```

Artık alakasız tahminlerin maskelenmesine izin vermek için softmaks çapraz entropi kaybını genişletebiliriz. Başlangıçta, tahmin edilen tüm andıçlar için maskeler bir olarak ayarlanır. Geçerli uzunluk verildikten sonra, herhangi bir dolgu andıcına karşılık gelen maske sıfır olarak ayarlanır. Sonunda, tüm andıçların kaybı, kayıptaki dolgu andıçlarının ilgisiz tahminlerini filtrelemek için maske ile çarpılacaktır.

```
#@save
class MaskedSoftmaxCELoss(nn.CrossEntropyLoss):
    """Maskelerle softmax çapraz entropi kaybı."""
    # `pred` şekli: (`batch_size`, `num_steps`, `vocab_size`)
    # `label` şekli: (`batch_size`, `num_steps`)
    # `valid_len` şekli: (`batch_size`,)
    def forward(self, pred, label, valid_len):
        weights = torch.ones_like(label)
        weights = sequence_mask(weights, valid_len)
        self.reduction='none'
        unweighted_loss = super(MaskedSoftmaxCELoss, self).forward(
            pred.permute(0, 2, 1), label)
        weighted_loss = (unweighted_loss * weights).mean(dim=1)
        return weighted_loss
```

Makulluk kontrolü için üç özdeş dizi oluşturabiliriz. Ardından, bu dizilerin geçerli uzunluklarının sırasıyla 4, 2 ve 0 olduğunu belirtebiliriz. Sonuç olarak, birinci dizinin kaybı ikinci dizininkinden iki kat kadar büyük olmalı, üçüncü dizi sıfır kaybına sahip olmalıdır.

```
loss = MaskedSoftmaxCELoss()
loss(torch.ones(3, 4, 10), torch.ones((3, 4), dtype=torch.long),
     torch.tensor([4, 2, 0]))
```

```
tensor([2.3026, 1.1513, 0.0000])
```

9.7.4 Eğitim

Aşağıdaki eğitim döngüsünde, Fig. 9.7.1 şeklinde gösterildiği gibi, kodçözücüye girdi olarak son andıç hariç özel dizi-başlangıç andıcını ve orijinal çıktı dizisini bitiştiririz. Buna *öğretici zorlama* denir çünkü orijinal çıktı dizisi (andıç etiketleri) kodçözücüye beslenir. Alternatif olarak, önceki zaman adımından *öngörülen* andıcı kodçözücüye geçerli girdi olarak da besleyebiliriz.

```
#@save
def train_seq2seq(net, data_iter, lr, num_epochs, tgt_vocab, device):
    """Bir modeli diziden diziye eğitin."""
    def xavier_init_weights(m):
        if type(m) == nn.Linear:
```

(continues on next page)

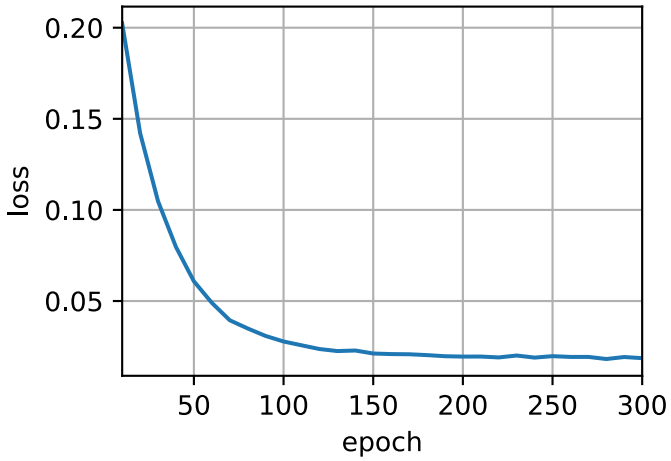
(continued from previous page)

```
            nn.init.xavier_uniform_(m.weight)
        if type(m) == nn.GRU:
            for param in m._flat_weights_names:
                if "weight" in param:
                    nn.init.xavier_uniform_(m._parameters[param])
    net.apply(xavier_init_weights)
    net.to(device)
    optimizer = torch.optim.Adam(net.parameters(), lr=lr)
    loss = MaskedSoftmaxCELoss()
    net.train()
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[10, num_epochs])
    for epoch in range(num_epochs):
        timer = d2l.Timer()
        metric = d2l.Accumulator(2)  # Eğitim kaybı toplamı, andıç sayısı
        for batch in data_iter:
            optimizer.zero_grad()
            X, X_valid_len, Y, Y_valid_len = [x.to(device) for x in batch]
            bos = torch.tensor([tgt_vocab['<bos>']] * Y.shape[0],
                               device=device).reshape(-1, 1)
            dec_input = torch.cat([bos, Y[:, :-1]], 1)  # Öğretici zorlama
            Y_hat, _ = net(X, dec_input, X_valid_len)
            l = loss(Y_hat, Y, Y_valid_len)
            l.sum().backward()  # `backward` için kaybı sayıl yap
            d2l.grad_clipping(net, 1)
            num_tokens = Y_valid_len.sum()
            optimizer.step()
            with torch.no_grad():
                metric.add(l.sum(), num_tokens)
        if (epoch + 1) % 10 == 0:
            animator.add(epoch + 1, (metric[0] / metric[1],))
    print(f'loss {metric[0] / metric[1]:.3f}, {metric[1] / timer.stop():.1f} '
          f'tokens/sec on {str(device)}')
```

Artık makine çevirisi veri kümesinde diziden-diziye öğrenme için bir RNN kodlayıcı-kodçözücü modeli oluşturabilir ve eğitebiliriz.

```
embed_size, num_hiddens, num_layers, dropout = 32, 32, 2, 0.1
batch_size, num_steps = 64, 10
lr, num_epochs, device = 0.005, 300, d2l.try_gpu()

train_iter, src_vocab, tgt_vocab = d2l.load_data_nmt(batch_size, num_steps)
encoder = Seq2SeqEncoder(
    len(src_vocab), embed_size, num_hiddens, num_layers, dropout)
decoder = Seq2SeqDecoder(
    len(tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
net = d2l.EncoderDecoder(encoder, decoder)
train_seq2seq(net, train_iter, lr, num_epochs, tgt_vocab, device)
```

```
loss 0.019, 11447.4 tokens/sec on cuda:0
```



9.7.5 Tahmin

Çıktı dizisi andıç andıç tahmin etmek için, her kodçözücü zaman adımında önceki zaman adımından tahmin edilen andıç kodçözücüye girdi olarak beslenir. Eğitime benzer şekilde, ilk zaman adımında dizi-başlangıç andıcı ("<bos>") kodçözücüye beslenir. Bu tahmin süreci Fig. 9.7.3 şeklinde gösterilmektedir. Dizi-sonu andıcı ("<eos>") tahmin edildiğinde, çıktı dizisinin tahmini tamamlanmış olur.

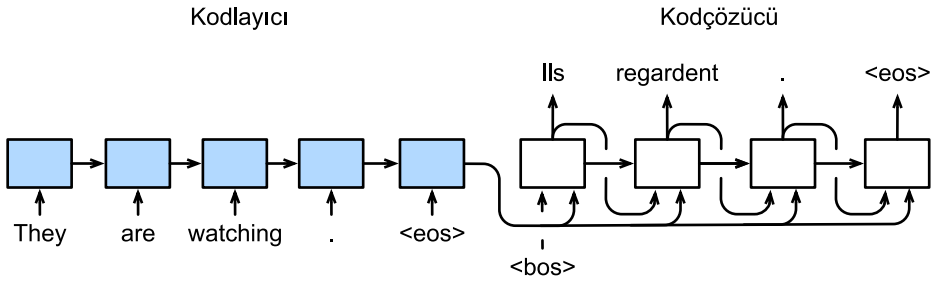


Fig. 9.7.3: Bir RNN kodlayıcı-kodçözücüsü kullanarak andıç andıç çıktı dizisini tahmin etme.

Section 9.8 içinde dizi üretimi için farklı stratejiler sunacağız.

```
#@save
def predict_seq2seq(net, src_sentence, src_vocab, tgt_vocab, num_steps,
                    device, save_attention_weights=False):
    """Diziden diziye tahmin et."""
    # Çıkarım için `net`'i değerlendirme moduna ayarlayın
    net.eval()
    src_tokens = src_vocab[src_sentence.lower().split(' ')] + [
        src_vocab['<eos>']]
    enc_valid_len = torch.tensor([len(src_tokens)], device=device)
    src_tokens = d2l.truncate_pad(src_tokens, num_steps, src_vocab['<pad>'])
    # Toplu iş eksenini ekleyin
    enc_X = torch.unsqueeze(
        torch.tensor(src_tokens, dtype=torch.long, device=device), dim=0)
    enc_outputs = net.encoder(enc_X, enc_valid_len)
    dec_state = net.decoder.init_state(enc_outputs, enc_valid_len)
```

(continues on next page)

(continued from previous page)

```
    # Toplu iş eksenini ekleyin
    dec_X = torch.unsqueeze(torch.tensor(
        [tgt_vocab['<bos>']], dtype=torch.long, device=device), dim=0)
    output_seq, attention_weight_seq = [], []
    for _ in range(num_steps):
        Y, dec_state = net.decoder(dec_X, dec_state)
        # Bir sonraki zaman adımında kod çözücünün girdisi olarak en yüksek
        # tahmin olasılığına sahip andıcı kullanıyoruz
        dec_X = Y.argmax(dim=2)
        pred = dec_X.squeeze(dim=0).type(torch.int32).item()
        # Dikkat ağırlıklarını kaydedin (daha sonra ele alınacaktır)
        if save_attention_weights:
            attention_weight_seq.append(net.decoder.attention_weights)
        # Dizi sonu andıcı tahmin edildiğinde, çıktı dizisinin oluşturulması
        # tamamlanır
        if pred == tgt_vocab['<eos>']:
            break
        output_seq.append(pred)
    return ' '.join(tgt_vocab.to_tokens(output_seq)), attention_weight_seq
```

9.7.6 Tahmin Edilen Dizilerin Değerlendirilmesi

Tahmin edilen bir diziyi etiket dizisi (gerçek referans değer) ile karşılaştırarak değerlendirebiliriz. BLEU (Bilingual Evaluation Understudy - İki Dilli Değerlendirme Dublörü), başlangıçta makine çevirisi sonuçlarını değerlendirmek için önerilmiş olsa da, farklı uygulamalar için çıktı dizilerinin kalitesini ölçmede yaygın olarak kullanılmaktadır. Prensip olarak, tahmin edilen dizideki herhangi bir n gram için, BLEU bu n-gramın etiket dizisinde görüp görünmediğini değerlendirir.

p_n ile n-gram hassasiyetini belirtelim; bu, öngörülen ve etiket dizilerindeki eşleşen n-gram adedinin tahmin edilen sıradaki n-gram adedine oranıdır. Açıklamak gerekirse A, B, C, D, D, E, F etiket dizimiz ve A, B, B, C, D tahmin edilen bir dizi olursa, elimizde $p_1 = 4/5$, $p_2 = 3/4$, $p_3 = 1/3$ ve $p_4 = 0$ olur. Ayrıca, $\text{len}_{\text{label}}$ ve len_{pred}'ün sırasıyla etiket dizisindeki ve tahmin edilen dizideki andıçların sayıları olmasına izin verin. Böylece, BLEU şöyle tanımlanır:

$$\exp\left(\min\left(0, 1 - \frac{\text{len}_{\text{label}}}{\text{len}_{\text{pred}}}\right)\right) \prod_{n=1}^{k} p_n^{1/2^n}, \tag{9.7.4}$$

burada k eşleşme için en uzun n-gramdır.

(9.7.4) denklemindeki BLEU tanımına dayanarak, tahmin edilen dizi etiket dizisi ile aynı olduğunda, BLEU değeri 1 olur. Dahası, daha uzun n-gramları eşleştirmek daha zor olduğundan, BLEU daha uzun n-gram hassasiyetine daha büyük bir ağırlık atar. Özellikle p_n sabit olduğunda n büyüdükçe $p_n^{1/2^n}$ artar (orjinal makale $p_n^{1/n}$ kullanır). Ayrıca, daha kısa dizileri tahmin etmek daha yüksek bir p_n değeri elde etme eğiliminde olduğundan, (9.7.4) denklemindeki çarpım teriminin öncesindeki katsayı daha kısa tahmin edilmiş dizileri cezalandırır. Örneğin, $k = 2$, A, B, C, D, E, F etiket dizisi ve A, B tahminlenen dizi ise, $p_1 = p_2 = 1$ olmasına rağmen, ceza çarpanı, $\exp(1 - 6/2) \approx 0.14$, BLEU değerini düşürür.

BLEU ölçüsünü aşağıdaki gibi uyguluyoruz.

```
def bleu(pred_seq, label_seq, k):  #@save
    """BLEU'yu hesapla."""
    pred_tokens, label_tokens = pred_seq.split(' '), label_seq.split(' ')
    len_pred, len_label = len(pred_tokens), len(label_tokens)
    score = math.exp(min(0, 1 - len_label / len_pred))
    for n in range(1, k + 1):
        num_matches, label_subs = 0, collections.defaultdict(int)
        for i in range(len_label - n + 1):
            label_subs[' '.join(label_tokens[i: i + n])] += 1
        for i in range(len_pred - n + 1):
            if label_subs[' '.join(pred_tokens[i: i + n])] > 0:
                num_matches += 1
                label_subs[' '.join(pred_tokens[i: i + n])] -= 1
        score *= math.pow(num_matches / (len_pred - n + 1), math.pow(0.5, n))
    return score
```

Sonunda, birkaç İngilizce cümleyi Fransızca'ya çevirmek ve sonuçların BLEU değerini hesaplamak için eğitilmiş RNN kodlayıcı-kodçözücüsünü kullanıyoruz.

```
engs = ['go .', "i lost .", 'he\'s calm .', 'i\'m home .']
fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']
for eng, fra in zip(engs, fras):
    translation, attention_weight_seq = predict_seq2seq(
        net, eng, src_vocab, tgt_vocab, num_steps, device)
    print(f'{eng} => {translation}, bleu {bleu(translation, fra, k=2):.3f}')
```

```
go . => va !, bleu 1.000
i lost . => j'ai perdu ., bleu 1.000
he's calm . => il est riche ., bleu 0.658
i'm home . => je suis chez moi <unk> ., bleu 0.803
```

9.7.7 Özet

- Kodlayıcı-kodçözücü mimarisinin tasarımını takiben, diziden-diziye öğrenme için bir model tasarlarken iki RNN kullanabiliriz.
- Kodlayıcıyı ve kodçözücüyü uygularken, çok katmanlı RNN'leri kullanabiliriz.
- Kayıp hesaplanırken olduğu gibi ilgisiz hesaplamaları filtrelemek için maskeler kullanabiliriz.
- Kodlayıcı-kodçözücü eğitiminde, öğretici zorlama yaklaşımı (tahminlerin aksine) orijinal çıktı dizilerini kodçözücüye besler.
- BLEU, tahmin edilen dizi ve etiket dizisi arasında n-gram eşleştirerek çıktı dizilerini değerlendirmek için popüler bir ölçüdür.

9.7.8 Alıştırmalar

1. Çeviri sonuçlarını iyileştirmek için hiper parametreleri ayarlayabilir misiniz?
2. Kayıp hesaplamasında maskeler kullanmadan deneyi yeniden çalıştırın. Ne sonuçlar gözlemliyorsunuz? Neden?
3. Kodlayıcı ve kodçözücü katman sayısı veya gizli birimlerin sayısı bakımından farklıysa, kodçözücünün gizli durumunu nasıl ilkleyebiliriz?
4. Eğitimde, eğitici zorlamayı kodçözücüye önceki zamanın tahminini besleme ile değiştirin. Bu performansı nasıl etkiler?
5. GRU'yu LSTM ile değiştirerek deneyi yeniden çalıştırın.
6. Kodçözücünün çıktı katmanını tasarlamanın başka yolları var mıdır?

Tartışmalar[118]

9.8 Işın Arama

Section 9.7 içinde, özel dizi sonu andıcı, "<eos>", tahmin edilene kadar çıktı dizisini andıç andıç ile tahmin ettik. Bu bölümde, bu *açgözlü arama* stratejisini formüle dökmeye ve onunla ilgili sorunları araştırmaya başlayacağız, daha sonra bu stratejiyi diğer seçeneklerle karşılaştıracağız: *Kapsamlı arama (exhaustive search)* ve *ışın arama (beam search)*.

Açgözlü aramaya biçimsel bir girişten önce, Section 9.7 içindeki aynı matematiksel gösterimi kullanarak arama problemini formülleştirelim. Herhangi bir t' zamanda adımında, $y_{t'}$ kodçüzücü çıktı olasılığı önceki çıktı altdizisi $y_1, \ldots, y_{t'-1}$ $y_1, \ldots, y_{t'-1}$'e ve girdi dizisinin bilgilerini kodlayan bağlam değişkeni $\mathbf{c}$'ye koşulludur. Hesaplama maliyetini ölçmek için, ("<eos>" içeren) çıktı kelime dağarcığını $\mathcal{Y}$ ile belirtelim. Yani bu kelime kümesinin küme büyüklüğü $|\mathcal{Y}|$ kelime dağarcığı büyüklüğündedir. Ayrıca, bir çıktı dizisinin en fazla andıç adedini T' olarak belirtelim. Sonuç olarak, hedefimiz tüm $\mathcal{O}(|\mathcal{Y}|^{T'})$ olası çıktı dizilerinden ideal bir çıktı aramaktır. Tabii ki, tüm bu çıktı dizileri için, gerçek çıktıda "<eos>" dahil sonrasındaki bölümler atılacaktır.

9.8.1 Açgözlü Arama

İlk olarak, basit bir stratejiye bir göz atalım: *Açgözlü arama*. Bu strateji Section 9.7 içindeki dizileri tahmin etmek için kullanılmıştır. Açgözlü aramada, çıktı dizisinin herhangi bir t' zaman adımında, çıktı olarak $\mathcal{Y}$'ten en yüksek koşullu olasılığa sahip andıcı ararız, yani,

$$y_{t'} = \operatorname*{argmax}_{y \in \mathcal{Y}} P(y \mid y_1, \ldots, y_{t'-1}, \mathbf{c}), \tag{9.8.1}$$

"<eos>" yayıldıktan veya çıktı dizisi maksimum uzunluğuna ulaştıktan sonra çıktı dizisi tamamlanmış olur.

Peki açgözlü arama ile ne yanlış gidebilir? Aslında, *en iyi dizi* maksimum $\prod_{t'=1}^{T'} P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \mathbf{c})$ değerine sahip çıktı dizisi olmalıdır, ki bu da girdi dizisine dayalı bir çıktı dizisi oluşturmanın koşullu olasılığıdır. Ne yazık ki, en iyi dizinin açgözlü arama ile elde edileceğinin garantisi yoktur.

[118] https://discuss.d2l.ai/t/1062

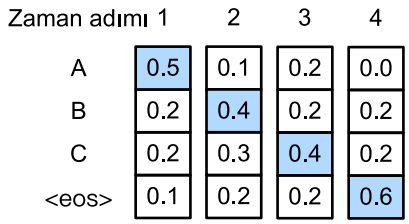


Fig. 9.8.1: Her adımda, açgözlü arama, en yüksek koşullu olasılığa sahip andıcı seçer.

Bir örnekle gösterelim. Çıktı sözlüğünde "A", "B", "C" ve "<eos>" dört andıcı olduğunu varsayalım. Fig. 9.8.1 şeklinde, her zaman adımının altındaki dört sayı, o zaman adımında sırasıyla "A", "B", "C" ve "<eos>" üretme koşullu olasılıklarını temsil eder. Her adımda, açgözlü arama, en yüksek koşullu olasılığa sahip andıcı seçer. Bu nedenle, Fig. 9.8.1 şeklinde "A", "B", "C" ve "<eos>" çıktı dizisi tahmin edilecektir. Bu çıktı dizisinin koşullu olasılığı $0.5 \times 0.4 \times 0.4 \times 0.6 = 0.048$'dir.

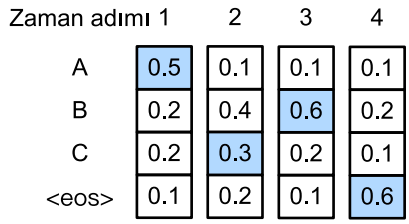


Fig. 9.8.2: Her zaman adımının altındaki dört sayı, o zaman adımında "A", "B", "C" ve "<eos>" oluşturmanın koşullu olasılıklarını temsil eder. 2. zaman adımında, ikinci en yüksek koşullu olasılığa sahip olan "C" andıcı seçilir.

Sonra, Fig. 9.8.2 şeklindeki başka bir örneğe bakalım. Fig. 9.8.1 şeklindekinin aksine, zaman adımında Fig. 9.8.2 şeklinde *ikinci* en yüksek koşullu olasılığa sahip "C" andıcını seçiyoruz. 3. zaman adımı dayandığı zaman adımları 1 ve 2, çıktı dizileri Fig. 9.8.1 örneğinde "A" ve "B"'den Fig. 9.8.2 örneğinde "A" ve "C"'ye değiştiğinden, Fig. 9.8.2 örneğinde her andıcın koşullu olasılığı 3. zaman adımında da değişti. 3. zaman adımında "B" andıcını seçtiğimizi varsayalım. Şimdi adım 4, Fig. 9.8.1 örneğinde ilk üç zaman adımın çıktısı "A", "B" ve "C" altdizisinden farklı olan "A", "C" ve "B" üzerinde koşulludur. Bu nedenle, Fig. 9.8.2 örneğindeki 4. adımda her andıcı üretmenin koşullu olasılığı da Fig. 9.8.1 örneğindeki durumdan farklıdır. Sonuç olarak, Fig. 9.8.2 örneğinde "A", "C", "B" ve "<eos>" çıktı dizisinin koşullu olasılığı $0.5 \times 0.3 \times 0.6 \times 0.6 = 0.054$'tür, bu da Fig. 9.8.1 örneğindeki açgözlü aramadakinden daha büyüktür. Bu örnekte, açgözlü arama ile elde edilen "A", "B", "C" ve"<eos>" çıktı dizisi en uygun sıra değildir.

9.8.2 Kapsamlı Arama

Amaç en uygun diziyi elde etmekse, *kapsamlı arama* kullanmayı düşünebiliriz: Olası tüm çıktı dizilerini koşullu olasılıklarıyla kapsamlı bir şekilde numaralandıralım, ardından en yüksek koşullu olasılığa sahip olanı çıktıyı alalım.

En iyi diziyi elde etmek için kapsamlı arama kullanabilsek de, hesaplama maliyeti $\mathcal{O}(|\mathcal{Y}|^{T'})$'in aşırı derecede yüksek olması muhtemeldir. Örneğin, $|\mathcal{Y}| = 10000$ ve $T' = 10$ olduğunda, $10000^{10} = 10^{40}$ tane dizi değerlendirmemiz gerekecek. Bu imkansıza yakın! Öte yandan, açgözlü aramanın hesaplama maliyeti $\mathcal{O}(|\mathcal{Y}|\,T')$'dir: Genellikle kapsamlı aramadakinden önemli ölçüde daha küçük-

tür. Örneğin, $|\mathcal{Y}| = 10000$ ve $T' = 10$ olduğunda, sadece $10000 \times 10 = 10^5$ tane dizi değerlendirmemiz gerekir.

9.8.3 Işın Arama

Sıra arama stratejileri ile ilgili kararlar, her iki uçtaki kolay sorularla birlikte bir spektrumda yatar. Eğer sadece doğruluk önemliyse? Açıkça görülüyor ki, kapsamlı arama. Ya sadece hesaplamalı maliyet önemliyse? Açıkça, açgözlü arama. Gerçek dünya uygulamaları genellikle bu iki uç arasında bir yerde karmaşık bir soru sorar.

Işın arama açgözlü aramanın geliştirilmiş bir versiyonudur. *Işın boyutu* adında bir k hiper parametresi vardır. Zaman adımında, en yüksek koşullu olasılıklara sahip k tane andıcı seçiyoruz. Her biri sırasıyla k aday çıktı dizilerinin ilk andıcı olacak. Sonraki her zaman adımında, önceki zaman adımındaki k aday çıktı dizilerine dayanarak, $k\,|\mathcal{Y}|$ olası seçeneklerden en yüksek koşullu olasılıklara sahip k tane aday çıktı dizisini seçmeye devam ediyoruz.

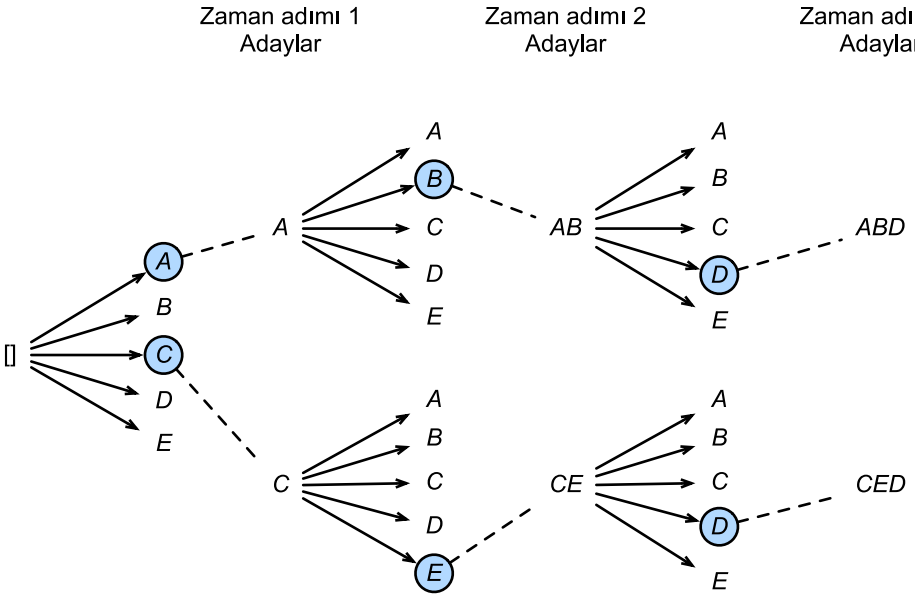


Fig. 9.8.3: Işın arama süreci (ışın boyutu: 2, bir çıktı dizisinin maksimum uzunluğu: 3). Aday çıktı dizileri A, C, AB, CE, ABD ve CED şeklindedir.

Fig. 9.8.3, bir örnek ile ışın arama sürecini gösterir. Çıktı kelime dağarcığının sadece beş öğe içerdiğini varsayalım: $\mathcal{Y} = \{A, B, C, D, E\}$ ve bunlardan biri "<eos>"dir. Işın boyutunun 2 ve bir çıktı dizisinin maksimum uzunluğunun 3 olduğunu düşünün. 1. zaman adımında, $P(y_1 \mid \mathbf{c})$ en yüksek koşullu olasılıklara sahip andıçların A ve C olduğunu varsayalım. 2. zaman adımında, tüm $y_2 \in \mathcal{Y}$, için hesaplarız:

$$\begin{aligned}P(A, y_2 \mid \mathbf{c}) &= P(A \mid \mathbf{c})P(y_2 \mid A, \mathbf{c}),\\ P(C, y_2 \mid \mathbf{c}) &= P(C \mid \mathbf{c})P(y_2 \mid C, \mathbf{c}),\end{aligned} \tag{9.8.2}$$

ve bu on değer arasında en büyük ikisini seçeriz, diyelim ki $P(A, B \mid \mathbf{c})$ ve $P(C, E \mid \mathbf{c})$. Sonra zaman adımı 3'te, tüm $y_3 \in \mathcal{Y}$ için hesaplarız:

$$\begin{aligned}P(A, B, y_3 \mid \mathbf{c}) &= P(A, B \mid \mathbf{c})P(y_3 \mid A, B, \mathbf{c}),\\ P(C, E, y_3 \mid \mathbf{c}) &= P(C, E \mid \mathbf{c})P(y_3 \mid C, E, \mathbf{c}),\end{aligned} \tag{9.8.3}$$

ve bu on değer arasında en büyük ikisini seçeriz, diyelim ki $P(A, B, D \mid \mathbf{c})$ ve $P(C, E, D \mid \mathbf{c})$. Sonuç olarak, altı aday çıktı dizisi elde ederiz: (i) A; (ii) C; (iii) A, B; (iv) C, E; (v) A, B, D ve (vi) C, E, D.

Sonunda, bu altı diziye dayalı nihai aday çıktı dizileri kümesini elde ederiz (örneğin, "<eos>" ve sonrasındaki tüm parçaları atın). Ardından, çıktı dizisi olarak aşağıdaki skorun en yüksek seviyesine sahip diziyi seçeriz:

$$\frac{1}{L^\alpha} \log P(y_1, \ldots, y_L \mid \mathbf{c}) = \frac{1}{L^\alpha} \sum_{t'=1}^{L} \log P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \mathbf{c}), \tag{9.8.4}$$

Burada L, son aday dizisinin uzunluğudur ve α genellikle 0.75 olarak ayarlanır. Daha uzun bir dizi (9.8.4) toplamında daha fazla logaritmik terime sahip olduğundan, paydadaki L^α terimi uzun dizileri cezalandırır.

Işın aramasının hesaplama maliyeti $\mathcal{O}(k \left|\mathcal{Y}\right| T')$'dir. Bu sonuç açgözlü arama ile kapsamlı arama arasında yer alır. Aslında, açgözlü arama, 1 ışın boyutuna sahip özel bir ışın araması türü olarak kabul edilebilir. Esnek bir ışın boyutu seçimi ile ışın arama, hesaplama maliyetine karşı doğruluk arasında bir denge sağlar.

9.8.4 Özet

- Dizi arama stratejileri, açgözlü arama, kapsamlı arama ve ışın aramasını içerir.
- Işın arama, ışın boyutunun esnek seçimi ile hesaplama maliyetine karşılık doğruluk arasında bir denge sağlar.

9.8.5 Alıştırmalar

1. Kapsamlı aramayı özel bir ışın araması türü olarak ele alabilir miyiz? Neden ya da neden olmasın?
2. Section 9.7 içindeki makine çeviri probleminde ışın aramasını uygulayın. Işın boyutu çeviri sonuçlarını ve tahmin hızını nasıl etkiler?
3. Section 8.5 içinde kullanıcı tarafından sağlanan önekleri takip eden metin üretmek için dil modelleme kullandık. Hangi tür bir arama stratejisi kullanılıyor? Bunu geliştirebilir misiniz?

Tartışmalar[119]

[119] https://discuss.d2l.ai/t/338

10 | Dikkat Mekanizmaları

Bir primatın görme sisteminin görme siniri, beynin tam olarak işleyebildiğini aşan devasa duyusal girdi alır. Neyse ki, tüm uyaranlar eşit yaratılmaz. Odaklanma ve bilinç yoğunlaşması primatların karmaşık görsel ortamda avlar ve yırtıcı hayvanlar gibi ilgi çekici nesnelere dikkatini yönlendirmesini sağladı. Bilginin sadece küçük bir kısmına dikkat etme yeteneği evrimsel öneme sahiptir ve bu da insanların yaşamasına ve başarılı olmasına izin verir.

Bilim adamları 19. yüzyıldan beri bilişsel sinirbilim alanında dikkati çalışıyor. Bu bölümde, dikkatin görsel bir sahnede nasıl konuşlandığını açıklayan popüler bir çerçeveyi inceleyerek başlayacağız. Bu çerçevedeki dikkat işaretlerinden esinlenerek, bu tür dikkat işaretlerinden faydalanan modeller tasarlayacağız. Özellikle, 1964 yılında Nadaraya-Waston çekirdek regresyonu, makine öğrenmesinin *dikkat mekanizmaları* ile basit bir kanıtlamasıdır.

Ardından, derin öğrenmede dikkat modellerinin tasarımında yaygın olarak kullanılan dikkat işlevlerini tanıtmaya devam edeceğiz. Özellikle, çift yönlü hizalanabilen ve türevlenebilen derin öğrenmede çığır açıcı bir dikkat modeli olan *Bahdanau dikkat*ini tasarlamak için bu işlevleri nasıl kullanacağımızı göstereceğiz.

Sonunda, daha yakın zaman *çoklu kafalı dikkat* ve *özdikkat* tasarımları ile donatılmış, sadece dikkat mekanizmalarına dayanan *dönüştürücü* mimarisini tanımlayacağız. Dönüştürücüler 2017'de önerilmelerinden bu yana dil, görme, konuşma ve pekiştirmeli öğrenme alanlarındaki modern derin öğrenme uygulamalarında yaygındır.

10.1 Dikkat İşaretleri

Bu kitaba gösterdiğiniz ilgi için teşekkür ederiz. Dikkat kıt bir kaynaktır: Şu anda bu kitabı okuyor ve gerisini görmezden geliyorsunuz. Böylece, paraya benzer şekilde, dikkatiniz bir fırsat maliyeti ile ödeniyor. Şu anda dikkat yatırımınızın değerli olmasını sağlamak için, güzel bir kitap üretmede dikkatimizi itinalı bir şekilde göstermeye son derece motive olduk. Dikkat, hayatın kemerindeki kilit taştır ve herhangi bir işin istisnacılığının anahtarını tutar.

Ekonomi, kıt kaynakların tahsisini incelediğinden, insanların dikkatinin değiştirilebilecek sınırlı, değerli ve kıt bir hammadde olarak ele alındığı dikkat ekonomisi çağındayız. Yararlanmak için üzerinde çok sayıda iş modeli geliştirilmiştir. Müzik veya video akışı hizmetlerinde ya reklamlarına dikkat ediyoruz ya da bunları gizlemek için para ödüyoruz. Çevrimiçi oyunlar dünyasında büyüme için, ya yeni oyuncular çeken savaşlara katılmaya dikkat ediyoruz ya da anında güçlü olmak için para ödüyoruz. Hiçbir şey bedavaya gelmez.

Sonuçta, çevremizdeki bilgiler kıt değilken dikkat kıttır. Görsel bir sahneyi incelerken, görsel sinirimiz saniyede 10^8 bit ölçeğinde bilgi alır ve bu beynimizin tam olarak işleyebileceğini çok aşar. Neyse ki, atalarımız deneyimlerimizden (veri olarak da bilinir) öğrenmişlerdi ki, *tüm duyusal girdiler eşit yaratılmamıştır*. İnsanlık tarihi boyunca, ilgi duyulan bilginin yalnızca bir kısmına

dikkati yöneltme yeteneği, beynimizin, avcıları, avları ve arkadaşları tespit etmek gibi, hayatta kalmak, büyümek ve sosyalleşmek için kaynakları daha akıllıca tahsis etmesini sağlamıştır.

10.1.1 Biyolojide Dikkat İşaretleri

Görsel dünyada dikkatimizin nasıl dağıtıldığını açıklamak için iki bileşenli bir çerçeve ortaya çıktı ve yaygın oldu. Bu fikir 1890'larda "Amerikan psikolojisinin babası" (James, 2007) olarak kabul edilen William James'e dayanmaktadır. Bu çerçevede, konular seçici olarak dikkatin sahne ışığını hem *istemsiz işaret* hem de *istemli işaret* kullanarak yönlendirir.

İstemsiz işaret, ortamdaki nesnelerin belirginliğine ve barizliğine dayanır. Önünüzde beş nesne olduğunu düşünün: Fig. 10.1.1 şeklinde gösterildiği bir gazete, bir araştırma makalesi, bir fincan kahve, bir defter ve bir kitap. Tüm kağıt ürünleri siyah beyaz basılıyken kahve fincanı kırmızıdır. Başka bir deyişle, bu kahve, bu görsel ortamda kendiliğinden belirgin ve bariz, otomatik ve istemsiz dikkat çekiyor. Böylece fovea'yı (görme keskinliğinin en yüksek olduğu makula merkezi) Fig. 10.1.1 şeklinde gösterildiği gibi kahvenin üzerine getirirsiniz.

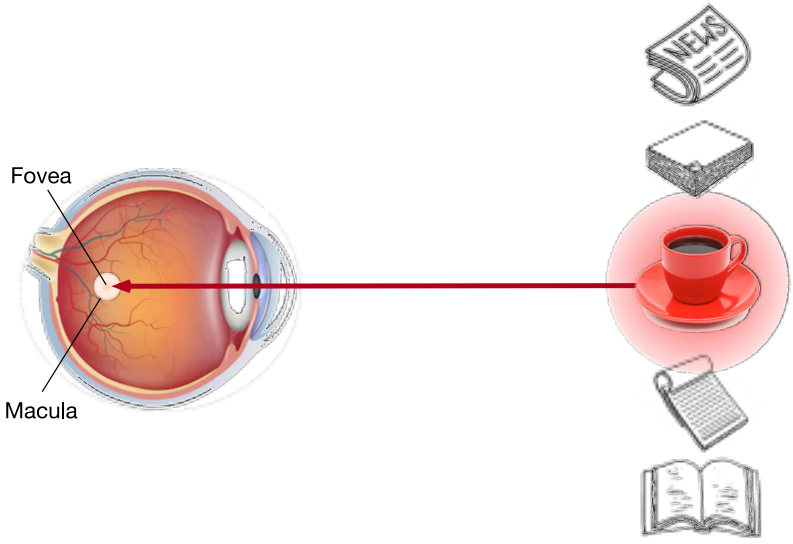


Fig. 10.1.1: Belirginliğe dayalı istemsiz işareti kullanarak (kırmızı fincan, kağıt olmayan), dikkat istemeden kahveye yönlendirilir.

Kahve içtikten sonra kafeinlenmiş olursunuz ve kitap okumak istersiniz. Yani başınızı çevirin, gözlerinizi yeniden odaklayın ve Fig. 10.1.2 şeklinde tasvir edildiği gibi kitaba bakın. Kahve, belirginliğe göre seçme konusunda sizi önyargılı yapar, Fig. 10.1.1 şeklindeki durumdan farklı olarak, bu görev bağımlı durumda bilişsel ve istemli kontrol altında kitabı seçersiniz. Değişken seçim kriterlerine dayalı istemli işaret kullanarak, bu dikkat biçimi daha kasıtlıdır. Ayrıca deneğin gönüllü çabaları ile daha güçlüdür.

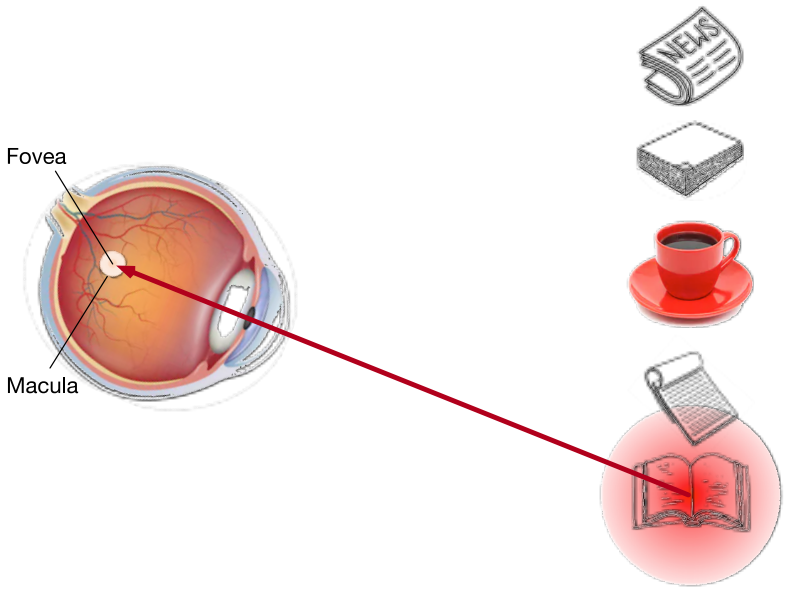


Fig. 10.1.2: Göreve bağlı istemli işaret (kitap okumayı istemek) kullanılarak, dikkat istemli kontrol altındaki kitaba yönlendirilir.

10.1.2 Sorgular, Anahtarlar ve Değerler

Dikkatli konuşlandırmayı açıklayan istemsiz ve istemli dikkat işaretlerinden esinlenerek, aşağıda bu iki dikkat işaretini birleştirerek dikkat mekanizmalarını tasarlamak için bir çerçeve anlatacağız.

Başlangıç olarak, yalnızca istemsiz işaretlerin mevcut olduğu daha basit durumu göz önünde bulundurun. Duyusal girdiler üzerinden seçimi önyargılı hale getirmek için, basitçe parametreleştirilmiş tam bağlı bir katman veya hatta parametrelenmemiş maksimum veya ortalama ortaklama kullanabiliriz.

Bu nedenle, dikkat mekanizmalarını tam bağlı katmanlardan veya ortaklama katmanlarından ayıran şey, istemli işaretlerin dahil edilmesidir. Dikkat mekanizmaları bağlamında, istemli işaretlere *sorgular* olarak atıfta bulunuyoruz. Herhangi bir sorgu göz önüne alındığında, dikkat mekanizmaları duyusal girdiler üzerinde seçimi (örneğin, ara öznitelik temsilleri) *dikkat ortaklama* yoluyla yönlendirir. Bu duyusal girdilere dikkat mekanizmaları bağlamında *değerler* denir. Daha genel olarak, her değer bir *anahtar* ile eşleştirilir ve bu durum, bu duyusal girdinin istemsiz işareti olarak düşünülebilir. Fig. 10.1.3 içinde gösterildiği gibi, verilen sorgu (istemli işaret), değerler (duyusal girdiler) üzerinde ek girdi seçimini yönlendiren anahtarlarla (istemsiz işaretler) etkileşime girebilmesi için dikkat ortaklamasını tasarlayabiliriz.

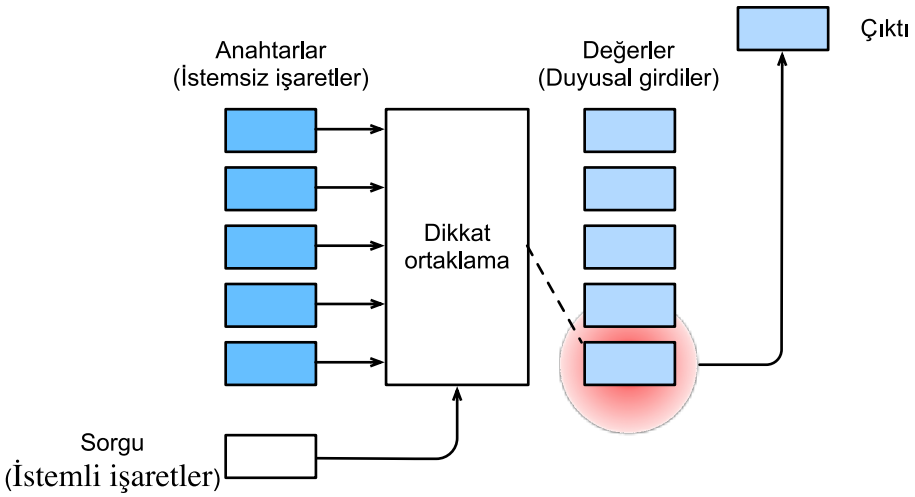


Fig. 10.1.3: Dikkat mekanizmaları, sorguları (istemli işaretleri) ve anahtarları (istemsiz işaretleri) içeren dikkat ortaklama aracılığıyla değerler (duyusal girdiler) üzerinden seçimi önyargılı kılar.

Dikkat mekanizmalarının tasarımı için birçok alternatif olduğunu unutmayın. Örneğin, (Mnih *et al.*, 2014) pekiştirmeli öğrenme yöntemleri kullanılarak eğitilebilen türevlenemeyen bir dikkat modeli tasarlayabiliriz. Fig. 10.1.3 şeklindeki çerçevenin hakimiyeti göz önüne alındığında, bu çerçevedeki modeller bu bölümdeki dikkatimizin merkezi olacak.

10.1.3 Dikkat Görselleştirme

Ortalama ortaklama ağırlıklarının tekdüze olduğu ağırlıklı bir girdi ortalaması olarak değerlendirilebilir. Pratikte dikkat ortaklama, verilen sorgu ile farklı anahtarlar arasında ağırlıkların hesaplandığı ağırlıklı ortalamayı kullanarak değerleri toplar.

```
import torch
from d2l import torch as d2l
```

Dikkat ağırlıklarını görselleştirmek için `show_heatmaps` işlevini tanımlıyoruz. `matrices` girdisi (görüntüleme için satır sayısı, görüntüleme için sütun sayısı, sorgu sayısı, anahtar sayısı) biçimine sahiptir.

```
#@save
def show_heatmaps(matrices, xlabel, ylabel, titles=None, figsize=(2.5, 2.5),
                  cmap='Reds'):
    """Matrislerin ısı haritalarını gösterir"""
    d2l.use_svg_display()
    num_rows, num_cols = matrices.shape[0], matrices.shape[1]
    fig, axes = d2l.plt.subplots(num_rows, num_cols, figsize=figsize,
                                 sharex=True, sharey=True, squeeze=False)
    for i, (row_axes, row_matrices) in enumerate(zip(axes, matrices)):
        for j, (ax, matrix) in enumerate(zip(row_axes, row_matrices)):
            pcm = ax.imshow(matrix.detach().numpy(), cmap=cmap)
            if i == num_rows - 1:
                ax.set_xlabel(xlabel)
            if j == 0:
```

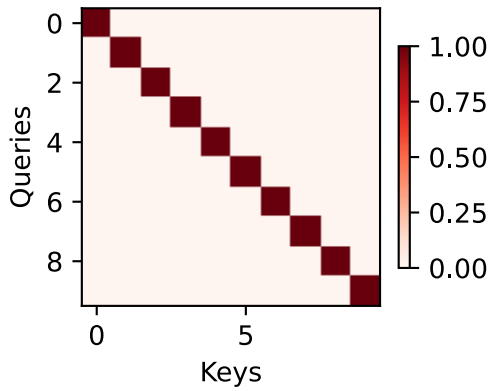
(continues on next page)

(continued from previous page)

```
            ax.set_ylabel(ylabel)
        if titles:
            ax.set_title(titles[j])
    fig.colorbar(pcm, ax=axes, shrink=0.6);
```

Gösterim için, dikkat ağırlığının yalnızca sorgu ve anahtar aynı olduğunda bir olduğu basit bir durum düşünün; aksi takdirde sıfırdır.

```
attention_weights = torch.eye(10).reshape((1, 1, 10, 10))
show_heatmaps(attention_weights, xlabel='Keys', ylabel='Queries')
```



Sonraki bölümlerde, dikkat ağırlıklarını görselleştirmek için sıklıkla bu işlevi çağıracağız.

10.1.4 Özet

- İnsanın dikkati sınırlı, değerli ve kıt bir kaynaktır.
- Denekler hem istemsiz hem de istemli işaretleri kullanarak dikkati seçici olarak yönlendirir. Birincisi, belirginliğe dayanır ve ikincisi görev bağımlıdır.
- Dikkat mekanizmaları, istemli işaretlerin dahil edilmesi nedeniyle tam bağlı katmanlardan veya ortaklama katmanlarından farklıdır.
- Dikkat mekanizmaları, sorguları (istemli işaretler) ve anahtarları (istemsiz işaretler) içeren dikkat ortaklama aracılığıyla değerler (duyusal girdiler) üzerinden seçimi önyargılı kılar. Anahtarlar ve değerler eşleştirilir.
- Sorgular ve anahtarlar arasındaki dikkat ağırlıklarını görselleştirebiliriz.

10.1.5 Alıştırmalar

1. Makine çevirisinde bir dizinin kodunu andıç andıç çözerken istemli işaret ne olabilir? İsteğe bağlı olmayan sinyaller ve duyusal girdiler nelerdir?
2. Rastgele bir 10×10'luk bir matris oluşturun ve her satırın geçerli bir olasılık dağılımı olduğundan emin olmak için softmaks işlemini kullanın. Çıktı dikkat ağırlıklarını görselleştirin.

Tartışmalar[120]

10.2 Dikkat Ortaklama: Nadaraya-Watson Çekirdek Bağlanımı

Artık Fig. 10.1.3 çerçevesinde dikkat mekanizmalarının ana bileşenlerini biliyorsunuz. Yeniden özetlemek için sorgular (istemli işaretler) ve anahtarlar (istemsiz işaretler) arasındaki etkileşimler *dikkat ortaklama* ile sonuçlanır. Dikkat ortaklama, çıktıyı üretmek için seçici olarak değerleri (duyusal girdiler) bir araya getirir. Bu bölümde, dikkat mekanizmalarının pratikte nasıl çalıştığına dair üst düzey bir görünüm vermek için dikkat ortaklamasını daha ayrıntılı olarak anlatacağız. Özellikle, 1964 yılında önerilen Nadaraya-Watson çekirdek bağlanım modeli, makine öğrenmesini dikkat mekanizmaları ile göstermek için basit ama eksiksiz bir örnektir.

```
import torch
from torch import nn
from d2l import torch as d2l
```

10.2.1 Veri Kümesi Oluşturma

İşleri basit tutmak için, aşağıdaki regresyon problemini ele alalım: $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ girdi-çıktı çiftlerinin bir veri kümesi verildiğinde $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, $\hat{y} = f(x)$ yeni bir x girdisinin çıktısını tahmin etmek için $\hat{y} = f(x)$ nasıl öğrenilir?

Burada ϵ gürültü terimi ile aşağıdaki doğrusal olmayan fonksiyona göre yapay bir veri kümesi oluşturuyoruz:

$$y_i = 2\sin(x_i) + x_i^{0.8} + \epsilon, \tag{10.2.1}$$

burada ϵ sıfır ortalama ve 0.5 standart sapma ile normal bir dağılıma uyar. Hem 50 eğitim örneği hem de 50 test örneği üretilir. Dikkat modelini daha sonra daha iyi görselleştirmek için, eğitim girdileri sıralanır.

```
n_train = 50  # Eğitim örneklerinin adedi
x_train, _ = torch.sort(torch.rand(n_train) * 5)   # Eğitim girdileri
```

```
def f(x):
    return 2 * torch.sin(x) + x**0.8

y_train = f(x_train) + torch.normal(0.0, 0.5, (n_train,))  # Eğitim çıktıları
x_test = torch.arange(0, 5, 0.1)  # Test örnekleri
```

(continues on next page)

[120] https://discuss.d2l.ai/t/1592

(continued from previous page)

```
y_truth = f(x_test)  # Test örneklerinin adedi gerçek referans değeri
n_test = len(x_test)  # Test örneklerinin adedi
n_test
```

```
50
```

Aşağıdaki işlev, tüm eğitim örneklerini (dairelerle temsil edilmiş), gürültü terimi olmadan gerçek referans değer veri üretme işlevi 'f'yi ("Truth" ile etiketlenmiş) ve öğrenilen tahmin işlevini ("Pred" ile etiketlenmiş) çizer.

```
def plot_kernel_reg(y_hat):
    d2l.plot(x_test, [y_truth, y_hat], 'x', 'y', legend=['Truth', 'Pred'],
             xlim=[0, 5], ylim=[-1, 5])
    d2l.plt.plot(x_train, y_train, 'o', alpha=0.5);
```

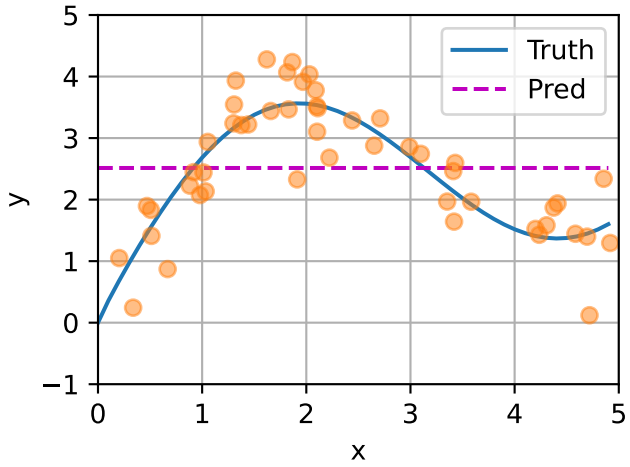
10.2.2 Ortalama Ortaklama

Bu regresyon problemi için belki de dünyanın "en aptalca" tahmin edicisiyle başlıyoruz: Ortalama ortaklama kullanarak tüm eğitim çıktıları üzerinde ortalama,

$$f(x) = \frac{1}{n}\sum_{i=1}^{n} y_i, \tag{10.2.2}$$

aşağıda çizilmiştir. Gördüğümüz gibi, bu tahminci gerçekten o kadar akıllı değil.

```
y_hat = torch.repeat_interleave(y_train.mean(), n_test)
plot_kernel_reg(y_hat)
```



10.2.3 Parametrik Olmayan Dikkat Ortaklama

Açıkçası, ortalama ortaklama girdileri, x_i, atlar. Girdi yerlerine göre y_i çıktılarını ağırlıklandırmak için Nadaraya (Nadaraya, 1964) ve Watson (Watson, 1964) tarafından daha iyi bir fikir önerildi:

$$f(x) = \sum_{i=1}^{n} \frac{K(x - x_i)}{\sum_{j=1}^{n} K(x - x_j)} y_i, \tag{10.2.3}$$

burada K bir *çekirdek*tir. (10.2.3) içindeki tahmin ediciye *Nadaraya-Watson çekirdek regresyonu* denir. Burada çekirdeklerin ayrıntılarına dalmayacağız. Fig. 10.1.3 içindeki dikkat mekanizmalarının çerçevesini hatırlayın. Dikkat açısından bakıldığında, (10.2.3) denklemini *dikkat ortaklama*nın daha genelleştirilmiş bir formunda yeniden yazabiliriz:

$$f(x) = \sum_{i=1}^{n} \alpha(x, x_i) y_i, \tag{10.2.4}$$

burada x sorgu ve (x_i, y_i) anahtar-değer çiftidir. (10.3.1) ve (10.2.2) karşılaştırılırsa, buradaki dikkat havuzlama y_i değerlerinin ağırlıklı bir ortalamasıdır. (10.3.1) içindeki *dikkat ağırlığı* $\alpha(x, x_i)$, x sorgu ve α ile modellenen anahtar x_i arasındaki etkileşime dayalı olarak karşılık gelen y_i değerine atanır. Herhangi bir sorgu için, tüm anahtar-değer çiftleri üzerindeki dikkat ağırlıkları geçerli bir olasılık dağılımıdır: Negatif değillerdir ve bire toplanırlar.

Dikkat ortaklama sezgileri kazanmak için, sadece aşağıda tanımlanan bir *Gauss çekirdeği*ni düşünün

$$K(u) = \frac{1}{\sqrt{2\pi}} \exp(-\frac{u^2}{2}). \tag{10.2.5}$$

Gauss çekirdeğini (10.3.1) ve (10.2.3) denklemlerine koyarsak

$$\begin{aligned} f(x) &= \sum_{i=1}^{n} \alpha(x, x_i) y_i \\ &= \sum_{i=1}^{n} \frac{\exp\left(-\frac{1}{2}(x - x_i)^2\right)}{\sum_{j=1}^{n} \exp\left(-\frac{1}{2}(x - x_j)^2\right)} y_i \\ &= \sum_{i=1}^{n} \text{softmax}\left(-\frac{1}{2}(x - x_i)^2\right) y_i. \end{aligned} \tag{10.2.6}$$

(10.2.6) içinde, verilen x sorgusuna daha yakın olan bir x_i anahtarı, anahtarın karşılık gelen y_i değerine atanan *daha büyük bir dikkat ağırlığı* aracılığıyla *daha fazla dikkat* alacaktır.

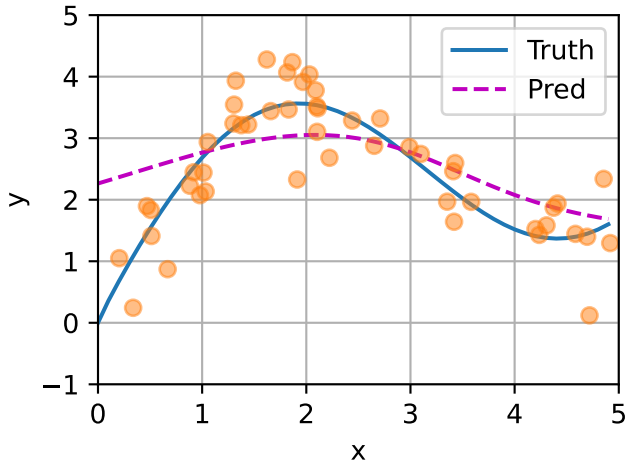
Nadaraya-Watson çekirdek regresyonu parametrik olmayan bir modeldir; bu nedenle (10.2.6), *parametrik olmayan dikkat ortaklama* örneğidir. Aşağıda, bu parametrik olmayan dikkat modeline dayanarak tahmini çiziyoruz. Tahmin edilen çizgi düzgün ve ortalama ortaklama tarafından üretilen gerçek referans değere daha yakındır.

```
# `X_repeat` şekli: (`n_test`, `n_train`), burada her satır aynı test
# girdilerini içerir (yani aynı sorgular)
X_repeat = x_test.repeat_interleave(n_train).reshape((-1, n_train))
# `x_train`'in anahtarları içerdiğine dikkat edin. `attention_weights` şekli:
# (`n_test`, `n_train`)'dir, burada her satır, her sorguya verilen değerler
# (`y_train`) arasında atanacak dikkat ağırlıklarını içerir
```

(continues on next page)

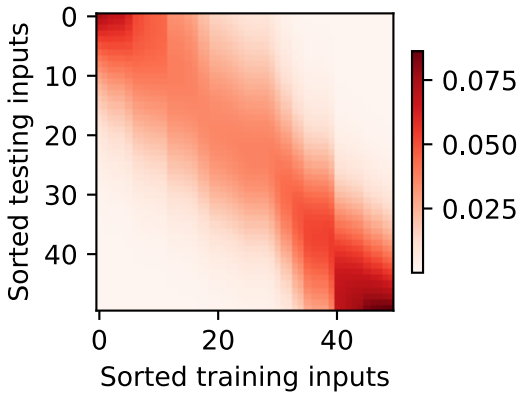
(continued from previous page)

```
attention_weights = nn.functional.softmax(-(X_repeat - x_train)**2 / 2, dim=1)
# `y_hat`'nin her bir öğesi, ağırlıkların dikkat ağırlıkları olduğu
# değerlerin ağırlıklı ortalamasıdır.
y_hat = torch.matmul(attention_weights, y_train)
plot_kernel_reg(y_hat)
```



Şimdi dikkat ağırlıklarına bir göz atalım. Burada test girdileri sorgulardır, eğitim girdileri ise anahtarlardır. Her iki girdi sıralandığından, sorgu-anahtar çifti ne kadar yakın olursa, dikkat ortaklamasında dikkat ağırlığı o kadar yüksek olur.

```
d2l.show_heatmaps(attention_weights.unsqueeze(0).unsqueeze(0),
                  xlabel='Sorted training inputs',
                  ylabel='Sorted testing inputs')
```



10.2.4 Parametrik Dikkat Ortaklama

Parametrik olmayan Nadaraya-Watson çekirdek regresyonu *tutarlılık* avantajından yararlanır: Yeterli veri verildiğinde bu model en uygun çözüme yakınlaşır. Bununla birlikte, öğrenilebilir parametreleri dikkat ortaklamasına kolayca tümleştirebiliriz.

Örnek olarak, (10.2.6) denkleminden biraz farklı olarak, aşağıdaki gibi x sorgu ve x_i anahtarı arasındaki uzaklık, öğrenilebilir bir parametre w ile çarpılır:

$$\begin{aligned} f(x) &= \sum_{i=1}^{n} \alpha(x, x_i) y_i \\ &= \sum_{i=1}^{n} \frac{\exp\left(-\frac{1}{2}((x - x_i)w)^2\right)}{\sum_{j=1}^{n} \exp\left(-\frac{1}{2}((x - x_j)w)^2\right)} y_i \\ &= \sum_{i=1}^{n} \text{softmax}\left(-\frac{1}{2}((x - x_i)w)^2\right) y_i. \end{aligned} \tag{10.2.7}$$

Bölümün geri kalanında, (10.2.7) denklemindeki dikkat ortaklama parametresini öğrenerek bu modeli eğiteceğiz.

Toplu Matris Çarpması

Minigruplar için dikkati daha verimli bir şekilde hesaplamak için, derin öğrenme çerçeveleri tarafından sağlanan toplu matris çarpma yardımcı programlarından yararlanabiliriz.

İlk minigrup n matrisleri $\mathbf{X}_1, \ldots, \mathbf{X}_n$ şekil $a \times b$ içerdiğini ve ikinci minibatch n matrisleri $b \times c$ şekilli $\mathbf{Y}_1, \ldots, \mathbf{Y}_n$ matrisleri içerdiğini varsayalım. Onların toplu matris çarpımı n şekil $a \times c$ matrisleri $\mathbf{X}_1\mathbf{Y}_1, \ldots, \mathbf{X}_n\mathbf{Y}_n$ ile sonuçlanır. Bu nedenle, iki şekil tensör verilen (n, a, b) ve (n, b, c), toplu matris çarpma çıktılarının şekli (n, a, c)'dir.

İlk minigrubun $a \times b$ şeklinde $\mathbf{X}_1, \ldots, \mathbf{X}_n$ n matrisi içerdiğini ve ikinci minigrubun $b \times c$ şeklinde $\mathbf{Y}_1, \ldots, \mathbf{Y}_n$ n matrisi içerdiğini varsayalım. Onların toplu matris çarpımı, $\mathbf{X}_1\mathbf{Y}_1, \ldots, \mathbf{X}_n\mathbf{Y}_n$ $a \times c$ şeklinde n matris ile sonuçlanır. Bu nedenle, (n, a, b) ve (n, b, c) şeklinde iki tensör verildiğinde, toplu matris çarpım çıktılarının şekli (n, a, c)'dir.

```
X = torch.ones((2, 1, 4))
Y = torch.ones((2, 4, 6))
torch.bmm(X, Y).shape
```

```
torch.Size([2, 1, 6])
```

Dikkat mekanizmaları bağlamında, bir minigruptaki değerlerin ağırlıklı ortalamalarını hesaplamak için minigrup matris çarpımını kullanabiliriz.

```
weights = torch.ones((2, 10)) * 0.1
values = torch.arange(20.0).reshape((2, 10))
torch.bmm(weights.unsqueeze(1), values.unsqueeze(-1))
```

```
tensor([[[ 4.5000]],

        [[14.5000]]])
```

Modeli Tanımlama

Minigrup matris çarpımını kullanarak, aşağıda (10.2.7) denklemindeki parametrik dikkat ortaklamayı temel alan Nadaraya-Watson çekirdek regresyonunun parametrik versiyonunu tanımlıyoruz.

```
class NWKernelRegression(nn.Module):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.w = nn.Parameter(torch.rand((1,), requires_grad=True))

    def forward(self, queries, keys, values):
        # `queries` ve `attention_weights` çıktısının şekli:
        # (sorgu sayısı, anahtar/değer çifti sayısı)
        queries = queries.repeat_interleave(keys.shape[1]).reshape((-1, keys.shape[1]))
        self.attention_weights = nn.functional.softmax(
            -((queries - keys) * self.w)**2 / 2, dim=1)
        # `values` (değerler) şekli: (sorgu sayısı, anahtar/değer çifti sayısı)
        return torch.bmm(self.attention_weights.unsqueeze(1),
                         values.unsqueeze(-1)).reshape(-1)
```

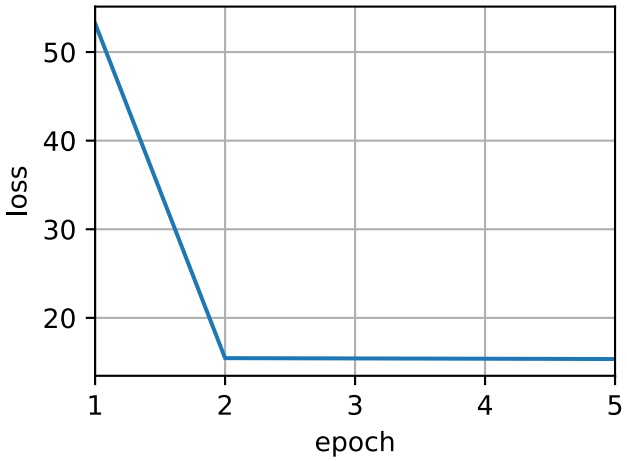
Eğitim

Aşağıda, dikkat modelini eğitmek için eğitim veri kümesini anahtar ve değerlere dönüştürürüz. Parametrik dikkat ortaklamada, herhangi bir eğitim girdisi çıktısını tahmin etmek için kendisi dışındaki tüm eğitim örneklerinden anahtar-değer çiftlerini alır.

```
# `X_tile`'in şekli: (`n_train`, `n_train`), burada her sütun aynı
#  eğitim girdilerini içerir
X_tile = x_train.repeat((n_train, 1))
# `Y_tile`'in şekli: (`n_train`, `n_train`), burada her sütun aynı
# eğitim çıktılarını içerir
Y_tile = y_train.repeat((n_train, 1))
# `keys`'in şekli: (`n_train`, `n_train` - 1)
keys = X_tile[(1 - torch.eye(n_train)).type(torch.bool)].reshape((n_train, -1))
# `values`'in şekli: (`n_train`, `n_train` - 1)
values = Y_tile[(1 - torch.eye(n_train)).type(torch.bool)].reshape((n_train, -1))
```

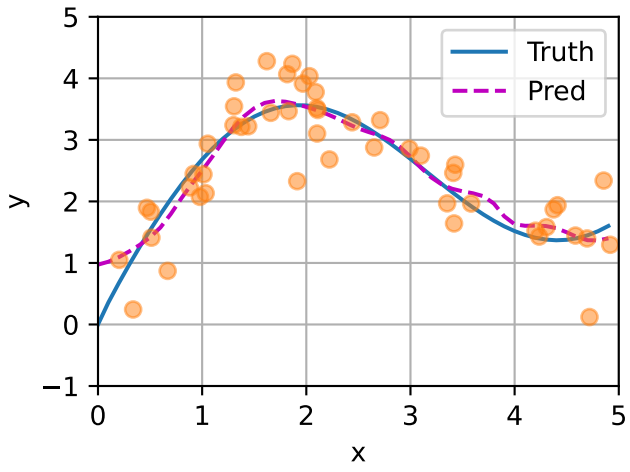
Kare kayıp ve rasgele gradyan inişi kullanarak, parametrik dikkat modelini eğitiriz.

```
net = NWKernelRegression()
loss = nn.MSELoss(reduction='none')
trainer = torch.optim.SGD(net.parameters(), lr=0.5)
animator = d2l.Animator(xlabel='epoch', ylabel='loss', xlim=[1, 5])

for epoch in range(5):
    trainer.zero_grad()
    l = loss(net(x_train, keys, values), y_train)
    l.sum().backward()
    trainer.step()
    print(f'epoch {epoch + 1}, loss {float(l.sum()):.6f}')
    animator.add(epoch + 1, float(l.sum()))
```



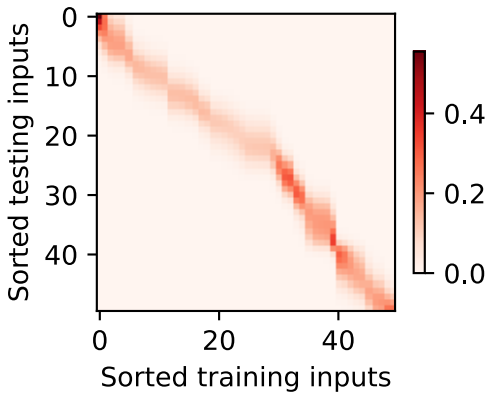
Parametrik dikkat modelini eğittikten sonra, tahminini çizebiliriz. Eğitim veri kümesini gürültüye oturtmaya çalışırken, tahmin edilen çizgi, daha önce çizilen parametrik olmayan karşılığından daha az pürüzsüzdür.

```
# `keys`'in şekli: (`n_test`, `n_train`), burada her sütun aynı eğitim
# girdilerini (yani aynı anahtarları) içerir
keys = x_train.repeat((n_test, 1))
# `value`'in şekli: (`n_test`, `n_train`)
values = y_train.repeat((n_test, 1))
y_hat = net(x_test, keys, values).unsqueeze(1).detach()
plot_kernel_reg(y_hat)
```



Parametrik olmayan dikkat ortaklama ile karşılaştırıldığında, öğrenilebilir ve parametrik ayarda büyük dikkat ağırlıkları olan bölge daha keskinleşir.

```
d2l.show_heatmaps(net.attention_weights.unsqueeze(0).unsqueeze(0),
                  xlabel='Sorted training inputs',
                  ylabel='Sorted testing inputs')
```



10.2.5 Özet

- Nadaraya-Watson çekirdek regresyonu, makine öğrenmesinin dikkat mekanizmaları ile bir örneğidir.
- Nadaraya-Watson çekirdek regresyonunun dikkat ortaklaması, eğitim çıktılarının ağırlıklı bir ortalamasıdır. Dikkat açısından bakıldığında, dikkat ağırlığı, bir sorgunun işlevine ve değerle eşleştirilmiş anahtara dayanan bir değere atanır.
- Dikkat ortaklama parametrik olabilir de olmayabilir de.

10.2.6 Alıştırmalar

1. Eğitim örneklerinin sayısını artırın. Parametrik olmayan Nadaraya-Watson çekirdek regresyonunu daha iyi öğrenebilir misin?
2. Parametrik dikkat ortaklama deneyinde öğrendiğimiz w'nin değeri nedir? Dikkat ağırlıklarını görselleştirirken ağırlıklı bölgeyi neden daha keskin hale getiriyor?
3. Daha iyi tahmin etmek için parametrik olmayan Nadaraya-Watson çekirdek regresyonuna nasıl hiper parametre ekleyebiliriz?
4. Bu bölümün çekirdek regresyonu için başka bir parametrik dikkat ortaklama tasarlayın. Bu yeni modeli eğitin ve dikkat ağırlıklarını görselleştirin.

Tartışmalar[121]

10.3 Dikkat Puanlama Fonksiyonları

Section 10.2 içinde, sorgular ve anahtarlar arasındaki etkileşimleri modellemek için bir Gauss çekirdeği kullandık. (10.2.6) denkleminde Gauss çekirdeğinin üssüne bir *dikkat puanlama fonksiyonu* (veya kısaca *puanlama fonksiyonu*) olarak muamele edilerek, bu fonksiyonun sonuçları temelde bir softmaks işlemine beslendi. Sonuç olarak, anahtarlarla eşleştirilmiş değerler üzerinde bir olasılık dağılımı (dikkat ağırlıkları) elde ettik. Sonunda, dikkat ortaklamasının çıktısı, bu dikkat ağırlıklarına dayanan değerlerin ağırlıklı bir toplamıdır.

[121] https://discuss.d2l.ai/t/1599

Üst seviyede, Fig. 10.1.3 şeklindeki dikkat mekanizmalarının çerçevesini oluşturmak için yukarıdaki algoritmayı kullanabiliriz. a ile dikkat ortaklama işlevini gösteren Fig. 10.3.1 şekli, dikkat havuzlama çıktısının ağırlıklı bir değer toplamı olarak nasıl hesaplanabileceğini göstermektedir. Dikkat ağırlıkları bir olasılık dağılımı olduğundan, ağırlıklı toplamı esas olarak ağırlıklı bir ortalamadır.

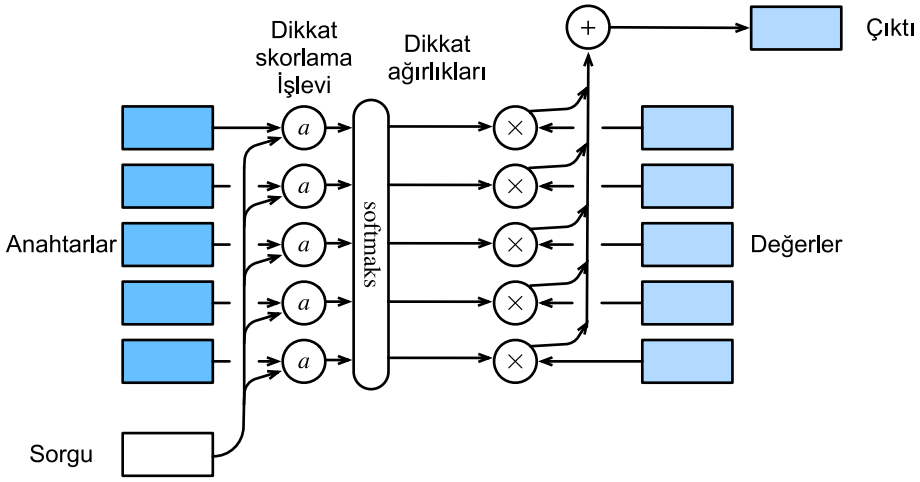


Fig. 10.3.1: Değerlerin ağırlıklı ortalaması olarak dikkat ortaklamasının çıktısını hesaplama.

Matematiksel olarak, $\mathbf{q} \in \mathbb{R}^q$ sorgumuz ve m $(\mathbf{k}_1, \mathbf{v}_1), \ldots, (\mathbf{k}_m, \mathbf{v}_m)$ anahtar-değer çiftlerimiz olduğunu varsayalım, öyle ki $\mathbf{k}_i \in \mathbb{R}^k$ ve $\mathbf{v}_i \in \mathbb{R}^v$. Dikkat ortaklama f, değerlerin ağırlıklı bir toplamı olarak örneklendirilir:

$$f(\mathbf{q}, (\mathbf{k}_1, \mathbf{v}_1), \ldots, (\mathbf{k}_m, \mathbf{v}_m)) = \sum_{i=1}^{m} \alpha(\mathbf{q}, \mathbf{k}_i)\mathbf{v}_i \in \mathbb{R}^v, \tag{10.3.1}$$

$\mathbf{q}$ ve anahtar $\mathbf{k}_i$ için dikkat ağırlığı (skaler), iki vektörü bir skalere eşleyen bir dikkat puanlama işlevi a'nın softmaks işlemi ile hesaplanır:

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \mathrm{softmax}(a(\mathbf{q}, \mathbf{k}_i)) = \frac{\exp(a(\mathbf{q}, \mathbf{k}_i))}{\sum_{j=1}^{m} \exp(a(\mathbf{q}, \mathbf{k}_j))} \in \mathbb{R}. \tag{10.3.2}$$

Gördüğümüz gibi, a dikkat puanlama fonksiyonunun farklı seçimleri farklı dikkat ortaklama davranışlarına yol açar. Bu bölümde, daha sonra daha gelişmiş dikkat mekanizmaları geliştirmek için kullanacağımız iki popüler puanlama fonksiyonunu tanıtıyoruz.

```
import math
import torch
from torch import nn
from d2l import torch as d2l
```

10.3.1 Maskeli Softmaks İşlemi

Daha önce de belirttiğimiz gibi, olasılık dağılımını dikkat ağırlıkları olarak elde etmek için bir softmaks işlemi kullanılır. Bazı durumlarda, tüm değerler dikkat ortaklamasına beslenmemelidir. Örneğin, Section 9.5 içinde verimli minigrup işleme için, bazı metin dizileri anlam taşımayan özel belirteçlerle doldurulmuştur. Değerler olarak yalnızca anlamlı belirteçler üzerinde bir dikkat ortaklamak için, softmaks hesaplarken bu belirtilen aralığın ötesinde olanları filtrelemek için geçerli bir sıra uzunluğu (belirteç sayısı olarak) belirtebiliriz. Bu şekilde, geçerli uzunluğun ötesinde herhangi bir değerin sıfır olarak maskelendiği aşağıdaki `masked_softmax` işlevinde böyle bir *maskelenmiş softmaks işlemi* uygulayabiliriz.

```
#@save
def masked_softmax(X, valid_lens):
    """Son eksendeki öğeleri maskeleyerek softmax işlemini gerçekleştirin."""
    # `X`: 3B tensör, `valid_lens`: 1B veya 2B tensör
    if valid_lens is None:
        return nn.functional.softmax(X, dim=-1)
    else:
        shape = X.shape
        if valid_lens.dim() == 1:
            valid_lens = torch.repeat_interleave(valid_lens, shape[1])
        else:
            valid_lens = valid_lens.reshape(-1)
        # Son eksende, maskelenmiş öğeleri, üsleri 0 olan çok büyük bir
        # negatif değerle değiştirin
        X = d2l.sequence_mask(X.reshape(-1, shape[-1]), valid_lens,
                              value=-1e6)
        return nn.functional.softmax(X.reshape(shape), dim=-1)
```

Bu işlevin nasıl çalıştığını göstermek için, bu iki örnek için geçerli uzunlukların sırasıyla iki ve üç olduğu iki tane 2×4 matris örneğinden oluşan bir minigrup düşünün. Maskelenmiş softmaks işleminin bir sonucu olarak, geçerli uzunlukların dışındaki değerlerin tümü sıfır olarak maskelenir.

```
masked_softmax(torch.rand(2, 2, 4), torch.tensor([2, 3]))
```

```
tensor([[[0.4434, 0.5566, 0.0000, 0.0000],
         [0.5291, 0.4709, 0.0000, 0.0000]],

        [[0.4512, 0.2650, 0.2839, 0.0000],
         [0.2631, 0.5151, 0.2218, 0.0000]]])
```

Benzer şekilde, her matris örneğindeki her satır için geçerli uzunlukları belirtmede iki boyutlu bir tensör de kullanabiliriz.

```
masked_softmax(torch.rand(2, 2, 4), torch.tensor([[1, 3], [2, 4]]))
```

```
tensor([[[1.0000, 0.0000, 0.0000, 0.0000],
         [0.3063, 0.3772, 0.3165, 0.0000]],

        [[0.3269, 0.6731, 0.0000, 0.0000],
         [0.1736, 0.3564, 0.1866, 0.2834]]])
```

10.3.2 Toplayıcı Dikkat

Genel olarak, sorgular ve anahtarlar farklı uzunluklarda vektörler olduğunda, puanlama işlevi olarak toplayıcı dikkat kullanabiliriz. Bir sorgu $\mathbf{q} \in \mathbb{R}^q$ ve bir anahtar $\mathbf{k} \in \mathbb{R}^k$, *toplayıcı dikkat* puanlama fonksiyonu göz önüne alındığında şöyledir:

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{w}_v^\top \tanh(\mathbf{W}_q\mathbf{q} + \mathbf{W}_k\mathbf{k}) \in \mathbb{R}, \tag{10.3.3}$$

burada $\mathbf{W}_q \in \mathbb{R}^{h\times q}$, $\mathbf{W}_k \in \mathbb{R}^{h\times k}$ ve $\mathbf{w}_v \in \mathbb{R}^h$ öğrenilebilir parametrelerdir. (10.3.3) denklemine eşdeğer olarak, sorgu ve anahtar bitiştirilir ve gizli birim sayısı h hiper parametresi olan bir tek gizli katmanlı MLP'ye beslenir. Etkinleştirme fonksiyonu olarak tanh'yi kullanarak ve ek girdi terimlerini devre dışı bırakarak, aşağıdakilere toplayıcı dikkat uyguluyoruz.

```
#@save
class AdditiveAttention(nn.Module):
    """Additive attention."""
    def __init__(self, key_size, query_size, num_hiddens, dropout, **kwargs):
        super(AdditiveAttention, self).__init__(**kwargs)
        self.W_k = nn.Linear(key_size, num_hiddens, bias=False)
        self.W_q = nn.Linear(query_size, num_hiddens, bias=False)
        self.w_v = nn.Linear(num_hiddens, 1, bias=False)
        self.dropout = nn.Dropout(dropout)

    def forward(self, queries, keys, values, valid_lens):
        queries, keys = self.W_q(queries), self.W_k(keys)
        # Boyut genişletmeden sonra, `queries`'in şekli: (`batch_size`,
        # sorgu sayısı, 1, `num_hiddens`) ve `keys`'in şekli: (`batch_size`, 1,
        # anahtar/değer çiftlerinin sayısı,`num_hiddens`).
        # Yayınlarken onları toplayın.
        features = queries.unsqueeze(2) + keys.unsqueeze(1)
        features = torch.tanh(features)
        # `self.w_v`nin sadece bir çıktısı var, bu yüzden şekilden
        # son tek boyutlu girişi kaldırıyoruz. `scores`'ın şekli (`batch_size`,
        # sorgu sayısı, anahtar/değer çifti sayısı)
        scores = self.w_v(features).squeeze(-1)
        self.attention_weights = masked_softmax(scores, valid_lens)
        # `values`'in şekli: (`batch_size`, anahtar/değer çiftlerinin sayısı,
        # değer boyutu)
        return torch.bmm(self.dropout(self.attention_weights), values)
```

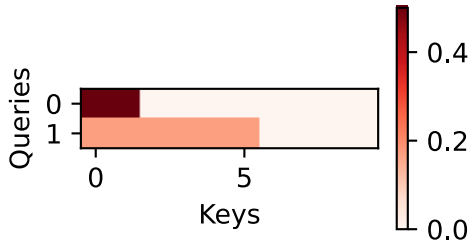
Yukarıdaki 'AdditiveAttention' sınıfını sorguların, anahtarların ve değerlerin şekillerinin (toplu iş boyutu, adım sayısı veya belirteçlerdeki sıra uzunluğu, öznitelik boyutu) sırasıyla (2, 1, 20), (2, 10, 2) ve (2, 10, 4) olduğu bir basit örnek ile gösterelim.

```
queries, keys = torch.normal(0, 1, (2, 1, 20)), torch.ones((2, 10, 2))
# `values` minigrubundaki iki değer matrisi aynıdır
values = torch.arange(40, dtype=torch.float32).reshape(1, 10, 4).repeat(
    2, 1, 1)
valid_lens = torch.tensor([2, 6])

attention = AdditiveAttention(key_size=2, query_size=20, num_hiddens=8,
                              dropout=0.1)
attention.eval()
attention(queries, keys, values, valid_lens)
```

```
tensor([[[ 2.0000,  3.0000,  4.0000,  5.0000]],

        [[10.0000, 11.0000, 12.0000, 13.0000]]], grad_fn=<BmmBackward0>)
```

Her ne kadar toplayıcı dikkat öğrenilebilir parametreler içerse de, bu örnekte her anahtar aynı olduğundan, önceden düzenlenmiş geçerli uzunluklara göre belirlenen dikkat ağırlıkları tekdüzedir.

```
d2l.show_heatmaps(attention.attention_weights.reshape((1, 1, 2, 10)),
                  xlabel='Keys', ylabel='Queries')
```



10.3.3 Ölçeklendirilmiş Nokta Çarpımı Dikkati

Puanlama fonksiyonu için hesaplama açısından daha verimli bir tasarım basitçe nokta çarpımı olabilir. Ancak, nokta çarpımı işlemi hem sorgu hem de anahtarın aynı vektör uzunluğuna sahip olmasını gerektirir, yani d. Sorgu ve anahtarın tüm öğelerinin sıfır ortalama ve birim varyanslı bağımsız rasgele değişkenler olduğunu varsayalım. Her iki vektörün nokta çarpımının sıfır ortalama ve d varyansı vardır. Nokta çarpımının varyansının vektör uzunluğundan bağımsız olarak hala bir kalmasını sağlamak için *ölçeklendirilmiş nokta çarpımı dikkati* puanlama işlevi

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{q}^\top \mathbf{k}/\sqrt{d} \tag{10.3.4}$$

nokta çarpımını $\sqrt{d}$ ile böler. Uygulamada, genellikle verimlilik için, sorguların ve anahtarların d uzunluğunda ve değerlerin v uzunluğunda olduğu n adet sorgu ve m adet anahtar/değer çifti için dikkat hesaplama gibi, minigruplar halinde düşünürüz. $\mathbf{Q} \in \mathbb{R}^{n\times d}$ sorgularının, $\mathbf{K} \in \mathbb{R}^{m\times d}$ anahtarlarının ve $\mathbf{V} \in \mathbb{R}^{m\times v}$ değerlerinin ölçeklendirilmiş nokta çarpımı dikkati şöyledir

$$\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)\mathbf{V} \in \mathbb{R}^{n\times v}. \tag{10.3.5}$$

Aşağıdaki ölçeklendirilmiş nokta çarpımı dikkati uygulamasında, model düzenlileştirmesi için hattan düşürme kullanıyoruz.

```
#@save
class DotProductAttention(nn.Module):
    """Ölçeklendirilmiş nokta çarpım dikkati."""
    def __init__(self, dropout, **kwargs):
        super(DotProductAttention, self).__init__(**kwargs)
        self.dropout = nn.Dropout(dropout)

    # `queries`'in şekli: (`batch_size`, sorgu sayısı, `d`)
```

(continues on next page)

(continued from previous page)

```
    # `keys`'in şekli: (`batch_size`, anahtar/değer çiftlerinin sayısı, `d`)
    # `values`'un şekli: (`batch_size`, anahtar/değer çiftlerinin sayısı, değer
    # boyut)
    # `valid_lens`'in şekli: (`batch_size`,) veya (`batch_size`, sorgu sayısı)
    def forward(self, queries, keys, values, valid_lens=None):
        d = queries.shape[-1]
        # keys`'in son iki boyutunu değiştirmek için "transpose_b=True`
        # diye ayarlayın
        scores = torch.bmm(queries, keys.transpose(1,2)) / math.sqrt(d)
        self.attention_weights = masked_softmax(scores, valid_lens)
        return torch.bmm(self.dropout(self.attention_weights), values)
```

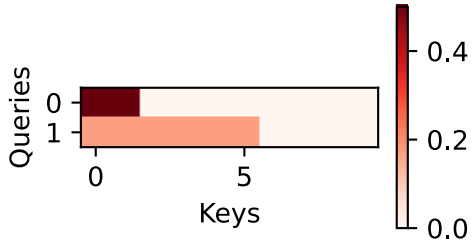
Yukarıdaki `DotProductAttention` sınıfını göstermek için, toplayıcı dikkat için önceki basit örnekteki aynı anahtarları, değerleri ve geçerli uzunlukları kullanıyoruz. Nokta çarpımı işlemi için, sorguların öznitelik boyutunu anahtarlarınkiyle aynı yapıyoruz.

```
queries = torch.normal(0, 1, (2, 1, 2))
attention = DotProductAttention(dropout=0.5)
attention.eval()
attention(queries, keys, values, valid_lens)
```

```
tensor([[[ 2.0000,  3.0000,  4.0000,  5.0000]],

        [[10.0000, 11.0000, 12.0000, 13.0000]]])
```

Toplayıcı dikkat gösterimindeki gibi, `keys` herhangi bir sorgu ile ayırt edilemeyen aynı elemanı içerdiğinden, tekdüze dikkat ağırlıkları elde edilir.

```
d2l.show_heatmaps(attention.attention_weights.reshape((1, 1, 2, 10)),
                  xlabel='Keys', ylabel='Queries')
```



10.3.4 Özet

- Dikkat puanlama fonksiyonunun farklı seçeneklerinin dikkat ortaklamasının farklı davranışlarına yol açtığı ağırlıklı bir değer ortalaması olarak dikkat ortaklama çıktısını hesaplayabiliriz.
- Sorgular ve anahtarlar farklı uzunluklarda vektörler olduğunda, toplayıcı dikkat puanlama işlevini kullanabiliriz. Aynı olduklarında, ölçeklendirilmiş nokta çarpımı dikkat puanlama işlevi hesaplama açısından daha verimlidir.

10.3.5 Alıştırmalar

1. Basit örnekteki anahtarları değiştirin ve dikkat ağırlıklarını görselleştirin. Toplayıcı dikkat ve ölçeklendirilmiş nokta çarpımı dikkati hala aynı dikkat ağırlıklarını veriyor mu? Neden ya da neden değil?
2. Yalnızca matris çarpımlarını kullanarak, farklı vektör uzunluklarına sahip sorgular ve anahtarlar için yeni bir puanlama işlevi tasarlayabilir misiniz?
3. Sorgular ve anahtarlar aynı vektör uzunluğuna sahip olduğunda, vektör toplamı puanlama fonksiyonu için nokta çarpımından daha iyi bir tasarım mıdır? Neden ya da neden değil?

Tartışmalar[122]

10.4 Bahdanau Dikkati

Section 9.7 içinde makine çevirisi problemini inceledik, burada diziden diziye öğrenme için iki RNN temelli bir kodlayıcı-kodçözücü mimarisi tasarladık. Özellikle, RNN kodlayıcısı bir değişken uzunluktaki diziyi sabit şekilli bağlam değişkenine dönüştürür, daha sonra RNN kodçözücüsü, üretilen belirteçlere ve bağlam değişkenine göre belirteç belirteç çıktı (hedef) dizisini oluşturur. Ancak, tüm girdi (kaynak) belirteçleri belirli bir belirteci kodlamak için yararlı olmasa da, girdi dizisinin tamamını kodlayan *aynı* bağlam değişkeni hala her kod çözme adımında kullanılır.

Belirli bir metin dizisi için el yazısı oluşturmayla ilgili ayrı ama bağlantılı bir zorlukta Graves, metin karakterlerini çok daha uzun kalem iziyle hizalamak için türevlenebilir bir dikkat modeli tasarladı; burada hizalama yalnızca bir yönde hareket eder (Graves, 2013). Hizalamayı öğrenme fikrinden esinlenen Bahdanau ve ark. (Bahdanau *et al.*, 2014) keskin bir tek yönlü hizalama sınırlaması olmaksızın türevlenebilir bir dikkat modeli önerdi. Bir belirteci tahmin ederken, tüm girdi belirteçleri ilgili değilse, model yalnızca girdi dizisinin geçerli tahminle ilgili bölümlerine hizalar (veya eşler). Bu, bağlam değişkeninin dikkat ortaklamasının bir çıktısı olarak ele alınarak elde edilir.

10.4.1 Model

Aşağıdaki RNN kodlayıcı-kodçözücüsü için Bahdanau dikkatini açıklarken, Section 9.7 içindeki aynı notasyonu takip edeceğiz. Yeni dikkat temelli model, (9.7.3) içindeki $\mathbf{c}$ bağlam değişkeninin herhangi bir t' kod çözme zaman adımında $\mathbf{c}_{t'}$ ile değiştirilmesi dışında Section 9.7 içindekiyle aynıdır. Girdi dizisinde T belirteçleri olduğunu varsayalım, kod çözme zamanı adımındaki bağlam değişkeni t' dikkat ortaklamasının çıktısıdır:

$$\mathbf{c}_{t'} = \sum_{t=1}^{T} \alpha(\mathbf{s}_{t'-1}, \mathbf{h}_t)\mathbf{h}_t, \tag{10.4.1}$$

burada zaman adım $t'-1$'deki kodçözücü gizli durumu $\mathbf{s}_{t'-1}$ sorgudur ve kodlayıcı gizli durumları $\mathbf{h}_t$ hem anahtarlar hem de değerlerdir ve α dikkat ağırlığı (10.3.2) ile tanımlanan toplayıcı dikkat puanlama işlevini kullanarak (10.3.2) içinde olduğu gibi hesaplanır.

Fig. 9.7.2 şeklindeki sıradan RNN kodlayıcı-kodçözücü mimarisinden biraz farklı olan Bahdanau dikkati ile aynı mimari Fig. 10.4.1 şeklinde tasvir edilmiştir.

[122] https://discuss.d2l.ai/t/1064

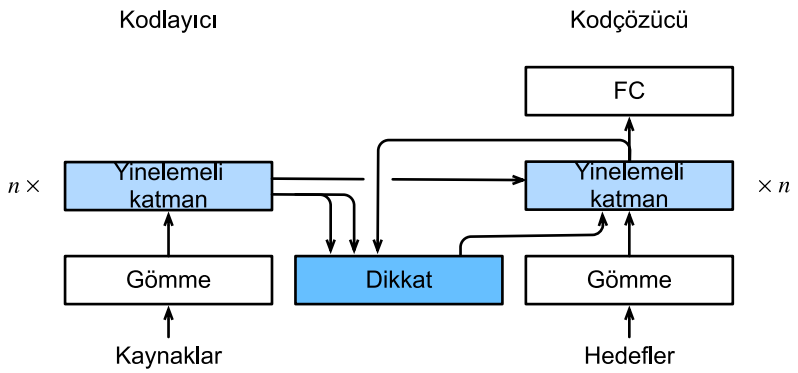


Fig. 10.4.1: Bahdanau dikkatli bir RNN kodlayıcı-kodçözücü modelindeki katmanlar.

```
import torch
from torch import nn
from d2l import torch as d2l
```

10.4.2 Çözücüyü Dikkat ile Tanımlama

RNN kodlayıcı-kodçözücüyü Bahdanau dikkati ile uygulamak için, sadece kodçözücüyü yeniden tanımlamamız gerekiyor. Öğrenilen dikkat ağırlıklarını daha rahat görselleştirmek için, aşağıdaki `AttentionDecoder` sınıfı dikkat mekanizmalarına sahip kodçözücüler için temel arabirimi tanımlar.

```
#@save
class AttentionDecoder(d2l.Decoder):
    """Temel dikkat tabanlı kodçözücü arabirimi."""
    def __init__(self, **kwargs):
        super(AttentionDecoder, self).__init__(**kwargs)

    @property
    def attention_weights(self):
        raise NotImplementedError
```

Şimdi, aşağıdaki `Seq2SeqAttentionDecoder` sınıfında RNN kodçözücüyü Bahdanau dikkatiyle uygulayalım. Kodçözücünün durumu, (i) tüm zaman adımlarında kodlayıcı son katman gizli durumları (dikkat anahtarları ve değerleri olarak); (ii) son zaman adımında kodlayıcı tüm katman gizli durumu (kod çözücünün gizli durumunu ilklemek için); ve (iii) geçerli kodlayıcı uzunluğu (dikkat ortaklamasındaki dolgu belirteçlerini hariç tutmak için) ile ilklenir. Her kod çözme zaman adımında, kod çözücü son katman gizli durumu önceki zaman adımında dikkat sorgusu olarak kullanılır. Sonuç olarak, hem dikkat çıktısı hem de girdi gömmesi RNN kod çözücünün girdisi olarak bitiştirilir.

```
class Seq2SeqAttentionDecoder(AttentionDecoder):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0, **kwargs):
        super(Seq2SeqAttentionDecoder, self).__init__(**kwargs)
        self.attention = d2l.AdditiveAttention(
            num_hiddens, num_hiddens, num_hiddens, dropout)
```

(continues on next page)

(continued from previous page)

```
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = nn.GRU(
            embed_size + num_hiddens, num_hiddens, num_layers,
            dropout=dropout)
        self.dense = nn.Linear(num_hiddens, vocab_size)

    def init_state(self, enc_outputs, enc_valid_lens, *args):
        # `outputs`'un şekli: (`num_steps`, `batch_size`, `num_hiddens`).
        # `hidden_state[0]`'in şekli: (`num_layers`, `batch_size`,
        # `num_hiddens`)
        outputs, hidden_state = enc_outputs
        return (outputs.permute(1, 0, 2), hidden_state, enc_valid_lens)

    def forward(self, X, state):
        # `enc_outputs`'un şekli: (`batch_size`, `num_steps`, `num_hiddens`).
        # `hidden_state[0]`'in şekli: (`num_layers`, `batch_size`,
        # `num_hiddens`)
        enc_outputs, hidden_state, enc_valid_lens = state
        # `X` çıktısının şekli: (`num_steps`, `batch_size`, `embed_size`)
        X = self.embedding(X).permute(1, 0, 2)
        outputs, self._attention_weights = [], []
        for x in X:
            # `query`'in şekli: (`batch_size`, 1, `num_hiddens`)
            query = torch.unsqueeze(hidden_state[-1], dim=1)
            # `context`'in şekli: (`batch_size`, 1, `num_hiddens`)
            context = self.attention(
                query, enc_outputs, enc_outputs, enc_valid_lens)
            # Öznitelik boyutunda bitiştir
            x = torch.cat((context, torch.unsqueeze(x, dim=1)), dim=-1)
            # `x`'i (1, `batch_size`, `embed_size` + `num_hiddens`) olarak
            # yeniden şekillendirin
            out, hidden_state = self.rnn(x.permute(1, 0, 2), hidden_state)
            outputs.append(out)
            self._attention_weights.append(self.attention.attention_weights)
        # Tam bağlı katman dönüşümünden sonra, `outputs` (çıktılar) şekli:
        # (`num_steps`, `batch_size`, `vocab_size`)
        outputs = self.dense(torch.cat(outputs, dim=0))
        return outputs.permute(1, 0, 2), [enc_outputs, hidden_state,
                                          enc_valid_lens]

    @property
    def attention_weights(self):
        return self._attention_weights
```

Aşağıda, 7 zaman adımlı 4 dizi girdisinden oluşan bir minigrup kullanarak Bahdanau dikkatiyle uygulanan kodçözücüyü test ediyoruz.

```
encoder = d2l.Seq2SeqEncoder(vocab_size=10, embed_size=8, num_hiddens=16,
                             num_layers=2)
encoder.eval()
decoder = Seq2SeqAttentionDecoder(vocab_size=10, embed_size=8, num_hiddens=16,
                                  num_layers=2)
decoder.eval()
X = torch.zeros((4, 7), dtype=torch.long)  # (`batch_size`, `num_steps`)
```

(continues on next page)

(continued from previous page)

```
state = decoder.init_state(encoder(X), None)
output, state = decoder(X, state)
output.shape, len(state), state[0].shape, len(state[1]), state[1][0].shape
```

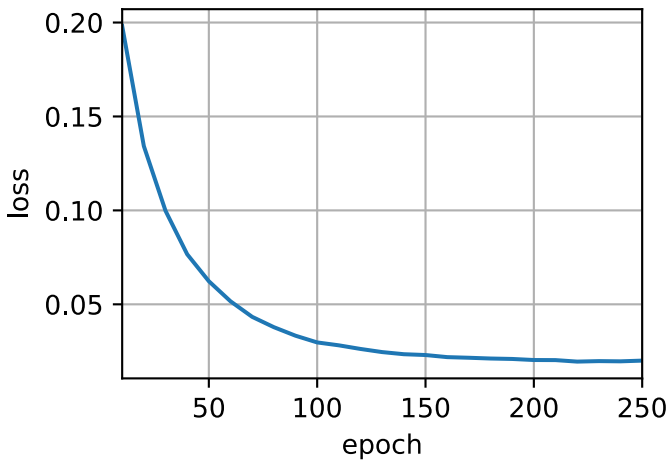
```
(torch.Size([4, 7, 10]), 3, torch.Size([4, 7, 16]), 2, torch.Size([4, 16]))
```

10.4.3 Eğitim

Section 9.7.4 içindekine benzer şekilde, burada hiper parametreleri belirtiyoruz, Bahdanau dikkatli bir kodlayıcı ve bir kodçözücü oluşturuyor ve bu modeli makine çevirisi için eğitiyoruz. Yeni eklenen dikkat mekanizması nedeniyle, bu eğitim Section 9.7.4 içindeki dikkat mekanizmaları olmadan çok daha yavaştır.

```
embed_size, num_hiddens, num_layers, dropout = 32, 32, 2, 0.1
batch_size, num_steps = 64, 10
lr, num_epochs, device = 0.005, 250, d2l.try_gpu()

train_iter, src_vocab, tgt_vocab = d2l.load_data_nmt(batch_size, num_steps)
encoder = d2l.Seq2SeqEncoder(
    len(src_vocab), embed_size, num_hiddens, num_layers, dropout)
decoder = Seq2SeqAttentionDecoder(
    len(tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
net = d2l.EncoderDecoder(encoder, decoder)
d2l.train_seq2seq(net, train_iter, lr, num_epochs, tgt_vocab, device)
```

```
loss 0.020, 5440.8 tokens/sec on cuda:0
```



Model eğitildikten sonra, onu birkaç İngilizce cümleyi Fransızca'ya çevirmek ve BLEU değerlerini hesaplamak için kullanıyoruz.

```
engs = ['go .', "i lost .", 'he\'s calm .', 'i\'m home .']
fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']
```

(continues on next page)

(continued from previous page)

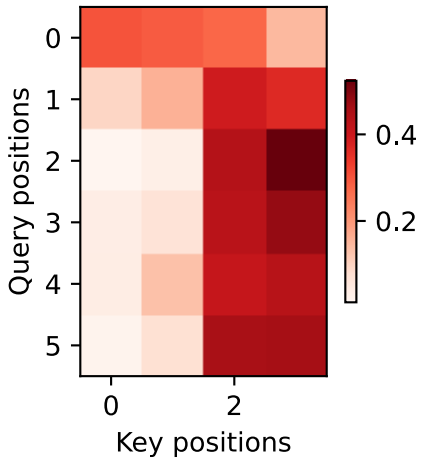
```
for eng, fra in zip(engs, fras):
    translation, dec_attention_weight_seq = d2l.predict_seq2seq(
        net, eng, src_vocab, tgt_vocab, num_steps, device, True)
    print(f'{eng} => {translation}, ',
          f'bleu {d2l.bleu(translation, fra, k=2):.3f}')
```

```
go . => va !,  bleu 1.000
i lost . => j'ai perdu .,  bleu 1.000
he's calm . => il est mouillé .,  bleu 0.658
i'm home . => je suis chez moi .,  bleu 1.000
```

```
attention_weights = torch.cat([step[0][0][0] for step in dec_attention_weight_seq], 0).
↪reshape((
    1, 1, -1, num_steps))
```

Son İngilizce cümleyi çevirirken dikkat ağırlıklarını görselleştirerek, her sorgunun anahtar değer çiftleri üzerinde tekdüze olmayan ağırlıklar atadığını görebiliriz. Her kod çözme adımında, girdi dizilerinin farklı bölümlerinin dikkat ortaklamasında seçici olarak toplandığını gösterir.

```
# Sıra sonu belirtecini eklemek için bir tane ekle
d2l.show_heatmaps(
    attention_weights[:, :, :, :len(engs[-1].split()) + 1].cpu(),
    xlabel='Key positions', ylabel='Query positions')
```



10.4.4 Özet

- Bir belirteci tahmin ederken, tüm girdi belirteçleri ilgili değilse, Bahdanau dikkatine sahip RNN kodlayıcı-kodçözücü seçici olarak girdi dizisinin farklı bölümlerini toplar. Bu, bağlam değişkeninin toplayıcı dikkat ortaklamasının bir çıktısı olarak ele alınarak elde edilir.
- RNN kodlayıcı-kodçözücüsünde, Bahdanau dikkati önceki zaman adımındaki kodçözücü gizli durumunu sorgu olarak ve kodlayıcı gizli durumlarını her zaman adımında hem anahtar hem de değerler olarak ele alır.

10.4.5 Alıştırmalar

1. Deneyde GRU'yu LSTM ile değiştirin.
2. Toplayıcı dikkat puanlama işlevini ölçeklendirilmiş nokta çarpımı ile değiştirerek deneyi tekrarlayın. Eğitim verimliliğini nasıl etkiler?

Tartışmalar[123]

10.5 Çoklu-Kafalı Dikkat

Pratikte, aynı sorgular, anahtarlar ve değerler kümesi verildiğinde, modelimizin, çeşitli aralıkların bir sıra içinde (örneğin, daha kısa menzile karşı daha uzun menzil) bağımlılıklarını yakalama gibi aynı dikkat mekanizmasının farklı davranışlarından elde edilen bilgileri birleştirmesini isteyebiliriz. Bu nedenle, dikkat mekanizmamızın sorguların, anahtarların ve değerlerin farklı temsil alt alanlarını ortaklaşa kullanmasına izin vermek yararlı olabilir.

Bu amaçla, tek bir dikkat ortaklaması yerine, sorgular, anahtarlar ve değerler h tane bağımsız olarak öğrenilen doğrusal izdüşümler ile dönüştürülebilir. Daha sonra bu h öngörülen sorgular, anahtarlar ve değerler paralel olarak dikkat ortaklaması içine beslenir. Nihayetinde, h dikkat ortaklama çıktıları bitiştirilir ve son çıktıyı üretmek için başka bir öğrenilmiş doğrusal izdüşüm ile dönüştürülür. Bu tasarıma *çoklu kafalı dikkat* denir, burada h dikkat ortaklama çıktılarının her biri *kafa*dır (Vaswani *et al.*, 2017). Öğrenilebilir doğrusal dönüşümler gerçekleştirmek için tam bağlı katmanları kullanan çoklu kafalı dikkat Fig. 10.5.1 şeklinde açıklanmıştır.

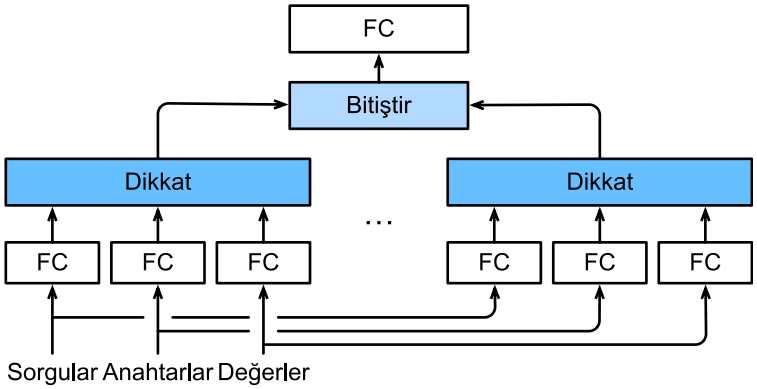


Fig. 10.5.1: Çoklu kafanın bir araya getirildiği ve ardından doğrusal olarak dönüştürüldüğü çoklu kafalı dikkat.

10.5.1 Model

Çoklu kafalı dikkatin uygulanmasını sağlamadan önce, bu modeli matematiksel olarak biçimlendirelim. Bir sorgu $\mathbf{q} \in \mathbb{R}^{d_q}$, bir anahtar $\mathbf{k} \in \mathbb{R}^{d_k}$ ve bir değer $\mathbf{v} \in \mathbb{R}^{d_v}$ göz önüne alındığında, her dikkat kafası $\mathbf{h}_i$ ($i = 1, \ldots, h$) aşağıdaki gibi hesaplanır

$$\mathbf{h}_i = f(\mathbf{W}_i^{(q)}\mathbf{q}, \mathbf{W}_i^{(k)}\mathbf{k}, \mathbf{W}_i^{(v)}\mathbf{v}) \in \mathbb{R}^{p_v}, \tag{10.5.1}$$

burada öğrenilebilir parametreler $\mathbf{W}_i^{(q)} \in \mathbb{R}^{p_q \times d_q}$, $\mathbf{W}_i^{(k)} \in \mathbb{R}^{p_k \times d_k}$ ve $\mathbf{W}_i^{(v)} \in \mathbb{R}^{p_v \times d_v}$ ve f, Section 10.3 içindeki toplayıcı dikkat ve ölçeklendirilmiş nokta çarpımı dikkat gibi dikkat ortaklamasıdır.

[123] https://discuss.d2l.ai/t/1065

Çoklu kafalı dikkat çıktısı, h kafalarının bitiştirilmesinin $\mathbf{W}_o \in \mathbb{R}^{p_o \times hp_v}$ öğrenilebilir parametreleri vasıtasıyla başka bir doğrusal dönüşümdür:

$$\mathbf{W}_o \begin{bmatrix} \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_h \end{bmatrix} \in \mathbb{R}^{p_o}. \tag{10.5.2}$$

Bu tasarıma dayanarak, her kafa girdisinin farklı bölümleriyle ilgilenebilir. Basit ağırlıklı ortalamadan daha gelişmiş fonksiyonlar ifade edilebilir.

```
import math
import torch
from torch import nn
from d2l import torch as d2l
```

10.5.2 Uygulama

Uygulamamızda, çoklu kafalı dikkatin her bir kafa için ölçeklendirilmiş nokta-çarpımı dikkatini seçiyoruz. Hesaplama maliyetinde ve parametreleştirme maliyetinde önemli bir artıştan kaçınmak için $p_q = p_k = p_v = p_o/h$ olarak ayarladık. Sorgu, anahtar ve değer için doğrusal dönüşümlerin çıktı sayısını $p_q h = p_k h = p_v h = p_o$ olarak ayarlarsak, h adet kafanın paralel olarak hesaplanabileceğini unutmayın. Aşağıdaki uygulamada, p_o, `num_hiddens` bağımsız değişkeni aracılığıyla belirtilir.

```
#@save
class MultiHeadAttention(nn.Module):
    """Multi-head attention."""
    def __init__(self, key_size, query_size, value_size, num_hiddens,
                 num_heads, dropout, bias=False, **kwargs):
        super(MultiHeadAttention, self).__init__(**kwargs)
        self.num_heads = num_heads
        self.attention = d2l.DotProductAttention(dropout)
        self.W_q = nn.Linear(query_size, num_hiddens, bias=bias)
        self.W_k = nn.Linear(key_size, num_hiddens, bias=bias)
        self.W_v = nn.Linear(value_size, num_hiddens, bias=bias)
        self.W_o = nn.Linear(num_hiddens, num_hiddens, bias=bias)

    def forward(self, queries, keys, values, valid_lens):
        # `queries`, `keys`, veya `values` şekli:
        # (`batch_size`,  anahtar-değer çiftleri veya sorgu sayısı, `num_hiddens`)
        # `valid_lens`'in şekli:
        # (`batch_size`,) or (`batch_size`, no. of queries)
        # Devirme sonrası, output `queries`, `keys`, veya `values` şekli:
        # (`batch_size` * `num_heads`,  anahtar-değer çiftleri veya sorgu sayısı,
        # `num_hiddens` / `num_heads`)
        queries = transpose_qkv(self.W_q(queries), self.num_heads)
        keys = transpose_qkv(self.W_k(keys), self.num_heads)
        values = transpose_qkv(self.W_v(values), self.num_heads)

        if valid_lens is not None:
            # 0 ekseninde, ilk öğeyi (skaler veya vektör) `num_heads` kez
            # kopyalayın, ardından sonraki öğeyi kopyalayın ve devam edin.
            valid_lens = torch.repeat_interleave(
```

(continues on next page)

(continued from previous page)

```
            valid_lens, repeats=self.num_heads, dim=0)

        # `output`'un şekli: (`batch_size` * `num_heads`, no. of queries,
        # `num_hiddens` / `num_heads`)
        output = self.attention(queries, keys, values, valid_lens)

        # `output_concat`'in şekli:
        # (`batch_size`, sorgu sayısı, `num_hiddens`)
        output_concat = transpose_output(output, self.num_heads)
        return self.W_o(output_concat)
```

Çoklu kafanın paralel hesaplanmasına izin vermek için, yukarıdaki `MultiHeadAttention` sınıfı aşağıda tanımlandığı gibi iki devrinim işlevi kullanır. Özellikle, `transpose_output` işlevi `transpose_qkv` işlevinin çalışmasını tersine çevirir.

```
#@save
def transpose_qkv(X, num_heads):
    """Çoklu dikkat kafasının paralel hesaplaması için aktarım."""
    # `X` girdisinin şekli:
    # (`batch_size`, anahtar-değer çiftleri veya sorgu sayısı, `num_hiddens`).
    # `X` çıktısının şekli:
    # (`batch_size`,anahtar-değer çiftleri veya sorgu sayısı, `num_heads`,
    # `num_hiddens` / `num_heads`)
    X = X.reshape(X.shape[0], X.shape[1], num_heads, -1)

    # `X` çıktısının şekli:
    # (`batch_size`, `num_heads`, anahtar-değer çiftleri veya sorgu sayısı,
    # `num_hiddens` / `num_heads`)
    X = X.permute(0, 2, 1, 3)

    # `output`'un şekli:
    # (`batch_size` * `num_heads`, anahtar-değer çiftleri veya sorgu sayısı,
    # `num_hiddens` / `num_heads`)
    return X.reshape(-1, X.shape[2], X.shape[3])


#@save
def transpose_output(X, num_heads):
    """`transpose_qkv` işlemini tersine çevir."""
    X = X.reshape(-1, num_heads, X.shape[1], X.shape[2])
    X = X.permute(0, 2, 1, 3)
    return X.reshape(X.shape[0], X.shape[1], -1)
```

Anahtarların ve değerlerin aynı olduğu bir basit örneği kullanarak uygulanan `MultiHeadAttention` sınıfını test edelim. Sonuç olarak, çoklu kafalı dikkat çıktısının şekli (`batch_size`, `num_queries`, `num_hiddens`) şeklindedir.

```
num_hiddens, num_heads = 100, 5
attention = MultiHeadAttention(num_hiddens, num_hiddens, num_hiddens,
                               num_hiddens, num_heads, 0.5)
attention.eval()
```

```
MultiHeadAttention(
  (attention): DotProductAttention(
    (dropout): Dropout(p=0.5, inplace=False)
  )
  (W_q): Linear(in_features=100, out_features=100, bias=False)
  (W_k): Linear(in_features=100, out_features=100, bias=False)
  (W_v): Linear(in_features=100, out_features=100, bias=False)
  (W_o): Linear(in_features=100, out_features=100, bias=False)
)
```

```
batch_size, num_queries, num_kvpairs, valid_lens = 2, 4, 6, torch.tensor([3, 2])
X = torch.ones((batch_size, num_queries, num_hiddens))
Y = torch.ones((batch_size, num_kvpairs, num_hiddens))
attention(X, Y, Y, valid_lens).shape
```

```
torch.Size([2, 4, 100])
```

10.5.3 Özet

- Çoklu kafalı dikkat, sorguların, anahtarların ve değerlerin farklı temsil altuzayları aracılığıyla aynı dikkat ortaklama bilgisini birleştirir.
- Çoklu kafalı dikkatin çoklu kafasını paralel olarak hesaplamak için uygun tensör düzenlemeleri gereklidir.

10.5.4 Alıştırmalar

1. Bu deneydeki çoklu kafanın dikkat ağırlıklarını görselleştirin.
2. Çoklu kafa dikkatine dayalı eğitilmiş bir modelimiz olduğunu ve tahmin hızını artırmak için en az önemli dikkat kafalarını budamak istediğimizi varsayalım. Bir dikkat kafasının önemini ölçmek için deneyleri nasıl tasarlayabiliriz.

Tartışmalar[124]

10.6 Özdikkat ve Konumsal Kodlama

Derin öğrenmede, bir diziyi kodlamak için sıklıkla CNN'leri veya RNN'leri kullanırız. Şimdi dikkat mekanizmalarıyla, bir belirteç dizisini dikkat ortaklamasına beslediğimizi hayal edin, böylece aynı belirteç kümesi sorgular, anahtarlar ve değerler gibi davranır. Özellikle, her sorgu tüm anahtar değer çiftlerine kulak verir ve bir dikkat çıktısı oluşturur. Sorgular, anahtarlar ve değerler aynı yerden geldiğinden, *özdikkat* (Lin *et al.*, 2017, Vaswani *et al.*, 2017), aynı zamanda *içe-dikkat* (Cheng *et al.*, 2016, Parikh *et al.*, 2016, Paulus *et al.*, 2017) olarak da adlandırılır, gerçeklenir. Bu bölümde, dizi düzeni için ek bilgilerin kullanılması da dahil olmak üzere, özdikkat kullanan dizi kodlamasını tartışacağız.

[124] https://discuss.d2l.ai/t/1635

```
import math
import torch
from torch import nn
from d2l import torch as d2l
```

10.6.1 Özdikkat

$\mathbf{x}_1, \ldots, \mathbf{x}_n$ $(1 \leq i \leq n)$ herhangi bir $\mathbf{x}_i \in \mathbb{R}^d$ $(1 \leq i \leq n)$ girdi belirteçleri dizisi göz önüne alındığında, özdikkat aynı uzunlukta $\mathbf{y}_1, \ldots, \mathbf{y}_n$ olan bir dizi çıkarır, burada

$$\mathbf{y}_i = f(\mathbf{x}_i, (\mathbf{x}_1, \mathbf{x}_1), \ldots, (\mathbf{x}_n, \mathbf{x}_n)) \in \mathbb{R}^d \tag{10.6.1}$$

(10.3.1) denklemindeki f dikkat ortaklama tanımına göredir. Çoklu kafalı dikkati kullanarak, aşağıdaki kod parçacığı, (toplu iş boyutu, zaman adımlarının sayısı veya belirteçlerdeki dizi uzunluğu, d) şekline sahip bir tensörün özdikkatini hesaplar. Çıktı tensörü aynı şekle sahiptir.

```
num_hiddens, num_heads = 100, 5
attention = d2l.MultiHeadAttention(num_hiddens, num_hiddens, num_hiddens,
                                   num_hiddens, num_heads, 0.5)
attention.eval()
```

```
MultiHeadAttention(
  (attention): DotProductAttention(
    (dropout): Dropout(p=0.5, inplace=False)
  )
  (W_q): Linear(in_features=100, out_features=100, bias=False)
  (W_k): Linear(in_features=100, out_features=100, bias=False)
  (W_v): Linear(in_features=100, out_features=100, bias=False)
  (W_o): Linear(in_features=100, out_features=100, bias=False)
)
```

```
batch_size, num_queries, valid_lens = 2, 4, torch.tensor([3, 2])
X = torch.ones((batch_size, num_queries, num_hiddens))
attention(X, X, X, valid_lens).shape
```

```
torch.Size([2, 4, 100])
```

10.6.2 CNN'lerin, RNN'lerin ve Özdikkatin Karşılaştırılması

n tane belirtecin bir dizisini eşit uzunlukta başka bir diziyle eşlemek için mimarileri karşılaştıralım, burada her girdi veya çıktı belirteci d boyutlu bir vektör ile temsil edilir. Özellikle CNN'leri, RNN'leri ve özdikkati dikkate alacağız. Hesaplama karmaşıklığını, ardışık işlemleri ve maksimum yol uzunluklarını karşılaştıracağız. Ardışık işlemlerin paralel hesaplamayı engellediğini unutmayın, dizi konumlarının herhangi bir kombinasyonu arasındaki daha kısa bir yol dizi içindeki uzun menzilli bağımlılıkları öğrenmeyi kolaylaştırır (Hochreiter *et al.*, 2001).

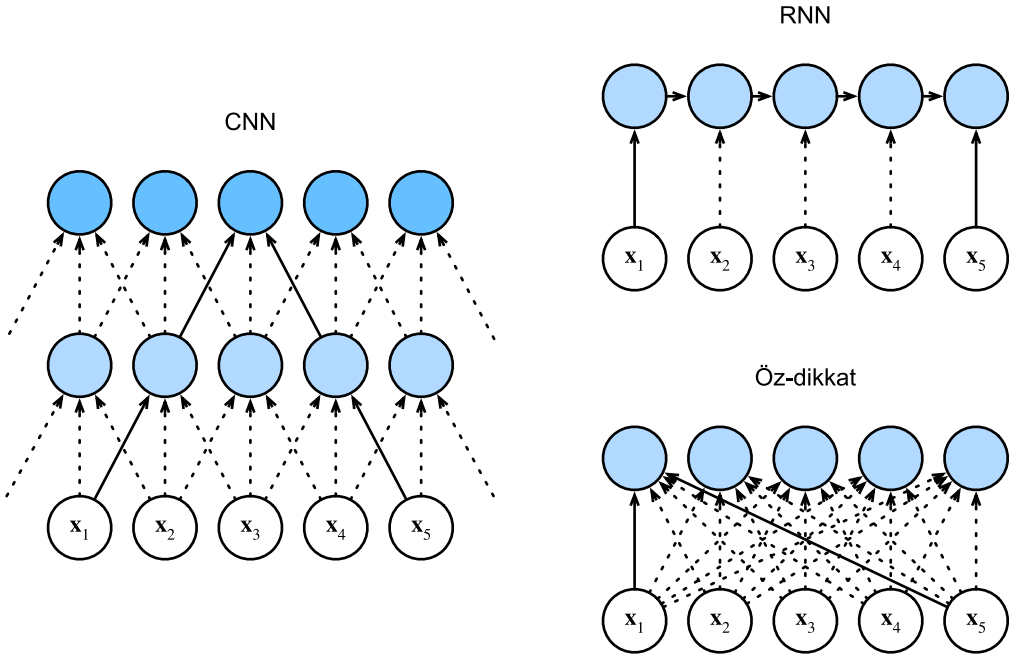


Fig. 10.6.1: CNN (dolgu belirteçleri atlanmıştır), RNN ve özdikkat mimarilerini karşılaştırma.

Çekirdek boyutu k olan bir evrişimli katman düşünün. Daha sonraki bölümlerde CNN'leri kullanarak dizi işleme hakkında daha fazla ayrıntı sağlayacağız. Şimdilik, sadece dizi uzunluğu n olduğundan, girdi ve çıktı kanallarının ikisinin de sayısı d olduğunu, evrişimli katmanın hesaplama karmaşıklığının $\mathcal{O}(knd^2)$ olduğunu bilmemiz gerekiyor. Fig. 10.6.1 şeklinin gösterdiği gibi, CNN'ler hiyerarşik olduğundan $\mathcal{O}(1)$ ardışık işlem vardır ve maksimum yol uzunluğu $\mathcal{O}(n/k)$'dir. Örneğin, $\mathbf{x}_1$ ve $\mathbf{x}_5$, Fig. 10.6.1 içinde çekirdek boyutu 3 olan iki katmanlı CNN'nin alıcı alanı içerisindedir.

RNN'lerin gizli durum güncellemesinde, $d \times d$ ağırlık matrisinin ve d boyutlu gizli durumun çarpımı $\mathcal{O}(d^2)$ hesaplama karmaşıklığına sahiptir. Dizi uzunluğu n olduğundan, yinelemeli katmanın hesaplama karmaşıklığı $\mathcal{O}(nd^2)$'dir. Fig. 10.6.1 şekline göre, paralelleştirilemeyen $\mathcal{O}(n)$ ardışık işlem vardır ve maksimum yol uzunluğu da $\mathcal{O}(n)$'dir.

Özdikkatte, sorgular, anahtarlar ve değerlerin tümü $n \times d$ matrislerdir. $n \times d$ matrisinin $d \times n$ matrisi ile çarpıldığı (10.3.5) denklemindeki ölçeklendirilmiş nokta çarpımı dikkatini düşünün, daha sonra $n \times n$ çıktı matrisi $n \times d$ matrisi ile çarpılır. Sonuç olarak, özdikkat $\mathcal{O}(n^2d)$ hesaplama karmaşıklığına sahiptir. Fig. 10.6.1 şeklinde görebileceğimiz gibi, her belirteç, özdikkat yoluyla başka bir belirteçle doğrudan bağlanır. Bu nedenle, hesaplama $\mathcal{O}(1)$ ardışık işlemle paralel olabilir ve maksimum yol uzunluğu da $\mathcal{O}(1)$'dir.

Sonuçta, hem CNN'ler hem de özdikkat paralel hesaplamanın keyfini çıkarır ve özdikkat en kısa maksimum yol uzunluğuna sahiptir. Bununla birlikte, dizi uzunluğuna göre ikinci dereceden hesaplama karmaşıklığı, özdikkati çok uzun diziler için engelleyici bir şekilde yavaşlatır.

10.6.3 Konumsal Kodlama

Bir dizinin belirteçlerini teker teker teker yinelemeli işleyen RNN'lerin aksine, özdikkat paralel hesaplama lehine ardışık işlemleri es geçer. Dizi düzeni bilgilerini kullanmak için girdi temsillerine *konumsal kodlama* ekleyerek mutlak veya göreceli konum bilgilerini aşılayabiliriz. Konumsal kodlamalar öğrenilebilir veya sabit olabilir. Aşağıda, sinüs ve kosinüs fonksiyonlarına dayanan sabit bir konumsal kodlamayı tanımlıyoruz (Vaswani *et al.*, 2017).

Girdi gösterimi $\mathbf{X} \in \mathbb{R}^{n \times d}$'nin n belirteçli bir dizi için d boyutlu gömmeler içerdiğini varsayalım. Konumsal kodlama, aynı şekle sahip bir konumsal yerleştirme matrisi $\mathbf{P} \in \mathbb{R}^{n \times d}$ kullanarak $i.$ satırında ve $(2j).$ veya $(2j+1).$ sütununda $\mathbf{X} + \mathbf{P}$ çıktısını verir;

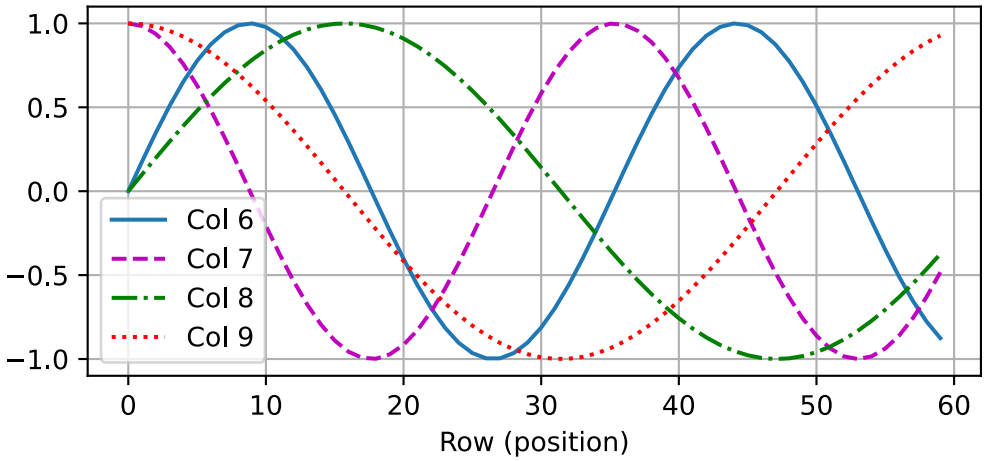
$$\begin{aligned} p_{i,2j} &= \sin\left(\frac{i}{10000^{2j/d}}\right), \\ p_{i,2j+1} &= \cos\left(\frac{i}{10000^{2j/d}}\right). \end{aligned} \tag{10.6.2}$$

İlk bakışta, bu trigonometrik fonksiyonlu tasarım garip görünüyor. Bu tasarımı açıklamadan önce, önce aşağıdaki `PositionalEncoding` sınıfında uygulayalım.

```
#@save
class PositionalEncoding(nn.Module):
    """Konumsal kodlama."""
    def __init__(self, num_hiddens, dropout, max_len=1000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(dropout)
        # Yeterince uzun `P` yarat
        self.P = torch.zeros((1, max_len, num_hiddens))
        X = torch.arange(max_len, dtype=torch.float32).reshape(
            -1, 1) / torch.pow(10000, torch.arange(
            0, num_hiddens, 2, dtype=torch.float32) / num_hiddens)
        self.P[:, :, 0::2] = torch.sin(X)
        self.P[:, :, 1::2] = torch.cos(X)

    def forward(self, X):
        X = X + self.P[:, :X.shape[1], :].to(X.device)
        return self.dropout(X)
```

Konumsal gömme matrisi $\mathbf{P}$'de, satırlar bir dizi içindeki konumlara karşılık gelir ve sütunlar farklı konum kodlama boyutlarını gösterir. Aşağıdaki örnekte, konumsal gömme matrisinin $6.$ ve $7.$ sütunlarının $8.$ ve $9.$ sütunlarından daha yüksek bir frekansa sahip olduğunu görebiliriz. $6.$ ve $7.$ ($8.$ ve $9.$ için aynı) sütunlar arasındaki uzaklık, sinüs ve kosinüs fonksiyonlarının değişmesinden kaynaklanmaktadır.

```
encoding_dim, num_steps = 32, 60
pos_encoding = PositionalEncoding(encoding_dim, 0)
pos_encoding.eval()
X = pos_encoding(torch.zeros((1, num_steps, encoding_dim)))
P = pos_encoding.P[:, :X.shape[1], :]
d2l.plot(torch.arange(num_steps), P[0, :, 6:10].T, xlabel='Row (position)',
         figsize=(6, 2.5), legend=["Col %d" % d for d in torch.arange(6, 10)])
```



Mutlak Konumsal Bilgiler

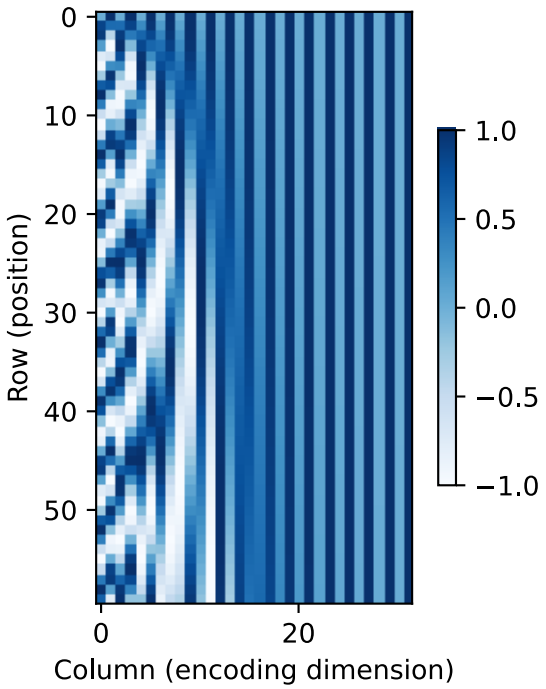
Kodlama boyutu boyunca monoton olarak azalan frekansın mutlak konumsal bilgilerle nasıl ilişkili olduğunu görmek için $0, 1, \ldots, 7$'nin ikilik temsilleri yazdıralım. Gördüğümüz gibi, en düşük bit, ikinci en düşük bit ve üçüncü en düşük bit her sayıda, her iki sayı ve her dört sayıda değişiyor.

```
for i in range(8):
    print(f'{i} in binary is {i:>03b}')
```

```
0 in binary is 000
1 in binary is 001
2 in binary is 010
3 in binary is 011
4 in binary is 100
5 in binary is 101
6 in binary is 110
7 in binary is 111
```

İkili temsillerde, daha yüksek bir bit, daha düşük bir bitten daha düşük bir frekansa sahiptir. Benzer şekilde, aşağıdaki ısı haritasında gösterildiği gibi, konumsal kodlama, trigonometrik fonksiyonlar kullanılarak kodlama boyutu boyunca frekansları azaltır. Çıktılar kayan virgüllü sayılar olduğundan, bu tür sürekli gösterimler ikili gösterimlerden daha fazla alan verimlidir.

```
P = P[0, :, :].unsqueeze(0).unsqueeze(0)
d2l.show_heatmaps(P, xlabel='Column (encoding dimension)',
                  ylabel='Row (position)', figsize=(3.5, 4), cmap='Blues')
```



Göreceli Konumsal Bilgiler

Mutlak konumsal bilgileri yakalamanın yanı sıra, yukarıdaki konumsal kodlama, bir modelin göreceli pozisyonlara göre dikkat kesilmeyi kolayca öğrenmesini de sağlar. Bunun nedeni, δ offset herhangi bir sabit pozisyon için $i + \delta$ konumundaki konumsal kodlamanın i konumundaki doğrusal bir izdüşümle temsil edilebilir olmasıdır.

Bu izdüşüm matematiksel olarak açıklanabilir. $\omega_j = 1/10000^{2j/d}$, (10.6.2) denkleminde herhangi bir $(p_{i,2j}, p_{i,2j+1})$ çifti doğrusal olarak herhangi bir sabit ofset δ için $(p_{i+\delta,2j}, p_{i+\delta,2j+1})$'ye izdüşürülebilir:

$$\begin{aligned}
&\begin{bmatrix} \cos(\delta\omega_j) & \sin(\delta\omega_j) \\ -\sin(\delta\omega_j) & \cos(\delta\omega_j) \end{bmatrix} \begin{bmatrix} p_{i,2j} \\ p_{i,2j+1} \end{bmatrix} \\
&= \begin{bmatrix} \cos(\delta\omega_j)\sin(i\omega_j) + \sin(\delta\omega_j)\cos(i\omega_j) \\ -\sin(\delta\omega_j)\sin(i\omega_j) + \cos(\delta\omega_j)\cos(i\omega_j) \end{bmatrix} \\
&= \begin{bmatrix} \sin\left((i+\delta)\omega_j\right) \\ \cos\left((i+\delta)\omega_j\right) \end{bmatrix} \\
&= \begin{bmatrix} p_{i+\delta,2j} \\ p_{i+\delta,2j+1} \end{bmatrix},
\end{aligned} \tag{10.6.3}$$

2×2 izdüşüm matrisi herhangi bir i konum endeksine bağlı değildir.

10.6.4 Özet

- Özdikkatinde, sorgular, anahtarlar ve değerler aynı yerden geliyor.
- Hem CNN'ler hem de özdikkat paralel hesaplamanın keyfini çıkarır ve özdikkat en kısa maksimum yol uzunluğuna sahiptir. Bununla birlikte, dizi uzunluğuna göre ikinci dereceden hesaplama karmaşıklığı, özdikkati çok uzun diziler için engelleyici bir şekilde yavaşlatır.
- Dizi düzeni bilgilerini kullanmak için, girdi temsillerine konumsal kodlama ekleyerek mutlak veya göreceli konum bilgilerini aşılayabiliriz.

10.6.5 Alıştırmalar

1. Konumsal kodlama ile özdikkat katmanlarını istifleyerek bir diziyi temsil edecek derin bir mimari tasarladığımızı varsayalım. Sorunlar ne olabilir?
2. Öğrenilebilir bir konumsal kodlama yöntemi tasarlayabilir misiniz?

Tartışmalar[125]

10.7 Dönüştürücü

Section 10.6.2 içinde CNN , RNN ve özdikkati karşılaştırdık. Özellikle, özdikkat hem paralel hesaplamanın hem de en kısa maksimum yol uzunluğunun keyfini sürer. Bu nedenle doğal olarak, özdikkat kullanarak derin mimariler tasarlamak caziptir. Girdi temsilleri için RNN'lere güvenen önceki özdikkat modellerinin aksine (Cheng *et al.*, 2016, Lin *et al.*, 2017, Paulus *et al.*, 2017), dönüştürücü modeli (Vaswani *et al.*, 2017) sadece herhangi bir evrişimli veya yinelemeli tabaka olmadan dikkat mekanizmalarına dayanmaktadır. Başlangıçta metin verilerinde diziden diziye öğrenme için önerilmiş olsa da, dönüştürücüler dil, görme, konuşma ve pekiştirmeli öğrenme alanlarında olduğu gibi çok çeşitli modern derin öğrenme uygulamalarında yaygın olmuştur.

10.7.1 Model

Kodlayıcı-kodçözücü mimarisinin bir örneği olarak, dönüştürücünün genel mimarisi Fig. 10.7.1 içinde sunulmuştur. Gördüğümüz gibi, dönüştürücü bir kodlayıcı ve bir kodçözücüden oluşur. Fig. 10.4.1 içinde diziden diziye öğrenmede Bahdanau dikkatinden farklı olarak, girdi (kaynak) ve çıktı (hedef) dizi gömmeleri, özdikkate dayalı modülleri istifleyen kodlayıcıya ve kodçözücüye beslenmeden önce, konumsal kodlama ile toplanır.

[125] https://discuss.d2l.ai/t/1652

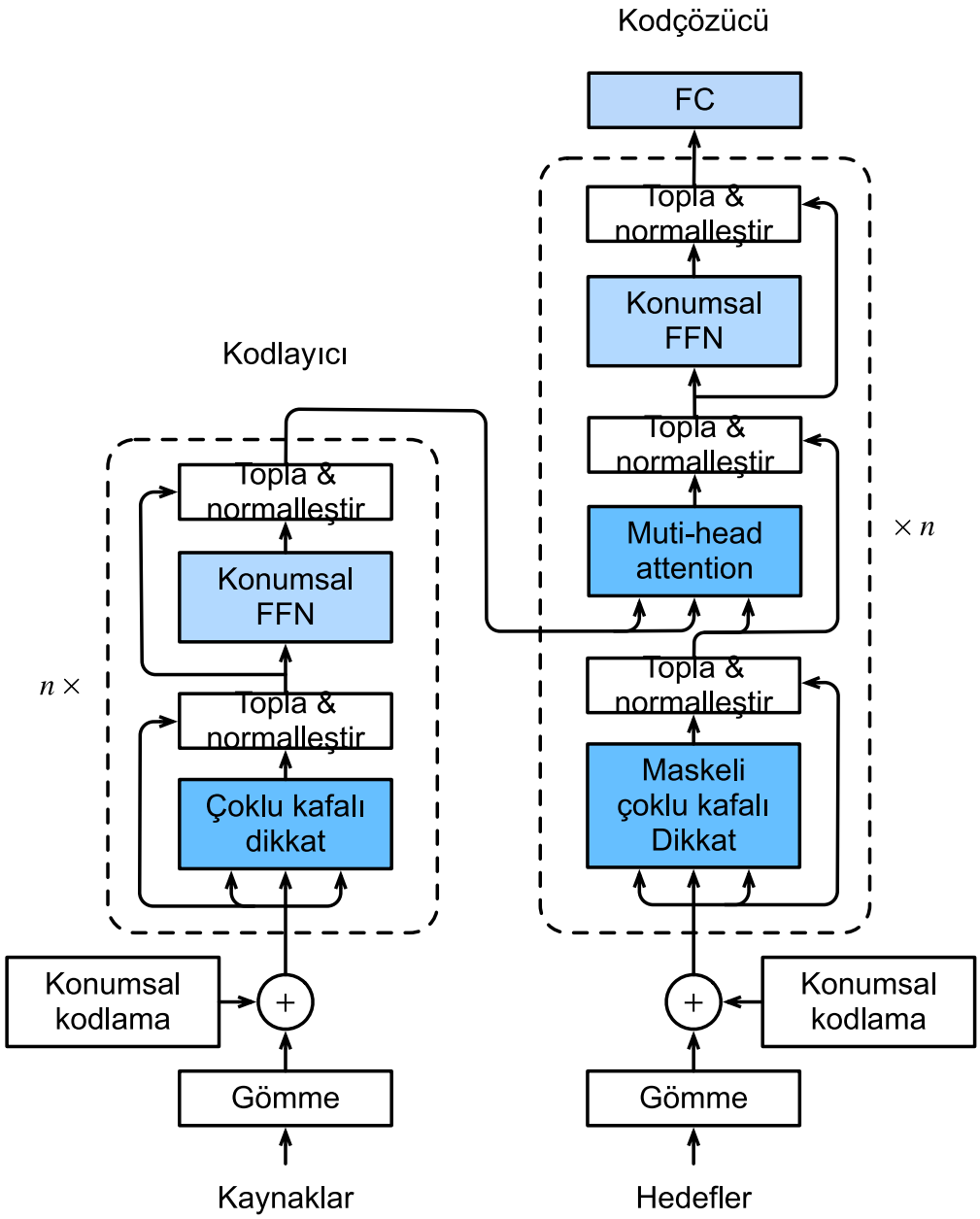


Fig. 10.7.1: Dönüştürücü mimarisi.

Şimdi Fig. 10.7.1 figüründeki dönüştürücü mimarisine genel bir bakış sunuyoruz. Yüksek düzeyde, dönüştürücü kodlayıcısı, her katmanın iki alt katmana sahip olduğu (ikisi de altkatman olarak ifade edilir) çoklu özdeş katmandan oluşan bir yığındır. Birincisi, çoklu kafalı bir özdikkat ortaklamasıdır ve ikincisi ise konumsal olarak ileriye besleme ağıdır. Özellikle, özdikkatteki kodlayıcıda, sorgular, anahtarlar ve değerler tüm önceki kodlayıcı katmanının çıktılarından gelir. Section 7.6 içindeki ResNet tasarımından esinlenerek, her iki alt katman etrafında artık bağlantı kullanılır. Dönüştürücüde, dizinin herhangi bir pozisyonunda $\mathbf{x} \in \mathbb{R}^d$ herhangi bir girdi için altkatman$(\mathbf{x}) \in \mathbb{R}^d$'ye ihtiyaç duyuyoruz, böylece $\mathbf{x} +$ altkatman$(\mathbf{x}) \in \mathbb{R}^d$, artık bağlantı $\mathbf{x} +$ altkatman$(\mathbf{x}) \in \mathbb{R}^d$ mümkündür. Artık bağlantıya bu ilavenin hemen ardından katman normalleştirmesi (Ba *et al.*, 2016) gelir. Sonuç olarak, dönüştürücü kodlayıcısı, girdi dizisinin her konumu için d boyutlu bir vektör temsilini çıkarır.

Dönüştürücü kodçözücü ayrıca artık bağlantılar ve katman normalleştirmeleri ile birden çok özdeş katman yığınıdır. Kodlayıcıda açıklanan iki alt katmanın yanı sıra, kodçözücü bu ikisi arasında kodlayıcı-kodçözücü dikkat olarak bilinen üçüncü bir alt katman ekler. Kodlayıcı-kod özücü dikkatinde, sorgular önceki kodçözücü katmanının çıktılarından ve anahtarlar ve değerler dönüştürücü kodlayıcı çıktılarından kaynaklanır. Kodçözücünün özdikkatinde, sorgular, anahtarlar ve değerler tüm önceki kodçözücü katmanının çıktılarından gelir. Bununla birlikte, kodçözücüdeki her pozisyonun, yalnızca kodçözücünün bu konuma kadar tüm pozisyonlara ilgi göstermesine izin verilir. Bu *maskelenmiş* dikkat, otomatik bağlanım özelliğini korur ve tahminin yalnızca üretilen çıktı belirteçlerine bağlı olmasını sağlar.

Section 10.5 içindeki ölçeklendirilmiş nokta çarpımlarına ve Section 10.6.3 içindeki konumsal kodlamaya dayanan çoklu kafalı dikkati zaten tanımladık ve uyguladık. Aşağıda, dönüştürücü modelinin geri kalanını uygulayacağız.

```
import math
import pandas as pd
import torch
from torch import nn
from d2l import torch as d2l
```

10.7.2 Konumsal Olarak İleriye Besleme Ağlar

Konumsal olarak ileriye besleme ağı, aynı MLP'yi kullanarak tüm dizi pozisyonlarındaki temsili dönüştürür. Bu yüzden ona *konumsal olarak* diyoruz. Aşağıdaki uygulamada, (toplu iş boyutu, zaman adımlarının sayısı veya belirteç dizi uzunluğu, gizli birimlerin sayısı veya öznitelik boyutu) şekline sahip `X` girdisi iki katmanlı bir MLP tarafından (parti boyutu, zaman adımlarının sayısı, `ffn_num_outputs`) şekilli çıktı tensörüne dönüştürülecektir .

```
#@save
class PositionWiseFFN(nn.Module):
    """Konumsal olarak ileriye besleme ağı."""
    def __init__(self, ffn_num_input, ffn_num_hiddens, ffn_num_outputs,
                 **kwargs):
        super(PositionWiseFFN, self).__init__(**kwargs)
        self.dense1 = nn.Linear(ffn_num_input, ffn_num_hiddens)
        self.relu = nn.ReLU()
        self.dense2 = nn.Linear(ffn_num_hiddens, ffn_num_outputs)

    def forward(self, X):
        return self.dense2(self.relu(self.dense1(X)))
```

Aşağıdaki örnek, tensörün en içteki boyutunun konumsal olarak ileriye besleme ağındaki çıktı sayısına değiştiğini göstermektedir. Aynı MLP tüm pozisyonlarda dönüştüğünden, tüm bu pozisyonlardaki girdiler aynı olduğunda, çıktıları da aynıdır.

```
ffn = PositionWiseFFN(4, 4, 8)
ffn.eval()
ffn(torch.ones((2, 3, 4)))[0]
```

```
tensor([[-0.3031,  0.0522, -0.8444, -0.3835,  0.0093, -0.2866, -0.0185,  0.0244],
        [-0.3031,  0.0522, -0.8444, -0.3835,  0.0093, -0.2866, -0.0185,  0.0244],
```

(continues on next page)

(continued from previous page)

```
        [-0.3031,  0.0522, -0.8444, -0.3835,  0.0093, -0.2866, -0.0185,  0.0244]],
       grad_fn=<SelectBackward0>)
```

10.7.3 Artık Bağlantı ve Katman Normalleştirmesi

Şimdi Fig. 10.7.1 figüründeki "topla ve normalleştir" bileşenine odaklanalım. Bu bölümün başında tanımladığımız gibi, bu, katman normalleştirmesinin hemen ardından geldiği bir artık bağlantıdır. Her ikisi de etkili derin mimarilerin anahtarıdır.

Section 7.5 bölümünde, toplu normalleştirmenin nasıl ortalandığını ve bir minigrup içindeki örnekler arasında nasıl yeniden ölçeklendiğini açıkladık. Katman normalleştirmesi, birincinin öznitelik boyutu boyunca normalleştirmesi dışında toplu normalleştirme ile aynıdır. Bilgisayarla görmede yaygın uygulamalarına rağmen, toplu normalleştirme deneysel olarak genellikle girdileri değişken uzunluktaki diziler olan doğal dil işleme görevlerinde katman normalleştirmesinden daha az etkilidir.

Aşağıdaki kod parçacığı katman normalleştirme ve toplu normalleştirme ile farklı boyutlar arasında normalleştirmeyi karşılaştırır.

```
ln = nn.LayerNorm(2)
bn = nn.BatchNorm1d(2)
X = torch.tensor([[1, 2], [2, 3]], dtype=torch.float32)
# Eğitim modunda `X`'den ortalama ve varyansı hesaplayın
print('layer norm:', ln(X), '\nbatch norm:', bn(X))
```

```
layer norm: tensor([[-1.0000,  1.0000],
        [-1.0000,  1.0000]], grad_fn=<NativeLayerNormBackward0>)
batch norm: tensor([[-1.0000, -1.0000],
        [ 1.0000,  1.0000]], grad_fn=<NativeBatchNormBackward0>)
```

Artık `AddNorm` sınıfını bir artık bağlantı ve ardından katman normalleştirme kullanarak uygulayabiliriz. Düzenlileştirme için hattan düşürme de uygulanır.

```
#@save
class AddNorm(nn.Module):
    """Artık bağlantı ve ardından katman normalleştirme."""
    def __init__(self, normalized_shape, dropout, **kwargs):
        super(AddNorm, self).__init__(**kwargs)
        self.dropout = nn.Dropout(dropout)
        self.ln = nn.LayerNorm(normalized_shape)

    def forward(self, X, Y):
        return self.ln(self.dropout(Y) + X)
```

Artık bağlantı, iki girdinin aynı şekle sahip olmasını gerektirir, böylece çıktı tensörünün toplama işleminden sonra da aynı şekle sahip olmasını sağlar.

```
add_norm = AddNorm([3, 4], 0.5) # Normalized_shape is input.size()[1:]
add_norm.eval()
add_norm(torch.ones((2, 3, 4)), torch.ones((2, 3, 4))).shape
```

```
torch.Size([2, 3, 4])
```

10.7.4 Kodlayıcı

Dönüştürücü kodlayıcıyı toparlamak için gerekli tüm bileşenlerle, kodlayıcı içinde tek bir katman uygulayarak başlayalım. Aşağıdaki `EncoderBlock` sınıfı iki alt katman içerir: Çoklu kafalı özdikkat ve konumsal ileri beslemeli ağlar, burada bir artık bağlantı ve ardından katman normalleştirme her iki alt katman etrafında kullanılır.

```
#@save
class EncoderBlock(nn.Module):
    """Dönüştürücü kodlayıcı bloğu."""
    def __init__(self, key_size, query_size, value_size, num_hiddens,
                 norm_shape, ffn_num_input, ffn_num_hiddens, num_heads,
                 dropout, use_bias=False, **kwargs):
        super(EncoderBlock, self).__init__(**kwargs)
        self.attention = d2l.MultiHeadAttention(
            key_size, query_size, value_size, num_hiddens, num_heads, dropout,
            use_bias)
        self.addnorm1 = AddNorm(norm_shape, dropout)
        self.ffn = PositionWiseFFN(
            ffn_num_input, ffn_num_hiddens, num_hiddens)
        self.addnorm2 = AddNorm(norm_shape, dropout)

    def forward(self, X, valid_lens):
        Y = self.addnorm1(X, self.attention(X, X, X, valid_lens))
        return self.addnorm2(Y, self.ffn(Y))
```

Gördüğümüz gibi, dönüştürücü kodlayıcısındaki herhangi bir katman girdisinin şeklini değiştirmez.

```
X = torch.ones((2, 100, 24))
valid_lens = torch.tensor([3, 2])
encoder_blk = EncoderBlock(24, 24, 24, 24, [100, 24], 24, 48, 8, 0.5)
encoder_blk.eval()
encoder_blk(X, valid_lens).shape
```

```
torch.Size([2, 100, 24])
```

Aşağıdaki dönüştürücü kodlayıcı uygulamasında, yukarıdaki `EncoderBlock` sınıflarının `num_layers` tane örneğini yığınlıyoruz. Değerleri her zaman -1 ile 1 arasında olan sabit konumsal kodlamayı kullandığımızdan, girdi gömmeyi ve konumsal kodlamayı toplamadan önce yeniden ölçeklendirmek için öğrenilebilir girdi gömmelerinin değerlerini gömme boyutunun kareköküyle çarparız.

```
#@save
class TransformerEncoder(d2l.Encoder):
    """Transformer encoder."""
    def __init__(self, vocab_size, key_size, query_size, value_size,
                 num_hiddens, norm_shape, ffn_num_input, ffn_num_hiddens,
```

(continues on next page)

(continued from previous page)

```
                 num_heads, num_layers, dropout, use_bias=False, **kwargs):
        super(TransformerEncoder, self).__init__(**kwargs)
        self.num_hiddens = num_hiddens
        self.embedding = nn.Embedding(vocab_size, num_hiddens)
        self.pos_encoding = d2l.PositionalEncoding(num_hiddens, dropout)
        self.blks = nn.Sequential()
        for i in range(num_layers):
            self.blks.add_module("block"+str(i),
                EncoderBlock(key_size, query_size, value_size, num_hiddens,
                             norm_shape, ffn_num_input, ffn_num_hiddens,
                             num_heads, dropout, use_bias))

    def forward(self, X, valid_lens, *args):
        # # Konumsal kodlama değerleri -1 ile 1 arasında olduğundan,
        # gömme değerleri, toplanmadan önce yeniden ölçeklendirmek için
        # gömme boyutunun kareköküyle çarpılır
        X = self.pos_encoding(self.embedding(X) * math.sqrt(self.num_hiddens))
        self.attention_weights = [None] * len(self.blks)
        for i, blk in enumerate(self.blks):
            X = blk(X, valid_lens)
            self.attention_weights[
                i] = blk.attention.attention.attention_weights
        return X
```

Aşağıda iki katmanlı bir dönüştürücü kodlayıcısı oluşturmak için hiper parametreleri belirtiyoruz. Dönüştürücü kodlayıcı çıktısının şekli (toplu iş boyutu, zaman adımlarının sayısı, `num_hiddens`) şeklindedir.

```
encoder = TransformerEncoder(
    200, 24, 24, 24, 24, [100, 24], 24, 48, 8, 2, 0.5)
encoder.eval()
encoder(torch.ones((2, 100), dtype=torch.long), valid_lens).shape
```

```
torch.Size([2, 100, 24])
```

10.7.5 Kodçözücü

Fig. 10.7.1 şeklinde gösterildiği gibi, dönüştürücü kodçözücüsü birden çok özdeş katmandan oluşur. Her katman, üç alt katman içeren aşağıdaki `DecoderBlock` sınıfında uygulanır: Kodçözücü özdikkat, kodlayıcı-kodçözücü dikkat ve konumsal olarak ileri beslemeli ağlar. Bu alt katmanlar çevrelerinde bir artık bağlantı ve ardından katman normalleştirmesi kullanır.

Bu bölümde daha önce de açıklandığı gibi, maskelenmiş çoklu kafalı kodçözücü özdikkatinde (ilk alt katman), sorgular, anahtarlar ve değerler önceki kodçözücü katmanının çıktılarından gelir. Diziden diziye modellerini eğitirken, çıktı dizisinin tüm pozisyonlarında (zaman adımları) belirteçleri bilinir. Bununla birlikte, tahmin esnasında çıktı dizisi belirteç belirteç oluşturulur; böylece, herhangi bir kodçözücü zaman adımında yalnızca üretilen belirteçler kodçözücünün özdikkatinde kullanılabilir. Kodçözücünün otomatik regresyonunu korumak için, maskeli özdikkat `dec_valid_lens`'ü belirtir, böylece herhangi bir sorgu yalnızca kodçözücünün sorgulama konumuna kadarki tüm konumlara ilgi gösterir.

```
class DecoderBlock(nn.Module):
    # Kodçözücüdeki `i`. blok
    def __init__(self, key_size, query_size, value_size, num_hiddens,
                 norm_shape, ffn_num_input, ffn_num_hiddens, num_heads,
                 dropout, i, **kwargs):
        super(DecoderBlock, self).__init__(**kwargs)
        self.i = i
        self.attention1 = d2l.MultiHeadAttention(
            key_size, query_size, value_size, num_hiddens, num_heads, dropout)
        self.addnorm1 = AddNorm(norm_shape, dropout)
        self.attention2 = d2l.MultiHeadAttention(
            key_size, query_size, value_size, num_hiddens, num_heads, dropout)
        self.addnorm2 = AddNorm(norm_shape, dropout)
        self.ffn = PositionWiseFFN(ffn_num_input, ffn_num_hiddens,
                                   num_hiddens)
        self.addnorm3 = AddNorm(norm_shape, dropout)

    def forward(self, X, state):
        enc_outputs, enc_valid_lens = state[0], state[1]
        # Eğitim sırasında, herhangi bir çıktı dizisinin tüm belirteçleri
        # aynı anda işlenir, bu nedenle `state[2][self.i]` ilklendiği gibi
        #  `None` olur. Tahmin sırasında herhangi bir çıktı dizisi
        # belirtecinin kodunu çözerken, `state[2][self.i]`, geçerli zaman
        # adımına kadar  `i`. bloğundaki kodu çözülmüş çıktının
        # temsillerini içerir.
        if state[2][self.i] is None:
            key_values = X
        else:
            key_values = torch.cat((state[2][self.i], X), axis=1)
        state[2][self.i] = key_values
        if self.training:
            batch_size, num_steps, _ = X.shape
            # `dec_valid_lens` şekli: (`batch_size`, `num_steps`),
            #  burada her satır [1, 2, ..., `num_steps`]
            dec_valid_lens = torch.arange(
                1, num_steps + 1, device=X.device).repeat(batch_size, 1)
        else:
            dec_valid_lens = None

        # Öz-dikkat
        X2 = self.attention1(X, key_values, key_values, dec_valid_lens)
        Y = self.addnorm1(X, X2)
        # Kodlayıcı - kodçözücüye dikkat. `enc_outputs` şekli:
        # (`batch_size`, `num_steps`, `num_hiddens`)
        Y2 = self.attention2(Y, enc_outputs, enc_outputs, enc_valid_lens)
        Z = self.addnorm2(Y, Y2)
        return self.addnorm3(Z, self.ffn(Z)), state
```

Kodlayıcı-kodçözücü dikkat ve artık bağlantılarda toplama işlemlerinde ölçeklendirilmiş nokta çarpımı işlemlerini kolaylaştırmak için, kodçözücünün öznitelik boyutu (`num_hiddens`) kodlayıcınınkiyle aynıdır.

```
decoder_blk = DecoderBlock(24, 24, 24, 24, [100, 24], 24, 48, 8, 0.5, 0)
decoder_blk.eval()
X = torch.ones((2, 100, 24))
```

(continues on next page)

(continued from previous page)

```
state = [encoder_blk(X, valid_lens), valid_lens, [None]]
decoder_blk(X, state)[0].shape
```

```
torch.Size([2, 100, 24])
```

Şimdi `DecoderBlock` `DecoderBlock` örneklerinden oluşan tam dönüştürücü kodçözücüyü oluşturuyoruz. Sonunda, tam bağlı bir katman tüm `vocab_size` olası çıktı belirteçleri için tahmin hesaplar. Hem dekoder öz-dikkat ağırlıkları hem de kodlayıcı-kodçözücü dikkat ağırlıkları daha sonra görselleştirme için saklanır.

Şimdi, `DecoderBlock`'un `num_layers` tane örnekten oluşan bütün dönüştürücü kodçözücüsünü oluşturuyoruz. Sonunda, tam bağlı bir katman, `vocab_size` boyutlu tüm olası çıktı belirteçleri için tahminleri hesaplar. Hem kodçözücü özdikkat ağırlıkları hem de kodlayıcı-kodçözücü dikkat ağırlıkları daha sonrasındaki görselleştirme için saklanır.

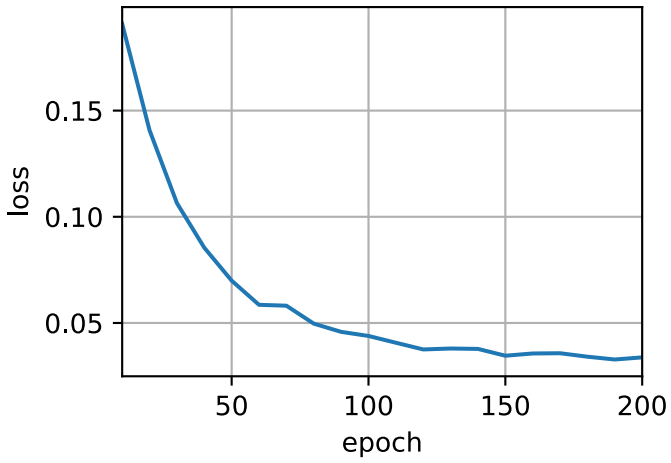
```
class TransformerDecoder(d2l.AttentionDecoder):
    def __init__(self, vocab_size, key_size, query_size, value_size,
                 num_hiddens, norm_shape, ffn_num_input, ffn_num_hiddens,
                 num_heads, num_layers, dropout, **kwargs):
        super(TransformerDecoder, self).__init__(**kwargs)
        self.num_hiddens = num_hiddens
        self.num_layers = num_layers
        self.embedding = nn.Embedding(vocab_size, num_hiddens)
        self.pos_encoding = d2l.PositionalEncoding(num_hiddens, dropout)
        self.blks = nn.Sequential()
        for i in range(num_layers):
            self.blks.add_module("block"+str(i),
                DecoderBlock(key_size, query_size, value_size, num_hiddens,
                             norm_shape, ffn_num_input, ffn_num_hiddens,
                             num_heads, dropout, i))
        self.dense = nn.Linear(num_hiddens, vocab_size)

    def init_state(self, enc_outputs, enc_valid_lens, *args):
        return [enc_outputs, enc_valid_lens, [None] * self.num_layers]

    def forward(self, X, state):
        X = self.pos_encoding(self.embedding(X) * math.sqrt(self.num_hiddens))
        self._attention_weights = [[None] * len(self.blks) for _ in range (2)]
        for i, blk in enumerate(self.blks):
            X, state = blk(X, state)
            # Kodçözücü özdikkat ağırlıkları
            self._attention_weights[0][
                i] = blk.attention1.attention.attention_weights
            # Kodlayıcı-kodçözücü dikkat ağırlıkları
            self._attention_weights[1][
                i] = blk.attention2.attention.attention_weights
        return self.dense(X), state

    @property
    def attention_weights(self):
        return self._attention_weights
```

10.7.6 Eğitim

Dönüştürücü mimarisini takip ederek bir kodlayıcı-kodçözücü modeli oluşturalım. Burada hem dönüştürücü kodlayıcının hem de dönüştürücü kodçözücünün 4 kafalı dikkat kullanan 2 katmana sahip olduğunu belirtiyoruz. Section 9.7.4 içindekine benzer şekilde, dönüştürücü modelini İngilizce-Fransızca makine çevirisi veri kümelerinde diziden diziye öğrenmeye yönelik eğitiyoruz.

```
num_hiddens, num_layers, dropout, batch_size, num_steps = 32, 2, 0.1, 64, 10
lr, num_epochs, device = 0.005, 200, d2l.try_gpu()
ffn_num_input, ffn_num_hiddens, num_heads = 32, 64, 4
key_size, query_size, value_size = 32, 32, 32
norm_shape = [32]

train_iter, src_vocab, tgt_vocab = d2l.load_data_nmt(batch_size, num_steps)

encoder = TransformerEncoder(
    len(src_vocab), key_size, query_size, value_size, num_hiddens,
    norm_shape, ffn_num_input, ffn_num_hiddens, num_heads,
    num_layers, dropout)
decoder = TransformerDecoder(
    len(tgt_vocab), key_size, query_size, value_size, num_hiddens,
    norm_shape, ffn_num_input, ffn_num_hiddens, num_heads,
    num_layers, dropout)
net = d2l.EncoderDecoder(encoder, decoder)
d2l.train_seq2seq(net, train_iter, lr, num_epochs, tgt_vocab, device)
```

```
loss 0.034, 4753.8 tokens/sec on cuda:0
```



Eğitimden sonra dönüştürücü modelini birkaç İngilizce cümleyi Fransızca'ya çevirmek ve BLEU puanlarını hesaplamak için kullanıyoruz.

```
engs = ['go .', "i lost .", 'he\'s calm .', 'i\'m home .']
fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']
for eng, fra in zip(engs, fras):
    translation, dec_attention_weight_seq = d2l.predict_seq2seq(
        net, eng, src_vocab, tgt_vocab, num_steps, device, True)
```

(continues on next page)

(continued from previous page)

```
    print(f'{eng} => {translation}, ',
          f'bleu {d2l.bleu(translation, fra, k=2):.3f}')
```

```
go . => va !,  bleu 1.000
i lost . => j'ai perdu .,  bleu 1.000
he's calm . => il est calme .,  bleu 1.000
i'm home . => je suis chez moi .,  bleu 1.000
```

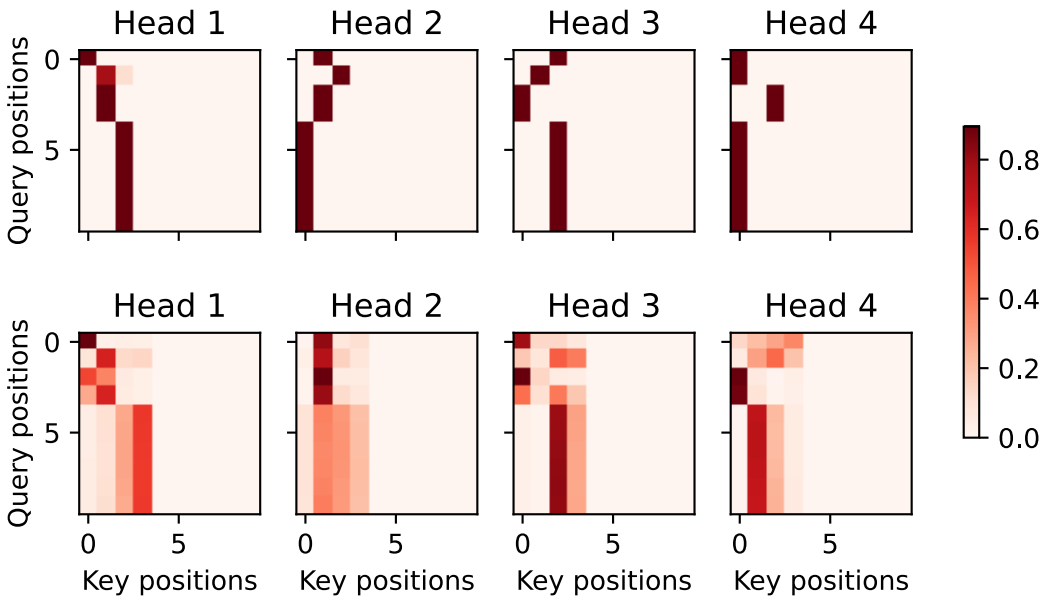
Son İngilizce cümleyi Fransızcaya çevirirken dönüştürücünün dikkat ağırlıklarını görselleştirmemize izin verin. Kodlayıcı özdikkat ağırlıklarının şekli (kodlayıcı katmanlarının sayısı, dikkat kafalarının sayısı, `num_steps` veya sorgu sayısı, `num_steps` veya anahtar değer çiftlerinin sayısı) şeklindedir.

```
enc_attention_weights = torch.cat(net.encoder.attention_weights, 0).reshape((num_layers, num_
↪heads,
    -1, num_steps))
enc_attention_weights.shape
```

```
torch.Size([2, 4, 10, 10])
```

Özdikkat kodlayıcısında, hem sorgular hem de anahtarlar aynı girdi dizisinden gelir. Dolgu belirteçleri anlam taşımadığından, girdi dizisinin belirlenmiş geçerli uzunluğu ile, dolgu belirteçlerinin konumlarına hiçbir sorgu ilgi göstermez. Aşağıda, çoklu kafalı iki katmanın dikkat ağırlıkları satır satır gösterilmektedir. Her kafa, sorguların, anahtarların ve değerlerin ayrı bir temsil altuzaylarına dayanarak bağımsız olarak ilgi gösterir.

```
d2l.show_heatmaps(
    enc_attention_weights.cpu(), xlabel='Key positions',
    ylabel='Query positions', titles=['Head %d' % i for i in range(1, 5)],
    figsize=(7, 3.5))
```



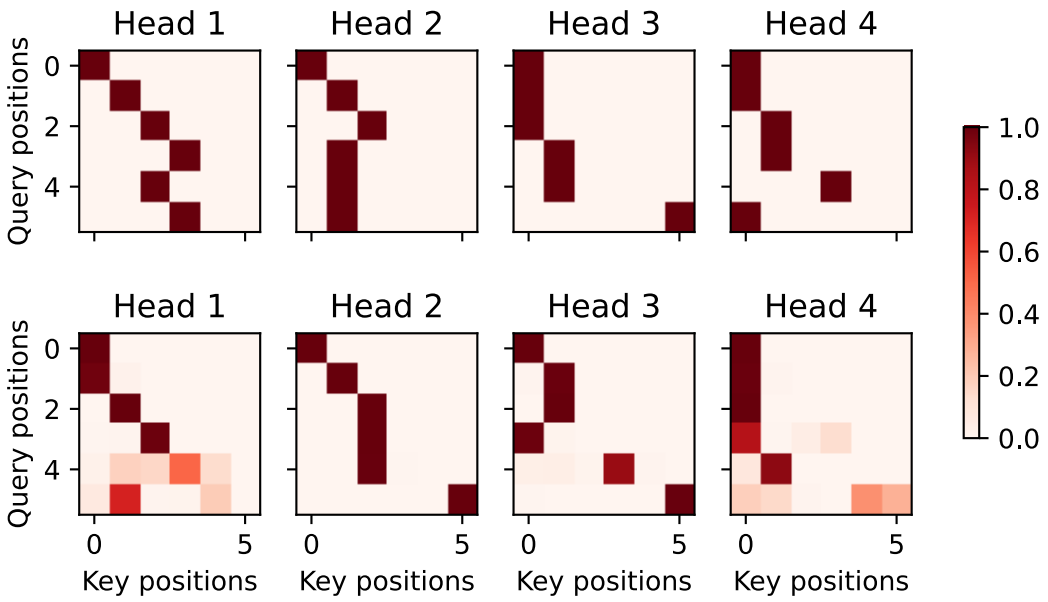
Hem kodçözücü özdikkat ağırlıklarını hem de kodlayıcı-kodçözücü dikkat ağırlıklarını görselleştirmek için daha fazla veri düzenlemeye ihtiyacımız var. Örneğin, maskelenmiş dikkat ağırlıklarını sıfırla dolduruyoruz. Kodçözücü özdikkat ağırlıklarının ve kodlayıcı-kodçözücü dikkat ağırlıklarının her ikisinin de aynı sorguları olduğunu unutmayın: Dizinin başlangıç belirteci ve ardından çıktı belirteçleri.

```
dec_attention_weights_2d = [head[0].tolist()
                            for step in dec_attention_weight_seq
                            for attn in step for blk in attn for head in blk]
dec_attention_weights_filled = torch.tensor(
    pd.DataFrame(dec_attention_weights_2d).fillna(0.0).values)
dec_attention_weights = dec_attention_weights_filled.reshape((-1, 2, num_layers, num_heads,␣
↪num_steps))
dec_self_attention_weights, dec_inter_attention_weights = \
    dec_attention_weights.permute(1, 2, 3, 0, 4)
dec_self_attention_weights.shape, dec_inter_attention_weights.shape
```

```
(torch.Size([2, 4, 6, 10]), torch.Size([2, 4, 6, 10]))
```

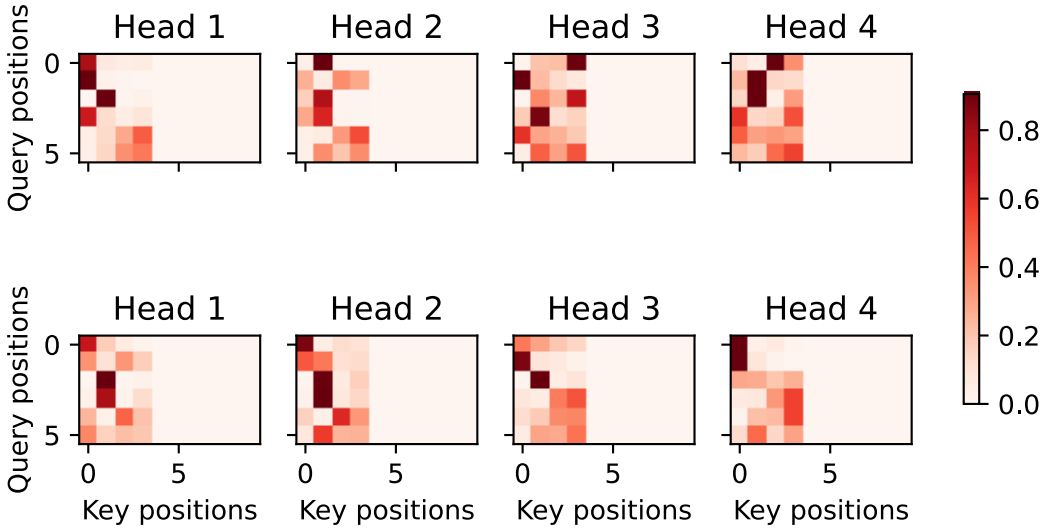
Kodçözücünün özdikkatinin otomatik bağlanım özelliği nedeniyle, hiçbir sorgu sorgu konumundan sonra anahtar/değer çiftlerine ilgi göstermez.

```
# Dizi başlangıç belirtecini içermek için 1 ekle
d2l.show_heatmaps(
    dec_self_attention_weights[:, :, :, :len(translation.split()) + 1],
    xlabel='Key positions', ylabel='Query positions',
    titles=['Head %d' % i for i in range(1, 5)], figsize=(7, 3.5))
```



Kodlayıcının özdikkatindeki duruma benzer şekilde, girdi dizisince belirtilen geçerli uzunluğu aracılığıyla, çıktı dizisinden gelen hiçbir sorgu bu girdi dizisinden dolgu belirteçlerine ilgi göstermez.

```
d2l.show_heatmaps(
    dec_inter_attention_weights, xlabel='Key positions',
    ylabel='Query positions', titles=['Head %d' % i for i in range(1, 5)],
    figsize=(7, 3.5))
```



Dönüştürücü mimarisi başlangıçta diziden-diziye öğrenme için önerilmiş olsa da, kitapta daha sonra keşfedeceğimiz gibi, dönüştürücü kodlayıcı ya da dönüştürücü kodçözücü genellikle farklı derin öğrenme görevleri için ayrı ayrı kullanılır.

10.7.7 Özet

- Dönüştürücü, kodlayıcı-kodçözücü mimarisinin bir örneğidir, ancak kodlayıcı veya kodçözücü uygulamada ayrı ayrı kullanılabilir.
- Dönüştürücüde, girdi dizisini ve çıktı dizisini temsil etmek için çoklu kafalı özdikkat kullanılır, ancak kodçözücünün maskelenmiş bir sürüm aracılığıyla otomatik bağlanım özelliğini korumak zorundadır.
- Hem artık bağlantılar hem de dönüştürücüdeki katman normalleştirmesi, bir çok derin modeli eğitmek için önemlidir.
- Dönüştürücü modelindeki konumsal olarak ileriye besleme ağı, aynı MLP'yi kullanarak tüm dizi konumlarındaki gösterimi dönüştürür.

10.7.8 Alıştırmalar

1. Deneylerde daha derin bir dönüştürücü eğitin. Eğitim hızını ve çeviri performansını nasıl etkiler?
2. Dönüştürücüdeki ölçeklendirilmiş nokta çarpımı dikkatini toplayıcı dikkati ile değiştirmek iyi bir fikir midir? Neden?
3. Dil modellemesi için dönüştürücü kodlayıcısını mı, kodçözücüyü mü veya her ikisini birden mi kullanmalıyız? Bu yöntem nasıl tasarlanabilir?

4. Girdi dizileri çok uzunsa dönüştürücüler için ne zorluklar olabilir? Neden?
5. Dönüştürücülerin hesaplama ve bellek verimliliğini nasıl arttırılabilir? İpucu: Tay ve ark. tarafından hazırlanan çalışmaya başvurabilirsiniz. (Tay *et al.*, 2020).
6. CNN kullanmadan imge sınıflandırma işleri için dönüştürücü tabanlı modelleri nasıl tasarlayabiliriz? İpucu: Görüntü dönüştürücüye başvurabilirsiniz (Dosovitskiy *et al.*, 2021)

Tartışmalar[126]

[126] https://discuss.d2l.ai/t/1066

11 | Eniyileme Algoritmaları

Kitabı bu noktaya kadar sırayla okurduysanız, şimdiden derin öğrenme modellerini eğitmek için bir takım eniyileme (optimizasyon) algoritmasını kullanmış oldunuz. Eğitim kümesinde değerlendirildiği gibi, model parametrelerini güncellemeye devam etmemize ve kayıp fonksiyonunun değerini en aza indirmemize izin veren araçlardı. Nitekim, basit bir ortamda amaç işlevleri en aza indirmek için bir kara kutu cihazı olarak eniyileme uygulayan herkes, böyle bir dizi sihirli yöntemin ("SGD" ve "Adam" gibi isimlerle) bulunduğunu bilerek kendini memnun edebilir.

Bununla birlikte, daha iyisini yapmak için biraz daha derin bilgi gereklidir. Optimizasyon algoritmaları derin öğrenme için önemlidir. Bir yandan karmaşık bir derin öğrenme modelini eğitmek saatler, günler, hatta haftalar sürebilir. Optimizasyon algoritmasının başarımı, modelin eğitim verimliliğini doğrudan etkiler. Öte yandan, farklı optimizasyon algoritmalarının ilkelerini ve onların hiper parametrelerinin rolünü anlamak, derin öğrenme modellerinin başarımını artırmak için hiper parametreleri hedeflenen bir şekilde kurmamızı sağlayacaktır.

Bu bölümde, yaygın derin öğrenme optimizasyon algoritmalarını derinlemesine araştırıyoruz. Derin öğrenmede ortaya çıkan neredeyse tüm optimizasyon problemleri *dışbükey (convex) olmayandır*. Bununla birlikte, *dışbükey* problemleri bağlamında algoritmaların tasarımının ve çözümlemesinin çok öğretici olduğu kanıtlanmıştır. Bu nedenle, bu bölüm dışbükey optimizasyonda bir özbilgi ve dışbükey bir amaç fonksiyon üzerinde çok basit bir rasgele gradyan iniş algoritması için kanıt içerir.

11.1 Eniyileme ve Derin Öğrenme

Bu bölümde, eniyileme ve derin öğrenme arasındaki ilişkiyi ve derin öğrenmede optimizasyonu kullanmanın zorluklarını tartışacağız. Derin öğrenme problemi için, genellikle önce *kayıp fonksiyonu* tanımlarız. Kayıp işlevini aldığımızda kayıpları en aza indirmek için bir optimizasyon algoritması kullanabiliriz. Optimizasyonda, bir kayıp fonksiyonu genellikle optimizasyon sorununun *amaç fonksiyonu* olarak adlandırılır. Geleneksel ve alışılmış olarak çoğu optimizasyon algoritması, *minimizasyon (en aza indirme)* ile ilgilidir. Eğer bir hedefi en üst düzeye çıkarmamız (maksimize etmemiz) gerekirse basit bir çözüm vardır: Sadece amaç işlevindeki işareti tersine çevirin.

11.1.1 Eniyilemenin Hedefi

Optimizasyon derin öğrenme için kayıp işlevini en aza indirmenin bir yolunu sağlasa da, özünde optimizasyonun ve derin öğrenmenin amaçları temelde farklıdır. Birincisi öncelikle bir amaç işlevini en aza indirmekle ilgiliyken, ikincisi, sınırlı miktarda veri verildiğinde uygun bir model bulmakla ilgilidir. Section 4.4 içinde, bu iki hedef arasındaki farkı ayrıntılı olarak tartıştık. Örneğin, eğitim hatası ve genelleme hatası genellikle farklılık gösterir: Optimizasyon algoritmasının amaç işlevi genellikle eğitim veri kümesine dayalı bir kayıp fonksiyonu olduğundan, optimizasyonun amacı eğitim hatasını azaltmaktır. Bununla birlikte, derin öğrenmenin amacı (veya daha geniş bir şekilde istatistiksel çıkarımın) genelleme hatasını azaltmaktır. İkincisini başarmak için, eğitim hatasını azaltmada optimizasyon algoritmasını kullanmanın yanı sıra aşırı öğrenme işlemine de dikkat etmeliyiz.

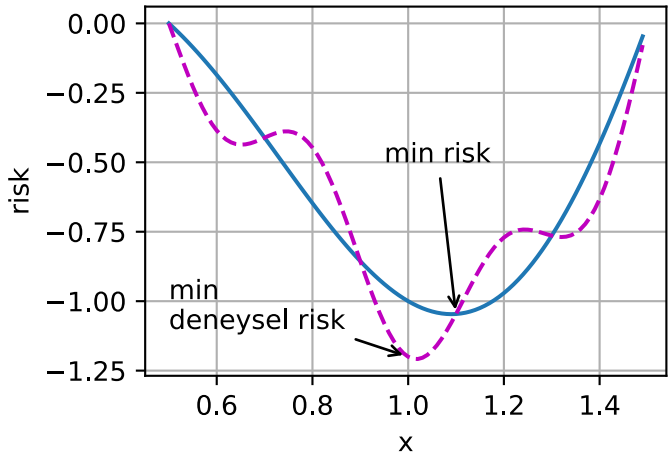
```
%matplotlib inline
import numpy as np
import torch
from mpl_toolkits import mplot3d
from d2l import torch as d2l
```

Yukarıda belirtilen farklı hedefleri göstermek için, riski ve deneysel riski ele alalım. Section 4.9.2 içinde açıklandığı gibi, deneysel risk, eğitim veri kümesinde ortalama bir kayıptır ve risk veri nüfusunun tamamında beklenen kayıptır. Aşağıda iki fonksiyon tanımlıyoruz: Risk fonksiyonu `f` ve deneysel risk fonksiyonu `g`. Sadece sınırlı miktarda eğitim verisi olduğunu varsayalım. Sonuç olarak, burada `g` `f`'den daha az pürüzsüzdür.

```
def f(x):
    return x * torch.cos(np.pi * x)

def g(x):
    return f(x) + 0.2 * torch.cos(5 * np.pi * x)
```

Aşağıdaki grafik, bir eğitim veri kümesinde deneysel riskin minimumunun, minimum riskten farklı bir konumda olabileceğini göstermektedir (genelleme hatası).

```
def annotate(text, xy, xytext):  #@save
    d2l.plt.gca().annotate(text, xy=xy, xytext=xytext,
                           arrowprops=dict(arrowstyle='->'))

x = torch.arange(0.5, 1.5, 0.01)
d2l.set_figsize((4.5, 2.5))
d2l.plot(x, [f(x), g(x)], 'x', 'risk')
annotate('min \ndeneysel risk', (1.0, -1.2), (0.5, -1.1))
annotate('min risk', (1.1, -1.05), (0.95, -0.5))
```



11.1.2 Derin Öğrenmede Eniyileme Zorlukları

Bu bölümde, özellikle bir modelin genelleme hatası yerine amaç işlevini en aza indirmede optimizasyon algoritmalarının başarımına odaklanacağız. Section 3.1 içinde optimizasyon problemlerinde analitik çözümler ve sayısal çözümler arasında ayrım yaptık. Derin öğrenmede, çoğu amaç fonksiyonu karmaşıktır ve analitik çözümleri yoktur. Bunun yerine, sayısal optimizasyon algoritmaları kullanmalıyız. Bu bölümdeki optimizasyon algoritmalarının tümü bu kategoriye girer.

Derin öğrenme optimizasyonunda birçok zorluk vardır. En eziyetli olanlardan bazıları yerel minimum, eyer noktaları ve kaybolan gradyanlardır. Onlara bir göz atalım.

Yerel En Düşüklükler

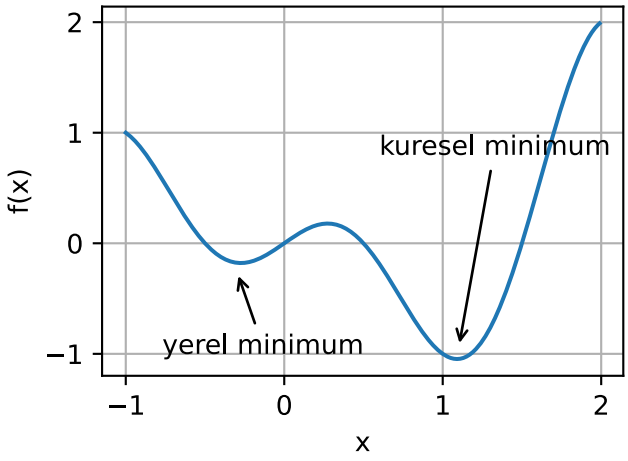
Herhangi bir $f(x)$ amaç işlevi için $f(x)$ değerinin x'teki değeri x civarındaki diğer noktalardaki $f(x)$ değerlerinden daha küçükse, $f(x)$ yerel minimum olabilir. x değerindeki $f(x)$ değeri, tüm etki alanı üzerinde amaç işlevin minimum değeriyse, $f(x)$ genel minimum değerdir.

Örneğin, aşağıdaki fonksiyon göz önüne alındığında

$$f(x) = x \cdot \cos(\pi x) \text{ for } -1.0 \leq x \leq 2.0, \tag{11.1.1}$$

bu fonksiyonun yerel minimum ve küresel minimum değerlerini yaklaşık olarak değerlendirebiliriz.

```
x = torch.arange(-1.0, 2.0, 0.01)
d2l.plot(x, [f(x), ], 'x', 'f(x)')
annotate('yerel minimum', (-0.3, -0.25), (-0.77, -1.0))
annotate('kuresel minimum', (1.1, -0.95), (0.6, 0.8))
```

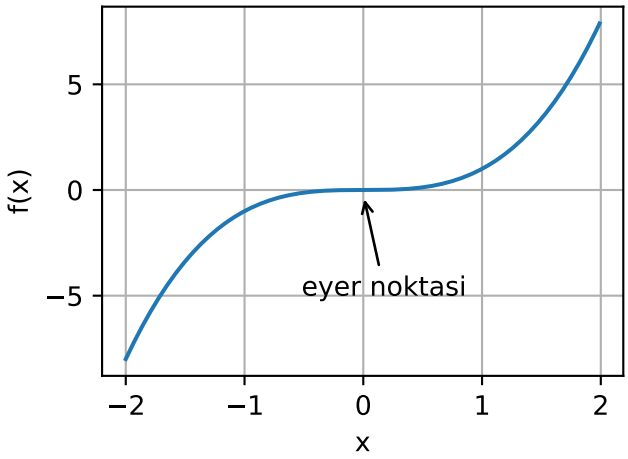


Derin öğrenme modellerinin amaç işlevi genellikle birçok yerel eniyi değere sahiptir. Bir optimizasyon probleminin sayısal çözümü yerel eniyi seviyesine yakın olduğunda, nihai yineleme ile elde edilen sayısal çözüm, amaç fonksiyonun çözümlerinin gradyanının sıfıra yaklaştığı veya olduğu için, objektif işlevi *küresel* yerine yalnızca *yerel* en aza indirebilir. Parametreyi yalnızca bir dereceye kadar gürültü yerel minimum seviyeden çıkarabilir. Aslında bu, minigrup üzerindeki gradyanların doğal çeşitliliğin parametreleri yerel en düşük yerinden çıkarabildiği minigrup rasgele gradyan inişinin yararlı özelliklerinden biridir.

Eyer Noktaları

Yerel minimumun yanı sıra eyer noktaları gradyanların kaybolmasının bir başka nedenidir. *Eyer noktası*, bir işlevin tüm gradyanların kaybolduğu ancak ne küresel ne de yerel minimum olmayan herhangi bir konumdur. $f(x) = x^3$ işlevini düşünün. Onun birinci ve ikinci türevi $x = 0$ için kaybolur. Eniyileme minimum olmasa bile, bu noktada durabilir.

```
x = torch.arange(-2.0, 2.0, 0.01)
d2l.plot(x, [x**3], 'x', 'f(x)')
annotate('eyer noktasi', (0, -0.2), (-0.52, -5.0))
```

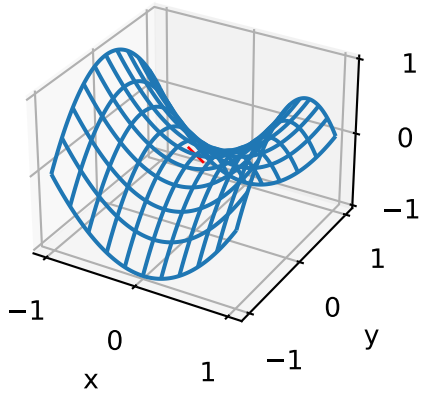


Aşağıdaki örnekte gösterildiği gibi, daha yüksek boyutlardaki eyer noktaları daha da gizli

tehlikedir. $f(x, y) = x^2 - y^2$ işlevini düşünün. $(0, 0)$ onun eyer noktası vardır. Bu, y'ye göre maksimum ve x'e göre minimum değerdir. Dahası, bu, matematiksel özelliğin adını aldığı bir eyer gibi *görünür*.

```
x, y = torch.meshgrid(
    torch.linspace(-1.0, 1.0, 101), torch.linspace(-1.0, 1.0, 101))
z = x**2 - y**2

ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x, y, z, **{'rstride': 10, 'cstride': 10})
ax.plot([0], [0], [0], 'rx')
ticks = [-1, 0, 1]
d2l.plt.xticks(ticks)
d2l.plt.yticks(ticks)
ax.set_zticks(ticks)
d2l.plt.xlabel('x')
d2l.plt.ylabel('y');
```

```
/home/d2l-worker/miniconda3/envs/d2l-tr-release-0/lib/python3.9/site-packages/torch/
↪functional.py:478: UserWarning: torch.meshgrid: in an upcoming release, it will be␣
↪required to pass the indexing argument. (Triggered internally at  ../aten/src/ATen/native/
↪TensorShape.cpp:2895.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
```



Bir fonksiyonun girdisinin k boyutlu bir vektör olduğunu ve çıktısının bir skaler olduğunu varsayıyoruz, bu nedenle Hessian matrisinin k tane özdeğere sahip olacağını varsayıyoruz (özayrışımlar üzerine çevrimiçi ek[127]'e bakın). Fonksiyonun çözümü yerel minimum, yerel maksimum veya işlev gradyanının sıfır olduğu bir konumda eyer noktası olabilir:

- Sıfır-gradyan konumundaki fonksiyonun Hessian matrisinin özdeğerleri pozitif olduğunda, işlev için yerel minimum değerlere sahibiz.
- Sıfır-gradyan konumundaki fonksiyonun Hessian matrisinin özdeğerleri negatif olduğunda, işlev için yerel maksimum değerlere sahibiz.
- Sıfır-gradyan konumundaki fonksiyonun Hessian matrisinin özdeğerleri negatif ve pozitif olduğunda, fonksiyon için bir eyer noktamız vardır.

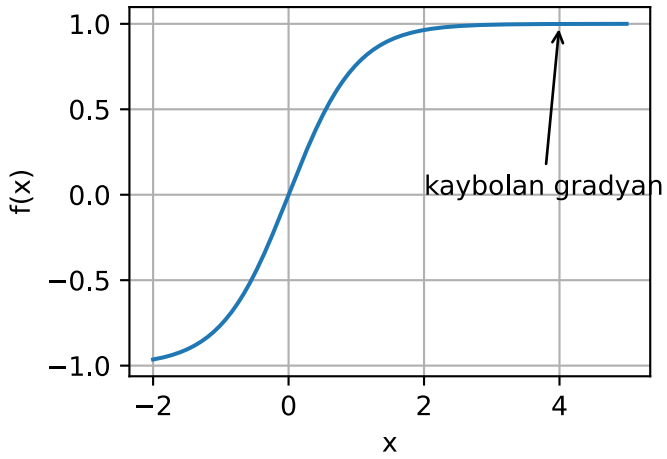
[127] https://tr.d2l.ai/chapter_appendix-mathematics-for-deep-learning/eigendecomposition.html

Yüksek boyutlu problemler için özdeğerlerin en azından *bazılarının* negatif olma olasılığı oldukça yüksektir. Bu eyer noktalarını yerel minimumlardan daha olasıdır yapar. Dışbükeyliği tanıtırken bir sonraki bölümde bu durumun bazı istisnalarını tartışacağız. Kısacası, dışbükey fonksiyonlar Hessian'ın özdeğerlerinin asla negatif olmadığı yerlerdir. Ne yazık ki, yine de, çoğu derin öğrenme problemi bu kategoriye girmiyor. Yine de optimizasyon algoritmaları incelemek için harika bir araçtır.

Kaybolan Gradyanlar

Muhtemelen aşılması gereken en sinsi sorun kaybolan gradyanlardır. Section 4.1.2 içinde yaygın olarak kullanılan etkinleştirme fonksiyonlarımızı ve türevlerini hatırlayın. Örneğin, $f(x) = \tanh(x)$ işlevini en aza indirmek istediğimizi ve $x = 4$'te başladığımızı varsayalım. Gördüğümüz gibi, f'nin gradyanı sıfıra yakındır. Daha özel durum olarak, $f'(x) = 1 - \tanh^2(x)$'dir ve o yüzden $f'(4) = 0.0013$'tür. Sonuç olarak, ilerleme kaydetmeden önce optimizasyon uzun süre takılıp kalacak. Bunun, derin öğrenme modellerinin, ReLU etkinleştirme işlevinin tanıtılmasından önce oldukça zor olmasının nedenlerinden biri olduğu ortaya çıkıyor.

```
x = torch.arange(-2.0, 5.0, 0.01)
d2l.plot(x, [torch.tanh(x)], 'x', 'f(x)')
annotate('kaybolan gradyan', (4, 1), (2, 0.0))
```



Gördüğümüz gibi, derin öğrenme için optimizasyon zorluklarla doludur. Neyse ki iyi performans ve yeni başlayanlar için bile kullanımı kolay bir dizi gürbüz algoritma var. Ayrıca, en iyi çözümü bulmak gerçekten gerekli değildir. Yerel en iyi veya hatta yaklaşık çözümler hala çok faydalıdır.

11.1.3 Özet

- Eğitim hatasının en aza indirilmesi, genelleme hatasını en aza indirmek için en iyi parametre kümesini bulduğumuzu garanti etmez.
- Optimizasyon sorunlarının birçok yerel minimumu olabilir.
- Problemlerin daha fazla eyer noktası olabilir, genellikle problemler dışbükey değildir.
- Kaybolan gradyanlar eniyilemenin durmasına neden olabilir. Genellikle sorunun yeniden parametrelendirilmesi yardımcı olur. Parametrelerin iyi ilklenmesi de faydalı olabilir.

11.1.4 Alıştırmalar

1. Gizli katmanında d boyutlu tek bir gizli katmanına ve tek bir çıktıya sahip basit bir MLP düşünün. Herhangi bir yerel minimum için en az $d!$ tane aynı davranan eşdeğer çözüm olduğunu gösterin.
2. Simetrik bir rastgele $\mathbf{M}$ matrisimiz olduğunu varsayalım, burada $M_{ij} = M_{ji}$ girdilerinin her biri p_{ij} olasılık dağılımından çekilir. Ayrıca, $p_{ij}(x) = p_{ij}(-x)$'nin, yani dağılımın simetrik olduğunu varsayalım (ayrıntılar için bkz. (Wigner, 1958)).
    1. Özdeğerler üzerindeki dağılımın da simetrik olduğunu kanıtlayın. Yani, herhangi bir özvektör $\mathbf{v}$ için, ilişkili özdeğer λ'nin $P(\lambda > 0) = P(\lambda < 0)$'i karşılama olasılığı.
    2. Yukarıdaki ifade neden $P(\lambda > 0) = 0.5$ anlamına *gelmez*?
3. Derin öğrenme optimizasyonunda yer alan hangi diğer zorlukları düşünebilirsiniz?
4. (Gerçek) bir topu (gerçek) bir eyer üzerinde dengelemek istediğinizi varsayalım.
    1. Neden bu kadar zordur?
    2. Optimizasyon algoritmaları için de bu etkiyi kullanabilir misiniz?

Tartışmalar[128]

11.2 Dışbükeylik

Dışbükeylik optimizasyon algoritmalarının tasarımında hayati bir rol oynamaktadır. Bunun nedeni, algoritmaları böyle bir bağlamda analiz etmenin ve test etmenin çok daha kolay olmasından kaynaklanmaktadır. Başka bir deyişle, algoritma dışbükey ayarda bile kötü başarım gösteriyorsa, genellikle başka türlü harika sonuçlar görmeyi ummamalıyız. Dahası, derin öğrenmedeki optimizasyon problemleri çoğunlukla dışbükey olmamasına rağmen, genellikle yerel minimumun yakınında dışbükey olanların bazı özelliklerini sergilerler. Bu, (Izmailov *et al.*, 2018) çalışmasındaki gibi heyecan verici yeni optimizasyon türlerine yol açabilir.

```
%matplotlib inline
import numpy as np
import torch
from mpl_toolkits import mplot3d
from d2l import torch as d2l
```

11.2.1 Tanımlar

Dışbükey analizden önce, *dışbükey kümeleri* ve *dışbükey fonksiyonları* tanımlamamız gerekir. Makine öğrenmesinde yaygın olarak uygulanan matematiksel araçlara yön verirler.

[128] https://discuss.d2l.ai/t/487

Dışbükey Kümeler

Kümeler dışbükeyliğin temelini oluşturur. Basitçe söylemek gerekirse, bir vektör uzayında bir $\mathcal{X}$ kümesi, $a, b \in \mathcal{X}$ için a ve b'yi bağlayan doğru parçası da $\mathcal{X}$'de ise *dışbüküm* olur. Matematiksel anlamda bu, tüm $\lambda \in [0, 1]$ için aşağıdaki ifadeye sahip olduğumuz anlamına gelir

$$\lambda a + (1 - \lambda)b \in \mathcal{X} \text{ ne zaman ki } a, b \in \mathcal{X}. \tag{11.2.1}$$

Kulağa biraz soyut geliyor. Fig. 11.2.1 figürünü düşünün. İçinde tamamı kapsanmayan doğru parçaları bulunduğundan ilk küme dışbükey değildir. Diğer iki kümede böyle bir sorun yaşanmaz.

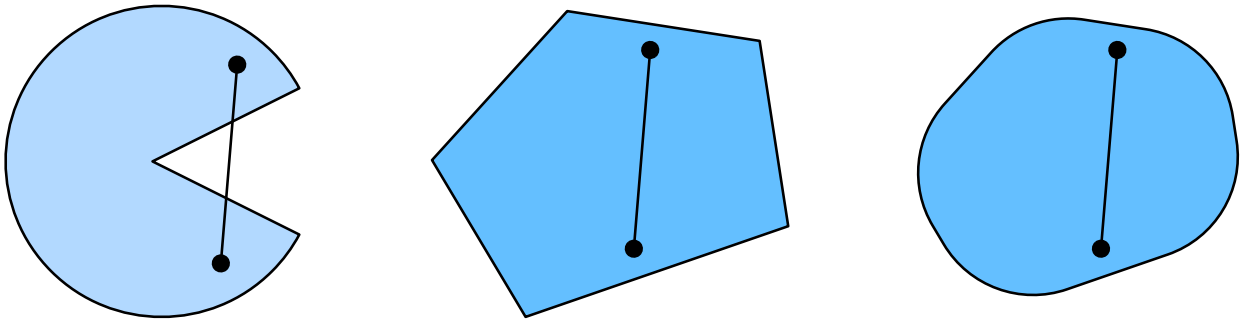


Fig. 11.2.1: İlk küme dışbükey değildir ve diğer ikisi dışbükeydir.

Onlarla bir şeyler yapamazsanız, kendi başlarına tanımlar özellikle yararlı değildir. Bu durumda Fig. 11.2.2 şeklinde gösterildiği gibi kesişimlere bakabiliriz. $\mathcal{X}$ ve $\mathcal{Y}$'nin dışbükey kümeler olduğunu varsayalım. O halde $\mathcal{X} \cap \mathcal{Y}$ de dışbükeydir. Bunu görmek için herhangi bir $a, b \in \mathcal{X} \cap \mathcal{Y}$'yi düşünün. $\mathcal{X}$ ve $\mathcal{Y}$ dışbükey olduğundan a ve b'yi bağlayan doğru parçaları hem $\mathcal{X}$ hem de $\mathcal{Y}$'de bulunur. Bu göz önüne alındığında, $\mathcal{X} \cap \mathcal{Y}$'de de yer almaları gerekiyor, böylece teoremimizi kanıtlıyor.

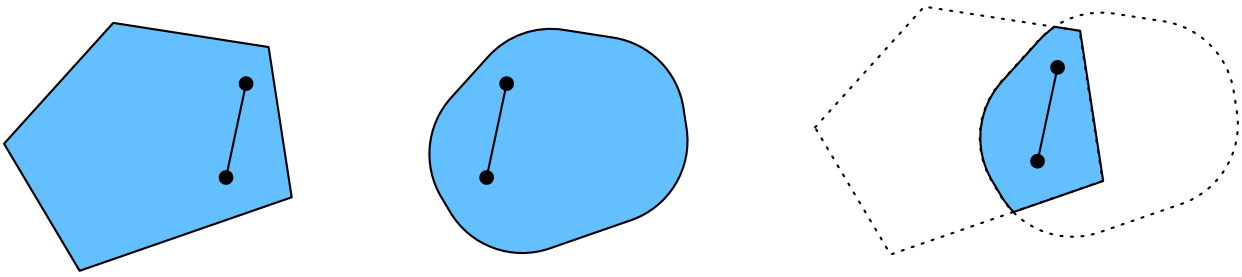


Fig. 11.2.2: İki dışbükey kümenin kesişimi dışbükeydir.

Bu sonucu az çaba ile güçlendirebiliriz: Dışbükey kümeler $\mathcal{X}_i$ göz önüne alındığında, kesişmeleri $\cap_i \mathcal{X}_i$ dışbükeydir. Tersinin doğru olmadığını görmek için, iki ayrık küme $\mathcal{X} \cap \mathcal{Y} = \emptyset$ düşünün. Şimdi $a \in \mathcal{X}$ ve $b \in \mathcal{Y}$'yi seçin. a ve b'yi birbirine bağlayan Fig. 11.2.3 içindeki doğru parçasının, $\mathcal{X}$'da veya $\mathcal{Y}$'da olmayan bir kısım içermesi gerekir, çünkü $\mathcal{X} \cap \mathcal{Y} = \emptyset$ olduğunu varsaydık. Dolayısıyla doğru parçası $\mathcal{X} \cup \mathcal{Y}$'da da değildir, bu da genel olarak dışbükey kümelerin birleşimlerinin dışbükey olması gerekmediğini kanıtlar.

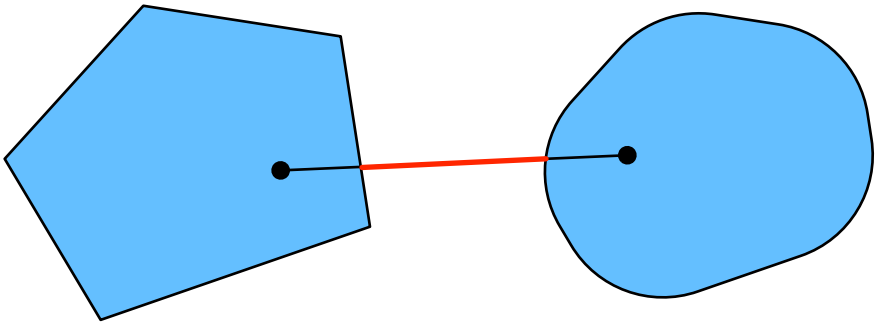


Fig. 11.2.3: İki dışbükey kümenin birleşiminin dışbükey olması gerekmez.

Derin öğrenmedeki problemler genellikle dışbükey kümeler üzerinde tanımlanır. Örneğin, $\mathbb{R}^d$, gerçek sayıların d boyutlu vektörleri kümesi, bir dışbükey kümedir (sonuçta, $\mathbb{R}^d$ içindeki herhangi iki nokta arasındaki çizgi $\mathbb{R}^d$ içinde kalır). Bazı durumlarda $\{\mathbf{x}|\mathbf{x} \in \mathbb{R}^d \text{ ve } \|\mathbf{x}\| \leq r\}$ tarafından tanımlanan r yarıçap topları gibi sınırlanmış uzunlukta değişkenlerle çalışırız.

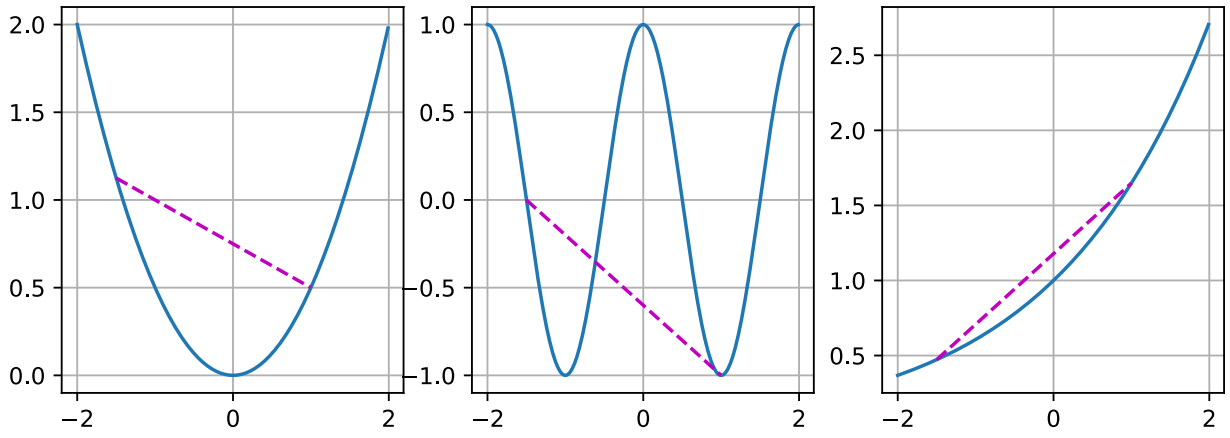
Dışbükey İşlevler

Artık dışbükey kümelere sahip olduğumuza göre *dışbükey fonksiyonları* f'yi tanıtabiliriz. Bir dışbükey $\mathcal{X}$ kümesi verildiğinde, eğer tüm $x, x' \in \mathcal{X}$ için ve elimizdeki tüm $\lambda \in [0, 1]$ için aşağıdaki ifade tutarsa $f : \mathcal{X} \to \mathbb{R}$ işlevi *dışbükeydir*:

$$\lambda f(x) + (1 - \lambda) f(x') \geq f(\lambda x + (1 - \lambda) x'). \tag{11.2.2}$$

Bunu göstermek için birkaç işlev çizelim ve hangilerinin gereksinimi karşıladığını kontrol edelim. Aşağıda hem dışbükey hem de dışbükey olmayan birkaç fonksiyon tanımlıyoruz.

```
f = lambda x: 0.5 * x**2  # Dışbükey
g = lambda x: torch.cos(np.pi * x)  # Dışbükey olmayan
h = lambda x: torch.exp(0.5 * x)  # Dışbükey

x, segment = torch.arange(-2, 2, 0.01), torch.tensor([-1.5, 1])
d2l.use_svg_display()
_, axes = d2l.plt.subplots(1, 3, figsize=(9, 3))
for ax, func in zip(axes, [f, g, h]):
    d2l.plot([x, segment], [func(x), func(segment)], axes=ax)
```



Beklendiği gibi kosinüs fonksiyonu *dışbükey* olmayan*, parabol ve üstel fonksiyon ise dışbükeydir. $\mathcal{X}$'in dışbükey bir küme olması koşulunun anlamlı olması için gerekli olduğunu unutmayın. Aksi takdirde $f(\lambda x + (1-\lambda)x')$'in sonucu iyi tanımlanmış olmayabilir.

Jensen'ın Eşitsizliği

Bir dışbükey f fonksiyonu göz önüne alındığında, en kullanışlı matematiksel araçlardan biri *Jensen eşitsizliği*dir. Dışbükeylik tanımının genelleştirilmesi anlamına gelir:

$$\sum_i \alpha_i f(x_i) \geq f\left(\sum_i \alpha_i x_i\right) \text{ ve } E_X[f(X)] \geq f\left(E_X[X]\right), \tag{11.2.3}$$

burada α_i negatif olmayan gerçel sayılar $\sum_i \alpha_i = 1$ ve X rasgele bir değişkenlerdir. Başka bir deyişle, dışbükey bir fonksiyonun beklentisi, ikincisinin genellikle daha basit bir ifade olduğu bir beklentinin dışbükey işlevinden daha az değildir. İlk eşitsizliği kanıtlamak için, dışbükeylik tanımını her seferinde toplamdaki bir terime tekrar tekrar uygularız.

Jensen'in eşitsizliğinin yaygın uygulamalarından biri daha karmaşık bir ifadeyi daha basit bir ifadeyle bağlamaktır. Örneğin, uygulaması kısmen gözlenen rastgele değişkenlerin log olabilirliği ile ilgili olabilir. Yani, kullandığımız

$$E_{Y \sim P(Y)}[-\log P(X \mid Y)] \geq -\log P(X), \tag{11.2.4}$$

çünkü $\int P(Y)P(X \mid Y)dY = P(X)$. Bu, varyasyonel yöntemlerde kullanılabilir. Burada Y tipik olarak gözlemlenmeyen rastgele değişkendir, $P(Y)$ bunun nasıl dağılabileceğine dair en iyi tahmindir ve $P(X)$, Y'nin integral uygulandığı dağılımdır. Örneğin, öbeklemede Y küme etiketleri olabilir ve $P(X \mid Y)$ öbek etiketleri uygularken üretici modeldir.

11.2.2 Özellikler

Dışbükey işlevler birçok yararlı özelliğe sahiptir. Aşağıda yaygın olarak kullanılan birkaç tanesini tanımlıyoruz.

Yerel Minimum Küresel Minimum Olduğunda

Her şeyden önce, dışbükey işlevlerin yerel minimumu da küresel minimumdur. Bunu aşağıdaki gibi tezatlıklarla kanıtlayabiliriz.

Bir dışbükey küme $\mathcal{X}$ üzerinde tanımlanmış bir dışbükey fonksiyon f düşünün. $x^* \in \mathcal{X}$'ın yerel bir minimum olduğunu varsayalım: Küçük bir pozitif p değeri vardır, öyle ki $x \in \mathcal{X}$ için $0 < |x - x^*| \leq p$ elimizde $f(x^*) < f(x)$.

Yerel minimum x^*'in f'nin küresel minimumu olmadığını varsayalım: $f(x') < f(x^*)$ için $x' \in \mathcal{X}$ vardır. Ayrıca $\lambda \in [0, 1)$ vardır, örneğin $\lambda = 1 - \frac{p}{|x^* - x'|}$ öyle ki $0 < |\lambda x^* + (1-\lambda)x' - x^*| \leq p$.

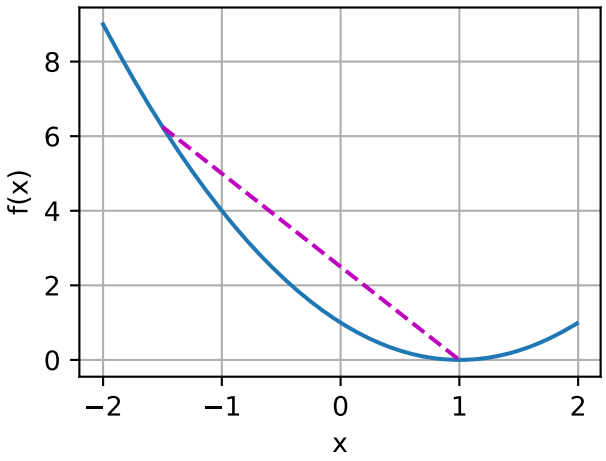
Ancak, dışbükey fonksiyonların tanımına göre,

$$\begin{aligned} f(\lambda x^* + (1-\lambda)x') &\leq \lambda f(x^*) + (1-\lambda) f(x') \\ &< \lambda f(x^*) + (1-\lambda) f(x^*) \\ &= f(x^*), \end{aligned} \tag{11.2.5}$$

bu x^*'in yerel bir minimum olduğu ifadesinde çelişmektedir. Bu nedenle, $x' \in \mathcal{X}$ için $f(x') < f(x^*)$ yok. Yerel minimum x^* da küresel minimum değerdir.

Örneğin, dışbükey işlev $f(x) = (x-1)^2$ yerel minimum $x = 1$ değerine sahiptir ve bu da küresel minimum değerdir.

```
f = lambda x: (x - 1) ** 2
d2l.set_figsize()
d2l.plot([x, segment], [f(x), f(segment)], 'x', 'f(x)')
```



Dışbükey fonksiyonlar için yerel minimum da küresel minimum olması çok uygundur. Fonksiyonları en aza indirirsek "takılıp kalmayız" anlamına gelir. Bununla birlikte, bunun birden fazla küresel minimum olamayacağı veya bir tane bile var olabileceği anlamına gelmediğini unutmayın. Örneğin, $f(x) = \max(|x| - 1, 0)$ işlevi $[-1, 1]$ aralığında minimum değerini alır. Tersine, $f(x) = \exp(x)$ işlevi $\mathbb{R}$ üzerinde minimum bir değere ulaşmaz: $x \to -\infty$ için 0'da asimtota dönüşür, ancak x için $f(x) = 0$ yoktur.

Dışbükey Fonksiyonların Aşağı Kümeleri Dışbükey Fonksiyonlardır

Dışbükey fonksiyonların *aşağıdaki kümeleri* aracılığıyla dışbükey kümeleri rahatça tanımlayabiliriz. Somut olarak, dışbükey bir $\mathcal{X}$ kümesi üzerinde tanımlanan dışbükey bir f fonksiyonu verildiğinde, herhangi bir aşağı küme

$$\mathcal{S}_b := \{x | x \in \mathcal{X} \text{ ve } f(x) \leq b\} \tag{11.2.6}$$

dışbükeydir.

Bunu çabucak kanıtlayalım. Herhangi bir $x, x' \in \mathcal{S}_b$ için $\lambda \in [0, 1]$ oldukça $\lambda x + (1-\lambda)x' \in \mathcal{S}_b$ olduğunu göstermemiz gerektiğini hatırlayın. $f(x) \leq b$ ve $f(x') \leq b$'den dolayı, dışbükeyliğin tanımı gereği aşağıdaki ifadeye sahip oluruz:

$$f(\lambda x + (1-\lambda)x') \leq \lambda f(x) + (1-\lambda) f(x') \leq b. \tag{11.2.7}$$

$f : \mathbb{R}^n \to \mathbb{R}$ fonksiyonunun ikinci türevi varsa, f'in dışbükey olup olmadığını kontrol etmek çok kolaydır. Tek yapmamız gereken f'ın Hessian'ının pozitif yarı-kesin olup olmadığını kontrol etmektir: $\nabla^2 f \succeq 0$, yani, $\nabla^2 f$ Hessian matrisini $\mathbf{H}$ ile ifade edersek $\mathbf{x}^\top \mathbf{H} \mathbf{x} \geq 0 \; \mathbf{x} \in \mathbb{R}^n$. Örneğin, $f(\mathbf{x}) = \frac{1}{2}\|\mathbf{x}\|^2$ işlevi $\nabla^2 f = \mathbf{1}$'ten dolayı dışbükeydir, yani Hessian bir birim matrisdir.

Biçimsel olarak, çift türevlenebilen tek boyutlu bir fonksiyon $f : \mathbb{R} \to \mathbb{R}$, sadece ve sadece ikinci türevi $f'' \geq 0$ ise dışbükeydir. Herhangi bir çift türevlenebilen çok boyutlu fonksiyon $f : \mathbb{R}^n \to \mathbb{R}$, sadece ve sadece Hessian $\nabla^2 f \succeq 0$ ise dışbükey olur.

İlk olarak, tek boyutlu durumu kanıtlamamız gerekiyor. f'nin dışbükeyliğinin $f'' \geq 0$'ı ima ettiğini görmek için

$$\frac{1}{2}f(x+\epsilon) + \frac{1}{2}f(x-\epsilon) \geq f\left(\frac{x+\epsilon}{2} + \frac{x-\epsilon}{2}\right) = f(x). \tag{11.2.8}$$

İkinci türev sonlu farklar üzerindeki limitle verildiğinde:

$$f''(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) + f(x-\epsilon) - 2f(x)}{\epsilon^2} \geq 0. \tag{11.2.9}$$

$f'' \geq 0$'ı görmek f'nin dışbükey olduğu anlamına gelir, burada $f'' \geq 0$'in f''nin monoton olarak azalmayan bir işlev olduğunu gösterdiğini kullandık. $a < x < b$ $\mathbb{R}$'de üç nokta olsun, öyleki $x = (1-\lambda)a + \lambda b$ ve $\lambda \in (0,1)$ olsun. Ortalama değer teoremine göre, öyle $\alpha \in [a,x]$ ve $\beta \in [x,b]$ vardır ki:

$$f'(\alpha) = \frac{f(x) - f(a)}{x-a} \text{ and } f'(\beta) = \frac{f(b) - f(x)}{b-x}. \tag{11.2.10}$$

Monotonluktan dolayı $f'(\beta) \geq f'(\alpha)$, dolayısıyla

$$\frac{x-a}{b-a}f(b) + \frac{b-x}{b-a}f(a) \geq f(x). \tag{11.2.11}$$

Çünkü $x = (1-\lambda)a + \lambda b$,

$$\lambda f(b) + (1-\lambda)f(a) \geq f((1-\lambda)a + \lambda b), \tag{11.2.12}$$

böylece dışbükeyliği kanıtlıyoruz.

İkincisi, çok boyutlu durumu kanıtlamadan önce bir önsava (lemma) ihtiyacımız var: $f : \mathbb{R}^n \to \mathbb{R}$, sadece ve sadece $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$

$$g(z) \stackrel{\text{def}}{=} f(z\mathbf{x} + (1-z)\mathbf{y}) \text{ öyle ki } z \in [0,1] \tag{11.2.13}$$

dışbükey ise dışbükeydir.

f'nin dışbükeyliğinin g'nin dışbükey olduğunu ima ettiğini kanıtlamak için, tüm $a, b, \lambda \in [0,1]$ için (böylece $0 \leq \lambda a + (1-\lambda)b \leq 1$)

$$\begin{aligned}
&g(\lambda a + (1-\lambda)b) \\
=&f\left((\lambda a + (1-\lambda)b)\,\mathbf{x} + (1 - \lambda a - (1-\lambda)b)\,\mathbf{y}\right) \\
=&f\left(\lambda\left(a\mathbf{x} + (1-a)\mathbf{y}\right) + (1-\lambda)\left(b\mathbf{x} + (1-b)\mathbf{y}\right)\right) \\
\leq&\lambda f\left(a\mathbf{x} + (1-a)\mathbf{y}\right) + (1-\lambda) f\left(b\mathbf{x} + (1-b)\mathbf{y}\right) \\
=&\lambda g(a) + (1-\lambda) g(b).
\end{aligned} \tag{11.2.14}$$

Tersini kanıtlamak için, tüm $\lambda \in [0, 1]$ ise şunu gösterebiliriz:

$$\begin{aligned} & f(\lambda \mathbf{x} + (1 - \lambda)\mathbf{y}) \\ =& g(\lambda \cdot 1 + (1 - \lambda) \cdot 0) \\ \leq& \lambda g(1) + (1 - \lambda) g(0) \\ =& \lambda f(\mathbf{x}) + (1 - \lambda) g(\mathbf{y}). \end{aligned} \tag{11.2.15}$$

Son olarak, yukarıdaki önsav ve tek boyutlu durumun sonucunu kullanarak, çok boyutlu durum aşağıdaki gibi kanıtlanabilir. Çok boyutlu bir fonksiyon $f : \mathbb{R}^n \to \mathbb{R}$, sadece ve sadece $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ $g(z) \stackrel{\text{def}}{=} f(z\mathbf{x} + (1 - z)\mathbf{y})$, burada $z \in [0, 1]$, dışbükey ise dışbükey olur. Tek boyutlu duruma göre, bu sadece ve sadece $g'' = (\mathbf{x} - \mathbf{y})^\top \mathbf{H}(\mathbf{x} - \mathbf{y}) \geq 0$ ($\mathbf{H} \stackrel{\text{def}}{=} \nabla^2 f$) tüm $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ için ($\mathbf{H} \stackrel{\text{def}}{=} \nabla^2 f$) olursa tutar, ki bu da pozitif yarı-kesin matrislerin tanımı $\mathbf{H} \succeq 0$ ile eşdeğerdir.

11.2.3 Kısıtlamalar

Dışbükey optimizasyonun güzel özelliklerinden biri, kısıtlamaları verimli bir şekilde ele almamıza izin vermesidir. Yani, *kısıtlı optimizasyon* biçimdeki sorunlarını çözmemize izin verir:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize et}} \; & f(\mathbf{x}) \\ \text{tabi } & c_i(\mathbf{x}) \leq 0 \text{ her } i \in \{1, \ldots, n\} \text{ için} , \end{aligned} \tag{11.2.16}$$

burada f amaç işlevi ve c_i fonksiyonları kısıtlama işlevleridir. Bunun ne yaptığını görmek için $c_1(\mathbf{x}) = \|\mathbf{x}\|_2 - 1$ durumunu düşünün. Bu durumda $\mathbf{x}$ parametreleri birim topla sınırlıdır. İkinci bir kısıtlama $c_2(\mathbf{x}) = \mathbf{v}^\top \mathbf{x} + b$ ise, bu yarı uzayda yatan $\mathbf{x}$'lerin tümüne karşılık gelir. Her iki kısıtlama da aynı anda tatmin etmek, bir topun bir diliminin seçilmesi anlamına gelir.

Lagrangian

Genel olarak, kısıtlı bir optimizasyon problemini çözmek zordur. Bunu ele almanın bir yolu fizikten kaynaklanan basit bir sezgidir. Bir kutunun içinde bir topu hayal edin. Top en düşük yere yuvarlanacak ve yerçekimi kuvvetleri, kutunun kenarlarının topa uygulayabileceği kuvvetlerle dengelenecektir. Kısacası, amaç fonksiyonun gradyanı (yani, yerçekimi), kısıtlama fonksiyonunun gradyanı ile dengelenecektir (topun duvarlarca "geri iterek" kutunun içinde kalması gerekir). Bazı kısıtlamaların aktif olmayabileceğini unutmayın: Topun dokunmadığı duvarlar topa herhangi bir güç uygulayamayacaktır.

Lagrangian L'nin türetilmesi es geçersek, yukarıdaki akıl yürütme, aşağıdaki eyer noktası optimizasyonu problemi ile ifade edilebilir:

$$L(\mathbf{x}, \alpha_1, \ldots, \alpha_n) = f(\mathbf{x}) + \sum_{i=1}^{n} \alpha_i c_i(\mathbf{x}) \text{ burada } \alpha_i \geq 0. \tag{11.2.17}$$

Burada α_i ($i = 1, \ldots, n$) değişkenleri, kısıtlamaların doğru şekilde uygulanmasını sağlayan *Lagrange çarpanları* olarak adlandırılır. Tüm i'lerde $c_i(\mathbf{x}) \leq 0$ sağlamak için yeterince büyük seçilir. Örneğin, $c_i(\mathbf{x}) < 0$'ın doğal olarak $c_i(\mathbf{x}) < 0$'ın $\alpha_i = 0$'ı seçtiği herhangi bir $\mathbf{x}$ için. Üstelik, bu, L'nin tüm α_i'ya göre *maksimize edilmek* ve aynı anda $\mathbf{x}$'a göre *minimize edilmek* istendiği bir eyer noktası optimizasyon problemidir. $L(\mathbf{x}, \alpha_1, \ldots, \alpha_n)$ fonksiyonuna nasıl ulaşılacağını açıklayan zengin bir yazın birikimi vardır. Amacımız için, asıl kısıtlı optimizasyon probleminin en iyi şekilde çözüldüğü L eyer noktasının nerede olduğunu bilmek yeterlidir.

Cezalar

Kısıtlı optimizasyon sorunlarını en azından *yaklaşık* tatmin etmenin bir yolu Lagrangian L'yi uyarlamaktır. $c_i(\mathbf{x}) \leq 0$'ı tatmin etmek yerine $\alpha_i c_i(\mathbf{x})$'i amaç fonksiyonu $f(x)$'e ekliyoruz. Bu, kısıtlamaların aşırı ihlal edilmemesini sağlar.

Aslında, başından beri bu hileyi kullanıyorduk. Section 4.5 içindeki ağırlık sönümünü düşünün. İçinde $\frac{\lambda}{2}\|\mathbf{w}\|^2$'i amaç işlevine $\mathbf{w}$'nin çok büyük olmamasını sağlamak için ekliyoruz. Kısıtlı optimizasyon bakış açısından bunun bize bazı r yarıçapı için $\|\mathbf{w}\|^2 - r^2 \leq 0$ sağlayacağını görebilirsiniz. λ değerinin ayarlanması, $\mathbf{w}$'nin boyutunu değiştirmemize izin verir.

Genel olarak, ceza eklemek, yaklaşık kısıtlama tatminini sağlamanın iyi bir yoludur. Pratikte bunun tam tatminden çok daha gürbüz olduğu ortaya çıkıyor. Dahası, dışbükey olmayan problemler için, dışbükey durumdaki (örn., eniyilik) kesin yaklaşımı çok çekici hale getiren özelliklerin çoğu artık tutmuyor.

İzdüşümler

Kısıtlamaları tatmin etmek için alternatif bir strateji izdüşümlerdir. Yine, daha önce onlarla karşılaştık, örneğin, Section 8.5 içinde gradyan kırpma ile uğraşırken. Orada bir gradyan θ ile sınırlanmış uzunluğa sahip olmasını sağladık

$$\mathbf{g} \leftarrow \mathbf{g} \cdot \min(1, \theta/\|\mathbf{g}\|). \tag{11.2.18}$$

Bu, θ yarıçaplı top üzerine $\mathbf{g}$'nin bir *izdüşümü* olduğu ortaya çıkarıyor. Daha genel olarak, bir dışbükey kümedeki bir izdüşüm $\mathcal{X}$ şöyle tanımlanır:

$$\text{Proj}_{\mathcal{X}}(\mathbf{x}) = \operatorname*{argmin}_{\mathbf{x}' \in \mathcal{X}} \|\mathbf{x} - \mathbf{x}'\|, \tag{11.2.19}$$

ki buda $\mathcal{X}$ içinde $\mathbf{x}$'e en yakın noktadır.

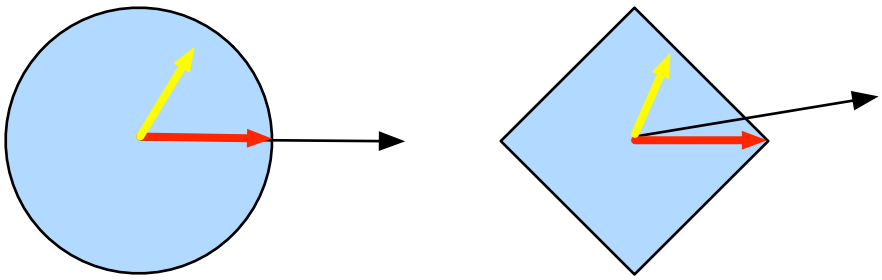


Fig. 11.2.4: Dışbükey izdüşümler.

İzdüşümlerin matematiksel tanımı biraz soyut gelebilir. Fig. 11.2.4 bunu biraz daha net bir şekilde açıklıyor. İçinde iki dışbükey küme, bir daire ve bir elmas var. Her iki kümedeki noktalar (sarı) izidüşümler esnasında değişmeden kalır. Her iki kümenin (siyah) dışındaki noktalar, asıl noktalara (siyah) yakın olan kümelerin içindeki (kırmızı) noktalara yansıtılır. L_2 topları için yön değişmeden kalırken, elmas durumunda görülebileceği gibi genel olarak böyle olması gerekmez.

Dışbükey izdüşümlerin kullanımlarından biri seyrek ağırlık vektörlerini hesaplamaktır. Bu durumda ağırlık vektörlerini L_1 topuna izdüşürüyoruz, bu da Fig. 11.2.4 içinde elmas durumunun genelleştirilmiş bir versiyonu olan bir L_1 topuna izdüşürüyoruz.

11.2.4 Özet

Derin öğrenme bağlamında dışbükey fonksiyonların temel amacı optimizasyon algoritmalarını özendirmek ve bunları ayrıntılı olarak anlamamıza yardımcı olmaktır. Sonrasında gradyan iniş ve rasgele gradyan iniş buna göre nasıl türetilebilir göreceksiniz.

- Dışbükey kümelerin kesişimleri dışbükeydir. Birleşimleri değil.
- Dışbükey bir fonksiyonun beklentisi, bir beklentinin dışbükey işlevinden daha az değildir (Jensen eşitsizliği).
- İki kez türevlenebilen bir fonksiyon, sadece ve sadece Hessian (ikinci türevlerin bir matrisi) pozitif yarı kesin ise dışbükeydir.
- Dışbükey kısıtlamalar Lagrangian aracılığıyla eklenebilir. Uygulamada, onları sadece amaç işlevine bir ceza ile ekleyebiliriz.
- İzdüşümler, orijinal noktaları dışbükey kümedeki en yakın noktalarla eşleştirirler.

11.2.5 Alıştırmalar

1. Kümedeki noktalar arasındaki tüm çizgileri çizerek ve çizgilerin içerilip içerilmediğine kontrol ederek bir kümenin dışbükeyliğini doğrulamak istediğimizi varsayalım.
    1. Sadece sınırdaki noktaları kontrol etmenin yeterli olduğunu kanıtlayın.
    2. Sadece kümenin köşelerini kontrol etmenin yeterli olduğunu kanıtlayın.
2. p-normunu kullanarak $\mathcal{B}_p[r] \stackrel{\text{def}}{=} \{\mathbf{x}|\mathbf{x} \in \mathbb{R}^d \text{ ve } \|\mathbf{x}\|_p \leq r\}$ yarıçap topu r ile belirtin. Tüm $p \geq 1$ için $\mathcal{B}_p[r]$'in dışbükey olduğunu kanıtlayın.
3. Verilen dışbükey fonksiyonlar f ve g için, $\max(f, g)$'nin de dışbükey olduğunu gösterin. $\min(f, g)$'nin dışbükey olmadığını kanıtlayın.
4. Softmax fonksiyonunun normalleştirilmesinin dışbükey olduğunu kanıtlayın. Daha özel olarak $f(x) = \log \sum_i \exp(x_i)$'in dışbükeyliğini kanıtlayın.
5. Doğrusal alt uzayların, yani $\mathcal{X} = \{\mathbf{x}|\mathbf{W}\mathbf{x} = \mathbf{b}\}$'nin dışbükey kümeler olduğunu kanıtlayın.
6. $\mathbf{b} = \mathbf{0}$ ile doğrusal alt uzaylarda $\text{Proj}_{\mathcal{X}}$ izdüşümünün bazı $\mathbf{M}$ matrisi için $\mathbf{M}\mathbf{x}$ olarak yazılabileceğini kanıtlayın.
7. İki kez türevlenebilir dışbükey f fonksiyonları için, bazı $\xi \in [0, \epsilon]$ için $f(x + \epsilon) = f(x) + \epsilon f'(x) + \frac{1}{2}\epsilon^2 f''(x + \xi)$ yazabileceğimizi gösterin.
8. Verilen $\mathbf{w} \in \mathbb{R}^d$ vektörü ve $\|\mathbf{w}\|_1 > 1$ ile L_1 birim topu üzerindeki izdüşümü hesaplayın.
    1. Bir ara adım olarak $\|\mathbf{w} - \mathbf{w}'\|^2 + \lambda\|\mathbf{w}'\|_1$ cezalandırılmış hedefini yazın ve belirli bir $\lambda > 0$ için çözümü hesaplayın.
    2. λ'nin "doğru" değerini çok fazla deneme yanılma olmadan bulabilir misiniz?
9. Bir dışbükey küme $\mathcal{X}$ ve iki vektör $\mathbf{x}$ ve $\mathbf{y}$ göz önüne alındığında, izdüşümlrin mesafeleri asla artırmadığını kanıtlayın, yani $\|\mathbf{x} - \mathbf{y}\| \geq \|\text{Proj}_{\mathcal{X}}(\mathbf{x}) - \text{Proj}_{\mathcal{X}}(\mathbf{y})\|$.

Tartışmalar[129]

[129] https://discuss.d2l.ai/t/350

11.3 Gradyan İnişi

Bu bölümde *gradyan iniş*inin altında yatan temel kavramları tanıtacağız. Derin öğrenmede nadiren kullanılmasına rağmen, gradyan inişi anlamak rasgele gradyan iniş algoritmalarını anlamak için anahtardır. Örneğin, optimizasyon problemi aşırı büyük bir öğrenme oranı nedeniyle ıraksayabilir. Bu olay halihazırda gradyan inişinde görülebilir. Benzer şekilde, ön şartlandırma gradyan inişinde yaygın bir tekniktir ve daha gelişmiş algoritmalara taşır. Basit bir özel durumla başlayalım.

11.3.1 Tek Boyutlu Gradyan İnişi

Bir boyuttaki gradyan iniş, gradyan iniş algoritmasının amaç işlevinin değerini neden azaltabileceğini açıklamak için mükemmel bir örnektir. Bir sürekli türevlenebilir gerçek değerli fonksiyon düşünün $f : \mathbb{R} \to \mathbb{R}$. Bir Taylor açılımı kullanarak şunu elde ederiz:

$$f(x + \epsilon) = f(x) + \epsilon f'(x) + \mathcal{O}(\epsilon^2). \tag{11.3.1}$$

Yani, birinci dereceden açılımda $f(x+\epsilon)$, $f(x)$ fonksiyon değeri ve x'deki birinci türev $f'(x)$ tarafından verilir. Küçük ϵ için negatif gradyan yönünde hareket etmenin f'nın azalacağını varsaymak mantıksız değildir. İşleri basit tutmak için sabit bir adım boyutu $\eta > 0$ seçip $\epsilon = -\eta f'(x)$'i seçeriz. Bunu yukarıdaki Taylor açılımı içine koyarsak:

$$f(x - \eta f'(x)) = f(x) - \eta f'^2(x) + \mathcal{O}(\eta^2 f'^2(x)). \tag{11.3.2}$$

Türev $f'(x) \neq 0$ kaybolmazsa $\eta f'^2(x) > 0$'dan dolayı ilerleme kaydediyoruz. Dahası, daha yüksek dereceden terimlerin alakasız hale gelmesi için η'yi her zaman yeterince küçük seçebiliriz. Bu nedenle şuraya varırız:

$$f(x - \eta f'(x)) \lessapprox f(x). \tag{11.3.3}$$

Bu demektir ki, bunu kullanırsak

$$x \leftarrow x - \eta f'(x) \tag{11.3.4}$$

x yinelemek için $f(x)$ işlevinin değeri azalabilir. Bu nedenle, gradyan inişinde ilk x değeri ve sabit bir $\eta > 0$ değeri seçiyoruz ve daha sonra onları durdurma koşuluna ulaşılana kadar x'i sürekli yinelemek için kullanıyoruz, örneğin $|f'(x)|$'in büyüklüğü yeterince küçük olduğunda veya yineleme sayısı belirli bir değere ulaşınca.

Basitlik için, gradyan inişinin nasıl uygulanacağını göstermek için $f(x) = x^2$ amaç işlevini seçiyoruz. $x = 0$'in $f(x)$'yi en aza indiren çözüm olduğunu bilmemize rağmen, x'in nasıl değiştiğini gözlemlemek için gene de bu basit işlevi kullanıyoruz.

```
%matplotlib inline
import numpy as np
import torch
from d2l import torch as d2l
```

```
def f(x):  # Amaç fonksiyonu
    return x ** 2

def f_grad(x):  # Amaç fonksiyonunun gradyanı (türevi)
    return 2 * x
```

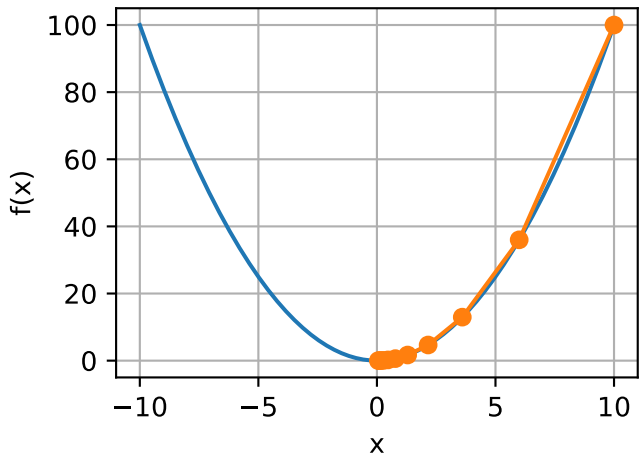
Ardından, ilk değer olarak $x = 10$'u kullanıyoruz ve $\eta = 0.2$'yi varsayıyoruz. x'i 10 kez yinelemek için gradyan iniş kullanırsak, sonunda x değerinin en iyi çözüme yaklaştığını görebiliriz.

```
def gd(eta, f_grad):
    x = 10.0
    results = [x]
    for i in range(10):
        x -= eta * f_grad(x)
        results.append(float(x))
    print(f'donem 10, x: {x:f}')
    return results

results = gd(0.2, f_grad)
```

```
donem 10, x: 0.060466
```

x üzerinde optimizasyonun ilerlemesi aşağıdaki gibi çizilebilir.

```
def show_trace(results, f):
    n = max(abs(min(results)), abs(max(results)))
    f_line = torch.arange(-n, n, 0.01)
    d2l.set_figsize()
    d2l.plot([f_line, results], [[f(x) for x in f_line], [
        f(x) for x in results]], 'x', 'f(x)', fmts=['-', '-o'])

show_trace(results, f)
```

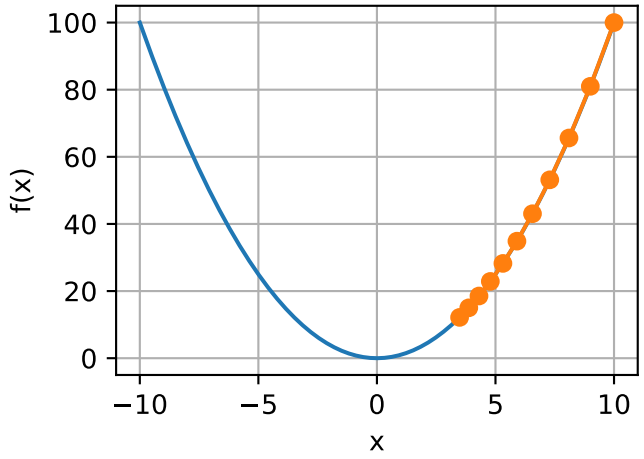


Öğrenme Oranı

η öğrenme oranı algoritma tasarımcısı tarafından ayarlanabilir. Çok küçük bir öğrenme oranı kullanırsak, x'in çok yavaş güncellenmesine neden olur ve daha iyi bir çözüm elde etmek için daha fazla yineleme gerektirir. Böyle bir durumda ne olduğunu göstermek için, $\eta = 0.05$ için aynı optimizasyon problemindeki ilerlemeyi göz önünde bulundurun. Gördüğümüz gibi, 10 adımdan sonra bile en uygun çözümden çok uzaktayız.

```
show_trace(gd(0.05, f_grad), f)
```

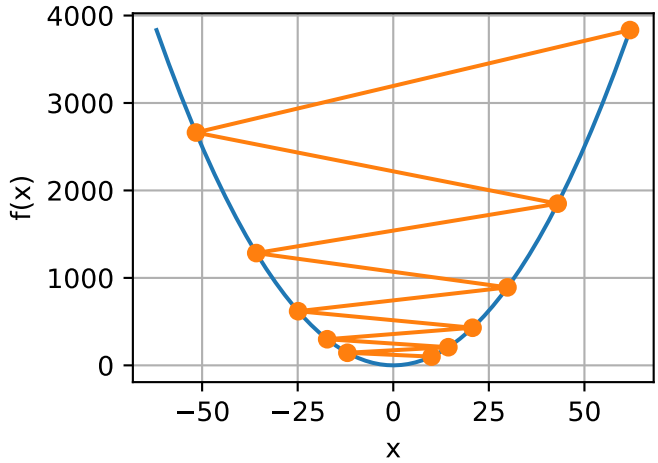
```
donem 10, x: 3.486784
```



Tersine, aşırı yüksek öğrenim oranı kullanırsak, $|\eta f'(x)|$ birinci dereceden Taylor genişleme formülü için çok büyük olabilir. Yani, (11.3.2) denklemdeki $\mathcal{O}(\eta^2 f'^2(x))$ terimi önemli hale gelebilir. Bu durumda, x yinelemesinin $f(x)$ değerini düşüreceğini garanti edemeyiz. Örneğin, öğrenme oranını $\eta = 1.1$ olarak ayarladığımızda, x optimal çözümü $x = 0$'i geçersiz kılar ve kademeli olarak ıraksar.

```
show_trace(gd(1.1, f_grad), f)
```

```
donem 10, x: 61.917364
```

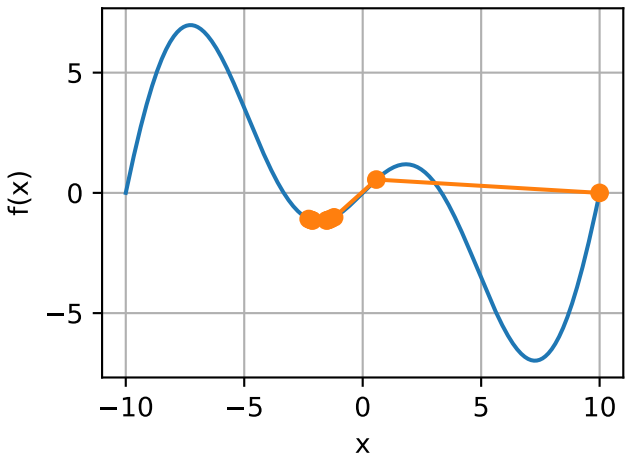


Yerel Minimum

Dışbükey olmayan fonksiyonlarde ne olduğunu göstermek için, bazı sabit c için $f(x) = x \cdot \cos(cx)$ durumunu düşünün. Bu işlevde sonsuz sayıda yerel minimum vardır. Öğrenme oranının seçimimize bağlı olarak ve sorunun ne kadar iyi şartlandırıldığına bağlı olarak, birçok çözümden birine varabiliriz. Aşağıdaki örnek, (gerçekçi olmayan) yüksek öğrenme oranının nasıl vasat bir yerel minimum seviyesine yol açacağını göstermektedir.

```
c = torch.tensor(0.15 * np.pi)

def f(x):  # Amaç fonksiyonu
    return x * torch.cos(c * x)

def f_grad(x):  # Amaç fonksiyonunun gradyanı (türevi)
    return torch.cos(c * x) - c * x * torch.sin(c * x)

show_trace(gd(2, f_grad), f)
```

```
donem 10, x: -1.528166
```



11.3.2 Çok Değişkenli Gradyan İnişi

Artık tek değişkenli durumun daha iyi bir sezgisine sahip olduğumuza göre, $\mathbf{x} = [x_1, x_2, \ldots, x_d]^\top$'nın bulunduğu hali ele alalım. Yani, amaç fonksiyonu $f : \mathbb{R}^d \to \mathbb{R}$ vektörleri skalerlere eşlesin. Buna göre gradyan çok değişkenli. d tane kısmi türevden oluşan bir vektördür:

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_d}\right]^\top. \tag{11.3.5}$$

Gradyandaki her kısmi türev elemanı $\partial f(\mathbf{x})/\partial x_i$, x_i girdisine göre $\mathbf{x}$'deki f değişiminin oranını gösterir. Tek değişkenli durumda daha önce olduğu gibi, ne yapmamız gerektiğine dair bir fikir edinmek için çok değişkenli fonksiyonlarla ilgili Taylor açılımını kullanabiliriz. Özellikle, elimizde şu var:

$$f(\mathbf{x} + \boldsymbol{\epsilon}) = f(\mathbf{x}) + \boldsymbol{\epsilon}^\top \nabla f(\mathbf{x}) + \mathcal{O}(\|\boldsymbol{\epsilon}\|^2). \tag{11.3.6}$$

Başka bir deyişle, ϵ'te ikinci dereceden terimlere kadar en dik iniş yönü $-\nabla f(\mathbf{x})$ negatif gradyan ile verilir. Uygun bir öğrenme hızı $\eta > 0$ seçilmesi, prototipik gradyan iniş algoritmasını sağlar:

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f(\mathbf{x}). \tag{11.3.7}$$

Algoritmanın pratikte nasıl davrandığını görmek için girdi olarak iki boyutlu vektörü, $\mathbf{x} = [x_1, x_2]^\top$, çıktı olarak bir skaleri ile bir amaç fonksiyonu, $f(\mathbf{x}) = x_1^2 + 2x_2^2$, oluşturalım. Gradyan $\nabla f(\mathbf{x}) = [2x_1, 4x_2]^\top$ ile verilir. $\mathbf{x}$'in yörüngesini $[-5, -2]$ ilk konumundan gradyan inişle gözlemleyeceğiz.

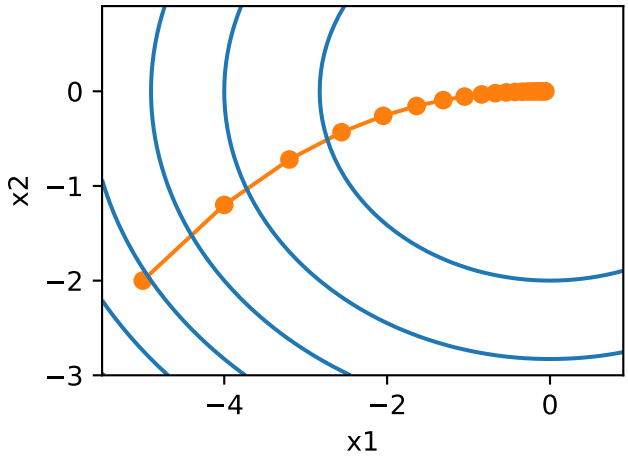
Başlangıç olarak, iki yardımcı fonksiyona daha ihtiyacımız var. Birincisi bir güncelleme işlevi kullanır ve ilk değere 20 kez uygular. İkinci yardımcı $\mathbf{x}$'in yörüngesini görselleştirir.

```
def train_2d(trainer, steps=20, f_grad=None):  #@save
    """Özelleştirilmiş bir eğitici ile 2B amaç işlevini optimize edin."""
    # `s1` ve `s2` daha sonra kullanılacak dahili durum değişkenleridir
    x1, x2, s1, s2 = -5, -2, 0, 0
    results = [(x1, x2)]
    for i in range(steps):
        if f_grad:
            x1, x2, s1, s2 = trainer(x1, x2, s1, s2, f_grad)
        else:
            x1, x2, s1, s2 = trainer(x1, x2, s1, s2)
        results.append((x1, x2))
    print(f'epoch {i + 1}, x1: {float(x1):f}, x2: {float(x2):f}')
    return results

def show_trace_2d(f, results):  #@save
    """Optimizasyon sırasında 2B değişkenlerin izini gösterin."""
    d2l.set_figsize()
    d2l.plt.plot(*zip(*results), '-o', color='#ff7f0e')
    x1, x2 = torch.meshgrid(torch.arange(-5.5, 1.0, 0.1),
                            torch.arange(-3.0, 1.0, 0.1))
    d2l.plt.contour(x1, x2, f(x1, x2), colors='#1f77b4')
    d2l.plt.xlabel('x1')
    d2l.plt.ylabel('x2')
```

Daha sonra, $\eta = 0.1$ öğrenme oranı için optimizasyon değişkeninin $\mathbf{x}$ yörüngesini gözlemliyoruz. 20 adımdan sonra $\mathbf{x}$ değerinin minimuma $[0, 0]$'da yaklaştığını görüyoruz. İlerleme oldukça yavaş olsa da oldukça iyi huyludur.

```
def f_2d(x1, x2):  # Amaç fonksiyonu
    return x1 ** 2 + 2 * x2 ** 2

def f_2d_grad(x1, x2):  # Amaç fonksiyonun gradyanı
    return (2 * x1, 4 * x2)

def gd_2d(x1, x2, s1, s2, f_grad):
    g1, g2 = f_grad(x1, x2)
    return (x1 - eta * g1, x2 - eta * g2, 0, 0)

eta = 0.1
show_trace_2d(f_2d, train_2d(gd_2d, f_grad=f_2d_grad))
```

```
epoch 20, x1: -0.057646, x2: -0.000073
/home/d2l-worker/miniconda3/envs/d2l-tr-release-0/lib/python3.9/site-packages/torch/
↪functional.py:478: UserWarning: torch.meshgrid: in an upcoming release, it will be␣
↪required to pass the indexing argument. (Triggered internally at  ../aten/src/ATen/native/
↪TensorShape.cpp:2895.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
```



11.3.3 Uyarlamalı Yöntemler

Section 11.3.1 içinde görebildiğimiz gibi, "doğru" η öğrenme oranını elde etmek çetrefillidir. Eğer çok küçük seçersek, çok az ilerleme kaydederiz. Eğer çok büyük seçersek, çözüm salınır ve en kötü ihtimalle ıraksayabilir. η'yi otomatik olarak belirleyebilirsek veya bir öğrenme oranı seçmek zorunda kalmaktan kurtulsak ne olur? Bu durumda sadece amaç fonksiyonun değerine ve gradyanlarına değil, aynı zamanda *eğri*sinin değerine de bakan ikinci dereceden yöntemler yardımcı olabilir. Bu yöntemler, hesaplama maliyeti nedeniyle doğrudan derin öğrenmeye uygulanamamakla birlikte, aşağıda belirtilen algoritmaların arzu edilen özelliklerinin çoğunu taklit eden gelişmiş optimizasyon algoritmalarının nasıl tasarlanacağına dair yararlı önseziler sağlarlar.

Newton Yöntemi

$f : \mathbb{R}^d \to \mathbb{R}$ fonksiyonun Taylor açılımının gözden geçirirsek, ilk dönemden sonra durmaya gerek yoktur. Aslında, böyle yazabilirsiniz:

$$f(\mathbf{x} + \boldsymbol{\epsilon}) = f(\mathbf{x}) + \boldsymbol{\epsilon}^\top \nabla f(\mathbf{x}) + \frac{1}{2}\boldsymbol{\epsilon}^\top \nabla^2 f(\mathbf{x})\boldsymbol{\epsilon} + \mathcal{O}(\|\boldsymbol{\epsilon}\|^3). \tag{11.3.8}$$

Hantal notasyondan kaçınmak için $\mathbf{H} \stackrel{\text{def}}{=} \nabla^2 f(\mathbf{x})$'i f'in Hessian'i olarak tanımlıyoruz, ki bu bir $d \times d$ matristir. Küçük d ve basit sorunlar için $\mathbf{H}$'nin hesaplaması kolaydır. Öte yandan derin sinir ağları için $\mathbf{H}$, $\mathcal{O}(d^2)$ girdilerinin depolanması maliyeti nedeniyle yasaklayıcı derecede büyük olabilir. Ayrıca geri yayma yoluyla hesaplamak çok pahalı olabilir. Şimdilik böyle düşünceleri göz ardı edelim ve hangi algoritmayı alacağımıza bakalım.

Sonuçta, minimum f, $\nabla f = 0$'i karşılar. Section 2.4.3 içindeki kalkülüs kurallarını izleyerek (11.3.8) türevlerini $\boldsymbol{\epsilon}$ ile ilgili olarak alıp ve yüksek dereceden terimleri göz ardı ederek şuna varırız:

$$\nabla f(\mathbf{x}) + \mathbf{H}\boldsymbol{\epsilon} = 0 \text{ ve böylece } \boldsymbol{\epsilon} = -\mathbf{H}^{-1}\nabla f(\mathbf{x}). \tag{11.3.9}$$

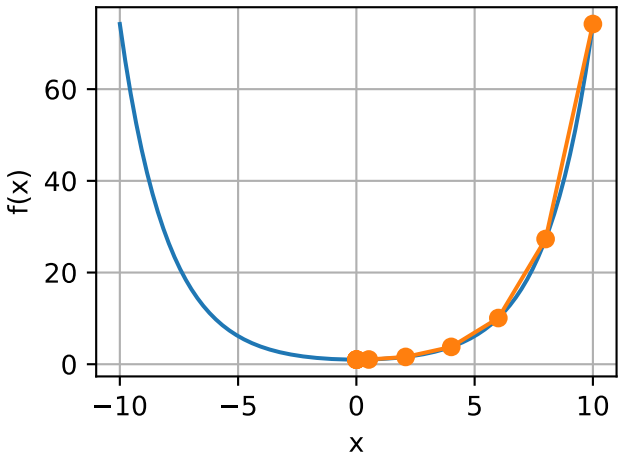
Yani, optimizasyon probleminin bir parçası olarak Hessian **H**'nin tersini almalıyız.

Basit bir örnek olarak, $f(x) = \frac{1}{2}x^2$ için $\nabla f(x) = x$ ve $\mathbf{H} = 1$'e sahibiz. Bu nedenle herhangi bir x için $\epsilon = -x$ elde ederiz. Başka bir deyişle, bir *tek* adım, herhangi bir ayarlamaya gerek kalmadan mükemmel bir şekilde yakınsama için yeterlidir! Ne yazık ki, burada biraz şanslıyız: Taylor açılımı $f(x + \epsilon) = \frac{1}{2}x^2 + \epsilon x + \frac{1}{2}\epsilon^2$'ten dolayı kesindir.

Diğer problemlerde neler olduğunu görelim. Bazı sabit c için dışbükey hiperbolik kosinüs fonksiyonu $f(x) = \cosh(cx)$ göz önüne alındığında, $x = 0$'daki küresel minimum seviyeye birkaç yinelemeden sonra ulaşıldığını görebiliriz.

```
c = torch.tensor(0.5)

def f(x):  # Amaç fonksiyonu
    return torch.cosh(c * x)

def f_grad(x):  # Amaç fonksiyonun gradyanı
    return c * torch.sinh(c * x)

def f_hess(x):  # Amaç fonksiyonunun Hessian'i
    return c**2 * torch.cosh(c * x)

def newton(eta=1):
    x = 10.0
    results = [x]
    for i in range(10):
        x -= eta * f_grad(x) / f_hess(x)
        results.append(float(x))
    print('donem 10, x:', x)
    return results

show_trace(newton(), f)
```

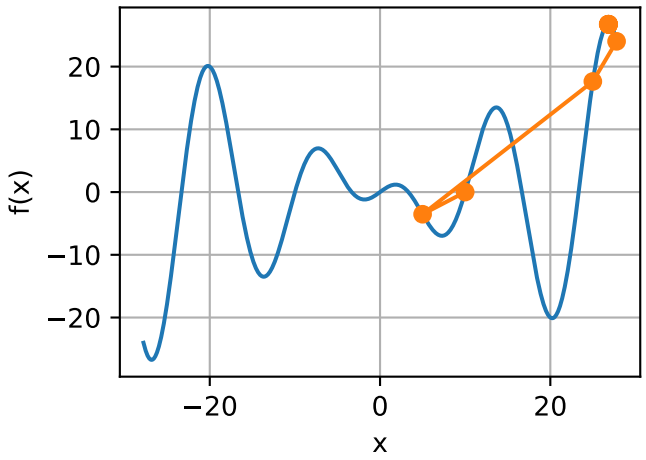
```
donem 10, x: tensor(0.)
```



Şimdi bir sabit c için $f(x) = x\cos(cx)$ gibi bir *dışbükey olmayan* işlevi düşünelim. Herşeyin sonunda, Newton'un yönteminde Hessian'a böldüğümüze dikkat edin. Bu, ikinci türev *negatif* ise f değeri *artma* yönüne doğru yürüyebileceğimiz anlamına gelir. Bu algoritmanın ölümcül kusurudur. Pratikte neler olduğunu görelim.

```
c = torch.tensor(0.15 * np.pi)

def f(x):  # Amaç fonksiyonu
    return x * torch.cos(c * x)

def f_grad(x):  # Amaç fonksiyonun gradyanı
    return torch.cos(c * x) - c * x * torch.sin(c * x)

def f_hess(x):  # Amaç fonksiyonun Hessian'i
    return - 2 * c * torch.sin(c * x) - x * c**2 * torch.cos(c * x)

show_trace(newton(), f)
```

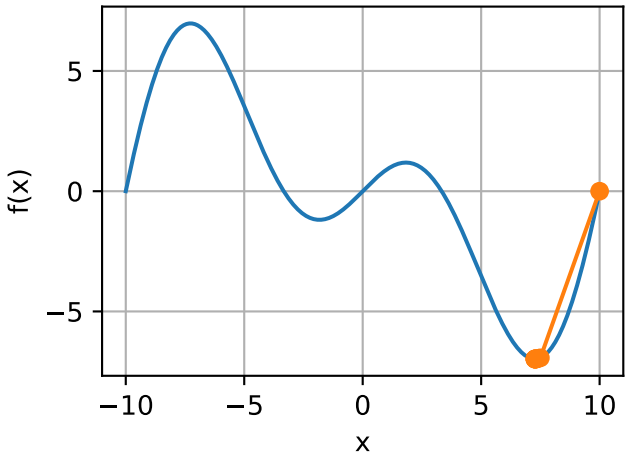
```
donem 10, x: tensor(26.8341)
```



Olağanüstü derecede yanlış gitti. Bunu nasıl düzeltebiliriz? Bunun yerine mutlak değerini alarak Hessian'i "düzeltmek" bir yol olabilir. Başka bir strateji de öğrenme oranını geri getirmektir. Bu amacı yenmek gibi görünüyor, ama tam olarak değil. İkinci dereceden bilgilere sahip olmak, eğrilik büyük olduğunda dikkatli olmamızı ve amaç fonksiyon daha düz olduğunda daha uzun adımlar atmamızı sağlar. Bunun biraz daha küçük bir öğrenme oranıyla nasıl çalıştığını görelim, $\eta = 0.5$ diyelim. Gördüğümüz gibi, oldukça verimli bir algoritmamız var.

```
show_trace(newton(0.5), f)
```

```
donem 10, x: tensor(7.2699)
```



Yakınsaklık Analizi

Biz sadece bazı dışbükey ve üç kez türevlenebilir amaç fonksiyonu f için Newton yönteminin yakınsama oranını analiz ediyoruz, burada ikinci türev sıfırdan farklı, yani $f'' > 0$. Çok değişkenli kanıt, aşağıdaki tek boyutlu argümanın basit bir uzantısıdır ve sezgi açısından bize pek yardımcı olmadığından ihmal edilir.

k'inci yinelemesinde x değerini $x^{(k)}$ ile belirtelim ve $e^{(k)} \stackrel{\text{def}}{=} x^{(k)} - x^*$ yinelemesinde eniyilikten uzaklık olmasına izin verelim. Taylor açılımı ile $f'(x^*) = 0$ durumu aşağıdaki gibi yazılabilir:

$$0 = f'(x^{(k)} - e^{(k)}) = f'(x^{(k)}) - e^{(k)} f''(x^{(k)}) + \frac{1}{2}(e^{(k)})^2 f'''(\xi^{(k)}), \tag{11.3.10}$$

ki bu bazı $\xi^{(k)} \in [x^{(k)} - e^{(k)}, x^{(k)}]$ için tutar. Yukarıdaki açılımı $f''(x^{(k)})$ ile bölmek aşağıdaki ifadeye yol açar:

$$e^{(k)} - \frac{f'(x^{(k)})}{f''(x^{(k)})} = \frac{1}{2}(e^{(k)})^2 \frac{f'''(\xi^{(k)})}{f''(x^{(k)})}. \tag{11.3.11}$$

$x^{(k+1)} = x^{(k)} - f'(x^{(k)})/f''(x^{(k)})$ güncellemesine sahip olduğumuzu hatırlayın. Bu güncelleme denklemini yerine koyarak ve her iki tarafın mutlak değerini alarak şuna ulaşırız:

$$\left|e^{(k+1)}\right| = \frac{1}{2}(e^{(k)})^2 \frac{\left|f'''(\xi^{(k)})\right|}{f''(x^{(k)})}. \tag{11.3.12}$$

Sonuç olarak, $\left|f'''(\xi^{(k)})\right| / (2f''(x^{(k)})) \leq c$ ile sınırlı olan bir bölgede olduğumuzda, dördüncü dereceden azalan bir hatamız var:

$$\left|e^{(k+1)}\right| \leq c(e^{(k)})^2. \tag{11.3.13}$$

Bir taraftan, optimizasyon araştırmacıları buna *doğrusal* yakınsama derler, oysa $\left|e^{(k+1)}\right| \leq \alpha \left|e^{(k)}\right|$ gibi bir koşul bir *sabit* yakınsama oranı olarak adlandırılır. Bu analizin bir dizi uyarıyla birlikte geldiğini unutmayın. İlk olarak, hızlı yakınsama bölgesine ulaşacağımıza dair gerçekten çok fazla bir garantimiz yok. Bunun yerine, sadece bir kez ulaştığımızda yakınsamanın çok hızlı olacağını biliyoruz. İkincisi, bu analiz f'in daha yüksek mertebeden türevlere kadar iyi-huylu olmasını gerektirir. f'ın değerlerini nasıl değiştirebileceği konusunda "şaşırtıcı" özelliklere sahip olmamasını sağlamaya geliyor.

Ön Şartlandırma

Oldukça şaşırtıcı bir şekilde hesaplamak ve tam Hessian'ı saklamak çok pahalıdır. Alternatifler bulmak bu nedenle arzu edilir. Meseleleri iyileştirmenin bir yolu da *ön şartlandırma*dır. Hessian'ın bütünüyle hesaplanmasını önler, sadece *köşesel* girdilerini hesaplar. Bu, aşağıdaki biçimdeki algoritmaların güncellenmesine yol açar:

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \mathrm{diag}(\mathbf{H})^{-1} \nabla f(\mathbf{x}). \tag{11.3.14}$$

Bu tam Newton'un yöntemi kadar iyi olmasa da, onu kullanmamaktan çok daha iyidir. Bunun neden iyi bir fikir olabileceğini görmek için bir değişkenin milimetre cinsinden yüksekliği ve diğeri kilometre cinsinden yüksekliği belirten bir durum göz önünde bulundurun. Her ikisi için doğal ölçeğin de metre olarak olduğunu varsayarsak, parametrelemelerde korkunç bir uyumsuzluğumuz var. Neyse ki, ön şartlandırma kullanmak bunu ortadan kaldırır. Gradyan inişi ile etkin bir şekilde ön koşullandırma, her değişken için farklı bir öğrenme oranı seçilmesi anlamına gelir ($\mathbf{x}$ vektörünün koordinatı). Daha sonra göreceğimiz gibi ön şartlandırma, rasgele gradyan iniş optimizasyon algoritmalarındaki bazı yaratıcılığı yönlendirir.

Doğru Üzerinde Arama ile Gradyan İnişi

Gradyan inişindeki en önemli sorunlardan biri, hedefi aşmamız veya yetersiz ilerleme kaydetmemizdir. Sorun için basit bir düzeltme, gradyan iniş ile birlikte doğru üzerinde arama kullanmaktır. Yani, $\nabla f(\mathbf{x})$ tarafından verilen yönü kullanırız ve ardından η öğrenme oranının $f(\mathbf{x} - \eta \nabla f(\mathbf{x}))$'yi en aza indirmek için ikili arama yaparız.

Bu algoritma hızla yakınsar (analiz ve kanıt için bkz., (Boyd and Vandenberghe, 2004)). Ancak, derin öğrenme amacıyla bu o kadar da mümkün değildir, çünkü doğru üzerinde aramanın her adımı, tüm veri kümesindeki amaç fonksiyonunu değerlendirmemizi gerektirir. Bunu tamamlamak çok maliyetlidir.

11.3.4 Özet

- Öğrenme oranları önemlidir. Çok büyükse ıraksarız, çok küçükse ilerleme kaydetmeyiz.
- Gradyan inişi yerel minimumda takılabilir.
- Yüksek boyutlarda öğrenme oranının ayarlanması karmaşıktır.
- Ön şartlandırma ayarların ölçeklendirilmesine yardımcı olabilir.
- Newton'un yöntemi dışbükey problemlerde düzgün çalışmaya başladıktan sonra çok daha hızlıdır.
- Dışbükey olmayan problemler için herhangi bir ayarlama yapmadan Newton'un yöntemini kullanmaktan sakının.

11.3.5 Alıştırmalar

1. Gradyan inişi için farklı öğrenme oranları ve amaç fonksiyonları ile deney yapın.
2. $[a, b]$ aralığında dışbükey işlevini en aza indirmek için doğru üzerinde arama gerçekleştirin.
    1. İkili arama için türevlere ihtiyacınız var mı, mesela $[a, (a + b)/2]$ veya $[(a + b)/2, b]$'yi seçip seçmeyeceğinize karar vermek için?
    2. Algoritma için yakınsama oranı ne kadar hızlıdır?
    3. Algoritmayı uygulayın ve $\log(\exp(x) + \exp(-2x - 3))$'ü en aza indirmek için uygulayın.
3. Gradyan inişinin son derece yavaş olduğu $\mathbb{R}^2$'te tanımlanan bir amaç fonksiyonu tasarlayın. İpucu: Farklı koordinatları farklı şekilde ölçeklendirin.
4. Aşağıdaki ön şartlandırmaları kullanarak Newton yönteminin hafifsiklet versiyonunu uygulayın:
    1. Ön koşul olarak köşegen Hessian kullanın.
    2. Gerçek (muhtemelen işaretli) değerler yerine bunun mutlak değerlerini kullanın.
    3. Bunu yukarıdaki probleme uygulayın.
5. Yukarıdaki algoritmayı bir takım amaç fonksiyonuna uygulayın (dışbükey olan veya olmayan). Koordinatları 45 derece döndürürseniz ne olur?

Tartışmalar[130]

11.4 Rasgele Gradyan İnişi

Daha önceki bölümlerde, eğitim prosedürümüzde rasgele gradyan inişi kullanmaya devam ettik, ancak, neden çalıştığını açıklamadık. Üzerine biraz ışık tutmak için, Section 11.3 içinde gradyan inişinin temel prensiplerini tanımladık. Bu bölümde, *rasgele gradyan inişini* daha ayrıntılı olarak tartışmaya devam ediyoruz.

```
%matplotlib inline
import math
import torch
from d2l import torch as d2l
```

11.4.1 Rasgele Gradyan Güncellemeleri

Derin öğrenmede, amaç işlevi genellikle eğitim veri kümelerindeki her örnek için kayıp fonksiyonlarının ortalamasıdır. n örnekten oluşan bir eğitim veri kümesi verildiğinde, $f_i(\mathbf{x})$'in $i.$ dizindeki eğitim örneğine göre kayıp işlevi olduğunu varsayarız, burada $\mathbf{x}$ parametre vektörüdür. Sonra şu amaç fonksiyonuna varırız

$$f(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^{n} f_i(\mathbf{x}). \tag{11.4.1}$$

[130] https://discuss.d2l.ai/t/351

$\mathbf{x}$'teki amaç fonksiyonunun gradyanı böyle hesaplanır:

$$\nabla f(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\mathbf{x}). \tag{11.4.2}$$

Gradyan inişi kullanılıyorsa, her bağımsız değişken yineleme için hesaplama maliyeti $\mathcal{O}(n)$'dir ve n ile doğrusal olarak büyür. Bu nedenle, eğitim veri kümesi daha büyük olduğunda, her yineleme için gradyan iniş maliyeti daha yüksek olacaktır.

Rasgele gradyan iniş (SGD), her yinelemede hesaplama maliyetini düşürür. Rasgele gradyan inişin her yinelemesinde, rastgele veri örnekleri için $i \in \{1, \ldots, n\}$ dizinden eşit olarak örneklemleriz ve $\mathbf{x}$'i güncelleştirmek için $\nabla f_i(\mathbf{x})$ gradyanını hesaplarız:

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f_i(\mathbf{x}), \tag{11.4.3}$$

burada η öğrenme oranıdır. Her yineleme için gradyan inişinin hesaplama maliyetinin $\mathcal{O}(n)$'den $\mathcal{O}(1)$ sabitine düştüğünü görebiliriz. Dahası, rasgele gradyan $\nabla f_i(\mathbf{x})$'in $\nabla f(\mathbf{x})$'in tam gradyanının tarafsız bir tahmini olduğunu vurgulamak istiyoruz, çünkü

$$\mathbb{E}_i \nabla f_i(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\mathbf{x}) = \nabla f(\mathbf{x}). \tag{11.4.4}$$

Bu, ortalama olarak rasgele gradyanın, gradyanın iyi bir tahmini olduğu anlamına gelir.

Şimdi, bir rasgele gradyan inişini benzetmek için gradyana ortalaması 0 ve varyansı 1 olan rastgele gürültü ekleyerek gradyan inişiyle karşılaştıracağız.

```
def f(x1, x2):  # Amaç fonksiyonu
    return x1 ** 2 + 2 * x2 ** 2

def f_grad(x1, x2):  # Amaç fonksiyonun gradyanı
    return 2 * x1, 4 * x2
```

```
def sgd(x1, x2, s1, s2, f_grad):
    g1, g2 = f_grad(x1, x2)
    # Gürültülü gradyan benzetimi
    g1 += torch.normal(0.0, 1, (1,))
    g2 += torch.normal(0.0, 1, (1,))
    eta_t = eta * lr()
    return (x1 - eta_t * g1, x2 - eta_t * g2, 0, 0)
```

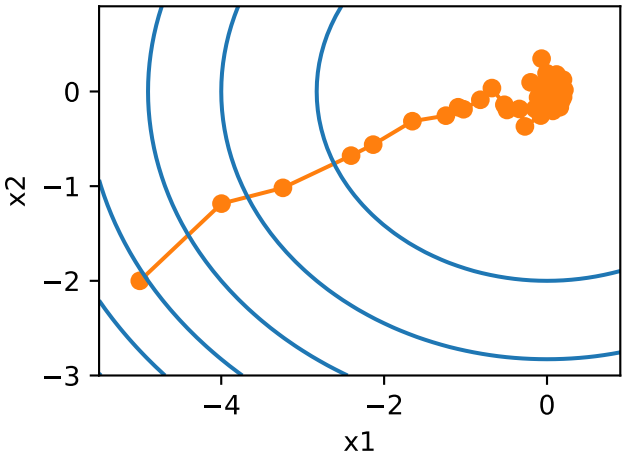
```
def constant_lr():
    return 1

eta = 0.1
lr = constant_lr  # Sabit öğrenme oranı
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=50, f_grad=f_grad))
```

```
epoch 50, x1: 0.174384, x2: -0.102252
/home/d2l-worker/miniconda3/envs/d2l-tr-release-0/lib/python3.9/site-packages/numpy/core/
↪shape_base.py:65: VisibleDeprecationWarning: Creating an ndarray from ragged nested␣
↪sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths␣
```

(continues on next page)

(continued from previous page)

```
↪or shapes) is deprecated. If you meant to do this, you must specify 'dtype=object' when␣
↪creating the ndarray.
  ary = asanyarray(ary)
/home/d2l-worker/miniconda3/envs/d2l-tr-release-0/lib/python3.9/site-packages/torch/
↪functional.py:478: UserWarning: torch.meshgrid: in an upcoming release, it will be␣
↪required to pass the indexing argument. (Triggered internally at  ../aten/src/ATen/native/
↪TensorShape.cpp:2895.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
```



Gördüğümüz gibi, rasgele gradyan inişindeki değişkenlerin yörüngesi, Section 11.3 içinde gradyan inişinde gözlemlediğimizden çok daha gürültülüdür. Bunun nedeni, gradyanın rasgele doğasından kaynaklanmaktadır. Yani, en düşük seviyeye yaklaştığımızda bile, $\eta \nabla f_i(\mathbf{x})$ aracılığıyla anlık gradyan tarafından aşılanan belirsizliğe hala tabiyiz. 50 adımdan sonra bile kalite hala o kadar iyi değil. Daha da kötüsü, ek adımlardan sonra iyileşmeyecektir (bunu onaylamak için daha fazla sayıda adım denemenizi öneririz). Bu bize tek alternatif bırakır: Öğrenme oranı η'yı değiştirmek. Ancak, bunu çok küçük seçersek, ilkin bir ilerleme kaydetmeyeceğiz. Öte yandan, eğer çok büyük alırsak, yukarıda görüldüğü gibi iyi bir çözüm elde edemeyiz. Bu çelişkili hedefleri çözmenin tek yolu, optimizasyon ilerledikçe öğrenme oranını *dinamik* azaltmaktır.

Bu aynı zamanda `sgd` adım işlevine `lr` öğrenme hızı işlevinin eklenmesinin de sebebidir. Yukarıdaki örnekte, ilişkili 'lr' işlevini sabit olarak ayarladığımız için öğrenme hızı planlaması için herhangi bir işlevsellik uykuda kalmaktadır.

11.4.2 Dinamik Öğrenme Oranı

η'nın zamana bağlı öğrenme hızı $\eta(t)$ ile değiştirilmesi, optimizasyon algoritmasının yakınsamasını kontrol etmenin karmaşıklığını arttırır. Özellikle, η'nın ne kadar hızlı sönmesi gerektiğini bulmamız gerekiyor. Çok hızlıysa, eniyilemeyi erken bırakacağız. Eğer çok yavaş azaltırsak, optimizasyon için çok fazla zaman harcarız. Aşağıdakiler, η'nın zamanla ayarlanmasında kullanılan birkaç temel stratejidir (daha sonra daha gelişmiş stratejileri tartışacağız):

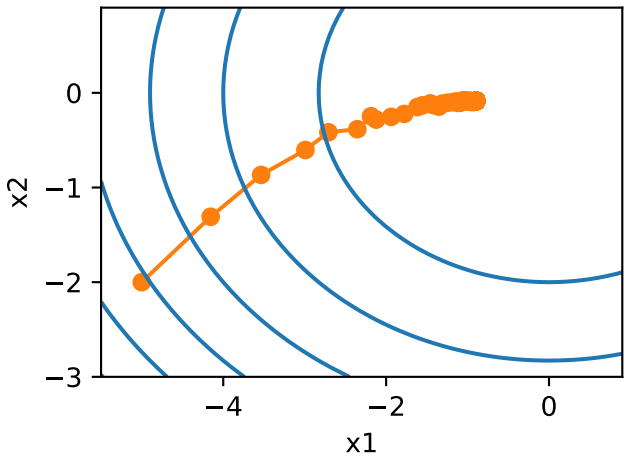
$$\begin{aligned} \eta(t) &= \eta_i \text{ eğer } t_i \leq t \leq t_{i+1} && \text{parçalı sabit} \\ \eta(t) &= \eta_0 \cdot e^{-\lambda t} && \text{üstel sönüm} \\ \eta(t) &= \eta_0 \cdot (\beta t + 1)^{-\alpha} && \text{polinomsal sönüm} \end{aligned} \tag{11.4.5}$$

İlk *parçalı sabit* senaryosunda, öğrenme oranını düşürüyoruz, mesela optimizasyondaki ilerleme durduğunda. Bu, derin ağları eğitmek için yaygın bir stratejidir. Alternatif olarak çok daha saldırgan bir *üstel sönüm* ile azaltabiliriz. Ne yazık ki bu genellikle algoritma yakınsamadan önce erken durmaya yol açar. Popüler bir seçim, $\alpha = 0.5$ ile *polinom sönüm*dür. Dışbükey optimizasyon durumunda, bu oranın iyi davrandığını gösteren bir dizi kanıt vardır.

Üstel sönmenin pratikte neye benzediğini görelim.

```
def exponential_lr():
    # Bu fonksiyonun dışında tanımlanan ve içinde güncellenen global değişken
    global t
    t += 1
    return math.exp(-0.1 * t)

t = 1
lr = exponential_lr
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=1000, f_grad=f_grad))
```

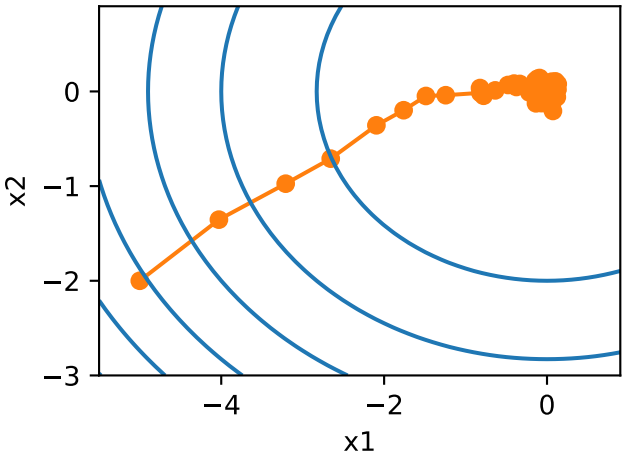
```
epoch 1000, x1: -0.895644, x2: -0.085209
```



Beklendiği gibi, parametrelerdeki varyans önemli ölçüde azaltılır. Bununla birlikte, bu, $\mathbf{x} = (0, 0)$ eniyi çözümüne yakınsamama pahasına gelir. Hatta sonra 1000 yineleme adımında sonra bile eniyi çözümden hala çok uzaktayız. Gerçekten de, algoritma yakınsamada tam anlamıyla başarısız. Öte yandan, öğrenme oranının adım sayısının ters karekökü ile söndüğü polinom sönmesi kullanırsak yakınsama sadece 50 adımdan sonra daha iyi olur.

```
def polynomial_lr():
    # Bu fonksiyonun dışında tanımlanan ve içinde güncellenen global değişken
    global t
    t += 1
    return (1 + 0.1 * t) ** (-0.5)

t = 1
lr = polynomial_lr
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=50, f_grad=f_grad))
```

```
epoch 50, x1: 0.123653, x2: 0.020916
```



Öğrenme oranının nasıl ayarlanacağı konusunda çok daha fazla seçenek var. Örneğin, küçük bir hızla başlayabilir, sonra hızlıca yükseltebilir ve daha yavaş da olsa tekrar azaltabiliriz. Hatta daha küçük ve daha büyük öğrenme oranları arasında geçiş yapabiliriz. Bu tür ayarlamaların çok çeşidi vardır. Şimdilik kapsamlı bir teorik analizin mümkün olduğu öğrenme oranı ayarlamalarına odaklanalım, yani dışbükey bir ortamda öğrenme oranları üzerine. Genel dışbükey olmayan problemler için anlamlı yakınsama garantileri elde etmek çok zordur, çünkü genel olarak doğrusal olmayan dışbükey problemlerin en aza indirilmesi NP zorludur. Bir araştırma için örneğin, Tibshirani'nin 2015'teki mükemmel ders notları[131]na bakınız.

11.4.3 Dışbükey Amaç İşlevleri için Yakınsaklık Analizi

Dışbükey amaç fonksiyonları için rasgele gradyan inişinin aşağıdaki yakınsama analizi isteğe bağlıdır ve öncelikle problem hakkında daha fazla sezgi iletmek için hizmet vermektedir. Kendimizi en basit kanıtlardan biriyle sınırlıyoruz (Nesterov and Vial, 2000). Önemli ölçüde daha gelişmiş ispat teknikleri mevcuttur, örn. amaç fonksiyonu özellikle iyi-huyluysa.

$f(\boldsymbol{\xi}, \mathbf{x})$'in amaç işlevinin $\boldsymbol{\xi}$'nin tümü için $\mathbf{x}$'de dışbükey olduğunu varsayalım. Daha somut olarak, rasgele gradyan iniş güncellemesini aklımızda bulunduruyoruz:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta_t \partial_{\mathbf{x}} f(\boldsymbol{\xi}_t, \mathbf{x}), \tag{11.4.6}$$

burada $f(\boldsymbol{\xi}_t, \mathbf{x})$, bir dağılımdan t adımında çekilen $\boldsymbol{\xi}_t$ eğitim örneğine göre amaç fonksiyonudur ve $\mathbf{x}$ model parametresidir. Aşağıda

$$R(\mathbf{x}) = E_{\boldsymbol{\xi}}[f(\boldsymbol{\xi}, \mathbf{x})] \tag{11.4.7}$$

beklenen riski ifade eder ve R^*, $\mathbf{x}$'e göre en düşük değerdir. Son olarak $\mathbf{x}^*$ küçültücü (minimizer) olsun ($\mathbf{x}$'in tanımlandığı etki alanında var olduğunu varsayıyoruz). Bu durumda t zamanında $\mathbf{x}_t$ ve risk küçültücü $\mathbf{x}^*$ arasındaki mesafeyi izleyebilir ve zamanla iyileşip iyileşmediğini görebiliriz:

$$\begin{aligned} &\|\mathbf{x}_{t+1} - \mathbf{x}^*\|^2 \\ =&\|\mathbf{x}_t - \eta_t \partial_{\mathbf{x}} f(\boldsymbol{\xi}_t, \mathbf{x}) - \mathbf{x}^*\|^2 \\ =&\|\mathbf{x}_t - \mathbf{x}^*\|^2 + \eta_t^2 \|\partial_{\mathbf{x}} f(\boldsymbol{\xi}_t, \mathbf{x})\|^2 - 2\eta_t \left\langle \mathbf{x}_t - \mathbf{x}^*, \partial_{\mathbf{x}} f(\boldsymbol{\xi}_t, \mathbf{x}) \right\rangle. \end{aligned} \tag{11.4.8}$$

[131] https://www.stat.cmu.edu/~ryantibs/convexopt-F15/lectures/26-nonconvex.pdf

Rasgele gradyan $\partial_{\mathbf{x}} f(\boldsymbol{\xi}_t, \mathbf{x})$'in L_2 normunun bir L sabiti ile sınırlandığını varsayıyoruz, dolayısıyla

$$\eta_t^2 \|\partial_{\mathbf{x}} f(\boldsymbol{\xi}_t, \mathbf{x})\|^2 \leq \eta_t^2 L^2. \tag{11.4.9}$$

Çoğunlukla $\mathbf{x}_t$ ve $\mathbf{x}^*$ arasındaki mesafenin *beklenti*de nasıl değiştiğiyle ilgileniyoruz. Aslında, herhangi bir belirli adım dizisi için, karşılaştığımız herhangi $\boldsymbol{\xi}_t$'ye bağlı olarak mesafe artabilir. Bu yüzden nokta çarpımını sınırlamamız gerekiyor. Herhangi bir dışbükey fonksiyon f için $f(\mathbf{y}) \geq f(\mathbf{x}) + \langle f'(\mathbf{x}), \mathbf{y} - \mathbf{x}\rangle$'nin tüm $\mathbf{x}$ ve $\mathbf{y}$ için, dışbükeylik ile şuna varırız:

$$f(\boldsymbol{\xi}_t, \mathbf{x}^*) \geq f(\boldsymbol{\xi}_t, \mathbf{x}_t) + \langle \mathbf{x}^* - \mathbf{x}_t, \partial_{\mathbf{x}} f(\boldsymbol{\xi}_t, \mathbf{x}_t)\rangle \,. \tag{11.4.10}$$

(11.4.9) ve (11.4.10) içindeki her iki eşitsizliği (11.4.8) denklemine yerleştirerek parametreler arasındaki mesafeye $t+1$ zamanında aşağıdaki gibi sınır koyarız:

$$\|\mathbf{x}_t - \mathbf{x}^*\|^2 - \|\mathbf{x}_{t+1} - \mathbf{x}^*\|^2 \geq 2\eta_t (f(\boldsymbol{\xi}_t, \mathbf{x}_t) - f(\boldsymbol{\xi}_t, \mathbf{x}^*)) - \eta_t^2 L^2. \tag{11.4.11}$$

Bu, mevcut kayıp ile optimal kayıp arasındaki fark $\eta_t L^2/2$'ten ağır bastığı sürece ilerleme kaydettiğimiz anlamına gelir. Bu fark sıfıra yakınsamaya bağlı olduğundan, η_t öğrenme oranının da *yok olması* gerekir.

Sonrasında (11.4.11) üzerinden beklentileri alırız. Şu sonuca varırız:

$$E\left[\|\mathbf{x}_t - \mathbf{x}^*\|^2\right] - E\left[\|\mathbf{x}_{t+1} - \mathbf{x}^*\|^2\right] \geq 2\eta_t [E[R(\mathbf{x}_t)] - R^*] - \eta_t^2 L^2. \tag{11.4.12}$$

Son adım $t \in \{1, \ldots, T\}$ için eşitsizlikler üzerinde toplamayı içerir. Toplam içeriye daralır ve düşük terimi düşürürsek şunu elde ederiz:

$$\|\mathbf{x}_1 - \mathbf{x}^*\|^2 \geq 2 \left(\sum_{t=1}^{T} \eta_t\right) [E[R(\mathbf{x}_t)] - R^*] - L^2 \sum_{t=1}^{T} \eta_t^2. \tag{11.4.13}$$

$\mathbf{x}_1$'in verildiğini ve böylece beklentinin düştüğünü unutmayın. Son tanımımız

$$\bar{\mathbf{x}} \stackrel{\text{def}}{=} \frac{\sum_{t=1}^{T} \eta_t \mathbf{x}_t}{\sum_{t=1}^{T} \eta_t}. \tag{11.4.14}$$

Çünkü

$$E\left(\frac{\sum_{t=1}^{T} \eta_t R(\mathbf{x}_t)}{\sum_{t=1}^{T} \eta_t}\right) = \frac{\sum_{t=1}^{T} \eta_t E[R(\mathbf{x}_t)]}{\sum_{t=1}^{T} \eta_t} = E[R(\mathbf{x}_t)], \tag{11.4.15}$$

Jensen'in eşitsizliği ile ($i = t$, (11.2.3) içinde $\alpha_i = \eta_t / \sum_{t=1}^{T} \eta_t$ ayarlarız) ve R dışbükeyliği ile $E[R(\mathbf{x}_t)] \geq E[R(\bar{\mathbf{x}})]$, şuna ulaşırız:

$$\sum_{t=1}^{T} \eta_t E[R(\mathbf{x}_t)] \geq \sum_{t=1}^{T} \eta_t E\left[R(\bar{\mathbf{x}})\right]. \tag{11.4.16}$$

Bunu (11.4.13) eşitsizliğinin içine koymak aşağıdaki sınırı verir:

$$[E[\bar{\mathbf{x}}]] - R^* \leq \frac{r^2 + L^2 \sum_{t=1}^{T} \eta_t^2}{2 \sum_{t=1}^{T} \eta_t}, \tag{11.4.17}$$

burada $r^2 \stackrel{\text{def}}{=} \|\mathbf{x}_1 - \mathbf{x}^*\|^2$, parametrelerin ilk seçimi ile nihai sonuç arasındaki mesafeye bağlıdır. Kısacası, yakınsama hızı, rasgele gradyanın normunun nasıl sınırlandığına (L) ve ilk parametre (r) değerinin eniyi değerden ne kadar uzakta olduğuna bağlıdır. Sınırın $\mathbf{x}_T$ yerine $\bar{\mathbf{x}}$ cinsinden olduğuna dikkat edin. $\bar{\mathbf{x}}$'in optimizasyon yolunun yumuşatılmış bir sürümü olduğu için bu durum böyledir. r, L ve T bilindiğinde öğrenme oranını $\eta = r/(L\sqrt{T})$ alabiliriz. Bu, üst sınırı $rL/\sqrt{T}$ olarak verir. Yani, $\mathcal{O}(1/\sqrt{T})$ oranıyla eniyi çözüme yakınsıyoruz.

11.4.4 Rasgele Gradyanlar ve Sonlu Örnekler

Şimdiye kadar, rasgele gradyan inişi hakkında konuşurken biraz hızlı ve gevşek hareket ettik. x_i örneklerini, tipik olarak y_i etiketleriyle bazı $p(x, y)$ dağılımlardan çektiğimizi ve bunu model parametrelerini bir şekilde güncellemek için kullandığımızı belirttik. Özellikle, sonlu bir örnek boyutu için $p(x, y) = \frac{1}{n} \sum_{i=1}^{n} \delta_{x_i}(x)\delta_{y_i}(y)$ ayrık dağılımın olduğunu savunduk. Bazı δ_{x_i} ve δ_{y_i} fonksiyonları için üzerinde rasgele gradyan inişini gerçekleştirmemizi sağlar.

Ancak, gerçekten yaptığımız şey bu değildi. Mevcut bölümdeki basit örneklerde, aksi takdirde rasgele olmayan bir gradyana gürültü ekledik, yani (x_i, y_i) çiftleri varmış gibi davrandık. Bunun burada haklı olduğu ortaya çıkıyor (ayrıntılı bir tartışma için alıştırmalara bakın). Daha rahatsız edici olan, önceki tüm tartışmalarda bunu açıkça yapmadığımızdır. Bunun neden tercih edilebilir olduğunu görmek için, tersini düşünün, yani ayrık dağılımdan *değiştirmeli* n gözlemleri örnekliyoruz. Rastgele i elemanını seçme olasılığı $1/n$'dır. Böylece onu *en azından* bir kez seçeriz:

$$P(\text{seç } i) = 1 - P(\text{yoksay } i) = 1 - (1 - 1/n)^n \approx 1 - e^{-1} \approx 0.63. \tag{11.4.18}$$

Benzer bir akıl yürütme, bir numunenin (yani eğitim örneği) *tam bir kez* seçme olasılığının şöyle verildiğini göstermektedir:

$$\binom{n}{1}\frac{1}{n}\left(1 - \frac{1}{n}\right)^{n-1} = \frac{n}{n-1}\left(1 - \frac{1}{n}\right)^{n} \approx e^{-1} \approx 0.37. \tag{11.4.19}$$

Bu, *değiştirmesiz* örneklemeye göreceli oranla artan bir varyansa ve azalan veri verimliliğine yol açar. Bu nedenle, pratikte ikincisini gerçekleştiriyoruz (ve bu kitap boyunca varsayılan seçimdir). Son olarak eğitim veri kümesindeki tekrarlanan geçişler ondan *farklı* rastgele sırayla geçer.

11.4.5 Özet

- Dışbükey problemler için, geniş bir öğrenme oranları seçeneği için rasgele gradyan inişinin eniyi çözüme yakınsayacağını kanıtlayabiliriz.
- Derin öğrenme için bu genellikle böyle değildir. Bununla birlikte, dışbükey problemlerin analizi, optimizasyona nasıl yaklaşılacağına dair, yani öğrenme oranını çok hızlı olmasa da aşamalı olarak azaltmak için, bize yararlı bilgiler verir.
- Öğrenme oranı çok küçük veya çok büyük olduğunda sorunlar ortaya çıkar. Pratikte uygun bir öğrenme oranı genellikle sadece birden fazla deneyden sonra bulunur.
- Eğitim veri kümesinde daha fazla örnek olduğunda, gradyan inişi için her yinelemede hesaplamak daha pahalıya mal olur, bu nedenle bu durumlarda rasgele gradyan inişi tercih edilir.
- Rasgele gradyan inişi için optimizasyon garantileri genel olarak dışbükey olmayan durumlarda mevcut değildir, çünkü kontrol gerektiren yerel minimum sayısı da üstel olabilir.

11.4.6 Alıştırmalar

1. Rasgele gradyan inişini farklı sayıda yineleme ile farklı öğrenme oranı düzenleri ile deneyin. Özellikle, yineleme sayısının bir fonksiyonu olarak optimal çözüm $(0, 0)$'dan uzaklığı çizin.
2. $f(x_1, x_2) = x_1^2 + 2x_2^2$ işlevi için gradyana normal gürültü eklemenin bir kayıp işlevini $f(\mathbf{x}, \mathbf{w}) = (x_1 - w_1)^2 + 2(x_2 - w_2)^2$ en aza indirmeye eşdeğer olduğunu kanıtlayın, burada $\mathbf{x}$ normal dağılımdan çekilmektedir.
3. $\{(x_1, y_1), \ldots, (x_n, y_n)\}$'ten değiştirmeli ve değiştirmesiz örnek aldığınızda rasgele gradyan inişinin yakınsamasını karşılaştırın.
4. Bir gradyan (veya onunla ilişkili bir koordinat) diğer tüm gradyanlardan tutarlı bir şekilde daha büyükse, rasgele gradyan inişi çözücüsünü nasıl değiştirirsiniz?
5. Bunu varsayalım: $f(x) = x^2(1 + \sin x)$. f'te kaç tane yerel minimum vardır? f', en aza indirgemek için tüm yerel minimumları değerlendirmek zorunda kalacak şekilde değiştirebilir misiniz?

Tartışmalar[132]

11.5 Minigrup Rasgele Gradyan İnişi

Şimdiye kadar gradyan tabanlı öğrenme yaklaşımında iki uç noktaya rastladık: Section 11.3 gradyanları hesaplamak ve parametreleri güncellemek için her seferinde bir geçiş yaparak tüm veri kümesini kullanır. Tersine Section 11.4 ilerleme kaydetmek için bir seferde bir gözlem işler. Her birinin kendi dezavantajları vardır. Gradyan inişi, özellikle veriler çok benzer olduğunda *veri verimli* değildir. Rasgele gradyan inişi, işlemciler ve GPU'lar vektörleştirmenin tam gücünden yararlanamayacağından, bilhassa *hesaplama açısından verimli* değildir. Bu, mutlu bir ortam olabileceğini gösteriyor ve aslında, şimdiye kadar tartıştığımız örneklerde bunu kullanıyorduk.

11.5.1 Vektörleştirme ve Önbellekler

Minigruplar kullanma kararının merkezinde hesaplama verimliliği vardır. Bu, en kolay şekilde birden çok GPU ve birden çok sunucuya paralelleştirme düşünüldüğünde anlaşılır. Bu durumda, her GPU'ya en az bir imge göndermemiz gerekiyor. 16 sunucu ve sunucu başına 8 GPU ile zaten 128'lik minigrup boyutuna ulaşıyoruz.

Tek GPU'lar ve hatta CPU'lar söz konusu olduğunda işler biraz daha inceliktir. Bu cihazların birden çok bellek türü, genellikle birden fazla işlem birimi türü ve aralarında farklı bant genişliği kısıtlamaları vardır. Örneğin, bir CPU'da az sayıda yazmaç (register) ve daha sonra L1, L2 ve hatta bazı durumlarda L3 önbellek (farklı işlemci çekirdekleri arasında paylaşılır) vardır. Bu önbellekler boyut ve gecikme süresini artırmaktadır (ve aynı zamanda bant genişliğini azaltmaktadır). Diyebiliriz ki, aslında işlemci ana bellek arayüzünün sağlayabildiğinden çok daha fazla işlem gerçekleştirebilir.

- 16 çekirdeğe ve AVX-512 vektörleştirmesine sahip 2 GHz CPU, saniyede $2 \cdot 10^9 \cdot 16 \cdot 32 = 10^{12}$ bayta kadar işleyebilir. GPU'ların kapasitesi bu sayının 100 katını kolayca aşar. Öte yandan, orta düzey bir sunucu işlemcisi 100 GB/s'den fazla bant genişliğine sahip olmayabilir, yani işlemcinin beslenmesini sağlamak için gerekenlerin onda birinden azı olabilir. İşleri daha da kötüleştirmek adına, tüm bellek erişimi eşit oluşturulmaz: Öncelikle, bellek arabirimleri

[132] https://discuss.d2l.ai/t/497

genellikle 64 bittir ya da daha geniştir (örneğin, GPU'larda 384 bite kadar), bu nedenle tek bir bayt okumak çok daha geniş bir erişim maliyetini doğurur.

- İlk erişim için önemli bir maliyet varken sıralı erişim nispeten ucuzdur (buna genellikle çoğuşma denir). Birden fazla soket, yonga ve diğer yapılara sahip olduğumuzda önbelleğe alma gibi akılda tutulması gereken çok daha fazla şey vardır. Bunun ayrıntılı bir tartışması, bu bölümün kapsamı dışındadır. Daha ayrıntılı bir tartışma için bu Wikipedia makalesi[133]'ne bakın.

Bu kısıtlamaları hafifletmenin yolu, işlemciye veri sağlamak için yeterince hızlı olan CPU önbellekleri hiyerarşisini kullanmaktır. Bu, derin öğrenmede toplu işlemenin arkasındaki itici güçtür. Konuları basit tutmak için, matris ile matris çarpımını düşünün, $\mathbf{A} = \mathbf{BC}$ diyelim. $\mathbf{A}$'yı hesaplamak için bir dizi seçeneğimiz var. Örneğin aşağıdakileri deneyebiliriz:

1. $\mathbf{A}_{ij} = \mathbf{B}_{i,:}\mathbf{C}_{:,j}^\top$'ü hesaplayabiliriz, yani nokta çarpımları vasıtasıyla eleman yönlü hesaplayabiliriz.
2. $\mathbf{A}_{:,j} = \mathbf{BC}_{:,j}^\top$'yi hesaplayabiliriz, yani, her seferinde bir sütun hesaplayabiliriz. Aynı şekilde $\mathbf{A}$'yı bir seferde bir satır, $\mathbf{A}_{i,:}$, hesaplamak da olabilir.
3. Sadece $\mathbf{A} = \mathbf{BC}$'yi hesaplayabiliriz.
4. $\mathbf{B}$ ve $\mathbf{C}$'yi daha küçük blok matrislerine parçalayıp $\mathbf{A}$'yı her seferde bir blok hesaplayabiliriz.

İlk seçeneği izlersek, $\mathbf{A}_{ij}$ öğesini hesaplamak istediğimiz her seferinde bir satır ve bir sütun vektörünü CPU'ya kopyalamalıyız. Daha da kötüsü, matris elemanlarının sıralı olarak hizalanması nedeniyle, bellekten okurken iki vektörden biri için birçok ayrık konuma erişmemiz gerekiyor. İkinci seçenek çok daha elverişlidir. İçinde B üzerinden geçiş yapmaya devam ederken sütun vektörünü $\mathbf{C}_{:,j}$'ü CPU önbelleğinde tutabiliyoruz. Bu, daha hızlı erişim ile bellek bant genişliği gereksinimini yarıya indirir. Tabii ki, seçenek 3 en çok arzu edilendir. Ne yazık ki, çoğu matris önbelleğe tamamen sığmayabilir (sonuçta tartıştığımız şey budur). Bununla birlikte, seçenek 4 pratik olarak kullanışlı bir alternatif sunar: Matrisin bloklarını önbelleğe taşıyabilir ve yerel olarak çoğaltabiliriz. Optimize edilmiş kütüphaneler bunu bizim için halledeceklerdir. Bu operasyonların pratikte ne kadar verimli olduğuna bir göz atalım.

Hesaplama verimliliğinin ötesinde, Python ve derin öğrenme çerçevesinin kendisi tarafından getirilen yük de düşündürücüdür. Python yorumlayıcısı her komutu çalıştırdığımızda MXNet motoruna, hesaplamalı çizgeye eklemesi ve zamanlama sırasında onunla ilgilenmesi gereken bir komut gönderdiğini hatırlayın. Bu tür yükler oldukça bezdirici olabilir. Kısacası, mümkün olduğunca vektörleştirme (ve matrisler) kullanılması şiddetle tavsiye edilir.

```
%matplotlib inline
import numpy as np
import torch
from torch import nn
from d2l import torch as d2l

timer = d2l.Timer()
A = torch.zeros(256, 256)
B = torch.randn(256, 256)
C = torch.randn(256, 256)
```

Eleman yönlü değer atama, $\mathbf{A}$'ya değer atamak için sırasıyla $\mathbf{B}$ ve $\mathbf{C}$'nin tüm satır ve sütunlarını yineler.

[133] https://en.wikipedia.org/wiki/Cache_hierarchy

```
# Her keresinde bir eleman olacak şekilde A = BC hesapla
timer.start()
for i in range(256):
    for j in range(256):
        A[i, j] = torch.dot(B[i, :], C[:, j])
timer.stop()
```

```
1.2039425373077393
```

Daha hızlı bir strateji sütun yönlü atama gerçekleştirmektir.

```
# Her keresinde bir sutün olacak şekilde A = BC hesapla
timer.start()
for j in range(256):
    A[:, j] = torch.mv(B, C[:, j])
timer.stop()
```

```
0.007898807525634766
```

Son olarak, en etkili yol, tüm işlemi bir blokta gerçekleştirmektir. İşlemlerin göreceli hızının ne olduğunu görelim.

```
# Bir seferde A = BC hesapla
timer.start()
A = torch.mm(B, C)
timer.stop()

# Çarpma ve toplama ayrı işlemler olsun (uygulamada kaynaşıktır)
gigaflops = [2/i for i in timer.times]
print(f'performance in Gigaflops: element {gigaflops[0]:.3f}, '
      f'column {gigaflops[1]:.3f}, full {gigaflops[2]:.3f}')
```

```
performance in Gigaflops: element 1.661, column 253.203, full 3394.823
```

11.5.2 Minigruplar

Geçmişte, parametreleri güncellemek için tek gözlemler yerine verilerin *minigrupları* okuyacağımızı düşünmeden kabul ettik. Şimdi bunun için kısa bir gerekçe veriyoruz. Tek gözlemlerin işlenmesi, oldukça pahalıdır ve altta yatan derin öğrenme çerçevesi adına önemli bir yük oluşturan birçok tek matris ile vektör (hatta vektör ile vektör) çarpımı gerçekleştirmemizi gerektirir. Bu, hem verilere uygulandığında bir ağın değerlendirilmesi (genellikle çıkarım olarak adlandırılır) hem de parametreleri güncellemek için gradyanları hesaplarken geçerlidir. Yani, bu $\mathbf{w} \leftarrow \mathbf{w} - \eta_t \mathbf{g}_t$ uyguladığımız her zaman geçerlidir:

$$\mathbf{g}_t = \partial_{\mathbf{w}} f(\mathbf{x}_t, \mathbf{w}) \tag{11.5.1}$$

Bu işlemin *hesaplama* verimliliğini bir seferde bir minigrup gözlem uygulayarak artırabiliriz. Yani, $\mathbf{g}_t$'yi tek bir gözlem yerine minigrup üzerinden gradyan ile değiştiriyoruz:

$$\mathbf{g}_t = \partial_{\mathbf{w}} \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} f(\mathbf{x}_i, \mathbf{w}) \tag{11.5.2}$$

Bunun $\mathbf{g}_t$'nin istatistiksel özelliklerine ne yaptığını görelim: Hem $\mathbf{x}_t$ hem de minigrup $\mathcal{B}_t$'nin tüm elemanları eğitim kümesinden tekdüze rastgele bir şekilde çekildiğinden, gradyanın beklentisi değişmeden kalır. Öte yandan varyans önemli ölçüde azaltılır. Minigrup gradyanı, ortalaması alınan $b := |\mathcal{B}_t|$ bağımsız gradyanlardan oluştuğu için, standart sapması $b^{-\frac{1}{2}}$ çarpanı kadar azalır. Bu, tek başına, iyi bir şeydir, çünkü güncellemelerin tam gradyan ile daha güvenilir bir şekilde hizalandığı anlamına gelir.

Safça bu, büyük bir minigrup $\mathcal{B}_t$ seçmenin evrensel olarak arzu edileceğini gösterir. Ne yazık ki, bir noktadan sonra, standart sapmadaki ek azalma, hesaplama maliyetindeki doğrusal artışa kıyasla minimumdur. Pratikte, bir GPU belleğine uyarken iyi hesaplama verimliliği sunacak kadar büyük bir minigrup seçiyoruz. Tasarrufları göstermek için koda biraz göz atalım. İçerisinde aynı matris matris çarpımını gerçekleştiriyoruz, ancak bu sefer bir seferde 64 sütunlu "minigruplar"a parçaladık.

```
timer.start()
for j in range(0, 256, 64):
    A[:, j:j+64] = torch.mm(B, C[:, j:j+64])
timer.stop()
print(f'performance in Gigaflops: block {2 / timer.times[3]:.3f}')
```

```
performance in Gigaflops: block 2141.043
```

Gördüğümüz gibi, minigrup üzerindeki hesaplama aslında tam matris kadar etkilidir. Dikkat edilmesi gereken unsur şudur: Section 7.5 içinde bir minigrup içindeki varyans miktarına büyük ölçüde bağımlı olan bir düzenlilik türü kullandık. İkincisini arttırdıkça, varyans azalır ve bununla birlikte toplu normalleşme nedeniyle gürültü aşılamanın faydası olur. Uygun şartların nasıl yeniden ölçekleneceği ve hesaplanacağı ile ilgili ayrıntılar için bkz. (Ioffe, 2017).

11.5.3 Veri Kümesini Okuma

Minigrupların verilerden nasıl verimli bir şekilde üretildiğine bir göz atalım. Aşağıda bu optimizasyon algoritmaları karşılaştırmak için farklı uçakların kanat gürültüsü[134]nü test etmek için NASA tarafından geliştirilmiş bir veri kümesi kullanıyoruz. Kolaylık olsun diye sadece ilk $1,500$ örneği kullanıyoruz. Veriler ön işleme için beyazlatılır, yani ortalamaları kaldırır ve varyansı koordinat başına 1'e yeniden ölçeklendiririz.

```
#@save
d2l.DATA_HUB['airfoil'] = (d2l.DATA_URL + 'airfoil_self_noise.dat',
                           '76e5be1548fd8222e5074cf0faae75edff8cf93f')

#@save
def get_data_ch11(batch_size=10, n=1500):
    data = np.genfromtxt(d2l.download('airfoil'),
                         dtype=np.float32, delimiter='\t')
    data = torch.from_numpy((data - data.mean(axis=0)) / data.std(axis=0))
    data_iter = d2l.load_array((data[:n, :-1], data[:n, -1]),
                               batch_size, is_train=True)
    return data_iter, data.shape[1]-1
```

[134] https://archive.ics.uci.edu/ml/datasets/Airfoil+Self-Noise

11.5.4 Sıfırdan Uygulama

Section 3.2 içindeki minigrup rasgele gradyan inişi uygulamasını hatırlayın. Aşağıda biraz daha genel bir uygulama sağlıyoruz. Kolaylık sağlamak için, bu bölümde daha sonra tanıtılan diğer optimizasyon algoritmalarıyla aynı çağrı imzasına sahiptir. Özellikle, `states` durum girdisini ekliyoruz ve hiper parametreyi `hyperparams` sözlüğüne yerleştiriyoruz. Buna ek olarak, eğitim işlevindeki her minigrup örneğinin kaybını ortalayacağız, böylece optimizasyon algoritmasındaki gradyanın iş boyutuna bölünmesi gerekmez.

```
def sgd(params, states, hyperparams):
    for p in params:
        p.data.sub_(hyperparams['lr'] * p.grad)
        p.grad.data.zero_()
```

Daha sonra, bu bölümün ilerleyen bölümlerinde tanıtılan diğer optimizasyon algoritmalarının kullanımını kolaylaştırmak için genel bir eğitim işlevi uyguluyoruz. Doğrusal regresyon modelini ilkletir ve modeli minigrup rasgele gradyan inişi ve daha sonra tanıtılan diğer algoritmalarla eğitmek için kullanılabilir.

```
#@save
def train_ch11(trainer_fn, states, hyperparams, data_iter,
               feature_dim, num_epochs=2):
    # İlkleme
    w = torch.normal(mean=0.0, std=0.01, size=(feature_dim, 1),
                     requires_grad=True)
    b = torch.zeros((1), requires_grad=True)
    net, loss = lambda X: d2l.linreg(X, w, b), d2l.squared_loss
    # Eğitim
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[0, num_epochs], ylim=[0.22, 0.35])
    n, timer = 0, d2l.Timer()
    for _ in range(num_epochs):
        for X, y in data_iter:
            l = loss(net(X), y).mean()
            l.backward()
            trainer_fn([w, b], states, hyperparams)
            n += X.shape[0]
            if n % 200 == 0:
                timer.stop()
                animator.add(n/X.shape[0]/len(data_iter),
                             (d2l.evaluate_loss(net, data_iter, loss),))
                timer.start()
    print(f'loss: {animator.Y[0][-1]:.3f}, {timer.avg():.3f} sec/epoch')
    return timer.cumsum(), animator.Y[0]
```

Toplu iş gradyan inişi için optimizasyonun nasıl ilerlediğini görelim. Bu, minigrup boyutunu 1500'e (yani toplam örnek sayısına) ayarlayarak elde edilebilir. Sonuç olarak model parametreleri dönem başına yalnızca bir kez güncellenir. Çok az ilerleme var. Aslında 6 adımdan sonra ilerleme durur.

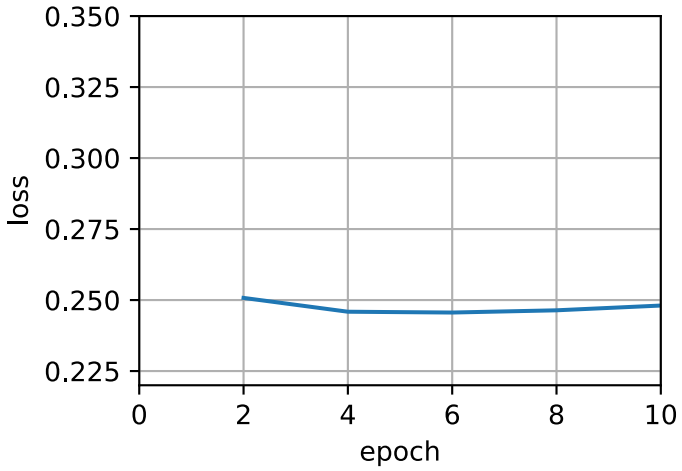
```
def train_sgd(lr, batch_size, num_epochs=2):
    data_iter, feature_dim = get_data_ch11(batch_size)
    return train_ch11(
```

(continues on next page)

(continued from previous page)

```
        sgd, None, {'lr': lr}, data_iter, feature_dim, num_epochs)

gd_res = train_sgd(1, 1500, 10)
```

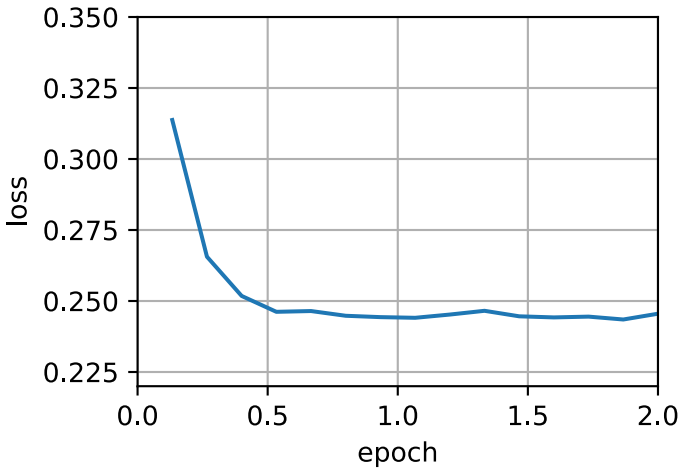
```
loss: 0.248, 0.040 sec/epoch
```



Toplu iş boyutu 1'e eşit olduğunda, optimizasyon için rasgele gradyan inişi kullanıyoruz. Uygulamanın basitliği için sabit (küçük olsa da) bir öğrenme oranı seçtik. Rasgele gradyan inişinde, örnek her işlendiğinde model parametreleri güncelleştirilir. Bizim durumumuzda bu, dönem başına 1500 güncellemedir. Gördüğümüz gibi, amaç fonksiyonunun değerindeki düşüş bir dönemden sonra yavaşlar. Her iki yöntem de bir dönem içinde 1500 örnek işlemiş olsa da, rasgele gradyan inişi bizim deneyde gradyan inişinden daha fazla zaman tüketir. Bunun nedeni, rasgele gradyan inişinin parametreleri daha sık güncellemesi ve tek tek gözlemleri birer birer işlemenin daha az verimli olmasıdır.

```
sgd_res = train_sgd(0.005, 1)
```

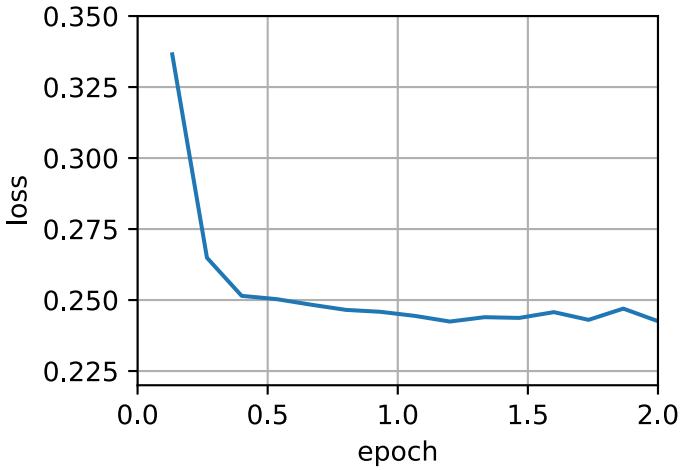
```
loss: 0.246, 0.090 sec/epoch
```



Son olarak, toplu iş boyutu 100'e eşit olduğunda, optimizasyon için minigrup rasgele gradyan inişi kullanıyoruz. Dönem başına gereken süre, rasgele gradyan inişi için gereken süreden ve toplu gradyan inişi için gereken süreden daha kısadır.

```
mini1_res = train_sgd(.4, 100)
```

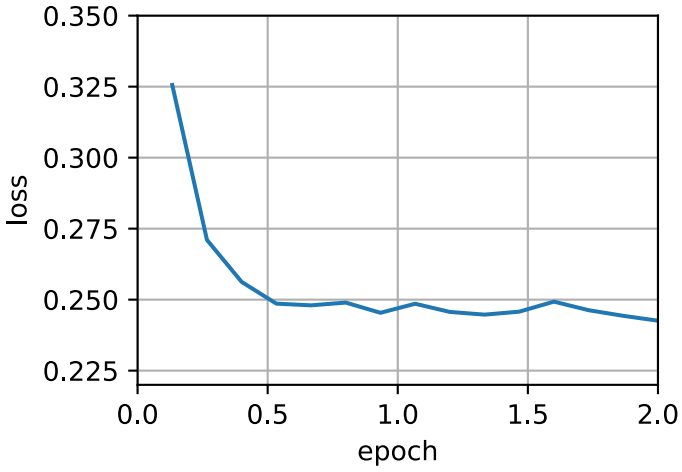
```
loss: 0.243, 0.003 sec/epoch
```



Topl iş boyutu 10'a düşürüldüğünde, her toplu iş yükünün yürütülmesi daha az verimli olduğundan, her dönemin süresi artar.

```
mini2_res = train_sgd(.05, 10)
```

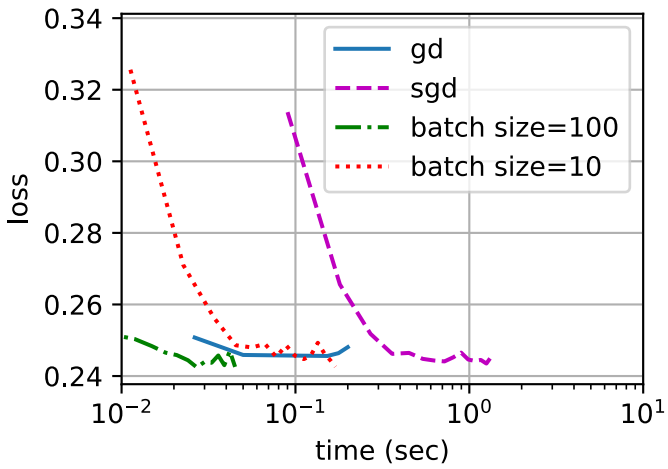
```
loss: 0.243, 0.011 sec/epoch
```



Şimdi önceki dört deney için zamanı ve kaybı karşılaştırabiliriz. Görüldüğü gibi, rasgele gradyan inişi, işlenen örneklerin sayısı açısından GD'den daha hızlı yakınsasa da, gradyanın her örnek için hesaplanması o kadar verimli olmadığından, GD'ye görece aynı kayba ulaşmak için daha fazla zaman kullanır. Minigrup rasgele gradyan inişi yakınsama hızını ve hesaplama verimliliğini

değiştirebilir. 10 'luk minigrup boyutu, rasgele gradyan inişinden daha etkilidir; 100'luk minigrup boyutu çalışma zamanı açısından GD'den daha iyi performans gösterir.

```
d2l.set_figsize([6, 3])
d2l.plot(*list(map(list, zip(gd_res, sgd_res, mini1_res, mini2_res))),
         'time (sec)', 'loss', xlim=[1e-2, 10],
         legend=['gd', 'sgd', 'batch size=100', 'batch size=10'])
d2l.plt.gca().set_xscale('log')
```



11.5.5 Özlü Uygulama

Gluon'da, optimizasyon algoritmalarını çağırmak için `Trainer` sınıfını kullanabiliriz. Bu, genel bir eğitim işlevini uygulamak için kullanılır. Bunu mevcut bölüm boyunca kullanacağız.

```
#@save
def train_concise_ch11(trainer_fn, hyperparams, data_iter, num_epochs=4):
    # İlkleme
    net = nn.Sequential(nn.Linear(5, 1))
    def init_weights(m):
        if type(m) == nn.Linear:
            torch.nn.init.normal_(m.weight, std=0.01)
    net.apply(init_weights)

    optimizer = trainer_fn(net.parameters(), **hyperparams)
    loss = nn.MSELoss(reduction='none')
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[0, num_epochs], ylim=[0.22, 0.35])
    n, timer = 0, d2l.Timer()
    for _ in range(num_epochs):
        for X, y in data_iter:
            optimizer.zero_grad()
            out = net(X)
            y = y.reshape(out.shape)
            l = loss(out, y)
            l.mean().backward()
            optimizer.step()
            n += X.shape[0]
```

(continues on next page)

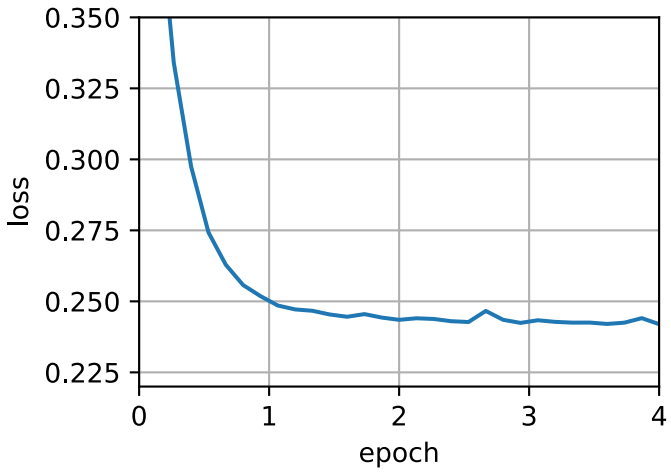
(continued from previous page)

```
            if n % 200 == 0:
                timer.stop()
                # `MSELoss` computes squared error without the 1/2 factor
                animator.add(n/X.shape[0]/len(data_iter),
                             (d2l.evaluate_loss(net, data_iter, loss) / 2,))
                timer.start()
    print(f'loss: {animator.Y[0][-1]:.3f}, {timer.avg():.3f} sec/epoch')
```

Son deneyi tekrarlarken Gluon'u kullanmak aynı davranışı gösterir.

```
data_iter, _ = get_data_ch11(10)
trainer = torch.optim.SGD
train_concise_ch11(trainer, {'lr': 0.01}, data_iter)
```

```
loss: 0.242, 0.014 sec/epoch
```



11.5.6 Özet

- Vektorleştirme, derin öğrenme çerçevesinden kaynaklanan azaltılmış ek yükü ve CPU ile GPU'larda daha iyi bellek yerelliği ve önbelleğe alma nedeniyle kodu daha verimli hale getirir.
- Rasgele gradyan inişinden kaynaklanan istatistiksel verimlilik ile aynı anda büyük veri yığınlarının işlenmesinden kaynaklanan hesaplama verimliliği arasında bir ödün verme vardır.
- Minigrup rasgele gradyan inişi her iki dünyanın en iyisini sunar: Hesaplama ve istatistiksel verimlilik.
- Minigrup rasgele gradyan inişinde, eğitim verilerinin rastgele bir yer değiştirmesi ile elde edilen veri yığınlarını işleriz (yani, her gözlem, rastgele sırada da olsa, her dönemde sadece bir kez işlenir).
- Eğitim sırasında öğrenme oranlarının sönümlenmesi tavsiye edilir.
- Genel olarak, minigrup rasgele gradyan inişi, saat süresi açısından ölçüldüğünde, daha küçük bir riske yakınsama için rasgele gradyan inişi ve gradyan inişinden daha hızlıdır.

11.5.7 Alıştırmalar

1. Toplu iş boyutunu ve öğrenme oranını değiştirin ve amaç fonksiyonun değerini ve her dönemde tüketilen süreye ilişkin düşüş oranını gözlemleyin.
2. MXNet belgelerini okuyun ve `Trainer` sınıfı `set_learning_rate` işlevini kullanarak minigrup rasgele gradyan inişinin öğrenme hızını her dönemden sonraki önceki değerinin 1/10'una düşürün.
3. Minigrup rasgele gradyan inişini, eğitim kümesinden *değiştirmeli örneklemler* kullanan bir sürümle karşılaştırın. Ne olur?
4. Kötü bir cin, size haber vermeden veri kümenizi çoğaltıyor (yani, her gözlem iki kez gerçekleşir ve veri kümeniz orijinal boyutunun iki katına çıkar, ancak size söylenmedi). Rasgele gradyan inişinin, minigrup rasgele gradyan inişinin ve gradyan inişinin davranışı nasıl değişir?

Tartışmalar[135]

11.6 Momentum

Section 11.4 içinde, rasgele gradyan inişini gerçekleştirirken, yani, gradyanın yalnızca gürültülü bir sürümünün mevcut olduğu optimizasyonu gerçekleştirirken neler olduğunu inceledik. Özellikle, gürültülü gradyanlar için gürültü karşısında öğrenme oranını seçmek söz konusu olduğunda aşırı temkinli olmamız gerektiğini fark ettik. Eğer çok hızlı azaltırsak yakınsama durur. Eğer çok hoşgörülü olursak, gürültü bizi eniyi cevaptan uzaklaştırmaya devam ettiğinden, yeterince iyi bir çözüme yaklaşmayı başaramayız.

11.6.1 Temel Bilgiler

Bu bölümde, özellikle uygulamada yaygın olan belirli optimizasyon problemleri türleri için daha etkili optimizasyon algoritmaları araştıracağız.

Sızdıran Ortalamalar

Önceki bölümde, hesaplamayı hızlandırmak için bir araç olarak minigrup SGD tartışmamızı gördünüz. Ayrıca, gradyanların ortalama varyans miktarını azalttığı güzel yan etkisi de vardı. Minigrup rasgele gradyan inişi şu şekilde hesaplanabilir:

$$\mathbf{g}_{t,t-1} = \partial_{\mathbf{w}} \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} f(\mathbf{x}_i, \mathbf{w}_{t-1}) = \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} \mathbf{h}_{i,t-1}. \tag{11.6.1}$$

Notasyonu basit tutmak için, $t-1$ zamanında güncellenen ağırlıkları kullanarak i örneğinin rasgele gradyan inişi olarak $\mathbf{h}_{i,t-1} = \partial_{\mathbf{w}} f(\mathbf{x}_i, \mathbf{w}_{t-1})$ kullandık. Bir minigrup üzerindeki gradyanların ortalamasının ötesinde bile varyans azaltmanın etkisinden faydalanabilsek güzel olurdu. Bu görevi yerine getirmek için bir seçenek gradyan hesaplamayı "sızıntılı ortalama" ile değiştirmektir:

$$\mathbf{v}_t = \beta \mathbf{v}_{t-1} + \mathbf{g}_{t,t-1} \tag{11.6.2}$$

[135] https://discuss.d2l.ai/t/1068

bazı $\beta \in (0, 1)$ için geçerlidir. Bu, anlık gradyanı birden çok *geçmiş* gradyan üzerinden ortalama alınmış bir gradyan ile etkili bir şekilde değiştirir. $\mathbf{v}$ *momentum* olarak adlandırılır. Amaç fonksiyonu yokuştan aşağı yuvarlanan ağır bir topun geçmiş kuvvetler üzerinde integral alması benzer şekilde geçmiş gradyanları biriktirir. Daha ayrıntılı olarak neler olup bittiğini görmek için $\mathbf{v}_t$'i özyinelemeli olarak aşağıdaki gibi açalım:

$$\mathbf{v}_t = \beta^2\mathbf{v}_{t-2} + \beta\mathbf{g}_{t-1,t-2} + \mathbf{g}_{t,t-1} = \ldots, = \sum_{\tau=0}^{t-1} \beta^\tau \mathbf{g}_{t-\tau,t-\tau-1}. \tag{11.6.3}$$

Büyük β, uzun menzilli ortalamaya karşılık gelirken, küçük β ise gradyan yöntemine göre yalnızca hafif bir düzeltme anlamına gelir. Yeni gradyan değişimi artık belirli bir örnekte en dik iniş yönünü değil, geçmiş gradyanların ağırlıklı ortalamasını işaret ediyor. Bu, üzerinde gradyanları hesaplama maliyeti olmadan bir toplu iş üzerinde ortalamanın faydalarının çoğunu gerçekleştirmemize olanak tanır. Bu ortalama yordamını daha sonra daha ayrıntılı olarak tekrar gözden geçireceğiz.

Yukarıdaki akıl yürütme, şimdi momentumlu gradyanlar gibi *hızlandırılmış* gradyan yöntemleri olarak bilinen şeyin temelini oluşturmuştur. Optimizasyon probleminin kötü koşullu olduğu durumlarda (yani ilerlemenin diğerlerinden çok daha yavaş olduğu, dar bir kanyona benzeyen bazı yönlerin olduğu yerlerde) çok daha etkili olmanın ek avantajından yararlanırlar. Ayrıca, daha kararlı iniş yönleri elde etmek için sonraki gradyanlar üzerinde ortalama yapmamıza izin verirlar. Gerçekten de, gürültüsüz dışbükey problemler için bile ivmenin yönü, momentumun neden çalıştığı ve neden bu kadar iyi çalıştığının temel nedenlerinden biridir.

Beklendiği gibi, etkinliği nedeniyle ivme derin öğrenme ve ötesinde optimizasyonda iyi çalışılmış bir konudur. Örneğin, ayrıntılı bir analiz ve etkileşimli animasyon için güzel açıklayıcı makale[136]'ye (Goh, 2017) bakın. Bu (Polyak, 1964) tarafından önerilmiştir. (Nesterov, 2018) dışbükey optimizasyon bağlamında ayrıntılı bir teorik tartışma sunar. Derin öğrenmede momentumun uzun zamandır faydalı olduğu bilinmektedir. Ayrıntılar için örneğin (Sutskever *et al.*, 2013) çalışmasındaki tartışmalarına bakın.

Kötü Koşullu Bir Problem

Momentum yönteminin geometrik özelliklerini daha iyi anlamak için, önemli ölçüde daha az hoş bir amaç fonksiyonu ile de olsa, gradyan inişini tekrar gözden geçiriyoruz. Section 11.3 içinde $f(\mathbf{x}) = x_1^2 + 2x_2^2$'i, yani orta miktarda çarpık elipsoid bir amaç kullandığımızı hatırlayın. Bu işlevi x_1 yönünde uzatarak daha da bozuyoruz

$$f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2. \tag{11.6.4}$$

Daha önce olduğu gibi f $(0, 0)$'da minimum seviyeye sahiptir. Bu fonksiyon x_1 yönünde *çok* düzdür. Bu yeni işlevde daha önce olduğu gibi gradyan inişi gerçekleştirdiğimizde ne olacağını görelim. Öğrenme oranını 0.4 seçiyoruz.

```
%matplotlib inline
import torch
from d2l import torch as d2l

eta = 0.4
def f_2d(x1, x2):
```

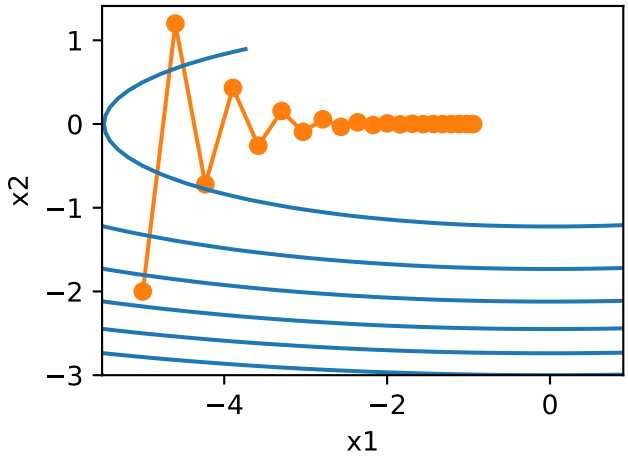
(continues on next page)

[136] https://distill.pub/2017/momentum/

(continued from previous page)

```
    return 0.1 * x1 ** 2 + 2 * x2 ** 2
def gd_2d(x1, x2, s1, s2):
    return (x1 - eta * 0.2 * x1, x2 - eta * 4 * x2, 0, 0)

d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))
```

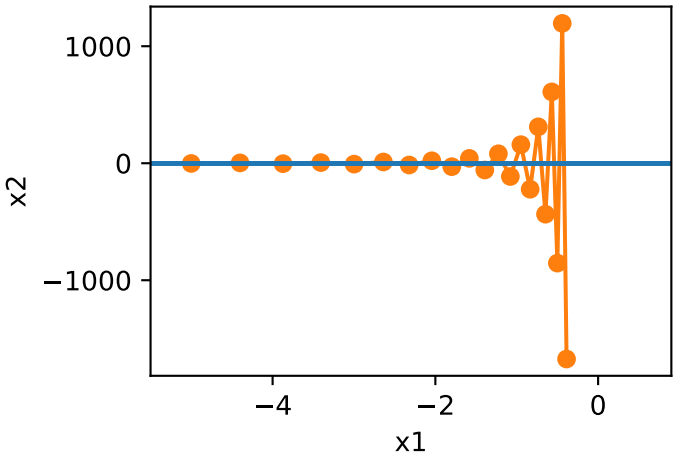
```
epoch 20, x1: -0.943467, x2: -0.000073
/home/d2l-worker/miniconda3/envs/d2l-tr-release-0/lib/python3.9/site-packages/torch/
↪functional.py:478: UserWarning: torch.meshgrid: in an upcoming release, it will be␣
↪required to pass the indexing argument. (Triggered internally at  ../aten/src/ATen/native/
↪TensorShape.cpp:2895.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
```



Yapı gereği, x_2 yönündeki gradyan *çok daha* yüksek ve yatay x_1 yönünden çok daha hızlı değişiyor. Böylece iki istenmeyen seçenek arasında sıkışmış olduk: Küçük bir öğrenme oranı seçersek, çözümün x_2 yönünde ıraksamamasını sağlarız, ancak x_1 yönünde yavaş yakınsama ile eyerleniriz. Tersine, büyük bir öğrenme oranı ile x_1 yönünde hızla ilerliyoruz ancak x_2'te ıraksıyoruz. Aşağıdaki örnek, 0.4'ten 0.6'ya kadar öğrenme hızında hafif bir artıştan sonra bile neler olduğunu göstermektedir. x_1 yönündeki yakınsama gelişir ancak genel çözüm kalitesi çok daha kötüdür.

```
eta = 0.6
d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))
```

```
epoch 20, x1: -0.387814, x2: -1673.365109
```



Momentum Yöntemi

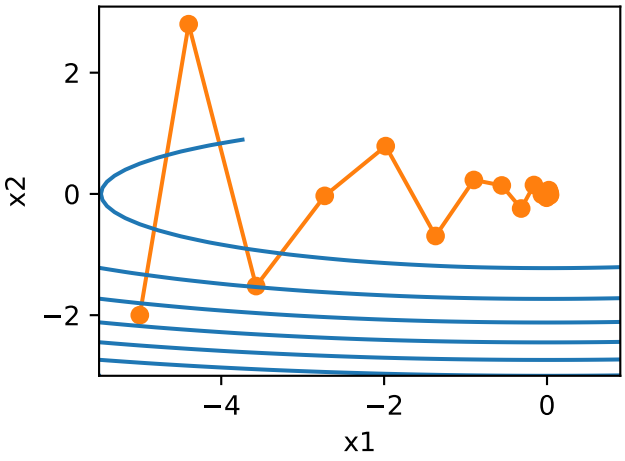
Momentum yöntemi, yukarıda açıklanan gradyan inişi problemini çözmemizi sağlar. Yukarıdaki optimizasyon izine baktığımızda geçmişteki gradyanların ortalama işe yarayacağını düşünebiliriz. Sonuçta, x_1 yönünde bu, iyi hizalanmış gradyanları bir araya getirecek ve böylece her adımda kapladığımız mesafeyi artıracaktır. Tersine, gradyanların salınım yaptığı x_2 yönünde, bir toplam gradyan birbirini iptal eden salınımlar nedeniyle adım boyutunu azaltacaktır. $\mathbf{g}_t$ gradyan yerine $\mathbf{v}_t$ kullanarak aşağıdaki güncelleştirme denklemlerini verir:

$$\begin{aligned}\mathbf{v}_t &\leftarrow \beta \mathbf{v}_{t-1} + \mathbf{g}_{t,t-1},\\ \mathbf{x}_t &\leftarrow \mathbf{x}_{t-1} - \eta_t \mathbf{v}_t.\end{aligned} \tag{11.6.5}$$

$\beta = 0$ için düzenli gradyan inişini kurtardığımızı unutmayın. Matematiksel özellikleri derinlemesine incelemeden önce, algoritmanın pratikte nasıl davrandığına hızlı bir göz atalım.

```
def momentum_2d(x1, x2, v1, v2):
    v1 = beta * v1 + 0.2 * x1
    v2 = beta * v2 + 4 * x2
    return x1 - eta * v1, x2 - eta * v2, v1, v2

eta, beta = 0.6, 0.5
d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
```

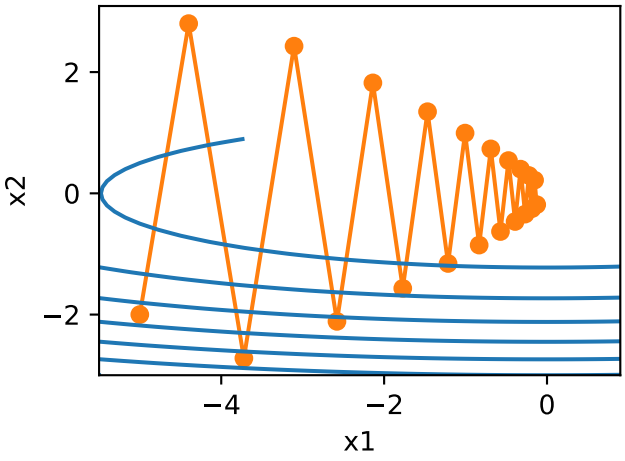
```
epoch 20, x1: 0.007188, x2: 0.002553
```



Gördüğümüz gibi, daha önce kullandığımız aynı öğrenme oranıyla bile, momentum hala iyi bir şekilde yakınsıyor. Momentum parametresini azalttığımızda neler olacağını görelim. $\beta = 0.25$'e kadar yarıya indirmek, neredeyse hiç yakınsamayan bir yörüngeye yol açar. Bununla birlikte, momentumsuz halinden çok daha iyidir (çözüm ıraksadığı zaman).

```
eta, beta = 0.6, 0.25
d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
```

```
epoch 20, x1: -0.126340, x2: -0.186632
```

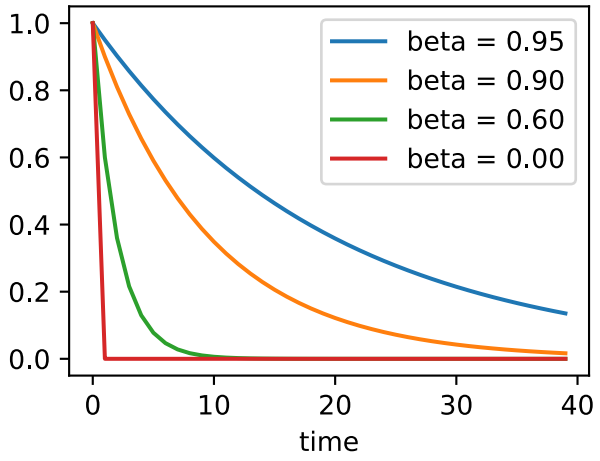


Rasgele gradyan inişi, özellikle minigrup rasgele gradyan inişi ile momentumu birleştirebileceğinizi unutmayın. Tek değişiklik, bu durumda $\mathbf{g}_{t,t-1}$ gradyanlarını $\mathbf{g}_t$ ile değiştirmemizdir. Son olarak, kolaylık sağlamak için $\mathbf{v}_0 = 0$'ı $t = 0$'da ilklettik. Sızdıran ortalamaların güncellemelere ne yaptığına bakalım.

Etkili Örneklem Ağırlığı

Hatırlayalım $\mathbf{v}_t = \sum_{\tau=0}^{t-1} \beta^\tau \mathbf{g}_{t-\tau,t-\tau-1}$. Limitte terimler $\sum_{\tau=0}^{\infty} \beta^\tau = \frac{1}{1-\beta}$'ye toplanır. Başka bir deyişle, gradyan inişte veya rasgele gradyan inişte η boyutunda bir adım atmak yerine, $\frac{\eta}{1-\beta}$ boyutunda bir adım atıyoruz ve aynı zamanda potansiyel olarak çok daha iyi davranan bir iniş yönüyle uğraşıyoruz. Bunlar ikisi bir arada faydalardır. β'yı farklı seçenekler için ağırlıklandırmanın nasıl davrandığını göstermek için aşağıdaki diyagramı göz önünde bulundurun.

```
d2l.set_figsize()
betas = [0.95, 0.9, 0.6, 0]
for beta in betas:
    x = torch.arange(40).detach().numpy()
    d2l.plt.plot(x, beta ** x, label=f'beta = {beta:.2f}')
d2l.plt.xlabel('time')
d2l.plt.legend();
```



11.6.2 Pratik Deneyler

Momentumun pratikte nasıl çalıştığını görelim, yani, uygun bir iyileştirici bağlamında kullanıldığında. Bunun için biraz daha ölçeklenebilir bir uygulamaya ihtiyacımız var.

Sıfırdan Uygulama

(Minigrup) rasgele gradyan inişi ile karşılaştırıldığında, momentum yönteminin bir dizi yardımcı değişken muhafaza etmesi gerekir, mesela hız. Gradyanlar (ve en iyileştirme probleminin değişkenleri) ile aynı şekle sahiptir. Aşağıdaki uygulamada bu değişkenleri `states` olarak adlandırıyoruz.

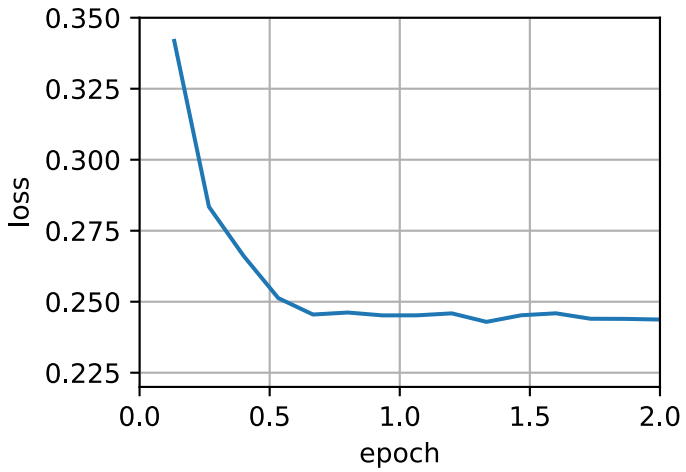
```
def init_momentum_states(feature_dim):
    v_w = torch.zeros((feature_dim, 1))
    v_b = torch.zeros(1)
    return (v_w, v_b)
```

```
def sgd_momentum(params, states, hyperparams):
    for p, v in zip(params, states):
        with torch.no_grad():
            v[:] = hyperparams['momentum'] * v + p.grad
            p[:] -= hyperparams['lr'] * v
        p.grad.data.zero_()
```

Bunun pratikte nasıl çalıştığını görelim.

```
def train_momentum(lr, momentum, num_epochs=2):
    d2l.train_ch11(sgd_momentum, init_momentum_states(feature_dim),
                   {'lr': lr, 'momentum': momentum}, data_iter,
                   feature_dim, num_epochs)

data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
train_momentum(0.02, 0.5)
```

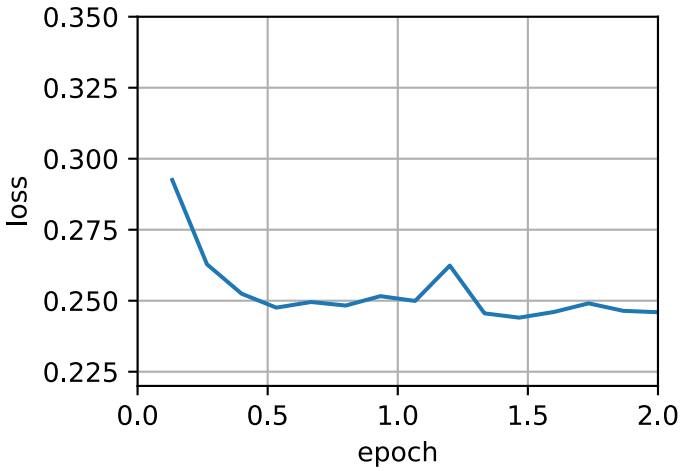
```
loss: 0.244, 0.014 sec/epoch
```



Momentum hiper parametresi `momentum`'u 0.9'a yükselttiğimizde, bu, $\frac{1}{1-0.9} = 10$ gibi önemli ölçüde daha büyük bir etkin örneklem boyutu anlamına gelir. Sorunları kontrol altında tutmak için öğrenme oranını azar azar 0.01'a indiriyoruz.

```
train_momentum(0.01, 0.9)
```

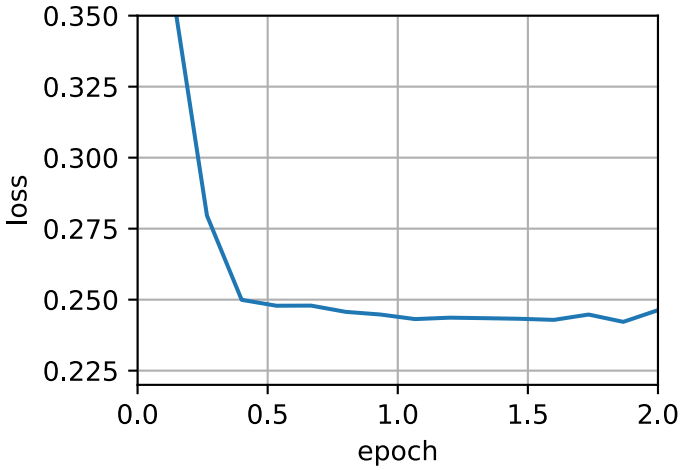
```
loss: 0.246, 0.013 sec/epoch
```



Öğrenme oranını azaltmak, pürüzsüz olmayan optimizasyon problemleriyle ilgili her türlü sorunu daha da giderir. 0.005 olarak ayarlamak iyi yakınsama özellikleri sağlar.

```
train_momentum(0.005, 0.9)
```

```
loss: 0.246, 0.014 sec/epoch
```

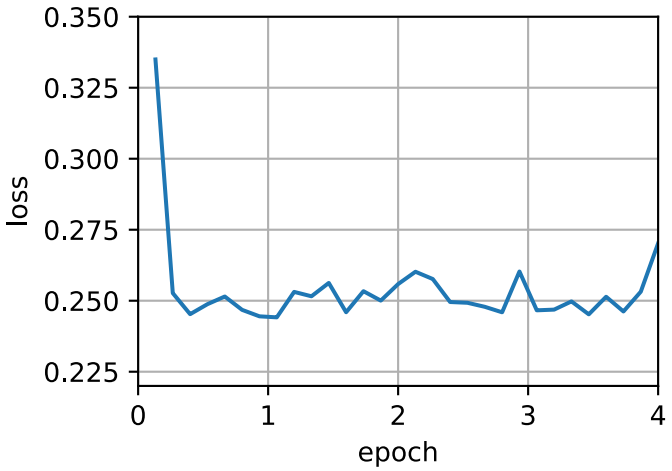


Özlü Uygulama

Standart sgd çözücüde momentum zaten yerleşik olduğundan Gluon'da yapılacak çok az şey var. Eşleşen parametrelerin ayarlanması çok benzer bir yörünge oluşturur.

```
trainer = torch.optim.SGD
d2l.train_concise_ch11(trainer, {'lr': 0.005, 'momentum': 0.9}, data_iter)
```

```
loss: 0.270, 0.013 sec/epoch
```



11.6.3 Kuramsal Analiz

Şimdiye kadar $f(x) = 0.1x_1^2 + 2x_2^2$'nin 2D örneği oldukça yapılandırılmış görünüyordu. Şimdi bunun, en azından dışbükey ikinci dereceden amaç fonksiyonlarının en aza indirilmesi durumunda karşılaşabileceği problem türlerinin açık bir temsilcisi olduğunu göreceğiz.

İkinci Dereceden Dışbükey Fonksiyonlar

Aşağıdaki işlevi düşünün:

$$h(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{x}^\top \mathbf{c} + b. \tag{11.6.6}$$

Bu genel bir ikinci dereceden fonksiyondur. Pozitif kesin matrisler $\mathbf{Q} \succ 0$ için, yani, pozitif özdeğerlere sahip matrisler için, bu minimum değer $b - \frac{1}{2}\mathbf{c}^\top \mathbf{Q}^{-1}\mathbf{c}$ ile $\mathbf{x}^* = -\mathbf{Q}^{-1}\mathbf{c}$ değerinde bir en küçük değer bulucuya sahiptir. Bu nedenle h'yi yeniden yazabiliriz

$$h(\mathbf{x}) = \frac{1}{2}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c})^\top \mathbf{Q}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c}) + b - \frac{1}{2}\mathbf{c}^\top \mathbf{Q}^{-1}\mathbf{c}. \tag{11.6.7}$$

Gradyan $\partial_{\mathbf{x}} f(\mathbf{x}) = \mathbf{Q}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c})$ ile verilir. Yani, $\mathbf{x}$ ile $\mathbf{Q}$ ile çarpılan en küçük değer bulucu arasındaki mesafe ile verilir. Sonuç olarak da momentum $\mathbf{Q}(\mathbf{x}_t - \mathbf{Q}^{-1}\mathbf{c})$ terimlerinin doğrusal bir kombinasyonudur.

$\mathbf{Q}$ pozitif kesin olduğundan $\mathbf{Q} = \mathbf{O}^\top \mathbf{\Lambda} \mathbf{O}$ üzerinden bir dikey (çevirme) matrisi $\mathbf{O}$ ve köşegen matrisi $\mathbf{\Lambda}$ için pozitif özdeğerlerine ayrıştırılabilir. Bu, çok basitleştirilmiş bir ifade elde etmek için $\mathbf{x}$'den $\mathbf{z} := \mathbf{O}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c})$'ye değişken değişikliği yapmamıza olanak tanır:

$$h(\mathbf{z}) = \frac{1}{2}\mathbf{z}^\top \mathbf{\Lambda} \mathbf{z} + b'. \tag{11.6.8}$$

Burada $b' = b - \frac{1}{2}\mathbf{c}^\top \mathbf{Q}^{-1}\mathbf{c}$. $\mathbf{O}$ sadece bir dikey matris olduğundan bu durum gradyanları anlamlı bir şekilde bozmaz. $\mathbf{z}$ ile ifade edilirse gradyan inişi aşağıdaki ifadeye dönüşür:

$$\mathbf{z}_t = \mathbf{z}_{t-1} - \mathbf{\Lambda}\mathbf{z}_{t-1} = (\mathbf{I} - \mathbf{\Lambda})\mathbf{z}_{t-1}. \tag{11.6.9}$$

Bu ifadedeki önemli gerçek gradyan inişinin farklı özuzaylar arasında *karışmadığıdır*. Yani, $\mathbf{Q}$'nun özsistemi açısından ifade edildiğinde optimizasyon problemi koordinat-yönlü bir şekilde ilerlemektedir. Bu aynı zamanda momentum için de geçerlidir.

$$\begin{aligned}\mathbf{v}_t &= \beta \mathbf{v}_{t-1} + \mathbf{\Lambda} \mathbf{z}_{t-1} \\ \mathbf{z}_t &= \mathbf{z}_{t-1} - \eta \left(\beta \mathbf{v}_{t-1} + \mathbf{\Lambda} \mathbf{z}_{t-1}\right) \\ &= (\mathbf{I} - \eta \mathbf{\Lambda}) \mathbf{z}_{t-1} - \eta \beta \mathbf{v}_{t-1}.\end{aligned} \tag{11.6.10}$$

Bunu yaparken ayrıca şu teoremi kanıtladık: Dışbükey bir ikinci derece fonksiyon için ivmeli ve ivmesiz gradyan inişi, ikinci dereceden matrisin özvektörleri yönünde koordinat yönlü optimizasyonuna ayrışır.

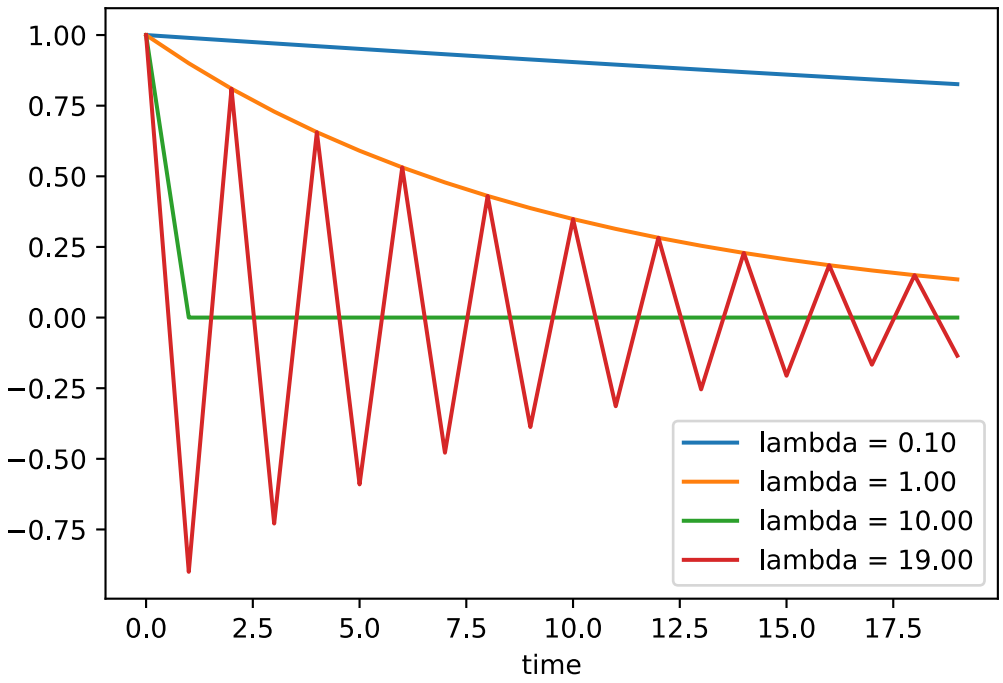
Skaler Fonksiyonlar

Yukarıdaki sonuç göz önüne alındığında, $f(x) = \frac{\lambda}{2} x^2$ işlevini en aza indirdiğimizde neler olduğunu görelim. Gradyan inişi için

$$x_{t+1} = x_t - \eta \lambda x_t = (1 - \eta \lambda) x_t. \tag{11.6.11}$$

$|1 - \eta\lambda| < 1$ olduğunda bu optimizasyon üstel bir oranda yakınsar çünkü t adımdan sonra $x_t = (1 - \eta\lambda)^t x_0$ olur. Bu, $\eta\lambda = 1$'e kadar η öğrenme oranını artırdıkça yakınsama oranının başlangıçta nasıl arttığını gösterir. Bunun ötesinde ıraksar ve $\eta\lambda > 2$ için optimizasyon problemi ıraksıyor.

```
lambdas = [0.1, 1, 10, 19]
eta = 0.1
d2l.set_figsize((6, 4))
for lam in lambdas:
    t = torch.arange(20).detach().numpy()
    d2l.plt.plot(t, (1 - eta * lam) ** t, label=f'lambda = {lam:.2f}')
d2l.plt.xlabel('time')
d2l.plt.legend();
```



Momentum durumunda yakınsaklığı analiz etmek için güncelleme denklemlerini iki skaler açısından yeniden yazarak başlıyoruz: Biri x ve diğeri momentum v için. Bu şuna yol açar:

$$\begin{bmatrix} v_{t+1} \\ x_{t+1} \end{bmatrix} = \begin{bmatrix} \beta & \lambda \\ -\eta\beta & (1-\eta\lambda) \end{bmatrix} \begin{bmatrix} v_t \\ x_t \end{bmatrix} = \mathbf{R}(\beta, \eta, \lambda) \begin{bmatrix} v_t \\ x_t \end{bmatrix}. \tag{11.6.12}$$

R'ı, yakınsama davranışını yöneten 2×2'yi göstermek için kullandık. t adımdan sonra ilk tercih $[v_0, x_0]$, $\mathbf{R}(\beta, \eta, \lambda)^t[v_0, x_0]$ olur. Bu nedenle, yakınsama hızını belirlemek **R**'nin özdeğerlerine kalmıştır. Harika bir animasyon için (Goh, 2017) Distill post[137]una ve ayrıntılı analiz için (Flammarion and Bach, 2015) bölümüne bakın. $0 < \eta\lambda < 2+2\beta$ olduğunda momentumun yakınsadığını gösterilebilir. Bu, gradyan inişi için $0 < \eta\lambda < 2$ ile karşılaştırıldığında daha geniş bir uygulanabilir parametre aralığıdır. Ayrıca, genel olarak β'nin büyük değerlerinin arzu edildiğini de göstermektedir. Daha fazla ayrıntı, makul miktarda teknik detay gerektirir ve ilgilenen okuyucunun orijinal yayınlara başvurmasını öneririz.

11.6.4 Özet

- Momentum, geçmiş gradyanlara göre sızan bir ortalama ile gradyanların yerini alır. Bu, yakınsamayı önemli ölçüde hızlandırır.
- Hem gürültüsüz gradyan inişi hem de (gürültülü) rasgele gradyan inişi için arzu edilir.
- Momentum, rasgele gradyan inişi için ortaya çıkma olasılığı çok daha yüksek olan optimizasyon sürecinin durdurulmasını önler.
- Geçmiş verilerin katlanarak azaltılması nedeniyle etkin gradyan sayısı $\frac{1}{1-\beta}$ ile verilir.
- Dışbükey ikinci derece problemler durumunda bu ayrıntılı olarak açıkça analiz edilebilir.
- Uygulama oldukça basittir ancak ek bir durum vektörü (momentum **v**) saklamamızı gerektirir.

11.6.5 Alıştırmalar

1. Momentum hiper parametrelerinin ve öğrenme oranlarının diğer kombinasyonlarını kullanın ve farklı deneysel sonuçları gözlemleyip analiz edin.
2. Birden fazla özdeğeriniz olduğu, yani $f(x) = \frac{1}{2}\sum_i \lambda_i x_i^2$, örn. $\lambda_i = 2^{-i}$ gibi bir ikinci derece polinom problem için GD ve momentum deneyin. x değerlerinin ilk $x_i = 1$ için nasıl azaldığını çizdirin.
3. $h(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top\mathbf{Q}\mathbf{x} + \mathbf{x}^\top\mathbf{c} + b$ için minimum değeri ve küçültücüyü türetin.
4. Biz momentumlu rasgele gradyan inişi gerçekleştirdiğinizde ne değişir? Momentumlu minigrup rasgele gradyan inişi kullandığımızda ne olur? Ya parametrelerle deney yaparsak ne olur?

Tartışmalar[138]

[137] https://distill.pub/2017/momentum/
[138] https://discuss.d2l.ai/t/1070

11.7 Adagrad

Seyrek olarak ortaya çıkan özniteliklerle öğrenme problemlerini göz önünde bulundurarak başlayalım.

11.7.1 Seyrek Öznitelikler ve Öğrenme Oranları

Bir dil modelini eğittiğimizi hayal edin. İyi bir doğruluk oranı elde etmek için genellikle eğitime devam ettiğimizde, çoğunlukla $\mathcal{O}(t^{-\frac{1}{2}})$ veya daha yavaş bir hızda öğrenme oranını düşürmek isteriz. Şimdi seyrek öznitelikler üzerinde bir model eğitimi düşünün, yani, sadece ender olarak ortaya çıkan öznitelikler. Bu doğal dil için yaygındır, örn. *ön şartlandırma* kelimesini görmemiz *öğrenme* kelimesini görmemizden çok daha az olasıdır. Bununla birlikte, hesaplamalı reklamcılık ve kişiselleştirilmiş işbirlikçi filtreleme gibi diğer alanlarda da yaygındır. Sonuçta, sadece az sayıda insan için ilgi çeken birçok şey vardır.

Sık görülen özniteliklerle ilişkili parametreler yalnızca bu öznitelikler ortaya çıktığında anlamlı güncellemeler alır. Azalan bir öğrenme oranı göz önüne alındığında, yaygın özniteliklerin parametrelerinin optimal değerlerine oldukça hızlı bir şekilde yakınsadığı bir duruma düşebiliriz, oysa seyrek öznitelikler için en uygun değerler belirlenmeden önce onları yeterince sık gözlemlemede yetersiz kalırız. Başka bir deyişle, öğrenme oranı ya sık görülen öznitelikler için çok yavaş ya da seyrek olanlar için çok hızlı azalır.

Bu sorunu çözmenin olası bir yolu, belirli bir özniteliği kaç kez gördüğümüzü saymak ve bunu öğrenme oranlarını ayarlamak için bir taksimetre gibi kullanmak olabilir. Yani, $\eta = \frac{\eta_0}{\sqrt{t+c}}$ formunda bir öğrenme oranı seçmek yerine $\eta_i = \frac{\eta_0}{\sqrt{s(i,t)+c}}$ kullanabiliriz. Burada $s(i,t)$ t zamana kadar gözlemlediğimiz öznitelik i için sıfır olmayanların sayısını sayar. Aslında bunun anlamlı bir ek yük olmadan uygulanması oldukça kolaydır. Bununla birlikte, oldukça seyrekliğe sahip olmadığımızda, bunun yerine sadece gradyanların genellikle çok küçük ve nadiren büyük olduğu veriye sahip olduğumuzda başarısız olur. Ne de olsa, gözlemlenen bir şeyin bir öznitelik olarak nitelendirmek veya nitelendirmemek arasındaki çizginin nerede çekileceği belirsizdir.

Adagrad, (Duchi *et al.*, 2011) tarafından, oldukça kaba olan $s(i,t)$ sayacını daha önce gözlemlenen gradyanların karelerinin toplamı ile değiştirerek bu sorunu giderir. Özellikle, öğrenme oranını ayarlamak için bir araç olarak $s(i,t+1) = s(i,t) + (\partial_i f(\mathbf{x}))^2$'yı kullanır. Bunun iki yararı vardır: Birincisi, artık bir gradyanın ne zaman yeterince büyük olduğuna karar vermemiz gerekmiyor. İkincisi, gradyanların büyüklüğü ile otomatik olarak ölçeklenir. Rutin olarak büyük gradyanlara karşılık gelen koordinatlar önemli ölçüde küçültülürken, küçük gradyanlara sahip diğerleri çok daha nazik bir muamele görür. Uygulamada bu, hesaplamalı reklamcılık ve ilgili problemler için çok etkili bir optimizasyon yöntemine yol açar. Ancak bu, en iyi ön koşullandırma bağlamında anlaşılan Adagrad'ın doğasında bulunan bazı ek faydaları gizler.

11.7.2 Ön Şartlandırma

Dışbükey optimizasyon problemleri algoritmaların özelliklerini analiz etmek için iyidir. Sonuçta, dışbükey olmayan sorunların çoğunda anlamlı teorik garantiler elde etmek zordur, ancak *sezgi* ve *kavrama* genellikle buraya da taşınır. $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{c}^\top \mathbf{x} + b$'yi en aza indirme sorununa bakalım.

Section 11.6 içinde gördüğümüz gibi, her koordinatın ayrı ayrı çözülebileceği çok basitleştirilmiş bir soruna varmak için bu sorunu özayrışma $\mathbf{Q} = \mathbf{U}^\top \mathbf{\Lambda} \mathbf{U}$ açısından yeniden yazmak mümkündür:

$$f(\mathbf{x}) = \bar{f}(\bar{\mathbf{x}}) = \frac{1}{2}\bar{\mathbf{x}}^\top \mathbf{\Lambda} \bar{\mathbf{x}} + \bar{\mathbf{c}}^\top \bar{\mathbf{x}} + b. \tag{11.7.1}$$

Burada $\mathbf{x} = \mathbf{U}\mathbf{x}$ ve dolayısıyla $\mathbf{c} = \mathbf{U}\mathbf{c}$'yi kullandık. Değiştirilen problemde, onun küçültücüsü $\bar{\mathbf{x}} = -\mathbf{\Lambda}^{-1}\bar{\mathbf{c}}$ ve minimum değeri $-\frac{1}{2}\bar{\mathbf{c}}^\top \mathbf{\Lambda}^{-1}\bar{\mathbf{c}} + b$ olarak bulunur. $\mathbf{\Lambda}$, $\mathbf{Q}$'nun özdeğerlerini içeren köşegen bir matris olduğundan bu işlem çok daha kolaydır.

$\mathbf{c}$'yi biraz dürtersek, f küçültücüsünde yalnızca küçük değişiklikler bulmayı umarız. Ne yazık ki durum böyle değil. $\mathbf{c}$'deki küçük değişiklikler $\bar{\mathbf{c}}$'de eşit derecede küçük değişikliklere yol açarken, f'nin (ve $\bar{f}$'nin sırasıyla) küçültücüsü için durum böyle değildir. Özdeğerler $\mathbf{\Lambda}_i$ büyük olduğunda $\bar{x}_i$'te ve minimum $\bar{f}$'de sadece küçük değişiklikler göreceğiz. Tersine, $\bar{x}_i$'teki küçük $\mathbf{\Lambda}_i$ değişiklikleri dramatik olabilir. En büyük ve en küçük özdeğer arasındaki oran, bir optimizasyon probleminin sağlamlık sayısı (condition number) olarak adlandırılır.

$$\kappa = \frac{\mathbf{\Lambda}_1}{\mathbf{\Lambda}_d}. \tag{11.7.2}$$

Sağlamlık sayısı κ büyükse, optimizasyon problemini doğru bir şekilde çözmek zordur. Geniş bir dinamik değer aralığını doğru bir şekilde elde etme konusunda dikkatli olduğumuzdan emin olmamız gerekir. Analizlerimiz bariz, ama biraz naif bir soruya yol açıyor: Problemi, tüm özdeğerler 1 olacak şekilde uzayı bozarak sorunu basitçe "düzeltemez miyiz?" Teorik olarak bu oldukça kolaydır: Problemi $\mathbf{x}$'den $\mathbf{z} := \mathbf{\Lambda}^{\frac{1}{2}}\mathbf{U}\mathbf{x}$'te bir taneye yeniden ölçeklendirmek için $\mathbf{Q}$'nin özdeğerlerine ve özvektörlerine ihtiyacımız var. Yeni koordinat sisteminde $\mathbf{x}^\top \mathbf{Q}\mathbf{x}$, $\|\mathbf{z}\|^2$'ye basitleştirilebilir. Ne yazık ki, bu oldukça pratik olmayan bir öneri. Özdeğerleri ve özvektörleri hesaplama genel olarak gerçek problemi çözmekten *çok daha* pahalıdır.

Özdeğerlerin tam hesaplanması pahalı olsa da, onları tahmin etmek ve hatta biraz yaklaşık hesaplama yapmak, hiçbir şey yapmamaktan çok daha iyi olabilir. Özellikle, $\mathbf{Q}$'nun köşegen girdilerini kullanabilir ve buna göre yeniden ölçekleyebiliriz. Bu, özdeğerlerin hesaplanmasından *çok daha* ucuzdur.

$$\tilde{\mathbf{Q}} = \mathrm{diag}^{-\frac{1}{2}}(\mathbf{Q})\mathbf{Q}\mathrm{diag}^{-\frac{1}{2}}(\mathbf{Q}). \tag{11.7.3}$$

Bu durumda $\tilde{\mathbf{Q}}_{ij} = \mathbf{Q}_{ij}/\sqrt{\mathbf{Q}_{ii}\mathbf{Q}_{jj}}$ ve özellikle tüm i için $\tilde{\mathbf{Q}}_{ii} = 1$ olur. Çoğu durumda bu sağlamlık sayısını önemli ölçüde basitleştirir. Örneğin, daha önce tartıştığımız vakalarda, problem eksen hizalandığından eldeki sorunu tamamen ortadan kaldıracaktır.

Ne yazık ki başka bir sorunla karşı karşıyayız: Derin öğrenmede genellikle amaç fonksiyonun ikinci türevine bile erişimimiz yok: $\mathbf{x} \in \mathbb{R}^d$ için bile bir minigrup üzerinde ikinci türev $\mathcal{O}(d^2)$'lik alan ve hesaplama için çalışma gerektirebilir, bu yüzden pratikte imkansız hale gelir. Adagrad'ın dahiyane fikri, Hessian'ın bu anlaşılması zor köşegeni için hem hesaplanması nispeten ucuz hem de etkili olan bir vekil kullanmasıdır - gradyanın kendisinin büyüklüğü.

Bunun neden çalıştığını görmek için $\bar{f}(\bar{\mathbf{x}})$'e bakalım. Elimizde bu var:

$$\partial_{\bar{\mathbf{x}}} \bar{f}(\bar{\mathbf{x}}) = \mathbf{\Lambda}\bar{\mathbf{x}} + \bar{\mathbf{c}} = \mathbf{\Lambda}\left(\bar{\mathbf{x}} - \bar{\mathbf{x}}_0\right), \tag{11.7.4}$$

Burada $\bar{\mathbf{x}}_0$, $\bar{f}$'nin küçültücüsüdür. Bu nedenle gradyanın büyüklüğü hem $\mathbf{\Lambda}$'ya hem de eniyi değere olan mesafeye bağlıdır. $\bar{\mathbf{x}} - \bar{\mathbf{x}}_0$ değişmeseydi, gereken tek şey bu olurdu. Sonuçta, bu durumda $\partial_{\bar{\mathbf{x}}} \bar{f}(\bar{\mathbf{x}})$ gradyanın büyüklüğü yeterlidir. AdaGrad bir rasgele gradyan inişi algoritması olduğundan, eniyi düzeyde bile sıfır olmayan varyansı olan gradyanları göreceğiz. Sonuç olarak, gradyanların varyansını Hessian ölçeği için ucuz bir vekil olarak güvenle kullanabiliriz. Kapsamlı bir analiz, bu bölümün kapsamının dışındadır (olsaydı birkaç sayfa olacaktı). Ayrıntılar için okuyucuyu (Duchi *et al.*, 2011)'e yönlendiriyoruz.

11.7.3 Algoritma

Yukarıdaki tartışmayı formüle dökelim. Geçmiş gradyan varyansı aşağıdaki gibi biriktirmek için $\mathbf{s}_t$ değişkenini kullanıyoruz.

$$\begin{aligned} \mathbf{g}_t &= \partial_{\mathbf{w}} l(y_t, f(\mathbf{x}_t, \mathbf{w})), \\ \mathbf{s}_t &= \mathbf{s}_{t-1} + \mathbf{g}_t^2, \\ \mathbf{w}_t &= \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \cdot \mathbf{g}_t. \end{aligned} \tag{11.7.5}$$

Burada işlem koordinat yönlü olarak akıllıca uygulanır. Yani, $\mathbf{v}^2$'nin v_i^2 girdileri vardır. Aynı şekilde $\frac{1}{\sqrt{v}}$'nin $\frac{1}{\sqrt{v_i}}$ girdileri ve $\mathbf{u} \cdot \mathbf{v}$'nin $u_i v_i$ girdileri vardır. Daha önce olduğu gibi η öğrenme oranıdır ve ϵ 0 ile bölmememizi sağlayan bir katkı sabitidir. Son olarak, $\mathbf{s}_0 = \mathbf{0}$'ı ilkleriz.

Tıpkı momentum durumunda olduğu gibi, koordinat başına bireysel bir öğrenme oranına izin vermek için yardımcı bir değişkeni takip etmeliyiz. Bu, ana maliyet tipik olarak $l(y_t, f(\mathbf{x}_t, \mathbf{w}))$ ve türevini hesaplamak olduğundan, SGD'ye göre Adagrad'ın maliyetini önemli ölçüde artırmaz.

$\mathbf{s}_t$'te kare gradyanların biriktirilmesinin $\mathbf{s}_t$'in esas olarak doğrusal hızda büyüdüğü anlamına geldiğini unutmayın (gradyanlar başlangıçta azaldığından, pratikte doğrusal olandan biraz daha yavaş). Bu, koordinat tabanında ayarlanmış olsa da $\mathcal{O}(t^{-\frac{1}{2}})$ öğrenme hızına yol açar. Dışbükey problemler için bu mükemmel bir şekilde yeterlidir. Derin öğrenmede, öğrenme oranını daha yavaş düşürmek isteyebiliriz. Bu, sonraki bölümlerde tartışacağımız bir dizi Adagrad varyantına yol açtı. Şimdilik, ikinci deredecen polinom dışbükey bir problemde nasıl davrandığını görelim. Daha önce olduğu gibi aynı problemi kullanıyoruz:

$$f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2. \tag{11.7.6}$$

Adagrad'ı daha önceki aynı öğrenme oranını kullanarak uygulayacağız, yani $\eta = 0.4$. Gördüğümüz gibi bağımsız değişkenin yinelemeli yörüngesi daha pürüzsüzdür. Bununla birlikte, s_t'nin biriktirici etkisi nedeniyle, öğrenme oranı sürekli olarak söner, bu nedenle bağımsız değişken yinelemenin sonraki aşamalarında çok fazla hareket etmez.

```
%matplotlib inline
import math
import torch
from d2l import torch as d2l
```

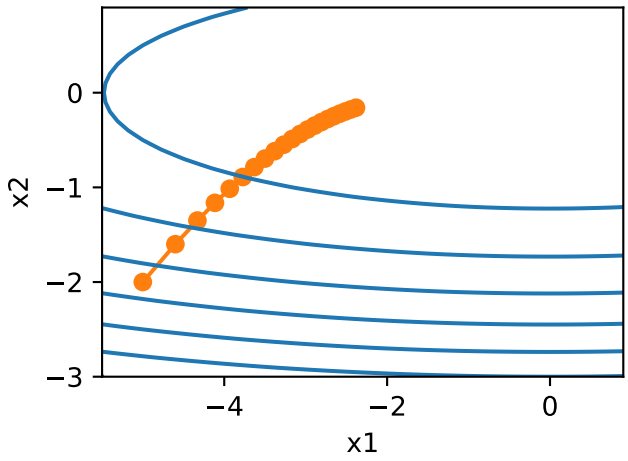
```
def adagrad_2d(x1, x2, s1, s2):
    eps = 1e-6
    g1, g2 = 0.2 * x1, 4 * x2
    s1 += g1 ** 2
    s2 += g2 ** 2
```

(continues on next page)

(continued from previous page)

```
    x1 -= eta / math.sqrt(s1 + eps) * g1
    x2 -= eta / math.sqrt(s2 + eps) * g2
    return x1, x2, s1, s2

def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2

eta = 0.4
d2l.show_trace_2d(f_2d, d2l.train_2d(adagrad_2d))
```

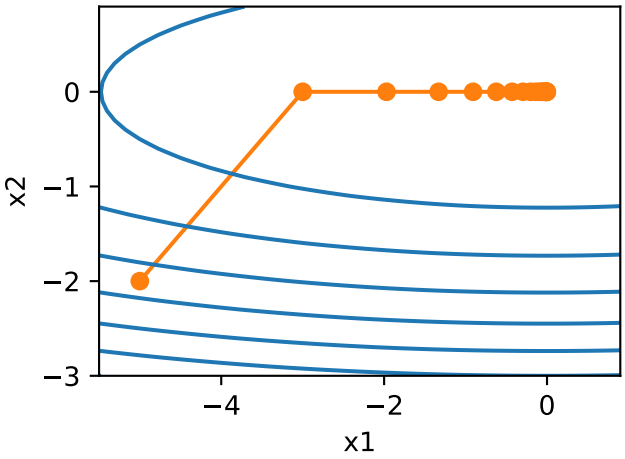
```
epoch 20, x1: -2.382563, x2: -0.158591
/home/d2l-worker/miniconda3/envs/d2l-tr-release-0/lib/python3.9/site-packages/torch/
↪functional.py:478: UserWarning: torch.meshgrid: in an upcoming release, it will be␣
↪required to pass the indexing argument. (Triggered internally at  ../aten/src/ATen/native/
↪TensorShape.cpp:2895.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
```



Öğrenme oranını 2'ye yükselttikçe çok daha iyi davranışlar görüyoruz. Bu durum, öğrenme oranındaki düşüşün gürültüsüz durumda bile oldukça saldırgan olabileceğini gösteriyor ve parametrelerin uygun şekilde yakınsamasını sağlamalıyız.

```
eta = 2
d2l.show_trace_2d(f_2d, d2l.train_2d(adagrad_2d))
```

```
epoch 20, x1: -0.002295, x2: -0.000000
```



11.7.4 Sıfırdan Uygulama

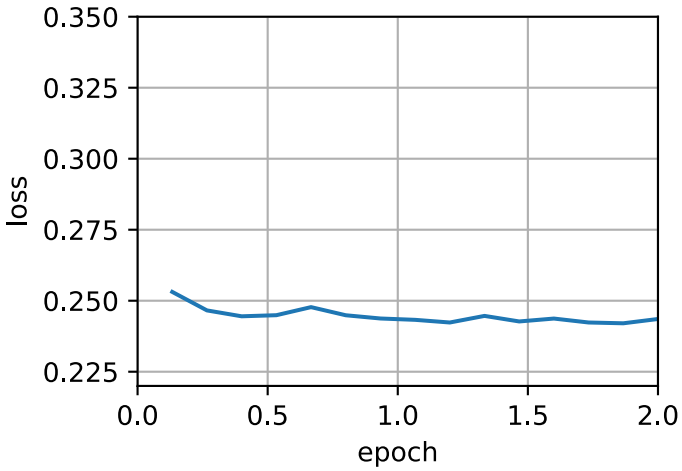
Tıpkı momentum yöntemi gibi, Adagrad'ın parametrelerle aynı şekle sahip bir durum değişkenini koruması gerekiyor.

```
def init_adagrad_states(feature_dim):
    s_w = torch.zeros((feature_dim, 1))
    s_b = torch.zeros(1)
    return (s_w, s_b)

def adagrad(params, states, hyperparams):
    eps = 1e-6
    for p, s in zip(params, states):
        with torch.no_grad():
            s[:] += torch.square(p.grad)
            p[:] -= hyperparams['lr'] * p.grad / torch.sqrt(s + eps)
        p.grad.data.zero_()
```

Section 11.5 içindeki deney ile karşılaştırıldığında, modeli eğitmek için daha büyük bir öğrenme hızı kullanıyoruz.

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(adagrad, init_adagrad_states(feature_dim),
               {'lr': 0.1}, data_iter, feature_dim);
```

```
loss: 0.244, 0.015 sec/epoch
```

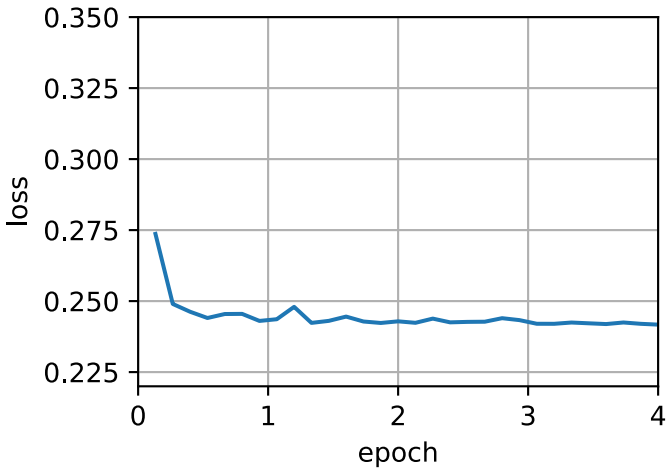


11.7.5 Kısa Uygulama

adagrad algoritmasının `Trainer` örneğini kullanarak, Gluon'daki Adagrad algoritmasını çağırabiliriz.

```
trainer = torch.optim.Adagrad
d2l.train_concise_ch11(trainer, {'lr': 0.1}, data_iter)
```

```
loss: 0.242, 0.015 sec/epoch
```



11.7.6 Özet

- Adagrad, koordinat tabanında öğrenme oranını dinamik olarak düşürür.
- Gradyanın büyüklüğünü, ilerlemenin ne kadar hızlı bir şekilde elde edildiğini ayarlamak için bir araç olarak kullanır - büyük gradyanlara sahip koordinatlar daha küçük bir öğrenme oranı ile telafi edilir.
- İkinci türevi tam hesaplamak, bellek ve hesaplama kısıtlamaları nedeniyle derin öğrenme problemlerinde genellikle mümkün değildir. Gradyan yararlı bir vekil olabilir.
- Optimizasyon problemi oldukça düzensiz bir yapıya sahipse Adagrad bozulmayı azaltmaya yardımcı olabilir.
- Adagrad, özellikle seyrek olarak ortaya çıkan terimler için öğrenme oranının daha yavaş azalması gereken seyrek öznitelikler için etkilidir.
- Derin öğrenme problemleri üzerinde Adagrad bazen öğrenme oranlarını düşürmede çok saldırgan olabilir. Section 11.10 bağlamında bunu hafifletmek için stratejileri tartışacağız.

11.7.7 Alıştırmalar

1. Bir dik matris $\mathbf{U}$ ve bir vektör $\mathbf{c}$ için şunu kanıtlayın: $\|\mathbf{c} - \delta\|_2 = \|\mathbf{U}\mathbf{c} - \mathbf{U}\delta\|_2$. Bu neden, değişkenlerin dik değişiminden sonra dürtmelerin büyüklüğünün değişmediği anlamına geliyor?
2. $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ ve ayrıca amaç fonksiyonunun 45 derece, yani $f(\mathbf{x}) = 0.1(x_1 + x_2)^2 + 2(x_1 - x_2)^2$ ile döndürüldüğü durum için Adagrad'ı deneyin. Farklı davranıyor mu?
3. $\mathbf{M}$ matrisinin λ_i özdeğerlerinin $|\lambda_i - \mathbf{M}_{jj}| \leq \sum_{k \neq j} |\mathbf{M}_{jk}|$'yi en az bir j seçeneği için Gerschgorin çember teoremini[139] sağladığını kanıtlayın.
4. Gerschgorin teoremi, çapraz olarak önceden koşullandırılmış matrisin $\mathrm{diag}^{-\frac{1}{2}}(\mathbf{M})\mathbf{M}\mathrm{diag}^{-\frac{1}{2}}(\mathbf{M})$'nin özdeğerleri hakkında bize ne anlatıyor?
5. Fashion MNIST'e uygulandığında Section 6.6 gibi uygun bir derin ağ için Adagrad'ı deneyin.
6. Öğrenme oranında daha az saldırgan bir sönme elde etmek için Adagrad'ı nasıl değiştirmeniz gerekir?

Tartışmalar[140]

11.8 RMSProp

Section 11.7 içindeki en önemli konulardan biri, öğrenme oranının önceden tanımlanmış bir zamanlamayla $\mathcal{O}(t^{-\frac{1}{2}})$ etkin bir şekilde azalmasıdır. Bu genellikle dışbükey problemler için uygun olsa da, derin öğrenmede karşılaşılanlar gibi dışbükey olmayan olanlar için ideal olmayabilir. Yine de, Adagrad'ın koordinat yönlü uyarlanması ön koşul olarak son derece arzu edilir.

(Tieleman and Hinton, 2012), oran zamanlamasını koordinat-uyarlamalı öğrenme oranlarından ayırmak için basit bir düzeltme olarak RMSProp algoritmasını önerdi. Sorun, Adagrad'ın $\mathbf{g}_t$ gradyanının karelerini bir durum vektörü $\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g}_t^2$ olarak biriktirmesidir. Sonuç olarak

[139] https://en.wikipedia.org/wiki/Gershgorin_circle_theorem
[140] https://discuss.d2l.ai/t/1072

$\mathbf{s}_t$, normalleştirme eksikliğinden dolayı sınır olmadan, esasında algoritma yakınsadıkça doğrusal olarak büyümeye devam eder.

Bu sorunu çözmenin bir yolu $\mathbf{s}_t/t$'yi kullanmaktır. $\mathbf{g}_t$'nin makul dağılımları için bu yakınsayacaktır. Ne yazık ki, prosedür tüm değerlerin yörüngesini hatırladığından, limit davranışının etkin olmaya başlaması çok uzun zaman alabilir. Alternatifi, momentum yönteminde kullandığımız gibi sızan bir ortalama kullanmaktır, yani bazı $\gamma > 0$ parametresi için $\mathbf{s}_t \leftarrow \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t^2$. Diğer tüm parçaları değiştirmeden tutmak, RMSProp'u verir.

11.8.1 Algoritma

Denklemleri ayrıntılı olarak yazalım.

$$\begin{aligned} \mathbf{s}_t &\leftarrow \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t^2, \\ \mathbf{x}_t &\leftarrow \mathbf{x}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t. \end{aligned} \tag{11.8.1}$$

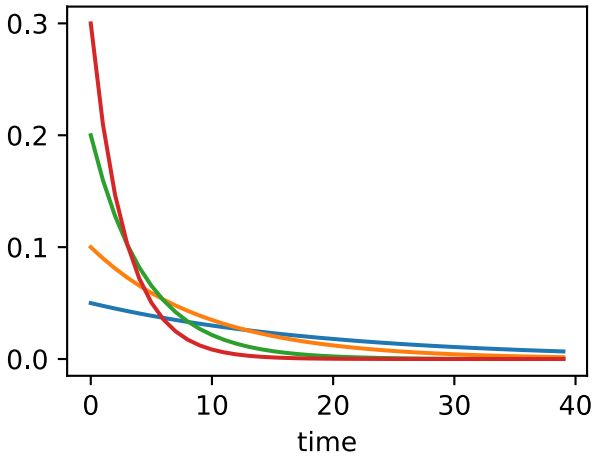
$\epsilon > 0$ sabiti, sıfır veya aşırı büyük adım boyutlarına bölünmediğimizden emin olmak için tipik olarak 10^{-6} olarak ayarlanır. Bu genişleme göz önüne alındığında, koordinat başına uygulanan ölçeklendirmeden bağımsız olarak η öğrenme oranını kontrol etmede serbestiz. Sızdıran ortalamalar açısından, momentum yöntemi için daha önce uygulanan mantığın aynısını uygulayabiliriz. $\mathbf{s}_t$ tanımını genişletmek şu ifadeye yol açar:

$$\begin{aligned} \mathbf{s}_t &= (1-\gamma)\mathbf{g}_t^2 + \gamma \mathbf{s}_{t-1} \\ &= (1-\gamma)\left(\mathbf{g}_t^2 + \gamma \mathbf{g}_{t-1}^2 + \gamma^2 \mathbf{g}_{t-2} + \ldots, \right). \end{aligned} \tag{11.8.2}$$

Daha önce Section 11.6 içinde olduğu gibi $1 + \gamma + \gamma^2 + \ldots, = \frac{1}{1-\gamma}$ kullanıyoruz. Bu nedenle ağırlıkların toplamı γ^{-1}'lik bir gözlem yarılanma ömrü ile 1'e normalleştirilir. Çeşitli γ seçenekleri için son 40 zaman adımının ağırlıklarını görselleştirelim.

```
import math
import torch
from d2l import torch as d2l
```

```
d2l.set_figsize()
gammas = [0.95, 0.9, 0.8, 0.7]
for gamma in gammas:
    x = torch.arange(40).detach().numpy()
    d2l.plt.plot(x, (1-gamma) * gamma ** x, label=f'gamma = {gamma:.2f}')
d2l.plt.xlabel('time');
```

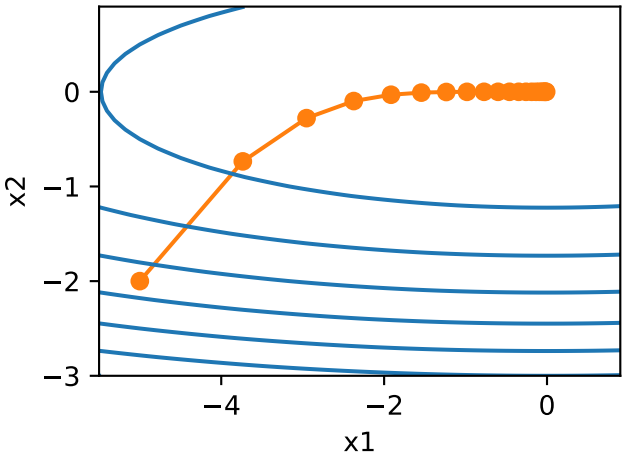


11.8.2 Sıfırdan Uygulama

Daha önce olduğu gibi, RMSProp yörüngesini gözlemlemek için $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ ikinci dereceden polinom işlevini kullanıyoruz. Section 11.7 içinde Adagrad'ı 0.4 öğrenme oranıyla kullandığımızda, öğrenme oranı çok hızlı düştüğü için değişkenlerin algoritmanın sonraki aşamalarında çok yavaş hareket ettiğini hatırlayın. η ayrı ayrı kontrol edildiğinden bu RMSProp ile gerçekleşmez.

```
def rmsprop_2d(x1, x2, s1, s2):
    g1, g2, eps = 0.2 * x1, 4 * x2, 1e-6
    s1 = gamma * s1 + (1 - gamma) * g1 ** 2
    s2 = gamma * s2 + (1 - gamma) * g2 ** 2
    x1 -= eta / math.sqrt(s1 + eps) * g1
    x2 -= eta / math.sqrt(s2 + eps) * g2
    return x1, x2, s1, s2

def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2

eta, gamma = 0.4, 0.9
d2l.show_trace_2d(f_2d, d2l.train_2d(rmsprop_2d))
```

```
epoch 20, x1: -0.010599, x2: 0.000000
/home/d2l-worker/miniconda3/envs/d2l-tr-release-0/lib/python3.9/site-packages/torch/
↪functional.py:478: UserWarning: torch.meshgrid: in an upcoming release, it will be␣
↪required to pass the indexing argument. (Triggered internally at  ../aten/src/ATen/native/
↪TensorShape.cpp:2895.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
```



Ardından, derin bir ağda kullanılacak RMSProp'u uyguluyoruz. Bu da son derece basit.

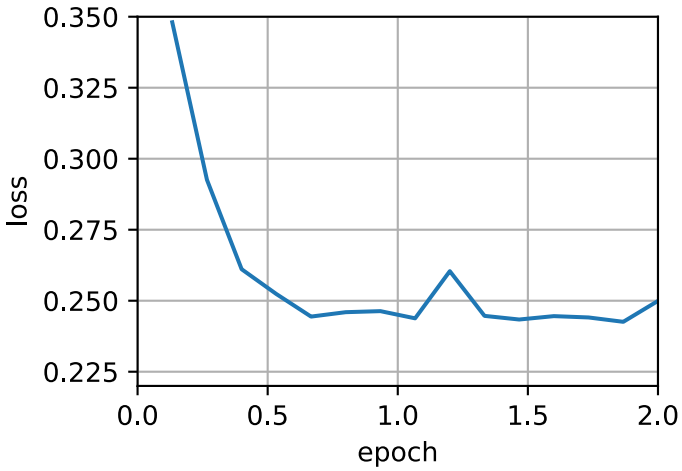
```
def init_rmsprop_states(feature_dim):
    s_w = torch.zeros((feature_dim, 1))
    s_b = torch.zeros(1)
    return (s_w, s_b)
```

```
def rmsprop(params, states, hyperparams):
    gamma, eps = hyperparams['gamma'], 1e-6
    for p, s in zip(params, states):
        with torch.no_grad():
            s[:] = gamma * s + (1 - gamma) * torch.square(p.grad)
            p[:] -= hyperparams['lr'] * p.grad / torch.sqrt(s + eps)
        p.grad.data.zero_()
```

İlk öğrenme oranını 0.01 ve ağırlıklandırma terimi γ'yı 0.9'a ayarladık. Yani, $\mathbf{s}$, kare gradyanın son $1/(1-\gamma) = 10$ adet gözlem üzerinden ortalama olarak toplanır.

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(rmsprop, init_rmsprop_states(feature_dim),
               {'lr': 0.01, 'gamma': 0.9}, data_iter, feature_dim);
```

```
loss: 0.250, 0.015 sec/epoch
```

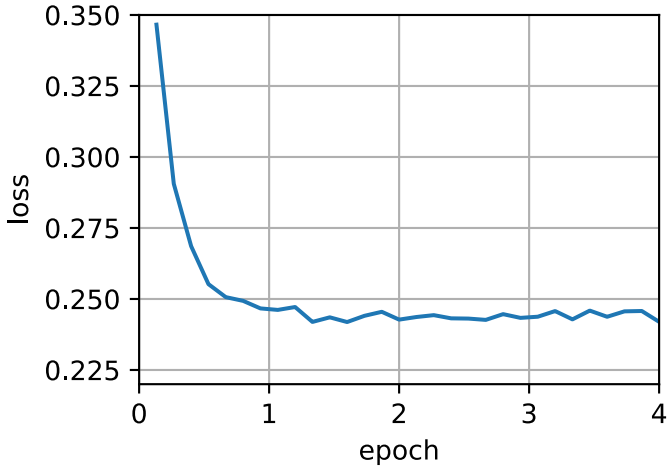


11.8.3 Kısa Uygulama

RMSProp oldukça popüler bir algoritma olduğundan `Trainer` örneğinde de mevcuttur. Tek yapmamız gereken `rmsprop` adlı bir algoritma kullanarak, γ'yı `gamma1` parametresine atamak.

```
trainer = torch.optim.RMSprop
d2l.train_concise_ch11(trainer, {'lr': 0.01, 'alpha': 0.9},
                       data_iter)
```

```
loss: 0.242, 0.014 sec/epoch
```



11.8.4 Özet

- RMSProp, her ikisi de katsayıları ölçeklendirmek için gradyanın karesini kullandığı için Adagrad'a çok benzer.
- RMSProp, sızıntı ortalamasını momentum ile paylaşır. Bununla birlikte, RMSProp, katsayı bazında ön koşullayıcıyı ayarlamak için bu tekniği kullanır.
- Öğrenme oranının pratikte deney yapan tarafından planlanması gerekir.
- γ katsayısı, koordinat başına ölçeği ayarlarken geçmişin ne kadar geri gideceğini belirler.

11.8.5 Alıştırmalar

1. $\gamma = 1$'i ayarlarsak deneysel olarak ne olur? Neden?
2. Optimizasyon problemi $f(\mathbf{x}) = 0.1(x_1 + x_2)^2 + 2(x_1 - x_2)^2$'yi en aza indirmek için döndürün. Yakınsamaya ne olur?
3. Fashion-MNIST eğitimi gibi gerçek bir makine öğrenmesi probleminde RMSProp'a ne olduğunu deneyin. Öğrenme oranını ayarlamak için farklı seçeneklerle denemeler yapın.
4. Optimizasyon ilerledikçe γ'yı ayarlamak ister misiniz? RMSProp buna ne kadar duyarlıdır?

Tartışmalar[141]

11.9 Adadelta

Adadelta, AdaGrad'ın başka bir çeşididir (Section 11.7). Temel fark, öğrenme oranının koordinatlara uyarlanabilir olduğu miktarı azaltmasıdır. Ayrıca, gelecekteki değişim için kalibrasyon olarak değişiklik miktarını kullandığından, geleneksel olarak bir öğrenme oranına sahip olmamak olarak adlandırılır. Algoritma (Zeiler, 2012) çalışmasında önerildi. Şimdiye kadar önceki algoritmaların tartışılması göz önüne alındığında oldukça basittir.

11.9.1 Algoritma

Özetle, Adadelta, gradyanın ikinci momentinin sızıntılı ortalamasını depolamak için $\mathbf{s}_t$ ve modeldeki parametrelerin değişiminin ikinci momentinin sızıntılı ortalamasını depolamak için $\Delta\mathbf{x}_t$ diye iki durum değişkeni kullanır. Yazarların orijinal gösterimini ve adlandırmalarını diğer yayınlarla ve uygulamalarla uyumluluk için kullandığımızı unutmayın (momentum, Adagrad, RMSProp ve Adadelta, Adagrad, RMSProp ve Adadelta'da aynı amaca hizmet eden bir parametreyi belirtmek için farklı Yunan değişkenleri kullanılmasının başka bir gerçek nedeni yoktur).

İşte Adadelta'nın teknik detayları verelim. Servis edilen parametre ρ olduğu göz önüne alındığında, Section 11.8'a benzer şekilde aşağıdaki sızdıran güncellemeleri elde ediyoruz:

$$\mathbf{s}_t = \rho \mathbf{s}_{t-1} + (1 - \rho)\mathbf{g}_t^2. \tag{11.9.1}$$

Section 11.8 ile arasındaki fark, güncellemeleri yeniden ölçeklendirilmiş $\mathbf{g}_t'$ gradyanı ile gerçekleştirmemizdir, yani

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \mathbf{g}_t'. \tag{11.9.2}$$

[141] https://discuss.d2l.ai/t/1074

Peki yeniden ölçeklendirilmiş gradyan $\mathbf{g}'_t$ nedir? Bunu aşağıdaki gibi hesaplayabiliriz:

$$\mathbf{g}'_t = \frac{\sqrt{\Delta\mathbf{x}_{t-1} + \epsilon}}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t, \tag{11.9.3}$$

burada $\Delta\mathbf{x}_{t-1}$ karesi yeniden ölçeklendirilmiş gradyanların sızdıran ortalaması $\mathbf{g}'_t$ dir. $\Delta\mathbf{x}_0$'ı 0 olacak şekilde ilkleriz ve her adımda $\mathbf{g}'_t$ ile güncelleriz, yani,

$$\Delta\mathbf{x}_t = \rho\Delta\mathbf{x}_{t-1} + (1-\rho){\mathbf{g}'_t}^2, \tag{11.9.4}$$

ve ϵ (10^{-5} gibi küçük bir değerdir) sayısal kararlılığı korumak için eklenir.

11.9.2 Uygulama

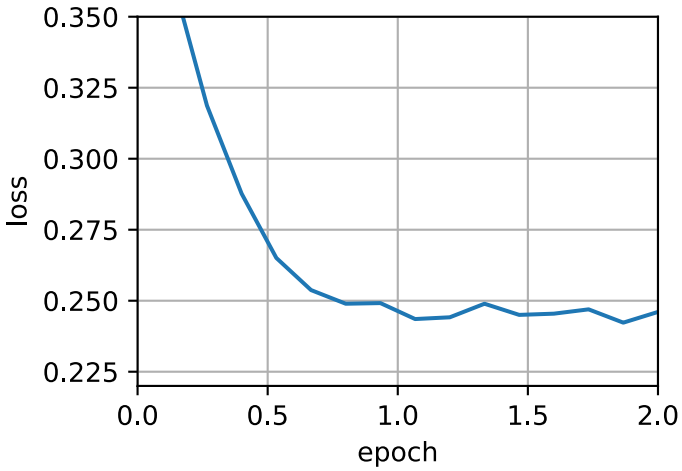
Adadelta, her değişken için iki durum değişkenini, $\mathbf{s}_t$ ve $\Delta\mathbf{x}_t$, korumalıdır. Bu, aşağıdaki uygulamayı verir.

```
%matplotlib inline
import torch
from d2l import torch as d2l


def init_adadelta_states(feature_dim):
    s_w, s_b = torch.zeros((feature_dim, 1)), torch.zeros(1)
    delta_w, delta_b = torch.zeros((feature_dim, 1)), torch.zeros(1)
    return ((s_w, delta_w), (s_b, delta_b))

def adadelta(params, states, hyperparams):
    rho, eps = hyperparams['rho'], 1e-5
    for p, (s, delta) in zip(params, states):
        with torch.no_grad():
            # [:] aracılığıyla yerinde güncellemeler
            s[:] = rho * s + (1 - rho) * torch.square(p.grad)
            g = (torch.sqrt(delta + eps) / torch.sqrt(s + eps)) * p.grad
            p[:] -= g
            delta[:] = rho * delta + (1 - rho) * g * g
        p.grad.data.zero_()
```

$\rho = 0.9$ seçilmesi, her parametre güncelleştirmesi için 10'luk yarı ömür süresine ulaşır. Bu oldukça iyi çalışma eğilimindedir. Aşağıdaki davranışları görürüz.

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(adadelta, init_adadelta_states(feature_dim),
               {'rho': 0.9}, data_iter, feature_dim);
```

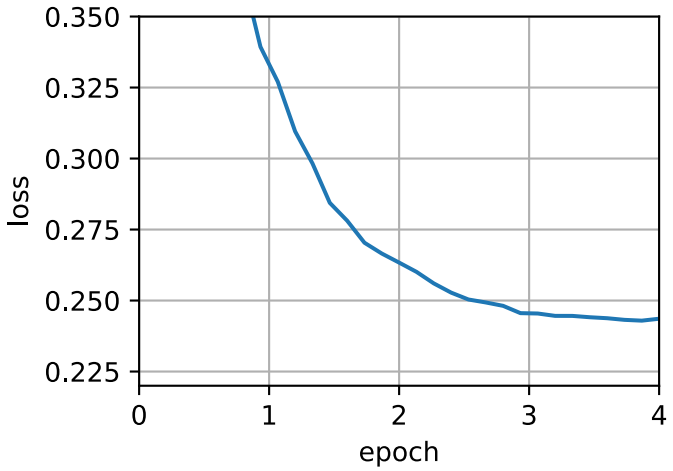
```
loss: 0.246, 0.016 sec/epoch
```



Kısa bir uygulama için `Trainer` sınıfından `adadelta` algoritmasını kullanıyoruz. Bu, çok daha sıkılmıştır bir çağırma için aşağıdaki tek-satırlık kodu verir.

```
trainer = torch.optim.Adadelta
d2l.train_concise_ch11(trainer, {'rho': 0.9}, data_iter)
```

```
loss: 0.244, 0.014 sec/epoch
```



11.9.3 Özet

- Adadelta'nın öğrenme oranı parametresi yoktur. Bunun yerine, öğrenme oranını uyarlamak için parametrelerin kendisindeki değişim oranını kullanır.
- Adadelta, gradyanın ikinci momentleri ve parametrelerdeki değişikliği depolamak için iki durum değişkenine ihtiyaç duyar.
- Adadelta, uygun istatistiklerin işleyen bir tahminini tutmak için sızdıran ortalamaları kullanır.

11.9.4 Alıştırmalar

1. ρ değerini ayarlayın. Ne olur?
2. $\mathbf{g}'_t$ kullanmadan algoritmanın nasıl uygulanacağını gösterin. Bu neden iyi bir fikir olabilir?
3. Adadelta gerçekten oranını bedavaya mı öğreniyor? Adadelta'yı bozan optimizasyon problemlerini bulabilir misin?
4. Yakınsama davranışlarını tartışmak için Adadelta'yı Adagrad ve RMSprop ile karşılaştırın.

Tartışmalar[142]

11.10 Adam

Bu bölüme kadar uzanan tartışmalarda etkili optimizasyon için bir dizi teknikle karşılaştık. Onları burada ayrıntılı olarak özetleyelim:

- Section 11.4 içinde optimizasyon problemlerini çözerken, örneğin bol veriye olan doğal dayanıklılığı nedeniyle gradyan inişinden daha etkili olduğunu gördük.
- Section 11.5 içinde bir minigrup içinde daha büyük gözlem kümeleri kullanarak vektörleştirmeden kaynaklanan önemli ek verimlilik sağladığını gördük. Bu, verimli çoklu makine, çoklu GPU ve genel paralel işleme için anahtardır.
- Section 11.6 yakınsamayı hızlandırmak için geçmiş gradyanların tarihçesini bir araya getirmeye yönelik bir mekanizma ekledi.
- Section 11.7 hesaplama açısından verimli bir ön koşul sağlamak için koordinat başına ölçekleme kullanıldı.
- Section 11.8 bir öğrenme hızı ayarlaması ile koordinat başına ölçeklendirmeyi ayrıştırdı.

Adam (Kingma and Ba, 2014), tüm bu teknikleri tek bir verimli öğrenme algoritmasına birleştirir. Beklendiği gibi, bu, derin öğrenmede kullanılacak daha sağlam ve etkili optimizasyon algoritmalarından biri olarak oldukça popüler hale gelen bir algoritmadır. Yine de sorunları yok değildir. Özellikle, (Reddi *et al.*, 2019), Adam'ın zayıf varyans kontrolü nedeniyle ıraksayabileceği durumlar olduğunu göstermektedir. Bir takip çalışması (Zaheer *et al.*, 2018) Adam için bu sorunları gideren Yogi denilen bir düzeltme önerdi. Bunun hakkında daha fazlasını birazdan vereceğiz. Şimdilik Adam algoritmasını gözden geçirelim.

11.10.1 Algoritma

Adam'ın temel bileşenlerinden biri, hem momentumu hem de gradyanın ikinci momentini tahmin etmek için üstel ağırlıklı hareketli ortalamaları (sızdıran ortalama olarak da bilinir) kullanmasıdır. Yani, durum değişkenleri kullanır

$$\begin{aligned}\mathbf{v}_t &\leftarrow \beta_1 \mathbf{v}_{t-1} + (1 - \beta_1) \mathbf{g}_t, \\ \mathbf{s}_t &\leftarrow \beta_2 \mathbf{s}_{t-1} + (1 - \beta_2) \mathbf{g}_t^2.\end{aligned} \tag{11.10.1}$$

Burada β_1 ve β_2 negatif olmayan ağırlıklandırma parametreleridir. Onlar için yaygın seçenekler $\beta_1 = 0.9$ ve $\beta_2 = 0.999$'dur. Yani, varyans tahmini momentum teriminden *çok daha yavaş*

[142] https://discuss.d2l.ai/t/1076

hareket eder. $\mathbf{v}_0 = \mathbf{s}_0 = 0$ şeklinde ilklersek, başlangıçta daha küçük değerlere karşı önemli miktarda taraflı olduğumuzu unutmayın. Bu, $\sum_{i=0}^{t} \beta^i = \frac{1-\beta^t}{1-\beta}$'nin terimleri yeniden normalleştirmesi gerçeğini kullanarak ele alınabilir. Buna göre normalleştirilmiş durum değişkenleri şu şekilde verilir:

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1 - \beta_1^t} \text{ ve } \hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1 - \beta_2^t}. \tag{11.10.2}$$

Uygun tahminlerle donanmış olarak şimdi güncelleme denklemlerini yazabiliriz. İlk olarak, RMSProp'a çok benzer bir şekilde gradyanı yeniden ölçeklendirerek aşağıdaki ifadeyi elde ederiz:

$$\mathbf{g}_t' = \frac{\eta \hat{\mathbf{v}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}. \tag{11.10.3}$$

RMSProp'un aksine güncellememiz gradyanın kendisi yerine $\hat{\mathbf{v}}_t$ momentumunu kullanır. Ayrıca, yeniden ölçeklendirme $\frac{1}{\sqrt{\hat{\mathbf{s}}_t}+\epsilon}$ yerine $\frac{1}{\sqrt{\hat{\mathbf{s}}_t}+\epsilon}$ kullanarak gerçekleştiği için hafif bir kozmetik fark vardır. Sonraki, pratikte tartışmasız olarak biraz daha iyi çalışır, dolayısıyla RMSProp'tan bir sapmadır. Sayısal kararlılık ve aslına uygunluk arasında iyi bir denge için tipik olarak $\epsilon = 10^{-6}$'yı seçeriz.

Şimdi güncellemeleri hesaplamak için tüm parçalara sahibiz. Bu biraz hayal kırıcıdır ve formun basit bir güncellemesi vardır:

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \mathbf{g}_t'. \tag{11.10.4}$$

Adam'ın tasarımı gözden geçirildiğinde ilham kaynağı açıktır. Momentum ve ölçek durum değişkenlerinde açıkça görülebilir. Oldukça tuhaf tanımları bizi terimleri yansızlaştırmaya zorlar (bu biraz farklı bir ilkleme ve güncelleme koşuluyla düzeltilebilir). İkincisi, her iki terim kombinasyonu RMSProp göz önüne alındığında oldukça basittir. Son olarak, açık öğrenme oranı η, yakınsama sorunlarını çözmek için adım uzunluğunu kontrol etmemizi sağlar.

11.10.2 Uygulama

Adam'ı sıfırdan uygulama çok da göz korkutucu değil. Kolaylık sağlamak için `hyperparams` sözlüğünde t zaman adımı sayacını saklıyoruz. Bunun ötesinde her şey basittir.

```
%matplotlib inline
import torch
from d2l import torch as d2l


def init_adam_states(feature_dim):
    v_w, v_b = torch.zeros((feature_dim, 1)), torch.zeros(1)
    s_w, s_b = torch.zeros((feature_dim, 1)), torch.zeros(1)
    return ((v_w, s_w), (v_b, s_b))

def adam(params, states, hyperparams):
    beta1, beta2, eps = 0.9, 0.999, 1e-6
    for p, (v, s) in zip(params, states):
        with torch.no_grad():
            v[:] = beta1 * v + (1 - beta1) * p.grad
            s[:] = beta2 * s + (1 - beta2) * torch.square(p.grad)
            v_bias_corr = v / (1 - beta1 ** hyperparams['t'])
```

(continues on next page)

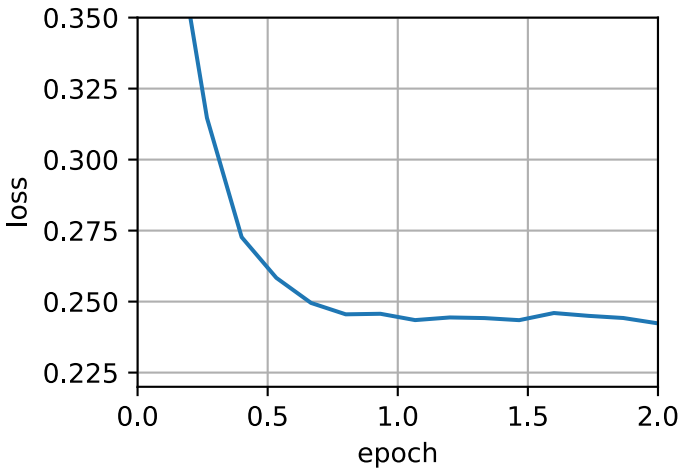
(continued from previous page)

```
            s_bias_corr = s / (1 - beta2 ** hyperparams['t'])
            p[:] -= hyperparams['lr'] * v_bias_corr / (torch.sqrt(s_bias_corr)
                                                       + eps)
        p.grad.data.zero_()
    hyperparams['t'] += 1
```

Modelini eğitmek için Adam'ı kullanmaya hazırız. $\eta = 0.01$ öğrenim oranını kullanıyoruz.

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(adam, init_adam_states(feature_dim),
               {'lr': 0.01, 't': 1}, data_iter, feature_dim);
```

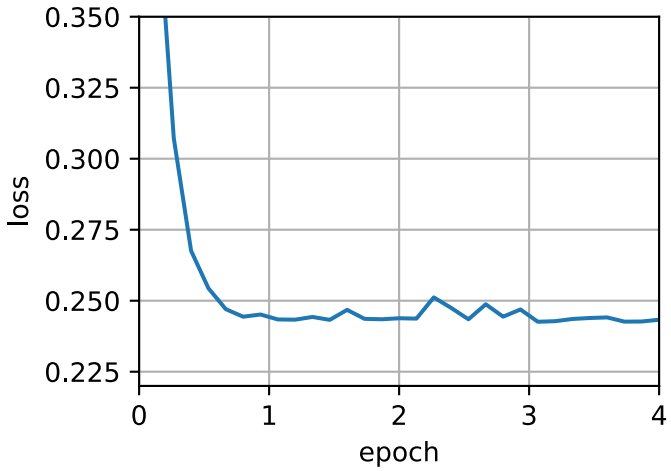
```
loss: 0.242, 0.016 sec/epoch
```



`adam` Gluon `trainer` optimizasyon kütüphanesinin bir parçası olarak sağlanan algoritmalardan biri olduğundan daha kısa bir uygulama barizdir. Bu nedenle, Gluon'daki bir uygulama için yalnızca yapılandırma parametrelerini geçmemiz gerekiyor.

```
trainer = torch.optim.Adam
d2l.train_concise_ch11(trainer, {'lr': 0.01}, data_iter)
```

```
loss: 0.243, 0.015 sec/epoch
```



11.10.3 Yogi

Adam'ın sorunlarından biri, $\mathbf{s}_t$'teki ikinci moment tahmini patladığında dışbükey ayarlarda bile yakınsamayabilmesidir. Bir düzeltme olarak (Zaheer *et al.*, 2018) $\mathbf{s}_t$ için arıtılmış bir güncelleme (ve ilkleme) önerdi. Neler olup bittiğini anlamak için, Adam güncellemesini aşağıdaki gibi yeniden yazalım:

$$\mathbf{s}_t \leftarrow \mathbf{s}_{t-1} + (1 - \beta_2) \left(\mathbf{g}_t^2 - \mathbf{s}_{t-1}\right). \tag{11.10.5}$$

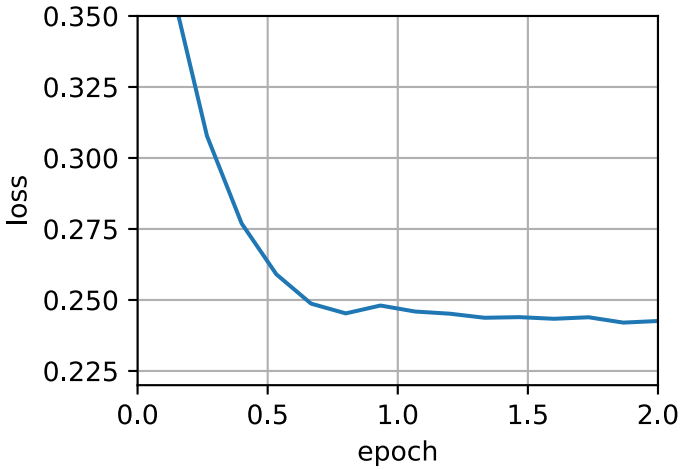
$\mathbf{g}_t^2$ yüksek varyansa sahip olduğunda veya güncellemeler seyrek olduğunda, $\mathbf{s}_t$ geçmiş değerleri çok çabuk unutabilir. Bunun için olası bir düzeltme $\mathbf{g}_t^2 - \mathbf{s}_{t-1}$'yı $\mathbf{g}_t^2 \odot \mathrm{sgn}(\mathbf{g}_t^2 - \mathbf{s}_{t-1})$ ile değiştirmektir. Artık güncellemenin büyüklüğü sapma miktarına bağlı değil. Bu Yogi güncellemelerini verir:

$$\mathbf{s}_t \leftarrow \mathbf{s}_{t-1} + (1 - \beta_2)\mathbf{g}_t^2 \odot \mathrm{sgn}(\mathbf{g}_t^2 - \mathbf{s}_{t-1}). \tag{11.10.6}$$

Yazarlar ayrıca momentumu sadece ilk noktasal tahminden ziyade daha büyük bir ilk toplu iş ile ilklemeyi tavsiye ediyor. Tartışmada önemli olmadıkları ve bu yakınsama olmasa bile oldukça iyi kaldığı için ayrıntıları atlıyoruz.

```
def yogi(params, states, hyperparams):
    beta1, beta2, eps = 0.9, 0.999, 1e-3
    for p, (v, s) in zip(params, states):
        with torch.no_grad():
            v[:] = beta1 * v + (1 - beta1) * p.grad
            s[:] = s + (1 - beta2) * torch.sign(
                torch.square(p.grad) - s) * torch.square(p.grad)
            v_bias_corr = v / (1 - beta1 ** hyperparams['t'])
            s_bias_corr = s / (1 - beta2 ** hyperparams['t'])
            p[:] -= hyperparams['lr'] * v_bias_corr / (torch.sqrt(s_bias_corr)
                                                       + eps)
        p.grad.data.zero_()
    hyperparams['t'] += 1

data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(yogi, init_adam_states(feature_dim),
               {'lr': 0.01, 't': 1}, data_iter, feature_dim);
```

```
loss: 0.243, 0.017 sec/epoch
```



11.10.4 Özet

- Adam, birçok optimizasyon algoritmasının özelliklerini oldukça sağlam bir güncelleme kuralı haline getirir.
- RMSProp temelinde oluşturulan Adam ayrıca minigrup rasgele gradyan üzerinde EWMA kullanır.
- Adam, momentumu ve ikinci bir momenti tahmin ederken yavaş başlatmayı ayarlamak için ek girdi düzeltmesini kullanır.
- Önemli varyansa sahip gradyanlar için yakınsama sorunlarıyla karşılaşabiliriz. Bunlar, daha büyük minigruplar kullanılarak veya $\mathbf{s}_t$ için geliştirilmiş bir tahmine geçilerek değiştirilebilir. Yogi böyle bir alternatif sunuyor.

11.10.5 Alıştırmalar

1. Öğrenme oranını ayarlayın ve deneysel sonuçları gözlemleyip analiz edin.
2. Eğer momentum ve ikinci moment güncellemeleri, ek girdi düzeltme gerektirmeyecek şekilde yeniden yazabilir misiniz?
3. Neden yakınsadığımızda η öğrenme oranını düşürmeniz gerekiyor?
4. Adam'ın ıraksadığında ve Yogi'nin yakınsadığı bir durum oluşturmaya mı çalışın.

Tartışmalar[143]

[143] https://discuss.d2l.ai/t/1078

11.11 Öğrenme Oranını Zamanlama

Şimdiye kadar öncelikle ağırlık vektörlerinin güncellendikleri *oran* yerine ağırlık vektörlerinin nasıl güncelleneceğine dair optimizasyon *algoritmalarına* odaklandık. Bununla birlikte, öğrenme oranının ayarlanması genellikle gerçek algoritma kadar önemlidir. Göz önünde bulundurulması gereken bir dizi husus vardır:

- En çok açıkçası öğrenme oranının *büyüklüğü* önemlidir. Çok büyükse, optimizasyon ıraksar, eğer çok küçükse, eğitilmesi çok uzun sürer veya yarı-optimal bir sonuç elde ederiz. Daha önce problemin sağlamlık sayısının önemli olduğunu gördük (ayrıntılar için bkz. Section 11.6). Sezgisel olarak, en hassas yöndeki değişim miktarının en az hassas olanına oranıdır.
- İkincisi, sönme oranı da aynı derecede önemlidir. Öğrenme oranı büyük kalırsa, en küçük seviyeyi atlayabilir ve böylece eniyiye ulaşamayabiliriz. Section 11.5 içinde bunu ayrıntılı olarak tartıştık ve Section 11.4 içinde performans garantilerini analiz ettik. Kısacası, sönme oranını istiyoruz, ama muhtemelen dışbükey problemler için $\mathcal{O}(t^{-\frac{1}{2}})$ daha yavaş olanlar iyi bir seçim olurdu.
- Eşit derecede önemli olan bir diğer yön ise *ilklemedir*. Bu, hem parametrelerin başlangıçta nasıl ayarlandığı (ayrıntılar için bkz.:numref:`sec_numerical_stability`) hem de başlangıçta nasıl evrimleştikleri de ilgilidir. Bu, *ısınma* diye tabir edilir, yani başlangıçta çözüme doğru ne kadar hızlı ilerlemeye başladığımızla ilgilidir. Başlangıçta büyük adımlar yararlı olmayabilir, özellikle de ilk parametre kümesi rastgele olduğundan. İlk güncelleme yönergeleri de oldukça anlamsız olabilir.
- Son olarak, döngüsel öğrenme hızı ayarlaması gerçekleştiren bir dizi optimizasyon türü vardır. Bu, bu bölümün kapsamı dışındadır. Okuyucuya (Izmailov *et al.*, 2018) çalışmasındaki ayrıntıları gözden geçirmesini tavsiye ederiz, örneğin, parametrelerin izlediği *yolun* üzerinden ortalama alarak daha iyi çözümler elde etmek gibi.

Öğrenme oranlarını yönetmek için gereken çok fazla ayrıntı olduğu göz önüne alındığında, çoğu derin öğrenme çerçevesinin bununla otomatik olarak başa çıkabilmesi için araçlar vardır. Bu bölümde, farklı zamanlamaların doğruluk üzerindeki etkilerini gözden geçireceğiz ve ayrıca bunun bir *öğrenme oranı zamanlayıcısı* aracılığıyla nasıl verimli bir şekilde yönetilebileceğini göstereceğiz.

11.11.1 Basit Örnek Problem

Kolayca hesaplanabilecek kadar ucuz, ancak bazı önemli hususları göstermek için yeterince açık olmayan bir basit örnek problem ile başlıyoruz. Bunun için Fashion-MNIST'e uygulandığı gibi LeNet'in biraz modern bir sürümünü seçin (`relu` yerine `sigmoid` aktivasyon, MaxPooling (Maksimum Havuzlama) yerine AveragePooling (Ortalama Havuzlama)). Dahası, ağı performans için melezleştiriyoruz. Kodun çoğu standart olduğundan, daha fazla ayrıntılı tartışma yapmadan sadece temel bilgileri sunuyoruz. Gerektiğinde anımsamak için bkz. Chapter 6.

```
%matplotlib inline
import math
import torch
from torch import nn
from torch.optim import lr_scheduler
from d2l import torch as d2l
```

(continues on next page)

(continued from previous page)

```
def net_fn():
    model = nn.Sequential(
        nn.Conv2d(1, 6, kernel_size=5, padding=2), nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Conv2d(6, 16, kernel_size=5), nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Flatten(),
        nn.Linear(16 * 5 * 5, 120), nn.ReLU(),
        nn.Linear(120, 84), nn.ReLU(),
        nn.Linear(84, 10))

    return model

loss = nn.CrossEntropyLoss()
device = d2l.try_gpu()

batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=batch_size)

# Kod, evrişimli sinir ağları bölümünün lenet kısmında tanımlanan
# `d2l.train_ch6` ile neredeyse aynıdır.
def train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
          scheduler=None):
    net.to(device)
    animator = d2l.Animator(xlabel='epoch', xlim=[0, num_epochs],
                            legend=['train loss', 'train acc', 'test acc'])

    for epoch in range(num_epochs):
        metric = d2l.Accumulator(3)  # train_loss, train_acc, num_examples
        for i, (X, y) in enumerate(train_iter):
            net.train()
            trainer.zero_grad()
            X, y = X.to(device), y.to(device)
            y_hat = net(X)
            l = loss(y_hat, y)
            l.backward()
            trainer.step()
            with torch.no_grad():
                metric.add(l * X.shape[0], d2l.accuracy(y_hat, y), X.shape[0])
            train_loss = metric[0] / metric[2]
            train_acc = metric[1] / metric[2]
            if (i + 1) % 50 == 0:
                animator.add(epoch + i / len(train_iter),
                             (train_loss, train_acc, None))

        test_acc = d2l.evaluate_accuracy_gpu(net, test_iter)
        animator.add(epoch+1, (None, None, test_acc))

        if scheduler:
            if scheduler.__module__ == lr_scheduler.__name__:
                # Using PyTorch In-Built scheduler
                scheduler.step()
            else:
                # Using custom defined scheduler
```

(continues on next page)

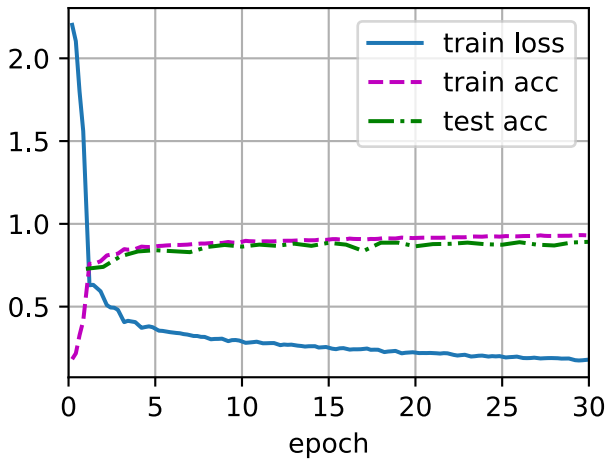
(continued from previous page)

```
            for param_group in trainer.param_groups:
                param_group['lr'] = scheduler(epoch)

    print(f'train loss {train_loss:.3f}, train acc {train_acc:.3f}, '
          f'test acc {test_acc:.3f}')
```

Bu algoritmayı 0.3 öğrenme oranı ve 30 yinelemeyle eğitmek için varsayılan ayarlarla çalıştırırsak ne olacağına bir göz atalım. Test doğruluğu açısından ilerleme bir noktanın ötesinde dururken eğitim doğruluğunun nasıl artmaya devam ettiğine dikkat edin. Her iki eğri arasındaki boşluk aşırı öğrenmeyi işaret eder.

```
lr, num_epochs = 0.3, 30
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr=lr)
train(net, train_iter, test_iter, num_epochs, loss, trainer, device)
```

```
train loss 0.181, train acc 0.930, test acc 0.892
```



11.11.2 Zamanlayıcılar

Öğrenme oranını ayarlamanın bir yolu, her adımda açıkça ayarlamaktır. Bu, `set_learning_rate` yöntemi ile elverişli bir şekilde elde edilir. Her dönemden sonra (hatta her minigrup işleminden sonra), örneğin, optimizasyonun nasıl ilerlediğine yanıt olarak dinamik bir şekilde, aşağı doğru ayarlayabiliriz.

```
lr = 0.1
trainer.param_groups[0]["lr"] = lr
print(f'learning rate is now {trainer.param_groups[0]["lr"]:.2f}')
```

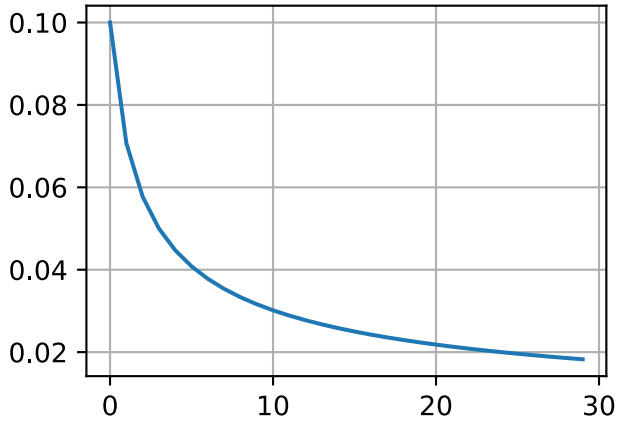
```
learning rate is now 0.10
```

Daha genel olarak bir zamanlayıcı tanımlamak istiyoruz. Güncelleme sayısı ile çağrıldığında, öğrenme oranının uygun değerini döndürür. Öğrenme oranını $\eta = \eta_0(t+1)^{-\frac{1}{2}}$'ye ayarlayan basit bir tane tanımlayalım.

```
class SquareRootScheduler:
    def __init__(self, lr=0.1):
        self.lr = lr

    def __call__(self, num_update):
        return self.lr * pow(num_update + 1.0, -0.5)
```

Davranışını bir dizi değer üzerinde çizelim.

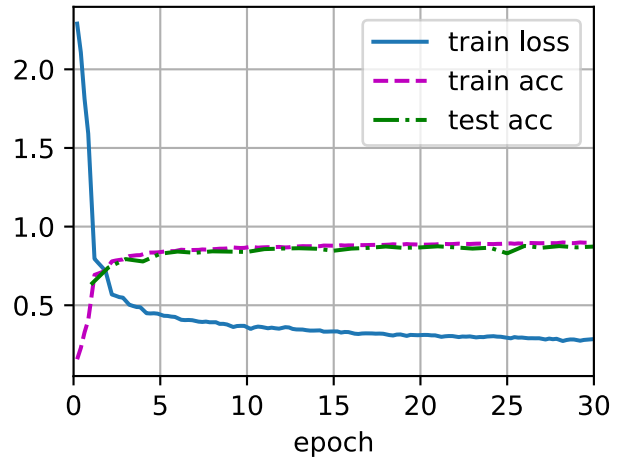
```
scheduler = SquareRootScheduler(lr=0.1)
d2l.plot(torch.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```



Şimdi bunun Fashion-MNIST eğitimi için nasıl bittiğini görelim. Zamanlayıcıyı eğitim algoritmasına ek bir argüman olarak sağlıyoruz.

```
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr)
train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
      scheduler)
```

```
train loss 0.283, train acc 0.897, test acc 0.873
```



Bu eskisinden biraz daha iyi çalıştı. İki şey öne çıkıyor: Eğri daha öncekinden daha pürüzsüzdü. İkincisi, daha az aşırı öğrenme vardı. Ne yazık ki, belirli stratejilerin neden *teoride* daha az aşırı öğrenmeye yol açtığı tam olarak çözülmüş bir soru değil. Daha küçük bir adım boyutunun sıfıra yakın ve dolayısıyla daha basit olan parametrelere yol açacağına dair argümanlar vardır. Bununla birlikte, bu fenomeni tam olarak açıklamaz, çünkü gerçekten erken durmayız, sadece öğrenme oranını nazikçe azaltırız.

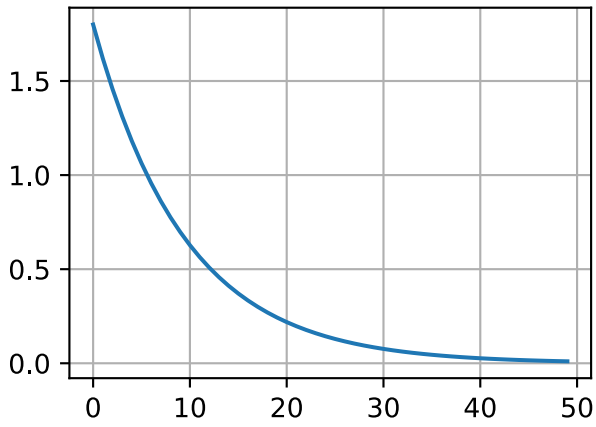
11.11.3 Politikalar

Tüm öğrenme oranı zamanlayıcılarını kapsayamasak da, aşağıda popüler politikalara kısa bir genel bakış vermeye çalışıyoruz. Yaygın seçenekler polinom sönme ve parçalı sabit zamanlamalardır. Bunun ötesinde, kosinüs öğrenme oranı zamanlamalarının bazı problemler üzerinde deneysel olarak iyi çalıştığı tespit edilmiştir. Son olarak, bazı problemlerde, büyük öğrenme oranları kullanmadan önce optimize ediciyi ısıtmak faydalıdır.

Çarpan Zamanlayıcı

Bir polinom sönmesine bir alternatif, $\eta_{t+1} \leftarrow \eta_t \cdot \alpha$ için $\alpha \in (0, 1)$ olan çarpımsal bir çözüm olabilir. Öğrenme oranının makul bir alt sınırın ötesinde sönmesini önlemek için güncelleme denklemi genellikle $\eta_{t+1} \leftarrow \max(\eta_{\min}, \eta_t \cdot \alpha)$ olarak değiştirilir.

```
class FactorScheduler:
    def __init__(self, factor=1, stop_factor_lr=1e-7, base_lr=0.1):
        self.factor = factor
        self.stop_factor_lr = stop_factor_lr
        self.base_lr = base_lr

    def __call__(self, num_update):
        self.base_lr = max(self.stop_factor_lr, self.base_lr * self.factor)
        return self.base_lr

scheduler = FactorScheduler(factor=0.9, stop_factor_lr=1e-2, base_lr=2.0)
d2l.plot(torch.arange(50), [scheduler(t) for t in range(50)])
```



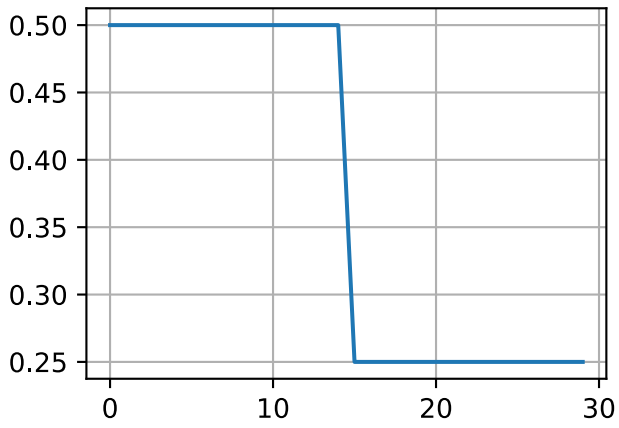
Bu, `lr_scheduler.FactorScheduler` nesnesi aracılığıyla MXNet'te yerleşik bir zamanlayıcı tarafından da gerçekleştirilebilir. Isınma süresi, ısınma modu (doğrusal veya sabit), istenen güncelleme

sayısı, vb. gibi birkaç parametre daha alır; ileriye gidersek yerleşik zamanlayıcıları uygun olarak kullanacağız ve yalnızca işlevselliklerini burada açıklayacağız. Gösterildiği gibi, gerekirse kendi zamanlayıcınızı oluşturmak oldukça basittir.

Çok Çarpanlı Zamanlayıcı

Derin ağları eğitmek için yaygın bir strateji, öğrenme oranını parçalı sabit tutmak ve sık sık belirli bir miktarda azaltmaktır. Diğer bir deyişle, bir dizi zaman verildiğinde hızın ne zaman düşürüleceğidir, örneğin $s = \{5, 10, 20\}$ ve $t \in s$ için $\eta_{t+1} \leftarrow \eta_t \cdot \alpha$'nın azalması. Değerlerin her adımda yarıya indirildiğini varsayarsak, bunu aşağıdaki gibi uygulayabiliriz.

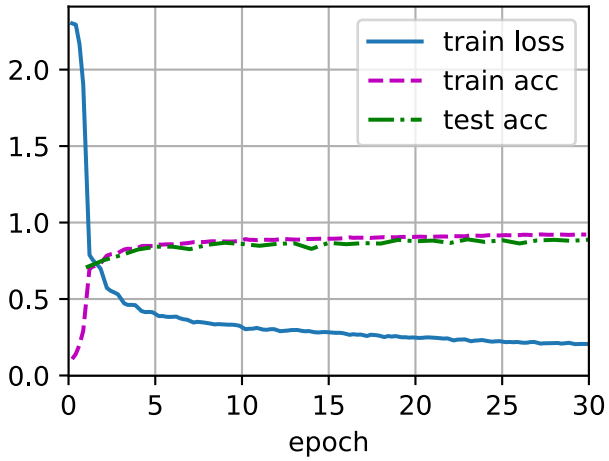
```
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr=0.5)
scheduler = lr_scheduler.MultiStepLR(trainer, milestones=[15, 30], gamma=0.5)

def get_lr(trainer, scheduler):
    lr = scheduler.get_last_lr()[0]
    trainer.step()
    scheduler.step()
    return lr

d2l.plot(torch.arange(num_epochs), [get_lr(trainer, scheduler)
                                    for t in range(num_epochs)])
```



Bu parçalı sabit öğrenme oranı zamanlamasının arkasındaki sezgi, ağırlık vektörlerinin dağılımı açısından sabit bir noktaya ulaşılana kadar optimizasyonun ilerlemesine izin vermesidir. O zaman (ve ancak o zaman) daha yüksek kaliteli bir vekil elde etmek için oranı iyi bir yerel minimuma düşürürüz. Aşağıdaki örnek, bunun nasıl biraz daha iyi çözümler üretebileceğini göstermektedir.

```
train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
      scheduler)
```

```
train loss 0.204, train acc 0.922, test acc 0.888
```



Kosinüs Zamanlayıcı

(Loshchilov and Hutter, 2016) tarafından oldukça şaşırtıcı bir sezgisel yöntem önerildi. Başlangıçta öğrenme oranını çok büyük ölçüde düşürmek istemeyebileceğimiz ve dahası, çözümü sonunda çok küçük bir öğrenme oranı kullanarak "ince ayarlamak" isteyebileceğimiz gözlemine dayanıyor. Bu, $t \in [0, T]$ aralığındaki öğrenme oranları için aşağıdaki işlevsel formla kosinüs benzeri bir zamanlama ile sonuçlanır.

$$\eta_t = \eta_T + \frac{\eta_0 - \eta_T}{2} (1 + \cos(\pi t/T)) \tag{11.11.1}$$

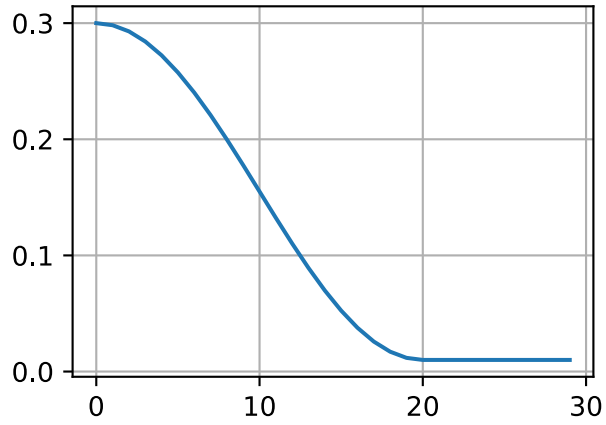
Burada η_0 ilk öğrenme oranı, η_T ise T zamanındaki hedef orandır. Ayrıca, $t > T$ için değeri tekrar artırmadan η_T'ya sabitleriz. Aşağıdaki örnekte, maksimum güncelleme adımını $T = 20$'ye ayarladık.

```
class CosineScheduler:
    def __init__(self, max_update, base_lr=0.01, final_lr=0,
                 warmup_steps=0, warmup_begin_lr=0):
        self.base_lr_orig = base_lr
        self.max_update = max_update
        self.final_lr = final_lr
        self.warmup_steps = warmup_steps
        self.warmup_begin_lr = warmup_begin_lr
        self.max_steps = self.max_update - self.warmup_steps

    def get_warmup_lr(self, epoch):
        increase = (self.base_lr_orig - self.warmup_begin_lr) \
                       * float(epoch) / float(self.warmup_steps)
        return self.warmup_begin_lr + increase

    def __call__(self, epoch):
        if epoch < self.warmup_steps:
            return self.get_warmup_lr(epoch)
        if epoch <= self.max_update:
            self.base_lr = self.final_lr + (
                self.base_lr_orig - self.final_lr) * (1 + math.cos(
                math.pi * (epoch - self.warmup_steps) / self.max_steps)) / 2
        return self.base_lr
```

(continues on next page)

(continued from previous page)

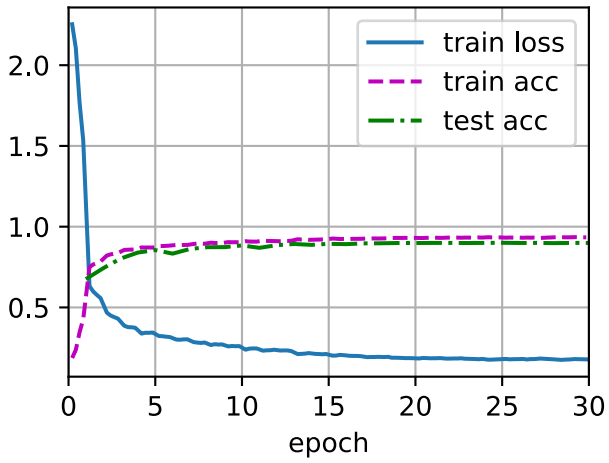
```
scheduler = CosineScheduler(max_update=20, base_lr=0.3, final_lr=0.01)
d2l.plot(torch.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```



Bilgisayarla görme bağlamında bu zamanlama gelişmiş sonuçlara yol aça*bilir*. Yine de, bu tür iyileştirmelerin garanti edilmediğini unutmayın (aşağıda görülebileceği gibi).

```
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr=0.3)
train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
      scheduler)
```

```
train loss 0.178, train acc 0.934, test acc 0.899
```

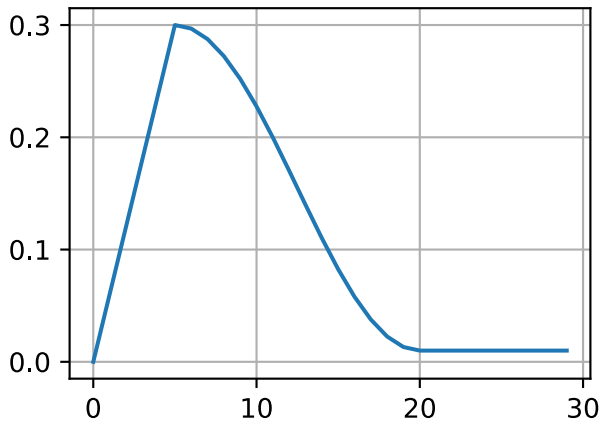


Isınma

Bazı durumlarda parametrelerin ilklenmesi, iyi bir çözümü garanti etmek için yeterli değildir. Bu, özellikle kararsız optimizasyon sorunlarına yol açabilecek bazı gelişmiş ağ tasarımları için bir sorundur. Başlangıçta ıraksamayı önlemek için yeterince küçük bir öğrenme oranı seçerek bunu ele alabiliriz. Ne yazık ki bu ilerlemenin yavaş olduğu anlamına gelir. Tersine, büyük bir öğrenme oranı başlangıçta ıraksamaya yol açar.

Bu ikilem için oldukça basit bir çözüm, öğrenme hızının başlangıçtaki maksimum değerine *arttığı* bir ısınma periyodu kullanmak ve hızı optimizasyon sürecinin sonuna kadar soğutmaktır. Basitlik için genellikle bu amaçla doğrusal bir artış kullanır. Bu, aşağıda belirtilen formda bir zamanlamaya yol açar.

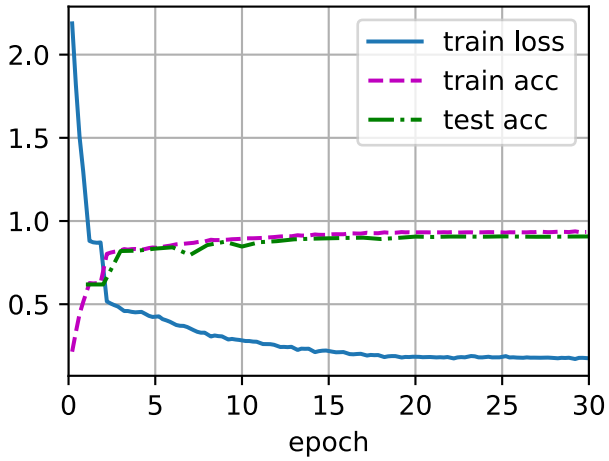
```
scheduler = CosineScheduler(20, warmup_steps=5, base_lr=0.3, final_lr=0.01)
d2l.plot(torch.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```



Ağın ilkin daha iyi yakınsadığına dikkat edin (özellikle ilk 5 dönemde performansı gözlemleyin).

```
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr=0.3)
train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
      scheduler)
```

```
train loss 0.177, train acc 0.934, test acc 0.907
```



Isınma herhangi bir zamanlayıcıya uygulanabilir (sadece kosinüs değil). Öğrenme oranı programları ve daha birçok deney hakkında daha ayrıntılı bir tartışma için ayrıca bkz. (Gotmare *et al.*, 2018). Özellikle, ısınma fazının çok derin ağlardaki parametrelerin sapma miktarını sınırladığını buldular. Bu sezgisel olarak mantıklıdır, çünkü ağın başlangıçta ilerleme kaydetmesi en çok zaman alan bölümlerinde rastgele ilkleme nedeniyle önemli farklılıklar bekleyebiliriz.

11.11.4 Özet

- Eğitim sırasında öğrenme oranının azaltılması, doğruluğun iyileştirilmesine ve (en şaşırtıcı şekilde) modelde azaltılmış aşırı öğrenmeye neden olabilir.
- İlerlemenin düzleştiği her durumda öğrenme hızının parçalı azalması pratikte etkilidir. Esasen bu, uygun bir çözüme verimli bir şekilde yakınlaşmamızı ve ancak daha sonra öğrenme oranını azaltarak parametrelerin doğal varyansını azaltmamızı sağlar.
- Kosinüs zamanlayıcıları bazı bilgisayarla görme problemleri için popülerdir. Bu tür bir zamanlayıcı ile ilgili ayrıntılar için bkz. GluonCV[144].
- Optimizasyondan önceki bir ısınma periyodu ıraksamayı önleyebilir.
- Optimizasyon, derin öğrenmede birden çok amaca hizmet eder. Eğitim hedefini en aza indirmenin yanı sıra, farklı optimizasyon algoritmaları ve öğrenme oranı zamanlama seçimleri, test kümesinde oldukça farklı miktarlarda genellemeye ve aşırı öğrenmeye (aynı miktarda eğitim hatası için) yol açabilir.

11.11.5 Alıştırmalar

1. Belirli bir sabit öğrenme hızı için optimizasyon davranışıyla deneyler yapın. Bu şekilde elde edebileceğiniz en iyi model nedir?
2. Öğrenme oranındaki düşüşün üssünü değiştirirseniz yakınsama nasıl değişir? Deneylerde kolaylık sağlamak için `PolyScheduler`'i kullanın.
3. Kosinüs zamanlayıcısını büyük bilgisayarla görme problemlerine uygulayın, örn. ImageNet'i eğitin. Diğer zamanlayıcılara göre performansı nasıl etkiler?

[144] http://gluon-cv.mxnet.io

4. Isınma ne kadar sürer?
5. Optimizasyon ve örneklemeyi ilişkilendirebilir misiniz? (Welling and Teh, 2011)'ten Rasgele Gradyan Langevin Dinamik sonuçlarını kullanarak başlayın.

Tartışmalar[145]

[145] https://discuss.d2l.ai/t/1080

12 | Hesaplamalı Performans

Derin öğrenmede, veri kümeleri ve modeller genellikle büyüktür, bu da ağır hesaplama içerir. Bu nedenle, hesaplama performansı çok önemlidir. Bu bölümde hesaplama performansını etkileyen önemli faktörler üzerinde durulacaktır: Buyuru programlama (imperative programming), sembolik programlama, eşzamansız hesaplama, otomatik paralelleştirme ve çoklu GPU hesaplama. Bu bölümü inceleyerek, örneğin, doğruluğu etkilemeden eğitim süresini azaltarak önceki bölümlerde uygulanan modellerin hesaplama performansını daha da artırabilirsiniz.

12.1 Derleyiciler ve Yorumlayıcılar

Şimdiye kadar, bu kitap bir programın durumunu değiştirmek için `print`, `+` ve `if` gibi ifadeleri kullanan buyuru programlamaya odaklanmıştır. Basit bir buyuru programın aşağıdaki örneğini göz önünde bulundurun.

```
def add(a, b):
    return a + b

def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g

print(fancy_func(1, 2, 3, 4))
```

```
10
```

Python bir *yorumlanan dil*dir. Yukarıdaki `fancy_func` işlevini değerlendirirken fonksiyonun gövdesini oluşturan işlemleri *sıra* ile gerçekleştirir. Yani, `e = add(a, b)`'yi değerlendirecek ve sonuçları e değişkeni olarak saklayacak ve böylece programın durumunu değiştirecektir. Sonraki iki ifade `f = add(c, d)` ve `g = add(e, f)` benzer şekilde yürütülecek, toplamalar gerçekleştirilecek ve sonuçları değişken olarak depolayacaktır. Fig. 12.1.1, veri akışını göstermektedir.

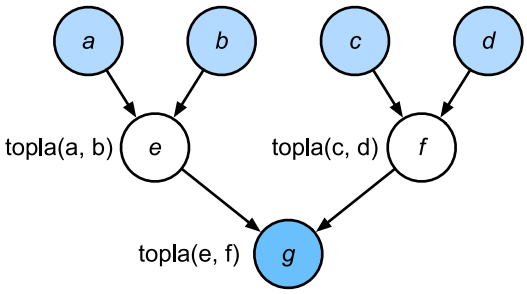


Fig. 12.1.1: Bir buyuru programda veri akışı.

Buyuru programlama her ne kadar uygun olsa da, verimsiz olabilir. Bir yandan, `add` işlevi `fancy_func` boyunca tekrar tekrar çağrılsa bile, Python üç işlevin çağrılarını tek tek yürütür. Bunlar, örneğin, bir GPU'da (veya birden fazla GPU'da) yürütülürse, Python yorumlayıcısından kaynaklanan yükü bezdirici hale gelebilir. Ayrıca, `fancy_func` içindeki tüm ifadeler yürütülünceye kadar `e` ve `f` değişken değerlerini kaydetmesi gerekecektir. Bunun nedeni, `e = add(a, b)` ve `f = add(c, d)` ifadeleri yürütüldükten sonra `e` ve `f` değişkenlerinin programın diğer bölümleri tarafından kullanılacağını bilmememizdir.

12.1.1 Sembolik Programlama

Hesaplamanın genellikle süreç tam olarak tanımlandıktan sonra gerçekleştirildiği alternatifi, *sembolik programlama*yı, göz önünde bulundurun. Bu strateji, Theano ve TensorFlow da dahil olmak üzere birden fazla derin öğrenme çerçevesi tarafından kullanılır (ikincisi buyuru uzantıları edindi). Genellikle aşağıdaki adımları içerir:

1. Yürütülecek işlemleri tanımlayın.
2. İşlemleri yürütülebilir bir programa derleyin.
3. Gerekli girdileri sağlayın ve yürütme için derlenmiş programı çağırın.

Bu, önemli miktarda optimizasyona izin verir. İlk olarak, birçok durumda Python yorumlayıcısını atlayabiliriz, böylece bir CPU üzerinde tek bir Python iş parçacığı ile eşleştirilmiş birden çok hızlı GPU'larda önemli hale gelebilecek bir performans darboğazını kaldırabiliriz. İkincisi, bir derleyici optimize edebilir ve yukarıdaki kodu `print((1 + 2) + (3 + 4))` veya hatta `print(10)` içine yeniden yazabilir. Bu, bir derleyici makine talimatlarına dönüştürmeden önce tam kodu görebildiği için mümkündür. Örneğin, artık bir değişken gerekmediğinde belleği serbest bırakabilir (veya asla tahsis etmez). Ya da kodu tamamen eşdeğer bir parçaya dönüştürebilir. Daha iyi bir fikir edinmek için aşağıdaki buyuru programlama benzetimini (sonuçta hepsi Python) düşünün.

```
def add_():
    return '''
def add(a, b):
    return a + b
'''

def fancy_func_():
    return '''
def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
```

(continues on next page)

(continued from previous page)

```
    g = add(e, f)
    return g
'''

def evoke_():
    return add_() + fancy_func_() + 'print(fancy_func(1, 2, 3, 4))'

prog = evoke_()
print(prog)
y = compile(prog, '', 'exec')
exec(y)
```

```
def add(a, b):
    return a + b

def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g
print(fancy_func(1, 2, 3, 4))
10
```

Buyuru (yorumlanan) programlama ve sembolik programlama arasındaki farklar şunlardır:

- Buyuru programlama daha kolaydır. Python'da buyuru programlama kullanıldığında, kodun çoğunluğu basittir ve yazması kolaydır. Ayrıca buyuru programlama kodunda hata ayıklamak daha kolaydır. Bunun nedeni, ilgili tüm ara değişken değerlerini elde etmenin ve yazdırmanın veya Python'un yerleşik hata ayıklama araçlarını kullanmanın daha kolay olmasıdır.
- Sembolik programlama daha verimli ve aktarması daha kolaydır. Sembolik programlama, derleme sırasında kodu optimize etmeyi kolaylaştırırken, programı Python'dan bağımsız bir formata aktarma yeteneğine de sahiptir. Bu, programın Python olmayan bir ortamda çalıştırılmasını sağlar, böylece Python yorumlayıcısı ile ilgili olası performans sorunlarından kaçınır.

12.1.2 Melez Programlama

Tarihsel olarak çoğu derin öğrenme çerçevesi buyuru veya sembolik yaklaşım arasında seçim yapar. Örneğin, Theano, TensorFlow (ilkinden esinlenerek), Keras ve CNTK modelleri sembolik olarak formüle eder. Tersine, Chainer ve PyTorch buyuru yaklaşımını benimsemektedir. TensorFlow 2.0 ve Keras'a sonraki düzeltmelerde buyuru modu eklendi.

Yukarıda belirtildiği gibi, PyTorch buyuru programlamaya dayanır ve dinamik hesaplama çizgeleri kullanır. Geliştiriciler, sembolik programlamanın taşınabilirliğini ve verimliliğini artırmak amacıyla, her iki programlama modelinin faydalarını birleştirmenin mümkün olup olmayacağını düşündü. Bu, kullanıcıların çoğu programı ürün düzeyinde bilgi işlem performansı ve konuşlandırma gerektiğinde çalıştırılmak üzere sembolik programlara dönüştürme yeteneğine sahipken, yalnızca buyuru programlama kullanarak geliştirmelerine ve hata ayıklamalarına olanak tanıyan bir meşale betiğine yol açtı.

12.1.3 `Sequential` Sınıfını Melezleme

Melezleştirmenin nasıl çalıştığını hissetmenin en kolay yolu, birden çok katmanlı derin ağları düşünmektir. Geleneksel olarak Python yorumlayıcısı, daha sonra bir CPU'ya veya GPU'ya iletilebilecek bir komut oluştururken tüm katmanlar için kodu yürütmesi gerekir. Tek (hızlı) bir bilgi işlem aygıtı için bu herhangi bir önemli soruna neden olmaz. Öte yandan, AWS P3dn.24xlarge örneği gibi gelişmiş bir 8 GPU'lu sunucusu kullanırsak Python tüm GPU'ları meşgul tutmaya çalışacaktır. Tek iş parçacıklı Python yorumlayıcısı burada darboğaz olur. `Sequential`'i `HybridSequential` ile değiştirerek kodun önemli bölümleri için bunu nasıl ele alabileceğimizi görelim. Basit bir MLP tanımlayarak başlıyoruz.

```
import torch
from torch import nn
from d2l import torch as d2l


# Ag fabrikasi
def get_net():
    net = nn.Sequential(nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, 2))
    return net

x = torch.randn(size=(1, 512))
net = get_net()
net(x)
```

```
tensor([[0.1160, 0.0584]], grad_fn=<AddmmBackward0>)
```

Modeli `torch.jit.script` işlevini kullanarak dönüştürerek, MLP'deki hesaplamayı derleyebiliyoruz ve optimize edebiliyoruz. Modelin hesaplama sonucu değişmeden kalır.

```
net = torch.jit.script(net)
net(x)
```

```
tensor([[0.1160, 0.0584]], grad_fn=<AddmmBackward0>)
```

Bu gerçek olamayacak kadar iyi görünüyor: Daha önce olduğu gibi aynı kodu yazın ve modeli `torch.jit.script`'ü kullanarak dönüştürün. Bu gerçekleştiğinde ağ optimize edilir (aşağıda performansı karşılaştıracağız).

Melezleştirme ile İvme

Derleme yoluyla elde edilen performans iyileştirmesini göstermek için `net(x)`'i melezleştirme öncesi ve sonrası değerlendirmek için gereken süreyi karşılaştırıyoruz. Önce bu zamanı ölçmek için bir sınıf tanımlayalım. Performansı ölçmek (ve geliştirmek) için yola çıktığımızda bu, bölüm boyunca kullanışlı olacaktır.

```
#@save
class Benchmark:
    """For measuring running time."""
    def __init__(self, description='Done'):
        self.description = description

    def __enter__(self):
        self.timer = d2l.Timer()
        return self

    def __exit__(self, *args):
        print(f'{self.description}: {self.timer.stop():.4f} sec')
```

Artık ağı bir kez meşale betikli ve bir kez de meşale betiği olmadan iki kez çağırabiliriz.

```
net = get_net()
with Benchmark('Without torchscript'):
    for i in range(1000): net(x)

net = torch.jit.script(net)
with Benchmark('With torchscript'):
    for i in range(1000): net(x)
```

```
Without torchscript: 1.5656 sec
With torchscript: 1.6088 sec
```

Yukarıdaki sonuçlarda görüldüğü gibi, `nn.Sequential` örneği `torch.jit.script` işlevi kullanılarak komut dosyası oluşturulduktan sonra, sembolik programlama kullanılarak bilgi işlem performansı artırılır.

Seri Hale Getirme

Modelleri derlemenin faydalarından biri, modeli ve parametrelerini diske seri hale getirebilmemizdir (kaydedebiliriz). Bu, bir modeli seçtiğiniz önişlemci dilinden bağımsız bir şekilde saklamamızı sağlar. Bu, eğitilmiş modelleri diğer cihazlara konuşlandırmamıza ve diğer önişlemci programlama dillerini kolayca kullanmamıza olanak tanır. Aynı zamanda kod genellikle buyuru programlamadan elde edilebileceğinden daha hızlıdır. `save` işlevini iş başında görelim.

```
net.save('my_mlp')
!ls -lh my_mlp*
```

```
-rw-rw-r-- 1 d2l-worker d2l-worker 652K Oct 17 22:47 my_mlp
```

12.1.4 Özet

- Buyuru programlama, kontrol akışı ve çok miktarda Python yazılım ekosistemi kullanma yeteneği ile kod yazmak mümkün olduğundan, yeni modeller tasarlamayı kolaylaştırır.
- Sembolik programlama, programı belirtmemizi ve çalıştırmadan önce derlemenizi gerektirir. Faydası arttırılmış performanstır.

12.1.5 Alıştırmalar

1. Önceki bölümlerde ilginizi çeken modelleri gözden geçirin. Onları yeniden uygulayarak hesaplama performanslarını artırabilir misiniz?

Tartışmalar[146]

12.2 Eşzamansız Hesaplama

Günümüzün bilgisayarları, birden fazla CPU çekirdeği (genellikle çekirdek başına birden fazla iş parçacığı), GPU başına birden çok işlem öğesi ve genellikle cihaz başına birden çok GPU'dan oluşan son derece paralel sistemlerdir. Kısacası, birçok farklı şeyi aynı anda, genellikle farklı cihazlarda işleyebiliriz. Ne yazık ki Python, en azından ekstra yardım almadan paralel ve eşzamansız kod yazmanın harika bir yolu değildir. Sonuçta, Python tek iş parçacıklıdır ve bunun gelecekte değişmesi pek olası değil. MXNet ve TensorFlow gibi derin öğrenme çerçeveleri, performansı artırmak için *eşzamansız programlama* modeli benimser, PyTorch ise Python'un kendi zamanlayıcısını kullanarak farklı bir performans değişimine yol açar. PyTorch için varsayılan olarak GPU işlemleri eşzamansızdır. GPU kullanan bir işlev çağırdığınızda, işlemler belirli bir aygıta sıralanır, ancak daha sonrasına kadar zorunlu olarak yürütülmez. Bu, CPU veya diğer GPU'lardaki işlemler de dahil olmak üzere paralel olarak daha fazla hesaplama yürütmemize olanak tanır.

Bu nedenle, eşzamansız programlamanın nasıl çalıştığını anlamak, hesaplama gereksinimlerini ve karşılıklı bağımlılıkları proaktif azaltarak daha verimli programlar geliştirmemize yardımcı olur. Bu, bellek yükünü azaltmamıza ve işlemci kullanımını artırmamıza olanak tanır.

```
import os
import subprocess
import numpy
import torch
from torch import nn
from d2l import torch as d2l
```

[146] https://discuss.d2l.ai/t/2490

12.2.1 Arka İşlemci Üzerinden Eşzamanlama

Isınmak için aşağıdaki basit örnek problemi göz önünde bulundurun: Rastgele bir matris oluşturmak ve çarpmak istiyoruz. Farkı görmek için bunu hem NumPy hem de PyTorch tensorunda yapalım. PyTorch `tensor`'ün bir GPU'da tanımlandığını unutmayın.

```
# GPU hesaplaması icin isinma
device = d2l.try_gpu()
a = torch.randn(size=(1000, 1000), device=device)
b = torch.mm(a, a)

with d2l.Benchmark('numpy'):
    for _ in range(10):
        a = numpy.random.normal(size=(1000, 1000))
        b = numpy.dot(a, a)

with d2l.Benchmark('torch'):
    for _ in range(10):
        a = torch.randn(size=(1000, 1000), device=device)
        b = torch.mm(a, a)
```

```
numpy: 1.4291 sec
torch: 0.0011 sec
```

PyTorch üzerinden yapılan kıyaslama çıktısı büyüklüğün kuvvetleri mertebesinde daha hızlıdır. NumPy nokta çarpımını CPU işlemcisinde yürütülür ve PyTorch matris çarpımını GPU'da yürütülür ve bu nedenle ikincisinin çok daha hızlı olması beklenir. Ama büyük zaman farkı, başka bir şeyin döndüğünü gösteriyor. Varsayılan olarak, PyTorch'ta GPU işlemleri eşzamansızdır. PyTorch'u geri dönmeden önce tüm hesaplamayı bitirmeye zorlamak daha önce neler olduğunu gösterir: Hesaplama arka işlemci tarafından yürütülür ve ön işlemci denetimi Python'a döndürür.

```
with d2l.Benchmark():
    for _ in range(10):
        a = torch.randn(size=(1000, 1000), device=device)
        b = torch.mm(a, a)
    torch.cuda.synchronize(device)
```

```
Done: 0.0032 sec
```

Genel olarak, PyTorch, örneğin Python aracılığıyla kullanıcılarla doğrudan etkileşimler için bir ön işlemciye ve sistem tarafından hesaplamayı gerçekleştirmek için kullanılan bir arka işlemciye sahiptir. Fig. 12.2.1 içinde gösterildiği gibi, kullanıcılar Python, R, Scala ve C++ gibi çeşitli ön işlemci dillerinde PyTorch programları yazabilir. Kullanılan ön işlemci programlama dili ne olursa olsun, PyTorch programlarının yürütülmesi öncelikle C++ uygulamalarının arka işlemcisinde gerçekleşir. Ön yüz dili tarafından verilen işlemler yürütme için arka işlemciye iletilir. Arka işlemci, sıraya alınmış görevleri sürekli olarak toplayan ve yürüten kendi iş parçacıklarını yönetir. Bunun çalışması için arka işlemcinin hesaplama çizgesindeki çeşitli adımlar arasındaki bağımlılıkları takip edebilmesi gerektiğini unutmayın. Bu nedenle, birbirine bağlı işlemleri paralel hale getirmek mümkün değildir.

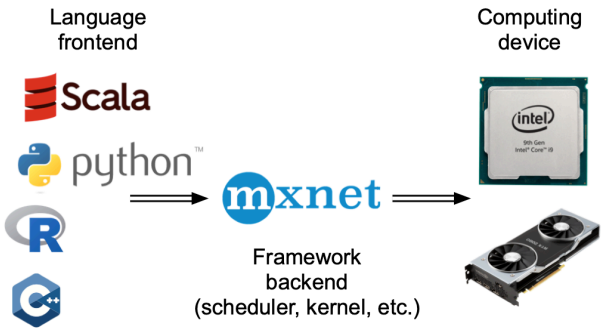


Fig. 12.2.1: Programlama dili ön işlemcileri ve derin öğrenme çerçevesi arka işlemcileri.

Bağımlılık çizgesini biraz daha iyi anlamak için başka bir basit örnek probleme bakalım.

```
x = torch.ones((1, 2), device=device)
y = torch.ones((1, 2), device=device)
z = x * y + 2
z
```

```
tensor([[3., 3.]], device='cuda:0')
```

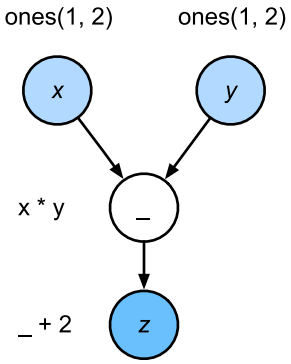


Fig. 12.2.2: Arka işlemci, hesaplama çizgesindeki çeşitli adımlar arasındaki bağımlılıkları izler.

Yukarıdaki kod parçacığı da Fig. 12.2.2 içinde gösterilmiştir. Python ön işlemci iş parçacığı ilk üç ifadeden birini çalıştırdığında, görevi arka işlemci kuyruğuna döndürür. Son ifadenin sonuçları *yazdırılmış* olması gerektiğinde, Python ön işlemci iş parçacığı C++ arka işlemci iş parçacığının z değişkenin sonucunu hesaplamayı tamamlamasını bekler. Bu tasarımın bir avantajı, Python ön işlemci iş parçacığının gerçek hesaplamaları gerçekleştirmesine gerek olmadığıdır. Böylece, Python'un performansından bağımsız olarak programın genel performansı üzerinde çok etkisi yoktur. Fig. 12.2.3, ön işlemci ve arka işlemcinin nasıl etkileşime girdiğini gösterir.

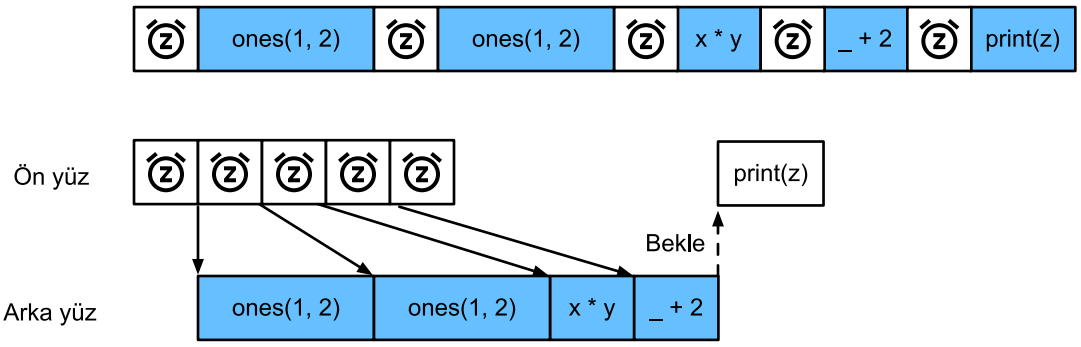


Fig. 12.2.3: Ön ve arka işlemci etkileşimleri.

12.2.2 Engeller ve Engelleyiciler

12.2.3 Hesaplamayı İyileştirme

12.2.4 Özet

- Derin öğrenme çerçeveleri, Python ön işlemcisini yürütme arka işlemcisinden ayırabilir. Bu, komutların arka işlemciye ve ilişkili paralellik içine hızlı eşzamansız olarak eklenmesine olanak tanır.
- Eşzamansızlık oldukça duyarlı bir ön işlemciye yol açar. Ancak, aşırı bellek tüketimine neden olabileceğinden görev kuyruğunu aşırı doldurmamak için dikkatli olun. Ön işlemciyi ve arka işlemciyi yaklaşık olarak senkronize ederken her minigrup için senkronize edilmesi önerilir.
- Çip satıcıları, derin öğrenmenin verimliliği hakkında çok daha ince tanımlanmış bir içgörü elde etmek için gelişmiş başarım çözümleme araçları sunar.

12.2.5 Alıştırmalar

1. CPU'da, bu bölümdeki aynı matris çarpma işlemlerini karşılaştırın. Arka işlemci üzerinden hala eşzamansızlık gözlemleyebilir misiniz?

Tartışmalar[147]

12.3 Otomatik Paralellik

Derin öğrenme çerçeveleri (örneğin, MXNet ve PyTorch) arka işlemcide otomatik olarak hesaplama çizgeleri oluşturur. Bir hesaplama çizgesi kullandığından, sistem tüm bağımlılıkların farkındadır ve hızı artırmak için paralel olarak birden çok birbirine bağımlı olmayan görevi seçici olarak yürütebilir. Örneğin, Section 12.2 içinde Fig. 12.2.2 bağımsız olarak iki değişkeni ilkler. Sonuç olarak sistem bunları paralel olarak yürütmeyi seçebilir.

Genellikle, tek bir operatör tüm CPU veya tek bir GPU'da tüm hesaplama kaynaklarını kullanır. Örneğin, `dot` operatörü, tek bir makinede birden fazla CPU işlemcisi olsa bile, tüm CPU'lardaki

[147] https://discuss.d2l.ai/t/2564

tüm çekirdekleri (ve iş parçacıklarını) kullanır. Aynısı tek bir GPU için de geçerlidir. Bu nedenle paralelleştirme, tek aygıtlı bilgisayarlar için o kadar yararlı değildir. Birden fazla cihazla işler daha önemli hale gelir. Paralelleştirme genellikle birden fazla GPU arasında en alakadar olmakla birlikte, yerel CPU'nun eklenmesi performansı biraz artıracaktır. Örneğin, bkz. (Hadjis *et al.*, 2016), bir GPU ve CPU'yu birleştiren bilgisayarla görme modellerini eğitmeye odaklanır. Otomatik olarak paralelleştirilen bir çerçevenin kolaylığı sayesinde aynı hedefi birkaç satır Python kodunda gerçekleştirebiliriz. Daha geniş bir şekilde, otomatik paralel hesaplama konusundaki tartışmamız, hem CPU'ları hem de GPU'ları kullanarak paralel hesaplamaya ve aynı zamanda hesaplama ve iletişimin paralelleşmesine odaklanmaktadır.

Bu bölümdeki deneyleri çalıştırmak için en az iki GPU'ya ihtiyacımız olduğunu unutmayın.

```
import torch
from d2l import torch as d2l
```

12.3.1 GPU'larda Paralel Hesaplama

Test etmek için bir referans iş yükü tanımlayarak başlayalım: Aşağıdaki `run` işlevi iki değişkene, `x_gpu1` ve `x_gpu2`, ayrılan verileri kullanarak seçeceğimiz cihazda 10 matris-matris çarpımı gerçekleştirir.

```
devices = d2l.try_all_gpus()
def run(x):
    return [x.mm(x) for _ in range(50)]

x_gpu1 = torch.rand(size=(4000, 4000), device=devices[0])
x_gpu2 = torch.rand(size=(4000, 4000), device=devices[1])
```

Şimdi işlevi verilere uyguluyoruz. Önbelleğe almanın sonuçlarda bir rol oynamadığından emin olmak için, ölçmeden önce her ikisine de tek bir geçiş yaparak cihazları ısıtırız. `torch.cuda.synchronize()`, CUDA cihazındaki tüm akışlardaki tüm çekirdeklerin tamamlaması için bekler. Senkronize etmemiz gereken bir `device` argümanı alır. Aygıt bağımsız değişkeni `None` (varsayılan) ise, `current_device()` tarafından verilen geçerli aygıtı kullanır.

```
run(x_gpu1)
run(x_gpu2)  # bütün cihazlari isindir
torch.cuda.synchronize(devices[0])
torch.cuda.synchronize(devices[1])

with d2l.Benchmark('GPU1 time'):
    run(x_gpu1)
    torch.cuda.synchronize(devices[0])

with d2l.Benchmark('GPU2 time'):
    run(x_gpu2)
    torch.cuda.synchronize(devices[1])
```

```
GPU1 time: 0.5000 sec
GPU2 time: 0.5136 sec
```

Her iki görev arasındaki `synchronize` ifadesini kaldırırsak, sistem her iki cihazda da otomatik olarak hesaplamayı paralel hale getirmekte serbesttir.

```
with d2l.Benchmark('GPU1 & GPU2'):
    run(x_gpu1)
    run(x_gpu2)
    torch.cuda.synchronize()
```

```
GPU1 & GPU2: 0.5039 sec
```

Yukarıdaki durumda, toplam yürütme süresi, parçalarının toplamından daha azdır, çünkü derin öğrenme çerçevesi, kullanıcı adına gelişmiş kod gerektirmeden her iki GPU cihazında hesaplamayı otomatik olarak planlar.

12.3.2 Paralel Hesaplama ve İletişim

Çoğu durumda, CPU ve GPU arasında veya farklı GPU'lar arasında, yani farklı cihazlar arasında veri taşımamız gerekir. Örneğin, bu durum, gradyanların birden fazla hızlandırıcı kart üzerinden toplamamız gereken dağıtılmış optimizasyonu gerçekleştirmek istediğimizde ortaya çıkar. Bunu GPU'da hesaplayarak benzetim yapalım ve sonuçları CPU'ya geri kopyalayalım.

```
def copy_to_cpu(x, non_blocking=False):
    return [y.to('cpu', non_blocking=non_blocking) for y in x]

with d2l.Benchmark('Run on GPU1'):
    y = run(x_gpu1)
    torch.cuda.synchronize()

with d2l.Benchmark('Copy to CPU'):
    y_cpu = copy_to_cpu(y)
    torch.cuda.synchronize()
```

```
Run on GPU1: 0.5059 sec
Copy to CPU: 2.3354 sec
```

Bu biraz verimsiz. Listenin geri kalanı hala hesaplanırken `y`'nin parçalarını CPU'ya kopyalamaya başlayabileceğimizi unutmayın. Bu durum, örneğin, bir minigrup işlemindeki (backprop) gradyanı hesapladığımızda ortaya çıkar. Bazı parametrelerin gradyanları diğerlerinden daha erken kullanılabilir olacaktır. Bu nedenle, GPU hala çalışırken PCI-Express veri yolu bant genişliğini kullanmaya başlamak ize avantaj sağlar. PyTorch'ta, `to()` ve `copy_()` gibi çeşitli işlevler, gereksiz olduğunda çağrı yapanın senkronizasyonu atlamasını sağlayan açık bir "non_blocking" argümanını kabul eder. `non_blocking=True` ayarı bu senaryoyu benzetmemize izin verir.

```
with d2l.Benchmark('Run on GPU1 and copy to CPU'):
    y = run(x_gpu1)
    y_cpu = copy_to_cpu(y, True)
    torch.cuda.synchronize()
```

```
Run on GPU1 and copy to CPU: 1.8978 sec
```

Her iki işlem için gereken toplam süre (beklendiği gibi) parçalarının toplamından daha azdır. Farklı bir kaynak kullandığı için bu görevin paralel hesaplamadan farklı olduğunu unutmayın:

CPU ve GPU'lar arasındaki veri yolu. Aslında, her iki cihazda da işlem yapabilir ve iletişim kurabiliriz, hepsini aynı anda. Yukarıda belirtildiği gibi, hesaplama ve iletişim arasında bir bağımlılık vardır: `y[i]` CPU'ya kopyalanmadan önce hesaplanmalıdır. Neyse ki, sistem toplam çalışma süresini azaltmak için `y[i]`'i hesaplarken `y[i-1]`'i kopyalayabilir.

Fig. 12.3.1 içinde gösterildiği gibi, bir CPU ve iki GPU üzerinde eğitim yaparken basit bir iki katmanlı MLP için hesaplama çizgesinin ve bağımlılıklarının bir gösterimi ile sonlandırıyoruz. Bundan kaynaklanan paralel programı manuel olarak planlamak oldukça acı verici olurdu. Eniyileme için çizge tabanlı bir hesaplama arka işlemcisine sahip olmanın avantajlı olduğu yer burasıdır.

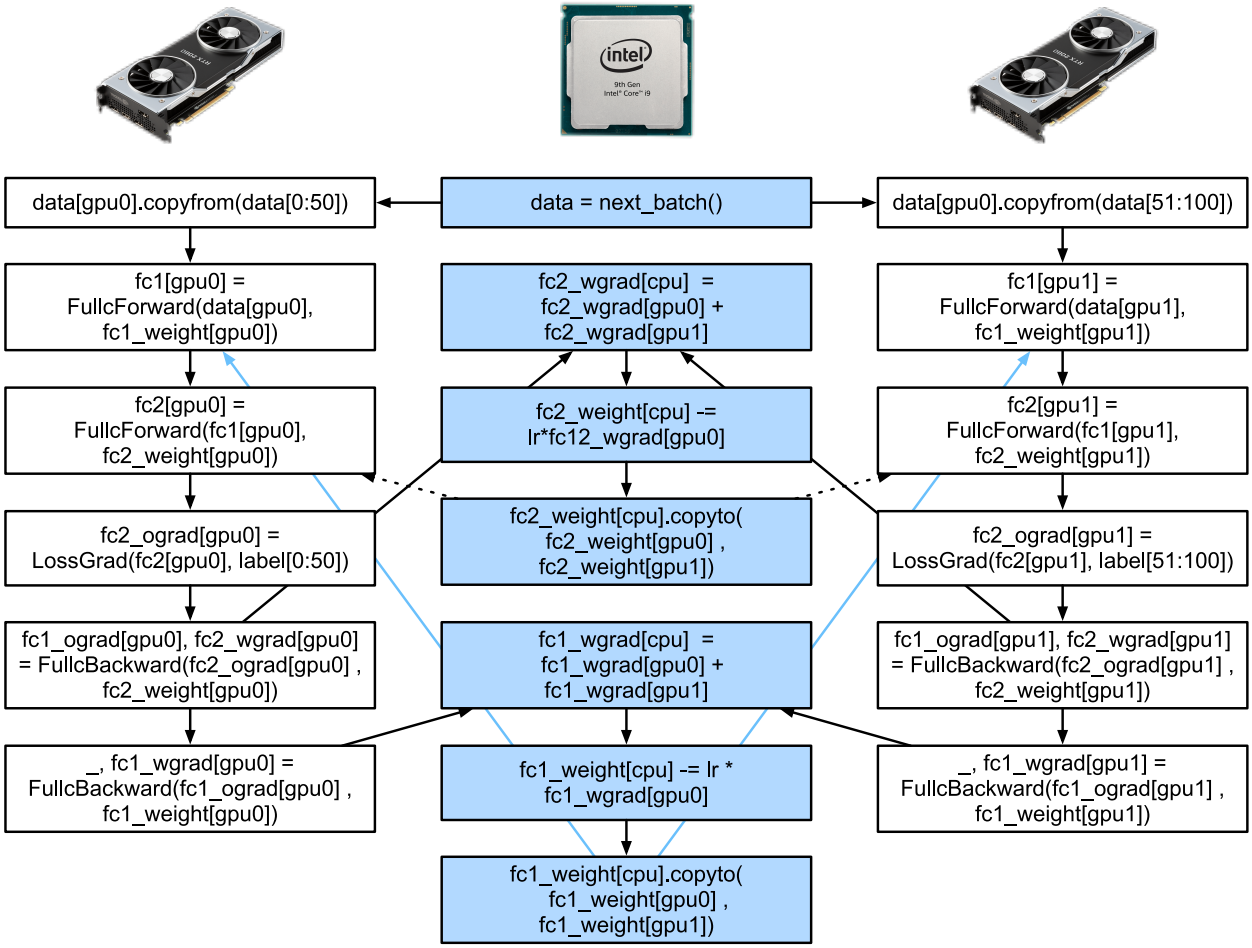


Fig. 12.3.1: Bir CPU ve iki GPU üzerindeki iki katmanlı MLP'nin hesaplama çizgesi ve bağımlılıkları.

12.3.3 Özet

- Modern sistemler, birden fazla GPU ve CPU gibi çeşitli cihazlara sahiptir. Paralel, eşzamansız olarak kullanılabilirler.
- Modern sistemler ayrıca PCI Express, depolama (genellikle katı hal sürücüleri veya ağlar aracılığıyla) ve ağ bant genişliği gibi iletişim için çeşitli kaynaklara sahiptir. En yüksek verimlilik için paralel olarak kullanılabilirler.
- Arka işlemci, otomatik paralel hesaplama ve iletişim yoluyla performansı artırabilir.

12.3.4 Alıştırmalar

1. Bu bölümde tanımlanan `run` işlevinde sekiz işlem gerçekleştirildi. Aralarında bağımlılık yoktur. Derin öğrenme çerçevesinin bunları otomatik olarak paralel yürüteceğini görmek için bir deney tasarlayın.
2. Tek bir operatörün iş yükü yeterince küçük olduğunda, paralelleştirmede tek bir CPU veya GPU'da bile yardımcı olabilir. Bunu doğrulamak için bir deney tasarlayın.
3. CPU'ları, GPU'ları ve her iki aygıt arasındaki iletişimde paralel hesaplamayı kullanan bir deney tasarlayın.
4. Kodunuzun verimli olduğunu doğrulamak için NVIDIA Nsight[148] gibi bir hata ayıklayıcısı kullanın.
5. Daha karmaşık veri bağımlılıkları içeren hesaplama görevleri tasarlayın ve performansı artırırken doğru sonuçları elde edip edemeyeceğinizi görmek için deneyler çalıştırın.

Tartışmalar[149]

12.4 Donanım

Yüksek performanslı sistemler oluşturmak, sorunun istatistiksel yönlerini yakalamak için algoritmaların ve modellerin iyi anlaşılmasını gerektirir. Aynı zamanda, temel donanım hakkında en azından bir nebze bilgi sahibi olmak da vazgeçilmezdir. Bu bölüm donanım ve sistem tasarımı üzerine yeterli bir dersin yerini almaz. Bunun yerine, bazı algoritmaların neden diğerlerinden daha verimli olduğunu ve nasıl iyi bir çıktı çıkarılacağını anlamak için bir başlangıç noktası görevi görebilir. İyi bir tasarım kolaylıkla büyük bir fark yaratabilir ve bu da bir ağı eğitebilmek (örneğin bir hafta içinde) ile hiç eğitememek (3 ay içinde, dolayısıyla son teslim tarihini kaçırarak) arasındaki farkı yaratabilir. Bilgisayarlara bakarak başlayacağız. Ardından CPU'lara ve GPU'lara daha dikkatli bakmak için yakınlaştıracağız. Son olarak, birden fazla bilgisayarın bir sunucu merkezinde veya bulutta nasıl bağlandığını gözden geçirmek için uzaklaştırıyoruz.

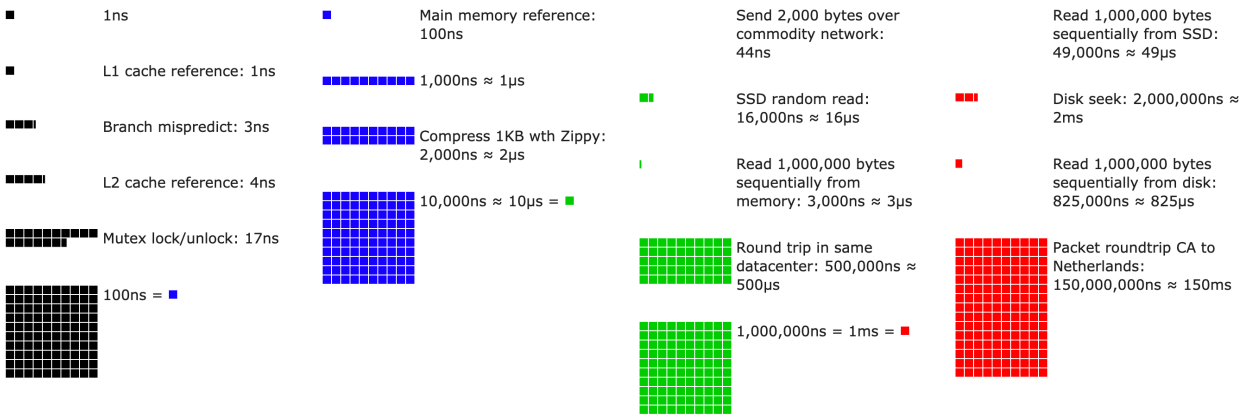


Fig. 12.4.1: Her programcının bilmesi gereken gecikme sayıları.

Sabırsız okuyucular Fig. 12.4.1 ile geçiştirebilirler. Colin Scott'ın interaktif yazı[150]sından alın-

[148] https://developer.nvidia.com/nsight-compute-2019_5
[149] https://discuss.d2l.ai/t/1681
[150] https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html

mıştır ve son on yıldaki ilerleme hakkında iyi bir genel bakış sunar. Orijinal sayılar Jeff Dean'in 2010 yılı Stanford konuşma[151]'sından kaynaklıdır. Aşağıdaki tartışma, bu sayıların gerekçelerini ve algoritmaları tasarlamada bize nasıl rehberlik edebilecekleri açıklamaktadır. Aşağıdaki tartışma çok yüksek düzeyde ve üstünkörüdür. Açıkça, gerçek bir kursun *ikamesi değildir*, bunun yerine sadece istatistiksel bir modelleyicinin uygun tasarım kararları vermesi için yeterli bilgiyi sağlamayı amaçlar. Bilgisayar mimarisine derinlemesine genel bir bakış için okuyucuyu ilgili çalışmaya (Hennessy and Patterson, 2011) ya da Arste Asanovic[152] tarafından verilen konuyla ilgili en yakın zamanlı derse yönlendiriyoruz.

12.4.1 Bilgisayarlar

Derin öğrenme araştırmacılarının ve uygulayıcılarının çoğu, makul miktarda belleğe, hesaplamaya, GPU gibi bir tür hızlandırıcıya veya bunların katlarına sahip bir bilgisayara erişebilir. Bir bilgisayar aşağıdaki temel bileşenlerden oluşur:

- Verdiğimiz programları yürütebilen (bir işletim sistemine ve diğer birçok şeyi çalıştırmaya ek olarak), tipik olarak 8 veya daha fazla çekirdekten oluşan bir işlemci (CPU olarak da adlandırılır).
- Ağırlık vektörleri ve etkinleştirmeleri gibi hesaplama sonuçlarını ve eğitim verilerini depolamak ve almak için bellek (RAM).
- 1 GB/s ila 100 GB/s arasında değişen hızlarda bir (bazen birden fazla) Ethernet ağ bağlantısı. Yüksek arka sunucularda daha gelişmiş ara bağlantılar bulunabilir.
- Sistemi bir veya daha fazla GPU'ya bağlamak için yüksek hızlı genişletme veri yolu (PCIe). Sunucular genellikle gelişmiş bir topolojiye bağlanan 8 hızlandırıcıya sahipken, masaüstü sistemleri kullanıcının bütçesine ve güç kaynağının boyutuna bağlı olarak 1 veya 2 taneye sahiptir.
- Birçok durumda PCIe veri yolu kullanılarak bağlanan manyetik sabit disk sürücüsü, katı hal sürücüsü gibi dayanıklı depolama. Eğitim verilerinin sisteme etkin bir şekilde aktarılmasını ve gerektiğinde ara kontrol noktalarının depolanmasını sağlar.

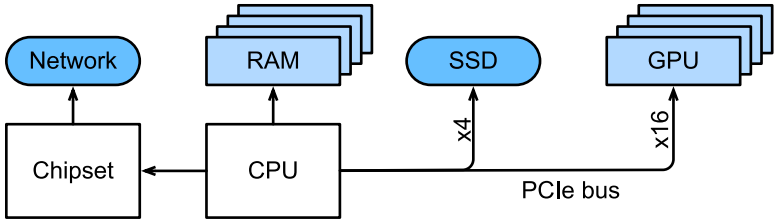


Fig. 12.4.2: Bilgisayar bileşenlerinin bağlanabilirliği.

Fig. 12.4.2 içinde belirttiği gibi, çoğu bileşen (ağ, GPU ve depolama) PCIe veri yolu boyunca CPU'ya bağlanır. Doğrudan CPU'ya bağlı birden fazla şeritten oluşur. Örneğin AMD'nin Threadripper 3'ü, her biri her iki yönde de 16 Gbit/s veri aktarımı özelliğine sahip 64 PCIe 4.0 şeridine sahiptir. Bellek, toplam 100 GB/s'ye kadar bant genişliği ile doğrudan CPU'ya bağlanır.

Bir bilgisayarda kod çalıştırdığımızda, verileri işlemcilere (CPU veya GPU'lar) karıştırmamız, hesaplama yapmamız ve ardından sonuçları işlemciden RAM'e ve dayanıklı depolamaya geri taşımamız gerekir. Bu nedenle, iyi bir performans elde etmek için sistemlerden herhangi biri büyük

[151] https://static.googleusercontent.com/media/research.google.com/en//people/jeff/Stanford-DL-Nov-2010.pdf
[152] http://inst.eecs.berkeley.edu/~cs152/sp19/

bir darboğaz haline gelmeden bunun sorunsuz bir şekilde çalıştığından emin olmalıyız. Örneğin, imgeleri yeterince hızlı yükleyemezsek işlemcinin yapacak hiçbir işi olmaz. Aynı şekilde, matrisleri CPU'ya (veya GPU'ya) yeterince hızlı taşıyamazsak, işleme öğeleri boşta kalacaktır. Son olarak, ağ üzerinden birden fazla bilgisayarı senkronize etmek istiyorsak, bu, hesaplamayı yavaşlatmamalıdır. Bir seçenek iletişimi ve hesaplamayı aralıklarla bağlamaktır. Çeşitli bileşenlere daha ayrıntılı bir göz atalım.

12.4.2 Bellek

En temelinde bellek kolayca erişilebilir olması gereken verileri depolamak için kullanılır. Günümüzde CPU RAM'i genellikle modül başına 20—25 GB/s bant genişliği sunan DDR4[153] çeşididir. Her modülün 64 bit genişliğinde bir veri yolu vardır. Genellikle bellek modülleri çiftleri birden çok kanala izin vermek için kullanılır. CPU'lar 2 ile 4 arasında bellek kanallarına sahiptir, yani 40 GB/s ile 100 GB/s arasında en yüksek bellek bant genişliğine sahiptirler. Genellikle kanal başına iki küme vardır. Örneğin AMD'nin Zen 3 Threadripper'ı 8 yuvasına sahiptir.

Bu sayılar etkileyici olsa da, aslında, hikayenin sadece bir kısmını anlatıyorlar. Bellekten bir bölümü okumak istediğimizde, önce bellek modülüne bilginin nerede bulunabileceğini söylemeliyiz. Yani, önce RAM'e *adresi* göndermemiz gerekiyor. Bu tamamlandıktan sonra sadece tek bir 64 bit kayıt veya uzun bir kayıt dizisini okumayı seçebiliriz. İkincisine *çoğuşma* denir. Özetle, belleğe bir adres göndermek ve aktarımı ayarlamak yaklaşık 100 ns alır (ayrıntılar kullanılan bellek yongalarının belirli zamanlama katsayılarına bağlıdır), sonraki her aktarım sadece 0.2 ns alır. Kısacası, ilk okuma, müteakip olanlardan 500 kat daha pahalıdır! Saniyede 10.000.000 adede kadar rasgele okuma yapabileceğimizi unutmayın. Bu bize, mümkün olduğunca rastgele bellek erişiminden kaçınmamızı ve bunun yerine seri okuma (ve yazma) kullanmamızı önerir.

Birden çok *küme* olduğunu dikkate aldığımızda konular biraz daha karmaşıktır. Her küme hafızayı büyük ölçüde bağımsız olarak okuyabilir. Bunun iki anlamı var. Bir yandan, rasgele okumaların etkin sayısı, belleğe eşit olarak yayılmaları koşuluyla 4 kata kadar daha yüksektir. Ayrıca çoğuşmalar da 4 kat daha hızlı olduğundan rastgele okuma yapmanın hala kötü bir fikir olduğu anlamına gelir. Öte yandan, 64 bitlik sınırlara bellek hizalaması nedeniyle, herhangi bir veri yapısını aynı sınırlarla hizalamak iyi bir fikirdir. Derleyiciler, uygun bayraklar ayarlandığında, bunu hemen hemen otomatik olarak[154] yapar. Meraklı okuyucular, Zeshan Chishti[155]'ninki gibi DRAM'ler üzerine bir dersi incelemeye teşvik edilir.

GPU belleği, CPU'lardan çok daha fazla işlem öğesine sahip olduklarından daha yüksek bant genişliği gereksinimlerine tabidir. Bunları ele almak için genel olarak iki seçenek vardır. Birincisi, bellek veri yolunu önemli ölçüde daha geniş yapmaktır. Örneğin, NVIDIA'nın RTX 2080 Ti'si 352 bit genişliğinde bir veri yoluna sahiptir. Bu, aynı anda çok daha fazla bilgi aktarılmasına olanak tanır. İkincisi, GPU'lar belirli yüksek performanslı bellek kullanır. NVIDIA'nın RTX ve Titan serisi gibi tüketici sınıfı cihazlar genellikle 500 GB/s'nin üzerinde toplam bant genişliğine sahip GDDR6[156] yongaları kullanır. Bir alternatif HBM (yüksek bant genişlikli bellek) modülleri kullanmaktır. Çok farklı bir arayüz kullanırlar ve özel bir silikon yonga levhası üzerindeki GPU'lara doğrudan bağlanırlar. Bu onları çok pahalı hale getirir ve bunların kullanımı genellikle NVIDIA Volta V100 serisi hızlandırıcılar gibi üst düzey sunucu yongaları ile sınırlıdır. Oldukça şaşırtıcı olmayan bir şekilde, GPU belleği, yüksek maliyeti nedeniyle CPU belleğinden genellikle *çok daha* küçüktür. Amaçlarımız için, performans özellikleri genel olarak benzer, sadece çok daha hızlıdır.

[153] https://en.wikipedia.org/wiki/DDR4_SDRAM
[154] https://en.wikipedia.org/wiki/Data_structure_alignment
[155] http://web.cecs.pdx.edu/~zeshan/ece585_lec5.pdf
[156] https://en.wikipedia.org/wiki/GDDR6_SDRAM

Bu kitabın amacı için detayları güvenle görmezden gelebiliriz. Sadece yüksek verim için GPU çekirdeklerini ayarlarken önemlidir.

12.4.3 Depolama

RAM'in bazı temel özelliklerinin *bant genişliği* ve *gecikme* olduğunu gördük. Aynı şey depolama cihazları için de geçerlidir, sadece farklılıklar daha da aşırı olabilir.

Sabit Disk Sürücüleri

Sabit disk sürücüleri (HDD'ler) yarım asırdan uzun süredir kullanılmaktadır. Özetle, herhangi bir parçada okumak veya yazmak için yerleştirilebilen kafaları olan bir dizi döner tabla içerirler. Üst düzey diskler 9 tabak üzerinde 16 TB'a kadar tutar. HDD'lerin en önemli avantajlarından biri, nispeten ucuz olmalarıdır. Birçok dezavantajlarından biri, tipik olarak feci arıza modları ve nispeten yüksek okuma gecikmeleridir.

İkincisini anlamak için, HDD'lerin yaklaşık 7.200 RPM'de (dakikadaki devir) döndüklerini göz önünde bulundurun. Eğer çok daha hızlı olsalardı tabaklara uygulanan merkezkaç kuvveti nedeniyle parçalanırlardı. Diskteki belirli bir sektöre erişmek söz konusu olduğunda bunun büyük bir dezavantajı vardır: Plaka pozisyonunda dönene kadar beklememiz gerekir (kafaları hareket ettirebiliriz, ancak gerçek diskleri hızlandıramayız). Bu nedenle, istenen veriler bulunana kadar 8 ms sürebilir. Bunun ortak bir yolu, HDD'lerin yaklaşık 100 IOP'de (saniyedeki girdi/çıktı işlemleri) çalışabileceğini söylemektir. Bu sayı esasen son yirmi yıldır değişmeden kaldı. Daha da kötüsü, bant genişliğini arttırmak eşit derecede zordur (100—200 MB/s sırasındadır). Sonuçta, her kafa bir bit parçasını okur, bu nedenle bit hızı sadece bilgi yoğunluğunun karekökü ile ölçeklenir. Sonuç olarak, HDD'ler hızla çok büyük veri kümeleri için arşiv depolamaya ve düşük dereceli depolamaya indirgeniyor.

Katı Hal Sürücüleri

Katı hal sürücüleri (SSD'ler) bilgileri kalıcı olarak depolamak için flash bellek kullanır. Bu, depolanan kayıtlara *çok daha hızlı* erişim sağlar. Modern SSD'ler 100.000 ila 500.000 IOP'de, yani HDD'lerden 3. derece üs kuvveti kadar daha hızlı çalışabilir. Ayrıca, bant genişliği 1—-3 GB/s'ye, yani HDD'lerden çok yukarı kademe bir hıza ulaşabilir. Bu gelişmeler kulağa gerçek olamayacak kadar iyi geliyor. Gerçekten de, SSD'ler tasarlanma şekli nedeniyle aşağıdaki uyarılarla birlikte gelirler.

- SSD'ler bilgileri bloklar halinde depolar (256 KB veya daha büyük). Sadece bir bütün olarak yazılabilirler, bu da kayda değer bir zaman alır. Sonuç olarak SSD'de bit düzeyinde rastgele yazmalar çok düşük performansa sahiptir. Aynı şekilde, bloğun okunması, silinmesi ve ardından yeni bilgilerle yeniden yazılması gerektiğinden, genel olarak veri yazmak önemli zaman alır. Şimdiye kadar SSD denetleyicileri ve bellenim (firmware) bunu azaltmak için algoritmalar geliştirdi. Bununla birlikte, yazmalar özellikle QLC (dörtlük düzeyde hücre) SSD'ler için çok daha yavaş olabilir. İyileştirilmiş performansın anahtarı, bir *işlem kuyruğu* tutmak, mümkünse okumaları ve büyük bloklar halinde yazmayı tercih etmektir.
- SSD'lerdeki bellek hücreleri nispeten hızlı bir şekilde yıpranır (genellikle birkaç bin yazmadan sonra). Aşınma seviyesi koruma algoritmaları bozulmayı birçok hücreye yayabilir. Yani, dosyaları takas etmek veya büyük günlük dosyası toplamaları için SSD'lerin kullanılması önerilmez.

- Son olarak, bant genişliğindeki büyük artış, bilgisayar tasarımcılarını doğrudan PCIe veri yoluna SSD'leri takmaya zorladı. Bunu işleyebilen NVMe (Non Volatile Memory enhanced) olarak adlandırılan sürücüler, 4 adede kadar PCIe şeridini kullanabilir. Bu, PCIe 4.0 üzerinde 8 GB/s'ye kadar tutarındadır.

Bulut Depolama

Bulut depolama, yapılandırılabilir bir performans aralığı sağlar. Yani, sanal makinelere depolama ataması, kullanıcılar tarafından seçildiği üzere hem miktar hem de hız açısından dinamiktir. Kullanıcıların, gecikme süresi çok yüksek olduğunda, örneğin birçok küçük kaydın olduğu eğitim sırasında, sağlanan IOP sayısını artırmalarını öneririz.

12.4.4 CPU'lar

Merkezi işlem birimleri (CPU'lar), herhangi bir bilgisayarın en önemli parçasıdır. Bir dizi temel bileşenden oluşurlar: Makine kodunu yürütebilen *işlemci çekirdekleri*, bunları birbirine bağlayan bir *veriyolu* (belirli topoloji işlemci modelleri, nesiller ve satıcılar arasında önemli ölçüde farklılık gösterir) ve ana bellekten okumalarla mümkün olandan daha yüksek bant genişliği ve daha düşük gecikmeli bellek erişimine izin vermek için *önbellekler*. Son olarak, neredeyse tüm modern CPU'lar, medya işleme ve makine öğrenmesinde yaygın oldukları için yüksek performanslı doğrusal cebire ve evrişimlere yardımcı olmak için *vektör işleme birimleri* içerir.

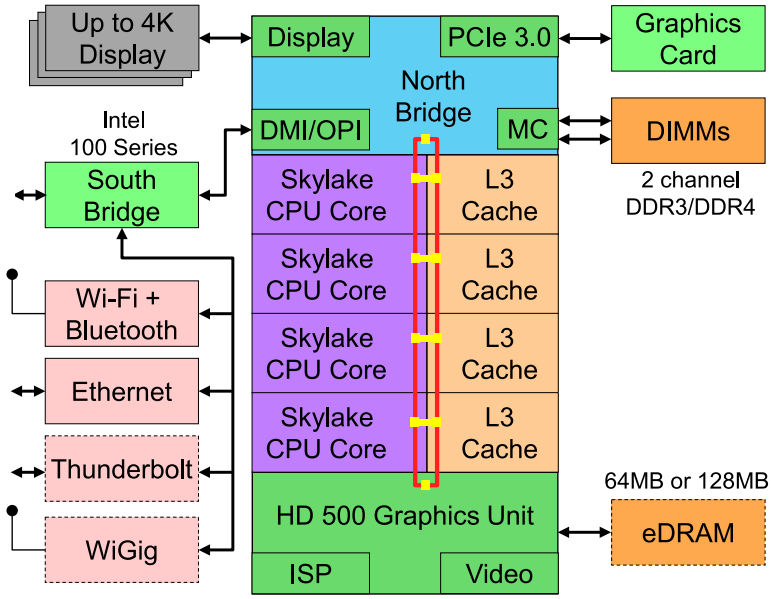


Fig. 12.4.3: Tüketiciler için Intel Skylake dört çekirdekli CPU.

Fig. 12.4.3 Intel Skylake tüketici sınıfı dört çekirdekli CPU'yu tasvir eder. Tümleşik bir GPU, önbellekleri ve dört çekirdeği bağlayan bir çember veriyolu vardır. Ethernet, WiFi, Bluetooth, SSD denetleyicisi ve USB gibi çevre birimleri yonga setinin bir parçasıdır ya da CPU'ya doğrudan takılıdır (PCIe).

Mikromimari

İşlemci çekirdeklerinin her biri oldukça gelişmiş bir bileşen kümesinden oluşur. Ayrıntılar nesiller ve satıcılar arasında farklılık gösterse de, temel işlevsellik oldukça standarttır. Ön işlemci talimatları yükler ve hangi yolun alınacağını tahmin etmeye çalışır (örn. kontrol akışı için). Talimatlar daha sonra birleştirici kodundan mikro talimatlara çevrilir. Birleştirici kodu genellikle bir işlemcinin yürüttüğü en düşük düzey kod değildir. Bunun yerine, karmaşık talimatlar daha alt düzey işlemler kümesine çözülebilir. Bunlar daha sonra gerçek yürütme çekirdeği tarafından işlenir. Genellikle sonraki, aynı anda birçok işlemi gerçekleştirebilir. Örneğin, Fig. 12.4.4 içindeki ARM Cortex A77 çekirdeği aynı anda 8 adede kadar işlem gerçekleştirebilir.

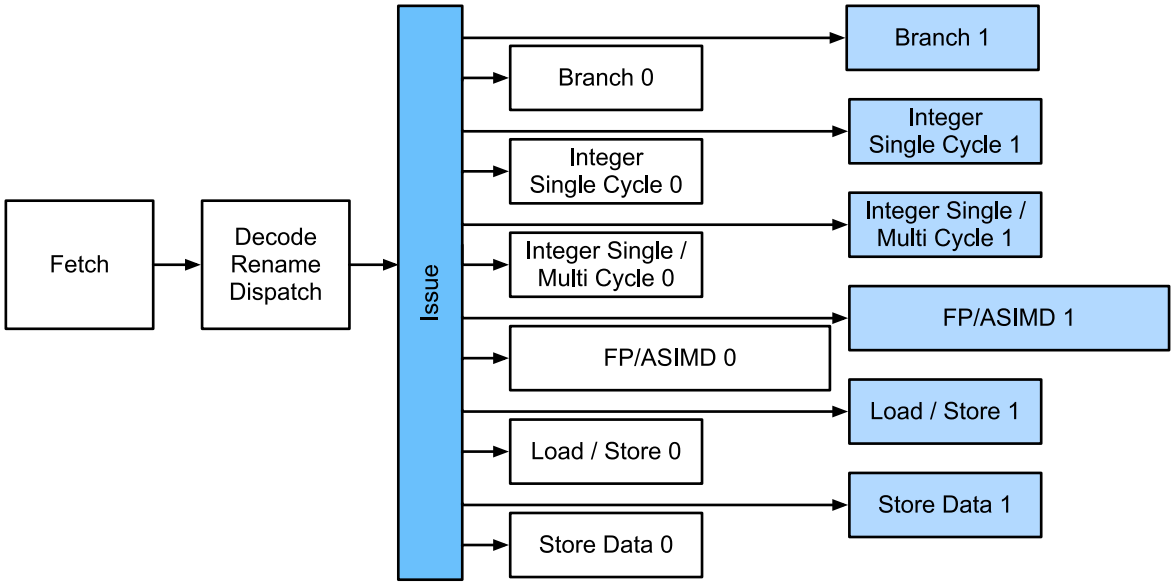


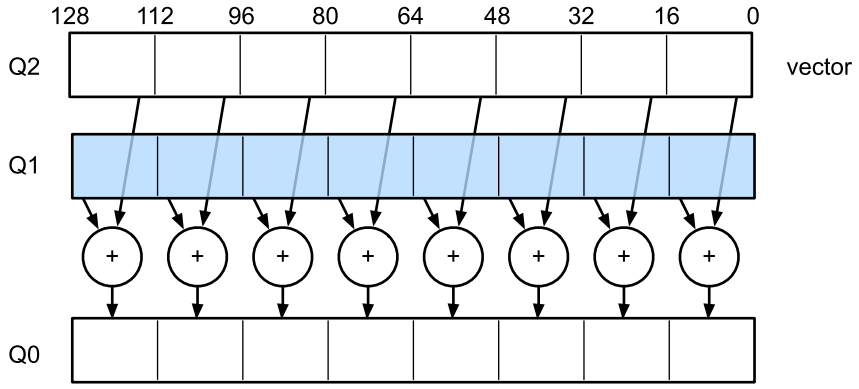
Fig. 12.4.4: ARM Cortex A77 Mikromimarisi.

Bu, verimli programların bağımsız olarak gerçekleştirilebilmesi şartıyla saat döngüsü başına birden fazla talimat gerçekleştirebileceği anlamına gelir. Tüm birimler eşit oluşturulmaz. Bazıları tamsayı talimatlarında uzmanlaşırken diğerleri kayan virgüllü sayı performansı için optimize edilmiştir. Verimi artırmak için işlemci aynı anda bir dallanma talimatında birden fazla kod yolunu izleyebilir ve ardından izlenmeyen dalların sonuçlarını atabilir. Bu nedenle dallanma tahmin birimleri (ön uçta) sadece en umut verici yolların takip edileceği şekilde önemlidir.

Vektörleştirme

Derin öğrenme aşırı derecede hesaplamaya açtır. Bu nedenle, işlemcileri makine öğrenmesine uygun hale getirmek için, bir saat döngüsünde birçok işlemi gerçekleştirmesi gerekir. Bu vektör birimleri vasıtasıyla elde edilir. Farklı isimleri vardır: ARM'de bunlara NEON denir, x86'da bunlara (yeni nesil) AVX2[157] birimleri denir. Ortak bir yönü, SIMD (tek komut çoklu veri) işlemlerini gerçekleştirebilmeleridir. Section 12.4.4, ARM'de bir saat döngüsünde 8 kısa tamsayının nasıl eklenebileceğini gösterir.

[157] https://en.wikipedia.org/wiki/Advanced_Vector_Extensions



Mimari seçeneklere bağlı olarak, bu tür yazmaçlar 512 bit'e kadar uzunluğa sahiptir ve 64 çifte kadar sayının katışımına izin verir. Örneğin, iki sayıyı çarpıp üçüncüye ekliyor olabiliriz, ki bu da kaynaşmış çarpma-toplama olarak da bilinir. Intel'in OpenVino[158], sunucu sınıfı CPU'larda derin öğrenme için hatrı sayılır verim elde ederken bunları kullanır. Yine de, bu sayının GPU'ların başarabilecekleri tarafından tamamen gölgede kaldığını unutmayın. Örneğin, NVIDIA'nın RTX 2080 Ti'sinde her biri herhangi bir zamanda böyle bir işlemi işleyebilen 4.352 CUDA çekirdeği vardır.

Önbellek

Şu durumu göz önünde bulundurun: 2 GHz frekansında çalışan, yukarıdaki Fig. 12.4.3 içinde gösterildiği gibi 4 çekirdekli mütevazı bir CPU çekirdeğimiz var. Ayrıca, bir IPC (saat başına talimatlar) sayısı 1 olduğunu ve birimlerin 256-bit genişliği etkin olan AVX2 olduğunu varsayalım. Ayrıca AVX2 işlemleri için kullanılan yazmaçlardan en az birinin bellekten alınması gerektiğini varsayalım. Bu, CPU'nun saat döngüsü başına $4 \times 256\,\text{bit} = 128\,\text{bayt}$ veri tükettiği anlamına gelir. Saniyede işlemciye $2 \times 10^9 \times 128 = 256 \times 10^9$ bayt aktaramazsak işleme elemanları boşta kalacak. Ne yazık ki böyle bir çipin bellek arayüzü sadece 20—-40 GB/s veri aktarımını destekler, yani, büyüklük bir kademe daha azdır. Telafisi, mümkün olduğunca bellekten *yeni* veri yüklemekten kaçınmak ve bunun yerine yerel olarak CPU'da önbelleğe almaktır. Önbelleklerin kullanışlı olduğu yer burasıdır. Genellikle aşağıdaki adlar veya kavramlar kullanılır:

- **Yazmaçlar** kesinlikle önbelleğin bir parçası değildir. Talimatların hazırlanmasına yardımcı olurlar. Yani, CPU yazmaçları bir CPU'nun herhangi bir gecikme cezası olmadan saat hızında erişebileceği bellek konumlarıdır. CPU'ların onlarca yazmacı vardır. Yazmaçları verimli bir şekilde kullanmak derleyiciye (veya programcıya) bağlıdır. Örneğin C programlama dili `register` anahtar sözcüğüne sahiptir.
- **L1 önbelleği** yüksek bellek bant genişliği gereksinimlerine karşı ilk savunma hattıdır. L1 önbellekleri küçüktür (tipik boyutlar 32—64 KB olabilir) ve genellikle veri ve talimatlar önbelleği diye bölünür. Veriler L1 önbelleğinde bulunduğunda, erişim çok hızlıdır. Orada bulunamazsa, arama önbellek hiyerarşisinde ilerler.
- **L2 önbelleği** bir sonraki duraktır. Mimari tasarımına ve işlemci boyutuna bağlı olarak özelleştirilmiş olabilirler. Yalnızca belirli bir çekirdek tarafından erişilebilir veya birden fazla çekirdek arasında paylaşılabilir. L2 önbellekler daha büyüktür (çekirdek başına genellikle 256—512 KB) ve L1'den daha yavaştır. Ayrıca, L2'deki bir şeye erişmek için öncelikle verilerin L1'de olmadığını kontrol etmemiz gerekir; bu da az miktarda ilave gecikme ekler.

[158] https://01.org/openvinotoolkit

- **L3 önbelleği** birden fazla çekirdek arasında paylaşılır ve oldukça büyük olabilir. AMD'nin Epyc 3 sunucu CPU'ları, birden fazla yongaya yayılmış 256 MB önbelleğe sahiptir. Daha yaygın olarak 4-8 MB aralığındadır.

Hangi bellek öğelerinin ihtiyaç duyulacağını tahmin etmek, çip tasarımında en önemli optimizasyon parametrelerinden biridir. Örneğin, çoğu önbelleğe alma algoritması geriye değil *ileriye okumaya* çalışacağından, belleği *ileri* yönde hareket ettirmeniz önerilir. Benzer şekilde, bellek erişim modellerini yerelde tutmak, performansı iyileştirmenin iyi bir yoludur.

Ön bellek eklemek çift kenarlı bir kılıçtır. Bir yandan işlemci çekirdeklerinin veriden yoksunluk çekmemesini sağlarlar. Aynı zamanda, işlemci gücünü artırmak için harcanabilecek alanı kullanarak yonga boyutunu da artırırlar. Dahası, *önbellek ıskalamaları* pahalı olabilir. Fig. 12.4.5 içinde gösterildiği gibi en kötü durum senaryosunu (*yanlış paylaşım*) düşünün. İşlemci 1'deki bir iş parçacığı verileri istediğinde, işlemci 0'da bir bellek konumu önbelleğe alınır. Onu elde etmek için, işlemcinin yaptığı işi durdurması, bilgileri ana belleğe geri yazması ve ardından işlemci 1'in bellekten okumasına izin vermesi gerekir. Bu işlem sırasında her iki işlemci de bekler. Oldukça potansiyel olarak, bu tür bir kod, verimli bir tek işlemcili uygulama ile karşılaştırıldığında, birden çok işlemcide *daha yavaş* çalışır. Bu, önbellek boyutlarının (fiziksel boyutlarının yanı sıra) pratik bir sınırının bulunmasının daha başka bir nedenidir.

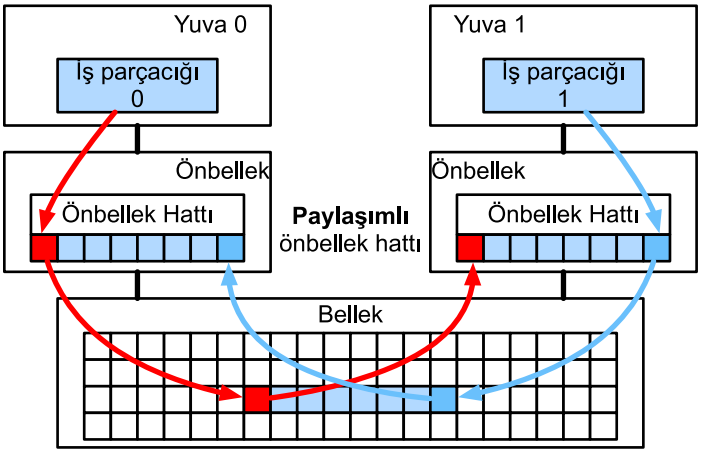


Fig. 12.4.5: Yanlış paylaşım (İmge Intel'in izniyle verilmektedir).

12.4.5 GPU'lar ve Diğer Hızlandırıcılar

Derin öğrenmenin GPU'lar olmadan başarılı olamayacağını iddia etmek abartı değildir. Aynı şekilde, GPU üreticilerinin servetlerinin derin öğrenme sayesinde önemli ölçüde arttığını iddia etmek oldukça makuldur. Donanım ve algoritmaların bu ortak evrimi, daha iyi ya da kötü derin öğrenmenin tercih edilen istatistiksel modelleme paradigması olduğu bir duruma yol açmıştır. Bu nedenle, GPU'ların ve TPU (Jouppi *et al.*, 2017) gibi ilgili hızlandırıcıların belirli faydalarını anlamak önemlidir.

Dikkat edilmesi gereken, uygulamada sıklıkla yapılan bir ayrımdır: Hızlandırıcılar ya eğitim ya da çıkarım için optimize edilmiştir. İkincisi için sadece bir ağdaki ileri yaymayı hesaplamamız gerekiyor. Geriye yayma için ara veri depolaması gerekmez. Ayrıca, çok hassas hesaplama gerekmeyebilir (FP16 veya INT8 genellikle yeterli). Öte yandan, eğitim sırasında tüm ara sonuçların gradyanlarını hesaplamak için depolama alanı gerekir. Dahası, gradyanların birikmesi, sayısal küçümenlikten (veya taşmadan) kaçınmak için daha yüksek hassasiyet gerektirir. Bu, FP16'nın (veya FP32 ile karışık hassasiyet) minimum gereksinim olduğu anlamına gelir. Tüm bunlar daha

hızlı ve daha büyük bellek (HBM2'ye karşı GDDR6) ve daha fazla işlem gücü gerektirir. Örneğin, NVIDIA'nın Turing[159] T4 GPU'ları çıkarım için optimize edilirken V100 GPU'ları eğitim için tercih edilir.

Section 12.4.4 içinde gösterildiği gibi vektörleştirmeyi hatırlayın. Bir işlemci çekirdeğine vektör birimleri eklemek, verimliliği önemli ölçüde artırmamızı sağladı. Örneğin, Section 12.4.4 içindeki örnekte aynı anda 16 işlemi gerçekleştirebildik. İlk olarak, sadece vektörler arasındaki işlemleri değil, matrisler arasındakileri de optimize eden işlemleri eklesek ne olur? Bu strateji tensör çekirdeklerine yol açtı (birazdan ele alınacaktır). İkincisi, daha çok çekirdek eklersek ne olur? Özetle, bu iki strateji GPU'lardaki tasarım kararlarını özetlemektedir. Fig. 12.4.6 temel işlem bloğu hakkında genel bir bakış sunar. 16 tamsayı ve 16 kayan virgüllü sayı birimi içerir. Buna ek olarak, iki tensör çekirdeği, derin öğrenmeyle ilgili ek işlemlerin dar bir alt kümesini hızlandırır. Her akış çoklu işlemcisi dört bloktan oluşur.

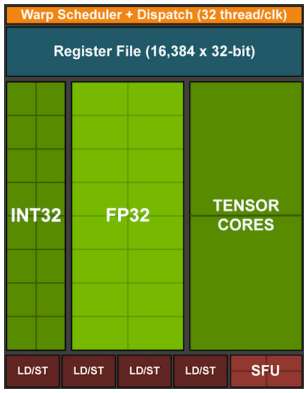


Fig. 12.4.6: NVIDIA Turing işleme bloğu (İmge NVIDIA'nın izniyle verilmektedir).

Ardından, 12 akış çoklu işlemcisi, üst düzey TU102 işlemcileri oluşturan grafik işleme öbekleri halinde gruplandırılır. Geniş bellek kanalları ve L2 önbellek düzeneği tamamlar. Fig. 12.4.7 ilgili ayrıntılara sahiptir. Böyle bir cihazın tasarlanmasının nedenlerinden biri, daha da sıkıştırılmış yongalara izin vermek ve verim sorunlarıyla başa çıkmak için ayrı ayrı blokları gerektiğinde ekleyebilmek veya çıkarabilmektir (hatalı modüller etkinleştirilmeyebilir). Neyse ki bu tür cihazları programlamak, CUDA ve çerçeve kodunun katmanlarının altında, sıradan derin öğrenme araştırmacılarından iyice gizlenmiştir. Özellikle, mevcut kaynakların olması koşuluyla, programlardan birinden fazlası GPU'da aynı anda yürütülebilir. Bununla birlikte, cihaz hafızasına sığmayan modelleri seçmekten kaçınmak için cihazların sınırlamalarının farkında olmakta fayda vardır.

[159] https://devblogs.nvidia.com/nvidia-turing-architecture-in-depth/

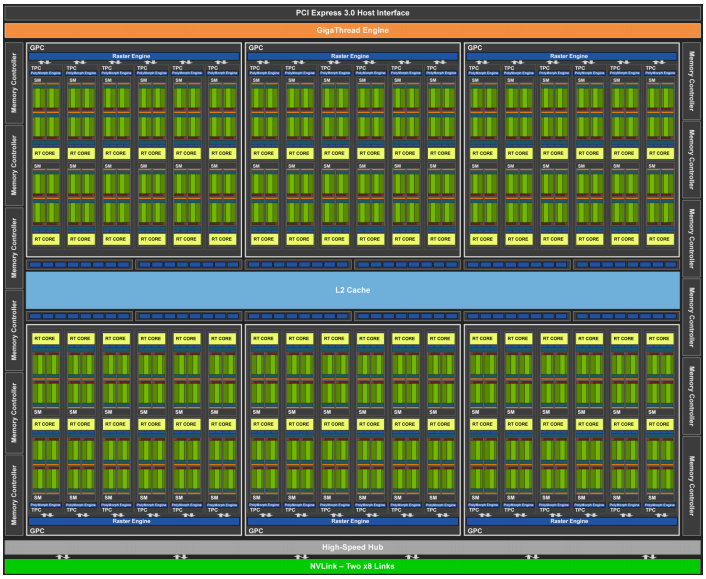


Fig. 12.4.7: NVIDIA Turing mimarisi (İmge NVIDIA'nın izniyle verilmektedir)

Daha ayrıntılı olarak bahsetmeye değer son yaklaşım *tensör çekirdekleri*dir. Bunlar, özellikle derin öğrenme için etkili olan daha optimize devreler eklemeye yönelik son trendin bir örneğidir. Örneğin, TPU hızlı matris çarpımı için (Kung, 1988) sistolik (kalp atış ritmine benzer) bir dizi ekledi. Oradaki tasarım, büyük operasyonların çok az kısmını (ilk nesil TPU'lar için) destekliyordu. Tensör çekirdekleri diğer uçtadır. Sayısal kesinliğe bağlı olarak 4×4 ve 16×16 matrisleri içeren küçük işlemler için optimize edilmişlerdir. Fig. 12.4.8 optimizasyonlara genel bir bakış sunar.

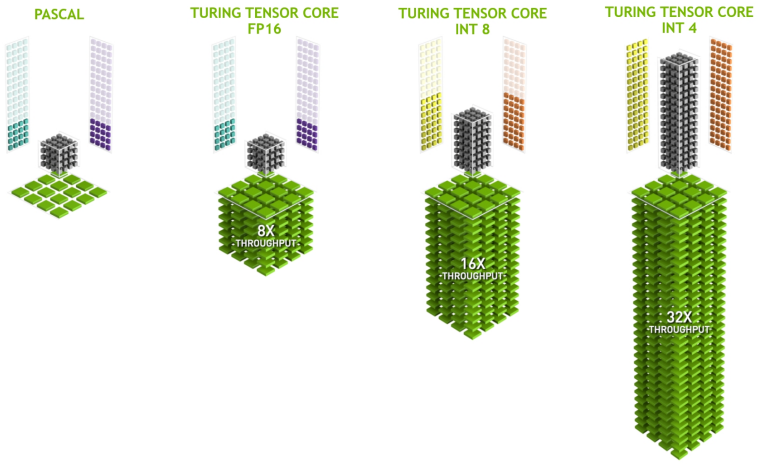


Fig. 12.4.8: Turing'de NVIDIA tensör çekirdekleri (İmge NVIDIA'nın izniyle verilmektedir).

Açıkçası hesaplama için optimize ederken bazı tavizlar veririz. Bunlardan biri, GPU'ların işkesmeleri ve seyrek verileri ele almakta çok iyi olmadığıdır. Gunrock[160] (Wang *et al.*, 2016) gibi önemli istisnalar olsa da, seyrek matrislerin ve vektörlerin erişim düzeni, GPU'ların mükemmel olduğu yüksek bant genişliği çoğuşma işlemleriyle iyi gitmez. Her iki hedefi de eşleştirmek aktif bir araştırma alanıdır. Çizgeler üzerinde derin öğrenmeye ayarlanmış bir kütüphane için bkz., DGL[161].

[160] https://github.com/gunrock/gunrock
[161] http://dgl.ai

12.4.6 Ağlar ve Veri Yolları

Optimizasyon için tek bir cihaz yetersiz olduğunda, işlemeyi senkronize etmek için veri aktarmamız gerekir. Ağların ve veri yollarının kullanışlı olduğu yer burasıdır. Bir dizi tasarım parametresine sahibiz: Bant genişliği, maliyet, mesafe ve esneklik. Bir uçta, oldukça iyi bir menzile sahip WiFi var, kullanımı çok kolay (telsiz, sonuçta), ucuz ama nispeten vasat bant genişliği ve gecikme sunuyor. Aklı başında hiçbir makine öğrenmesi araştırmacısı bunu bir sunucu kümesi oluşturmak için kullanmaz. Aşağıda derin öğrenme için uygun olan ara bağlantılara odaklanıyoruz.

- **PCIe**, şerit başına çok yüksek bant genişliğine sahip noktadan noktaya bağlantılar (16 şeritli bir yuvada PCIe 4.0'da 32 GB/s'ye kadar) için özel bir veri yoludur. Gecikme, tek basamaklı mikrosaniye (5 µs) seviyesindedir. PCIe bağlantıları değerlidir. İşlemcilerde yalnızca sınırlı sayıda bulunur: AMD'nin EPYC 3'ünün 128 şeridi vardır, Intel'in Xeon'unda yonga başına 48 şerit vardır; masaüstü sınıfı CPU'larda rakamlar sırasıyla 20 (Ryzen 9) ve 16 (Core i9). GPU'lar tipik olarak 16 şerit olduğundan, bu CPU'ya tam bant genişliğinde bağlanabilen GPU sayısını sınırlar. Sonuçta, bağlantıları depolama ve Ethernet gibi diğer yüksek bant genişlikli çevre birimleriyle paylaşmaları gerekir. RAM erişiminde olduğu gibi, azaltılmış paket yükü nedeniyle büyük toplu aktarımlar tercih edilir.
- **Ethernet**, bilgisayarları bağlamanın en yaygın kullanılan yoludur. PCIe göre önemli ölçüde daha yavaş olsa da, kurulum için çok ucuz ve esnektir ve çok daha uzun mesafeleri kapsar. Düşük dereceli sunucular için tipik bant genişliği 1 GBit/s'dir. Üst düzey cihazlar (örneğin, bulutta C5 örnekleri[162]) 10 ile 100 GBit/s arasında bant genişliği sunar. Önceki tüm durumlarda olduğu gibi veri iletimi önemli ek yük getirir. Doğrudan ham Ethernet'i neredeyse hiç kullanmadığımıza, fiziksel bağlantı (UDP veya TCP/IP gibi) üstünde yürütülen bir protokol kullandığımıza dikkat edin. Bu daha fazla ek yük getirir. PCIe gibi Ethernet de, bir bilgisayar ve bir anahtar gibi iki aygıtı bağlamak için tasarlanmıştır.
- **Anahtarlar** birden fazla cihazı, herhangi bir çiftinin (tipik olarak tam bant genişliği) noktadan noktaya bağlantıyı aynı anda gerçekleştirebileceği şekilde bağlamamıza izin verir. Örneğin, Ethernet anahtarları 40 sunucuyu yüksek kesitsel bant genişliğinde bağlayabilir. Anahtarların geleneksel bilgisayar ağlarına özgü olmadığını unutmayın. Hatta PCIe şeritleri de anahtarlanabilir[163]. Bu durum, örneğin, P2 örnekleri[164] için olduğu gibi bir ana işlemciye çok sayıda GPU bağlarken gerçekleşir.
- **NVLink**, çok yüksek bant genişliği ara bağlantıları söz konusu olduğunda PCIe için bir alternatiftir. Bağlantı başına 300 Gbit/s'ye kadar veri aktarım hızı sunar. Sunucu GPU'larında (Volta V100) altı bağlantı bulunurken, tüketici sınıfı GPU'larda (RTX 2080 Ti) yalnızca bir bağlantı bulunur ve 100 Gbit/sn hızında çalışır. GPU'lar arasında yüksek veri aktarımı sağlamak için NCCL[165]'i kullanmanızı öneririz.

[162] https://aws.amazon.com/ec2/instance-types/c5/
[163] https://www.broadcom.com/products/pcie-switches-bridges/pcie-switches
[164] https://aws.amazon.com/ec2/instance-types/p2/
[165] https://github.com/NVIDIA/nccl

12.4.7 Daha Fazla Gecikme Miktarları

Section 12.4.7 ve Section 12.4.7 içindeki özetler bu sayıların güncellenmiş bir sürümünü tutan Eliot Eshelman[166]'in GitHub gist[167]'inden gelmektedir.

Eylem	Zaman	Ek Bilgi
L1 önbellek referans/isabet	1.5 ns	4 döngüsü
Kayan virgüllü sayı toplama/çarpma/FMA	1.5 ns	4 döngüsü
L2 önbellek referans/isabet	5 ns	12 ~ 17 döngüsü
Dal yanlış tahmin	6 ns	15 ~ 20 döngü
L3 önbellek isabet (paylaşılmayan önbellek)	16 ns	42 döngü
L3 önbellek isabet (başka bir çekirdekte paylaşılan)	25 ns	65 döngü
Mutex kilitle/kilidi aç	25 ns	
L3 önbellek isabet (başka bir çekirdekte değiştirilmiş)	29 ns	75 döngü
L3 önbellek isabet (uzak CPU soket)	40 ns	100 ~ 300 döngü (40 ~ 116 ns)
Başka bir CPU'ya QPI sıçraması CPU (sıçrama başına)	40 ns	
64MB bellek ref. (yerel CPU)	46 ns	TinyMemBench on Broadwell E5-2690v4
64MB bellek ref. (uzak CPU)	70 ns	TinyMemBench on Broadwell E5-2690v4
256MB bellek ref. (yerel CPU)	75 ns	TinyMemBench on Broadwell E5-2690v4
Intel Optane rasgele yazma	94 ns	UCSD Non-Volatile Systems Lab
256MB bellek ref. (uzak CPU)	120 ns	TinyMemBench on Broadwell E5-2690v4
Intel Optane rasgele okuma	305 ns	UCSD Non-Volatile Systems Lab
Send 4KB over 100 Gbps HPC fabric	1 µs	Intel Omni-Path üzerinden MVAPICH2
Google Snappy ile 1KB sıkıştırma	3 µs	
10 Gbps ethernet üzerinden 4KB gönderme	10 µs	
NVMe SSD'e 4KB rasgele yazma	30 µs	DC P3608 NVMe SSD (QOS 99% is 500µs)
NVLink GPU ile 1MB aktarım	30 µs	~33GB/s on NVIDIA 40GB NVLink
PCI-E GPU ile 1MB aktarım	80 µs	~12GB/s on PCIe 3.0 x16 link
NVMe SSD'den 4KB rasgele okuma	120 µs	DC P3608 NVMe SSD (QOS 99%)
NVMe SSD'ten sıralı 1MB okuma	208 µs	~4.8GB/s DC P3608 NVMe SSD
SATA SSD'e rasgele 4KB yazma	500 µs	DC S3510 SATA SSD (QOS 99.9%)
SATA SSD'den rasgele 4KB okuma	500 µs	DC S3510 SATA SSD (QOS 99.9%)
Aynı veri merkezinde çift yönlü gönderim	500 µs	Tek yönlü yoklama (ping) ~250µs
SATA SSD'den sıralı 1MB okuma	2 ms	~550MB/s DC S3510 SATA SSD
Diskten sıralı 1MB okuma	5 ms	~200MB/s sunucu HDD
Rasgele Disk Erişimi (arama+devir)	10 ms	
Paket gönderimi CA->Netherlands->CA	150 ms	

Table: Genel Gecikme Miktarları.

[166] https://gist.github.com/eshelman
[167] https://gist.github.com/eshelman/343a1c46cb3fba142c1afdcdeec17646

Eylem	Zaman	Ek Bilgi
GPU Paylaşımlı Bellek erişimi	30 ns	30~90 döngü (küme çakışmaları gecikme ekler)
GPU Global Bellek erişimi	200 ns	200~800 döngü
GPU'da CUDE çekirdiği başlatma	10 µs	Ana bilgisayar CPU'nun GPU'ya çekirdek başlatmasını emretmesi
NVLink GPU ile 1MB aktarım	30 µs	~33GB/s on NVIDIA 40GB NVLink
PCI-E GPU ile 1MB aktarım	80 µs	~12GB/s on PCI-Express x16 link

Table: NVIDIA Tesla GPU'ları için Gecikme Miktarları.

12.4.8 Özet

- Cihazların operasyonlar için ek yükleri vardır. Bu nedenle, çok sayıda küçük transfer yerine az sayıda büyük transferi hedeflemek önemlidir. Bu RAM, SSD'ler, ağlar ve GPU'lar için geçerlidir.
- Vektörleştirme performans için anahtardır. Hızlandırıcınızın belirli yeteneklerinin farkında olduğunuzdan emin olun. Örneğin, bazı Intel Xeon CPU'lar INT8 işlemleri için özellikle iyidir, NVIDIA Volta GPU'ları FP16 matris matris işlemlerinde mükemmeldir ve NVIDIA Turing FP16, INT8 ve INT4 işlemlerinde parlar.
- Küçük veri türleri nedeniyle sayısal taşma, eğitim sırasında (ve daha az ölçüde çıkarım sırasında) bir sorun olabilir.
- Zamansal örtüşme (aliasing) performansı önemli ölçüde düşürebilir. Örneğin, 64 bit CPU'larda bellek hizalaması 64 bit sınırlara göre yapılmalıdır. GPU'larda, evrişim boyutlarını, örn. çekirdekleri tensör olarak hizalanmış tutmak iyi bir fikirdir.
- Algoritmalarınızı donanımla (örneğin, bellek ayak izi ve bant genişliği) eşleştirin. Parametreleri önbelleklere sığdırırken büyük hız (büyük ölçekte) elde edilebilir.
- Deneysel sonuçları doğrulamadan önce kağıt üzerinde yeni bir algoritmanın performansını karalamanızı öneririz. Bir büyüklük ölçeğindeki veya daha fazlasındaki tutarsızlıklar endişe nedenleridir.
- Performans darboğazlarında hata ayıklamak için profil oluşturucuları kullanın.
- Eğitim ve çıkarım donanımları fiyat ve performans açısından farklı haz noktalarına sahiptir.

12.4.9 Alıştırmalar

1. Harici bellek arayüzüne göre hizalanmış veya yanlış hizalanmış belleğe erişim hızı arasında herhangi bir fark olup olmadığını test etmek için C kodu yazın. İpucu: Önbelleğe alma etkilerine dikkat edin.
2. Belleğe sırayla veya belirli bir sıra aralığı ile erişmek arasındaki hız farkını test edin.
3. CPU'daki önbellek boyutlarını nasıl ölçebilirsiniz?
4. Maksimum bant genişliği için birden fazla bellek kanalında verileri nasıl düzenlersiniz? Çok sayıda küçük iş parçacığınız olsaydı nasıl yerleştirirdiniz?
5. Kurumsal sınıf bir HDD 10.000 rpm'de dönüyor. Bir HDD'nin verileri okuyabilmesi için en kötü durumda harcaması gereken minimum süre nedir (kafaların neredeyse anında hareket ettiğini varsayabilirsiniz)? 2,5" HDD'ler ticari sunucular için neden popüler hale geliyor (3,5" ve 5,25" sürücülere göre)?
6. HDD üreticisinin depolama yoğunluğunu inç kare başına 1 Tbit'ten inç kare başına 5 Tbit'e çıkardığını varsayalım. 2,5" HDD'de bir halkada ne kadar bilgi saklayabilirsiniz? İç ve dış parçalar arasında bir fark var mıdır?
7. 8 bitten 16 bit veri türüne gitmek, silikon miktarını yaklaşık dört kat artırır. Neden? NVIDIA neden kendi Turing GPU'larına INT4 işlemlerini eklemiş olabilir?
8. Bellekte geriye doğru okuma karşılık ileriye doğru okumak ne kadar hızlıdır? Bu sayı farklı bilgisayarlar ve CPU satıcıları arasında farklılık gösterir mi? Neden? C kodu yazın ve onunla deney yapın.
9. Diskinizin önbellek boyutunu ölçebilir misiniz? Tipik bir HDD için nedir? SSD'lerin önbelleğe ihtiyacı var mıdır?
10. Ethernet üzerinden ileti gönderirken paket ek yükünü ölçün. UDP ve TCP/IP bağlantıları arasındaki farkı arayın.
11. Doğrudan bellek erişimi, CPU dışındaki aygıtların doğrudan belleğe (veya bellekten)) yazmasını (ve okumasını) sağlar. Bu neden iyi bir fikirdir?
12. Turing T4 GPU'nun performans sayılarına bakın. FP16'dan INT8 ve INT4'e giderken performans neden "sadece" ikiye katlanıyor?
13. San Francisco ve Amsterdam arasında gidiş-dönüş bir paket için gereken en kısa süre nedir? İpucu: Mesafenin 10.000 km olduğunu varsayabilirsiniz.

Tartışmalar[168]

[168] https://discuss.d2l.ai/t/363

12.5 Birden Fazla GPU Eğitmek

Şimdiye kadar modellerin CPU'lar ve GPU'lar üzerinde nasıl verimli bir şekilde eğitileceğini tartıştık. Hatta derin öğrenme çerçevelerinin Section 12.3 içinde hesaplama ve iletişimi otomatik olarak nasıl paralelleştirmesine izin verdiğini gösterdik. Ayrıca Section 5.5 içinde `nvidia-smi` komutunu kullanarak bir bilgisayardaki mevcut tüm GPU'ların nasıl listeleneceğini de gösterdik. Konuşmadığımız şey derin öğrenme eğitiminin nasıl paralelleştirileceğidir. Bunun yerine, bir şekilde verilerin birden fazla cihaza bölüneceğini ve çalışmasının sağlanacağını ima ettik. Mevcut bölüm ayrıntıları doldurur ve sıfırdan başladığınızda bir ağın paralel olarak nasıl eğitileceğini gösterir. Üst düzey API'lerde işlevsellikten nasıl yararlanılacağına ilişkin ayrıntılar Section 12.6 içinde kümelendirilmiştir. Section 11.5 içinde açıklananlar gibi minigrup rasgele gradyan iniş algoritmalarına aşina olduğunuzu varsayıyoruz.

12.5.1 Sorunu Bölmek

Basit bir bilgisayarla görme problemi ve biraz modası geçmiş bir ağ ile başlayalım, örn. birden fazla evrişim katmanı, ortaklama ve en sonda muhtemelen birkaç tam bağlı katman. Yani, LeNet (LeCun *et al.*, 1998) veya AlexNet (Krizhevsky *et al.*, 2012)'e oldukça benzeyen bir ağ ile başlayalım. Birden fazla GPU (eğer bir masaüstü sunucusu ise 2, AWS g4dn.12xlarge üzerinde 4, p3.16xlarge üzerinde 8 veya p2.16xlarge üzerinde 16), aynı anda basit ve tekrarlanabilir tasarım seçimlerinden yararlanarak iyi bir hız elde edecek şekilde eğitimi parçalara bölmek istiyoruz. Sonuçta birden fazla GPU hem *bellek* hem de *hesaplama* yeteneğini artırır. Özetle, sınıflandırmak istediğimiz bir minigrup eğitim verisi göz önüne alındığında aşağıdaki seçeneklere sahibiz.

İlk olarak, ağı birden fazla GPU arasında bölümleyebiliriz. Yani, her GPU belirli bir katmana akan verileri girdi olarak alır, sonraki birkaç katmanda verileri işler ve ardından verileri bir sonraki GPU'ya gönderir. Bu, tek bir GPU'nun işleyebileceği şeylerle karşılaştırıldığında verileri daha büyük ağlarla işlememize olanak tanır. Ayrıca, GPU başına bellek ayak izi iyi kontrol edilebilir (toplam ağ ayak izinin bir kısmıdır).

Ancak, katmanlar arasındaki arayüz (ve dolayısıyla GPU'lar) sıkı eşzamanlama gerektirir. Bu, özellikle hesaplamalı iş yükleri katmanlar arasında düzgün bir şekilde eşleştirilmemişse zor olabilir. Sorun çok sayıda GPU için daha da şiddetlenir. Katmanlar arasındaki arayüz, etkinleştirme ve gradyanlar gibi büyük miktarda veri aktarımı gerektirir. Bu, GPU veri yollarının bant genişliğini kasabilir. Ayrıca, yoğun işlem gerektiren ancak sıralı işlemler, bölümleme açısından önemsizdir. Bu konuda en iyi yaklaşım için örneğin (Mirhoseini *et al.*, 2017)'e bakın. Zor bir sorun olmaya devam ediyor ve apaçık olmayan problemlerde iyi (doğrusal) ölçeklendirme elde etmenin mümkün olup olmadığı belirsizdir. Birden fazla GPU'ları birbirine zincirlemek için mükemmel bir çerçeve veya işletim sistemi desteği olmadığı sürece bunu önermiyoruz.

İkincisi, işi katmanlara bölüşebiliriz. Örneğin, tek bir GPU'da 64 kanalı hesaplamak yerine, sorunu her biri 16 kanal için veri üreten 4 GPU'ya bölebiliriz. Aynı şekilde, tam bağlı bir katman için çıktı birimlerinin sayısını bölebiliriz. Fig. 12.5.1 ((Krizhevsky *et al.*, 2012)'ten alınmıştır), bu stratejinin çok küçük bir bellek ayak izine sahip GPU'larla uğraşmak için kullanıldığı bu tasarımı göstermektedir (aynı anda 2 GB). Bu, kanalların (veya birimlerin) sayısının çok küçük olmaması koşuluyla hesaplama açısından iyi ölçeklendirmeye izin verir. Ayrıca, kullanılabilir bellek doğrusal ölçeklendiğinden, birden fazla GPU gitgide artan daha büyük ağları işleyebilir.

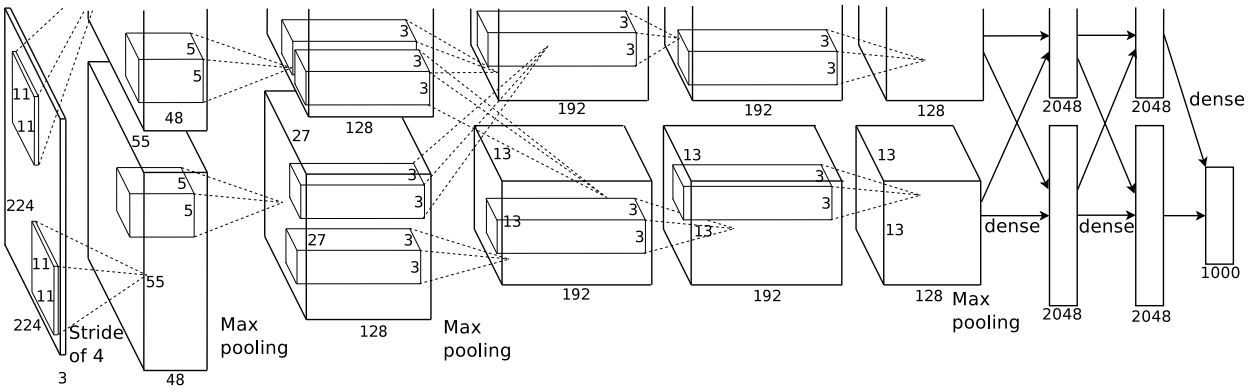


Fig. 12.5.1: Sınırlı GPU belleği nedeniyle orijinal AlexNet tasarımında model paralelliği.

Bununla birlikte, her katman diğer tüm katmanların sonuçlarına bağlı olduğundan, *çok büyük* eşzamanlama veya bariyer işlemlerine ihtiyacımız var. Ayrıca, aktarılması gereken veri miktarı, GPU'lar arasındaki katmanlara dağıtılırken olduğundan daha büyük olabilir. Bu nedenle, bant genişliği maliyeti ve karmaşıklığı nedeniyle bu yaklaşımı önermiyoruz.

Son olarak, verileri birden fazla GPU arasında bölümlendirebiliriz. Bu şekilde tüm GPU'lar farklı gözlemlerde de olsa aynı tür çalışmaları gerçekleştirir. Gradyanlar, eğitim verilerinin her minigrup işleminden sonra GPU'lar arasında toplanır. Bu en basit yaklaşımdır ve her durumda uygulanabilir. Sadece her minigrup işleminden sonra eşzamanlı hale getirmemiz gerekiyor. Yani, diğerleri hala hesaplanırken gradyan parametreleri alışverişine başlamak son derece arzu edilir. Dahası, daha fazla sayıda GPU daha büyük minigrup boyutlarına yol açarak eğitim verimliliğini arttırır. Ancak, daha fazla GPU eklemek daha büyük modeller eğitmemize izin vermez.

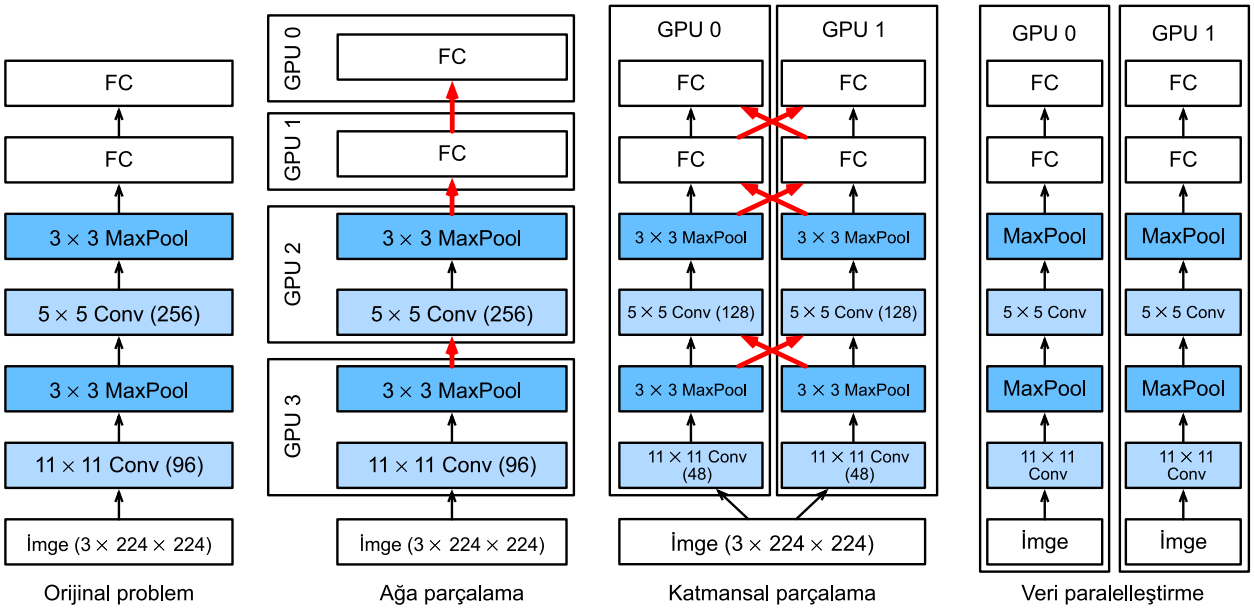


Fig. 12.5.2: Çoklu GPU'larda paralelleştirme. Soldan sağa: orijinal problem, ağ bölümleme, katman bazında bölümleme, veri paralelliği.

Birden fazla GPU üzerinde farklı paralelleştirme yollarının karşılaştırılması Fig. 12.5.2 içinde tasvir edilmiştir. Yeterince büyük belleğe sahip GPU'lara erişimimiz olması koşuluyla, genel olarak veri paralelliği ilerlemenin en uygun yoludur. Dağıtılmış eğitim için bölümlemenin ayrın-

tılı bir açıklaması için ayrıca bkz. (Li *et al.*, 2014). GPU belleği derin öğrenmenin ilk günlerinde bir sorundu. Şimdiye kadar bu sorun aşırı sıradışı durumlar haricinde herkes için çözülmüştür. Aşağıda veri paralelliğine odaklanıyoruz.

12.5.2 Veri Paralelleştirme

Bir makinede k tane GPU olduğunu varsayalım. Eğitilecek model göz önüne alındığında, GPU'lardaki parametre değerleri aynı ve eşzamanlı olsa da, her GPU bağımsız olarak eksiksiz bir model parametreleri kümesini koruyacaktır. Örnek olarak, Fig. 12.5.3 $k = 2$ olduğunda veri paralelleştirme ile eğitimi göstermektedir.

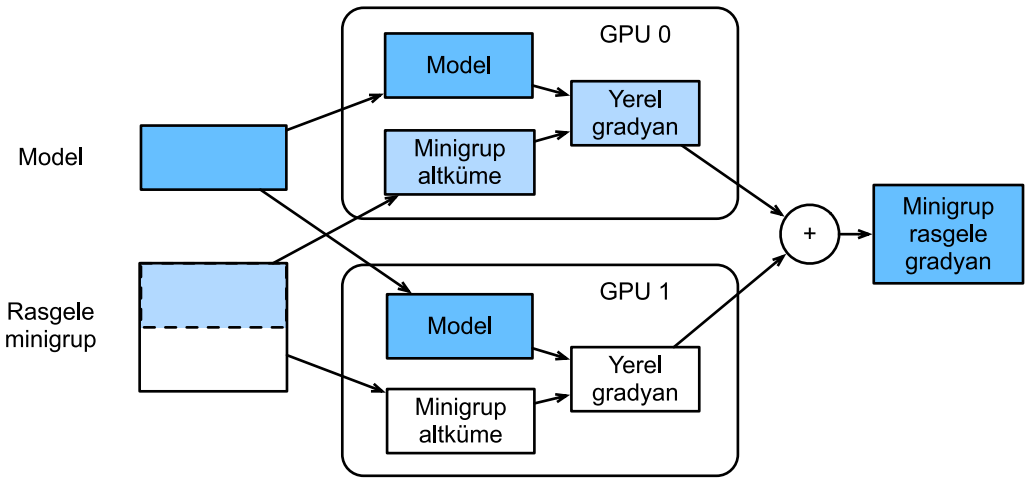


Fig. 12.5.3: İki GPU'da veri paralelliği kullanılarak minigrup rasgele gradyan inişinin hesaplanması.

Genel olarak, eğitim aşağıdaki gibi devam eder:

- Eğitimin herhangi bir yinelemesinde, rastgele bir minigrup verildiğinde, toplu iş örnekleri k tane parçaya ayırır ve GPU'lara eşit olarak dağıtırız.
- Her GPU, atandığı minigrup altkümesine göre model parametrelerinin kaybını ve gradyanlarını hesaplar.
- k tane GPU'nun her birinin yerel gradyanları, geçerli minigrup rasgele gradyanı elde etmek için toplanır.
- Toplam gradyan her GPU'ya yeniden dağıtılır.
- Her GPU, koruduğu model parametrelerinin tamamını güncelleştirmek için bu minigrup rasgele gradyanını kullanır.

Pratikte, k tane GPU üzerinde eğitim yaparken minigrup boyutunu k-kat *artırdığımızı*, böylece her GPU'nun yalnızca tek bir GPU üzerinde eğitim yapıyormuşuz gibi aynı miktarda iş yapması gerektiğini unutmayın. 16 GPU'lu bir sunucuda bu, minigrup boyutunu önemli ölçüde artırabilir ve buna göre öğrenme oranını artırmamız gerekebilir. Ayrıca, Section 7.5 içindeki toplu normalleştirmenin, örneğin GPU başına ayrı bir toplu normalleştirme katsayısı tutularak ayarlanması gerektiğini unutmayın. Aşağıda, çoklu GPU eğitimini göstermek için basit bir ağ kullanacağız.

```
%matplotlib inline
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

12.5.3 Basit Bir Örnek Ağ

LeNet'i Section 6.6 içinde tanıtıldığı gibi kullanıyoruz (hafif değiştirmeler ile). Parametre değişimini ve eşzamanlılığı ayrıntılı olarak göstermek için sıfırdan tanımlıyoruz.

```
# Model parametrelerini ilklet
scale = 0.01
W1 = torch.randn(size=(20, 1, 3, 3)) * scale
b1 = torch.zeros(20)
W2 = torch.randn(size=(50, 20, 5, 5)) * scale
b2 = torch.zeros(50)
W3 = torch.randn(size=(800, 128)) * scale
b3 = torch.zeros(128)
W4 = torch.randn(size=(128, 10)) * scale
b4 = torch.zeros(10)
params = [W1, b1, W2, b2, W3, b3, W4, b4]

# Modeli tanımla
def lenet(X, params):
    h1_conv = F.conv2d(input=X, weight=params[0], bias=params[1])
    h1_activation = F.relu(h1_conv)
    h1 = F.avg_pool2d(input=h1_activation, kernel_size=(2, 2), stride=(2, 2))
    h2_conv = F.conv2d(input=h1, weight=params[2], bias=params[3])
    h2_activation = F.relu(h2_conv)
    h2 = F.avg_pool2d(input=h2_activation, kernel_size=(2, 2), stride=(2, 2))
    h2 = h2.reshape(h2.shape[0], -1)
    h3_linear = torch.mm(h2, params[4]) + params[5]
    h3 = F.relu(h3_linear)
    y_hat = torch.mm(h3, params[6]) + params[7]
    return y_hat

# Çapraz Entropi Kayıp İşlevi
loss = nn.CrossEntropyLoss(reduction='none')
```

12.5.4 Veri Eşzamanlama

Verimli çoklu GPU eğitimi için iki temel işleme ihtiyacımız var. Öncelikle birden fazla cihaza parametre listesini dağıtabilme ve gradyanları (`get_params`) iliştirme yeteneğine sahip olmamız gerekir. Parametreler olmadan ağı bir GPU üzerinde değerlendirmek imkansızdır. İkincisi, birden fazla cihazda parametreleri toplama yeteneğine ihtiyacımız var, yani bir `allreduce` işlevine ihtiyacımız var.

```
def get_params(params, device):
    new_params = [p.to(device) for p in params]
```

(continues on next page)

(continued from previous page)

```
    for p in new_params:
        p.requires_grad_()
    return new_params
```

Model parametrelerini bir GPU'ya kopyalayarak deneyelim.

```
new_params = get_params(params, d2l.try_gpu(0))
print('b1 agirligi:', new_params[1])
print('b1 grad:', new_params[1].grad)
```

```
b1 agirligi: tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,␣
↪0., 0.],
       device='cuda:0', requires_grad=True)
b1 grad: None
```

Henüz herhangi bir hesaplama yapmadığımız için, ek girdi parametresi ile ilgili gradyan hala sıfırdır. Şimdi birden fazla GPU arasında dağıtılmış bir vektör olduğunu varsayalım. Aşağıdaki `allreduce` işlevi tüm vektörleri toplar ve sonucu tüm GPU'lara geri gönderir. Bunun işe yaraması için verileri sonuçları toplayan cihaza kopyalamamız gerektiğini unutmayın.

```
def allreduce(data):
    for i in range(1, len(data)):
        data[0][:] += data[i].to(data[0].device)
    for i in range(1, len(data)):
        data[i][:] = data[0].to(data[i].device)
```

Farklı cihazlarda farklı değerlere sahip vektörler oluşturarak ve bunları toplayarak bunu test edelim.

```
data = [torch.ones((1, 2), device=d2l.try_gpu(i)) * (i + 1) for i in range(2)]
print('allreduce oncesi:\n', data[0], '\n', data[1])
allreduce(data)
print('allreduce sonrasi:\n', data[0], '\n', data[1])
```

```
allreduce oncesi:
 tensor([[1., 1.]], device='cuda:0')
 tensor([[2., 2.]], device='cuda:1')
allreduce sonrasi:
 tensor([[3., 3.]], device='cuda:0')
 tensor([[3., 3.]], device='cuda:1')
```

12.5.5 Veri Dağıtımı

Bir minigrubu birden çok GPU boyunca eşit olarak dağıtmak için basit bir yardımcı işleve ihtiyacımız vardır. Örneğin, iki GPU'da verilerin yarısının GPU'lardan birine kopyalanmasını istiyoruz. Daha kullanışlı ve daha özlü olduğu için, 4×5 matrisinde denemek için derin öğrenme çerçevesindeki yerleşik işlevi kullanıyoruz.

```
data = torch.arange(20).reshape(4, 5)
devices = [torch.device('cuda:0'), torch.device('cuda:1')]
split = nn.parallel.scatter(data, devices)
print('girdi :', data)
print('suraya yukle', devices)
print('cikti:', split)
```

```
girdi : tensor([[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]])
suraya yukle [device(type='cuda', index=0), device(type='cuda', index=1)]
cikti: (tensor([[0, 1, 2, 3, 4],
        [5, 6, 7, 8, 9]], device='cuda:0'), tensor([[10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]], device='cuda:1'))
```

Daha sonra yeniden kullanım için hem verileri hem de etiketleri parçalara bölen bir `split_batch` işlevi tanımlıyoruz.

```
#@save
def split_batch(X, y, devices):
    """`X` ve `y`'yi birçok cihaza parçala."""
    assert X.shape[0] == y.shape[0]
    return (nn.parallel.scatter(X, devices),
            nn.parallel.scatter(y, devices))
```

12.5.6 Eğitim

Artık çoklu-GPU eğitimini tek bir minigrupta uygulayabiliriz. Uygulaması öncelikle bu bölümde açıklanan veriyi paralelleştirme yaklaşımına dayanmaktadır. Verileri birden fazla GPU arasında eşzamanlamak için az önce tartıştığımız `allreduce` ve `split_and_load` yardımcı fonksiyonlarını kullanacağız. Paralellik elde etmek için herhangi bir özel kod yazmamıza gerek olmadığını unutmayın. Hesaplama çizgesinin bir minigrup içindeki cihazlar arasında herhangi bir bağımlılığı olmadığından, paralel olarak *otomatik* yürütülür.

```
def train_batch(X, y, device_params, devices, lr):
    X_shards, y_shards = split_batch(X, y, devices)
    # Kayıp her GPU'da ayrı ayrı hesaplanır
    ls = [loss(lenet(X_shard, device_W), y_shard).sum()
          for X_shard, y_shard, device_W in zip(
              X_shards, y_shards, device_params)]
    for l in ls:  # Geri yayma her GPU'da ayrı ayrı uygulanır
        l.backward()
    # Her GPU'dan gelen tüm gradyanları toplayın ve bunları tüm GPU'lara yayınlayın
```

(continues on next page)

(continued from previous page)

```
    with torch.no_grad():
        for i in range(len(device_params[0])):
            allreduce([device_params[c][i].grad for c in range(len(devices))])
    # Model parametreleri her GPU'da ayrı ayrı güncellenir
    for param in device_params:
        d2l.sgd(param, lr, X.shape[0]) # Burada tam boyutlu bir toplu iş kullanıyoruz
```

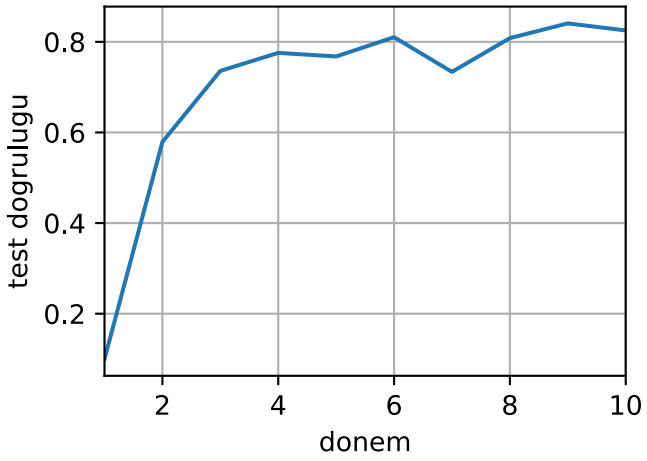
Şimdi eğitim fonksiyonunu tanımlayabiliriz. Önceki bölümlerde kullanılanlardan biraz farklıdır: GPU'ları tahsis etmeliyiz ve tüm model parametrelerini tüm cihazlara kopyalamalıyız. Açıkçası her grup birden çok GPU ile başa çıkmak için `train_batch` işlevi kullanılarak işlenir. Kolaylık sağlamak (ve kodun özlü olması) için doğruluğu tek bir GPU üzerinde hesaplıyoruz, ancak diğer GPU'lar boşta olduğundan bu *verimsiz*dir.

```
def train(num_gpus, batch_size, lr):
    train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
    devices = [d2l.try_gpu(i) for i in range(num_gpus)]
    # Model parametrelerini `num_gpus` tane GPU'ya kopyalayın
    device_params = [get_params(params, d) for d in devices]
    num_epochs = 10
    animator = d2l.Animator('donem', 'test dogrulugu', xlim=[1, num_epochs])
    timer = d2l.Timer()
    for epoch in range(num_epochs):
        timer.start()
        for X, y in train_iter:
            # Tek bir minigrup için çoklu GPU eğitimi gerçekleştirin
            train_batch(X, y, device_params, devices, lr)
            torch.cuda.synchronize()
        timer.stop()
        # Modeli GPU 0'da değerlendirin
        animator.add(epoch + 1, (d2l.evaluate_accuracy_gpu(
            lambda x: lenet(x, device_params[0]), test_iter, devices[0]),))
    print(f'test dogrulugu: {animator.Y[0][-1]:.2f}, {timer.avg():.1f} saniye/donem '
          f'on {str(devices)}')
```

Bunun tek bir GPU üzerinde ne kadar iyi çalıştığını görelim. İlk olarak 256'lık iş boyutunu ve 0.2 öğrenme oranını kullanıyoruz.

```
train(num_gpus=1, batch_size=256, lr=0.2)
```

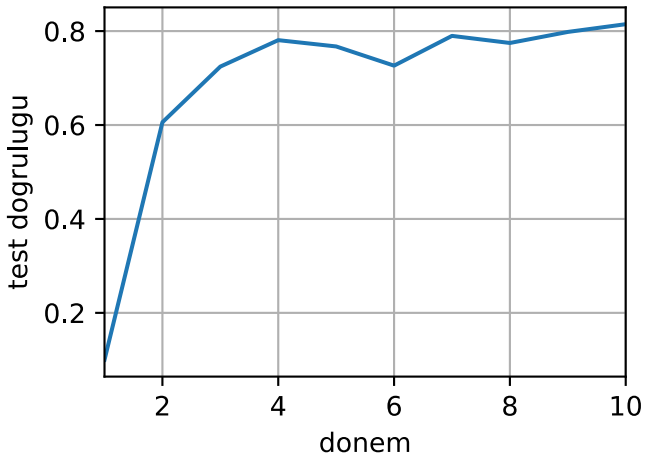
```
test dogrulugu: 0.83, 3.5 saniye/donem on [device(type='cuda', index=0)]
```



Toplu iş boyutunu ve öğrenme oranını değişmeden tutarak ve GPU sayısını 2'ye artırarak, test doğruluğunun önceki deneye kıyasla kabaca aynı kaldığını görebiliriz. Optimizasyon algoritmaları açısından aynıdırlar. Ne yazık ki burada kazanılacak anlamlı bir hız yok: Model basitçe çok küçük; dahası, çoklu GPU eğitimini uygulamaya yönelik biraz gelişmiş yaklaşımımızın önemli Python ek yükünden muzdarip olduğu küçük bir veri kümesine sahibiz. Daha karmaşık modellerle ve daha gelişmiş paralelleşme yollarıyla karşılaşacağız. Fashion-MNIST için yine ne olacağını görelim.

```
train(num_gpus=2, batch_size=256, lr=0.2)
```

```
test dogrulugu: 0.81, 2.9 saniye/donem on [device(type='cuda', index=0), device(type='cuda',␣
↪index=1)]
```



12.5.7 Özet

- Derin ağ eğitimini birden fazla GPU üzerinden bölmenin birden çok yolu vardır. Katmanlar arasında, karşılıklı katmanlar arasında veya veriler arasında bölebiliriz. İlk ikisi, sıkı bir şekilde koreografisi yapılmış veri aktarımları gerektirir. Veri paralelliği en basit stratejidir.
- Veri paralel eğitimi basittir. Bununla birlikte, bu, verimli olması için etkili minigrup boyutunu artırır.
- Veri paralelliğinde veriler birden çok GPU'ya bölünür; burada her GPU'nun kendi ileri ve geri işlemlerini yürütür ve daha sonra gradyanlar toplanır ve sonuçlar GPU'lara geri yayınlanır.
- Daha büyük minigruplar için biraz daha yüksek öğrenme oranları kullanabiliriz.

12.5.8 Alıştırmalar

1. k tane GPU'da eğitim yaparken, minigrup boyutunu b'den $k \cdot b$'e değiştirin, yani GPU sayısına göre ölçeklendirin.
2. Farklı öğrenme oranları için doğruluğu karşılaştırın. GPU sayısıyla nasıl ölçeklenir?
3. Farklı GPU'larda farklı parametreleri toplayan daha verimli bir `allreduce` işlevini uygulayın. Neden daha verimlidir?
4. Çoklu GPU test doğruluğu hesaplamasını gerçekleştirin.

Tartışmalar[169]

12.6 Çoklu GPU için Özlü Uygulama

Her yeni model için sıfırdan paralellik uygulamak eğlenceli değildir. Ayrıca, yüksek performans için eşzamanlama araçlarının optimize edilmesinde önemli fayda vardır. Aşağıda, derin öğrenme çerçevelerinin üst düzey API'lerini kullanarak bunun nasıl yapılacağını göstereceğiz. Matematik ve algoritmalar Section 12.5' içindekiler ile aynıdır. Şaşırtıcı olmayan bir şekilde, bu bölümün kodunu çalıştırmak için en az iki GPU'ya ihtiyacınız olacaktır.

```
import torch
from torch import nn
from d2l import torch as d2l
```

12.6.1 Basit Örnek Bir Ağ

Hala yeterince kolay ve hızlı eğitilen Section 12.5 içindeki LeNet'ten biraz daha anlamlı bir ağ kullanalım. Bir ResNet-18 türevini (He *et al.*, 2016) seçiyoruz. Girdi imgeleri küçük olduğundan onu biraz değiştiriyoruz. Özellikle, Section 7.6 içindekinden farkı, başlangıçta daha küçük bir evrişim çekirdeği, uzun adım ve dolgu kullanmamızdır. Ayrıca, maksimum ortaklama katmanını kaldırıyoruz.

[169] https://discuss.d2l.ai/t/1669

```
#@save
def resnet18(num_classes, in_channels=1):
    """Biraz değiştirilmiş ResNet-18 modeli."""
    def resnet_block(in_channels, out_channels, num_residuals,
                     first_block=False):
        blk = []
        for i in range(num_residuals):
            if i == 0 and not first_block:
                blk.append(d2l.Residual(in_channels, out_channels,
                                        use_1x1conv=True, strides=2))
            else:
                blk.append(d2l.Residual(out_channels, out_channels))
        return nn.Sequential(*blk)

    # Bu model daha küçük bir evrişim çekirdeği, adım ve dolgu kullanır ve
    # maksimum ortaklama katmanını kaldırır.
    net = nn.Sequential(
        nn.Conv2d(in_channels, 64, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(64),
        nn.ReLU())
    net.add_module("resnet_block1", resnet_block(64, 64, 2, first_block=True))
    net.add_module("resnet_block2", resnet_block(64, 128, 2))
    net.add_module("resnet_block3", resnet_block(128, 256, 2))
    net.add_module("resnet_block4", resnet_block(256, 512, 2))
    net.add_module("global_avg_pool", nn.AdaptiveAvgPool2d((1,1)))
    net.add_module("fc", nn.Sequential(nn.Flatten(),
                                       nn.Linear(512, num_classes)))
    return net
```

12.6.2 Ağ İlkleme

Eğitim döngüsünün içindeki ağı ilkleteceğiz. İlkleme yöntemleri üzerinde bir tazeleme için bkz. Section 4.8.

```
net = resnet18(10)
# GPU'ların bir listesini alın
devices = d2l.try_all_gpus()
# Ağı eğitim döngüsü içinde ilkleteceğiz
```

12.6.3 Eğitim

Daha önce olduğu gibi, eğitim kodunun verimli paralellik için birkaç temel işlevi yerine getirmesi gerekir:

- Ağ parametrelerinin tüm cihazlarda ilklenmesi gerekir.
- Veri kümesi üzerinde yineleme yaparken minigruplar tüm cihazlara bölünmelidir.
- Kaybı ve gradyanı cihazlar arasında paralel olarak hesaplarız.
- Gradyanlar toplanır ve parametreler buna göre güncellenir.

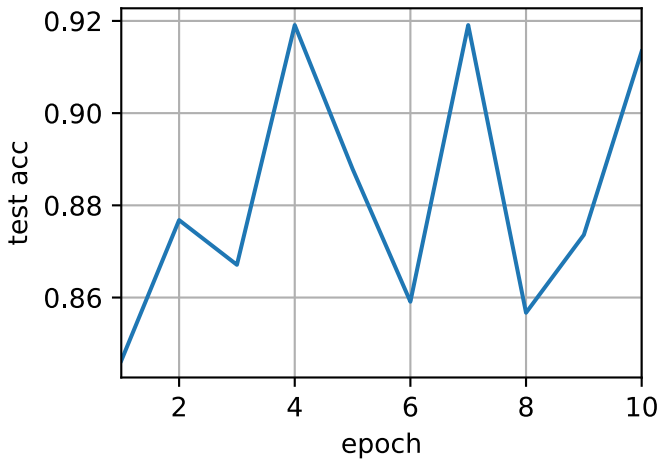
Sonunda, ağın nihai performansını bildirmek için doğruluğu (yine paralel olarak) hesaplıyoruz. Eğitim rutini, verileri bölmemiz ve toplamamız gerekmesi dışında, önceki bölümlerdeki uygulamalara oldukça benzer.

```
def train(net, num_gpus, batch_size, lr):
    train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
    devices = [d2l.try_gpu(i) for i in range(num_gpus)]
    def init_weights(m):
        if type(m) in [nn.Linear, nn.Conv2d]:
            nn.init.normal_(m.weight, std=0.01)
    net.apply(init_weights)
    # Set the model on multiple GPUs
    net = nn.DataParallel(net, device_ids=devices)
    trainer = torch.optim.SGD(net.parameters(), lr)
    loss = nn.CrossEntropyLoss()
    timer, num_epochs = d2l.Timer(), 10
    animator = d2l.Animator('epoch', 'test acc', xlim=[1, num_epochs])
    for epoch in range(num_epochs):
        net.train()
        timer.start()
        for X, y in train_iter:
            trainer.zero_grad()
            X, y = X.to(devices[0]), y.to(devices[0])
            l = loss(net(X), y)
            l.backward()
            trainer.step()
        timer.stop()
        animator.add(epoch + 1, (d2l.evaluate_accuracy_gpu(net, test_iter),))
    print(f'test acc: {animator.Y[0][-1]:.2f}, {timer.avg():.1f} sec/epoch '
          f'on {str(devices)}')
```

Bunun pratikte nasıl çalıştığını görelim. Isınma olarak ağı tek bir GPU'da eğitelim.

```
train(net, num_gpus=1, batch_size=256, lr=0.1)
```

```
test acc: 0.91, 13.6 sec/epoch on [device(type='cuda', index=0)]
```

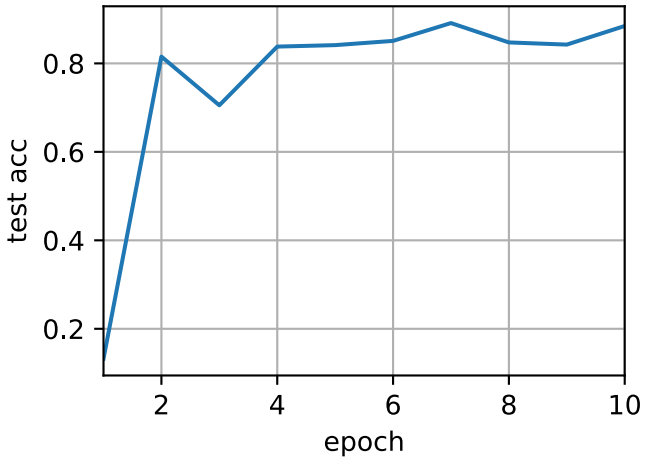


Sonra eğitim için 2 GPU kullanıyoruz. Section 12.5 içinde değerlendirilen LeNet ile

karşılaştırıldığında, ResNet-18 modeli oldukça daha karmaşıktır. Paralelleşmenin avantajını gösterdiği yer burasıdır. Hesaplama zamanı, parametreleri eşzamanlama zamanından anlamlı bir şekilde daha büyüktür. Paralelleştirme için ek yük daha az alakalı olduğundan, bu ölçeklenebilirliği artırır.

```
train(net, num_gpus=2, batch_size=512, lr=0.2)
```

```
test acc: 0.88, 8.2 sec/epoch on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



12.6.4 Özet

- Veriler, verilerin bulunabileceği cihazlarda otomatik olarak değerlendirilir.
- O cihazdaki parametrelere erişmeye çalışmadan önce her cihazdaki ağları ilklemeye özen gösterin. Aksi takdirde bir hatayla karşılaşırsınız.
- Optimizasyon algoritmaları otomatik olarak birden fazla GPU üzerinde toplanır.

12.6.5 Alıştırmalar

1. Bu bölümde ResNet-18 kullanılıyor. Farklı dönemleri, toplu iş boyutlarını ve öğrenme oranlarını deneyin. Hesaplama için daha fazla GPU kullanın. Bunu 16 GPU ile (örn. bir AWS p2.16xlarge örneğinde) denerseniz ne olur?
2. Bazen, farklı cihazlar farklı bilgi işlem gücü sağlar. GPU'ları ve CPU'yu aynı anda kullanabiliriz. İşi nasıl bölmeliyiz? Çabaya değer mi? Neden? Neden olmasın?

Tartışmalar[170]

[170] https://discuss.d2l.ai/t/1403

12.7 Parametre Sunucuları

Tek bir GPU'dan birden çok GPU'ya ve ardından, muhtemelen tümü birden çok rafa ve ağ anahtarına yayılmış birden çok GPU içeren birden çok sunucuya geçerken, dağıtılmış ve paralel eğitim için algoritmalarımızın çok daha karmaşık hale gelmesi gerekiyor. Farklı ara bağlantılarının çok farklı bant genişliğine sahip olması nedeniyle ayrıntılar önemlidir (örn. NVLink uygun bir ortamda 6 bağlantı için 100 GB/s'ye kadar sunabilir, PCIe 4.0 (16 şeritli) 32 GB/s, yüksek hızlı 100GbE Ethernet bile sadece 10 GB/s'ye ulaşır). Aynı zamanda istatistiksel bir modelleyicinin ağ ve sistemlerde uzman olmasını beklemek mantıksızdır.

Parametre sunucusunun temel fikri, dağıtılmış gizli değişken modeller bağlamında (Smola and Narayanamurthy, 2010)'da tanıtıldı. İtme ve çekme anlambiliminin bir açıklaması, (Ahmed *et al.*, 2012), ardından onu izleyen (Ahmed *et al.*, 2012) ve sistem ve açık kaynak kütüphanesinin bir açıklaması ve ardından onu izleyen (Li *et al.*, 2014) bu konudaki çalışmalardır. Aşağıda verimlilik için gerekli bileşenleri motive edeceğiz.

12.7.1 Veri-Paralel Eğitim

Dağıtılmış eğitime veri paralel eğitim yaklaşımını inceleyelim. Pratikte uygulanması önemli ölçüde daha basit olduğu için, bu bölümdeki diğerlerini hariç tutmak için bunu kullanacağız. GPU'ların günümüzde çok fazla belleği olduğundan, paralellik için başka herhangi bir stratejinin tercih edildiği (grafikler üzerinde derin öğrenmenin yanı sıra) neredeyse hiçbir kullanım durumu yoktur. Section 12.5 içinde uyguladığımız veri paralelliğinin varyantını açıklamaktadır. Bunun en önemli yönü, güncelleştirilmiş parametreler tüm GPU'lara yeniden yayınlanmadan önce gradyanların toplanmasının GPU 0'da gerçekleşmesidir.

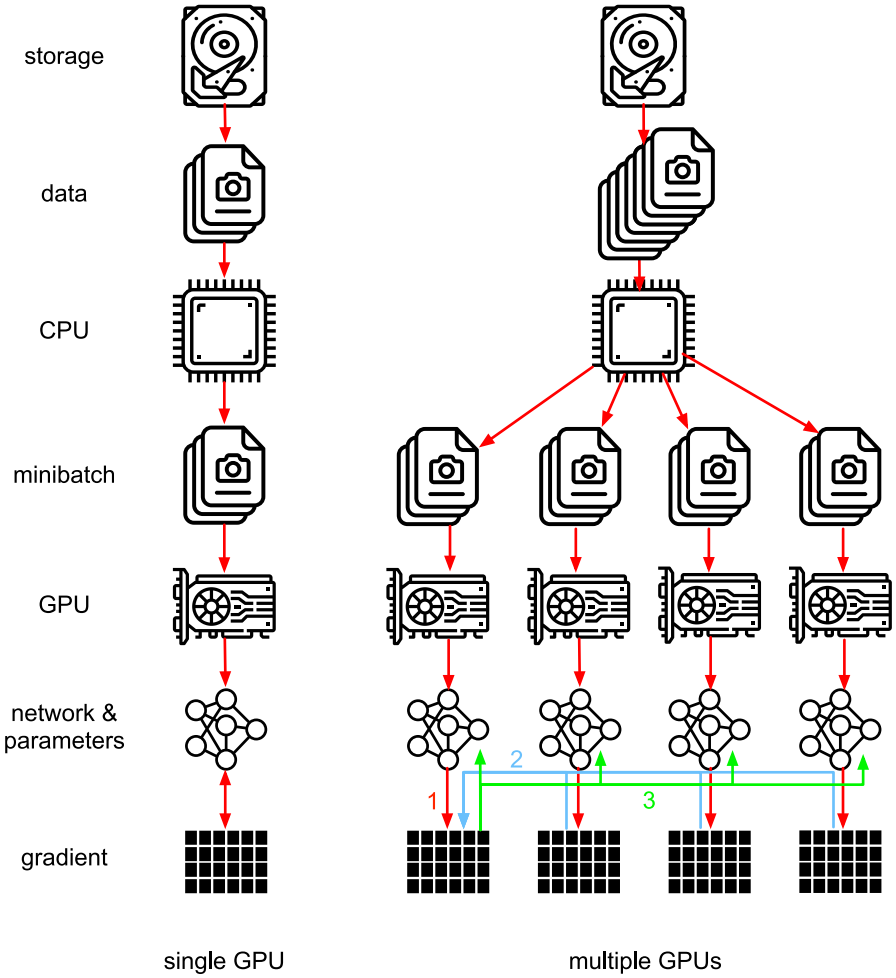


Fig. 12.7.1: Sol: Tek GPU eğitimi. Sağda: Çoklu GPU eğitiminin bir çeşidi: (1) kaybı ve gradyanı hesaplarız, (2) tüm gradyanlar tek bir GPU'da toplanır, (3) parametre güncellemesi gerçekleşir ve parametreler tüm GPU'lara yeniden dağıtılır.

Geriye dönüp bakıldığında, GPU 0'da toplama kararı oldukça geçici görünüyor. Sonuçta, CPU üzerinde de bir araya getirebiliriz. Aslında, bazı parametreleri bir GPU'da ve bazılarını diğerinde toplamaya bile karar verebiliriz. Optimizasyon algoritmasının bunu desteklemesi şartıyla, yapamamamızın gerçek bir nedeni yoktur. Örneğin, $\mathbf{g}_1, \ldots, \mathbf{g}_4$ ile ilişkili gradyanları olan dört parametre vektörümüz varsa, gradyanları her $\mathbf{g}_i$ $(i = 1, \ldots, 4)$ için bir GPU'da toplayabiliriz.

Bu muhakeme keyfi ve anlamsız görünüyor. Sonuçta, matematik baştan sona aynıdır. Bununla birlikte, Section 12.4 içinde tartışıldığı gibi farklı veri yollarının farklı bant genişliğine sahip olduğu gerçek fiziksel donanımlarla uğraşıyoruz. Fig. 12.7.2 şeklinde açıklandığı gibi gerçek bir 4 yönlü GPU sunucusunu düşünün. Özellikle iyi bağlıysa, 100 GbE ağ kartına sahip olabilir. Daha tipik sayılar, 100 MB/sn ila 1 GB/sn arasında etkili bant genişliğine sahip 1—-10 GbE aralığındadır. İşlemcilerin tüm GPU'lara doğrudan bağlanmak için çok az PCIe şeridi olduğundan (örneğin, tüketici sınıfı Intel CPU'ların 24 şeritli olması) çoklayıcı[171]ya ihtiyacımız var. 16x Gen3 bağlantısındaki CPU'dan gelen bant genişliği 16 GB/sn'dir. Bu aynı zamanda GPU'ların *her birinin* anahtara bağlanma hızıdır. Bu, cihazlar arasında iletişim kurmanın daha etkili olduğu anlamına gelir.

[171] https://www.broadcom.com/products/pcie-switches-bridges/pcie-switches

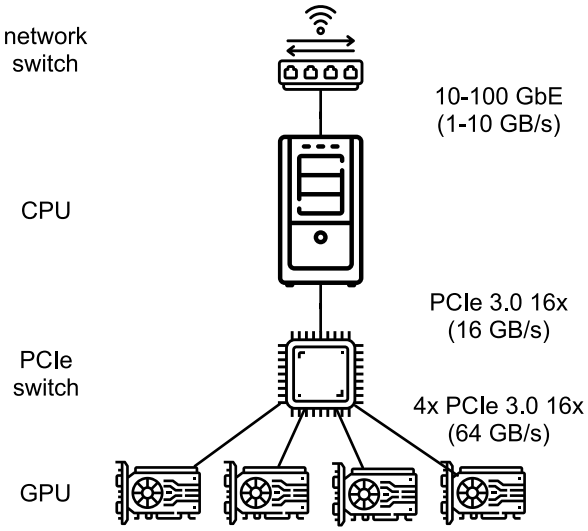


Fig. 12.7.2: A 4-yönlü GPU sunucusu.

Tartışma uğruna gradyanların 160 MB olduğunu varsayalım. Bu durumda, gradyanları kalan 3 GPU'dan dördüncüye göndermek için 30 ms gerekir (her aktarım 10 ms = 160 MB/16 GB/s alır). Ağırlık vektörlerini iletmek için 30 ms daha eklersek toplam 60 ms'ye ulaşırız. Tüm verileri CPU'ya gönderirsek, dört GPU'nun *her birinin* verileri CPU'ya göndermesi gerektiğinden, toplam 80 ms verimle 40 ms'lik bir cezaya maruz kalırız. Son olarak gradyanları her biri 40 MB'lık 4 parçaya ayırabileceğimizi varsayalım. PCIe anahtarı tüm bağlantılar arasında tam bant genişliği işlemi sunduğundan, artık parçaların her birini farklı bir GPU'da *eşzamanlı olarak* toplayabiliriz. 30 ms yerine bu 7.5 ms sürer ve bir eşzamanlama işlemi toplam 15 ms sürer. Kısacası, parametreleri nasıl senkronize ettiğimize bağlı olarak aynı işlem 15 ms'den 80 ms'ye kadar herhangi bir zaman sürebilir. Fig. 12.7.3, parametrelerin değişimi için farklı stratejileri gösterir.

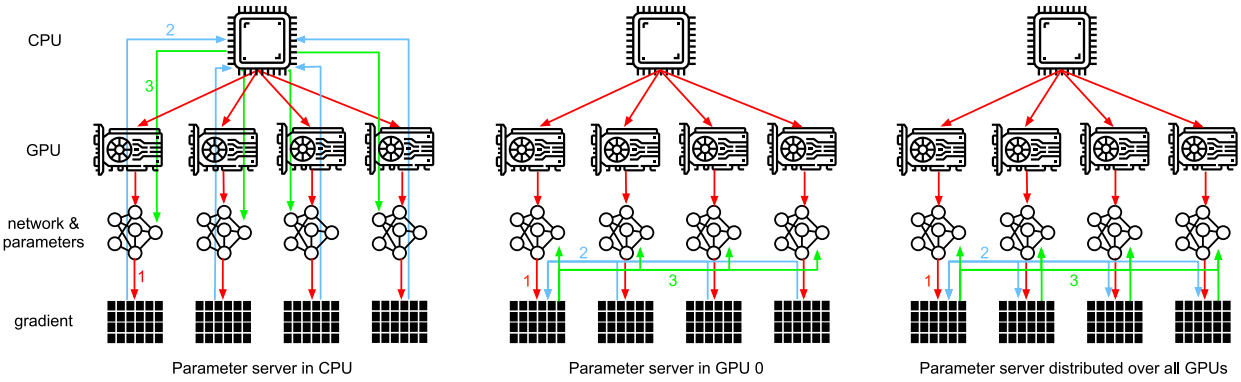


Fig. 12.7.3: Parametre eşzamanlama stratejileri.

Performansı artırmaya gelince elimizde başka bir araç daha olduğunu unutmayın: Derin bir ağda yukarıdan aşağıya tüm gradyanları hesaplamak biraz zaman alır. Bazı parametre grupları için gradyanları, diğerleri için hesaplamakla meşgulken bile senkronize etmeye başlayabiliriz. Örneğin bkz. (Sergeev and Del Balso, 2018) ve bunun nasıl yapılacağına ilişkin ayrıntılar için Horovod[172] adresine bakınız.

[172] https://github.com/horovod/horovod

12.7.2 Halka Eşzamanlaması

Modern derin öğrenme donanımı üzerinde eşzamanlama söz konusu olduğunda genellikle önemli ölçüde ısmarlama ağ bağlantısıyla karşılaşırız. Örneğin, AWS p3.16xlarge ve NVIDIA DGX-2 örnekleri Fig. 12.7.4 şeklindeki bağlantı yapısını paylaşır. Her GPU, en iyi 16 GB/s hızında çalışan bir PCIe bağlantısı üzerinden bir ana işlemciye bağlanır. Ayrıca her bir GPU'nun 6 NVLink bağlantısı vardır ve bunların her biri çift yönlü 300 Gbit/s'yi aktarabilir. Bu, yön başına bağlantı başına 18 GB/s civarındadır. Kısacası, toplam NVLink bant genişliği PCIe bant genişliğinden önemli ölçüde daha yüksektir. Soru, onu en verimli şekilde nasıl kullanacağıdır.

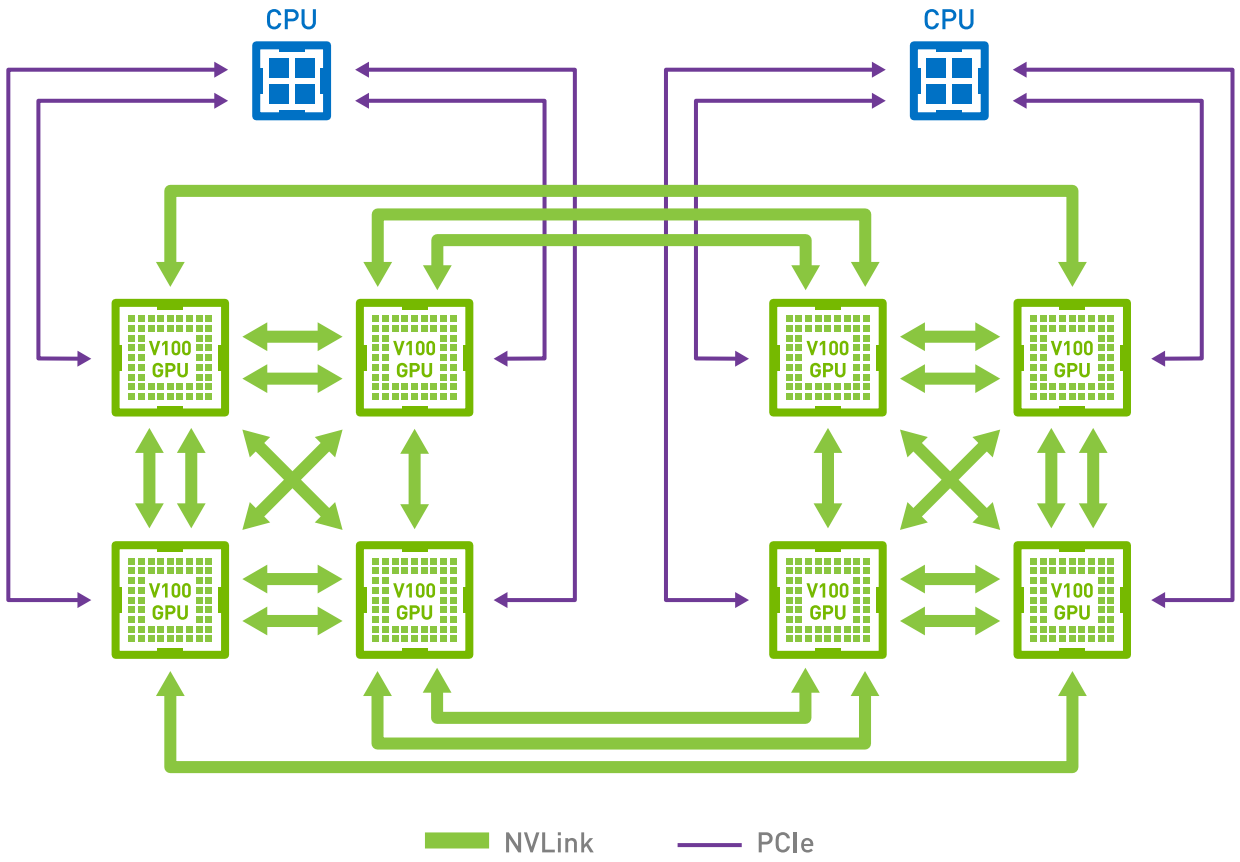


Fig. 12.7.4: 8 tane V100 GPU sunucularında NVLink bağlantısı (İmge NVIDIA'nın izniyle verilmektedir).

En uygun eşzamanlama stratejisinin ağı iki halkaya ayırmak ve bunları doğrudan verileri senkronize etmek için kullanmak olduğu ortaya çıkıyor (Wang *et al.*, 2018). Fig. 12.7.5, ağın çift NVLink bant genişliği ile bir (1-2-3-4-5-6-7-8-1) halkasına ve düzenli bant genişliği ile bir (1-4-6-3-5-8-2-7-1) halkasına ayrılabileceğini gösterir. Bu durumda verimli bir eşzamanlama protokolü tasarlamak önemsizdir.

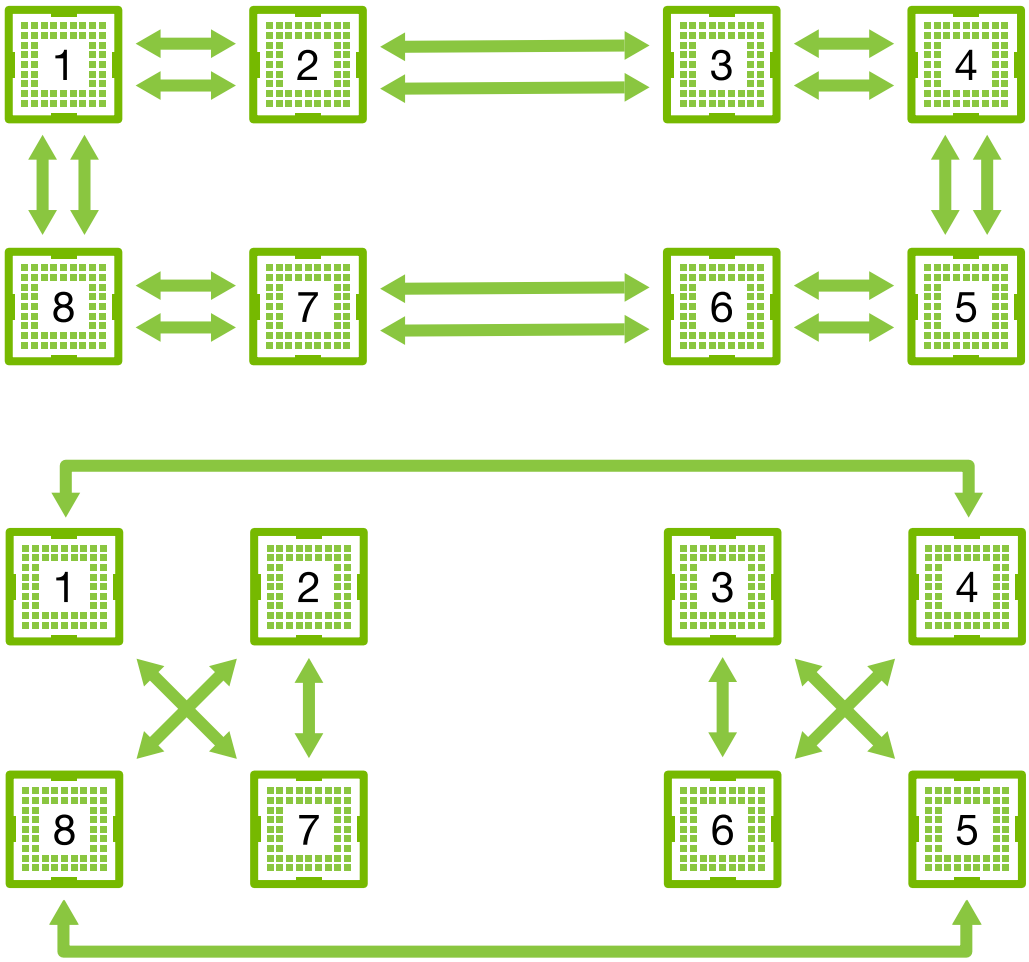


Fig. 12.7.5: NVLink ağının iki halkaya ayrıştırılması.

Aşağıdaki düşünce deneyini düşünün: n tane bilgi işlem düğümlü (veya GPU'lu) bir halka göz önüne alındığında, ilk düğümden ikinci düğüme gradyanlar gönderebiliriz. Orada yerel gradyana eklenir ve üçüncü düğüme gönderilir, vb. $n-1$ adımından sonra toplam gradyan son ziyaret edilen düğümde bulunabilir. Yani, gradyanları toplama zamanı, düğüm sayısı ile doğrusal olarak büyür. Ama bunu yaparsak algoritma oldukça verimsiz olur. Sonuçta, herhangi bir zamanda iletişim kuran sadece bir düğüm vardır. Gradyanları n parçaya bölersek ve i düğümünden başlayarak i öbeğini senkronize etmeye başlarsak ne olur? Her yığın boyutu $1/n$ olduğundan toplam süre artık $(n-1)/n \approx 1$'dir. Başka bir deyişle, gradyanları birleştirmek için harcanan süre, halkanın boyutunu arttırdığımızdan *büyümez*. Bu oldukça şaşırtıcı bir sonuçtur. Fig. 12.7.6, $n = 4$ düğümün adımların sırasını göstermektedir.

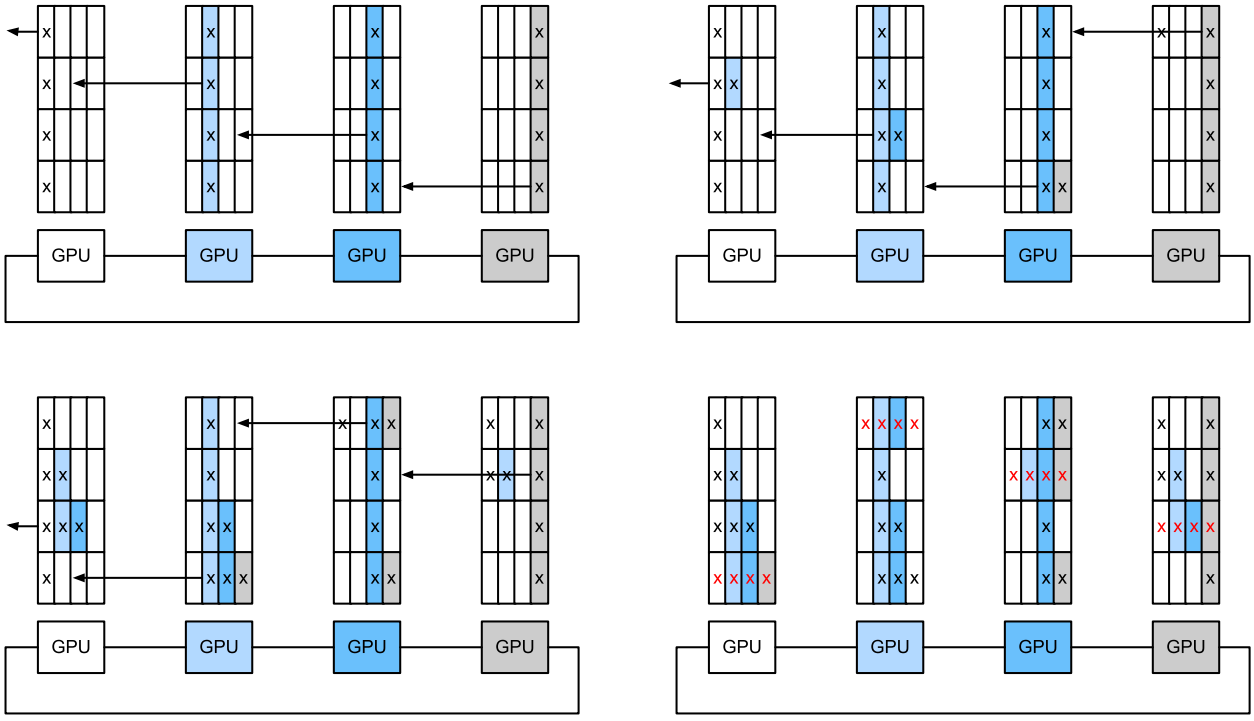


Fig. 12.7.6: 4 düğüm arasında halka senkronizasyonu. Her düğüm, birleştirilmiş gradyan sağ komşusunda bulunana kadar gradyan parçalarını sol komşusuna iletmeye başlar.

8 V100 GPU'da 160 MB senkronize etme örneğini kullanırsak yaklaşık $2 \cdot 160\text{MB}/(3 \cdot 18\text{GB/s}) \approx 6\text{ms}$'e ulaşırız. Bu, şu anda 8 GPU kullanıyor olsak da PCIe veri yolu kullanmaktan daha iyidir. Pratikte bu sayıların biraz daha kötü olduğunu unutmayın, çünkü derin öğrenme çerçeveleri genellikle iletişimi büyük çoğuşma transferlerinde birleştirmeyi başaramaz.

Halka eşzamanlama diğer eşzamanlama algoritmalarından temelde farklı olmasının yaygın bir yanlış anlaşılma olduğunu unutmayın. Tek fark, eşzamanlama yolunun basit bir ağaçla karşılaştırıldığında biraz daha ayrıntılı olmasıdır.

12.7.3 Çoklu Makine Eğitimi

Birden fazla makinede dağıtılmış eğitim ek bir zorluk getirir: Yalnızca, bazı durumlarda çok daha yavaş olabilen, nispeten daha düşük bant genişliğine sahip bir yapı üzerinden bağlanan sunucularla iletişim kurmamız gerekiyor. Cihazlar arasında eşzamanlama çetrefillidir. Sonuçta, eğitim kodu çalıştıran farklı makineler çok farklı hızlara sahip olacaktır. Bu nedenle eşzamanlı dağıtılmış optimizasyonu kullanmak istiyorsak bunları senkronize etmeliyiz. Fig. 12.7.7 dağıtılmış paralel eğitimin nasıl gerçekleştiğini göstermektedir.

1. Her makinede bir (farklı) veri grubu okunur, birden fazla GPU'ya bölünür ve GPU belleğine aktarılır. Tahminler ve gradyanlar her GPU toplu işleminde ayrı ayrı hesaplanır.
2. Tüm yerel GPU'lardan gelen gradyanlar tek bir GPU'da toplanır (veya bunların bir kısmı farklı GPU'lar üzerinde toplanır).
3. Gradyanlar CPU'lara gönderilir.
4. CPU'lar gradyanları tüm gradyanları toplayan bir merkezi parametre sunucusuna gönderir.

5. Toplanan gradyanlar daha sonra parametreleri güncelleştirmek için kullanılır ve güncelleştirilmiş parametreler tek tek CPU'lara geri yayınlanır.
6. Bilgiler bir (veya birden çok) GPU'ya gönderilir.
7. Güncelleştirilmiş parametreler tüm GPU'lara yayılır.

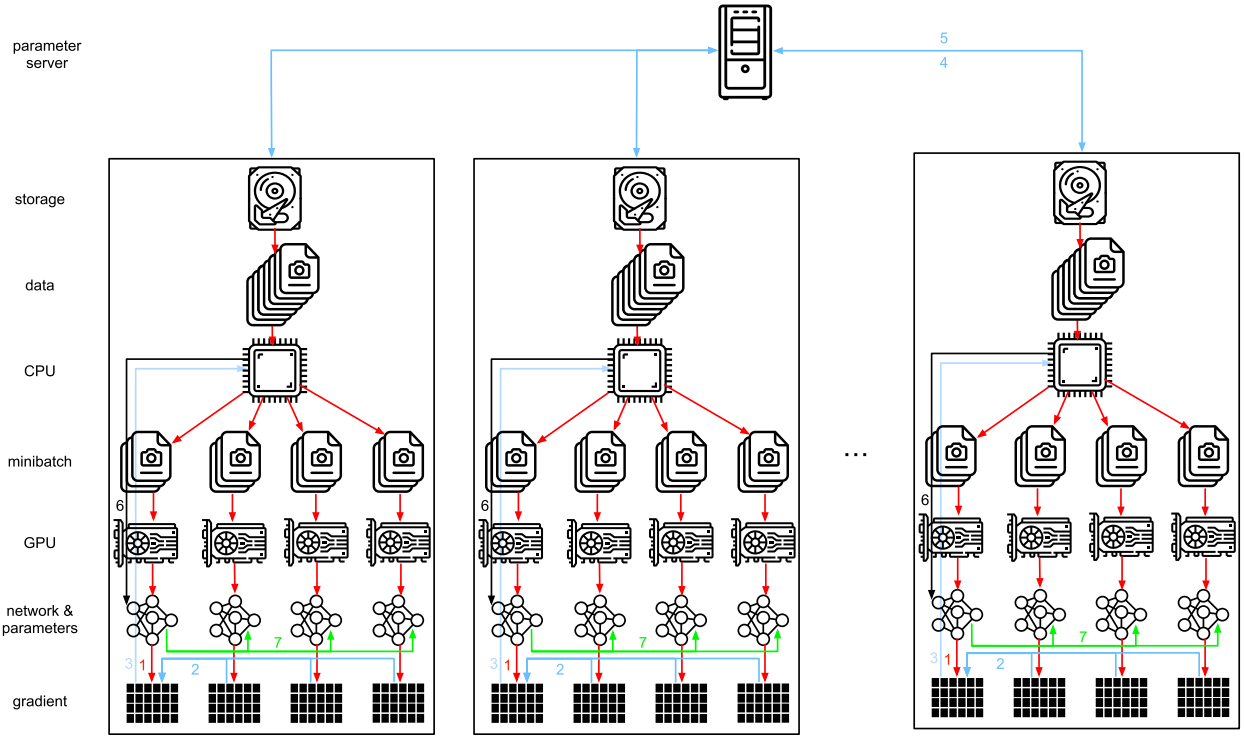


Fig. 12.7.7: Çoklu makine çoklu GPU dağıtılmış paralel eğitim.

Bu operasyonların her biri oldukça basit görünüyor. Aslında, tek bir makinede verimli bir şekilde gerçekleştirilebilirler. Birden fazla makineye baktığımızda, merkezi parametre sunucusunun darboğaz haline geldiğini görebiliriz. Sonuçta, sunucu başına bant genişliği sınırlıdır, bu nedenle m işçi makine için tüm gradyanları sunucuya göndermek için gereken süre $\mathcal{O}(m)$'dir. Sunucu sayısını n'e yükselterek bu engeli aşabiliriz. Bu noktada her sunucunun yalnızca parametrelerin $\mathcal{O}(1/n)$'unu depolaması gerekir, bu nedenle güncellemeler ve optimizasyon için toplam süre $\mathcal{O}(m/n)$ olur. Her iki sayının da eşleştirilmesi, kaç işçi makine ile uğraştığımıza bakılmaksızın sürekli ölçeklendirme sağlar. Uygulamada, *aynı* makineleri hem işçi makine hem de sunucu olarak kullanıyoruz. Fig. 12.7.8 tasarımı göstermektedir (ayrıntılar için ayrıca bkz. (Li *et al.*, 2014)). Özellikle, birden fazla makinenin makul olmayan gecikmeler olmadan çalışmasını sağlamak önemsizdir. Engellerle ilgili ayrıntıları atlıyoruz ve aşağıda eşzamanlı olan ve olmayan güncellemelere kısaca değineceğiz.

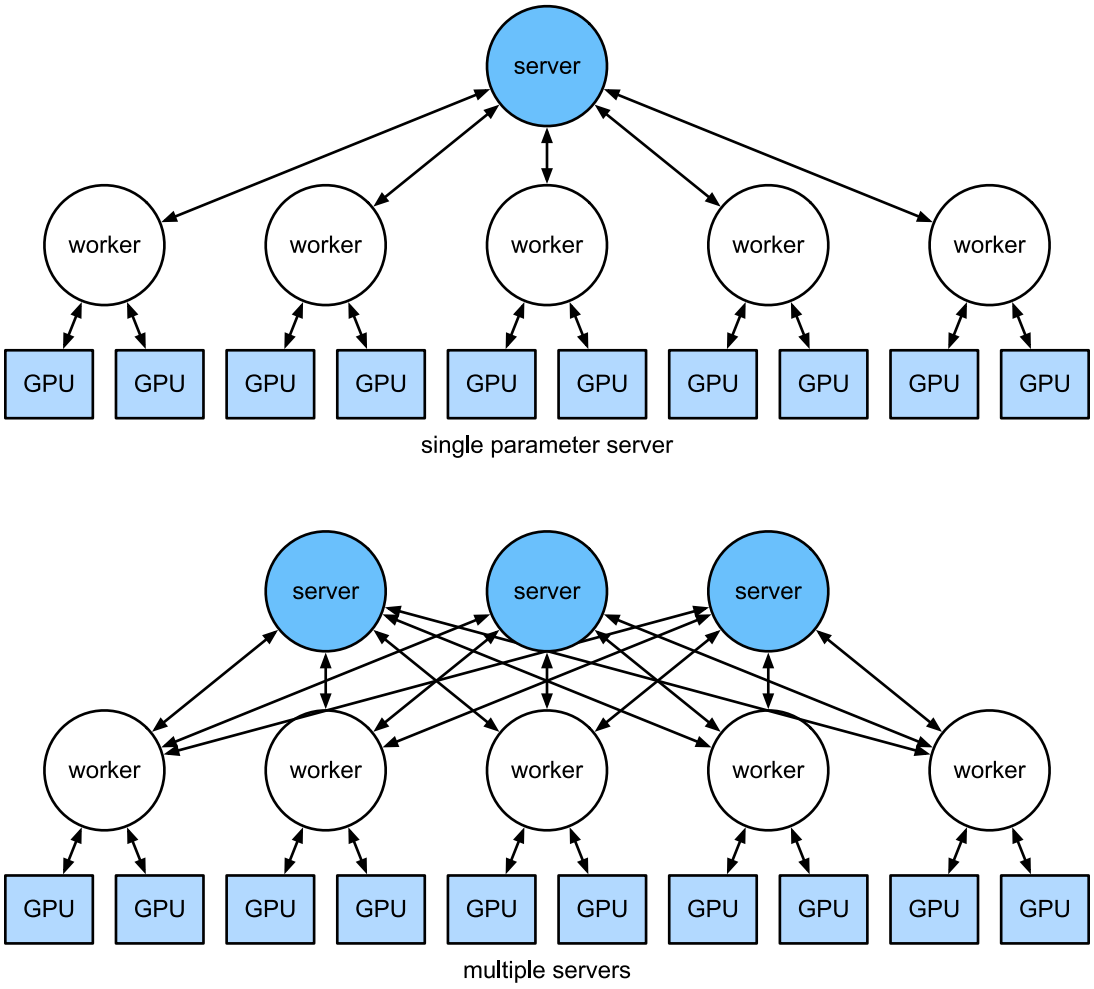


Fig. 12.7.8: Üst: Bant genişliği sınırlı olduğundan tek parametreli bir sunucu bir darboğazdır. Alt: Birden çok parametre sunucusu, toplam bant genişliğine sahip parametrelerin bölümlerini depolar.

12.7.4 Anahtar–Değer Depoları

Pratikte dağıtılmış çoklu GPU eğitimi için gerekli adımların uygulanması önemsiz değildir. Bu nedenle, yeniden tanımlanmış güncelleme semantiğine sahip bir *anahtar-değer deposu* gibi ortak bir soyutlama kullanmak faydalı olur.

Birçok işçi makine ve birçok GPU arasında i gradyan hesaplaması aşağıdadır:

$$\mathbf{g}_i = \sum_{k \in \text{işçiler}} \sum_{j \in \text{GPUs}} \mathbf{g}_{ijk}, \tag{12.7.1}$$

burada $\mathbf{g}_{ijk}$, işçi k GPU j'da bölünmüş i gradyanının bir parçasıdır. Bu işlemdeki kilit nokta, bunun bir *değişken indirgeme* olmasıdır, yani birçok vektörü bire çevirir ve işlemin uygulanma sırası önemli değildir. Hangi gradyanın ne zaman alındığı üzerinde hassas kontrole sahip olmadığımızdan bu, amaçlarımız için mükemmeldir. Ayrıca, bu işlemin farklı i'ler arasında bağımsız olduğunu unutmayın.

Bu, aşağıdaki iki işlemi tanımlamamıza olanak sağlar: Gradyanları biriken *itme* ve toplam gradyanları alan *çekme*. Birçok farklı gradyan kümesine sahip olduğumuzdan (sonuçta birçok katmana sahibiz), gradyanları bir i anahtarı ile indekslememiz gerekiyor. Dynamo'da (DeCandia *et*

al., 2007) tanıtıldığı gibi, anahtar–değer depolarına olan bu benzerlik tesadüf değildir. Bunlar da, özellikle parametreleri birden çok sunucuya dağıtmak söz konusu olduğunda birçok benzer özelliği karşılar.

Anahtar-değer depoları için itme ve çekme işlemleri aşağıdaki gibi açıklanır:

- **itme (anahtar, değer)** bir işçiden ortak bir depolamaya belirli bir gradyan (değer) gönderir. Orada değer biriktirilir, örneğin, toplanarak.
- **çekme (anahtar, değer)** ortak depodan, örneğin tüm işçi öğelerinin gradyanlarını birleştirdikten sonra toplam değeri alır.

Basit bir itme ve çekme işleminin arkasındaki eşzamanlama ile ilgili tüm karmaşıklığı gizleyerek, optimizasyonu basit terimlerle ifade edebilmek isteyen istatistiksel modelleyicilerin endişelerini ve dağıtılmış senkronizasyonun doğasında olan karmaşıklıkla başa çıkması gereken sistem mühendislerinin endişelerini ayırabiliriz.

12.7.5 Özet

- Eşzamanlamanın, bir sunucu içindeki belirli ağ altyapısına ve bağlantısına son derece uyarlanabilir olması gerekir. Bu, eşzamanlama için gereken sürede önemli bir fark yaratabilir.
- Halka senkronizasyonu p3 ve DGX-2 sunucuları için en uygun olabilir. Diğerleri için muhtemelen o kadar değildir.
- Artırılmış bant genişliği için birden fazla parametre sunucusu eklerken hiyerarşik eşzamanlama stratejisi iyi çalışır.

12.7.6 Alıştırmalar

1. Halka eşzamanlamasını daha da artırabilir misin? İpucu: Her iki yönde de mesaj gönderebilirsiniz.
2. Eşzamansız iletişime izin vermek mümkün mü (hesaplama hala devam ederken)? Performansı nasıl etkiler?
3. Uzun süren bir hesaplama sırasında bir sunucuyu kaybedersek ne olur? Hesaplamanın tamamen yeniden başlatılmasını önlemek için bir *hata toleransı* mekanizmasını nasıl tasarlayabiliriz?

Tartışmalar[173]

[173] https://discuss.d2l.ai/t/366

13 | Bilgisayarla Görme

Tıbbi teşhis, kendi kendini yöneten araçlar, kameralı izleme veya akıllı filtreler olsun, bilgisayarla görme alanındaki birçok uygulama mevcut ve gelecekteki yaşamlarımızla yakından ilişkilidir. Son yıllarda derin öğrenme, bilgisayarla görme sistemlerinin performansını artırmak için dönüştürücü güç olmuştur. En gelişmiş bilgisayarla görme uygulamalarının derin öğrenmeden neredeyse ayrılmaz olduğu söylenebilir. Bu bölüm bilgisayarla görme alanı üzerinde duracak ve yakın zamanda akademide ve endüstride etkili olan yöntemleri ve uygulamaları inceleyecektir.

Chapter 6 ve Chapter 7 içinde, bilgisayarla görmede yaygın olarak kullanılan çeşitli evrişimli sinir ağlarını inceledik ve bunları basit imge sınıflandırma görevlerine uyguladık. Bu bölümün başında, model genelleştirmesini geliştirebilecek iki yöntemi, yani *imge artırımı* ve *ince ayarlama*, tanımlayacağız ve bunları imge sınıflandırmasına uygulayacağız. Derin sinir ağları imgeleri birden çok katmanda etkili bir şekilde temsil edebildiğinden, bu tür katmanlı gösterimler, *nesne tespiti*, *anlamsal bölümleme* ve *stil aktarımı* gibi çeşitli bilgisayarla görme görevlerinde başarıyla kullanılmıştır. Bilgisayarla görmede katmanlı temsillerden yararlanmanın temel fikrini takiben, nesne tespiti için önemli bileşenler ve teknikler ile başlayacağız. Ardından, imgelerin anlamsal bölümlemesi için *tamamen evrişimli ağları* nasıl kullanılacağını göstereceğiz. Sonra bu kitabın kapağı gibi imgeler oluşturmak için stil aktarım tekniklerinin nasıl kullanılacağını açıklayacağız. Sonunda, bu bölümün materyallerini ve önceki birkaç bölümü iki bilindik bilgisayarla görme kıyaslama veri kümesine uygulayarak sonlandırıyoruz.

13.1 İmge Artırması

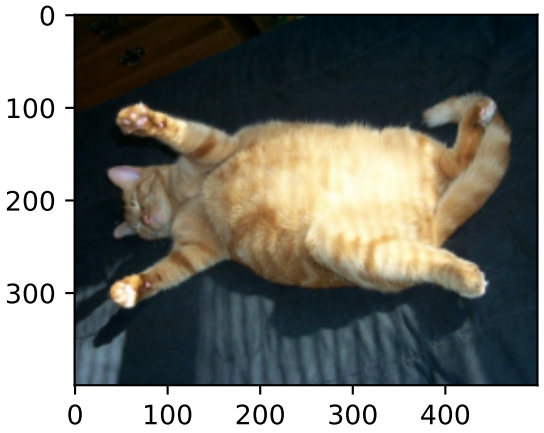
Section 7.1 içinde, büyük veri kümelerinin, çeşitli uygulamalarda derin sinir ağlarının başarısı için bir ön koşul olduğunu belirttik. *İmge artırma* eğitim imgelerinde bir dizi rastgele değişiklikten sonra benzer ama farklı eğitim örnekleri üretir ve böylece eğitim kümesinin boyutunu genişletir. Alternatif olarak, imge artırma, eğitim örneklerinin rastgele ayarlanması modellerin belirli niteliklere daha az güvenmesine ve böylece genelleme yeteneklerini geliştirmesine izin vermesi gerçeğiyle motive edilebilir. Örneğin, bir imgeyi, ilgi nesnesinin farklı pozisyonlarda görünmesini sağlamak için farklı şekillerde kırpabiliriz, böylece bir modelin nesnenin konumuna bağımlılığını azaltabiliriz. Ayrıca, modelin renge duyarlılığını azaltmak için parlaklık ve renk gibi faktörleri de ayarlayabiliriz. Muhtemelen zamanında imge artırmanın AlexNet'in başarısı için vazgeçilmez olduğu doğrudur. Bu bölümde bilgisayarla görmede yaygın olarak kullanılan bu tekniği tartışacağız.

```
%matplotlib inline
import torch
import torchvision
from torch import nn
from d2l import torch as d2l
```

13.1.1 Yaygın İmge Artırma Yöntemleri

Yaygın imge artırma yöntemlerini incelememizde, aşağıdaki 400×500 imgesini bir örnek olarak kullanacağız.

```
d2l.set_figsize()
img = d2l.Image.open('../img/cat1.jpg')
d2l.plt.imshow(img);
```



Çoğu imge artırma yönteminin belirli bir rastgelelik derecesi vardır. İmge artırmanın etkisini gözlemlememizi kolaylaştırmak için, sonraki adımda `apply` yardımcı bir işlev tanımlıyoruz. Bu işlev `img` girdi imgesinde `aug` imge artırma yöntemini birden çok kez çalıştırır ve tüm sonuçları gösterir.

```
def apply(img, aug, num_rows=2, num_cols=4, scale=1.5):
    Y = [aug(img) for _ in range(num_rows * num_cols)]
    d2l.show_images(Y, num_rows, num_cols, scale=scale)
```

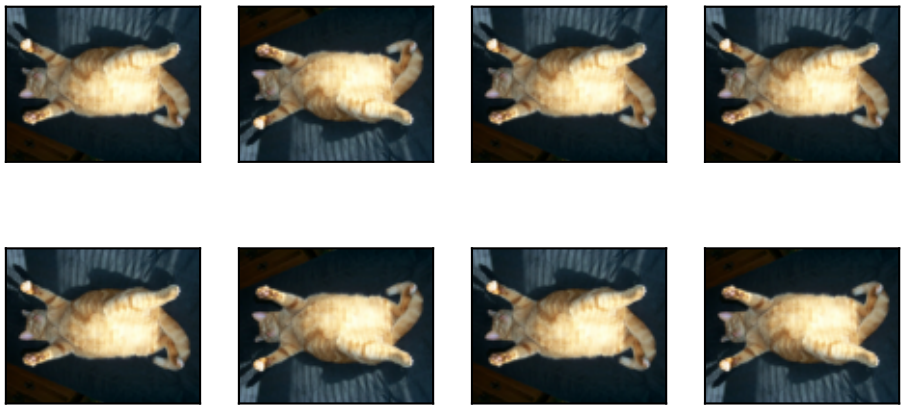
Döndürme ve Kırpma

Resmin sola ve sağa doğru döndürülmesi genellikle nesnenin kategorisini değiştirmez. Bu, en eski ve en yaygın kullanılan imge artırma yöntemlerinden biridir. Daha sonra, `transforms` modülünü `RandomHorizontalFlip` örneğini oluşturmak için kullanıyoruz, ki imgeyi sola ve sağa döndürebiliriz.

```
apply(img, torchvision.transforms.RandomHorizontalFlip())
```

Yukarı ve aşağı döndürme sola ve sağa döndürmek kadar yaygın değildir. Ancak en azından bu örnek imge için, yukarı ve aşağı döndürmek tanımayı engellemez. Ardından, bir imgeyi %50 şansla yukarı ve aşağı döndürmek için `RandomVerticalFlip` örneği oluşturuyoruz.

```
apply(img, torchvision.transforms.RandomVerticalFlip())
```



Kullandığımız örnek imgede, kedi imgenin ortasındadır, ancak bu genel olarak böyle olmayabilir. Section 6.5 içinde, ortaklama katmanının evrişimli bir katmanın hedef konuma duyarlılığını azaltabileceğini açıkladık. Buna ek olarak, nesnelerin imgedeki farklı ölçeklerde farklı konumlarda görünmesini sağlamak için imgeyi rastgele kırpabiliriz, bu da bir modelin hedef konuma duyarlılığını da azaltabilir.

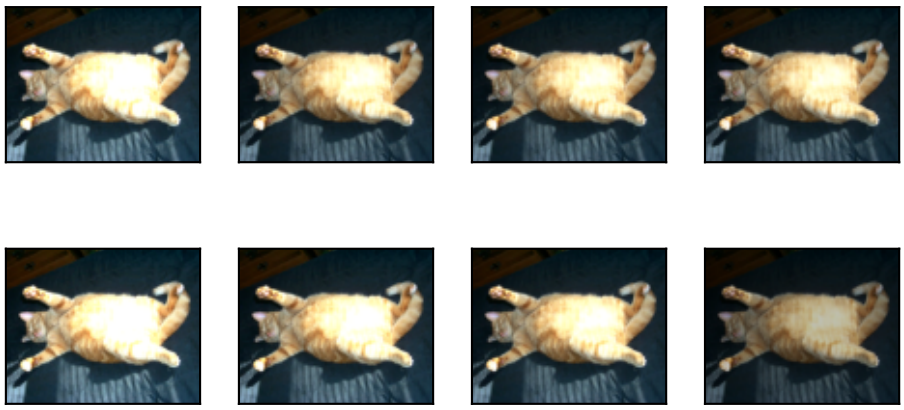
Aşağıdaki kodda, her seferinde orijinal alanın $\%10 \sim \%100$ alanı olan bir alanı rastgele kırpıyoruz ve bu alanın genişliğinin yüksekliğine oranı $0.5 \sim 2$'den rastgele seçilir. Ardından, bölgenin genişliği ve yüksekliği 200 piksele ölçeklenir. Aksi belirtilmedikçe, bu bölümdeki a ile b arasındaki rastgele sayı, $[a, b]$ aralığından rastgele ve tek biçimli örneklemeyle elde edilen sürekli bir değeri ifade eder.

```
shape_aug = torchvision.transforms.RandomResizedCrop(
    (200, 200), scale=(0.1, 1), ratio=(0.5, 2))
apply(img, shape_aug)
```

Renkleri Değiştirme

Diğer bir artırma yöntemi ise renkleri değiştirmektir. İmge renginin dört cephesini değiştirebiliriz: Parlaklık, zıtlık, doygunluk ve renk tonu. Aşağıdaki örnekte, imgenin parlaklığını orijinal imgenin %50'si $(1 - 0.5)$ ile %150'si $(1 + 0.5)$ arasında bir değere değiştiriyoruz.

```
apply(img, torchvision.transforms.ColorJitter(
    brightness=0.5, contrast=0, saturation=0, hue=0))
```



Benzer şekilde, imgenin tonunu rastgele değiştirebiliriz.

```
apply(img, torchvision.transforms.ColorJitter(
    brightness=0, contrast=0, saturation=0, hue=0.5))
```

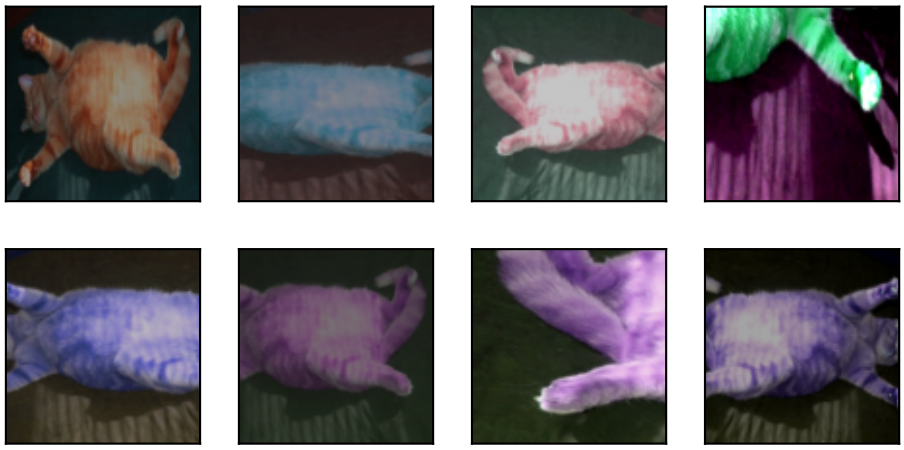
Ayrıca bir `RandomColorJitter` örneği oluşturabilir ve imgenin `brightness` (parlaklık), `contrast` (zıtlık), `saturation` (doygunluk) ve `hue`'ini (renk tonu) aynı anda nasıl rastgele değiştirebileceğimizi ayarlayabiliriz.

```
color_aug = torchvision.transforms.ColorJitter(
    brightness=0.5, contrast=0.5, saturation=0.5, hue=0.5)
apply(img, color_aug)
```

Çoklu İmge Artırma Yöntemlerini Birleştirme

Pratikte, birden fazla imge artırma yöntemini birleştireceğiz. Örneğin, yukarıda tanımlanan farklı imge artırma yöntemlerini birleştirebilir ve bunları bir `Compose` (beste) örneği aracılığıyla her imgeye uygulayabiliriz.

```
augs = torchvision.transforms.Compose([
    torchvision.transforms.RandomHorizontalFlip(), color_aug, shape_aug])
apply(img, augs)
```

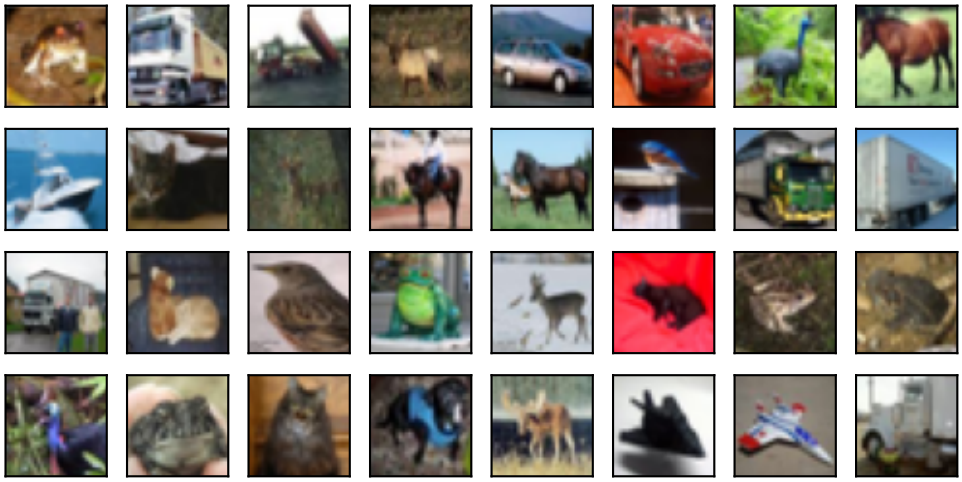


13.1.2 İmge Artırması ile Eğitim

İmge artırma ile bir model eğitelim. Burada daha önce kullandığımız Fashion-MNIST veri kümesi yerine CIFAR-10 veri kümesini kullanıyoruz. Bunun nedeni, Fashion-MNIST veri kümesindeki nesnelerin konumu ve boyutu normalleştirilirken, CIFAR-10 veri kümesindeki nesnelerin rengi ve boyutunun daha önemli farklılıklara sahip olmasıdır. CIFAR-10 veri kümelerindeki ilk 32 eğitim imgesi aşağıda gösterilmiştir.

```
all_images = torchvision.datasets.CIFAR10(train=True, root="../data",
                                          download=True)
d2l.show_images([all_images[i][0] for i in range(32)], 4, 8, scale=0.8);
```

```
Files already downloaded and verified
```



Tahmin sırasında kesin sonuçlar elde etmek için genellikle sadece eğitim örneklerine imge artırma uygularız ve tahmin sırasında rastgele işlemlerle imge artırma kullanmayız. Burada sadece en basit rastgele sol-sağ döndürme yöntemini kullanıyoruz. Buna ek olarak, bir minigrup imgeyi derin öğrenme çerçevesinin gerektirdiği biçime dönüştürmek için `ToTensor` örneğini kul-

lanıyoruz, yani 0 ile 1 arasında 32 bit kayan virgüllü sayıları (toplu iş boyutu, kanal sayısı, yükseklik, genişlik) şeklindedirler.

```
train_augs = torchvision.transforms.Compose([
     torchvision.transforms.RandomHorizontalFlip(),
     torchvision.transforms.ToTensor()])

test_augs = torchvision.transforms.Compose([
     torchvision.transforms.ToTensor()])
```

Ardından, imgeyi okumayı ve imge artırmasını uygulamayı kolaylaştırmak için yardımcı bir işlev tanımlar. PyTorch'un veri kümesi tarafından sağlanan `transform` bağımsız değişkeni, imgeleri döndürmek için artırma uygular. `DataLoader`'a ayrıntılı bir giriş için lütfen Section 3.5 kısmına bakın.

```
def load_cifar10(is_train, augs, batch_size):
    dataset = torchvision.datasets.CIFAR10(root="../data", train=is_train,
                                           transform=augs, download=True)
    dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                    shuffle=is_train, num_workers=d2l.get_dataloader_workers())
    return dataloader
```

Çoklu-GPU Eğitimi

ResNet-18 modelini Section 7.6 içindeki CIFAR-10 veri kümesi üzerinde eğitiyoruz. Section 12.6 kısmından çoklu GPU eğitimine girişi hatırlayın. Aşağıda, birden çok GPU kullanarak modeli eğitmek ve değerlendirmek için bir işlev tanımlıyoruz.

```
#@save
def train_batch_ch13(net, X, y, loss, trainer, devices):
    """Birden çok GPU'lu bir minigrup için eğitim (Bölüm 13'te tanımlanmıştır)."""
    if isinstance(X, list):
        # BERT ince ayarı için gerekli (daha sonra ele alınacaktır)
        X = [x.to(devices[0]) for x in X]
    else:
        X = X.to(devices[0])
    y = y.to(devices[0])
    net.train()
    trainer.zero_grad()
    pred = net(X)
    l = loss(pred, y)
    l.sum().backward()
    trainer.step()
    train_loss_sum = l.sum()
    train_acc_sum = d2l.accuracy(pred, y)
    return train_loss_sum, train_acc_sum
```

```
#@save
def train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs,
               devices=d2l.try_all_gpus()):
    """Birden çok GPU'lu bir modeli eğitin (Bölüm 13'te tanımlanmıştır)."""
    timer, num_batches = d2l.Timer(), len(train_iter)
```

(continues on next page)

(continued from previous page)

```
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0, 1],
                            legend=['train loss', 'train acc', 'test acc'])
    net = nn.DataParallel(net, device_ids=devices).to(devices[0])
    for epoch in range(num_epochs):
        # Eğitim kaybının toplamı, eğitim doğruluğunun toplamı, örneklerin sayısı,␣
↪tahminlerin sayısı
        metric = d2l.Accumulator(4)
        for i, (features, labels) in enumerate(train_iter):
            timer.start()
            l, acc = train_batch_ch13(
                net, features, labels, loss, trainer, devices)
            metric.add(l, acc, labels.shape[0], labels.numel())
            timer.stop()
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (metric[0] / metric[2], metric[1] / metric[3],
                              None))
        test_acc = d2l.evaluate_accuracy_gpu(net, test_iter)
        animator.add(epoch + 1, (None, None, test_acc))
    print(f'loss {metric[0] / metric[2]:.3f}, train acc '
          f'{metric[1] / metric[3]:.3f}, test acc {test_acc:.3f}')
    print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec on '
          f'{str(devices)}')
```

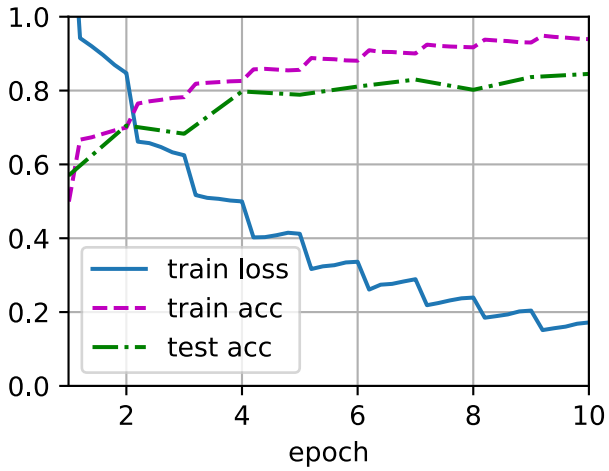
Şimdi modeli imge artırma ile eğitmek için `train_with_data_aug` işlevini tanımlayabiliriz. Bu işlev mevcut tüm GPU'ları alır, optimizasyon algoritması olarak Adam'ı kullanır, eğitim veri kümesine imge artırma uygular ve son olarak modeli eğitmek ve değerlendirmek için tanımlanmış `train_ch13` işlevini çağırır.

```
batch_size, devices, net = 256, d2l.try_all_gpus(), d2l.resnet18(10, 3)

def init_weights(m):
    if type(m) in [nn.Linear, nn.Conv2d]:
        nn.init.xavier_uniform_(m.weight)

net.apply(init_weights)

def train_with_data_aug(train_augs, test_augs, net, lr=0.001):
    train_iter = load_cifar10(True, train_augs, batch_size)
    test_iter = load_cifar10(False, test_augs, batch_size)
    loss = nn.CrossEntropyLoss(reduction="none")
    trainer = torch.optim.Adam(net.parameters(), lr=lr)
    train_ch13(net, train_iter, test_iter, loss, trainer, 10, devices)
```

Rastgele sol-sağ döndürmeye dayalı imge artırma kullanarak modeli eğitelim.

```
train_with_data_aug(train_augs, test_augs, net)
```

```
loss 0.172, train acc 0.939, test acc 0.845
5522.5 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



13.1.3 Özet

- İmge artırma, modellerin genelleme yeteneğini geliştirmek için mevcut eğitim verilerine dayanan rastgele imgeler üretir.
- Tahmin sırasında kesin sonuçlar elde etmek için genellikle sadece eğitim örneklerine imge artırma uygularız ve tahmin sırasında rastgele işlemlerle imge artırma kullanmayız.
- Derin öğrenme çerçeveleri, aynı anda uygulanabilen birçok farklı imge artırma yöntemi sağlar.

13.1.4 Alıştırmalar

1. İmge artırma özelliğini, `train_with_data_aug(test_augs, test_augs)`, kullanmadan modeli eğitin. İmge artırmasını kullanırken ve kullanmazken eğitim ve test doğruluğunu karşılaştırın. Bu karşılaştırmalı deney, imge artırmanin aşırı uyumu azaltabileceği argümanını destekleyebilir mi? Neden?
2. CIFAR-10 veri kümesinde model eğitiminde birden çok farklı imge artırma yöntemini birleştirin. Test doğruluğunu arttırıyor mu?
3. Derin öğrenme çerçevesinin çevrimiçi belgelerine bakın. Başka hangi imge artırma yöntemlerini de sağlar?

Tartışmalar[174]

[174] https://discuss.d2l.ai/t/1404

13.2 İnce Ayar

Daha önceki bölümlerde, Fashion-MNIST eğitim veri kümesindeki sadece 60000 resim ile modellerin nasıl eğitileceğini tartıştık. Ayrıca, 10 milyondan fazla imge ve 1000 nesneye sahip olan, akademide en yaygın kullanılan büyük ölçekli imge veri kümesi olan ImageNet'i de tanımladık. Ancak, genellikle karşılaştığımız veri kümesinin boyutu iki veri kümesinin arasındadır.

İmgelerden farklı sandalye türlerini tanımak istediğimizi ve kullanıcılara satın alma bağlantılarını önerdiğimizi varsayalım. Olası bir yöntem, önce 100 ortak sandalyeyi tanımlamak, her sandalye için farklı açılardan 1000 imge almak ve daha sonra toplanan imge veri kümesinde bir sınıflandırma modeli eğitmektir. Bu sandalye veri kümesi Fashion-MNIST veri kümesinden daha büyük olsa da, örneklerin sayısı hala ImageNet'in onda birinden daha azdır. Bu, bu sandalye veri kümesinde ImageNet için uygun karmaşık modellerin aşırı öğrenmesine neden olabilir. Ayrıca, sınırlı sayıda eğitim örneği nedeniyle, eğitimli modelin doğruluğu pratik gereklilikleri karşılamayabilir.

Yukarıdaki sorunları gidermek için, bariz bir çözüm daha fazla veri toplamaktır. Bununla birlikte, verilerin toplanması ve etiketlenmesi çok zaman ve para alabilir. Örneğin, ImageNet veri kümesini toplamak için, araştırmacılar araştırma fonundan milyonlarca dolar harcadı. Geçerli veri toplama maliyeti önemli ölçüde düşmüş olsa da, bu maliyet yine de göz ardı edilemez.

Diğer bir çözüm, *kaynak veri kümesinden* öğrenilen bilgileri *hedef veri kümesine* aktarmak için *öğrenme aktarımı* uygulamaktır. Örneğin, ImageNet veri kümelerindeki imgelerin çoğunun sandalyelerle ilgisi olmasa da, bu veri kümesinde eğitilen model, kenarları, dokuları, şekilleri ve nesne bileşimini tanımlamaya yardımcı olabilecek daha genel imge özelliklerini ortaya çıkarabilir. Bu benzer özellikler sandalyeleri tanımak için de etkili olabilir.

13.2.1 Adımlar

Bu bölümde, öğrenme aktarımında yaygın bir teknik tanıtacağız: *İnce ayar*. Fig. 13.2.1 içinde gösterildiği gibi, ince ayar aşağıdaki dört adımdan oluşur:

1. Bir sinir ağı modelini (yani *kaynak modeli*) kaynak veri kümesinde (örn. ImageNet veri kümesi) ön eğitin.
2. Yeni bir sinir ağı modeli oluşturun, yani *hedef modeli*. Bu, çıktı katmanı dışındaki tüm model tasarımlarını ve parametrelerini kaynak modelden kopyalar. Bu model parametrelerinin kaynak veri kümesinden öğrenilen bilgileri içerdiğini ve bu bilginin hedef veri kümesine de uygulanacağını varsayıyoruz. Kaynak modelin çıktı katmanının kaynak veri kümesinin etiketleriyle yakından ilişkili olduğunu varsayıyoruz; bu nedenle hedef modelde kullanılmaz.
3. Hedef modele çıktı sayısı hedef veri kümesindeki kategorilerin sayısı kadar olan bir çıktı katmanı ekleyin. Ardından bu katmanın model parametrelerini rastgele ilkletin.
4. Hedef modeli hedef veri kümelerinde eğitin, mesela sandalye veri kümesi. Çıktı katmanı sıfırdan eğitilirken, diğer tüm katmanların parametreleri kaynak modelin parametrelerine göre ince ayarlanır.

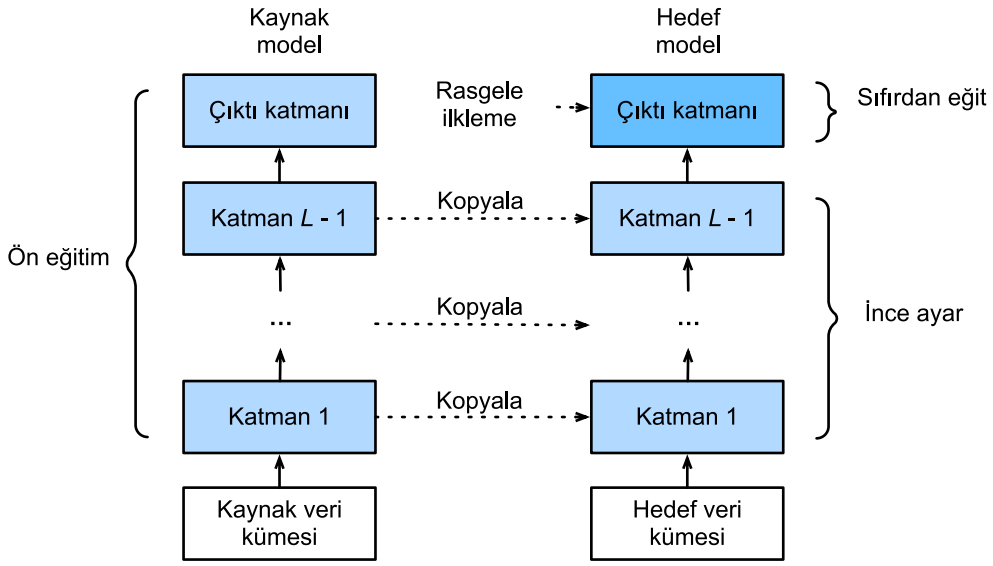


Fig. 13.2.1: İnce ayar.

Hedef veri kümeleri kaynak veri kümelerinden çok daha küçük olduğunda ince ayar modellerin genelleme yeteneğini geliştirmeye yardımcı olur.

13.2.2 Sosisli Sandviç Tanıma

Somut bir vaka aracılığıyla ince ayarı gösterelim: Sosisli sandviç tanıma. ImageNet veri kümesinde önceden eğitilmiş bir ResNet modelini küçük bir veri kümesine ince ayar yapacağız. Bu küçük veri kümesi, sosisli sandviçli ve sandviçsiz binlerce imgeden oluşur. İmgelerden sosisli sandviçleri tanımak için ince ayarlanmış model kullanacağız.

```
%matplotlib inline
import os
import torch
import torchvision
from torch import nn
from d2l import torch as d2l
```

Veri Kümesini Okuma

Kullandığımız sosisli sandviç veri kümesi çevrimiçi imgelerden alındı. Bu veri kümesi, sosisli sandviç içeren 1400 pozitif etiketli imgeden ve diğer yiyecekleri içeren birçok negatif etiketli imgeden oluşur. Her iki sınıfın 1000 imgesi eğitim için kullanılır ve geri kalanı test içindir.

İndirilen veri kümesini açtıktan sonra, iki klasör, `hotdog/train` ve `hotdog/test`, elde ediyoruz. Her iki klasörde de `hotdog` (sosisli sandviç) ve `not-hotdog` (sosisli sandviç olmayan) alt klasörleri vardır; bunlardan herbiri karşılık gelen sınıfın imgelerini içerir.

```
#@save
d2l.DATA_HUB['hotdog'] = (d2l.DATA_URL + 'hotdog.zip',
                          'fba480ffa8aa7e0febbb511d181409f899b9baa5')
```

(continues on next page)

(continued from previous page)

```
data_dir = d2l.download_extract('hotdog')
```

Eğitim ve test veri kümelerindeki tüm imge dosyalarını okumak için iki örnek oluşturuyoruz.

```
train_imgs = torchvision.datasets.ImageFolder(os.path.join(data_dir, 'train'))
test_imgs = torchvision.datasets.ImageFolder(os.path.join(data_dir, 'test'))
```

İlk 8 pozitif örnek ve son 8 negatif imge aşağıda gösterilmiştir. Gördüğünüz gibi, resimler boyut ve en-boy oranı bakımından farklılık gösterir.

```
hotdogs = [train_imgs[i][0] for i in range(8)]
not_hotdogs = [train_imgs[-i - 1][0] for i in range(8)]
d2l.show_images(hotdogs + not_hotdogs, 2, 8, scale=1.4);
```

Eğitim sırasında, önce imgeden rastgele boyut ve rastgele en-boy oranına sahip bir alanı kırpıyoruz ve sonra bu alanı 224×224 girdi imgesine ölçeklendiriyoruz. Test sırasında, imgenin hem yüksekliğini hem de genişliğini 256 piksele ölçeklendirir ve ardından merkezi bir 224×224 alanı girdi olarak kırpırız. Buna ek olarak, üç RGB (kırmızı, yeşil ve mavi) renk kanalı için değerlerini kanala göre standartlaştırıyoruz. Bir kanalın ortalama değeri, bu kanalın her değerinden çıkarılır ve sonuç bu kanalın standart sapmasına bölünür.

```
# Her kanalı standartlaştırmak için üç RGB kanalının ortalamalarını ve standart sapmalarını␣
↪belirtin
normalize = torchvision.transforms.Normalize(
    [0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

train_augs = torchvision.transforms.Compose([
    torchvision.transforms.RandomResizedCrop(224),
    torchvision.transforms.RandomHorizontalFlip(),
    torchvision.transforms.ToTensor(),
    normalize])

test_augs = torchvision.transforms.Compose([
    torchvision.transforms.Resize(256),
    torchvision.transforms.CenterCrop(224),
    torchvision.transforms.ToTensor(),
    normalize])
```

Modelin Tanımlanması ve İlklenmesi

Kaynak model olarak ImageNet veri kümesi üzerinde önceden eğitilmiş olan ResNet-18'i kullanıyoruz. Burada, önceden eğitilmiş model parametrelerini otomatik olarak indirmek için `pretrained=True`'yi belirtiyoruz. Bu model ilk kez kullanılıyorsa, indirmek için İnternet bağlantısı gereklidir.

```
pretrained_net = torchvision.models.resnet18(pretrained=True)
```

```
/home/d2l-worker/miniconda3/envs/d2l-tr-release-0/lib/python3.9/site-packages/
↪torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is␣
↪deprecated since 0.13 and will be removed in 0.15, please use 'weights' instead.
  warnings.warn(
/home/d2l-worker/miniconda3/envs/d2l-tr-release-0/lib/python3.9/site-packages/
↪torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum␣
↪or None for 'weights' are deprecated since 0.13 and will be removed in 0.15. The␣
↪current behavior is equivalent to passing weights=ResNet18_Weights.IMAGENET1K_
↪V1. You can also use weights=ResNet18_Weights.DEFAULT to get the most up-to-date␣
↪weights.
  warnings.warn(msg)
```

Önceden eğitilmiş kaynak model örneği, bir dizi öznitelik katmanı ve bir çıktı katmanı `fc` içerir. Bu bölümün temel amacı, tüm katmanların ancak çıktı tabakasının model parametrelerinin ince ayarlanmasını kolaylaştırmaktır. Kaynak modelin `fc` üye değişkeni aşağıda verilmiştir.

```
pretrained_net.fc
```

```
Linear(in_features=512, out_features=1000, bias=True)
```

Tam bağlı bir katman olarak ResNet'in son küresel ortalama ortaklama çıktılarını ImageNet veri kümesinin 1000 sınıf çıktısına dönüştürür. Daha sonra hedef model olarak yeni bir sinir ağı kurarız. Son katmandaki çıktı sayısının hedef veri kümesindeki sınıf sayısına (1000 yerine) ayarlanması dışında, önceden eğitilmiş kaynak modeliyle aynı şekilde tanımlanır.

Aşağıdaki kodda, hedef model örneğinin üye değişken özniteliklerindeki model parametreleri, kaynak modelin karşılık gelen katmanının model parametrelerine ilkletilir. Özniteliklerdeki model parametreleri ImageNet veri kümesinde önceden eğitildiğinden ve yeterince iyi olduğundan, bu parametrelerin ince ayarlanması için genellikle yalnızca küçük bir öğrenme oranına ihtiyaç vardır.

Üye değişken çıktısındaki model parametreleri rastgele ilkletilir ve genellikle sıfırdan eğitmek için daha büyük bir öğrenme oranı gerektirir. Trainer (eğtici) örneğindeki öğrenme oranının η olduğunu varsayarsak, üye değişken çıktısındaki model parametrelerinin öğrenme oranını yinelemede 10η olarak ayarlarız.

Aşağıdaki kodda, hedef model örneğinin `finetune_net` çıktı katmanından önceki model parametreleri, kaynak modelden karşılık gelen katmanların model parametrelerine ilklenir. Bu model parametreleri ImageNet'te ön eğitim yoluyla elde edildiğinden etkilidir. Bu nedenle, bu tür önceden eğitilmiş parametreleri *ince ayarlamak* için yalnızca küçük bir öğrenme oranını kullanabiliriz. Buna karşılık, çıktı katmanındaki model parametreleri rastgele ilklenir ve genellikle sıfırdan öğrenilmesi için daha büyük bir öğrenme oranı gereklidir. Temel öğrenme oranının η olursa, çıktı katmanındaki model parametrelerini yinelemek için 10η öğrenme oranı kullanılacaktır.

```
finetune_net = torchvision.models.resnet18(pretrained=True)
finetune_net.fc = nn.Linear(finetune_net.fc.in_features, 2)
nn.init.xavier_uniform_(finetune_net.fc.weight);
```

Modeli İnce Ayarlama

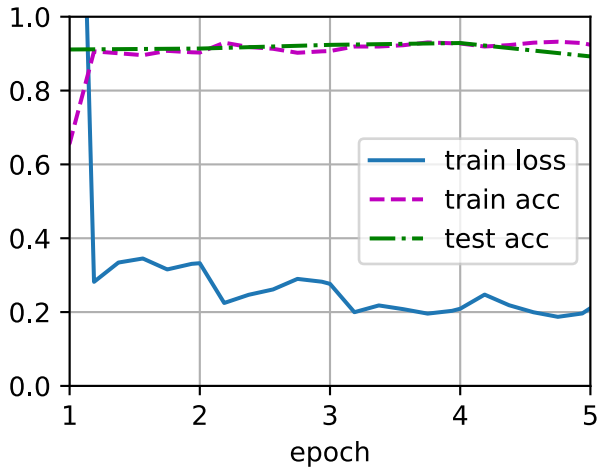
İlk olarak, ince ayar kullanan `train_fine_tuning` eğitim fonksiyonunu tanımlıyoruz, böylece birden çok kez çağrılabilir.

```
# If `param_group=True`, the model parameters in the output layer will be
# updated using a learning rate ten times greater
def train_fine_tuning(net, learning_rate, batch_size=128, num_epochs=5,
                      param_group=True):
    train_iter = torch.utils.data.DataLoader(torchvision.datasets.ImageFolder(
        os.path.join(data_dir, 'train'), transform=train_augs),
        batch_size=batch_size, shuffle=True)
    test_iter = torch.utils.data.DataLoader(torchvision.datasets.ImageFolder(
        os.path.join(data_dir, 'test'), transform=test_augs),
        batch_size=batch_size)
    devices = d2l.try_all_gpus()
    loss = nn.CrossEntropyLoss(reduction="none")
    if param_group:
        params_1x = [param for name, param in net.named_parameters()
             if name not in ["fc.weight", "fc.bias"]]
        trainer = torch.optim.SGD([{'params': params_1x},
                                   {'params': net.fc.parameters(),
                                    'lr': learning_rate * 10}],
                                  lr=learning_rate, weight_decay=0.001)
    else:
        trainer = torch.optim.SGD(net.parameters(), lr=learning_rate,
                                  weight_decay=0.001)
    d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs,
                   devices)
```

Ön eğitim yoluyla elde edilen model parametreleri *ince ayarlayabilmek* için temel öğrenme oranını küçük bir değere ayarladık. Önceki ayarlara dayanarak, hedef modelin çıktı katmanı parametrelerini on kat daha büyük bir öğrenme oranı kullanarak sıfırdan eğiteceğiz.

```
train_fine_tuning(finetune_net, 5e-5)
```

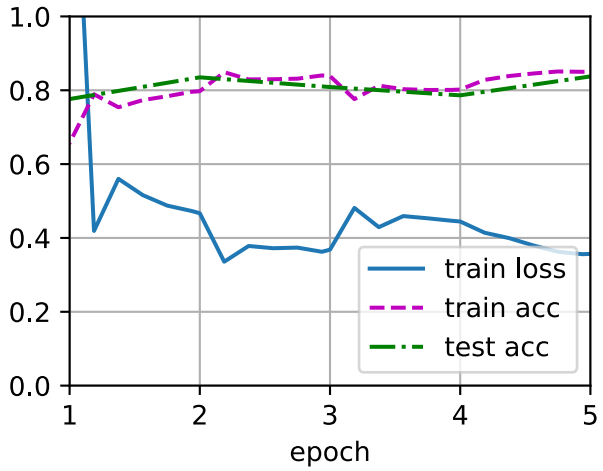
```
loss 0.211, train acc 0.924, test acc 0.892
1062.8 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



Karşılaştırma için, özdeş bir model tanımlarız, ancak tüm model parametrelerini rastgele değerlere ilklenir. Tüm modelin sıfırdan eğitilmesi gerektiğinden, daha büyük bir öğrenme oranı kullanabiliriz.

```
scratch_net = torchvision.models.resnet18()
scratch_net.fc = nn.Linear(scratch_net.fc.in_features, 2)
train_fine_tuning(scratch_net, 5e-4, param_group=False)
```

```
loss 0.357, train acc 0.851, test acc 0.838
1608.3 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



Gördüğümüz gibi, ince ayarlı model aynı dönem için daha iyi performans gösterme eğilimindedir, çünkü ilk parametre değerleri daha etkilidir.

13.2.3 Özet

- Öğrenme aktarımı kaynak veri kümesinden öğrenilen bilgiyi hedef veri kümesine aktarır. İnce ayar, öğrenim aktarımı için yaygın bir tekniktir.
- Hedef model, çıktı katmanı hariç, kaynak modelden parametreleriyle tüm model tasarımlarını kopyalar ve hedef veri kümesine göre bu parametreleri ince ayarlar. Buna karşılık, hedef modelin çıktı katmanının sıfırdan eğitilmesi gerekir.
- Genel olarak, ince ayar parametreleri daha küçük bir öğrenme oranı kullanırken, çıktı katmanını sıfırdan eğitmek daha büyük bir öğrenme oranı kullanabilir.

13.2.4 Alıştırmalar

1. `finetune_net`'in öğrenme oranını artırmaya devam edin. Modelin doğruluğu nasıl değişir?
2. Karşılaştırmalı deneyde `finetune_net` ve `scratch_net`'in hiper parametrelerini daha detaylı ayarlayın. Hala doğrulukta farklılık gösteriyorlar mı?
3. `finetune_net` çıktı katmanından önceki parametreleri kaynak modelininkine ayarlayın ve eğitim sırasında bunları *güncellemeyin*. Modelin doğruluğu nasıl değişir? Aşağıdaki kodu kullanabilirsiniz.

```
for param in finetune_net.parameters():
    param.requires_grad = False
```

4. Aslında, `ImageNet` veri kümesinde bir "sosisli sandviç" sınıfı var. Çıktı katmanındaki karşılık gelen ağırlık parametresi aşağıdaki kodla elde edilebilir. Bu ağırlık parametresinden nasıl yararlanabiliriz?

```
weight = pretrained_net.fc.weight
hotdog_w = torch.split(weight.data, 1, dim=0)[934]
hotdog_w.shape
```

```
torch.Size([1, 512])
```

Tartışmalar[175]

13.3 Nesne Algılama ve Kuşatan Kutular

Önceki bölümlerde (örn. Section 7.1—-Section 7.4), imge sınıflandırma için çeşitli modeller tanıttık. İmge sınıflandırma görevlerinde, resimde sadece *bir* ana nesne olduğunu varsayıyoruz ve sadece kategorisini nasıl tanıyacağımıza odaklanıyoruz. Bununla birlikte, ilgili imgede sıklıkla *çoklu* nesne vardır. Sadece kategorilerini değil, aynı zamanda imgedeki belirli konumlarını da bilmek istiyoruz. Bilgisayarla görmede, *nesne algılama* (veya *nesne tanıma*) gibi görevlere atıfta bulunuyoruz.

Nesne algılama birçok alanda yaygın olarak uygulanmıştır. Örneğin, kendi kendine sürüş, çekilen video görüntülerinde araçların, yayaların, yolların ve engellerin konumlarını tespit ederek

[175] https://discuss.d2l.ai/t/1439

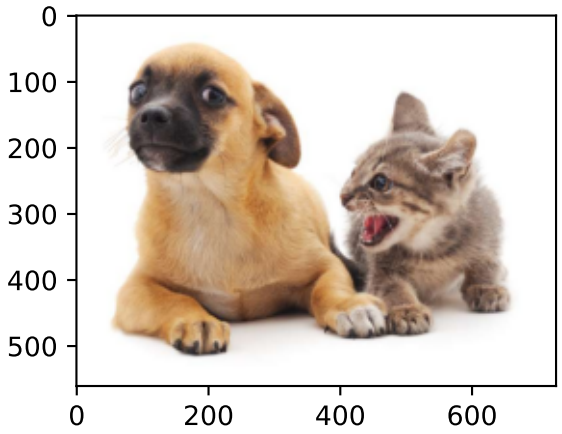
seyahat rotalarını planlamalıdır. Ayrıca, robotlar bu tekniği, bir ortamdaki gezinti boyunca ilgilenen nesneleri tespit etmek ve yerini belirlemek için kullanabilir. Ayrıca, güvenlik sistemlerinin davetsiz misafir veya bomba gibi sıradışı nesneleri algılaması gerekebilir.

Sonraki birkaç bölümde, nesne algılama için çeşitli derin öğrenme yöntemlerini tanıtacağız. Nesnelerin *konumlarına* bir giriş ile başlayacağız.

```
%matplotlib inline
import torch
from d2l import torch as d2l
```

Bu bölümde kullanılacak örnek imgeyi yükleyeceğiz. Görüntünün sol tarafında bir köpek ve sağda bir kedi olduğunu görebiliriz. Bu imgedeki iki ana nesne bunlardır.

```
d2l.set_figsize()
img = d2l.plt.imread('../img/catdog.jpg')
d2l.plt.imshow(img);
```



13.3.1 Kuşatan Kutular

Nesne algılamada, genellikle bir nesnenin uzamsal konumunu tanımlamak için bir *kuşatan kutu* kullanırız. Kuşatan kutu dikdörtgen olup, dikdörtgenin sol üst köşesinin x ve y koordinatları ve sağ alt köşenin koordinatları ile belirlenir. Yaygın olarak kullanılan bir diğer kuşatan kutu gösterimi, kuşatan kutu merkezinin (x, y) eksen koordinatları ve kutunun genişliği ve yüksekliğidir.

Burada iki temsil arasında dönüştürecek işlevleri tanımlıyoruz : `box_corner_to_center` iki köşeli temsilden merkez-genişlik yüksekliği temsiline dönüştürür ve `box_center_to_corner` tersini yapar. `boxes` girdi argümanı, iki boyutlu (n, 4) şekilli bir tensör olmalıdır, burada n sınırlayıcı kutuların sayısıdır.

```
#@save
def box_corner_to_center(boxes):
    """(sol üst, sağ alt) konumundan (merkez, genişlik, yükseklik) konumuna dönüştür."""
    x1, y1, x2, y2 = boxes[:, 0], boxes[:, 1], boxes[:, 2], boxes[:, 3]
    cx = (x1 + x2) / 2
    cy = (y1 + y2) / 2
    w = x2 - x1
```

(continues on next page)

(continued from previous page)

```
    h = y2 - y1
    boxes = torch.stack((cx, cy, w, h), axis=-1)
    return boxes

#@save
def box_center_to_corner(boxes):
    """(sol üst, sağ alt) konumundan (merkez, genişlik, yükseklik) konumuna dönüştür."""
    cx, cy, w, h = boxes[:, 0], boxes[:, 1], boxes[:, 2], boxes[:, 3]
    x1 = cx - 0.5 * w
    y1 = cy - 0.5 * h
    x2 = cx + 0.5 * w
    y2 = cy + 0.5 * h
    boxes = torch.stack((x1, y1, x2, y2), axis=-1)
    return boxes
```

Koordinat bilgilerine göre resimdeki köpek ve kedinin kuşatan kutularını tanımlayacağız. İmgedeki koordinatların orijini imgenin sol üst köşesidir ve sağ ve aşağı sırasıyla x ve y eksenlerinin pozitif yönleridir.

```
# Burada `bbox` kuşatan kutunun kısaltmasıdır
dog_bbox, cat_bbox = [60.0, 45.0, 378.0, 516.0], [400.0, 112.0, 655.0, 493.0]
```

İki kuşatan kutu dönüştürme işlevinin doğruluğunu iki kez dönüştürerek doğrulayabiliriz.

```
boxes = torch.tensor((dog_bbox, cat_bbox))
box_center_to_corner(box_corner_to_center(boxes)) == boxes
```

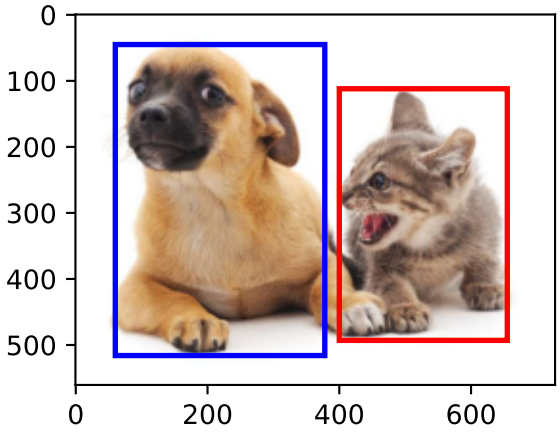
```
tensor([[True, True, True, True],
        [True, True, True, True]])
```

Doğru olup olmadıklarını kontrol etmek için resimdeki kuşatan kutuları çizelim. Çizmeden önce, `bbox_to_rect` yardımcı işlevini tanımlayacağız. `matplotlib` paketinin kuşatan kutu biçimindeki kuşatan kutusunu temsil eder.

```
#@save
def bbox_to_rect(bbox, color):
    """Kuşatan kutuyu matplotlib biçimine dönüştürün."""
    # Kuşatan kutunun (sol üst x, sol üst y, sağ alt x, sağ alt y) biçimini matplotlib␣
↪biçimine dönüştürün: ((sol üst x, sol üst y), genişlik, yükseklik)
    return d2l.plt.Rectangle(
        xy=(bbox[0], bbox[1]), width=bbox[2]-bbox[0], height=bbox[3]-bbox[1],
        fill=False, edgecolor=color, linewidth=2)
```

İmgeye kuşatan kutuları ekledikten sonra, iki nesnenin ana hatlarının temelde iki kutunun içinde olduğunu görebiliriz.

```
fig = d2l.plt.imshow(img)
fig.axes.add_patch(bbox_to_rect(dog_bbox, 'blue'))
fig.axes.add_patch(bbox_to_rect(cat_bbox, 'red'));
```



13.3.2 Özet

- Nesne algılama sadece imgede ilgili tüm nesneleri değil, aynı zamanda konumlarını da tanır. Pozisyon genellikle dikdörtgen bir kuşatan kutu ile temsil edilir.
- Yaygın olarak kullanılan iki kuşatan kutu temsili arasında dönüşüm yapabiliriz.

13.3.3 Alıştırmalar

1. Başka bir imge bulun ve nesneyi içeren bir kuşatan kutuyu etiketlemeyi deneyin. Kuşatan kutuları ve kategorileri etiketlemeyi karşılaştırın: Hangisi genellikle daha uzun sürer?
2. Neden `box_center_to_corner` ve `box_corner_to_center` girdi bağımsız değişkeninin "kutuları" nın en içteki boyutu her zaman 4'tür?

Tartışmalar[176]

13.4 Çapa Kutuları

Nesne algılama algoritmaları genellikle girdi imgesinde çok sayıda bölgeyi örnekler, bu bölgelerin ilgilenilen nesneleri içerip içermediğini belirler ve nesnelerin *gerçek referans değeri kuşatan kutularını* daha doğru bir şekilde tahmin etmek için bölgelerin sınırlarını ayarlar. Farklı modeller farklı bölge örnekleme düzenleri benimseyebilir. Burada bu tür yöntemlerden birini sunuyoruz: Her pikselde ortalanmış değişen ölçeklere ve en-boy oranlarına sahip birden çok kuşatan kutu oluşturur. Bu kuşatan kutulara *çapa kutuları* denir. Section 13.7 içinde çapa kutularına dayalı bir nesne algılama modeli tasarlayacağız.

Öncelikle, sadece daha özlü çıktılar için doğruluk yazdırmayı değiştirelim.

```
%matplotlib inline
import torch
from d2l import torch as d2l

torch.set_printoptions(2)  # Doğruluğunu basmayı basitleştirin
```

[176] https://discuss.d2l.ai/t/1527

13.4.1 Çoklu Çapa Kutuları Oluşturma

Girdi imgesinin h yüksekliğine ve w genişliğine sahip olduğunu varsayalım. İmgenin her pikselinde ortalanmış farklı şekillere sahip çapa kutuları oluşturuyoruz. *Ölçek* $s \in (0, 1]$ ve *en-boy oranı* (genişliğin yüksekliğe oranı) $r > 0$ olsun. O zaman çapa kutusunun genişliği ve yüksekliği sırasıyla $ws\sqrt{r}$ ve $hs/\sqrt{r}$'dür. Merkez konumu verildiğinde, bilinen genişlik ve yüksekliğe sahip bir çapa kutusu belirlendiğini unutmayın.

Farklı şekillerde birden çok çapa kutusu oluşturmak için, bir dizi ölçek $s_1, \ldots, s_n$ ve bir dizi en boy oranı $r_1, \ldots, r_m$ ayarlayalım. Bu ölçeklerin ve en boy oranlarının tüm kombinasyonlarını merkez olarak her pikselle birlikte kullanırken, girdi imgesinde toplam $whnm$ çapa kutusu bulunur. Bu çapa kutuları gerçek referans değerleri kuşatan kutuları kapsayabilmesine rağmen, hesaplama karmaşıklığı basitçe çok yüksektir. Uygulamada, yalnızca s_1 veya r_1 içeren kombinasyonları dikkate alalım:

$$(s_1, r_1), (s_1, r_2), \ldots, (s_1, r_m), (s_2, r_1), (s_3, r_1), \ldots, (s_n, r_1). \tag{13.4.1}$$

Yani, aynı pikselde ortalanmış çapa kutularının sayısı $n + m - 1$'dir. Tüm girdi imgesi için toplam $wh(n + m - 1)$ çapa kutusu oluşturacağız.

Yukarıdaki çapa kutuları oluşturma yöntemi aşağıdaki `multibox_prior` işlevinde uygulanır. Girdi imgesini, ölçeklerin bir listesini ve en-boy oranlarının bir listesini belirleriz, daha sonra bu işlev tüm çapa kutularını döndürür.

```
#@save
def multibox_prior(data, sizes, ratios):
    """Her pikselde ortalanmış farklı şekillere sahip çapa kutuları oluşturun."""
    in_height, in_width = data.shape[-2:]
    device, num_sizes, num_ratios = data.device, len(sizes), len(ratios)
    boxes_per_pixel = (num_sizes + num_ratios - 1)
    size_tensor = torch.tensor(sizes, device=device)
    ratio_tensor = torch.tensor(ratios, device=device)
    # Bağlantıyı bir pikselin merkezine taşımak için kaydırmalar gereklidir.
    # Bir pikselin yüksekliği=1 ve genişliği=1 olduğundan, merkezlerimizi 0.5
    # ile kaydırmayı seçiyoruz.
    offset_h, offset_w = 0.5, 0.5
    steps_h = 1.0 / in_height  # y ekseninde ölçeklenmiş adımlar
    steps_w = 1.0 / in_width  # x ekseninde ölçeklenmiş adımlar

    # Çapa kutuları için tüm merkez noktalarını oluştur
    center_h = (torch.arange(in_height, device=device) + offset_h) * steps_h
    center_w = (torch.arange(in_width, device=device) + offset_w) * steps_w
    shift_y, shift_x = torch.meshgrid(center_h, center_w)
    shift_y, shift_x = shift_y.reshape(-1), shift_x.reshape(-1)

    # Daha sonra çapa kutusu köşe koordinatları (xmin, xmax, ymin, ymax)
    # oluşturmak için kullanılacak yükseklik ve genişliklerin `boxes_per_pixel`
    # sayısını oluşturun
    w = torch.cat((size_tensor * torch.sqrt(ratio_tensor[0]),
                   sizes[0] * torch.sqrt(ratio_tensor[1:])))\
                   * in_height / in_width  # Dikdörtgen girdileri yönet
    h = torch.cat((size_tensor / torch.sqrt(ratio_tensor[0]),
                   sizes[0] / torch.sqrt(ratio_tensor[1:])))
    # Yarım yükseklik ve yarım genişlik elde etmek için 2'ye bölün
    anchor_manipulations = torch.stack((-w, -h, w, h)).T.repeat(
```

(continues on next page)

(continued from previous page)

```
                                    in_height * in_width, 1) / 2

    # Her merkez noktasında `boxes_per_pixel` sayısı çapa kutusu olacaktır,
    # bu nedenle `boxes_per_pixel` tekrarlarıyla tüm çapa kutusu
    # merkezlerinin bir ızgarasını oluşturun
    out_grid = torch.stack([shift_x, shift_y, shift_x, shift_y],
                dim=1).repeat_interleave(boxes_per_pixel, dim=0)
    output = out_grid + anchor_manipulations
    return output.unsqueeze(0)
```

Döndürülen çapa kutusu değişkeni `Y`'nin şeklinin (parti boyutu, çapa kutusu sayısı, 4) olduğunu görebiliriz.

```
img = d2l.plt.imread('../img/catdog.jpg')
h, w = img.shape[:2]

print(h, w)
X = torch.rand(size=(1, 3, h, w))  # Girdi verisi oluştur
Y = multibox_prior(X, sizes=[0.75, 0.5, 0.25], ratios=[1, 2, 0.5])
Y.shape
```

```
561 728
/home/d2l-worker/miniconda3/envs/d2l-tr-release-0/lib/python3.9/site-packages/torch/
↪functional.py:478: UserWarning: torch.meshgrid: in an upcoming release, it will be␣
↪required to pass the indexing argument. (Triggered internally at  ../aten/src/ATen/native/
↪TensorShape.cpp:2895.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
```

```
torch.Size([1, 2042040, 4])
```

Çapa kutusu değişkeninin şeklini değiştirdikten sonra `Y` olarak (imge yüksekliği, imge genişliği, aynı piksel üzerinde ortalanmış çapa kutularının sayısı, 4), belirtilen piksel konumuna ortalanmış tüm çapa kutularını elde edebiliriz. Aşağıda (250, 250) merkezli ilk çapa kutusuna erişiyoruz. Dört öğeye sahiptir: Sol üst köşedeki (x, y) eksen koordinatları ve çapa kutusunun sağ alt köşesindeki (x, y) eksen koordinatları. Her iki eksenin koordinat değerleri sırasıyla imgenin genişliği ve yüksekliğine bölünür; böylece değer aralığı 0 ile 1 arasındadır.

```
boxes = Y.reshape(h, w, 5, 4)
boxes[250, 250, 0, :]
```

```
tensor([0.06, 0.07, 0.63, 0.82])
```

İmgedeki tüm çapa kutularını bir pikselde ortalanmış olarak gösterirken, imge üzerinde birden çok kuşatan kutu çizmek için aşağıdaki `show_bboxes` işlevini tanımlarız.

```
#@save
def show_bboxes(axes, bboxes, labels=None, colors=None):
    """Çapa kutularını göster."""
```

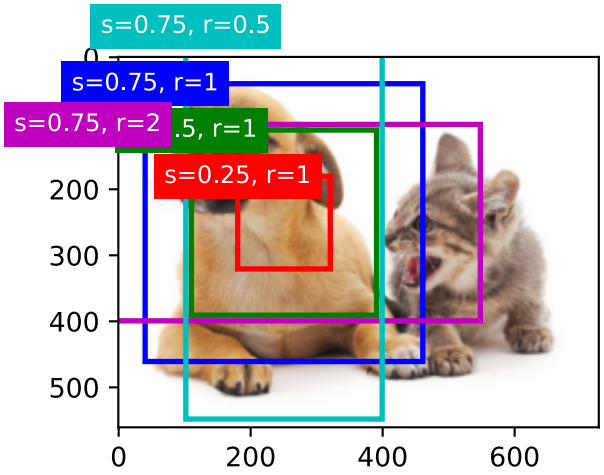
(continues on next page)

(continued from previous page)

```
    def make_list(obj, default_values=None):
        if obj is None:
            obj = default_values
        elif not isinstance(obj, (list, tuple)):
            obj = [obj]
        return obj

    labels = make_list(labels)
    colors = make_list(colors, ['b', 'g', 'r', 'm', 'c'])
    for i, bbox in enumerate(bboxes):
        color = colors[i % len(colors)]
        rect = d2l.bbox_to_rect(bbox.detach().numpy(), color)
        axes.add_patch(rect)
        if labels and len(labels) > i:
            text_color = 'k' if color == 'w' else 'w'
            axes.text(rect.xy[0], rect.xy[1], labels[i],
                      va='center', ha='center', fontsize=9, color=text_color,
                      bbox=dict(facecolor=color, lw=0))
```

Gördüğümüz gibi, boxes değişkenindeki x ve y eksenlerinin koordinat değerleri sırasıyla imgenin genişliği ve yüksekliğine bölünmüştür. Çapa kutularını çizerken, orijinal koordinat değerlerini geri yüklememiz gerekir; bu nedenle, aşağıda bbox_scale değişkeni tanımlarız. Şimdi, resimde (250, 250) merkezli tüm çapa kutularını çizebiliriz. Gördüğünüz gibi, 0.75 ölçekli ve 1 en boy oranına sahip mavi çapa kutusu, imgedeki köpeği iyi çevreler.

```
d2l.set_figsize()
bbox_scale = torch.tensor((w, h, w, h))
fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, boxes[250, 250, :, :] * bbox_scale,
            ['s=0.75, r=1', 's=0.5, r=1', 's=0.25, r=1', 's=0.75, r=2',
             's=0.75, r=0.5'])
```



13.4.2 Birleşim (IoU) üzerinde Kesişim

Sadece bir çapa kutusunun köpeği imgede "iyi" çevrelediğini belirttik. Nesnenin gerçek referans değeri kuşatan kutusu biliniyorsa, burada "iyi" nasıl ölçülebilir? Sezgisel olarak, çapa kutusu ile gerçek referans değeri kuşatan kutu arasındaki benzerliği ölçebiliriz. *Jaccard indeksi*nin iki küme arasındaki benzerliği ölçebileceğini biliyoruz. $\mathcal{A}$ ve $\mathcal{B}$ kümeleri verildiğinde, bunların Jaccard indeksi kesişimlerinin boyutunun birleşimlerinin boyutuna bölünmesidir:

$$J(\mathcal{A},\mathcal{B}) = \frac{|\mathcal{A} \cap \mathcal{B}|}{|\mathcal{A} \cup \mathcal{B}|}. \tag{13.4.2}$$

Aslında, herhangi bir kuşatan kutunun piksel alanını bir piksel kümesi olarak düşünebiliriz. Bu şekilde, iki kuşatan kutunun benzerliğini piksel kümelerinin Jaccard indeksi ile ölçebiliriz. İki kuşatan kutu için, Jaccard indeksini genellikle Fig. 13.4.1 içinde gösterildiği gibi, kesişme alanlarının birleşme alanlarına oranı olan *bileşim üzerinden kesişme* (*IoU*) olarak adlandırırız. Bir IoU aralığı 0 ile 1 arasındadır: 0 iki kuşatan kutunun hiç örtüşmediği anlamına gelirken 1, iki kuşatan kutunun eşit olduğunu gösterir.

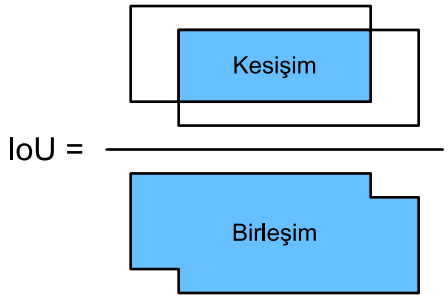


Fig. 13.4.1: IoU, kesişim alanının iki kuşatan kutunun birleşim alanına oranıdır.

Bu bölümün geri kalanında, çapa kutuları ile gerçek referans değeri kuşatan kutular ve farklı çapa kutuları arasındaki benzerliği ölçmek için IoU kullanacağız. İki çapa veya kuşatan kutular listesi göz önüne alındığında, aşağıdaki `box_iou`, bu iki listede çift yönlü IoU hesaplar.

```
#@save
def box_iou(boxes1, boxes2):
    """İki çapa veya kuşatan kutu listesinde ikili IoU hesaplayın."""
    box_area = lambda boxes: ((boxes[:, 2] - boxes[:, 0]) *
                              (boxes[:, 3] - boxes[:, 1]))
    # Shape of `boxes1`, `boxes2`, `areas1`, `areas2`: (no. of boxes1, 4),
    # (no. of boxes2, 4), (no. of boxes1,), (no. of boxes2,)
    areas1 = box_area(boxes1)
    areas2 = box_area(boxes2)
    # Shape of `inter_upperlefts`, `inter_lowerrights`, `inters`: (no. of
    # boxes1, no. of boxes2, 2)
    inter_upperlefts = torch.max(boxes1[:, None, :2], boxes2[:, :2])
    inter_lowerrights = torch.min(boxes1[:, None, 2:], boxes2[:, 2:])
    inters = (inter_lowerrights - inter_upperlefts).clamp(min=0)
    # Shape of `inter_areas` and `union_areas`: (no. of boxes1, no. of boxes2)
    inter_areas = inters[:, :, 0] * inters[:, :, 1]
    union_areas = areas1[:, None] + areas2 - inter_areas
    return inter_areas / union_areas
```

13.4.3 Eğitim Verilerinde Çapa Kutuları Etiketleme

Bir eğitim veri kümesinde, her çapa kutusunu bir eğitim örneği olarak değerlendiririz. Bir nesne algılama modelini eğitmek için, her bir çapa kutusu için *sınıf* ve *uzaklık* etiketlerine ihtiyacımız var; burada birincisi, çapa kutusuyla ilgili nesnenin sınıfıdır ve ikincisi ise çapa kutusuna göre gerçek referans değeri kuşatan kutunun uzaklığıdır. Tahmin sırasında, her imge için birden çok çapa kutusu oluşturur, tüm bağlantı kutuları için sınıfları ve uzaklıkları tahmin eder, tahmini kuşatan kutuları elde etmek için konumlarını tahmin edilen uzaklıklara göre ayarlarız ve son olarak yalnızca belirli kriterleri karşılayan tahmini kuşatan kutuları çıktılarız.

Bildiğimiz gibi, bir nesne algılama eğitim kümesi, *gerçek referans değeri kuşatan kutular* ve çevrelenmiş nesnelerin sınıfları için etiketlerle birlikte gelir. Oluşturulan herhangi bir *çapa kutusu* etiketlemek için, çapa kutusuna en yakın olan *atanmış* gerçek referans değeri kuşatan kutunun etiketlenmiş konumuna ve sınıfına başvururuz. Aşağıda, çapa kutularına en yakın gerçek referans değeri kuşatan kutuları atamak için bir algoritma tanımlıyoruz.

Çapa Kutularına Gerçek Referans Değeri Kuşatan Kutuları Atama

Bir imge göz önüne alındığında, çapa kutularının $A_1, A_2, \ldots, A_{n_a}$ ve gerçek referans değeri kuşatan kutuların $B_1, B_2, \ldots, B_{n_b}$ olduğunu varsayalım, burada $n_a \geq n_b$. $i.$ satırında ve $j.$ sütununda x_{ij} elemanı, A_i çapa kutusunun ve B_j gerçek referans değer kuşatan kutusunun IoU değeri olan bir $\mathbf{X} \in \mathbb{R}^{n_a \times n_b}$ matrisini tanımlayalım. Algoritma aşağıdaki adımlardan oluşur:

1. $\mathbf{X}$ matrisindeki en büyük elemanı bulun ve satır ve sütun indekslerini sırasıyla i_1 ve j_1 olarak belirtin. Daha sonra B_{j_1} gerçek referans değeri kuşatan kutu A_{i_1} çapa kutusuna atanır. Bu oldukça sezgiseldir, çünkü A_{i_1} ve B_{j_1}, tüm çapa kutuları ve gerçek referans değeri kuşatan kutular arasında en yakın olanlardır. İlk atamadan sonra, $\mathbf{X}$ matrisindeki $i_1.$ satırındaki ve $j_1.$ sütunundaki tüm öğeleri atın.
2. $\mathbf{X}$ matrisinde kalan elemanların en büyüğünü bulun ve satır ve sütun indekslerini sırasıyla i_2 ve j_2 olarak belirtin. B_{j_2} gerçek referans değeri kuşatan kutusunu A_{i_2} çapa kutusuna atarız ve $\mathbf{X}$ matrisindeki $i_2.$ satırındaki ve $j_2.$ sütunundaki tüm öğeleri atarız.
3. Bu noktada, $\mathbf{X}$ matrisindeki iki satır ve iki sütundaki öğeler atılmıştır. $\mathbf{X}$ matrisindeki n_b sütunundaki tüm öğeler atılana kadar devam ederiz. Şu anda, n_b çapa kutusunun her birine bir gerçek referans değeri kuşatan kutu atadık.
4. Sadece kalan $n_a - n_b$ tane çapa kutusundan geçiş yapın. Örneğin, herhangi bir A_i çapa kutusu verildiğinde, $i.$ matrisinin $\mathbf{X}$ satırı boyunca A_i ile en büyük IoU'ya sahip B_j gerçek referans değeri kuşatan kutusunu bulun ve yalnızca bu IoU önceden tanımlanmış bir eşikten büyükse B_j öğesini A_i öğesine atayın.

Yukarıdaki algoritmayı somut bir örnek kullanarak gösterelim. Fig. 13.4.2 içinde(solda) gösterildiği gibi, $\mathbf{X}$ matrisindeki maksimum değerin x_{23} olduğunu varsayarak x_{23}, B_3 numaralı gerçek referans değeri kuşatan kutusunu A_2 numaralı çapa kutusuna atarız. Daha sonra, matrisin satır 2 ve sütun 3'teki tüm elemanlarını atırız, kalan elemanlarda (gölgeli alan) en büyük x_{71}'i buluruz ve A_7 çapa kutusuna B_1 numaralı gerçek referans değeri kuşatan kutuyu atarız. Daha sonra, Fig. 13.4.2 (ortada) içinde gösterildiği gibi, matrisin 7. satırı ve 1. sütunundaki tüm öğeleri atın, kalan elemanlardaki (gölgeli alan) en büyük x_{54} öğesini bulun ve B_4 gerçek referans değeri kuşatan kutusunu A_5 çapa kutusuna atayın. Son olarak, Fig. 13.4.2 içinde (sağda) gösterildiği gibi, matrisin 5. satırı ve 4. sütunundaki tüm elemanları atın, kalan elemanlardaki (gölgeli alan) en büyük x_{92} değerini bulun ve B_2 gerçek referans değeri kuşatan kutusunu A_9 çapa kutusuna atayın. Bundan sonra, yalnızca geri kalan A_1, A_3, A_4, A_6, A_8 çapa kutularından geçmemiz ve eşiğe göre gerçek

referans değeri kuşatan kutular atanıp atanmayacağına karar vermemiz gerekiyor.

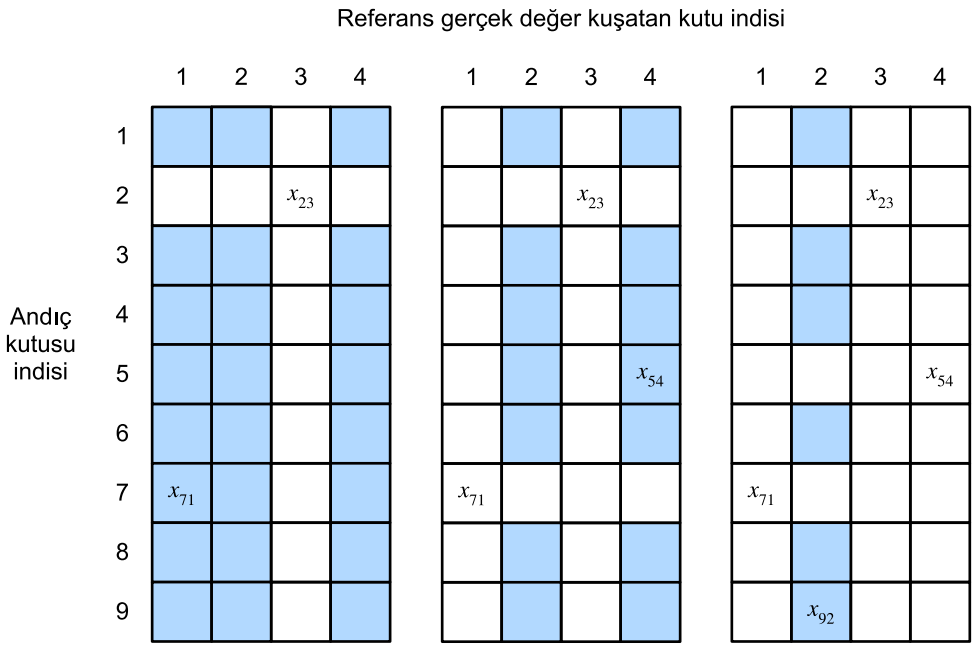


Fig. 13.4.2: Çapa kutularına gerçek referans değeri kuşatan kutular atama.

Bu algoritma aşağıdaki `assign_anchor_to_bbox` işlevinde uygulanır.

```
#@save
def assign_anchor_to_bbox(ground_truth, anchors, device, iou_threshold=0.5):
    """En yakın gerçek referans değeri kuşatan kutuları çapa kutularına atayın."""
    num_anchors, num_gt_boxes = anchors.shape[0], ground_truth.shape[0]
    # i. satır ve j. sütundaki x_ij öğesi, i çapa kutusunun ve j gerçek referans değeri␣
↪kuşatan kutusunu IoU'sudur.
    jaccard = box_iou(anchors, ground_truth)
    # Her çapa için atanmış gerçek referans değeri kuşatan kutuyu tutmak için
    # tensörü ilklet
    anchors_bbox_map = torch.full((num_anchors,), -1, dtype=torch.long,
                                  device=device)
    # Eşiğe göre gerçek referans değeri kuşatan kutuları atayın
    max_ious, indices = torch.max(jaccard, dim=1)
    anc_i = torch.nonzero(max_ious >= 0.5).reshape(-1)
    box_j = indices[max_ious >= 0.5]
    anchors_bbox_map[anc_i] = box_j
    col_discard = torch.full((num_anchors,), -1)
    row_discard = torch.full((num_gt_boxes,), -1)
    for _ in range(num_gt_boxes):
        max_idx = torch.argmax(jaccard)  # En büyük IoU'yu bul
        box_idx = (max_idx % num_gt_boxes).long()
        anc_idx = (max_idx / num_gt_boxes).long()
        anchors_bbox_map[anc_idx] = box_idx
        jaccard[:, box_idx] = col_discard
        jaccard[anc_idx, :] = row_discard
    return anchors_bbox_map
```

Etiketleme Sınıfları ve Uzaklıklar

Artık her çapa kutusu için sınıfı ve uzaklığı etiketleyebiliriz. Bir A çapa kutusunun bir B gerçek referans değer kuşatan kutusuna atandığını varsayalım. Bir yandan, A çapa kutusu sınıfı B'ninkiyle aynı olarak etiketlenecektir. Öte yandan, A çapa kutusunun uzaklığı, B ve A arasındaki merkezi koordinatlar arasındaki göreli konuma göre bu iki kutu arasındaki göreli boyutla birlikte etiketlenecektir. Veri kümesindeki farklı kutuların değişen konumları ve boyutları verildiğinde, sığdırılması daha kolay olan daha düzgün dağıtılmış ofsetlere yol açabilecek bu göreli konumlara ve boyutlara dönüşümler uygulayabiliriz. Burada genel bir dönüşümü tanımlıyoruz. A ve B'nin merkezi koordinatları (x_a, y_a) ve (x_b, y_b) olarak verildiğinde, sırasıyla genişlikleri w_a ve w_b ve yükseklikleri h_a ve h_b olarak verilir. A ofsetini şu şekilde etiketleyebiliriz:

$$\left(\frac{\frac{x_b - x_a}{w_a} - \mu_x}{\sigma_x}, \frac{\frac{y_b - y_a}{h_a} - \mu_y}{\sigma_y}, \frac{\log \frac{w_b}{w_a} - \mu_w}{\sigma_w}, \frac{\log \frac{h_b}{h_a} - \mu_h}{\sigma_h} \right), \tag{13.4.3}$$

burada sabitlerin varsayılan değerleri $\mu_x = \mu_y = \mu_w = \mu_h = 0, \sigma_x = \sigma_y = 0.1$ ve $\sigma_w = \sigma_h = 0.2$'dir. Bu dönüştürme, aşağıda `offset_boxes` işlevinde uygulanmaktadır.

```
#@save
def offset_boxes(anchors, assigned_bb, eps=1e-6):
    """Çapa kutu ofsetlerini dönüştür"""
    c_anc = d2l.box_corner_to_center(anchors)
    c_assigned_bb = d2l.box_corner_to_center(assigned_bb)
    offset_xy = 10 * (c_assigned_bb[:, :2] - c_anc[:, :2]) / c_anc[:, 2:]
    offset_wh = 5 * torch.log(eps + c_assigned_bb[:, 2:] / c_anc[:, 2:])
    offset = torch.cat([offset_xy, offset_wh], axis=1)
    return offset
```

Bir çapa kutusuna bir gerçek referans değeri kuşatan kutu atanmamışsa, sadece çapa kutusunun sınıfını "arka plan" olarak etiketleriz. Sınıfları arka plan olan çapa kutuları genellikle *negatif* çapa kutuları olarak adlandırılır ve geri kalanı ise *pozitif* çapa kutuları olarak adlandırılır. Aşağıdaki `multibox_target` işlevini gerçek referans değeri kuşatan kutuları (`labels` bağımsız değişkeni) kullanarak sınıfları ve çapa kutuları (`anchors` bağımsız değişkeni) için uzaklıkları etiketlemek için uyguluyoruz. Bu işlev, arka plan sınıfını sıfıra ayarlar ve yeni bir sınıfın tamsayı dizinini bir artırır.

```
#@save
def multibox_target(anchors, labels):
    """Gerçeği referans değeri kuşatan kutuları kullanarak çapa kutularını etiketleyin."""
    batch_size, anchors = labels.shape[0], anchors.squeeze(0)
    batch_offset, batch_mask, batch_class_labels = [], [], []
    device, num_anchors = anchors.device, anchors.shape[0]
    for i in range(batch_size):
        label = labels[i, :, :]
        anchors_bbox_map = assign_anchor_to_bbox(
            label[:, 1:], anchors, device)
        bbox_mask = ((anchors_bbox_map >= 0).float().unsqueeze(-1)).repeat(
            1, 4)
        # Sınıf etiketlerini ve atanmış kuşatan kutu koordinatlarını sıfırlarla ilklet
        class_labels = torch.zeros(num_anchors, dtype=torch.long,
                                   device=device)
        assigned_bb = torch.zeros((num_anchors, 4), dtype=torch.float32,
                                  device=device)
        # Çapa kutularının sınıflarını kendilerine atanan gerçek referans değeri
```

(continues on next page)

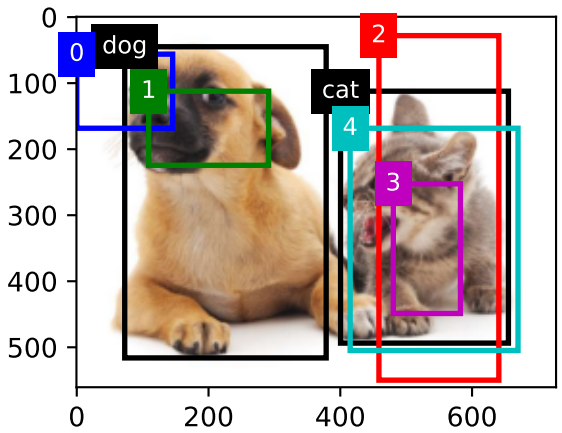
(continued from previous page)

```
        # kuşatan kutularını kullanarak etiketleyin. Bir çapa kutusu
        # atanmamışsa, sınıfını arka plan olarak etiketliyoruz (değer sıfır
        # olarak kalıyor)
        indices_true = torch.nonzero(anchors_bbox_map >= 0)
        bb_idx = anchors_bbox_map[indices_true]
        class_labels[indices_true] = label[bb_idx, 0].long() + 1
        assigned_bb[indices_true] = label[bb_idx, 1:]
        # Ofset dönüşümü
        offset = offset_boxes(anchors, assigned_bb) * bbox_mask
        batch_offset.append(offset.reshape(-1))
        batch_mask.append(bbox_mask.reshape(-1))
        batch_class_labels.append(class_labels)
    bbox_offset = torch.stack(batch_offset)
    bbox_mask = torch.stack(batch_mask)
    class_labels = torch.stack(batch_class_labels)
    return (bbox_offset, bbox_mask, class_labels)
```

Bir Örnek

Çapa kutusu etiketlemeyi somut bir örnekle gösterelim. İlk öğenin sınıf olduğu (köpek için 0 ve kedi için 1) ve kalan dört öğenin sol üst köşede ve sağ alt köşedeki (x, y)-ekseni koordinatlarının (aralık 0 ile 1 arasındadır) olduğu, yüklenen imgedeki köpek ve kedi için gerçek referans değeri kuşatan kutular tanımlarız. Ayrıca sol üst köşenin ve sağ alt köşenin koordinatlarını kullanarak etiketlenecek beş çapa kutusu oluşturuyoruz: $A_0, \ldots, A_4$ (indeks 0'dan başlar). Sonra resimdeki bu gerçek referans değeri kuşatan kutuları ve çapa kutularını çiziyoruz.

```
ground_truth = torch.tensor([[0, 0.1, 0.08, 0.52, 0.92],
                         [1, 0.55, 0.2, 0.9, 0.88]])
anchors = torch.tensor([[0, 0.1, 0.2, 0.3], [0.15, 0.2, 0.4, 0.4],
                    [0.63, 0.05, 0.88, 0.98], [0.66, 0.45, 0.8, 0.8],
                    [0.57, 0.3, 0.92, 0.9]])

fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, ground_truth[:, 1:] * bbox_scale, ['dog', 'cat'], 'k')
show_bboxes(fig.axes, anchors * bbox_scale, ['0', '1', '2', '3', '4']);
```



Yukarıda tanımlanan `multibox_target` işlevini kullanarak, köpek ve kedi için bu çapa kutularının sınıflarını ve ofsetlerini gerçek referans değeri kuşatan kutulara dayanarak etiketleyebiliriz. Bu örnekte, arka plan, köpek ve kedi sınıflarının indeksleri sırasıyla 0, 1 ve 2'dir. Aşağıda, çapa kutularının ve gerçek referans değeri kuşatan kutuların örnekleri için bir boyut ekliyoruz.

```
labels = multibox_target(anchors.unsqueeze(dim=0),
                         ground_truth.unsqueeze(dim=0))
```

Döndürülen sonuçta, hepsi tensör formatında olan üç öğe vardır. Üçüncü öğe, girdi çapa kutularının etiketli sınıflarını içerir.

Aşağıdaki döndürülen sınıf etiketlerini, çapa kutusu ve resimdeki gerçek referans değeri kuşatan kutu konumlarına göre analiz edelim. İlk olarak, tüm çapa kutuları ve gerçek referans değeri kuşatan kutu çiftleri arasında, A_4 bağlantı kutusunun ve kedinin gerçek referans değeri kuşatan kutusunun IoU'su en büyüğüdür. Böylece, A_4'nın sınıfı kedi olarak etiketlenmiştir. A_4 veya kedinin gerçek referans değeri kuşatan kutusunu içeren çiftleri çıkarırken, geri kalanlar arasında A_1 çapa kutusunun ve köpeğin gerçek referans değeri kuşatan kutusunun çifti en büyük IoU'ya sahiptir. Büylece A_1 sınıfı köpek olarak etiketlenir. Ardından, kalan üç etiketlenmemiş çapa kutusundan geçmemiz gerekiyor: A_0, A_2 ve A_3. A_0 için, en büyük IoU'ya sahip gerçek referans değeri kuşatan kutunun sınıfı köpektir, ancak IoU önceden tanımlanmış eşiğin (0.5) altındadır, bu nedenle sınıf arka plan olarak etiketlenir; A_2 için, en büyük IoU'ya sahip gerçek referans değeri kuşatan kutunun sınıfı kedidir ve IoU eşiği aştığından sınıf kedi olarak etiketlenir; A_3 için, en büyük IoU'ya sahip gerçek referans değeri kuşatan kutunun sınıfı kedidir, ancak değer eşiğin altındadır, bu nedenle sınıf arka plan olarak etiketlenir.

```
labels[2]
```

```
tensor([[0, 1, 2, 0, 2]])
```

Döndürülen ikinci öğe şeklin bir maske değişkenidir (toplu iş boyutu, çapa kutularının sayısının dört katı). Maske değişkenindeki her dört öğe, her bir çapa kutusunun dört ofset değerine karşılık gelir. Arka plan algılama umurumuzda olmadığından, bu negatif sınıfın ofsetleri amaç işlevini etkilememelidir. Eleman yönlü çarpımlar sayesinde, maske değişkenindeki sıfırlar, amaç işlevini hesaplamadan önce negatif sınıf ofsetlerini filtreler.

```
labels[1]
```

```
tensor([[0., 0., 0., 0., 1., 1., 1., 1., 1., 1., 1., 1., 0., 0., 0., 0., 1., 1.,
         1., 1.]])
```

İlk döndürülen öğe, her çapa kutusu için etiketlenmiş dört ofset değeri içerir. Negatif sınıf çapa kutularının ofsetlerinin sıfır olarak etiketlendiğini unutmayın.

```
labels[0]
```

```
tensor([[-0.00e+00, -0.00e+00, -0.00e+00, -0.00e+00,  1.40e+00,  1.00e+01,
          2.59e+00,  7.18e+00, -1.20e+00,  2.69e-01,  1.68e+00, -1.57e+00,
         -0.00e+00, -0.00e+00, -0.00e+00, -0.00e+00, -5.71e-01, -1.00e+00,
          4.17e-06,  6.26e-01]])
```

13.4.4 Maksimum Olmayanı Bastırma ile Kuşatan Kutuları Tahmin Etme

Tahmin esnasında, imge için birden çok çapa kutusu oluşturur ve her biri için sınıfları ve ofsetleri tahmin ederiz. Bir *tahmin edilen kuşatan kutu* böylece, tahmin edilen ofseti olan bir çapa kutusuna göre elde edilir. Aşağıda, girdiler olarak çapaları ve ofset tahminlerini alan ve tahmin edilen kuşatan kutu koordinatlarını döndürmek için ters ofset dönüşümleri uygulayan `offset_inverse` işlevini uyguluyoruz.

```
#@save
def offset_inverse(anchors, offset_preds):
    """Tahmin edilen uzaklıklara sahip çapa kutularına dayalı kuşatan kutuları tahmin edin."""
    anc = d2l.box_corner_to_center(anchors)
    pred_bbox_xy = (offset_preds[:, :2] * anc[:, 2:] / 10) + anc[:, :2]
    pred_bbox_wh = torch.exp(offset_preds[:, 2:] / 5) * anc[:, 2:]
    pred_bbox = torch.cat((pred_bbox_xy, pred_bbox_wh), axis=1)
    predicted_bbox = d2l.box_center_to_corner(pred_bbox)
    return predicted_bbox
```

Çok sayıda bağlantı kutusu olduğunda, aynı nesneyi çevreleyen birçok benzer (önemli ölçüde örtüşen) tahmin edilen kuşatan kutular potansiyel olarak çıktılanabilir. Çıktıyı basitleştirmek için, aynı nesneye ait benzer tahmin edilen kuşatan kutuları *maksimum olmayanı bastırma (non-maximum suppression)* (NMS) kullanarak birleştirebiliriz.

Şimdi maksimum olmayanı bastırma nasıl çalışır anlayalım. Tahmin edilen bir kuşatan kutu B için nesne algılama modeli her sınıf için tahmin edilen olasılığı hesaplar. p ile tahmin edilen en büyük olasılığı ifade edersek, bu olasılığa karşılık gelen sınıf, B için tahmin edilen sınıftır. Daha özel belirtirsek, p'yi tahmini kuşatan kutu B'nin *güveni* (skoru) olarak adlandırıyoruz. Aynı imgede, tahmin edilen tüm arka plan olmayan kuşatan kutular, bir L listesi oluşturmak için azalan düzende güvene göre sıralanır. Ardından, aşağıdaki adımlarda sıralanmış listeyi L'yi değiştiriyoruz.:

1. L'ten en yüksek güvenle tahmin edilen kuşatan kutu B_1'yi temel olarak seçin ve L'den B_1 ile IoU'su önceden tanımlanmış ϵ eşiğini aşan temel dışı tahmin edilen tüm kuşatan kutuları kaldırın. Bu noktada, L tahmin edilen kuşatan kutuyu en yüksek güvenle tutar, ancak buna çok benzer olan başkalarını atar. Özetle, *maksimum olmayan* güven puanı olanlar *bastırılır*.
2. Başka bir temel olarak L'dan ikinci en yüksek güvenle tahmin edilen kuşatan kutu B_2'yi seçin ve L'dan B_2 ile IoU ϵ'u aşan temel olmayan tüm tahmini kuşatan kutuları kaldırın.
3. L'deki tüm tahmini kuşatan kutuları temel olarak kullanılıncaya kadar yukarıdaki işlemi yineleyin. Şu anda, L'deki tahmini kuşatan kutuların herhangi bir çiftinin IoU'su ϵ eşiğinin altındadır; bu nedenle, hiçbir çift birbiriyle çok benzer değildir.
4. L listedeki tüm tahmini kuşatan kutuları çıktı olarak verin.

Aşağıdaki `nms` işlevi, güven skorlarını azalan sırada sıralar ve indekslerini döndürür.

```
#@save
def nms(boxes, scores, iou_threshold):
    """Tahmin edilen kuşatan kutuların güven puanlarını sıralama."""
    B = torch.argsort(scores, dim=-1, descending=True)
    keep = []  # Tutulacak tahmini sınırlayıcı kutuların endeksleri
    while B.numel() > 0:
        i = B[0]
```

(continues on next page)

(continued from previous page)

```
        keep.append(i)
        if B.numel() == 1: break
        iou = box_iou(boxes[i, :].reshape(-1, 4),
                      boxes[B[1:], :].reshape(-1, 4)).reshape(-1)
        inds = torch.nonzero(iou <= iou_threshold).reshape(-1)
        B = B[inds + 1]
    return torch.tensor(keep, device=boxes.device)
```

Aşağıdaki `multibox_detection` işlevi kuşatan kutuları tahmin ederken maksimum olmayanı bastırmayı uygulamak için tanımlıyoruz. Uygulamanın biraz karmaşık olduğunu görürseniz endişelenmeyin: Uygulamadan hemen sonra somut bir örnekle nasıl çalıştığını göstereceğiz.

```
#@save
def multibox_detection(cls_probs, offset_preds, anchors, nms_threshold=0.5,
                       pos_threshold=0.009999999):
    """Maksimum olmayan bastırma kullanarak kuşatan kutuları tahmin edin."""
    device, batch_size = cls_probs.device, cls_probs.shape[0]
    anchors = anchors.squeeze(0)
    num_classes, num_anchors = cls_probs.shape[1], cls_probs.shape[2]
    out = []
    for i in range(batch_size):
        cls_prob, offset_pred = cls_probs[i], offset_preds[i].reshape(-1, 4)
        conf, class_id = torch.max(cls_prob[1:], 0)
        predicted_bb = offset_inverse(anchors, offset_pred)
        keep = nms(predicted_bb, conf, nms_threshold)
        # Tüm "tutmayan" dizinleri bulun ve sınıfı arka plana ayarlayın
        all_idx = torch.arange(num_anchors, dtype=torch.long, device=device)
        combined = torch.cat((keep, all_idx))
        uniques, counts = combined.unique(return_counts=True)
        non_keep = uniques[counts == 1]
        all_id_sorted = torch.cat((keep, non_keep))
        class_id[non_keep] = -1
        class_id = class_id[all_id_sorted]
        conf, predicted_bb = conf[all_id_sorted], predicted_bb[all_id_sorted]
        # Burada `pos_threshold`, pozitif (arka plan dışı) tahminler için bir eşik değeridir
        below_min_idx = (conf < pos_threshold)
        class_id[below_min_idx] = -1
        conf[below_min_idx] = 1 - conf[below_min_idx]
        pred_info = torch.cat((class_id.unsqueeze(1),
                               conf.unsqueeze(1),
                               predicted_bb), dim=1)
        out.append(pred_info)
    return torch.stack(out)
```

Şimdi yukarıdaki uygulamaları dört çapa kutusuna sahip somut bir örneğe uygulayalım. Basitlik için, tahmin edilen ofsetlerin tümünün sıfır olduğunu varsayıyoruz. Bu, tahmin edilen kuşatan kutuların çapa kutuları olduğu anlamına gelir. Arka plan, köpek ve kedi arasındaki her sınıf için, tahmin edilen olasılığını da tanımlıyoruz.

```
anchors = torch.tensor([[0.1, 0.08, 0.52, 0.92], [0.08, 0.2, 0.56, 0.95],
                        [0.15, 0.3, 0.62, 0.91], [0.55, 0.2, 0.9, 0.88]])
offset_preds = torch.tensor([0] * anchors.numel())
cls_probs = torch.tensor([[0] * 4,  # Tahmini arka plan olabilirliği
```

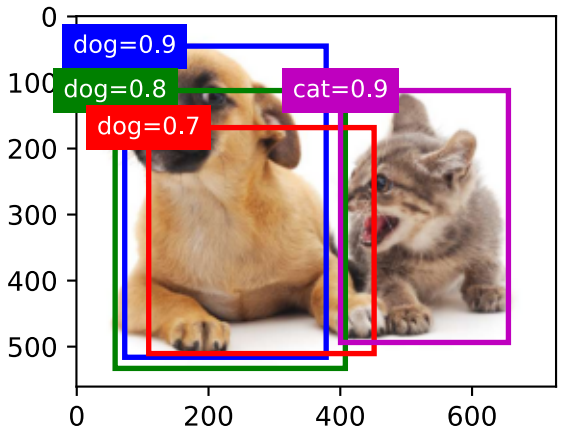
(continues on next page)

(continued from previous page)

```
                    [0.9, 0.8, 0.7, 0.1],  # Tahmini köpek olabilirliği
                    [0.1, 0.2, 0.3, 0.9]])  #  Tahmini kedi olabilirliği
```

Bu tahmini kuşatan kutuları resimdeki güvenleriyle çizebiliriz.

```
fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, anchors * bbox_scale,
            ['dog=0.9', 'dog=0.8', 'dog=0.7', 'cat=0.9'])
```



Şimdi, eşiğin 0.5'e ayarlandığı maksimum olmayanı bastırmayı gerçekleştirmek için `multibox_detection` işlevini çağırabiliriz. Tensör girdideki örnekler için bir boyut eklediğimizi unutmayın.

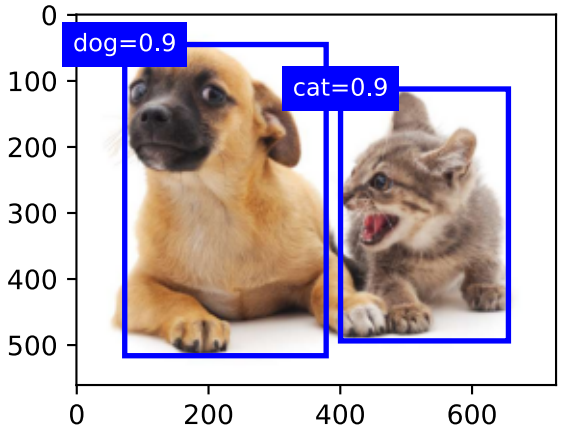
Döndürülen sonucun şeklinin (parti boyutu, çapa kutusu sayısı, 6) olduğunu görebiliriz. En iç boyuttaki altı eleman, aynı tahmini kuşatan kutunun çıktı bilgilerini verir. İlk eleman, 0'dan başlayan tahmini sınıf dizinidir (0 köpektir ve 1 kedidir). -1 değeri, arka planı veya maksimum olmayanı bastırmada kaldırmayı gösterir. İkinci eleman, tahmin edilen kuşatan kutunun güvenidir. Kalan dört öğe, tahmini kuşatan kutunun sırasıyla sol üst köşesinin ve sağ alt köşesinin (x, y) eksen koordinatlarıdır (aralık 0 ile 1 arasındadır).

```
output = multibox_detection(cls_probs.unsqueeze(dim=0),
                            offset_preds.unsqueeze(dim=0),
                            anchors.unsqueeze(dim=0),
                            nms_threshold=0.5)
output
```

```
tensor([[[ 0.00,  0.90,  0.10,  0.08,  0.52,  0.92],
         [ 1.00,  0.90,  0.55,  0.20,  0.90,  0.88],
         [-1.00,  0.80,  0.08,  0.20,  0.56,  0.95],
         [-1.00,  0.70,  0.15,  0.30,  0.62,  0.91]]])
```

Sınıfı -1 olan bu tahmini kuşatan kutuları kaldırdıktan sonra maksimum olmayanı bastırma ile tutulan son tahmini kuşatan kutuyu çıktı yapabiliriz.

```
fig = d2l.plt.imshow(img)
for i in output[0].detach().numpy():
    if i[0] == -1:
        continue
    label = ('dog=', 'cat=')[int(i[0])] + str(i[1])
    show_bboxes(fig.axes, [torch.tensor(i[2:]) * bbox_scale], label)
```



Pratikte, maksimum olmayanı bastırmayı gerçekleştirmeden önce bile daha düşük güvenli tahmini kuşatan kutuları kaldırabilir, böylece bu algoritmadaki hesaplamayı azaltabiliriz. Aynı zamanda maksimum olmayanı bastırmanın çıktısını da sonradan işleyebiliriz, örneğin, nihai çıktıda yalnızca daha yüksek güvenli sonuçları tutarabiliriz.

13.4.5 Özet

- İmgenin her pikselinde ortalanmış farklı şekillere sahip çapa kutuları oluşturuyoruz.
- Jaccard indeksi olarak da bilinen birleşme (IoU) üzerindeki kesişme, iki kuşatan kutunun benzerliğini ölçer. Kesişme alanlarının birleşme alanlarına oranıdır.
- Bir eğitim kümesinde, her bir çapa kutusu için iki tip etikete ihtiyacımız vardır. Biri, çapa kutusundaki ilgili nesnenin sınıfıdır ve diğeri ise çapa kutusuna göre gerçek referans değeri kuşatan kutunun ofsetidir.
- Tahmin sırasında, benzer tahmini kuşatan kutuları kaldırmak için maksimum olmayanı bastırma (NMS) kullanabiliriz ve böylece çıktıyı basitleştiririz.

13.4.6 Alıştırmalar

1. `multibox_prior` işlevinde `sizes` ve `ratios` değerlerini değiştirin. Oluşturulan çapa kutularındaki değişiklikler nelerdir?
2. 0.5 IoU ile iki kuşatan kutu oluşturun ve görselleştirin. Birbirleriyle nasıl örtüşürler?
3. Section 13.4.3 ve Section 13.4.4 içindeki `anchors` değişkenini değiştirin. Sonuçlar nasıl değişir?
4. Maksimum olmayanı bastırma, tahmini kuşatan kutuları *kaldırarak* bastıran açgözlü bir algoritmadır. Bu kaldırılmış olanlardan bazılarının gerçekten yararlı olması mümkün mü?

Yumuşak bastırma için bu algoritma nasıl değiştirilebilir? Soft-NMS (Bodla *et al.*, 2017)'e başvurabilirsiniz.

5. El yapımı olmaktan ziyade, maksimum olmayanı bastırma öğrenilebilir mi?

Tartışmalar[177]

13.5 Çoklu Ölçekli Nesne Algılama

Section 13.4 içinde, bir girdi imgesinin her pikselinde ortalanmış birden çok çapa kutusu oluşturduk. Esasen bu çapa kutuları imgenin farklı bölgelerinin örneklerini temsil eder. Ancak, *her* piksel için oluşturulmuşlarsa hesaplayamayacak kadar çok çapa kutusu elde edebiliriz. Bir 561×728'lik girdi imgesi düşünün. Merkez alarak her piksel için farklı şekillerde beş çapa kutusu oluşturulursa, iki milyondan fazla çapa kutusunun ($561 \times 728 \times 5$) imge üzerinde etiketlenmesi ve tahmin edilmesi gerekir.

13.5.1 Çoklu Ölçekli Çapa Kutuları

Bir imgedeki çapa kutularını azaltmanın zor olmadığını fark edebilirsiniz. Örneğin, üzerlerinde ortalanmış çapa kutuları oluşturmak için girdi imgesindeki piksellerin küçük bir bölümünü tekdüze bir şekilde örnekleyebiliriz. Buna ek olarak, farklı ölçeklerde farklı boyutlarda farklı çapa kutuları üretebiliriz. Sezgisel olarak, daha küçük nesnelerin imgede daha büyük olanlardan daha fazla görünme olasılığı daha yüksektir. Örnek olarak, 1×1, 1×2 ve 2×2 nesneler 2×2'lik imgede sırasıyla 4, 2 ve 1 olası şekilde görünebilir. Bu nedenle, daha küçük nesneleri algılamak için daha küçük çapa kutuları kullanırken daha fazla bölgeyi örnekleyebiliriz, daha büyük nesneler için daha az bölgeyi örnekleyebiliriz.

Birden çok ölçekte çapa kutuları nasıl oluşturulacağını göstermek için bir imgeyi okuyalım. Yüksekliği ve genişliği sırasıyla 561 ve 728 piksel olsun.

```
%matplotlib inline
import torch
from d2l import torch as d2l

img = d2l.plt.imread('../img/catdog.jpg')
h, w = img.shape[:2]
h, w
```

```
(561, 728)
```

Section 6.2 içinde, evrişimli bir katmanın iki boyutlu bir dizi çıktısını bir öznitelik haritası olarak adlandırdığımızı hatırlayın. Öznitelik haritası şeklini tanımlayarak, herhangi bir imgedeki düzgün örneklenmiş çapa kutularının merkezlerini belirleyebiliriz.

`display_anchors` işlevi aşağıda tanımlanmıştır. Çapa kutusu merkezi olarak her birim (piksel) ile öznitelik haritasında (`fmap`) çapa kutuları (`anchors`) oluştururuz. Çapa kutularındaki (x, y) eksen koordinat değerleri (`anchors`), öznitelik haritasının genişlik ve yüksekliğine bölünmüş olduğundan (`fmap`), ki bu değerler 0 ve 1 arasındadır, öznitelik haritasındaki çapa kutularının göreli konumlarını belirtir.

[177] https://discuss.d2l.ai/t/1603

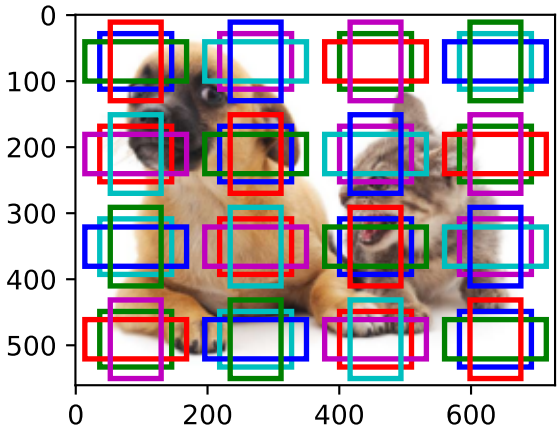
Çapa kutularının merkezleri (`anchors`) öznitelik haritasındaki tüm birimlere (`fmap`) yayıldığından, bu merkezler göreceli uzamsal konumları bakımından herhangi bir girdi imgesine tekdüze şekilde dağıtılmalıdır. Daha somut olarak, öznitelik haritasının sırasıyla `fmap_w` ve `fmap_h`, genişliği ve yüksekliği, göz önüne alındığında, aşağıdaki işlev, herhangi bir girdi imgesindeki `fmap_h` satırlarındaki ve `fmap_w` sütunlarındaki pikselleri *tekdüze* bir şekilde örnekleyecektir. Bu eşit örneklenen pikseller üzerinde ortalanan s ölçek çapa kutuları (s listenin uzunluğunun 1 olduğu varsayılarak) ve farklı en-boy oranları (`ratios`) oluşturulur.

```
def display_anchors(fmap_w, fmap_h, s):
    d2l.set_figsize()
    # İlk iki boyuttaki değerler çıktıyı etkilemez
    fmap = torch.zeros((1, 10, fmap_h, fmap_w))
    anchors = d2l.multibox_prior(fmap, sizes=s, ratios=[1, 2, 0.5])
    bbox_scale = torch.tensor((w, h, w, h))
    d2l.show_bboxes(d2l.plt.imshow(img).axes,
                    anchors[0] * bbox_scale)
```

İlk olarak, küçük nesnelerin algılanmasını düşünün. Görüntülendiğinde ayırt edilmeyi kolaylaştırmak için, buradaki farklı merkezlere sahip çapa kutuları çakışmaz: Çapa kutusu ölçeği 0.15 olarak ayarlanır ve öznitelik haritasının yüksekliği ve genişliği 4'e ayarlanır. Çapa kutularının merkezlerinin imge üzerindeki 4 satır ve 4 sütuna eşit olarak dağıtıldığını görebiliriz.

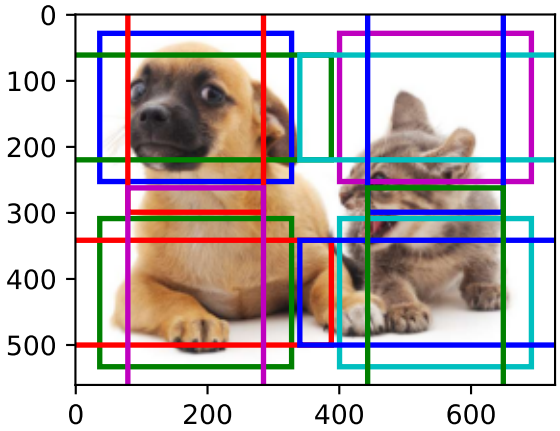
```
display_anchors(fmap_w=4, fmap_h=4, s=[0.15])
```

```
/home/d2l-worker/miniconda3/envs/d2l-tr-release-0/lib/python3.9/site-packages/torch/
↪functional.py:478: UserWarning: torch.meshgrid: in an upcoming release, it will be␣
↪required to pass the indexing argument. (Triggered internally at  ../aten/src/ATen/native/
↪TensorShape.cpp:2895.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
```



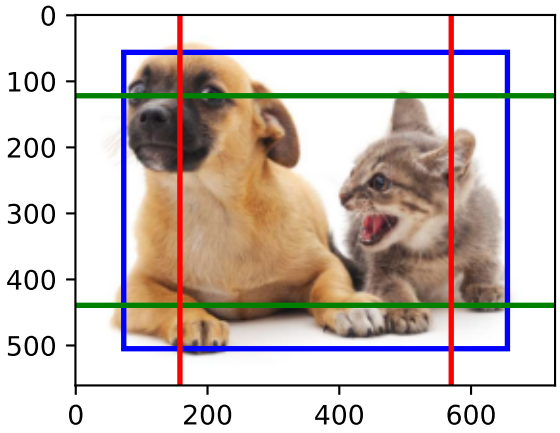
Öznitelik haritasının yüksekliğini ve genişliğini yarıya indirmeye ve daha büyük nesneleri algılamak için daha büyük çapa kutuları kullanmaya geçiyoruz. Ölçek 0.4 olarak ayarlandığında, bazı çapa kutuları birbirleriyle çakışır.

```
display_anchors(fmap_w=2, fmap_h=2, s=[0.4])
```



Son olarak, öznitelik haritasının yüksekliğini ve genişliğini yarıya indirir ve çapa kutusu ölçeğini 0.8 seviyesine çıkarırız. Şimdi çapa kutusunun merkezi imgenin merkezidir.

```
display_anchors(fmap_w=1, fmap_h=1, s=[0.8])
```



13.5.2 Çoklu Ölçekli Algılama

Çoklu ölçekli çapa kutuları oluşturduğumuzdan, bunları farklı ölçeklerde çeşitli boyutlardaki nesneleri algılamak için kullanacağız. Aşağıda, Section 13.7 içinde uygulayacağımız CNN tabanlı çoklu ölçekli nesne algılama yöntemini tanıtıyoruz.

Bazı ölçekte, $h \times w$ şeklinde c tane öznitelik haritası olduğunu varsayalım. Section 13.5.1 içindeki yöntemi kullanarak, her kümenin aynı merkeze sahip a çapa kutusuna sahip olduğu hw çapa kutusu kümeleri oluştururuz. Örneğin, Section 13.5.1 içindeki deneylerde ilk ölçekte, on (kanal sayısı) 4×4 öznitelik haritası verildiğinde, her kümenin aynı merkeze sahip 3 çapa kutusu içerdiği 16 adet çapa kutusu kümesi oluşturduk. Daha sonra, her çapa kutusu gerçek referans değerinin kuşatan kutusunun ofseti ve sınıfı ile etiketlenir. Geçerli ölçekte, nesne algılama modelinin, farklı kümelerin farklı merkezlere sahip olduğu girdi imgesindeki hw çapa kutuları kümelerinin sınıflarını ve ofsetlerini tahmin etmesi gerekir.

Buradaki c öznitelik haritalarının girdi imgesine dayalı CNN ileri yayma tarafından elde edilen ara çıktılar olduğunu varsayalım. Her öznitelik haritasında hw farklı uzamsal konum bulunduğundan, aynı uzamsal konumun c birime sahip olduğu düşünülebilir. Section 6.2 içindeki alıcı alan

tanımına göre, öznitelik haritalarının aynı uzamsal konumundaki bu c birimleri, girdi imgesinde aynı alıcı alana sahiptir: Aynı alıcı alanındaki girdi imgesi bilgisini temsil ederler. Bu nedenle, aynı uzamsal konumdaki öznitelik haritalarının c birimi, bu uzamsal konum kullanılarak oluşturulan a tane çapa kutusunun sınıflarına ve uzaklıklarına dönüştürebiliriz. Özünde, girdi imgesindeki alıcı alana yakın olan çapa kutularının sınıflarını ve ofsetlerini tahmin etmek için belirli bir alıcı alandaki girdi imgesinin bilgisini kullanırız.

Farklı katmanlardaki öznitelik haritaları, girdi imgesinde farklı boyutlarda alıcı alanlara sahip olduğunda, farklı boyutlardaki nesneleri algılamak için kullanılabilirler. Örneğin, çıktı katmanına daha yakın olan öznitelik haritalarının birimlerinin daha geniş alıcı alanlara sahip olduğu bir sinir ağı tasarlayabiliriz, böylece girdi imgesinde daha büyük nesneleri algılayabilirler.

Özetle, çoklu ölçekli nesne algılama için derin sinir ağları aracılığıyla imgelerin katmansal temsillerini birden çok düzeyde kullanabiliriz. Bunun Section 13.7 içinde somut bir örnekle nasıl çalıştığını göstereceğiz.

13.5.3 Özet

- Çoklu ölçekte, farklı boyutlardaki nesneleri algılamak için farklı boyutlarda çapa kutuları üretebiliriz.
- Öznitelik haritalarının şeklini tanımlayarak, herhangi bir imgedeki tekdüze örneklenmiş çapa kutularının merkezlerini belirleyebiliriz.
- Girdi imgesinde bir alıcı alana yakın olan çapa kutularının sınıflarını ve uzaklıklarını tahmin etmek için o belirli alıcı alandaki girdi imgesinin bilgisini kullanırız.
- Derin öğrenme sayesinde, çoklu ölçekli nesne algılama için imgelerin katmansal temsillerini birden çok düzeyde kullanabiliriz.

13.5.4 Alıştırmalar

1. Section 7.1 içindeki tartışmalarımıza göre, derin sinir ağları imgeler için soyutlama düzeylerini artırarak hiyerarşik öznitelikleri öğrenir. Çoklu ölçekli nesne algılamada, farklı ölçeklerdeki öznitelik haritaları farklı soyutlama düzeylerine karşılık geliyor mu? Neden ya da neden değil?
2. Section 13.5.1 içindeki deneylerde ilk ölçekte (`fmap_w=4`, `fmap_h=4`), çakışabilecek tekdüze dağıtılmış çapa kutuları oluşturun.
3. $1 \times c \times h \times w$ şeklinde bir öznitelik haritası değişkeni verilsin, burada c, h ve w, öznitelik haritalarının sırasıyla kanal sayısı, yüksekliği ve genişliğidir. Bu değişkeni çapa kutularının sınıflarına ve ofsetlerine nasıl dönüştürebilirsiniz? Çıktının şekli nedir?

Tartışmalar[178]

[178] https://discuss.d2l.ai/t/1607

13.6 Nesne Algılama Veri Kümesi

Nesne algılama alanında MNIST ve Fashion-MNIST gibi küçük bir veri kümeleri bulunmamaktadır. Nesne algılama modellerini hızlı bir şekilde göstermek için küçük bir veri kümesi topladık ve etiketledik. İlk olarak ofisimizdeki bedava muzların fotoğraflarını çektik ve farklı dönüşlerde ve boyutlarda 1000 muz imgesi oluşturduk. Sonra her bir muz imgesini bir arka plan görüntüsü üzerinde rastgele bir konuma yerleştirdik. Sonunda, resimlerdeki bu muzlar için kuşatan kutuları etiketledik.

13.6.1 Veri Kümesini İndirme

Tüm imgeler ve csv etiket dosyasıyla birlikte muz algılama veri kümesi doğrudan internetten indirilebilir.

```
%matplotlib inline
import os
import pandas as pd
import torch
import torchvision
from d2l import torch as d2l
```

```
#@save
d2l.DATA_HUB['banana-detection'] = (
    d2l.DATA_URL + 'banana-detection.zip',
    '5de26c8fce5ccdea9f91267273464dc968d20d72')
```

13.6.2 Veri Kümesini Okuma

Aşağıdaki `read_data_bananas` işlevinde muz algılama veri kümesini okuyacağız. Veri kümesi, bir csv dosyasında nesne sınıfı etiketlerini ve gerçek referans değeri kuşatan kutunun sol üst ve sağ alt köşelerdeki koordinatları içerir.

```
#@save
def read_data_bananas(is_train=True):
    """Muz algılama veri kümesi imgelerini ve etiketlerini okuyun."""
    data_dir = d2l.download_extract('banana-detection')
    csv_fname = os.path.join(data_dir, 'bananas_train' if is_train
                             else 'bananas_val', 'label.csv')
    csv_data = pd.read_csv(csv_fname)
    csv_data = csv_data.set_index('img_name')
    images, targets = [], []
    for img_name, target in csv_data.iterrows():
        images.append(torchvision.io.read_image(
            os.path.join(data_dir, 'bananas_train' if is_train else
                         'bananas_val', 'images', f'{img_name}')))
        # Burada `target` (hedef), tüm imgelerin aynı muz sınıfına
        # sahip olduğu (sınıf, sol üst x, sol üst y, sağ alt x, sağ alt y)
        # içerir (indeks 0)
        targets.append(list(target))
    return images, torch.tensor(targets).unsqueeze(1) / 256
```

İmgeleri ve etiketleri okumak için `read_data_bananas` işlevini kullanarak, aşağıdaki `BananasDataset` sınıfı, muz algılama veri kümesini yüklemek için özelleştirilmiş bir `Dataset` örneği oluşturmamızı sağlayacaktır.

```
#@save
class BananasDataset(torch.utils.data.Dataset):
    """Muz algılama veri kümesini yüklemek için özelleştirilmiş bir veri kümesi."""
    def __init__(self, is_train):
        self.features, self.labels = read_data_bananas(is_train)
        print('read ' + str(len(self.features)) + (f' training examples' if
              is_train else f' validation examples'))

    def __getitem__(self, idx):
        return (self.features[idx].float(), self.labels[idx])

    def __len__(self):
        return len(self.features)
```

Son olarak, `load_data_bananas` fonksiyonunu hem eğitim hem de test kümeleri için iki veri yineleyici örneği döndürecek şekilde tanımlarız. Test veri kümesi için onu rastgele sırayla okumaya gerek yoktur.

```
#@save
def load_data_bananas(batch_size):
    """Muz algılama veri kümesini yükleyin."""
    train_iter = torch.utils.data.DataLoader(BananasDataset(is_train=True),
                                             batch_size, shuffle=True)
    val_iter = torch.utils.data.DataLoader(BananasDataset(is_train=False),
                                           batch_size)
    return train_iter, val_iter
```

Bir minigrup okuyalım ve bu minigruptaki hem imgelerin hem de etiketlerin şekillerini yazdıralım. İmge minigrubunun şekli (grup boyutu, kanal sayısı, yükseklik, genişlik) tanıdık görünüyor: Önceki imge sınıflandırma görevlerimizle aynı. Minigrup etiketinin şekli (parti boyutu, m, 5) şeklindedir; burada m, veri kümesinde herhangi bir imgenin sahip olabileceği mümkün olan en büyük kuşatan kutu sayısıdır.

Minigruplarda hesaplama daha verimli olmasına rağmen, tüm imge örneklerinin bitiştirme yoluyla bir minigrup oluşturması için aynı sayıda kuşatan kutu içermeleri gerektirir. Genel olarak, imgeler farklı sayıda kuşatan kutuya sahip olabilir; bu nedenle m'den daha az kuşatan kutuya sahip imgeler, m'e ulaşılana kadar geçersiz kuşatan kutularla doldurulacaktır. Daha sonra her kuşatan kutunun etiketi 5 uzunluğunda bir dizi ile temsil edilir. Dizideki ilk öğe, kuşatan kutudaki nesnenin sınıfıdır ve burada -1 dolgu için geçersiz bir kuşatan kutusunu gösterir. Dizinin kalan dört öğesi, kuşatan kutunun sol üst köşesinin ve sağ alt köşesinin (x, y) koordinat değerleridir (aralık 0 ile 1 arasındadır). Muz veri kümesi için, her imgede sadece bir kuşatan kutu olduğundan elimizde $m = 1$ var.

```
batch_size, edge_size = 32, 256
train_iter, _ = load_data_bananas(batch_size)
batch = next(iter(train_iter))
batch[0].shape, batch[1].shape
```

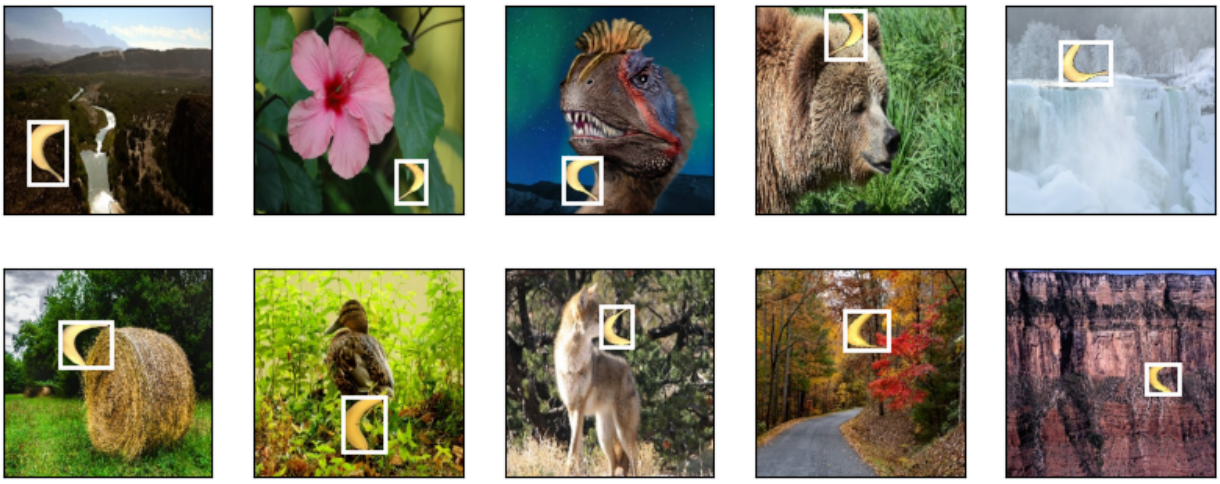
```
read 1000 training examples
read 100 validation examples
```

```
(torch.Size([32, 3, 256, 256]), torch.Size([32, 1, 5]))
```

13.6.3 Kanıtlama

Gerçek referans değeri etiketli kuşatan kutularıyla on imge gösterelim. Muzun dönüşlerinin, boyutlarının ve konumlarının tüm bu imgelerde değiştiğini görebiliyoruz. Tabii ki, bu sadece basit bir yapay veri kümesidir. Uygulamada, gerçek dünya veri kümeleri genellikle çok daha karmaşıktır.

```
imgs = (batch[0][0:10].permute(0, 2, 3, 1)) / 255
axes = d2l.show_images(imgs, 2, 5, scale=2)
for ax, label in zip(axes, batch[1][0:10]):
    d2l.show_bboxes(ax, [label[0][1:5] * edge_size], colors=['w'])
```



13.6.4 Özet

- Topladığımız muz algılama veri kümesi, nesne algılama modellerini göstermek için kullanılabilir.
- Nesne algılama için veri yükleme, imge sınıflandırmasındakine benzer. Bununla birlikte, nesne algılamasında etiketler ayrıca imge sınıflandırmasında eksik olan gerçek referans değeri kuşatan kutularla ilgili bilgileri de içerir.

13.6.5 Alıştırmalar

1. Muz algılama veri kümelerinde gerçek referans değeri kuşatan kutularla diğer imgeleri gösterin. Kuşatan kutulara ve nesnelere göre nasıl farklılık gösterirler?
2. Nesne algılamaya rastgele kırpma gibi veri artırımı uygulamak istediğimizi varsayalım. Bu imge sınıflandırmasındakinden nasıl farklı olabilir? İpucu: Kırpılmış bir imge nesnenin yalnızca küçük bir bölümünü içeriyorsa ne olur?

Tartışmalar[179]

13.7 Tek Atışta Çoklu Kutu Algılama

Section 13.3—-Section 13.6 içinde kuşatan kutuları, çapa kutularını, çoklu ölçekli nesne algılamayı ve nesne algılama için veri kümesini kullanıma sunduk. Şimdi bir nesne algılama modeli tasarlamak için bu tür arkaplan bilgisini kullanmaya hazırız: Tek atışta çoklu kutu algılama (SSD) (Liu *et al.*, 2016). Bu model basittir, hızlıdır ve yaygın olarak kullanılmaktadır. Bu, çok büyük miktardaki nesne algılama modellerinden sadece biri olmasına rağmen, bu bölümdeki tasarım ilkeleri ve uygulama detaylarından bazıları diğer modeller için de geçerlidir.

13.7.1 Model

Fig. 13.7.1, tek atışta çoklu kutu algılama tasarımına genel bir bakış sağlar. Bu model esas olarak bir temel ağdan ve ardından birkaç çoklu ölçekli öznitelik haritası bloğundan oluşur. Temel ağ, girdi imgesinden öznitelikleri ayıklamak içindir, bu nedenle derin bir CNN kullanabilir. Örneğin, özgün tek atışta çoklu kutu algılama makalesi (Liu *et al.*, 2016) sınıflandırma katmanında önce budanmış bir VGG ağı benimser, her ne kadar ResNet de yaygın olarak kullanılır olsa da. Tasarımımız sayesinde, daha küçük nesneleri algılamak için daha fazla çapa kutusu üretmek için temel ağ çıktısının daha büyük öznitelik haritaları yapmasını sağlarız. Daha sonra, her çoklu ölçekli öznitelik harita bloğu önceki bloktan öznitelik haritalarının yüksekliğini ve genişliğini azaltır (örn. yarı yarıya) ve öznitelik haritalarının her biriminin girdi imgesindeki alıcı alanını artırmasını sağlar.

Section 13.5 içinde derin sinir ağları tarafından imgelerin katmansal gösterimleri yoluyla çoklu ölçekli nesne algılama tasarımını hatırlayın. Fig. 13.7.1 içinde gösterilen üst kısmına daha yakın olan çoklu ölçekli öznitelik haritaları daha küçük olduğundan ancak daha büyük alıcı alanlara sahip olduklarından, daha az ama daha büyük nesneleri algılamak için uygundur.

Özetle, temel ağı ve birkaç çoklu ölçekli öznitelik haritası bloğu aracılığıyla, tek atışta çoklu kutu algılama, farklı boyutlarda değişen sayıda çapa kutusu oluşturur ve bu çapa kutularının sınıflarını ve ofsetlerini (dolayısıyla kuşatan kutuları) tahmin ederek değişen boyuttaki nesneleri algılar; bu nedenle, bu bir çoklu ölçekli nesne algılama modelidir.

[179] https://discuss.d2l.ai/t/1608

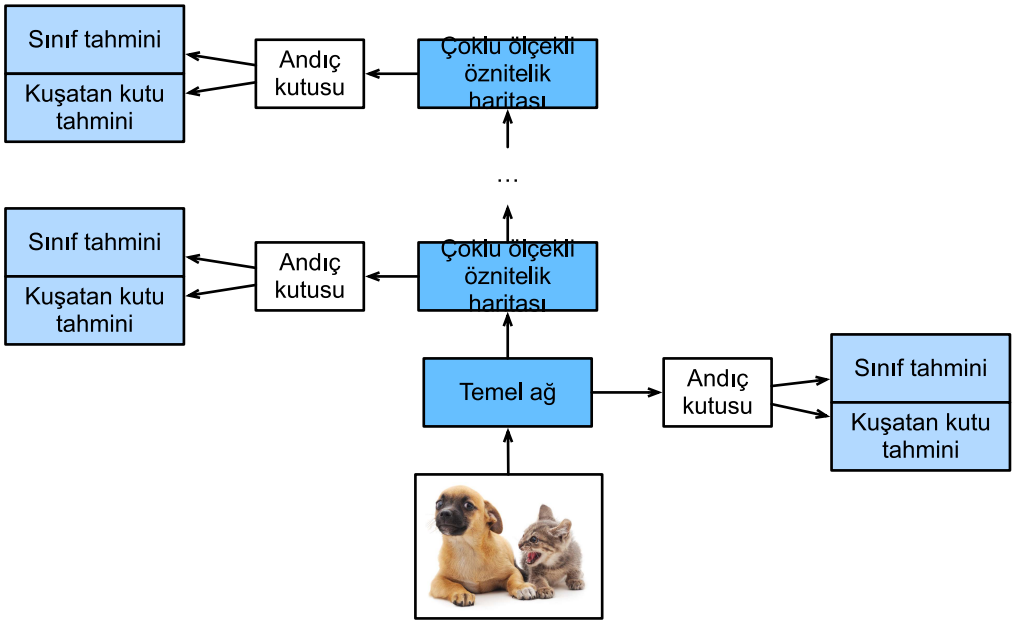


Fig. 13.7.1: Çoklu ölçekli bir nesne algılama modeli olarak, tek atışta çoklu kutu algılama temel olarak bir temel ağdan ve ardından birkaç çoklu ölçekli öznitelik haritası bloğundan oluşur.

Aşağıda, Fig. 13.7.1 içindeki farklı blokların uygulama ayrıntılarını açıklayacağız. Başlangıç olarak, sınıf ve kuşatan kutu tahmininin nasıl uygulanacağını tartışıyoruz.

Sınıf Tahmin Katmanı

Nesne sınıflarının sayısı q olsun. O zaman çapa kutuları $q+1$ sınıfa sahiptir, burada sınıf 0 arkaplandır. Bazı ölçeklerde, öznitelik haritalarının yüksekliği ve genişliğinin sırasıyla h ve w olduğunu varsayalım. a tane çapa kutusu, bu öznitelik haritalarının her mekansal konumu ile merkezi olarak oluşturulduğunda, toplam hwa çapa kutusunun sınıflandırılması gerekir. Bu durum genellikle büyük olasılıkla ağır parametreleme maliyetleri nedeniyle tam bağlı katmanlarla sınıflandırmayı olanaksız kılar. Section 7.3 içinde sınıfları tahmin etmek için evrişimli katman kanallarını nasıl kullandığımızı hatırlayın. Tek atışta çoklu kutu algılama, model karmaşıklığını azaltmak için aynı tekniği kullanır.

Özellikle, sınıf tahmini katmanı, öznitelik haritalarının genişliğini veya yüksekliğini değiştirmeden bir evrişimli katman kullanır. Bu şekilde, öznitelik haritalarının aynı uzamsal boyutlarında (genişlik ve yükseklik) çıktılar ve girdiler arasında birebir karşılık olabilir. Daha somut olarak, çıktı öznitelik haritalarının kanallarının herhangi bir (x, y) uzamsal konumu, girdi öznitelik haritalarının (x, y) merkezli tüm çapa kutuları için sınıf tahminlerini temsil eder. Geçerli tahminler üretmek için, $a(q+1)$ çıktı kanalı olmalıdır; burada aynı uzamsal konum için $i(q+1)+j$ dizinli çıktı kanalı, i $(0 \leq i < a)$ çapa kutusu için j $(0 \leq j \leq q)$ sınıfının tahminini temsil eder.

Aşağıda, sırasıyla `num_anchors` ve `num_classes` argümanları vasıtasıyla a'yı ve q'yu belirten böyle bir sınıf tahmini katmanı tanımlıyoruz. Bu katman, 1 dolgulu bir 3×3'lük evrişimli tabaka kullanır. Bu evrişimli katmanın girdisinin ve çıktısının genişliği ve yüksekliği değişmeden kalır.

```
%matplotlib inline
import torch
import torchvision
```

(continues on next page)

(continued from previous page)

```
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l


def cls_predictor(num_inputs, num_anchors, num_classes):
    return nn.Conv2d(num_inputs, num_anchors * (num_classes + 1),
                     kernel_size=3, padding=1)
```

Kuşatan Kutu Tahmin Katmanı

Kuşatan kutu tahmin katmanının tasarımı, sınıf tahmini katmanınkine benzer. Tek fark, her bir çapa kutusu için çıktı sayısında yatmaktadır: Burada $q + 1$ sınıfından ziyade dört ofset tahmin etmeliyiz.

```
def bbox_predictor(num_inputs, num_anchors):
    return nn.Conv2d(num_inputs, num_anchors * 4, kernel_size=3, padding=1)
```

Çoklu Ölçekler için Tahminleri Bitiştirme

Belirttiğimiz gibi, tek atışta çok kutulu algılama, çapa kutuları oluşturmak ve sınıflarını ve uzaklıklarını tahmin etmek için çoklu ölçekli öznitelik haritaları kullanır. Farklı ölçeklerde, öznitelik haritalarının şekilleri veya aynı birimde ortalanmış çapa kutularının sayısı değişebilir. Bu nedenle, farklı ölçeklerde tahmin çıktılarının şekilleri değişebilir.

Aşağıdaki örnekte, aynı minigrup için `Y1` ve `Y2` olmak üzere iki farklı ölçekte öznitelik haritası oluşturuyoruz; burada `Y2`'nin yüksekliği ve genişliği `Y1`'inkinin yarısı kadardır. Örnek olarak sınıf tahminini ele alalım. Her birim için sırasıyla `Y1` ve `Y2`'teki 5 ve 3 çapa kutusunun oluşturulduğunu varsayalım. Nesne sınıflarının sayısının 10 olduğunu varsayalım. Öznitelik haritaları `Y1` ve `Y2` için sınıf tahmini çıktılarındaki kanal sayısı sırasıyla $5 \times (10 + 1) = 55$ ve $3 \times (10 + 1) = 33$'tür, burada (toplu iş boyutu, kanal sayısı, yükseklik, genişlik) her iki çıktının şeklidir.

```
def forward(x, block):
    return block(x)

Y1 = forward(torch.zeros((2, 8, 20, 20)), cls_predictor(8, 5, 10))
Y2 = forward(torch.zeros((2, 16, 10, 10)), cls_predictor(16, 3, 10))
Y1.shape, Y2.shape
```

```
(torch.Size([2, 55, 20, 20]), torch.Size([2, 33, 10, 10]))
```

Gördüğümüz gibi, toplu iş boyutu hariç, diğer üç boyutun hepsinin farklı boyutları var. Daha verimli hesaplama için bu iki tahmin çıktısını bitiştirmek için bu tensörleri daha tutarlı bir formata dönüştüreceğiz.

Kanal boyutunun aynı merkeze sahip çapa kutuları için tahminleri tuttuğunu unutmayın. İlk önce bu boyutu en içteki boyuta taşıyacağız. Toplu iş boyutu farklı ölçekler için aynı kaldığından, tahmin çıktısını (toplu iş boyutu, yükseklik $\times$ genişlik $\times$ kanal sayısı) şekline sahip iki boyutlu bir tensöre dönüştürebiliriz. O zaman bu tür çıktıları 1. boyut boyunca farklı ölçeklerde birleştiririz.

```
def flatten_pred(pred):
    return torch.flatten(pred.permute(0, 2, 3, 1), start_dim=1)

def concat_preds(preds):
    return torch.cat([flatten_pred(p) for p in preds], dim=1)
```

Bu şekilde, `Y1` ve `Y2` kanal, yükseklik ve genişliklerde farklı boyutlara sahip olmasına rağmen, bu iki tahmin çıktısını aynı minigrup için iki farklı ölçekte birleştirebiliriz.

```
concat_preds([Y1, Y2]).shape
```

```
torch.Size([2, 25300])
```

Örnek Seyreltme Bloğu

Nesneleri çoklu ölçekte algılamak için girdi öznitelik haritalarının yüksekliğini ve genişliğini yarıya indiren aşağıdaki örnek seyreltme bloğu `down_sample_blk`'i tanımlarız. Aslında, bu blok Section 7.2.1 içindeki VGG bloklarının tasarımını uygular. Daha somut olarak, her bir örnek seyreltme bloğu, 1 dolgulu iki 3×3 evrişimli katmandan ve ardından 2 uzun adımlı bir 2×2 maksimum ortaklama katmanından oluşur. Bildiğimiz gibi, 1 dolgulu 3×3 evrişimli katmanlar, öznitelik haritalarının şeklini değiştirmez. Ancak, sonraki 2×2 maksimum ortaklama, girdi öznitelik haritalarının yüksekliğini ve genişliğini yarı yarıya azaltır. Bu örnek seyreltme bloğunun hem girdi hem de çıktı öznitelik haritaları için $1 \times 2 + (3 - 1) + (3 - 1) = 6$ olduğundan, çıktıdaki her birim girdi üzerinde 6×6 alıcı alanına sahiptir. Bu nedenle, aşağı örnek seyreltme bloğu, çıktı öznitelik haritalarında her birim alıcı alanını genişletir.

```
def down_sample_blk(in_channels, out_channels):
    blk = []
    for _ in range(2):
        blk.append(nn.Conv2d(in_channels, out_channels,
                             kernel_size=3, padding=1))
        blk.append(nn.BatchNorm2d(out_channels))
        blk.append(nn.ReLU())
        in_channels = out_channels
    blk.append(nn.MaxPool2d(2))
    return nn.Sequential(*blk)
```

Aşağıdaki örnekte, inşa edilmiş örnek seyreltme bloğumuz girdi kanallarının sayısını değiştirir ve girdi öznitelik haritalarının yüksekliğini ve genişliğini yarıya indirir.

```
forward(torch.zeros((2, 3, 20, 20)), down_sample_blk(3, 10)).shape
```

```
torch.Size([2, 10, 10, 10])
```

Temel Ağ Bloğu

Temel ağ bloğu, girdi imgelerinden öznitelikleri ayıklamak için kullanılır. Basitlik açısından, her bloktaki kanal sayısını iki katına çıkaran üç örnek seyreltme bloğundan oluşan küçük bir temel ağ oluşturuyoruz. 256×256'lik girdi imgesi göz önüne alındığında, bu temel ağ bloğu 32×32 tane $(256/2^3 = 32)$'lik öznitelik haritası çıkarır.

```
def base_net():
    blk = []
    num_filters = [3, 16, 32, 64]
    for i in range(len(num_filters) - 1):
        blk.append(down_sample_blk(num_filters[i], num_filters[i+1]))
    return nn.Sequential(*blk)

forward(torch.zeros((2, 3, 256, 256)), base_net()).shape
```

```
torch.Size([2, 64, 32, 32])
```

Tam Model

Tek atışta çoklu kutu algılama modeli beş bloktan oluşur. Her blok tarafından üretilen öznitelik haritaları, (i) çapa kutuları oluşturmak ve (ii) bu çapa kutularının sınıflarını ve ofsetlerini tahmin etmek için kullanılır. Bu beş blok arasında, ilki temel ağ bloğudur, ikincisinden dördüncüsüne kadarkiler örnek seyreltme bloklarıdır ve son blok hem yüksekliği hem de genişliği 1'e düşürmek için küresel maksimum ortaklama kullanır. Teknik olarak, ikincisinden beşinciye bütün bloklar, Fig. 13.7.1 içindeki çoklu ölçekli öznitelik harita bloklarıdır.

```
def get_blk(i):
    if i == 0:
        blk = base_net()
    elif i == 1:
        blk = down_sample_blk(64, 128)
    elif i == 4:
        blk = nn.AdaptiveMaxPool2d((1,1))
    else:
        blk = down_sample_blk(128, 128)
    return blk
```

Şimdi her blok için ileri yaymayı tanımlıyoruz. İmge sınıflandırma görevlerinden farklı olarak, buradaki çıktılar arasında (i) CNN öznitelik haritaları `Y`, (ii) geçerli ölçekte `Y` kullanılarak oluşturulan çapa kutuları ve (iii) (`Y`'ye dayalı) bu çapa kutuları için tahmin edilen sınıfları ve uzaklıkları (ofsetleri) içerir.

```
def blk_forward(X, blk, size, ratio, cls_predictor, bbox_predictor):
    Y = blk(X)
    anchors = d2l.multibox_prior(Y, sizes=size, ratios=ratio)
    cls_preds = cls_predictor(Y)
    bbox_preds = bbox_predictor(Y)
    return (Y, anchors, cls_preds, bbox_preds)
```

Fig. 13.7.1 içinde, tepeye daha yakın olan çoklu ölçekli bir öznitelik harita bloğunun daha büyük nesneleri algılamak için olduğunu hatırlayın; bu nedenle, daha büyük çapa kutuları oluşturması

gerekir. Yukarıdaki ileri yaymada, her çoklu ölçekli öznitelik harita bloğunda, çağrılan `multibox_prior` işlevinin `sizes` argümanı (Section 13.4 içinde açıklanmıştır) vasıtasıyla iki ölçekli değerlerden oluşan bir liste geçeririz. Aşağıda, 0.2 ile 1.05 arasındaki aralık, beş bloktaki daha küçük ölçek değerlerini belirlemek için eşit olarak beş bölüme ayrılmıştır: 0.2, 0.37, 0.54, 0.71 ve 0.88. Daha sonra daha büyük ölçek değerleri $\sqrt{0.2 \times 0.37} = 0.272$, $\sqrt{0.37 \times 0.54} = 0.447$ vb. şeklinde verilir.

```
sizes = [[0.2, 0.272], [0.37, 0.447], [0.54, 0.619], [0.71, 0.79],
         [0.88, 0.961]]
ratios = [[1, 2, 0.5]] * 5
num_anchors = len(sizes[0]) + len(ratios[0]) - 1
```

Şimdi tüm modeli, `TinySSD`, aşağıdaki gibi tanımlayabiliriz.

```
class TinySSD(nn.Module):
    def __init__(self, num_classes, **kwargs):
        super(TinySSD, self).__init__(**kwargs)
        self.num_classes = num_classes
        idx_to_in_channels = [64, 128, 128, 128, 128]
        for i in range(5):
            # Eşdeğer atama  `self.blk_i = get_blk(i)`
            setattr(self, f'blk_{i}', get_blk(i))
            setattr(self, f'cls_{i}', cls_predictor(idx_to_in_channels[i],
                                                    num_anchors, num_classes))
            setattr(self, f'bbox_{i}', bbox_predictor(idx_to_in_channels[i],
                                                      num_anchors))

    def forward(self, X):
        anchors, cls_preds, bbox_preds = [None] * 5, [None] * 5, [None] * 5
        for i in range(5):
            # Burada `getattr(self, 'blk_%d' % i)` `self.blk_i`'ye ulaşır
            X, anchors[i], cls_preds[i], bbox_preds[i] = blk_forward(
                X, getattr(self, f'blk_{i}'), sizes[i], ratios[i],
                getattr(self, f'cls_{i}'), getattr(self, f'bbox_{i}'))
        anchors = torch.cat(anchors, dim=1)
        cls_preds = concat_preds(cls_preds)
        cls_preds = cls_preds.reshape(
            cls_preds.shape[0], -1, self.num_classes + 1)
        bbox_preds = concat_preds(bbox_preds)
        return anchors, cls_preds, bbox_preds
```

256×256'lik imgelerden oluşan `X` minigrubundan bir model örneği oluşturup ileri yaymayı gerçekleştirmek için kullanıyoruz.

Bu bölümde daha önce gösterildiği gibi, ilk blok 32×32 öznitelik haritalarını çıkarır. İkinci ila dördüncü örnek seyreltme bloklarının yükseklik ve genişliği yarıya indirdiğini ve beşinci bloğun genel ortaklama kullandığını hatırlayın. Özellik haritalarının uzamsal boyutları boyunca her birim için 4 çapa kutusu oluşturulduğundan, beş ölçekli her imge için toplam $(32^2 + 16^2 + 8^2 + 4^2 + 1) \times 4 = 5444$ çapa kutusu oluşturulur.

```
net = TinySSD(num_classes=1)
X = torch.zeros((32, 3, 256, 256))
anchors, cls_preds, bbox_preds = net(X)
```

(continues on next page)

(continued from previous page)

```
print('output anchors:', anchors.shape)
print('output class preds:', cls_preds.shape)
print('output bbox preds:', bbox_preds.shape)
```

```
output anchors: torch.Size([1, 5444, 4])
output class preds: torch.Size([32, 5444, 2])
output bbox preds: torch.Size([32, 21776])
/home/d2l-worker/miniconda3/envs/d2l-tr-release-0/lib/python3.9/site-packages/torch/
↪functional.py:478: UserWarning: torch.meshgrid: in an upcoming release, it will be␣
↪required to pass the indexing argument. (Triggered internally at  ../aten/src/ATen/native/
↪TensorShape.cpp:2895.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
```

13.7.2 Eğitim

Şimdi tek atışta çoklu kutu algılama modelini nesne algılama için nasıl eğiteceğimizi açıklayacağız.

Veri Kümesini Okuma ve Modeli İlkleme

Başlangıç olarak, Section 13.6 içinde açıklanan muz algılama veri kümesini okuyalım.

```
batch_size = 32
train_iter, _ = d2l.load_data_bananas(batch_size)
```

```
read 1000 training examples
read 100 validation examples
```

Muz algılama veri kümesindeki tek bir sınıf var. Modeli tanımladıktan sonra parametrelerini ilklememiz ve optimizasyon algoritmasını tanımlamamız gerekir.

```
device, net = d2l.try_gpu(), TinySSD(num_classes=1)
trainer = torch.optim.SGD(net.parameters(), lr=0.2, weight_decay=5e-4)
```

Kayıp ve Değerlendirme Fonksiyonlarını Tanımlama

Nesne algılaması iki tür kayba sahiptir. İlk kayıp, çapa kutularının sınıflarıyla ilgilidir: Hesaplama, imge sınıflandırması için kullandığımız çapraz entropi kayıp işlevini yeniden kullanabilir. İkinci kayıp pozitif (arkaplan olmayan) çapa kutularının ofsetlerini ile ilgilenir: Bu bir bağlanım problemidir. Bununla birlikte, bu bağlanım sorunu için, burada Section 3.1.3 içinde açıklanan kare kaybı kullanmıyoruz. Bunun yerine, L_1 norm kaybını, tahmin ve gerçek referans değer arasındaki farkın mutlak değerini kullanıyoruz. Maske değişkeni `bbox_masks`, kayıp hesaplamasında negatif çapa kutularını ve geçersiz (dolgulu) çapa kutularını filtreler. Sonunda, model için yitim fonksiyonunu elde etmek için çapa kutusu sınıf kaybını ve çapa kutusu ofset kaybını toplarız.

```
cls_loss = nn.CrossEntropyLoss(reduction='none')
bbox_loss = nn.L1Loss(reduction='none')

def calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels, bbox_masks):
    batch_size, num_classes = cls_preds.shape[0], cls_preds.shape[2]
    cls = cls_loss(cls_preds.reshape(-1, num_classes),
                   cls_labels.reshape(-1)).reshape(batch_size, -1).mean(dim=1)
    bbox = bbox_loss(bbox_preds * bbox_masks,
                     bbox_labels * bbox_masks).mean(dim=1)
    return cls + bbox
```

Sınıflandırma sonuçlarını değerlendirmek için doğruluğu kullanabiliriz. Ofsetler için kullanılan L_1 norm kaybı nedeniyle, tahmini kuşatan kutuları değerlendirmek için *ortalama mutlak hata* kullanıyoruz. Bu tahmin sonuçları, üretilen çapa kutularından ve bunlar için tahmin edilen ofsetlerden elde edilir.

```
def cls_eval(cls_preds, cls_labels):
    # Sınıf tahmini sonuçları son boyutta olduğundan,
    # `argmax`'ın bu boyutu belirtmesi gerekiyor
    return float((cls_preds.argmax(dim=-1).type(
        cls_labels.dtype) == cls_labels).sum())

def bbox_eval(bbox_preds, bbox_labels, bbox_masks):
    return float((torch.abs((bbox_labels - bbox_preds) * bbox_masks)).sum())
```

Modeli Eğitme

Modelin eğitimini yaparken, çoklu ölçekli çapa kutuları (`anchors`) oluşturmamız ve ileri yaymada sınıflarını (`cls_preds`) ve ofsetleri (`bbox_preds`) tahmin etmemiz gerekir. Daha sonra `Y` etiket bilgisine dayanarak bu tür üretilen çapa kutularının sınıflarını (`cls_labels`) ve ofsetlerini (`bbox_labels`) etiketleriz. Son olarak, sınıfların ve ofsetlerin tahmini ve etiketlenmiş değerlerini kullanarak kayıp işlevini hesaplıyoruz. Özlü uygulamalar için, test veri kümesinin değerlendirilmesini burada atlıyoruz.

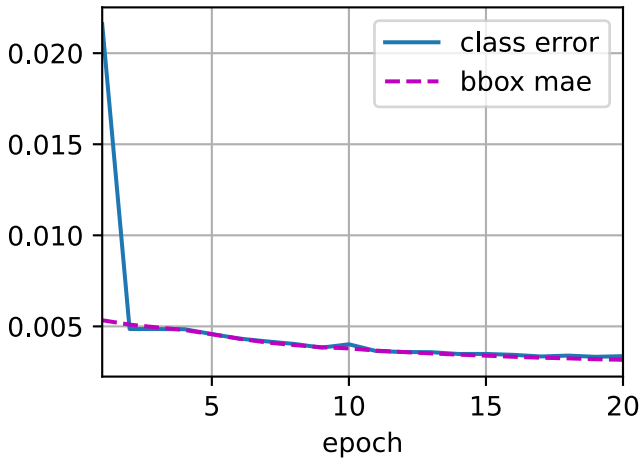
```
num_epochs, timer = 20, d2l.Timer()
animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                        legend=['class error', 'bbox mae'])
net = net.to(device)
for epoch in range(num_epochs):
    # Eğitim doğruluğu toplamı, eğitim doğruluğu toplamında örnek sayısı,
    # Mutlak hata toplamı, mutlak hata toplamında örnek sayısı
    metric = d2l.Accumulator(4)
    net.train()
    for features, target in train_iter:
        timer.start()
        trainer.zero_grad()
        X, Y = features.to(device), target.to(device)
        # Çok ölçekli bağlantı kutuları oluşturun ve sınıflarını
        # ve ofsetlerini tahmin edin
        anchors, cls_preds, bbox_preds = net(X)
        # Bu çapa kutularının sınıflarını ve uzaklıklarını etiketleyin
        bbox_labels, bbox_masks, cls_labels = d2l.multibox_target(anchors, Y)
```

(continues on next page)

(continued from previous page)

```
        # Sınıfların ve ofsetlerin tahmin edilen ve etiketlenen
        # değerlerini kullanarak kayıp fonksiyonunu hesaplayın
        l = calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels,
                      bbox_masks)
        l.mean().backward()
        trainer.step()
        metric.add(cls_eval(cls_preds, cls_labels), cls_labels.numel(),
                   bbox_eval(bbox_preds, bbox_labels, bbox_masks),
                   bbox_labels.numel())
    cls_err, bbox_mae = 1 - metric[0] / metric[1], metric[2] / metric[3]
    animator.add(epoch + 1, (cls_err, bbox_mae))
print(f'class err {cls_err:.2e}, bbox mae {bbox_mae:.2e}')
print(f'{len(train_iter.dataset) / timer.stop():.1f} examples/sec on '
      f'{str(device)}')
```

```
class err 3.37e-03, bbox mae 3.17e-03
5908.7 examples/sec on cuda:0
```



13.7.3 Tahminleme

Tahmin sırasında amaç, imge üzerindeki tüm nesneleri tespit etmektir. Aşağıda, bir test imgesini okuruz ve yeniden boyutlandırırız, onu evrişimli katmanların gerektirdiği dört boyutlu bir tensöre dönüştürürüz.

```
X = torchvision.io.read_image('../img/banana.jpg').unsqueeze(0).float()
img = X.squeeze(0).permute(1, 2, 0).long()
```

Aşağıdaki `multibox_detection` işlevini kullanarak, tahmini kuşatan kutuları çapa kutularından ve tahmin edilen ofsetlerden elde edilir. Daha sonra, benzer tahmini kuşatan kutuları kaldırmak için maksimum olmayanı bastırma kullanılır.

```
def predict(X):
    net.eval()
    anchors, cls_preds, bbox_preds = net(X.to(device))
```

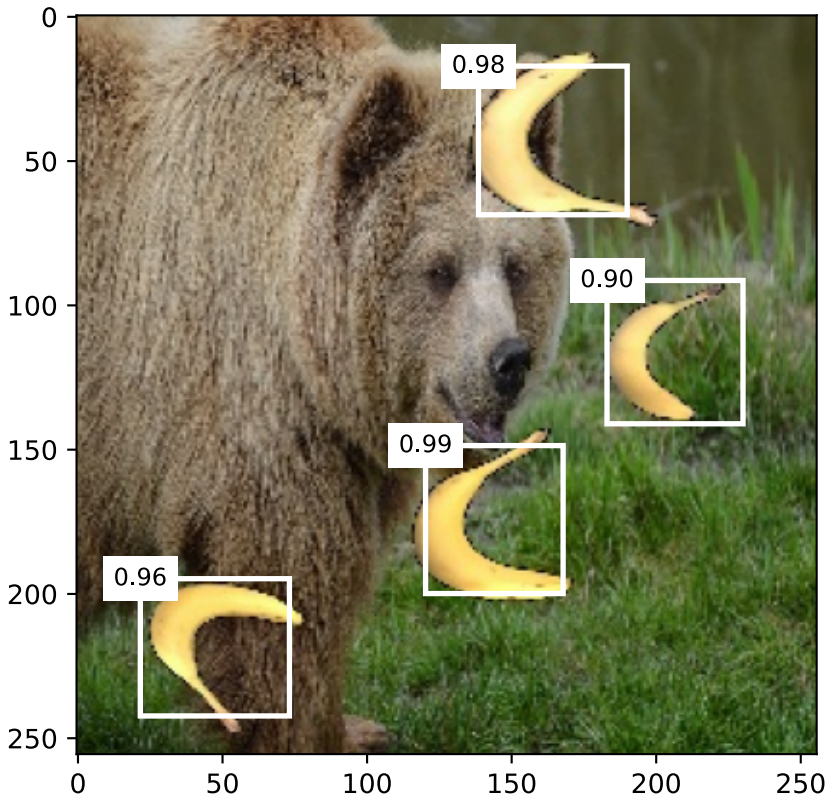
(continues on next page)

(continued from previous page)

```
    cls_probs = F.softmax(cls_preds, dim=2).permute(0, 2, 1)
    output = d2l.multibox_detection(cls_probs, bbox_preds, anchors)
    idx = [i for i, row in enumerate(output[0]) if row[0] != -1]
    return output[0, idx]

output = predict(X)
```

Son olarak, çıktı olarak 0.9 veya üzeri güvene sahip tüm tahmini kuşatan kutuları gösteriyoruz.

```
def display(img, output, threshold):
    d2l.set_figsize((5, 5))
    fig = d2l.plt.imshow(img)
    for row in output:
        score = float(row[1])
        if score < threshold:
            continue
        h, w = img.shape[0:2]
        bbox = [row[2:6] * torch.tensor((w, h, w, h), device=row.device)]
        d2l.show_bboxes(fig.axes, bbox, '%.2f' % score, 'w')

display(img, output.cpu(), threshold=0.9)
```



13.7.4 Özet

- Tek atışta çoklu kutu algılama, çoklu ölçekli bir nesne algılama modelidir. Temel ağı ve birkaç çoklu ölçekli öznitelik harita bloğu aracılığıyla, tek atışta çoklu kutulu algılama, farklı boyutlarda değişen sayıda çapa kutusu oluşturur ve bu çapa kutularının sınıflarını ve ofsetlerini (dolayısıyla kuşatan kutuları) tahmin ederek değişen boyuttaki nesneleri algılar.
- Tek atışta çoklu kutu algılama modelini eğitirken, kayıp işlevi, çapa kutusu sınıflarının ve ofsetlerinin tahmin edilen ve etiketlenmiş değerlerine göre hesaplanır.

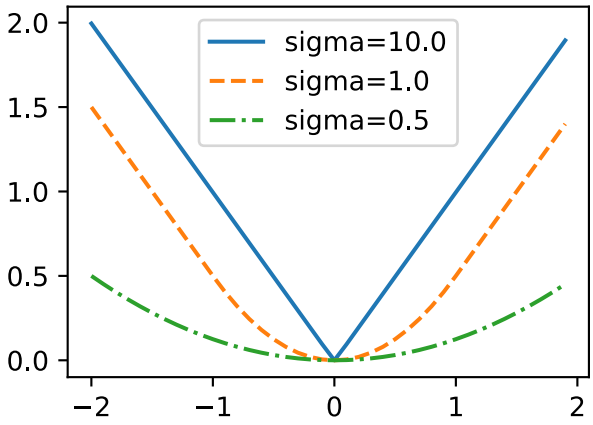
13.7.5 Alıştırmalar

1. Kayıp işlevini geliştirerek tek atışta çoklu kutu algılamasını geliştirebilir misiniz? Örneğin, tahmini ofsetler için pürüzsüz L_1 norm kaybı ile L_1 norm kaybını değiştirin. Bu kayıp fonksiyonu, hiper parametre σ tarafından kontrol edilen pürüzsüzlük için sıfır çevresinde bir kare işlevi kullanır:

$$f(x) = \begin{cases} (\sigma x)^2/2, & \text{if } |x| < 1/\sigma^2 \\ |x| - 0.5/\sigma^2, & \text{otherwise} \end{cases} \tag{13.7.1}$$

σ çok büyük olduğunda, bu kayıp L_1 norm kaybına benzer. Değeri daha küçük olduğunda, kayıp fonksiyonu daha pürüzsüzdür.

```
def smooth_l1(data, scalar):
    out = []
    for i in data:
        if abs(i) < 1 / (scalar ** 2):
            out.append(((scalar * i) ** 2) / 2)
        else:
            out.append(abs(i) - 0.5 / (scalar ** 2))
    return torch.tensor(out)

sigmas = [10, 1, 0.5]
lines = ['-', '--', '-.']
x = torch.arange(-2, 2, 0.1)
d2l.set_figsize()

for l, s in zip(lines, sigmas):
    y = smooth_l1(x, scalar=s)
    d2l.plt.plot(x, y, l, label='sigma=%.1f' % s)
d2l.plt.legend();
```

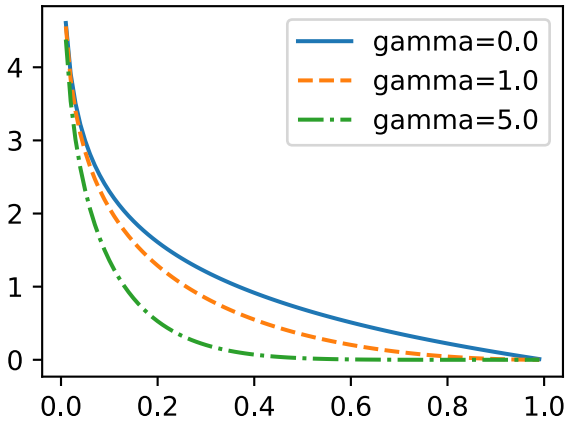


Ayrıca, deneyde sınıf tahmini için çapraz entropi kaybı kullandık: p_j ile p_j ile gerçek referans değer sınıfı j için tahmin edilen olasılığı ifade edersek, çapraz entropi kaybı $-\log p_j$ olur. Ayrıca odak kaybı (Lin *et al.*, 2017) kullanabilirsiniz: $\gamma > 0$ ve $\alpha > 0$ hiper parametre ile, bu kayıp şu şekilde tanımlanır:

$$-\alpha(1-p_j)^\gamma \log p_j. \tag{13.7.2}$$

Gördüğümüz gibi, γ'i arttırmak, iyi sınıflandırılmış örnekler için göreli kaybı etkili bir şekilde azaltabilir (örneğin, $p_j > 0.5$), böylece eğitim yanlış sınıflandırılmış bu zor örneklere daha fazla odaklanabilir.

```
def focal_loss(gamma, x):
    return -(1 - x) ** gamma * torch.log(x)

x = torch.arange(0.01, 1, 0.01)
for l, gamma in zip(lines, [0, 1, 5]):
    y = d2l.plt.plot(x, focal_loss(gamma, x), l, label='gamma=%.1f' % gamma)
d2l.plt.legend();
```



2. Alan kısıtları nedeniyle, bu bölümdeki tek atışta çoklu kutu algılama modelinin bazı uygulama ayrıntılarını atladık. Modeli aşağıdaki yönlerden daha da geliştirebilir misiniz?
    1. Nesne imgeye kıyasla çok daha küçük olduğunda, model girdi imgesini daha büyük boyutlandırabilir.

2. Genellikle çok sayıda negatif çapa kutusu vardır. Sınıf dağılımını daha dengeli hale getirmek için negatif çapa kutularını azaltabiliriz.
3. Kayıp işlevinde, sınıf kaybına ve ofset kaybına farklı ağırlık hiper parametreleri atayın.
4. Nesne algılama modelini değerlendirmek için, tek atışta çoklu kutu algılama çalışması (Liu *et al.*, 2016) gibi diğer yöntemleri kullanın.

Tartışmalar[180]

13.8 Bölge tabanlı CNN'ler (R-CNN'ler)

Section 13.7 içinde açıklanan tek atışta çoklu kutu algılamanın yanı sıra, bölge tabanlı CNN'ler veya CNN özniteliklerine (R-CNN) sahip bölgeler de nesne algılamaya (Girshick *et al.*, 2014) derin öğrenmeyi uygulamanın öncü yaklaşımları arasında yer almaktadır. Bu bölümde R-CNN ve onun iyileştirilmiş serilerini tanıtacağız: Hızlı R-CNN (Girshick, 2015), daha hızlı R-CNN (Ren *et al.*, 2015) ve maske R-CNN (He *et al.*, 2017). Sınırlı alan nedeniyle, sadece bu modellerin tasarımına odaklanacağız.

13.8.1 R-CNN'ler

R-CNN ilk olarak girdi imgesinden birçok (örneğin, 2000) *bölge önerisi* ayıklar (örneğin, çapa kutuları bölge önerileri olarak da kabul edilebilir), sınıflarını ve kuşatan kutuları (örneğin, ofsetler) etiketler (Girshick *et al.*, 2014). Sonra bir CNN, her bölge önerisi üzerinde ileri yaymayı gerçekleştirmek için öznitelikleri kullanır. Sonrasında, her bölge önerisinin öznitelikleri, bu bölge önerisinin sınıfını ve kuşatan kutusunu tahmin etmek için kullanılır.

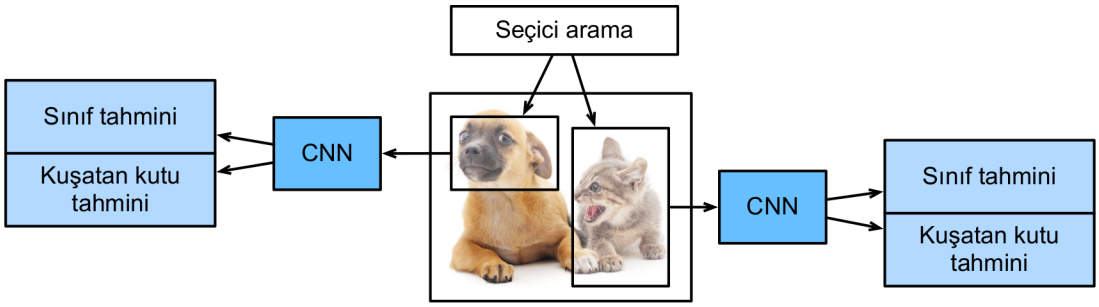


Fig. 13.8.1: R-CNN modeli.

Fig. 13.8.1 R-CNN modelini gösterir. Daha somut olarak, R-CNN aşağıdaki dört adımdan oluşur:

1. (Uijlings *et al.*, 2013) girdi imgesinden birden fazla yüksek kaliteli bölge önerisi ayıklamak için *seçici arama* gerçekleştirin. Önerilen bu bölgeler genellikle farklı şekil ve boyutlarda çoklu ölçeklerde seçilir. Her bölge önerisi bir sınıf ve bir gerçek referans değeri kuşatan kutu ile etiketlenecektir.
2. Önceden eğitilmiş bir CNN seçin ve çıktı katmanından önce budayın. Her bölge önerisini ağın gerektirdiği girdi boyutuna yeniden boyutlandırın ve bölge önerisi için ayıklanan öznitelikleri ileri yayma yoluyla çıktılayın.

[180] https://discuss.d2l.ai/t/1604

3. Her bölge önerisinin çıkarılan özniteliklerini ve etiketlenmiş sınıfını örnek olarak ele alın. Her destek vektör makinesinin, örneğin belirli bir sınıfı içerip içermediğini ayrı ayrı belirlediği nesneleri sınıflandırmak için birden fazla destek vektör makinesini eğitin.

4. Örnek olarak her bölge önerisinin ayıklanan öznitelikleri ve etiketli kuşatan kutusunu alın. Gerçek referans değer kuşatan kutuyu tahmin etmek için doğrusal bağlanım modelini eğitin.

R-CNN modeli imge özniteliklerini etkili bir şekilde ayıklamak için önceden eğitilmiş CNN'ler kullansa da yavaştır. Tek bir girdi imgesinden binlerce bölge önerisini seçtiğimizi düşünün: Bu nesne algılamayı gerçekleştirmek için binlerce CNN ileri yaymasını gerektirir. Bu devasa bilgi işlem yükünden dolayı, R-CNN'lerin gerçek dünyadaki uygulamalarda yaygın olarak kullanılmasını mümkün değildir.

13.8.2 Hızlı R-CNN

Bir R-CNN'nin ana performans darboğazı, her bölge önerisi için hesaplamayı paylaşmadan bağımsız CNN ileri yaymasında yatmaktadır. Bu bölgelerde genellikle çakışmalar olduğundan, bağımsız öznitelik ayıklamaları çok fazla tekrarlanan hesaplamaya yol açar. *Hızlı R-CNN*'nin R-CNN'den sağladığı en önemli iyileştirmelerden biri, CNN ileri yaymasının yalnızca tüm imge üzerinde (Girshick, 2015) gerçekleştirilmesidir.

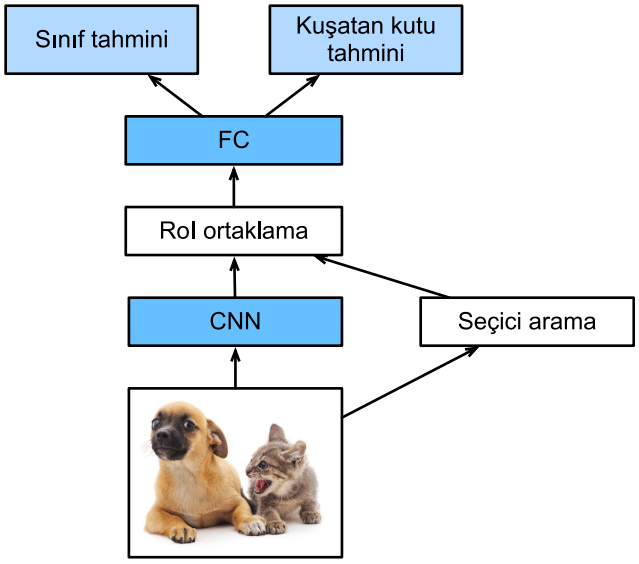


Fig. 13.8.2: Hızlı R-CNN modeli.

Fig. 13.8.2 hızlı R-CNN modelini açıklar. Başlıca hesaplamaları şöyledir:

1. R-CNN ile karşılaştırıldığında, hızlı R-CNN'de, öznitelik ayıklama için CNN'nin girdisi, bireysel bölge önerilerinden ziyade tüm imgedir. Dahası, bu CNN eğitilebilir. Bir girdi imgesi göz önüne alındığında, CNN çıktısının şekli $1 \times c \times h_1 \times w_1$ olsun.

2. Seçici aramanın n bölge önerileri oluşturduğunu varsayalım. Bu bölge önerileri (farklı şekillerde) CNN çıktısındaki ilgi bölgelerini (farklı şekillerdeki) işaretler. Daha sonra kolayca birleştirilebilmesi için ilgili bu bölgelerin aynı şekle sahip öznitelikleri ayıklanır (yükseklik h_2 ve genişlik w_2 diye belirtilir). Bunu başarmak için hızlı R-CNN, *ilgili bölge (RoI) ortaklama* katmanı sunar: CNN çıktısı ve bölge önerileri bu katmana girilir ve tüm bölge önerileri için daha da ayıklanmış $n \times c \times h_2 \times w_2$ şekilli bitiştirilmiş öznitelikleri ortaya çıkarır.

3. Tam bağlı bir katman kullanarak, bitiştirilmiş öznitelikleri $n \times d$ şeklinde bir çıktıya dönüştürün; burada d model tasarımına bağlıdır.
4. n bölge önerilerinin her biri için sınıfı ve kuşatan kutuyu tahmin edin. Daha somut olarak, sınıf ve kuşatan kutu tahmininde, tam bağlı katman çıktısını $n \times q$ şeklinde (q sınıf sayısıdır) ve $n \times 4$ şeklinde çıktıya dönüştürün. Sınıf tahmini softmaks bağlanım kullanır.

Hızlı R-CNN'de önerilen ilgi bölgesi ortaklama katmanı Section 6.5 içinde tanıtılan ortaklama katmanından farklıdır. Ortaklama katmanında, ortaklama penceresinin, dolgunun ve adımın boyutlarını belirterek çıktı şeklini dolaylı olarak kontrol ediyoruz. Buna karşılık, doğrudan ilgi bölgesi ortaklama katmanı çıktı şeklini belirtebilirsiniz.

Örneğin, her bölge için çıktı yüksekliğini ve genişliğini sırasıyla h_2 ve w_2 olarak belirtelim. $h \times w$ şeklindeki herhangi bir ilgi bölgesinin penceresi için, bu pencere, her alt pencerenin şekli yaklaşık $(h/h_2) \times (w/w_2)$ olduğu $h_2 \times w_2$ bir alt pencere ızgarasına ayrılmıştır. Pratikte, herhangi bir alt pencerenin yüksekliği ve genişliği tamsayıya yuvarlanır ve en büyük eleman, alt pencerenin çıktısı olarak kullanılacaktır. Bu nedenle, ilgi bölgeleri farklı şekillere sahip olsa bile ilgi bölgesi ortaklama katmanı aynı şekle sahip öznitelikler ayıklayabilir.

Açıklayıcı bir örnek olarak, Fig. 13.8.3 içinde, 4×4 girdisinde sol üst 3×3 ilgi bölgesi seçilir. Bu ilgi bölgesinde 2×2'lik çıktı elde etmek için bir 2×2 ilgi bölgesi ortaklama katmanı kullanıyoruz. Dört bölünmüş alt pencerenin her birinin 0, 1, 4 ve 5 (5 maksimumdur) öğelerini içerdiğini unutmayın; 2 ve 6 (6 maksimumdur); 8 ve 9 (9 maksimumdur) ve 10.

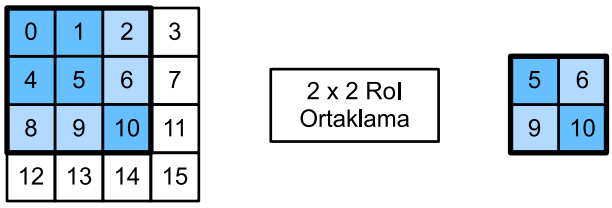


Fig. 13.8.3: 2×2'lik ilgi bölgesi ortaklama katmanı

Aşağıda ilgi bölgesi ortaklama katmanı hesaplanması gösterilmektedir. CNN çıkarılan öznitelikleri, `X`'in, yüksekliğinin ve genişliğinin, her ikisinin de 4 olduğunu ve yalnızca tek bir kanal olduğunu varsayalım.

```
import torch
import torchvision

X = torch.arange(16.).reshape(1, 1, 4, 4)
X
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]]]])
```

Girdi imgesinin hem yüksekliğinin hem de genişliğinin 40 piksel olduğunu ve seçici aramanın bu imgede iki bölge önerisi oluşturduğunu varsayalım. Her bölge önerisi beş öğe olarak ifade edilir: Nesne sınıfını, ardından, sol üst ve sağ alt köşelerinin (x, y)-koordinatları takip eder.

```
rois = torch.Tensor([[0, 0, 0, 20, 20], [0, 0, 10, 30, 30]])
```

`X`'in yüksekliği ve genişliği, girdi imgesinin yüksekliğinin ve genişliğinin $1/10$'u olduğu için, iki bölge önerisinin koordinatları belirtilen `spatial_scale` argümanına göre 0.1 ile çarpılır. Böylece iki ilgi bölgesi `X` üzerinde sırasıyla `X[:, :, 0:3, 0:3]` ve `X[:, :, 1:4, 0:4]` olarak işaretlenmiştir. Son olarak 2×2'lık ilgi bölgesi ortaklamasında, her bir ilgi bölgesi, aynı şekilde 2×2 olan öznitelikleri daha fazla çıkarmak için bir alt pencere ızgarasına bölünür.

```
torchvision.ops.roi_pool(X, rois, output_size=(2, 2), spatial_scale=0.1)
```

```
tensor([[[[ 5.,  6.],
          [ 9., 10.]]],


        [[[ 9., 11.],
          [13., 15.]]]])
```

13.8.3 Daha Hızlı R-CNN

Nesne algılamada daha doğru olması için hızlı R-CNN modeli genellikle seçici aramada çok sayıda bölge teklifi üretmelidir. Doğruluk kaybı olmadan bölge önerilerini azaltmak için *daha hızlı R-CNN* seçici aramayı *bölge önerisi ağı* (Ren *et al.*, 2015) ile değiştirmeyi önermektedir.

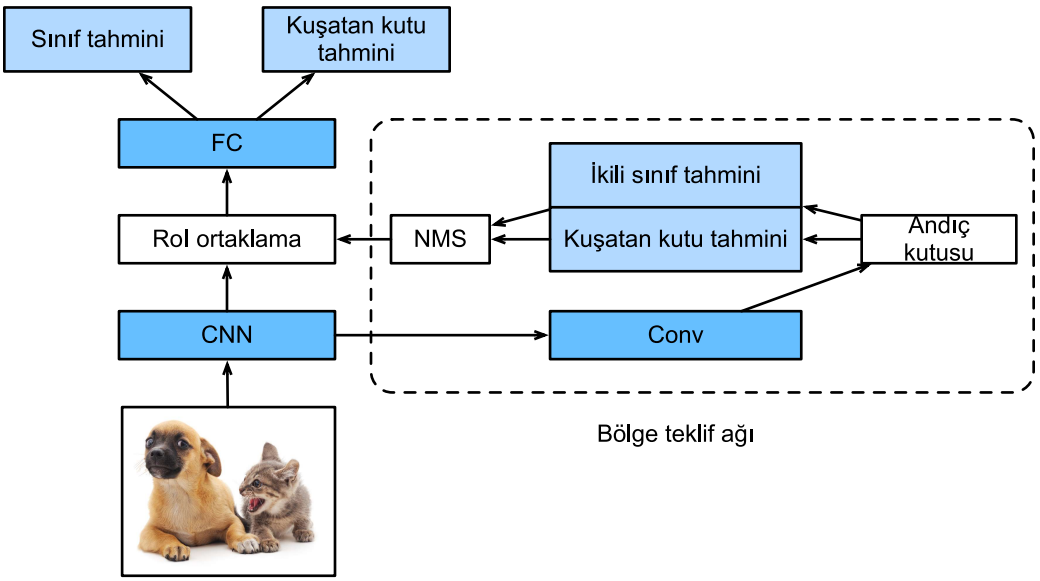


Fig. 13.8.4: Daha hızlı R-CNN model.

Fig. 13.8.4 daha hızlı R-CNN modelini gösterir. Hızlı R-CNN ile karşılaştırıldığında, daha hızlı R-CNN yalnızca bölge önerisi yöntemini seçici aramadan bölge önerisi ağına değiştirir. Modelin geri kalanı değişmeden kalır. Bölge önerisi ağı aşağıdaki adımlarla çalışır:

1. CNN çıktısını c kanal ile yeni bir çıktıya dönüştürmek için 1 dolgulu 3×3 evrişimli katman kullanın. Bu şekilde, CNN çıkarılan öznitelik haritalarının uzamsal boyutları boyunca her birimi c uzunluğunda yeni bir öznitelik vektörü alır.

2. Öznitelik haritalarının her pikselinde ortalanan, farklı ölçeklerde ve en boy oranlarında çoklu çapa kutusu oluşturun ve bunları etiketleyin.
3. Her bir çapa kutusunun ortasındaki c uzunluklu öznitelik vektörünü kullanarak, bu çapa kutusu için ikili sınıfı (arkaplan veya nesneler) ve kuşatan kutuyu tahmin edin.
4. Tahmin edilen sınıfları nesneler olan tahmin edilen kuşatan kutuları düşünün. Maksimum olmayanı bastırmayı kullanarak örtüşen sonuçları kaldırın. Nesneler için kalan tahmini kuşatan kutuları, ilgi bölgesi ortaklama katmanının ihtiyaç duyduğu bölge önerileridir.

Daha hızlı R-CNN modelinin bir parçası olarak, bölge önerisi ağının modelin geri kalanı ile birlikte eğitildiğini belirtmek gerekir. Başka bir deyişle, daha hızlı R-CNN'nin amaç işlevi yalnızca nesne algılamada sınıf ve kuşatan kutu tahminini değil, aynı zamanda bölge önerisi ağındaki çapa kutularının ikili sınıfı ve kuşatan kutu tahminini de içerir. Uçtan uca eğitim sonucunda bölge önerisi ağı, verilerden öğrenilen daha az sayıda bölge teklifiyle nesne algılamada doğru kalabilmek için yüksek kaliteli bölge tekliflerinin nasıl üretileceğini öğrenir.

13.8.4 Maske R-CNN

Eğitim veri kümesinde, nesnenin piksel düzeyindeki konumları imgelerde de etiketlenmişse, *mask R-CNN*, nesne algılamanın (He *et al.*, 2017) doğruluğunu daha da geliştirmek için bu tür ayrıntılı etiketleri etkili bir şekilde kullanabilir.

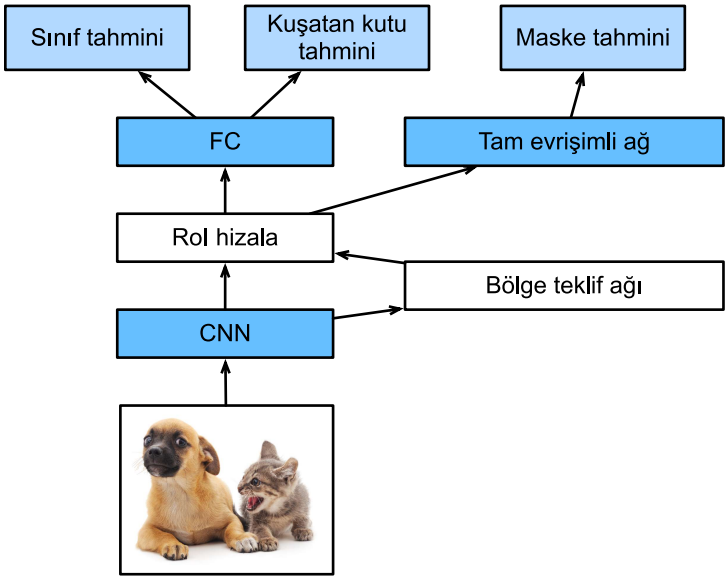


Fig. 13.8.5: Maske R-CNN modeli.

Fig. 13.8.5 içinde gösterildiği gibi, maske R-CNN daha hızlı R-CNN'den devşirilmiştir. Özellikle, maske R-CNN ilgi bölgesi ortaklama katmanını *ilgi bölgesi (RoI) hizalama* katmanı ile değiştirir. Bu ilgi bölgesi hizalama katmanı, öznitellik haritalarındaki uzamsal bilgileri korumak için ikili aradeğerleme kullanır; bu da piksel düzeyinde tahmin için daha uygundur. Bu katmanın çıktısı, tüm ilgi bölgeleri için aynı şekildeki öznitelik haritaları içerir. Bunlar, yalnızca her bir ilgi bölgesi için sınıfı ve kuşatan kutuyu değil, aynı zamanda ek bir tam evrişimli ağ aracılığıyla nesnenin piksel düzeyinde konumunu da tahmin etmek için kullanılırlar. Bir imgenin piksel düzeyinde anlamını tahmin etmek için tam evrişimli bir ağ kullanma hakkında daha fazla ayrıntı, bu ünitenin sonraki bölümlerinde sağlanacaktır.

13.8.5 Özet

- R-CNN, girdi imgesinden birçok bölge önerisi çıkarır, özniteliklerini ayıklamak için her bölge önerisi üzerinde ileri yaymayı gerçekleştirmek için bir CNN kullanır, ardından bu öznitelikleri bu bölge önerisinin sınıfını ve kuşatan kutuyu tahmin etmek için kullanır.
- Hızlı R-CNN, R-CNN'den önemli iyileştirmelerinden biri CNN ileri yaymasının sadece tüm imge üzerinde gerçekleştirilir olmasıdır. Aynı zamanda ilgi bölgesi ortaklama katmanını takdim eder, böylece aynı şekle sahip öznitelikler daha farklı şekillere sahip ilgi bölgeleri için çıkartılabilir.
- Daha hızlı R-CNN, hızlı R-CNN'de kullanılan seçici aramayı ortak eğitilmiş bir bölge öneri ağıyla değiştirir, böylece eski, daha az sayıda bölge önerisiyle nesne algılamada doğru kalabilir.
- Daha hızlı R-CNN'e dayanan maske R-CNN, nesne algılamanın doğruluğunu daha da artırmak için piksel düzeyinde etiketlerden yararlanmak için ek olarak tam evrişimli bir ağ sunar.

13.8.6 Alıştırmalar

1. Nesne algılamasını, kuşatan kutuları ve sınıf olasılıklarını tahmin etme gibi, tek bir bağlanım sorunu olarak çerçeveleyebilir miyiz? YOLO model (Redmon *et al.*, 2016) tasarımına başvurabilirsiniz.
2. Tek atışta çoklu kutu algılamayı bu bölümde tanıtılan yöntemlerle karşılaştırın. Büyük farklılıklar nelerdir? (Zhao *et al.*, 2019)'deki Şekil 2'ye başvurabilirsiniz.

Tartışmalar[181]

13.9 Anlamsal Bölümleme ve Veri Kümesi

Section 13.3—Section 13.8 içinde nesne algılama görevleri tartışılırken, imgelerdeki nesneleri etiketlemek ve tahmin etmek için dikdörtgen kuşatan kutular kullanıldı. Bu bölümde, bir imgenin farklı anlamsal sınıflara ait bölgelere nasıl bölüneceğine odaklanan *anlamsal bölümleme* sorununu tartışacaktır. Nesne algılamasından farklı olarak, anlamsal bölümleme piksel düzeyinde imgelerde ne olduğunu tanır ve anlar: Anlamsal bölgelerin etiketlenmesi ve tahmini piksel düzeydedir. Fig. 13.9.1, anlamsal bölümlemede imgenin köpek, kedi ve arkaplanının etiketlerini gösterir. Nesne algılama ile karşılaştırıldığında, anlamsal bölümlemede etiketlenmiş piksel düzeyinde sınırlar açıkça daha ince tanelidir.

[181] https://discuss.d2l.ai/t/1409

Fig. 13.9.1: Anlamsal bölümlemede köpek, kedi ve görüntünün arka planının etiketleri.

13.9.1 İmge Bölümleme ve Örnek Bölümleme

Bilgisayarla görme alanında anlamsal bölümlemeye benzer iki önemli görev vardır, yani imge bölümleme ve örnek bölümleme. Onları anlamsal bölümlemeden kısaca aşağıdaki gibi ayırt edeceğiz.

- *İmge bölümleme* bir imgeyi birkaç kurucu bölgeye böler. Bu tür bir soruna yönelik yöntemler genellikle imgedeki pikseller arasındaki korelasyonu kullanır. Eğitim sırasında imge pikselleri hakkında etiket bilgisine ihtiyaç duymaz ve bölümlere ayrılmış bölgelerin tahmin sırasında elde etmeyi umduğumuz anlamsal bilgilere sahip olacağını garanti edemez. Fig. 13.9.1 içindeki imgeyi girdi olarak alarak, imge bölümleme köpeği iki bölgeye bölebilir: Biri ağırlıklı olarak siyah olan ağız ve gözleri kaplar, diğeri ise esas olarak sarı olan vücudun geri kalanını kaplar.
- *Örnek bölümleme* aynı zamanda *eşzamanlı algılama ve bölümleme* olarak da adlandırılır. İmgedeki her nesne örneğinin piksel düzeyinde bölgelerinin nasıl tanınacağını inceler. Anlamsal bölümlemeden farklı olarak, örnek bölümlemenin yalnızca anlamı değil, aynı zamanda farklı nesne örneklerini de ayırt etmesi gerekir. Örneğin, görüntüde iki köpek varsa, örnek bölümlemenin bir pikselin iki köpekten hangisine ait olduğunu ayırt etmesi gerekir.

13.9.2 Pascal VOC2012 Anlamsal Bölümleme Veri Kümesi

En önemli anlamsal bölümleme veri kümesi Pascal VOC2012[182]'dir. Aşağıda, bu veri kümesine bir göz atacağız.

```
%matplotlib inline
import os
import torch
import torchvision
from d2l import torch as d2l
```

Veri kümesinin tar dosyası yaklaşık 2 GB'dir, bu nedenle dosyayı indirmek biraz zaman alabilir. Ayıklanan veri kümesi `../data/VOCdevkit/VOC2012` adresindedir.

```
#@save
d2l.DATA_HUB['voc2012'] = (d2l.DATA_URL + 'VOCtrainval_11-May-2012.tar',
                           '4e443f8a2eca6b1dac8a6c57641b67dd40621a49')
```

(continues on next page)

[182] http://host.robots.ox.ac.uk/pascal/VOC/voc2012/

(continued from previous page)

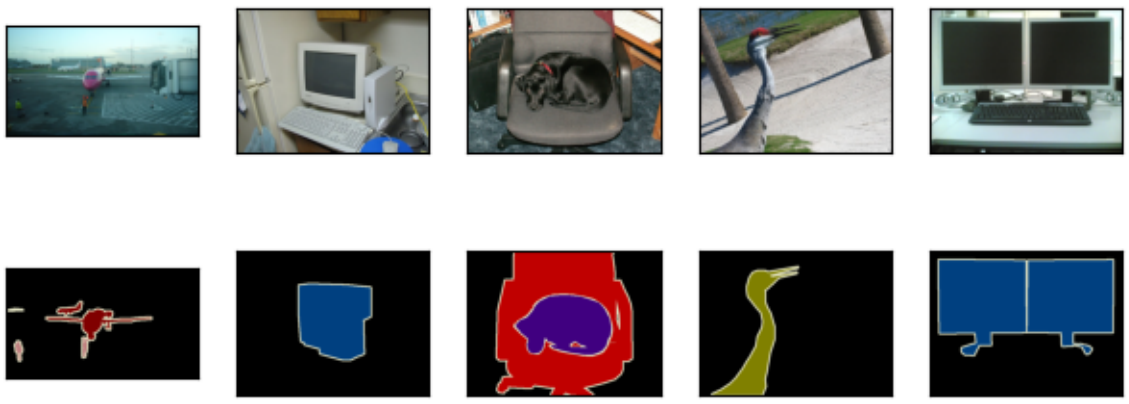
```
voc_dir = d2l.download_extract('voc2012', 'VOCdevkit/VOC2012')
```

`../data/VOCdevkit/VOC2012`'a girdikten sonra, veri kümesinin farklı bileşenlerini görebiliriz. `ImageSets/Segmentation` adresi, eğitim ve test örneklerini belirten metin dosyaları içerirken, `JPEGImages` ve `SegmentationClass` adresleri sırasıyla her örnek için girdi imgesini ve etiketini depolar. Buradaki etiket aynı zamanda etiketlenmiş girdi imgesiyle aynı boyutta imge biçimindedir. Ayrıca, herhangi bir etiket imgesinde aynı renge sahip pikseller aynı anlamsal sınıfa aittir. Aşağıda, `read_voc_images` işlevi tüm girdi imgelerini ve etiketlerini belleğe okumak için tanımlanır.

```
#@save
def read_voc_images(voc_dir, is_train=True):
    """Tüm VOC öznitelikleri okuyun ve resimleri etiketleyin."""
    txt_fname = os.path.join(voc_dir, 'ImageSets', 'Segmentation',
                             'train.txt' if is_train else 'val.txt')
    mode = torchvision.io.image.ImageReadMode.RGB
    with open(txt_fname, 'r') as f:
        images = f.read().split()
    features, labels = [], []
    for i, fname in enumerate(images):
        features.append(torchvision.io.read_image(os.path.join(
            voc_dir, 'JPEGImages', f'{fname}.jpg')))
        labels.append(torchvision.io.read_image(os.path.join(
            voc_dir, 'SegmentationClass' ,f'{fname}.png'), mode))
    return features, labels

train_features, train_labels = read_voc_images(voc_dir, True)
```

İlk beş girdi imgesini ve etiketlerini çizdiriyoruz. Etiket imgelerinde, beyaz ve siyah sırasıyla kenarlıkları ve arkaplanı temsil ederken, diğer renkler farklı sınıflara karşılık gelir.

```
n = 5
imgs = train_features[0:n] + train_labels[0:n]
imgs = [img.permute(1,2,0) for img in imgs]
d2l.show_images(imgs, 2, n);
```



Ardından, bu veri kümesindeki tüm etiketler için RGB renk değerlerini ve sınıf adları numara-

landırırız.

```
#@save
VOC_COLORMAP = [[0, 0, 0], [128, 0, 0], [0, 128, 0], [128, 128, 0],
                [0, 0, 128], [128, 0, 128], [0, 128, 128], [128, 128, 128],
                [64, 0, 0], [192, 0, 0], [64, 128, 0], [192, 128, 0],
                [64, 0, 128], [192, 0, 128], [64, 128, 128], [192, 128, 128],
                [0, 64, 0], [128, 64, 0], [0, 192, 0], [128, 192, 0],
                [0, 64, 128]]

#@save
VOC_CLASSES = ['background', 'aeroplane', 'bicycle', 'bird', 'boat',
               'bottle', 'bus', 'car', 'cat', 'chair', 'cow',
               'diningtable', 'dog', 'horse', 'motorbike', 'person',
               'potted plant', 'sheep', 'sofa', 'train', 'tv/monitor']
```

Yukarıda tanımlanan iki sabit ile etiketteki her piksel için sınıf dizinini bulabiliriz. Yukarıdaki RGB renk değerlerinden sınıf dizinlerine eşleştirmeyi oluşturmak için `voc_colormap2label` işlevini ve bu Pascal VOC2012 veri kümesindeki herhangi bir RGB değerlerini sınıf dizinleriyle eşlemek için `voc_label_indices` işlevini tanımlıyoruz.

```
#@save
def voc_colormap2label():
    """VOC etiketleri için RGB'den sınıf dizinlerine eşleme oluşturun."""
    colormap2label = torch.zeros(256 ** 3, dtype=torch.long)
    for i, colormap in enumerate(VOC_COLORMAP):
        colormap2label[
            (colormap[0] * 256 + colormap[1]) * 256 + colormap[2]] = i
    return colormap2label

#@save
def voc_label_indices(colormap, colormap2label):
    """VOC etiketlerindeki tüm RGB değerlerini sınıf dizinleriyle eşleyin."""
    colormap = colormap.permute(1, 2, 0).numpy().astype('int32')
    idx = ((colormap[:, :, 0] * 256 + colormap[:, :, 1]) * 256
           + colormap[:, :, 2])
    return colormap2label[idx]
```

Örneğin, ilk örnek imgede, uçağın ön kısmının sınıf indeksi 1 iken arka plan indeksi 0 olur.

```
y = voc_label_indices(train_labels[0], voc_colormap2label())
y[105:115, 130:140], VOC_CLASSES[1]
```

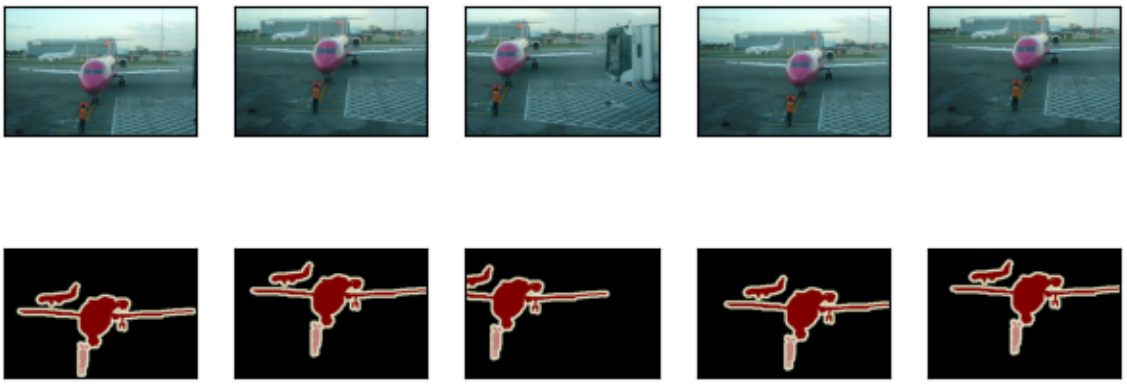
```
(tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
         [0, 0, 0, 0, 0, 0, 0, 1, 1, 1],
         [0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 0, 0, 0, 1, 1]]),
 'aeroplane')
```

Veri Ön İşleme

Section 7.1—-Section 7.4 içinde olduğu gibi önceki deneylerde imgeler modelin gerekli girdi şekline uyacak şekilde yeniden ölçeklendirildi. Ancak, anlamsal bölümlemede, bunun yapılması, tahmin edilen piksel sınıflarının girdi imgesinin orijinal şekline geri ölçeklenmesini gerektirir. Bu tür yeniden ölçeklendirme, özellikle farklı sınıflara sahip bölümlenmiş bölgeler için yanlış olabilir. Bu sorunu önlemek için, imgeyi yeniden ölçekleme yerine *sabit* bir şekle kırpıyoruz. Özellikle, imge artırımındaki rasgele kırpmayı kullanarak, girdi imgesinin ve etiketinin aynı alanını keseriz.

```
#@save
def voc_rand_crop(feature, label, height, width):
    """Hem öznitelik hem de etiket resimlerini rastgele kırpın."""
    rect = torchvision.transforms.RandomCrop.get_params(
        feature, (height, width))
    feature = torchvision.transforms.functional.crop(feature, *rect)
    label = torchvision.transforms.functional.crop(label, *rect)
    return feature, label
```

```
imgs = []
for _ in range(n):
    imgs += voc_rand_crop(train_features[0], train_labels[0], 200, 300)

imgs = [img.permute(1, 2, 0) for img in imgs]
d2l.show_images(imgs[::2] + imgs[1::2], 2, n);
```



Özel Anlamsal Bölümleme Veri Kümesi Sınıfı

Yüksek düzey API'ler tarafından sağlanan `Dataset` sınıfını devralarak özel bir anlamsal bölümleme veri kümesi sınıfı `VOCSegDataset`'i tanımlıyoruz. `__getitem__` işlevini uygulayarak, veri kümesindeki `idx` olarak dizinlenmiş girdi imgesine ve bu imgedeki her pikselin sınıf dizinine keyfi olarak erişebiliriz. Veri kümelerindeki bazı imgeler rasgele kırpma çıktı boyutundan daha küçük bir boyuta sahip olduğundan, bu örnekler özel bir `filter` işlevi tarafından filtrelenir. Buna ek olarak, girdi imgelerinin üç RGB kanalının değerlerini standartlaştırmak için `normalize_image` işlevini de tanımlıyoruz.

```
#@save
class VOCSegDataset(torch.utils.data.Dataset):
    """VOC veri kümesini yüklemek için özelleştirilmiş bir veri kümesi."""

    def __init__(self, is_train, crop_size, voc_dir):
        self.transform = torchvision.transforms.Normalize(
            mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
        self.crop_size = crop_size
        features, labels = read_voc_images(voc_dir, is_train=is_train)
        self.features = [self.normalize_image(feature)
                         for feature in self.filter(features)]
        self.labels = self.filter(labels)
        self.colormap2label = voc_colormap2label()
        print('read ' + str(len(self.features)) + ' examples')

    def normalize_image(self, img):
        return self.transform(img.float() / 255)

    def filter(self, imgs):
        return [img for img in imgs if (
            img.shape[1] >= self.crop_size[0] and
            img.shape[2] >= self.crop_size[1])]

    def __getitem__(self, idx):
        feature, label = voc_rand_crop(self.features[idx], self.labels[idx],
                                       *self.crop_size)
        return (feature, voc_label_indices(label, self.colormap2label))

    def __len__(self):
        return len(self.features)
```

Veri Kümesini Okuma

Eğitim kümesinin ve test kümesinin örneklerini oluşturmak için özel `VOCSegDataset` sınıfını kullanıyoruz. Rastgele kırpılmış imgelerin çıktı şeklinin 320×480 olduğunu belirttiğimizi varsayalım. Aşağıda, eğitim kümesinde ve test kümesinde tutulan örneklerin sayısını görebiliriz.

```
crop_size = (320, 480)
voc_train = VOCSegDataset(True, crop_size, voc_dir)
voc_test = VOCSegDataset(False, crop_size, voc_dir)
```

```
read 1114 examples
read 1078 examples
```

Toplu iş boyutunu 64 olarak ayarlarken, eğitim kümesi için veri yineleyicisini tanımlarız. İlk mini-igrup şeklini yazdıralım. İmge sınıflandırması veya nesne algılamasından farklı olarak, buradaki etiketler üç boyutlu tensörlerdir.

```
batch_size = 64
train_iter = torch.utils.data.DataLoader(voc_train, batch_size, shuffle=True,
                                         drop_last=True,
```

(continues on next page)

(continued from previous page)

```
                                    num_workers=d2l.get_dataloader_workers())
for X, Y in train_iter:
    print(X.shape)
    print(Y.shape)
    break
```

```
torch.Size([64, 3, 320, 480])
torch.Size([64, 320, 480])
```

Her Şeyi Biraraya Koyma

Son olarak, Pascal VOC2012 anlamsal bölümleme veri kümesini indirmek ve okumak için aşağıdaki `load_data_voc` işlevini tanımlıyoruz. Hem eğitim hem de test veri kümeleri için veri yineleyicilerini döndürür.

```
#@save
def load_data_voc(batch_size, crop_size):
    """Load the VOC semantic segmentation dataset."""
    voc_dir = d2l.download_extract('voc2012', os.path.join(
        'VOCdevkit', 'VOC2012'))
    num_workers = d2l.get_dataloader_workers()
    train_iter = torch.utils.data.DataLoader(
        VOCSegDataset(True, crop_size, voc_dir), batch_size,
        shuffle=True, drop_last=True, num_workers=num_workers)
    test_iter = torch.utils.data.DataLoader(
        VOCSegDataset(False, crop_size, voc_dir), batch_size,
        drop_last=True, num_workers=num_workers)
    return train_iter, test_iter
```

13.9.3 Özet

- Anlamsal bölümleme, imgeyi farklı anlamsal sınıflara ait bölgelere bölerek piksel düzeyinde bir imgede ne olduğunu tanır ve anlar.
- En önemli anlamsal bölümleme veri kümesi Pascal VOC2012'dir.
- Anlamsal bölümlemede, girdi imgesi ve etiketi piksel üzerinde bire bir karşılık geldiğinden, girdi imgesi yeniden ölçeklenmek yerine rastgele sabit bir şekle kırpılır.

13.9.4 Alıştırmalar

1. Otonom araçlarda ve tıbbi imge teşhislerinde anlamsal bölümleme nasıl uygulanabilir? Başka uygulamalar düşünebiliyor musun?
2. Section 13.1 içindeki veri artırımı açıklamalarını hatırlayın. İmge sınıflandırmasında kullanılan imge artırma yöntemlerinden hangisinin anlamsal bölümlemede uygulanması mümkün olmayacaktır?

13.10 Devrik Evrişim

Bugüne kadar gördüğümüz CNN katmanları, evrişimli katmanlar (Section 6.2) ve ortaklama katmanları (Section 6.5) gibi, genellikle girdinin uzamsal boyutlarını (yükseklik ve genişlik) azaltır (örnek seyreltme) veya değişmeden tutar. Piksel düzeyinde sınıflandırılan anlamsal bölümlemede, girdi ve çıktının mekansal boyutları aynı ise uygun olacaktır. Örneğin, bir çıktı pikselindeki kanal boyutu, girdi pikselinin sınıflandırma sonuçlarını aynı uzamsal konumda tutabilir.

Bunu başarmak için, özellikle uzamsal boyutlar CNN katmanları tarafından azaltıldıktan sonra, ara öznitelik haritalarının mekansal boyutlarını artırabilecek (örnek sıklaştırma) yapabilen başka bir CNN katmanlarını kullanabiliriz. Bu bölümde, evrişim tarafından örnek seyreltme işlemlerini tersine çevirmek için *kesirli adımlı evrişim* (Dumoulin and Visin, 2016) olarak da adlandırılan *devrik evrişimi* tanıtacağız.

```
import torch
from torch import nn
from d2l import torch as d2l
```

13.10.1 Temel İşlem

Şimdilik kanalları görmezden gelerek, 1 adımlı ve dolgusuz temel devrik evrişim işlemiyle başlayalım. Bir $n_h \times n_w$ girdi tensörü ve bir $k_h \times k_w$ çekirdeği verildiğini varsayalım. Çekirdek penceresini her satırda n_w kez ve her sütundaki n_h kez 1 adımıyla kaydırılması toplam $n_h n_w$ ara sonuç verir. Her ara sonuç sıfır olarak ilklenen bir $(n_h + k_h - 1) \times (n_w + k_w - 1)$ tensördür. Her ara tensörün hesaplanması için, girdi tensöründe bulunan her eleman çekirdek ile çarpılır, böylece sonuçta ortaya çıkan $k_h \times k_w$ tensörü her ara tensörde bir kısmın yerini alır. Her ara tensördeki değiştirilen kısmın konumunun, hesaplama için kullanılan girdi tensöründe elemanın konumuna karşılık geldiğini unutmayın. Sonunda, çıktı üretmek için tüm ara sonuçlar toplanır.

Örneğin, Fig. 13.10.1, 2×2 girdi tensörü için 2×2 çekirdeği ile devrik evrişimin nasıl hesaplandığını göstermektedir.

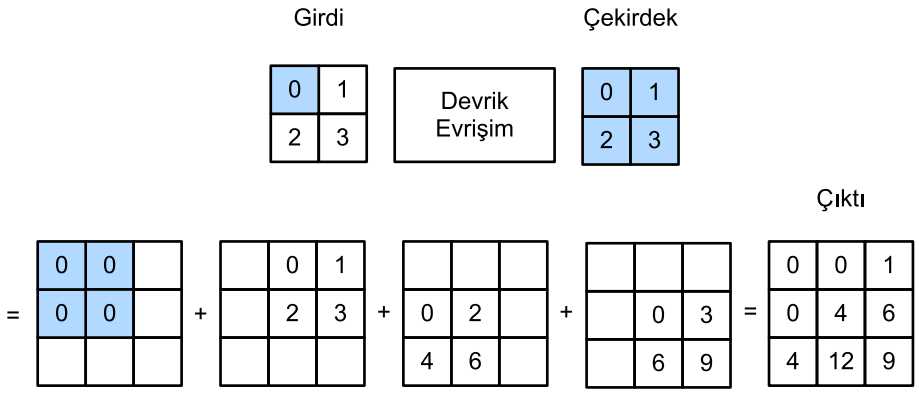


Fig. 13.10.1: 2×2 çekirdek ile devrik evrişim. Gölgeli kısımlar, bir ara tensörün bir kısmı ile hesaplama için kullanılan girdi ve çekirdek tensör elemanlarıdır.

[183] https://discuss.d2l.ai/t/1480

Bir girdi matrisi `X` ve bir çekirdek matrisi `K` için bu temel devrik evrişim işlemi `trans_conv`'u uygulayabiliriz.

```
def trans_conv(X, K):
    h, w = K.shape
    Y = torch.zeros((X.shape[0] + h - 1, X.shape[1] + w - 1))
    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            Y[i: i + h, j: j + w] += X[i, j] * K
    return Y
```

Çekirdek yoluyla girdi öğelerini *azaltan* olağan evrişimin (Section 6.2) aksine, devrik evrişim, girdi öğelerini çekirdek yoluyla *yayınlar*, böylece girdiden daha büyük bir çıktı üretir. Temel iki boyutlu devrik evrişim işleminin yukarıdaki uygulamanın çıktısını doğrulamak için Fig. 13.10.1 şeklindeki girdi tensörü `X`'i ve çekirdek tensörü `K`'yi oluşturabiliriz.

```
X = torch.tensor([[0.0, 1.0], [2.0, 3.0]])
K = torch.tensor([[0.0, 1.0], [2.0, 3.0]])
trans_conv(X, K)
```

```
tensor([[ 0.,  0.,  1.],
        [ 0.,  4.,  6.],
        [ 4., 12.,  9.]])
```

Alternatif olarak, girdi `X` ve çekirdek `K` dört boyutlu tensörler olduğunda, aynı sonuçları elde etmek için üst seviye API'leri kullanabiliriz.

```
X, K = X.reshape(1, 1, 2, 2), K.reshape(1, 1, 2, 2)
tconv = nn.ConvTranspose2d(1, 1, kernel_size=2, bias=False)
tconv.weight.data = K
tconv(X)
```

```
tensor([[[[ 0.,  0.,  1.],
          [ 0.,  4.,  6.],
          [ 4., 12.,  9.]]]], grad_fn=<ConvolutionBackward0>)
```

13.10.2 Dolgu, Adım ve Çoklu Kanallar

Dolgunun girdiye uygulandığı düzenli evrişimden farklı olarak, devrik evrişim içindeki çıktıya uygulanır. Örneğin, yükseklik ve genişliğin her iki tarafındaki dolgu sayısı 1 olarak belirtilirken, ilk ve son satırlar ve sütunlar devrik evrişim çıktısından kaldırılır.

```
tconv = nn.ConvTranspose2d(1, 1, kernel_size=2, padding=1, bias=False)
tconv.weight.data = K
tconv(X)
```

```
tensor([[[[4.]]]], grad_fn=<ConvolutionBackward0>)
```

Devrik evrişimde, girdi için değil, ara sonuçlar (böylece çıktı) için adımlar belirtilir. Fig. 13.10.1 içinde bahsedilen aynı girdi ve çekirdek tensörleri kullanılırken, adımın 1'den 2'ye değiştirilmesi,

ara tensörlerin, dolayısıyla Fig. 13.10.2 şeklinde gösterilen çıktı tensörünün, hem yüksekliğini hem de ağırlığını artırır.

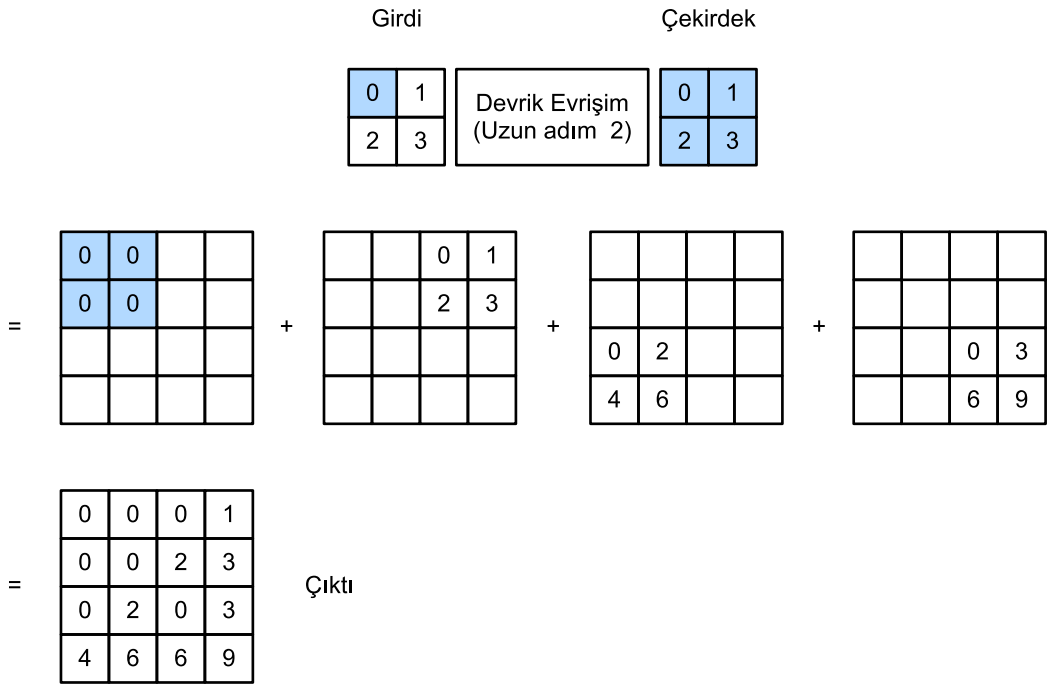


Fig. 13.10.2: 2 uzun adımlı 2×2 çekirdekli devrik evrişim. Gölgeli kısımlar, hesaplama için kullanılan girdi ve çekirdek tensör elemanlarının yanı sıra bir ara tensörün bir kısmıdır.

Aşağıdaki kod parçacığı, Fig. 13.10.2'te 2 uzun adım için devrik evrişim çıktısını doğrulayabilir.

```
tconv = nn.ConvTranspose2d(1, 1, kernel_size=2, stride=2, bias=False)
tconv.weight.data = K
tconv(X)
```

```
tensor([[[[0., 0., 0., 1.],
          [0., 0., 2., 3.],
          [0., 2., 0., 3.],
          [4., 6., 6., 9.]]]], grad_fn=<ConvolutionBackward0>)
```

Çoklu girdi ve çıktı kanalı için, devrik evrişim normal evrişim ile aynı şekilde çalışır. Girdinin c_i kanallara sahip olduğunu ve devrik evrişimin her girdi kanalına bir $k_h \times k_w$ çekirdek tensörü atadığını varsayalım. Birden fazla çıktı kanalı belirtildiğinde, her çıktı kanalı için bir $c_i \times k_h \times k_w$ çekirdeğine sahip olacağız.

Her durumda, X'i $Y = f(X)$ çıktısı almak için bir f evrişim katmanına beslersek ve X içindeki kanalların sayısı olan çıktı kanallarının sayısı dışında f ile aynı hiper parametrelere sahip bir g devrik evrimsel katman oluşturursak, o zaman $g(Y)$ X ile aynı şekle sahip olacaktır. Bu, aşağıdaki örnekte gösterilebilir.

```
X = torch.rand(size=(1, 10, 16, 16))
conv = nn.Conv2d(10, 20, kernel_size=5, padding=2, stride=3)
tconv = nn.ConvTranspose2d(20, 10, kernel_size=5, padding=2, stride=3)
tconv(conv(X)).shape == X.shape
```

```
True
```

13.10.3 Matris Devirme ile Bağlantı

Devrik evrişim, matris devirme sonrasından adlandırılmıştır. Açıklamak için, önce matris çarpımlarını kullanarak evrişimlerin nasıl uygulanacağını görelim. Aşağıdaki örnekte, 3×3'lük girdi `X` ve 2×2'lik evrişim çekirdeği `K` tanımlıyoruz ve `Y` evrişim çıktısını hesaplamak için `corr2d` işlevini kullanıyoruz.

```
X = torch.arange(9.0).reshape(3, 3)
K = torch.tensor([[1.0, 2.0], [3.0, 4.0]])
Y = d2l.corr2d(X, K)
Y
```

```
tensor([[27., 37.],
        [57., 67.]])
```

Daha sonra, çok sayıda sıfır içeren seyrek ağırlık matrisi `W` olarak evrişim çekirdeği `K`'yi yeniden yazıyoruz. Ağırlık matrisinin şekli $(4, 9)$ olup sıfır olmayan elemanlar evrişim çekirdeği `K`'den gelmektedir.

```
def kernel2matrix(K):
    k, W = torch.zeros(5), torch.zeros((4, 9))
    k[:2], k[3:5] = K[0, :], K[1, :]
    W[0, :5], W[1, 1:6], W[2, 3:8], W[3, 4:] = k, k, k, k
    return W

W = kernel2matrix(K)
W
```

```
tensor([[1., 2., 0., 3., 4., 0., 0., 0., 0.],
        [0., 1., 2., 0., 3., 4., 0., 0., 0.],
        [0., 0., 0., 1., 2., 0., 3., 4., 0.],
        [0., 0., 0., 0., 1., 2., 0., 3., 4.]])
```

9 uzunluğunda bir vektör elde etmek için girdi `X`'i satır satır bitiştirin. Daha sonra `W` ve vektörleştirilmiş `X`'in matris çarpımı, 4 uzunluğunda bir vektör verir. Yeniden şekillendirdikten sonra, yukarıdaki orijinal evrişim işlemindeki aynı `Y` sonucunu elde edebiliriz: Matris çarpımlarını kullanarak evrişimleri uyguladık.

```
Y == torch.matmul(W, X.reshape(-1)).reshape(2, 2)
```

```
tensor([[True, True],
        [True, True]])
```

Aynı şekilde, matris çarpımlarını kullanarak devrik evrişimleri de uygulayabiliriz. Aşağıdaki örnekte, yukarıdaki olağan evrişimin 2×2 çıktısı `Y`'yi devrik evrişime girdi olarak alıyoruz. Bu işlemi matrisleri çarparak uygulamak için, `W` ağırlık matrisini yeni şekli $(9, 4)$ ile devirmemiz yeterlidir.

```
Z = trans_conv(Y, K)
Z == torch.matmul(W.T, Y.reshape(-1)).reshape(3, 3)
```

```
tensor([[True, True, True],
        [True, True, True],
        [True, True, True]])
```

Matrisleri çarparak evrişimi uygulamayı düşünün. Bir girdi vektörü $\mathbf{x}$ ve bir ağırlık matrisi $\mathbf{W}$ göz önüne alındığında, evrişimin ileri yayma fonksiyonu, girdi ağırlık matrisi ile çarpımından bir $\mathbf{y} = \mathbf{W}\mathbf{x}$ vektör çıktısı verilerek uygulanabilir. Geriye yayma zincir kuralını ve $\nabla_{\mathbf{x}}\mathbf{y} = \mathbf{W}^\top$'i izlediğinden, evrişimin geriye yayma fonksiyonu, girdisini devrik ağırlık matrisi $\mathbf{W}^\top$ ile çarparak uygulanabilir. Bu nedenle, devrik evrişim katmanı sadece ileri yayma fonksiyonunu ve evrişim tabakasının geri yayma fonksiyonunu değiştirebilir: İleri yayma ve geri yayma fonksiyonları sırasıyla $\mathbf{W}^\top$ ve $\mathbf{W}$ ile girdi vektörünü çarpar.

13.10.4 Özet

- Çekirdek üzerinden girdi elemanlarını azaltan olağan evrişimin aksine, devrik evrişim çekirdek üzerinden girdi öğelerini yayınlar ve böylece girdiden daha büyük bir çıktı üretir.
- X'i `Y` $= f(\mathsf{X})$ çıktısı almak için bir f evrişim katmanına beslersek ve çıktı kanallarının sayısının `X` içindeki kanalların sayısı olması dışında f ile aynı hiper parametrelere sahip bir devrik evrişim katmanı g oluşturursak, $g(Y)$ `X` ile aynı şekle sahip olacaktır.
- Matris çarpımlarını kullanarak evrişimleri gerçekleştirebiliriz. Devrik evrişimli katman sadece ileri yayma fonksiyonunu ve evrişimli katmanın geri yayma işlevini değiştirebilir.

13.10.5 Alıştırmalar

1. Section 13.10.3 içinde, evrişim girdisi `X` ve devrik evrişim çıktısı `Z` aynı şekle sahiptir. Aynı değere sahipler mi? Neden?
2. Evrişimleri uygulamak için matris çarpımlarını kullanmak verimli midir? Neden?

Tartışmalar[184]

13.11 Tam Evrişimli Ağlar

Section 13.9 içinde tartışıldığı gibi, anlamsal bölümleme imgeleri piksel düzeyinde sınıflandırır. Bir tam evrişimli ağ (FCN), imge piksellerini (Long *et al.*, 2015) piksel sınıflarına dönüştürmek için bir evrişimli sinir ağı kullanır. İmge sınıflandırması veya nesne algılama için daha önce karşılaştığımız CNN'lerden farklı olarak, tam evrişimli bir ağ, ara öznitelik haritalarının yüksekliğini ve genişliğini girdi imgesindekine geri dönüştürür: bu, Section 13.10 içinde tanıtılan devrik evrişimli katman ile elde edilir. Sonuç olarak, sınıflandırma çıktısı ve girdi imgesi, piksel düzeyinde bire bir karşılığa sahiptir: Herhangi bir çıktı pikselindeki kanal boyutu, girdi pikseli için sınıflandırma sonuçlarını aynı uzamsal konumda tutar.

[184] https://discuss.d2l.ai/t/1450

```
%matplotlib inline
import torch
import torchvision
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

13.11.1 Model

Burada tam evrişimli ağ modelinin temel tasarımını açıklıyoruz. Fig. 13.11.1 içinde gösterildiği gibi, bu model ilk olarak imge özniteliklerini ayıklamak için bir CNN kullanır, daha sonra 1×1 evrişimli katman aracılığıyla kanal sayısını sınıf sayısına dönüştürür ve son olarak öznitelik haritalarının yüksekliğini ve genişliğini Section 13.10 içinde tanıtılan devrik evrişim yoluyla girdi imgesine dönüştürür. Sonuç olarak, model çıktısı, girdi imgesi ile aynı yüksekliğe ve genişliğe sahiptir; burada çıktı kanalı, aynı uzaysal konumda girdi pikseli için tahmin edilen sınıfları içerir.

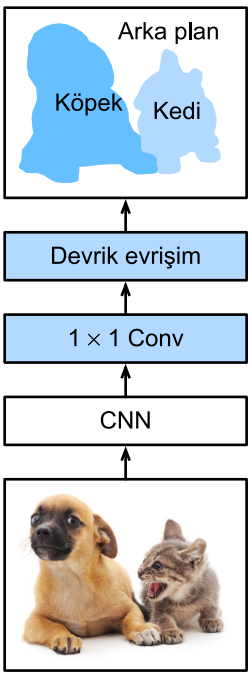


Fig. 13.11.1: Tam evrişimli ağ.

Aşağıda, imge özelliklerini ayıklamak için ImageNet veri kümesi üzerinde önceden eğitilmiş bir ResNet-18 modeli kullanıyor ve model örneğini `pretrained_net` olarak belirtiyoruz. Bu modelin son birkaç katmanı küresel ortalama ortaklama katmanı ve tam bağlı bir katman içerir: Bunlar tam evrişimli ağda gerekli değildir.

```
pretrained_net = torchvision.models.resnet18(pretrained=True)
list(pretrained_net.children())[-3:]
```

```
/home/d2l-worker/miniconda3/envs/d2l-tr-release-0/lib/python3.9/site-packages/
↪torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is␣
↪deprecated since 0.13 and will be removed in 0.15, please use 'weights' instead.
  warnings.warn(
```

```
/home/d2l-worker/miniconda3/envs/d2l-tr-release-0/lib/python3.9/site-packages/
↪torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum␣
↪or None for 'weights' are deprecated since 0.13 and will be removed in 0.15. The␣
↪current behavior is equivalent to passing weights=ResNet18_Weights.IMAGENET1K_
↪V1. You can also use weights=ResNet18_Weights.DEFAULT to get the most up-to-date␣
↪weights.
  warnings.warn(msg)
```

```
[Sequential(
   (0): BasicBlock(
     (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
     (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
     (relu): ReLU(inplace=True)
     (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
     (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
     (downsample): Sequential(
       (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
       (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
     )
   )
   (1): BasicBlock(
     (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
     (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
     (relu): ReLU(inplace=True)
     (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
     (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
   )
 ),
 AdaptiveAvgPool2d(output_size=(1, 1)),
 Linear(in_features=512, out_features=1000, bias=True)]
```

Ardından, tam evrişimli ağ örneği `net`'i oluşturuyoruz. Son global ortalama ortaklama katmanı ve çıktıya en yakın tam bağlı katman dışında ResNet-18'deki tüm önceden eğitilmiş katmanları kopyalar.

```
net = nn.Sequential(*list(pretrained_net.children())[:-2])
```

Yüksekliği ve genişliği sırasıyla 320 ve 480 olan bir girdi göz önüne alındığında, `net`'in ileri yayması girdi yüksekliğini ve genişliğini orijinalin 1/32'sine, yani 10 ve 15'e düşürür.

```
X = torch.rand(size=(1, 3, 320, 480))
net(X).shape
```

```
torch.Size([1, 512, 10, 15])
```

Ardından, çıktı kanallarının sayısını Pascal VOC2012 veri kümesinin sınıf sayısına (21) dönüştürmek için bir 1×1 evrişimli katman kullanıyoruz. Son olarak, girdi imgesinin yüksekliğine ve genişliğine geri döndürmek için öznitelik haritalarının yüksekliğini ve genişliğini 32 kat artırmaya ihtiyacımız var. Section 6.3 içindeki bir evrişimli katmanının çıktı şeklini nasıl hesaplayacağınızı hatırlayın. $(320 - 64 + 16 \times 2 + 32)/32 = 10$ ve $(480 - 64 + 16 \times 2 + 32)/32 = 15$ olduğundan, çekirdeğin yüksekliği ve genişliği 64'e, dolguyu ise 16'ya ayarlayarak 32 uzun adımlı olan devrik bir evrişim katmanı oluşturuyoruz. Genel olarak, s uzun adımı için, dolgu $s/2$ ($s/2$ bir tam sayı olduğu varsayılarak) ve çekirdek yüksekliği ve genişliği $2s$'dir, devrik evrişimin girdi

yüksekliğini ve genişliğini s kat artıracağını görebilirsiniz.

```
num_classes = 21
net.add_module('final_conv', nn.Conv2d(512, num_classes, kernel_size=1))
net.add_module('transpose_conv', nn.ConvTranspose2d(num_classes, num_classes,
                                    kernel_size=64, padding=16, stride=32))
```

13.11.2 Devrik Evrişimli Katmanları İlkleme

Devrik evrişimli katmanların öznitelik haritalarının yüksekliğini ve genişliğini artırabileceğini zaten biliyoruz. İmge işlemede, bir imgeyi büyütmemiz gerekebilir, yani, *örnek sıklaştırma*. *Çift doğrusal aradeğerleme* yaygın olarak kullanılan örnek sıklaştırma tekniklerinden biridir. Ayrıca, devrik evrişimli tabakaların ilklenmesi için de sıklıkla kullanılır.

Çift doğrusal aradeğerlemeyi açıklamak için, bir girdi imgesi göz önüne alındığında, örnek sıklaştırılan çıktı imgesinin her pikselini hesaplamak istediğimizi varsayalım. Çıktı imgesinin pikselini (x, y) koordinatında hesaplamak için, ilk önce (x, y) ile girdi imgesindeki (x', y') koordinatını eşleyin, örneğin, girdi boyutunun çıktı boyutuna oranı. Eşlenen x' ve y''nin gerçel sayılar olduğuna dikkat edin. Ardından, girdi imgesinde (x', y') koordinatına en yakın dört pikseli bulun. Son olarak, (x, y) koordinatındaki çıktı imgesinin pikseli, girdi imgesindeki bu dört en yakın piksele ve onların (x', y')'dan göreli mesafelerine dayanarak hesaplanır.

Çift doğrusal aradeğerleme örnek sıklaştırması, aşağıdaki `bilinear_kernel` işlevi tarafından oluşturulan çekirdek ile devrik evrişimli katman tarafından gerçekleştirilebilir. Alan kısıtlamaları nedeniyle, algoritma tasarımı hakkında tartışmadan sadece aşağıdaki `bilinear_kernel` işlevinin uygulanmasını sağlıyoruz.

```
def bilinear_kernel(in_channels, out_channels, kernel_size):
    factor = (kernel_size + 1) // 2
    if kernel_size % 2 == 1:
        center = factor - 1
    else:
        center = factor - 0.5
    og = (torch.arange(kernel_size).reshape(-1, 1),
          torch.arange(kernel_size).reshape(1, -1))
    filt = (1 - torch.abs(og[0] - center) / factor) * \
           (1 - torch.abs(og[1] - center) / factor)
    weight = torch.zeros((in_channels, out_channels,
                          kernel_size, kernel_size))
    weight[range(in_channels), range(out_channels), :, :] = filt
    return weight
```

Bir devrik evrişimli katman tarafından uygulanan çift doğrusal aradeğerleme örnek sıklaştırmasını deneyelim. Yüksekliği ve ağırlığı iki katına çıkaran ve çekirdeğini `bilinear_kernel` işleviyle ilkleten bir devrik evrişimli katman oluşturuyoruz.

```
conv_trans = nn.ConvTranspose2d(3, 3, kernel_size=4, padding=1, stride=2,
                                bias=False)
conv_trans.weight.data.copy_(bilinear_kernel(3, 3, 4));
```

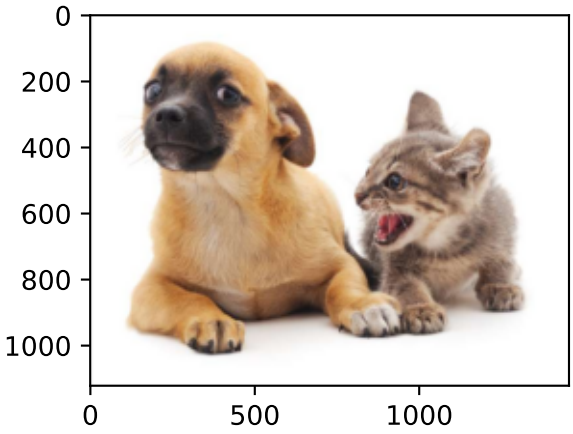
`X` imgesini okuyun ve örnekleme sıklaştırma çıktısını `Y`'ye atayın. İmgeyi yazdırmak için kanal boyutunun konumunu ayarlamamız gerekiyor.

```
img = torchvision.transforms.ToTensor()(d2l.Image.open('../img/catdog.jpg'))
X = img.unsqueeze(0)
Y = conv_trans(X)
out_img = Y[0].permute(1, 2, 0).detach()
```

Gördüğümüz gibi, devrik evrişimli tabaka, imgenin yüksekliğini ve genişliğini iki kat arttırır. Koordinatlardaki farklı ölçekler haricinde, çift doğrusal aradeğerleme ile büyütülmüş imge ve Section 13.3 içinde basılan orijinal imge aynı görünüyor.

```
d2l.set_figsize()
print('input image shape:', img.permute(1, 2, 0).shape)
d2l.plt.imshow(img.permute(1, 2, 0));
print('output image shape:', out_img.shape)
d2l.plt.imshow(out_img);
```

```
input image shape: torch.Size([561, 728, 3])
output image shape: torch.Size([1122, 1456, 3])
```



Bir tam evrişimli ağda, çift doğrusal aradeğerleme örnek sıklaştırma ile devrik evrişimli katmanı ilkliyoruz. 1×1 evrişimli katman için Xavier ilkleme kullanıyoruz.

```
W = bilinear_kernel(num_classes, num_classes, 64)
net.transpose_conv.weight.data.copy_(W);
```

13.11.3 Veri Kümesini Okuma

Section 13.9 içinde tanıtıldığı gibi anlamsal bölümleme veri kümesini okuduk. Rastgele kırpmanın çıktı imgesi şekli 320×480 olarak belirtilir: Hem yükseklik hem de genişlik 32 ile bölünebilir.

```
batch_size, crop_size = 32, (320, 480)
train_iter, test_iter = d2l.load_data_voc(batch_size, crop_size)
```

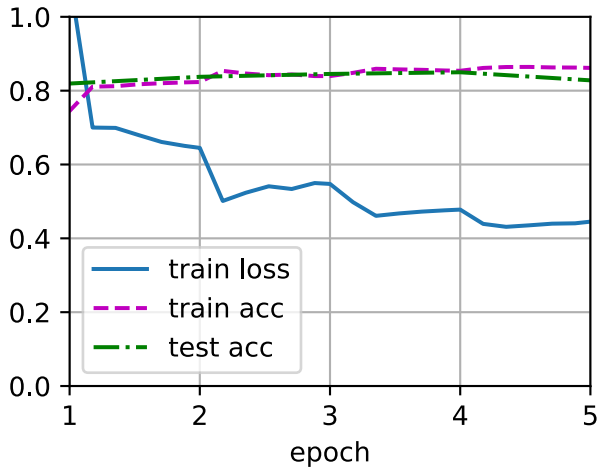
```
read 1114 examples
read 1078 examples
```

13.11.4 Eğitim

Şimdi oluşturduğumuz tam evrişimli ağımızı eğitebiliriz. Buradaki kayıp fonksiyonu ve doğruluk hesaplaması, önceki bölümlerin imge sınıflandırılmasındakilerden farklı değildir. Her piksel için sınıfı tahmin etmek için devrik evrişimli katmanın çıktı kanalını kullandığımızdan, kanal boyutu kayıp hesaplamasında belirtilir. Buna ek olarak, doğruluk, tüm pikseller için tahmin edilen sınıfın doğruluğuna göre hesaplanır.

```
def loss(inputs, targets):
    return F.cross_entropy(inputs, targets, reduction='none').mean(1).mean(1)

num_epochs, lr, wd, devices = 5, 0.001, 1e-3, d2l.try_all_gpus()
trainer = torch.optim.SGD(net.parameters(), lr=lr, weight_decay=wd)
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.445, train acc 0.862, test acc 0.828
251.3 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



13.11.5 Tahminleme

Tahmin ederken, her kanaldaki girdi imgesini standartlaştırmamız ve imgeyi CNN'nin gerek duyduğu dört boyutlu girdi formatına dönüştürmemiz gerekir.

```
def predict(img):
    X = test_iter.dataset.normalize_image(img).unsqueeze(0)
    pred = net(X.to(devices[0])).argmax(dim=1)
    return pred.reshape(pred.shape[1], pred.shape[2])
```

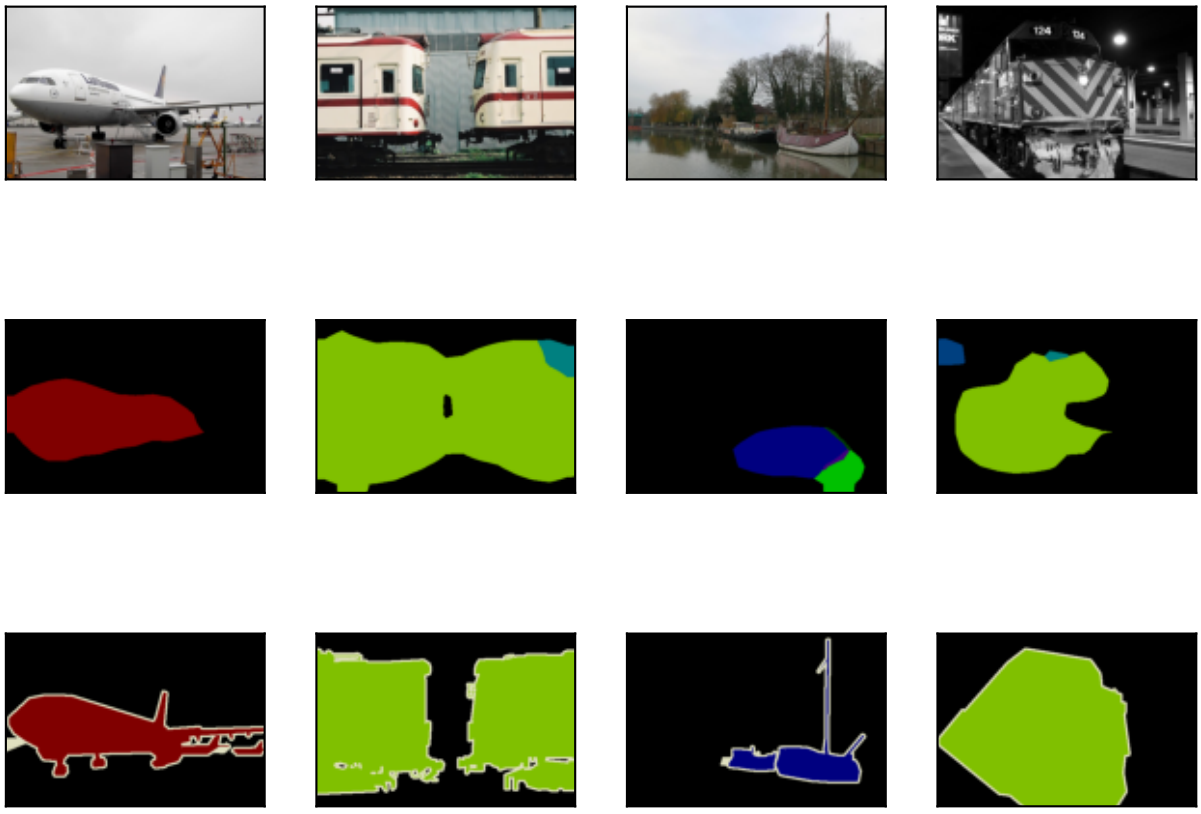
Her pikselin tahmin edilen sınıfını görselleştirmek için, tahmini sınıfı veri kümesindeki etiket rengine geri eşleriz.

```
def label2image(pred):
    colormap = torch.tensor(d2l.VOC_COLORMAP, device=devices[0])
    X = pred.long()
    return colormap[X, :]
```

Test veri kümelerindeki imgeler boyut ve şekil bakımından farklılık gösterir. Model, bir girdi imgesinin yüksekliği veya genişliği 32 ile bölünmez olduğunda, 32 adımlı bir devrik evrişimli katman kullandığından, devrik evrişimli katmanın çıktı yüksekliği veya genişliği girdi imgesinin şeklinden sapacaktır. Bu sorunu gidermek için, imgedeki 32 tamsayı katları olan yükseklik ve genişliğe sahip çoklu dikdörtgen alanı kırpabilir ve bu alanlardaki piksellerde ileri yaymayı ayrı ayrı gerçekleştirebiliriz. Bu dikdörtgen alanların birleşmesinin girdi imgesini tamamen örtmesi gerektiğini unutmayın. Bir piksel birden fazla dikdörtgen alanla kaplandığında, aynı piksel için ayrı alanlardaki devrik evrişim çıktılarının ortalaması sınıfı tahmin etmek için softmaks işlemine girilebilir.

Basitlik açısından, sadece birkaç büyük test imgesi okuruz ve imgenin sol üst köşesinden başlayarak tahmin için 320×480'lik bir alan kırpıyoruz. Bu test imgeleri için kırpılmış alanlarını, tahmin sonuçlarını ve gerçek referans değeri satır satır yazdırıyoruz.

```
voc_dir = d2l.download_extract('voc2012', 'VOCdevkit/VOC2012')
test_images, test_labels = d2l.read_voc_images(voc_dir, False)
n, imgs = 4, []
for i in range(n):
    crop_rect = (0, 0, 320, 480)
    X = torchvision.transforms.functional.crop(test_images[i], *crop_rect)
    pred = label2image(predict(X))
    imgs += [X.permute(1,2,0), pred.cpu(),
             torchvision.transforms.functional.crop(
                 test_labels[i], *crop_rect).permute(1,2,0)]
d2l.show_images(imgs[::3] + imgs[1::3] + imgs[2::3], 3, n, scale=2);
```



13.11.6 Özet

- Tam evrişimli ağ önce imge özniteliklerini ayıklamak için bir CNN kullanır, daha sonra 1×1'lik bir evrişimli katman aracılığıyla kanal sayısını sınıf sayısına dönüştürür ve son olarak öznitelik haritalarının yüksekliğini ve genişliğini devrik evrişim yoluyla girdi imgesine dönüştürür.
- Tam evrişimli bir ağda, devrik evrişimli tabakayı ilklemek için çift doğrusal aradeğerlendirme örnek sıklaştırmayı kullanabiliriz.

13.11.7 Alıştırmalar

1. Deneyde devrik evrişimli katman için Xavier ilkleme kullanırsak, sonuç nasıl değişir?
2. Hiper parametreleri ayarlayarak modelin doğruluğunu daha da iyileştirebilir misiniz?
3. Test imgelerindeki tüm piksellerin sınıflarını tahmin edin.
4. Orijinal tam evrişimli ağ makalesi, bazı ara CNN katmanlarının (Long *et al.*, 2015) çıktılarını da kullanır. Bu fikri uygulamaya çalışın.

Tartışmalar[185]

[185] https://discuss.d2l.ai/t/1582

13.12 Sinir Stil Transferi

Eğer bir fotoğraf meraklısı iseniz, filtreye aşina olabilirsiniz. Fotoğrafların renk stilini değiştirebilir, böylece manzara fotoğrafları daha keskin hale gelir veya portre fotoğrafları beyaz tonlara sahip olur. Ancak, bir filtre genellikle fotoğrafın yalnızca bir yönünü değiştirir. Bir fotoğrafa ideal bir stil uygulamak için muhtemelen birçok farklı filtre kombinasyonunu denemeniz gerekir. Bu işlem, bir modelin hiper parametrelerini ayarlamak kadar karmaşıktır.

Bu bölümde, bir imgenin stilini otomatik olarak başka bir imgeye (örn. *stil aktarımı* (Gatys *et al.*, 2016)) uygulamak için CNN'nin katmanlı temsillerinden yararlanacağız. Bu görevde iki girdi imgesi gerekir: Biri *içerik imgesi*, diğeri ise *stil imgesi*. İçerik imgesini stil imgesine yakın hale getirmek için sinir ağlarını kullanacağız. Örneğin, Fig. 13.12.1 içindeki içerik imgesi Seattle'ın banliyölerindeki Rainier Milli Parkı'nda tarafımızdan çekilen bir manzara fotoğrafıdır ve stil imgesi sonbahar meşe ağaçları temalı bir yağlı boya tablosudur. Sentezlenmiş çıktı imgesinde, stil imgesinin yağlı fırça darbeleri uygulanarak, içerik imgesindeki nesnelerin ana şekli korunurken daha canlı renkler elde edilir.

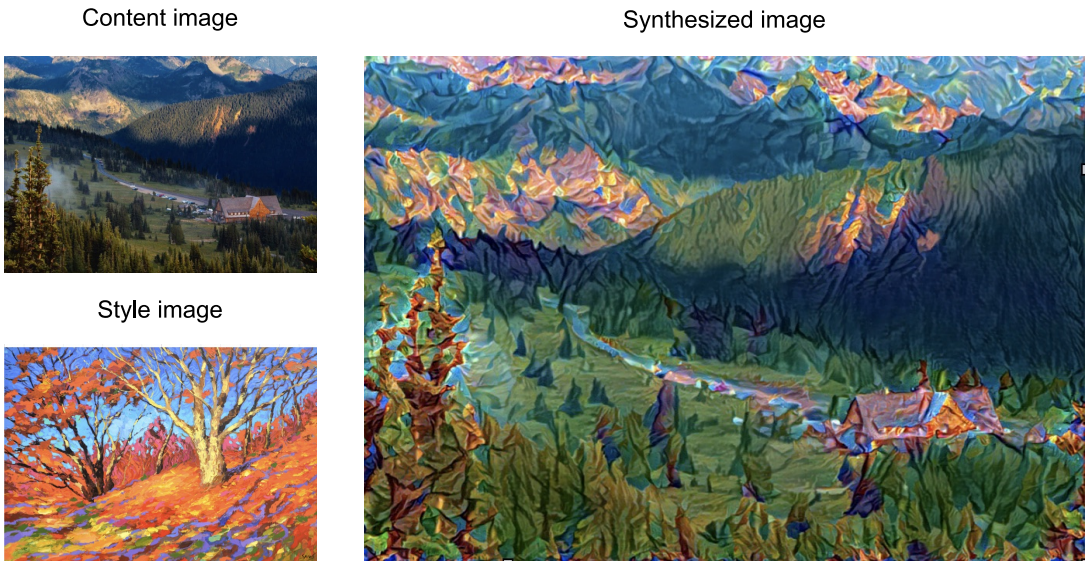


Fig. 13.12.1: Verilen içerik ve stil imgeleri, stil aktarımı sentezlenmiş bir imge verir.

13.12.1 Yöntem

Fig. 13.12.2, CNN tabanlı stil aktarım yöntemini basitleştirilmiş bir örnekle gösterir. İlk olarak, sentezlenen imgeyi, örneğin içerik imgesine ilkleriz. Bu sentezlenen imge, stil aktarımı işlemi sırasında güncellenmesi gereken tek değişkendir, yani eğitim sırasında güncellenecek model parametreleri. Daha sonra imge özniteliklerini ayıklamak için önceden eğitilmiş bir CNN seçiyoruz ve eğitim sırasında model parametrelerini donduruyoruz. Bu derin CNN imgeler için hiyerarşik öznitelikleri ayıklamak için çoklu katman kullanır. İçerik öznitelikleri veya stil öznitelikleri olarak bu katmanlardan bazılarının çıktısını seçebiliriz. Örnek olarak Fig. 13.12.2 figürünü ele alın. Buradaki önceden eğitilmiş sinir ağı, ikinci katmanın içerik özniteliklerini çıkardığı ve birinci ve üçüncü katmanlar stil özniteliklerini çıkardığı 3 evrişimli katmana sahiptir.

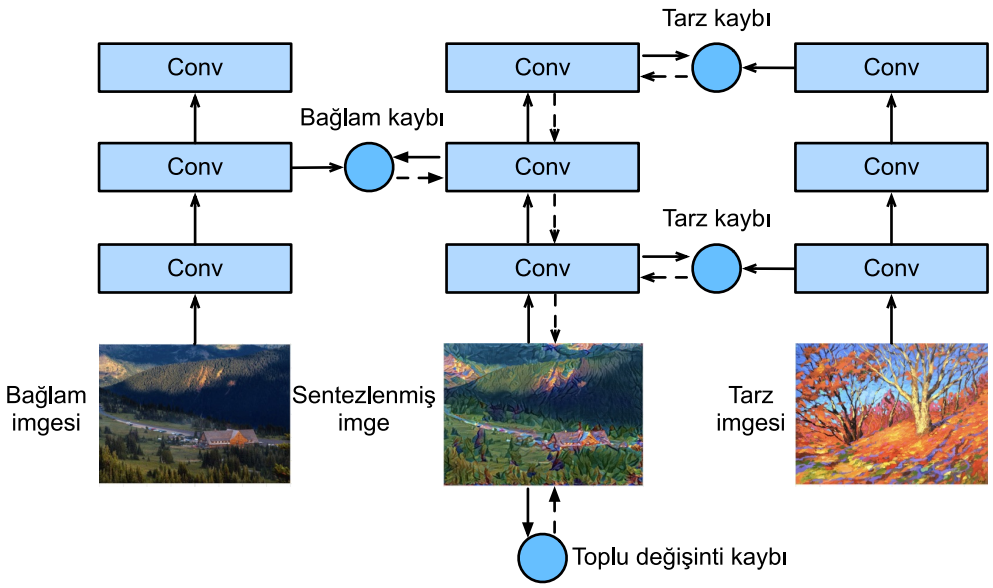


Fig. 13.12.2: CNN tabanlı stil aktarım süreci. Düz çizgiler ileri yayma yönünü ve noktalı çizgiler geriye yaymayı gösterir.

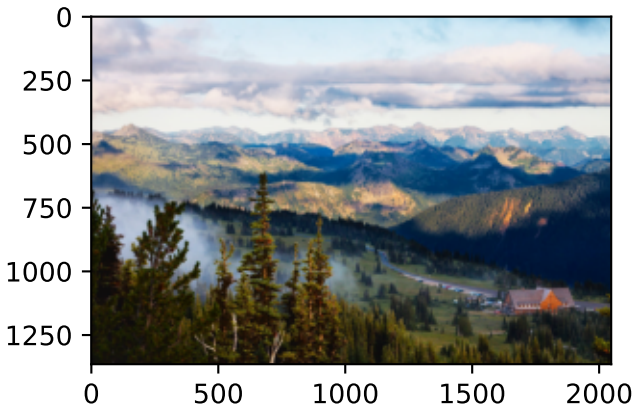
Daha sonra, ileri yayma yoluyla stil aktarımının kayıp işlevini hesaplarız (katı okların yönü) ve model parametrelerini (çıktı için sentezlenmiş imge) geri yayma (kesikli okların yönü) ile güncelleriz. Stil aktarımında yaygın olarak kullanılan kayıp fonksiyonu üç bölümden oluşur: (i) *içerik kaybı* sentezlenen imgeyi ve içerik imgesini içerik özniteliklerinde yakınlaştırır; (ii) *stil kaybı* sentezlenen imge ve stil imgesini stil özniteliklerinde yakınlaştırır; ve (iii) *toplam değişim kaybı* sentezlenen imgede gürültü azaltmaya yardım eder. Son olarak, model eğitimi bittiğinde, son sentezlenmiş imgeyi oluşturmak için stil aktarımının model parametrelerini çıktı olarak veririz.

Aşağıda, somut bir deney yoluyla stil aktarımının teknik detaylarını açıklayacağız.

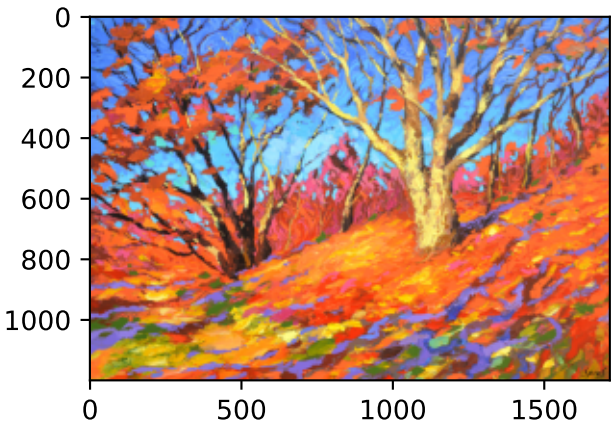
13.12.2 İçerik ve Stil İmgelerini Okuma

İlk olarak, içerik ve stil imgelerini okuyoruz. Basılı koordinat eksenlerinden, bu imgelerin farklı boyutlarda olduğunu söyleyebiliriz.

```
%matplotlib inline
import torch
import torchvision
from torch import nn
from d2l import torch as d2l

d2l.set_figsize()
content_img = d2l.Image.open('../img/rainier.jpg')
d2l.plt.imshow(content_img);
```




```
style_img = d2l.Image.open('../img/autumn-oak.jpg')
d2l.plt.imshow(style_img);
```



13.12.3 Ön İşleme ve Sonradan İşleme

Aşağıda, ön işleme ve sonradan işleme imgeleri için iki işlev tanımlıyoruz. `preprocess` işlevi, girdi imgesinin üç RGB kanalının her birini standartlaştırır ve sonuçları CNN girdi biçimine dönüştürür. `postprocess` işlevi, standartlaştırmadan önce çıktı imgesinde piksel değerlerini orijinal değerlerine geri yükler. İmge yazdırma işlevi, her pikselin 0'dan 1'e kadar kayan virgüllü sayı değerine sahip olmasını gerektirdiğinden, 0'dan küçük veya 1'den büyük herhangi bir değeri sırasıyla 0 veya 1 ile değiştiririz.

```
rgb_mean = torch.tensor([0.485, 0.456, 0.406])
rgb_std = torch.tensor([0.229, 0.224, 0.225])

def preprocess(img, image_shape):
    transforms = torchvision.transforms.Compose([
        torchvision.transforms.Resize(image_shape),
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize(mean=rgb_mean, std=rgb_std)])
    return transforms(img).unsqueeze(0)
```

(continues on next page)

(continued from previous page)

```
def postprocess(img):
    img = img[0].to(rgb_std.device)
    img = torch.clamp(img.permute(1, 2, 0) * rgb_std + rgb_mean, 0, 1)
    return torchvision.transforms.ToPILImage()(img.permute(2, 0, 1))
```

13.12.4 Öznitelik Ayıklama

İmge özniteliklerini (Gatys *et al.*, 2016) ayıklamak için ImageNet veri kümesinde önceden eğitilmiş VGG-19 modelini kullanıyoruz.

```
pretrained_net = torchvision.models.vgg19(pretrained=True)
```

```
/home/d2l-worker/miniconda3/envs/d2l-tr-release-0/lib/python3.9/site-packages/
↪torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is␣
↪deprecated since 0.13 and will be removed in 0.15, please use 'weights' instead.
  warnings.warn(
/home/d2l-worker/miniconda3/envs/d2l-tr-release-0/lib/python3.9/site-packages/
↪torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum␣
↪or None for 'weights' are deprecated since 0.13 and will be removed in 0.15. The␣
↪current behavior is equivalent to passing weights=VGG19_Weights.IMAGENET1K_V1. You␣
↪can also use weights=VGG19_Weights.DEFAULT to get the most up-to-date weights.
  warnings.warn(msg)
```

İmgenin içerik özniteliklerini ve stil özniteliklerini ayıklamak için, VGG ağındaki belirli katmanların çıktısını seçebiliriz. Genel olarak, girdi katmanına ne kadar yakın olursa, imgenin ayrıntılarını çıkarmak daha kolay olur ve tersi yönde de, imgenin küresel bilgilerini çıkarmak daha kolay olur. Sentezlenen imgede içerik imgesinin ayrıntılarını aşırı derecede tutmaktan kaçınmak için imgenin içerik özniteliklerinin çıktısını almak için *içerik katmanı* olarak çıktıya daha yakın bir VGG katmanı seçiyoruz. Yerel ve küresel stil özniteliklerini ayıklamak için farklı VGG katmanlarının çıktısını da seçiyoruz. Bu katmanlar *stil katmanları* olarak da adlandırılır. Section 7.2 içinde belirtildiği gibi, VGG ağı 5 evrişimli blok kullanır. Deneyde, dördüncü evrişimli bloğun son evrişimli katmanını içerik katmanı olarak ve her bir evrişimli bloğun ilk evrişimli katmanını stil katmanı olarak seçiyoruz. Bu katmanların indeksleri `pretrained_net` örneğini yazdırarak elde edilebilir.

```
style_layers, content_layers = [0, 5, 10, 19, 28], [25]
```

VGG katmanlarını kullanarak öznitelikleri ayıklarken, yalnızca girdi katmanından içerik katmanına veya çıktı katmanına en yakın stil katmanına kadar tüm bunları kullanmamız gerekir. Yalnızca öznitelik ayıklama için kullanılacak tüm VGG katmanlarını koruyan yeni bir ağ örneği, `net`, oluşturalım.

```
net = nn.Sequential(*[pretrained_net.features[i] for i in
                      range(max(content_layers + style_layers) + 1)])
```

`X` girdisi göz önüne alındığında, sadece ileri yayma `net(X)`'i çağırırsak, yalnızca son katmanın çıktısını alabiliriz. Ara katmanların çıktılarına da ihtiyacımız olduğundan, katman bazında hesaplama yapmalı, içerik ve stil katmanı çıktılarını korumalıyız.

```
def extract_features(X, content_layers, style_layers):
    contents = []
    styles = []
    for i in range(len(net)):
        X = net[i](X)
        if i in style_layers:
            styles.append(X)
        if i in content_layers:
            contents.append(X)
    return contents, styles
```

Aşağıda iki işlev tanımlanmıştır: `get_contents` işlevi içerik imgesinden içerik özniteliklerini ayıklar ve `get_styles` işlevi stil imgesinden stil özniteliklerini ayıklar. Eğitim sırasında önceden eğitilmiş VGG'nin model parametrelerini güncellemeye gerek olmadığından, eğitim başlamadan bile içerik ve stil özniteliklerini ayıklayabiliriz. Sentezlenen imge, stil aktarımı için güncellenecek bir model parametreleri kümesi olduğundan, yalnızca sentezlenen imgenin içerik ve stil özniteliklerini eğitim sırasında `extract_features` işlevini çağırarak ayıklayabiliriz.

```
def get_contents(image_shape, device):
    content_X = preprocess(content_img, image_shape).to(device)
    contents_Y, _ = extract_features(content_X, content_layers, style_layers)
    return content_X, contents_Y

def get_styles(image_shape, device):
    style_X = preprocess(style_img, image_shape).to(device)
    _, styles_Y = extract_features(style_X, content_layers, style_layers)
    return style_X, styles_Y
```

13.12.5 Kayıp Fonksiyonunu Tanımlama

Şimdi stil aktarımı için kayıp işlevini açıklayacağız. Kayıp fonksiyonu içerik kaybı, stil kaybı ve toplam değişim kaybından oluşur.

İçerik Kaybı

Doğrusal bağlanımdaki kayıp işlevine benzer şekilde, içerik kaybı, kare kayıp fonksiyonu aracılığıyla, sentezlenen imge ile içerik imgesi arasındaki içerik özniteliklerindeki farkı ölçer. Kare kayıp fonksiyonunun iki girdisi, `extract_features` fonksiyonu tarafından hesaplanan içerik katmanının çıktılarıdır.

```
def content_loss(Y_hat, Y):
    # We detach the target content from the tree used to dynamically compute
    # the gradient: this is a stated value, not a variable. Otherwise the loss
    # will throw an error.
    return torch.square(Y_hat - Y.detach()).mean()
```

Stil Kaybı

İçerik kaybına benzer şekilde stil kaybı, aynı zamanda sentezlenen imge ile stil imgesi arasındaki stil farkını ölçmek için kare kaybı işlevini kullanır. Herhangi bir stil katmanının stil çıktısını ifade etmeden önce stil katmanı çıktısını hesaplamak için `extract_features` işlevini kullanırız. Çıktının 1 örneği, c kanalları, h yüksekliği ve w genişliği olduğunu varsayalım, bu çıktıyı c satırları ve hw sütunlarıyla $\mathbf{X}$ matrisine dönüştürebiliriz. Bu matris, her birinin uzunluğu hw olan c adet $\mathbf{x}_1, \ldots, \mathbf{x}_c$ vektörlerinin birleşimi olarak düşünülebilir. Burada $\mathbf{x}_i$ vektörü, i kanalının stil özniteliğini temsil eder.

Bu vektörlerin *Gram matrisinde* $\mathbf{X}\mathbf{X}^\top \in \mathbb{R}^{c \times c}$, i satırındaki ve j sütunundaki x_{ij} öğesi $\mathbf{x}_j$ vektörlerinin nokta çarpımıdır. i ve j kanallarının stil özniteliklerinin korelasyonunu temsil eder. Bu Gram matrisini herhangi bir stil katmanının stil çıktısını temsil etmek için kullanırız. hw değeri daha büyük olduğunda, büyük olasılıkla Gram matrisinde daha büyük değerlere yol açtığını unutmayın. Gram matrisinin yüksekliği ve genişliğinin ikisinin de kanal sayısının c olduğunu da unutmayın. Stil kaybının bu değerlerden etkilenmemesine izin vermek için, aşağıdaki `gram` işlevi Gram matrisini elemanlarının sayısına (yani chw) böler.

```
def gram(X):
    num_channels, n = X.shape[1], X.numel() // X.shape[1]
    X = X.reshape((num_channels, n))
    return torch.matmul(X, X.T) / (num_channels * n)
```

Açıkçası, stil kaybı için kare kayıp fonksiyonunun iki Gram matris girdisi, sentezlenen imge ve stil imgesi için stil katmanı çıktılarına dayanır. Burada stil imgesine dayanan Gram matrisi `gram_Y`'nin önceden hesaplandığı varsayılmaktadır.

```
def style_loss(Y_hat, gram_Y):
    return torch.square(gram(Y_hat) - gram_Y.detach()).mean()
```

Toplam Değişim Kaybı

Bazen, öğrenilen sentezlenen imge çok yüksek frekanslı gürültüye, yani özellikle parlak veya koyu piksellere sahiptir. Bir yaygın gürültü azaltma yöntemi *toplam değişim gürültü arındırma*dır. (i, j) koordinatında piksel değerini $x_{i,j}$ ile belirtin. Toplam değişim kaybının azaltma

$$\sum_{i,j} |x_{i,j} - x_{i+1,j}| + |x_{i,j} - x_{i,j+1}| \tag{13.12.1}$$

, sentezlenen imgedeki komşu piksellerin değerlerini yakınlaştırır.

```
def tv_loss(Y_hat):
    return 0.5 * (torch.abs(Y_hat[:, :, 1:, :] - Y_hat[:, :, :-1, :]).mean() +
                  torch.abs(Y_hat[:, :, :, 1:] - Y_hat[:, :, :, :-1]).mean())
```

Kayıp Fonksiyonu

Stil aktarımının kayıp fonksiyonu, içerik kaybı, stil kaybı ve toplam değişim kaybının ağırlıklı toplamıdır. Bu ağırlık hiper parametrelerini ayarlayarak sentezlenen imgede, içerik tutma, stil aktarma ve gürültü azaltma arasında denge kurabiliriz.

```
content_weight, style_weight, tv_weight = 1, 1e3, 10

def compute_loss(X, contents_Y_hat, styles_Y_hat, contents_Y, styles_Y_gram):
    # Sırasıyla içerik, stil ve toplam varyans kayıplarını hesaplayın
    contents_l = [content_loss(Y_hat, Y) * content_weight for Y_hat, Y in zip(
        contents_Y_hat, contents_Y)]
    styles_l = [style_loss(Y_hat, Y) * style_weight for Y_hat, Y in zip(
        styles_Y_hat, styles_Y_gram)]
    tv_l = tv_loss(X) * tv_weight
    # Bütün kayıpları toplayın
    l = sum(10 * styles_l + contents_l + [tv_l])
    return contents_l, styles_l, tv_l, l
```

13.12.6 Sentezlenmiş İmgeyi İlkleme

Stil transferinde, sentezlenen imge, eğitim sırasında güncellenmesi gereken tek değişkendir. Böylece, basit bir model, `SynthesizedImage` tanımlayabilir ve sentezlenen imgeyi model parametreleri olarak ele alabiliriz. Bu modelde, ileri yayma sadece model parametrelerini döndürür.

```
class SynthesizedImage(nn.Module):
    def __init__(self, img_shape, **kwargs):
        super(SynthesizedImage, self).__init__(**kwargs)
        self.weight = nn.Parameter(torch.rand(*img_shape))

    def forward(self):
        return self.weight
```

Sonrasında, `get_inits` işlevini tanımlıyoruz. Bu işlev, sentezlenmiş bir imge modeli örneği oluşturur ve `X` imgesine ilkler. Çeşitli stil katmanlarındaki stil imgesi için Gram matrisleri, `styles_Y_gram`, eğitimden önce hesaplanır.

```
def get_inits(X, device, lr, styles_Y):
    gen_img = SynthesizedImage(X.shape).to(device)
    gen_img.weight.data.copy_(X.data)
    trainer = torch.optim.Adam(gen_img.parameters(), lr=lr)
    styles_Y_gram = [gram(Y) for Y in styles_Y]
    return gen_img(), styles_Y_gram, trainer
```

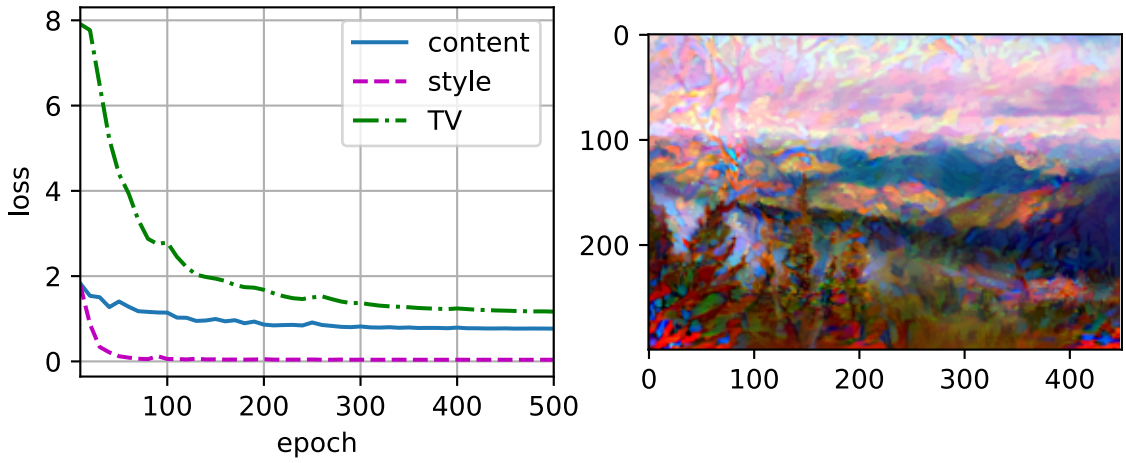
13.12.7 Eğitim

Modeli stil aktarımı için eğitirken, sentezlenen imgenin içerik özniteliklerini ve stil özniteliklerini sürekli olarak ayıklarız ve kayıp işlevini hesaplarız. Aşağıda eğitim döngüsünü tanımlıyoruz.

```
def train(X, contents_Y, styles_Y, device, lr, num_epochs, lr_decay_epoch):
    X, styles_Y_gram, trainer = get_inits(X, device, lr, styles_Y)
    scheduler = torch.optim.lr_scheduler.StepLR(trainer, lr_decay_epoch, 0.8)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[10, num_epochs],
                            legend=['content', 'style', 'TV'],
                            ncols=2, figsize=(7, 2.5))
    for epoch in range(num_epochs):
        trainer.zero_grad()
        contents_Y_hat, styles_Y_hat = extract_features(
            X, content_layers, style_layers)
        contents_l, styles_l, tv_l, l = compute_loss(
            X, contents_Y_hat, styles_Y_hat, contents_Y, styles_Y_gram)
        l.backward()
        trainer.step()
        scheduler.step()
        if (epoch + 1) % 10 == 0:
            animator.axes[1].imshow(postprocess(X))
            animator.add(epoch + 1, [float(sum(contents_l)),
                                     float(sum(styles_l)), float(tv_l)])
    return X
```

Şimdi modeli eğitmeye başlıyoruz. İçerik ve stil imgelerinin yüksekliğini ve genişliğini 300 x 450 piksele yeniden ölçeklendiriyoruz. Sentezlenen imgeyi ilklemek için içerik imgesini kullanırız.

```
device, image_shape = d2l.try_gpu(), (300, 450)  # PIL Image (h, w)
net = net.to(device)
content_X, contents_Y = get_contents(image_shape, device)
_, styles_Y = get_styles(image_shape, device)
output = train(content_X, contents_Y, styles_Y, device, 0.3, 500, 50)
```



Sentezlenen imgenin içerik imgesinin manzarasını ve nesnelerini koruduğunu ve aynı zamanda stil imgesinin rengini aktardığını görebiliriz. Örneğin, sentezlenen imgenin stil imgesinde olduğu gibi renk blokları vardır. Bu blokların bazıları fırça darbelerinin ince dokusuna bile sahiptir.

13.12.8 Özet

- Stil aktarımında yaygın olarak kullanılan kayıp fonksiyonu üç bölümden oluşur: (i) İçerik kaybı, sentezlenen imgeyi ve içerik imgesini içerik özniteliklerinde yakınlaştırır; (ii) stil kaybı, sentezlenen imge ve stil imgesini stil özniteliklerinde yakınlaştırır; ve (iii) toplam değişim kaybı sentezlenmiş imgedeki gürültüyü azaltmayı sağlar.
- Eğitim sırasında sentezlenen imgeyi sürekli olarak model parametreleri olarak güncellemek için imge özniteliklerini ayıklamak ve kayıp işlevini en aza indirmek için önceden eğitilmiş bir CNN kullanabiliriz.
- Stil katmanlarından stil çıktılarını temsil etmek için Gram matrisleri kullanırız.

13.12.9 Alıştırmalar

1. Farklı içerik ve stil katmanları seçtiğinizde çıktı nasıl değişir?
2. Kayıp işlevindeki ağırlık hiper parametrelerini ayarlayın. Çıktıda daha fazla içerik mi yoksa daha az gürültü mü var?
3. Farklı içerik ve stil imgeleri kullanın. Sentezlenmiş daha ilginç imgeler oluşturabilir misiniz?
4. Metin için stil aktarımı uygulayabilir miyiz? İpucu: Hu ve arkadaşlarının araştırma makalesine başvurabilirsiniz (Hu *et al.*, 2020).

Tartışmalar[186]

13.13 Kaggle'da İmge Sınıflandırması (CIFAR-10)

Şimdiye kadar, doğrudan tensör formatında imge veri kümelerini elde etmek için derin öğrenme çerçevelerinin üst düzey API'lerini kullanıyoruz. Ancak, özel imge veri kümeleri genellikle imge dosyaları halinde gelir. Bu bölümde, ham imge dosyalarından başlayacağız ve düzenleyeceğiz, okuyacağız, ardından bunları adım adım tensör formatına dönüştüreceğiz.

Bilgisayarla görmede önemli bir veri kümesi olan Section 13.1 içinde CIFAR-10 veri kümesi ile deney yaptık. Bu bölümde, CIFAR-10 imge sınıflandırmasının Kaggle yarışmasını uygulamak için önceki bölümlerde öğrendiğimiz bilgileri uygulayacağız. Yarışmanın web adresi https://www.kaggle.com/c/cifar-10

Fig. 13.13.1 yarışmanın web sayfasındaki bilgileri gösterir. Sonuçları göndermek için bir Kaggle hesabına kayıt olmanız gerekir.

[186] https://discuss.d2l.ai/t/1476

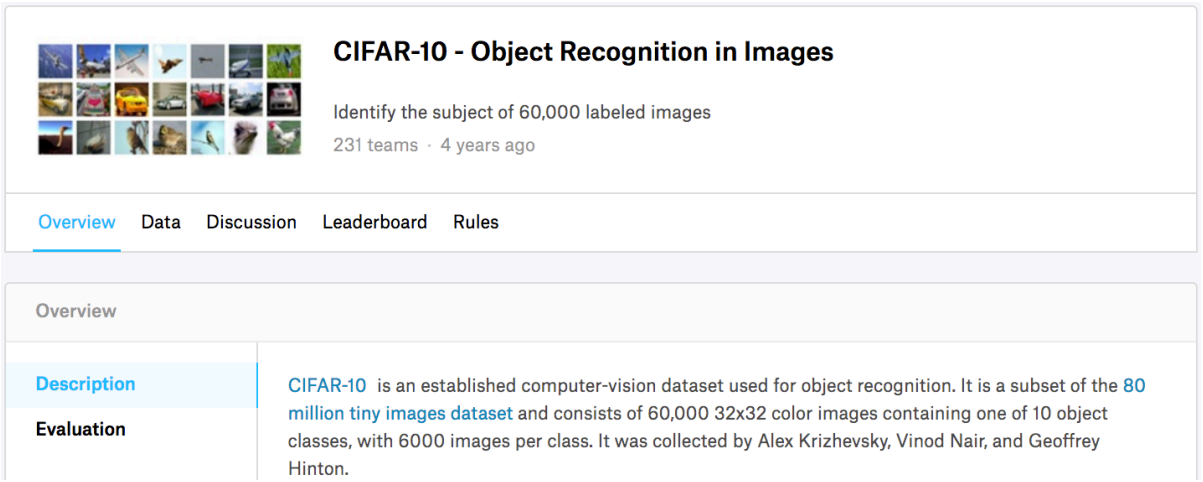


Fig. 13.13.1: CIFAR-10 imge sınıflandırma yarışması web sayfası bilgileri. Yarışma veri kümesi "Data" ("Veri") sekmesine tıklanarak elde edilebilir.

```
import collections
import math
import os
import shutil
import pandas as pd
import torch
import torchvision
from torch import nn
from d2l import torch as d2l
```

13.13.1 Veri Kümesini Elde Etme ve Düzenleme

Yarışma veri kümesi, sırasıyla 50000 ve 300000 imge içeren bir eğitim kümesi ve bir test kümesine ayrılmıştır. Test kümesinde, değerlendirme için 10000 imge kullanılacak, kalan 290000 imgeler değerlendirilmeyecek: Bunlar sadece test kümesinin *manuel* etiketli sonuçlarıyla hile yapmayı zorlaştırmak için dahil edilmiştir. Bu veri kümesindeki imgeler, yüksekliği ve genişliği 32 piksel olan png renkli (RGB kanalları) imge dosyalarıdır. İmgeler, uçaklar, arabalar, kuşlar, kediler, geyik, köpekler, kurbağalar, atlar, tekneler ve kamyonlar olmak üzere toplam 10 kategoriyi kapsar. Fig. 13.13.1 şeklinin sol üst köşesi veri kümesindeki uçakların, arabaların ve kuşların bazı imgelerini gösterir.

Veri Kümesini İndirme

Kaggle'a girdi yaptıktan sonra Fig. 13.13.1 içinde gösterilen CIFAR-10 imge sınıflandırma yarışması web sayfasındaki "Veri" ("Data") sekmesine tıklayabilir ve "Tümünü İndir" ("Download All") butonuna tıklayarak veri kümesini indirebiliriz. İndirilen dosyayı `../data`'da açtıktan ve içinde `train.7z` ve `test.7z`'yı açtıktan sonra, tüm veri kümesini aşağıdaki yollarda bulacaksınız:

- `../data/cifar-10/train/[1-50000].png`
- `../data/cifar-10/test/[1-300000].png`
- `../data/cifar-10/trainLabels.csv`

- `../data/cifar-10/sampleSubmission.csv`

`train` ve `test` dizinlerinin sırasıyla eğitim ve test imgelerini içerdiği, `trainLabels.csv` eğitim imgeleri için etiketler sağlar ve `sample_submission.csv` örnek bir gönderim dosyasıdır.

Başlamayı kolaylaştırmak için ilk 1000 eğitim imgesi ve 5 rastgele test imgesi içeren veri kümesinin küçük ölçekli bir örneğini sağlıyoruz. Kaggle yarışmasının tam veri kümesini kullanmak için aşağıdaki `demo` değişkenini `False` olarak ayarlamanız gerekir.

```
#@save
d2l.DATA_HUB['cifar10_tiny'] = (d2l.DATA_URL + 'kaggle_cifar10_tiny.zip',
                                '2068874e4b9a9f0fb07ebe0ad2b29754449ccacd')

# If you use the full dataset downloaded for the Kaggle competition, set
# `demo` to False
demo = True

if demo:
    data_dir = d2l.download_extract('cifar10_tiny')
else:
    data_dir = '../data/cifar-10/'
```

Veri Kümesini Düzenleme

Model eğitimini ve testlerini kolaylaştırmak için veri kümeleri düzenlememiz gerekiyor. Önce csv dosyasındaki etiketleri okuyalım. Aşağıdaki işlev, dosya adının uzantısız kısmını etiketine eşleyen bir sözlük döndürür.

```
#@save
def read_csv_labels(fname):
    """Read `fname` to return a filename to label dictionary."""
    with open(fname, 'r') as f:
        # Skip the file header line (column name)
        lines = f.readlines()[1:]
    tokens = [l.rstrip().split(',') for l in lines]
    return dict(((name, label) for name, label in tokens))

labels = read_csv_labels(os.path.join(data_dir, 'trainLabels.csv'))
print('# training examples:', len(labels))
print('# classes:', len(set(labels.values())))
```

```
# training examples: 1000
# classes: 10
```

Ardından, `reorg_train_valid` işlevini esas eğitim kümesinden geçerleme kümesini bölmek için tanımlıyoruz. Bu işlevdeki `valid_ratio` argümanı, geçerleme kümesindeki örneklerin sayısının orijinal eğitim kümesindeki örneklerin sayısına oranıdır. Daha somut olarak, sınıfın en az örnek içeren imge sayısı n ve oranı da r olsun. Geçerleme kümesi her sınıf için $\max(\lfloor nr \rfloor, 1)$ imge ayırır. Örnek olarak `valid_ratio=0.1`'i kullanalım. Orijinal eğitim kümesi 50000 imgeye sahip olduğundan, `train_valid_test/train` yolunda eğitim için kullanılan 45000 imge olacak, diğer 5000 imge `train_valid_test/valid` yolunda geçerleme kümesi olarak bölünecek. Veri kümesini düzenledikten sonra, aynı sınıfın imgeleri aynı klasörün altına yerleştirilir.

```
#@save
def copyfile(filename, target_dir):
    """Bir dosyayı hedef dizine kopyalayın."""
    os.makedirs(target_dir, exist_ok=True)
    shutil.copy(filename, target_dir)

#@save
def reorg_train_valid(data_dir, labels, valid_ratio):
    """Doğrulama kümesini orijinal eğitim kümesinden ayırın."""
    # Eğitim veri kümesinde en az örneğe sahip sınıfın örnek sayısı
    n = collections.Counter(labels.values()).most_common()[-1][1]
    # Geçerleme kümesi için sınıf başına örnek sayısı
    n_valid_per_label = max(1, math.floor(n * valid_ratio))
    label_count = {}
    for train_file in os.listdir(os.path.join(data_dir, 'train')):
        label = labels[train_file.split('.')[0]]
        fname = os.path.join(data_dir, 'train', train_file)
        copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                     'train_valid', label))
        if label not in label_count or label_count[label] < n_valid_per_label:
            copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                         'valid', label))
            label_count[label] = label_count.get(label, 0) + 1
        else:
            copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                         'train', label))
    return n_valid_per_label
```

Aşağıdaki `reorg_test` işlevi tahmin sırasında veri yükleme için test kümesini düzenler.

```
#@save
def reorg_test(data_dir):
    """Tahmin sırasında veri yüklemesi için test kümesini düzenleyin."""
    for test_file in os.listdir(os.path.join(data_dir, 'test')):
        copyfile(os.path.join(data_dir, 'test', test_file),
                 os.path.join(data_dir, 'train_valid_test', 'test',
                              'unknown'))
```

Son olarak, `read_csv_labels`, `reorg_train_valid` ve `reorg_test` yukarıda tanımlanan işlevleri çağırmak için bir işlev kullanıyoruz.

```
def reorg_cifar10_data(data_dir, valid_ratio):
    labels = read_csv_labels(os.path.join(data_dir, 'trainLabels.csv'))
    reorg_train_valid(data_dir, labels, valid_ratio)
    reorg_test(data_dir)
```

Burada, veri kümesinin küçük ölçekli örneği için toplu iş boyutunu yalnızca 32 olarak ayarlıyoruz. Kaggle yarışmasının tüm veri kümesini eğitip test ederken, `batch_size` 128 gibi daha büyük bir tamsayıya ayarlanmalıdır. Eğitim örneklerinin %10'unu hiper parametrelerin ayarlanması için geçerleme kümesi olarak ayırdık.

```
batch_size = 32 if demo else 128
valid_ratio = 0.1
reorg_cifar10_data(data_dir, valid_ratio)
```

13.13.2 İmge Artırma

Aşırı öğrenmeyi bertaraf etmek için imge artırımı kullanıyoruz. Örneğin, imgeler eğitim sırasında rastgele yatay olarak çevrilebilir. Renkli imgelerin üç RGB kanalı için standartlaştırma da gerçekleştirebiliriz. Aşağıda ayarlayabileceğiniz bu işlemlerin bazıları listelenmektedir.

```
transform_train = torchvision.transforms.Compose([
    # İmgeyi hem yükseklik hem de genişlikte 40 piksellik bir kareye ölçeklendirin
    torchvision.transforms.Resize(40),
    # Orijinal imgenin alanının 0.64 ila 1 katı arasında küçük bir kare
    # oluşturmak için hem yükseklik hem de genişlikte 40 piksellik bir kare
    # imgeyi rastgele kırpın ve ardından hem yükseklik hem de genişlikte
    # 32 piksellik bir kareye ölçeklendirin
    torchvision.transforms.RandomResizedCrop(32, scale=(0.64, 1.0),
                                             ratio=(1.0, 1.0)),
    torchvision.transforms.RandomHorizontalFlip(),
    torchvision.transforms.ToTensor(),
    # İmgenin her kanalını standartlaştırın
    torchvision.transforms.Normalize([0.4914, 0.4822, 0.4465],
                                     [0.2023, 0.1994, 0.2010])])
```

Test sırasında, değerlendirme sonuçlarındaki rastgeleliği ortadan kaldırmak için yalnızca imgeler üzerinde standartlaştırma gerçekleştiriyoruz.

```
transform_test = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize([0.4914, 0.4822, 0.4465],
                                     [0.2023, 0.1994, 0.2010])])
```

13.13.3 Veri Kümesini Okuma

Ardından, ham imge dosyalarından oluşan düzenlenmiş veri kümesini okuruz. Her örnek bir imge ve bir etiket içerir.

```
train_ds, train_valid_ds = [torchvision.datasets.ImageFolder(
    os.path.join(data_dir, 'train_valid_test', folder),
    transform=transform_train) for folder in ['train', 'train_valid']]

valid_ds, test_ds = [torchvision.datasets.ImageFolder(
    os.path.join(data_dir, 'train_valid_test', folder),
    transform=transform_test) for folder in ['valid', 'test']]
```

Eğitim sırasında yukarıda tanımlanan tüm imge artırım işlemlerini belirtmemiz gerekir. Geçerleme kümesi hiper parametre ayarlama sırasında model değerlendirmesi için kullanıldığında, imge artırımdan rastgelelik getirilmemelidir. Son tahminden önce, tüm etiketlenmiş verileri tam olarak kullanmak için modeli birleştirilmiş eğitim kümesi ve geçerleme kümesi üzerinde eğitiriz.

```
train_iter, train_valid_iter = [torch.utils.data.DataLoader(
    dataset, batch_size, shuffle=True, drop_last=True)
    for dataset in (train_ds, train_valid_ds)]

valid_iter = torch.utils.data.DataLoader(valid_ds, batch_size, shuffle=False,
```

(continues on next page)

(continued from previous page)

```
                                        drop_last=True)

test_iter = torch.utils.data.DataLoader(test_ds, batch_size, shuffle=False,
                                        drop_last=False)
```

13.13.4 Modeli Tanımlama

Section 7.6'te açıklanan ResNet-18 modelini tanımlıyoruz.

```
def get_net():
    num_classes = 10
    net = d2l.resnet18(num_classes, 3)
    return net

loss = nn.CrossEntropyLoss(reduction="none")
```

13.13.5 Eğitim Fonksiyonunu Tanımlama

Modelleri seçeceğiz ve hiper parametreleri geçerleme kümesindeki modelin performansına göre ayarlayacağız. Aşağıda, model eğitim fonksiyonunu, `train`, tanımlıyoruz.

```
def train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
          lr_decay):
    trainer = torch.optim.SGD(net.parameters(), lr=lr, momentum=0.9,
                              weight_decay=wd)
    scheduler = torch.optim.lr_scheduler.StepLR(trainer, lr_period, lr_decay)
    num_batches, timer = len(train_iter), d2l.Timer()
    legend = ['train loss', 'train acc']
    if valid_iter is not None:
        legend.append('valid acc')
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                            legend=legend)
    net = nn.DataParallel(net, device_ids=devices).to(devices[0])
    for epoch in range(num_epochs):
        net.train()
        metric = d2l.Accumulator(3)
        for i, (features, labels) in enumerate(train_iter):
            timer.start()
            l, acc = d2l.train_batch_ch13(net, features, labels,
                                          loss, trainer, devices)
            metric.add(l, acc, labels.shape[0])
            timer.stop()
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (metric[0] / metric[2], metric[1] / metric[2],
                              None))
        if valid_iter is not None:
            valid_acc = d2l.evaluate_accuracy_gpu(net, valid_iter)
            animator.add(epoch + 1, (None, None, valid_acc))
        scheduler.step()
```

(continues on next page)

(continued from previous page)

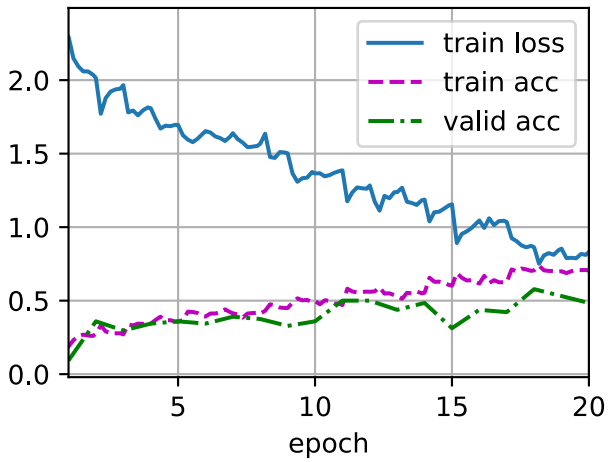
```
    measures = (f'train loss {metric[0] / metric[2]:.3f}, '
                f'train acc {metric[1] / metric[2]:.3f}')
    if valid_iter is not None:
        measures += f', valid acc {valid_acc:.3f}'
    print(measures + f'\n{metric[2] * num_epochs / timer.sum():.1f}'
          f' examples/sec on {str(devices)}')
```

13.13.6 Modeli Eğitme ve Geçerleme

Şimdi, modeli eğitebilir ve geçerleyebiliriz. Aşağıdaki tüm hiper parametreler ayarlanabilir. Örneğin, dönem sayısını artırabiliriz. `lr_period` ve `lr_decay` sırasıyla 4 ve 0.9 olarak ayarlandığında, optimizasyon algoritmasının öğrenme oranı her 4 dönem sonrasında 0.9 ile çarpılır. Sadece gösterim kolaylığı için, burada sadece 20 dönemlik eğitim yapıyoruz.

```
devices, num_epochs, lr, wd = d2l.try_all_gpus(), 20, 2e-4, 5e-4
lr_period, lr_decay, net = 4, 0.9, get_net()
train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
      lr_decay)
```

```
train loss 0.832, train acc 0.704, valid acc 0.484
1049.5 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



13.13.7 Test Kümesini Sınıflandırma ve Kaggle'da Sonuçları Teslim Etme

Hiper parametrelerle umut verici bir model elde ettikten sonra, modeli yeniden eğitmek ve test kümesini sınıflandırmak için tüm etiketli verileri (geçerleme kümesi dahil) kullanırız.

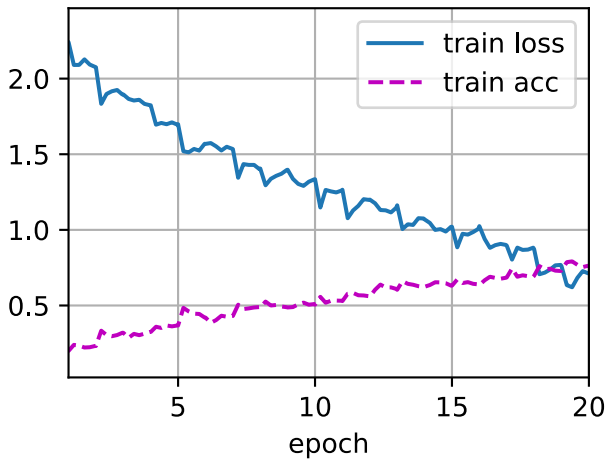
```
net, preds = get_net(), []
train(net, train_valid_iter, None, num_epochs, lr, wd, devices, lr_period,
      lr_decay)

for X, _ in test_iter:
```

(continues on next page)

(continued from previous page)

```
    y_hat = net(X.to(devices[0]))
    preds.extend(y_hat.argmax(dim=1).type(torch.int32).cpu().numpy())
sorted_ids = list(range(1, len(test_ds) + 1))
sorted_ids.sort(key=lambda x: str(x))
df = pd.DataFrame({'id': sorted_ids, 'label': preds})
df['label'] = df['label'].apply(lambda x: train_valid_ds.classes[x])
df.to_csv('submission.csv', index=False)
```

```
train loss 0.717, train acc 0.759
1209.9 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



Yukarıdaki kod, biçimi Kaggle yarışmasının gereksinimini karşılayan bir `submission.csv` dosyası oluşturacaktır. Sonuçları Kaggle'a gönderme yöntemi, Section 4.10 içindeki yönteme benzerdir.

13.13.8 Özet

- Gerekli formata düzenledikten sonra ham imge dosyalarını içeren veri kümelerini okuyabiliriz.
- Bir imge sınıflandırma yarışmasında evrişimli sinir ağları ve imge artırmayı kullanabiliriz.

13.13.9 Alıştırmalar

1. Bu Kaggle yarışması için CIFAR-10 veri kümesinin tamamını kullanın. Hiper parametreleri `batch_size = 128`, `num_epochs = 100`, `lr = 0.1`, `lr_period = 50` ve `lr_decay = 0.1` olarak ayarlayın. Bu yarışmada hangi doğruluk ve sıralamayı elde edebileceğinizi görün. Onları daha da geliştirebilir misin?
2. İmge artırmayı kullanmadığınızda hangi doğruluğu elde edebilirsiniz?

Tartışmalar[187]

[187] https://discuss.d2l.ai/t/1479

13.14 Kaggle Üzerinde Köpek Cinsi Tanımlama (ImageNet Köpekler)

Bu bölümde, Kaggle'da köpek cinsi tanımlama problemini uygulayacağız. Bu yarışmanın web adresi: https://www.kaggle.com/c/dog-breed-identification

Bu yarışmada 120 farklı köpek ırkı tanınacak. Aslında, bu yarışma için veri kümesi ImageNet veri kümesinin bir alt kümesidir. Section 13.13 içindeki CIFAR-10 veri kümesindeki imgelerin aksine, ImageNet veri kümesindeki imgeler hem daha yüksek hem de daha geniş farklı boyutlardadır. Fig. 13.14.1, yarışmanın web sayfasındaki bilgileri gösterir. Sonuçlarınızı göndermek için bir Kaggle hesabına ihtiyacınız var.

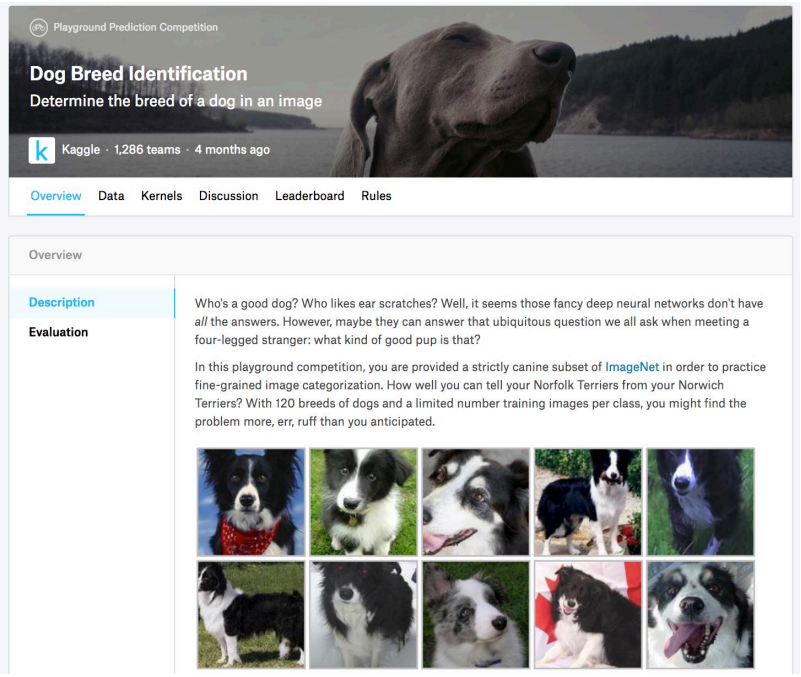


Fig. 13.14.1: Köpek cinsi tanımlama yarışması web sitesi. Yarışma veri kümesi "Data" ("Veri") sekmesine tıklanarak elde edilebilir.

```
import os
import torch
import torchvision
from torch import nn
from d2l import torch as d2l
```

13.14.1 Veri Kümesini Elde Etme ve Düzenleme

Yarışma veri kümesi, sırasıyla üç RGB (renkli) kanalın 10222 ve 10357 JPEG, imgelerini içeren birer eğitim kümesine ve test kümesine ayrılmıştır. Eğitim veri kümesinde Labradors, Kaniş, Dachshunds, Samoyeds, Huskies, Chihuahuas ve Yorkshire Terriers gibi 120 köpek ırkı vardır.

Veri Kümesini İndirme

Kaggle'a giriş yaptıktan sonra, Fig. 13.14.1 içinde gösterilen yarışma web sayfasındaki "Veri" ("Data") sekmesine tıklayabilir ve "Tümünü İndir" ("Download All") düğmesine tıklayarak veri kümesini indirebilirsiniz. İndirilen dosyayı `../data`'da açtıktan sonra, tüm veri kümesini aşağıdaki yollarda bulacaksınız:

- ../data/dog-breed-identification/labels.csv
- ../data/dog-breed-identification/sample_submission.csv
- ../data/dog-breed-identification/train
- ../data/dog-breed-identification/test

Yukarıdaki yapının `train/` ve `test/` klasörlerinin sırasıyla eğitim ve test köpek imgeleri içeren Section 13.13 içindeki CIFAR-10 yarışmasına benzer olduğunu fark etmiş olabilirsiniz ve `labels.csv` eğitim imgeleri için etiketler içerir. Benzer şekilde, başlamayı kolaylaştırmak için, yukarıda belirtilen: `train_valid_test_tiny.zip` veri kümesinin küçük bir örneklemini sağlıyoruz. Kaggle yarışması için tam veri kümesini kullanacaksanız, aşağıdaki `demo` değişkenini `False` olarak değiştirmeniz gerekir.

```
#@save
d2l.DATA_HUB['dog_tiny'] = (d2l.DATA_URL + 'kaggle_dog_tiny.zip',
                            '0cb91d09b814ecdc07b50f31f8dcad3e81d6a86d')

# Kaggle yarışması için indirilen tam veri kümesini kullanıyorsanız,
# aşağıdaki değişkeni `False` olarak değiştirin
demo = True
if demo:
    data_dir = d2l.download_extract('dog_tiny')
else:
    data_dir = os.path.join('..', 'data', 'dog-breed-identification')
```

Veri Kümesini Düzenleme

Veri kümesini Section 13.13 içinde yaptığımız şeye benzer şekilde düzenleyebiliriz, yani esas eğitim kümesindeki bir geçerleme kümesini ayırabilir ve imgeleri etiketlere göre gruplandırılmış alt klasörlere taşıyabiliriz.

Aşağıdaki `reorg_dog_data` işlevi eğitim veri etiketlerini okur, geçerleme kümesini böler ve eğitim kümesini düzenler.

```
def reorg_dog_data(data_dir, valid_ratio):
    labels = d2l.read_csv_labels(os.path.join(data_dir, 'labels.csv'))
    d2l.reorg_train_valid(data_dir, labels, valid_ratio)
    d2l.reorg_test(data_dir)


batch_size = 32 if demo else 128
valid_ratio = 0.1
reorg_dog_data(data_dir, valid_ratio)
```

13.14.2 İmge Artırma

Bu köpek cins veri kümesinin, imgeleri Section 13.13 içindeki CIFAR-10 veri kümesinden daha büyük olan ImageNet veri kümesinin bir alt kümesi olduğunu hatırlayın. Aşağıda, nispeten daha büyük imgeler için yararlı olabilecek birkaç imge artırma işlemi listelenmektedir.

```
transform_train = torchvision.transforms.Compose([
    # Orijinal alanın 0.08 ila 1'i arasında bir alana ve 3/4 ile 4/3 arasında
    # yükseklik-genişlik oranına sahip bir imge elde etmek için imgeyi
    # rastgele kırpın. Ardından, yeni bir 224 x 224 imge oluşturmak için
    # imgeyi ölçeklendirin.
    torchvision.transforms.RandomResizedCrop(224, scale=(0.08, 1.0),
                                             ratio=(3.0/4.0, 4.0/3.0)),
    torchvision.transforms.RandomHorizontalFlip(),
    # Parlaklığı, zıtlığı ve doygunluğu rastgele değiştirin
    torchvision.transforms.ColorJitter(brightness=0.4,
                                       contrast=0.4,
                                       saturation=0.4),
    # Rastgele gürültü ekle
    torchvision.transforms.ToTensor(),
    # İmgenin her kanalını standartlaştırın
    torchvision.transforms.Normalize([0.485, 0.456, 0.406],
                                     [0.229, 0.224, 0.225])])
```

Tahmin sırasında yalnızca imge ön işleme işlemlerini rastgelelik olmadan kullanıyoruz.

```
transform_test = torchvision.transforms.Compose([
    torchvision.transforms.Resize(256),
    # İmgenin ortasından 224 x 224 karelik bir alanı kırpın
    torchvision.transforms.CenterCrop(224),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize([0.485, 0.456, 0.406],
                                     [0.229, 0.224, 0.225])])
```

13.14.3 Veri Kümesi Okuma

Section 13.13 içinde olduğu gibi, ham imge dosyalarından oluşan düzenlenmiş veri kümesini okuyabiliriz.

```
train_ds, train_valid_ds = [torchvision.datasets.ImageFolder(
    os.path.join(data_dir, 'train_valid_test', folder),
    transform=transform_train) for folder in ['train', 'train_valid']]

valid_ds, test_ds = [torchvision.datasets.ImageFolder(
    os.path.join(data_dir, 'train_valid_test', folder),
    transform=transform_test) for folder in ['valid', 'test']]
```

Aşağıda, Section 13.13 içinde olduğu gibi veri yineleyici örneklerini oluşturuyoruz.

```
train_iter, train_valid_iter = [torch.utils.data.DataLoader(
    dataset, batch_size, shuffle=True, drop_last=True)
    for dataset in (train_ds, train_valid_ds)]
```

(continues on next page)

(continued from previous page)

```
valid_iter = torch.utils.data.DataLoader(valid_ds, batch_size, shuffle=False,
                                         drop_last=True)

test_iter = torch.utils.data.DataLoader(test_ds, batch_size, shuffle=False,
                                        drop_last=False)
```

13.14.4 Önceden Eğitilmiş Modelleri İnce Ayarlama

Yine, bu yarışma için veri kümesi ImageNet veri kümesinin bir alt kümedir. Bu nedenle, tam ImageNet veri kümesinde önceden eğitilmiş bir model seçmek için Section 13.2 içinde tartışılan yaklaşımı kullanabilir ve bunu özel bir küçük ölçekli çıktı ağına beslenecek imge özniteliklerini ayıklamak için kullanabiliriz. Derin öğrenme çerçevelerinin üst seviye API'leri, ImageNet veri kümesi üzerinde önceden eğitilmiş geniş bir model yelpazesi sunar. Burada, bu modelin çıktı katmanının girdisini (yani ayıklanan öznitelikler) yeniden kullandığımız önceden eğitilmiş bir ResNet-34 modeli seçiyoruz. Daha sonra orijinal çıktı katmanını, iki tam bağlı katman yığını gibi eğitilebilecek küçük bir özel çıktı ağı ile değiştirebiliriz. Section 13.2 içindeki deneyden farklı olarak, aşağıdaki öznitelik ayıklamak için kullanılan önceden eğitilmiş modeli yeniden eğitmez. Bu, gradyanların depolanması için eğitim süresini ve hafızasını azaltır.

Tüm ImageNet veri kümesi için üç RGB kanalının araçlarını ve standart sapmalarını kullanarak imgeleri standartlaştırdığımızı hatırlayın. Aslında, bu aynı zamanda ImageNet'te önceden eğitilmiş model tarafından uygulanan standartlaştırma işlemi ile de tutarlıdır.

```
def get_net(devices):
    finetune_net = nn.Sequential()
    finetune_net.features = torchvision.models.resnet34(pretrained=True)
    # Yeni bir çıktı ağı tanımlayın (120 çıktı kategorisi var)
    finetune_net.output_new = nn.Sequential(nn.Linear(1000, 256),
                                            nn.ReLU(),
                                            nn.Linear(256, 120))
    # Modeli cihazlara taşı
    finetune_net = finetune_net.to(devices[0])
    # Öznitelik katmanlarının parametrelerini dondur
    for param in finetune_net.features.parameters():
        param.requires_grad = False
    return finetune_net
```

Kaybın hesaplanmasından önce, önce önceden eğitilmiş modelin çıktı katmanının girdisini, yani ayıklanan özniteliği, elde ederiz. Daha sonra bu özniteliği, kaybı hesaplamak için küçük özel çıktı ağımızın girdisi olarak kullanırız.

```
loss = nn.CrossEntropyLoss(reduction='none')

def evaluate_loss(data_iter, net, devices):
    l_sum, n = 0.0, 0
    for features, labels in data_iter:
        features, labels = features.to(devices[0]), labels.to(devices[0])
        outputs = net(features)
        l = loss(outputs, labels)
        l_sum += l.sum()
```

(continues on next page)

(continued from previous page)

```
        n += labels.numel()
    return l_sum / n
```

13.14.5 Eğitim Fonksiyonunu Tanımlama

Modeli seçip, modelin geçerleme kümesindeki performansına göre hiper parametreleri ayarlayacağız. `train` model eğitim işlevi yalnızca küçük özel çıktı ağının parametrelerini yineler.

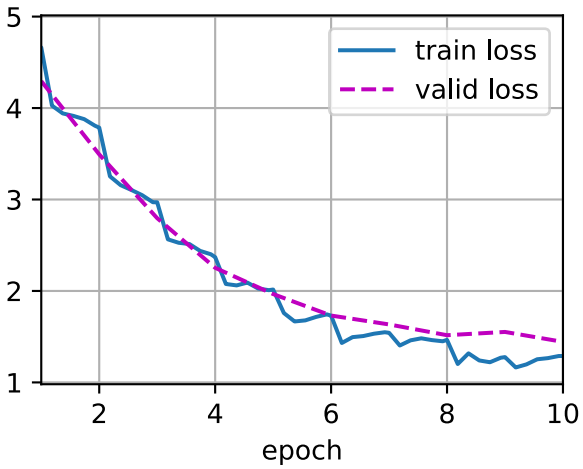
```
def train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
          lr_decay):
    # Yalnızca küçük özel çıktı ağını eğitin
    net = nn.DataParallel(net, device_ids=devices).to(devices[0])
    trainer = torch.optim.SGD((param for param in net.parameters()
                               if param.requires_grad), lr=lr,
                              momentum=0.9, weight_decay=wd)
    scheduler = torch.optim.lr_scheduler.StepLR(trainer, lr_period, lr_decay)
    num_batches, timer = len(train_iter), d2l.Timer()
    legend = ['train loss']
    if valid_iter is not None:
        legend.append('valid loss')
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                            legend=legend)
    for epoch in range(num_epochs):
        metric = d2l.Accumulator(2)
        for i, (features, labels) in enumerate(train_iter):
            timer.start()
            features, labels = features.to(devices[0]), labels.to(devices[0])
            trainer.zero_grad()
            output = net(features)
            l = loss(output, labels).sum()
            l.backward()
            trainer.step()
            metric.add(l, labels.shape[0])
            timer.stop()
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (metric[0] / metric[1], None))
        measures = f'train loss {metric[0] / metric[1]:.3f}'
        if valid_iter is not None:
            valid_loss = evaluate_loss(valid_iter, net, devices)
            animator.add(epoch + 1, (None, valid_loss.detach().cpu()))
        scheduler.step()
    if valid_iter is not None:
        measures += f', valid loss {valid_loss:.3f}'
    print(measures + f'\n{metric[1] * num_epochs / timer.sum():.1f}'
          f' examples/sec on {str(devices)}')
```

13.14.6 Modeli Eğitme ve Geçerleme

Şimdi modeli eğitebilir ve geçerleyebiliriz. Aşağıdaki hiper parametrelerin tümü ayarlanabilir. Örneğin, dönem sayısı artırılabilir. `lr_period` ve `lr_decay` sırasıyla 2 ve 0.9 olarak ayarlandığından, eniyileme algoritmasının öğrenme oranı her 2 dönem sonrasında 0.9 ile çarpılır.

```
devices, num_epochs, lr, wd = d2l.try_all_gpus(), 10, 1e-4, 1e-4
lr_period, lr_decay, net = 2, 0.9, get_net(devices)
train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
      lr_decay)
```

```
train loss 1.289, valid loss 1.443
611.3 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```

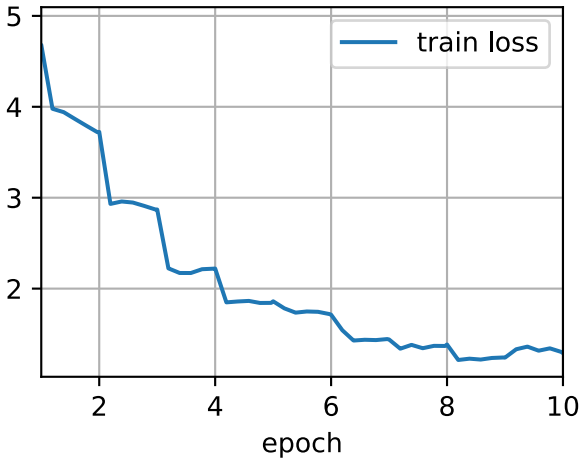


13.14.7 Test Kümesini Sınıflandırma ve Kaggle'da Sonuçları Teslim Etme

Section 13.13 içindeki son adıma benzer şekilde, sonunda tüm etiketli veriler (geçerleme kümesi dahil) modeli eğitmek ve test kümesini sınıflandırmak için kullanılır. Sınıflandırma için eğitilmiş özel çıktı ağını kullanacağız.

```
net = get_net(devices)
train(net, train_valid_iter, None, num_epochs, lr, wd, devices, lr_period,
      lr_decay)

preds = []
for data, label in test_iter:
    output = torch.nn.functional.softmax(net(data.to(devices[0])), dim=0)
    preds.extend(output.cpu().detach().numpy())
ids = sorted(os.listdir(
    os.path.join(data_dir, 'train_valid_test', 'test', 'unknown')))
with open('submission.csv', 'w') as f:
    f.write('id,' + ','.join(train_valid_ds.classes) + '\n')
    for i, output in zip(ids, preds):
        f.write(i.split('.')[0] + ',' + ','.join(
            [str(num) for num in output]) + '\n')
```

```
train loss 1.290
911.8 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



Yukarıdaki kod, Section 4.10 içinde açıklanan şekilde Kaggle'a teslim edilecek bir `submission.csv` dosyası oluşturacaktır.

13.14.8 Özet

- ImageNet veri kümelerindeki imgeler CIFAR-10 imgelerinden (farklı boyutlarda) daha büyüktür. Farklı bir veri kümesindeki görevler için imge artırma işlemlerini değiştirebiliriz.
- ImageNet veri kümesinin bir alt kümesini sınıflandırmak için, öznitelikleri ayıklamak ve yalnızca özel bir küçük ölçekli çıktı ağı eğitebilmek için ImageNet veri kümesinin tüm ImageNet veri kümesinde önceden eğitilmiş modellerini kullanabiliriz. Bu, daha az hesaplama süresi ve bellek maliyetine yol açacaktır.

13.14.9 Alıştırmalar

1. Tüm Kaggle yarışma veri kümesini kullanırken, diğer bazı hiper parametreleri `lr = 0.01`, `lr_period = 10` ve `lr_period = 10` ve `lr_decay = 0.1` olarak ayarlarken `batch_size` (toplu iş boyutu) ve `num_epochs` (dönem sayısı) artırdığınızda hangi sonuçları elde edebilirsiniz?
2. Bir önceden eğitilmiş daha derin model kullanırsanız daha iyi sonuçlar alır mısınız? Hiper parametreleri nasıl ayarlarsınız? Sonuçları daha da iyileştirebilir misiniz?

Tartışmalar[188]

[188] https://discuss.d2l.ai/t/1481

14 | Doğal Dil İşleme: Ön Eğitim

İnsanların iletişim kurması gerekir. İnsan durumunun bu temel ihtiyacı dışında, günlük olarak çok miktarda yazılı metin oluşturulmuştur. Sosyal medyadaki, sohbet uygulamalarındaki, e-postalardaki, ürün incelemelerindeki, haber makalelerindeki, araştırma kağıtlarındaki ve kitaplardaki zengin metinler göz önüne alındığında, yardım sunmak veya insan dillerine dayalı kararlar vermek için bilgisayarların onları anlamasını sağlamak hayati hale gelir.

Doğal dil işleme doğal dilleri kullanarak bilgisayarlar ve insanlar arasındaki etkileşimleri inceler. Uygulamada, Section 8.3 içindeki dil modelleri ve Section 9.5 içindeki makine çevirisi modelleri gibi metin (insan doğal dili) verilerini işlemek ve analiz etmek için doğal dil işleme tekniklerinin kullanılması çok yaygındır.

Metni anlamak için, temsillerini öğrenerek başlayabiliriz. Büyük metin kaynaklarındaki mevcut metin dizileri yararlanarak, *öz gözetimli öğrenme*, metnin bazı gizli kısımlarını çevreleyen metnin başka bir kısmını kullanarak tahmin etmek gibi, metin temsillerini önceden eğitmek için yaygın olarak kullanılmıştır. Bu şekilde modeller, *pahalı* etiketleme çabaları olmadan *kitle* metin verilerinden gözetim yoluyla öğrenirler!

Bu bölümde göreceğimiz gibi, her kelimeyi veya alt kelimeyi bireysel bir belirteç olarak ele alırken, her belirtecin temsili, word2vec, GloVe veya alt kelime gömme modellerini büyük metin kaynakları üzerinde kullanılarak önceden eğitilebilir. Ön eğitim sonrasında, her belirteçin temsili bir vektör olabilir, ancak bağlam ne olursa olsun aynı kalır. Örneğin, "banka" vektör temsili "biraz para yatırmak için bankaya git" ve "oturmak için banka git" de aynıdır. Böylece, daha birçok yeni ön eğitim modeli, aynı belirteçin temsilini farklı bağlamlara uyarlar. Bunların arasında, dönüştürücü (transformer) kodlayıcısına dayanan çok daha derin bir öz gözetimli model olan BERT vardır. Bu bölümde, Fig. 14.1 figüründe vurgulandığı gibi, metin için bu tür temsillerin nasıl ön eğitileceğine odaklanacağız.

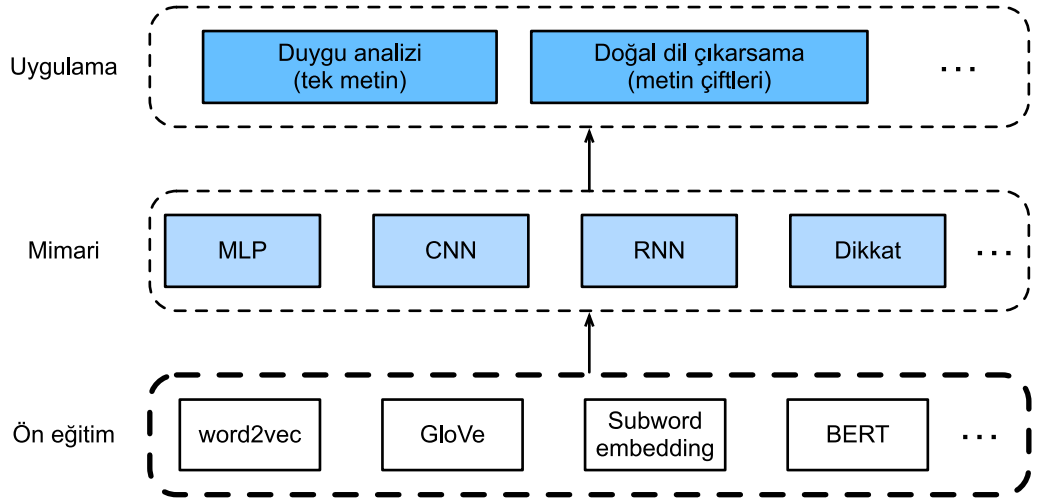


Fig. 14.1: Önceden eğitilmiş metin temsilleri, farklı akışaşağı doğal dil işleme uygulamaları için çeşitli derin öğrenme mimarilerine beslenebilir. Bu bölüm, akışyukarı metin temsili ön eğitimine odaklanmaktadır.

Büyük resmin görünmesi için, Fig. 14.1 önceden eğitilmiş metin temsillerinin farklı akışaşağı doğal dil işleme uygulamaları için çeşitli derin öğrenme mimarilerine beslenebileceğini göstermektedir. Onları Chapter 15 içinde ele alacağız.

14.1 Sözcük Gömme (word2vec)

Doğal dil anlamları ifade etmek için kullanılan karmaşık bir sistemdir. Bu sistemde, sözcükler anlamın temel birimidir. Adından da anlaşılacağı gibi, *sözcük vektörleri*, sözcükleri temsil etmek için kullanılan vektörlerdir ve ayrıca öznitelik vektörleri veya sözcüklerin temsilleri olarak da düşünülebilir. Sözcükleri gerçek vektörlere eşleme tekniği *sözcük gömme* olarak adlandırılır. Son yıllarda, sözcük gömme yavaş yavaş doğal dil işlemenin temel bilgisi haline gelmiştir.

14.1.1 Bire Bir Vektörler Kötü Bir Seçimdir

Section 8.5 içinde sözcükleri temsil etmek için bire bir vektörler kullandık (karakterler sözcüklerdir). Sözlükteki farklı sözcüklerin sayısının (sözlük boyutu) N olduğunu ve her sözcüğün 0 ila $N-1$ arasında farklı bir tamsayıya (dizin) karşılık geldiğini varsayalım. İndeksi i olan herhangi bir sözcük için bire bir vektör temsilini elde etmek için, tüm 0'lardan bir N-uzunluklu vektör oluşturuyoruz ve i konumundaki elemanı 1'e ayarlıyoruz. Bu şekilde, her sözcük N uzunluğunda bir vektör olarak temsil edilir ve doğrudan sinir ağları tarafından kullanılabilir.

Bire bir sözcük vektörlerinin oluşturulması kolay olsa da, genellikle iyi bir seçim değildir. Ana nedeni, bire bir sözcük vektörlerinin, sık sık kullandığımız *kosinüs benzerliği* gibi farklı sözcükler arasındaki benzerliği doğru bir şekilde ifade edememesidir. Vektörler $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ için kosinüs benzerliği aralarındaki açının kosinüsüdür:

$$\frac{\mathbf{x}^\top \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|} \in [-1, 1]. \tag{14.1.1}$$

İki farklı sözcüğün bire bir vektörleri arasındaki kosinüs benzerliği 0 olduğundan, bire bir vektörler sözcükler arasındaki benzerlikleri kodlayamaz.

14.1.2 Öz-Gözetimli word2vec

Yukarıdaki sorunu gidermek için word2vec[189] aracı önerilmiştir. Her sözcüğü sabit uzunlukta bir vektöre eşler ve bu vektörler farklı sözcükler arasındaki benzerlik ve benzeşim ilişkisini daha iyi ifade edebilir. word2vec aracı iki model içerir, yani *skip-gram* (Mikolov *et al.*, 2013) ve *sürekli sözcük torbası* (CBOW) (Mikolov *et al.*, 2013). Anlamsal olarak anlamlı temsiller için, eğitimleri, metin kaynaklarındaki bazı sözcükleri çevreleyen sözcüklerin bazılarını tahmin etmek olarak görülebilen koşullu olasılıklara dayanır. Gözetim, etiketsiz verilerden geldiğinden, hem skip-gram hem de sürekli sözcük torbası öz-gözetimli modellerdir.

Aşağıda, bu iki modeli ve eğitim yöntemlerini tanıtacağız.

14.1.3 Skip-Gram Modeli

Skip-gram modeli, bir sözcüğün etrafındaki sözcükleri bir metin dizisinde oluşturmak için kullanılabileceğini varsayar. Örnek olarak "the", "man", "loves", "his", "son" metin dizisini alın. *Merkez sözcük* olarak "loves""i seçelim ve içerik penceresi boyutunu 2'ye ayarlayalım. Fig. 14.1.1 içinde gösterildiği gibi, "loves" merkez sözcüğü göz önüne alındığında, skip-gram modeli, *bağlam sözcüklerini* üretmek için koşullu olasılığı göz önünde bulundurur: "the", "man", "his" ve "son", hepsi orta sözcükten en fazla 2 sözcük uzak değildir:

$$P(\text{"the"}, \text{"man"}, \text{"his"}, \text{"son"} \mid \text{"loves"}). \tag{14.1.2}$$

Bağlam sözcüklerinin bağımsız olarak merkez sözcük (yani koşullu bağımsızlık) verildiğinde oluşturulduğunu varsayalım. Bu durumda, yukarıdaki koşullu olasılık aşağıdaki gibi yeniden yazılabilir:

$$P(\text{"the"} \mid \text{"loves"}) \cdot P(\text{"man"} \mid \text{"loves"}) \cdot P(\text{"his"} \mid \text{"loves"}) \cdot P(\text{"son"} \mid \text{"loves"}). \tag{14.1.3}$$

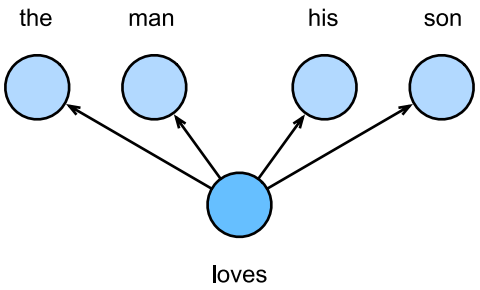


Fig. 14.1.1: skip-gram modeli, bir merkez kelime verilen çevredeki bağlam kelimelerini üretmenin koşullu olasılığını dikkate alır.

Skip-gram modelinde, her sözcüğün koşullu olasılıkları hesaplamak için iki adet d boyutlu vektör gösterimi vardır. Daha somut bir şekilde, sözlükte indeksi i olan herhangi bir sözcük için bir *merkez* sözcük ve bir *bağlam* sözcük kullanıldığında sırasıyla $\mathbf{v}_i \in \mathbb{R}^d$ ve $\mathbf{u}_i \in \mathbb{R}^d$ ile belirtilen iki vektör olarak ifade edelim. Merkez kelimesi w_c (sözlükte c indeksi ile) verilen herhangi bir bağlam kelimesi w_o (sözlükte o indeksi ile) oluşturmanın koşullu olasılığı, vektör nokta çarpımları üzerinde bir softmaks işlemi ile modellenebilir:

$$P(w_o \mid w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}, \tag{14.1.4}$$

[189] https://code.google.com/archive/p/word2vec/

burada sözcük indeksi $\mathcal{V} = \{0, 1, \ldots, |\mathcal{V}| - 1\}$ diye ayarlanır. T uzunluğunda bir metin dizisi göz önüne alındığında, burada t adımındaki sözcük $w^{(t)}$ olarak gösterilir. Bağlam sözcüklerinin herhangi bir merkez sözcük verildiğinde bağımsız olarak oluşturulduğunu varsayın. m boyutlu bağlam penceresi boyutu için, skip-gram modelinin olabilirlik fonksiyonu, herhangi bir merkez kelime verildiğinde tüm bağlam kelimelerini üretme olasılığıdır:

$$\prod_{t=1}^{T} \prod_{-m \leq j \leq m,\ j \neq 0} P(w^{(t+j)} \mid w^{(t)}), \tag{14.1.5}$$

burada 1'den az veya T'den daha büyük herhangi bir zaman adımı atlanabilir.

Eğitim

Skip-gram modeli parametreleri, sözcük dağarcığındaki her sözcük için merkez sözcük vektörü ve bağlam sözcük vektörüdür. Eğitimde, olasılık fonksiyonunu (yani maksimum olabilirlik tahmini) maksimuma çıkararak model parametrelerini öğreniriz. Bu, aşağıdaki kayıp işlevini en aza indirmeye eşdeğerdir:

$$-\sum_{t=1}^{T} \sum_{-m \leq j \leq m,\ j \neq 0} \log P(w^{(t+j)} \mid w^{(t)}). \tag{14.1.6}$$

Kaybı en aza indirmek için rasgele gradyan inişi kullanırken, model parametrelerini güncellemek için bu alt dizi için (rasgele) gradyanı hesaplarken her yinelemede rastgele daha kısa bir alt diziyi örnekleyebiliriz. Bu (rasgele) gradyanı hesaplamak için, merkez sözcük vektörü ve bağlam sözcük vektörüne göre logaritmik koşullu olasılığının gradyanlarını elde etmemiz gerekir. Genel olarak, (14.1.4) ifadesine göre, herhangi w_c merkez kelimesinin ve w_o bağlam kelimesinin herhangi bir çiftini içeren logaritmik koşullu olasılığı aşağıdaki gibidir:

$$\log P(w_o \mid w_c) = \mathbf{u}_o^\top \mathbf{v}_c - \log \left(\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c) \right). \tag{14.1.7}$$

Türev alma sayesinde, merkez sözcük vektörü $\mathbf{v}_c$ ile ilgili olarak gradyanı aşağıdaki gibi elde edebiliriz

$$\begin{aligned} \frac{\partial \log P(w_o \mid w_c)}{\partial \mathbf{v}_c} &= \mathbf{u}_o - \frac{\sum_{j \in \mathcal{V}} \exp(\mathbf{u}_j^\top \mathbf{v}_c) \mathbf{u}_j}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \\ &= \mathbf{u}_o - \sum_{j \in \mathcal{V}} \left(\frac{\exp(\mathbf{u}_j^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \right) \mathbf{u}_j \\ &= \mathbf{u}_o - \sum_{j \in \mathcal{V}} P(w_j \mid w_c) \mathbf{u}_j. \end{aligned} \tag{14.1.8}$$

(14.1.8) içindeki hesaplamanın, sözlükteki tüm kelimelerin merkez kelime w_c ile olan koşullu olasılıklarını gerektirdiğini unutmayın. Diğer kelime vektörlerinin gradyanları da aynı şekilde elde edilebilir.

Eğitimden sonra, sözlükte indeks i olan herhangi bir sözcük için, $\mathbf{v}_i$ (merkez sözcük olarak) ve $\mathbf{u}_i$ (bağlam sözcüğü olarak) iki sözcük vektörü elde ederiz. Doğal dil işleme uygulamalarında, skip-gram modelinin merkez sözcük vektörleri genellikle sözcük temsilleri olarak kullanılır.

14.1.4 Sürekli Sözcük Torbası (CBOW) Modeli

Sürekli sözcük torbası (CBOW) modeli, skip-gram modeline benzer. Skip-gram modelinden en büyük fark, sürekli sözcük torbası modelinin, bir merkez sözcüğün metin dizisindeki çevreleyen bağlam sözcüklerine dayanarak oluşturulduğunu varsaymasıdır. Örneğin, aynı "the", "man", "loves", "his", ve "son" metin dizisinde , "loves" merkez kelimedir ve bağlam penceresi boyutu 2'dir, sürekli sözcük torbası modeli, "the", "man", "his" ve "son" bağlam sözcüklerine dayalı olarak "loves" merkez kelimesini üretme koşullu olasılığını dikkate alır (`fig_cbow` içinde gösterildiği gibi),

$$P(\text{"loves"} \mid \text{"the"}, \text{"man"}, \text{"his"}, \text{"son"}). \tag{14.1.9}$$

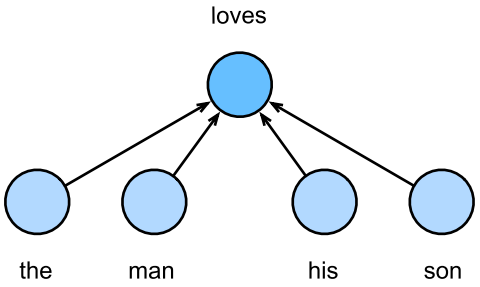


Fig. 14.1.2: Sürekli sözcük torbası modeli, çevreleyen bağlam kelimeleri verilen merkez kelimeyi üretmenin koşullu olasılığını dikkate alır. :label: fig_cbow

Sürekli sözcük torba modelinde çoklu bağlam sözcükleri bulunduğundan, bu bağlam sözcük vektörleri koşullu olasılığın hesaplanmasında ortalanır. Özellikle, sözlükte i dizinine sahip herhangi bir kelime için bir *bağlam* sözcüğü ve bir *merkez* sözcüğü olarak kullanıldığındaki iki vektörü sırasıyla $\mathbf{v}_i \in \mathbb{R}^d$ ve $\mathbf{u}_i \in \mathbb{R}^d$ ile belirtin (anlamlar skip-gram modelinde yer değiştirilir). Çevresindeki bağlam sözcükleri $w_{o_1}, \ldots, w_{o_{2m}}$ (sözlükte dizin c ile) $w_{o_1}, \ldots, w_{o_{2m}}$ (sözlükteki dizin $o_1, \ldots, o_{2m}$) verilen herhangi bir merkez sözcük üretme koşullu olasılığı şöyle modellenebilir:

$$P(w_c \mid w_{o_1}, \ldots, w_{o_{2m}}) = \frac{\exp\left(\frac{1}{2m}\mathbf{u}_c^\top (\mathbf{v}_{o_1} + \ldots, +\mathbf{v}_{o_{2m}})\right)}{\sum_{i \in \mathcal{V}} \exp\left(\frac{1}{2m}\mathbf{u}_i^\top (\mathbf{v}_{o_1} + \ldots, +\mathbf{v}_{o_{2m}})\right)}. \tag{14.1.10}$$

Kısa olması için $\mathcal{W}_o = \{w_{o_1}, \ldots, w_{o_{2m}}\}$ ve $\bar{\mathbf{v}}_o = \left(\mathbf{v}_{o_1} + \ldots, +\mathbf{v}_{o_{2m}}\right)/(2m)$ olsun. Daha sonra (14.1.10) aşağıdaki gibi basitleştirilebilir

$$P(w_c \mid \mathcal{W}_o) = \frac{\exp\left(\mathbf{u}_c^\top \bar{\mathbf{v}}_o\right)}{\sum_{i \in \mathcal{V}} \exp\left(\mathbf{u}_i^\top \bar{\mathbf{v}}_o\right)}. \tag{14.1.11}$$

T uzunluğunda bir metin dizisi göz önüne alındığında, burada t adımdaki sözcük $w^{(t)}$ olarak gösterilir. Bağlam penceresi boyutu m için, sürekli sözcük torbasının olasılık fonksiyonu, bağlam sözcükleri verildiğinde tüm merkez sözcükleri üretme olasılığıdır:

$$\prod_{t=1}^{T} P(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}). \tag{14.1.12}$$

Eğitim

Sürekli sözcük torbası modellerini eğitmek, skip-gram modellerini eğitmekle hemen hemen aynıdır. Sürekli sözcük torba modelinin maksimum olabilirlik tahmini aşağıdaki kayıp fonksiyonunu en aza indirmeye eşdeğerdir:

$$-\sum_{t=1}^{T} \log P(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}). \tag{14.1.13}$$

Dikkatli incelersek,

$$\log\ P(w_c \mid \mathcal{W}_o) = \mathbf{u}_c^\top \bar{\mathbf{v}}_o - \log\ \left(\sum_{i \in \mathcal{V}} \exp\left(\mathbf{u}_i^\top \bar{\mathbf{v}}_o\right)\right). \tag{14.1.14}$$

Türev alma sayesinde, herhangi bir bağlam sözcük vektörü $\mathbf{v}_{o_i}$ $(i = 1, \ldots, 2m)$'ye göre gradyanını elde edebilirsiniz :

$$\frac{\partial \log\ P(w_c \mid \mathcal{W}_o)}{\partial \mathbf{v}_{o_i}} = \frac{1}{2m}\left(\mathbf{u}_c - \sum_{j \in \mathcal{V}} \frac{\exp(\mathbf{u}_j^\top \bar{\mathbf{v}}_o)\mathbf{u}_j}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \bar{\mathbf{v}}_o)}\right) = \frac{1}{2m}\left(\mathbf{u}_c - \sum_{j \in \mathcal{V}} P(w_j \mid \mathcal{W}_o)\mathbf{u}_j\right). \tag{14.1.15}$$

Diğer sözcük vektörlerinin gradyanları aynı şekilde elde edilebilir. Skip-gram modelinin aksine, sürekli sözcük torbası modeli genellikle sözcük temsilleri olarak bağlam sözcük vektörlerini kullanır.

14.1.5 Özet

- Sözcük vektörleri sözcükleri temsil etmek için kullanılan vektörlerdir ve ayrıca öznitelik vektörleri veya sözcüklerin temsilleri olarak da düşünülebilir. Sözcükleri gerçek vektörlere eşleme tekniğine sözcük gömme denir.
- Word2vec aracı hem skip-gram hem de sürekli sözcük torbası modellerini içerir.
- Skip-gram modeli, bir sözcüğün etrafındaki sözcükleri bir metin dizisinde üretmek için kullanılabileceğini varsayar; sürekli sözcük torbası modeli, bir merkez sözcüğün çevresindeki bağlam sözcüklerine dayanarak oluşturulduğunu varsayar.

14.1.6 Alıştırmalar

1. Her gradyanı hesaplamanın hesaplama karmaşıklığı nedir? Sözlük boyutu büyükse ne sorun olabilir?
2. İngilizce bazı sabit ifadeler, "new york" gibi, birden fazla sözcükten oluşur. Onların sözcük vektörleri nasıl eğitilir? İpucu: 4. Bölümdeki word2vec makalesine bakınız (Mikolov *et al.*, 2013).
3. Skip-gram modelini örnek alarak word2vec tasarımını düşünelim. Skip-gram modelindeki iki sözcük vektörünün nokta çarpımıyla kosinüs benzerliği arasındaki ilişki nedir? Benzer anlama sahip bir çift sözcük için, sözcük vektörlerinin (skip-gram modeli tarafından eğitilmiş) kosinüs benzerliği neden yüksek olabilir?

Tartışmalar[190]

[190] https://discuss.d2l.ai/t/381

14.2 Yaklaşık Eğitim

Section 14.1 içindeki tartışmalarımızı hatırlayın. Skip-gram modelinin ana fikri, logaritmik kaybın (14.1.7) tersine karşılık gelen (14.1.4) içinde verilen w_c merkez sözcüğü bazında bir w_o bağlam sözcüğü oluşturmanın koşullu olasılığını hesaplamak için softmaks işlemlerini kullanmaktır.

Softmaks işleminin doğası gereği, bir bağlam sözcüğü $\mathcal{V}$ sözlüğündeki herhangi biri olabileceğinden, (14.1.7) denkleminin tersine, sözcük dağarcığının tüm boyutu kadar olan öğelerin toplamını içerir. Sonuç olarak, (14.1.8) denklemindeki skip-gram modeli için gradyan hesaplamasının ve (14.1.15) denklemindeki sürekli sözcük torba modelinin ikisi de toplamı içerir. Ne yazık ki, büyük bir sözlük üzerinde (genellikle yüz binlerce veya milyonlarca sözcükle) toplamı bu tür gradyanlar için hesaplama maliyeti çok büyüktür!

Yukarıda bahsedilen hesaplama karmaşıklığını azaltmak için, bu bölüm iki yaklaşıklama eğitim yöntemi sunacaktır: *Negatif örnekleme* ve *hiyerarşik softmaks*. Skip-gram modeli ile sürekli sözcük torbası modeli arasındaki benzerlik nedeniyle, bu iki yaklaşıklama eğitim yöntemini tanımlamak için skip-gram modelini örnek olarak alacağız.

14.2.1 Negatif Örnekleme

Negatif örnekleme, orijinal amaç işlevini değiştirir. Bir w_c merkez sözcüğünün bağlam penceresi göz önüne alındığında, herhangi bir w_o (bağlam) sözcüğünün bu bağlam penceresinden gelmesi olayı olasılığı olarak modellenir:

$$P(D = 1 \mid w_c, w_o) = \sigma(\mathbf{u}_o^\top \mathbf{v}_c), \tag{14.2.1}$$

burada σ ile sigmoid etkinleştirme fonksiyonunun tanımı kullanılır:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}. \tag{14.2.2}$$

Sözcük gömmeyi eğitmek için metin dizilerindeki tüm bu olayların bileşik olasılığını en üst düzeye çıkararak başlayalım. Özellikle, T uzunluğunda bir metin dizisi göz önüne alındığında, t zaman adımındaki sözcüğü $w^{(t)}$ ile belirtin ve bağlam penceresi boyutunun m olmasına izin verin, bileşik olasılığın en üst değere çıkarılmasını düşünün:

$$\prod_{t=1}^{T} \prod_{-m \leq j \leq m,\ j \neq 0} P(D = 1 \mid w^{(t)}, w^{(t+j)}). \tag{14.2.3}$$

Ancak, (14.2.3) yalnızca olumlu örnekler içeren olayları dikkate alır. Sonuç olarak, (14.2.3) içindeki bileşik olasılık, yalnızca tüm kelime vektörleri sonsuza eşitse 1'e maksimize edilir. Tabii ki, bu tür sonuçlar anlamsızdır. Amaç işlevini daha anlamlı hale getirmek için *negatif örnekleme*, önceden tanımlanmış bir dağılımdan örneklenen negatif örnekler ekler.

w_o bağlam kelimesinin bir merkez w_c kelimesinin bağlam penceresinden gelmesi olayını S ile belirtin. w_o içeren bu olay için, önceden tanımlanmış bir $P(w)$ dağılımından, bu bağlam penceresinden olmayan K *gürültü sözcükleri*ni örnekler. w_k ($k = 1, \ldots, K$) gürültü sözcüğünün w_c bağlam penceresinden gelmeme olayını N_k ile belirtin. Hem olumlu hem de olumsuz örnekleri içeren bu olayların, $S, N_1, \ldots, N_K$, karşılıklı dışlanan olaylar olduğunu varsayın. (14.2.3) denklemindeki negatif örnekleme, bileşik olasılığı (sadece olumlu örnekler içeren) aşağıdaki gibi yeniden yazar:

$$\prod_{t=1}^{T} \prod_{-m \leq j \leq m,\ j \neq 0} P(w^{(t+j)} \mid w^{(t)}), \tag{14.2.4}$$

$S, N_1, \ldots, N_K$ olayları aracılığıyla koşullu olasılık şöyle yaklaşıklanır:

$$P(w^{(t+j)} \mid w^{(t)}) = P(D = 1 \mid w^{(t)}, w^{(t+j)}) \prod_{k=1,\ w_k \sim P(w)}^{K} P(D = 0 \mid w^{(t)}, w_k). \tag{14.2.5}$$

Sırasıyla, bir metin dizisinin t adımında $w^{(t)}$ sözcüğünün ve bir gürültü w_k sözcüğünün indekslerini i_t ve h_k ile gösterin. (14.2.5)'teki koşullu olasılıklara göre logaritmik kayıp şöyledir:

$$\begin{aligned}
-\log P(w^{(t+j)} \mid w^{(t)}) &= -\log P(D = 1 \mid w^{(t)}, w^{(t+j)}) - \sum_{k=1,\ w_k \sim P(w)}^{K} \log P(D = 0 \mid w^{(t)}, w_k) \\
&= -\log \sigma\left(\mathbf{u}_{i_{t+j}}^\top \mathbf{v}_{i_t}\right) - \sum_{k=1,\ w_k \sim P(w)}^{K} \log\left(1 - \sigma\left(\mathbf{u}_{h_k}^\top \mathbf{v}_{i_t}\right)\right) \\
&= -\log \sigma\left(\mathbf{u}_{i_{t+j}}^\top \mathbf{v}_{i_t}\right) - \sum_{k=1,\ w_k \sim P(w)}^{K} \log \sigma\left(-\mathbf{u}_{h_k}^\top \mathbf{v}_{i_t}\right).
\end{aligned} \tag{14.2.6}$$

Artık her eğitim adımındaki gradyanlar için hesaplama maliyetinin sözlük boyutuyla ilgisi olmadığını, ancak doğrusal olarak K'ya bağlı olduğunu görebiliyoruz. Hiper parametre K'yı daha küçük bir değere ayarlarken, negatif örnekleme ile her eğitim adımındaki gradyanlar için hesaplama maliyeti daha küçüktür.

14.2.2 Hiyerarşik Softmaks

Alternatif bir yaklaşıklama eğitimi yöntemi olarak, *hiyerarşik softmaks* ikili ağacı kullanır ve bu veri yapısı Fig. 14.2.1 içinde gösterilen bir veri yapısıdır; burada ağacın her bir yaprak düğümü $\mathcal{V}$ sözlüğünde bir sözcüğü temsil eder.

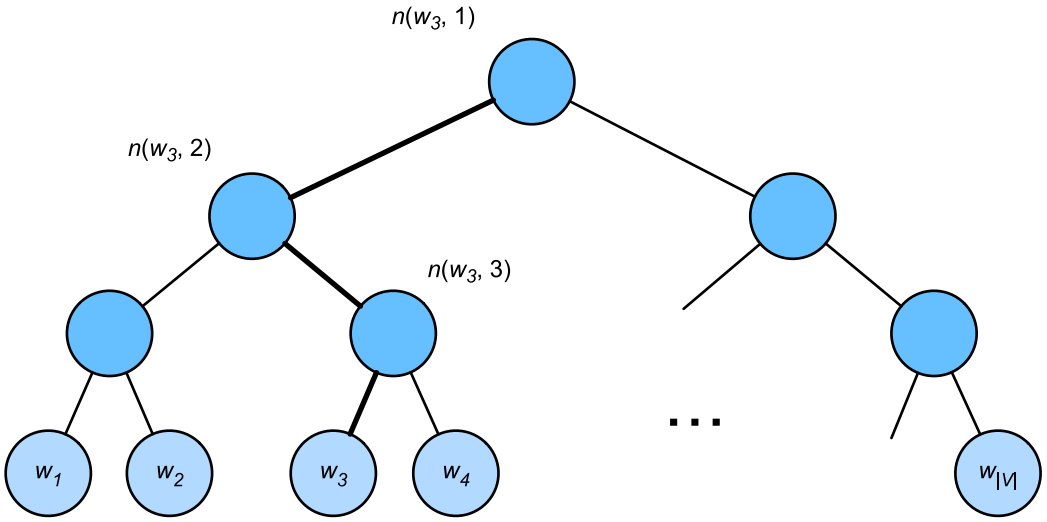


Fig. 14.2.1: Ağacın her yaprak düğümünün sözlükte bir kelimeyi temsil ettiği yaklaşıklama eğitimi için hiyerarşik softmaks.

İkili ağaçtaki w kelimesini temsil eden kök düğümden yaprak düğüme giden yoldaki düğümlerin sayısını (her iki uç dahil) $L(w)$ ile belirtin. $n(w, j)$, bağlam kelime vektörü $\mathbf{u}_{n(w,j)}$ olacak şekilde,

bu yoldaki j. düğüm olsun. Örneğin, Fig. 14.2.1 içinde $L(w_3) = 4$ olsun. (14.1.4) denklemindeki koşullu olasılık hiyerarşik softmaks ile yaklaşıklanır:

$$P(w_o \mid w_c) = \prod_{j=1}^{L(w_o)-1} \sigma\left([\![n(w_o, j+1) = \text{leftChild}(n(w_o, j))]\!] \cdot \mathbf{u}_{n(w_o,j)}^\top \mathbf{v}_c\right), \tag{14.2.7}$$

σ işlevi (14.2.2) denleminde tanımlanır ve leftChild(n), n düğümünün sol alt düğümüdür: x doğruysa, $[\![x]\!] = 1$; aksi halde $[\![x]\!] = -1$.

Göstermek için, Fig. 14.2.1 içinde w_c sözcüğü verildiğinde w_3 sözcüğünü üretme koşullu olasılığını hesaplayalım. Bu, w_c arasında $\mathbf{v}_c$ sözcük vektörü ve yaprak olmayan düğüm vektörleri arasında nokta çarpımlarını gerektirir (Fig. 14.2.1 içindeki kalın yol), kökten w_3'e kadar sola, sağa, sonra sola ilerler:

$$P(w_3 \mid w_c) = \sigma(\mathbf{u}_{n(w_3,1)}^\top \mathbf{v}_c) \cdot \sigma(-\mathbf{u}_{n(w_3,2)}^\top \mathbf{v}_c) \cdot \sigma(\mathbf{u}_{n(w_3,3)}^\top \mathbf{v}_c). \tag{14.2.8}$$

$\sigma(x) + \sigma(-x) = 1$ olduğundan, $\mathcal{V}$ sözlüğündeki tüm sözcükleri herhangi bir w_c sözcüğüne dayalı olarak üretmenin koşullu olasılıklarının toplamının 1 olduğunu tutar:

$$\sum_{w \in \mathcal{V}} P(w \mid w_c) = 1. \tag{14.2.9}$$

Neyse ki, $L(w_o) - 1$ ikili ağaç yapısı nedeniyle $\mathcal{O}(\log_2|\mathcal{V}|)$ düzeyinde olduğundan, sözlük boyutu $\mathcal{V}$ çok büyüktür, hiyerarşik softmaks kullanan her eğitim adımının hesaplama maliyeti, yaklaşıklama eğitimi olmadan yapılana kıyasla önemli ölçüde azalır.

14.2.3 Özet

- Negatif örnekleme, hem pozitif hem de negatif örnekler içeren karşılıklı dışlanan olayları göz önünde bulundurarak kayıp işlevini oluşturur. Eğitim için hesaplama maliyeti doğrusal olarak her adımdaki gürültü sözcüklerinin sayısına bağlıdır.
- Hiyerarşik softmaks, kök düğümünden ikili ağacın yaprak düğümüne giden yolu kullanarak kayıp işlevini oluşturur. Eğitim için hesaplama maliyeti, her adımdaki sözlük boyutunun logaritmasına bağlıdır.

14.2.4 Alıştırmalar

1. Negatif örneklemede gürültü sözcüklerini nasıl örnekleyebiliriz?
2. (14.2.9) denkleminin tuttuğunu doğrulayın.
3. Sırasıyla negatif örnekleme ve hiyerarşik softmaks kullanarak sürekli kelime torbası modeli nasıl eğitilir?

Tartışmalar[191]

[191] https://discuss.d2l.ai/t/382

14.3 Sözcük Gömme Ön Eğitimi İçin Veri Kümesi

Artık word2vec modellerinin teknik ayrıntılarını ve yaklaşıklama eğitim yöntemlerini bildiğimize göre, uygulamalarını inceleyelim. Özellikle, Section 14.1 içinde skip-gram modelini ve Section 14.2 içinde negatif örneklemeyi örnek olarak alacağız. Bu bölümde, sözcük gömme modeli ön eğitimi için veri kümesi ile başlıyoruz: Verilerin orijinal biçimi eğitim sırasında yinelenebilen minigruplar haline dönüştürülecektir.

```
import math
import os
import random
import torch
from d2l import torch as d2l
```

14.3.1 Veri Kümesini Okuma

Burada kullandığımız veri kümesi Penn Tree Bank (PTB)[192]'dir. Bu külliyat, Wall Street Journal makalelerinden örneklenmiştir ve eğitim, geçerleme ve test kümelerine bölünmüştür. Özgün biçimde, metin dosyasının her satırı boşluklarla ayrılmış bir sözcükler cümlesini temsil eder. Burada her sözcüğe bir belirteç gibi davranıyoruz.

```
#@save
d2l.DATA_HUB['ptb'] = (d2l.DATA_URL + 'ptb.zip',
                       '319d85e578af0cdc590547f26231e4e31cdf1e42')

#@save
def read_ptb():
    """Load the PTB dataset into a list of text lines."""
    data_dir = d2l.download_extract('ptb')
    # Read the training set.
    with open(os.path.join(data_dir, 'ptb.train.txt')) as f:
        raw_text = f.read()
    return [line.split() for line in raw_text.split('\n')]

sentences = read_ptb()
f'# sentences: {len(sentences)}'
```

```
'# sentences: 42069'
```

Eğitim kümesini okuduktan sonra, 10 kereden az görünen herhangi bir sözcüğün "<unk>" belirteci ile değiştirildiği külliyat için bir sözcük dağarcığı oluşturuyoruz. Özgün veri kümesinin nadir (bilinmeyen) sözcükleri temsil eden "<unk>" belirteçleri de içerdiğini unutmayın.

```
vocab = d2l.Vocab(sentences, min_freq=10)
f'vocab size: {len(vocab)}'
```

```
'vocab size: 6719'
```

[192] https://catalog.ldc.upenn.edu/LDC99T42

14.3.2 Alt Örnekleme

Metin verileri genellikle "the", "a" ve "in" gibi yüksek frekanslı sözcüklere sahiptir: Hatta çok büyük külliyatta milyarlarca kez ortaya çıkabilirler. Ancak, bu sözcükler genellikle bağlam pencerelerinde birçok farklı sözcükle birlikte ortaya çıkar ve çok az yararlı sinyaller sağlar. Örneğin, bir bağlam penceresinde ("çip") "chip" sözcüğünü göz önünde bulundurun: Sezgisel olarak düşük frekanslı bir "intel" sözcüğüyle birlikte oluşması, eğitimde yüksek frekanslı bir sözcük "a" ile birlikte oluşmasından daha yararlıdır. Dahası, büyük miktarda (yüksek frekanslı) sözcüklerle eğitim yavaştır. Böylece, sözcük gömme modellerini eğitiyorken, yüksek frekanslı sözcükler *alt örneklenebilir* (Mikolov *et al.*, 2013). Özellikle, veri kümesindeki dizine alınmış her bir w_i sözcüğü aşağıdaki olasılıkla atılacaktır.

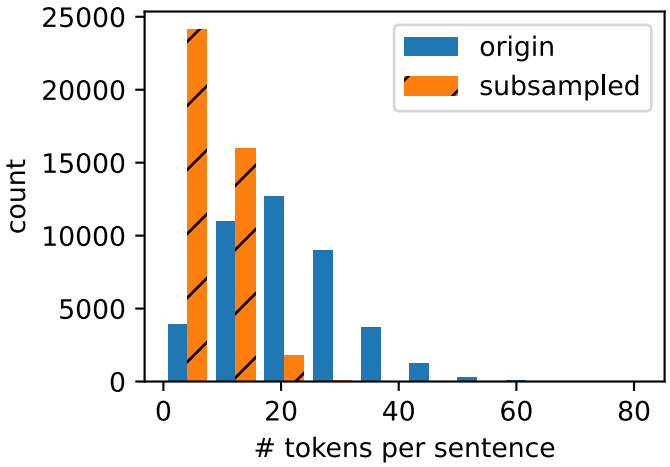
$$P(w_i) = \max\left(1 - \sqrt{\frac{t}{f(w_i)}}, 0\right), \tag{14.3.1}$$

burada $f(w_i)$, w_i sözcüklerinin sayısının veri kümelerindeki toplam sözcük sayısına oranıdır ve t sabit (deneyde 10^{-4}) bir hiper parametredir. Sadece göreli frekans $f(w_i) > t$ (yüksek frekanslı) olursa w_i sözcüğü atılabilir ve sözcüğün göreli frekansı ne kadar yüksek olursa, atılma olasılığı o kadar yüksektir.

```
#@save
def subsample(sentences, vocab):
    """Yüksek frekanslı sözcükleri alt örnekle."""
    # '<unk>' andıçlarını hariç tut
    sentences = [[token for token in line if vocab[token] != vocab.unk]
                 for line in sentences]
    counter = d2l.count_corpus(sentences)
    num_tokens = sum(counter.values())

    # Alt örnekleme sırasında `token` tutulursa True döndür
    def keep(token):
        return(random.uniform(0, 1) <
               math.sqrt(1e-4 / counter[token] * num_tokens))

    return ([[token for token in line if keep(token)] for line in sentences],
            counter)

subsampled, counter = subsample(sentences, vocab)
```

Aşağıdaki kod parçacığı, alt örnekleme öncesi ve sonrasında cümle başına belirteç sayısının histogramını çizer. Beklendiği gibi, alt örnekleme, yüksek frekanslı sözcükleri düşürerek cümleleri önemli ölçüde kısaltır ve bu da eğitim hızlandırmasına yol açacaktır.

```
d2l.show_list_len_pair_hist(['origin', 'subsampled'], '# tokens per sentence',
                            'count', sentences, subsampled);
```



Bireysel belirteçler için, yüksek frekanslı “the” sözcüğünün örnekleme oranı 1/20’den azdır.

```
def compare_counts(token):
    return (f'# of "{token}": '
            f'before={sum([l.count(token) for l in sentences])}, '
            f'after={sum([l.count(token) for l in subsampled])}')

compare_counts('the')
```

```
'# of "the": before=50770, after=2090'
```

Buna karşılık, düşük frekanslı “join” sözcüğü tamamen tutulur.

```
compare_counts('join')
```

```
'# of "join": before=45, after=45'
```

Alt örneklemeden sonra, belirteçleri külliyat için indekslerine eşliyoruz.

```
corpus = [vocab[line] for line in subsampled]
corpus[:3]
```

```
[[], [392, 2115, 18], [22, 5277, 3054, 1580]]
```

14.3.3 Merkezi Sözcükleri ve Bağlam Sözcüklerini Ayıklamak

Aşağıdaki `get_centers_and_contexts` işlevi, `corpus`’ten tüm merkez sözcükleri ve onların bağlam sözcüklerini ayıklar. Bağlam penceresi boyutu olarak rastgele olarak `1` ile `max_window_size` arasında bir tamsayıya eşit olarak örnekler. Herhangi bir merkez sözcük için, uzaklığı örneklenen bağlam penceresi boyutunu aşmayan sözcükler, bağlam sözcükleridir.

```
#@save
def get_centers_and_contexts(corpus, max_window_size):
    """skip-gramdaki merkez sözcüklerini ve bağlam sözcüklerini döndürür."""
    centers, contexts = [], []
    for line in corpus:
        # Bir "merkez sözcük-bağlam sözcüğü" çifti oluşturmak için
        # her cümlede en az 2 kelime olması gerekir
        if len(line) < 2:
            continue
        centers += line
        for i in range(len(line)):  #  `i` merkezli bağlam penceresi
            window_size = random.randint(1, max_window_size)
            indices = list(range(max(0, i - window_size),
                                 min(len(line), i + 1 + window_size)))
            # Merkez sözcüğü bağlam sözcüklerinden çıkar
            indices.remove(i)
            contexts.append([line[idx] for idx in indices])
    return centers, contexts
```

Ardından, sırasıyla 7 ve 3 sözcükten oluşan iki cümle içeren yapay bir veri kümesi oluşturuyoruz. Maksimum bağlam penceresi boyutunun 2 olmasını sağlayın ve tüm merkez sözcüklerini ve onların bağlam sözcüklerini yazdırın.

```
tiny_dataset = [list(range(7)), list(range(7, 10))]
print('dataset', tiny_dataset)
for center, context in zip(*get_centers_and_contexts(tiny_dataset, 2)):
    print('center', center, 'has contexts', context)
```

```
dataset [[0, 1, 2, 3, 4, 5, 6], [7, 8, 9]]
center 0 has contexts [1, 2]
center 1 has contexts [0, 2]
center 2 has contexts [0, 1, 3, 4]
center 3 has contexts [1, 2, 4, 5]
center 4 has contexts [2, 3, 5, 6]
center 5 has contexts [4, 6]
center 6 has contexts [5]
center 7 has contexts [8, 9]
center 8 has contexts [7, 9]
center 9 has contexts [7, 8]
```

PTB veri kümesi üzerinde eğitim yaparken, maksimum bağlam penceresi boyutunu 5'e ayarladık. Aşağıdaki, veri kümelerindeki tüm merkez sözcüklerini ve onların bağlam sözcüklerini ayıklar.

```
all_centers, all_contexts = get_centers_and_contexts(corpus, 5)
f'# center-context pairs: {sum([len(contexts) for contexts in all_contexts])}'
```

```
'# center-context pairs: 1500630'
```

14.3.4 Negatif Örnekleme

Yaklaşıklama eğitim için negatif örnekleme kullanıyoruz. Gürültülü sözcükleri önceden tanımlanmış bir dağıtıma göre örneklemek için aşağıdaki `RandomGenerator` sınıfını tanımlıyoruz; burada (muhtemelen normalleştirilmemiş) örnekleme dağılımı `sampling_weights` argümanı üzerinden geçirilir.

```
#@save
class RandomGenerator:
    """Örnekleme ağırlıklarına göre {1, ..., n} arasından rastgele çek."""
    def __init__(self, sampling_weights):
        # Hariç tut
        self.population = list(range(1, len(sampling_weights) + 1))
        self.sampling_weights = sampling_weights
        self.candidates = []
        self.i = 0

    def draw(self):
        if self.i == len(self.candidates):
            # `k` rastgele örnekleme sonuçlarını önbelleğe al
            self.candidates = random.choices(
                self.population, self.sampling_weights, k=10000)
            self.i = 0
        self.i += 1
        return self.candidates[self.i - 1]
```

Örneğin, $P(X = 1) = 2/9, P(X = 2) = 3/9$ ve $P(X = 3) = 4/9$ örnekleme olasılıkları ile 1, 2 ve 3 indeksleri arasından 10 adet X rasgele değişkenini çekebiliriz.

Bir çift merkez sözcük ve bağlam sözcüğü için, rastgele `K` (deneyde 5) gürültü sözcüğü örnekleriz. Word2vec makalesindeki önerilere göre, w gürültü sözcüğünün $P(w)$ örnekleme olasılığı, 0.75'in (Mikolov *et al.*, 2013) gücüne yükseltilmiş olarak sözlükteki göreli frekansına ayarlanır.

```
#@save
def get_negatives(all_contexts, vocab, counter, K):
    """Negatif örneklemede gürültü sözcüklerini döndür."""
    # Sözlükte 1, 2, ... (dizin 0 hariç tutulan bilinmeyen belirteçtir)
    # olan kelimeler için örnekleme ağırlıkları
    sampling_weights = [counter[vocab.to_tokens(i)]**0.75
                        for i in range(1, len(vocab))]
    all_negatives, generator = [], RandomGenerator(sampling_weights)
    for contexts in all_contexts:
        negatives = []
        while len(negatives) < len(contexts) * K:
            neg = generator.draw()
            # Gürültü sözcükleri bağlam sözcükleri olamaz
            if neg not in contexts:
                negatives.append(neg)
        all_negatives.append(negatives)
    return all_negatives

all_negatives = get_negatives(all_contexts, vocab, counter, 5)
```

14.3.5 Minigruplarda Eğitim Örneklerini Yükleme

Bağlam sözcükleri ve örneklenmiş gürültü sözcükleri ile birlikte tüm merkezi sözcükler çıkarıldıktan sonra, eğitim sırasında yinelemeli olarak yüklenebilecek örneklerin minigruplarına dönüştürülecektir.

Bir minigrupta, i. örnek bir merkez sözcük ve onun n_i bağlam sözcükleri ve m_i gürültü sözcükleri içerir. Değişen bağlam penceresi boyutları nedeniyle $n_i + m_i$ farklı i'ler için değişiklik gösterir. Böylece, her örnek için bağlam sözcüklerini ve gürültü sözcüklerini `contexts_negatives` değişkeninde bitiştiririz ve bitiştirme uzunluğu $\max_i n_i + m_i$'a (`max_len`) ulaşana kadar sıfırlarla dolgularız. Kaybın hesaplanmasında dolguları hariç tutmak için bir maske değişkeni, `masks`, tanımlıyoruz. `masks`'taki elemanlar ve `contexts_negatives`'teki elemanlar arasında bire bir karşılık vardır, burada `masks`'daki sıfırlar (aksi takdirde birler) `contexts_negatives`'teki dolgulara karşılık gelir.

Pozitif ve negatif örnekleri ayırt etmek için `contexts_negatives` içindeki gürültü sözcüklerinden `labels` değişkeni aracılığıyla bağlam sözcüklerini ayırırız. `masks`'e benzer şekilde, `labels`'daki elemanlar ve `contexts_negatives`'teki elemanlar arasında bire bir karşılık vardır ve `labels`'daki birler (aksi takdirde sıfırlar) `contexts_negatives`'teki bağlam sözcüklerine (olumlu örneklere) karşılık gelir.

Yukarıdaki fikir aşağıdaki `batchify` işlevinde uygulanmaktadır. Girdi `data` toplu boyutuna eşit uzunlukta bir listedir; her öğesi `center` merkez sözcüklerinden, `context` bağlam sözcüklerinden ve `negative` gürültü sözcüklerinden oluşan bir örnektir. Bu işlev, maske değişkeni de dahil olmak üzere eğitim sırasında hesaplamalar için yüklenebilecek bir minigrup döndürür.

```
#@save
def batchify(data):
    """Negatif örnekleme ile skip-gram için bir minigrup örnek döndür."""
    max_len = max(len(c) + len(n) for _, c, n in data)
    centers, contexts_negatives, masks, labels = [], [], [], []
    for center, context, negative in data:
        cur_len = len(context) + len(negative)
        centers += [center]
        contexts_negatives += [context + negative + [0] * (max_len - cur_len)]
        masks += [[1] * cur_len + [0] * (max_len - cur_len)]
        labels += [[1] * len(context) + [0] * (max_len - len(context))]
    return (torch.tensor(centers).reshape((-1, 1)), torch.tensor(
        contexts_negatives), torch.tensor(masks), torch.tensor(labels))
```

Bu işlevi iki örnekten oluşan bir minigrup kullanarak test edelim.

```
x_1 = (1, [2, 2], [3, 3, 3, 3])
x_2 = (1, [2, 2, 2], [3, 3])
batch = batchify((x_1, x_2))

names = ['centers', 'contexts_negatives', 'masks', 'labels']
for name, data in zip(names, batch):
    print(name, '=', data)
```

```
centers = tensor([[1],
        [1]])
contexts_negatives = tensor([[2, 2, 3, 3, 3, 3],
```

(continues on next page)

(continued from previous page)

```
        [2, 2, 2, 3, 3, 0]])
masks = tensor([[1, 1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1, 0]])
labels = tensor([[1, 1, 0, 0, 0, 0],
        [1, 1, 1, 0, 0, 0]])
```

14.3.6 Her Şeyi Bir Araya Getirmek

Son olarak, PTB veri kümesini okuyan ve veri yineleyicisini ve sözcük dağarcığını döndüren `load_data_ptb` işlevini tanımlıyoruz.

```
#@save
def load_data_ptb(batch_size, max_window_size, num_noise_words):
    """Download the PTB dataset and then load it into memory."""
    num_workers = d2l.get_dataloader_workers()
    sentences = read_ptb()
    vocab = d2l.Vocab(sentences, min_freq=10)
    subsampled, counter = subsample(sentences, vocab)
    corpus = [vocab[line] for line in subsampled]
    all_centers, all_contexts = get_centers_and_contexts(
        corpus, max_window_size)
    all_negatives = get_negatives(
        all_contexts, vocab, counter, num_noise_words)

    class PTBDataset(torch.utils.data.Dataset):
        def __init__(self, centers, contexts, negatives):
            assert len(centers) == len(contexts) == len(negatives)
            self.centers = centers
            self.contexts = contexts
            self.negatives = negatives

        def __getitem__(self, index):
            return (self.centers[index], self.contexts[index],
                    self.negatives[index])

        def __len__(self):
            return len(self.centers)

    dataset = PTBDataset(all_centers, all_contexts, all_negatives)

    data_iter = torch.utils.data.DataLoader(dataset, batch_size, shuffle=True,
                                    collate_fn=batchify,
                                    num_workers=num_workers)
    return data_iter, vocab
```

Veri yineleyicisinin ilk minigrubunu yazdıralım.

```
data_iter, vocab = load_data_ptb(512, 5, 5)
for batch in data_iter:
    for name, data in zip(names, batch):
        print(name, 'shape:', data.shape)
    break
```

```
centers shape: torch.Size([512, 1])
contexts_negatives shape: torch.Size([512, 60])
masks shape: torch.Size([512, 60])
labels shape: torch.Size([512, 60])
```

14.3.7 Özet

- Yüksek frekanslı sözcükler eğitimde o kadar yararlı olmayabilir. Eğitimde hızlanmak için onları alt örnekleyebiliriz.
- Hesaplama verimliliği için örnekleri minigruplar halinde yükleriz. Dolguları dolgu olmayanlardan ve olumlu örnekleri olumsuz olanlardan ayırt etmek için başka değişkenler tanımlayabiliriz.

14.3.8 Alıştırmalar

1. Alt örnekleme kullanmıyorsa, bu bölümdeki kodun çalışma süresi nasıl değişir?
2. `RandomGenerator` sınıfı `k` rasgele örnekleme sonuçlarını önbelleğe alır. `k`'yi diğer değerlere ayarlayın ve veri yükleme hızını nasıl etkilediğini görün.
3. Bu bölümün kodundaki hangi diğer hiper parametreler veri yükleme hızını etkileyebilir?

Tartışmalar[193]

14.4 word2vec Ön Eğitimi

Section 14.1 içinde tanımlanan skip-gram modelini uygulamaya devam ediyoruz. Ardından, PTB veri kümesindeki negatif örnekleme kullanarak word2vec ön eğiteceğiz. Her şeyden önce, Section 14.3 içinde açıklanan `d2l.load_data_ptb` işlevini çağırarak veri yineleyicisini ve bu veri kümesi için sözcük dağarcığını elde edelim.

```
import math
import torch
from torch import nn
from d2l import torch as d2l

batch_size, max_window_size, num_noise_words = 512, 5, 5
data_iter, vocab = d2l.load_data_ptb(batch_size, max_window_size,
                                     num_noise_words)
```

[193] https://discuss.d2l.ai/t/1330

14.4.1 Skip-Gram Modeli

Katmanları ve toplu matris çarpımlarını kullanarak skip-gram modelini uyguluyoruz. İlk olarak, gömme katmanların nasıl çalıştığını inceleyelim.

Gömme Katmanı

Section 9.7 içinde açıklandığı gibi, bir katman, bir belirteç dizinini öznitelik vektörüyle eşler. Bu katmanın ağırlığı, satır sayısı sözlük boyutuna (`input_dim`) ve sütun sayısı her belirteç için vektör boyutuna (`output_dim`) eşit olan bir matristir. Bir sözcük gömme modeli eğitildikten sonra, bu ağırlık ihtiyacımız olan şeydir.

```
embed = nn.Embedding(num_embeddings=20, embedding_dim=4)
print(f'Parameter embedding_weight ({embed.weight.shape}, '
      f'dtype={embed.weight.dtype})')
```

```
Parameter embedding_weight (torch.Size([20, 4]), dtype=torch.float32)
```

Gömülü katmanın girdisi, bir belirteç (sözcük) dizinidir. Herhangi bir belirteç dizini i için vektör gösterimi, gömme katmanındaki ağırlık matrisinin $i.$ satırından elde edilebilir. Vektör boyutu (`output_dim`) 4 olarak ayarlandığından, gömme katmanı (2, 3) şekle sahip bir minigrup işlemi için (2, 3, 4) şekilli vektörler döndürür.

```
x = torch.tensor([[1, 2, 3], [4, 5, 6]])
embed(x)
```

```
tensor([[[-1.7168,  1.5441, -0.8471,  0.6239],
         [-0.8008,  0.1647, -0.0691, -0.4562],
         [-0.4796,  1.1483,  1.5281,  0.7885]],

        [[ 1.3627, -0.1596, -0.4472, -1.0897],
         [-0.2692,  0.7414,  0.3256, -0.1186],
         [ 0.2164, -0.8939,  0.3935,  2.7814]]], grad_fn=<EmbeddingBackward0>)
```

İleri Yaymayı Tanımlama

İleri yayılımda, skip-gram modelinin girdisi, (toplu iş boyutu, 1) şekilli `center` merkez sözcük indekslerini ve `max_len`'in Section 14.3.5 içinde tanımlandığı (toplu iş boyutu, `max_len`) şeklindeki `contexts_and_negatives` bitiştirilmiş bağlam ve gürültü sözcük indekslerini içerir. Bu iki değişken önce belirteç dizinlerinden gömme katmanı aracılığıyla vektörlere dönüştürülür, daha sonra toplu matris çarpımı (Section 10.2.4 içinde açıklanmıştır) (toplu iş boyutu, 1, `max_len`) şekilli bir çıktı döndürür. Çıktıdaki her eleman, bir merkez sözcük vektörü ile bir bağlam veya gürültü sözcük vektörünün nokta çarpımıdır .

```
def skip_gram(center, contexts_and_negatives, embed_v, embed_u):
    v = embed_v(center)
    u = embed_u(contexts_and_negatives)
    pred = torch.bmm(v, u.permute(0, 2, 1))
    return pred
```

Bazı örnek girdiler için bu `skip_gram` işlevinin çıktı şeklini yazdıralım.

```
skip_gram(torch.ones((2, 1), dtype=torch.long),
          torch.ones((2, 4), dtype=torch.long), embed, embed).shape
```

```
torch.Size([2, 1, 4])
```

14.4.2 Eğitim

Skip-gram modelini negatif örnekleme ile eğitmeden önce, öncelikle kayıp işlevini tanımlayalım.

İkili Çapraz Entropi Kaybı

Section 14.2.1 içinde negatif örnekleme için kayıp fonksiyonunun tanımına göre ikili çapraz entropi kaybını kullanacağız.

```
class SigmoidBCELoss(nn.Module):
    # Maskeleme ile ikili çapraz entropi kaybı
    def __init__(self):
        super().__init__()

    def forward(self, inputs, target, mask=None):
        out = nn.functional.binary_cross_entropy_with_logits(
            inputs, target, weight=mask, reduction="none")
        return out.mean(dim=1)

loss = SigmoidBCELoss()
```

Section 14.3.5 içindeki maske değişkeninin ve etiket değişkeninin tanımlarını hatırlayın. Aşağıdaki ifade, verilen değişkenler için ikili çapraz entropi kaybını hesaplar.

```
pred = torch.tensor([[1.1, -2.2, 3.3, -4.4]] * 2)
label = torch.tensor([[1.0, 0.0, 0.0, 0.0], [0.0, 1.0, 0.0, 0.0]])
mask = torch.tensor([[1, 1, 1, 1], [1, 1, 0, 0]])
loss(pred, label, mask) * mask.shape[1] / mask.sum(axis=1)
```

```
tensor([0.9352, 1.8462])
```

Aşağısı, ikili çapraz entropi kaybında sigmoid etkinleştirme fonksiyonu kullanılarak yukarıdaki sonuçların nasıl hesaplandığını (daha az verimli bir şekilde) göstermektedir. İki çıktıyı, maskelenmemiş tahminlere göre ortalaması alınan iki normalleştirilmiş kayıp olarak düşünebiliriz.

```
def sigmd(x):
    return -math.log(1 / (1 + math.exp(-x)))

print(f'{(sigmd(1.1) + sigmd(2.2) + sigmd(-3.3) + sigmd(4.4)) / 4:.4f}')
print(f'{(sigmd(-1.1) + sigmd(-2.2)) / 2:.4f}')
```

```
0.9352
1.8462
```

Model Parametrelerini İlkleme

Sırasıyla merkez sözcükler ve bağlam sözcükleri olarak kullanıldıklarından sözcük dağarcığındaki tüm sözcükler için iki gömme katmanı tanımlarız. `embed_size` sözcük vektör boyutu 100 olarak ayarlanır.

```
embed_size = 100
net = nn.Sequential(nn.Embedding(num_embeddings=len(vocab),
                                 embedding_dim=embed_size),
                    nn.Embedding(num_embeddings=len(vocab),
                                 embedding_dim=embed_size))
```

Eğitim Döngüsünün Tanımlanması

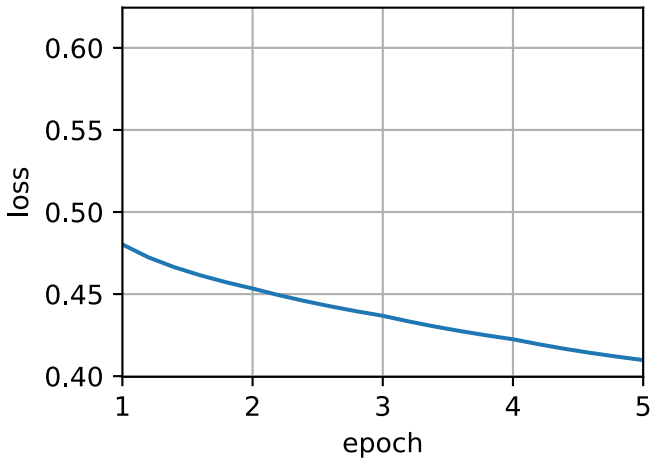
Eğitim döngüsü aşağıda tanımlanmıştır. Dolgunun varlığı nedeniyle, kayıp fonksiyonunun hesaplanması önceki eğitim fonksiyonlarına kıyasla biraz farklıdır.

```
def train(net, data_iter, lr, num_epochs, device=d2l.try_gpu()):
    def init_weights(m):
        if type(m) == nn.Embedding:
            nn.init.xavier_uniform_(m.weight)
    net.apply(init_weights)
    net = net.to(device)
    optimizer = torch.optim.Adam(net.parameters(), lr=lr)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[1, num_epochs])
    # Normalleştirilmiş kayıpların toplamı, normalleştirilmiş kayıpların sayısı
    metric = d2l.Accumulator(2)
    for epoch in range(num_epochs):
        timer, num_batches = d2l.Timer(), len(data_iter)
        for i, batch in enumerate(data_iter):
            optimizer.zero_grad()
            center, context_negative, mask, label = [
                data.to(device) for data in batch]

            pred = skip_gram(center, context_negative, net[0], net[1])
            l = (loss(pred.reshape(label.shape).float(), label.float(), mask)
                     / mask.sum(axis=1) * mask.shape[1])
            l.sum().backward()
            optimizer.step()
            metric.add(l.sum(), l.numel())
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (metric[0] / metric[1],))
    print(f'loss {metric[0] / metric[1]:.3f}, '
          f'{metric[1] / timer.stop():.1f} tokens/sec on {str(device)}')
```

Artık negatif örnekleme kullanarak bir skip-gram modelini eğitebiliriz.

```
lr, num_epochs = 0.002, 5
train(net, data_iter, lr, num_epochs)
```

```
loss 0.410, 361758.5 tokens/sec on cuda:0
```



14.4.3 Sözcük Gömmelerini Uygulama

Word2vec modelini eğittikten sonra, eğitimli modeldeki sözcük vektörlerinin kosinüs benzerliğini kullanarak sözlükten bir girdi sözcüğüne en çok benzeyen sözcükleri bulabiliriz.

```
def get_similar_tokens(query_token, k, embed):
    W = embed.weight.data
    x = W[vocab[query_token]]
    # Kosinüs benzerliğini hesaplayın. Sayısal kararlılık için 1e-9 ekleyin
    cos = torch.mv(W, x) / torch.sqrt(torch.sum(W * W, dim=1) *
                                      torch.sum(x * x) + 1e-9)
    topk = torch.topk(cos, k=k+1)[1].cpu().numpy().astype('int32')
    for i in topk[1:]:  # Remove the input words
        print(f'cosine sim={float(cos[i]):.3f}: {vocab.to_tokens(i)}')

get_similar_tokens('chip', 3, net[0])
```

```
cosine sim=0.679: microprocessor
cosine sim=0.665: intel
cosine sim=0.626: processes
```

14.4.4 Özet

- Gömülü katmanlar ve ikili çapraz entropi kaybını kullanarak negatif örnekleme ile bir skip-gram modelini eğitebiliriz.
- Sözcük gömme uygulamaları, sözcük vektörlerinin kosinüs benzerliğine dayanan belirli bir sözcük için anlamsal olarak benzer sözcükleri bulmayı içerir.

14.4.5 Alıştırmalar

1. Eğitilmiş modeli kullanarak, diğer girdi sözcükleri için anlamsal olarak benzer sözcükleri bulun. Hiper parametreleri ayarlayarak sonuçları iyileştirebilir misiniz?
2. Bir eğitim külliyatı çok büyük olduğunda, *model parametrelerini güncellerken* mevcut mini-grup içindeki merkez sözcükler için bağlam sözcükleri ve gürültü sözcüklerini sık sık örnekleriz. Başka bir deyişle, aynı merkez sözcük farklı eğitim dönemlerinde farklı bağlam sözcüklerine veya gürültü sözcüklerine sahip olabilir. Bu yöntemin faydaları nelerdir? Bu eğitim yöntemini uygulamaya çalışın.

Tartışmalar[194]

14.5 Küresel Vektörler ile Sözcük Gömme (GloVe)

Bağlam pencerelerindeki sözcük-sözcük birlikte oluşumları zengin anlamsal bilgiler taşıyabilir. Örneğin, büyük bir külliyatta "katı" sözcüğünün "buhar" yerine "buz" ile birlikte ortaya çıkması daha olasıdır, ancak "gaz" sözcüğü muhtemelen "buhar" ile birlikte bulunur. Ayrıca, bu tür ortak olayların küresel külliyat istatistikleri önceden hesaplanabilir: Bu daha verimli eğitime yol açabilir. Sözcük gömme için tüm külliyat içindeki istatistiksel bilgileri kullanmak için, önce Section 14.1.3 içindeki skip-gram modelini tekrar gözden geçirelim, ancak ortak oluşum sayıları gibi küresel külliyat istatistiklerini kullanarak yorumlayalım.

14.5.1 Küresel Külliyat İstatistikleri ile Skip-Gram

Skip-gram modelinde w_i sözcüğü verildiğinde w_j sözcüğünün $P(w_j \mid w_i)$ koşullu olasılığını q_{ij} ile ifade edersek, elimizde şu vardır:

$$q_{ij} = \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_i)}{\sum_{k \in \mathcal{V}} \exp(\mathbf{u}_k^\top \mathbf{v}_i)}, \tag{14.5.1}$$

burada herhangi i dizini için $\mathbf{v}_i$ ve $\mathbf{u}_i$, sırasıyla merkez sözcük ve bağlam sözcüğü olarak w_i sözcüğünü temsil eden vektörlerdir ve $\mathcal{V} = \{0, 1, \ldots, |\mathcal{V}| - 1\}$ sözcük dağarcığının dizin kümesidir.

Külliyatta birden çok kez oluşabilecek w_i sözcüğünü düşünün. Tüm külliyatta, w_i'nin merkez kelime olarak alındığı tüm bağlam kelimeleri, *aynı elemanın birden çok örneğine izin veren* bir *çoklu küme* $\mathcal{C}_i$ kelime indeksi oluşturur. Herhangi bir öğe için, örnek sayısı onun *çokluğu* olarak adlandırılır. Bir örnekle göstermek için, w_i sözcüğünün iki kez külliyat içinde gerçekleştiğini ve iki bağlam penceresinde merkez sözcük olarak w_i alan bağlam sözcüklerinin indekslerinin k, j, m, k

[194] https://discuss.d2l.ai/t/1335

ve k, l, k, j olduğunu varsayalım. Böylece, j, k, l, m elemanlarının çokluğu sırasıyla 2, 4, 1, 1 olan $\mathcal{C}_i = \{j, j, k, k, k, k, l, m\}$ çoklu kümelidir.

Şimdi x_{ij} olarak çoklu küme $\mathcal{C}_i$'de j elemanının çokluğunu gösterelim. Bu, tüm külliyattaki aynı bağlam penceresindeki w_j (bağlam sözcüğü olarak) ve w_i (orta sözcük olarak) sözcüğünün küresel birlikte bulunma sayısıdır. Bu tür küresel külliyat istatistiklerini kullanarak, skip-gram modelinin kayıp fonksiyonu aşağıdaki ifadeye eşdeğerdir:

$$-\sum_{i\in\mathcal{V}}\sum_{j\in\mathcal{V}} x_{ij}\log\, q_{ij}. \tag{14.5.2}$$

Ayrıca x_i ile $|\mathcal{C}_i|$'e eşdeğer olan w_i'nin merkez sözcük olarak gerçekleştiği bağlam pencerelerindeki tüm bağlam sözcüklerinin sayısını belirtiyoruz. p_{ij}, merkez kelime w_i verildiğinde bağlam kelimesi w_j'yi oluşturmak için koşullu olasılık x_{ij}/x_i olsun, (14.5.2) aşağıdaki gibi yeniden yazılabilir:

$$-\sum_{i\in\mathcal{V}} x_i\sum_{j\in\mathcal{V}} p_{ij}\log\, q_{ij}. \tag{14.5.3}$$

(14.5.3) içinde $-\sum_{j\in\mathcal{V}} p_{ij}\log\, q_{ij}$, küresel külliyat istatistiklerinin p_{ij}'nin koşullu dağılımının çapraz entropisi ve model tahminlerinin q_{ij}'in koşullu dağılımını hesaplar. Yukarıda açıklandığı gibi bu kayıp da x_i tarafından ağırlıklandırılmıştır. (14.5.3) içinde kayıp fonksiyonunun en aza indirilmesi, tahmini koşullu dağılımın küresel külliyat istatistiklerinden koşullu dağılıma yaklaşmasına olanak tanır.

Olasılık dağılımları arasındaki mesafeyi ölçmek için yaygın olarak kullanılsa da, çapraz entropi kayıp fonksiyonu burada iyi bir seçim olmayabilir. Bir yandan, Section 14.2 içinde belirttiğimiz gibi, q_{ij}'i düzgün bir şekilde normalleştirme maliyeti, hesaplama açısından pahalı olabilecek tüm sözcük dağarcığının toplamına neden olur. Öte yandan, büyük bir külliyattan gelen çok sayıda nadir olay genellikle çapraz entropi kaybı ile çok fazla ağırlıkla modellenir.

14.5.2 GloVe Modeli

Bunun ışığında, *GloVe* modeli, skip-gram modelinde (Pennington *et al.*, 2014) kare kaybına dayanarak üç değişiklik yapar:

1. $p'_{ij} = x_{ij}$ ve $q'_{ij} = \exp(\mathbf{u}_j^\top \mathbf{v}_i)$ değişkenlerini kullanın, bunlar olasılık dağılımları değildir ve her ikisinin de logaritmasını alın, bu da kare kayıp terimidir: $\left(\log\, p'_{ij} - \log\, q'_{ij}\right)^2 = \left(\mathbf{u}_j^\top \mathbf{v}_i - \log\, x_{ij}\right)^2$.
2. Her w_i sözcüğü için iki sayıl model parametresi ekleyin: Merkez sözcük ek girdisi b_i'dir ve bağlam sözcüğü ek girdisi c_i'dir.
3. Her kayıp teriminin ağırlığını $h(x_{ij})$ ağırlık fonksiyonu ile değiştirin, burada $h(x)$ $[0, 1]$ aralığında artıyor.

Her şeyi bir araya getirirsek, GloVe eğitimi, aşağıdaki kayıp fonksiyonunu en aza indirmektir:

$$\sum_{i\in\mathcal{V}}\sum_{j\in\mathcal{V}} h(x_{ij})\left(\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j - \log\, x_{ij}\right)^2. \tag{14.5.4}$$

Ağırlık fonksiyonu için önerilen bir seçim şu şekildedir: eğer $x < c$ (örn. $c = 100$) ise $h(x) = (x/c)^\alpha$ (örn. $\alpha = 0.75$); aksi takdirde $h(x) = 1$. Bu durumda, $h(0) = 0$ olduğundan, herhangi

bir $x_{ij} = 0$ için kare kayıp terimi hesaplama verimliliği için atlanabilir. Örneğin, eğitim için minigrup rasgele gradyan inişi kullanırken, her yinelemede gradyanları hesaplamak ve model parametrelerini güncellemek için rasgele *sıfır olmayan* x_{ij} minigrup örnekleriz. Bu sıfır olmayan x_{ij}'lerin önceden hesaplanmış küresel külliyat istatistikleri olduğunu unutmayın; bu nedenle, modele *küresel vektörlerden (Global Vectors)* dolayı Glove denir.

w_i sözcüğü w_j sözcüğünün bağlam penceresinde görünürse, *tersi de geçerlidir*. Bu nedenle, $x_{ij} = x_{ji}$ olur. p_{ij} asimetrik koşullu olasılığa uyan word2vec'in aksine, GloVe simetrik log x_{ij}'a uyar. Bu nedenle, GloVe modelinde herhangi bir sözcüğün merkez sözcük vektörü ve bağlam sözcük vektörü matematiksel olarak eşdeğerdir. Ancak pratikte, farklı ilkleme değerleri nedeniyle, aynı sözcük eğitimden sonra bu iki vektörde yine de farklı değerler alabilir: GloVe bunları çıktı vektörü olarak toplar.

14.5.3 Eş-Oluşum Olasılıklarının Oranından GloVe Yorumlaması

GloVe modelini başka bir bakış açısından da yorumlayabiliriz. Section 14.5.1 içindeki aynı gösterimi kullanarak, $p_{ij} \stackrel{\text{def}}{=} P(w_j \mid w_i)$ külliyatta merkez sözcük olarak w_i elimizdeyken w_j bağlam sözcüğünü üretme koşullu olasılığı olsun. Section 14.5.3, "buz" ve "buhar" sözcükleri verilen çeşitli ortak oluşum olasılıkları ve bunların büyük bir külliyatın istatistiklerine dayanan oranlarını listeler.

w_k=	katı	gaz	su	moda
$p_1 = P(w_k \mid \text{buz})$	0.00019	0.000066	0.003	0.000017
$p_2 = P(w_k \mid \text{buhar})$	0.000022	0.00078	0.0022	0.000018
p_1/p_2	8.9	0.085	1.36	0.96

Table: Kelime-kelime birlikte bulunma olasılıkları ve büyük bir külliyattaki oranları ((Pennington *et al.*, 2014) içindeki Tablo 1'den uyarlanmıştır:)

Aşağıdakileri Section 14.5.3 içinden gözlemleyebiliriz:

- "Buz" ile ilgili ancak "buhar" ile ilgisiz bir w_k sözcüğü, $w_k =$ katı gibi, için birlikte oluşma olasılıklarının daha büyük bir oranda olmasını, 8.9 gibi, bekleriz.
- w_k "buhar" ile ilgili ancak $w_k =$ gaz gibi "buz" ile ilgisiz bir sözcük için, 0.085 gibi birlikte oluşma olasılıklarının daha küçük bir oranda olmasını bekleriz.
- $w_k =$ su gibi "buz" ve "buhar" ile ilgili bir sözcük, w_k, için, 1.36 gibi 1'e yakın bir birlikte oluşma olasılıkları oranı bekleriz.
- $w_k =$ moda gibi "buz" ve "buhar" ile alakasız olan w_k sözcüğü için, 0.96 gibi 1'e yakın bir birlikte oluşma olasılıkları oranı bekleriz.

Birlikte oluşum olasılıklarının oranının sözcükler arasındaki ilişkiyi sezgisel olarak ifade edebileceği görülebilir. Böylece, bu orana uyacak şekilde üç sözcük vektöründen oluşan bir fonksiyon tasarlayabiliriz. w_i merkez kelime ve w_j ve w_k bağlam kelimeleri olmak üzere p_{ij}/p_{ik} birlikte meydana gelme olasılıklarının oranı için, f işlevini kullanarak bu oranı sağlamak istiyoruz:

$$f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) \approx \frac{p_{ij}}{p_{ik}}. \tag{14.5.5}$$

f için birçok olası tasarım arasında, sadece aşağıdakilerden makul bir seçim seçiyoruz. Birlikte oluşum olasılıklarının oranı bir sayıl olduğundan, f'nin $f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) = f\left((\mathbf{u}_j - \mathbf{u}_k)^\top \mathbf{v}_i\right)$ gibi sayıl

bir fonksiyon olmasını isteriz. (14.5.5) içindeki j ve k sözcük endeksleri değiştirince $f(x)f(-x) = 1$'i tutması gerekir, bu nedenle olasılıklardan biri $f(x) = \exp(x)$ olur, yani

$$f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) = \frac{\exp\left(\mathbf{u}_j^\top \mathbf{v}_i\right)}{\exp\left(\mathbf{u}_k^\top \mathbf{v}_i\right)} \approx \frac{p_{ij}}{p_{ik}}. \tag{14.5.6}$$

Şimdi α'nın sabit olduğu $\exp\left(\mathbf{u}_j^\top \mathbf{v}_i\right) \approx \alpha p_{ij}$'yi seçelim. $p_{ij} = x_{ij}/x_i$'den dolayı, her iki tarafta da logaritmayı aldıktan sonra $\mathbf{u}_j^\top \mathbf{v}_i \approx \log\, \alpha + \log\, x_{ij} - \log\, x_i$ elde ediyoruz. $-\log\, \alpha + \log\, x_i$'e uyacak ek girdi terimleri kullanabiliriz, örneğin merkez sözcük ek girdisi b_i ve bağlam sözcüğü ek girdisi c_j:

$$\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j \approx \log\, x_{ij}. \tag{14.5.7}$$

(14.5.7) denklemindeki kare hatası ağırlıklarla ölçülünce, (14.5.4) içindeki GloVe kayıp fonksiyonu elde edilir.

14.5.4 Özet

- Skip-gram modeli, sözcük-sözcük birlikte oluşum sayımları gibi küresel külliyat istatistikleri kullanılarak yorumlanabilir.
- Çapraz entropi kaybı, özellikle büyük bir külliyatında iki olasılık dağılımının farkını ölçmek için iyi bir seçim olmayabilir. GloVe, önceden hesaplanmış küresel külliyat istatistiklerine uymak için kare kaybını kullanır.
- Merkez sözcük vektörü ve bağlam sözcük vektörü, Glove'deki herhangi bir sözcük için matematiksel olarak eşdeğerdir.
- GloVe sözcük-sözcük birlikte oluşum olasılıklarının oranından yorumlanabilir.

14.5.5 Alıştırmalar

1. w_i ve w_j sözcükleri aynı bağlam penceresinde birlikte ortaya çıkarsa, koşullu olasılığın, p_{ij}, hesaplama yöntemini yeniden tasarlamak için metin dizisindeki mesafelerini nasıl kullanabiliriz? İpucu: GloVe makalesindeki 4.2. kısmına bakınız (Pennington *et al.*, 2014).
2. Herhangi bir sözcük için, onun merkez sözcük ek girdisi ve bağlam sözcük ek girdisi Glove'da matematiksel olarak eşdeğer midir? Neden?

Tartışmalar[195]

14.6 Sözcük Benzerliği ve Benzeşim

Section 14.4 içinde, küçük bir veri kümesi üzerinde bir word2vec modelini eğittik ve bunu bir girdi sözcüğü için anlamsal olarak benzer kelimeleri bulmak için uyguladık. Uygulamada, büyük külliyatlar üzerinde önceden eğitilmiş kelime vektörleri, daha sonra Chapter 15 içinde ele alınacak olan doğal dil işleme görevlerine uygulanabilir. Büyük külliyatlardan önceden eğitilmiş kelime vektörlerinin anlamını basit bir şekilde göstermek için, bunları sözcük benzerliği ve benzeşimi görevlerine uygulayalım.

[195] https://discuss.d2l.ai/t/385

```
import os
import torch
from torch import nn
from d2l import torch as d2l
```

14.6.1 Önceden Eğitilmiş Sözcük Vektörlerini Yükleme

Aşağıda, GloVe web sitesinden[196] indirilebilen 50, 100 ve 300 boyutlarında önceden eğitilmiş GloVe gömme listeleri verilmektedir. Önceden eğitilmiş fastText gömmeleri birden çok dilde mevcuttur. Burada indirilebilen bir İngilizce sürümünü düşünelim (300-boyutlu "wiki.en") fastText websitesi[197].

```
#@save
d2l.DATA_HUB['glove.6b.50d'] = (d2l.DATA_URL + 'glove.6B.50d.zip',
                                '0b8703943ccdb6eb788e6f091b8946e82231bc4d')

#@save
d2l.DATA_HUB['glove.6b.100d'] = (d2l.DATA_URL + 'glove.6B.100d.zip',
                                 'cd43bfb07e44e6f27cbcc7bc9ae3d80284fdaf5a')

#@save
d2l.DATA_HUB['glove.42b.300d'] = (d2l.DATA_URL + 'glove.42B.300d.zip',
                                  'b5116e234e9eb9076672cfeabf5469f3eec904fa')

#@save
d2l.DATA_HUB['wiki.en'] = (d2l.DATA_URL + 'wiki.en.zip',
                           'c1816da3821ae9f43899be655002f6c723e91b88')
```

Bu önceden eğitilmiş GloVe ve fastText gömmelerini yüklemek için aşağıdaki `TokenEmbedding` sınıfını tanımlıyoruz.

```
#@save
class TokenEmbedding:
    """Andıç Gömme."""
    def __init__(self, embedding_name):
        self.idx_to_token, self.idx_to_vec = self._load_embedding(
            embedding_name)
        self.unknown_idx = 0
        self.token_to_idx = {token: idx for idx, token in
                             enumerate(self.idx_to_token)}

    def _load_embedding(self, embedding_name):
        idx_to_token, idx_to_vec = ['<unk>'], []
        data_dir = d2l.download_extract(embedding_name)
        # GloVe websitesi: https://nlp.stanford.edu/projects/glove/
        # fastText websitesi: https://fasttext.cc/
        with open(os.path.join(data_dir, 'vec.txt'), 'r') as f:
            for line in f:
                elems = line.rstrip().split(' ')
                token, elems = elems[0], [float(elem) for elem in elems[1:]]
```

(continues on next page)

[196] https://nlp.stanford.edu/projects/glove/
[197] https://fasttext.cc/

(continued from previous page)

```
                # fastText'teki en üst satır gibi başlık bilgilerini atla
                if len(elems) > 1:
                    idx_to_token.append(token)
                    idx_to_vec.append(elems)
        idx_to_vec = [[0] * len(idx_to_vec[0])] + idx_to_vec
        return idx_to_token, torch.tensor(idx_to_vec)

    def __getitem__(self, tokens):
        indices = [self.token_to_idx.get(token, self.unknown_idx)
                   for token in tokens]
        vecs = self.idx_to_vec[torch.tensor(indices)]
        return vecs

    def __len__(self):
        return len(self.idx_to_token)
```

Aşağıda 50 boyutlu GloVe gömmelerini yüklüyoruz (Wikipedia alt kümesi üzerinde önceden eğitilmiş). `TokenEmbedding` örneğini oluştururken, belirtilen gömme dosyasının henüz yoksa indirilmesi gerekir.

```
glove_6b50d = TokenEmbedding('glove.6b.50d')
```

Sözcük dağarcığı boyutunu çıktılayın. Sözcük bilgisi 400000 kelime (belirteç) ve özel bir bilinmeyen belirteci içerir.

```
len(glove_6b50d)
```

```
400001
```

Sözcük hazinesinde bir kelimenin indeksini alabiliriz ve tersi de geçerlidir.

```
glove_6b50d.token_to_idx['beautiful'], glove_6b50d.idx_to_token[3367]
```

```
(3367, 'beautiful')
```

14.6.2 Önceden Eğitilmiş Sözcük Vektörlerini Uygulama

Yüklenen GloVe vektörlerini kullanarak, anlamlarını aşağıdak sözcük benzerliği ve benzeşim görevlerine uygulayarak göstereceğiz.

Sözcük Benzerliği

Section 14.4.3 içindekine benzer şekilde, sözcük vektörleri arasındaki kosinüs benzerliklerine dayanan bir girdi sözcüğü için anlamsal olarak benzer kelimeleri bulmak için aşağıdaki knn (k en yakın komşu) işlevini uygularız.

```
def knn(W, x, k):
    # Sayısal kararlılık için 1e-9 ekleyin
    cos = torch.mv(W, x.reshape(-1,)) / (
        torch.sqrt(torch.sum(W * W, axis=1) + 1e-9) *
        torch.sqrt((x * x).sum()))
    _, topk = torch.topk(cos, k=k)
    return topk, [cos[int(i)] for i in topk]
```

Daha sonra, `TokenEmbedding` örneğinden önceden eğitilmiş sözcük vektörlerini kullanarak benzer kelimeleri ararız.

```
def get_similar_tokens(query_token, k, embed):
    topk, cos = knn(embed.idx_to_vec, embed[[query_token]], k + 1)
    for i, c in zip(topk[1:], cos[1:]):  # Girdi sözcüğünü hariç tut
        print(f'cosine sim={float(c):.3f}: {embed.idx_to_token[int(i)]}')
```

`glove_6b50d`'teki önceden eğitilmiş kelime vektörlerinin kelime dağarcığı 400000 kelime ve özel bir bilinmeyen belirteç içerir. Girdi kelimesi ve bilinmeyen belirteci hariç, bu kelime arasında "chip (çip))" kelimesine en anlamsal olarak benzer üç kelimeyi bulalım.

```
get_similar_tokens('chip', 3, glove_6b50d)
```

```
cosine sim=0.856: chips
cosine sim=0.749: intel
cosine sim=0.749: electronics
```

Aşağıda "baby (bebek)" ve " beautiful (güzel)" kelimelerinin benzerleri çıktılanır.

```
get_similar_tokens('baby', 3, glove_6b50d)
```

```
cosine sim=0.839: babies
cosine sim=0.800: boy
cosine sim=0.792: girl
```

```
get_similar_tokens('beautiful', 3, glove_6b50d)
```

```
cosine sim=0.921: lovely
cosine sim=0.893: gorgeous
cosine sim=0.830: wonderful
```

Kelime Benzeşimi

Benzer kelimeleri bulmanın yanı sıra, kelime vektörlerini kelime benzetme görevlerine de uygulayabiliriz. Örneğin, "erkek":"kadın"::"oğul":"kız" bir kelime benzetme şeklidir: "erkek" ile "kadın", "kız" ile "oğul" benzerdir. Daha özel olarak, kelime benzeşim tamamlama görevi şu şekilde tanımlanabilir: Bir kelime benzeşimi $a : b :: c : d$ için, ilk üç kelime olan a, b ve c verildiğinde, d bulunsun. w kelimesinin vektörünü $\text{vec}(w)$ ile belirtilsin. Benzetmeyi tamamlamak için, vektörü $\text{vec}(c) + \text{vec}(b) - \text{vec}(a)$ sonucuna en çok benzeyen kelimeyi bulacağız.

```
def get_analogy(token_a, token_b, token_c, embed):
    vecs = embed[[token_a, token_b, token_c]]
    x = vecs[1] - vecs[0] + vecs[2]
    topk, cos = knn(embed.idx_to_vec, x, 1)
    return embed.idx_to_token[int(topk[0])]  # Bilinmeyen kelimeleri çıkar
```

Yüklü kelime vektörlerini kullanarak "erkek-kadın" benzeşimini doğrulayalım.

```
get_analogy('man', 'woman', 'son', glove_6b50d)
```

```
'daughter'
```

Aşağıdaki bir "başkent-ülke" benzeşimini tamamlar: "Pekin":"Çin"::"Tokyo":"Japonya". Bu, önceden eğitilmiş kelime vektörlerindeki anlamı gösterir.

```
get_analogy('beijing', 'china', 'tokyo', glove_6b50d)
```

```
'japan'
```

"Kötü":"en kötü"::"büyük":"en büyük" gibi "sıfat-üstün sıfat" benzetme için, önceden eğitilmiş kelime vektörlerinin sözdizimsel bilgileri yakalayabileceğini görebiliriz.

```
get_analogy('bad', 'worst', 'big', glove_6b50d)
```

```
'biggest'
```

Önceden eğitilmiş kelime vektörlerinde geçmiş zaman kavramını göstermek için, sözdizimini "şimdiki zaman - geçmiş zaman" benzetmesini kullanarak test edebiliriz: "yap (do)":"yaptım (did)"::"go (git)":"went (gittim)".

```
get_analogy('do', 'did', 'go', glove_6b50d)
```

```
'went'
```

14.6.3 Özet

- Uygulamada, büyük külliyatlar üzerinde önceden eğitilmiş kelime vektörleri doğal dil işleme görevlerine uygulanabilir.
- Önceden eğitilmiş kelime vektörleri kelime benzerliği ve benzeşimi görevlerine uygulanabilir.

14.6.4 Alıştırmalar

1. `TokenEmbedding('wiki.en')` kullanarak fastText sonuçlarını test edin.
2. Kelime dağarcığı son derece büyük olduğunda, benzer kelimeleri nasıl bulabiliriz veya bir kelime benzeşimini daha hızlı tamamlayabiliriz?

Tartışmalar[198]

14.7 Dönüştürücülerden Çift Yönlü Kodlayıcı Temsiller (BERT)

Doğal dil anlama için çeşitli kelime gömme modelleri tanıttık. Ön eğitim sonrasında çıktı, her satırın önceden tanımlanmış bir sözcük dağarcığını temsil eden bir vektör olduğu bir matris olarak düşünülebilir. Aslında, bu kelime gömme modellerinin tümü *bağlam bağımsız*dır. Bu özelliği göstererek başlayalım.

14.7.1 Bağlam Bağımsızlıktan Bağlam Duyarlılığına

Section 14.4 ve Section 14.6 içindeki deneyleri hatırlayın. Örneğin, word2vec ve GloVe her ikisi de kelimenin bağlamından bağımsız olarak (varsa) aynı önceden eğitilmiş vektörünü aynı sözcüğe atar. Resmi olarak, herhangi bir belirteç x'in bağlamdan bağımsız gösterimi, yalnızca x'i girdisi olarak alan bir $f(x)$ işlevidir. Doğal dillerde çokanlamlılık ve karmaşık anlambilim bolluğu göz önüne alındığında, bağlamdan bağımsız temsillerin bariz sınırlamaları vardır. Örneğin, "topu ateşledi" ve "topa vurdu" bağlamlarındaki "top" kelimesinin tamamen farklı anlamları vardır; bu nedenle, aynı kelimeye bağlamlara bağlı olarak farklı temsiller atanabilir.

Bu, kelimelerin temsillerinin bağlamlarına bağlı olduğu *bağlam duyarlı* kelime temsillerinin gelişimini teşvik eder. Bu nedenle, x belirtecinin bağlam duyarlı bir temsili, hem x'e hem de onun bağlamı $c(x)$'e bağlı olan bir $f(x, c(x))$ işlevidir. Yaygın bağlam duyarlı temsiller arasında TagLM (dil-model-artırılmış dizi etiketleyici) (Peters *et al.*, 2017), CoVe (Context Vectors - Bağlam Vektörleri) (McCann *et al.*, 2017) ve ELMo (Embeddings from Language Models - Dil Modellerinden Gömmeler) (Peters *et al.*, 2018) bulunur.

Örneğin, tüm diziyi girdi olarak alarak, ELMo, girdi dizisinden her sözcüğe bir temsil atan bir işlevdir. Özellikle, ELMo, önceden eğitilmiş çift yönlü LSTM'den tüm ara katman temsillerini çıktı temsili olarak birleştirir. Daha sonra ELMo temsili, mevcut modelde ELMo temsilini ve belirteçlerin orijinal temsilini (örneğin, Glove) bitiştirmek gibi ek özniteliklerle bir aşağı akış görevinin mevcut gözetimli modeline eklenecektir. Bir yandan, önceden eğitilmiş çift yönlü LSTM modelindeki tüm ağırlıklar ELMo temsilleri toplandıktan sonra dondurulur. Öte yandan, mevcut gözetimli model belirli bir görev için ayrıca özelleştirilmiştir. O dönemde farklı görevler için farklı en iyi modellerden yararlanarak ELMo, altı doğal dil işleme görevinde son teknolojiyi iyileştirdi: Duygu

[198] https://discuss.d2l.ai/t/1336

analizi, doğal dil çıkarımı, anlamsal rol etiketleme, referans çözünürlüğü, adlandırılmış varlık tanıma ve soru yanıtlama.

14.7.2 Göreve Özgülükten Görevden Bağımsızlığa

ELMo, çeşitli doğal dil işleme görevleri kümesine yönelik çözümleri önemli ölçüde geliştirmiş olsa da, her çözüm hala *göreve özgü* mimariye dayanıyor. Bununla birlikte, her doğal dil işleme görevi için belirli bir mimari oluşturmak pratik olarak aşikar değildir. GPT (Generative Pre-Training - Üretici Ön İşleme) modeli, bağlama duyarlı gösterimler (Radford *et al.*, 2018) için genel bir *görev bağımsız* modeli tasarlama çabasını temsil eder. Bir dönüştürücü kod çözücü üzerine inşa edilen GPT, metin dizilerini temsil etmek için kullanılacak bir dil modelini ön eğitir. Bir aşağı akış göreve GPT uygulanırken, dil modelinin çıktısı görevin etiketini tahmin etmek için ek bir doğrusal çıktı katmanına beslenir. Önceden eğitilmiş modelin parametrelerini donduran ELMo ile keskin bir zıtlıkla, GPT aşağı akış görevinin gözetimli öğrenmesi sırasında önceden eğitilmiş dönüştürücü kod çözücüsündeki parametrelerin *tümünü* ince ayarlar. GPT, doğal dil çıkarımı, soru cevaplama, cümle benzerliği ve sınıflandırma gibi on iki görev üzerinde değerlendirildi ve model mimarisinde minimum değişikliklerle dokuzunda son teknolojiyi iyileştirdi.

Bununla birlikte, dil modellerinin özbağlanımlı doğası nedeniyle, GPT sadece ileriye bakar (soldan sağa). "Banka" solundaki bağlamda duyarlı olduğu için "Para yatırmak için bankaya gittim" ve "Oturmak için banka gittim" bağlamında, farklı anlamlara sahip olsa da, GPT "banka" için aynı temsili geri dönecektir.

14.7.3 BERT: Her İki Dünyanın En İyilerini Birleştirme

Gördüğümüz gibi, ELMo bağlamı iki yönlü olarak kodlar, ancak göreve özgü mimarileri kullanır; GPT ise görevden bağımsızdır ancak bağlamı soldan sağa kodlar. Her iki dünyanın en iyilerini bir araya getiren BERT (Bidirectional Encoder Representations from Transformers - Dönüştürücülerden Çift Yönlü Kodlayıcı Temsilleri) bağlamı çift yönlü olarak kodlar (Devlin *et al.*, 2018) ve çeşitli doğal dil işleme görevleri için minimum mimari değişiklikleri gerektirir. Önceden eğitilmiş bir dönüştürücü kodlayıcısı kullanarak BERT, çift yönlü bağlamına dayalı herhangi bir belirteci temsil edebilir. Aşağı akış görevlerinin gözetimli öğrenmesi sırasında BERT iki açıdan GPT'ye benzer. İlk olarak, BERT temsilleri, her belirteç için tahmin etmek gibi görevlerin niteliğine bağlı olarak model mimarisinde minimum değişikliklerle birlikte eklenen bir çıktı katmanına beslenecektir. İkinci olarak, önceden eğitilmiş dönüştürücü kodlayıcının tüm parametreleri ince ayarlanırken, ek çıktı katmanı sıfırdan eğitilecektir. Fig. 14.7.1, ELMo, GPT ve BERT arasındaki farkları tasvir eder.

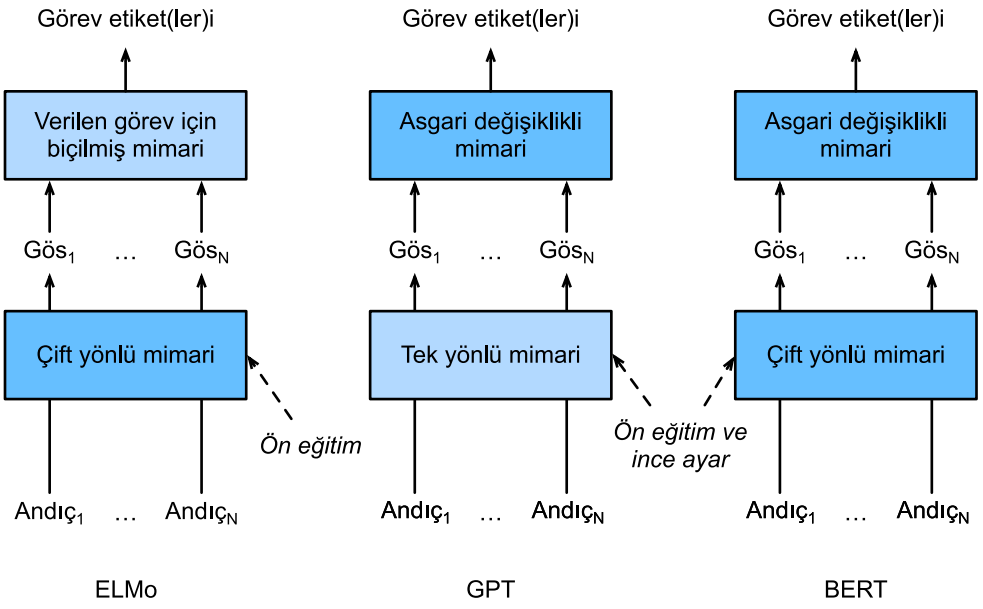


Fig. 14.7.1: ELMo, GPT, ve BERT karşılaştırması.

BERT, (i) tek metin sınıflandırması (örn. duygu analizi), (ii) metin çifti sınıflandırması (örneğin, doğal dil çıkarımı), (iii) soru yanıtlama, (iv) metin etiketleme (örneğin, adlandırılmış varlık tanıma) genel kategorilerinde on bir doğal dil işleme görevinde son teknolojiyi iyileştirdi. 2018 yılında, bağlam duyarlı ELMo'dan görev bağımsız GPT ve BERT'e kadar, doğal diller için derin temsillerin kavramsal olarak basit ama deneysel olarak güçlü ön eğitimi, çeşitli doğal dil işleme görevlerindeki çözümlerde devrim yaratmıştır.

Bu bölümün geri kalanında BERT ön eğitimine dalacağız. Doğal dil işleme uygulamaları Chapter 15 içinde açıklandığında, aşağı akış uygulamaları için BERT ince ayarını göstereceğiz.

```
import torch
from torch import nn
from d2l import torch as d2l
```

14.7.4 Girdi Temsili

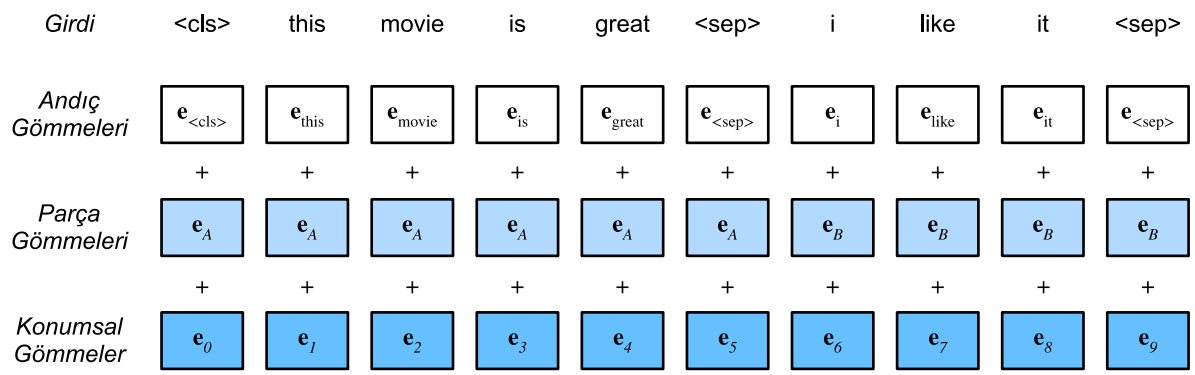
Doğal dil işlemede, bazı görevler (örn. duygu analizi) girdi olarak tek bir metin alırken, diğer bazı görevlerde (örneğin, doğal dil çıkarımı) girdi bir çift metin dizisidir. BERT girdi dizisi, hem tek metin hem de metin çiftlerini açıkça temsil eder. Birincisinde, BERT girdi dizisi özel sınıflandırma belirteci "<cls>", bir metin dizisinin belirteçleri ve "<sep>" özel ayırma belirteci bitiştirilmedir. İkincisinde, BERT girdi dizisi "<cls>", ilk metin dizisinin belirteçleri, "<sep>", ikinci metin dizisinin belirteçleri ve "<sep>" bitiştirilmedir. "BERT girdi dizisi" terminolojisini diğer "diziler" türlerinden sürekli olarak ayırt edeceğiz. Örneğin, bir *BERT girdi dizisi*, bir *metin dizisi* veya iki *metin dizisi* içerebilir.

Metin çiftlerini ayırt etmek için, öğrenilmiş bölüm gömmeleri $\mathbf{e}_A$ ve $\mathbf{e}_B$ sırasıyla birinci dizinin ve ikinci dizinin belirteç gömmelerine eklenir. Tek metin girdileri için sadece $\mathbf{e}_A$ kullanılır.

Aşağıdaki `get_tokens_and_segments` girdi olarak bir veya iki cümle alır, daha sonra BERT girdi dizisinin belirteçlerini ve bunlara karşılık gelen bölüm kimliklerini döndürür.

```
#@save
def get_tokens_and_segments(tokens_a, tokens_b=None):
    """BERT girdi dizisinin belirteçlerini ve bunların bölüm kimliklerini alın."""
    tokens = ['<cls>'] + tokens_a + ['<sep>']
    # 0 and 1 are marking segment A and B, respectively
    segments = [0] * (len(tokens_a) + 2)
    if tokens_b is not None:
        tokens += tokens_b + ['<sep>']
        segments += [1] * (len(tokens_b) + 1)
    return tokens, segments
```

BERT, çift yönlü mimari olarak dönüştürücü kodlayıcıyı seçer. Dönüştürücü kodlayıcısında yaygın olarak, BERT girdi dizisinin her pozisyonuna konumsal gömme eklenir. Ancak, orijinal dönüştürücü kodlayıcısından farklı olan BERT, *öğrenilebilen* konumsal gömme kullanır. Özetlemek gerekirse, `fig_bert-input`, BERT girdi dizisinin gömmelerinin, belirteç gömmelerinin, bölüm gömmelerinin ve konum gömmelerinin toplamı olduğunu gösterir.



.. _fig_bert-input:

Aşağıdaki `BERTEncoder` sınıfı Section 10.7 içinde uygulanan gibi `TransformerEncoder` sınıfına benzer. `TransformerEncoder`'dan farklı olan `BERTEncoder`, bölüm gömme ve öğrenilebilir konumsal gömme parçaları kullanır.

```
#@save
class BERTEncoder(nn.Module):
    """BERT kodlayıcı."""
    def __init__(self, vocab_size, num_hiddens, norm_shape, ffn_num_input,
                 ffn_num_hiddens, num_heads, num_layers, dropout,
                 max_len=1000, key_size=768, query_size=768, value_size=768,
                 **kwargs):
        super(BERTEncoder, self).__init__(**kwargs)
        self.token_embedding = nn.Embedding(vocab_size, num_hiddens)
        self.segment_embedding = nn.Embedding(2, num_hiddens)
        self.blks = nn.Sequential()
        for i in range(num_layers):
            self.blks.add_module(f"{i}", d2l.EncoderBlock(
                key_size, query_size, value_size, num_hiddens, norm_shape,
                ffn_num_input, ffn_num_hiddens, num_heads, dropout, True))
        # BERT'te konumsal gömmeler öğrenilebilir, bu nedenle yeterince
        # uzun bir konumsal gömme parametresi oluşturuyoruz
```

(continues on next page)

(continued from previous page)

```
        self.pos_embedding = nn.Parameter(torch.randn(1, max_len,
                                                      num_hiddens))

    def forward(self, tokens, segments, valid_lens):
        # Aşağıdaki kod parçacığında `X`'in şekli değişmeden kalır:
        # (parti boyutu, maksimum dizi uzunluğu, `num_hiddens`)
        X = self.token_embedding(tokens) + self.segment_embedding(segments)
        X = X + self.pos_embedding.data[:, :X.shape[1], :]
        for blk in self.blks:
            X = blk(X, valid_lens)
        return X
```

Kelime dağarcığının 10000 olduğunu varsayalım. `BERTEncoder`'in ileri çıkarımını göstermek için, bunun bir örneğini oluşturalım ve parametrelerini ilkleyelim.

```
vocab_size, num_hiddens, ffn_num_hiddens, num_heads = 10000, 768, 1024, 4
norm_shape, ffn_num_input, num_layers, dropout = [768], 768, 2, 0.2
encoder = BERTEncoder(vocab_size, num_hiddens, norm_shape, ffn_num_input,
                      ffn_num_hiddens, num_heads, num_layers, dropout)
```

`tokens`'ı, her bir belirtecin kelime dağarcığının bir indeksi olduğu, uzunluğu 8 olan 2 BERT girdi dizisi olarak tanımlarız. `BERTEncoder`'ın girdi belirteçleri (`tokens`) ile ileri çıkarımı, her belirtecin, uzunluğu `num_hiddens` hiper parametresi tarafından önceden tanımlanmış bir vektör tarafından temsil edildiği kodlanmış sonucu döndürür. Bu hiper parametre genellikle dönüştürücü kodlayıcının *gizli boyutu* (gizli birim sayısı) olarak adlandırılır.

```
tokens = torch.randint(0, vocab_size, (2, 8))
segments = torch.tensor([[0, 0, 0, 0, 1, 1, 1, 1], [0, 0, 0, 1, 1, 1, 1, 1]])
encoded_X = encoder(tokens, segments, None)
encoded_X.shape
```

```
torch.Size([2, 8, 768])
```

14.7.5 Ön Eğitim Görevleri

`BERTEncoder`'ın ileri çıkarımı, girdi metninin her belirteci ve eklenen özel "<cls>" ve "<seq>" belirteçlerinin BERT temsilini verir. Ardından, BERT ön eğitimi için kayıp fonksiyonunu hesaplamak için bu temsilleri kullanacağız. Ön eğitim aşağıdaki iki görevden oluşur: Maskeli dil modelleme ve sonraki cümle tahmini.

Maskeli Dil Modelleme

Section 8.3 içinde gösterildiği gibi, bir dil modeli, sol tarafındaki bağlamı kullanarak bir belirteci öngörür. Her belirteci temsil ederken bağlamı iki yönlü olarak kodlamak için BERT, belirteçleri rastgele maskeler ve maskelenmiş belirteçleri öz gözetimli bir şekilde tahmin etmek için çift yönlü bağlamdaki belirteçleri kullanır. Bu görev *maskeli dil modeli* olarak adlandırılır.

Bu ön eğitim görevinde, belirteçlerin %15'i tahmin için maskelenmiş belirteçler olarak rastgele seçilecektir. Etiketini kullanarak hile yapmadan maskelenmiş bir belirteci tahmin etmek için, basit bir yaklaşım, belirteci her zaman BERT girdi dizisinde özel bir "<mask>" belirteci ile değiştirmektir. Bununla birlikte, yapay özel belirteç "<mask>" asla ince ayarda görünmeyecektir. Ön eğitim ve ince ayar arasında böyle bir uyuşmazlığı önlemek için, eğer bir belirteç tahmin için maskelenmişse (örneğin, "harika" maskelenecek ve "bu film harika" olarak tahmin edilmek üzere seçilir), girdide şu şekilde değiştirilir:

- Zamanın %80'i için özel bir "<mask>" belirteci ile (örneğin, "bu film harika", "bu film <mask>" olur);
- Zamanın %10'u için rastgele bir belirteç (örneğin, "bu film harika" "bu film içki" olur);
- Zamanın %10'unda değişmeyen etiket belirteci (örneğin, "bu film harika" "bu film harika" kalır).

%15'lik zamanın %10'u için rastgele bir belirteç eklendiğini unutmayın. Bu ara sıra olan gürültü, BERT'i, çift yönlü bağlam kodlamasında maskelenmiş belirtece (özellikle etiket belirteci değişmeden kaldığında) karşı daha az yanlı olmasını teşvik eder.

BERT ön eğitiminin maskelenmiş dil modeli görevinde maskeli belirteçleri tahmin etmek için aşağıdaki `MaskLM` sınıfını uyguluyoruz. Tahmin, tek gizli katmanlı bir MLP kullanır (`self.mlp`). İleri çıkarımda, iki girdi gerekir: `BERTEncoder`'ının kodlanmış sonucu ve tahmin için belirteç konumları. Çıktı, bu pozisyonlardaki tahmin sonuçlarıdır.

```
#@save
class MaskLM(nn.Module):
    """BERT'in maskeli dil modeli görevi."""
    def __init__(self, vocab_size, num_hiddens, num_inputs=768, **kwargs):
        super(MaskLM, self).__init__(**kwargs)
        self.mlp = nn.Sequential(nn.Linear(num_inputs, num_hiddens),
                                 nn.ReLU(),
                                 nn.LayerNorm(num_hiddens),
                                 nn.Linear(num_hiddens, vocab_size))

    def forward(self, X, pred_positions):
        num_pred_positions = pred_positions.shape[1]
        pred_positions = pred_positions.reshape(-1)
        batch_size = X.shape[0]
        batch_idx = torch.arange(0, batch_size)
        # Varsayalım ki `batch_size` = 2, `num_pred_positions` = 3 olsun,
        # o halde `batch_idx` `np.array([0, 0, 0, 1, 1, 1])` olur
        batch_idx = torch.repeat_interleave(batch_idx, num_pred_positions)
        masked_X = X[batch_idx, pred_positions]
        masked_X = masked_X.reshape((batch_size, num_pred_positions, -1))
        mlm_Y_hat = self.mlp(masked_X)
        return mlm_Y_hat
```

`MaskLM`'in ileri çıkarımını göstermek için, `mlm` örneğini oluşturup ilkleriz. `BERTEncoder`'in ileri

çıkarımından encoded_X'in 2 BERT girdi dizisini temsil ettiğini hatırlayın. mlm_positions'i encoded_X BERT girdi dizisinde tahmin etmek için 3 endeksli olarak tanımlarız. mlm'nin ileri çıkarımı, encoded_X'in tüm mlm_positions maskeli konumlarında mlm_Y_hat tahmin sonuçlarını döndürür. Her tahmin için, sonucun boyutu kelime dağarcığına eşittir.

```
mlm = MaskLM(vocab_size, num_hiddens)
mlm_positions = torch.tensor([[1, 5, 2], [6, 1, 5]])
mlm_Y_hat = mlm(encoded_X, mlm_positions)
mlm_Y_hat.shape
```

```
torch.Size([2, 3, 10000])
```

Maskeler altında tahmin edilen mlm_Y_hat belirteçlerinin mlm_Y gerçek referans değer etiketleri ile, BERT ön eğitiminde maskeli dil modeli görevinin çapraz entropi kaybını hesaplayabiliriz.

```
mlm_Y = torch.tensor([[7, 8, 9], [10, 20, 30]])
loss = nn.CrossEntropyLoss(reduction='none')
mlm_l = loss(mlm_Y_hat.reshape((-1, vocab_size)), mlm_Y.reshape(-1))
mlm_l.shape
```

```
torch.Size([6])
```

Sonraki Cümle Tahmini

Maskelenmiş dil modellemesi sözcükleri temsil etmek için çift yönlü bağlamı kodlayabilse de, metin çiftleri arasındaki mantıksal ilişkiyi açıkça modellemez. BERT, iki metin dizisi arasındaki ilişkiyi anlamaya yardımcı olmak için, ön eğitimde ikili sınıflandırma görevi, *sonraki cümle tahmini* olarak değerlendirir. Ön eğitim için cümle çiftleri oluştururken, çoğu zaman "Doğru" (True) etiketi ile ardışık cümlelerdir; zamanın diğer yarısı için ikinci cümle rastgele "Yanlış" (False) etiketi ile külliyattan örneklenir.

Aşağıdaki NextSentencePred sınıfı, ikinci cümlenin BERT girdi dizisindeki ilk cümlenin bir sonraki cümle olup olmadığını tahmin etmek için tek gizli katmanlı bir MLP kullanır. Dönüştürücü kodlayıcısındaki öz-dikkat nedeniyle, "<cls>" özel belirtecinin BERT temsili, her iki cümleyi de girdiden kodlar. Bu nedenle, MLP sınıflandırıcısının çıktı katmanı (self.output) girdi olarak X'i alır, burada X, girdisi kodlanmış "<cls>" belirteci olan MLP gizli katmanının çıktısıdır.

```
#@save
class NextSentencePred(nn.Module):
    """BERT'in sonraki cümle tahmini görevi."""
    def __init__(self, num_inputs, **kwargs):
        super(NextSentencePred, self).__init__(**kwargs)
        self.output = nn.Linear(num_inputs, 2)

    def forward(self, X):
        # `X` shape: (batch size, `num_hiddens`)
        return self.output(X)
```

Bir NextSentencePred örneğinin ileri çıkarımının her BERT girdi dizisi için ikili tahminleri döndürdüğünü görebiliriz.

```
# PyTorch by default won't flatten the tensor as seen in mxnet where, if
# flatten=True, all but the first axis of input data are collapsed together
encoded_X = torch.flatten(encoded_X, start_dim=1)
# input_shape for NSP: (batch size, `num_hiddens`)
nsp = NextSentencePred(encoded_X.shape[-1])
nsp_Y_hat = nsp(encoded_X)
nsp_Y_hat.shape
```

```
torch.Size([2, 2])
```

2 tane ikili sınıflandırmanın çapraz entropi kaybı da hesaplanabilir.

```
nsp_y = torch.tensor([0, 1])
nsp_l = loss(nsp_Y_hat, nsp_y)
nsp_l.shape
```

```
torch.Size([2])
```

Yukarıda bahsedilen ön eğitim görevlerindeki tüm etiketlerin elle etiketleme çabası olmaksızın ön eğitim külliyattından zahmetsiz olarak elde edilebileceği dikkat çekicidir. Orijinal BERT, BookCorpus (Zhu *et al.*, 2015) ve İngilizce Wikipedia'nin bitiştirilmesi üzerinde ön eğitilmiştir. Bu iki metin külliyatı çok büyüktür: Sırasıyla 800 milyon kelime ve 2.5 milyar kelime vardır.

14.7.6 Her Şeyleri Bir Araya Getirmek

BERT ön eğitimi yaparken, nihai kayıp fonksiyonu hem maskeli dil modelleme hem de sonraki cümle tahmini için kayıp fonksiyonlarının doğrusal bir kombinasyonudur. Artık `BERTModel` sınıfını `BERTEncoder`, `MaskLM` ve `NextSentencePred`'in üç sınıfını örnekleyerek tanımlayabiliriz. İleri çıkarım kodlanmış BERT temsilleri `encoded_X`, maskeli dil modelleme tahminleri `mlm_Y_hat` ve sonraki cümle tahminleri `nsp_Y_hat` döndürür.

```
#@save
class BERTModel(nn.Module):
    """BERT modeli."""
    def __init__(self, vocab_size, num_hiddens, norm_shape, ffn_num_input,
                 ffn_num_hiddens, num_heads, num_layers, dropout,
                 max_len=1000, key_size=768, query_size=768, value_size=768,
                 hid_in_features=768, mlm_in_features=768,
                 nsp_in_features=768):
        super(BERTModel, self).__init__()
        self.encoder = BERTEncoder(vocab_size, num_hiddens, norm_shape,
                    ffn_num_input, ffn_num_hiddens, num_heads, num_layers,
                    dropout, max_len=max_len, key_size=key_size,
                    query_size=query_size, value_size=value_size)
        self.hidden = nn.Sequential(nn.Linear(hid_in_features, num_hiddens),
                                    nn.Tanh())
        self.mlm = MaskLM(vocab_size, num_hiddens, mlm_in_features)
        self.nsp = NextSentencePred(nsp_in_features)

    def forward(self, tokens, segments, valid_lens=None, pred_positions=None):
```

(continues on next page)

(continued from previous page)

```
        encoded_X = self.encoder(tokens, segments, valid_lens)
        if pred_positions is not None:
            mlm_Y_hat = self.mlm(encoded_X, pred_positions)
        else:
            mlm_Y_hat = None
        # Bir sonraki cümle tahmini için MLP sınıflandırıcısının gizli katmanı.
        # 0, '<cls>' belirtecinin dizinidir
        nsp_Y_hat = self.nsp(self.hidden(encoded_X[:, 0, :]))
        return encoded_X, mlm_Y_hat, nsp_Y_hat
```

14.7.7 Özet

- Word2vec ve GloVe gibi sözcük gömme modelleri bağlam bağımsızdır. Aynı önceden eğitilmiş vektörü, kelimenin bağlamından bağımsız olarak (varsa) aynı sözcüğe atar. Doğal dillerde çokanlamlılık veya karmaşık anlamları iyi ele almaları zordur.
- ELMo ve GPT gibi bağlam duyarlı sözcük temsilleri için, sözcüklerin temsilleri bağlamlarına bağlıdır.
- ELMo bağlamı iki yönlü olarak kodlar ancak göreve özgü mimarileri kullanır (ancak, her doğal dil işleme görevi için belirli bir mimariyi oluşturmak pratik olarak aşikar değildir); GPT göreve özgüdür ancak bağlamı soldan sağa kodlar.
- BERT her iki dünyanın en iyilerini birleştirir: Bağlamı çift yönlü olarak kodlar ve bir çok çeşitli doğal dil işleme görevleri için asgari mimari değişiklikleri gerektirir.
- BERT girdi dizisinin gömmeleri belirteç gömme, bölüm gömme ve konumsal gömme toplamıdır.
- BERT ön eğitim iki görevden oluşur: Maskeli dil modelleme ve sonraki cümle tahmini. Birincisi, sözcükleri temsil etmek için çift yönlü bağlamı kodlayabilirken, ikincisi ise metin çiftleri arasındaki mantıksal ilişkiyi açıkça modeller.

14.7.8 Alıştırmalar

1. BERT neden başarılıdır?
2. Diğer tüm şeyler eşit olunca, maskeli bir dil modeli, soldan sağa dil modelinden daha fazla veya daha az ön eğitim adımı gerektirir mi? Neden?
3. BERT'in orijinal uygulamasında `BERTEncoder` içindeki (`d2l.EncoderBlock` aracılığıyla) konumsal ileri besleme ağı da ve `MaskLM`'deki tam bağlı katman da Gauss hata doğrusal birimini (GELU) (Hendrycks and Gimpel, 2016) etkinleştirme işlevi olarak kullanır. GELU ve ReLU arasındaki farkı araştırın.

Tartışmalar[199]

[199] https://discuss.d2l.ai/t/1490

14.8 BERT Ön Eğitimi için Veri Kümesi

BERT modelini Section 14.7 içinde uygulandığı şekilde ön eğitirken, iki ön eğitim görevini kolaylaştırmak için veri kümesini ideal formatta oluşturmamız gerekir: Maskeli dil modelleme ve sonraki cümle tahmini. Bir yandan, orijinal BERT modeli, bu kitabın en okuyucuları için koşmayı zorlaştıran iki büyük külliyatın, BookCorpus ve İngilizce Wikipedia'nın (bkz. Section 14.7.5), bitiştirilmesinde ön eğitilmiştir. Öte yandan, kullanıma hazır önceden eğitilmiş BERT modeli tıp gibi belirli alanlardan gelen uygulamalar için uygun olmayabilir. Bu nedenden, BERT'i özelleştirilmiş bir veri kümesi üzerinde ön eğitmek popüler hale gelmektedir. BERT ön eğitiminin gösterilmesini kolaylaştırmak için daha küçük bir külliyat, WikiText-2 (Merity *et al.*, 2016) kullanıyoruz.

Section 14.3 içinde word2vec ön eğitimi için kullanılan PTB veri kümesiyle karşılaştırıldığında, WikiText-2 (i) orijinal noktalama işaretlerini korur ve bir sonraki cümle tahmini için uygun hale getirir; (ii) orijinal harf büyüklüğünü (büyük-küçük) ve sayıları korur; (iii) iki kat daha büyüktür.

```
import os
import random
import torch
from d2l import torch as d2l
```

WikiText-2 veri kümesinde, her satır herhangi bir noktalama işareti ile önceki belirteci arasına boşluk eklenen bir paragrafı temsil eder. En az iki cümle içeren paragraflar korunur. Cümleleri bölerken, noktayı sadece basitlik için sınırlayıcı olarak kullanıyoruz. Daha karmaşık cümle bölme teknikleriyle ilgili tartışmaları bu bölümün sonundaki alıştırmalara bırakıyoruz.

```
#@save
d2l.DATA_HUB['wikitext-2'] = (
    'https://s3.amazonaws.com/research.metamind.io/wikitext/'
    'wikitext-2-v1.zip', '3c914d17d80b1459be871a5039ac23e752a53cbe')

#@save
def _read_wiki(data_dir):
    file_name = os.path.join(data_dir, 'wiki.train.tokens')
    with open(file_name, 'r') as f:
        lines = f.readlines()
    # Büyük harfler küçük harflere dönüştürülür.
    paragraphs = [line.strip().lower().split(' . ')
                  for line in lines if len(line.split(' . ')) >= 2]
    random.shuffle(paragraphs)
    return paragraphs
```

14.8.1 Ön Eğitim Görevleri için Yardımcı İşlevleri Tanımlama

Aşağıda, iki BERT ön eğitim görevi için yardımcı fonksiyonları uygulayarak başlıyoruz: Sonraki cümle tahmini ve maskeli dil modelleme. Bu yardımcı işlevler, daha sonra, bu ham metin külliyatı, BERT ön eğitimi için ideal biçimli veri kümesine dönüştürülürken çağrılacaktır.

Sonraki Cümle Tahmini Görevi Oluşturma

Section 14.7.5 açıklamalarına göre, `_get_next_sentence` işlevi ikili sınıflandırma görevi için bir eğitim örneği oluşturur.

```
#@save
def _get_next_sentence(sentence, next_sentence, paragraphs):
    if random.random() < 0.5:
        is_next = True
    else:
        # `paragraphs` bir listelerin listelerinin listesidir
        next_sentence = random.choice(random.choice(paragraphs))
        is_next = False
    return sentence, next_sentence, is_next
```

Aşağıdaki işlev `_get_next_sentence` işlevini çağırarak `paragraph` girdisinden sonraki cümle tahmini için eğitim örnekleri oluşturur. Burada `paragraph`, her cümlenin bir belirteç listesi olduğu bir cümleler listesidir. `max_len` bağımsız değişkeni, ön eğitim sırasında BERT girdi dizisinin maksimum uzunluğunu belirtir.

```
#@save
def _get_nsp_data_from_paragraph(paragraph, paragraphs, vocab, max_len):
    nsp_data_from_paragraph = []
    for i in range(len(paragraph) - 1):
        tokens_a, tokens_b, is_next = _get_next_sentence(
            paragraph[i], paragraph[i + 1], paragraphs)
        # 1 tane '<cls>' belirteci ve 2 tane '<sep>' belirteci düşünün
        if len(tokens_a) + len(tokens_b) + 3 > max_len:
            continue
        tokens, segments = d2l.get_tokens_and_segments(tokens_a, tokens_b)
        nsp_data_from_paragraph.append((tokens, segments, is_next))
    return nsp_data_from_paragraph
```

Maskeli Dil Modelleme Görevi Oluşturma

BERT girdi dizisinden maskelenmiş dil modelleme görevi için eğitim örnekleri oluşturmak için aşağıdaki `_replace_mlm_tokens` işlevini tanımlıyoruz. Girdilerinde, `tokens`, BERT girdi dizisini temsil eden belirteçlerin bir listesidir, `candidate_pred_positions`, özel belirteçler hariç BERT girdi dizisinin belirteç indekslerinin bir listesidir (maskeli dil modelleme görevinde özel belirteçler tahmin edilmez) ve `num_mlm_preds` tahminlerin sayısını gösterir (tahmin etmek için %15 rastgele belirteci geri çağırın). Section 14.7.5 içindeki maskelenmiş dil modelleme görevinin tanımını takiben, her tahmin konumunda, girdi özel bir "<mask>" belirteci veya rastgele bir belirteç ile değiştirilebilir veya değişmeden kalabilir. Sonunda, işlev olası değiştirmeden sonra girdi belirteçlerini, tahminlerin gerçekleştiği belirteç endekslerini ve bu tahminler için etiketleri döndürür.

```
#@save
def _replace_mlm_tokens(tokens, candidate_pred_positions, num_mlm_preds,
                        vocab):
    # Girdinin değiştirilmiş '<mask>' veya rastgele belirteçler içerebileceği
    # maskelenmiş bir dil modelinin girdisi için belirteçlerin
```

(continues on next page)

(continued from previous page)

```
    # yeni bir kopyasını oluşturun
    mlm_input_tokens = [token for token in tokens]
    pred_positions_and_labels = []
    # Maskeli dil modelleme görevinde tahmin için %15 rastgele
    # belirteç almak için karıştır
    random.shuffle(candidate_pred_positions)
    for mlm_pred_position in candidate_pred_positions:
        if len(pred_positions_and_labels) >= num_mlm_preds:
            break
        masked_token = None
        # Zamanın %80'inde sözcüğü '<mask>' simgesiyle değiştirin
        if random.random() < 0.8:
            masked_token = '<mask>'
        else:
            # Zamanın %10'unda kelimeyi değiştirmeden bırakın
            if random.random() < 0.5:
                masked_token = tokens[mlm_pred_position]
            # Zamanın %10'unda kelimeyi rastgele bir kelimeyle değiştirin
            else:
                masked_token = random.choice(vocab.idx_to_token)
        mlm_input_tokens[mlm_pred_position] = masked_token
        pred_positions_and_labels.append(
            (mlm_pred_position, tokens[mlm_pred_position]))
    return mlm_input_tokens, pred_positions_and_labels
```

Yukarıda bahsedilen `_replace_mlm_tokens` işlevini çağırarak, aşağıdaki işlev bir BERT girdi dizisini (`tokens`) girdi olarak alır ve girdi belirteçlerinin dizinlerini (Section 14.7.5 içinde açıklandığı gibi olası belirteç değişiminden sonra), belirteç tahminlerin gerçekleştiği indeksleri ve bu tahminler için etiket indekslerini döndürür.

```
#@save
def _get_mlm_data_from_tokens(tokens, vocab):
    candidate_pred_positions = []
    # `tokens` bir dizgiler listesidir
    for i, token in enumerate(tokens):
        # Maskeli dil modelleme görevinde özel belirteçler tahmin edilmez
        if token in ['<cls>', '<sep>']:
            continue
        candidate_pred_positions.append(i)
    # Rastgele belirteçlerin %15'i maskelenmiş dil modelleme görevinde tahmin edilir
    num_mlm_preds = max(1, round(len(tokens) * 0.15))
    mlm_input_tokens, pred_positions_and_labels = _replace_mlm_tokens(
        tokens, candidate_pred_positions, num_mlm_preds, vocab)
    pred_positions_and_labels = sorted(pred_positions_and_labels,
                                       key=lambda x: x[0])
    pred_positions = [v[0] for v in pred_positions_and_labels]
    mlm_pred_labels = [v[1] for v in pred_positions_and_labels]
    return vocab[mlm_input_tokens], pred_positions, vocab[mlm_pred_labels]
```

14.8.2 Metni Ön Eğitim Veri Kümesine Dönüştürme

Şimdi BERT ön eğitimi için bir `Dataset` sınıfını özelleştirmeye neredeyse hazırız. Bundan önce, girdilere özel "<mask>" belirteçlerini eklemek için `_pad_bert_inputs` bir yardımcı işlevi tanımlamamız gerekiyor. Onun argümanı `examples`, iki ön eğitim görevi için yardımcı işlevlerden `_get_nsp_data_from_paragraph` ve `_get_mlm_data_from_tokens` çıktılarını içerir.

```
#@save
def _pad_bert_inputs(examples, max_len, vocab):
    max_num_mlm_preds = round(max_len * 0.15)
    all_token_ids, all_segments, valid_lens,  = [], [], []
    all_pred_positions, all_mlm_weights, all_mlm_labels = [], [], []
    nsp_labels = []
    for (token_ids, pred_positions, mlm_pred_label_ids, segments,
         is_next) in examples:
        all_token_ids.append(torch.tensor(token_ids + [vocab['<pad>']] * (
            max_len - len(token_ids)), dtype=torch.long))
        all_segments.append(torch.tensor(segments + [0] * (
            max_len - len(segments)), dtype=torch.long))
        # `valid_lens`, '<pad>' belirteçlerinin sayısını hariç tutar
        valid_lens.append(torch.tensor(len(token_ids), dtype=torch.float32))
        all_pred_positions.append(torch.tensor(pred_positions + [0] * (
            max_num_mlm_preds - len(pred_positions)), dtype=torch.long))
        # Dolgulu belirteçlerin tahminleri, 0 ağırlıklar çarpımı
        # yoluyla kayıpta filtrelenecektir.
        all_mlm_weights.append(
            torch.tensor([1.0] * len(mlm_pred_label_ids) + [0.0] * (
                max_num_mlm_preds - len(pred_positions)),
                dtype=torch.float32))
        all_mlm_labels.append(torch.tensor(mlm_pred_label_ids + [0] * (
            max_num_mlm_preds - len(mlm_pred_label_ids)), dtype=torch.long))
        nsp_labels.append(torch.tensor(is_next, dtype=torch.long))
    return (all_token_ids, all_segments, valid_lens, all_pred_positions,
            all_mlm_weights, all_mlm_labels, nsp_labels)
```

İki ön eğitim görevinin eğitim örneklerini üretmek için yardımcı fonksiyonları ve girdi dolgulama için yardımcı işlevini bir araya getirerek, BERT ön eğitimi için WikiText-2 veri kümesi olarak aşağıdaki `_WikiTextDataset` sınıfını özelleştiriyoruz. `__getitem__` işlevini uygulayarak, WikiText-2 külliyatından bir çift cümleden oluşturulan ön eğitim (maskeli dil modelleme ve sonraki cümle tahmini) örneklerine keyfi olarak erişebiliriz.

Orijinal BERT modeli kelime boyutu 30000 (Wu *et al.*, 2016) olan WordPiece gömmeleri kullanır. WordPiece'nin belirteçlere ayırma yöntemi, `subsec_Byte_Pair_Encoding` içinde orijinal sekizli çifti kodlama algoritmasının hafif bir değişiğidir. Basitlik için, belirteçlere ayırmak için `d2l.tokenize` işlevini kullanıyoruz. Beş kereden az görünen seyrek belirteçler filtrelenir.

```
#@save
class _WikiTextDataset(torch.utils.data.Dataset):
    def __init__(self, paragraphs, max_len):
        # Girdi `paragraphs[i]`, bir paragrafı temsil eden cümle dizilerinin
        # bir listesidir; çıktı `paragraphs[i]` bir paragrafı temsil eden
        # cümlelerin bir listesidir, burada her cümle bir belirteç listesidir
        paragraphs = [d2l.tokenize(
            paragraph, token='word') for paragraph in paragraphs]
```

(continues on next page)

(continued from previous page)

```
        sentences = [sentence for paragraph in paragraphs
                     for sentence in paragraph]
        self.vocab = d2l.Vocab(sentences, min_freq=5, reserved_tokens=[
            '<pad>', '<mask>', '<cls>', '<sep>'])
        # Bir sonraki cümle tahmini görevi için veri alın
        examples = []
        for paragraph in paragraphs:
            examples.extend(_get_nsp_data_from_paragraph(
                paragraph, paragraphs, self.vocab, max_len))
        # Maskeli dil modeli görevi için veri alın
        examples = [(_get_mlm_data_from_tokens(tokens, self.vocab)
                     + (segments, is_next))
                    for tokens, segments, is_next in examples]
        # Girdiyi dolgula
        (self.all_token_ids, self.all_segments, self.valid_lens,
         self.all_pred_positions, self.all_mlm_weights,
         self.all_mlm_labels, self.nsp_labels) = _pad_bert_inputs(
            examples, max_len, self.vocab)

    def __getitem__(self, idx):
        return (self.all_token_ids[idx], self.all_segments[idx],
                self.valid_lens[idx], self.all_pred_positions[idx],
                self.all_mlm_weights[idx], self.all_mlm_labels[idx],
                self.nsp_labels[idx])

    def __len__(self):
        return len(self.all_token_ids)
```

`_read_wiki` işlevini ve `_WikiTextDataset` sınıfını kullanarak, WikiText-2 veri kümesini indirmek ve ondan ön eğitim örnekleri oluşturmak için aşağıdaki `load_data_wiki`'yı tanımlıyoruz.

```
#@save
def load_data_wiki(batch_size, max_len):
    """WikiText-2 veri kümesini yükleyin."""
    num_workers = d2l.get_dataloader_workers()
    data_dir = d2l.download_extract('wikitext-2', 'wikitext-2')
    paragraphs = _read_wiki(data_dir)
    train_set = _WikiTextDataset(paragraphs, max_len)
    train_iter = torch.utils.data.DataLoader(train_set, batch_size,
                                             shuffle=True, num_workers=num_workers)
    return train_iter, train_set.vocab
```

Toplu iş boyutunu 512 olarak ve BERT girdi dizisinin maksimum uzunluğunu 64 olarak ayarlayarak, BERT ön eğitim örneklerinden oluşan bir minigrubun şekillerini yazdırıyoruz. Her BERT girdi dizisinde, maskelenmiş dil modelleme görevi için 10 (64×0.15) konumun tahmin edildiğini unutmayın.

```
batch_size, max_len = 512, 64
train_iter, vocab = load_data_wiki(batch_size, max_len)

for (tokens_X, segments_X, valid_lens_x, pred_positions_X, mlm_weights_X,
     mlm_Y, nsp_y) in train_iter:
    print(tokens_X.shape, segments_X.shape, valid_lens_x.shape,
```

(continues on next page)

(continued from previous page)

```
        pred_positions_X.shape, mlm_weights_X.shape, mlm_Y.shape,
        nsp_y.shape)
    break
```

```
torch.Size([512, 64]) torch.Size([512, 64]) torch.Size([512]) torch.Size([512, 10]) torch.
↪Size([512, 10]) torch.Size([512, 10]) torch.Size([512])
```

Sonunda, kelime dağarcığına bir göz atalım. Sık görülen belirteçleri filtreledikten sonra bile, PTB veri kümesinden iki kat daha büyüktür.

```
len(vocab)
```

```
20256
```

14.8.3 Özet

- PTB veri kümesiyle karşılaştırıldığında, WikiText-2 veri kümesi orijinal noktalama işaretlerini, büyük/küçük harf ve sayıları korur ve iki kat daha büyüktür.
- WikiText-2 külliyatındaki bir çift cümleden oluşturulan ön eğitim (maskeli dil modellemesi ve sonraki cümle tahmini) örneklerine keyfi olarak erişebiliriz.

14.8.4 Alıştırmalar

1. Basitlik açısından, nokta cümleleri bölmede tek sınırlayıcı olarak kullanılır. SpaCy ve NLTK gibi diğer cümle ayırma tekniklerini deneyin. Örnek olarak NLTK'yi ele alalım. Önce NLTK'yi kurmanız gerekiyor: `pip install nltk`. Kodda, ilk `import nltk` çağırın. Daha sonra Punkt cümle belirteci ayıklayıcıyı indirin: `nltk.download('punkt')`. `sentences = 'This is great ! Why not ?'` gibi cümleleri ayırmak için `nltk.tokenize.sent_tokenize(sentences)` çağırmak iki cümlelik bir liste döndürür: `['This is great !', 'Why not ?']`
2. Seyrek belirteçleri filtrelemezsek kelime dağarcığı boyutu ne olur?

Tartışmalar[200]

14.9 BERT Ön Eğitimi

Section 14.7 içinde uygulanan BERT modeli ve Section 14.8 içindeki WikiText-2 veri kümesinden oluşturulan ön eğitim örnekleriyle bu bölümde BERT'in WikiText-2 veri kümesi üzerinde ön eğitimini yapacağız.

```
import torch
from torch import nn
from d2l import torch as d2l
```

[200] https://discuss.d2l.ai/t/1496

Başlamak için WikiText-2 veri kümesini maskelenmiş dil modellemesi ve sonraki cümle tahmini için ön eğitim örneklerinin minigrubu olarak yükleriz. Toplu iş boyutu 512 ve BERT girdi dizisinin maksimum uzunluğu 64'tür. Orijinal BERT modelinde maksimum uzunluğun 512 olduğunu unutmayın.

```
batch_size, max_len = 512, 64
train_iter, vocab = d2l.load_data_wiki(batch_size, max_len)
```

14.9.1 BERT Ön Eğitimi

Orijinal BERT'in farklı model boyutlarında (Devlin *et al.*, 2018) iki sürümü vardır. Temel model ($\text{BERT}_{\text{BASE}}$) 768 gizli birim (gizli boyut) ve 12 öz-dikkat kafası olan 12 katman (dönüştürücü kodlayıcı blokları) kullanır. Büyük model ($\text{BERT}_{\text{LARGE}}$), 1024 gizli birimli ve 16 öz-dikkat kafalı 24 katman kullanır. Dikkate değer şekilde, birincisinin 110 milyon parametresi varken, ikincisi 340 milyon parametreye sahiptir. Kolayca gösterim için, 2 katman, 128 gizli birim ve 2 öz-dikkat kafası kullanarak küçük bir BERT tanımlıyoruz.

```
net = d2l.BERTModel(len(vocab), num_hiddens=128, norm_shape=[128],
                    ffn_num_input=128, ffn_num_hiddens=256, num_heads=2,
                    num_layers=2, dropout=0.2, key_size=128, query_size=128,
                    value_size=128, hid_in_features=128, mlm_in_features=128,
                    nsp_in_features=128)
devices = d2l.try_all_gpus()
loss = nn.CrossEntropyLoss()
```

Eğitim döngüsünü tanımlamadan önce, `_get_batch_loss_bert` yardımcı işlevini tanımlıyoruz. Eğitim örneklerinin parçası göz önüne alındığında, bu işlev hem maskelenmiş dil modellemesi hem de sonraki cümle tahmini görevlerinin kaybını hesaplar. BERT ön eğitiminin son kaybı sadece hem maskeli dil modelleme kaybının hem de bir sonraki cümle tahmini kaybının toplamıdır.

```
#@save
def _get_batch_loss_bert(net, loss, vocab_size, tokens_X,
                         segments_X, valid_lens_x,
                         pred_positions_X, mlm_weights_X,
                         mlm_Y, nsp_y):
    # İleri ilet
    _, mlm_Y_hat, nsp_Y_hat = net(tokens_X, segments_X,
                                  valid_lens_x.reshape(-1),
                                  pred_positions_X)
    # Hesaplanan maskeli dil modeli kaybı
    mlm_l = loss(mlm_Y_hat.reshape(-1, vocab_size), mlm_Y.reshape(-1)) *\
    mlm_weights_X.reshape(-1, 1)
    mlm_l = mlm_l.sum() / (mlm_weights_X.sum() + 1e-8)
    # Sonraki cümle tahmin kaybını hesapla
    nsp_l = loss(nsp_Y_hat, nsp_y)
    l = mlm_l + nsp_l
    return mlm_l, nsp_l, l
```

Yukarıda belirtilen iki yardımcı işlevini çağıran aşağıdaki `train_bert` işlevi, WikiText-2 (`train_iter`) veri kümesindeki BERT (`net`) ön eğitim prosedürünü tanımlar. BERT eğitimi çok uzun sürebilir. `train_ch13` işlevinde olduğu gibi eğitim için dönemlerin sayısını belirtmek yerine

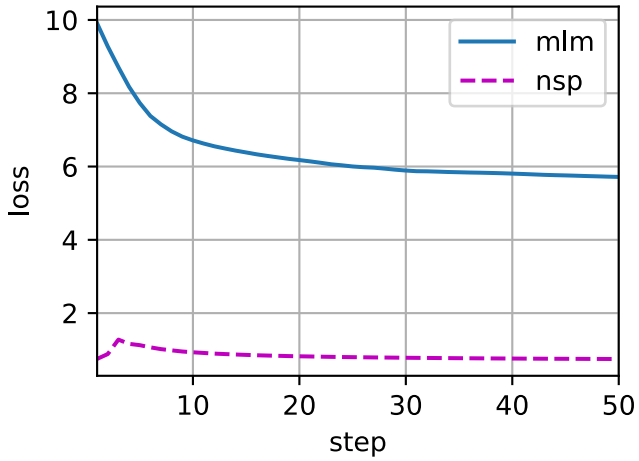
(bkz. Section 13.1), aşağıdaki işlevin `num_steps` girdisi, eğitim için yineleme adımlarının sayısını belirtir.

```
def train_bert(train_iter, net, loss, vocab_size, devices, num_steps):
    net = nn.DataParallel(net, device_ids=devices).to(devices[0])
    trainer = torch.optim.Adam(net.parameters(), lr=0.01)
    step, timer = 0, d2l.Timer()
    animator = d2l.Animator(xlabel='step', ylabel='loss',
                            xlim=[1, num_steps], legend=['mlm', 'nsp'])
    # Maskeli dil modelleme kayıplarının toplamı,
    # sonraki cümle tahmin kayıplarının toplamı,
    # cümle çiftlerinin sayısı, adet
    metric = d2l.Accumulator(4)
    num_steps_reached = False
    while step < num_steps and not num_steps_reached:
        for tokens_X, segments_X, valid_lens_x, pred_positions_X,\
            mlm_weights_X, mlm_Y, nsp_y in train_iter:
            tokens_X = tokens_X.to(devices[0])
            segments_X = segments_X.to(devices[0])
            valid_lens_x = valid_lens_x.to(devices[0])
            pred_positions_X = pred_positions_X.to(devices[0])
            mlm_weights_X = mlm_weights_X.to(devices[0])
            mlm_Y, nsp_y = mlm_Y.to(devices[0]), nsp_y.to(devices[0])
            trainer.zero_grad()
            timer.start()
            mlm_l, nsp_l, l = _get_batch_loss_bert(
                net, loss, vocab_size, tokens_X, segments_X, valid_lens_x,
                pred_positions_X, mlm_weights_X, mlm_Y, nsp_y)
            l.backward()
            trainer.step()
            metric.add(mlm_l, nsp_l, tokens_X.shape[0], 1)
            timer.stop()
            animator.add(step + 1,
                         (metric[0] / metric[3], metric[1] / metric[3]))
            step += 1
            if step == num_steps:
                num_steps_reached = True
                break

    print(f'MLM loss {metric[0] / metric[3]:.3f}, '
          f'NSP loss {metric[1] / metric[3]:.3f}')
    print(f'{metric[2] / timer.sum():.1f} sentence pairs/sec on '
          f'{str(devices)}')
```

BERT ön eğitimi sırasında hem maskeli dil modelleme kaybını hem de sonraki cümle tahmini kaybını çizdirebiliriz.

```
train_bert(train_iter, net, loss, len(vocab), devices, 50)
```

```
MLM loss 5.717, NSP loss 0.748
4163.7 sentence pairs/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



14.9.2 BERT ile Metni Temsil Etme

BERT ön eğitiminden sonra onu, tek metin, metin çiftleri veya bunlardaki herhangi bir belirteci temsil etmek için kullanabiliriz. Aşağıdaki işlev, `tokens_a` ve `tokens_b`'teki tüm belirteçler için BERT (`net`) temsillerini döndürür.

```
def get_bert_encoding(net, tokens_a, tokens_b=None):
    tokens, segments = d2l.get_tokens_and_segments(tokens_a, tokens_b)
    token_ids = torch.tensor(vocab[tokens], device=devices[0]).unsqueeze(0)
    segments = torch.tensor(segments, device=devices[0]).unsqueeze(0)
    valid_len = torch.tensor(len(tokens), device=devices[0]).unsqueeze(0)
    encoded_X, _, _ = net(token_ids, segments, valid_len)
    return encoded_X
```

"A crane is flying" cümlesini düşünün. Section 14.7.4 içinde tartışıldığı gibi BERT girdi temsilini hatırlayın. Özel belirteçleri ekledikten sonra "<cls>" (sınıflandırma için kullanılır) ve "<sep>" (ayırma için kullanılır), BERT girdi dizisi altı uzunluğa sahiptir. Sıfır "<cls>" belirteci dizini olduğundan, `encoded_text[:, 0, :]` tüm girdi cümlesinin BERT temsilidir. Çokanlamlılık belirteci "crane"yi değerlendirmek için, belirteçin BERT temsilinin ilk üç öğesini de yazdırıyoruz.

```
tokens_a = ['a', 'crane', 'is', 'flying']
encoded_text = get_bert_encoding(net, tokens_a)
# Tokens: '<cls>', 'a', 'crane', 'is', 'flying', '<sep>'
encoded_text_cls = encoded_text[:, 0, :]
encoded_text_crane = encoded_text[:, 2, :]
encoded_text.shape, encoded_text_cls.shape, encoded_text_crane[0][:3]
```

```
(torch.Size([1, 6, 128]),
 torch.Size([1, 128]),
 tensor([-0.0641, -2.2734,  1.1625], device='cuda:0', grad_fn=<SliceBackward0>))
```

Şimdi bir cümle çifti düşünün "a crane driver came" ("bir vinç sürücüsü geldi") ve "he just left" ("az önce gitti"). Benzer şekilde, `encoded_pair[:, 0, :]`, önceden eğitilmiş BERT'ten tüm cümle çiftinin kodlanmış sonucudur. Çokanlamlılık belirteci "crane"nin (vinç veya turna) ilk üç öğesinin, bağlam farklı olduğu zamanlardan farklı olduğunu unutmayın. Bu BERT temsillerinin bağlam duyarlı olduğunu destekler.

```
tokens_a, tokens_b = ['a', 'crane', 'driver', 'came'], ['he', 'just', 'left']
encoded_pair = get_bert_encoding(net, tokens_a, tokens_b)
# Andıçlar: '<cls>', 'a', 'crane', 'driver', 'came', '<sep>', 'he', 'just',
# 'left', '<sep>'
encoded_pair_cls = encoded_pair[:, 0, :]
encoded_pair_crane = encoded_pair[:, 2, :]
encoded_pair.shape, encoded_pair_cls.shape, encoded_pair_crane[0][:3]
```

```
(torch.Size([1, 10, 128]),
 torch.Size([1, 128]),
 tensor([-0.1420, -0.1808,  0.0342], device='cuda:0', grad_fn=<SliceBackward0>))
```

Chapter 15 içinde, aşağı akış doğal dil işleme uygulamaları için önceden eğitilmiş bir BERT modelinde ince ayar yapacağız.

14.9.3 Özet

- Orijinal BERT, temel modelin 110 milyon parametreye sahip olduğu ve büyük modelin 340 milyon parametreye sahip olduğu iki sürüme sahiptir.
- BERT ön eğitiminden sonra onu, tek metin, metin çiftleri veya bunlardaki herhangi bir belirteci temsil etmek için kullanabiliriz.
- Deneyde, aynı belirteç, bağlamları farklı olduğunda farklı BERT temsiline sahiptir. Bu BERT temsillerinin bağlam duyarlı olduğunu destekler.

14.9.4 Alıştırmalar

1. Deneyde, maskeli dil modelleme kaybının bir sonraki cümle tahmini kaybından önemli ölçüde daha yüksek olduğunu görebiliriz. Neden?
2. BERT girdi dizisinin maksimum uzunluğunu 512 olarak ayarlayın (orijinal BERT modeliyle aynı). Orijinal BERT modelinin $\text{BERT}_{\text{LARGE}}$ gibi yapılandırmalarını kullanın. Bu bölümü çalıştırırken herhangi bir hatayla karşılaşıyor musunuz? Neden?

Tartışmalar[201]

[201] https://discuss.d2l.ai/t/1497

15 | Doğal Dil İşleme: Uygulamalar

Metin dizilerinde belirteçleri nasıl temsil edeceğimizi ve Chapter 14 içinde temsillerini nasıl eğitileceğini gördük. Bu tür önceden eğitilmiş metin temsilleri, farklı alt akış doğal dil işleme görevleri için çeşitli modellere beslenebilir.

Aslında, daha önceki bölümler zaten bazı doğal dil işleme uygulamalarını tartıştık; *ön eğitimsiz*, sadece derin öğrenme mimarilerini açıklamak için. Örneğin, Chapter 8 içinde, roman benzeri metinler üretmede dil modelleri tasarlamak için RNN'lere güvendik. Chapter 9 ve Chapter 10 içinde, makine çevirisi için RNN'lere ve dikkat mekanizmalarına dayanan modeller de tasarladık.

Ancak, bu kitap tüm bu tür uygulamaları kapsamlı bir şekilde açıklamak niyetinde değildir. Bunun yerine, odak noktamız *doğal dil işleme problemlerini ele almak için dillerin (derin) temsili öğreniminin nasıl uygulanacağıdır*. Önceden eğitilmiş metin temsilleri göz önüne alındığında, bu bölüm iki popüler ve temsili aşağı akış doğal dil işleme görevini inceleyecektir: Duygu analizi ve doğal dil çıkarımı, sırasıyla tek metin ve metin çiftlerinin ilişkilerini analiz eder.

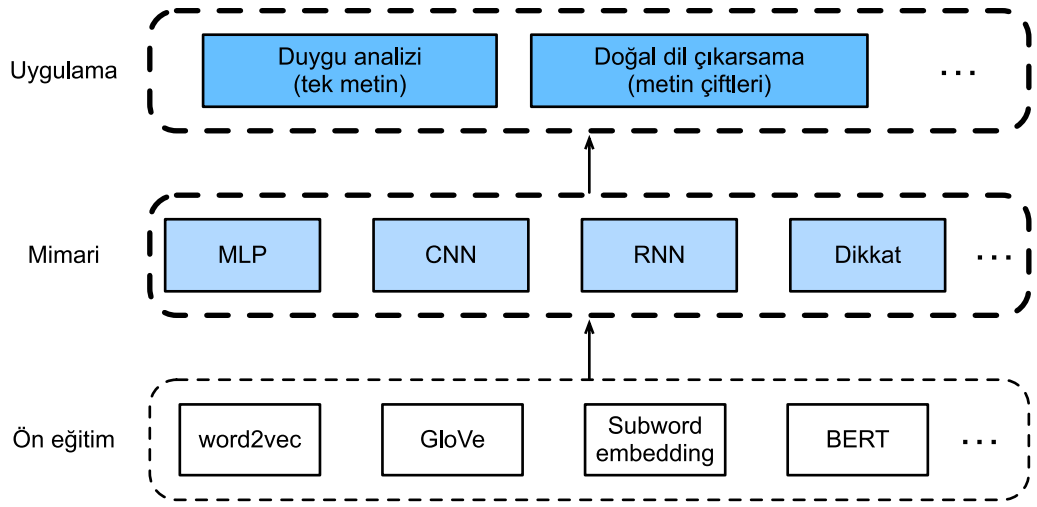


Fig. 15.1: Önceden eğitilmiş metin temsilleri, farklı akış aşağı doğal dil işleme uygulamaları için çeşitli derin öğrenme mimarilerine beslenebilir. Bu bölüm, farklı aşağı akış doğal dil işleme uygulamaları için modellerin nasıl tasarlanacağına odaklanır.

Fig. 15.1 üzerinde tasvir edildiği gibi, bu bölüm, MLP'ler, CNN'ler, RNN'ler ve dikkat gibi farklı derin öğrenme mimarileri türlerini kullanarak doğal dil işleme modellerinin tasarlanmasına ilişkin temel fikirleri açıklamaya odaklanmaktadır. Herhangi bir önceden eğitilmiş metin temsillerini Fig. 15.1 içindeki her iki uygulama için herhangi bir mimariyle birleştirmek mümkün olsa da, birkaç temsili kombinasyon seçiyoruz. Özellikle, duygu analizi için RNN'lere ve CNN'lere dayalı popüler mimarileri araştıracağız. Doğal dil çıkarımı için, metin çiftlerinin nasıl analiz edileceğini göstermek için dikkati ve MLP'leri seçiyoruz. Sonunda, bir dizi düzeyinde (tek metin sınıflandır-

ması ve metin çifti sınıflandırması) ve bir belirteç düzeyinde (metin etiketleme ve soru yanıtlama) gibi çok çeşitli doğal dil işleme uygulamaları için önceden eğitilmiş bir BERT modeline nasıl ince ayar yapılacağını gösteriyoruz. Somut deneysel bir durum olarak, BERT doğal dil çıkarımı için ince ayar yapacağız.

Section 14.7 içinde tanıtılan gibi BERT, çok çeşitli doğal dil işleme uygulamaları için minimal mimari değişiklikleri gerektirir. Bununla birlikte, bu fayda, aşağı akış uygulamaları için çok sayıda BERT parametresinin ince ayarlanması pahasına gelir. Uzay veya zaman sınırlı olduğunda, MLP'ler, CNN'ler, RNN'ler ve dikkat temelli modeller daha uygulanabilirdir. Aşağıda, duygu analizi uygulamasıyla başlıyoruz ve sırasıyla RNN'lere ve CNN'lere dayalı model tasarımını gösteriyoruz.

15.1 Duygu Analizi ve Veri Kümesi

Çevrimiçi sosyal medya ve inceleme platformlarının çoğalmasıyla birlikte, karar verme süreçlerini desteklemek için büyük bir potansiyel taşıyan bir çok görüş içeren veri kaydedildi. *Duygu analizi* ürün incelemeleri, blog yorumları ve forum tartışmaları gibi insanların ürettikleri metinlerdeki duyguları inceler. Siyaset (örn. politikalara yönelik kamu duygularının analizi), finans (örneğin, pazarın duygularının analizi) ve pazarlama (örneğin, ürün araştırması ve marka yönetimi) kadar çeşitli alanlarda geniş uygulamalara sahiptir.

Duygular ayrık kutuplar veya ölçekler (örneğin, pozitif ve negatif) olarak sınıflandırılabileceğinden, duygu analizini, değişen uzunluktaki bir metin dizisini sabit uzunlukta bir metin kategorisine dönüştüren bir metin sınıflandırma görevi olarak düşünebiliriz. Bu bölümde, duygu analizi için Stanford'un film incelemesi büyük veri kümesini[202] kullanacağız. IMDb'den indirilen 25000 film incelemesini içeren bir eğitim kümesi ve bir test kümesinden oluşur. Her iki veri kümesinde de, farklı duygu kutuplarını gösteren eşit sayıda "pozitif" ve "negatif" etiketleri bulunur.

```
import os
import torch
from torch import nn
from d2l import torch as d2l
```

15.1.1 Veri Kümesini Okuma

İlk olarak, bu IMDb inceleme veri kümesini `../data/aclImdb` yoluna indirin ve genişletin.

```
#@save
d2l.DATA_HUB['aclImdb'] = (
    'http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz',
    '01ada507287d82875905620988597833ad4e0903')

data_dir = d2l.download_extract('aclImdb', 'aclImdb')
```

Ardından, eğitim ve test veri kümelerini okuyun. Her örnek bir inceleme ve onun etiketidir: "Pozitif" için 1 ve "negatif" için 0.

[202] https://ai.stanford.edu/~amaas/data/sentiment/

```
#@save
def read_imdb(data_dir, is_train):
    """IMDb inceleme veri kümesi metin dizilerini ve etiketlerini okuyun."""
    data, labels = [], []
    for label in ('pos', 'neg'):
        folder_name = os.path.join(data_dir, 'train' if is_train else 'test',
                                   label)
        for file in os.listdir(folder_name):
            with open(os.path.join(folder_name, file), 'rb') as f:
                review = f.read().decode('utf-8').replace('\n', '')
                data.append(review)
                labels.append(1 if label == 'pos' else 0)
    return data, labels

train_data = read_imdb(data_dir, is_train=True)
print('# trainings:', len(train_data[0]))
for x, y in zip(train_data[0][:3], train_data[1][:3]):
    print('label:', y, 'review:', x[0:60])
```

```
# trainings: 25000
label: 1 review: Henry Hathaway was daring, as well as enthusiastic, for his
label: 1 review: An unassuming, subtle and lean film, "The Man in the White S
label: 1 review: Eddie Murphy really made me laugh my ass off on this HBO sta
```

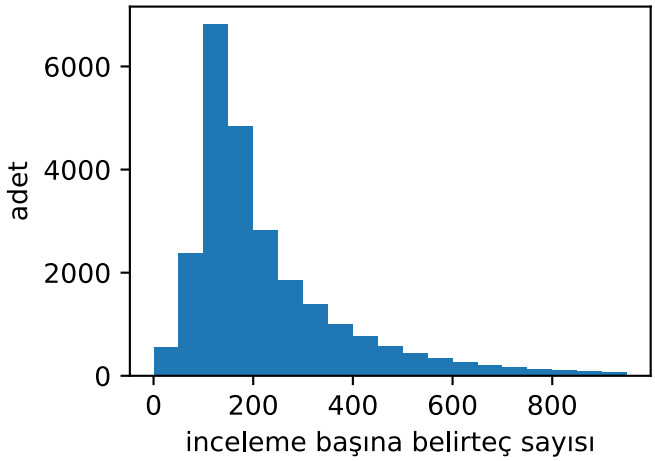
15.1.2 Veri Kümesini Ön İşlemesi

Her kelimeye bir belirteç gibi davranarak ve 5 kereden az görünen sözcükleri filtreleyerek, eğitim veri kümesinden bir kelime dağarcığı oluşturuyoruz.

```
train_tokens = d2l.tokenize(train_data[0], token='word')
vocab = d2l.Vocab(train_tokens, min_freq=5, reserved_tokens=['<pad>'])
```

Belirteçlendirme işleminden sonra, inceleme uzunluklarının belirteçler cinsinden histogramını çizelim.

```
d2l.set_figsize()
d2l.plt.xlabel('inceleme başına belirteç sayısı')
d2l.plt.ylabel('adet')
d2l.plt.hist([len(line) for line in train_tokens], bins=range(0, 1000, 50));
```



Beklediğimiz gibi, incelemeler değişen uzunluklara sahiptir. Bu tür incelemelerin minigrup işlemlerini her seferinde işlemek için, Section 9.5 içindeki makine çevirisi veri kümesi için ön işleme adımına benzer olan budama ve dolgulama ile her incelemenin uzunluğunu 500 olarak ayarladık.

```
num_steps = 500  # dizi uzunluğu
train_features = torch.tensor([d2l.truncate_pad(
    vocab[line], num_steps, vocab['<pad>']) for line in train_tokens])
print(train_features.shape)
```

```
torch.Size([25000, 500])
```

15.1.3 Veri Yineleyiciler Oluşturma

Şimdi veri yineleyiciler oluşturabiliriz. Her yinelemede, bir örnek minigrubu döndürülür.

```
train_iter = d2l.load_array((train_features, torch.tensor(train_data[1])), 64)

for X, y in train_iter:
    print('X:', X.shape, ', y:', y.shape)
    break
print('# batches:', len(train_iter))
```

```
X: torch.Size([64, 500]) , y: torch.Size([64])
# batches: 391
```

15.1.4 Her Şeyleri Bir Araya Koymak

Son olarak, yukarıdaki adımları `load_data_imdb` işlevinde sarıyoruz. İşlev eğitim ve test veri yineleyicileri ve IMDb inceleme veri kümesinin kelime dağarcığını döndürür.

```
#@save
def load_data_imdb(batch_size, num_steps=500):
    """Veri yineleyicilerini ve IMDb inceleme veri kümesinin kelime dağarcığını döndür."""
    data_dir = d2l.download_extract('aclImdb', 'aclImdb')
    train_data = read_imdb(data_dir, True)
    test_data = read_imdb(data_dir, False)
    train_tokens = d2l.tokenize(train_data[0], token='word')
    test_tokens = d2l.tokenize(test_data[0], token='word')
    vocab = d2l.Vocab(train_tokens, min_freq=5)
    train_features = torch.tensor([d2l.truncate_pad(
        vocab[line], num_steps, vocab['<pad>']) for line in train_tokens])
    test_features = torch.tensor([d2l.truncate_pad(
        vocab[line], num_steps, vocab['<pad>']) for line in test_tokens])
    train_iter = d2l.load_array((train_features, torch.tensor(train_data[1])),
                                batch_size)
    test_iter = d2l.load_array((test_features, torch.tensor(test_data[1])),
                               batch_size,
                               is_train=False)
    return train_iter, test_iter, vocab
```

15.1.5 Özet

- Duygu analizi, değişen uzunluktaki bir metin dizisini sabit uzunlukta bir metin kategorisine dönüştüren bir metin sınıflandırma problemi olarak kabul edilen, insanların ürettikleri metindeki duygularını inceler.
- Ön işlemeden sonra Stanford'un film inceleme büyük veri kümesini (IMDb inceleme veri kümesini) kelime dağarcıklı veri yineleyicilerine yükleyebiliriz.

15.1.6 Alıştırmalar

1. Duygu analizi modellerinin eğitimi hızlandırmak için bu bölümdeki hangi hiper parametreleri değiştirebiliriz?
2. Amazon incelemeleri[203] veri kümesini duygu analizi için veri yineleyicilerine ve etiketlere yükleme için bir işlev uygulayabilir misiniz?

Tartışmalar[204]

[203] https://snap.stanford.edu/data/web-Amazon.html
[204] https://discuss.d2l.ai/t/1387

15.2 Duygu Analizi: Yinemeli Sinir Ağlarının Kullanımı

Kelime benzerliği ve benzetme görevleri gibi, biz de duygu analizine önceden eğitilmiş kelime vektörleri de uygulayabiliriz. Section 15.1 içindeki IMDb inceleme veri kümesi çok büyük olmadığından, büyük ölçekli külliyat üzerinde önceden eğitilmiş metin temsillerinin kullanılması modelin aşırı öğrenmesini azaltabilir. Fig. 15.2.1 içinde gösterilen özel bir örnek olarak, önceden eğitilmiş GloVe modelini kullanarak her belirteci temsil edeceğiz ve bu belirteç temsillerini metin dizisi gösterimini, ki duygu analizi çıktılarına dönüştürülecektir, elde etmek için çok katmanlı çift yönlü bir RNN'ye besleyeceğiz (Maas *et al.*, 2011). Aynı aşağı akım uygulaması için daha sonra farklı bir mimari seçimi ele alacağız.

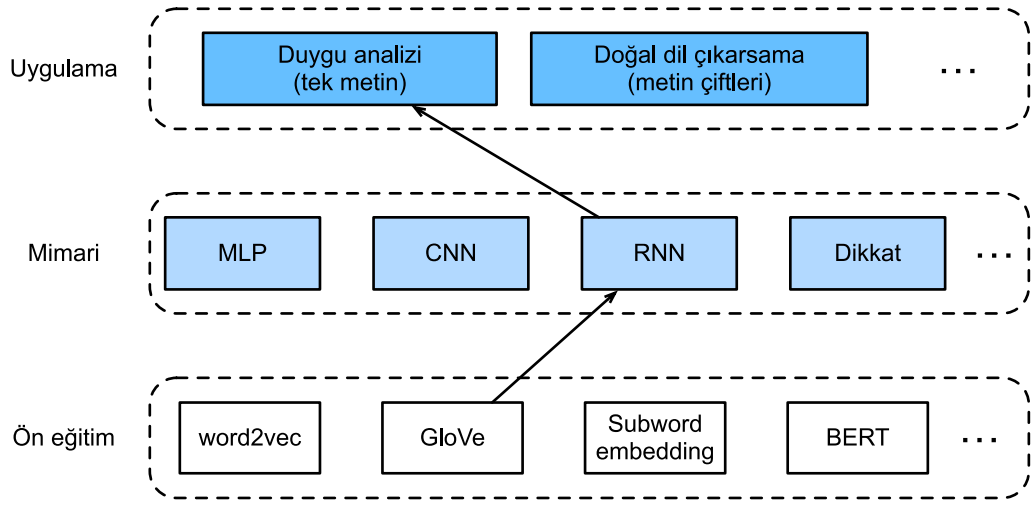


Fig. 15.2.1: Bu bölüm, duygu analizi için önceden eğitilmiş GloVe'yi RNN tabanlı bir mimariye besler.

```
import torch
from torch import nn
from d2l import torch as d2l

batch_size = 64
train_iter, test_iter, vocab = d2l.load_data_imdb(batch_size)
```

15.2.1 RNN'lerle Tek Metni Temsil Etme

Duygu analizi gibi metin sınıflandırmalarında, değişen uzunluktaki bir metin dizisi sabit uzunlukta kategorilere dönüştürülür. Aşağıdaki `BiRNN` sınıfında, bir metin dizisinin her belirteci, gömme katman (`self.embedding`) aracılığıyla bireysel önceden eğitilmiş GloVe temsilini alırken, tüm dizi çift yönlü RNN (`self.encoder`) ile kodlanır. Daha somut olarak, iki yönlü LSTM'nin hem ilk hem de son zaman adımlarındaki gizli durumları (son katmanda), metin dizisinin temsili olarak bitiştirilir. Bu tek metin gösterimi daha sonra iki çıktıya ("pozitif" ve "negatif") sahip tam bağlı bir katman (`self.encoder`) tarafından çıktı kategorilerine dönüştürülür.

```
class BiRNN(nn.Module):
    def __init__(self, vocab_size, embed_size, num_hiddens,
                 num_layers, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        super(BiRNN, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # Çift yönlü bir RNN elde etmek için `bidirectional` (çift yönlü)
        # öğesini True olarak ayarlayın
        self.encoder = nn.LSTM(embed_size, num_hiddens, num_layers=num_layers,
                               bidirectional=True)
        self.decoder = nn.Linear(4 * num_hiddens, 2)

    def forward(self, inputs):
        # `inputs` şekli (parti boyutu, zaman adımı sayısı)'dır. LSTM,
        # girdisinin ilk boyutunun zamansal boyut olmasını gerektirdiğinden,
        # girdi, belirteç temsilleri elde edilmeden önce değiştirilir. Çıktı
        # şekli (zaman adımı sayısı, parti boyutu, kelime vektör boyutu)
        embeddings = self.embedding(inputs.T)
        self.encoder.flatten_parameters()
        # Farklı zaman adımlarında son gizli katmanın gizli durumlarını
        # döndürür. `outputs` şekli
        # (zaman adımı sayısı, iş boyutu, 2 * gizli birim sayısı)'dır.
        outputs, _ = self.encoder(embeddings)
        # Tam bağlı katmanın girdisi olarak ilk ve son zaman adımlarında gizli
        # durumları bitiştirin.
        # Şekli (parti boyutu, 4 * gizli birim sayısı)'dır
        encoding = torch.cat((outputs[0], outputs[-1]), dim=1)
        outs = self.decoder(encoding)
        return outs
```

Duygu analizi için tek bir metni temsil etmek üzere iki gizli katman içeren çift yönlü bir RNN oluşturalım.

```
embed_size, num_hiddens, num_layers, devices = 100, 100, 2, d2l.try_all_gpus()
net = BiRNN(len(vocab), embed_size, num_hiddens, num_layers)
```

```
def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.xavier_uniform_(m.weight)
    if type(m) == nn.LSTM:
        for param in m._flat_weights_names:
            if "weight" in param:
                nn.init.xavier_uniform_(m._parameters[param])
net.apply(init_weights);
```

15.2.2 Önceden Eğitilmiş Sözcük Vektörlerini Yükleme

Aşağıda önceden eğitilmiş 100 boyutlu (`embed_size` ile tutarlı olması gerekir) kelime dağarcığındaki belirteçler için GloVe gömmelerini yüklüyoruz.

```
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
```

Kelime dağarcığındaki tüm belirteçler için vektörlerin şeklini yazdırın.

```
embeds = glove_embedding[vocab.idx_to_token]
embeds.shape
```

```
torch.Size([49346, 100])
```

Bu önceden eğitilmiş kelime vektörlerini incelemelerde belirteçleri temsil etmek için kullanıyoruz ve eğitim sırasında bu vektörleri güncellemeyeceğiz.

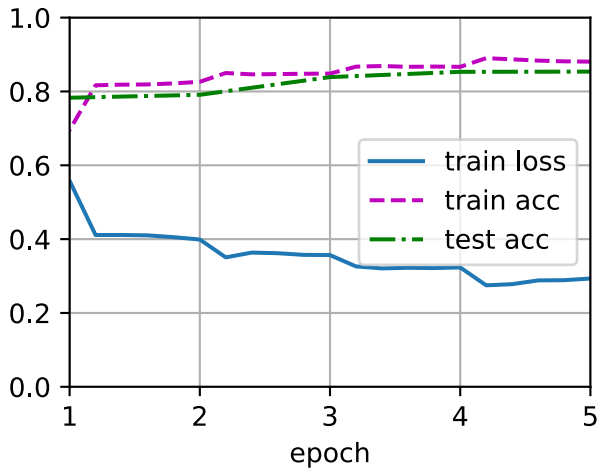
```
net.embedding.weight.data.copy_(embeds)
net.embedding.weight.requires_grad = False
```

15.2.3 Model Eğitimi ve Değerlendirilmesi

Şimdi çift yönlü RNN'leri duygu analizi için eğitebiliriz.

```
lr, num_epochs = 0.01, 5
trainer = torch.optim.Adam(net.parameters(), lr=lr)
loss = nn.CrossEntropyLoss(reduction="none")
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.293, train acc 0.881, test acc 0.854
693.3 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



Eğitimli modeli, `net`, kullanarak bir metin dizisinin duygusunu tahmin etmek için aşağıdaki işlevi tanımlıyoruz.

```
#@save
def predict_sentiment(net, vocab, sequence):
    """Bir metin dizisinin duygusunu tahmin edin."""
    sequence = torch.tensor(vocab[sequence.split()], device=d2l.try_gpu())
    label = torch.argmax(net(sequence.reshape(1, -1)), dim=1)
    return 'positive' if label == 1 else 'negative'
```

Son olarak, iki basit cümlenin duygusunu tahmin etmek için eğitimli modeli kullanalım.

```
predict_sentiment(net, vocab, 'this movie is so great')
```

```
'positive'
```

```
predict_sentiment(net, vocab, 'this movie is so bad')
```

```
'negative'
```

15.2.4 Özet

- Önceden eğitilmiş sözcük vektörleri, bir metin dizisinde tek tek belirteçleri temsil edebilir.
- Çift yönlü RNN'ler, örneğin ilk ve son zaman adımlarında gizli durumlarının bitiştirilmesi yoluyla olduğu gibi bir metin dizisini temsil edebilir. Bu tek metin gösterimi, tam bağlı bir katman kullanılarak kategorilere dönüştürülebilir.

15.2.5 Alıştırmalar

1. Dönem sayısını artırın. Eğitim ve test doğruluklarını iyileştirebilir misiniz? Diğer hiper parametreleri ayarlamaya ne dersiniz?
2. 300 boyutlu GloVe gömme gibi daha büyük önceden eğitilmiş sözcük vektörlerini kullanın. Sınıflandırma doğruluğunu arttırıyor mu?
3. SpaCy andıçlamasını kullanarak sınıflandırma doğruluğunu artırabilir miyiz? SpaCy (`pip install spacy`) ve İngilizce paketini (`python -m spacy download en`) yüklemeniz gerekir. Kodda, önce spaCy'i (`import spacy`) içe aktarın. Ardından, spaCy İngilizce paketini yükleyin (`spacy_en = spacy.load('en')`). Son olarak, `def tokenizer(text): return [tok.text for tok in spacy_en.tokenizer(text)]` işlevini tanımlayın ve orijinal `tokenizer` işlevini değiştirin. GloVe ve spaCy içinde ifade belirteçlerinin farklı biçimlerine dikkat edin. Örneğin, "new york" ifade belirteci GloVe'de "new-york" biçimini ve spaCy andıçlamasından sonra "new york" biçimini alır.

Tartışmalar[205]

15.3 Duygu Analizi: Evrişimli Sinir Ağlarının Kullanımı

Chapter 6 içinde, bitişik pikseller gibi yerel özelliklere uygulanan iki boyutlu CNN'lerle iki boyutlu imge verilerini işlemek için mekanizmaları inceledik. Aslında bilgisayarla görme için tasarlanmış olsa da, CNN'ler doğal dil işleme için de yaygın olarak kullanılmaktadır. Basitçe söylemek gerekirse, herhangi bir metin dizisini tek boyutlu bir imge olarak düşünün. Bu şekilde, tek boyutlu CNN'ler metindeki n gramlar gibi yerel özellikleri işleyebilir.

Bu bölümde, tek metni temsil etmede bir CNN mimarisinin nasıl tasarlanacağını göstermek için *textCNN* modelini kullanacağız (Kim, 2014). Duygu analizi için GloVe ön eğitimi ile RNN mimarisi

[205] https://discuss.d2l.ai/t/1424

kullanan Fig. 15.2.1 ile karşılaştırıldığında, Fig. 15.3.1 mimarisindeki tek fark mimarinin seçiminde yatmaktadır.

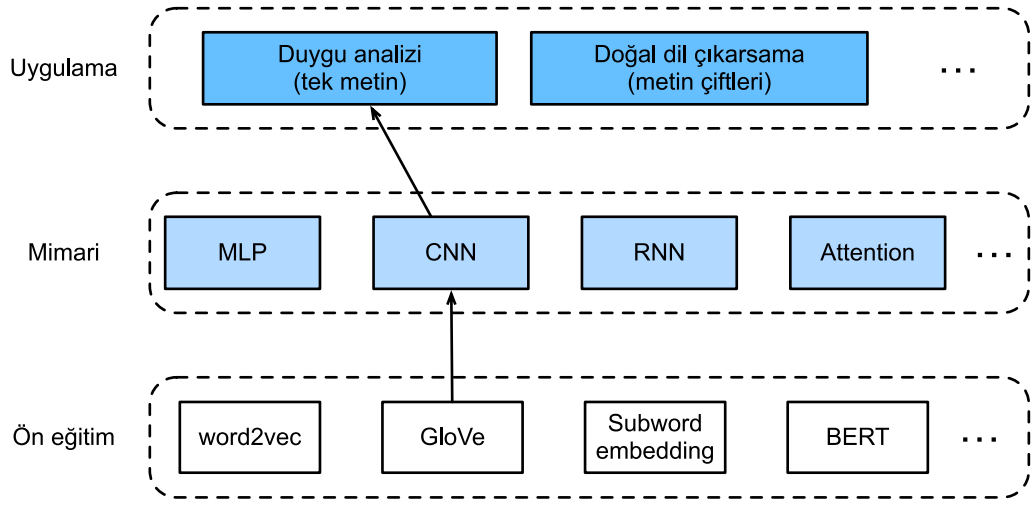


Fig. 15.3.1: Bu bölüm, duygu analizi için önceden eğitilmiş GloVe'i CNN tabanlı bir mimariye besler.

```
import torch
from torch import nn
from d2l import torch as d2l

batch_size = 64
train_iter, test_iter, vocab = d2l.load_data_imdb(batch_size)
```

15.3.1 Tek Boyutlu Evrişimler

Modeli tanıtmadan önce, tek boyutlu bir evrişimin nasıl çalıştığını görelim. Bunun çapraz korelasyon işlemine dayanan iki boyutlu bir evrişimin sadece özel bir durumu olduğunu unutmayın.

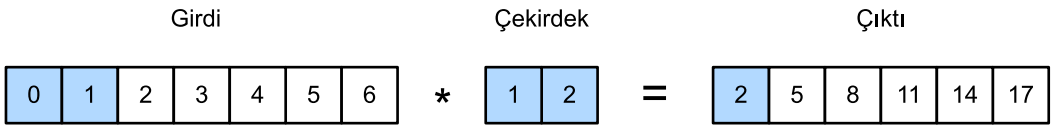


Fig. 15.3.2: Tek boyutlu çapraz korelasyon işlemi. Gölgeli kısımlar, çıktı hesaplaması için kullanılan girdi ve çekirdek tensör elemanlarının yanı sıra ilk çıktı elemanıdır: $0 \times 1 + 1 \times 2 = 2$.

Fig. 15.3.2 içinde gösterildiği gibi, tek boyutlu durumda, evrişim penceresi girdi tensör boyunca soldan sağa doğru kayar. Kayma sırasında, belirli bir konumdaki evrişim penceresinde bulunan girdi alt tensör (örn. Fig. 15.3.2 içindeki 0 ve 1) ve çekirdek tensör (örneğin, Fig. 15.3.2 içindeki 1 ve 2) eleman yönlü çarpılır. Bu çarpımların toplamı, çıktı tensörünün karşılık gelen pozisyonunda tek sayıl değeri (örneğin, Fig. 15.3.2 içindeki $0 \times 1 + 1 \times 2 = 2$) verir.

Aşağıdaki `corr1d` işlevinde tek boyutlu çapraz korelasyon uyguluyoruz. Bir girdi tensör `X` ve bir çekirdek tensör `K` göz önüne alındığında, `Y` çıktı tensörünü döndürür.

```
def corr1d(X, K):
    w = K.shape[0]
    Y = torch.zeros((X.shape[0] - w + 1))
    for i in range(Y.shape[0]):
        Y[i] = (X[i: i + w] * K).sum()
    return Y
```

Yukarıdaki tek boyutlu çapraz korelasyon uygulamasının çıktısını doğrulamak için Fig. 15.3.2 içindeki girdi tensörü `X`'i ve çekirdek tensörü `K`'yi oluşturabiliriz.

```
X, K = torch.tensor([0, 1, 2, 3, 4, 5, 6]), torch.tensor([1, 2])
corr1d(X, K)
```

```
tensor([ 2.,  5.,  8., 11., 14., 17.])
```

Birden çok kanallı tek boyutlu girdi için, evrişim çekirdeğinin aynı sayıda girdi kanalına sahip olması gerekir. Ardından, her kanal için, tek boyutlu çıktı tensörünü üretmek için tüm kanallar üzerindeki sonuçları toplayarak, girdinin tek boyutlu tensörü ve evrişim çekirdeğinin tek boyutlu tensörü üzerinde bir çapraz korelasyon işlemi gerçekleştirilir. Fig. 15.3.3 3 adet girdi kanalıyla tek boyutlu çapraz korelasyon işlemini gösterir.

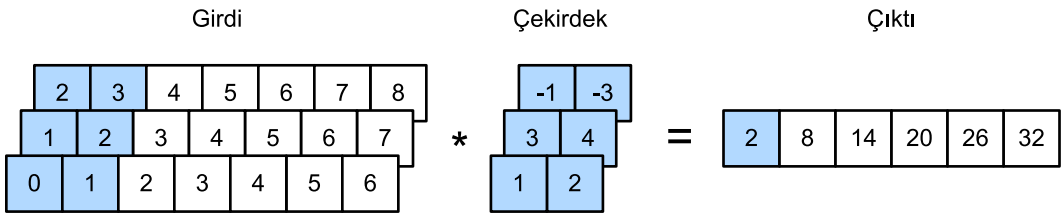


Fig. 15.3.3: 3 adet girdi kanalı ile tek boyutlu çapraz korelasyon işlemi. Gölgeli kısımlar, çıktı hesaplaması için kullanılan girdi ve çekirdek tensör elemanlarının yanı sıra ilk çıktı elemanıdır: $0 \times 1 + 1 \times 2 + 1 \times 3 + 2 \times 4 + 2 \times (-1) + 3 \times (-3) = 2$.

Birden fazla girdi kanalı için tek boyutlu çapraz korelasyon işlemini uygulayabilir ve Fig. 15.3.3 içindeki sonuçları doğrulayabiliriz.

```
def corr1d_multi_in(X, K):
    # İlk olarak, `X` ve `K`'nin 0. boyutu (kanal boyutu) boyunca yineleyin.
    # Ardından, bunları toplayın
    return sum(corr1d(x, k) for x, k in zip(X, K))

X = torch.tensor([[0, 1, 2, 3, 4, 5, 6],
              [1, 2, 3, 4, 5, 6, 7],
              [2, 3, 4, 5, 6, 7, 8]])
K = torch.tensor([[1, 2], [3, 4], [-1, -3]])
corr1d_multi_in(X, K)
```

```
tensor([ 2.,  8., 14., 20., 26., 32.])
```

Çoklu girdi kanallı tek boyutlu çapraz korelasyonların, tek girdi kanallı iki boyutlu çapraz korelasyonlara eşdeğer olduğuna dikkat edin. Göstermek için, Fig. 15.3.3 içindeki çoklu girdi kanallı

tek boyutlu çapraz korelasyonun eşdeğer bir formu, Fig. 15.3.4 içindeki tek girdi kanallı iki boyutlu çapraz korelasyondur; burada evrişim çekirdeğinin yüksekliği girdi tensörünküyle aynı olmalıdır.

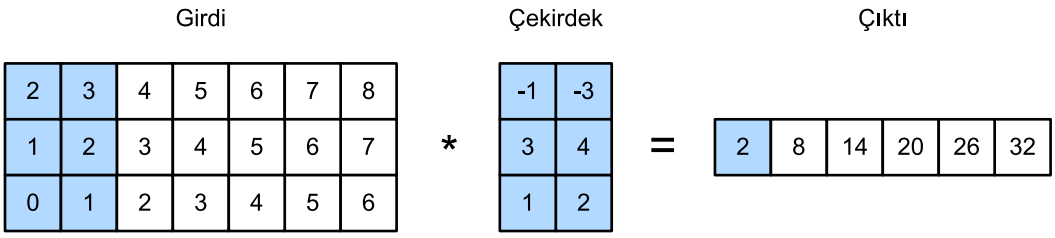


Fig. 15.3.4: Tek bir girdi kanalı ile iki boyutlu çapraz korelasyon işlemi. Gölgeli kısımlar, çıktı hesaplaması için kullanılan girdi ve çekirdek tensör elemanlarının yanı sıra ilk çıktı elemanıdır: $2 \times (-1) + 3 \times (-3) + 1 \times 3 + 2 \times 4 + 0 \times 1 + 1 \times 2 = 2$.

Fig. 15.3.2 ve Fig. 15.3.3 içindeki her iki çıktıda da yalnızca bir kanal vardır. Section 6.4.2 içinde açıklanan çoklu çıktı kanallarına sahip iki boyutlu evrişimlerle aynı şekilde, tek boyutlu evrişimler için de çoklu çıktılı kanallar belirtebiliriz.

15.3.2 Zaman Üzerinden Maksimum Ortaklama

Benzer şekilde, zaman adımları boyunca en önemli öznitelik olarak dizi temsillerinden en yüksek değeri çıkarmak için ortaklamayı kullanabiliriz. textCNN'de kullanılan *zaman üzerinden maksimum ortaklama*, tek boyutlu küresel maksimum ortaklama gibi çalışır (Collobert *et al.*, 2011). Her kanalın farklı zaman adımlarında değerleri depoladığı çok kanallı bir girdi için, her kanaldaki çıktı o kanal için maksimum değerdir. Zaman üzerinden maksimum ortaklamanın, farklı kanallarda farklı sayıda zaman adımına izin verdiğini unutmayın.

15.3.3 TextCNN Modeli

Tek boyutlu evrişim ve zaman üzerinden maksimum ortaklama kullanarak, textCNN modeli girdi olarak önceden eğitilmiş bireysel belirteç temsillerini alır, sonra aşağı akış uygulama için dizi temsillerini elde eder ve dönüştürür.

d boyutlu vektörlerle temsil edilen n belirteci olan tek bir metin dizisi için girdi tensörünün kanal genişliği, yüksekliği ve sayısı sırasıyla n, 1 ve d'dir. textCNN modeli, girdiyi çıktıya aşağıdaki gibi dönüştürür:

1. Birden çok tek boyutlu evrişim çekirdeğini tanımlar ve girdiler üzerinde ayrı olarak evrişim işlemlerini gerçekleştirir. Farklı genişliklere sahip evrişim çekirdekleri, farklı sayıdaki bitişik belirteçler arasındaki yerel öznitelikleri yakalayabilir.
2. Tüm çıktı kanallarında zaman üzerinden maksimum ortaklama gerçekleştirin ve ardından tüm skaler ortaklama çıktılarını bir vektör olarak bitiştirin.
3. Tam bağlı katmanı kullanarak bitiştirilmiş vektörü çıktı kategorilerine dönüştürün. Hattan düşürme, aşırı öğrenmeyi azaltmak için kullanılabilir.

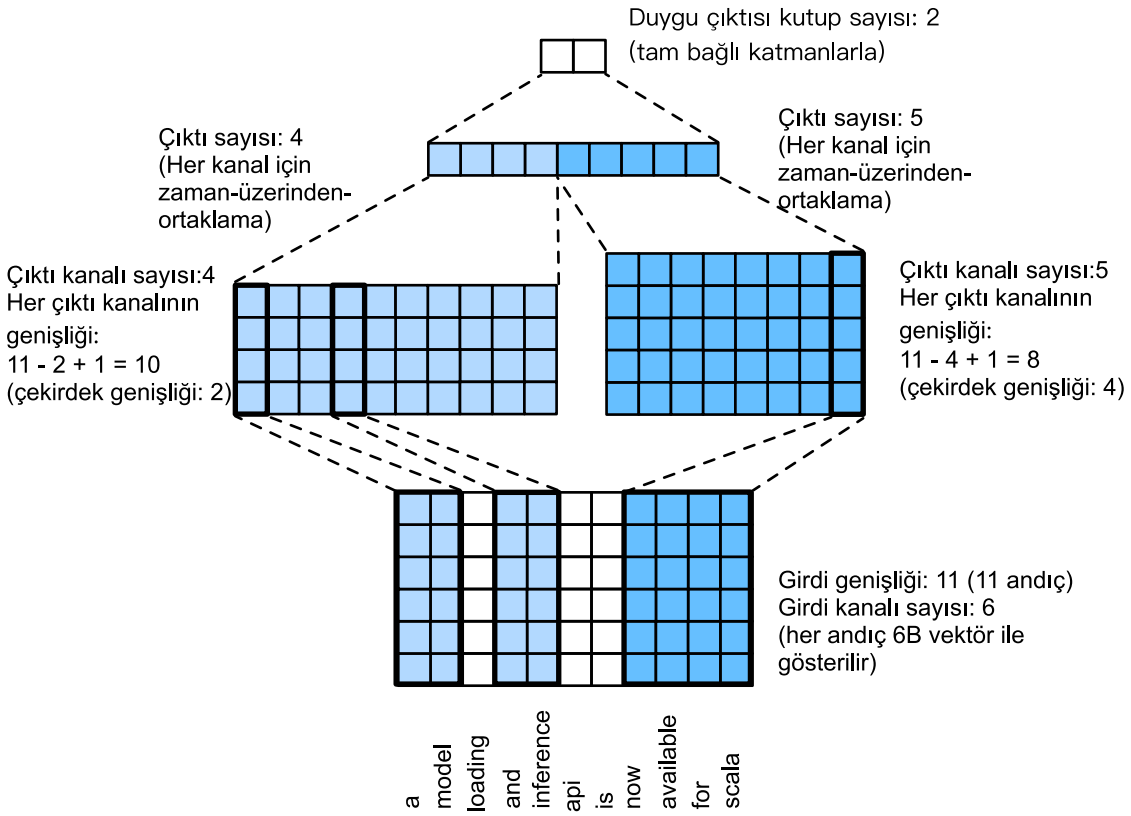


Fig. 15.3.5: textCNN model mimarisi.

Fig. 15.3.5, textCNN'in model mimarisini somut bir örnekle göstermektedir. Girdi, her belirtecin 6 boyutlu vektörlerle temsil edildiği 11 belirteçli bir cümledir. Bu yüzden genişliği 11 olan 6 kanallı bir girdiye sahibiz. Sırasıyla 4 ve 5 çıktı kanalı ile 2 ve 4 genişliklerindeki iki tek boyutlu evrişim çekirdeğini tanımlayın. $11 - 2 + 1 = 10$ genişliğinde 4 çıktı kanalı ve $11 - 4 + 1 = 8$ genişliğinde 5 çıktı kanalı üretirler. Bu 9 kanalın farklı genişliklerine rağmen, zaman üzerinden maksimum ortaklama, bitiştirilmiş 9 boyutlu bir vektör verir ve bu da en sonunda ikili duygu tahminleri için 2 boyutlu bir çıktı vektörüne dönüştürülür.

Modeli Tanımlama

textCNN modelini aşağıdaki sınıfta uyguluyoruz. Section 15.2 içindeki çift yönlü RNN modeliyle karşılaştırıldığında, yinelemeli katmanları evrişimli katmanlarla değiştirmenin yanı sıra, iki gömme katmanı da kullanıyoruz: Biri eğitilebilir ağırlıklara ve diğeri sabit ağırlıklara sahiptir.

```
class TextCNN(nn.Module):
    def __init__(self, vocab_size, embed_size, kernel_sizes, num_channels,
                 **kwargs):
        super(TextCNN, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # Eğitilmeyecek gömme katmanı
        self.constant_embedding = nn.Embedding(vocab_size, embed_size)
        self.dropout = nn.Dropout(0.5)
        self.decoder = nn.Linear(sum(num_channels), 2)
        # Zaman üzerinden maksimum ortaklama katmanının parametresi yoktur,
```

(continues on next page)

(continued from previous page)

```
        # bu nedenle bu örnek paylaşılabilir
        self.pool = nn.AdaptiveAvgPool1d(1)
        self.relu = nn.ReLU()
        # Birden çok tek boyutlu evrişim katmanı oluşturun
        self.convs = nn.ModuleList()
        for c, k in zip(num_channels, kernel_sizes):
            self.convs.append(nn.Conv1d(2 * embed_size, c, k))

    def forward(self, inputs):
        # Vektörler boyunca (iş boyutu, belirteç sayısı, belirteç vektör
        # boyutu) şekilli iki gömme katmanı çıktısını bitiştirin
        embeddings = torch.cat((
            self.embedding(inputs), self.constant_embedding(inputs)), dim=2)
        # Tek boyutlu evrişimli katmanların girdi formatına göre, tensörü
        # yeniden düzenleyin, böylece ikinci boyut kanalları depolar
        embeddings = embeddings.permute(0, 2, 1)
        # Her bir tek boyutlu evrişim katmanı için, maksimum zaman üzerinden
        # ortaklamadan sonra, (iş boyutu, kanal sayısı, 1) şekilli bir tensör
        # elde edilir. Son boyutu kaldırın ve kanallar boyunca bitiştirin.
        encoding = torch.cat([
            torch.squeeze(self.relu(self.pool(conv(embeddings)))), dim=-1)
            for conv in self.convs], dim=1)
        outputs = self.decoder(self.dropout(encoding))
        return outputs
```

Bir textCNN örneği oluşturalım. 3, 4 ve 5 çekirdek genişliklerine sahip 3 adet evrişimli katmana sahiptir ve hepsinde 100 çıktı kanalı bulunur.

```
embed_size, kernel_sizes, nums_channels = 100, [3, 4, 5], [100, 100, 100]
devices = d2l.try_all_gpus()
net = TextCNN(len(vocab), embed_size, kernel_sizes, nums_channels)

def init_weights(m):
    if type(m) in (nn.Linear, nn.Conv1d):
        nn.init.xavier_uniform_(m.weight)

net.apply(init_weights);
```

Önceden Eğitilmiş Sözcük Vektörlerini Yükleme

Section 15.2 ile aynı şekilde, önceden eğitilmiş 100 boyutlu GloVe gömme yerleştirmelerini, ilkletilmiş belirteç temsilleri olarak yükleriz. Bu belirteç temsilleri (gömme ağırlıkları) `embedding`'da eğitilecek ve `constant_embedding`'te sabitlenecektir.

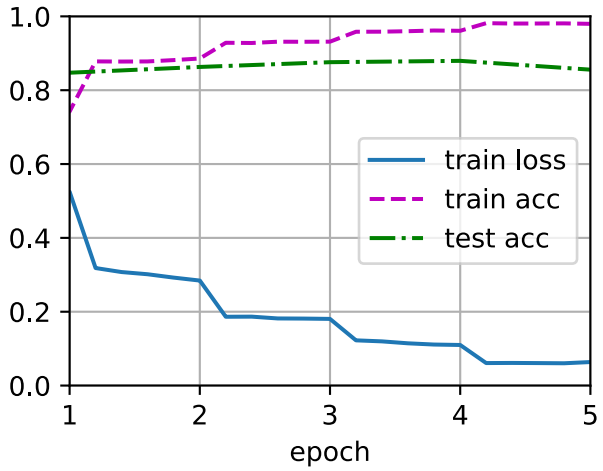
```
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
embeds = glove_embedding[vocab.idx_to_token]
net.embedding.weight.data.copy_(embeds)
net.constant_embedding.weight.data.copy_(embeds)
net.constant_embedding.weight.requires_grad = False
```

Model Eğitimi ve Değerlendirilmesi

Şimdi textCNN modelini duygu analizi için eğitebiliriz.

```
lr, num_epochs = 0.001, 5
trainer = torch.optim.Adam(net.parameters(), lr=lr)
loss = nn.CrossEntropyLoss(reduction="none")
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.064, train acc 0.980, test acc 0.856
3380.2 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



Aşağıda iki basit cümlede duygu tahmini için eğitilmiş model kullanıyoruz.

```
d2l.predict_sentiment(net, vocab, 'this movie is so great')
```

```
'positive'
```

```
d2l.predict_sentiment(net, vocab, 'this movie is so bad')
```

```
'negative'
```

15.3.4 Özet

- Tek boyutlu CNN'ler metinlerdeki n-gramlar gibi yerel öznitelikleri işleyebilir.
- Çoklu girdi kanallı tek boyutlu çapraz korelasyonlar, tek girdi kanallı iki boyutlu çapraz korelasyonlara eşdeğerdir.
- Zaman üzerinden maksimum ortaklama, farklı kanallarda farklı sayıda zaman adımına olanak tanır.

- textCNN modeli, tek boyutlu evrişimli katmanları ve zaman üzerinden maksimum ortaklama katmanları kullanarak tek tek belirteç temsillerini aşağı akış uygulama çıktılarına dönüştürür.

15.3.5 Alıştırmalar

1. Hiper parametreleri ayarlayın ve duygu analizi için iki mimariyi, Section 15.2 içindeki ve bu bölümdeki, örneğin sınıflandırma doğruluğu ve hesaplama verimliliği gibi, karşılaştırın.
2. Section 15.2 alıştırmalarında tanıtılan yöntemleri kullanarak modelin sınıflandırma doğruluğunu daha da iyileştirebilir misiniz?
3. Girdi temsillerine konumsal kodlama ekleyin. Sınıflandırma doğruluğunu arttırıyor mu?

Tartışmalar[206]

15.4 Doğal Dil Çıkarımı ve Veri Kümesi

Section 15.1 içinde, duygu analizi sorununu tartıştık. Bu görev, tek bir metin dizisini, duygu kutupları kümesi gibi önceden tanımlanmış kategorilere sınıflandırmayı amaçlamaktadır. Bununla birlikte, bir cümlenin diğerinden çıkarılıp çıkarılamayacağına karar vermek veya anlamsal olarak eşdeğer cümleleri tanımlayarak fazlalıkları ortadan kaldırmak gerektiğinde, bir metin dizisini nasıl sınıflandıracağını bilmek yetersizdir. Bunun yerine, metin dizileri çiftleri üzerinde akıl yürütebilmemiz gerekir.

15.4.1 Doğal Dil Çıkarımı

Doğal dil çıkarımı, her ikisinin de bir metin dizisi olduğu bir *öncülden* bir *hipotezin* çıkarılıp çıkarılamayacağını inceler. Başka bir deyişle, doğal dil çıkarımı, bir çift metin dizisi arasındaki mantıksal ilişkiyi belirler. Bu tür ilişkiler genellikle üç tipe ayrılır:

- *Gerekçe*: Hipotez, öncülden çıkarılabilir.
- *Çelişki*: Hipotezin olumsuzlanması öncülden çıkarılabilir.
- *Tarafsızlık*: Diğer tüm durumlar.

Doğal dil çıkarımı aynı zamanda metinsel gerekçe görevi olarak da bilinir. Örneğin, aşağıdaki çift, *gerekçe* olarak etiketlenecektir, çünkü hipotezdeki "sevgi göstermek" öncülünden "birbirine sarılmak" çıkarılabilir.

> Öncül: İki kadın birbirlerine sarılıyor.
>
> Hipotez: İki kadın sevgi gösteriyor.

"Kodlama örneğini çalıştırmak", "uyku" yerine "uyumamayı" gösterdiğinden, aşağıda bir *çelişki* örneği verilmiştir.

> Öncül: Bir adam Derin Öğrenmeye Dalış'tan kodlama örneğini çalıştırıyor.
>
> Hipotez: Adam uyuyor.

[206] https://discuss.d2l.ai/t/1425

Üçüncü örnek bir *tarafsızlık* ilişkisini gösterir, çünkü ne "ünlü" ne de "ünlü değil" "bizim için performans gösteriyor" gerçeğinden çıkarılamaz.

Öncül: Müzisyenler bizim için performans gösteriyorlar.

Hipotez: Müzisyenler ünlüdür.

Doğal dil çıkarımı doğal dili anlamak için merkezi bir konu olmuştur. Bilgi getiriminden açık alan sorularını yanıtlamaya kadar geniş uygulamalara sahiptir. Bu sorunu incelemek için popüler bir doğal dil çıkarım kıyaslama veri kümesini araştırarak başlayacağız.

15.4.2 Stanford Doğal Dil Çıkarımı (SNLI) Veri Kümesi

Stanford Doğal Dil Çıkarımı (Stanford Natural Language Inference - SNLI) Külliyatı 500000'in üzerinde etiketli İngilizce cümle çiftleri içeren bir koleksiyondur (Bowman *et al.*, 2015). Ayıklanan SNLI veri kümesini `../data/snli_1.0` yoluna indirip saklıyoruz.

```
import os
import re
import torch
from torch import nn
from d2l import torch as d2l

#@save
d2l.DATA_HUB['SNLI'] = (
    'https://nlp.stanford.edu/projects/snli/snli_1.0.zip',
    '9fcde07509c7e87ec61c640c1b2753d9041758e4')

data_dir = d2l.download_extract('SNLI')
```

Veri Kümesini Okuma

Orijinal SNLI veri kümesi, deneylerimizde gerçekten ihtiyacımız olandan çok daha zengin bilgi içeriyor. Bu nedenle, veri kümesinin yalnızca bir kısmını çıkarmak için bir `read_snli` işlevi tanımlarız, ardından öncüllerin, hipotezlerin ve bunların etiketlerinin listesini döndürürüz.

```
#@save
def read_snli(data_dir, is_train):
    """SNLI veri kümesini öncüller, hipotezler ve etiketler halinde okuyun."""
    def extract_text(s):
        # Bizim tarafımızdan kullanılmayacak bilgileri kaldırın
        s = re.sub('\\(', '', s)
        s = re.sub('\\)', '', s)
        # Ardışık iki veya daha fazla boşluğu boşlukla değiştirin
        s = re.sub('\\s{2,}', ' ', s)
        return s.strip()
    label_set = {'entailment': 0, 'contradiction': 1, 'neutral': 2}
    file_name = os.path.join(data_dir, 'snli_1.0_train.txt'
                             if is_train else 'snli_1.0_test.txt')
    with open(file_name, 'r') as f:
        rows = [row.split('\t') for row in f.readlines()[1:]]
    premises = [extract_text(row[1]) for row in rows if row[0] in label_set]
    hypotheses = [extract_text(row[2]) for row in rows if row[0] in label_set]
```

(continues on next page)

(continued from previous page)

```
    labels = [label_set[row[0]] for row in rows if row[0] in label_set]
    return premises, hypotheses, labels
```

Şimdi ilk 3 çift öncül ve hipotezin yanı sıra onların etiketlerini yazdıralım (“0”, “1” ve “2” sırasıyla “gerekçe”, “çelişki” ve “tarafsızlık“a karşılık gelir).

```
train_data = read_snli(data_dir, is_train=True)
for x0, x1, y in zip(train_data[0][:3], train_data[1][:3], train_data[2][:3]):
    print('premise:', x0)
    print('hypothesis:', x1)
    print('label:', y)
```

```
premise: A person on a horse jumps over a broken down airplane .
hypothesis: A person is training his horse for a competition .
label: 2
premise: A person on a horse jumps over a broken down airplane .
hypothesis: A person is at a diner , ordering an omelette .
label: 1
premise: A person on a horse jumps over a broken down airplane .
hypothesis: A person is outdoors , on a horse .
label: 0
```

Eğitim kümesinin yaklaşık 550000 çifti vardır ve test kümesi yaklaşık 10000 çifte sahiptir. Aşağıdakiler, hem eğitim kümesindeki hem de test kümesindeki üç etiketin, “gerekçe”, “çelişki” ve “tarafsızlık”, dengelendiğini göstermektedir.

```
test_data = read_snli(data_dir, is_train=False)
for data in [train_data, test_data]:
    print([[row for row in data[2]].count(i) for i in range(3)])
```

```
[183416, 183187, 182764]
[3368, 3237, 3219]
```

Veri kümesini Yüklemek İçin Bir Sınıf Tanımlama

Aşağıda, Gluon’daki `Dataset` sınıfından türetilmiş SNLI veri kümesini yüklemek için bir sınıf tanımlıyoruz. Sınıf kurucusundaki `num_steps` bağımsız değişkeni, bir metin dizisinin uzunluğunu belirtir, böylece dizilerin her minigrup işlemi aynı şekle sahip olur. Başka bir deyişle, daha uzun dizideki ilk `num_steps` olanlardan sonraki belirteçler kırpılırken, “<pad>” özel belirteçleri uzunlukları `num_steps` olana kadar daha kısa dizilere eklenecektir. `__getitem__` işlevini uygulayarak, `idx` endeksi ile öncüle, hipoteze ve etikete keyfi olarak erişebiliriz.

```
#@save
class SNLIDataset(torch.utils.data.Dataset):
    """SNLI veri kümesini yüklemek için özelleştirilmiş bir veri kümesi."""
    def __init__(self, dataset, num_steps, vocab=None):
        self.num_steps = num_steps
        all_premise_tokens = d2l.tokenize(dataset[0])
```

(continues on next page)

(continued from previous page)

```
        all_hypothesis_tokens = d2l.tokenize(dataset[1])
        if vocab is None:
            self.vocab = d2l.Vocab(all_premise_tokens + all_hypothesis_tokens,
                                   min_freq=5, reserved_tokens=['<pad>'])
        else:
            self.vocab = vocab
        self.premises = self._pad(all_premise_tokens)
        self.hypotheses = self._pad(all_hypothesis_tokens)
        self.labels = torch.tensor(dataset[2])
        print('read ' + str(len(self.premises)) + ' examples')

    def _pad(self, lines):
        return torch.tensor([d2l.truncate_pad(
            self.vocab[line], self.num_steps, self.vocab['<pad>'])
                         for line in lines])

    def __getitem__(self, idx):
        return (self.premises[idx], self.hypotheses[idx]), self.labels[idx]

    def __len__(self):
        return len(self.premises)
```

Her Şeyi Bir Araya Koymak

Artık `read_snli` işlevini ve `SNLIDataset` sınıfını SNLI veri kümesini indirmek ve eğitim kümesinin kelime dağarcığıyla birlikte hem eğitim hem de test kümeleri için `DataLoader` örneklerini döndürmek için çalıştırabiliriz. Eğitim kümesinden inşa edilen kelime dağarcığını test kümesininki gibi kullanmamız dikkat çekicidir. Sonuç olarak, test kümesindeki herhangi bir yeni belirteç, eğitim kümesinde eğitilen modelce bilinmeyecektir.

```
#@save
def load_data_snli(batch_size, num_steps=50):
    """SNLI veri kümesini indirin ve veri yineleyicilerini ve kelime dağarcığını döndürün."""
    num_workers = d2l.get_dataloader_workers()
    data_dir = d2l.download_extract('SNLI')
    train_data = read_snli(data_dir, True)
    test_data = read_snli(data_dir, False)
    train_set = SNLIDataset(train_data, num_steps)
    test_set = SNLIDataset(test_data, num_steps, train_set.vocab)
    train_iter = torch.utils.data.DataLoader(train_set, batch_size,
                                             shuffle=True,
                                             num_workers=num_workers)
    test_iter = torch.utils.data.DataLoader(test_set, batch_size,
                                            shuffle=False,
                                            num_workers=num_workers)
    return train_iter, test_iter, train_set.vocab
```

Burada toplu iş boyutunu 128'e ve dizi uzunluğunu 50'ye ayarlıyoruz ve veri yineleyicilerini ve kelime dağarcığını elde etmek için `load_data_snli` işlevini çağırıyoruz. Sonra kelime dağarcığı boyutunu yazdırıyoruz.

```
train_iter, test_iter, vocab = load_data_snli(128, 50)
len(vocab)
```

```
read 549367 examples
read 9824 examples
```

```
18678
```

Şimdi ilk minigrubun şeklini yazdırıyoruz. Duygu analizinin aksine, iki girdi, `X[0]` ve `X[1]`, öncül ve hipotez çiftlerini temsil ediyor.

```
for X, Y in train_iter:
    print(X[0].shape)
    print(X[1].shape)
    print(Y.shape)
    break
```

```
torch.Size([128, 50])
torch.Size([128, 50])
torch.Size([128])
```

15.4.3 Özet

- Doğal dil çıkarımı, her ikisinin de bir metin dizisi olduğu, bir öncülden bir hipotezin çıkarılıp çıkarılamayacağını inceler.
- Doğal dil çıkarımlarında, öncüller ve hipotezler arasındaki ilişkiler, gerekçe, çelişki ve tarafsızlık olarak sayılabilir.
- Stanford Doğal Dil Çıkarımı (Stanford Natural Language Inference - SNLI) Külliyatı, doğal dil çıkarımında popüler bir kıyaslama veri kümesidir.

15.4.4 Alıştırmalar

1. Makine çevirisi uzun zamandır bir çıktı çevirisi ile bir gerçek referans değer çeviri arasındaki yüzeysel n-gramlar eşleşmesine dayalı olarak değerlendirilmektedir. Doğal dil çıkarımını kullanarak makine çevirisi sonuçlarını değerlendirmek için bir ölçü tasarlayabilir misiniz?
2. Kelime dağarcığı boyutunu azaltmak için hiper parametreleri nasıl değiştirebiliriz?

Tartışmalar[207]

[207] https://discuss.d2l.ai/t/1388

15.5 Doğal Dil Çıkarımı: Dikkati Kullanma

Section 15.4 içinde doğal dil çıkarım görevini ve SNLI veri kümesini tanıttık. Karmaşık ve derin mimarilere dayanan birçok model göz önüne alındığında, Parikh ve ark. doğal dil çıkarımını dikkat mekanizmaları ile ele almayı önerdi ve bunu "ayrıştırılabilir dikkat modeli" olarak adlandırdı (Parikh *et al.*, 2016). Bu, yinelemeli veya evrişimli katmanları olmayan bir modelle sonuçlanır ve SNLI veri kümesinde o anda çok daha az parametre ile en iyi sonucu elde eder. Bu bölümde, Fig. 15.5.1 içinde tasvir edildiği gibi doğal dil çıkarımı için bu dikkat tabanlı yöntemi (MLP'lerle) açıklayacağız ve uygulayacağız.

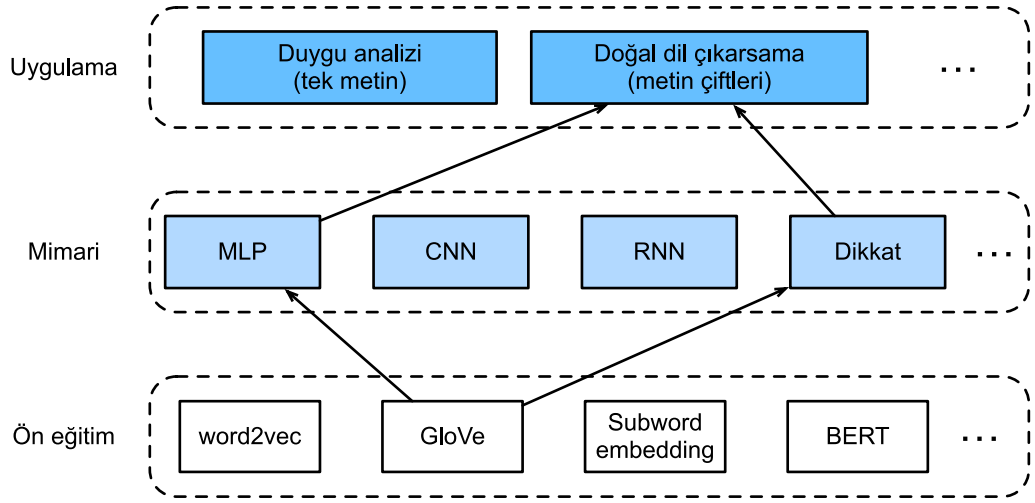


Fig. 15.5.1: Bu bölüm, önceden eğitilmiş GloVe'i, doğal dil çıkarımı için dikkat ve MLP'lere dayalı bir mimariye besler.

15.5.1 Model

Öncüllerdeki ve hipotezlerdeki belirteçlerin sırasını korumaktan daha basit olarak, sadece bir metin dizisindeki belirteçleri diğerindeki her belirteçle hizalayabiliriz ve tersi de geçerlidir, daha sonra öncüller ve hipotezler arasındaki mantıksal ilişkileri tahmin etmek için bu bilgileri karşılaştırabilir ve birleştirebiliriz. Makine çevirisinde kaynak ve hedef cümleler arasında belirteçlerin hizalanmasına benzer şekilde, öncül ve hipotezler arasındaki belirteçlerin hizalanması dikkat mekanizmaları ile düzgün bir şekilde gerçekleştirilebilir.

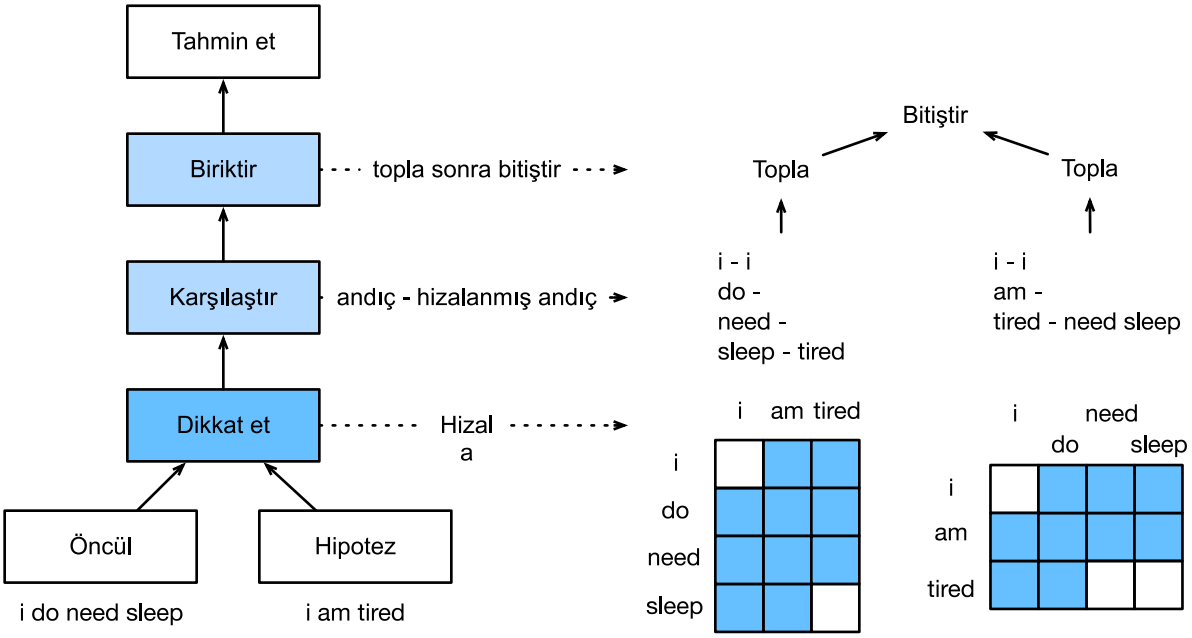


Fig. 15.5.2: Dikkat mekanizmalarını kullanan doğal dil çıkarımı.

Fig. 15.5.2 dikkat mekanizmalarını kullanan doğal dil çıkarım yöntemini tasvir eder. Üst düzeyde, ortak eğitilmiş üç adımdan oluşur: Dikkat etmek, karşılaştırmak ve biriktirmek. Onları aşağıda adım adım göstereceğiz.

```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

Dikkat Etmek

İlk adım, bir metin dizisindeki belirteçleri diğer dizideki her belirteçle hizalamaktır. Öncülün "benim uykuya ihtiyacım var" ve hipotezin "ben yorgunum" olduğunu varsayalım. Anlamsal benzerlik nedeniyle, hipotezdeki "ben" ile öncül içindeki "ben"'i hizalamak ve hipotezdeki "yorgun"'u öncül içindeki "uyku" ile hizalamak isteyebiliriz. Benzer şekilde, öncüldeki "ben"i hipotezdeki "ben" ile hizalamak ve öncüldeki "uyku" ve "ihtiyaç"'ı hipotezdeki "yorgun" ile aynı hizaya getirmek isteyebiliriz. Bu tür bir hizalamanın, ideal olarak büyük ağırlıkların hizalanacak belirteçlerle ilişkilendirildiği, ağırlıklı ortalama kullanılarak *yumuşak* olduğunu unutmayın. Gösterim kolaylığı için, Fig. 15.5.2 böyle bir hizalamayı *sert* bir şekilde gösterir.

Şimdi dikkat mekanizmalarını kullanarak yumuşak hizalamayı daha ayrıntılı olarak tanımlıyoruz. $\mathbf{A} = (\mathbf{a}_1, \ldots, \mathbf{a}_m)$ ve $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ ile, sırasıyla $\mathbf{a}_i, \mathbf{b}_j \in \mathbb{R}^d$ $(i = 1, \ldots, m, j = 1, \ldots, n)$ d boyutlu bir sözcük vektörü olan belirteç sayısı m ve n olan öncülü ve hipotezi belirtin. Yumuşak hizalama için $e_{ij} \in \mathbb{R}$ dikkat ağırlıklarını aşağıdaki gibi hesaplıyoruz

$$e_{ij} = f(\mathbf{a}_i)^\top f(\mathbf{b}_j), \tag{15.5.1}$$

burada f işlevi, aşağıdaki `mlp` işlevinde tanımlanan bir MLP'dir. f'in çıktı boyutu, `mlp`'nin `num_hiddens` argümanı tarafından belirlenir.

```
def mlp(num_inputs, num_hiddens, flatten):
    net = []
    net.append(nn.Dropout(0.2))
    net.append(nn.Linear(num_inputs, num_hiddens))
    net.append(nn.ReLU())
    if flatten:
        net.append(nn.Flatten(start_dim=1))
    net.append(nn.Dropout(0.2))
    net.append(nn.Linear(num_hiddens, num_hiddens))
    net.append(nn.ReLU())
    if flatten:
        net.append(nn.Flatten(start_dim=1))
    return nn.Sequential(*net)
```

(15.5.1) denkleminde f'in $\mathbf{a}_i$ ve $\mathbf{b}_j$ girdilerini girdi olarak bir çift almak yerine ayrı ayrı aldığı vurgulanmalıdır. Bu *ayrıştırma* püf noktası, mn kere uygulama (ikinci dereceden karmaşıklık) yerine f'in yalnızca $m + n$ kere uygulamasına (doğrusal karmaşıklık) yol açar.

(15.5.1) içindeki dikkat ağırlıklarını normalleştirerek, varsayımdaki tüm belirteç vektörlerinin ağırlıklı ortalamasını hesaplıyoruz ve bu hipotezin temsilini elde etmek için i ile endeksli belirteç ile yumuşak bir şekilde hizalanan hipotezin temsilini elde ediyoruz:

$$\boldsymbol{\beta}_i = \sum_{j=1}^{n} \frac{\exp(e_{ij})}{\sum_{k=1}^{n} \exp(e_{ik})} \mathbf{b}_j. \tag{15.5.2}$$

Benzer şekilde, hipotezde j ile endekslenen her belirteç için öncül belirteçlerinin yumuşak hizalamasını hesaplarız:

$$\boldsymbol{\alpha}_j = \sum_{i=1}^{m} \frac{\exp(e_{ij})}{\sum_{k=1}^{m} \exp(e_{kj})} \mathbf{a}_i. \tag{15.5.3}$$

Aşağıda, hipotezlerin (`beta`) girdi öncülleri `A` ile yumuşak hizalamasını ve öncüllerin (`alpha`) girdi hipotezleri `B` ile yumuşak hizalamasını hesaplamak için `Attend` sınıfını tanımlıyoruz.

```
class Attend(nn.Module):
    def __init__(self, num_inputs, num_hiddens, **kwargs):
        super(Attend, self).__init__(**kwargs)
        self.f = mlp(num_inputs, num_hiddens, flatten=False)

    def forward(self, A, B):
        # `A`/`B`'nin şekli: (`batch_size`, A/B dizisindeki belirteç sayısı,
        # `embed_size`)
        # `f_A`/`f_B`'nin şekli: (`batch_size`, A/B dizisindeki belirteç
        # sayısı, `num_hiddens`)
        f_A = self.f(A)
        f_B = self.f(B)
        # `e`'nin şekli: (`batch_size`, A dizisindeki belirteç sayısı,
        # B dizisindeki belirteç sayısı)
        e = torch.bmm(f_A, f_B.permute(0, 2, 1))
        # `beta`'nın şekli: (`batch_size`, A dizisindeki belirteç sayısı,
        # `embed_size`), burada B dizisi, A dizisindeki her bir belirteç ile
        # (`beta`'nın 1. ekseni) yumuşak bir şekilde hizalanır.
        beta = torch.bmm(F.softmax(e, dim=-1), B)
        # `alpha`'nın şekli: (`batch_size`, dizi B'deki belirteç sayısı,
```

(continues on next page)

(continued from previous page)

```
        # `embed_size`), burada A dizisi, B dizisindeki her bir belirteç ile
        # (`alpha`'nın 1. ekseni) yumuşak bir şekilde hizalanır
        alpha = torch.bmm(F.softmax(e.permute(0, 2, 1), dim=-1), A)
        return beta, alpha
```

Karşılaştırmak

Bir sonraki adımda, bir dizideki bir belirteci, bu belirteçle yumuşak bir şekilde hizalanan diğer diziyle karşılaştırırız. Yumuşak hizalamada, bir dizideki tüm belirteçlerin, muhtemelen farklı dikkat ağırlıklarına sahip olsa da, diğer dizideki bir belirteçle karşılaştırılacağını unutmayın. Kolay gösterim için, Fig. 15.5.2 belirteçleri *sert* bir şekilde hizalanmış belirteçlerle eşleştirir. Örneğin, dikkat etme adımının öncüldeki "ihtiyaç" ve "uyku"nun her ikisinin de hipotezdeki "yorgun" ile aynı hizada olduğunu belirlediğini varsayalım, "yorgun–uykuya ihtiyacım var" çifti karşılaştırılacaktır.

Karşılaştırma adımında, bir diziden belirteçlerin (operatör $[\cdot, \cdot]$) ve diğer diziden hizalanmış belirteçlerin bitiştirilmesini g (bir MLP) işlevine besleriz:

$$\begin{aligned}\mathbf{v}_{A,i} &= g([\mathbf{a}_i, \boldsymbol{\beta}_i]), i = 1, \ldots, m\\ \mathbf{v}_{B,j} &= g([\mathbf{b}_j, \boldsymbol{\alpha}_j]), j = 1, \ldots, n.\end{aligned} \tag{15.5.4}$$

(15.5.4) içinde, $\mathbf{v}_{A,i}$, öncüldeki i belirteci ve i belirteci ile yumuşak bir şekilde hizalanan tüm hipotez belirteçleri arasındaki karşılaştırmadır; $\mathbf{v}_{B,j}$ ise hipotezdeki j belirteci ile j belirteci ile yumuşak bir şekilde hizalanmış tüm öncül belirteçler arasındaki karşılaştırmadır. Aşağıdaki `Compare` sınıfı, böyle bir karşılaştırma adımı tanımlar.

```
class Compare(nn.Module):
    def __init__(self, num_inputs, num_hiddens, **kwargs):
        super(Compare, self).__init__(**kwargs)
        self.g = mlp(num_inputs, num_hiddens, flatten=False)

    def forward(self, A, B, beta, alpha):
        V_A = self.g(torch.cat([A, beta], dim=2))
        V_B = self.g(torch.cat([B, alpha], dim=2))
        return V_A, V_B
```

Biriktirmek

İki karşılaştırma vektörü kümesi $\mathbf{v}_{A,i}$ ($i = 1, \ldots, m$) ve $\mathbf{v}_{B,j}$ ($j = 1, \ldots, n$) ile, son adımda mantıksal ilişkiyi çıkarmak için bu tür bilgileri biriktireceğiz. Her iki kümeyi toplayarak başlıyoruz:

$$\mathbf{v}_A = \sum_{i=1}^{m} \mathbf{v}_{A,i}, \quad \mathbf{v}_B = \sum_{j=1}^{n} \mathbf{v}_{B,j}. \tag{15.5.5}$$

Daha sonra mantıksal ilişkinin sınıflandırma sonucunu elde etmek için h (bir MLP) işlevine her iki özetleme sonuçlarının bitiştirilmesini besleriz:

$$\hat{\mathbf{y}} = h([\mathbf{v}_A, \mathbf{v}_B]). \tag{15.5.6}$$

Biriktirme adımı aşağıdaki `Aggregate` sınıfında tanımlanır.

```
class Aggregate(nn.Module):
    def __init__(self, num_inputs, num_hiddens, num_outputs, **kwargs):
        super(Aggregate, self).__init__(**kwargs)
        self.h = mlp(num_inputs, num_hiddens, flatten=True)
        self.linear = nn.Linear(num_hiddens, num_outputs)

    def forward(self, V_A, V_B):
       # Her iki karşılaştırma vektörü kümesini toplayın
        V_A = V_A.sum(dim=1)
        V_B = V_B.sum(dim=1)
        # Her iki özetleme sonucunun bitiştirmesini bir MLP'ye besleyin
        Y_hat = self.linear(self.h(torch.cat([V_A, V_B], dim=1)))
        return Y_hat
```

Her Şeyi Bir Araya Koyma

Dikkat etme, karşılaştırma ve biriktirme adımlarını bir araya getirerek, bu üç adımı birlikte eğitmek için ayrıştırılabilir dikkat modelini tanımlarız.

```
class DecomposableAttention(nn.Module):
    def __init__(self, vocab, embed_size, num_hiddens, num_inputs_attend=100,
                 num_inputs_compare=200, num_inputs_agg=400, **kwargs):
        super(DecomposableAttention, self).__init__(**kwargs)
        self.embedding = nn.Embedding(len(vocab), embed_size)
        self.attend = Attend(num_inputs_attend, num_hiddens)
        self.compare = Compare(num_inputs_compare, num_hiddens)
        # 3 olası çıktı vardır: gerekçe, çelişki ve tarafsızlık
        self.aggregate = Aggregate(num_inputs_agg, num_hiddens, num_outputs=3)

    def forward(self, X):
        premises, hypotheses = X
        A = self.embedding(premises)
        B = self.embedding(hypotheses)
        beta, alpha = self.attend(A, B)
        V_A, V_B = self.compare(A, B, beta, alpha)
        Y_hat = self.aggregate(V_A, V_B)
        return Y_hat
```

15.5.2 Model Eğitimi ve Değerlendirilmesi

Şimdi SNLI veri kümesinde tanımlanmış ayrıştırılabilir dikkat modelini eğiteceğiz ve değerlendireceğiz. Veri kümesini okuyarak başlıyoruz.

Veri Kümesini Okuma

SNLI veri kümesini Section 15.4 içinde tanımlanan işlevi kullanarak indirip okuyoruz. Toplu iş boyutu ve dizi uzunluğu sırasıyla 256 ve 50 olarak ayarlanır.

```
batch_size, num_steps = 256, 50
train_iter, test_iter, vocab = d2l.load_data_snli(batch_size, num_steps)
```

```
read 549367 examples
read 9824 examples
```

Modeli Oluşturma

Girdi belirteçlerini temsil etmek için önceden eğitilmiş 100 boyutlu GloVe gömmeyi kullanıyoruz. Böylece, $\mathbf{a}_i$ ve $\mathbf{b}_j$ vektörlerinin (15.5.1) içindeki boyutunu önceden 100 olarak tanımlarız. (15.5.1) içindeki f ve (15.5.4) içindeki g fonksiyonlarının çıktı boyutu 200 olarak ayarlanmıştır. Ardından bir model örneği oluşturur, parametrelerini ilkler ve girdi belirteçlerinin vektörlerini ilklemek için GloVe gömmesini yükleriz.

```
embed_size, num_hiddens, devices = 100, 200, d2l.try_all_gpus()
net = DecomposableAttention(vocab, embed_size, num_hiddens)
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
embeds = glove_embedding[vocab.idx_to_token]
net.embedding.weight.data.copy_(embeds);
```

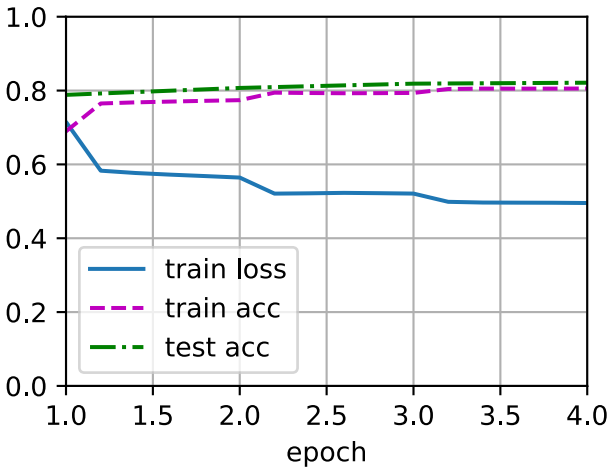
Model Eğitimi ve Değerlendirilmesi

Section 12.5 içindeki `split_batch` işlevinin aksine, metin dizileri (veya imgeler) gibi tek girdileri alan `split_batch` işlevinin aksine, minigruplarda öncüller ve hipotezler gibi birden fazla girdiyi almak için `split_batch_multi_inputs` işlevi tanımlıyoruz.

Şimdi modeli SNLI veri kümesinde eğitebilir ve değerlendirebiliriz.

```
lr, num_epochs = 0.001, 4
trainer = torch.optim.Adam(net.parameters(), lr=lr)
loss = nn.CrossEntropyLoss(reduction="none")
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.495, train acc 0.806, test acc 0.821
18014.7 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



Modeli Kullanma

Son olarak, bir çift öncül ve hipotez arasındaki mantıksal ilişkiyi ortaya çıkaracak tahmin işlevini tanımlayın.

```
#@save
def predict_snli(net, vocab, premise, hypothesis):
    """Öncül ve hipotez arasındaki mantıksal ilişkiyi tahmin edin."""
    net.eval()
    premise = torch.tensor(vocab[premise], device=d2l.try_gpu())
    hypothesis = torch.tensor(vocab[hypothesis], device=d2l.try_gpu())
    label = torch.argmax(net([premise.reshape((1, -1)),
                          hypothesis.reshape((1, -1))]), dim=1)
    return 'entailment' if label == 0 else 'contradiction' if label == 1 \
            else 'neutral'
```

Eğitilmiş modeli, örnek bir cümle çifti için doğal dil çıkarım sonucunu elde etmek için kullanabiliriz.

```
predict_snli(net, vocab, ['he', 'is', 'good', '.'], ['he', 'is', 'bad', '.'])
```

```
'contradiction'
```

15.5.3 Özet

- Ayrıştırılabilir dikkat modeli, öncüller ve hipotezler arasındaki mantıksal ilişkileri tahmin etmek için üç adımdan oluşur: Dikkat etmek, karşılaştırmak ve biriktirmek.
- Dikkat mekanizmalarıyla, bir metin dizisinde belirteçleri diğerindeki her belirteçle hizalayabiliriz ve tam tersi de geçerlidir. Bu hizalama, ideal olarak büyük ağırlıkların hizalanacak belirteçlerle ilişkilendirildiği ağırlıklı ortalama kullanıldığından yumuşaktır.
- Ayrıştırma püf noktası, dikkat ağırlıklarını hesaplarken ikinci dereceden karmaşıklıktan daha fazla arzu edilen doğrusal bir karmaşıklığa yol açar.

- Doğal dil çıkarımı gibi aşağı akış doğal dil işleme görevi için girdi temsili olarak önceden eğitilmiş sözcük vektörlerini kullanabiliriz.

15.5.4 Alıştırmalar

1. Modeli diğer hiper parametre kombinasyonları ile eğitin. Test kümesinde daha iyi doğruluk elde edebilir misiniz?
2. Doğal dil çıkarımı için ayrıştırılabilir dikkat modelinin en büyük sakıncaları nelerdir?
3. Herhangi bir cümle çifti için anlamsal benzerlik düzeyini (örneğin, 0 ile 1 arasında sürekli bir değer) elde etmek istediğimizi varsayalım. Veri kümesini nasıl toplayıp etiketleyeceğiz? Dikkat mekanizmaları ile bir model tasarlayabilir misiniz?

Tartışmalar[208]

15.6 Dizi Düzeyinde ve Belirteç Düzeyinde Uygulamalar için BERT İnce Ayarı

Bu bölümün önceki kısımlarında, RNN'ler, CNN'ler, dikkat ve MLP'lere dayanan doğal dil işleme uygulamaları için farklı modeller tasarladık. Bu modeller, alan veya zaman kısıtlaması olduğunda faydalıdır, ancak her doğal dil işleme görevi için belirli bir model oluşturmak pratik olarak imkansızdır. Section 14.7 içinde, çok çeşitli doğal dil işleme görevleri için asgari mimari değişiklikleri gerektiren bir ön eğitim modeli olan BERT tanıttık. Bir yandan, önerildiği sırasında BERT çeşitli doğal dil işleme görevlerinde son teknolojiyi iyileştirdi. Öte yandan, Section 14.9 içinde belirtildiği gibi, orijinal BERT modelinin iki versiyonu 110 milyon ve 340 milyon parametre ile geliyor. Bu nedenle, yeterli hesaplama kaynağı olduğunda, aşağı akış doğal dil işleme uygulamaları için BERT ince ayarını düşünebiliriz.

Aşağıda, doğal dil işleme uygulamalarının bir alt kümesini dizi düzeyinde ve belirteç düzeyinde genelleştiriyoruz. Dizi düzeyinde, metin girdisinin BERT temsilini tek metin sınıflandırmasında ve metin çifti sınıflandırmasında veya bağlanımda çıktı etiketine nasıl dönüştüreceğimizi gösteriyoruz. Belirteç düzeyinde, metin etiketleme ve soru yanıtlama gibi yeni uygulamaları kısaca tanıtacağız ve BERT'in girdilerini nasıl temsil edebileceğine ve çıktı etiketlerine dönüştürebileceğine ışık tutacağız. İnce ayar sırasında BERT tarafından farklı uygulamalar için gerekli olan "asgari mimari değişiklikleri" ek tam bağlı katmanlardır. Aşağı akış uygulamasının gözetimli öğrenmesi sırasında ek katmanların parametreleri sıfırdan öğrenilirken, önceden eğitilmiş BERT modelindeki tüm parametreler ince ayarlanır.

[208] https://discuss.d2l.ai/t/1530

15.6.1 Tek Metin Sınıflandırması

Tek metin sınıflandırma girdi olarak tek bir metin dizisi alır ve sınıflandırma sonucunu çıkarır. Bu bölümde incelediğimiz duygu analizinin yanısıra, Dilsel Kabul Edilebilirlik Külliyatı (Corpus of Linguistic Acceptability - CoLA), belirli bir cümlenin dilbilgisi açısından kabul edilebilir olup olmadığına karar veren tek metin sınıflandırması için bir veri kümedir (Warstadt *et al.*, 2019). Mesela, "Çalışmalıyım." kabul edilebilir ama "Çalışmalıyorum" değil.

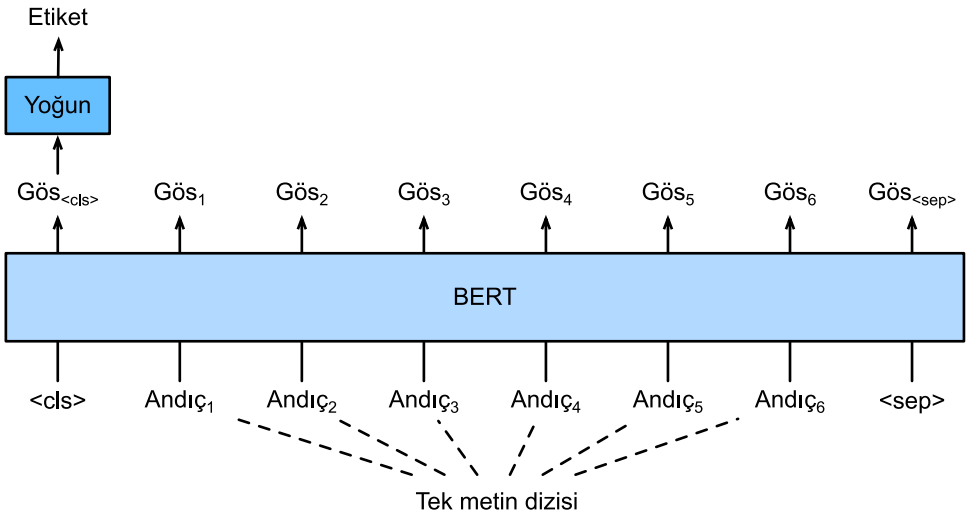


Fig. 15.6.1: Duygu analizi ve dilsel kabul edilebilirliği test etme gibi tek metin sınıflandırma uygulamaları için BERT ince ayarı. Tek girdi metninin altı belirteci olduğunu varsayalım.

Section 14.7 BERT girdi temsilini açıklar. BERT girdi dizisi, hem tek metini hem de metin çiftlerini açıkça temsil eder; burada "<cls>" özel sınıflandırma belirteci dizi sınıflandırma için kullanılır ve "<sep>" özel sınıflandırma belirteci tek metnin sonunu işaretler veya bir metin çiftini ayırır. Fig. 15.6.1 içinde gösterildiği gibi, tek metin sınıflandırma uygulamalarında, "<cls>" özel sınıflandırma belirtecinin BERT temsili, girdi metni dizisinin tüm bilgilerini kodlar. Tek girdi metninin temsili olarak, tüm ayrık etiket değerlerinin dağılımını yapmak için tam bağlı (yoğun) katmanlardan oluşan küçük bir MLP içine beslenecektir.

15.6.2 Metin Çifti Sınıflandırma veya Bağlanım

Bu bölümde doğal dil çıkarımını da inceledik. Bir çift metni sınıflandıran bir uygulama türü olan *metin çifti sınıflandırma*ya aittir.

Girdi olarak bir metin çifti alıp sürekli bir değer çıktısı veren *anlamsal metinsel benzerlik*, popüler bir *metin çifti bağlanımı* görevidir. Bu görev cümlelerin anlamsal benzerliğini ölçer. Örneğin, Anlamsal Metinsel Benzerlik Kıyaslama veri kümelerinde, bir çift cümlenin benzerlik puanı 0 (anlam çakışmayan) ile 5 (yani eşdeğerlik) arasında değişen bir sıra ölçeğidir (Cer *et al.*, 2017). Amaç bu skorları tahmin etmektir. Anlamsal Metin Benzerliği Kıyaslama veri kümesinden örnekler şunlardır (cümle 1, cümle 2, benzerlik puanı):

- "Bir uçak kalkıyor.", "Bir tayyare kalkıyor.", 5.000;
- "Bir kadın bir şeyler yiyor.", "Bir kadın et yiyor.", 3.000;
- "Bir kadın dans ediyor.", "Bir adam konuşuyor.", 0.000.

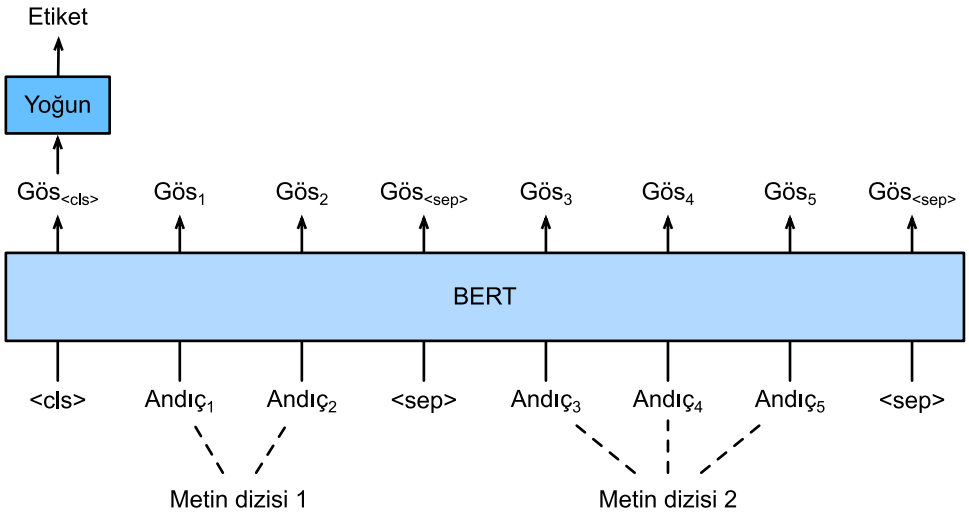


Fig. 15.6.2: Doğal dil çıkarımı ve anlamsal metin benzerliği gibi metin çifti sınıflandırması veya bağlanım uygulamaları için BERT ince ayarı. Girdi metni çiftinin iki ve üç belirteci olduğunu varsayalım.

Fig. 15.6.1 içindeki tek metin sınıflandırmasıyla karşılaştırıldığında, Fig. 15.6.2 içindeki metin çifti sınıflandırması için BERT ince ayarı girdi temsilinde farklıdır. Anlamsal metinsel benzerlik gibi metin çifti bağlanım görevleri için, sürekli bir etiket değeri çıktısı ve ortalama kare kaybının kullanılması gibi önemsiz değişiklikler uygulanabilir: Bunlar bağlanım için yaygındır.

15.6.3 Metin Etiketleme

Şimdi, her belirtece bir etiketin atandığı *metin etiketleme* gibi belirteç düzeyinde görevleri ele alalım. Metin etiketleme görevleri arasında *konuşma parçası etiketleme*, her kelimeye cümledeki kelimenin rolüne göre bir konuşma parçası etiketi atar (örn., sıfat ve belirleyici). Örneğin, Penn Treebank II etiket kümesine göre, "John Smith'in arabası yeni" cümlesi "NNP (isim, uygun tekil) NNP POS (iyelik sonu) NN (isim, tekil veya kitle) VB (temel form, fiil) JJ (sıfat)" olarak etiketlenmelidir.

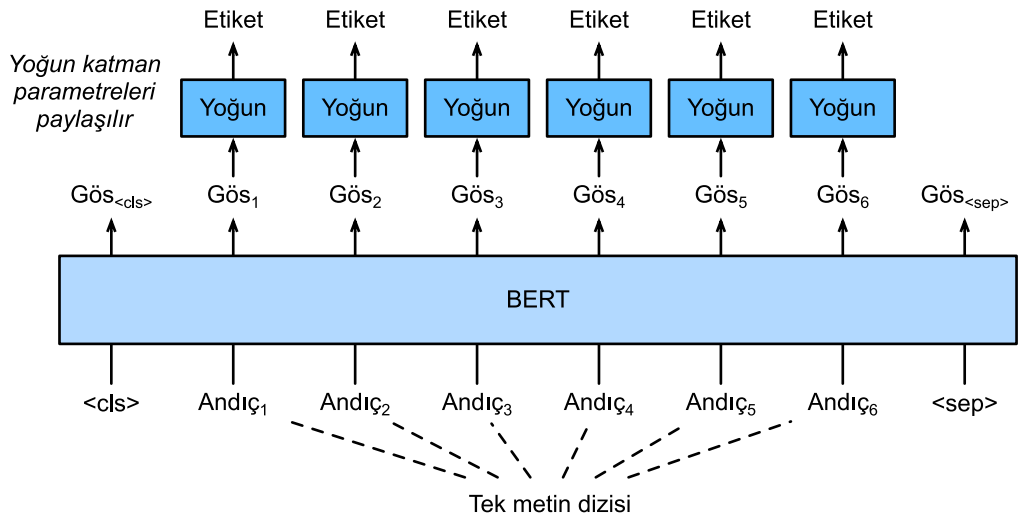


Fig. 15.6.3: Konuşma parçası etiketleme gibi metin etiketleme uygulamaları için BERT ince ayarı. Tek girdi metninin altı belirteci olduğunu varsayalım.

Metin etiketleme uygulamaları için BERT ince ayarı Fig. 15.6.3 içinde gösterilmiştir. Fig. 15.6.1 ile karşılaştırıldığında, tek fark, metin etiketlemede, girdi metninin *her belirtecinin* BERT gösteriminin, konuşma parçası etiketi gibi, belirteç etiketinin çıktısını almak için aynı ek tam bağlı katmanlara beslenmesidir.

15.6.4 Soru Yanıtlama

Başka bir belirteç düzeyinde uygulama olan *soru cevaplama* okuma anlama yeteneklerini yansıtır. Örneğin, Stanford Soru Yanıtlama Veri Kümesi (SQuAD v1.1), her sorunun cevabının, sorunun ilgili olduğu pasajdan sadece bir metin parçası (metin aralığı) olduğu okuma parçaları ve sorulardan oluşur (Rajpurkar *et al.*, 2016). Açıklamak için, bir metin parçası "Bazı uzmanlar bir maskenin etkinliğinin sonuçsuz olduğunu bildiriyor. Bununla birlikte, maske üreticileri, N95 solunum maskeleri gibi ürünlerinin virüse karşı koruyabileceği konusunda ısrar ediyorlar." ve bir soru "N95 solunum maskelerinin virüse karşı koruyabileceğini kim söylüyor?" düşünün. Cevap, parçadaki "maske üreticileri" metin alanı olmalıdır. Böylece, SQuAD v1.1'deki hedef, bir çift soru ve metin parçası verildiğinde parçadaki metin aralığının başlangıcını ve sonunu tahmin etmektir.

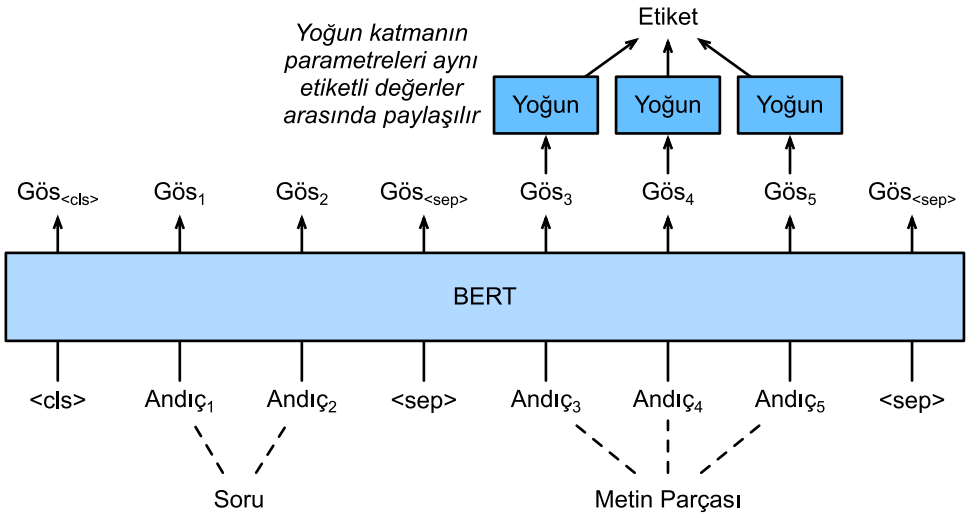


Fig. 15.6.4: Soru cevaplama için BERT ince ayarı. Girdi metni çiftinin iki ve üç belirteci olduğunu varsayalım.

Soru yanıtlamada BERT ince ayarı yapmak için, soru ve metin parçası BERT girdisinde sırasıyla birinci ve ikinci metin dizisi olarak paketlenir. Metin aralığının başlangıcının konumunu tahmin etmek için, aynı ek tam bağlı katman, metin parçasındaki i konumunun herhangi bir belirtecinin BERT temsilini s_i skaler puanına dönüştürecektir. Tüm metin parçası belirteçlerinin bu puanları, ayrıca, softmaks işlemi tarafından bir olasılık dağılımına dönüştürülür, böylece metin parçasındaki her i belirteç konumu, p_i metin aralığının başlangıcı olma olasılığına atanır. Metin aralığının sonunu tahmin etmek, ek tam bağlı katmandaki parametrelerin, başlangıcı tahmin etmek için kullanılanlardan bağımsız olması dışında, yukarıdakiyle aynıdır. Sonunu tahmin ederken, herhangi bir i konumunun metin parçası belirteci, aynı tam bağlı katman tarafından bir e_i skaler skoruna dönüştürülür. Fig. 15.6.4, soru yanıtlama için BERT ince ayarını tasvir eder.

Soru yanıtlamak için, gözetimli öğrenmenin eğitim amaç işlevi, gerçek referans değerin başlangıç ve bitiş pozisyonlarının logaritmik olabilirliğini en üst düzeye çıkarmak kadar basittir. Aralığı tahmin ederken, i konumundan j ($i \leq j$) konumuna kadar geçerli bir aralık için $s_i + e_j$ puanını hesaplayabilir ve en yüksek puanlı aralığı çıktı olarak verebiliriz.

15.6.5 Özet

- BERT, tek metin sınıflandırması (örn. Duygu analizi ve dilsel kabul edilebilirliği test etme), metin çifti sınıflandırması veya bağlanım (örneğin, doğal dil çıkarım ve anlamsal metinsel benzerlik), metin etiketleme (örn. konuşma bölümü etiketleme) ve soru yanıtlama gibi dizi düzeyinde ve belirteç düzeyinde doğal dil işleme uygulamaları için asgari mimari değişiklikleri (ek tam bağlı katmanlar) gerektirir.
- Aşağı akış uygulamasının gözetimli öğrenme sırasında ek katmanların parametreleri sıfırdan öğrenilirken, önceden eğitilmiş BERT modelindeki tüm parametreler ince ayarlanır.

15.6.6 Alıştırmalar

1. Haber makaleleri için bir arama motoru algoritması tasarlayalım. Sistem bir sorgu aldığında (örneğin, "koronavirüs salgını sırasında petrol endüstrisi"), sorgu ile en alakalı haber makalelerinin sıralı listesini döndürmelidir. Büyük bir haber makalesi havuzumuz ve çok sayıda sorgumuz olduğunu varsayalım. Sorunu basitleştirmek için en alakalı makalenin her sorgu için etiketlenmiş olduğunu varsayalım. Algoritma tasarımında negatif örneklemeyi (bkz. Section 14.2.1) ve BERT'i nasıl uygulayabiliriz?
2. Dil modellerinin eğitiminde BERT'ten nasıl yararlanabiliriz?
3. Makine çevirisinde BERT'ten yararlanabilir miyiz?

Tartışmalar[209]

15.7 Doğal Dil Çıkarımı: BERT İnce Ayarı

Bu ünitenin önceki kısımlarında, SNLI veri kümesinde (Section 15.4 içindde açıklandığı gibi) doğal dil çıkarım görevi için dikkat tabanlı bir mimari (Section 15.5 içinde) tasarladık. Şimdi BERT ince ayarı yaparak bu görevi tekrar gözden geçiriyoruz. Section 15.6 içinde tartışıldığı gibi, doğal dil çıkarımı bir dizi düzeyinde metin çifti sınıflandırma sorunudur ve BERT ince ayarı yalnızca Fig. 15.7.1 içinde gösterildiği gibi ek bir MLP tabanlı mimari gerektirir.

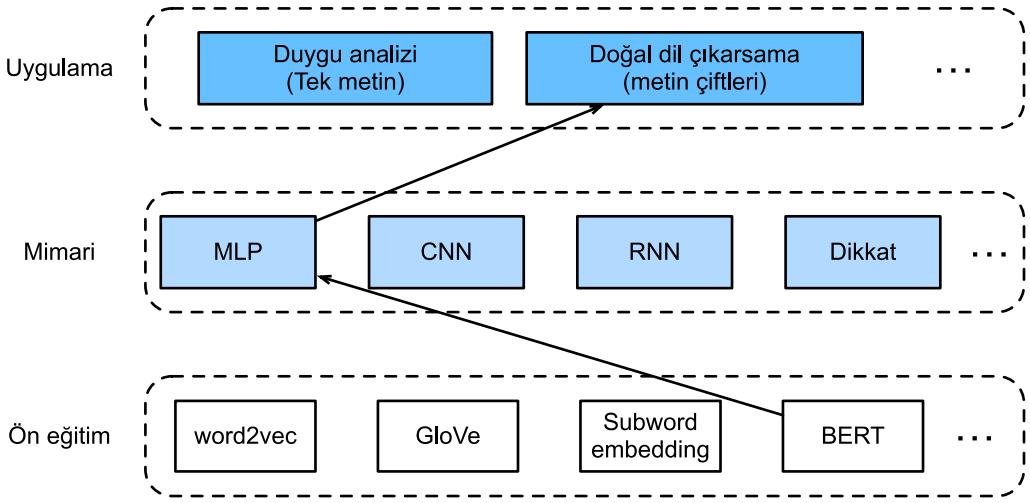


Fig. 15.7.1: Bu bölüm, doğal dil çıkarımı için önceden eğitilmiş BERT'i MLP tabanlı bir mimariye besler.

Bu bölümde, BERT'in önceden eğitilmiş küçük bir sürümünü indireceğiz, ardından SNLI veri kümesinde doğal dil çıkarımı için ince ayarını yapacağız.

```
import json
import multiprocessing
import os
import torch
from torch import nn
from d2l import torch as d2l
```

[209] https://discuss.d2l.ai/t/396

15.7.1 Önceden Eğitilmiş BERT'i Yükleme

BERT'i Section 14.8 ve Section 14.9 içinde WikiText-2 veri kümelerinde nasıl ön eğitebileceğimizi açıkladık (orijinal BERT modelinin çok daha büyük külliyatlar üzerinde önceden eğitildiğini unutmayın). Section 14.9 içinde tartışıldığı gibi, orijinal BERT modelinin yüz milyonlarca parametresi vardır. Aşağıda, önceden eğitilmiş BERT'in iki versiyonunu sunuyoruz: "bert.base" ince ayar yapmak için çok sayıda hesaplama kaynağı gerektiren orijinal BERT temel modeli kadar büyüktür, "bert.small" ise gösterimi kolaylaştırmak için küçük bir versiyondur.

```
d2l.DATA_HUB['bert.base'] = (d2l.DATA_URL + 'bert.base.torch.zip',
                             '225d66f04cae318b841a13d32af3acc165f253ac')
d2l.DATA_HUB['bert.small'] = (d2l.DATA_URL + 'bert.small.torch.zip',
                              'c72329e68a732bef0452e4b96a1c341c8910f81f')
```

Önceden eğitilmiş BERT modeli, sözcük dağarcığını tanımlayan bir "vocab.json" dosyasını ve önceden eğitilmiş parametrelerin olduğu "pretrained.params" dosyasını içerir. Önceden eğitilmiş BERT parametrelerini yüklemek için aşağıdaki `load_pretrained_model` işlevini uyguluyoruz.

```
def load_pretrained_model(pretrained_model, num_hiddens, ffn_num_hiddens,
                          num_heads, num_layers, dropout, max_len, devices):
    data_dir = d2l.download_extract(pretrained_model)
    # Önceden tanımlanmış kelimeleri yüklemek için boş bir kelime
    # dağarcığı tanımla
    vocab = d2l.Vocab()
    vocab.idx_to_token = json.load(open(os.path.join(data_dir, 'vocab.json')))
    vocab.token_to_idx = {token: idx for idx, token in enumerate(
        vocab.idx_to_token)}
    bert = d2l.BERTModel(len(vocab), num_hiddens, norm_shape=[256],
                         ffn_num_input=256, ffn_num_hiddens=ffn_num_hiddens,
                         num_heads=4, num_layers=2, dropout=0.2,
                         max_len=max_len, key_size=256, query_size=256,
                         value_size=256, hid_in_features=256,
                         mlm_in_features=256, nsp_in_features=256)
    # Önceden eğitilmiş BERT parametrelerini yükle
    bert.load_state_dict(torch.load(os.path.join(data_dir,
                                                 'pretrained.params')))
    return bert, vocab
```

Makinelerin çoğunda gösterimi kolaylaştırmak için, bu bölümde önceden eğitilmiş BERT'in küçük versiyonunu ("bert.small") yükleyip ona ince ayar yapacağız. Alıştırmada, test doğruluğunu önemli ölçüde artırmak için çok daha büyük "bert.base"'in nasıl ince ayar yapılacağını göstereceğiz.

```
devices = d2l.try_all_gpus()
bert, vocab = load_pretrained_model(
    'bert.small', num_hiddens=256, ffn_num_hiddens=512, num_heads=4,
    num_layers=2, dropout=0.1, max_len=512, devices=devices)
```

15.7.2 BERT İnce Ayarı İçin Veri Kümesi

SNLI veri kümesinde aşağı akış görevi doğal dil çıkarımı için, özelleştirilmiş bir veri kümesi sınıfı `SNLIBERTDataset` tanımlıyoruz. Her örnekte, öncül ve hipotez bir çift metin dizisi oluşturur ve Fig. 15.6.2 üzerinde tasvir edildiği gibi bir BERT girdi dizisine paketlenir. Hatırlayalım; Section 14.7.4 bölüm kimlikleri bir BERT girdi dizisinde öncül ve hipotezi ayırt etmek için kullanılır. Bir BERT girdi dizisinin (`max_len`) önceden tanımlanmış maksimum uzunluğu ile, girdi metin çiftinin daha uzun olanının son belirteci, `max_len` karşılanana kadar ortadan kaldırılmaya devam eder. BERT ince ayarında SNLI veri kümesinin oluşturulmasını hızlandırmak için, paralel olarak eğitim veya test örnekleri oluşturmada 4 işçi işlem kullanıyoruz.

```
class SNLIBERTDataset(torch.utils.data.Dataset):
    def __init__(self, dataset, max_len, vocab=None):
        all_premise_hypothesis_tokens = [[
            p_tokens, h_tokens] for p_tokens, h_tokens in zip(
            *[d2l.tokenize([s.lower() for s in sentences])
              for sentences in dataset[:2]])]

        self.labels = torch.tensor(dataset[2])
        self.vocab = vocab
        self.max_len = max_len
        (self.all_token_ids, self.all_segments,
         self.valid_lens) = self._preprocess(all_premise_hypothesis_tokens)
        print('read ' + str(len(self.all_token_ids)) + ' examples')

    def _preprocess(self, all_premise_hypothesis_tokens):
        pool = multiprocessing.Pool(4)  # Use 4 worker processes
        out = pool.map(self._mp_worker, all_premise_hypothesis_tokens)
        all_token_ids = [
            token_ids for token_ids, segments, valid_len in out]
        all_segments = [segments for token_ids, segments, valid_len in out]
        valid_lens = [valid_len for token_ids, segments, valid_len in out]
        return (torch.tensor(all_token_ids, dtype=torch.long),
                torch.tensor(all_segments, dtype=torch.long),
                torch.tensor(valid_lens))

    def _mp_worker(self, premise_hypothesis_tokens):
        p_tokens, h_tokens = premise_hypothesis_tokens
        self._truncate_pair_of_tokens(p_tokens, h_tokens)
        tokens, segments = d2l.get_tokens_and_segments(p_tokens, h_tokens)
        token_ids = self.vocab[tokens] + [self.vocab['<pad>']] \
                            * (self.max_len - len(tokens))
        segments = segments + [0] * (self.max_len - len(segments))
        valid_len = len(tokens)
        return token_ids, segments, valid_len

    def _truncate_pair_of_tokens(self, p_tokens, h_tokens):
        # BERT girdisi için '<CLS>', '<SEP>' ve '<SEP>' belirteçleri
        # için yer ayırın
        while len(p_tokens) + len(h_tokens) > self.max_len - 3:
            if len(p_tokens) > len(h_tokens):
                p_tokens.pop()
            else:
                h_tokens.pop()
```

(continues on next page)

(continued from previous page)

```
    def __getitem__(self, idx):
        return (self.all_token_ids[idx], self.all_segments[idx],
                self.valid_lens[idx]), self.labels[idx]

    def __len__(self):
        return len(self.all_token_ids)
```

SNLI veri kümesini indirdikten sonra, `SNLIBERTDataset` sınıfından örnek yaratarak eğitim ve test örnekleri oluşturuyoruz. Bu tür örnekler, doğal dil çıkarımlarının eğitimi ve test edilmesi sırasında minigruplarda okunacaktır.

```
# Yetersiz bellek hatası varsa `batch_size` değerini azaltın. Orijinal BERT
# modelinde, `max_len` = 512'dir.
batch_size, max_len, num_workers = 512, 128, d2l.get_dataloader_workers()
data_dir = d2l.download_extract('SNLI')
train_set = SNLIBERTDataset(d2l.read_snli(data_dir, True), max_len, vocab)
test_set = SNLIBERTDataset(d2l.read_snli(data_dir, False), max_len, vocab)
train_iter = torch.utils.data.DataLoader(train_set, batch_size, shuffle=True,
                                   num_workers=num_workers)
test_iter = torch.utils.data.DataLoader(test_set, batch_size,
                                  num_workers=num_workers)
```

```
read 549367 examples
read 9824 examples
```

15.7.3 BERT İnce Ayarı

Fig. 15.6.2 içinde belirtildiği gibi, doğal dil çıkarımı için BERT ince ayarı yalnızca iki tam bağlı katmandan oluşan ek bir MLP gerektirir (aşağıdaki `BERTClassifier` sınıfında `self.hidden` ve `self.output`). Bu MLP, hem öncül hem de hipotezin bilgilerini kodlayan özel "<cls>" belirtecinin BERT temsilini, doğal dil çıkarımının üç çıktısına dönüştürür: Gerekçe, çelişki ve tarafsızlık.

```
class BERTClassifier(nn.Module):
    def __init__(self, bert):
        super(BERTClassifier, self).__init__()
        self.encoder = bert.encoder
        self.hidden = bert.hidden
        self.output = nn.Linear(256, 3)

    def forward(self, inputs):
        tokens_X, segments_X, valid_lens_x = inputs
        encoded_X = self.encoder(tokens_X, segments_X, valid_lens_x)
        return self.output(self.hidden(encoded_X[:, 0, :]))
```

Aşağıda, önceden eğitilmiş BERT modeli `bert`, aşağı akış uygulaması için `BERTClassifier` örneği `net`'e beslenir. BERT ince ayarının ortak uygulamalarında, sadece ek MLP'nin (`net.output`) çıktı tabakasının parametreleri sıfırdan öğrenilecektir. Önceden eğitilmiş BERT kodlayıcısının (`net.encoder`) ve ek MLP'nin gizli katmanının (`net.hidden`) tüm parametreleri ince ayarlanacaktır.

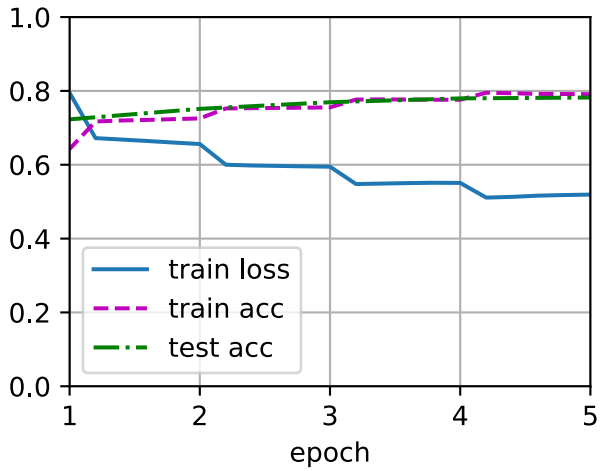
```
net = BERTClassifier(bert)
```

Section 14.7 içinde hem `MaskLM` sınıfının hem de `NextSentencePred` sınıfının konuşlanmış MLP'lerinde parametrelere sahip olduğunu hatırlayın. Bu parametreler önceden eğitilmiş `bert` BERT modelinin ve dolayısıyla `net` içindeki parametrelerin bir parçasıdır. Bununla birlikte, bu tür parametreler sadece maskeli dil modelleme kaybını ve ön eğitim sırasında bir sonraki cümle tahmini kaybını hesaplamak içindir. Bu iki kayıp işlevi, aşağı akış uygulamalarının ince ayarını yapmakla ilgisizdir, bu nedenle, BERT ince ayarı yapıldığında `MaskLM` ve `NextSentencePred`'de kullanılan MLP'lerin parametreleri güncellenmez (bozulmaz).

Dondurulmuş gradyanlara sahip parametrelere izin vermek için `ignore_stale_grad=True` işareti `d2l.train_batch_ch13`'in `step` işlevinde ayarlanır. Bu işlevi, SNLI'nin eğitim kümesini (`train_iter`) ve test kümesini (`test_iter`) kullanarak `net` modelini eğitmek ve değerlendirmek için kullanıyoruz. Sınırlı hesaplama kaynakları nedeniyle, eğitim ve test doğruluğu daha da iyileştirilebilir: Tartışmalarını alıştırmalara bırakıyoruz.

```
lr, num_epochs = 1e-4, 5
trainer = torch.optim.Adam(net.parameters(), lr=lr)
loss = nn.CrossEntropyLoss(reduction='none')
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.519, train acc 0.791, test acc 0.782
10280.0 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]
```



15.7.4 Özet

- SNLI veri kümesindeki doğal dil çıkarımı gibi aşağı akış uygulamaları için önceden eğitilmiş BERT modeline ince ayar yapabiliriz.
- İnce ayar sırasında BERT modeli aşağı akış uygulaması için modelin bir parçası haline gelir. Sadece ön eğitim kaybı ile ilgili parametreler ince ayar sırasında güncellenmeyecektir.

15.7.5 Alıştırmalar

1. Hesaplama kaynağınız izin veriyorsa, orijinal BERT temel modeli kadar büyük olan, çok daha büyük, önceden eğitilmiş BERT modeline ince ayar yapın. `load_pretrained_model` işlevinde bağımsız değişkenleri şu şekilde ayarlayın: 'bert.small' yerine 'bert.base' ile kullanın, sırasıyla `num_hiddens=256`, `ffn_num_hiddens=512`, `num_heads=4` ve `num_layers=2` ile 768, 3072, 12 ve 12 değerlerini değiştirin. İnce ayar dönemlerini artırarak (ve muhtemelen diğer hiper parametreleri ayarlayarak), 0.86'dan daha yüksek bir test doğruluğu elde edebilir misiniz?
2. Bir çift dizi uzunluk oranlarına göre nasıl budanır? Bu çift budama yöntemini ve `SNLIBERTDataset` sınıfında kullanılan yöntemi karşılaştırın. Artıları ve eksileri nelerdir?

Tartışmalar[210]

[210] https://discuss.d2l.ai/t/1526

16 | Tavsiye Sistemleri

Shuai Zhang (*Amazon*), **Aston Zhang** (*Amazon*) ve **Yi Tay** (*Google*)

Tavsiye sistemleri endüstride yaygın olarak kullanılmaktadır ve günlük hayatımızda her yerde bulunmaktadır. Bu sistemler, çevrimiçi alışveriş siteleri (örneğin amazon.com), müzik/film hizmetleri sitesi (ör. Netflix ve Spotify), mobil uygulama mağazaları (örneğin, IOS uygulama mağazası ve google play), çevrimiçi reklamcılık gibi birçok alanda kullanılmaktadır.

Tavsiye sistemlerinin temel amacı, kullanıcıların izleyecekleri filmler, okuyacakları metinler veya satın alacakları ürünler gibi ilgili öğeleri keşfetmelerine yardımcı olmaktır, böylece keyifli bir kullanıcı deneyimi yaratır. Ayrıca, tavsiye sistemleri, çevrimiçi perakendecilerin artan gelir elde etmek için uyguladığı en güçlü makine öğrenmesi sistemleri arasındadır. Tavsiye sistemleri, proaktif arama çabalarını azaltarak ve kullanıcıları hiç aramadıkları tekliflerle şaşırtarak arama motorlarının yerini alıyor. Birçok şirket, daha etkili tavsiye sistemleri yardımıyla kendilerini rakiplerinin önünde konumlandırmayı başardı. Bu nedenle, tavsiye sistemleri sadece günlük hayatımızın merkezinde olmakla kalmaz, aynı zamanda bazı sektörlerde de son derece vazgeçilmezdir.

Bu bölümde, tavsiye sistemlerinin temellerini ve gelişmelerini, farklı veri kaynaklarına sahip tavsiye sistemlerinin oluşturulmasına yönelik bazı genel temel teknikleri ve bunların uygulamalarını inceleyeceğiz. Özellikle, bir kullanıcının muhtemel bir öğeye verebileceği derecelendirmeyi nasıl tahmin edeceğinizi, öğelerin tavsiye listesini nasıl oluşturacağınızı ve bol miktarda özniteliklerden tıklama oranını nasıl tahmin edeceğinizi öğreneceksiniz. Bu görevler gerçek dünyadaki uygulamalarda yaygındır. Bu bölümü inceleyerek, gerçek dünya tavsiye problemlerini sadece klasik yöntemlerle değil, daha gelişmiş derin öğrenme tabanlı modellerle de çözmeye ilişkin uygulamalı deneyime sahip olacaksınız.

16.1 Tavsiye Sistemlerine Genel Bakış

Son on yılda İnternet, iletişim kurma, haber okuma, ürün satın alma ve film izleme şeklimizi derinden değiştiren büyük ölçekli çevrimiçi hizmetler için bir platform haline geldi. Bu arada, benzeri görülmemiş sayıda öğe (filmler, haberler, kitaplar ve ürünlere atıfta bulunmak için *öğe* terimini kullanıyoruz.) çevrimiçi olarak sunulan, tercih ettiğimiz öğeleri keşfetmemize yardımcı olabilecek bir sistem gerektirir. Bu nedenle tavsiye sistemleri, kişiselleştirilmiş hizmetleri kolaylaştırabilen ve bireysel kullanıcılara özel deneyim sunabilen güçlü bilgi filtreleme araçlarıdır. Kısacası, tavsiye sistemleri, seçimleri yönetilebilir hale getirmek için mevcut veri zenginliğinin kullanılmasında önemli bir rol oynamaktadır. Günümüzde öneri sistemleri Amazon, Netflix ve YouTube gibi bir dizi çevrimiçi hizmet sağlayıcısının özünde yer almaktadır. Fig. 1.3.3 içinde Amazon tarafından önerilen derin öğrenme kitaplarının örneğini hatırlayın. Tavsiye sistemlerini kullanmanın faydaları iki yönlüdür: Bir yandan, kullanıcıların öğeleri bulma çabalarını büyük ölçüde azaltabilir ve aşırı bilgi yüklemesi sorununu hafifletebilir. Öte yandan, çevrimiçi servis

sağlayıcılara iş değeri katabilir ve önemli bir gelir kaynağıdır. Bu ünite, uygulamaya konan örneklerle birlikte tavsiye sistemleri alanında derin öğrenme ile ilgili temel kavramları, klasik modelleri ve son ilerlemeleri tanıtacaktır.

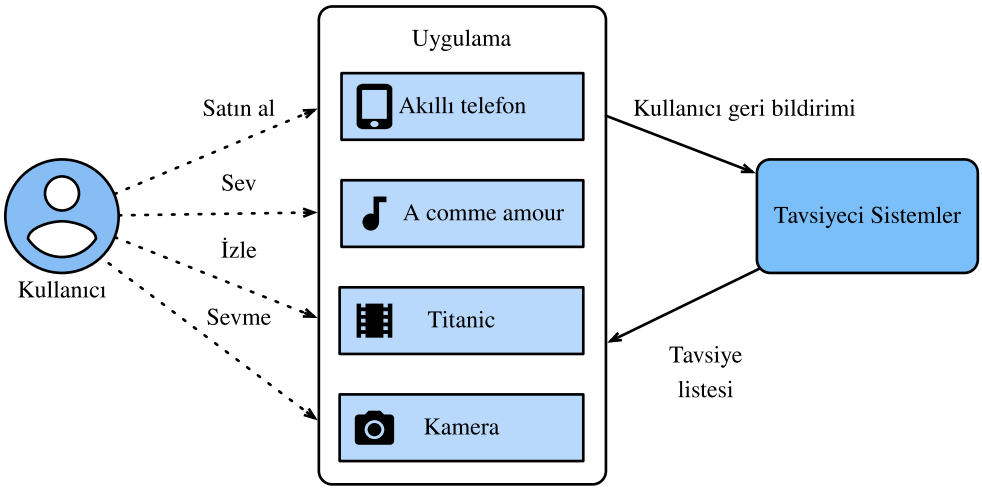


Fig. 16.1.1: Tavsiye Sürecinin Resimlendirilmesi

16.1.1 İşbirlikçi Filtreleme

Yolculuğa, tavsiye sistemlerindeki önemli kavram ile başlıyoruz: İlk olarak Goblen sistemi tarafından icat edilen işbirlikçi filtreleme (İF) (Goldberg *et al.*, 1992), "İnsanlar, haber gruplarına gönderilen çok miktarda e-posta ve iletiyi işlerken filtreleme işlemini gerçekleştirmede birbirlerine yardımcı olmak için işbirliği yapıyor" diye ifade edilmiştir. Bu terim daha fazla anlam ile zenginleştirilmiştir. Geniş anlamda, birden çok kullanıcı, aracılar ve veri kaynağı arasında işbirliği içeren teknikleri kullanarak bilgi veya desenler için filtreleme işlemidir. İF birçok biçime sahiptir, ortaya çıkışından bu yana önerilen çok sayıda İF yöntemi vardır.

Genel olarak, İF teknikleri, bellek tabanlı İF, model tabanlı İF, and onların melezleri ile kategorize edilebilir (Su and Khoshgoftaar, 2009). Temsilci bellek tabanlı İF teknikleri, kullanıcı tabanlı İF ve öğe tabanlı İF (Sarwar *et al.*, 2001) gibi en yakın komşu tabanlı İF tekniklerdir. Matris ayrıştırma gibi saklı çarpan modelleri model tabanlı İF örnekleridir. Bellek tabanlı İF, benzerlik değerlerini ortak öğelere göre hesapladığı için seyrek ve büyük ölçekli verilerle uğraşırken sınırlamalara sahiptir. Model tabanlı yöntemler, seyreklik ve ölçeklenebilirlik ile başa çıkmadaki daha iyi yeteneği ile daha popüler hale gelmektedir. Birçok model tabanlı İF yaklaşımı sinir ağları ile genişletilebilir, bu da derin öğrenmedeki (Zhang *et al.*, 2019) hesaplama hızlandırmasıyla daha esnek ve ölçeklenebilir modellere yol açar. Genel olarak, İF yalnızca tahmin ve tavsiyeler yapmak için kullanıcı-öğe etkileşim verilerini kullanır. İF'nin yanı sıra, içerik tabanlı ve bağlam tabanlı öneri sistemleri, öğelerin/kullanıcıların içerik açıklamalarını, zaman damgaları ve konumlar gibi bağlamsal sinyaller ile birleştirmede de yararlıdır. Açıkçası, farklı girdi verileri mevcut olduğunda model türlerini/yapılarını ayarlamamız gerekebilir.

16.1.2 Açık Geri Bildirim ve Kapalı Geri Bildirim

Kullanıcıların tercihini öğrenmek için sistem onlardan geri bildirim toplayacaktır. Geri bildirim açık veya örtülü (Hu *et al.*, 2008) olabilir. Örneğin, IMDB[211] filmler için bir ila on yıldız arasında değişen yıldız derecelendirmelerini toplar. YouTube, kullanıcıların tercihlerini göstermeleri için başparmak yukarı ve aşağı düğmelerini sağlar. Açık geri bildirim toplamanın, kullanıcıların ilgi alanlarını proaktif olarak belirtmelerini gerektirdiği açıktır. Bununla birlikte, birçok kullanıcı ürünleri derecelendirmeye isteksiz olabileceğinden, açık geri bildirimler her zaman hazır değildir. Göreceli konuşursak, örtülü geri bildirimler genellikle kullanıcı tıklamaları gibi örtülü davranışların modellenmesiyle ilgili olduğu için genellikle kolayca kullanılabilir. Bu nedenle, birçok tavsiye sistemi, kullanıcı davranışlarını gözlemleyerek kullanıcının görüşünü dolaylı olarak yansıtan örtülü geri bildirimlere odaklanmıştır. Satın alma geçmişi, tarama geçmişi, saatler ve hatta fare hareketleri dahil olmak üzere çeşitli örtülü geri bildirim formları vardır. Örneğin, aynı yazar tarafından birçok kitap satın alan bir kullanıcı muhtemelen bu yazarı sever. Örtülü geri bildirimin doğal olarak gürültülü olduğunu unutmayın. Sadece tercihlerini ve gerçek motiflerini *tahmin edebiliriz*. Bir kullanıcı bir filmi izlemişse, o film hakkında olumlu bir görüşe sahip olduğunu göstermez.

16.1.3 Tavsiye Görevleri

Geçtiğimiz yıllarda bir dizi tavsiye görevi araştırılmıştır. Uygulama alanına göre film tavsiyesi, haber tavsiyesi, ilgi noktası tavsiyesi (Ye *et al.*, 2011) vb. vardır. Görevleri geri bildirim ve girdi verilerinin türlerine göre ayırt etmek de mümkündür, örneğin, derecelendirme tahmini görevi açık derecelendirmeleri tahmin etmeyi amaçlamaktadır. Zirve-n tavsiyesi (öğe sıralaması), örtük geri bildirimlere dayanarak her kullanıcı için kişisel olarak tüm öğeleri sıralar. Zaman damgası bilgileri de dahil edilmişse, (Quadrana *et al.*, 2018) dizi duyarlı tavsiye oluşturabiliriz. Bir başka popüler görev, örtülü geri bildirimlere dayanan tıklama oranı tahmini olarak adlandırılır, ancak çeşitli kategorik öznitelikler kullanılabilir. Yeni kullanıcılara tavsiyede bulunmaya ve mevcut kullanıcılara yeni öğeler önermeye soğuk başlangıç tavsiyesi denir (Schein *et al.*, 2002).

16.1.4 Özet

- Tavsiye sistemleri bireysel kullanıcılar ve endüstriler için önemlidir. İşbirlikçi filtreleme, tavsiyede önemli bir kavramdır.
- İki tür geri bildirim vardır: Örtülü geri bildirim ve açık geri bildirim. Son on yılda bir dizi tavsiye görevi araştırılmıştır.

16.1.5 Alıştırmalar

1. Tavsiye sistemlerinin günlük yaşamınızı nasıl etkilediğini açıklayabilir misiniz?
2. Hangi ilginç tavsiye görevlerinin araştırılabileceğini düşünüyorsunuz?

Tartışmalar[212]

[211] https://www.imdb.com/
[212] https://discuss.d2l.ai/t/398

17 | Çekişmeli Üretici Ağlar

17.1 Çekişmeli Üretici Ağlar

Bu kitabın çoğunda, nasıl tahmin yapabileceğimiz hakkında konuştuk. O yada bu şekilde, veri örneklerini etiketlere eşleyen derin sinir ağlarını öğrendik. Bu tür öğrenmeye ayrımcı öğrenme denir, çünkü kedi ve köpek fotoğrafları arasında ayrım yapabilmelerini istiyoruz. Sınıflandırıcılar ve bağlanımcılar (regressor), ayrımcı öğrenmenin örnekleridir. Geri yayma ile eğitilen sinir ağları, büyük karmaşık veri kümelerinde ayrımcı öğrenme hakkında bildiğimizi düşündüğümüz her şeyi altüst etmeyi başardı. Yüksek çözünürlüklü imgelerdeki sınıflandırma doğruluk oranı, sadece 5-6 yıl içinde işe yaramazlık düzeyinden insan düzeyine (bazı kısıtlamalarla) geldi. Bu bölümde, derin sinir ağlarının şaşırtıcı derecede iyi yaptığı diğer ayrımcı görevler hakkında tartışacağız.

Ancak makine öğrenmesinde ayrımcı problemler çözmekten daha ötesi vardır. Örneğin, herhangi bir etiket içermeyen büyük bir veri kümesi verildiğinde, bu verinin karakterini kısaca özetleyen bir model öğrenmek isteyebiliriz. Böyle bir model verildiğinde, eğitim verisinin dağılımına benzeyen sentetik veri örnekleri örnekleyebiliriz. Örneğin, büyük bir yüz fotoğrafı külliyatı verildiğinde, makul bir şekilde, aynı veri kümesinden gelmiş gibi görünen yeni bir foto-gerçekçi imge oluşturabilmeyi isteyebiliriz. Bu tür öğrenmeye üretici modelleme denir.

Yakın zamana kadar, yeni foto-gerçekçi görüntüleri sentezleyebilecek bir yöntem yoktu. Ancak derin sinir ağlarının ayrımcı öğrenmedeki başarısı yeni olasılıkların ortaya çıkmasına vesile oldu. Son üç yıldaki büyük bir yönelim de genellikle gözetimli öğrenme problemi gibi düşünmediğimiz problemlerdeki zorlukların üstesinden gelmede ayrımcı derin ağların kullanılması olmuştur. Yinelemeli sinir ağı dil modelleri, bir kez eğitildikten sonra üretici bir model olarak davranabilen (sonraki karakteri tahmin etmek için eğitilmiş) ayrımcı ağ kullanmanın iyi bir örneğidir.

2014'te çığır açan bir makalede, Çekişmeli Üretici Ağlar (GAN'lar) tanıtıldı (Goodfellow *et al.*, 2014), iyi üretici modeller elde etmek için ayrımcı modellerin gücünden yararlanmada akıllıca yeni bir yol gösterildi. GAN'lar, eğer gerçek veriler ile sahte veriler ayrılamıyorsa, o zaman veri üreticinin başarılı olduğu fikrine dayanırlar. İstatistikte buna ikili-örneklem testi denir - $X = \{x_1, \ldots, x_n\}$ ve $X' = \{x'_1, \ldots, x'_n\}$ veri kümelerinin aynı dağılımdan çekilmiş olup olmadığı sorusuna cevap vermek için yapılan bir testtir. Çoğu istatistik makalesi ile GAN arasındaki temel fark, ikincisinin bu fikri yapıcı bir şekilde kullanmasıdır. Başka bir deyişle, "hey, bu iki veri kümesi aynı dağılımdan gelmiş gibi görünmüyor" demek için bir model eğitmek yerine, ikili-örneklem testini[213] üretici bir modele eğitim sinyalleri sağlamak için kullandılar. Bu bize gerçek verilere benzeyen bir şey üretene kadar veri üreticiyi iyileştirme olanağını tanır. Sınıflandırıcımız son teknoloji bir derin sinir ağı da olsa bile üreticinin en azından sınıflandırıcıyı kandırabilmesi gerekir.

[213] https://en.wikipedia.org/wiki/Two-sample_hypothesis_testing

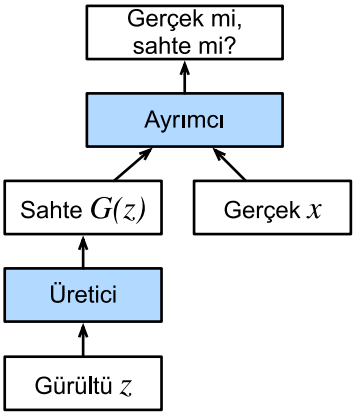


Fig. 17.1.1: Çekişmeli Üretici Ağlar

GAN mimarisi şu şekilde gösterilmektedir Fig. 17.1.1. Gördüğünüz gibi, GAN mimarisinde iki parça bulunur - öncelikle, potansiyel olarak gerçek gibi görünen verileri üretebilecek bir cihaza ihtiyacımız vardır (bir derin ağ düşünebiliriz, fakat oyun oluşturma motoru gibi herhangi bir şey de olabilir). İmgelerle uğraşıyorsak, bu imge üretmeyi gerektirir. Mesela konuşmayla uğraşıyorsak, ses dizileri üretmeyi gerektirir. Buna üretici ağ diyoruz. İkinci bileşen, ayrımcı ağdır. Sahte ve gerçek verileri birbirinden ayırt etmeye çalışır. Her iki ağ da birbiriyle rekabet içindedir. Üretici ağ, ayrımcı ağı kandırmaya çalışır. Bir noktada ayrımcı ağ yeni sahte verilere uyum sağlar. Bu bilgi de üretici ağı iyileştirmek için kullanılır.

Ayrımcı, x girdisinin gerçek mi (gerçek veriden) yoksa sahte mi (üreticiden) olduğunu ayırt eden bir ikili sınıflandırıcıdır. Tipik olarak, ayrımcı $\mathbf{x}$ girdisi için bir skaler tahmin, $o \in \mathbb{R}$, verir ve bunun için mesela gizli boyutu 1 olan yoğun bir katman kullanabilir ve ardından da tahmin olasılığı, $D(\mathbf{x}) = 1/(1 + e^{-o})$, için sigmoid fonksiyonunu uygulayabilir. y etiketinin gerçek veriler için 1 ve sahte veriler için 0 olduğunu varsayalım. Ayrımcıyı, çapraz entropi kaybını en aza indirecek şekilde eğitiriz, *yani*,

$$\min_D \{-y \log D(\mathbf{x}) - (1-y) \log(1 - D(\mathbf{x}))\}, \tag{17.1.1}$$

Üretici için, önce bir rasgelelik kaynağından bir $\mathbf{z} \in \mathbb{R}^d$ parametresi çekeriz, *örneğin*, normal bir dağılım, $\mathbf{z} \sim \mathcal{N}(0, 1)$, kullanabiliriz. Gizli değişkeni genellikle $\mathbf{z}$ ile gösteriyoruz. Daha sonra $\mathbf{x}' = G(\mathbf{z})$ oluşturmak için bir işlev uygularız. Üreticinin amacı, ayrımcıyı $\mathbf{x}' = G(\mathbf{z})$'yi gerçek veri olarak *yani*, $D(G(\mathbf{z})) \approx 1$ diye, sınıflandırması için kandırmaktır. Başka bir deyişle, belirli bir D ayrımcısı için, $y = 0$, olduğunda çapraz entropi kaybını en yükseğe çıkarmak için G üreticisinin parametrelerini güncelleriz, *yani*

$$\max_G \{-(1-y) \log(1 - D(G(\mathbf{z})))\} = \max_G \{-\log(1 - D(G(\mathbf{z})))\}. \tag{17.1.2}$$

Üretici mükemmel bir iş çıkarırsa, o zaman $D(\mathbf{x}') \approx 1$ olur, böylece yukarıdaki kayıp 0'a yaklaşır, bu da gradyanların ayrımcı için anlamlı bir iyileştirme sağlayamayacak kadar küçük olmasına neden olur. Bu yüzden, genellikle aşağıdaki kaybı en aza indirmeyi deneriz:

$$\min_G \{-y \log(D(G(\mathbf{z})))\} = \min_G \{-\log(D(G(\mathbf{z})))\}, \tag{17.1.3}$$

Bu da $y = 1$ etiketini vererek $\mathbf{x}' = G(\mathbf{z})$'yi ayrımcıya beslemek demektir.

Özetle, D ve G, kapsamlı amaç işlevine sahip bir "minimax" oyunu oynuyorlar:

$$min_D max_G \{-E_{x \sim \text{Data}} log D(\mathbf{x}) - E_{z \sim \text{Noise}} log(1 - D(G(\mathbf{z})))\}. \tag{17.1.4}$$

GAN uygulamalarının çoğu imge bağlamındadır. Sizlere gösterme amacıyla, önce çok daha basit bir dağılım oturtmakla yetineceğiz. Bir Gaussian için dünyanın en verimsiz parametre tahmincisini oluşturmak için GAN'ları kullanırsak ne olacağını göreceğiz. Hadi başlayalım.

```
%matplotlib inline
import torch
from torch import nn
from d2l import torch as d2l
```

17.1.1 Bir Miktar "Gerçek" Veri Üretme

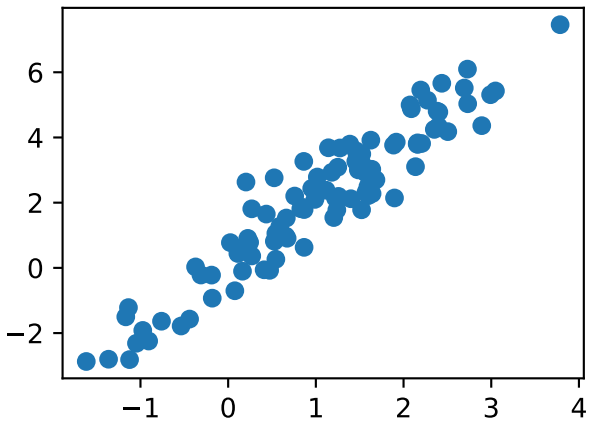
Bu dünyanın en yavan örneği olacağından, basit bir Gauss'tan çekilen verileri üretiyoruz.

```
X = torch.normal(0.0, 1, (1000, 2))
A = torch.tensor([[1, 2], [-0.1, 0.5]])
b = torch.tensor([1, 2])
data = torch.matmul(X, A) + b
```

Bakalım elimizde neler var. Bu, ortalaması b ve kovaryans matrisi $A^T A$ olan keyfi bir şekilde kaydırılmış bir Gaussian olacaktır.

```
d2l.set_figsize()
d2l.plt.scatter(data[:100, 0].detach().numpy(), data[:100, 1].detach().numpy());
print(f'Kovaryans matrisi\n{torch.matmul(A.T, A)}')
```

```
Kovaryans matrisi
tensor([[1.0100, 1.9500],
        [1.9500, 4.2500]])
```




```
batch_size = 8
data_iter = d2l.load_array((data,), batch_size)
```

17.1.2 Üretici

Üretici ağımız mümkün olan en basit ağ olacak - tek katmanlı bir doğrusal model. Bunun nedeni, bu doğrusal ağı bir Gauss veri üreticisi ile yönlendirecek olmamız. Böylece, kelimenin birebir anlamıyla, sadece nesneleri mükemmel bir şekilde taklit etmek için gerekli parametreleri öğrenmesi gerekecektir.

```
net_G = nn.Sequential(nn.Linear(2, 2))
```

17.1.3 Ayrımcı

Ayrımcı için biraz daha hassas olacağız: İşleri biraz daha ilginç hale getirmek için 3 katmanlı bir MLP kullanacağız.

```
net_D = nn.Sequential(
    nn.Linear(2, 5), nn.Tanh(),
    nn.Linear(5, 3), nn.Tanh(),
    nn.Linear(3, 1))
```

17.1.4 Eğitim

İlk olarak ayrımcıyı güncellemek için bir işlev tanımlayalım.

```
#@save
def update_D(X, Z, net_D, net_G, loss, trainer_D):
    """Ayrımcıyı güncelle."""
    batch_size = X.shape[0]
    ones = torch.ones((batch_size,), device=X.device)
    zeros = torch.zeros((batch_size,), device=X.device)
    trainer_D.zero_grad()
    real_Y = net_D(X)
    fake_X = net_G(Z)
    # `net_G` için gradyanı hesaplamanıza gerek yok, onu
    # gradyan hesaplamalarından koparın
    fake_Y = net_D(fake_X.detach())
    loss_D = (loss(real_Y, ones.reshape(real_Y.shape)) +
              loss(fake_Y, zeros.reshape(fake_Y.shape))) / 2
    loss_D.backward()
    trainer_D.step()
    return loss_D
```

Üreticiyi de benzer şekilde güncelleiyoruz. Burada çapraz entropi kaybını tekrar kullanıyoruz ama sahte verinin etiketini 0'dan 1'e çeviriyoruz.

```
#@save
def update_G(Z, net_D, net_G, loss, trainer_G):
    """Üreticiyi güncelle."""
    batch_size = Z.shape[0]
    ones = torch.ones((batch_size,), device=Z.device)
    trainer_G.zero_grad()
```

(continues on next page)

(continued from previous page)

```
    # Hesaplamadan kurtulmak için `update_D`'den `fake_X`'i yeniden
    # kullanabiliriz
    fake_X = net_G(Z)
    # `net_D` değiştirildiğinden `fake_Y`'nin yeniden hesaplanması gerekiyor
    fake_Y = net_D(fake_X)
    loss_G = loss(fake_Y, ones.reshape(fake_Y.shape))
    loss_G.backward()
    trainer_G.step()
    return loss_G
```

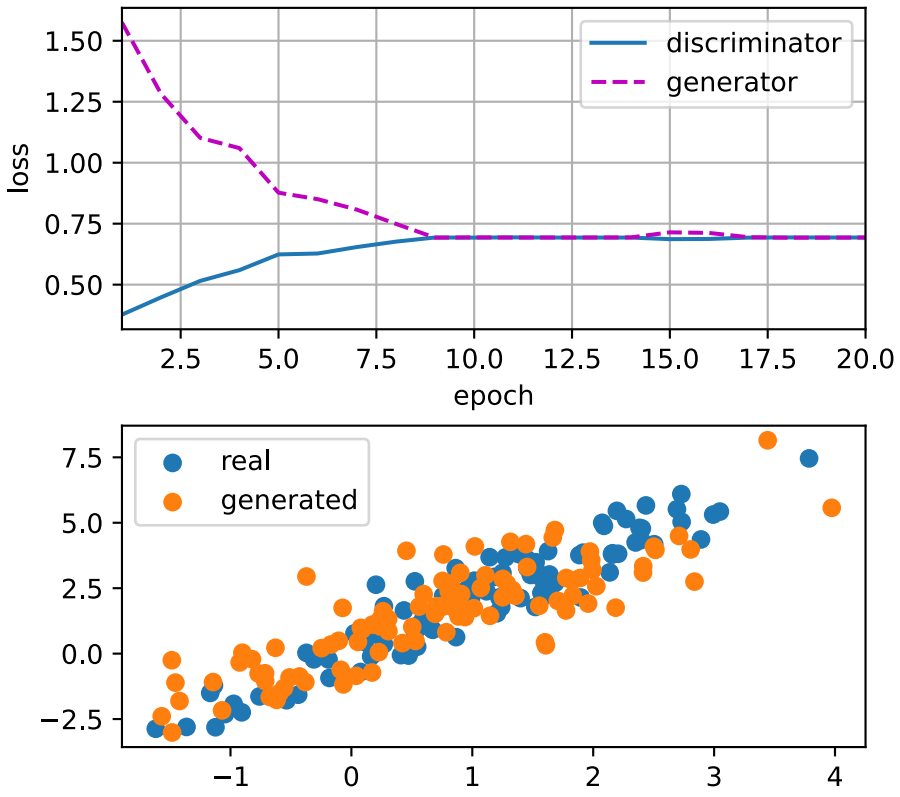
Hem ayrımcı hem de üretici, çapraz entropi kaybıyla ikili lojistik bağlanım uygular. Eğitim sürecini kolaylaştırmak için Adam'ı kullanıyoruz. Her yinelemede, önce ayrımcıyı ve ardından da üreticiyi güncelliyoruz. Ayrıca hem kayıpları hem de üretilen örnekleri görselleştiriyoruz.

```
def train(net_D, net_G, data_iter, num_epochs, lr_D, lr_G, latent_dim, data):
    loss = nn.BCEWithLogitsLoss(reduction='sum')
    for w in net_D.parameters():
        nn.init.normal_(w, 0, 0.02)
    for w in net_G.parameters():
        nn.init.normal_(w, 0, 0.02)
    trainer_D = torch.optim.Adam(net_D.parameters(), lr=lr_D)
    trainer_G = torch.optim.Adam(net_G.parameters(), lr=lr_G)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[1, num_epochs], nrows=2, figsize=(5, 5),
                            legend=['discriminator', 'generator'])
    animator.fig.subplots_adjust(hspace=0.3)
    for epoch in range(num_epochs):
        # Bir dönem eğit
        timer = d2l.Timer()
        metric = d2l.Accumulator(3)  # loss_D, loss_G, num_examples
        for (X,) in data_iter:
            batch_size = X.shape[0]
            Z = torch.normal(0, 1, size=(batch_size, latent_dim))
            metric.add(update_D(X, Z, net_D, net_G, loss, trainer_D),
                       update_G(Z, net_D, net_G, loss, trainer_G),
                       batch_size)
        # Üretilen örnekleri görselleştirin
        Z = torch.normal(0, 1, size=(100, latent_dim))
        fake_X = net_G(Z).detach().numpy()
        animator.axes[1].cla()
        animator.axes[1].scatter(data[:, 0], data[:, 1])
        animator.axes[1].scatter(fake_X[:, 0], fake_X[:, 1])
        animator.axes[1].legend(['real', 'generated'])
        # Kayıpları göster
        loss_D, loss_G = metric[0]/metric[2], metric[1]/metric[2]
        animator.add(epoch + 1, (loss_D, loss_G))
    print(f'loss_D {loss_D:.3f}, loss_G {loss_G:.3f}, '
          f'{metric[2] / timer.stop():.1f} examples/sec')
```

Şimdi, Gauss dağılımına oturacak hiper parametreleri belirliyoruz.

```
lr_D, lr_G, latent_dim, num_epochs = 0.05, 0.005, 2, 20
train(net_D, net_G, data_iter, num_epochs, lr_D, lr_G,
      latent_dim, data[:100].detach().numpy())
```

```
loss_D 0.693, loss_G 0.693, 1431.8 examples/sec
```



17.1.5 Özet

- Çekişmeli üretici ağlar (GAN'lar), iki derin ağdan oluşur: Üretici ve ayrımcı.
- Üretici, çapraz entropi kaybını en yükseğe çıkararak *yani*, $\max \log(D(\mathbf{x}'))$ yoluyla, ayrımcıyı kandırmak için gerçek imgeye olabildiğince yakın imgeler oluşturur.
- Ayrımcı, çapraz entropi kaybını en aza indirerek, oluşturulan imgeleri gerçek imgelerden ayırt etmeye çalışır, *yani*, $\min -y \log D(\mathbf{x}) - (1-y) \log(1 - D(\mathbf{x}))$ optimize edilir.

17.1.6 Alıştırmalar

- Üreticinin kazandığı yerde bir denge var mıdır, *mesela* ayrımcının sonlu örnekler üzerinden iki dağılımı ayırt edemediği gibi?

Tartışmalar[214]

[214] https://discuss.d2l.ai/t/1082

17.2 Derin Evrişimli Çekişmeli Üretici Ağlar

Section 17.1 içinde, GAN'ların nasıl çalıştığına dair temel fikirleri tanıttık. Tekdüze veya normal dağılım gibi basit, örneklenmesi kolay bir dağılımdan örnekler alabileceklerini ve bunları bazı veri kümelerinin dağılımıyla eşleşiyor gibi görünen örneklere dönüştürebileceklerini gösterdik. Burada bir 2B Gauss dağılımını eşleştirme örneğimiz amaca uygunken, özellikle heyecan verici bir örnek değil.

Bu bölümde, foto-gerçekçi imgeler oluşturmak için GAN'ları nasıl kullanabileceğinizi göstereceğiz. Modellerimizi derin evrişimli GAN'lara (DCGAN) dayandıracağız (Radford *et al.*, 2015). Ayrımcı bilgisayarla görme problemleri için çok başarılı olduğu kanıtlanmış evrişimli mimariyi ödünç alacağız ve GAN'lar aracılığıyla foto-gerçekçi imgeler oluşturmak için nasıl kullanılabileceklerini göstereceğiz.

```
import warnings
import torch
import torchvision
from torch import nn
from d2l import torch as d2l
```

17.2.1 Pokemon Veri Kümesi

Kullanacağımız veri kümesi, pokemondb[215] adresinden elde edilen Pokemon görselleri koleksiyonudur. İlk önce bu veri kümesini indirelim, çıkaralım ve yükleyelim.

```
#@save
d2l.DATA_HUB['pokemon'] = (d2l.DATA_URL + 'pokemon.zip',
                           'c065c0e2593b8b161a2d7873e42418bf6a21106c')

data_dir = d2l.download_extract('pokemon')
pokemon = torchvision.datasets.ImageFolder(data_dir)
```

```
Downloading ../data/pokemon.zip from http://d2l-data.s3-accelerate.amazonaws.com/pokemon.zip.
↪..
```

Her bir imgeyi 64×64 şekilde yeniden boyutlandırıyoruz. `ToTensor` dönüşümü piksel değerini $[0, 1]$ aralığına izdüşürürken bizim üreticimiz $[-1, 1]$ aralığından çıktılar elde etmek için tanh işlevini kullanacak. Bu nedenle, verileri değer aralığına uyması için 0.5 ortalama ve 0.5 standart sapma ile normalize ediyoruz.

```
batch_size = 256
transformer = torchvision.transforms.Compose([
    torchvision.transforms.Resize((64, 64)),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(0.5, 0.5)
])
pokemon.transform = transformer
data_iter = torch.utils.data.DataLoader(
```

(continues on next page)

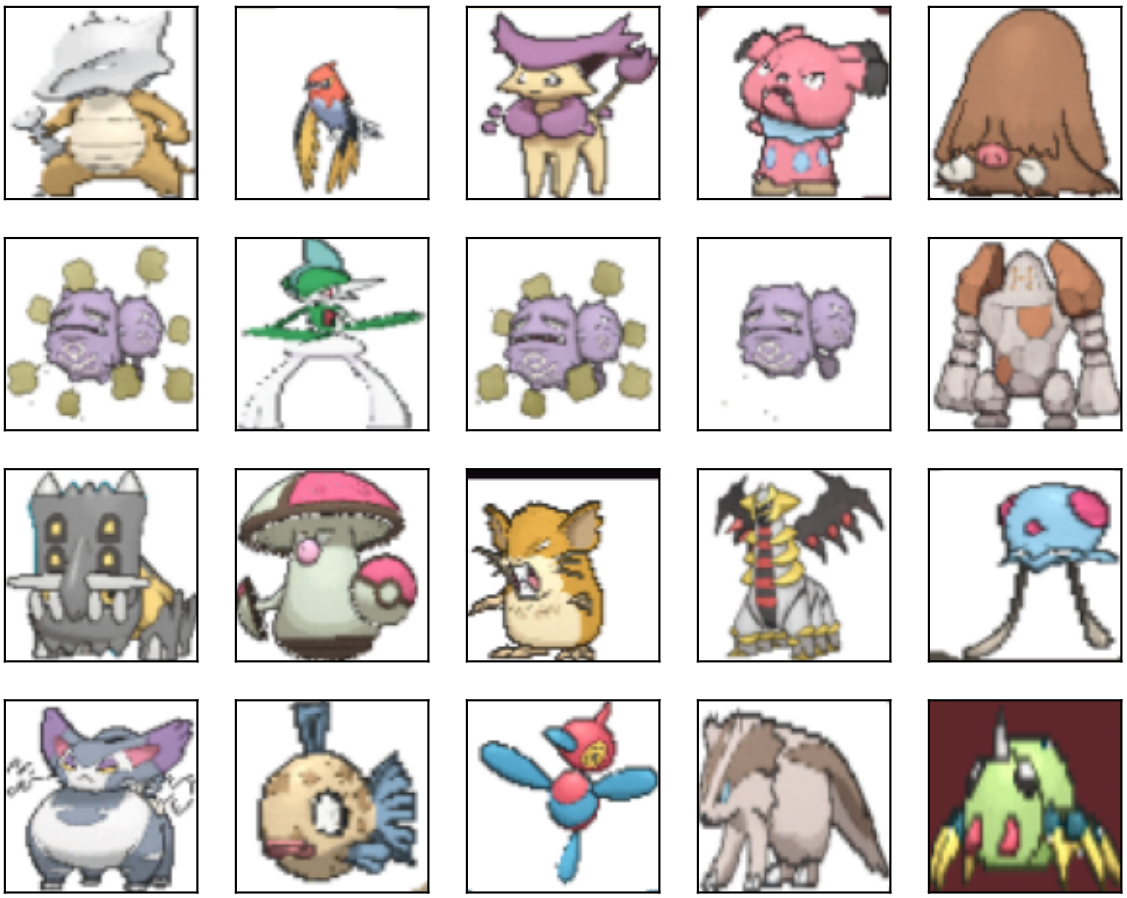
[215] https://pokemondb.net/sprites

(continued from previous page)

```
    pokemon, batch_size=batch_size,
    shuffle=True, num_workers=d2l.get_dataloader_workers())
```

İlk 20 imgeyi görselleştirelim.

```
warnings.filterwarnings('ignore')
d2l.set_figsize((4, 4))
for X, y in data_iter:
    imgs = X[0:20,:,:,:].permute(0, 2, 3, 1)/2+0.5
    d2l.show_images(imgs, num_rows=4, num_cols=5)
    break
```



17.2.2 Üretici

Üreticinin, d-uzunluklu vektör olan $\mathbf{z} \in \mathbb{R}^d$ gürültü değişkenini, genişliği ve yüksekliği 64×64 olan bir RGB imgesine eşlemesi gerekir. Section 13.11 bölümünde, girdi boyutunu büyütmek için devrik evrişim katmanı (bkz. Section 13.10) kullanan tam evrişimli ağı tanıttık. Üreticinin temel bloğu, devrik bir evrişim katmanı, ardından toptan normalleştirme ve ReLU etkinleştirmesi içerir.

```
class G_block(nn.Module):
    def __init__(self, out_channels, in_channels=3, kernel_size=4, strides=2,
```

(continues on next page)

(continued from previous page)

```
                padding=1, **kwargs):
        super(G_block, self).__init__(**kwargs)
        self.conv2d_trans = nn.ConvTranspose2d(in_channels, out_channels,
                                kernel_size, strides, padding, bias=False)
        self.batch_norm = nn.BatchNorm2d(out_channels)
        self.activation = nn.ReLU()

    def forward(self, X):
        return self.activation(self.batch_norm(self.conv2d_trans(X)))
```

Varsayılan olarak, devrik evrişim katmanı $k_h = k_w = 4$'lük çekirdek, $s_h = s_w = 2$'lik uzun adımlar (stride) ve $p_h = p_w = 1$'lik dolgu (padding) kullanır. $n_h^{'} \times n_w^{'} = 16 \times 16$ girdi şekli ile, üretici bloğu girdinin genişliğini ve yüksekliğini iki katına çıkaracaktır.

$$\begin{aligned} n_h^{'} \times n_w^{'} &= [(n_h k_h - (n_h - 1)(k_h - s_h) - 2p_h] \times [(n_w k_w - (n_w - 1)(k_w - s_w) - 2p_w] \\ &= [(k_h + s_h(n_h - 1) - 2p_h] \times [(k_w + s_w(n_w - 1) - 2p_w] \\ &= [(4 + 2 \times (16 - 1) - 2 \times 1] \times [(4 + 2 \times (16 - 1) - 2 \times 1] \\ &= 32 \times 32. \end{aligned} \tag{17.2.1}$$

```
x = torch.zeros((2, 3, 16, 16))
g_blk = G_block(20)
g_blk(x).shape
```

```
torch.Size([2, 20, 32, 32])
```

Devrik evrişim katmanını 4×4 çekirdek, 1×1 uzun adımları ve sıfır dolgu olarak değiştirelim. 1×1 girdi boyutuyla çıktının genişliği ve yüksekliği sırasıyla 3 artar.

```
x = torch.zeros((2, 3, 1, 1))
g_blk = G_block(20, strides=1, padding=0)
g_blk(x).shape
```

```
torch.Size([2, 20, 4, 4])
```

Üretici, girdinin hem genişliğini hem de yüksekliğini 1'den 32'ye çıkaran dört temel bloktan oluşur. Aynı zamanda, önce saklı değişkeni 64×8 kanala izdüşürür ve ardından her seferinde kanalları yarıya indirir. Sonunda, çıktının üretilmesi için bir devrik evrişim katmanı kullanılır. Genişliği ve yüksekliği istenen 64×64 şekline uyacak şekilde ikiye katlar ve kanal sayısını 3'e düşürür. Çıktı değerlerini $(-1, 1)$ aralığına yansıtmak için tanh etkinleştirme işlevi uygulanır.

```
n_G = 64
net_G = nn.Sequential(
    G_block(in_channels=100, out_channels=n_G*8,
            strides=1, padding=0),                  # Çıktı: (64 * 8, 4, 4)
    G_block(in_channels=n_G*8, out_channels=n_G*4), # Çıktı: (64 * 4, 8, 8)
    G_block(in_channels=n_G*4, out_channels=n_G*2), # Çıktı: (64 * 2, 16, 16)
    G_block(in_channels=n_G*2, out_channels=n_G),   # Çıktı: (64, 32, 32)
    nn.ConvTranspose2d(in_channels=n_G, out_channels=3,
                       kernel_size=4, stride=2, padding=1, bias=False),
    nn.Tanh())  # Çıktı: (3, 64, 64)
```

Üreticinin çıktı şeklini doğrulamak için 100 boyutlu bir saklı değişken oluşturalım.

```
x = torch.zeros((1, 100, 1, 1))
net_G(x).shape
```

```
torch.Size([1, 3, 64, 64])
```

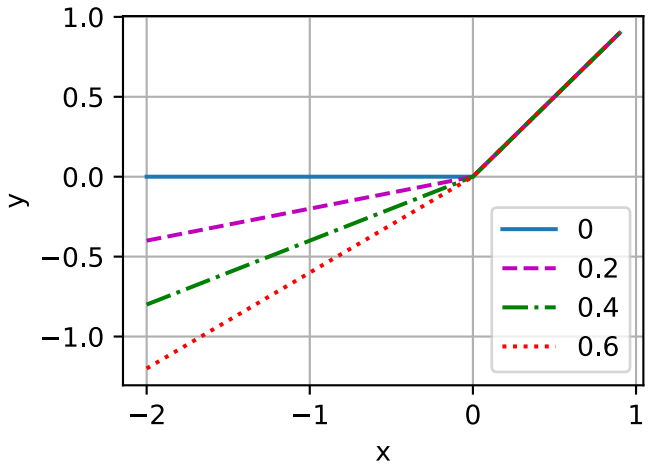
17.2.3 Ayrımcı

Ayrımcı, etkinleştirme işlevi olarak sızıntılı (leaky) ReLU kullanması dışında normal bir evrişimli ağdır. $\alpha \in [0, 1]$ verildiğinde, tanım şöyledir:

$$\text{sızıntılı ReLU}(x) = \begin{cases} x & \text{eğer } x > 0 \\ \alpha x & \text{diğer türlü} \end{cases}. \tag{17.2.2}$$

Görüldüğü gibi, $\alpha = 0$ ise normal ReLU, $\alpha = 1$ ise bir birim fonksiyondur. $\alpha \in (0, 1)$ için, sızıntılı ReLU, negatif bir girdi için sıfır olmayan bir çıktı veren doğrusal olmayan bir fonksiyondur. Bir nöronun her zaman negatif bir değer verebileceği ve bu nedenle ReLU'nun gradyanı 0 olduğu için herhangi bir ilerleme kaydedemeyeceği "ölmekte olan ReLU" (dying ReLU) problemini çözmeyi amaçlamaktadır.

```
alphas = [0, .2, .4, .6, .8, 1]
x = torch.arange(-2, 1, 0.1)
Y = [nn.LeakyReLU(alpha)(x).detach().numpy() for alpha in alphas]
d2l.plot(x.detach().numpy(), Y, 'x', 'y', alphas)
```



Ayrımcının temel bloğu, bir evrişim katmanı ve ardından bir toptan normalleştirme katmanı ve bir sızıntılı ReLU etkinleştirmesidir. Evrişim katmanının hiper parametreleri, üretici bloğundaki devrik evrişim katmanına benzer.

```
class D_block(nn.Module):
    def __init__(self, out_channels, in_channels=3, kernel_size=4, strides=2,
                 padding=1, alpha=0.2, **kwargs):
```

(continues on next page)

(continued from previous page)

```
        super(D_block, self).__init__(**kwargs)
        self.conv2d = nn.Conv2d(in_channels, out_channels, kernel_size,
                                strides, padding, bias=False)
        self.batch_norm = nn.BatchNorm2d(out_channels)
        self.activation = nn.LeakyReLU(alpha, inplace=True)

    def forward(self, X):
        return self.activation(self.batch_norm(self.conv2d(X)))
```

Varsayılan ayarlara sahip temel bir blok, girdilerin genişliğini ve yüksekliğini, Section 6.3 içinde gösterdiğimiz gibi, yarıya düşürecektir. Örneğin, $k_h = k_w = 4$ çekirdek, $s_h = s_w = 2$ uzun adım, $p_h = p_w = 1$ dolgu ve $n_h = n_w = 16$ girdi şekli verildiğinde, çıktının şekli şöyle olacaktır:

$$\begin{aligned} n_h^{'} \times n_w^{'} &= \lfloor (n_h - k_h + 2p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + 2p_w + s_w)/s_w \rfloor \\ &= \lfloor (16 - 4 + 2 \times 1 + 2)/2 \rfloor \times \lfloor (16 - 4 + 2 \times 1 + 2)/2 \rfloor \\ &= 8 \times 8. \end{aligned} \tag{17.2.3}$$

```
x = torch.zeros((2, 3, 16, 16))
d_blk = D_block(20)
d_blk(x).shape
```

```
torch.Size([2, 20, 8, 8])
```

Ayrımcı üreticinin bir yansımasıdır.

```
n_D = 64
net_D = nn.Sequential(
    D_block(n_D),  # Çıktı: (64, 32, 32)
    D_block(in_channels=n_D, out_channels=n_D*2),  # Çıktı: (64 * 2, 16, 16)
    D_block(in_channels=n_D*2, out_channels=n_D*4),  # Çıktı: (64 * 4, 8, 8)
    D_block(in_channels=n_D*4, out_channels=n_D*8),  # Çıktı: (64 * 8, 4, 4)
    nn.Conv2d(in_channels=n_D*8, out_channels=1,
              kernel_size=4, bias=False))  # Çıktı: (1, 1, 1)
```

Tek bir tahmin değeri elde etmek için son katmanda çıktı kanalı 1 olan bir evrişim katmanı kullanılır.

```
x = torch.zeros((1, 3, 64, 64))
net_D(x).shape
```

```
torch.Size([1, 1, 1, 1])
```

17.2.4 Eğitim

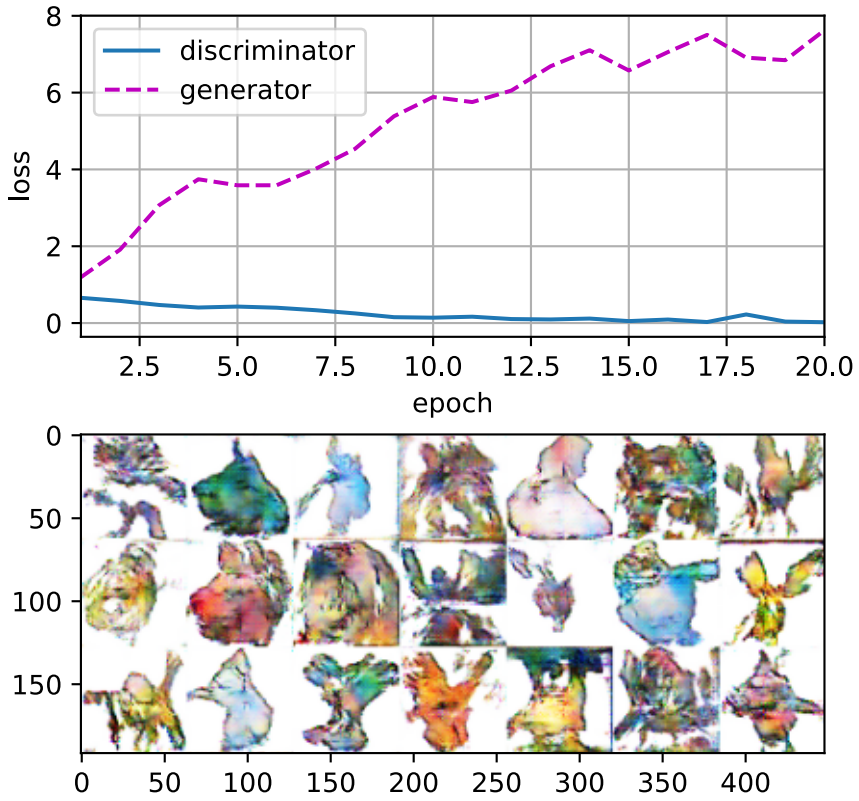
Temel GAN, Section 17.1, ile karşılaştırıldığında, birbirlerine benzer olduklarından hem üretici hem de ayrımcı için aynı öğrenme oranını kullanıyoruz. Ek olarak, Adam'daki (Section 11.10) β_1'yı 0.9'dan 0.5'e değiştiriyoruz. Üretici ve ayrımcı birbiriyle çekiştiği için, hızlı değişen gradyanlarla ilgilenmek için momentumun, ki geçmiş gradyanların üstel ağırlıklı hareketli ortalamasıdır, düzgünlüğünü azaltır. Ayrıca, rastgele üretilen `Z` gürültüsü bir 4B tensördür ve bu nedenle hesaplamayı hızlandırmak için GPU kullanırız.

```
def train(net_D, net_G, data_iter, num_epochs, lr, latent_dim,
          device=d2l.try_gpu()):
    loss = nn.BCEWithLogitsLoss(reduction='sum')
    for w in net_D.parameters():
        nn.init.normal_(w, 0, 0.02)
    for w in net_G.parameters():
        nn.init.normal_(w, 0, 0.02)
    net_D, net_G = net_D.to(device), net_G.to(device)
    trainer_hp = {'lr': lr, 'betas': [0.5,0.999]}
    trainer_D = torch.optim.Adam(net_D.parameters(), **trainer_hp)
    trainer_G = torch.optim.Adam(net_G.parameters(), **trainer_hp)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[1, num_epochs], nrows=2, figsize=(5, 5),
                            legend=['discriminator', 'generator'])
    animator.fig.subplots_adjust(hspace=0.3)
    for epoch in range(1, num_epochs + 1):
        # Bir dönem eğit
        timer = d2l.Timer()
        metric = d2l.Accumulator(3)  # loss_D, loss_G, num_examples
        for X, _ in data_iter:
            batch_size = X.shape[0]
            Z = torch.normal(0, 1, size=(batch_size, latent_dim, 1, 1))
            X, Z = X.to(device), Z.to(device)
            metric.add(d2l.update_D(X, Z, net_D, net_G, loss, trainer_D),
                       d2l.update_G(Z, net_D, net_G, loss, trainer_G),
                       batch_size)
        # Üretilen örnekleri göster
        Z = torch.normal(0, 1, size=(21, latent_dim, 1, 1), device=device)
        # Yapay verileri N(0, 1) olarak normalleştirin
        fake_x = net_G(Z).permute(0, 2, 3, 1) / 2 + 0.5
        imgs = torch.cat(
            [torch.cat([
                fake_x[i * 7 + j].cpu().detach() for j in range(7)], dim=1)
             for i in range(len(fake_x)//7)], dim=0)
        animator.axes[1].cla()
        animator.axes[1].imshow(imgs)
        # Kayıpları göster
        loss_D, loss_G = metric[0] / metric[2], metric[1] / metric[2]
        animator.add(epoch, (loss_D, loss_G))
    print(f'loss_D {loss_D:.3f}, loss_G {loss_G:.3f}, '
          f'{metric[2] / timer.stop():.1f} examples/sec on {str(device)}')
```

Modeli sadece gösterim amaçlı az sayıda dönemle eğitiyoruz. Daha iyi performans için, `num_epochs` değişkeni daha büyük bir sayıya ayarlayabiliriz.

```
latent_dim, lr, num_epochs = 100, 0.005, 20
train(net_D, net_G, data_iter, num_epochs, lr, latent_dim)
```

```
loss_D 0.022, loss_G 7.624, 1068.4 examples/sec on cuda:0
```



17.2.5 Özet

- DCGAN mimarisi, ayrımcı için dört evrişimli katmana ve üretici için dört "kesirli uzun adımlı" evrişimli katmana sahiptir.
- Ayrımcı, toptan normalleştirme (girdi katmanı hariç) ve sızıntılı ReLU etkinleştirmeleri olan 4 katman uzun adımlı evrişimlerdir.
- Sızıntılı ReLU, negatif bir girdi için sıfır olmayan bir çıktı veren doğrusal olmayan bir fonksiyondur. "Ölen ReLU" sorununu çözmeyi amaçlar ve gradyanların mimari boyunca daha kolay akmasına yardımcı olur.

17.2.6 Alıştırmalar

1. Sızıntılı ReLU yerine standart ReLU etkinleştirmesi kullanırsak ne olur?
2. DCGAN'ı Fashion-MNIST'e uygulayın ve hangi kategorinin işe yarayıp hangilerinin yaramadığını görün.

Tartışmalar[216]

[216] https://discuss.d2l.ai/t/1083

18 | Ek: Derin Öğrenme için Matematik

Brent Werness (*Amazon*), **Rachel Hu** (*Amazon*) ve bu kitabın yazarları

Modern derin öğrenmenin harika yanlarından biri, altındaki matematiğin çoğunun tam olarak idrak edilmeden anlaşılabilir ve kullanılabilir olmasıdır. Bu, alanın olgunlaştığının bir işaretidir. Çoğu yazılım geliştiricisinin artık hesaplanabilir işlevler teorisi hakkında endişelenmesi gerekmediği gibi, derin öğrenme uygulayıcılarının da en büyük olabilirlik (maximum likelihood) öğrenmesinin teorik temelleri hakkında endişelenmesi gerekmemesidir.

Ancak henüz tam olarak sonda değiliz.

Uygulamada, bazen mimari seçimlerin gradyan akışını nasıl etkilediğini veya belirli bir kayıp fonksiyonu ile eğitim yaptığınızda saklı varsayımları anlamanız gerekecektir. Entropinin (düzensizlik) bu dünyada neyi ölçtüğünü ve modelinizde karakter başına bitlerin tam olarak ne anlama geldiğini anlamanıza nasıl yardımcı olabileceğini bilmeniz gerekebilir. Bunların hepsi daha derin matematiksel anlayış gerektirir.

Bu ek, modern derin öğrenmenin temel teorisini anlamak için ihtiyacınız olan matematiksel altyapıyı sağlamayı amaçlamaktadır, ancak tam kapsamlı değildir. Doğrusal cebiri daha derinlemesine incelemeye başlayacağız. Çeşitli dönüşümlerin verilerimiz üzerindeki etkilerini görselleştirmemizi sağlayacak tüm genel doğrusal cebirsel nesnelerin ve işlemlerin geometrik bir anlayışını geliştiriyoruz. Temel unsurlardan biri, öz ayrışımların (eigen-decomposition) temellerinin geliştirilmesidir.

Daha sonra, gradyanın neden en dik iniş yönü olduğunu ve neden geri yaymanın olduğu şekli aldığını tam olarak anlayabileceğimiz noktaya kadar türevsel hesap (diferansiyel kalkülüs) teorisini geliştireceğiz. Daha sonra integral hesabı, bir sonraki konumuz, olasılık teorisini desteklemek için gereken ölçüde tartışılacak.

Pratikte sıklıkla karşılaşılan sorunlar kesin değildir ve bu nedenle belirsiz şeyler hakkında konuşmak için bir dile ihtiyacımız vardır. Rastgele değişkenler teorisini ve en sık karşılaşılan dağılımları gözden geçiriyoruz, böylece modelleri olasılıksal olarak tartışabiliriz. Bu, olasılıksal bir sınıflandırma tekniği olan saf (naif) Bayes sınıflandırıcısının temelini sağlar.

Olasılık teorisi ile yakından ilgili olan şey, istatistik alanıdır. İstatistik, kısa bir bölümde hakkını vererek incelemek için çok büyük bir alan olsa da, özellikle tüm makine öğrenmesi uygulayıcılarının bilmesi gereken temel kavramları tanıtacağız: Tahmin edicileri değerlendirmek ve karşılaştırmak, hipotez testleri yapmak ve güven aralıkları oluşturmak.

Son olarak, bilgi depolama ve aktarımının matematiksel alanı olan bilgi teorisi konusuna dönüyoruz. Bu, bir modelin bir araştırma alanında ne kadar bilgi tuttuğunu nicel olarak tartışabileceğimiz temel dili sağlar.

Birlikte ele alındığında bunlar, derin öğrenmeyi derinlemesine anlamaya giden yola başlamak için gereken matematiksel kavramların özünü oluştururlar.

18.1 Geometri ve Doğrusal Cebirsel İşlemler

Section 2.3 içinde, doğrusal cebirin temelleriyle karşılaştık ve verilerimizi dönüştürürken genel işlemleri ifade etmek için nasıl kullanılabileceğini gördük. Doğrusal cebir, derin öğrenmede ve daha geniş anlamda makine öğrenmesinde yaptığımız işlerin çoğunun altında yatan temel matematiksel sütunlardan biridir. Section 2.3, modern derin öğrenme modellerinin mekaniğini iletmek için yeterli mekanizmayı içerirken, konuyla ilgili daha çok şey var. Bu bölümde daha derine ineceğiz, doğrusal cebir işlemlerinin bazı geometrik yorumlarını vurgulayacağız ve özdeğerler (eigenvalues) ve özvektörler (eigenvectors) dahil birkaç temel kavramı tanıtacağız.

18.1.1 Vektörlerin Geometrisi

İlk olarak, uzaydaki noktalar veya yönler olarak vektörlerin iki ortak geometrik yorumunu tartışmamız gerekiyor. Temel olarak bir vektör, aşağıdaki Python listesi gibi bir sayı listesidir.

```
v = [1, 7, 0, 1]
```

Matematikçiler bunu genellikle bir *sütun* veya *satır* vektörü olarak yazarlar;

$$\mathbf{x} = \begin{bmatrix} 1 \\ 7 \\ 0 \\ 1 \end{bmatrix}, \tag{18.1.1}$$

veya

$$\mathbf{x}^\top = \begin{bmatrix} 1 & 7 & 0 & 1 \end{bmatrix}. \tag{18.1.2}$$

Bunlar genellikle veri örneklerinin sütun vektörleri ve ağırlıklı toplamları oluşturmada kullanılan ağırlıkların satır vektörleri olduğu farklı yorumlara sahiptir. Ancak esnek olmak faydalı olabilir. Section 2.3 bölümünde açıkladığımız gibi, tek bir vektörün varsayılan yönelimi bir sütun vektörü olsa da, tablo halindeki bir veri kümesini temsil eden herhangi bir matris için, her bir veri örneğini matriste bir satır vektörü olarak ele almak daha gelenekseldir.

Bir vektör verildiğinde, ona vermemiz gereken ilk yorum uzayda bir nokta olduğudur. İki veya üç boyutta, bu noktaları, *köken (orijin)* adı verilen sabit bir referansa kıyasla uzaydaki konumlarını belirtmek için vektör bileşenlerini kullanarak görselleştirebiliriz. Bu, şurada görülebilir Fig. 18.1.1.

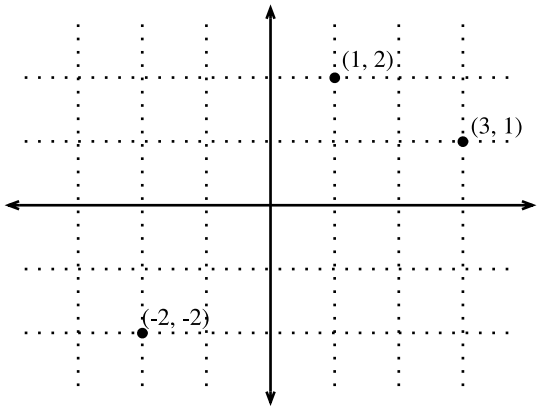


Fig. 18.1.1: Vektörleri düzlemdeki noktalar olarak görselleştirmenin bir örneği. Vektörün ilk bileşeni x koordinatını verir, ikinci bileşen y koordinatını verir. Görselleştirilmesi çok daha zor olsa da, daha yüksek boyutlar da benzerdir.

Bu geometrik bakış açısı, sorunu daha soyut bir düzeyde ele almamızı sağlar. Artık resimleri kedi veya köpek olarak sınıflandırmak gibi başa çıkılmaz görünen bir problemle karşılaşmadığımızdan, görevleri soyut olarak uzaydaki nokta toplulukları olarak değerlendirmeye ve görevi iki farklı nokta kümesini nasıl ayıracağımızı keşfetmek olarak resmetmeye başlayabiliriz.

Buna paralel olarak, insanların genellikle vektörleri aldıkları ikinci bir bakış açısı vardır: Uzayda yönler olarak. $\mathbf{v} = [3, 2]^\top$ vektörünü başlangıç noktasından 3 birim sağda ve 2 birim yukarıda bir konum olarak düşünmekle kalmayabiliriz, aynı zamanda onu sağa doğru 3 adım ve yukarı doğru 2 adım şekilde yönün kendisi olarak da düşünebiliriz. Bu şekilde, şekildeki tüm vektörleri aynı kabul ederiz Fig. 18.1.2.

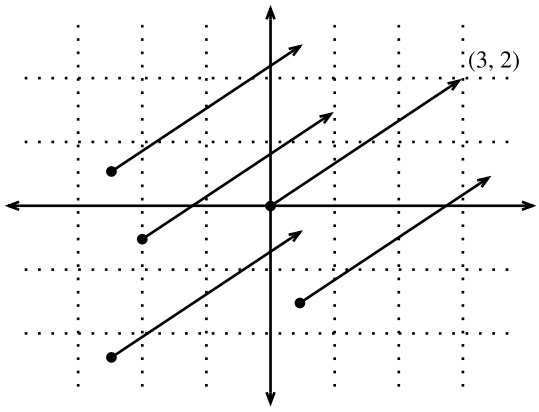


Fig. 18.1.2: Herhangi bir vektör, düzlemde bir ok olarak görselleştirilebilir. Bu durumda, çizilen her vektör $(3, 2)$ vektörünün bir temsilidir.

Bu değişik gösterimin faydalarından biri, vektör toplama işlemini görsel olarak anlamlandırabilmemizdir. Özellikle, bir vektör tarafından verilen yönleri izliyoruz ve şekil Fig. 18.1.3 içinde görüldüğü gibi, sonra diğerinin verdiği yönleri takip ediyoruz.

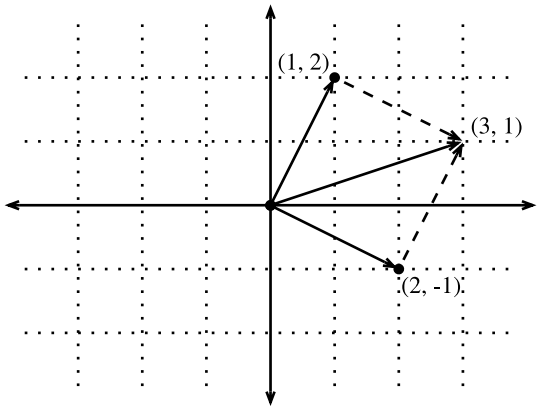


Fig. 18.1.3: Önce bir vektörü, sonra diğerini takip ederek vektör toplamayı görselleştirebiliriz.

Vektör çıkarma işleminin benzer bir yorumu vardır. $\mathbf{u} = \mathbf{v} + (\mathbf{u} - \mathbf{v})$ özdeşliğini göz önünde bulundurursak, $\mathbf{u} - \mathbf{v}$ vektörü, bizi $\mathbf{v}$ noktasından $\mathbf{u}$ noktasına götüren yöndür.

18.1.2 Nokta (İç) Çarpımları ve Açılar

Section 2.3 içinde gördüğümüz gibi, $\mathbf{u}$ ve $\mathbf{v}$ gibi iki sütun vektörü alırsak, bunların nokta çarpımını aşağıdaki işlemi hesaplayarak oluşturabiliriz:

$$\mathbf{u}^\top \mathbf{v} = \sum_i u_i \cdot v_i. \tag{18.1.3}$$

(18.1.3) simetrik olduğundan, klasik çarpmanın gösterimini kopyalayacağız ve şöyle yazacağız:

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{u}^\top \mathbf{v} = \mathbf{v}^\top \mathbf{u}, \tag{18.1.4}$$

Böylece vektörlerin sırasını değiştirmenin aynı cevabı vereceği gerçeğini vurgulamış olacağız.

İç çarpım (18.1.3) ayrıca geometrik bir yorumu da kabul eder: O da iki vektör arasındaki açı ile yakından ilgilidir. Fig. 18.1.4 içinde gösterilen açıyı düşünün.

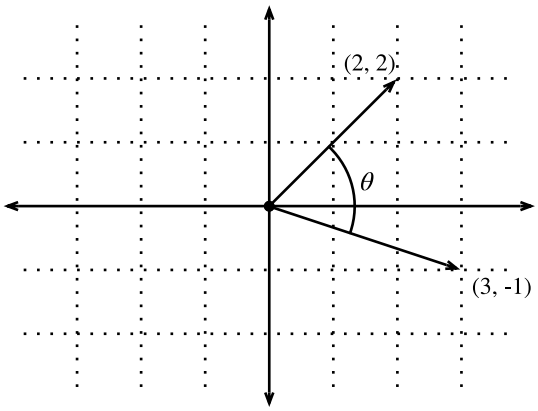


Fig. 18.1.4: Düzlemdeki herhangi iki vektör arasında iyi tanımlanmış bir θ açısı vardır. Bu açının iç çarpıma yakından bağlı olduğunu göreceğiz.

Başlamak için iki belli vektörü ele alalım:

$$\mathbf{v} = (r, 0) \text{ ve } \mathbf{w} = (s\cos(\theta), s\sin(\theta)). \tag{18.1.5}$$

$\mathbf{v}$ vektörü r uzunluğundadır ve x eksenine paralel uzanır, $\mathbf{w}$ vektörü s uzunluğundadır ve x ekseni ile arasında θ açısı vardır. Bu iki vektörün iç çarpımını hesaplarsak, şunu görürüz:

$$\mathbf{v} \cdot \mathbf{w} = rs\cos(\theta) = \|\mathbf{v}\|\|\mathbf{w}\|\cos(\theta). \tag{18.1.6}$$

Bazı basit cebirsel işlemlerle, terimleri yeniden düzenleyebiliriz.

$$\theta = \arccos\left(\frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\|\|\mathbf{w}\|}\right). \tag{18.1.7}$$

Kısacası, bu iki belli vektör için, normlarla birleştirilmiş iç çarpım bize iki vektör arasındaki açıyı söyler. Aynı gerçek genel olarak doğrudur. Burada ifadeyi türetmeyeceğiz, ancak $\|\mathbf{v} - \mathbf{w}\|^2$'yi iki şekilde yazmayı düşünürsek, biri nokta çarpımı ile, diğeri geometrik olarak kosinüsler yasasının kullanımı ile, tam ilişkiyi elde edebiliriz. Gerçekten de, herhangi iki vektör, $\mathbf{v}$ ve $\mathbf{w}$ için, aralarındaki açı:

$$\theta = \arccos\left(\frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\|\|\mathbf{w}\|}\right). \tag{18.1.8}$$

Hesaplamadaki hiçbir şey iki boyutluluğu referans almadığı için bu güzel bir sonuçtur. Aslında bunu üç veya üç milyon boyutta da sorunsuz olarak kullanabiliriz.

Basit bir örnek olarak, bir çift vektör arasındaki açıyı nasıl hesaplayacağımızı görelim:

```
%matplotlib inline
import torch
import torchvision
from IPython import display
from torchvision import transforms
from d2l import torch as d2l


def angle(v, w):
    return torch.acos(v.dot(w) / (torch.norm(v) * torch.norm(w)))

angle(torch.tensor([0, 1, 2], dtype=torch.float32), torch.tensor([2.0, 3, 4]))
```

```
tensor(0.4190)
```

Şu anda kullanmayacağız, ancak açılarının $\pi/2$ (veya eşdeğer olarak $90°$) olduğu vektörleri *dik* olarak isimlendireceğimizi bilmekte fayda var. Yukarıdaki denklemi inceleyerek, bunun $\theta = \pi/2$ olduğunda gerçekleştiğini görürüz, bu $\cos(\theta) = 0$ ile aynı şeydir. Bunun gerçekleşmesinin tek yolu, nokta çarpımın kendisinin sıfır olmasıdır ve ancak ve ancak $\mathbf{v} \cdot \mathbf{w} = 0$ ise iki vektör dik olur. Bu, nesneleri geometrik olarak anlarken faydalı bir formül olacaktır.

Şu soruyu sormak mantıklıdır: Açıyı hesaplamak neden yararlıdır? Cevap, verinin sahip olmasını beklediğimiz türden değişmezlikten gelir. Bir imge ve her piksel değerinin aynı, ancak parlaklığın $\%10$ olduğu kopya bir imge düşünün. Tek tek piksellerin değerleri genel olarak asıl değerlerden uzaktır. Bu nedenle, hakiki imge ile daha karanlık olan arasındaki mesafe hesaplanırsa, mesafe büyük olabilir. Gene de, çoğu makine öğrenmesi uygulaması için *içerik* aynıdır—kedi/köpek sınıflandırıcısı söz konusu olduğunda yine de bir kedinin imgesidir. Ancak, açıyı düşünürsek, herhangi bir $\mathbf{v}$ vektörü için $\mathbf{v}$ ve $0.1 \cdot \mathbf{v}$ arasındaki açının sıfır olduğunu görmek zor değildir. Bu, ölçeklemenin vektörlerin yönlerini koruduğu ve sadece uzunluğu değiştirdiği gerçeğine karşılık gelir. Açı, koyu imgeyi aynı kabul edecektir.

Buna benzer örnekler her yerdedir. Metinde, aynı şeyleri söyleyen iki kat daha uzun bir belge yazarsak tartışılan konunun değişmemesini isteyebiliriz. Bazı kodlamalar için (herhangi sözcük dağarcığındaki kelimelerin kaç kere geçtiğinin sayılması gibi), bu, belgeyi kodlayan vektörün ikiye çarpılmasına karşılık gelir ki, böylece yine açıyı kullanabiliriz.

Kosinüs Benzerliği

Açının iki vektörün yakınlığını ölçmek için kullanıldığı makine öğrenmesi bağlamlarında, uygulayıcılar benzerlik miktarını ifade etmek için *kosinüs benzerliği* terimini kullanırlar.

$$\cos(\theta) = \frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\|\|\mathbf{w}\|}. \tag{18.1.9}$$

İki vektör aynı yönü gösterdiğinde kosinüs maksimum 1, zıt yönleri gösterdiklerinde minimum -1 ve birbirlerine dik iseler 0 değerini alır. Yüksek boyutlu vektörlerin bileşenleri ortalama 0 ile rastgele örneklenirse, kosinüslerinin neredeyse her zaman 0'a yakın olacağını unutmayın.

18.1.3 Hiperdüzlemler

Vektörlerle çalışmaya ek olarak, doğrusal cebirde ileri gitmek için anlamanız gereken bir diğer önemli nesne olan *hiperdüzlem*, bir doğrunun (iki boyut) veya bir düzlemin (üç boyut) daha yüksek boyutlarına genellenmesidir. d boyutlu bir vektör uzayında, bir hiperdüzlemin $d-1$ boyutu vardır ve uzayı iki yarı-uzaya böler.

Bir örnekle başlayalım. Sütun vektörümüzün $\mathbf{w} = [2, 1]^\top$ olduğunu varsayalım. "$\mathbf{w} \cdot \mathbf{v} = 1$ olan $\mathbf{v}$ noktaları nedir?", bilmek istiyoruz. Yukarıda (18.1.8) içindeki nokta çarpımları ile açılar arasındaki bağlantıyı hatırlayarak, bunun aşağıdaki denkleme eşdeğer olduğunu görebiliriz:

$$\|\mathbf{v}\|\|\mathbf{w}\| \cos(\theta) = 1 \iff \|\mathbf{v}\| \cos(\theta) = \frac{1}{\|\mathbf{w}\|} = \frac{1}{\sqrt{5}}. \tag{18.1.10}$$

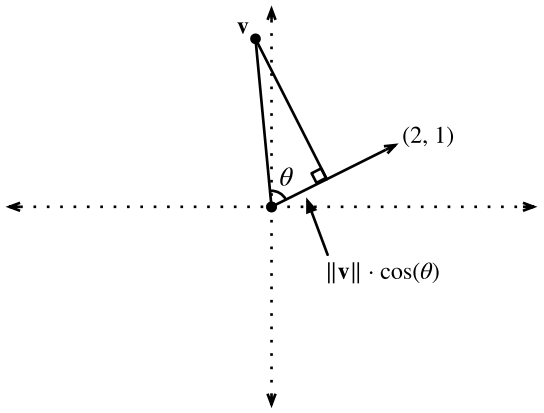


Fig. 18.1.5: Trigonometriyi anımsarsak, $\|\mathbf{v}\| \cos(\theta)$ formülünün $\mathbf{v}$ vektörünün $\mathbf{w}$ yönüne izdüşümünün uzunluğu olduğunu görürüz.

Bu ifadenin geometrik anlamını düşünürsek, Fig. 18.1.5 içinde gösterildiği gibi, bunun $\mathbf{v}$'nin $\mathbf{w}$ yönündeki izdüşümünün uzunluğunun tam olarak $1/\|\mathbf{w}\|$ olduğunu söylemeye eşdeğer olduğunu görürüz. Bunun doğru olduğu tüm noktalar kümesi, $\mathbf{w}$ vektörüne dik açıda olan bir doğrudur.

İstersek, bu doğrunun denklemini bulabilir ve bunun $2x + y = 1$ veya eşdeğer olarak $y = 1 - 2x$ olduğunu görebiliriz.

Şimdi $\mathbf{w} \cdot \mathbf{v} > 1$ veya $\mathbf{w} \cdot \mathbf{v} < 1$ ile nokta kümesini sorduğumuzda ne olduğuna bakarsak, bunların sırasıyla $1/\|\mathbf{w}\|$'den daha uzun veya daha kısa izdüşümlerin (projeksiyon) olduğu durumlar olduğunu görebiliriz. Dolayısıyla, bu iki eşitsizlik çizginin her iki tarafını da tanımlar. Bu şekilde, Fig. 18.1.6 içinde gördüğümüz gibi, uzayımızı iki yarıya bölmenin bir yolunu bulduk, öyleki bir taraftaki tüm noktaların nokta çarpımı bir eşiğin altında ve diğer tarafında ise yukarıdadır.

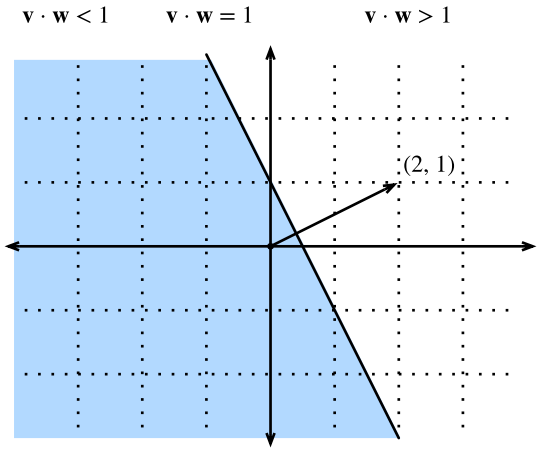


Fig. 18.1.6: Şimdi ifadenin eşitsizlik versiyonunu ele alırsak, hiperdüzlemimizin (bu durumda: sadece bir çizgi) uzayı iki yarıma ayırdığını görürüz.

Daha yüksek boyuttaki hikaye hemen hemen aynıdır. Şimdi $\mathbf{w} = [1, 2, 3]^\top$ alırsak ve $\mathbf{w} \cdot \mathbf{v} = 1$ ile üç boyuttaki noktaları sorarsak, verilen $\mathbf{w}$ vektörüne dik açıda bir düzlem elde ederiz. İki eşitsizlik yine düzlemin iki tarafını şu şekilde, Fig. 18.1.7, gösterildiği gibi tanımlar .

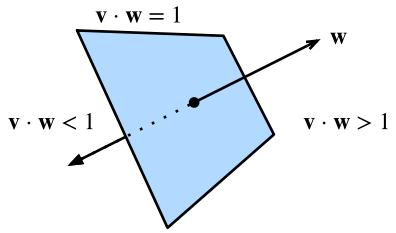


Fig. 18.1.7: Herhangi bir boyuttaki hiperdüzlemler, uzayı ikiye böler.

Bu noktada görselleştirme yeteneğimiz tükenirken, bizi bunu onlarca, yüzlerce veya milyarlarca boyutta yapmaktan hiçbir şey alıkoyamaz. Bu genellikle makinenin öğrendiği modeller hakkında düşünürken ortaya çıkar. Örneğin Section 3.4 içindeki gibi doğrusal sınıflandırma modellerini farklı hedef sınıfları ayıran hiperdüzlemleri bulma yöntemleri olarak anlayabiliriz. Bu bağlamda, bu tür hiperdüzlemlere genellikle *karar düzlemleri* adı verilir. Derin eğitilmiş sınıflandırma modellerinin çoğu, bir eşiksiz en büyük işleve (softmak) beslenen doğrusal bir katmanla sona erer, bu nedenle derin sinir ağının rolü, hedef sınıfların hiperdüzlemler tarafından temiz bir şekilde ayrılabileceği şekilde doğrusal olmayan bir gömme bulmak olarak yorumlanabilir.

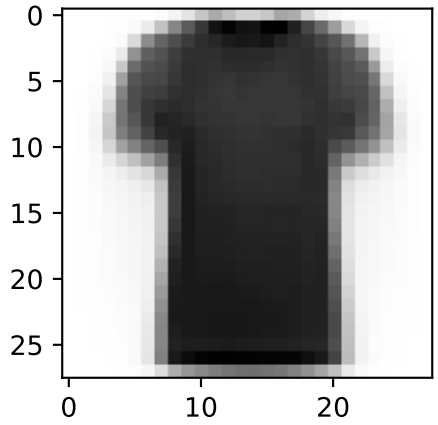
El yapımı bir örnek vermek gerekirse, Fashion MNIST veri kümesinden (Section 3.5) küçük tişört ve pantolon resimlerini sınıflandırmak için sadece ortalamalarını birleştiren bir vektör alıp karar

düzlemini ve göz kararı kaba bir eşiği tanımlayarak makul bir model oluşturabileceğimize dikkat edin. İlk önce verileri yükleyeceğiz ve ortalamaları hesaplayacağız.

```
# Veri kümesine yükle
trans = []
trans.append(transforms.ToTensor())
trans = transforms.Compose(trans)
train = torchvision.datasets.FashionMNIST(root="../data", transform=trans,
                                          train=True, download=True)
test = torchvision.datasets.FashionMNIST(root="../data", transform=trans,
                                         train=False, download=True)

X_train_0 = torch.stack(
    [x[0] * 256 for x in train if x[1] == 0]).type(torch.float32)
X_train_1 = torch.stack(
    [x[0] * 256 for x in train if x[1] == 1]).type(torch.float32)
X_test = torch.stack(
    [x[0] * 256 for x in test if x[1] == 0 or x[1] == 1]).type(torch.float32)
y_test = torch.stack([torch.tensor(x[1]) for x in test
                      if x[1] == 0 or x[1] == 1]).type(torch.float32)

# Ortalamaları hesapla
ave_0 = torch.mean(X_train_0, axis=0)
ave_1 = torch.mean(X_train_1, axis=0)
```

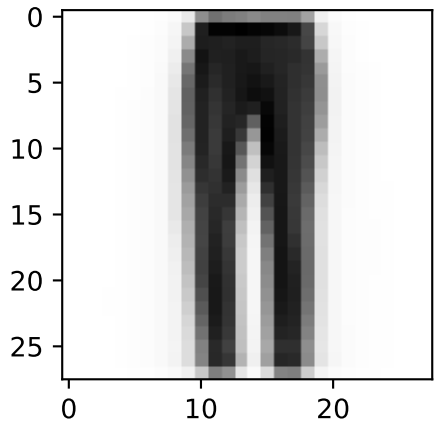
Bu ortalamaları ayrıntılı olarak incelemek bilgilendirici olabilir, bu yüzden neye benzediklerini çizelim. Bu durumda, ortalamanın gerçekten de bir tişörtün bulanık görüntüsüne benzediğini görüyoruz.

```
# Ortalama tişörtü çizdir
d2l.set_figsize()
d2l.plt.imshow(ave_0.reshape(28, 28).tolist(), cmap='Greys')
d2l.plt.show()
```



İkinci durumda, yine ortalamanın bulanık bir pantolon görüntüsüne benzediğini görüyoruz.

```
# Ortalama pantolonu çizdir
d2l.plt.imshow(ave_1.reshape(28, 28).tolist(), cmap='Greys')
d2l.plt.show()
```



Tamamen makine öğrenmeli bir çözümde, eşiği veri kümesinden öğrenecektik. Bu durumda, el ile eğitim verilerinde iyi görünen bir eşiği göz kararı aldık.

```
# Göz elde edilen eşiği ile test kümesi doğruluğunu yazdırın
w = (ave_1 - ave_0).T
# '@' is Matrix Multiplication operator in pytorch.
predictions = X_test.reshape(2000, -1) @ (w.flatten()) > -1500000

# Doğruluk
torch.mean((predictions.type(y_test.dtype) == y_test).float(), dtype=torch.float64)
```

```
/tmp/ipykernel_5428/1764750836.py:2: UserWarning: The use of x.T on tensors of␣
↪dimension other than 2 to reverse their shape is deprecated and it will throw␣
↪an error in a future release. Consider x.mT to transpose batches of matricesor x.
↪permute(*torch.arange(x.ndim - 1, -1, -1)) to reverse the dimensions of a tensor.␣
↪(Triggered internally at  ../aten/src/ATen/native/TensorShape.cpp:2985.)
  w = (ave_1 - ave_0).T
```

```
tensor(0.7870, dtype=torch.float64)
```

18.1.4 Doğrusal Dönüşümlerin Geometrisi

Section 2.3 ve yukarıdaki tartışmalar sayesinde, vektörlerin geometrisine, uzunluklara ve açılara dair sağlam bir anlayışa sahibiz. Bununla birlikte, tartışmayı atladığımız önemli bir nesne var ve bu, matrislerle temsil edilen doğrusal dönüşümlerin geometrik bir şekilde anlaşılmasıdır. Potansiyel olarak farklı yüksek boyutlu iki uzay arasında verileri dönüştürken matrislerin neler yapabileceğini tam olarak içselleştirmek, önemli bir uygulama gerektirir ve bu ek bölümün kapsamı dışındadır. Bununla birlikte, sezgimizi iki boyutta oluşturmaya başlayabiliriz.

Bir matrisimiz olduğunu varsayalım:

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}. \tag{18.1.11}$$

Bunu rastgele bir $\mathbf{v} = [x, y]^\top$ vektörüne uygulamak istersek, çarpar ve görürüz ki

$$\begin{aligned}\mathbf{A}\mathbf{v} &= \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\ &= \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix} \\ &= x \begin{bmatrix} a \\ c \end{bmatrix} + y \begin{bmatrix} b \\ d \end{bmatrix} \\ &= x \left\{ \mathbf{A} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\} + y \left\{ \mathbf{A} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\}.\end{aligned} \tag{18.1.12}$$

Bu, net bir şeyin bir şekilde anlaşılmaz hale geldiği garip bir hesaplama gibi görünebilir. Bununla birlikte, bize bir matrisin *herhangi* bir vektörü *iki belirli vektöre* göre nasıl dönüştürdüğünü yazabileceğimizi söyler: $[1, 0]^\top$ and $[0, 1]^\top$. Bu bir an için düşünmeye değer. Esasen sonsuz bir problemi (herhangi bir gerçel sayı çiftine olanı) sonlu bir probleme (bu belirli vektörlere ne olduğuna) indirgedik. Bu vektörler, uzayımızdaki herhangi bir vektörü bu *taban vektörlerin* ağırlıklı toplamı olarak yazabileceğimiz örnek bir *taban*dır.

Belli bir matrisi kullandığımızda ne olacağını çizelim

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ -1 & 3 \end{bmatrix}. \tag{18.1.13}$$

Belirli $\mathbf{v} = [2, -1]^\top$ vektörüne bakarsak, bunun $2 \cdot [1, 0]^\top + -1 \cdot [0, 1]^\top$ olduğunu görürüz, dolayısıyla A matrisinin bunu $2(\mathbf{A}[1, 0]^\top) + -1(\mathbf{A}[0, 1])^\top = 2[1, -1]^\top - [2, 3]^\top = [0, -5]^\top$'e göndereceğini biliyoruz. Bu mantığı dikkatlice takip edersek, diyelim ki tüm tamsayı nokta çiftlerinin ızgarasını (grid) göz önünde bulundurarak, matris çarpımının ızgarayı eğriltebileceğini, döndürebileceğini ve ölçekleyebileceğini görürüz, ancak ızgara yapısı numref:`fig_grid-transform` içinde gördüğünüz gibi kalmalıdır.

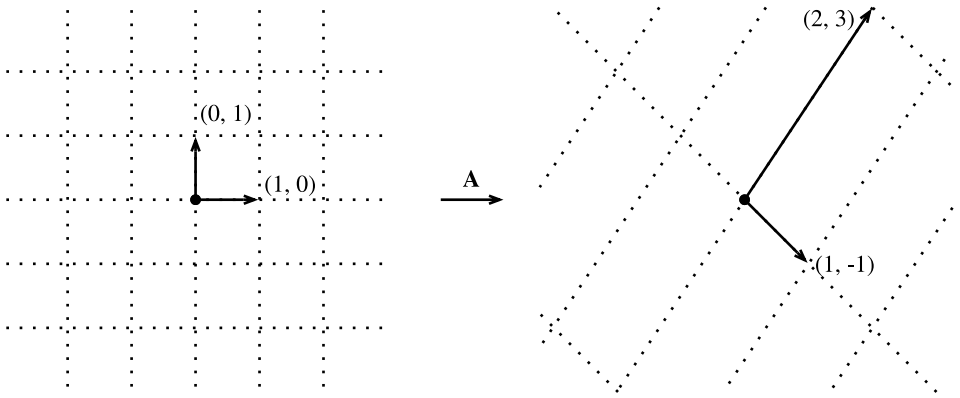


Fig. 18.1.8: Verilen temel vektörlere göre hareket eden **A** matrisi. Tüm ızgaranın onunla birlikte nasıl taşındığına dikkat edin.

Bu, matrisler tarafından temsil edilen doğrusal dönüşümler hakkında içselleştirilmesi gereken en önemli sezgisel noktadır. Matrisler, uzayın bazı bölümlerini diğerlerinden farklı şekilde bozma yeteneğine sahip değildir. Tüm yapabilecekleri, uzayımızdaki hakiki koordinatları almak ve onları eğriltmek, döndürmek ve ölçeklendirmektir.

Bazı çarpıklıklar şiddetli olabilir. Örneğin matris

$$\mathbf{B} = \begin{bmatrix} 2 & -1 \\ 4 & -2 \end{bmatrix}, \tag{18.1.14}$$

iki boyutlu düzlemin tamamını tek bir çizgiye sıkıştırır. Bu tür dönüşümleri tanımlamak ve bunlarla çalışmak daha sonraki bir bölümün konusudur, ancak geometrik olarak bunun yukarıda gördüğümüz dönüşüm türlerinden temelde farklı olduğunu görebiliriz. Örneğin, **A** matrisinden gelen sonuç, orijinal ızgaraya "geri eğilebilir". **B** matrisinden gelen sonuçlar olamaz çünkü $[1, 2]^\top$ vektörünün nereden geldiğini asla bilemeyiz — bu $[1, 1]^\top$ veya $[0, -1]^\top$ mıydı?

Bu resim 2×2 matrisi için olsa da, hiçbir şey öğrenilen dersleri daha yüksek boyutlara taşımamızı engellemiyor. $[1, 0, \ldots, 0]$ gibi benzer taban vektörleri alırsak ve matrisimizin onları nereye gönderdiğini görürsek, matris çarpımının, uğraştığımız boyut uzayında tüm uzayı nasıl bozduğu hakkında bir fikir edinmeye başlayabiliriz.

18.1.5 Doğrusal Bağımlılık

Matrisi tekrar düşünün

$$\mathbf{B} = \begin{bmatrix} 2 & -1 \\ 4 & -2 \end{bmatrix}. \tag{18.1.15}$$

Bu, tüm düzlemi $y = 2x$ tek doğruda yaşaması için sıkıştırır. Şimdi şu soru ortaya çıkıyor: Bunu sadece matrise bakarak tespit etmemizin bir yolu var mı? Cevap, gerçekten edebiliriz. $\mathbf{b}_1 = [2, 4]^\top$ ve $\mathbf{b}_2 = [-1, -2]^\top$, **B**'nin iki sütunu olsun. **B** matrisi tarafından dönüştürülen her şeyi, matrisin sütunlarının ağırlıklı toplamı olarak yazabileceğimizi unutmayın: $a_1\mathbf{b}_1 + a_2\mathbf{b}_2$ gibi. Buna *doğrusal birleşim (kombinasyon)* diyoruz. $\mathbf{b}_1 = -2 \cdot \mathbf{b}_2$ olması, bu iki sütunun herhangi bir doğrusal kombinasyonunu tamamen, mesela, $\mathbf{b}_2$ cinsinden yazabileceğimiz anlamına gelir, çünkü

$$a_1\mathbf{b}_1 + a_2\mathbf{b}_2 = -2a_1\mathbf{b}_2 + a_2\mathbf{b}_2 = (a_2 - 2a_1)\mathbf{b}_2. \tag{18.1.16}$$

Bu, sütunlardan birinin uzayda tek bir yön tanımlamadığından bir bakıma gereksiz olduğu anlamına gelir. Bu matrisin tüm düzlemi tek bir çizgiye indirdiğini gördüğümüz için bu bizi çok şaşırtmamalı. Dahası, $\mathbf{b}_1 = -2 \cdot \mathbf{b}_2$ doğrusal bağımlılığının bunu yakaladığını görüyoruz. Bunu iki vektör arasında daha simetrik hale getirmek için şöyle yazacağız:

$$\mathbf{b}_1 + 2 \cdot \mathbf{b}_2 = 0. \tag{18.1.17}$$

Genel olarak, eğer aşağıdaki denklem için *hepsi sıfıra eşit olmayan* $a_1, \ldots, a_k$ katsayıları varsa, bir $\mathbf{v}_1, \ldots, \mathbf{v}_k$ vektörler topluluğunun *doğrusal olarak bağımlı* olduğunu söyleyeceğiz:

$$\sum_{i=1}^{k} a_i\mathbf{v_i} = 0. \tag{18.1.18}$$

Bu durumda, vektörlerden birini diğerlerinin birtakım birleşimi olarak çözebilir ve onu etkili bir şekilde gereksiz hale getirebiliriz. Bu nedenle, bir matrisin sütunlarındaki doğrusal bir bağımlılık, matrisimizin uzayı daha düşük bir boyuta sıkıştırdığına bir kanıttır. Doğrusal bağımlılık yoksa, vektörlerin *doğrusal olarak bağımsız* olduğunu söyleriz. Bir matrisin sütunları doğrusal olarak bağımsızsa, sıkıştırma gerçekleşmez ve işlem geri alınabilir.

18.1.6 Kerte (Rank)

Genel bir $n \times m$ matrisimiz varsa, matrisin hangi boyut uzayına eşlendiğini sormak mantıklıdır. Cevabımız *kerte* olarak bilinen bir kavram olacaktır. Önceki bölümde, doğrusal bir bağımlılığın uzayın daha düşük bir boyuta sıkıştırılmasına tanıklık ettiğini ve bu nedenle bunu kerte kavramını tanımlamak için kullanabileceğimizi ifade ettik. Özellikle, bir $\mathbf{A}$ matrisinin kertesi, sütunların tüm alt kümeleri arasındaki doğrusal bağımsız en büyük sütun sayısıdır. Örneğin, matris

$$\mathbf{B} = \begin{bmatrix} 2 & 4 \\ -1 & -2 \end{bmatrix}, \tag{18.1.19}$$

kerte$(B) = 1$'e sahiptir, çünkü iki sütunu doğrusal olarak bağımlıdır, ancak iki sütun da kendi başına doğrusal olarak bağımlı değildir. Daha zorlu bir örnek için,

$$\mathbf{C} = \begin{bmatrix} 1 & 3 & 0 & -1 & 0 \\ -1 & 0 & 1 & 1 & -1 \\ 0 & 3 & 1 & 0 & -1 \\ 2 & 3 & -1 & -2 & 1 \end{bmatrix}, \tag{18.1.20}$$

ve $\mathbf{C}$'nin kertesinin iki olduğunu gösterebiliriz, çünkü örneğin ilk iki sütun doğrusal olarak bağımsızdır, ancak herhangi bir dört sütunlu toplulukta üç sütun bağımlıdır.

Bu prosedür, açıklandığı gibi, çok verimsizdir. Verdiğimiz matrisin sütunlarının her alt kümesine bakmayı gerektirir ve bu nedenle sütun sayısına bağlı, potansiyel olarak üsteldir. Daha sonra bir matrisin kertesini hesaplamanın hesaplama açısından daha verimli bir yolunu göreceğiz, ancak şimdilik, kavramın iyi tanımlandığını görmek ve anlamı anlamak yeterlidir.

18.1.7 Tersinirlik (Invertibility)

Yukarıda doğrusal olarak bağımlı sütunları olan bir matris ile çarpmanın geri alınamayacağını gördük, yani girdiyi her zaman kurtarabilecek ters işlem yoktur. Bununla birlikte, tam kerteli bir matrisle çarpma durumunda (yani, $\mathbf{A}$, $n \times n$'lik n kerteli matristir), bunu her zaman geri alabilmeliyiz. Şu matrisi düşünün:

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}. \tag{18.1.21}$$

Bu, köşegen boyunca birlerin ve başka yerlerde sıfırların bulunduğu matristir. Buna *birim* matris diyoruz. Uygulandığında verilerimizi değiştirmeden bırakan matristir. $\mathbf{A}$ matrisimizin yaptıklarını geri alan bir matris bulmak için, şu şekilde bir $\mathbf{A}^{-1}$ matrisi bulmak istiyoruz:

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I}. \tag{18.1.22}$$

Buna bir sistem olarak bakarsak, $n \times n$ bilinmeyenli ($\mathbf{A}^{-1}$'nin girdileri) ve $n \times n$ denklemimiz var ($\mathbf{A}^{-1}\mathbf{A}$ çarpımının her girdisi ve $\mathbf{I}$'nin her girdisi arasında uyulması gereken eşitlik), bu nedenle genel olarak bir çözümün var olmasını beklemeliyiz. Nitekim bir sonraki bölümde sıfır olmadığı sürece çözüm bulabileceğimizi gösterme özelliğine sahip olan *determinant* adlı bir miktar göreceğiz. Böyle bir $\mathbf{A}^{-1}$ matrise, *ters* matris diyoruz. Örnek olarak, eğer $\mathbf{A}$ genel bir 2×2 matris ise

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \tag{18.1.23}$$

o zaman tersinin şöyle olduğunu görebiliriz:

$$\frac{1}{ad - bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}. \tag{18.1.24}$$

Yukarıdaki formülün verdiği ters ile çarpmanın pratikte işe yaradığını görmek için test edebiliriz.

```
M = torch.tensor([[1, 2], [1, 4]], dtype=torch.float32)
M_inv = torch.tensor([[2, -1], [-0.5, 0.5]])
M_inv @ M
```

```
tensor([[1., 0.],
        [0., 1.]])
```

Sayısal (Numerik) Sorunlar

Bir matrisin tersi teoride yararlı olsa da, pratikte bir problemi çözmek için çoğu zaman matris tersini *kullanmak* istemediğimizi söylemeliyiz. Genel olarak,

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{18.1.25}$$

gibi doğrusal denklemleri çözmek için sayısal olarak çok daha kararlı algoritmalar vardır, aşağıdaki gibi tersini hesaplamaktan ve çarpmaktan daha çok tercih edebileceğimiz yöntemlerdir.

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}. \tag{18.1.26}$$

Küçük bir sayıya bölünmenin sayısal kararsızlığa yol açması gibi, düşük kerteye olmaya yakın bir matrisin ters çevrilmesi de kararsızlığa neden olabilir.

Dahası, $\mathbf{A}$ matrisinin *seyrek* olması yaygındır, yani sadece az sayıda sıfır olmayan değer içerir. Örnekleri araştıracak olsaydık, bunun tersin de seyrek olduğu anlamına gelmediğini görürdük. $\mathbf{A}$, yalnızca 5 milyon tanesi sıfır olmayan girdileri olan 1 milyona 1 milyonluk bir matris olsa bile (ve bu nedenle yalnızca bu 5 milyon girdiyi saklamamız gerekir), tersi genellikle hemen hemen hepsi eksi değer olmayan tüm girdilere sahip olacaktır ki tüm 1M^2 girdiyi saklamamızı gerektirir — bu da 1 trilyon girdidir!

Doğrusal cebir ile çalışırken sıkça karşılaşılan çetrefilli sayısal sorunlara tam olarak dalacak vaktimiz olmasa da, ne zaman dikkatli bir şekilde ilerlemeniz gerektiği konusunda size biraz önsezi sağlamak istiyoruz ve pratikte genellikle ters çevirmekten kaçınmak yararlı bir kuraldır.

18.1.8 Determinant

Doğrusal cebirin geometrik görünümü, *determinant* olarak bilinen temel bir miktarı yorumlamanın sezgisel bir yolunu sunar. Önceki ızgara görüntüsünü, ama şimdi vurgulanmış bölgeyle (Fig. 18.1.9) düşünün.

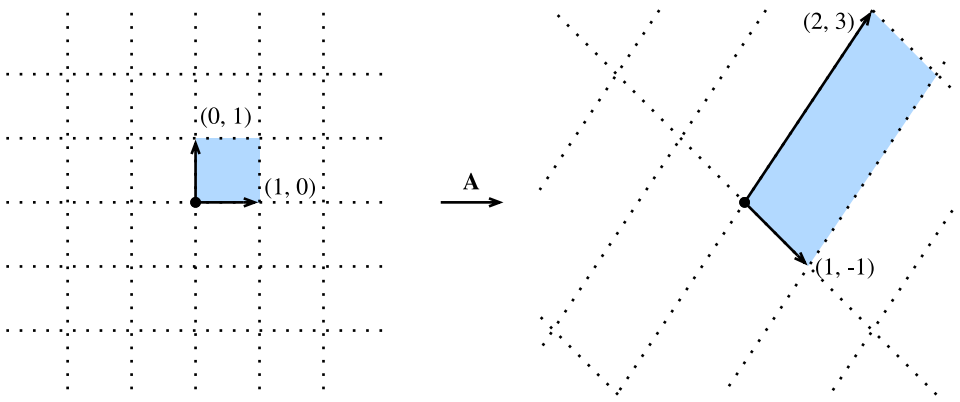


Fig. 18.1.9: **A** matrisi yine ızgarayı bozuyor. Bu sefer, vurgulanan kareye ne olduğuna özellikle dikkat çekmek istiyoruz.

Vurgulanan kareye bakın. Bu, kenarları $(0, 1)$ ve $(1, 0)$ ile verilen bir karedir ve dolayısıyla bir birim alana sahiptir. **A** bu kareyi dönüştürdükten sonra, bunun bir paralelkenar olduğunu görürüz. Bu paralelkenarın başladığımızdaki aynı alana sahip olması için hiçbir neden yok ve aslında burada gösterilen özel durumda aşağıdaki matristir.

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ -1 & 3 \end{bmatrix}, \tag{18.1.27}$$

Bu paralelkenarın alanını hesaplamak ve alanın 5 olduğunu elde etmek koordinat geometrisinde bir alıştırmadır.

Genel olarak, bir matrisimiz varsa,

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \tag{18.1.28}$$

biraz hesaplamayla elde edilen paralelkenarın alanının $ad - bc$ olduğunu görebiliriz. Bu alan, *determinant* olarak adlandırılır.

Bunu bazı örnek kodlarla hızlıca kontrol edelim.

```
torch.det(torch.tensor([[1, -1], [2, 3]], dtype=torch.float32))
```

```
tensor(5.)
```

Aramızdaki kartal gözlüler, bu ifadenin sıfır, hatta negatif olabileceğini fark edecek. Negatif terim için, bu genel olarak matematikte ele alınan bir ifade meselesidir: Eğer matris şekli ters çevirirse, alanın aksine çevrildiğini söyleriz. Şimdi determinant sıfır olduğunda daha fazlasını öğreneceğimizi görelim.

Bir düşünelim.

$$\mathbf{B} = \begin{bmatrix} 2 & 4 \\ -1 & -2 \end{bmatrix}. \tag{18.1.29}$$

Bu matrisin determinantını hesaplarsak, $2 \cdot (-2) - 4 \cdot (-1) = 0$ elde ederiz. Yukarıdaki anlayışımıza göre, bu mantıklı. **B**, orijinal görüntüdeki kareyi sıfır alana sahip bir çizgi parçasına sıkıştırır. Ve aslında, dönüşümden sonra sıfır alana sahip olmanın tek yolu, daha düşük boyutlu bir alana

sıkıştırılmaktır. Böylece, aşağıdaki sonucun doğru olduğunu görüyoruz: Bir A matrisinin, ancak ve ancak determinantı sıfıra eşit değilse tersi hesaplanabilir.

Son bir yorum olarak, düzlemde herhangi bir figürün çizildiğini hayal edin. Bilgisayar bilimcileri gibi düşünürsek, bu şekli küçük kareler toplamına ayırabiliriz, böylece şeklin alanı özünde sadece ayrıştırmadaki karelerin sayısı olur. Şimdi bu rakamı bir matrisle dönüştürürsek, bu karelerin her birini, determinant tarafından verilen alana sahip olan paralelkenarlara göndeririz. Herhangi bir şekil için determinantın, bir matrisin herhangi bir şeklin alanını ölçeklendirdiği (işaretli) sayıyı verdiğini görüyoruz.

Daha büyük matrisler için belirleyicilerin hesaplanması zahmetli olabilir, ancak sezgi aynıdır. Determinant, $n \times n$ matrislerin n-boyutlu hacimlerini ölçeklendiren faktör olarak kalır.

18.1.9 Tensörler ve Genel Doğrusal Cebir İşlemleri

Section 2.3'de tensör kavramı tanıtıldı. Bu bölümde, tensör daralmalarına (büzülmesine) (matris çarpımının tensör eşdeğeri) daha derinlemesine dalacağız ve bir dizi matris ve vektör işlemi üzerinde nasıl birleşik bir görünüm sağlayabileceğini göreceğiz.

Matrisler ve vektörlerle, verileri dönüştürmek için onları nasıl çarpacağımızı biliyorduk. Bize yararlı olacaksa, tensörler için de benzer bir tanıma ihtiyacımız var. Matris çarpımını düşünün:

$$\mathbf{C} = \mathbf{AB}, \tag{18.1.30}$$

Veya eşdeğer olarak

$$c_{i,j} = \sum_k a_{i,k} b_{k,j}. \tag{18.1.31}$$

Bu model tensörler için tekrar edebileceğimiz bir modeldir. Tensörler için, neyin toplanacağına dair evrensel olarak seçilebilecek tek bir durum yoktur, bu yüzden tam olarak hangi indeksleri toplamak istediğimizi belirlememiz gerekir. Örneğin düşünün,

$$y_{il} = \sum_{jk} x_{ijkl} a_{jk}. \tag{18.1.32}$$

Böyle bir dönüşüme *tensör daralması* denir. Tek başına matris çarpımının olduğundan çok daha esnek bir dönüşüm ailesini temsil edebilir.

Sık kullanılan bir gösterimsel sadeleştirme olarak, toplamın ifadede birden fazla kez yer alan indislerin üzerinde olduğunu fark edebiliriz, bu nedenle insanlar genellikle, toplamın örtülü olarak tüm tekrarlanan indisler üzerinden alındığı *Einstein gösterimi* ile çalışır. Bu aşağıdaki kompakt ifadeyi verir:

$$y_{il} = x_{ijkl} a_{jk}. \tag{18.1.33}$$

Doğrusal Cebirden Yaygın Örnekler

Daha önce gördüğümüz doğrusal cebirsel tanımların kaçının bu sıkıştırılmış tensör gösteriminde ifade edilebileceğini görelim:

- $\mathbf{v} \cdot \mathbf{w} = \sum_i v_i w_i$
- $\|\mathbf{v}\|_2^2 = \sum_i v_i v_i$

- $(\mathbf{Av})_i = \sum_j a_{ij} v_j$
- $(\mathbf{AB})_{ik} = \sum_j a_{ij} b_{jk}$
- $\mathrm{tr}(\mathbf{A}) = \sum_i a_{ii}$

Bu şekilde, çok sayıda özel gösterimi kısa tensör ifadeleriyle değiştirebiliriz.

Kodla İfade Etme

Tensörler de kod içinde esnek bir şekilde çalıştırılabilir. Section 2.3 içinde görüldüğü gibi, aşağıda gösterildiği gibi tensörler oluşturabiliriz.

```
# Tensörleri tanımla
B = torch.tensor([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
A = torch.tensor([[1, 2], [3, 4]])
v = torch.tensor([1, 2])

# Şekilleri yazdır
A.shape, B.shape, v.shape
```

```
(torch.Size([2, 2]), torch.Size([2, 2, 3]), torch.Size([2]))
```

Einstein toplamı doğrudan uygulanır. Einstein toplamında ortaya çıkan indisler bir dizi olarak aktarılabilir ve ardından işlem yapılan tensörler eklenebilir. Örneğin, matris çarpımını uygulamak için, yukarıda görülen Einstein toplamını ($\mathbf{Av} = a_{ij} v_j$) düşünebilir ve uygulamayı (gerçeklemeyi) elde etmek için indisleri söküp atabiliriz:

```
# Matris çarpımını yeniden uygula
torch.einsum("ij, j -> i", A, v), A@v
```

```
(tensor([ 5, 11]), tensor([ 5, 11]))
```

Bu oldukça esnek bir gösterimdir. Örneğin, geleneksel olarak şu şekilde yazılanı hesaplamak istiyorsak,

$$c_{kl} = \sum_{ij} \mathbf{b}_{ijk} \mathbf{a}_{il} v_j. \tag{18.1.34}$$

Einstein toplamı aracılığıyla şu şekilde uygulanabilir:

```
torch.einsum("ijk, il, j -> kl", B, A, v)
```

```
tensor([[ 90, 126],
        [102, 144],
        [114, 162]])
```

Bu gösterim insanlar için okunabilir ve etkilidir, ancak herhangi bir nedenle programlı olarak bir tensör daralması üretmeniz gerekirse hantaldır. Bu nedenle `einsum`, her tensör için tamsayı indisleri sağlayarak alternatif bir gösterim sağlar. Örneğin, aynı tensör daralması şu şekilde de yazılabilir:

```
# PyTorch bu tür gösterimi desteklemez.
```

Her iki gösterim, tensör daralmalarının kodda kısa ve verimli bir şekilde temsiline izin verir.

18.1.10 Özet

- Vektörler, uzayda geometrik olarak noktalar veya yönler olarak yorumlanabilir.
- Nokta çarpımları, keyfi olarak yüksek boyutlu uzaylar için açı kavramını tanımlar.
- Hiperdüzlemler, doğruların ve düzlemlerin yüksek boyutlu genellemeleridir. Genellikle bir sınıflandırma görevinde son adım olarak kullanılan karar düzlemlerini tanımlamak için kullanılabilirler.
- Matris çarpımı, geometrik olarak, temel koordinatların tekdüze bozulmaları olarak yorumlanabilir. Vektörleri dönüştürmenin çok kısıtlı, ancak matematiksel olarak temiz bir yolunu temsil ederler.
- Doğrusal bağımlılık, bir vektör topluluğunun beklediğimizden daha düşük boyutlu bir uzayda olduğunu anlamanın bir yoludur (diyelim ki 2 boyutunda bir uzayda yaşayan 3 vektörünüz var). Bir matrisin kertesi, doğrusal olarak bağımsız olan sütunlarının en büyük alt kümesinin ebadıdır.
- Bir matrisin tersi tanımlandığında, matris tersi bulma, ilkinin eylemini geri alan başka bir matris bulmamızı sağlar. Matris tersi bulma teoride faydalıdır, ancak sayısal kararsızlık nedeniyle pratikte dikkat gerektirir.
- Determinantlar, bir matrisin bir alanı ne kadar genişlettiğini veya daralttığını ölçmemizi sağlar. Sıfır olmayan bir determinant, tersinir (tekil olmayan) bir matris anlamına gelir ve sıfır değerli bir determinant, matrisin tersinemez (tekil) olduğu anlamına gelir.
- Tensör daralmaları ve Einstein toplamı, makine öğrenmesinde görülen hesaplamaların çoğunu ifade etmek için düzgün ve temiz bir gösterim sağlar.

18.1.11 Alıştırmalar

1. Aralarındaki açı nedir?

$$\vec{v}_1 = \begin{bmatrix} 1 \\ 0 \\ -1 \\ 2 \end{bmatrix}, \qquad \vec{v}_2 = \begin{bmatrix} 3 \\ 1 \\ 0 \\ 1 \end{bmatrix} \tag{18.1.35}$$

2. Doğru veya yanlış: $\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$ ve $\begin{bmatrix} 1 & -2 \\ 0 & 1 \end{bmatrix}$ birbirinin tersi mi?
3. Düzlemde 100m^2 alanına sahip bir şekil çizdiğimizi varsayalım. Şeklin aşağıdaki matrise göre dönüştürüldükten sonraki alanı nedir?

$$\begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix}. \tag{18.1.36}$$

4. Aşağıdaki vektör kümelerinden hangisi doğrusal olarak bağımsızdır?

- $\left\{\begin{pmatrix}1\\0\\-1\end{pmatrix}, \begin{pmatrix}2\\1\\-1\end{pmatrix}, \begin{pmatrix}3\\1\\1\end{pmatrix}\right\}$
- $\left\{\begin{pmatrix}3\\1\\1\end{pmatrix}, \begin{pmatrix}1\\1\\1\end{pmatrix}, \begin{pmatrix}0\\0\\0\end{pmatrix}\right\}$
- $\left\{\begin{pmatrix}1\\1\\0\end{pmatrix}, \begin{pmatrix}0\\1\\-1\end{pmatrix}, \begin{pmatrix}1\\0\\1\end{pmatrix}\right\}$

5. Bazı a, b, c ve d değerleri için $A = \begin{bmatrix}c\\d\end{bmatrix} \cdot \begin{bmatrix}a & b\end{bmatrix}$ olarak yazılmış bir matrisiniz olduğunu varsayalım . Doğru mu yanlış mı: Böyle bir matrisin determinantı her zaman 0'dır?
6. $e_1 = \begin{bmatrix}1\\0\end{bmatrix}$ ve $e_2 = \begin{bmatrix}0\\1\end{bmatrix}$ vektörleri diktir. Ae_1 ve Ae_2'nin dik olması için A matrisindeki koşul nedir?
7. Rasgele bir A matrisi için Einstein gösterimi ile tr($\mathbf{A}^4$)'u nasıl yazabilirsiniz?

Tartışmalar[217]

18.2 Özayrışmalar

Özdeğerler genellikle doğrusal cebiri incelerken karşılaşacağımız en yararlı kavramlardan biridir, ancak yeni başlayanlar olarak önemlerini gözden kaçırmak kolaydır. Aşağıda, özayrışmayı tanıtıyoruz ve neden bu kadar önemli olduğuna dair bir fikir aktarmaya çalışıyoruz.

Aşağıdaki girdileri içeren bir A matrisimiz olduğunu varsayalım:

$$\mathbf{A} = \begin{bmatrix}2 & 0\\0 & -1\end{bmatrix}. \tag{18.2.1}$$

Herhangi bir $\mathbf{v} = [x, y]^\top$ vektörüne A uygularsak, bir $\mathbf{Av} = [2x, -y]^\top$ vektörü elde ederiz. Bunun sezgisel bir yorumu var: Vektörü x yönünde iki kat geniş olacak şekilde uzatın ve ardından y yönünde ters çevirin.

Ancak, bir şeyin değişmeden kaldığı *bazı* vektörler vardır. Yani $[1, 0]^\top$, $[2, 0]^\top$'e ve $[0, 1]^\top$, $[0, -1]^\top$'e gönderilir. Bu vektörler hala aynı doğrudadır ve tek değişiklik, matrisin onları sırasıyla 2 ve -1 çarpanı ile genişletmesidir. Bu tür vektörlere *özvektörler* diyoruz ve bunların uzatıldıkları çarpan da *özdeğerler*dir.

Genel olarak, bir λ sayısı ve şöyle bir $\mathbf{v}$ vektörü bulabilirsek

$$\mathbf{Av} = \lambda\mathbf{v} \tag{18.2.2}$$

$\mathbf{v}$ A için bir özvektör ve λ bir özdeğerdir deriz.

[217] https://discuss.d2l.ai/t/1084

18.2.1 Özdeğerleri Bulma

Onları nasıl bulacağımızı anlayalım. Her iki taraftan $\lambda\mathbf{v}$ çıkararak ve ardından vektörü dışarıda bırakarak, yukarıdakinin şuna eşdeğer olduğunu görürüz:

$$(\mathbf{A} - \lambda\mathbf{I})\mathbf{v} = 0. \tag{18.2.3}$$

(18.2.3) denkleminin gerçekleşmesi için, $(\mathbf{A} - \lambda\mathbf{I})$'nın bir yönü sıfıra kadar sıkıştırması gerektiğini görüyoruz, bu nedenle tersinir değildir ve bu nedenle determinant sıfırdır. Böylece, *özdeğerleri*, λ değerinin ne zaman $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$ olduğunu bularak bulabiliriz. Özdeğerleri bulduktan sonra, ilişkili *özvektör(leri)* bulmak için $\mathbf{Av} = \lambda\mathbf{v}$'yı çözebiliriz.

Bir örnek

Bunu daha zorlu bir matrisle görelim

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 2 & 3 \end{bmatrix}. \tag{18.2.4}$$

$\det(\mathbf{A} - \lambda\mathbf{I}) = 0$ olarak düşünürsek, bunun $0 = (2 - \lambda)(3 - \lambda) - 2 = (4 - \lambda)(1 - \lambda)$ polinom denklemine eşdeğer olduğunu görürüz. Böylece, iki özdeğer 4 ve 1'dir. İlişkili vektörleri bulmak için bunu çözmemiz gerekir:

$$\begin{bmatrix} 2 & 1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} \text{ ve } \begin{bmatrix} 2 & 1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 4x \\ 4y \end{bmatrix}. \tag{18.2.5}$$

Bunu sırasıyla $[1, -1]^\top$ ve $[1, 2]^\top$ vektörleriyle çözebiliriz.

Bunu yerleşik `numpy.linalg.eig` rutinini kullanarak kodda kontrol edebiliriz.

```
%matplotlib inline
import torch
from IPython import display
from d2l import torch as d2l

torch.eig(torch.tensor([[2, 1], [2, 3]], dtype=torch.float64),
          eigenvectors=True)
```

```
/tmp/ipykernel_4566/1653148152.py:6: UserWarning: torch.eig is deprecated in favor of torch.
↪linalg.eig and will be removed in a future PyTorch release.
torch.linalg.eig returns complex tensors of dtype cfloat or cdouble rather than real tensors␣
↪mimicking complex tensors.
L, _ = torch.eig(A)
should be replaced with
L_complex = torch.linalg.eigvals(A)
and
L, V = torch.eig(A, eigenvectors=True)
should be replaced with
L_complex, V_complex = torch.linalg.eig(A) (Triggered internally at  ../aten/src/ATen/native/
↪BatchLinearAlgebra.cpp:3427.)
  torch.eig(torch.tensor([[2, 1], [2, 3]], dtype=torch.float64),
```

```
torch.return_types.eig(
eigenvalues=tensor([[1., 0.],
        [4., 0.]], dtype=torch.float64),
eigenvectors=tensor([[-0.7071, -0.4472],
        [ 0.7071, -0.8944]], dtype=torch.float64))
```

`numpy` kütüphanesinin özvektörleri bir uzunluğunda normalleştirdiğini, oysa bizimkileri keyfi uzunlukta kabul ettiğimizi unutmayın. Ek olarak, işaret seçimi keyfidir. Bununla birlikte, hesaplanan vektörler, aynı özdeğerlerle elle bulduklarımıza paraleldir.

18.2.2 Matris Ayrıştırma

Önceki örnek ile bir adım daha devam edelim.

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ -1 & 2 \end{bmatrix}, \tag{18.2.6}$$

Sütunları $\mathbf{A}$ matrisinin özvektörleri olduğu matris olsun.

$$\mathbf{\Sigma} = \begin{bmatrix} 1 & 0 \\ 0 & 4 \end{bmatrix}, \tag{18.2.7}$$

Köşegen üzerinde ilişkili özdeğerleri olan matris olsun. Özdeğerlerin ve özvektörlerin tanımı bize şunu söyler:

$$\mathbf{A}\mathbf{W} = \mathbf{W}\mathbf{\Sigma}. \tag{18.2.8}$$

W matrisi ters çevrilebilir, bu yüzden her iki tarafı da sağdan W^{-1} ile çarpabiliriz ve görürüz ki:

$$\mathbf{A} = \mathbf{W}\mathbf{\Sigma}\mathbf{W}^{-1}. \tag{18.2.9}$$

Bir sonraki bölümde bunun bazı güzel sonuçlarını göreceğiz, ancak şimdilik sadece, doğrusal olarak bağımsız özvektörlerin tam bir topluluğunu bulabildiğimiz sürece böyle bir ayrışmanın var olacağını bilmemiz gerekiyor (böylece W tersinirdir).

18.2.3 Özayrışmalar Üzerinde İşlemler

Özayrışmalar, (18.2.9), ilgili güzel bir şey, genellikle karşılaştığımız birçok işlemi özayrışmalar açısından temiz bir şekilde yazabilmemizdir. İlk örnek olarak şunları düşünün:

$$\mathbf{A}^n = \overbrace{\mathbf{A}\cdots\mathbf{A}}^{n \text{ kere}} = \overbrace{(\mathbf{W}\mathbf{\Sigma}\mathbf{W}^{-1})\cdots(\mathbf{W}\mathbf{\Sigma}\mathbf{W}^{-1})}^{n \text{ kere}} = \mathbf{W}\overbrace{\mathbf{\Sigma}\cdots\mathbf{\Sigma}}^{n \text{ kere}}\mathbf{W}^{-1} = \mathbf{W}\mathbf{\Sigma}^n\mathbf{W}^{-1}. \tag{18.2.10}$$

Bu bize, bir matrisin herhangi bir pozitif kuvveti için, özayrışmasının özdeğerlerin sadece aynı kuvvete yükseltilmesiyle elde edildiğini söyler. Aynısı negatif kuvvetler için de gösterilebilir, bu nedenle bir matrisi tersine çevirmek istiyorsak, yapmamız gereken yalnızca

$$\mathbf{A}^{-1} = \mathbf{W}\mathbf{\Sigma}^{-1}\mathbf{W}^{-1}, \tag{18.2.11}$$

veya başka bir deyişle, her bir özdeğeri ters çevirin. Bu, her özdeğer sıfır olmadığı sürece işe yarayacaktır, bu nedenle tersinebilirin sıfır özdeğer olmamasıyla aynı olduğunu görürüz.

Aslında, ek çalışma şunu gösterebilir: Eğer $\lambda_1, \ldots, \lambda_n$ bir matrisin özdeğerleriyse, o zaman o matrisin determinantı

$$\det(\mathbf{A}) = \lambda_1 \cdots \lambda_n, \tag{18.2.12}$$

veya tüm özdeğerlerin çarpımıdır. Bu sezgisel olarak mantıklıdır, çünkü $\mathbf{W}$ ne kadar esnetme yaparsa, W^{-1} bunu geri alır, dolayısıyla sonunda gerçekleşen tek uzatma köşegen matris $\mathbf{\Sigma}$ ile çarpma yoluyla olur, ki o da köşegen elemanların çarpımına göre hacimleri uzatır.

Son olarak, kertenin matrisinizin en büyük doğrusal olarak bağımsız sütunlarının sayısı olduğunu hatırlayın. Özayrışmayı yakından inceleyerek, kertenin $\mathbf{A}$'nın sıfır olmayan özdeğerlerinin sayısıyla aynı olduğunu görebiliriz.

Örnekler devam edebilirdi, ancak umarız ki mesele açıktır: Özayrışmalar, birçok doğrusal-cebirsel hesaplamayı basitleştirebilir ve birçok sayısal (numerik) algoritmanın ve doğrusal cebirde yaptığımız analizlerin çoğunun altında yatan temel bir işlemdir.

18.2.4 Simetrik Matrislerin Özayrışmaları

Yukarıdaki işlemin çalışması için yeterli doğrusal olarak bağımsız özvektör bulmak her zaman mümkün değildir. Örneğin matris

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \tag{18.2.13}$$

tek bir özvektöre, $(1, 0)^\top$, sahiptir. Bu tür matrisleri işlemek için, ele alabileceğimizden daha gelişmiş tekniklere ihtiyacımız var (Jordan Normal Formu veya Tekil Değer Ayrıştırması gibi). Dikkatimizi sık sık tam bir özvektörler kümesinin varlığını garanti edebileceğimiz matrislere sınırlamamız gerekecek.

En sık karşılaşılan aile, $\mathbf{A} = \mathbf{A}^\top$ olan *simetrik matrislerdir*. Bu durumda, W'i bir *dikgen (ortogonal) matris* olarak alabiliriz — sütunlarının tümü birbirine dik açıda birim uzunluklu vektörler olan bir matris, burada $\mathbf{W}^\top = \mathbf{W}^{-1}$'dir - ve tüm özdeğerler gerçel olacaktır. Böylece, bu özel durumda (18.2.9) denklemini şöyle yazabiliriz

$$\mathbf{A} = \mathbf{W}\mathbf{\Sigma}\mathbf{W}^\top. \tag{18.2.14}$$

18.2.5 Gershgorin Çember Teoremi

Özdeğerlerle sezgisel olarak akıl yürütmek genellikle zordur. Rasgele bir matris sunulursa, özdeğerleri hesaplamadan ne oldukları hakkında söylenebilecek çok az şey vardır. Bununla birlikte, en büyük değerler köşegen üzerindeyse, iyi bir tahmin yapmayı kolaylaştıran bir teorem vardır.

$\mathbf{A} = (a_{ij})$ herhangi bir $(n \times n)$ kare matris olsun. Şöyle tanımlayalım: $r_i = \sum_{j \neq i} |a_{ij}|$. $\mathcal{D}_i$ karmaşık düzlemde a_{ii} merkezli r_i yarıçaplı diski temsil etsin. Daha sonra, $\mathbf{A}$'nın her özdeğeri $\mathcal{D}_i$'den birinin içinde bulunur.

Bunu açmak biraz zaman alabilir, o yüzden bir örneğe bakalım. Şu matrisi düşünün:

$$\mathbf{A} = \begin{bmatrix} 1.0 & 0.1 & 0.1 & 0.1 \\ 0.1 & 3.0 & 0.2 & 0.3 \\ 0.1 & 0.2 & 5.0 & 0.5 \\ 0.1 & 0.3 & 0.5 & 9.0 \end{bmatrix}. \tag{18.2.15}$$

Elimizde $r_1 = 0.3$, $r_2 = 0.6$, $r_3 = 0.8$ ve $r_4 = 0.9$ var. Matris simetriktir, bu nedenle tüm özdeğerler gerçeldir. Bu, tüm özdeğerlerimizin aşağıdaki aralıklardan birinde olacağı anlamına gelir.

$$[a_{11} - r_1, a_{11} + r_1] = [0.7, 1.3], \tag{18.2.16}$$

$$[a_{22} - r_2, a_{22} + r_2] = [2.4, 3.6], \tag{18.2.17}$$

$$[a_{33} - r_3, a_{33} + r_3] = [4.2, 5.8], \tag{18.2.18}$$

$$[a_{44} - r_4, a_{44} + r_4] = [8.1, 9.9]. \tag{18.2.19}$$

Sayısal hesaplamanın gerçekleştirilmesi, özdeğerlerin yaklaşık 0.99, 2.97, 4.95, 9.08 olduğunu ve rahatlıkla sağlanan aralıklar içinde olduğunu gösterir.

```
A = torch.tensor([[1.0, 0.1, 0.1, 0.1],
                  [0.1, 3.0, 0.2, 0.3],
                  [0.1, 0.2, 5.0, 0.5],
                  [0.1, 0.3, 0.5, 9.0]])

v, _ = torch.eig(A)
v
```

```
tensor([[0.9923, 0.0000],
        [9.0803, 0.0000],
        [4.9539, 0.0000],
        [2.9734, 0.0000]])
```

Bu şekilde, özdeğerler yaklaşık olarak tahmin edilebilir ve köşegenin diğer tüm öğelerden önemli ölçüde daha büyük olması durumunda yaklaşımlar oldukça doğru olacaktır.

Bu küçük bir şey, ancak özayrışma gibi karmaşık ve incelikli bir konuda, yapabileceğimiz herhangi bir sezgisel kavrayışa sahip olmak iyidir.

18.2.6 Yararlı Bir Uygulama: Yinelenen Eşlemelerin Gelişimi

Artık özvektörlerin prensipte ne olduğunu anladığımıza göre, bunların sinir ağı davranışının merkezinde olan bir problemin derinlemesine anlaşılmasını sağlamak için nasıl kullanılabileceklerini görelim: Uygun ağırlık ilklenmesi.

Uzun Vadeli Davranış Olarak Özvektörler

Derin sinir ağlarının ilklenmesinin tam matematiksel araştırması, metnin kapsamı dışındadır, ancak özdeğerlerin bu modellerin nasıl çalıştığını görmemize nasıl yardımcı olabileceğini anlamak için burada bir basit örnek sürümünü görebiliriz. Bildiğimiz gibi, sinir ağları, doğrusal olmayan işlemlerle doğrusal dönüşüm katmanlarını serpiştirerek çalışır. Burada basitleştirmek için, doğrusal olmayanlığın olmadığını ve dönüşümün tek bir tekrarlanan matris işlemi A olduğunu varsayacağız, böylece modelimizin çıktısı şu şekildedir:

$$\mathbf{v}_{out} = \mathbf{A} \cdot \mathbf{A} \cdots \mathbf{A}\mathbf{v}_{in} = \mathbf{A}^N \mathbf{v}_{in}. \tag{18.2.20}$$

Bu modeller ilklendirildiğinde, A Gauss girdileri olan rastgele bir matris olarak alınır, bu yüzden onlardan birini yapalım. Somut olmak gerekirse, ortalama sıfır, değişinti (varyans) bir olan Gauss dağılımından gelen bir 5×5 matrisi ile başlıyoruz.

```
torch.manual_seed(42)

k = 5
A = torch.randn(k, k, dtype=torch.float64)
A
```

```
tensor([[ 0.2996,  0.2424,  0.2832, -0.2329,  0.6712],
        [ 0.7818, -1.7903, -1.7484,  0.1735, -0.1182],
        [-1.7446, -0.4695,  0.4573,  0.5177, -0.2771],
        [-0.6641,  0.6551,  0.2616, -1.5265, -0.3311],
        [-0.6378,  0.1072,  0.7096,  0.3009, -0.2869]], dtype=torch.float64)
```

Rastgele Verilerde Davranış

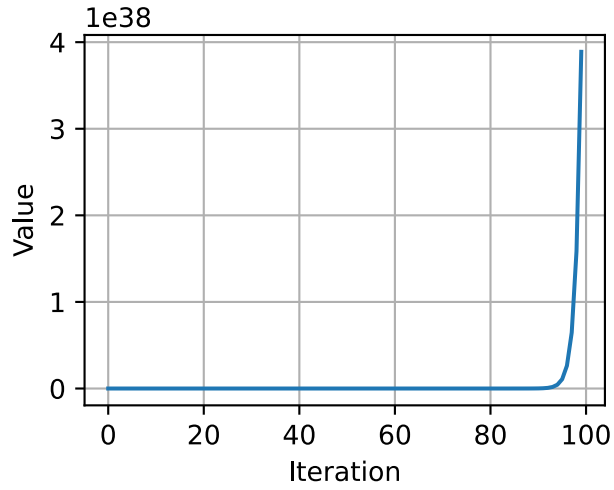
Oyuncak modelimizde basitlik sağlamak için, $\mathbf{v}_{in}$ ile beslediğimiz veri vektörünün rastgele beş boyutlu bir Gauss vektörü olduğunu varsayacağız. Ne olmasını istediğimizi düşünelim. Bağlam için, bir imge gibi girdi verilerini, imgenin bir kedi resmi olma olasılığı gibi bir tahmine dönüştürmeye çalıştığımız genel bir makine öğrenmesi problemini düşünelim. Tekrarlanan **A** uygulaması rastgele bir vektörü çok uzun olacak şekilde esnetirse, o zaman girdideki küçük değişiklikler çıktıdaki büyük değişikliklere yükseltilir — girdi imgesindeki küçük değişiklikler çok farklı tahminlere yol açar. Bu doğru görünmüyor!

Diğer taraftan, **A** rasgele vektörleri daha kısa olacak şekilde küçültürse, o zaman birçok katmandan geçtikten sonra, vektör esasen hiçbir şeye (sıfıra yakın) küçülür ve çıktı girdiye bağlı olmaz. Bu da açıkça doğru değil!

Çıktımızın girdimize bağlı olarak değiştiğinden emin olmak için büyüme ve bozulma arasındaki dar çizgide yürümemiz gerekir, ancak çok fazla değil!

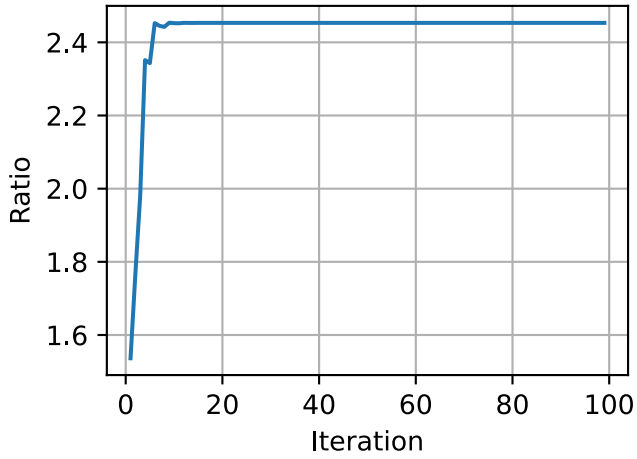
A matrisimizi rastgele bir girdi vektörüyle tekrar tekrar çarptığımızda ne olacağını görelim ve normunu takip edelim.

```
# `A`'yı tekrar tekrar uyguladıktan sonra normların dizisini hesapla
v_in = torch.randn(k, 1, dtype=torch.float64)

norm_list = [torch.norm(v_in).item()]
for i in range(1, 100):
    v_in = A @ v_in
    norm_list.append(torch.norm(v_in).item())

d2l.plot(torch.arange(0, 100), norm_list, 'Iteration', 'Value')
```



Norm kontrolsüz bir şekilde büyüyor! Nitekim bölüm listesini alırsak, bir desen göreceğiz.

```
# Normların ölçeklendirme çarpanını hesapla
norm_ratio_list = []
for i in range(1, 100):
    norm_ratio_list.append(norm_list[i]/norm_list[i - 1])

d2l.plot(torch.arange(1, 100), norm_ratio_list, 'Iteration', 'Ratio')
```



Yukarıdaki hesaplamanın son kısmına bakarsak, rastgele vektörün `1.974459321485[...]` çarpanı ile esnediğini görürüz, burada son kısım biraz kayar, ancak esneme çarpanı sabittir.

Özvektörlerle İlişkilendirme

Özvektörlerin ve özdeğerlerin, bir şeyin gerildiği (esnetildiği) miktara karşılık geldiğini gördük, ancak bu, belirli vektörler ve belirli gerilmeler içindi. **A** için ne olduklarına bir göz atalım. Burada bir uyarı: Hepsini görmek için karmaşık sayılara gitmemiz gerekeceği ortaya çıkıyor. Bunları esnemeler ve dönüşler olarak düşünebilirsiniz. Karmaşık sayının normunu (gerçek ve sanal kısımların karelerinin toplamının karekökü) alarak, bu esneme çarpanını ölçebiliriz. Bunları da sıralayabiliriz.

```
# Özdeğerleri hesapla
eigs = torch.eig(A)[0][:,0].tolist()
norm_eigs = [torch.abs(torch.tensor(x)) for x in eigs]
norm_eigs.sort()
print(f'norms of eigenvalues: {norm_eigs}')
```

```
norms of eigenvalues: [tensor(0.3490), tensor(0.5691), tensor(0.5691), tensor(1.1828),␣
↪tensor(2.4532)]
```

Bir Gözlem

Burada biraz beklenmedik bir şey görüyoruz: Daha önce rasgele bir vektöre uzun vadeli gerilmesi için **A** matrisimizi uygularken tanımladığımız sayı *tam olarak* (on üç ondalık basamağa kadar doğru!) **A**'nın en büyük özdeğeridir. Bu açıkça bir tesadüf değil!

Ama şimdi geometrik olarak ne olduğunu düşünürsek, bu mantıklı gelmeye başlar. Rastgele bir vektör düşünün. Bu rasgele vektör her yönü biraz işaret ediyor, bu nedenle özellikle en büyük özdeğerle ilişkili **A** özvektörüyle çok az olsa bile bir miktar aynı yönü gösteriyor. Bu o kadar önemlidir ki *ana (esas) özdeğer* ve *ana (esas) özvektör* olarak adlandırılır. **A**'yı uyguladıktan sonra, rastgele vektörümüz her olası özvektörle ilişkili olduğu gibi mümkün olan her yönde gerilir, ancak en çok bu özvektörle ilişkili yönde esnetilir. Bunun anlamı, A'da uygulandıktan sonra, rasgele vektörümüzün daha uzun olması ve ana özvektör ile hizalanmaya daha yakın bir yönü göstermesidir. Matrisi birçok kez uyguladıktan sonra, ana özvektörle hizalama gittikçe daha yakın hale gelir, ta ki tüm pratik amaçlar için rastgele vektörümüz temel özvektöre dönüştürülene kadar! Aslında bu algoritma, bir matrisin en büyük özdeğerini ve özvektörünü bulmak için *kuvvet yinelemesi* olarak bilinen şeyin temelidir. Ayrıntılar için, örneğin, bkz. (Van Loan and Golub, 1983).

Normalleştirmeyi (Düzgelemeyi) Düzeltme

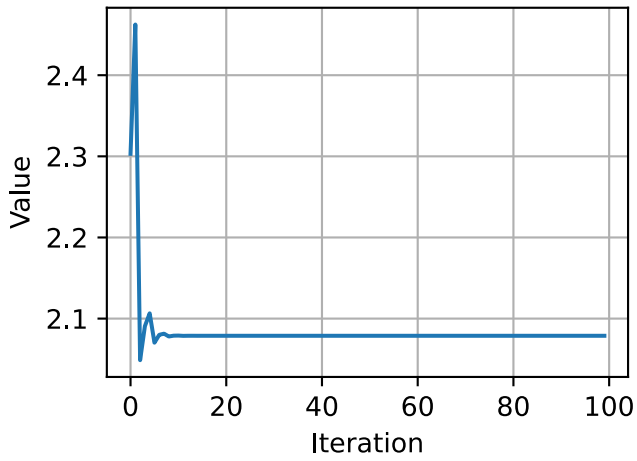
Şimdi, yukarıdaki tartışmalardan, rastgele bir vektörün esnetilmesini veya ezilmesini istemediğimiz sonucuna vardık, rastgele vektörlerin tüm süreç boyunca yaklaşık aynı boyutta kalmasını istiyoruz. Bunu yapmak için, şimdi matrisimizi bu ana özdeğere göre yeniden ölçeklendiriyoruz, böylece en büyük özdeğer şimdi eskinin yerine sadece bir olur. Bakalım bu durumda ne olacak.

```
# `A` matrisini yeniden ölçeklendir
A /= norm_eigs[-1]

# Aynı deneyi tekrar yapın
v_in = torch.randn(k, 1, dtype=torch.float64)
```

(continues on next page)

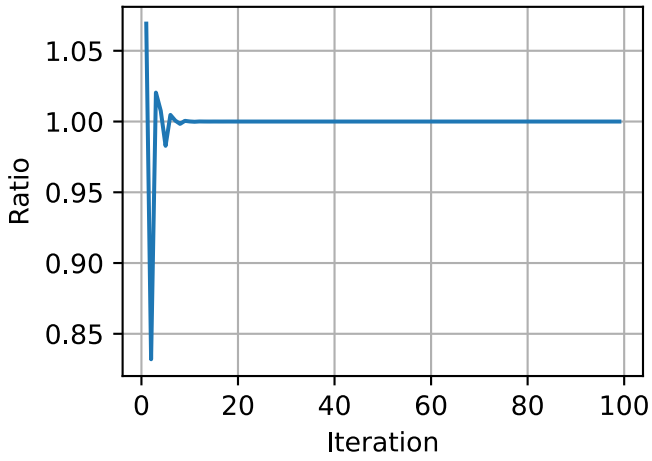
(continued from previous page)

```
norm_list = [torch.norm(v_in).item()]
for i in range(1, 100):
    v_in = A @ v_in
    norm_list.append(torch.norm(v_in).item())

d2l.plot(torch.arange(0, 100), norm_list, 'Iteration', 'Value')
```



Aynı zamanda ardışık normlar arasındaki oranı daha önce olduğu gibi çizebiliriz ve gerçekten dengelendiğini görebiliriz.

```
# Oranı da çiz
norm_ratio_list = []
for i in range(1, 100):
    norm_ratio_list.append(norm_list[i]/norm_list[i-1])

d2l.plot(torch.arange(1, 100), norm_ratio_list, 'Iteration', 'Ratio')
```



18.2.7 Sonuçlar

Şimdi tam olarak ne umduysak görüyoruz! Matrisleri ana özdeğere göre normalleştirdikten sonra, rastgele verilerin eskisi gibi patlamadığını görüyoruz, bunun yerine nihai belirli bir değerde dengelenirler. Bunları ilk ana özdeğerlerden yapabilmek güzel olurdu ve matematiğine derinlemesine bakarsak, bağımsız, ortalaması sıfır varyansı bir olan Gauss dağılımlı girdileri olan büyük bir rastgele matrisin en büyük özdeğerinin, ortalamada yaklaşık $\sqrt{n}$ veya bizim durumumuzda $\sqrt{5} \approx 2.2$ civarında olduğunu görebiliriz; bu *dairesel yasa* olarak bilinen büyüleyici bir gerçekten kaynaklanır (Ginibre, 1965). Rastgele matrislerin özdeğerlerinin (ve tekil değerler olarak adlandırılan ilgili bir konunun) arasındaki ilişkinin, şu adreste (Pennington *et al.*, 2017) ve sonraki çalışmalarda tartışıldığı gibi sinir ağlarının uygun şekilde ilklendirilmesiyle derin bağlantıları olduğu gösterilmiştir.

18.2.8 Özet

- Özvektörler, yön değiştirmeden bir matris tarafından esnetilen vektörlerdir.
- Özdeğerler, özvektörlerin matris uygulamasıyla gerildikleri miktardır.
- Bir matrisin özayrışımı, birçok işlemin özdeğerler üzerindeki işlemlere indirgenmesine izin verebilir.
- Gershgorin Çember Teoremi, bir matrisin özdeğerleri için yaklaşık değerler sağlayabilir.
- Yinelenen matris kuvvetlerinin davranışı, öncelikle en büyük özdeğerin boyutuna bağlıdır. Bu anlayış, sinir ağı ilkleme teorisinde birçok uygulamaya sahiptir.

18.2.9 Alıştırmalar

1. Aşağıdaki matrisin özdeğerleri ve özvektörleri nedir?

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \tag{18.2.21}$$

2. Aşağıdaki matrisin özdeğerleri ve özvektörleri nelerdir ve bu örnekte bir öncekine kıyasla garip olan nedir?

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix} \tag{18.2.22}$$

3. Özdeğerleri hesaplamadan, aşağıdaki matrisin en küçük özdeğerinin 0.5'den az olması mümkün müdür? *Not*: Bu problemi kafanızda yapılabilirsiniz.

$$\mathbf{A} = \begin{bmatrix} 3.0 & 0.1 & 0.3 & 1.0 \\ 0.1 & 1.0 & 0.1 & 0.2 \\ 0.3 & 0.1 & 5.0 & 0.0 \\ 1.0 & 0.2 & 0.0 & 1.8 \end{bmatrix}. \tag{18.2.23}$$

Tartışmalar[218]

[218] https://discuss.d2l.ai/t/1086

18.3 Tek Değişkenli Hesap

Section 2.4 içinde, türevsel (diferansiyel) hesabın (kalkülüsün) temel elemanlarını gördük. Bu bölüm, analizin temellerine ve bunu makine öğrenmesi bağlamında nasıl anlayıp uygulayabileceğimize daha derin bir dalış yapıyor.

18.3.1 Türevsel Hesap

Türevsel hesap, temelde fonksiyonların küçük değişiklikler altında nasıl davrandığının incelenmesidir. Bunun neden derin öğrenmenin özü olduğunu görmek için bir örnek ele alalım.

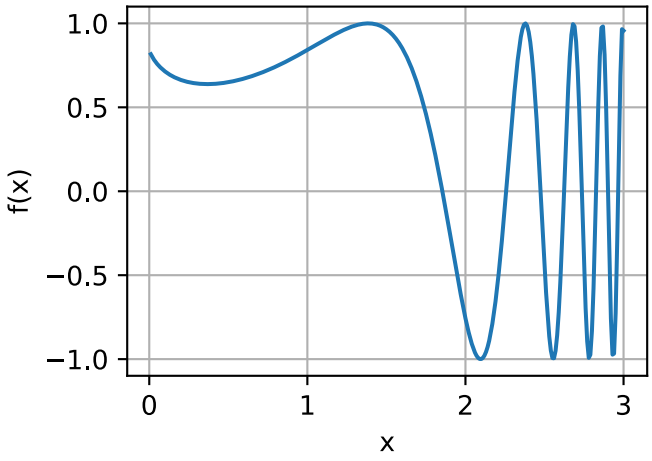
Kolaylık olması açısından ağırlıkların tek bir $\mathbf{w} = (w_1, \ldots, w_n)$ vektörüne bitiştirildiği derin bir sinir ağımız olduğunu varsayalım. Bir eğitim veri kümesi verildiğinde, bu veri kümesindeki sinir ağımızın $\mathcal{L}(\mathbf{w})$ olarak yazacağımız kaybını dikkate alıyoruz.

Bu işlev olağanüstü derecede karmaşıktır ve belirli mimarinin tüm olası modellerinin performansını bu veri kümesinde şifreler, dolayısıyla hangi $\mathbf{w}$ ağırlık kümesinin kaybı en aza indireceğini söylemek neredeyse imkansızdır. Bu nedenle, pratikte genellikle ağırlıklarımızı *rastgele* ilkleyerek başlarız ve ardından yinelemeli olarak kaybı olabildiğince hızlı düşüren yönde küçük adımlar atarız.

O zaman soru, yüzeyde daha kolay olmayan bir şeye dönüşür: Ağırlıkları olabildiğince çabuk düşüren yönü nasıl buluruz? Bunu derinlemesine incelemek için önce tek bir ağırlığa sahip durumu inceleyelim: Tek bir x gerçel değeri için $L(\mathbf{w}) = L(x)$.

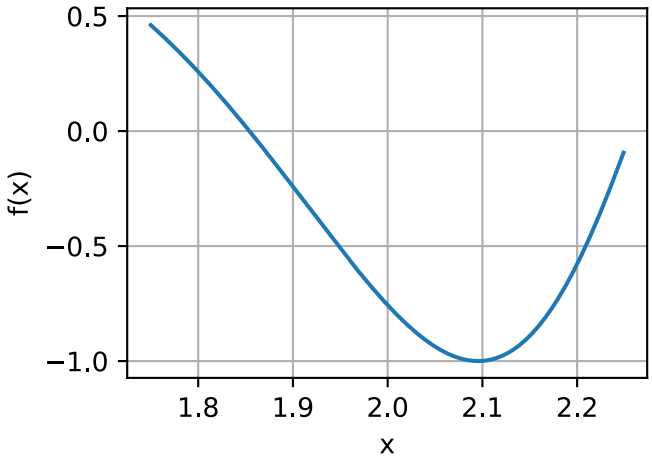
Şimdi x'i alalım ve onu küçük bir miktar $x + \epsilon$ olarak değiştirdiğimizde ne olacağını anlamaya çalışalım. Somut olmak istiyorsanız, $\epsilon = 0.0000001$ gibi bir sayı düşünün. Neler olduğunu görselleştirmemize yardımcı olmak için, $[0, 3]$ üzerinden $f(x) = \sin(x^x)$ gibi bir örnek fonksiyonun grafiğini çizelim.

```
%matplotlib inline
import torch
from IPython import display
from d2l import torch as d2l

torch.pi = torch.acos(torch.zeros(1)).item() * 2  # Pi'yi tanımla

# Normal bir aralıktaki bir fonksiyon çiz
x_big = torch.arange(0.01, 3.01, 0.01)
ys = torch.sin(x_big**x_big)
d2l.plot(x_big, ys, 'x', 'f(x)')
```



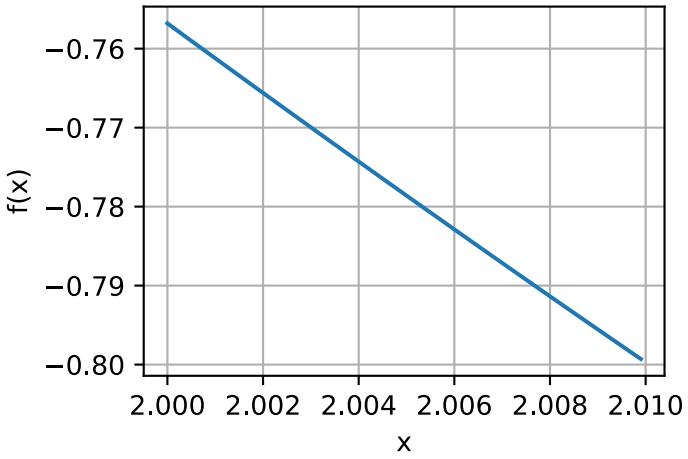
Bu büyük ölçekte, işlevin davranışı basit değildir. Ancak, aralığımızı $[1.75, 2.25]$ gibi daha küçük bir değere düşürürsek, grafiğin çok daha basit hale geldiğini görürüz.

```
# Aynı işlevi küçük bir aralıkta çizin
x_med = torch.arange(1.75, 2.25, 0.001)
ys = torch.sin(x_med**x_med)
d2l.plot(x_med, ys, 'x', 'f(x)')
```



Bunu aşırıya götürürsek, küçücük bir bölüme yakınlaştırırsak, davranış çok daha basit hale gelir: Bu sadece düz bir çizgidir.

```
# Aynı işlevi küçük bir aralıkta çizin
x_small = torch.arange(2.0, 2.01, 0.0001)
ys = torch.sin(x_small**x_small)
d2l.plot(x_small, ys, 'x', 'f(x)')
```



Bu, tek değişkenli analizin temel gözlemidir: Tanıdık fonksiyonların davranışı, yeterince küçük bir aralıktaki bir doğru ile modellenebilir. Bu, çoğu fonksiyon için, fonksiyonun x değerini biraz kaydırdığımızda, $f(x)$ çıktısının da biraz kayacağını beklemenin mantıklı olduğu anlamına gelir. Yanıtlamamız gereken tek soru, "Girdideki değişime kıyasla çıktıdaki değişim ne kadar büyük? Yarısı kadar büyük mü? İki kat büyük mü?"

Bu nedenle, bir fonksiyonun çıktısındaki değişim oranını, fonksiyonun girdisindeki küçük bir değişiklik için dikkate alabiliriz. Bunu kurallı olarak yazabiliriz:

$$\frac{L(x+\epsilon) - L(x)}{(x+\epsilon) - x} = \frac{L(x+\epsilon) - L(x)}{\epsilon}. \tag{18.3.1}$$

Bu, kod içinde oynamaya başlamak için zaten yeterli. Örneğin, $L(x) = x^2 + 1701(x-4)^3$ olduğunu bildiğimizi varsayalım, o zaman bu değerin $x = 4$ noktasında ne kadar büyük olduğunu aşağıdaki gibi görebiliriz.

```
# Fonksiyonumuzu tanımlayalım
def L(x):
    return x**2 + 1701*(x-4)**3

# Birkaç epsilon için farkı epsilon'a bölerek yazdırın
for epsilon in [0.1, 0.001, 0.0001, 0.00001]:
    print(f'epsilon = {epsilon:.5f} -> {(L(4+epsilon) - L(4)) / epsilon:.5f}')
```

```
epsilon = 0.10000 -> 25.11000
epsilon = 0.00100 -> 8.00270
epsilon = 0.00010 -> 8.00012
epsilon = 0.00001 -> 8.00001
```

Şimdi, eğer dikkatli olursak, bu sayının çıktısının şüpheli bir şekilde 8'e yakın olduğunu fark edeceğiz. Aslında, ϵ'u düşürürsek, değerin giderek 8'e yaklaştığını göreceğiz. Böylece, doğru bir şekilde, aradığımız değerin (girdideki bir değişikliğin çıktıyı değiştirdiği derece) $x = 4$ noktasında 8 olması gerektiği sonucuna varabiliriz. Bir matematikçinin bu gerçeği kodlama şekli aşağıdadır:

$$\lim_{\epsilon \to 0} \frac{L(4+\epsilon) - L(4)}{\epsilon} = 8. \tag{18.3.2}$$

Biraz tarihsel bir bilgi olarak: Sinir ağı araştırmalarının ilk birkaç on yılında, bilim adamları bu algoritmayı (*sonlu farklar yöntemi*) küçük bir oynama altında bir kayıp fonksiyonunun nasıl

değiştiğini değerlendirmek için kullandılar; sadece ağırlıkları değiştirin ve kayıp nasıl değişir izleyin. Bu, sayısal olarak verimsizdir ve bir değişkendeki tek bir değişikliğin kaybı nasıl etkilediğini görmek için kayıp fonksiyonunun iki değerlendirmesini gerektirir. Bunu birkaç bin parametreyle bile yapmaya çalışsaydık, tüm veri kümesi üzerinde ağın birkaç bin değerlendirme yapmasını gerektirecekti! (Rumelhart *et al.*, 1988) çalışmasında sunulan *geri yayma algoritması* ile ağırlıkların herhangi bir değişikliğinin, veri kümesi üzerindeki ağın tek bir tahminiyle aynı hesaplama süresinde kaybı nasıl değiştireceğini hesaplamak için bir yol sağladığı 1986 yılına kadar çözülemedi.

Örneğimizdeki bu 8 değeri, x'in farklı değerleri için farklıdır, bu nedenle, x'in bir işlevi olarak tanımlamak mantıklıdır. Daha resmi olarak, bu değere bağlı değişim oranına *türev* denir ve şöyle yazılır:

$$\frac{df}{dx}(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}. \tag{18.3.3}$$

Türev için farklı metinler farklı gösterimler kullanacaktır. Örneğin, aşağıdaki tüm gösterimler aynı şeyi gösterir:

$$\frac{df}{dx} = \frac{d}{dx}f = f' = \nabla_x f = D_x f = f_x. \tag{18.3.4}$$

Çoğu yazar tek bir gösterim seçer ve ona sadık kalır, ancak bu bile garanti edilmez. Bunların hepsine aşina olmak en iyisidir. Karmaşık bir ifadenin türevini almak istemiyorsak, bu metin boyunca $\frac{df}{dx}$ gösterimini kullanacağız, bu durumda aşağıdaki gibi ifadeler yazmak için $\frac{d}{dx}f$ kullanacağız

$$\frac{d}{dx}\left[x^4 + \cos\left(\frac{x^2+1}{2x-1}\right)\right]. \tag{18.3.5}$$

Çoğu zaman, x değerinde küçük bir değişiklik yaptığımızda bir fonksiyonun nasıl değiştiğini görmek için türev tanımını (18.3.3) yeniden açmak sezgisel olarak kullanışlıdır:

$$\begin{aligned} \frac{df}{dx}(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon} &\implies \frac{df}{dx}(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon} \\ &\implies \epsilon \frac{df}{dx}(x) \approx f(x+\epsilon) - f(x) \\ &\implies f(x+\epsilon) \approx f(x) + \epsilon \frac{df}{dx}(x). \end{aligned} \tag{18.3.6}$$

Son denklem açıkça belirtilmeye değer. Bize, herhangi bir işlevi alırsanız ve girdiyi küçük bir miktar değiştirirseniz, çıktının türevin ölçeklendirdiği küçük miktarda değişeceğini söyler.

Bu şekilde türevi, girdideki bir değişiklikten çıktıda ne kadar büyük değişiklik elde ettiğimizi söyleyen ölçeklendirme çarpanı olarak anlayabiliriz.

18.3.2 Kalkülüs Kuralları

Şimdi, açık bir fonksiyonun türevinin nasıl hesaplanacağını anlama görevine dönüyoruz. Kalkülüsün tam bir biçimsel incelemesi, her şeyi ilk ilkelerden türetecektir. Burada bu cazibeye kapılmayacağız, bunun yerine karşılaşılan genel kuralların anlaşılmasını sağlayacağız.

Yaygın Türevler

Section 2.4 içinde görüldüğü gibi, türevleri hesaplarken hesaplamayı birkaç temel işleve indirgemek için çoğu zaman bir dizi kural kullanılabilir. Referans kolaylığı için burada tekrar ediyoruz.

- **Sabitlerin türevi.** $\frac{d}{dx}c = 0$.
- **Doğrusal fonksiyonların türevi.** $\frac{d}{dx}(ax) = a$.
- **Kuvvet kuralı.** $\frac{d}{dx}x^n = nx^{n-1}$.
- **Üstellerin türevi.** $\frac{d}{dx}e^x = e^x$.
- **Logaritmanın türevi.** $\frac{d}{dx}\log(x) = \frac{1}{x}$.

Türev Kuralları

Her türevin ayrı ayrı hesaplanması ve bir tabloda depolanması gerekirse, türevsel hesap neredeyse imkansız olurdu. Yukarıdaki türevleri genelleştirebilmemiz ve $f(x) = \log\left(1 + (x-1)^{10}\right)$ türevini bulmak gibi daha karmaşık türevleri hesaplayabilmemiz matematiğin bir armağanıdır. Section 2.4 içinde bahsedildiği gibi, bunu yapmanın anahtarı, fonksiyonları aldığımızda ve bunları çeşitli şekillerde birleştirdiğimizde, özellikle toplamlar, çarpımlar ve bileşimler, olanları kodlamaktır.

- **Toplam kuralı.** $\frac{d}{dx}\left(g(x) + h(x)\right) = \frac{dg}{dx}(x) + \frac{dh}{dx}(x)$
- **Çarpım kuralı.** $\frac{d}{dx}\left(g(x) \cdot h(x)\right) = g(x)\frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x)$.
- **Zincir kuralı.** $\frac{d}{dx}g(h(x)) = \frac{dg}{dh}(h(x)) \cdot \frac{dh}{dx}(x)$.

Bu kuralları anlamak için (18.3.6) ifadesini nasıl kullanabileceğimize bir bakalım. Toplam kuralı için aşağıdaki akıl yürütme zincirini düşünün:

$$\begin{aligned} f(x+\epsilon) &= g(x+\epsilon) + h(x+\epsilon) \\ &\approx g(x) + \epsilon\frac{dg}{dx}(x) + h(x) + \epsilon\frac{dh}{dx}(x) \\ &= g(x) + h(x) + \epsilon\left(\frac{dg}{dx}(x) + \frac{dh}{dx}(x)\right) \\ &= f(x) + \epsilon\left(\frac{dg}{dx}(x) + \frac{dh}{dx}(x)\right). \end{aligned} \tag{18.3.7}$$

Bu sonucu $f(x+\epsilon) \approx f(x) + \epsilon\frac{df}{dx}(x)$ gerçeğiyle karşılaştırdığımızda istenildiği gibi $\frac{df}{dx}(x) = \frac{dg}{dx}(x) + \frac{dh}{dx}(x)$ olduğunu görürüz. Buradaki sezgi şudur: x girdisini değiştirdiğimizde, g ve h çıktının değişmesine $\frac{dg}{dx}(x)$ ve $\frac{dh}{dx}(x)$ ile birlikte katkıda bulunur.

Çarpım daha inceliklidir ve bu ifadelerle nasıl çalışılacağı konusunda yeni bir gözlem gerektirecektir. Önceden olduğu gibi (18.3.6) ifadesini kullanarak başlayacağız :

$$\begin{aligned} f(x+\epsilon) &= g(x+\epsilon) \cdot h(x+\epsilon) \\ &\approx \left(g(x) + \epsilon\frac{dg}{dx}(x)\right) \cdot \left(h(x) + \epsilon\frac{dh}{dx}(x)\right) \\ &= g(x) \cdot h(x) + \epsilon\left(g(x)\frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x)\right) + \epsilon^2\frac{dg}{dx}(x)\frac{dh}{dx}(x) \\ &= f(x) + \epsilon\left(g(x)\frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x)\right) + \epsilon^2\frac{dg}{dx}(x)\frac{dh}{dx}(x). \end{aligned} \tag{18.3.8}$$

Bu, yukarıda yapılan hesaplamaya benzer ve aslında görürüz ki cevabımız ($\frac{df}{dx}(x) = g(x)\frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x)$) ϵ'un yanında duruyor, ancak ϵ^2 terimininin boyutu ile ilgili bir durum var. ϵ^2'nin gücü ϵ^1'in gücünden daha yüksek olduğu için buna *yüksek dereceli bir terim* diyeceğiz. Daha sonraki bir bölümde bazen bunların kaydını tutmak isteyeceğimizi göreceğiz, ancak şimdilik $\epsilon = 0.0000001$ ise $\epsilon^2 = 0.0000000000001$'nın çok daha küçük olduğunu gözlemleyeceğiz. $\epsilon \to 0$ gider iken, daha yüksek dereceli terimleri güvenle göz ardı edebiliriz. Bu ek bölümünde genel bir kural olarak, iki terimin daha yüksek mertebeden terimlere kadar eşit olduğunu belirtmek için "$\approx$" kullanacağız. Ancak, daha resmi olmak istiyorsak, fark oranını inceleyebiliriz.

$$\frac{f(x+\epsilon) - f(x)}{\epsilon} = g(x)\frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x) + \epsilon\frac{dg}{dx}(x)\frac{dh}{dx}(x), \tag{18.3.9}$$

$\epsilon \to 0$ giderken, sağdaki terimin de sıfıra gittiğini görürüz.

Son olarak, zincir kuralı ile, (18.3.6) ifadesini kullanmadan önceki gibi tekrar ilerleyebiliriz ve görürüz ki

$$\begin{aligned} f(x+\epsilon) &= g(h(x+\epsilon)) \\ &\approx g\left(h(x) + \epsilon\frac{dh}{dx}(x)\right) \\ &\approx g(h(x)) + \epsilon\frac{dh}{dx}(x)\frac{dg}{dh}(h(x)) \\ &= f(x) + \epsilon\frac{dg}{dh}(h(x))\frac{dh}{dx}(x), \end{aligned} \tag{18.3.10}$$

Burada ikinci satırda g fonksiyonunun girdisinin ($h(x)$) minik bir miktar, $\epsilon\frac{dh}{dx}(x)$ kadar, kaydırıldığını görüyoruz.

Bu kurallar, esasen istenen herhangi bir ifadeyi hesaplamak için bize bir dizi esnek araç sağlar. Örneğin,

$$\begin{aligned} \frac{d}{dx}\left[\log\left(1+(x-1)^{10}\right)\right] &= \left(1+(x-1)^{10}\right)^{-1}\frac{d}{dx}\left[1+(x-1)^{10}\right] \\ &= \left(1+(x-1)^{10}\right)^{-1}\left(\frac{d}{dx}[1] + \frac{d}{dx}[(x-1)^{10}]\right) \\ &= \left(1+(x-1)^{10}\right)^{-1}\left(0 + 10(x-1)^9\frac{d}{dx}[x-1]\right) \\ &= 10\left(1+(x-1)^{10}\right)^{-1}(x-1)^9 \\ &= \frac{10(x-1)^9}{1+(x-1)^{10}}. \end{aligned} \tag{18.3.11}$$

Her satırda sırasıyla aşağıdaki kurallar kullanmıştır:

1. Zincir kuralı ve logaritmanın türevi.
2. Toplam kuralı.
3. Sabitlerin türevi, zincir kuralı ve kuvvet kuralı.
4. Toplam kuralı, doğrusal fonksiyonların türevi, sabitlerin türevi.

Bu örneği yaptıktan sonra iki şey netleşmiş olmalıdır:

1. Toplamları, çarpımları, sabitleri, üsleri, üstelleri ve logaritmaları kullanarak yazabileceğimiz herhangi bir fonksiyonun türevi bu kuralları takip ederek mekanik olarak hesaplanabilir.

2. Bir insanın bu kuralları takip etmesi yorucu ve hataya açık olabilir!

Neyse ki, bu iki gerçek birlikte ileriye doğru bir yol gösteriyor: Bu, mekanikleştirme için mükemmel bir aday! Aslında bu bölümde daha sonra tekrar ele alacağımız geri yayma tam olarak da budur.

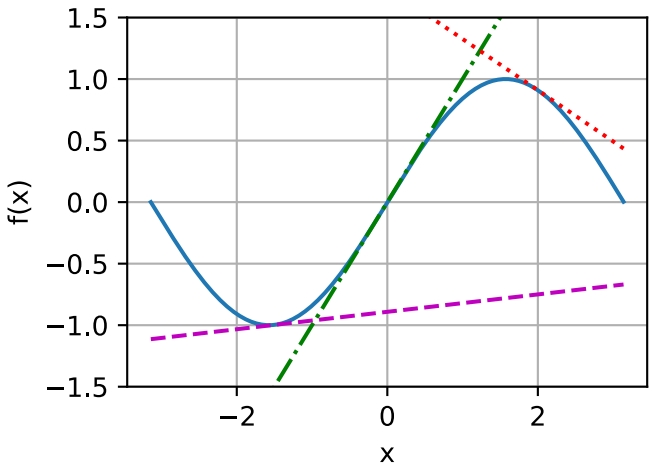
Doğrusal Yaklaşıklama

Türevlerle çalışırken, yukarıda kullanılan yaklaşıklamayı geometrik olarak yorumlamak genellikle yararlıdır. Özellikle, aşağıdaki denklemin

$$f(x+\epsilon) \approx f(x) + \epsilon \frac{df}{dx}(x), \tag{18.3.12}$$

f değerine $(x, f(x))$ noktasından geçen ve $\frac{df}{dx}(x)$ eğimine sahip bir çizgi ile yaklaştığına dikkat edin. Bu şekilde, türevin, aşağıda gösterildiği gibi f fonksiyonuna doğrusal bir yaklaşıklama verdiğini söylüyoruz:

```
# Sinüsü hesapla
xs = torch.arange(-torch.pi, torch.pi, 0.01)
plots = [torch.sin(xs)]

# Biraz doğrusal yaklaşım hesaplayın. d(sin(x)) / dx = cos(x) kullanın.
for x0 in [-1.5, 0.0, 2.0]:
    plots.append(torch.sin(torch.tensor(x0)) + (xs - x0) *
                 torch.cos(torch.tensor(x0)))

d2l.plot(xs, plots, 'x', 'f(x)', ylim=[-1.5, 1.5])
```



Şimdi yüzeyde garip görünebilecek bir şey yapalım. Bir f fonksiyonunu alın ve $\frac{df}{dx}$ türevini hesaplayın. Bu bize herhangi bir noktada f'nin değişim oranını verir.

Bununla birlikte, $\frac{df}{dx}$ türevi, bir işlev olarak görülebilir, bu nedenle hiçbir şey bizi $\frac{df}{dx}$ türevini, $\frac{d^2f}{dx^2} = \frac{df}{dx}\left(\frac{df}{dx}\right)$, hesaplamaktan alıkoyamaz. Buna f'nin ikinci türevi diyeceğiz. Bu fonksiyon, f'nin değişim oranının değişim oranıdır veya başka bir deyişle, değişim oranının nasıl değiştiğidir. $n.$ türev denen türevi elde etmek için herhangi bir sayıda art arda türev uygulayabiliriz. Gösterimi temiz tutmak için, $n.$ türevi şu şekilde göstereceğiz:

$$f^{(n)}(x) = \frac{d^n f}{dx^n} = \left(\frac{d}{dx}\right)^n f. \tag{18.3.13}$$

Bunun *neden* yararlı bir fikir olduğunu anlamaya çalışalım. Aşağıda, $f^{(2)}(x)$, $f^{(1)}(x)$ ve $f(x)$'i görselleştiriyoruz.

İlk olarak, ikinci türevin $f^{(2)}(x)$ pozitif bir sabit olduğu durumu düşünün. Bu, birinci türevin eğiminin pozitif olduğu anlamına gelir. Sonuç olarak, birinci türev, $f^{(1)}(x)$, negatif olarak başlayabilir, bir noktada sıfır olur ve sonra sonunda pozitif olur. Bu bize esas fonksiyonumuz f'nin eğimini anlatır; dolayısıyla f fonksiyonunun kendisi azalır, düzleşir, sonra artar. Başka bir deyişle, f işlevi yukarı doğru eğrilir ve Fig. 18.3.1 şeklinde gösterildiği gibi tek bir minimuma sahiptir.

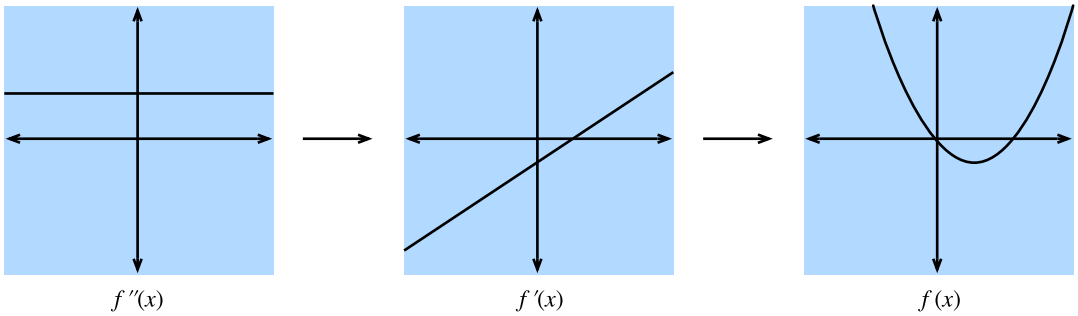


Fig. 18.3.1: İkinci türevin pozitif bir sabit olduğunu varsayarsak, artmakta olan ilk türev, fonksiyonun kendisinin bir minimuma sahip olduğu anlamına gelir.

İkincisi, eğer ikinci türev negatif bir sabitse, bu, birinci türevin azaldığı anlamına gelir. Bu, ilk türevin pozitif başlayabileceği, bir noktada sıfır olacağı ve sonra negatif olacağı anlamına gelir. Dolayısıyla, f işlevinin kendisi artar, düzleşir ve sonra azalır. Başka bir deyişle, f işlevi aşağı doğru eğilir ve Fig. 18.3.2 şeklinde gösterildiği gibi tek bir maksimuma sahiptir.

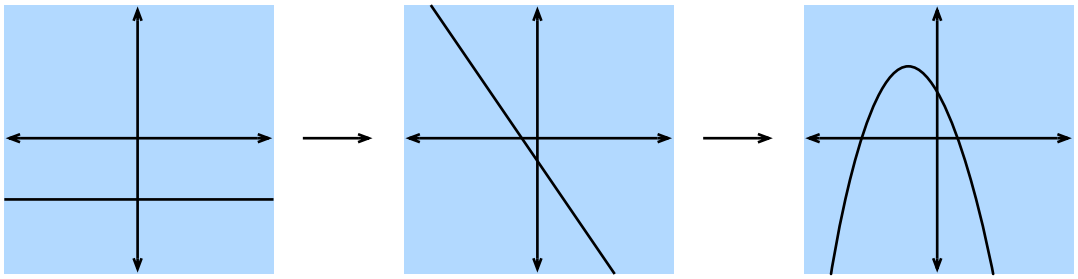


Fig. 18.3.2: İkinci türevin negatif bir sabit olduğunu varsayarsak, azalmakta olan ilk türev, fonksiyonun kendisinin bir maksimuma sahip olduğu anlamına gelir.

Üçüncüsü, eğer ikinci türev her zaman sıfırsa, o zaman ilk türev asla değişmeyecektir—sabittir! Bu, f'nin sabit bir oranda arttığı (veya azaldığı) anlamına gelir ve f'nin kendisi de Fig. 18.3.3 içinde gösterildiği gibi düz bir doğrudur.

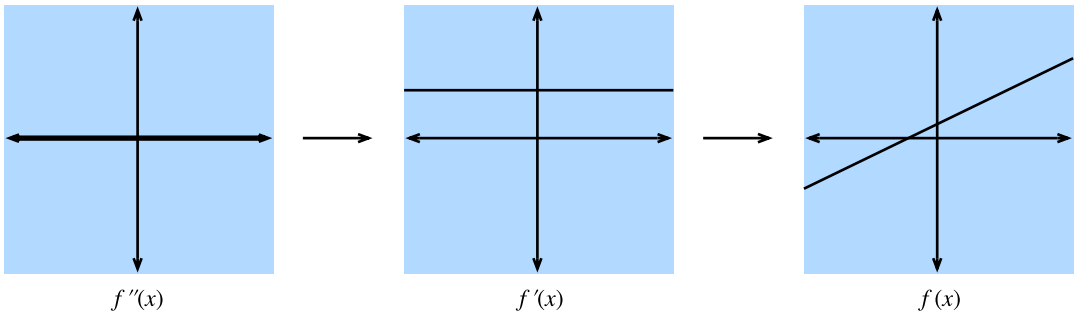


Fig. 18.3.3: İkinci türevin sıfır olduğunu varsayarsak, ilk türev sabittir, bu da fonksiyonun kendisinin düz bir doğru olduğu anlamına gelir.

Özetlemek gerekirse, ikinci türev, f fonksiyonunun eğrilerinin şeklinin açıklanması olarak yorumlanabilir. Pozitif bir ikinci türev yukarı doğru bir eğriye yol açarken, negatif bir ikinci türev f'nin aşağı doğru eğrildiği ve sıfır ikinci türev ise f'nin hiç eğrilmediği anlamına gelir.

Bunu bir adım daha ileri götürelim. $g(x) = ax^2 + bx + c$ işlevini düşünün. Birlikte şunu hesaplayabiliriz

$$\begin{aligned}\frac{dg}{dx}(x) &= 2ax + b \\ \frac{d^2g}{dx^2}(x) &= 2a.\end{aligned} \tag{18.3.14}$$

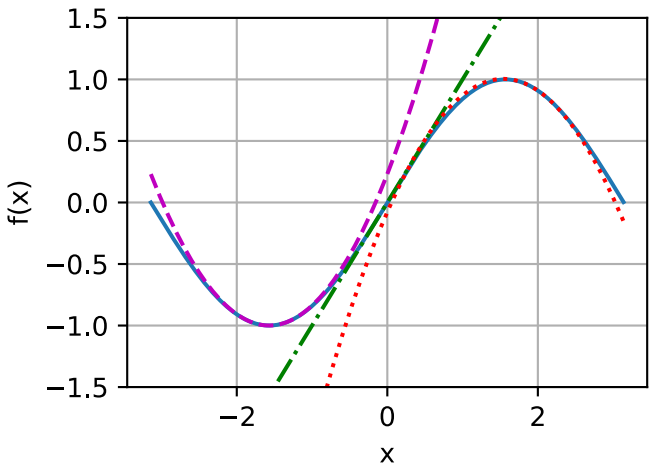
Aklımızda belirli bir orijinal fonksiyon $f(x)$ varsa, ilk iki türevi hesaplayabilir ve a, b ve c değerlerini bu hesaplamayla eşleşecek şekilde bulabiliriz. İlk türevin düz bir doğru ile en iyi yaklaşıklamayı verdiğini gördüğümüz önceki bölüme benzer şekilde, bu yapı bir ikinci dereceden en iyi yaklaşımı sağlar. Bunu $f(x) = \sin(x)$ için görselleştirelim.

```
# Sinüsü hesapla.
xs = torch.arange(-torch.pi, torch.pi, 0.01)
plots = [torch.sin(xs)]
```

(continues on next page)

(continued from previous page)

```
# Biraz ikinci dereceden yaklaşım hesaplayın. d(sin(x)) / dx = cos(x) kullanın.
for x0 in [-1.5, 0.0, 2.0]:
    plots.append(torch.sin(torch.tensor(x0)) + (xs - x0) *
                 torch.cos(torch.tensor(x0)) - (xs - x0)**2 *
                 torch.sin(torch.tensor(x0)) / 2)

d2l.plot(xs, plots, 'x', 'f(x)', ylim=[-1.5, 1.5])
```



Bu fikri bir sonraki bölümde bir *Taylor serisi* fikrine genişleteceğiz.

Taylor Serisi

Taylor serisi, bir x_0 noktası için ilk n türevlerinin değerleri, $\{f(x_0), f^{(1)}(x_0), f^{(2)}(x_0), \ldots, f^{(n)}(x_0)\}$ verilirse, $f(x)$ fonksiyonuna yaklaşıklamak için bir yöntem sağlar. Buradaki fikir, x_0'da verilen tüm türevlerle eşleşen bir n dereceli polinom bulmak olacaktır.

Önceki bölümde $n = 2$ durumunu gördük ve biraz cebir bunu gösterecektir:

$$f(x) \approx \frac{1}{2}\frac{d^2 f}{dx^2}(x_0)(x - x_0)^2 + \frac{df}{dx}(x_0)(x - x_0) + f(x_0). \tag{18.3.15}$$

Yukarıda gördüğümüz gibi, 2 paydası, x^2'nin türevini iki defa aldığımızda elde ettiğimiz 2'yi iptal etmek için oradadır, diğer terimlerin hepsi sıfırdır. Aynı mantık, birinci türev ve değerin kendisi için de geçerlidir.

Mantığı $n = 3$ değerine uygularsak, şu sonuca varacağız:

$$f(x) \approx \frac{\frac{d^3 f}{dx^3}(x_0)}{6}(x - x_0)^3 + \frac{\frac{d^2 f}{dx^2}(x_0)}{2}(x - x_0)^2 + \frac{df}{dx}(x_0)(x - x_0) + f(x_0). \tag{18.3.16}$$

burada $6 = 3 \times 2 = 3!$, x^3'ün üçüncü türevini alırsak önümüze gelen sabitten gelir.

Dahası, bir n dereceli polinom elde edebiliriz.

$$P_n(x) = \sum_{i=0}^{n} \frac{f^{(i)}(x_0)}{i!}(x - x_0)^i. \tag{18.3.17}$$

buradaki eşdeğer gösterim aşağıdadır.

$$f^{(n)}(x) = \frac{d^n f}{dx^n} = \left(\frac{d}{dx}\right)^n f. \tag{18.3.18}$$

Aslında, $P_n(x)$, $f(x)$ fonksiyonumuza yaklaşan en iyi n-dereceli polinom olarak görülebilir.

Yukarıdaki tahminlerin hatasına tam olarak dalmayacak olsak da, sonsuz limitten bahsetmeye değer. Bu durumda, $\cos(x)$ veya e^x gibi iyi huylu işlevler (gerçel analitik işlevler olarak bilinir) için, sonsuz sayıda terim yazabilir ve tam olarak aynı işlevi yaklaşık olarak tahmin edebiliriz

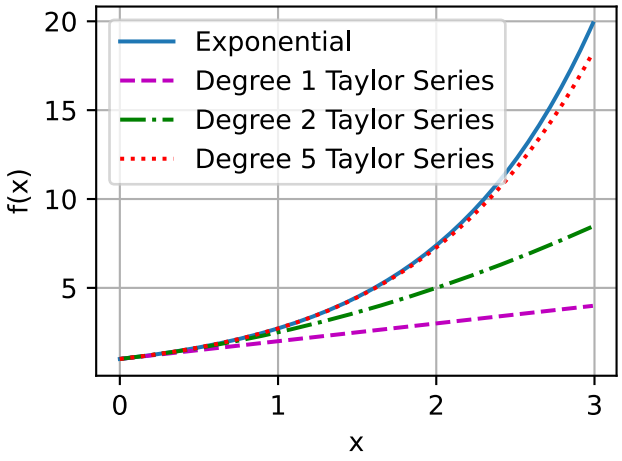
$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n. \tag{18.3.19}$$

Örnek olarak $f(x) = e^x$'i alalım. e^x kendisinin türevi olduğundan, $f^{(n)}(x) = e^x$ olduğunu biliyoruz. Bu nedenle, e^x, Taylor serisi $x_0 = 0$ alınarak yeniden yapılandırılabilir, yani,

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \cdots. \tag{18.3.20}$$

Bunun kodda nasıl çalıştığını görelim ve Taylor yaklaşımının derecesini artırmanın bizi istenen e^x fonksiyonuna nasıl yaklaştırdığını görelim.

```
# Üstel işlevi hesaplayın
xs = torch.arange(0, 3, 0.01)
ys = torch.exp(xs)

# Birkaç Taylor serisi yaklaşımını hesaplayın
P1 = 1 + xs
P2 = 1 + xs + xs**2 / 2
P5 = 1 + xs + xs**2 / 2 + xs**3 / 6 + xs**4 / 24 + xs**5 / 120

d2l.plot(xs, [ys, P1, P2, P5], 'x', 'f(x)', legend=[
    "Exponential", "Degree 1 Taylor Series", "Degree 2 Taylor Series",
    "Degree 5 Taylor Series"])
```



Taylor serisinin iki ana uygulaması vardır:

1. *Teorik uygulamalar*: Genellikle çok karmaşık bir fonksiyonu anlamaya çalıştığımızda, Taylor serisini kullanmak onu doğrudan çalışabileceğimiz bir polinom haline dönüştürebilmemizi sağlar.
2. *Sayısal (numerik) uygulamalar*: e^x veya $\cos(x)$ gibi bazı işlevlerin hesaplanması makineler için zordur. Değer tablolarını sabit bir hassasiyette depolayabilirler (ve bu genellikle yapılır), ancak yine de "$\cos(1)$'in 1000'inci basamağı nedir?" gibi açık sorular kalır. Taylor serileri, bu tür soruları cevaplamak için genellikle yardımcı olur.

18.3.3 Özet

- Türevler, girdiyi küçük bir miktar değiştirdiğimizde fonksiyonların nasıl değişeceğini ifade etmek için kullanılabilir.
- Temel türevler, karmaşık türevleri bulmak için türev kuralları kullanılarak birleştirilebilir.
- Türevler, ikinci veya daha yüksek dereceden türevleri elde etmek için yinelenebilir. Derecedeki her artış, işlevin davranışı hakkında daha ayrıntılı bilgi sağlar.
- Tek bir veri örneğinin türevlerindeki bilgileri kullanarak, Taylor serisinden elde edilen polinomlarla iyi huylu fonksiyonları yaklaşık elde edebiliriz.

18.3.4 Alıştırmalar

1. $x^3 - 4x + 1$'in türevi nedir?
2. $\log(\frac{1}{x})$'in türevi nedir?
3. Doğru veya yanlış: $f'(x) = 0$ ise, f işlevi x'te maksimum veya minimuma sahip midir?
4. $x \geq 0$ için $f(x) = x\log(x)$ minimumu nerededir (burada f'nin $f(0)$'da 0'ın limit değerini aldığını varsayıyoruz)?

Tartışmalar[219]

18.4 Çok Değişkenli Hesap

Artık tek değişkenli bir fonksiyonun türevleri hakkında oldukça güçlü bir anlayışa sahip olduğumuza göre, potansiyel olarak milyarlarca ağırlığa sahip bir kayıp (yitim) fonksiyonunu düşündüğümüz esas sorumuza dönelim.

[219] https://discuss.d2l.ai/t/1088

18.4.1 Yüksek Boyutlu Türev Alma

Section 18.3 bize, bu milyarlarca ağırlıktan diğer her birini sabit bırakarak sadece birini değiştirirsek ne olacağını bildiğimizi söyler! Bu, tek değişkenli bir fonksiyondan başka bir şey değildir, bu yüzden şöyle yazabiliriz

$$L(w_1 + \epsilon_1, w_2, \ldots, w_N) \approx L(w_1, w_2, \ldots, w_N) + \epsilon_1 \frac{d}{dw_1} L(w_1, w_2, \ldots, w_N). \tag{18.4.1}$$

Diğer değişkenleri sabit tutarken bir değişkendeki türeve *kısmi türev* diyeceğiz ve (18.4.1) denklemindeki türev için $\frac{\partial}{\partial w_1}$ gösterimini kullanacağız.

Şimdi bunu alalım ve w_2'yi biraz $w_2 + \epsilon_2$ olarak değiştirelim:

$$\begin{aligned}
L(w_1 + \epsilon_1, w_2 + \epsilon_2, \ldots, w_N) &\approx L(w_1, w_2 + \epsilon_2, \ldots, w_N) + \epsilon_1 \frac{\partial}{\partial w_1} L(w_1, w_2 + \epsilon_2, \ldots, w_N + \epsilon_N) \\
&\approx L(w_1, w_2, \ldots, w_N) \\
&\quad + \epsilon_2 \frac{\partial}{\partial w_2} L(w_1, w_2, \ldots, w_N) \\
&\quad + \epsilon_1 \frac{\partial}{\partial w_1} L(w_1, w_2, \ldots, w_N) \\
&\quad + \epsilon_1 \epsilon_2 \frac{\partial}{\partial w_2} \frac{\partial}{\partial w_1} L(w_1, w_2, \ldots, w_N) \\
&\approx L(w_1, w_2, \ldots, w_N) \\
&\quad + \epsilon_2 \frac{\partial}{\partial w_2} L(w_1, w_2, \ldots, w_N) \\
&\quad + \epsilon_1 \frac{\partial}{\partial w_1} L(w_1, w_2, \ldots, w_N).
\end{aligned} \tag{18.4.2}$$

Bir kez daha, $\epsilon_1 \epsilon_2$'nin daha yüksek bir terim olduğu fikrini kullandık ve önceki bölümde gördüğümüz ϵ^2 ile aynı şekilde (18.4.1) denkleminden atabildik. Bu şekilde devam ederek şunu yazabiliriz:

$$L(w_1 + \epsilon_1, w_2 + \epsilon_2, \ldots, w_N + \epsilon_N) \approx L(w_1, w_2, \ldots, w_N) + \sum_i \epsilon_i \frac{\partial}{\partial w_i} L(w_1, w_2, \ldots, w_N). \tag{18.4.3}$$

Bu bir karmaşa gibi görünebilir, ancak sağdaki toplamın tam olarak bir iç çarpıma benzediğini fark ederek bunu daha tanıdık hale getirebiliriz.

$$\boldsymbol{\epsilon} = [\epsilon_1, \ldots, \epsilon_N]^\top \text{ and } \nabla_{\mathbf{x}} L = \left[\frac{\partial L}{\partial x_1}, \ldots, \frac{\partial L}{\partial x_N}\right]^\top, \tag{18.4.4}$$

o zaman da:

$$L(\mathbf{w} + \boldsymbol{\epsilon}) \approx L(\mathbf{w}) + \boldsymbol{\epsilon} \cdot \nabla_{\mathbf{w}} L(\mathbf{w}). \tag{18.4.5}$$

$\nabla_{\mathbf{w}} L$'yı L'nin *gradyanı* olarak adlandıracağız.

Denklem (18.4.5) bir an üstünde düşünmeye değerdir. Tam olarak bir boyutta karşılaştığımız formata sahip, sadece her şeyi vektörlere ve nokta çarpımlarına dönüştürdük. Girdiye herhangi ufak bir dürtme verildiğinde L fonksiyonunun nasıl değişeceğini yaklaşık olarak söylememizi sağlar. Bir sonraki bölümde göreceğimiz gibi, bu bize gradyanda bulunan bilgileri kullanarak nasıl öğrenebileceğimizi geometrik olarak anlamamız için önemli bir araç sağlayacaktır.

Ama önce bu yaklaşıklamayı bir örnekle iş başında görelim. Şu fonksiyon ile çalıştığımızı varsayalım:

$$f(x, y) = \log(e^x + e^y) \text{ ve gradyanı } \nabla f(x, y) = \left[\frac{e^x}{e^x + e^y}, \frac{e^y}{e^x + e^y}\right]. \tag{18.4.6}$$

$(0, \log(2))$ gibi bir noktaya bakarsak görürüz ki

$$f(x, y) = \log(3) \text{ ve gradyanı } \nabla f(x, y) = \left[\frac{1}{3}, \frac{2}{3}\right]. \tag{18.4.7}$$

Bu nedenle, $(\epsilon_1, \log(2) + \epsilon_2)$ konumunda f'ye yaklaşmak istiyorsak, şu özel örneğe sahip olmamız gerektiğini görürüz (18.4.5):

$$f(\epsilon_1, \log(2) + \epsilon_2) \approx \log(3) + \frac{1}{3}\epsilon_1 + \frac{2}{3}\epsilon_2. \tag{18.4.8}$$

Yaklaşıklamanın ne kadar iyi olduğunu görmek için bunu kodda test edebiliriz.

```
%matplotlib inline
import numpy as np
import torch
from IPython import display
from mpl_toolkits import mplot3d
from d2l import torch as d2l


def f(x, y):
    return torch.log(torch.exp(x) + torch.exp(y))
def grad_f(x, y):
    return torch.tensor([torch.exp(x) / (torch.exp(x) + torch.exp(y)),
                         torch.exp(y) / (torch.exp(x) + torch.exp(y))])

epsilon = torch.tensor([0.01, -0.03])
grad_approx = f(torch.tensor([0.]), torch.log(
    torch.tensor([2.]))) + epsilon.dot(
    grad_f(torch.tensor([0.]), torch.log(torch.tensor(2.))))
true_value = f(torch.tensor([0.]) + epsilon[0], torch.log(
    torch.tensor([2.])) + epsilon[1])
f'approximation: {grad_approx}, true Value: {true_value}'
```

```
'approximation: tensor([1.0819]), true Value: tensor([1.0821])'
```

18.4.2 Gradyanların Geometrisi ve Gradyan (Eğim) İnişi

(18.4.5) denklemindeki ifadeyi tekrar düşünün:

$$L(\mathbf{w} + \boldsymbol{\epsilon}) \approx L(\mathbf{w}) + \boldsymbol{\epsilon} \cdot \nabla_{\mathbf{w}} L(\mathbf{w}). \tag{18.4.9}$$

Diyelim ki, bunu L kaybımızı en aza indirmeye yardımcı olmak için kullanmak istiyoruz. İlk önce Section 2.5 içinde açıklanan gradyan iniş algoritmasını geometrik olarak anlayalım. Yapacağımız şey şudur:

1. $\mathbf{w}$ başlangıç parametreleri için rastgele bir seçimle başlayın.

2. L'nin $\mathbf{w}$ seviyesinde en hızlı azalmasını sağlayan $\mathbf{v}$ yönünü bulun.
3. Bu yönde küçük bir adım atın: $\mathbf{w} \to \mathbf{w} + \epsilon\mathbf{v}$.
4. Tekrar edin.

Tam olarak nasıl yapılacağını bilmediğimiz tek şey, ikinci adımdaki $\mathbf{v}$ vektörünü hesaplamaktır. Böyle bir yöne *en dik iniş yönü* diyeceğiz. Section 18.1 konusundan nokta çarpımlarının geometrik anlamını kullanarak, şunu, (18.4.5), yeniden yazabileceğimizi görüyoruz:

$$L(\mathbf{w} + \mathbf{v}) \approx L(\mathbf{w}) + \mathbf{v} \cdot \nabla_{\mathbf{w}} L(\mathbf{w}) = L(\mathbf{w}) + \|\nabla_{\mathbf{w}} L(\mathbf{w})\| \cos(\theta). \tag{18.4.10}$$

Kolaylık olması açısından yönümüzü birim uzunluğa sahip olacak şekilde aldığımızı ve $\mathbf{v}$ ile $\nabla_{\mathbf{w}} L(\mathbf{w})$ arasındaki açı için θ'yı kullandığımızı unutmayın. L'nin olabildiğince hızlı azalan yönünü bulmak istiyorsak, bu ifadeyi olabildiğince negatif olarak ifade etmek isteriz. Seçtiğimiz yönün bu denkleme girmesinin tek yolu $\cos(\theta)$ sayesindedir ve bu yüzden bu kosinüsü olabildiğince negatif yapmak istiyoruz. Şimdi, kosinüs şeklini hatırlarsak, bunu mümkün olduğunca negatif yapmak için $\cos(\theta) = -1$ yapmamız veya eşdeğer olarak gradyan ile seçtiğimiz yön arasındaki açıyı π radyan olacak şekilde, diğer anlamda 180 derece yapmamız gerekecektir. Bunu başarmanın tek yolu, tam ters yöne gitmektir: $\nabla_{\mathbf{w}} L(\mathbf{w})$ yönünün tam tersini gösteren $\mathbf{v}$'yi seçin!

Bu bizi makine öğrenmesindeki en önemli matematiksel kavramlardan birine getiriyor: $-\nabla_{\mathbf{w}} L(\mathbf{w})$ yönündeki en dik yokuş noktaların yönü. Böylece resmi olmayan algoritmamız aşağıdaki gibi yeniden yazılabilir.

1. $\mathbf{w}$ ilk parametreleri için rastgele bir seçimle başlayın.
2. $\nabla_{\mathbf{w}} L(\mathbf{w})$'yi hesaplayın.
3. Ters yönde küçük bir adım atın: $\mathbf{w} \to \mathbf{w} - \epsilon\nabla_{\mathbf{w}} L(\mathbf{w})$.
4. Tekrar edin.

Bu temel algoritma, birçok araştırmacı tarafından birçok şekilde değiştirilmiş ve uyarlanmıştır, ancak temel kavram hepsinde aynı kalır. Kaybı olabildiğince hızlı azaltan yönü bulmak için gradyanı kullanın ve bu yönde bir adım atmak için parametreleri güncelleyin.

18.4.3 Matematiksel Optimizasyon (Eniyileme) Üzerine Bir Not

Bu kitap boyunca, derin öğrenme ortamında karşılaştığımız tüm işlevlerin açıkça en aza indirilemeyecek kadar karmaşık olmasından dolayı pratik nedenlerle doğrudan sayısal eniyileme tekniklerine odaklanıyoruz.

Bununla birlikte, yukarıda elde ettiğimiz geometrik anlayışın bize fonksiyonları doğrudan optimize etme hakkında ne söylediğini düşünmek faydalı bir alıştırmadır.

Bir $L(\mathbf{x})$ işlevini en aza indiren $\mathbf{x}_0$'nin değerini bulmak istediğimizi varsayalım. Diyelim ki birisi bize bir değer veriyor ve bize L'yi en aza indiren şeyin bu değer olduğunu söylüyor. Yanıtın makul olup olmadığını kontrol edebileceğimiz bir şey var mı?

Tekrar düşünün (18.4.5):

$$L(\mathbf{x}_0 + \boldsymbol{\epsilon}) \approx L(\mathbf{x}_0) + \boldsymbol{\epsilon} \cdot \nabla_{\mathbf{x}} L(\mathbf{x}_0). \tag{18.4.11}$$

Gradyan sıfır değilse, L'nin daha küçük değerini bulmak için $-\epsilon\nabla_{\mathbf{x}} L(\mathbf{x}_0)$ yönünde bir adım atabileceğimizi biliyoruz. Bu nedenle, gerçekten en düşük değerde isek, böyle bir durum olamaz!

$\mathbf{x}_0$ bir minimum ise, $\nabla_{\mathbf{x}} L(\mathbf{x}_0) = 0$ olduğu sonucuna varabiliriz. $\nabla_{\mathbf{x}} L(\mathbf{x}_0) = 0$ ifadesi gerçek olan noktaları *kritik nokta* diye çağırıyoruz.

Bu güzel bir bilgi, çünkü bazı nadir durumlarda gradyanın sıfır olduğu tüm noktaları açıkça *bulabiliriz* ve en küçük değere sahip olanı bulabiliriz.

Somut bir örnek için bu işlevi düşünün:

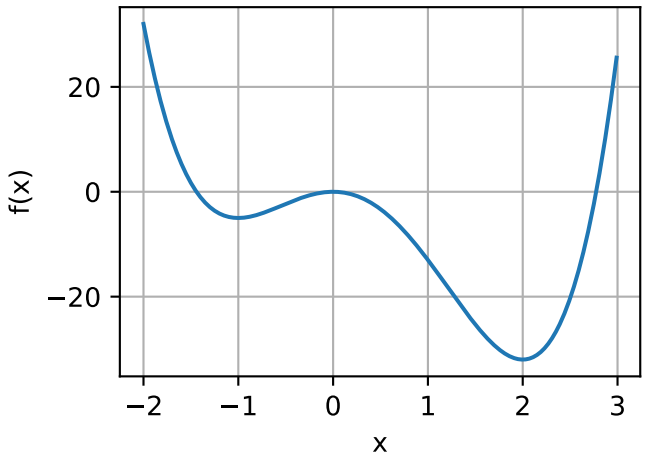
$$f(x) = 3x^4 - 4x^3 - 12x^2. \tag{18.4.12}$$

Bu fonksiyonun türevi aşağıdadır:

$$\frac{df}{dx} = 12x^3 - 12x^2 - 24x = 12x(x-2)(x+1). \tag{18.4.13}$$

Minimumun olası konumları $x = -1, 0, 2$'dir, burada fonksiyon sırasıyla $-5, 0, -32$ değerlerini alır ve böylece $x = 2$ olduğunda fonksiyonumuzu küçülttüğümüz sonucuna varabiliriz. Hızlı bir çizim bunu doğrular.

```
x = torch.arange(-2, 3, 0.01)
f = (3 * x**4) - (4 * x**3) - (12 * x**2)

d2l.plot(x, f, 'x', 'f(x)')
```



Bu, teorik veya sayısal olarak çalışırken bilinmesi gereken önemli bir gerçeğin altını çiziyor: Bir işlevi en aza indirebileceğimiz (veya en yükseğe çıkarabileceğimiz) olası noktalar sıfıra eşit bir gradyana sahip olacaktır, ancak sıfır gradyanlı her nokta gerçek *küresel (global)* minimum (veya maksimum) değildir.

18.4.4 Çok Değişkenli Zincir Kuralı

Pek çok terim oluşturarak yapabileceğimiz dört değişkenli (w, x, y ve z) bir fonksiyonumuz olduğunu varsayalım:

$$\begin{aligned} f(u, v) &= (u+v)^2 \\ u(a, b) &= (a+b)^2, \qquad v(a, b) = (a-b)^2, \\ a(w, x, y, z) &= (w+x+y+z)^2, \qquad b(w, x, y, z) = (w+x-y-z)^2. \end{aligned} \tag{18.4.14}$$

Sinir ağları ile çalışırken bu tür denklem zincirleri yaygındır, bu nedenle bu tür işlevlerin gradyanlarının nasıl hesaplanacağını anlamaya çalışmak çok önemlidir. Hangi değişkenlerin doğrudan birbiriyle ilişkili olduğuna bakarsak, bu bağlantının görsel ipuçlarını Fig. 18.4.1 içinde görmeye başlayabiliriz.

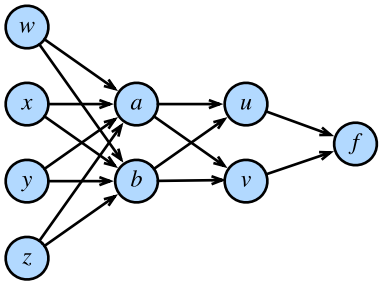


Fig. 18.4.1: Düğümlerin değerleri temsil ettiği ve kenarların işlevsel bağımlılığı gösterdiği yukarıda geçen işlev ilişkileri.

Hiçbir şey bizi sadece (18.4.14) denkleminden her şeyi birleştirmekten ve bunu yazmaktan alıkoyamaz

$$f(w,x,y,z) = \left(\left((w+x+y+z)^2 + (w+x-y-z)^2\right)^2 + \left((w+x+y+z)^2 - (w+x-y-z)^2\right)^2\right)^2. \tag{18.4.15}$$

O zaman türevi sadece tek değişkenli türevler kullanarak alabiliriz, ancak bunu yaparsak, kendimizi hızlı bir şekilde terimlere boğulmuş buluruz, çoğu da tekrar eder! Gerçekten de, örneğin şunu görebiliriz:

$$\begin{aligned}\frac{\partial f}{\partial w} = & 2\left(2\left(2(w+x+y+z) - 2(w+x-y-z)\right)\left((w+x+y+z)^2 - (w+x-y-z)^2\right) + \right.\\ & 2\left(2(w+x-y-z) + 2(w+x+y+z)\right)\left((w+x-y-z)^2 + (w+x+y+z)^2\right)\Big) \times \\ & \left(\left((w+x+y+z)^2 - (w+x-y-z)^2\right)^2 + \left((w+x-y-z)^2 + (w+x+y+z)^2\right)^2\right).\end{aligned} \tag{18.4.16}$$

Daha sonra $\frac{\partial f}{\partial x}$'i de hesaplamak isteseydik, birçok tekrarlanan terim ve iki türev arasında birçok *paylaşılan* tekrarlanan terimle tekrar benzer bir denklem elde ederdik. Bu, büyük miktarda boşa harcanan işi temsil ediyor ve eğer türevleri bu şekilde hesaplamamız gerekseydi, tüm derin öğrenme devrimi başlamadan önce durmuş olurdu!

Sorunu çözelim. a'yı değiştirdiğimizde f'nin nasıl değiştiğini anlamaya çalışarak başlayacağız, esasen w, x, y ve z değerlerinin mevcut olmadığını varsayarak yapacağız. Gradyan ile ilk kez çalışırken yaptığımız gibi akıl yürüteceğiz. Bir a alalım ve ona küçük bir miktar ϵ ekleyelim.

$$\begin{aligned} & f(u(a+\epsilon,b), v(a+\epsilon,b)) \\ \approx & f\left(u(a,b) + \epsilon\frac{\partial u}{\partial a}(a,b), v(a,b) + \epsilon\frac{\partial v}{\partial a}(a,b)\right) \\ \approx & f(u(a,b),v(a,b)) + \epsilon\left[\frac{\partial f}{\partial u}(u(a,b),v(a,b))\frac{\partial u}{\partial a}(a,b) + \frac{\partial f}{\partial v}(u(a,b),v(a,b))\frac{\partial v}{\partial a}(a,b)\right]. \end{aligned} \tag{18.4.17}$$

İlk satır kısmi türev tanımından, ikincisi gradyan tanımından gelir. $\frac{\partial f}{\partial u}(u(a,b),v(a,b))$ ifadesinde olduğu gibi, her türevi tam olarak nerede değerlendirdiğimizi izlemek gösterimsel olarak külfetlidir, bu nedenle sık sık bunu çok daha akılda kalıcı olarak kısaltırız.

$$\frac{\partial f}{\partial a} = \frac{\partial f}{\partial u}\frac{\partial u}{\partial a} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial a}. \tag{18.4.18}$$

Sürecin anlamını düşünmekte fayda var. $f(u(a, b), v(a, b))$ biçimindeki bir fonksiyonun a'daki bir değişiklikle değerini nasıl değiştirdiğini anlamaya çalışıyoruz. Bunun meydana gelebileceği iki yol vardır: $a \to u \to f$ ve $a \to v \to f$. Bu katkıların her ikisini de zincir kuralı aracılığıyla hesaplayabiliriz: $\frac{\partial w}{\partial u} \cdot \frac{\partial u}{\partial x}$ ve $\frac{\partial w}{\partial v} \cdot \frac{\partial v}{\partial x}$ hesaplanırlar ve toplanırlar.

Sağdaki işlevlerin, Fig. 18.4.2 şeklinde gösterildiği gibi soldakilere bağlı olan işlevlere bağımlı olduğu farklı bir işlev ağımız olduğunu hayal edin.

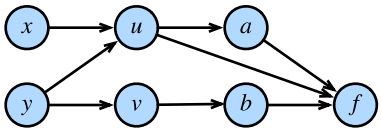


Fig. 18.4.2: Zincir kuralının daha ince bir başka örneği.

$\frac{\partial f}{\partial y}$ gibi bir şeyi hesaplamak için, y'den f'e kadar tüm (bu durumda 3) yolları toplamamız gerekir.

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial u}\frac{\partial u}{\partial y} + \frac{\partial f}{\partial u}\frac{\partial u}{\partial y} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial v}\frac{\partial v}{\partial y}. \tag{18.4.19}$$

Zincir kuralını bu şekilde anlamak, gradyanların ağlar boyunca nasıl aktığını ve neden LSTM'lerdeki (Section 9.2) veya artık (residual) katmanlardaki (Section 7.6) gibi çeşitli mimari seçimlerin gradyan akışını kontrol etmeye yardımcı olarak öğrenme sürecini şekillendirdiğini anlamaya çalışırken büyük kazançlar sağlayacaktır.

18.4.5 Geri Yayma Algoritması

Önceki bölümdeki (18.4.14) örneğine dönelim:

$$\begin{aligned} f(u, v) &= (u + v)^2 \\ u(a, b) &= (a + b)^2, \qquad v(a, b) = (a - b)^2, \\ a(w, x, y, z) &= (w + x + y + z)^2, \qquad b(w, x, y, z) = (w + x - y - z)^2. \end{aligned} \tag{18.4.20}$$

Diyelim ki $\frac{\partial f}{\partial w}$'yi hesaplıyoruz, çok değişkenli zincir kuralını uygularsak şunu görebiliriz:

$$\begin{aligned} \frac{\partial f}{\partial w} &= \frac{\partial f}{\partial u}\frac{\partial u}{\partial w} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial w}, \\ \frac{\partial u}{\partial w} &= \frac{\partial u}{\partial a}\frac{\partial a}{\partial w} + \frac{\partial u}{\partial b}\frac{\partial b}{\partial w}, \\ \frac{\partial v}{\partial w} &= \frac{\partial v}{\partial a}\frac{\partial a}{\partial w} + \frac{\partial v}{\partial b}\frac{\partial b}{\partial w}. \end{aligned} \tag{18.4.21}$$

$\frac{\partial f}{\partial w}$ hesaplamak için bu ayrıştırmayı kullanmayı deneyelim. Burada ihtiyacımız olan tek şeyin çeşitli tek adımlık kısmi türevler olduğuna dikkat edin:

$$\begin{aligned} \frac{\partial f}{\partial u} &= 2(u + v), \quad \frac{\partial f}{\partial v} = 2(u + v), \\ \frac{\partial u}{\partial a} &= 2(a + b), \quad \frac{\partial u}{\partial b} = 2(a + b), \\ \frac{\partial v}{\partial a} &= 2(a - b), \quad \frac{\partial v}{\partial b} = -2(a - b), \\ \frac{\partial a}{\partial w} &= 2(w + x + y + z), \quad \frac{\partial b}{\partial w} = 2(w + x - y - z). \end{aligned} \tag{18.4.22}$$

Bunu kodda yazarsak, bu oldukça yönetilebilir bir ifade olur.

```
# Girdilerden çıktılara fonksiyonun değerini hesapla
w, x, y, z = -1, 0, -2, 1
a, b = (w + x + y + z)**2, (w + x - y - z)**2
u, v = (a + b)**2, (a - b)**2
f = (u + v)**2
print(f'    f at {w}, {x}, {y}, {z} is {f}')

# Tek adımlı kısmileri hesapla
df_du, df_dv = 2*(u + v), 2*(u + v)
du_da, du_db, dv_da, dv_db = 2*(a + b), 2*(a + b), 2*(a - b), -2*(a - b)
da_dw, db_dw = 2*(w + x + y + z), 2*(w + x - y - z)

# Girdilerden çıktılara nihai sonucu hesapla
du_dw, dv_dw = du_da*da_dw + du_db*db_dw, dv_da*da_dw + dv_db*db_dw
df_dw = df_du*du_dw + df_dv*dv_dw
print(f'df/dw at {w}, {x}, {y}, {z} is {df_dw}')
```

```
    f at -1, 0, -2, 1 is 1024
df/dw at -1, 0, -2, 1 is -4096
```

Ancak, bunun hala $\frac{\partial f}{\partial x}$ gibi bir şeyi hesaplamayı kolaylaştırmadığını unutmayın. Bunun nedeni, zincir kuralını uygulamayı seçtiğimiz *yoldur*. Yukarıda ne yaptığımıza bakarsak, elimizden geldiğince paydada her zaman ∂w tuttuk. Bu şekilde, w değişkenin her değişkeni nasıl değiştirdiğini görerek zincir kuralını uygulamayı seçtik. İstediğimiz buysa, bu iyi bir fikir olabilir. Bununla birlikte, derin öğrenmedeki motivasyonumuza geri dönün: Her parametrenin *kaybı* nasıl değiştirdiğini görmek istiyoruz. Esasında, yapabildiğimiz her yerde ∂f'yi payda tutan zincir kuralını uygulamak istiyoruz!

Daha açık olmak gerekirse, şunu yazabileceğimize dikkat edin:

$$\begin{aligned}\frac{\partial f}{\partial w} &= \frac{\partial f}{\partial a}\frac{\partial a}{\partial w} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial w},\\ \frac{\partial f}{\partial a} &= \frac{\partial f}{\partial u}\frac{\partial u}{\partial a} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial a},\\ \frac{\partial f}{\partial b} &= \frac{\partial f}{\partial u}\frac{\partial u}{\partial b} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial b}.\end{aligned} \tag{18.4.23}$$

Zincir kuralının bu uygulaması bizim açıkça $\frac{\partial f}{\partial u}, \frac{\partial f}{\partial v}, \frac{\partial f}{\partial a}, \frac{\partial f}{\partial b}$, ve $\frac{\partial f}{\partial w}$'ları hesaplamamızı gerektirir. Aşağıdaki denklemleri de dahil etmekten bizi hiçbir şey alıkoyamaz:

$$\begin{aligned}\frac{\partial f}{\partial x} &= \frac{\partial f}{\partial a}\frac{\partial a}{\partial x} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial x},\\ \frac{\partial f}{\partial y} &= \frac{\partial f}{\partial a}\frac{\partial a}{\partial y} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial y},\\ \frac{\partial f}{\partial z} &= \frac{\partial f}{\partial a}\frac{\partial a}{\partial z} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial z}.\end{aligned} \tag{18.4.24}$$

Sonra tüm ağdaki *herhangi bir* düğümü değiştirdiğimizde f değerinin nasıl değiştiğini takip edebiliyoruz. Haydi uygulayalım.

```
# Girdilerden çıktılara fonksiyonun değerini hesapla
w, x, y, z = -1, 0, -2, 1
a, b = (w + x + y + z)**2, (w + x - y - z)**2
```

(continues on next page)

(continued from previous page)

```
u, v = (a + b)**2, (a - b)**2
f = (u + v)**2
print(f'f at {w}, {x}, {y}, {z} is {f}')

# Yukarıdaki ayrıştırmayı kullanarak türevi hesapla
# İlk önce tek adımlı kısmileri hesapla
df_du, df_dv = 2*(u + v), 2*(u + v)
du_da, du_db, dv_da, dv_db = 2*(a + b), 2*(a + b), 2*(a - b), -2*(a - b)
da_dw, db_dw = 2*(w + x + y + z), 2*(w + x - y - z)
da_dx, db_dx = 2*(w + x + y + z), 2*(w + x - y - z)
da_dy, db_dy = 2*(w + x + y + z), -2*(w + x - y - z)
da_dz, db_dz = 2*(w + x + y + z), -2*(w + x - y - z)

# Şimdi herhangi bir değeri çıktıdan girdiye değiştirdiğimizde f'nin nasıl değiştiğini␣
↪hesapla
df_da, df_db = df_du*du_da + df_dv*dv_da, df_du*du_db + df_dv*dv_db
df_dw, df_dx = df_da*da_dw + df_db*db_dw, df_da*da_dx + df_db*db_dx
df_dy, df_dz = df_da*da_dy + df_db*db_dy, df_da*da_dz + df_db*db_dz

print(f'df/dw at {w}, {x}, {y}, {z} is {df_dw}')
print(f'df/dx at {w}, {x}, {y}, {z} is {df_dx}')
print(f'df/dy at {w}, {x}, {y}, {z} is {df_dy}')
print(f'df/dz at {w}, {x}, {y}, {z} is {df_dz}')
```

```
f at -1, 0, -2, 1 is 1024
df/dw at -1, 0, -2, 1 is -4096
df/dx at -1, 0, -2, 1 is -4096
df/dy at -1, 0, -2, 1 is -4096
df/dz at -1, 0, -2, 1 is -4096
```

Türevleri, girdilerden çıktılara, ileriye doğru, hesaplamaktansa, f'den girdilere doğru hesapladığımız gerçeği (yukarıdaki ilk kod parçacığında yaptığımız gibi), bu algoritmaya adını veren şeydir: *Geri yayma*. İki adım olduğunu unutmayın: 1. Fonksiyonun değerini ve önden arkaya tek adımlık kısmi değerlerini hesaplayın. Yukarıda yapılmasa da, bu tek bir *ileri geçişte* birleştirilebilir. 2. Arkadan öne doğru f gradyanını hesaplayın. Biz buna *geriye doğru geçiş* diyoruz.

Bu, her derin öğrenme algoritmasının, bir geçişte ağdaki her ağırlığa göre kaybın gradyanının hesaplanmasına izin vermek, uyguladığı şeydir. Böyle bir ayrışmaya sahip olmamız şaşırtıcı bir gerçektir.

Bunu nasıl içeri işlediğimizi görmek için bu örneğe hızlıca bir göz atalım.

```
# ndarrays olarak ilklet, ardından gradyanları ekle
w = torch.tensor([-1.], requires_grad=True)
x = torch.tensor([0.], requires_grad=True)
y = torch.tensor([-2.], requires_grad=True)
z = torch.tensor([1.], requires_grad=True)
# Hesaplamayı her zamanki gibi yap, gradyanları takip et
a, b = (w + x + y + z)**2, (w + x - y - z)**2
u, v = (a + b)**2, (a - b)**2
f = (u + v)**2
```

(continues on next page)

(continued from previous page)

```
# Geriye doğru geçişi uygula
f.backward()

print(f'df/dw at {w.data.item()}, {x.data.item()}, {y.data.item()}, '
      f'{z.data.item()} is {w.grad.data.item()}')
print(f'df/dx at {w.data.item()}, {x.data.item()}, {y.data.item()}, '
      f'{z.data.item()} is {x.grad.data.item()}')
print(f'df/dy at {w.data.item()}, {x.data.item()}, {y.data.item()}, '
      f'{z.data.item()} is {y.grad.data.item()}')
print(f'df/dz at {w.data.item()}, {x.data.item()}, {y.data.item()}, '
      f'{z.data.item()} is {z.grad.data.item()}')
```

```
df/dw at -1.0, 0.0, -2.0, 1.0 is -4096.0
df/dx at -1.0, 0.0, -2.0, 1.0 is -4096.0
df/dy at -1.0, 0.0, -2.0, 1.0 is -4096.0
df/dz at -1.0, 0.0, -2.0, 1.0 is -4096.0
```

Yukarıda yaptığımız şeylerin tümü, `f.backwards()` çağrısıyla otomatik olarak yapılabilir.

18.4.6 Hessianlar

Tek değişkenli hesapta olduğu gibi, bir işleve tek başına gradyanı kullanmaktan daha iyi bir yaklaşıklama elde edebilmek için daha yüksek dereceli türevleri düşünmek yararlıdır.

Birkaç değişkenli fonksiyonların daha yüksek dereceden türevleriyle çalışırken karşılaşılan anlık bir sorun vardır ve bu da çok sayıda olmasıdır. $f(x_1, \ldots, x_n)$ fonksiyonun n değişkeni varsa, o zaman n^2 tane ikinci türev alabiliriz, yani herhangi bir i ve j seçeneği için:

$$\frac{d^2 f}{dx_i dx_j} = \frac{d}{dx_i}\left(\frac{d}{dx_j} f\right). \tag{18.4.25}$$

Bu, geleneksel olarak *Hessian* adı verilen bir matris içinde toplanır:

$$\mathbf{H}_f = \begin{bmatrix} \frac{d^2 f}{dx_1 dx_1} & \cdots & \frac{d^2 f}{dx_1 dx_n} \\ \vdots & \ddots & \vdots \\ \frac{d^2 f}{dx_n dx_1} & \cdots & \frac{d^2 f}{dx_n dx_n} \end{bmatrix}. \tag{18.4.26}$$

Bu matrisin her girdisi bağımsız değildir. Aslında, her iki *karışık kısmi* (birden fazla değişkene göre kısmi türevler) var ve sürekli olduğu sürece, herhangi bir i ve j için şunu söyleyebiliriz:

$$\frac{d^2 f}{dx_i dx_j} = \frac{d^2 f}{dx_j dx_i}. \tag{18.4.27}$$

Bunu, önce bir işlevi x_i yönünde ve ardından x_j yönünde dürtmeyi ve ardından bunun sonucunu önce x_j ve sonra x_i yönünde dürtersek ne olacağıyla karşılaştırmak izler; burada her iki sıranın da f'nin çıktısında aynı nihai değişikliğe yol açtığı bilgisine ulaşırız.

Tek değişkenlerde olduğu gibi, fonksiyonun bir noktanın yakınında nasıl davrandığına dair daha iyi bir fikir edinmek için bu türevleri kullanabiliriz. Özellikle, tek bir değişkende gördüğümüz gibi, $\mathbf{x}_0$ noktasının yakınında en uygun ikinci derece fonksiyonu bulmak için kullanabiliriz.

Bir örnek görelim. $f(x_1, x_2) = a + b_1 x_1 + b_2 x_2 + c_{11} x_1^2 + c_{12} x_1 x_2 + c_{22} x_2^2$ olduğunu varsayalım. Bu, iki değişkenli bir ikinci dereceden fonksiyon genel formudur. Fonksiyonun değerine, gradyanına ve Hessian (18.4.26) değerine bakarsak, hepsi sıfır noktasında şöyle gözükür:

$$\begin{aligned} f(0, 0) &= a, \\ \nabla f(0, 0) &= \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}, \\ \mathbf{H} f(0, 0) &= \begin{bmatrix} 2c_{11} & c_{12} \\ c_{12} & 2c_{22} \end{bmatrix}, \end{aligned} \tag{18.4.28}$$

esas polinomumuzu şöyle diyerek geri elde edebiliriz;

$$f(\mathbf{x}) = f(0) + \nabla f(0) \cdot \mathbf{x} + \frac{1}{2} \mathbf{x}^\top \mathbf{H} f(0) \mathbf{x}. \tag{18.4.29}$$

Genel olarak, bu genişletmeyi herhangi bir $\mathbf{x}_0$ noktasında hesaplasaydık, şunu görürdük

$$f(\mathbf{x}) = f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0) \cdot (\mathbf{x} - \mathbf{x}_0) + \frac{1}{2} (\mathbf{x} - \mathbf{x}_0)^\top \mathbf{H} f(\mathbf{x}_0) (\mathbf{x} - \mathbf{x}_0). \tag{18.4.30}$$

Bu, herhangi bir boyuttaki girdi için çalışır ve bir noktadaki herhangi bir işleve en iyi yaklaşıklayan ikinci derece polinomu sağlar. Bir örnek vermek gerekirse, fonksiyonun grafiğini çizelim.

$$f(x, y) = x e^{-x^2 - y^2}. \tag{18.4.31}$$

Gradyan ve Hessian şöyle hesaplanabilir:

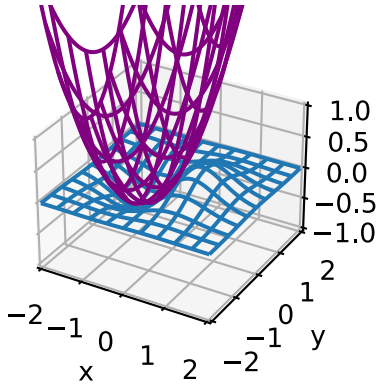
$$\nabla f(x, y) = e^{-x^2 - y^2} \begin{pmatrix} 1 - 2x^2 \\ -2xy \end{pmatrix} \text{ and } \mathbf{H} f(x, y) = e^{-x^2 - y^2} \begin{pmatrix} 4x^3 - 6x & 4x^2 y - 2y \\ 4x^2 y - 2y & 4xy^2 - 2x \end{pmatrix}. \tag{18.4.32}$$

Böylece, biraz cebirle, $[-1, 0]^\top$'deki yaklaşık ikinci dereceden polinomu görebiliriz:

$$f(x, y) \approx e^{-1} \left(-1 - (x + 1) + (x + 1)^2 + y^2\right). \tag{18.4.33}$$

```
# Izgarayı oluştur ve işlevi hesapla
x, y = torch.meshgrid(torch.linspace(-2, 2, 101),
                      torch.linspace(-2, 2, 101))

z = x*torch.exp(- x**2 - y**2)

# (1, 0)'da gradyan ve Hessian ile ikinci dereceden yaklaştırmayı hesapla
w = torch.exp(torch.tensor([-1.]))*(-1 - (x + 1) + 2 * (x + 1)**2 + 2 * y**2)

# Pİşlevi çiz
ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x.numpy(), y.numpy(), z.numpy(),
                  **{'rstride': 10, 'cstride': 10})
ax.plot_wireframe(x.numpy(), y.numpy(), w.numpy(),
                  **{'rstride': 10, 'cstride': 10}, color='purple')
d2l.plt.xlabel('x')
d2l.plt.ylabel('y')
d2l.set_figsize()
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
ax.set_zlim(-1, 1)
ax.dist = 12
```

```
/home/d2l-worker/miniconda3/envs/d2l-tr-release-0/lib/python3.9/site-packages/torch/
↪functional.py:478: UserWarning: torch.meshgrid: in an upcoming release, it will be␣
↪required to pass the indexing argument. (Triggered internally at  ../aten/src/ATen/native/
↪TensorShape.cpp:2895.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
```



Bu, Section 11.3 içinde tartışılan Newton algoritmasının temelini oluşturur; sayısal optimizasyonu yinelemeli olarak uygulayarak en uygun ikinci derece polinomu bulur, sonra da tam olarak bu ikinci dereceden polinomu en aza indiririz.

18.4.7 Biraz Matris Hesabı

Matrisleri içeren fonksiyonların türevlerinin oldukça iyi olduğu ortaya çıktı. Bu bölüm gösterimsel olarak ağır hale gelebilir, bu nedenle ilk okumada atlanabilir, ancak genel matris işlemlerini içeren fonksiyonların türevlerinin genellikle başlangıçta tahmin edebileceğinden çok daha temiz olduğunu bilmek yararlıdır, özellikle de merkezi matris işlemlerinin derin öğrenme uygulamaları için ne kadar olduğu göz önüne alındığında.

Bir örnekle başlayalım. Bir sabit sütun vektörümüz $\boldsymbol{\beta}$ olduğunu ve $f(\mathbf{x}) = \boldsymbol{\beta}^\top \mathbf{x}$ çarpım fonksiyonunu almak ve $\mathbf{x}$'i değiştirdiğimizde iç çarpım nasıl değişir anlamak istediğimizi varsayalım.

Makine öğrenmesinde matris türevleriyle çalışırken faydalı olacak bir gösterim parçası, kısmi türevlerimizi türevin paydada bulunduğu vektör, matris veya tensörün şekline dönüştürdüğümüz *payda düzenli matris türevi* olarak adlandırılır. Bu durumda şöyle yazacağız

$$\frac{df}{d\mathbf{x}} = \begin{bmatrix} \frac{df}{dx_1} \\ \vdots \\ \frac{df}{dx_n} \end{bmatrix}, \tag{18.4.34}$$

Burada $\mathbf{x}$ sütun vektörünün şekline eşleştirdik.

İşlevimizi bileşenlere yazarsak:

$$f(\mathbf{x}) = \sum_{i=1}^{n} \beta_i x_i = \beta_1 x_1 + \cdots + \beta_n x_n. \tag{18.4.35}$$

Şimdi x_1 göre kısmi türevi alırsak, ilk terim hariç her şeyin sıfır olduğuna dikkat edin, sadece x_1 ile β_1 ile çarpılır, böylece bunu elde ederiz:

$$\frac{df}{dx_1} = \beta_1, \tag{18.4.36}$$

ya da daha genel olarak

$$\frac{df}{dx_i} = \beta_i. \tag{18.4.37}$$

Şimdi bunu bir matris olarak görmek için yeniden birleştirebiliriz

$$\frac{df}{d\mathbf{x}} = \begin{bmatrix} \frac{df}{dx_1} \\ \vdots \\ \frac{df}{dx_n} \end{bmatrix} = \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_n \end{bmatrix} = \boldsymbol{\beta}. \tag{18.4.38}$$

Bu, matris hesabı ile ilgili olarak bu bölümde sık sık karşılaşacağımız birkaç etkeni göstermektedir:

- İlk olarak, hesaplamalar daha çok işin içine girecek.
- İkinci olarak, nihai sonuçlar ara süreçten çok daha temizdir ve her zaman tek değişkenli duruma benzer görünecektir. Bu durumda, $\frac{d}{dx}(bx) = b$ ve $\frac{d}{d\mathbf{x}}(\boldsymbol{\beta}^\top \mathbf{x}) = \boldsymbol{\beta}$'nin ikisi de benzerdir.
- Üçüncüsü, devrikler genellikle herhangi bir yerden ortaya çıkabilirler. Bunun temel nedeni, paydanın şeklini eşleştirmemizdir, böylece matrisleri çarptığımızda, esas terimin şekline geri dönmek için devrikler almamız gerekecek.

Önsezi oluşturmaya devam etmek için biraz daha zor bir hesaplama deneyelim. Bir sütun vektörümüz $\mathbf{x}$ ve kare matrisimiz A olduğunu ve şunu hesaplamak istediğimizi varsayalım.

$$\frac{d}{d\mathbf{x}}(\mathbf{x}^\top A\mathbf{x}). \tag{18.4.39}$$

Gösterimde daha kolay oynama yaparak ileriye doğru ilerlemek için, bu problemi Einstein gösterimini kullanarak ele alalım. Bu durumda fonksiyonu şu şekilde yazabiliriz:

$$\mathbf{x}^\top A\mathbf{x} = x_i a_{ij} x_j. \tag{18.4.40}$$

Türevimizi hesaplamak için, her k için, değerin ne olduğunu anlamamız gerekir:

$$\frac{d}{dx_k}(\mathbf{x}^\top A\mathbf{x}) = \frac{d}{dx_k} x_i a_{ij} x_j. \tag{18.4.41}$$

Çarpım kuralını uygulayalım:

$$\frac{d}{dx_k} x_i a_{ij} x_j = \frac{dx_i}{dx_k} a_{ij} x_j + x_i a_{ij} \frac{dx_j}{dx_k}. \tag{18.4.42}$$

$\frac{dx_i}{dx_k}$ gibi bir terim için, bunun $i = k$ için bir ve aksi halde sıfır olduğunda görmek zor değildir. Bu, i ve k değerlerinin farklı olduğu her terimin bu toplamdan kaybolduğu anlamına gelir, bu nedenle bu ilk toplamda kalan tek terim, $i = k$ olanlardır. Aynı çıkarsama, $j = k$'ya ihtiyacımız olduğu ikinci terim için de geçerlidir. Böylece:

$$\frac{d}{dx_k} x_i a_{ij} x_j = a_{kj} x_j + x_i a_{ik}. \tag{18.4.43}$$

Şimdi, Einstein gösterimindeki indislerin isimleri keyfidir — i ve j'nin farklı olması bu noktada bu hesaplama için önemsizdir, bu yüzden ikisinin de i kullanması için yeniden indeksleyebiliriz:

$$\frac{d}{dx_k} x_i a_{ij} x_j = a_{ki} x_i + x_i a_{ik} = (a_{ki} + a_{ik}) x_i. \tag{18.4.44}$$

Şimdi, burası daha ileri gitmek için biraz pratik yapmaya ihtiyacımız olan yerdir. Bu sonucu matris işlemleri açısından belirlemeye çalışalım. $a_{ki} + a_{ik}$, $\mathbf{A} + \mathbf{A}^\top$'nin k, i-inci bileşenidir.

$$\frac{d}{dx_k} x_i a_{ij} x_j = [\mathbf{A} + \mathbf{A}^\top]_{ki} x_i. \tag{18.4.45}$$

Benzer şekilde, bu terim artık $\mathbf{A} + \mathbf{A}^\top$ matrisinin $\mathbf{x}$ vektörü ile çarpımıdır, dolayısıyla şunu görüyoruz

$$\left[\frac{d}{d\mathbf{x}}(\mathbf{x}^\top A \mathbf{x})\right]_k = \frac{d}{dx_k} x_i a_{ij} x_j = [(\mathbf{A} + \mathbf{A}^\top)\mathbf{x}]_k. \tag{18.4.46}$$

Böylece, (18.4.39) denkleminden istenen türevin $k.$ girdisinin sağdaki vektörün $k.$ girdisi olduğunu ve dolayısıyla ikisinin aynı olduğunu görüyoruz. Sonunda:

$$\frac{d}{d\mathbf{x}}(\mathbf{x}^\top A \mathbf{x}) = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}. \tag{18.4.47}$$

Bu, önceki sonucumuzdan önemli ölçüde daha fazla çalışma gerektirdi, ancak nihai sonuç küçük. Bundan daha fazlası, geleneksel tek değişkenli türevler için aşağıdaki hesaplamayı düşünün:

$$\frac{d}{dx}(xax) = \frac{dx}{dx}ax + xa\frac{dx}{dx} = (a + a)x. \tag{18.4.48}$$

Eşdeğer olarak $\frac{d}{dx}(ax^2) = 2ax = (a + a)x$'dir. Yine, tek değişkenli sonuca benzeyen, ancak içine devrik atılmış bir sonuç elde ederiz.

Bu noktada, model oldukça şüpheli görünmelidir, bu yüzden nedenini anlamaya çalışalım. Bunun gibi matris türevlerini aldığımızda, öncelikle aldığımız ifadenin başka bir matris ifadesi olacağını varsayalım: Onu matrislerin çarpımları ve toplamları ve bunların devrikleri cinsinden yazabileceğimiz bir ifade. Böyle bir ifade varsa, tüm matrisler için doğru olması gerekecektir. Özellikle, 1×1 matrisler için doğru olması gerekecektir, ki bu durumda matris çarpımı sadece sayıların çarpımıdır, matris toplamı sadece toplamdır ve devrik hiçbir şey yapmaz! Başka bir deyişle, aldığımız ifade ne olursa olsun tek değişkenli ifadeyle *eşleşmelidir*. Bu, biraz pratikle, yalnızca ilişkili tek değişkenli ifadenin neye benzemesi gerektiğini bilerek matris türevlerini tahmin edebileceğiniz anlamına gelir!

Bunu deneyelim. $\mathbf{X}$'nin bir $n \times m$ matrisi, $\mathbf{U}$'nun $n \times r$ ve $\mathbf{V}$'nin $r \times m$ olduğunu varsayalım. Hesaplamayı deneyelim

$$\frac{d}{d\mathbf{V}} \|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = ? \tag{18.4.49}$$

Bu hesaplama, matris çarpanlarına ayırma adı verilen bir alanda önemlidir. Ancak bizim için bu, hesaplanması gereken bir türevdir. Bunun 1×1 matrisler için ne olacağını hayal etmeye çalışalım. Bu durumda ifadeyi alırız

$$\frac{d}{dv}(x - uv)^2 = -2(x - uv)u, \tag{18.4.50}$$

Burada türev oldukça standarttır. Bunu tekrar bir matris ifadesine dönüştürmeye çalışırsak,

$$\frac{d}{d\mathbf{V}} \|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = -2(\mathbf{X} - \mathbf{U}\mathbf{V})\mathbf{U}. \tag{18.4.51}$$

Ancak buna bakarsak pek işe yaramıyor. $\mathbf{X}$'in $n \times m$ olduğunu ve $\mathbf{U}\mathbf{V}$ öyle olduğunu hatırlayın, dolayısıyla $2(\mathbf{X} - \mathbf{U}\mathbf{V})$, $n \times m$'dir. Öte yandan, $\mathbf{U}$, $n \times r$'dir ve boyutlar eşleşmediğinden, $n \times m$ ve a $n \times r$ matrisleri çarpamayız!

$\frac{d}{d\mathbf{V}}$'i elde etmek istiyoruz, bu $\mathbf{V}$ ile aynı şekle sahip, ki bu $r \times m$. Öyleyse bir şekilde bir $n \times m$ matrisi ve bir $n \times r$ matrisi almalıyız, bunları bir $r \times m$ matris elde etmek için birbirleriyle çarpmalıyız (belki bazı devriklerle). Bunu $\mathbf{U}^\top$'yu $(\mathbf{X} - \mathbf{UV})$ ile çarparak yapabiliriz. Böylelikle, (18.4.49) için çözümü tahmin edebiliriz:

$$\frac{d}{d\mathbf{V}}\|\mathbf{X} - \mathbf{UV}\|_2^2 = -2\mathbf{U}^\top(\mathbf{X} - \mathbf{UV}). \tag{18.4.52}$$

Bunun işe yaradığını göstermek için, ayrıntılı bir hesaplama sağlamazsak ihmal etmiş oluruz. Bu pratik kuralın işe yaradığına zaten inanıyorsanız, bu türetmeyi atlamaktan çekinmeyin. Hesaplayalım:

$$\frac{d}{d\mathbf{V}}\|\mathbf{X} - \mathbf{UV}\|_2^2, \tag{18.4.53}$$

Her a ve b için bulmalıyız.

$$\frac{d}{dv_{ab}}\|\mathbf{X} - \mathbf{UV}\|_2^2 = \frac{d}{dv_{ab}}\sum_{i,j}\left(x_{ij} - \sum_k u_{ik}v_{kj}\right)^2. \tag{18.4.54}$$

$\mathbf{X}$ and $\mathbf{U}$'nin tüm girdilerinin $\frac{d}{dv_{ab}}$ bakımında sabitler olduğunu hatırlayarak, türevi toplamın içine itebiliriz, ve zincir kuralını kareye uygularız:

$$\frac{d}{dv_{ab}}\|\mathbf{X} - \mathbf{UV}\|_2^2 = \sum_{i,j} 2\left(x_{ij} - \sum_k u_{ik}v_{kj}\right)\left(-\sum_k u_{ik}\frac{dv_{kj}}{dv_{ab}}\right). \tag{18.4.55}$$

Önceki türetmede olduğu gibi, $\frac{dv_{kj}}{dv_{ab}}$'nın yalnızca $k = a$ ve $j = b$ ise sıfırdan farklı olduğunu görebiliriz. Bu koşullardan herhangi biri geçerli değilse, toplamdaki terim sıfırdır ve onu özgürce atabiliriz. Bunu görüyoruz:

$$\frac{d}{dv_{ab}}\|\mathbf{X} - \mathbf{UV}\|_2^2 = -2\sum_i\left(x_{ib} - \sum_k u_{ik}v_{kb}\right)u_{ia}. \tag{18.4.56}$$

Buradaki önemli bir incelik, $k = a$ şartının iç toplamın içinde oluşmamasıdır, çünkü k iç terimin içinde topladığımız yapay bir değişken. Gösterimsel olarak daha temiz bir örnek için nedenini düşünelim:

$$\frac{d}{dx_1}\left(\sum_i x_i\right)^2 = 2\left(\sum_i x_i\right). \tag{18.4.57}$$

Bu noktadan itibaren, toplamın bileşenlerini belirlemeye başlayabiliriz. İlk olarak,

$$\sum_k u_{ik}v_{kb} = [\mathbf{UV}]_{ib}. \tag{18.4.58}$$

Yani toplamın içindeki tüm ifade:

$$x_{ib} - \sum_k u_{ik}v_{kb} = [\mathbf{X} - \mathbf{UV}]_{ib}. \tag{18.4.59}$$

Bu, artık türevimizi şu şekilde yazabileceğimiz anlamına gelir:

$$\frac{d}{dv_{ab}}\|\mathbf{X} - \mathbf{UV}\|_2^2 = -2\sum_i[\mathbf{X} - \mathbf{UV}]_{ib}u_{ia}. \tag{18.4.60}$$

Bunun bir matrisin a, b öğesi gibi görünmesini istiyoruz ki böylece bir matris ifadesine ulaşmak için önceki örnekte olduğu gibi tekniği kullanabilelim, bu da indislerin sırasını u_{ia} üzerinden değiştirmemiz gerektiği anlamına gelir. $u_{ia} = [\mathbf{U}^\top]_{ai}$ olduğunu fark edersek, bunu yazabiliriz:

$$\frac{d}{dv_{ab}}\|\mathbf{X} - \mathbf{UV}\|_2^2 = -2\sum_i [\mathbf{U}^\top]_{ai}[\mathbf{X} - \mathbf{UV}]_{ib}. \tag{18.4.61}$$

Bu bir matris çarpımıdır ve dolayısıyla şu sonuca varabiliriz:

$$\frac{d}{dv_{ab}}\|\mathbf{X} - \mathbf{UV}\|_2^2 = -2[\mathbf{U}^\top(\mathbf{X} - \mathbf{UV})]_{ab}. \tag{18.4.62}$$

Böylece çözümü şu şekilde yazabiliriz (18.4.49)

$$\frac{d}{d\mathbf{V}}\|\mathbf{X} - \mathbf{UV}\|_2^2 = -2\mathbf{U}^\top(\mathbf{X} - \mathbf{UV}). \tag{18.4.63}$$

Bu, yukarıda tahmin ettiğimiz çözüme uyuyor!

Bu noktada şunu sormak mantıklıdır, "Neden öğrendiğim tüm hesap (kalkülüs) kurallarının matris versiyonlarını yazamıyorum? Bunun hala mekanik olduğu açık. Neden bunun üstesinden gelmiyoruz!" Gerçekten de böyle kurallar var ve (Petersen *et al.*, 2008) mükemmel bir özet sunuyor. Bununla birlikte, matris işlemlerinin tekli değerlere kıyasla birleştirilebileceği çok sayıda yol olması nedeniyle, tek değişkenli olanlara göre çok daha fazla matris türev kuralı vardır. Genellikle en iyisi indislerle çalışmak veya uygun olduğunda bunu otomatik türev almaya bırakmaktır.

18.4.8 Özet

- Daha yüksek boyutlarda, bir boyuttaki türevlerle aynı amaca hizmet eden gradyanları tanımlayabiliriz. Bunlar, girdilerde rastgele küçük bir değişiklik yaptığımızda çok değişkenli bir fonksiyonun nasıl değiştiğini görmemizi sağlar.
- Geri yayma algoritması, birçok kısmi türevin verimli bir şekilde hesaplanmasına izin vermek için çok değişkenli zincir kuralını düzenlemenin bir yöntemi olarak görülebilir.
- Matris hesabı, matris ifadelerinin türevlerini öz olarak yazmamızı sağlar.

18.4.9 Alıştırmalar

1. $\boldsymbol{\beta}$ sütun vektörü verildiğinde, hem $f(\mathbf{x}) = \boldsymbol{\beta}^\top\mathbf{x}$ hem de $g(\mathbf{x}) = \mathbf{x}^\top\boldsymbol{\beta}$ türevlerini hesaplayınız. Neden aynı cevabı alıyorsunuz?
2. $\mathbf{v}$ bir n boyutlu vektör olsun. $\frac{\partial}{\partial\mathbf{v}}\|\mathbf{v}\|_2$ nedir?
3. $L(x, y) = \log(e^x + e^y)$ olsun. Gradyanını hesaplayınız. Gradyanın bileşenlerinin toplamı nedir?
4. $f(x, y) = x^2y + xy^2$ olsun. Tek kritik noktanın $(0, 0)$ olduğunu gösterin. $f(x, x)$'i dikkate alarak, $(0, 0)$'ın maksimum mu, minimum mu olduğunu veya hiçbiri olmadığını belirleyin.
5. $f(\mathbf{x}) = g(\mathbf{x}) + h(\mathbf{x})$ işlevini küçülttüğümüzü varsayalım. $\nabla f = 0$ koşulunu g ve h cinsinden geometrik olarak nasıl yorumlayabiliriz?

Tartışmalar[220]

[220] https://discuss.d2l.ai/t/1090

18.5 Tümlev (Integral) Kalkülüsü

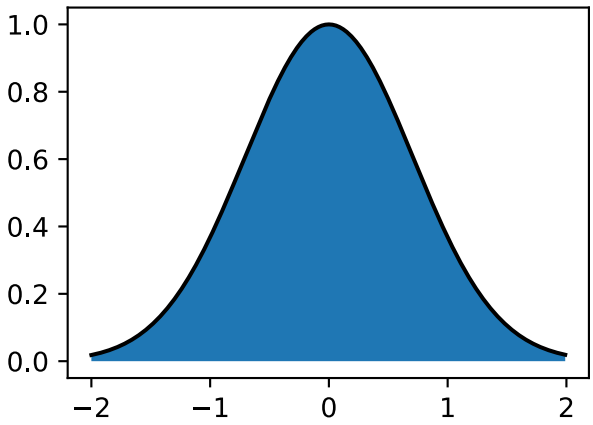
Türev alma, geleneksel bir matematik eğitiminin içeriğinin yalnızca yarısını oluşturur. Diğer yapıtaşı, integral alma, oldukça ayrık bir soru gibi görünerek başlar, "Bu eğrinin altındaki alan nedir?" Görünüşte alakasız olsa da, integral alma, *kalkülüsün temel teoremi* olarak bilinen ilişki aracılığıyla türev alma ile sıkı bir şekilde iç içe geçmiştir.

Bu kitapta tartıştığımız makine öğrenmesi düzeyinde, derin bir türev alma anlayışına ihtiyacımız olmayacak. Gene de, daha sonra karşılaşacağımız diğer uygulamalara zemin hazırlamak için kısa bir giriş sağlayacağız.

18.5.1 Geometrik Yorum

Bir $f(x)$ fonksiyonumuz olduğunu varsayalım. Basit olması için, $f(x)$'nin negatif olmadığını varsayalım (asla sıfırdan küçük bir değer almıyor). Denemek ve anlamak istediğimiz şudur: $f(x)$ ile x ekseni arasındaki alan nedir?

```
%matplotlib inline
import torch
from IPython import display
from mpl_toolkits import mplot3d
from d2l import torch as d2l

x = torch.arange(-2, 2, 0.01)
f = torch.exp(-x**2)

d2l.set_figsize()
d2l.plt.plot(x, f, color='black')
d2l.plt.fill_between(x.tolist(), f.tolist())
d2l.plt.show()
```



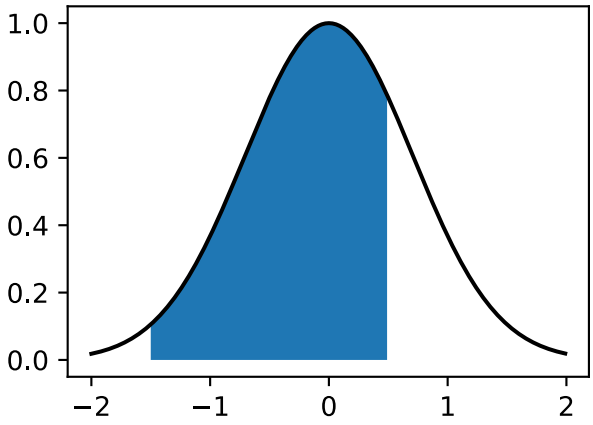
Çoğu durumda, bu alan sonsuz veya tanımsız olacaktır ($f(x) = x^2$ altındaki alanı düşünün), bu nedenle insanlar genellikle bir çift uç arasındaki alan hakkında konuşacaklardır, örneğin a ve b.

```
x = torch.arange(-2, 2, 0.01)
f = torch.exp(-x**2)
```

(continues on next page)

(continued from previous page)

```
d2l.set_figsize()
d2l.plt.plot(x, f, color='black')
d2l.plt.fill_between(x.tolist()[50:250], f.tolist()[50:250])
d2l.plt.show()
```



Bu alanı aşağıdaki integral (tümlev) sembolü ile göstereceğiz:

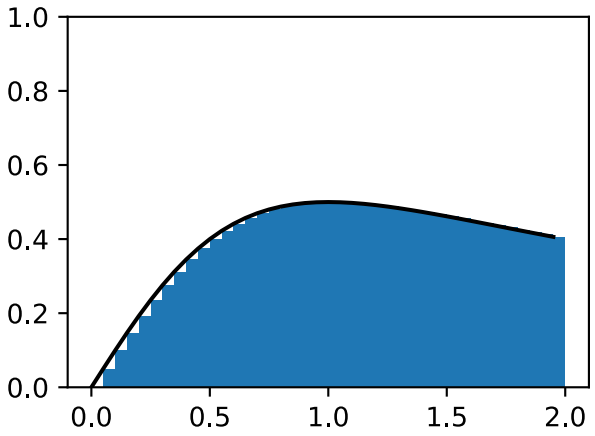
$$\text{Alan}(\mathcal{A}) = \int_a^b f(x)\ dx. \tag{18.5.1}$$

İç değişken, bir $\sum$ içindeki bir toplamın indisine çok benzeyen bir yapay değişkendir ve bu nedenle bu, istediğimiz herhangi bir iç değerle eşdeğer olarak yazılabilir:

$$\int_a^b f(x)\ dx = \int_a^b f(z)\ dz. \tag{18.5.2}$$

Bu tür integralleri nasıl yaklaşık olarak tahmin etmeye çalışabileceğimizin ve anlayabileceğimizin geleneksel bir yolu var: a ile b arasındaki bölgeyi alıp onu N dikey dilimlere böldüğümüzü hayal edebiliriz. N büyükse, her dilimin alanını bir dikdörtgenle tahmin edebilir ve ardından eğrinin altındaki toplam alanı elde etmek için alanları toplayabiliriz. Bunu kodla yapan bir örneğe bakalım. Gerçek değeri nasıl elde edeceğimizi daha sonraki bir bölümde göreceğiz.

```
epsilon = 0.05
a = 0
b = 2

x = torch.arange(a, b, epsilon)
f = x / (1 + x**2)

approx = torch.sum(epsilon*f)
true = torch.log(torch.tensor([5.])) / 2

d2l.set_figsize()
d2l.plt.bar(x, f, width=epsilon, align='edge')
d2l.plt.plot(x, f, color='black')
d2l.plt.ylim([0, 1])
d2l.plt.show()

f'approximation: {approx}, truth: {true}'
```




```
'approximation: 0.7944855690002441, truth: tensor([0.8047])'
```

Sorun şu ki, sayısal (numerik) olarak yapılabilse de, bu yaklaşımı analitik olarak sadece aşağıdaki gibi en basit işlevler için yapabiliriz.

$$\int_a^b x \, dx. \tag{18.5.3}$$

Yukarıdaki koddaki örneğimizden biraz daha karmaşık bir şey, mesela:

$$\int_a^b \frac{x}{1+x^2} \, dx. \tag{18.5.4}$$

böyle doğrudan bir yöntemle çözebileceğimizin ötesinde bir örnektir.

Bunun yerine farklı bir yaklaşım benimseyeceğiz. Alan kavramıyla sezgisel olarak çalışacağız ve integralleri bulmak için kullanılan ana hesaplama aracını öğreneceğiz: *Kalkülüsün temel teoremi*. Bu, integral alma (tümleme) çalışmamızın temeli olacaktır.

18.5.2 Kalkülüsün Temel Teoremi

Integral alma teorisinin derinliklerine inmek için bir fonksiyon tanıtalım:

$$F(x) = \int_0^x f(y)dy. \tag{18.5.5}$$

Bu işlev, x'i nasıl değiştirdiğimize bağlı olarak 0 ile x arasındaki alanı ölçer. İhtiyacımız olan her şeyin burada olduğuna dikkat edin:

$$\int_a^b f(x) \, dx = F(b) - F(a). \tag{18.5.6}$$

Bu, Fig. 18.5.1 içindeki gibi alanı en uzak uç noktaya kadar ölçebildiğimiz ve ardından yakın uç noktaya kadarki alanı çıkarabileceğimiz gerçeğinin matematiksel bir kodlamasıdır.

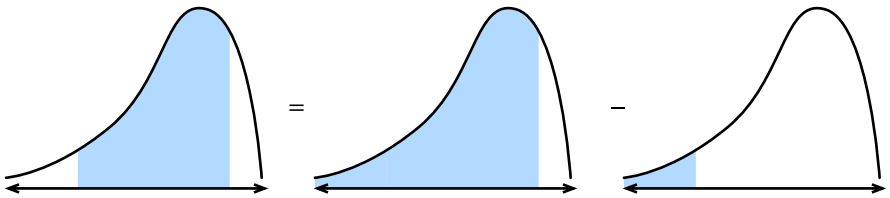


Fig. 18.5.1: İki nokta arasındaki bir eğrinin altındaki alanı hesaplama sorununu bir noktanın solundaki alanı hesaplamaya neden indirgeyeceğimizi görselleştirelim.

Dolayısıyla, herhangi bir aralıktaki integralin ne olduğunu $F(x)$'in ne olduğunu bularak bulabiliriz.

Bunun için bir deney yapalım. Kalkülüste sık sık yaptığımız gibi, değeri çok az değiştirdiğimizde ne olduğunu görelim. Yukarıdaki yorumdan biliyoruz ki

$$F(x+\epsilon) - F(x) = \int_x^{x+\epsilon} f(y)\ dy. \tag{18.5.7}$$

Bu bize, fonksiyonun küçük bir fonksiyon şeridinin altındaki alana göre değiştiğini söyler.

Bu, bir yaklaşıklama yaptığımız noktadır. Bunun gibi küçük bir alana bakarsak, bu alan, yüksekliği $f(x)$ ve taban genişliği ϵ olan dikdörtgenin alanına yakın görünüyor. Aslında, $\epsilon \to 0$ olunca bu yaklaşıklamanın gittikçe daha iyi hale geldiği gösterilebilir. Böylece şu sonuca varabiliriz:

$$F(x+\epsilon) - F(x) \approx \epsilon f(x). \tag{18.5.8}$$

Bununla birlikte, şimdi fark edebiliriz ki F'nin türevini hesaplıyor olsaydık, tam olarak beklediğimiz kalıp bu olurdu! Böylece şu oldukça şaşırtıcı gerçeği görüyoruz:

$$\frac{dF}{dx}(x) = f(x). \tag{18.5.9}$$

Bu, *kalkülüsün temel teoremidir*. Bunu genişletilmiş biçimde şöyle yazabiliriz:

$$\frac{d}{dx}\int_{-\infty}^{x} f(y)\ dy = f(x). \tag{18.5.10}$$

Alan bulma kavramını alır (ki muhtemelen oldukça zordur) ve onu bir ifade türevine (çok daha anlaşılmış bir şey) indirger. Yapmamız gereken son bir yorum, bunun bize $F(x)$'nin tam olarak ne olduğunu söylemediğidir. Aslında, herhangi bir C için $F(x) + C$ aynı türeve sahiptir. Bu, integral alma teorisinde hayatın gerçeğidir. Belirli integrallerle çalışırken sabitlerin düştüğüne ve dolayısıyla sonuçla ilgisiz olduklarına dikkat edin.

$$\int_a^b f(x)\ dx = (F(b) + C) - (F(a) + C) = F(b) - F(a). \tag{18.5.11}$$

Bu soyut olarak bir anlam ifade etmiyor gibi görünebilir, ancak bunun bizde integral hesaplamaya yepyeni bir bakış açısı kazandırdığını anlamak için biraz zaman ayıralım. Amacımız artık bir çeşit parçala ve topla işlemi yapmak ve alanı geri kurtarmayı denemek değil; bunun yerine sadece türevi sahip olduğumuz fonksiyon olan başka bir fonksiyon bulmamız gerekiyor! Bu inanılmaz bir olay çünkü artık pek çok zorlu integrali sadece Section 18.3.2 içindeki tabloyu ters çevirerek listeleyebiliriz. Örneğin, x^n'nin türevinin nx^{n-1} olduğunu biliyoruz. Böylece, temel teoremi, (18.5.10), kullanarak şunu söyleyebiliriz:

$$\int_0^x ny^{n-1}\ dy = x^n - 0^n = x^n. \tag{18.5.12}$$

Benzer şekilde, e^x'nin türevinin kendisi olduğunu biliyoruz, yani

$$\int_0^x e^x \, dx = e^x - e^0 = e^x - 1. \tag{18.5.13}$$

Bu şekilde, türevsel hesaptan gelen fikirlerden özgürce yararlanarak bütün integral alma teorisini geliştirebiliriz. Her integral kuralı bu olgudan türetilir.

18.5.3 Değişkenlerin Değişimi

Aynen türev almada olduğu gibi, integrallerin hesaplanmasını daha izlenebilir kılan bir dizi kural vardır. Aslında, türevsel hesabın her kuralı (çarpım kuralı, toplam kuralı ve zincir kuralı gibi), integral hesaplamada karşılık gelen bir kurala sahiptir (formül sırasıyla, parçalayarak integral alma, integralin doğrusallığı ve değişkenler değişimi). Bu bölümde, listeden tartışmasız en önemli olanı inceleyeceğiz: değişkenlerin değişimi formülü.

İlk olarak, kendisi bir integral olan bir fonksiyonumuz olduğunu varsayalım:

$$F(x) = \int_0^x f(y) \, dy. \tag{18.5.14}$$

Diyelim ki, $F(u(x))$'i elde etmek için onu başka bir fonksiyonla oluşturduğumuzda bu fonksiyonun nasıl göründüğünü bilmek istediğimizi varsayalım. Zincir kuralı ile biliyoruz ki:

$$\frac{d}{dx}F(u(x)) = \frac{dF}{du}(u(x)) \cdot \frac{du}{dx}. \tag{18.5.15}$$

Yukarıdaki gibi, temel teoremi, (18.5.10), kullanarak bunu integral alma ilgili bir ifadeye dönüştürebiliriz. Böylece:

$$F(u(x)) - F(u(0)) = \int_0^x \frac{dF}{du}(u(y)) \cdot \frac{du}{dy} \, dy. \tag{18.5.16}$$

F'nin kendisinin bir integral olduğunu hatırlamak, sol tarafın yeniden yazılabilmesine olanak verir.

$$\int_{u(0)}^{u(x)} f(y) \, dy = \int_0^x \frac{dF}{du}(u(y)) \cdot \frac{du}{dy} \, dy. \tag{18.5.17}$$

Benzer şekilde, F'nin bir integral olduğunu hatırlamak, temel teoremi, (18.5.10), kullanarak $\frac{dF}{dx} = f$'i tanımamıza izin verir ve böylece şu sonuca varabiliriz:

$$\int_{u(0)}^{u(x)} f(y) \, dy = \int_0^x f(u(y)) \cdot \frac{du}{dy} \, dy. \tag{18.5.18}$$

Bu, *değişkenlerin değişimi* formülüdür.

Daha sezgisel bir türetme için, x ile $x + \epsilon$ arasında bir $f(u(x))$ integralini aldığımızda ne olacağını düşünün. Küçük bir ϵ için, bu integral, yaklaşık olarak $\epsilon f(u(x))$, yani ilişkili dikdörtgenin alanıdır. Şimdi bunu $u(x)$ ile $u(x + \epsilon)$ arasındaki $f(y)$ integraliyle karşılaştıralım. $u(x + \epsilon) \approx u(x) + \epsilon \frac{du}{dx}(x)$ olduğunu biliyoruz, bu nedenle bu dikdörtgenin alanı yaklaşık $\epsilon \frac{du}{dx}(x) f(u(x))$'dir. Bu nedenle, bu iki dikdörtgenin alanını uyuşacak hale getirmek için Fig. 18.5.2 içinde gösterildiği gibi ilkini $\frac{du}{dx}(x)$ ile çarpmamız gerekiyor.

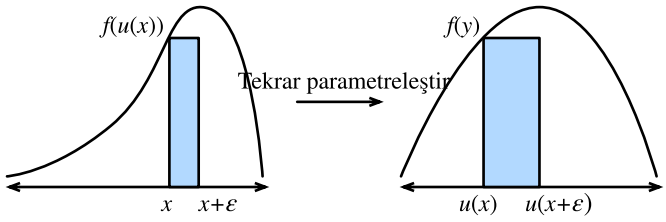


Fig. 18.5.2: Değişkenlerin değişimi altında tek bir ince dikdörtgenin dönüşümünü görselleştirelim.

Bu bize şunu söyler:

$$\int_{x}^{x+\epsilon} f(u(y))\frac{du}{dy}(y)\,dy = \int_{u(x)}^{u(x+\epsilon)} f(y)\,dy. \tag{18.5.19}$$

Bu, tek bir küçük dikdörtgen için ifade edilen değişken değişimi formülüdür.

$u(x)$ ve $f(x)$ doğru seçilirse, bu inanılmaz derecede karmaşık integrallerin hesaplanmasına izin verebilir. Örneğin, $f(y) = 1$ ve $u(x) = e^{-x^2}$ seçersek (bu $\frac{du}{dx}(x) = -2xe^{-x^2}$ anlamına gelir), bu örnek şunu gösterebilir:

$$e^{-1} - 1 = \int_{e^{-0}}^{e^{-1}} 1\,dy = -2\int_{0}^{1} ye^{-y^2}\,dy, \tag{18.5.20}$$

Tekrar düzenlersek:

$$\int_{0}^{1} ye^{-y^2}\,dy = \frac{1-e^{-1}}{2}. \tag{18.5.21}$$

18.5.4 İşaret Gösterimlerine Bir Yorum

Keskin gözlü okuyucular yukarıdaki hesaplamalarla ilgili tuhaf bir şey gözlemleyecekler. Yani, aşağıdaki gibi hesaplamalar

$$\int_{e^{-0}}^{e^{-1}} 1\,dy = e^{-1} - 1 < 0, \tag{18.5.22}$$

negatif sayılar üretebilirler. Alanlar hakkında düşünürken, negatif bir değer görmek garip olabilir ve bu nedenle bu gösterimin ne olduğunu araştırmaya değer.

Matematikçiler işaretli alanlar kavramını benimser. Bu kendini iki şekilde gösterir. İlk olarak, bazen sıfırdan küçük olan $f(x)$ fonksiyonunu düşünürsek, alan da negatif olacaktır. Yani örneğin

$$\int_{0}^{1} (-1)\,dx = -1. \tag{18.5.23}$$

Benzer şekilde, soldan sağa değil sağdan sola ilerleyen integraller de negatif alan olarak tanımlanır.

$$\int_{0}^{-1} 1\,dx = -1. \tag{18.5.24}$$

Standart alan (pozitif bir fonksiyonda soldan sağa) her zaman pozitiftir. Ters çevirerek elde edilen herhangi bir şey (örneğin, negatif bir sayının integralini elde etmek için x eksenini ters çevirmek

veya bir integrali yanlış sırada elde etmek için y eksenini ters çevirmek gibi) negatif bir alan üretecektir. Aslında, iki kez ters yüz etme, pozitif alana sahip olmak için birbirini götüren bir çift negatif işaret verecektir.

$$\int_0^{-1} (-1)\,dx = 1. \tag{18.5.25}$$

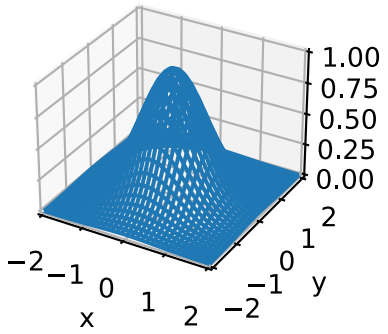
Bu tartışma size tanıdık geliyorsa, normaldir! Section 18.1 içinde determinantın işaretli alanı nasıl aynı şekilde temsil ettiğini tartıştık.

18.5.5 Çoklu İntegraller

Bazı durumlarda daha yüksek boyutlarda çalışmamız gerekecektir. Örneğin, $f(x, y)$ gibi iki değişkenli bir fonksiyonumuz olduğunu ve x'nin $[a, b]$ ve y'nin ise $[c, d]$ arasında değiştiğinde f altındaki hacim nedir bilmek istediğimizi varsayalım.

```
# Izgara oluştur ve işlevi hesapla
x, y = torch.meshgrid(torch.linspace(-2, 2, 101), torch.linspace(-2, 2, 101))
z = torch.exp(- x**2 - y**2)

# İşlevi çiz
ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x, y, z)
d2l.plt.xlabel('x')
d2l.plt.ylabel('y')
d2l.plt.xticks([-2, -1, 0, 1, 2])
d2l.plt.yticks([-2, -1, 0, 1, 2])
d2l.set_figsize()
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
ax.set_zlim(0, 1)
ax.dist = 12
```

```
/home/d2l-worker/miniconda3/envs/d2l-tr-release-0/lib/python3.9/site-packages/torch/
↪functional.py:478: UserWarning: torch.meshgrid: in an upcoming release, it will be␣
↪required to pass the indexing argument. (Triggered internally at  ../aten/src/ATen/native/
↪TensorShape.cpp:2895.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
```



Şöyle yazabiliriz:

$$\int_{[a,b]\times[c,d]} f(x,y)\ dx\ dy. \tag{18.5.26}$$

Bu integrali hesaplamak istediğimizi varsayalım. Bizim iddiamız, bunu önce x'deki integrali yinelemeli olarak hesaplayarak ve sonra da y'deki integrale kayarak yapabileceğimizdir, yani

$$\int_{[a,b]\times[c,d]} f(x,y)\ dx\ dy = \int_c^d \left(\int_a^b f(x,y)\ dx\right)\ dy. \tag{18.5.27}$$

Bunun neden olduğunu görelim.

Fonksiyonu $\epsilon \times \epsilon$ karelere böldüğümüzü ve i, j tamsayı koordinatlarıyla indekslediğimizi düşünün. Bu durumda integralimiz yaklaşık olarak şöyledir:

$$\sum_{i,j} \epsilon^2 f(\epsilon i, \epsilon j). \tag{18.5.28}$$

Problemi ayrıklaştırdığımızda, bu karelerdeki değerleri istediğimiz sırayla toplayabiliriz ve değerleri değiştirme konusunda endişelenmeyiz. Bu, Fig. 18.5.3 şeklinde gösterilmektedir. Özellikle şunu söyleyebiliriz:

$$\sum_j \epsilon \left(\sum_i \epsilon f(\epsilon i, \epsilon j)\right). \tag{18.5.29}$$

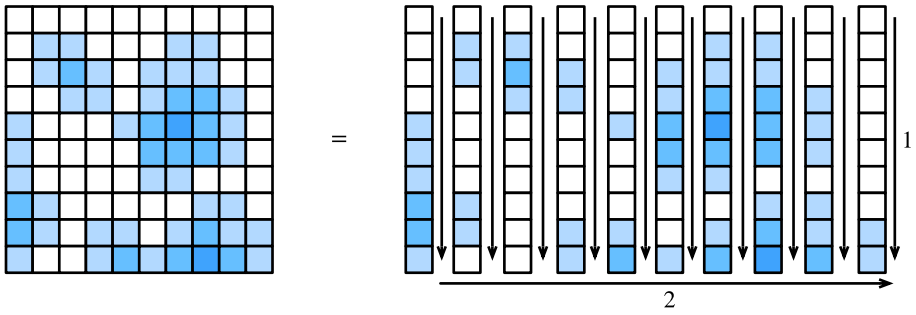


Fig. 18.5.3: İlk önce sütunlar üzerinden bir toplam olarak birçok karede bir toplamın nasıl ayrıştırılacağını (1), ardından sütun toplamlarının nasıl toplanacağını (2) gösterelim.

İç kısımdaki toplam, tam olarak integralin ayrıklaştırılmasıdır.

$$G(\epsilon j) = \int_a^b f(x, \epsilon j)\ dx. \tag{18.5.30}$$

Son olarak, bu iki ifadeyi birleştirirsek şunu elde ettiğimize dikkat edin:

$$\sum_j \epsilon G(\epsilon j) \approx \int_c^d G(y)\ dy = \int_{[a,b]\times[c,d]} f(x,y)\ dx\ dy. \tag{18.5.31}$$

Böylece hepsini bir araya getirirsek, buna sahip oluruz:

$$\int_{[a,b]\times[c,d]} f(x,y)\ dx\ dy = \int_c^d \left(\int_a^b f(x,y)\ dx\right)\ dy. \tag{18.5.32}$$

Dikkat edin, bir kez ayrıklaştırıldı mı, yaptığımız tek şey, bir sayı listesi eklediğimiz sırayı yeniden düzenlemek oldu. Bu, burada hiçbir zorluk yokmuş gibi görünmesine neden olabilir, ancak bu sonuç (*Fubini Teoremi* olarak adlandırılır) her zaman doğru değildir! Makine öğrenmesi (sürekli fonksiyonlar) yapılırken karşılaşılan matematik türü için herhangi bir endişe yoktur, ancak başarısız olduğu durumlardan örnekler oluşturmak mümkündür (örneğin $f(x, y) = xy(x^2 - y^2)/(x^2 + y^2)^3$ fonksiyonunu $[0, 2] \times [0, 1]$ dikdörtgenin üzerinde deneyin).

İntegrali önce x cinsinden ve ardından y cinsinden yapma seçeneğinin keyfi olduğuna dikkat edin. Önce y için, sonra da x için yapmayı eşit derecede seçebilirdik:

$$\int_{[a,b]\times[c,d]} f(x, y)\, dx\, dy = \int_a^b \left(\int_c^d f(x, y)\, dy \right) dx. \tag{18.5.33}$$

Çoğu zaman, vektör gösterimine yoğunlaşacağız ve $U = [a, b] \times [c, d]$ için bunun şöyle olduğunu söyleyeceğiz:

$$\int_U f(\mathbf{x})\, d\mathbf{x}. \tag{18.5.34}$$

18.5.6 Çoklu İntegrallerde Değişkenlerin Değiştirilmesi

Tek değişkenlerdeki gibi (18.5.18), daha yüksek boyutlu bir integral içindeki değişkenleri değiştirme yeteneği önemli bir araçtır. Sonucu türetme yapmadan özetleyelim.

Integral alma alanımızı yeniden parametrelendiren bir işleve ihtiyacımız var. Bunu $\phi : \mathbb{R}^n \to \mathbb{R}^n$ olarak alabiliriz, bu n tane gerçel değişkeni alıp başka bir n gerçele döndüren herhangi bir işlevdir. İfadeleri temiz tutmak için, ϕ'nin *bire-bir* olduğunu varsayacağız, bu da onun hiçbir zaman kendi üzerine katlanmadığını söylemektir ($\phi(\mathbf{x}) = \phi(\mathbf{y}) \implies \mathbf{x} = \mathbf{y}$).

Bu durumda şunu söyleyebiliriz:

$$\int_{\phi(U)} f(\mathbf{x})\, d\mathbf{x} = \int_U f(\phi(\mathbf{x}))\, |\det(D\phi(\mathbf{x}))|\, d\mathbf{x}. \tag{18.5.35}$$

$D\phi$, ϕ'nin *Jacobi matrisi*dir, yani $\boldsymbol{\phi} = (\phi_1(x_1, \ldots, x_n), \ldots, \phi_n(x_1, \ldots, x_n))$'nin kısmi türevlerinin matrisidir.

$$D\boldsymbol{\phi} = \begin{bmatrix} \frac{\partial \phi_1}{\partial x_1} & \cdots & \frac{\partial \phi_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial \phi_n}{\partial x_1} & \cdots & \frac{\partial \phi_n}{\partial x_n} \end{bmatrix}. \tag{18.5.36}$$

Yakından baktığımızda, bunun tek değişkenli zincir kuralına, (18.5.18), benzer olduğunu görüyoruz, ancak $\frac{du}{dx}(x)$ terimini $|\det(D\phi(\mathbf{x}))|$ ile değiştirdik. Bu terimi nasıl yorumlayabileceğimize bir bakalım. $\frac{du}{dx}(x)$ teriminin x eksenimizi u'yu uygulayarak ne kadar uzattığımızı söylemek için var olduğunu hatırlayın. Daha yüksek boyutlarda aynı işlem, $\boldsymbol{\phi}$ uygulayarak küçük bir karenin (veya küçük *hiper küpün*) alanını (veya hacmini veya hiper hacmini) ne kadar uzatacağımızı belirlemektir. Eğer $\boldsymbol{\phi}$ bir matrisle çarpımsa, determinantın zaten cevabı nasıl verdiğini biliyoruz.

Biraz çalışmayla, *Jacobi matrisi*nin çok değişkenli bir fonksiyona, yani $\boldsymbol{\phi}$'ye, bir noktada matrisin türevleri ve gradyanları olan doğrular veya düzlemlerle yaklaşık olarak tahmin edebileceğimiz gibi en iyi yaklaşıklamayı sağladığı gösterilebilir. Böylece, Jacobi matrisinin determinantı, bir boyutta tanımladığımız ölçeklendirme çarpanını tam olarak kopyalar.

Bunun ayrıntılarını doldurmak biraz çalışma gerektirir, bu yüzden şimdi net değilse endişelenmeyin. Daha sonra kullanacağımız bir örnek görelim. Bu integrali düşünün:

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} e^{-x^2-y^2} \, dx \, dy. \tag{18.5.37}$$

Bu integral ile doğrudan oynamak bizi hiçbir yere götürmez, ancak değişkenleri değiştirirsek, önemli ilerleme kaydedebiliriz. $\phi(r, \theta) = (r\cos(\theta), r\sin(\theta))$ olmasına izin verirsek (bu $x = r\cos(\theta)$, $y = r\sin(\theta)$ demektir), sonra bunun aynı şey olduğunu görmek için değişken değişimi formülünü uygulayabiliriz:

$$\int_0^{\infty} \int_0^{2\pi} e^{-r^2} \left|\det(D\phi(\mathbf{x}))\right| \, d\theta \, dr, \tag{18.5.38}$$

ve

$$\left|\det(D\phi(\mathbf{x}))\right| = \left|\det \begin{bmatrix} \cos(\theta) & -r\sin(\theta) \\ \sin(\theta) & r\cos(\theta) \end{bmatrix}\right| = r(\cos^2(\theta) + \sin^2(\theta)) = r. \tag{18.5.39}$$

Böylece integral:

$$\int_0^{\infty} \int_0^{2\pi} re^{-r^2} \, d\theta \, dr = 2\pi \int_0^{\infty} re^{-r^2} \, dr = \pi, \tag{18.5.40}$$

Burada son eşitlik, bölüm Section 18.5.3 içinde kullandığımız hesaplamanın aynısını izler.

Sürekli rastgele değişkenleri incelediğimizde, Section 18.6, bu integral ile tekrar karşılaşacağız.

18.5.7 Özet

- Integral alma teorisi, alanlar veya hacimler hakkındaki soruları yanıtlamamıza izin verir.
- Kalkülüsün temel teoremi, alanın türevinin bir noktaya kadar integrali alınan fonksiyonun değeri tarafından verildiği gözlemi sayesinde alanları hesaplamak için türevler hakkındaki bilgiden yararlanmamızı sağlar.
- Daha yüksek boyutlardaki integraller, tek değişkenli integralleri sırasıyla izleyerek hesaplanabilir.

18.5.8 Alıştırmalar

1. $\int_1^2 \frac{1}{x} \, dx$ nedir?
2. $\int_0^{\sqrt{\pi}} x\sin(x^2) \, dx$ ifadesini hesaplamak için değişken değişimi formülünü kullanınız.
3. $\int_{[0,1]^2} xy \, dx \, dy$ nedir?
4. $\int_0^2 \int_0^1 xy(x^2-y^2)/(x^2+y^2)^3 \, dy \, dx$ ve $\int_0^1 \int_0^2 f(x,y) = xy(x^2-y^2)/(x^2+y^2)^3 \, dx \, dy$ hesaplamak için değişken değişimi formülünü kullanınız ve farkı görünüz.

Tartışmalar[221]

[221] https://discuss.d2l.ai/t/1092

18.6 Rastgele Değişkenler

Section 2.6 içinde, bizim durumumuzda ya sonlu olası değerler kümesini ya da tamsayıları alan rastgele değişkenlere atıfta bulunan ayrık rastgele değişkenlerle nasıl çalışılacağının temellerini gördük. Bu bölümde, herhangi bir gerçel değeri alabilen rastgele değişkenler olan *sürekli rastgele değişkenler* teorisini geliştiriyoruz.

18.6.1 Sürekli Rastgele Değişkenler

Sürekli rastgele değişkenler, ayrık rastgele değişkenlerden önemli ölçüde daha incelikli bir konudur. Yapılması gereken adil bir benzetme, buradaki teknik sıçramanın sayı listeleri toplama ve işlevlerin integralini alma arasındaki sıçramayla karşılaştırılabilir olmasıdır. Bu nedenle, teoriyi geliştirmek için biraz zaman ayırmamız gerekecek.

Ayrık Değerden Sürekli Değere

Sürekli rastgele değişkenlerle çalışırken karşılaşılan ek teknik zorlukları anlamak için düşüncesel bir deney yapalım. Dart tahtasına bir dart fırlattığımızı ve tahtanın ortasından tam olarak 2cm uzağa saplanma olasılığını bilmek istediğimizi varsayalım.

Başlangıç olarak, tek basamaklı bir doğrulukla, yani 0cm, 1cm, 2cm gibi bölmeler kullanarak ölçtüğümüzü hayal ediyoruz. Dart tahtasına diyelim ki 100 dart atıyoruz ve eğer bunlardan 20'si 2cm bölmesine düşerse, attığımız dartların $\%20$'sinin tahtada merkezden 2cm uzağa saplandığı sonucuna varıyoruz.

Ancak daha yakından baktığımızda, bu sorumuzla örtüşmüyor! Tam eşitlik istiyorduk, oysa bu bölmeler diyelim ki 1.5cm ile 2.5cm arasındaki her şeyi tutuyor.

Kesintisiz, daha ileriye devam edelim. Daha da keskin bir şekilde ölçüyoruz, diyelim ki 1.9cm, 2.0cm, 2.1cm ve şimdi, belki de 100 darttan 3'ünün 2.0cm hattında tahtaya saplandığını görüyoruz. Böylece olasılığın $\%3$ olduğu sonucuna vardık.

Ancak bu hiçbir şeyi çözmez! Sorunu bir basamak daha aşağıya ittik. Biraz soyutlayalım. İlk k hanesinin $2.00000\ldots$ ile eşleşme olasılığını bildiğimizi ve ilk $k+1$ hanesi için eşleşme olasılığını bilmek istediğimizi düşünün. $k+1.$ basamağının aslında $\{0,1,2,\ldots,9\}$ kümesinden rastgele bir seçim olduğunu varsaymak oldukça mantıklıdır. En azından, mikrometre mertebesindeki değeri merkezden uzaklaşarak 7 veya 3'e karşılık gelmeye tercih etmeye zorlayacak fiziksel olarak anlamlı bir süreç düşünemiyoruz.

Bunun anlamı, özünde ihtiyaç duyduğumuz her ek doğruluk basamağının eşleştirme olasılığını 10'luk bir faktörle azaltması gerektiğidir. Ya da başka bir deyişle, bunu beklerdik:

$$P(\text{mesafe } 2.00\ldots, \text{'ye } k \text{ basamak yakın}) \approx p \cdot 10^{-k}. \tag{18.6.1}$$

Değer p esasen ilk birkaç basamakta olanları kodlar ve 10^{-k} gerisini halleder.

Ondalık sayıdan sonra konumu $k=4$ basamağa kadar doğru bildiğimize dikkat edin. Bu, değerin $[1.99995, 2.00005]$ aralığında olduğunu bildiğimiz anlamına gelir, bu da $2.00005 - 1.99995 = 10^{-4}$ uzunluğunda bir aralıktır. Dolayısıyla, bu aralığın uzunluğunu ϵ olarak adlandırırsak, diyebiliriz ki

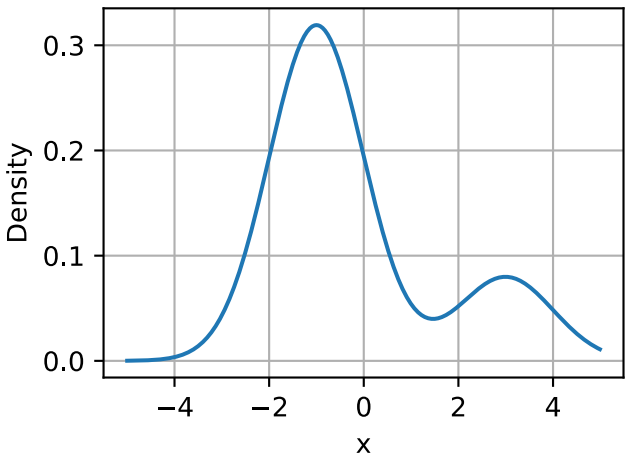
$$P(2\text{'ye } \epsilon\text{-ebatlı aralık mesafesinde}) \approx \epsilon \cdot p. \tag{18.6.2}$$

Bunu son bir adım daha ileri götürelim. Bunca zamandır 2 noktasını düşünüyorduk, ama diğer noktaları asla düşünmedik. Orada temelde hiçbir şey farklı değildir, ancak p değeri muhtemelen farklı olacaktır. En azından bir dart atıcısının, 20cm yerine 2cm gibi merkeze daha yakın bir noktayı vurma olasılığının daha yüksek olduğunu umuyoruz. Bu nedenle, p değeri sabit değildir, bunun yerine x noktasına bağlı olmalıdır. Bu da bize şunu beklememiz gerektiğini söylüyor:

$$P(x\text{'ye } \epsilon\text{-ebatlı aralık mesafesinde}) \approx \epsilon \cdot p(x). \tag{18.6.3}$$

Aslında, (18.6.3) tam olarak *olasılık yoğunluk fonksiyonunu* tanımlar. Bu, bir noktayı başka yakın bir noktaya göre vurma olasılığını kodlayan bir $p(x)$ fonksiyonudur. Böyle bir fonksiyonun neye benzeyebileceğini görselleştirelim.

```
%matplotlib inline
import torch
from IPython import display
from d2l import torch as d2l

torch.pi = torch.acos(torch.zeros(1)).item() * 2  # Pi'yi tanımla

# Bazı rastgele değişkenler için olasılık yoğunluk fonksiyonunu çiz
x = torch.arange(-5, 5, 0.01)
p = 0.2*torch.exp(-(x - 3)**2 / 2)/torch.sqrt(2 * torch.tensor(torch.pi)) + \
    0.8*torch.exp(-(x + 1)**2 / 2)/torch.sqrt(2 * torch.tensor(torch.pi))

d2l.plot(x, p, 'x', 'Density')
```



İşlev değerinin büyük olduğu konumlar, rastgele değeri bulma olasılığımızın daha yüksek olduğu bölgeleri gösterir. Düşük kısımlar, rastgele değeri bulamaya yatkın olmadığımız alanlardır.

Olasılık Yoğunluk Fonksiyonları

Şimdi bunu daha ayrıntılı inceleyelim. Bir rastgele değişken X için olasılık yoğunluk fonksiyonunun sezgisel olarak ne olduğunu daha önce görmüştük, yani yoğunluk fonksiyonu bir $p(x)$ fonksiyonudur.

$$P(X \text{ } x\text{'ye } \epsilon\text{-ebatlı aralık mesafesinde}) \approx \epsilon \cdot p(x). \tag{18.6.4}$$

Peki bu $p(x)$'nin özellikleri için ne anlama geliyor?

Birincisi, olasılıklar asla negatif değildir, dolayısıyla $p(x) \geq 0$ olmasını bekleriz.

İkinci olarak, $\mathbb{R}$'yi ϵ genişliğinde sonsuz sayıda dilime böldüğümüzü varsayalım, diyelim ki $(\epsilon \cdot i, \epsilon \cdot (i+1)]$ gibi. Bunların her biri için, (18.6.4) denkleminden biliyoruz, olasılık yaklaşık olarak

$$P(X \text{ } x\text{'ye } \epsilon\text{-ebatlı aralık mesafesinde}) \approx \epsilon \cdot p(\epsilon \cdot i), \tag{18.6.5}$$

bu yüzden hepsi üzerinden toplanabilmeli

$$P(X \in \mathbb{R}) \approx \sum_i \epsilon \cdot p(\epsilon \cdot i). \tag{18.6.6}$$

Bu, Section 18.5 içinde tartışılan bir integral yaklaşımından başka bir şey değildir, dolayısıyla şunu söyleyebiliriz:

$$P(X \in \mathbb{R}) = \int_{-\infty}^{\infty} p(x) \, dx. \tag{18.6.7}$$

$P(X \in \mathbb{R}) = 1$ olduğunu biliyoruz, çünkü rasgele değişkenin *herhangi bir* sayı alması gerektiğinden, herhangi bir yoğunluk için şu sonuca varabiliriz:

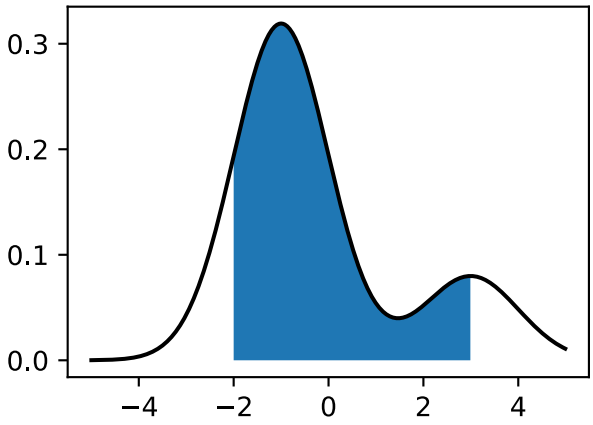
$$\int_{-\infty}^{\infty} p(x) \, dx = 1. \tag{18.6.8}$$

Aslında, bu konuyu daha ayrıntılı olarak incelersek, herhangi a ve b için şunu görürüz:

$$P(X \in (a, b]) = \int_a^b p(x) \, dx. \tag{18.6.9}$$

Bunu, daha önce olduğu gibi aynı ayrık yaklaşıklama yöntemlerini kullanarak kodda yaklaşıklaştırabiliriz. Bu durumda mavi bölgeye düşme olasılığını tahmin edebiliriz.

```
# Sayısal integral alma kullanarak olasılığı yaklaşıkla
epsilon = 0.01
x = torch.arange(-5, 5, 0.01)
p = 0.2*torch.exp(-(x - 3)**2 / 2) / torch.sqrt(2 * torch.tensor(torch.pi)) +\
    0.8*torch.exp(-(x + 1)**2 / 2) / torch.sqrt(2 * torch.tensor(torch.pi))

d2l.set_figsize()
d2l.plt.plot(x, p, color='black')
d2l.plt.fill_between(x.tolist()[300:800], p.tolist()[300:800])
d2l.plt.show()

f'approximate Probability: {torch.sum(epsilon*p[300:800])}'
```




```
'approximate Probability: 0.7736171483993530'
```

Bu iki özelliğin, olası olasılık yoğunluk fonksiyonlarının (veya yaygın olarak karşılaşılan kısaltma için *o.y.f.* *(p.d.f)*'ler) uzayını tastamam tanımladığı ortaya çıkar. Negatif olmayan fonksiyonlar $p(x) \geq 0$ için

$$\int_{-\infty}^{\infty} p(x)\ dx = 1. \tag{18.6.10}$$

Bu işlevi, rastgele değişkenimizin belirli bir aralıkta olma olasılığını elde etmek için integral alarak yorumluyoruz:

$$P(X \in (a, b]) = \int_{a}^{b} p(x)\ dx. \tag{18.6.11}$$

Section 18.8 içinde bir dizi yaygın dağılımı göreceğiz, ancak soyut olarak çalışmaya devam edelim.

Birikimli (Kümülatif) Dağılım Fonksiyonları

Önceki bölümde, o.y.f. kavramını gördük. Uygulamada, bu sürekli rastgele değişkenleri tartışmak için yaygın olarak karşılaşılan bir yöntemdir, ancak önemli bir görünmez tuzak vardır: o.y.f.'nin değerlerinin kendileri olasılıklar değil, olasılıkları elde etmek için integralini almamız gereken bir fonksiyondur. $1/10$'dan daha uzun bir aralık için 10'dan fazla olmadığı sürece, yoğunluğun 10'dan büyük olmasının yanlış bir tarafı yoktur. Bu sezgiye aykırı olabilir, bu nedenle insanlar genellikle *birikimli dağılım işlevi* veya *bir olasılık olan* b.d.f. açısından düşünürler.

Özellikle (18.6.11) kullanarak b.d.f'yi tanımlarız. $p(x)$ yoğunluğuna sahip rastgele bir değişken X için

$$F(x) = \int_{-\infty}^{x} p(x)\ dx = P(X \leq x). \tag{18.6.12}$$

Birkaç özelliği inceleyelim.

- $F(x) \to 0$ iken $x \to -\infty$.
- $F(x) \to 1$ iken $x \to \infty$.
- $F(x)$ azalmaz ($y > x \implies F(y) \geq F(x)$).

- X sürekli bir rasgele değişkense $F(x)$ süreklidir (sıçrama yoktur).

Dördüncü maddede, X ayrık olsaydı bunun doğru olmayacağına dikkat edin, mesela 0 ve 1 değerlerini $1/2$ olasılıkla alalım. Bu durumda

$$F(x) = \begin{cases} 0 & x < 0, \\ \frac{1}{2} & x < 1, \\ 1 & x \geq 1. \end{cases} \tag{18.6.13}$$

Bu örnekte, bdf ile çalışmanın faydalarından birini, aynı çerçevede sürekli veya ayrık rastgele değişkenlerle veya ötesi ikisinin karışımlarıyla başa çıkma becerisini görüyoruz (Yazı tura atın: Tura gerlirse zar atın, yazı gelirse bir dart atışının dart tahtasının merkezinden mesafesini ölçün).

Ortalama

Rastgele değişkenler, X, ile uğraştığımızı varsayalım. Dağılımın kendisini yorumlamak zor olabilir. Bir rastgele değişkenin davranışını kısaca özetleyebilmek genellikle yararlıdır. Rastgele bir değişkenin davranışını özetlememize yardımcı olan sayılara *özet istatistikleri* denir. En sık karşılaşılanlar *ortalama, varyans (değişinti)* ve *standart sapmadır*.

Ortalama, rastgele bir değişkenin ortalama değerini kodlar. p_i olasılıklarıyla x_i değerlerini alan ayrık bir rastgele değişkenimiz, X, varsa, ortalama, ağırlıklı ortalama ile verilir: Rastgele değişken değerlerinin bu değerleri alma olasılığıyla çarpımlarının toplamıdır:

$$\mu_X = E[X] = \sum_i x_i p_i. \tag{18.6.14}$$

Ortalamayı yorumlamamız gereken anlam (dikkatli olarak), bize rastgele değişkenin nerede bulunma eğiliminde olduğunu söylemesidir.

Bu bölümde inceleyeceğimiz minimalist bir örnek olarak, $a-2$ değerini p, $a+2$ değerini p ve a değerini $1-2p$ olasılıkla alan rastgele değişken olarak X'i alalım. (18.6.14) denklemini kullanarak, olası herhangi bir a ve p seçimi için ortalama hesaplayabiliriz:

$$\mu_X = E[X] = \sum_i x_i p_i = (a-2)p + a(1-2p) + (a+2)p = a. \tag{18.6.15}$$

Böylece ortalamanın a olduğunu görüyoruz. Rastgele değişkenimizi ortaladığımız konum a olduğundan, bu sezgimizle eşleşir.

Yararlı oldukları için birkaç özelliği özetleyelim.

- Herhangi bir rastgele değişken X ve a ve b sayıları için, $\mu_{aX+b} = a\mu_X + b$ olur.
- İki rastgele değişkenimiz varsa X ve Y, $\mu_{X+Y} = \mu_X + \mu_Y$ olur.

Ortalamalar, rastgele bir değişkenin ortalama davranışını anlamak için yararlıdır, ancak ortalama, tam bir sezgisel anlayışa sahip olmak için bile yeterli değildir. Satış başına $10\$ \pm 1\$$ kar etmek, aynı ortalama değere sahip olmasına rağmen satış başına $10\$ \pm 15\$$ kar etmekten çok farklıdır. İkincisi çok daha büyük bir dalgalanma derecesine sahiptir ve bu nedenle çok daha büyük bir riski temsil eder. Bu nedenle, rastgele bir değişkenin davranışını anlamak için en az bir ölçüye daha ihtiyacımız olacak: Bir rasgele değişkenin ne kadar geniş dalgalandığına dair bir ölçü.

Varyanslar

Bu bizi rastgele bir değişkenin *varyansını* düşünmeye götürür. Bu, rastgele bir değişkenin ortalamadan ne kadar saptığının nicel bir ölçüsüdür. $X - \mu_X$ ifadesini düşünün. Bu, rastgele değişkenin ortalamasından sapmasıdır. Bu değer pozitif ya da negatif olabilir, bu yüzden onu pozitif yapmak için bir şeyler yapmalıyız ki böylece sapmanın büyüklüğünü ölçebilelim.

Denenecek makul bir şey, $|X - \mu_X|$'a bakmaktır ve bu gerçekten de *ortalama mutlak sapma* olarak adlandırılan faydalı bir miktara yol açar, ancak diğer matematik ve istatistik alanlarıyla olan bağlantılarından dolayı, insanlar genellikle farklı bir çözüm kullanır.

Özellikle $(X - \mu_X)^2$'ye bakarlar. Bu miktarın tipik boyutuna ortalamasını alarak bakarsak, varyansa ulaşırız:

$$\sigma_X^2 = \text{Var}(X) = E\left[(X - \mu_X)^2\right] = E[X^2] - \mu_X^2. \tag{18.6.16}$$

(18.6.16) denklemindeki son eşitlik, ortadaki tanımı genişleterek ve beklentinin özelliklerini uygulayarak devam eder.

X'in p olasılıkla $a-2$ değerini, p olasılıkla $a+2$ ve $1-2p$ olasılıkla a değerini alan rastgele değişken olduğu örneğimize bakalım. Bu durumda $\mu_X = a$'dır, dolayısıyla hesaplamamız gereken tek şey $E\left[X^2\right]$'dir. Bu kolaylıkla yapılabilir:

$$E\left[X^2\right] = (a-2)^2 p + a^2(1-2p) + (a+2)^2 p = a^2 + 8p. \tag{18.6.17}$$

Böylece görürüz ki (18.6.16) tanımıyla varyansımız:

$$\sigma_X^2 = \text{Var}(X) = E[X^2] - \mu_X^2 = a^2 + 8p - a^2 = 8p. \tag{18.6.18}$$

Bu sonuç yine mantıklıdır. p en büyük $1/2$ olabilir ve bu da yazı tura ile $a-2$ veya $a+2$ seçmeye karşılık gelir. Bunun 4 olması, hem $a-2$ hem de $a+2$'nin ortalamadan 2 birim uzakta ve $2^2 = 4$ olduğu gerçeğine karşılık gelir. Spektrumun (izgenin) diğer ucunda, eğer $p = 0$ ise, bu rasgele değişken her zaman 0 değerini alır ve bu nedenle hiçbir varyansı yoktur.

Aşağıda varyansın birkaç özelliğini listeleyeceğiz:

- Herhangi bir rastgele değişken X için, $\text{Var}(X) \geq 0$, ancak ve ancak X bir sabitse $\text{Var}(X) = 0$'dır.
- Herhangi bir rastgele değişken X ve a ve b sayıları için, $\text{Var}(aX + b) = a^2\text{Var}(X)$'dır.
- İki *bağımsız* rastgele değişkenimiz varsa, X ve Y, $\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y)$'dir.

Bu değerleri yorumlarken biraz tutukluk olabilir. Özellikle, bu hesaplama yoluyla birimleri takip edersek ne olacağını hayal edelim. Web sayfasındaki bir ürüne atanan yıldız derecelendirmesiyle çalıştığımızı varsayalım. Daha sonra a, $a-2$ ve $a+2$ değerlerinin tümü yıldız birimleriyle ölçülür. Benzer şekilde, ortalama, μ_X, daha sonra yıldızlarla da ölçülür (ağırlıklı ortalama). Bununla birlikte, varyansa ulaşırsak, hemen bir sorunla karşılaşırız, bu da *yıldız kare* birimleri cinsinden $(X - \mu_X)^2$'ye bakmak istediğimizdendir. Bu, varyansın kendisinin orijinal ölçümlerle karşılaştırılamayacağı anlamına gelir. Bunu yorumlanabilir hale getirmek için orijinal birimlerimize dönmemiz gerekecek.

Standart Sapmalar

Bu özet istatistik, karekök alınarak varyanstan her zaman çıkarılabilir! Böylece *standart sapmayı* tanımlıyoruz:

$$\sigma_X = \sqrt{\text{Var}(X)}. \tag{18.6.19}$$

Örneğimizde bu, standart sapmanın $\sigma_X = 2\sqrt{2p}$ olduğu anlamına gelir. İnceleme örneğimiz için yıldız birimleriyle uğraşıyorsak, σ_X yine yıldız birimindedir.

Varyans için sahip olduğumuz özellikler, standart sapma için yeniden ifade edilebilir.

- Herhangi bir rastgele değişken X için , $\sigma_X \geq 0$'dır.
- Herhangi bir rastgele değişken X ve a ve b sayıları için, $\sigma_{aX+b} = |a|\sigma_X$'dır.
- İki *bağımsız* rastgele değişkenimiz, X ve Y, varsa, $\sigma_{X+Y} = \sqrt{\sigma_X^2 + \sigma_Y^2}$ olur.

Şu anda şunu sormak doğaldır, "Eğer standart sapma orijinal rasgele değişkenimizin birimlerindeyse, bu rasgele değişkenle ilgili olarak çizebileceğimiz bir şeyi temsil eder mi?" Cevap yankılanan bir evettir! Aslında ortalamanın bize rastgele değişkenimizin tipik konumunu söylediğine benzer şekilde, standart sapma o rastgele değişkenin tipik varyasyon aralığını verir. Bunu, Chebyshev eşitsizliği olarak bilinen şeyle sıkı hale getirebiliriz:

$$P\left(X \notin [\mu_X - \alpha\sigma_X, \mu_X + \alpha\sigma_X]\right) \leq \frac{1}{\alpha^2}. \tag{18.6.20}$$

Veya sözlü olarak ifade etmek gerekirse, $\alpha = 10$ durumunda, herhangi bir rasgele değişkenden alınan örneklerin %99'u, ortalamadan 10 standart sapmalık aralığa düşer. Bu, standart özet istatistiklerimize anında bir anlam sağlar.

Bu ifadenin ne kadar ince olduğunu görmek için, X'in p olasılıkla $a-2$, p olasılıkla $a+2$ ve $1-2p$ olasılıkla a değerini alan rastgele değişken olduğu, mevcut örneğimize tekrar bakalım. Ortalamanın a ve standart sapmanın $2\sqrt{2p}$ olduğunu gördük. Bu demektir ki, Chebyshev'in eşitsizliğini, (18.6.20), $\alpha = 2$ ile alırsak, ifadenin şöyle olduğunu görürüz:

$$P\left(X \notin [a - 4\sqrt{2p}, a + 4\sqrt{2p}]\right) \leq \frac{1}{4}. \tag{18.6.21}$$

Bu, zamanın %75'inde, bu rastgele değişkenin herhangi bir p değeri için bu aralığa denk geleceği anlamına gelir. Şimdi, $p \to 0$ olarak, bu aralığın da tek bir a noktasına yakınsadığına dikkat edin. Ancak rasgele değişkenimizin yalnızca $a-2, a$ ve $a+2$ değerlerini aldığını biliyoruz, bu nedenle sonunda $a-2$ ve $a+2$ aralığının dışında kalacağından emin olabiliriz! Soru, bunun hangi p'de olduğu. Bu yüzden bunu çözmek istiyoruz: Hangi p'de $a + 4\sqrt{2p} = a + 2$ yapar, ki bu $p = 1/8$ olduğunda çözülür, bu da dağılımdan $1/4$'ten fazla örneklemin aralığın dışında kalmayacağı iddiamızı ihlal etmeden gerçekleştirebilecek *tam olarak* ilk p'dir ($1/8$ sola ve $1/8$ sağa).

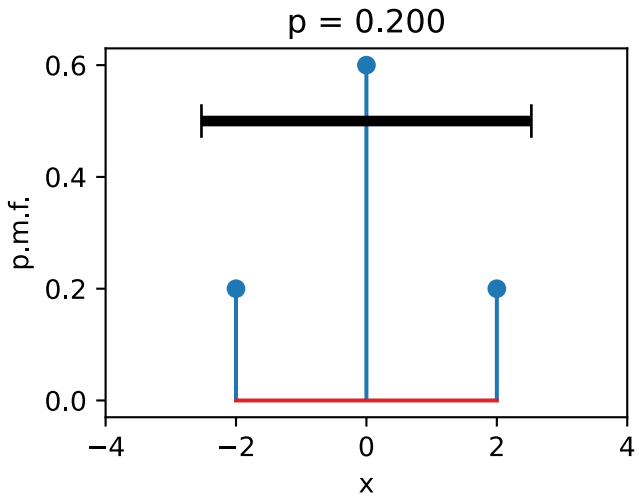
Bunu görselleştirelim. Üç değeri alma olasılığını olasılıkla orantılı yüksekliği olan üç dikey çubuk olarak göstereceğiz. Aralık ortada yatay bir çizgi olarak çizilecektir. İlk grafik, aralığın güvenli bir şekilde tüm noktaları içerdiği $p > 1/8$ için ne olduğunu gösterir.

```
# Bu rakamları çizmek için bir yardımcı tanımlayın
def plot_chebyshev(a, p):
    d2l.set_figsize()
    d2l.plt.stem([a-2, a, a+2], [p, 1-2*p, p], use_line_collection=True)
    d2l.plt.xlim([-4, 4])
```

(continues on next page)

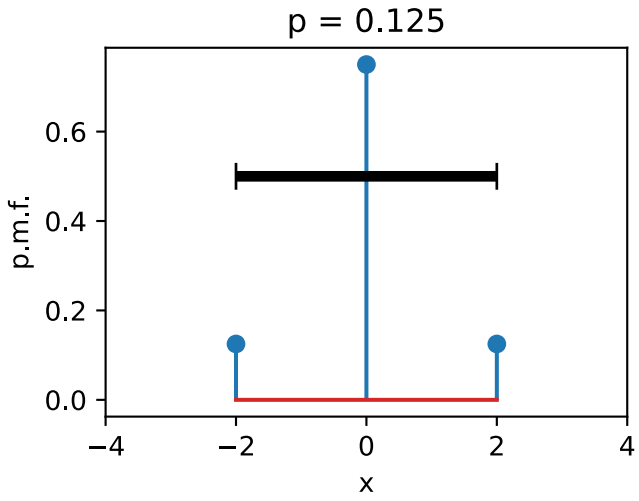
(continued from previous page)

```
    d2l.plt.xlabel('x')
    d2l.plt.ylabel('p.m.f.')

    d2l.plt.hlines(0.5, a - 4 * torch.sqrt(2 * p),
                   a + 4 * torch.sqrt(2 * p), 'black', lw=4)
    d2l.plt.vlines(a - 4 * torch.sqrt(2 * p), 0.53, 0.47, 'black', lw=1)
    d2l.plt.vlines(a + 4 * torch.sqrt(2 * p), 0.53, 0.47, 'black', lw=1)
    d2l.plt.title(f'p = {p:.3f}')

    d2l.plt.show()

# p > 1/8 olduğundaki aralığı çiz
plot_chebyshev(0.0, torch.tensor(0.2))
```



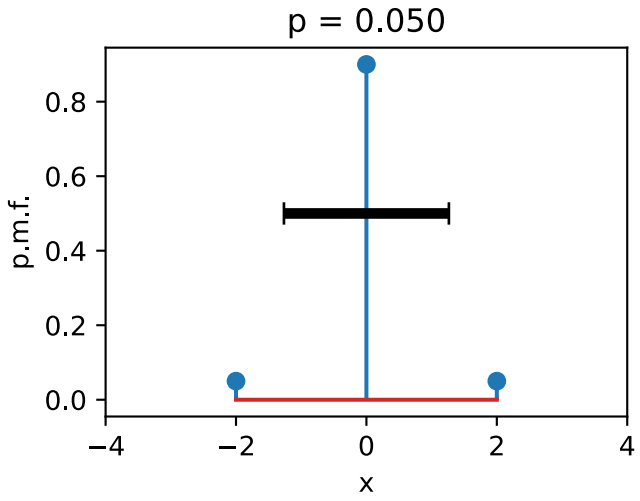
İkinci görsel, $p = 1/8$'de aralığın tam olarak iki noktaya dokunduğunu gösterir. Bu, eşitsizliğin doğru tutulurken daha küçük bir aralık alınamayacağı için eşitsizliğin *keskin* olduğunu gösterir.

```
# p = 1/8 olduğundaki aralığı çiz
plot_chebyshev(0.0, torch.tensor(0.125))
```



Üçüncüsü, $p < 1/8$ için aralığın yalnızca merkezi içerdiğini gösterir. Bu, eşitsizliği geçersiz kılmaz, çünkü yalnızca olasılığın $1/4$'ten fazlasının aralığın dışında kalmamasını sağlamamız gerekiyor, bu da bir kez $p < 1/8$ olduğunda,dır iki nokta $a - 2$ ve $a + 2$ yok sayılabilir.

```
# p < 1/8 olduğundaki aralığı çiz
plot_chebyshev(0.0, torch.tensor(0.05))
```



Süreklilikteki Ortalamalar ve Varyanslar

Bunların tümü ayrık rastgele değişkenler açısından olmuştur, ancak sürekli rastgele değişkenler durumu benzerdir. Bunun nasıl çalıştığını sezgisel olarak anlamak için, $(\epsilon i, \epsilon(i+1)]$ ile verilen gerçel sayı doğrusunu ϵ uzunluğundaki aralıklara böldüğümüzü hayal edin. Bunu yaptıktan sonra, sürekli rastgele değişkenimiz artık ayrık sayılır ve (18.6.14) kullanarak şunu söyleyebiliriz:

$$\begin{aligned} \mu_X &\approx \sum_i (\epsilon i) P(X \in (\epsilon i, \epsilon(i+1)]) \\ &\approx \sum_i (\epsilon i) p_X(\epsilon i) \epsilon, \end{aligned} \tag{18.6.22}$$

Burada p_X, X'in yoğunluğudur. Bu, $xp_X(x)$ integralinin yaklaşıklama bir değeridir, dolayısıyla şunu çıkarabiliriz:

$$\mu_X = \int_{-\infty}^{\infty} x p_X(x) \, dx. \tag{18.6.23}$$

Benzer şekilde (18.6.16) kullanılarak varyans şöyle yazılabilir:

$$\sigma_X^2 = E[X^2] - \mu_X^2 = \int_{-\infty}^{\infty} x^2 p_X(x) \, dx - \left(\int_{-\infty}^{\infty} x p_X(x) \, dx \right)^2. \tag{18.6.24}$$

Yukarıda ortalama, varyans ve standart sapma hakkında belirtilen her şey bu durumda hala geçerlidir. Örneğin, aşağıdaki yoğunluğa sahip rastgele değişkeni düşünelim:

$$p(x) = \begin{cases} 1 & x \in [0, 1], \\ 0 & \text{diğer türlü}. \end{cases} \tag{18.6.25}$$

Hesaplabiliriz ki:

$$\mu_X = \int_{-\infty}^{\infty} xp(x)\,dx = \int_0^1 x\,dx = \frac{1}{2}. \tag{18.6.26}$$

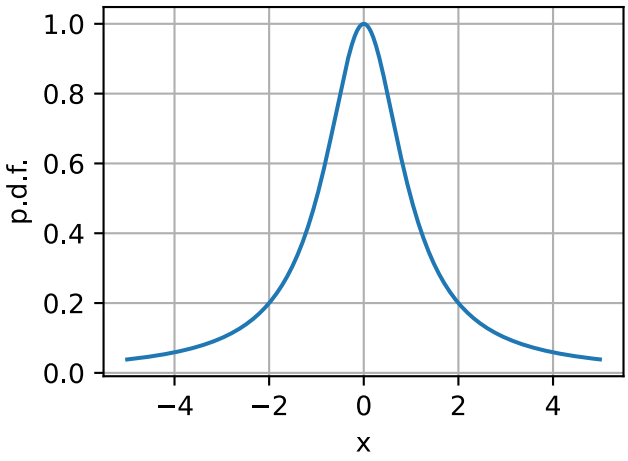
ve

$$\sigma_X^2 = \int_{-\infty}^{\infty} x^2 p(x)\,dx - \left(\frac{1}{2}\right)^2 = \frac{1}{3} - \frac{1}{4} = \frac{1}{12}. \tag{18.6.27}$$

Bir uyarı olarak, *Cauchy dağılımı* olarak bilinen bir örneği daha inceleyelim. Aşağıda o.y.f.'si verilen dağılımdır.

$$p(x) = \frac{1}{1+x^2}. \tag{18.6.28}$$

```
# Cauchy dağılımınının oyf'sini çiz
x = torch.arange(-5, 5, 0.01)
p = 1 / (1 + x**2)

d2l.plot(x, p, 'x', 'p.d.f.')
```



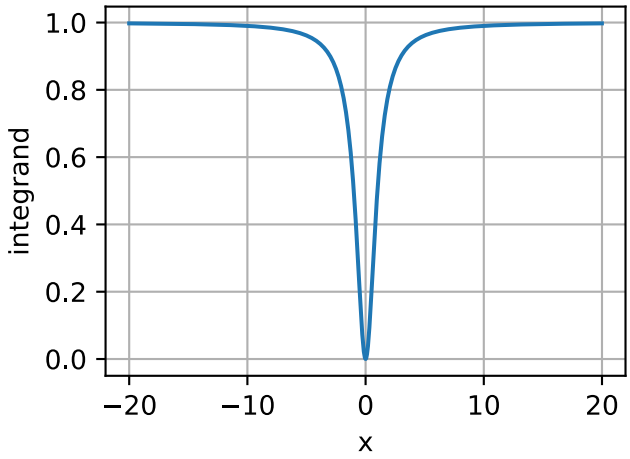
Bu fonksiyon masum görünür ve bir integral tablosuna başvurmak, onun altında birim alan olduğunu gösterecek ve böylece sürekli bir rastgele değişken tanımlayacaktır.

Neyin yanlış gittiğini görmek için, bunun varyansını hesaplamaya çalışalım. Hesaplama şunu kullanmayı içerir (18.6.16)

$$\int_{-\infty}^{\infty} \frac{x^2}{1+x^2}\,dx. \tag{18.6.29}$$

İç kısımdaki işlev şuna benzer:

```
# Varyansı hesaplamak için gereken integrali çiz
x = torch.arange(-20, 20, 0.01)
p = x**2 / (1 + x**2)

d2l.plot(x, p, 'x', 'integrand')
```



Bu fonksiyonun altında sonsuz bir alan vardır, çünkü esasen sıfırda çökerken diğer yerlerde bir sabittir ve bunu gösterebiliriz.

$$\int_{-\infty}^{\infty} \frac{x^2}{1+x^2}\,dx = \infty. \tag{18.6.30}$$

Bu, iyi tanımlanmış bir sonlu varyansa sahip olmadığı anlamına gelir.

Ancak daha derin bakmak daha da rahatsız edici bir sonuç gösterir. Ortalama değerini (18.6.14) kullanarak hesaplamaya çalışalım. Değişken değişiğimi formülünü kullanalım,

$$\mu_X = \int_{-\infty}^{\infty} \frac{x}{1+x^2}\,dx = \frac{1}{2}\int_{1}^{\infty} \frac{1}{u}\,du. \tag{18.6.31}$$

İçerideki integral, logaritmanın tanımıdır, dolayısıyla bu özünde $\log(\infty) = \infty$'dur, dolayısıyla iyi tanımlanmış bir ortalama değer de yoktur!

Makine öğrenmesi bilimcileri, modellerini, çoğu zaman bu sorunlarla uğraşmamıza gerek kalmayacak şekilde tanımlarlar, öyle ki vakaların büyük çoğunluğunda iyi tanımlanmış ortalamalara ve varyanslara sahip rastgele değişkenlerle ilgileneceğiz. Bununla birlikte, *ağır kuyruklu* rastgele değişkenler (yani, büyük değerler alma olasılıklarının ortalama veya varyans gibi şeyleri tanımlanmamış hale getirecek kadar büyük olduğu rastgele değişkenler) fiziksel sistemleri modellemede yardımcı olur, bu nedenle var olduklarını bilmeye değerdir.

Bileşik Yoğunluk İşlevleri

Yukarıdaki çalışmanın tümü, tek bir gerçel değerli rastgele değişkenle çalıştığımızı varsayar. Peki ya iki veya daha fazla potansiyel olarak yüksek düzeyde ilişkili rastgele değişkenle uğraşıyorsak? Bu durum, makine öğrenmesinde normaldir: Bir imgedeki (i, j) koordinatındaki pikselin kırmızı değerini kodlayan $R_{i,j}$ gibi rastgele değişkenler veya t zamanında hisse senedi fiyatı tarafından verilen rastgele değişken olan P_t. Yakındaki pikseller benzer renge sahip olma eğilimindedir ve yakın zamanlardakiler benzer fiyatlara sahip olma eğilimindedir. Bunları ayrı rastgele değişkenler olarak ele alamayız ve başarılı bir model oluşturmayı bekleyemeyiz (Section 18.9 içinde böyle bir varsayım nedeniyle düşük performans gösteren bir model göreceğiz). Bu ilişkili sürekli rastgele değişkenleri işlemek için matematik dili geliştirmemiz gerekiyor.

Neyse ki Section 18.5 içindeki çoklu integraller ile böyle bir dil geliştirebiliriz. Basitlik açısından, ilişkilendirilebilecek iki X, Y rastgele değişkenimiz olduğunu varsayalım. Sonra, tek bir değişken

durumunda olduğu gibi, şu soruyu sorabiliriz:

$$P(X \text{ } x\text{'ye } \epsilon\text{-ebatlı aralık mesafesindeve} Y \text{ } y\text{'ye } \epsilon\text{-ebatlı aralık mesafesinde}). \tag{18.6.32}$$

Tek değişkenli duruma benzer akıl yürütme, bunun yaklaşık olması gerektiğini gösterir:

$$P(X \text{ } x\text{'ye } \epsilon\text{-ebatlı aralık mesafesindeve} Y \text{ } y\text{'ye } \epsilon\text{-ebatlı aralık mesafesinde}) \approx \epsilon^2 p(x, y), \tag{18.6.33}$$

bu bazı $p(x, y)$ işlevleri içindir. Bu, X ve Y bileşik yoğunluğu olarak adlandırılır. Tek değişken durumunda gördüğümüz gibi benzer özellikler bunun için doğrudur. Yani:

- $p(x, y) \geq 0$;
- $\int_{\mathbb{R}^2} p(x, y) \, dx \, dy = 1$;
- $P((X, Y) \in \mathcal{D}) = \int_{\mathcal{D}} p(x, y) \, dx \, dy$.

Bu şekilde, birden çok, potansiyel olarak ilişkili rastgele değişkenlerle başa çıkabiliriz. İkiden fazla rastgele değişkenle çalışmak istersek, çok değişkenli yoğunluğu, $p(\mathbf{x}) = p(x_1, \ldots, x_n)$'yi dikkate alarak istediğimiz kadar çok koordinata genişletebiliriz. Negatif olmama ve birinin toplam integraline sahip olma gibi aynı özellikler hala geçerlidir.

Marjinal Dağılımlar

Birden çok değişkenle uğraşırken, çoğu zaman ilişkileri görmezden gelmek ve "Bu tek değişken nasıl dağıtılır?" diye sormak isteriz. Böyle bir dağılım, *marjinal dağılım* olarak adlandırılır.

Somut olmak gerekirse, bileşik yoğunluğu $p_{X,Y}(x, y)$ ile verilen iki rastgele değişkenimiz, X, Y, olduğunu varsayalım. Yoğunluğun hangi rastgele değişkenler için olduğunu belirtmek için alt indis kullanacağız. Marjinal dağılımı bulma sorusu bu işlevi alıp $p_X(x)$ bulmak için kullanmaktır.

Çoğu şeyde olduğu gibi, neyin doğru olması gerektiğini anlamak için sezgisel resmimize dönmek en iyisidir. Yoğunluğun p_X işlevi olduğunu hatırlayın, böylece

$$P(X \in [x, x + \epsilon]) \approx \epsilon \cdot p_X(x). \tag{18.6.34}$$

Y'den söz edilmiyor, ancak bize verilen tek şey $p_{X,Y}$ ise, bir şekilde Y eklememiz gerekir. İlk önce bunun aynı olduğunu gözlemleyelim:

$$P(X \in [x, x + \epsilon], \text{ ve } Y \in \mathbb{R}) \approx \epsilon \cdot p_X(x). \tag{18.6.35}$$

Yoğunluğumuz bize bu durumda ne olduğunu doğrudan söylemiyor, y cinsinden de küçük aralıklara bölmemiz gerekiyor, böylece bunu şu şekilde yazabiliriz:

$$\begin{aligned} \epsilon \cdot p_X(x) &\approx \sum_i P(X \in [x, x + \epsilon], \text{ ve } Y \in [\epsilon \cdot i, \epsilon \cdot (i + 1)]) \\ &\approx \sum_i \epsilon^2 p_{X,Y}(x, \epsilon \cdot i). \end{aligned} \tag{18.6.36}$$

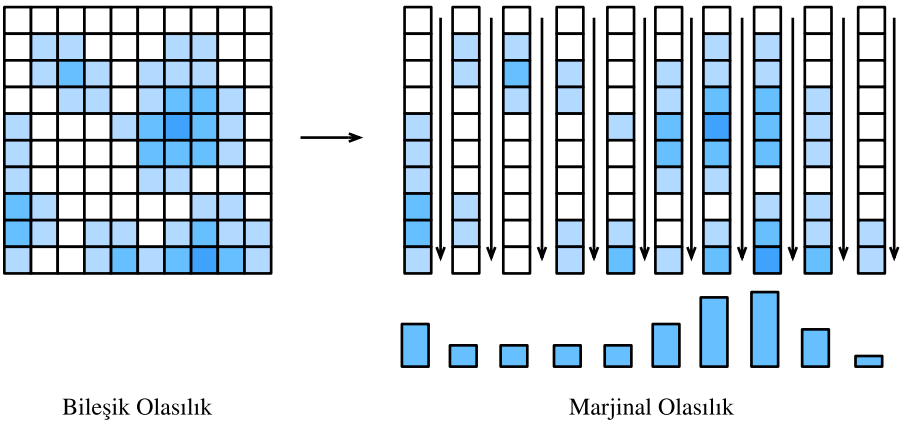


Fig. 18.6.1: Olasılık dizimizin sütunları boyunca toplayarak, sadece x ekseni boyunca temsil edilen rastgele değişken için marjinal dağılımı elde edebiliriz.

Bu bize, Fig. 18.6.1 şeklindeki gibi bir satırdaki bir dizi kare boyunca yoğunluk değerini toplamamızı söyler. Aslında, her iki taraftan bir epsilon çarpanı iptal ettikten ve sağdaki toplamın y'nin üzerindeki integral olduğunu gördükten sonra, şu sonuca varabiliriz:

$$\begin{aligned} p_X(x) &\approx \sum_i \epsilon p_{X,Y}(x, \epsilon \cdot i) \\ &\approx \int_{-\infty}^{\infty} p_{X,Y}(x, y)\ dy. \end{aligned} \tag{18.6.37}$$

Böylece görürüz ki:

$$p_X(x) = \int_{-\infty}^{\infty} p_{X,Y}(x, y)\ dy. \tag{18.6.38}$$

Bu bize, marjinal bir dağılım elde etmek için umursamadığımız değişkenlerin integralini almamızı söyler. Bu sürece genellikle gereksiz değişkenleri *integralle dışarı atma* veya *marjinalleştirme* adı verilir.

Kovaryans (Eşdeğişirlik)

Birden fazla rastgele değişkenle uğraşırken, bilinmesinin ileride yardımcı olacağı ek bir özet istatistik vardır: *Kovaryans*. Bu, iki rastgele değişkenin birlikte dalgalanma derecesini ölçer.

Başlamak için X ve Y olmak üzere iki rasgele değişkenimiz olduğunu varsayalım, bunların ayrık olduklarını varsayalım, p_{ij} olasılıkla (x_i, y_j) değerlerini alalım. Bu durumda kovaryans şu şekilde tanımlanır:

$$\sigma_{XY} = \text{Cov}(X, Y) = \sum_{i,j} (x_i - \mu_X)(y_j - \mu_Y) p_{ij}. = E[XY] - E[X]E[Y]. \tag{18.6.39}$$

Bunu sezgisel olarak düşünmek için: Aşağıdaki rastgele değişken çiftini düşünün. X'in 1 ve 3 değerlerini ve Y'nin -1 ve 3 değerlerini aldığını varsayalım. Aşağıdaki olasılıklara sahip olduğu-

muzu varsayalım:

$$\begin{aligned}
P(X = 1 \text{ ve } Y = -1) &= \frac{p}{2},\\
P(X = 1 \text{ ve } Y = 3) &= \frac{1-p}{2},\\
P(X = 3 \text{ ve } Y = -1) &= \frac{1-p}{2},\\
P(X = 3 \text{ ve } Y = 3) &= \frac{p}{2},
\end{aligned} \tag{18.6.40}$$

Burada p, $[0, 1]$ içinde bir parametredir. $p = 1$ ise her ikisi de her zaman aynı anda minimum veya maksimum değerlerdedir ve $p = 0$ ise çevrilmiş değerlerini aynı anda almaları garanti edilir (biri küçükken diğeri büyüktür). $p = 1/2$ ise, bu durumda dört olasılığın tümü eşit olasılıktadır ve ikisi de ilişkili olmamalıdır. Kovaryansı hesaplayalım. İlk olarak, $\mu_X = 2$ ve $\mu_Y = 1$ olduğuna not edin, böylece (18.6.39) kullanarak hesaplama yapabiliriz:

$$\begin{aligned}
\mathrm{Cov}(X, Y) &= \sum_{i,j} (x_i - \mu_X)(y_j - \mu_Y) p_{ij}\\
&= (1-2)(-1-1)\frac{p}{2} + (1-2)(3-1)\frac{1-p}{2} + (3-2)(-1-1)\frac{1-p}{2} + (3-2)(3-1)\frac{p}{2}\\
&= 4p - 2.
\end{aligned} \tag{18.6.41}$$

$p = 1$ olduğunda (her ikisinin de aynı anda maksimum pozitif veya negatif olduğu durum) 2'lik bir kovaryansa sahiptir. $p = 0$ olduğunda (çevrildikleri durum) kovaryans -2'dir. Son olarak, $p = 1/2$ (ilgisiz oldukları durum) olduğunda kovaryans 0'dır. Böylece kovaryansın bu iki rastgele değişkenin nasıl ilişkili olduğunu ölçtüğünü görüyoruz.

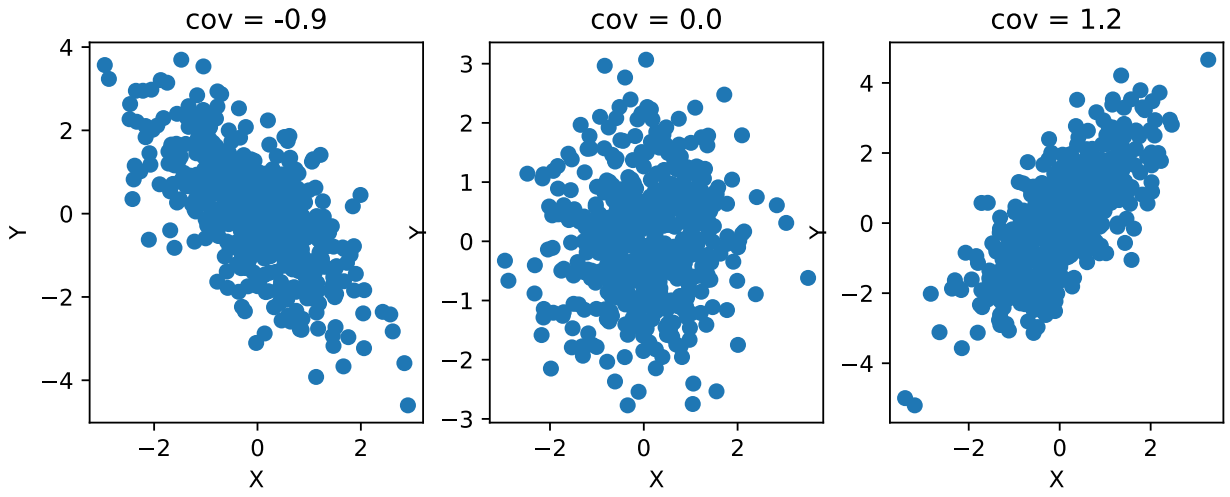
Kovaryansla ilgili bir not, yalnızca bu doğrusal ilişkileri ölçmesidir. $X = Y^2$ gibi daha karmaşık ilişkiler, Y'nin $\{-2, -1, 0, 1, 2\}$'dan eşit olasılıkla rastgele seçildiği durumlarda, gözden kaçabilir. Aslında hızlı bir hesaplama, biri diğerinin deterministik bir fonksiyonu olmasına rağmen, bu rastgele değişkenlerin kovaryansının sıfır olduğunu gösterir.

Sürekli rastgele değişkenler için, hemen hemen aynı hikaye geçerlidir. Bu noktada, ayrık ve sürekli arasındaki geçişi yapmakta oldukça rahatız, bu nedenle herhangi bir türetme olmaksızın (18.6.39) denkleminin sürekli benzerini sağlayacağız.

$$\sigma_{XY} = \int_{\mathbb{R}^2} (x - \mu_X)(y - \mu_Y) p(x, y)\, dx\, dy. \tag{18.6.42}$$

Görselleştirme için, ayarlanabilir kovaryanslı rastgele değişkenlerinden bir topluluğa bakalım.

```
# Birkaç rastgele değişken ayarlanabilir kovaryansı çiz
covs = [-0.9, 0.0, 1.2]
d2l.plt.figure(figsize=(12, 3))
for i in range(3):
    X = torch.randn(500)
    Y = covs[i]*X + torch.randn(500)

    d2l.plt.subplot(1, 4, i+1)
    d2l.plt.scatter(X.numpy(), Y.numpy())
    d2l.plt.xlabel('X')
    d2l.plt.ylabel('Y')
    d2l.plt.title(f'cov = {covs[i]}')
d2l.plt.show()
```



Kovaryansların bazı özelliklerini görelim:

- Herhangi bir rastgele değişken X için, $\mathrm{Cov}(X, X) = \mathrm{Var}(X)$'dir.
- X, Y rasgele değişkenleri ve a ve b sayıları için, $\mathrm{Cov}(aX + b, Y) = \mathrm{Cov}(X, aY + b) = a\mathrm{Cov}(X, Y)$'dir.
- X ve Y bağımsızsa, $\mathrm{Cov}(X, Y) = 0$'dır.

Ek olarak, daha önce gördüğümüz bir ilişkiyi genişletmek için kovaryansı kullanabiliriz. X ve Y'nin iki bağımsız rastgele değişken olduğunu hatırlayın.

$$\mathrm{Var}(X + Y) = \mathrm{Var}(X) + \mathrm{Var}(Y). \tag{18.6.43}$$

Kovaryans bilgisi ile bu ilişkiyi genişletebiliriz. Aslında, biraz cebir genel olarak şunu gösterebilir:

$$\mathrm{Var}(X + Y) = \mathrm{Var}(X) + \mathrm{Var}(Y) + 2\mathrm{Cov}(X, Y). \tag{18.6.44}$$

Bu, ilişkili rastgele değişkenler için varyans toplama kuralını genelleştirmemize olanak tanır.

Korelasyon

Ortalamalar ve varyanslar durumunda yaptığımız gibi, şimdi birimleri ele alalım. X bir birimde ölçülürse (inç diyelim) ve Y başka bir birimde ölçülürse (dolar diyelim), kovaryans bu iki birimin çarpımı inç $\times$ dolar cinsinden ölçülür. Bu birimleri yorumlamak zor olabilir. Bu durumda sık sık isteyeceğimiz şey, birimsiz ilişkililik ölçümüdür. Aslında, çoğu zaman kesin nicel korelasyonu önemsemiyoruz, bunun yerine korelasyonun aynı yönde olup olmadığını ve ilişkinin ne kadar güçlü olduğunu soruyoruz.

Neyin mantıklı olduğunu görmek için bir düşünce deneyi yapalım. Rastgele değişkenlerimizi inç ve dolar cinsinden inç ve sent olarak dönüştürdüğümüzü varsayalım. Bu durumda rastgele Y değişkeni 100\$ ile çarpılır. Tanım üzerinde çalışırsak bu, $\mathrm{Cov}(X, Y)$'nın 100\$ ile çarpılacağı anlamına gelir. Böylece, bu durumda birimlerin değişmesinin kovaryansı 100\$lık bir çarpan kadar değiştirdiğini görüyoruz. Bu nedenle, birim değişmez korelasyon ölçümümüzü bulmak için, yine 100\$ ile ölçeklenen başka bir şeye bölmemiz gerekecek. Gerçekten de bariz bir adayımız var, standart sapma! Böylece, *korelasyon katsayısını* tanımlarsak

$$\rho(X, Y) = \frac{\mathrm{Cov}(X, Y)}{\sigma_X \sigma_Y}, \tag{18.6.45}$$

Bunun birimsiz bir değer olduğunu görüyoruz. Biraz matematik, bu sayının -1 ile 1 arasında olduğunu ve 1'in maksimum pozitif korelasyona sahip olduğunu, -1'in ise maksimum negatif korelasyon olduğunu gösterebilir.

Yukarıdaki açık ayrık örneğimize dönersek, $\sigma_X = 1$ ve $\sigma_Y = 2$ olduğunu görebiliriz, böylece iki rastgele değişken arasındaki korelasyonu (18.6.45) kullanarak hesaplayabiliriz:

$$\rho(X, Y) = \frac{4p - 2}{1 \cdot 2} = 2p - 1. \tag{18.6.46}$$

Bu şimdi -1 ile 1 arasında değişiyor ve beklenen davranış 1'in en çok ilişkili olduğu ve -1'in ise minimum düzeyde ilişkili olduğu anlamına gelmesidir.

Başka bir örnek olarak, herhangi bir rastgele değişken olarak X'i ve X'in herhangi bir doğrusal deterministik fonksiyonu olarak $Y = aX + b$'yi düşünün. Sonra, bunu hesaplayabiliriz:

$$\sigma_Y = \sigma_{aX+b} = |a|\sigma_X, \tag{18.6.47}$$

$$\text{Cov}(X, Y) = \text{Cov}(X, aX + b) = a\text{Cov}(X, X) = a\text{Var}(X), \tag{18.6.48}$$

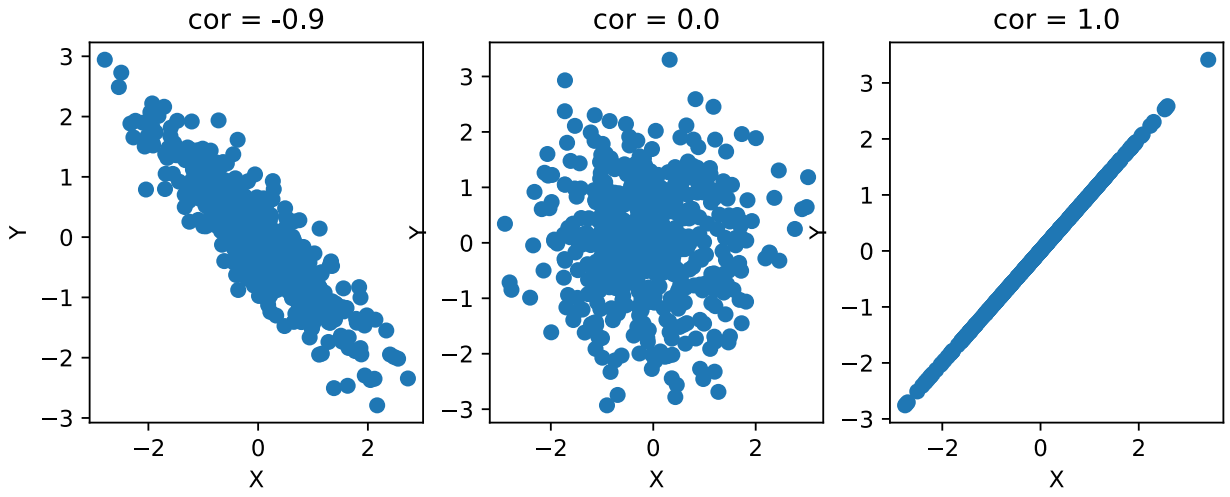
böylece (18.6.45) ile beraber:

$$\rho(X, Y) = \frac{a\text{Var}(X)}{|a|\sigma_X^2} = \frac{a}{|a|} = \text{sign}(a). \tag{18.6.49}$$

Böylece korelasyonun herhangi bir $a > 0$ için $+1$ olduğunu ve herhangi bir $a < 0$ için -1 olduğunu görürüz, yani korelasyon iki rasgele değişkenin ilişkisinin derecesini ve yönünü ölçer, varyasyonun aldığı ölçeği değil.

Yine ayarlanabilir korelasyonlu rastgele değişkenlerin bir koleksiyonunu çizelim.

```
# Birkaç rastgele değişken ayarlanabilir korelasyonu çiz
cors = [-0.9, 0.0, 1.0]
d2l.plt.figure(figsize=(12, 3))
for i in range(3):
    X = torch.randn(500)
    Y = cors[i] * X + torch.sqrt(torch.tensor(1) -
                                 cors[i]**2) * torch.randn(500)

    d2l.plt.subplot(1, 4, i + 1)
    d2l.plt.scatter(X.numpy(), Y.numpy())
    d2l.plt.xlabel('X')
    d2l.plt.ylabel('Y')
    d2l.plt.title(f'cor = {cors[i]}')
d2l.plt.show()
```



Korelasyonun birkaç özelliğini aşağıda listeleyelim.

- Herhangi bir rastgele değişken X için , $\rho(X, X) = 1$.
- X, Y rasgele değişkenleri ve a ve b sayıları için, $\rho(aX + b, Y) = \rho(X, aY + b) = \rho(X, Y)$.
- X ve Y, sıfır olmayan varyansla bağımsızsa, $\rho(X, Y) = 0$.

Son bir not olarak, bu formüllerden bazılarının tanıdık geldiğini hissedebilirsiniz. Gerçekten, her şeyi $\mu_X = \mu_Y = 0$ varsayarak genişletirsek, görürüz ki

$$\rho(X, Y) = \frac{\sum_{i,j} x_i y_i p_{ij}}{\sqrt{\sum_{i,j} x_i^2 p_{ij}} \sqrt{\sum_{i,j} y_j^2 p_{ij}}}. \tag{18.6.50}$$

Bu, terimlerin toplamının karekköküne bölünen bir terimler çarpımının toplamına benziyor. Bu tam olarak, p_{ij} ile ağırlıklandırılmış farklı koordinatlara sahip iki $\mathbf{v}, \mathbf{w}$ vektörü arasındaki açının kosinüsü için olan formüldür:

$$\cos(\theta) = \frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\|\|\mathbf{w}\|} = \frac{\sum_i v_i w_i}{\sqrt{\sum_i v_i^2} \sqrt{\sum_i w_i^2}}. \tag{18.6.51}$$

Aslında, normların standart sapmalarla ilişkili olduğunu ve korelasyonların açıların kosinüsü olduğunu düşünürsek, geometriden elde ettiğimiz sezgilerin çoğu rastgele değişkenler hakkında düşünmeye uygulanabilir.

18.6.2 Özet

- Sürekli rastgele değişkenler, bir dizi değer sürekliliği alabilen rastgele değişkenlerdir. Ayrık rastgele değişkenlere kıyasla, çalışmayı daha zor hale getiren bazı teknik zorlukları vardır.
- Olasılık yoğunluk fonksiyonu, bir aralıktaki eğrinin altındaki alanın o aralıkta bir örnek nokta bulma olasılığını gösteren bir fonksiyon vererek, sürekli rastgele değişkenlerle çalışmamızı sağlar.
- Birikimli dağılım işlevi, rastgele değişkenin belirli bir eşik değerinden daha düşük olduğunu gözlemleme olasılığıdır. Ayrık ve sürekli değişkenleri birleştiren kullanışlı bir alternatif bakış açısı sağlayabilir.

- Ortalama, rastgele bir değişkenin ortalama değeridir.
- Varyans, rastgele değişken ile onun ortalaması arasındaki farkın beklenen (ortalama) karesidir.
- Standart sapma, varyansın karekköküdür. Rasgele değişkenin alabileceği değerlerin aralığını ölçmek olarak düşünülebilir.
- Chebyshev'in eşitsizliği, çoğu zaman rastgele değişkeni içeren açık bir aralık vererek bu sezgiyi içleştirmemize izin verir.
- Bileşik yoğunluklar, ilişkili rastgele değişkenlerle çalışmamıza izin verir. İstenilen rastgele değişkenin dağılımını elde etmek için istenmeyen rastgele değişkenlerin integralini alarak bileşik yoğunluklarını marjinalize edebiliriz.
- Kovaryans ve korelasyon katsayısı, iki ilişkili rastgele değişken arasındaki herhangi bir doğrusal ilişkiyi ölçmenin bir yol sağlar.

18.6.3 Alıştırmalar

1. $x \geq 1$ için yoğunluğu $p(x) = \frac{1}{x^2}$ ve aksi takdirde $p(x) = 0$ ile verilen rastgele değişkenimiz olduğunu varsayalım. $P(X > 2)$ nedir?
2. Laplace dağılımı, yoğunluğu $p(x) = \frac{1}{2}e^{-|x|}$ ile verilen rastgele bir değişkendir. Bu fonksiyonun ortalaması ve standart sapması nedir? Bir ipucu, $\int_0^\infty xe^{-x}\,dx = 1$ ve $\int_0^\infty x^2e^{-x}\,dx = 2$.
3. Sokakta size doğru yürüyorum ve "Ortalaması \$1 \$, standart sapması 2 olan rastgele bir değişkenim var ve örneklerimin $\%25$'inin 9'dan daha büyük bir değer aldığını gözlemledim." dedim. Bana inanır mısınız? Neden evet ya da neden hayır?
4. $x, y \in [0, 1]$ için $p_{XY}(x, y) = 4xy$ ve aksi takdirde $p_{XY}(x, y) = 0$ ile bileşik yoğunlukları verilen iki rastgele değişkenimizin X, Y olduğunu varsayalım. X ve Y'nin kovaryansı nedir?

Tartışmalar[222]

18.7 Maksimum (Azami) Olabilirlik

Makine öğrenmesinde en sık karşılaşılan düşünce yöntemlerinden biri, maksimum olabilirlik bakış açısıdır. Bu, bilinmeyen parametrelere sahip olasılıklı bir modelle çalışırken, verileri en yüksek olasılığa sahip kılan parametrelerin en olası parametreler olduğu kavramıdır.

18.7.1 Maksimum Olabilirlik İlkesi

Bunun, üzerinde düşünmeye yardımcı olabilecek Bayesçi bir yorumu vardır. $\boldsymbol{\theta}$ parametrelerine ve X veri örneklerine sahip bir modelimiz olduğunu varsayalım. Somutluk açısından, $\boldsymbol{\theta}$'nın, bir bozuk paranın ters çevrildiğinde tura gelme olasılığını temsil eden tek bir değer ve X'in bağımsız bir bozuk para çevirme dizisi olduğunu hayal edebiliriz. Bu örneğe daha sonra derinlemesine bakacağız.

[222] https://discuss.d2l.ai/t/1094

Modelimizin parametreleri için en olası değeri bulmak istiyorsak, bu, bunu bulmak istediğimiz anlamına gelir:

$$\text{argmax}\, P(\boldsymbol{\theta} \mid X). \tag{18.7.1}$$

Bayes kuralı gereği yukarıdaki ifade ile aşağıdaki gibi yazılabilir:

$$\text{argmax}\, \frac{P(X \mid \boldsymbol{\theta})P(\boldsymbol{\theta})}{P(X)}. \tag{18.7.2}$$

Verileri oluşturmanın parametreden bağımsız bir olasılığı olan $P(X)$ ifadesi, $\boldsymbol{\theta}$'ya hiç bağlı değildir ve bu nedenle en iyi $\boldsymbol{\theta}$ seçeneği değiştirilmeden atılabilir. Benzer şekilde, şimdi hangi parametre kümesinin diğerlerinden daha iyi olduğuna dair önceden bir varsayımımız olmadığını farzedebiliriz, bu yüzden $P(\boldsymbol{\theta})$'nın da $\boldsymbol{\theta}$'ya bağlı olmadığını beyan edebiliriz! Bu, örneğin, yazı tura atma örneğimizde, tura gelme olasılığının önceden adil olup olmadığına dair herhangi bir inanca sahip olmadan $[0, 1]$ arasında herhangi bir değer olabileceği durumlarda anlamlıdır (genellikle *bilgisiz önsel* olarak anılır). Böylece, Bayes kuralı uygulamamızın, en iyi $\boldsymbol{\theta}$ seçimimizin $\boldsymbol{\theta}$ için maksimum olasılık tahmini olduğunu gösterdiğini görüyoruz:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta}}{\text{argmax}}\, P(X \mid \boldsymbol{\theta}). \tag{18.7.3}$$

Ortak bir terminoloji için, $(P(X \mid \boldsymbol{\theta}))$ parametreleri verilen verilerin olasılığı *olabilirlik* olarak adlandırılır.

Somut Bir Örnek

Bunun nasıl çalıştığını somut bir örnekle görelim. Tura atma olasılığını temsil eden tek bir θ parametremiz olduğunu varsayalım. O zaman yazı atma olasılığı $1 - \theta$ olur ve bu nedenle, gözlemlenen verimiz X, n_T tura ve n_Y yazıdan oluşan bir diziyse, bağımsız olasılıkları çarparak görürüz ki:

$$P(X \mid \theta) = \theta^{n_H}(1 - \theta)^{n_T}. \tag{18.7.4}$$

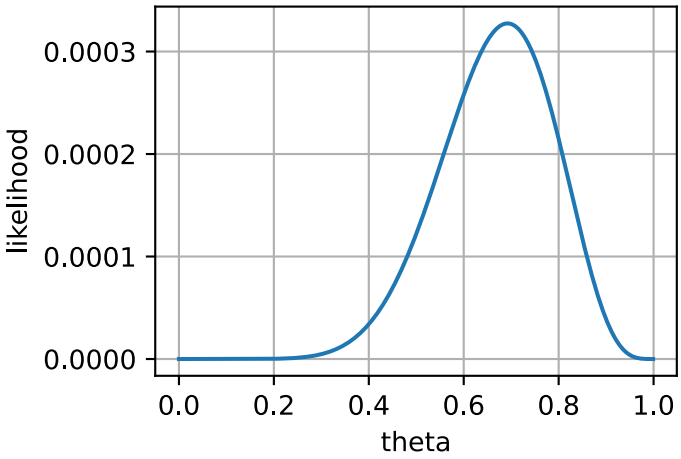
13 tane madeni parayı atarsak ve $n_T = 9$ ve $n_Y = 4$ olan "TTTYTYYTTTTTY" dizisini alırsak, bunun şu olduğunu görürüz:

$$P(X \mid \theta) = \theta^9(1 - \theta)^4. \tag{18.7.5}$$

Bu örnekle ilgili güzel bir şey, cevabın nasıl geleceğini bilmemizdir. Gerçekten de, sözlü olarak, "13 para attım ve 9 tura geldi, tura gelmesi olasılığı için en iyi tahminimiz nedir?", herkes doğru bir şekilde $9/13$ olarak tahmin edecektir. Bu maksimum olabilirlik yönteminin bize vereceği şey, bu sayıyı ilk ilkelerden çok daha karmaşık durumlara genelleyecek bir şekilde elde etmenin bir yoludur.

Örneğimiz için, $P(X \mid \theta)$ grafiği aşağıdaki gibidir:

```
%matplotlib inline
import torch
from d2l import torch as d2l

theta = torch.arange(0, 1, 0.001)
p = theta**9 * (1 - theta)**4.

d2l.plot(theta, p, 'theta', 'likelihood')
```



Bunun maksimum değeri, beklentisi $9/13 \approx 0.7\ldots$'a yakın bir yerde. Bunun tam olarak orada olup olmadığını görmek için kalkülüse dönebiliriz. Maksimumda fonksiyonun gradyanının düz olduğuna dikkat edin. Böylece, türevin sıfır olduğu yerde θ değerlerini bularak ve en yüksek olasılığı veren yerde maksimum olabilirlik tahminini (18.7.1) bulabiliriz. Hesaplayalım:

$$\begin{aligned} 0 &= \frac{d}{d\theta} P(X \mid \theta) \\ &= \frac{d}{d\theta} \theta^9 (1-\theta)^4 \\ &= 9\theta^8(1-\theta)^4 - 4\theta^9(1-\theta)^3 \\ &= \theta^8(1-\theta)^3(9-13\theta). \end{aligned} \tag{18.7.6}$$

Bunun üç çözümü vardır: 0, 1 ve $9/13$. İlk ikisi açıkça minimumdur, dizimize 0 olasılık atadıkları için maksimum değildirler. Nihai değer, dizimize sıfır olasılık *atamaz* ve bu nedenle, maksimum olasılık tahmini $\hat{\theta} = 9/13$ olmalıdır.

18.7.2 Sayısal Optimizasyon (Eniyileme) ve Negatif Logaritmik-Olabilirliği

Önceki örnek güzel, ama ya milyarlarca parametremiz ve veri örneğimiz varsa?

İlk olarak, tüm veri örneklerinin bağımsız olduğunu varsayarsak, pratik olarak olabilirliliğin kendisini artık pek çok olasılığın bir çarpımı olarak değerlendiremeyeceğimize dikkat edin. Aslında, her olasılık $[0, 1]$ arasındadır, diyelim ki tipik olarak yaklaşık $1/2$ değerindedir ve $(1/2)^{1000000000}$ çarpımı makine hassasiyetinin çok altındadır. Bununla doğrudan çalışamayız.

Ancak, logaritmanın çarpımları toplamlara dönüştürdüğünü hatırlayın, bu durumda

$$\log((1/2)^{1000000000}) = 1000000000 \cdot \log(1/2) \approx -301029995.6\ldots \tag{18.7.7}$$

Bu sayı, 32-bitlik kayan virgüllü sayı tek duyarlılığına bile mükemmel şekilde uyuyor. Bu nedenle, *log-olabilirliği* göz önünde bulundurmalıyız.

$$\log(P(X \mid \boldsymbol{\theta})). \tag{18.7.8}$$

$x \mapsto \log(x)$ işlevi arttığından, olabilirliliği en üst düzeye çıkarmak, log-olabilirliliği en üst düzeye çıkarmakla aynı şeydir. Nitekim Section 18.9 içinde, naif Bayes sınıflandırıcısının belirli bir örneğiyle çalışırken bu mantığın uygulandığını göreceğiz.

Genellikle kaybı en aza indirmek istediğimiz kayıp işlevleriyle çalışırız. *Negatif logaritmik-olabilirlik* olan $-\log(P(X \mid \boldsymbol{\theta}))$'ı alarak maksimum olabilirliliği bir kaybın en aza indirilmesine çevirebiliriz.

Bunu örnekle görselleştirmek için, yazı tura atma problemini önceden düşünün ve kapalı form çözümünü bilmiyormuşuz gibi davranın. Bunu hesaplayabiliriz

$$-\log(P(X \mid \boldsymbol{\theta})) = -\log(\theta^{n_H}(1-\theta)^{n_T}) = -(n_H \log(\theta) + n_T \log(1-\theta)). \tag{18.7.9}$$

Bu, koda yazılabilir ve milyarlarca bozuk para atmak için bile serbestçe optimize edilebilir.

```
# Verilerimizi ayarlayın
n_H = 8675309
n_T = 25624

# Parametrelerimizi ilkle
theta = torch.tensor(0.5, requires_grad=True)

# Gradyan inişi gerçekleştir
lr = 0.00000000001
for iter in range(10):
    loss = -(n_H * torch.log(theta) + n_T * torch.log(1 - theta))
    loss.backward()
    with torch.no_grad():
        theta -= lr * theta.grad
    theta.grad.zero_()

# Çıktıyı kontrol et
theta, n_H / (n_H + n_T)
```

```
(tensor(0.5017, requires_grad=True), 0.9970550284664874)
```

İnsanların negatif logaritma olasılıklarını kullanmayı sevmesinin tek nedeni sayısal kolaylık değildir. Tercih edilmesinin birkaç nedeni daha var.

Logaritmik-olabilirliliğini düşünmemizin ikinci nedeni, kalkülüs kurallarının basitleştirilmiş uygulamasıdır. Yukarıda tartışıldığı gibi, bağımsızlık varsayımları nedeniyle, makine öğrenmesinde karşılaştığımız çoğu olasılık, bireysel olasılıkların çarpımıdır.

$$P(X \mid \boldsymbol{\theta}) = p(x_1 \mid \boldsymbol{\theta}) \cdot p(x_2 \mid \boldsymbol{\theta}) \cdots p(x_n \mid \boldsymbol{\theta}). \tag{18.7.10}$$

Bu demektir ki, bir türevi hesaplamak için çarpım kuralını doğrudan uygularsak,

$$\begin{aligned}\frac{\partial}{\partial \boldsymbol{\theta}} P(X \mid \boldsymbol{\theta}) = &\left(\frac{\partial}{\partial \boldsymbol{\theta}} P(x_1 \mid \boldsymbol{\theta})\right) \cdot P(x_2 \mid \boldsymbol{\theta}) \cdots P(x_n \mid \boldsymbol{\theta}) \\ &+ P(x_1 \mid \boldsymbol{\theta}) \cdot \left(\frac{\partial}{\partial \boldsymbol{\theta}} P(x_2 \mid \boldsymbol{\theta})\right) \cdots P(x_n \mid \boldsymbol{\theta}) \\ &\vdots \\ &+ P(x_1 \mid \boldsymbol{\theta}) \cdot P(x_2 \mid \boldsymbol{\theta}) \cdots \left(\frac{\partial}{\partial \boldsymbol{\theta}} P(x_n \mid \boldsymbol{\theta})\right).\end{aligned} \tag{18.7.11}$$

Bu, $(n-1)$ toplama ile birlikte $n(n-1)$ çarpımı gerektirir, bu nedenle işlem zamanı girdilerin ikinci dereceden polinomuyla orantılıdır! Yeterli akıllılık terimleri gruplayarak bunu doğrusal

zamana indirgeyecektir, ancak biraz düşünmeyi gerektirir. Negatif logaritmik-olabilirlikte ise, bunun yerine

$$-\log\left(P(X \mid \boldsymbol{\theta})\right) = -\log(P(x_1 \mid \boldsymbol{\theta})) - \log(P(x_2 \mid \boldsymbol{\theta})) \cdots - \log(P(x_n \mid \boldsymbol{\theta})), \tag{18.7.12}$$

o da böyle buna dönüşür

$$-\frac{\partial}{\partial \boldsymbol{\theta}} \log\left(P(X \mid \boldsymbol{\theta})\right) = \frac{1}{P(x_1 \mid \boldsymbol{\theta})} \left(\frac{\partial}{\partial \boldsymbol{\theta}} P(x_1 \mid \boldsymbol{\theta})\right) + \cdots + \frac{1}{P(x_n \mid \boldsymbol{\theta})} \left(\frac{\partial}{\partial \boldsymbol{\theta}} P(x_n \mid \boldsymbol{\theta})\right). \tag{18.7.13}$$

Bu sadece n bölme ve $n-1$ toplam gerektirir ve dolayısıyla girdilerle doğrusal zamanlıdır.

Negatif logaritmik-olabilirliği göz önünde bulundurmanın üçüncü ve son nedeni, bilgi teorisi ile olan ilişkidir ve bunu Section 18.11 içinde ayrıntılı olarak tartışacağız. Bu, rastgele bir değişkendeki bilginin veya rasgeleliğin derecesini ölçmek için bir yol sağlayan titiz bir matematiksel teoridir. Bu alanda çalışmanın temel konusu entropidir.

$$H(p) = -\sum_i p_i \log_2(p_i), \tag{18.7.14}$$

Bir kaynağın rasgeleliğini ölçer. Bunun ortalama $-\log$ olasılığından başka bir şey olmadığına dikkat edin ve bu nedenle, negatif logaritmik-olabilirliğimizi alıp veri örneklerinin sayısına bölersek, çapraz-entropi (cross-entropy) olarak bilinen göreceli bir entropi elde ederiz. Tek başına bu teorik yorum, model performansını ölçmenin bir yolu olarak, veri kümesi üzerinden ortalama negatif logaritmik-olabilirliliği rapor etmeyi motive etmek için yeterince zorlayıcı olacaktır.

18.7.3 Sürekli Değişkenler için Maksimum Olabilirlik

Şimdiye kadar yaptığımız her şey, ayrık rastgele değişkenlerle çalıştığımızı varsayıyor; ancak ya sürekli olanlarla çalışmak istersek?

Kısaca özet, olasılığın tüm örneklerini olasılık yoğunluğu ile değiştirmemiz dışında hiçbir şeyin değişmemesidir. Yoğunlukları küçük harfli p ile yazdığımızı hatırlarsak, bu, örneğin şimdi şunu söylediğimiz anlamına gelir:

$$-\log\left(p(X \mid \boldsymbol{\theta})\right) = -\log(p(x_1 \mid \boldsymbol{\theta})) - \log(p(x_2 \mid \boldsymbol{\theta})) \cdots - \log(p(x_n \mid \boldsymbol{\theta})) = -\sum_i \log(p(x_i \mid \theta)). \tag{18.7.15}$$

Soru, "Bu neden geçerli?" haline gelir. Sonuçta, yoğunlukları tanıtmamızın nedeni, belirli sonuçların kendilerinin elde edilme olasılıklarının sıfır olmasıydı ve dolayısıyla herhangi bir parametre kümesi için verilerimizi üretme olasılığımızın sıfır olmaz mı?

Aslında soru budur ve neden yoğunluklara geçebileceğimizi anlamak, epsilonlara ne olduğunu izlemeye yönelik bir alıştırmadır.

Önce hedefimizi yeniden tanımlayalım. Sürekli rastgele değişkenler için artık tam olarak doğru değeri elde etme olasılığını hesaplamak istemediğimizi, bunun yerine ϵ aralığında eşleştirme yapmak istediğimizi varsayalım. Basit olması için, verilerimizin aynı şekilde dağıtılmış rastgele değişkenler $X_1, \ldots, X_N$'nin tekrarlanan gözlemleri, $x_1, \ldots, x_N$, olduğunu varsayıyoruz. Daha önce gördüğümüz gibi, bu şu şekilde yazılabilir:

$$\begin{aligned} &P(X_1 \in [x_1, x_1+\epsilon], X_2 \in [x_2, x_2+\epsilon], \ldots, X_N \in [x_N, x_N+\epsilon] \mid \boldsymbol{\theta}) \\ \approx &\epsilon^N p(x_1 \mid \boldsymbol{\theta}) \cdot p(x_2 \mid \boldsymbol{\theta}) \cdots p(x_n \mid \boldsymbol{\theta}). \end{aligned} \tag{18.7.16}$$

Böylece, bunun negatif logaritmasını alırsak bunu elde ederiz:

$$\begin{aligned} &-\log(P(X_1 \in [x_1, x_1+\epsilon], X_2 \in [x_2, x_2+\epsilon], \ldots, X_N \in [x_N, x_N+\epsilon] \mid \boldsymbol{\theta})) \\ \approx &-N\log(\epsilon) - \sum_i \log(p(x_i \mid \boldsymbol{\theta})). \end{aligned} \tag{18.7.17}$$

Bu ifadeyi incelersek, ϵ'un olduğu tek yer $-N\log(\epsilon)$ toplamsal sabitidir. Bu, $\boldsymbol{\theta}$ parametrelerine hiç bağlı değildir, dolayısıyla en uygun $\boldsymbol{\theta}$ seçimi, ϵ seçimimize bağlı değildir! Dört veya dört yüz basamaklı da talep edersek, en iyi $\boldsymbol{\theta}$ seçimi aynı kalır, böylece epsilon'u serbestçe dışarıda bırakıp optimize etmek istediğimiz şey bu olur:

$$-\sum_i \log(p(x_i \mid \boldsymbol{\theta})). \tag{18.7.18}$$

Böylelikle, olasılıkları olasılık yoğunlukları ile değiştirerek, maksimum olabilirlik bakış açısının sürekli rastgele değişkenlerle, kesikli olanlar kadar kolay bir şekilde çalışabileceğini görüyoruz.

18.7.4 Özet

- Maksimum olabilirlik ilkesi bize, belirli bir veri kümesi için en uygun modelin en yüksek olasılıkla verileri üreten model olduğunu söyler.
- Genellikle insanlar çeşitli nedenlerde dolayı negatif logaritmik-olabilirlik ile çalışırlar: Sayısal kararlılık, çarpımların toplamlara dönüştürülmesi (ve bunun sonucunda gradyan hesaplamalarının basitleştirilmesi) ve bilgi teorisine yönelik teorik bağlar.
- Ayrık ortamda motive etmek en basiti olsa da, veri noktalarına atanan olasılık yoğunluğunu en üst düzeye çıkararak sürekli ortamda da serbestçe genelleştirilebilir.

18.7.5 Alıştırmalar

1. Rastgele bir değişkenin bir α değeri için $\frac{1}{\alpha}e^{-\alpha x}$ yoğunluğuna sahip olduğunu bildiğinizi varsayalım. Rastgele değişkenden 3 sayısını tek gözlem olarak elde ediyorsunuz. α için maksimum olabilirlik tahmini nedir?
2. Ortalama değeri bilinmeyen ancak varyansı 1 olan bir Gauss'tan alınmış $\{x_i\}_{i=1}^N$ örnekten oluşan bir veri kümeniz olduğunu varsayalım. Ortalama için maksimum olabilirlik tahmini nedir?

Tartışmalar[223]

18.8 Dağılımlar

Artık hem kesikli hem de sürekli ortamda olasılıkla nasıl çalışılacağını öğrendiğimize göre, şimdi karşılaşılan yaygın dağılımlardan bazılarını öğrenelim. Makine öğrenmesi alanına bağlı olarak, bunlardan çok daha fazlasına aşina olmamız gerekebilir; derin öğrenmenin bazı alanları için potansiyel olarak hiç kullanılmıyorlar. Ancak bu, aşina olunması gereken iyi bir temel listedir. Önce bazı ortak kütüphaneleri içeri aktaralım.

[223] https://discuss.d2l.ai/t/1096

```
%matplotlib inline
from math import erf, factorial
import torch
from IPython import display
from d2l import torch as d2l

torch.pi = torch.acos(torch.zeros(1)) * 2  # Pi'yi tanımla
```

18.8.1 Bernoulli

Bu, genellikle karşılaşılan en basit rastgele değişkendir. Bu rastgele değişken, p olasılıkla 1 ve $1-p$ olasılıkla 0 gelen bir yazı tura atmayı kodlar. Bu dağılımla rastgele bir değişkenimiz X varsa, şunu yazacağız:

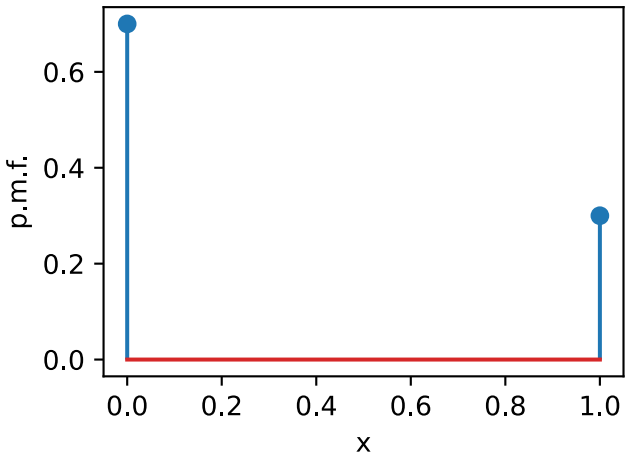
$$X \sim \text{Bernoulli}(p). \tag{18.8.1}$$

Birikimli dağılım fonksiyonu şöyledir:

$$F(x) = \begin{cases} 0 & x < 0, \\ 1-p & 0 \leq x < 1, \\ 1 & x >= 1. \end{cases} \tag{18.8.2}$$

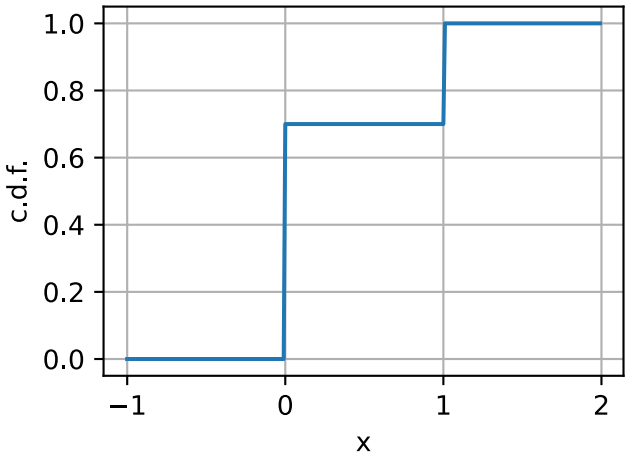
Olasılık kütle fonksiyonu aşağıda çizilmiştir.

```
p = 0.3

d2l.set_figsize()
d2l.plt.stem([0, 1], [1 - p, p], use_line_collection=True)
d2l.plt.xlabel('x')
d2l.plt.ylabel('p.m.f.')
d2l.plt.show()
```



Şimdi, (18.8.2) birikimli dağılım fonksiyonunu çizelim.

```
x = torch.arange(-1, 2, 0.01)

def F(x):
    return 0 if x < 0 else 1 if x > 1 else 1 - p

d2l.plot(x, torch.tensor([F(y) for y in x]), 'x', 'c.d.f.')
```



Eğer $X \sim \text{Bernoulli}(p)$ ise, o zaman:

- $\mu_X = p$,
- $\sigma_X^2 = p(1-p)$.

Bir Bernoulli rastgele değişkeninden keyfi şekilli bir diziyi aşağıdaki gibi örnekleyebiliriz.

```
1*(torch.rand(10, 10) < p)
```

```
tensor([[0, 0, 0, 0, 0, 1, 0, 0, 0, 1],
        [1, 0, 0, 0, 0, 0, 1, 0, 0, 0],
        [1, 0, 0, 0, 0, 0, 0, 1, 1, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
        [0, 0, 1, 0, 0, 0, 0, 1, 1, 0],
        [1, 1, 0, 0, 0, 1, 0, 0, 0, 0],
        [0, 0, 0, 1, 0, 1, 0, 1, 1, 0],
        [0, 1, 0, 0, 1, 0, 0, 1, 0, 0],
        [1, 0, 1, 1, 1, 1, 1, 1, 0, 0]])
```

18.8.2 Ayrık Tekdüze Dağılım

Bir sonraki yaygın karşılaşılan rastgele değişken, ayrık bir tekdüzedir. Buradaki tartışmamız için, $\{1, 2, \ldots, n\}$ tam sayılarında desteklendiğini varsayacağız, ancak herhangi bir tüm değerler kümesi serbestçe seçilebilir. Bu bağlamda *tekdüze* kelimesinin anlamı, olabilir her değerin eşit derecede olası olmasıdır. $i \in \{1, 2, 3, \ldots, n\}$ değerinin olasılığı $p_i = \frac{1}{n}$'dir. Bu dağılımla X rastgele değişkenini şu şekilde göstereceğiz:

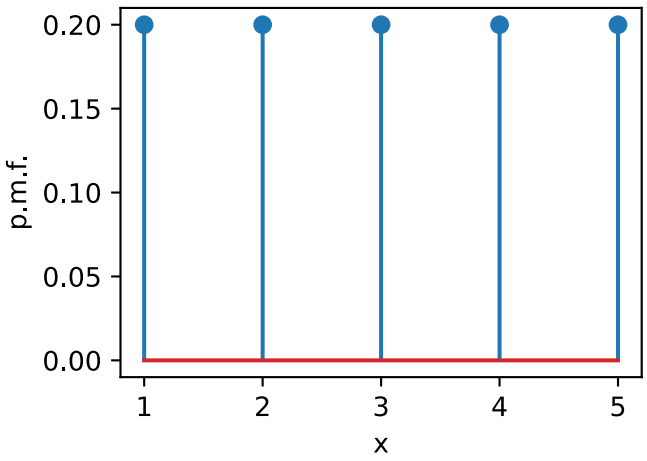
$$X \sim U(n). \tag{18.8.3}$$

Birikimli dağılım fonksiyonunu böyledir:

$$F(x) = \begin{cases} 0 & x < 1, \\ \frac{k}{n} & k \leq x < k+1 \text{ öyleki } 1 \leq k < n, \\ 1 & x >= n. \end{cases} \tag{18.8.4}$$

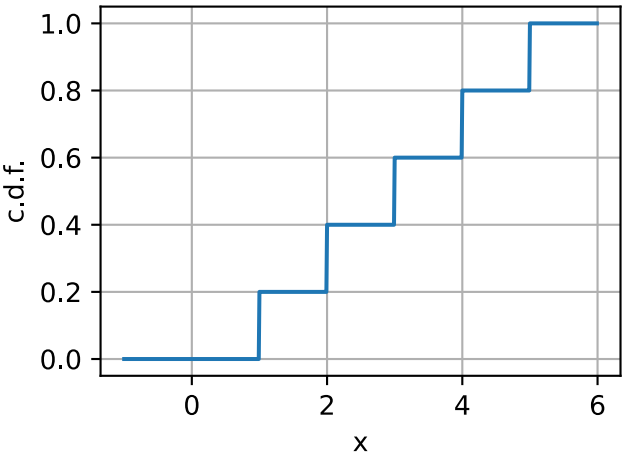
İlk olarak olasılık kütle fonksiyonunu çizelim.

```
n = 5

d2l.plt.stem([i+1 for i in range(n)], n*[1 / n], use_line_collection=True)
d2l.plt.xlabel('x')
d2l.plt.ylabel('p.m.f.')
d2l.plt.show()
```



Şimdi, (18.8.4) birikimli dağılım fonksiyonunu çizelim.

```
x = torch.arange(-1, 6, 0.01)

def F(x):
    return 0 if x < 1 else 1 if x > n else torch.floor(x) / n

d2l.plot(x, torch.tensor([F(y) for y in x]), 'x', 'c.d.f.')
```



Eğer $X \sim U(n)$ ise, o zaman:

- $\mu_X = \frac{1+n}{2}$,
- $\sigma_X^2 = \frac{n^2-1}{12}$.

Aşağıdaki gibi, ayrık bir tekdüze rastgele değişkenden keyfi şekilli bir diziyi örnekleyebiliriz.

```
torch.randint(1, n, size=(10, 10))
```

```
tensor([[3, 3, 4, 1, 3, 3, 3, 1, 1, 4],
        [2, 3, 4, 4, 4, 2, 2, 4, 4, 3],
        [4, 1, 3, 2, 3, 4, 4, 2, 3, 1],
        [1, 1, 2, 4, 2, 2, 3, 3, 3, 2],
        [2, 1, 3, 2, 4, 1, 2, 3, 4, 1],
        [3, 2, 4, 1, 3, 2, 4, 4, 4, 1],
        [2, 4, 1, 2, 3, 4, 4, 2, 3, 3],
        [3, 1, 4, 3, 4, 2, 3, 3, 1, 1],
        [3, 3, 1, 4, 1, 1, 2, 2, 4, 2],
        [1, 4, 3, 4, 1, 4, 1, 2, 2, 2]])
```

18.8.3 Sürekli Tekdüze Dağılım

Şimdi, sürekli tekdüze dağılımı tartışalım. Bu rastgele değişkenin arkasındaki fikir şudur: Ayrık tekdüze dağılımdaki n'yi arttırırsak ve bunu $[a, b]$ aralığına sığacak şekilde ölçeklendirirsek; sadece $[a, b]$ aralığında, hepsi eşit olasılıkla, keyfi bir değer seçen sürekli bir rastgele değişkene yaklaşacağız. Bu dağılımı şu şekilde göstereceğiz

$$X \sim U(a, b). \tag{18.8.5}$$

Olasılık yoğunluk fonksiyonu şöyledir:

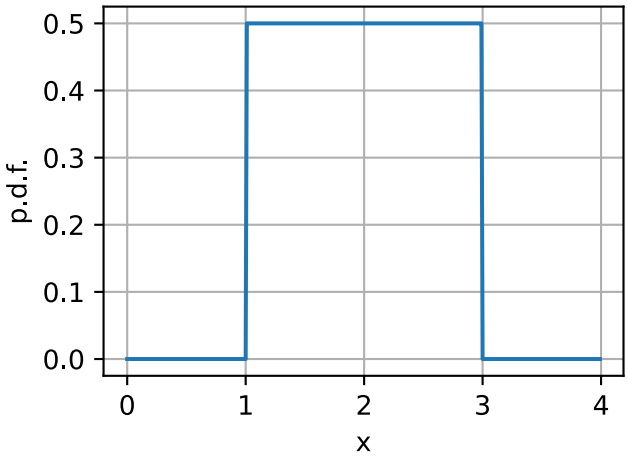
$$p(x) = \begin{cases} \frac{1}{b-a} & x \in [a, b], \\ 0 & x \notin [a, b]. \end{cases} \tag{18.8.6}$$

Birikimli dağılım fonksiyonunu şöyledir:

$$F(x) = \begin{cases} 0 & x < a, \\ \frac{x-a}{b-a} & x \in [a, b], \\ 1 & x >= b. \end{cases} \tag{18.8.7}$$

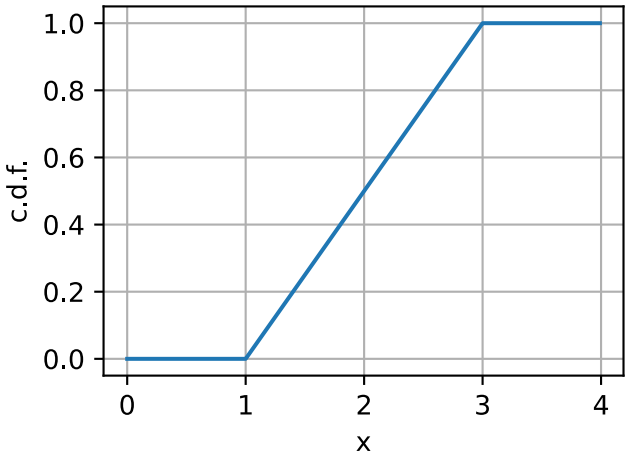
Önce, (18.8.6) olasılık yoğunluk dağılımı fonksiyonunu çizelim.

```
a, b = 1, 3

x = torch.arange(0, 4, 0.01)
p = (x > a).type(torch.float32)*(x < b).type(torch.float32)/(b-a)
d2l.plot(x, p, 'x', 'p.d.f.')
```



Şimdi, (18.8.7) birikimli dağılım fonksiyonunu çizelim.

```
def F(x):
    return 0 if x < a else 1 if x > b else (x - a) / (b - a)

d2l.plot(x, torch.tensor([F(y) for y in x]), 'x', 'c.d.f.')
```



Eğer $X \sim U(a, b)$ ise, o zaman:

- $\mu_X = \frac{a+b}{2}$,
- $\sigma_X^2 = \frac{(b-a)^2}{12}$.

Aşağıdaki gibi tekdüze bir rastgele değişkenden keyfi şekilli bir diziyi örnekleyebiliriz. Bunun $U(0, 1)$'den varsayılan örnekler olduğuna dikkat edin, bu nedenle farklı bir aralık istiyorsak, onu ölçeklendirmemiz gerekir.

```
(b - a) * torch.rand(10, 10) + a
```

```
tensor([[2.4319, 2.3027, 2.5008, 1.0314, 2.4978, 2.2231, 2.7293, 2.7454, 1.9746,
         2.9582],
        [2.6556, 2.6539, 1.9899, 2.1583, 1.6333, 1.4400, 1.4093, 1.8292, 1.8658,
         2.6306],
        [2.7257, 1.8550, 2.9031, 2.6528, 1.5819, 2.8775, 1.1190, 2.8663, 2.3309,
         2.3822],
        [2.7110, 1.7304, 2.2318, 2.1238, 1.1947, 2.8726, 1.4815, 2.0132, 2.3229,
         1.1324],
        [2.0713, 2.1257, 2.0809, 1.9627, 2.3050, 1.8019, 2.0731, 1.3395, 1.7201,
         2.6306],
        [2.6130, 1.2258, 2.8866, 2.6555, 2.5732, 1.1039, 1.9652, 2.1065, 1.3629,
         1.1522],
        [1.2170, 2.9644, 2.8276, 1.4999, 2.6880, 2.9844, 2.7410, 1.1381, 2.3237,
         2.5571],
        [2.6271, 1.8841, 2.8675, 2.8451, 2.7440, 1.3168, 2.3813, 1.4330, 1.7728,
         2.1355],
        [2.6843, 2.6480, 2.6468, 2.9452, 1.8513, 2.2427, 1.6430, 1.1144, 1.2585,
         1.2484],
        [2.2447, 1.8161, 2.6843, 2.2231, 1.2743, 2.0706, 2.1140, 2.7791, 1.0218,
         1.1806]])
```

18.8.4 Binom (İki Terimli) Dağılım

İşleri biraz daha karmaşık hale getirelim ve *iki terimli (binom)* rastgele değişkeni inceleyelim. Bu rastgele değişken, her birinin başarılı olma olasılığı p olan n bağımsız deneyler dizisi gerçekleştirmekten ve kaç tane başarı görmeyi beklediğimizi sormaktan kaynaklanır.

Bunu matematiksel olarak ifade edelim. Her deney, başarıyı kodlamak için 1 ve başarısızlığı kodlamak için 0 kullanacağımız bağımsız bir rastgele değişken X_i'dir. Her biri p olasılığı ile başarılı olan bağımsız bir yazı tura olduğu için, $X_i \sim \text{Bernoulli}(p)$ diyebiliriz. Sonra, iki terimli rastgele değişken aşağıdaki gibidir:

$$X = \sum_{i=1}^{n} X_i. \tag{18.8.8}$$

Bu durumda, böyle yazacağız:

$$X \sim \text{Binomial}(n, p). \tag{18.8.9}$$

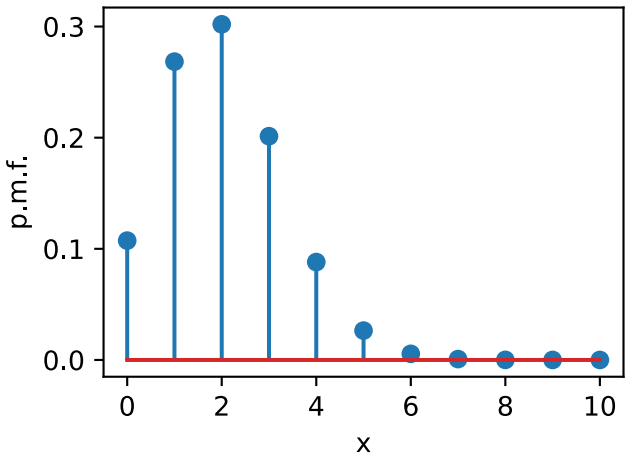
Birikimli dağılım işlevini elde etmek için, tam olarak k başarı elde etmenin gerçekleşebileceği hepsi $ p^{k(1-p)}{nk}$ olasılıklı $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ yolu olduğunu fark etmemiz gerekir. Böylece birikimli

dağılım işlevi bu olur:

$$F(x) = \begin{cases} 0 & x < 0, \\ \sum_{m \leq k} \binom{n}{m} p^m (1-p)^{n-m} & k \leq x < k+1 \text{ öyleki } 0 \leq k < n, \\ 1 & x >= n. \end{cases} \tag{18.8.10}$$

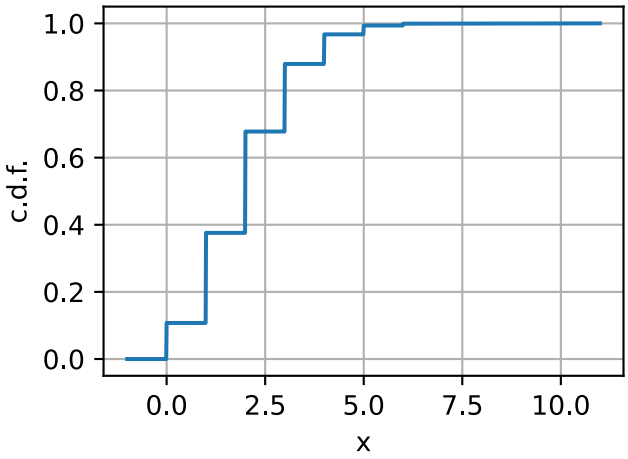
İlk olarak olasılık kütle fonksiyonu çizelim.

```
n, p = 10, 0.2

# Binom katsayısını hesapla
def binom(n, k):
    comb = 1
    for i in range(min(k, n - k)):
        comb = comb * (n - i) // (i + 1)
    return comb

pmf = torch.tensor([p**i * (1-p)**(n - i) * binom(n, i) for i in range(n + 1)])

d2l.plt.stem([i for i in range(n + 1)], pmf, use_line_collection=True)
d2l.plt.xlabel('x')
d2l.plt.ylabel('p.m.f.')
d2l.plt.show()
```



Şimdi, (18.8.10) birikimli dağılım fonksiyonunu çizelim.

```
x = torch.arange(-1, 11, 0.01)
cmf = torch.cumsum(pmf, dim=0)

def F(x):
    return 0 if x < 0 else 1 if x > n else cmf[int(x)]

d2l.plot(x, torch.tensor([F(y) for y in x.tolist()]), 'x', 'c.d.f.')
```



Eğer $X \sim \text{Binomial}(n, p)$ ise, o zaman:

- $\mu_X = np$,
- $\sigma_X^2 = np(1-p)$.

Bu, n Bernoulli rastgele değişkenlerinin toplamı üzerindeki beklenen değerin doğrusallığından ve bağımsız rastgele değişkenlerin toplamının varyansının varyansların toplamı olduğu gerçeğinden kaynaklanır. Bu aşağıdaki gibi örneklenebilir.

```
m = torch.distributions.binomial.Binomial(n, p)
m.sample(sample_shape=(10, 10))
```

```
tensor([[4., 0., 1., 1., 0., 2., 2., 4., 0., 3.],
        [3., 0., 2., 3., 1., 2., 0., 0., 1., 2.],
        [1., 4., 6., 1., 2., 2., 2., 1., 3., 2.],
        [1., 2., 3., 1., 2., 2., 4., 1., 2., 4.],
        [2., 0., 0., 1., 3., 1., 3., 0., 1., 3.],
        [2., 2., 5., 3., 0., 4., 2., 2., 2., 2.],
        [1., 2., 3., 1., 0., 1., 4., 4., 2., 4.],
        [3., 3., 2., 1., 4., 2., 1., 0., 3., 4.],
        [0., 1., 2., 2., 6., 3., 4., 3., 2., 2.],
        [1., 3., 2., 2., 4., 1., 5., 0., 3., 0.]])
```

18.8.5 Poisson Dağılımı

Şimdi bir düşünce deneyi yapalım. Bir otobüs durağında duruyoruz ve önümüzdeki dakika içinde kaç otobüsün geleceğini bilmek istiyoruz. $X^{(1)} \sim \text{Bernoulli}(p)$'i ele alarak başlayalım, bu basitçe bir otobüsün bir dakikalık pencerede varma olasılığıdır. Bir şehir merkezinden uzaktaki otobüs durakları için bu oldukça iyi bir yaklaşıklama olabilir. Bir dakikada birden fazla otobüs göremeyebiliriz.

Ancak, yoğun bir bölgedeysek, iki otobüsün gelmesi mümkün veya hatta muhtemeldir. Bunu, rastgele değişkenimizi ilk 30 saniye veya ikinci 30 saniye için ikiye bölerek modelleyebiliriz. Bu durumda bunu yazabiliriz:

$$X^{(2)} \sim X_1^{(2)} + X_2^{(2)}, \tag{18.8.11}$$

burada $X^{(2)}$ tüm toplamdır ve $X_i^{(2)} \sim \text{Bernoulli}(p/2)$. Toplam dağılım bu durumda $X^{(2)} \sim \text{Binomial}(2, p/2)$ olur.

Neden burada duralım? O dakikayı n parçaya ayırmaya devam edelim. Yukarısıyla aynı mantıkla, bunu görüyoruz:

$$X^{(n)} \sim \text{Binomial}(n, p/n). \tag{18.8.12}$$

Bu rastgele değişkenleri düşünün. Önceki bölümden şunu biliyoruz (18.8.12), ortalama $\mu_{X^{(n)}} = n(p/n) = p$ ve varyans $\sigma^2_{X^{(n)}} = n(p/n)(1-(p/n)) = p(1-p/n)$'dir. $n \to \infty$ alırsak, bu sayıların ortalama $\mu_{X^{(\infty)}} = p$ ve varyans $\sigma^2_{X^{(\infty)}} = p$ şeklinde sabitlendiğini görebiliriz. Bu, bu sonsuz alt bölüm limitinde tanımlayabileceğimiz bazı rastgele değişkenlerin *olabileceğini* gösterir.

Bu çok fazla sürpriz olmamalı, çünkü gerçek dünyada sadece otobüslerin geliş sayısını sayabiliyoruz, ancak matematiksel modelimizin iyi tanımlandığını görmek güzel. Bu tartışma, *nadir olaylar yasası* olarak kurallı yapılabilir.

Bu akıl yürütmeyi dikkatlice takip ederek aşağıdaki modele ulaşabiliriz. $\{0, 1, 2, \ldots\}$ değerlerini aşağıdaki olasılıkla alan rastgele bir değişken ise $X \sim \text{Poisson}(\lambda)$ diyeceğiz.

$$p_k = \frac{\lambda^k e^{-\lambda}}{k!}. \tag{18.8.13}$$

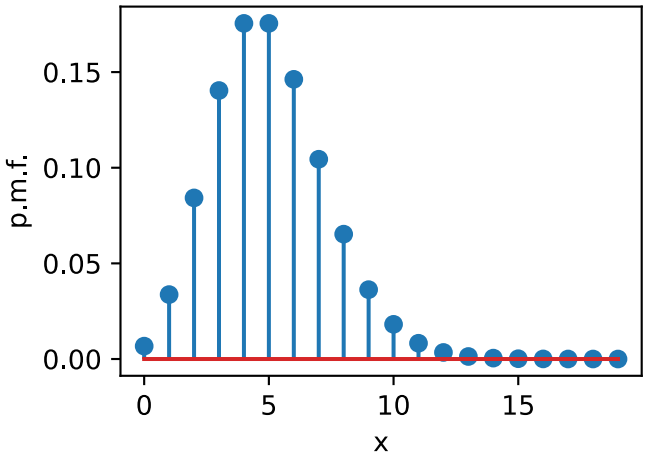
$\lambda > 0$ değeri *oran* (veya *şekil* parametresi) olarak bilinir ve bir zaman biriminde beklediğimiz ortalama varış sayısını belirtir.

Birikimli dağılım işlevini elde etmek için bu olasılık kütle işlevini toplayabiliriz.

$$F(x) = \begin{cases} 0 & x < 0, \\ e^{-\lambda} \sum_{m=0}^{k} \frac{\lambda^m}{m!} & k \leq x < k+1 \text{ öyleki } 0 \leq k. \end{cases} \tag{18.8.14}$$

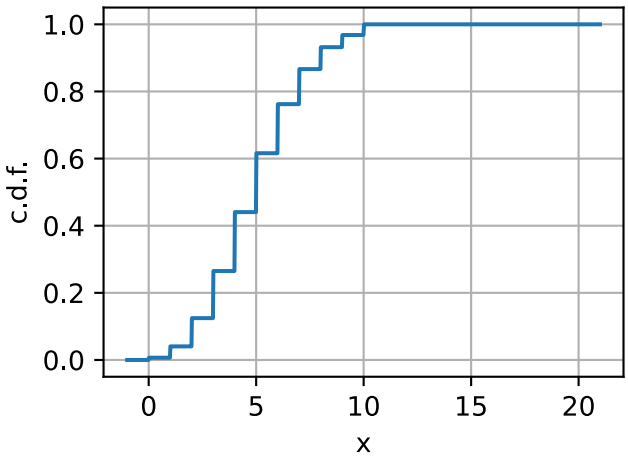
İlk olarak (18.8.13) olasılık kütle fonksiyonu çizelim.

```
lam = 5.0

xs = [i for i in range(20)]
pmf = torch.tensor([torch.exp(torch.tensor(-lam)) * lam**k
                    / factorial(k) for k in xs])

d2l.plt.stem(xs, pmf, use_line_collection=True)
d2l.plt.xlabel('x')
d2l.plt.ylabel('p.m.f.')
d2l.plt.show()
```



Şimdi, (18.8.14) birikimli dağılım fonksiyonunu çizelim.

```
x = torch.arange(-1, 21, 0.01)
cmf = torch.cumsum(pmf, dim=0)
def F(x):
    return 0 if x < 0 else 1 if x > n else cmf[int(x)]

d2l.plot(x, torch.tensor([F(y) for y in x.tolist()]), 'x', 'c.d.f.')
```



Yukarıda gördüğümüz gibi, ortalamalar ve varyanslar özellikle nettir. Eğer $X \sim \text{Poisson}(\lambda)$ ise, o zaman:

- $\mu_X = \lambda$,
- $\sigma_X^2 = \lambda$.

Bu aşağıdaki gibi örneklenebilir.

```
m = torch.distributions.poisson.Poisson(lam)
m.sample((10, 10))
```

```
tensor([[ 4.,  5.,  7.,  6.,  4.,  8.,  6.,  8.,  4.,  4.],
        [ 5.,  8.,  3.,  4.,  3.,  5.,  5.,  0.,  7.,  6.],
        [ 6.,  6.,  5.,  2., 11.,  7.,  6.,  4.,  7.,  3.],
        [ 6.,  2.,  6.,  1.,  6., 11.,  2.,  5.,  5.,  5.],
        [ 8.,  6.,  2.,  7.,  8.,  3.,  4., 11.,  3.,  4.],
        [ 6.,  7.,  5.,  4.,  4.,  5.,  4.,  3.,  3.,  6.],
        [ 4.,  3.,  3.,  8.,  6.,  3.,  4.,  3.,  7.,  5.],
        [ 7.,  7.,  5.,  5.,  6.,  4.,  2.,  5., 10.,  5.],
        [ 6.,  5.,  7.,  4.,  8.,  4.,  4.,  5.,  5.,  6.],
        [ 3.,  4.,  2.,  9.,  6.,  8.,  4.,  6.,  1.,  3.]])
```

18.8.6 Gauss Dağılımı

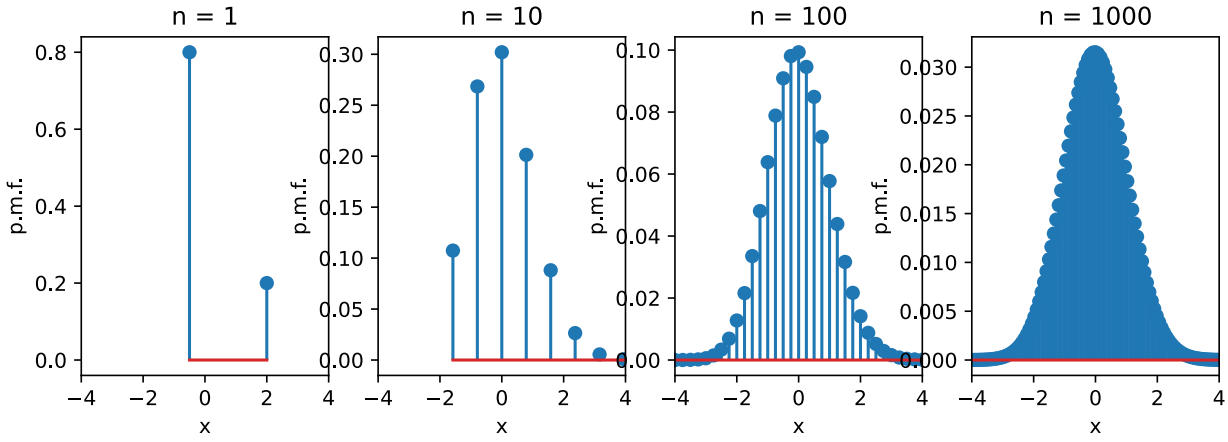
Şimdi farklı ama ilişkili bir deney yapalım. Tekrar n bağımsız Bernoulli(p) ölçümlerini, X_i'yi, gerçekleştirdiğimizi varsayalım. Bunların toplamının dağılımı $X^{(n)} \sim$ Binomial(n, p) şeklindedir. Burada n arttıkça ve p azaldıkça bir limit almak yerine, p'yi düzeltelim ve sonra $n \to \infty$ diyelim. Bu durumda $\mu_{X^{(n)}} = np \to \infty$ ve $\sigma^2_{X^{(n)}} = np(1-p) \to \infty$ olur, bu nedenle bu limitin iyi tanımlanması gerektiğini düşünmek için hiçbir neden yoktur.

Ancak, tüm umudumuzu kaybetmeyelim! Onları tanımlayarak ortalama ve varyansın iyi davranmasını sağlayalım:

$$Y^{(n)} = \frac{X^{(n)} - \mu_{X^{(n)}}}{\sigma_{X^{(n)}}}. \tag{18.8.15}$$

Burada ortalama sıfır ve varyans bir olduğu görülebilir ve bunun nedenle bazı sınırlayıcı dağılıma yakınsayacağına inanmak mantıklıdır. Bu dağılımların neye benzediğini çizersek, işe yarayacağına daha da ikna olacağız.

```
p = 0.2
ns = [1, 10, 100, 1000]
d2l.plt.figure(figsize=(10, 3))
for i in range(4):
    n = ns[i]
    pmf = torch.tensor([p**i * (1-p)**(n-i) * binom(n, i)
                        for i in range(n + 1)])
    d2l.plt.subplot(1, 4, i + 1)
    d2l.plt.stem([(i - n*p)/torch.sqrt(torch.tensor(n*p*(1 - p)))
                  for i in range(n + 1)], pmf,
                 use_line_collection=True)
    d2l.plt.xlim([-4, 4])
    d2l.plt.xlabel('x')
    d2l.plt.ylabel('p.m.f.')
    d2l.plt.title("n = {}".format(n))
d2l.plt.show()
```



Unutulmaması gereken bir şey: Poisson durumu ile karşılaştırıldığında, şimdi standart sapmaya bölüyoruz, bu da olası sonuçları gittikçe daha küçük alanlara sıkıştırdığımız anlamına gelir. Bu, limitimizin artık ayrık olmayacağının, bilakis sürekli olacağının bir göstergesidir.

Burada oluşan şeyin türetilmesi bu kitabın kapsamı dışındadır, ancak *merkezi limit teoremi*, $n \to \infty$ iken bunun Gauss Dağılımını (veya bazen normal dağılımı) vereceğini belirtir. Daha açık bir şekilde, herhangi bir a, b için:

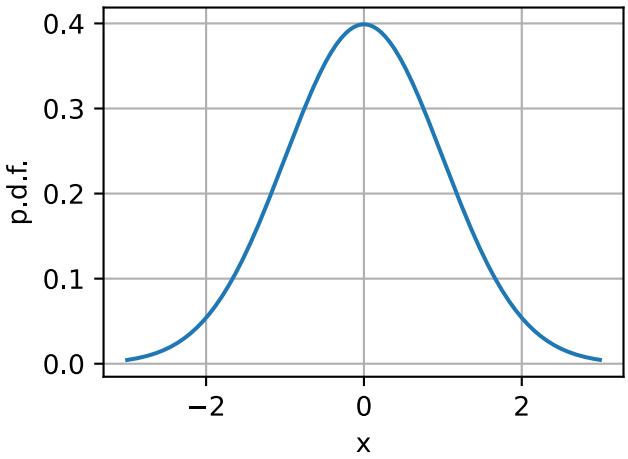
$$\lim_{n\to\infty} P(Y^{(n)} \in [a, b]) = P(\mathcal{N}(0, 1) \in [a, b]), \tag{18.8.16}$$

burada rastgele bir değişkenin normalde verilen ortalama μ ve varyans σ^2 ile dağıldığını söyleriz, $X \sim \mathcal{N}(\mu, \sigma^2)$ ise, X yoğunluğu şöyledir:

$$p_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}. \tag{18.8.17}$$

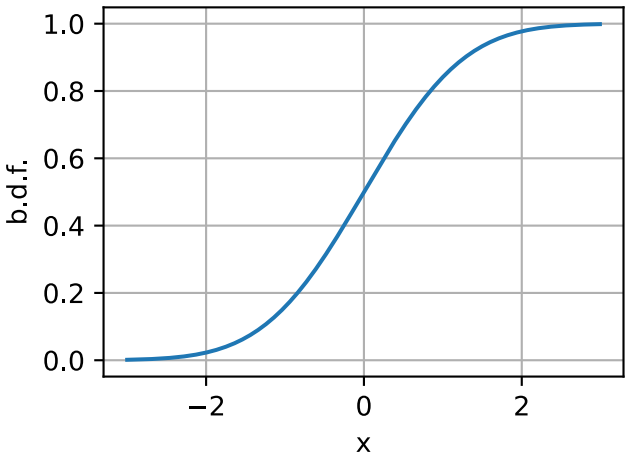
Önce, (18.8.17) olasılık yoğunluk dağılım fonksiyonunu çizelim.

```
mu, sigma = 0, 1

x = torch.arange(-3, 3, 0.01)
p = 1 / torch.sqrt(2 * torch.pi * sigma**2) * torch.exp(
    -(x - mu)**2 / (2 * sigma**2))

d2l.plot(x, p, 'x', 'p.d.f.')
```



Şimdi, birikimli dağılım fonksiyonunu çizelim. Bu ek bölümün kapsamı dışındadır, ancak Gauss b.d.f.'nun daha temel işlevlerden tanımlı kapalı-şekil formülü yoktur. Bu integrali sayısal olarak hesaplamanın bir yolunu sağlayan `erf`'i kullanacağız.

```
def phi(x):
    return (1.0 + erf((x - mu) / (sigma * torch.sqrt(torch.tensor(2.))))) / 2.0

d2l.plot(x, torch.tensor([phi(y) for y in x.tolist()]), 'x', 'b.d.f.')
```



Meraklı okuyucular bu terimlerin bazılarını tanıyacaktır. Aslında, bu integralla Section 18.5 içinde karşılaştık. Aslında, $p_X(x)$'nin toplamda bir birim alana sahip olduğunu ve dolayısıyla geçerli bir yoğunluk olduğunu görmek için tam olarak bu hesaplamaya ihtiyacımız var.

Bozuk para atmalarla çalışma seçimimiz, hesaplamaları kısalttı, ancak bu seçimle ilgili hiçbir şey temel (zorunlu) değildi. Gerçekten de, bağımsız aynı şekilde dağılmış rastgele değişkenlerden, X_i'den, oluşan herhangi bir koleksiyon alırsak ve aşağıdaki gibi hesaplarsak:

$$X^{(N)} = \sum_{i=1}^{N} X_i. \tag{18.8.18}$$

O zaman:

$$\frac{X^{(N)} - \mu_{X^{(N)}}}{\sigma_{X^{(N)}}} \tag{18.8.19}$$

yaklaşık olarak Gauss olacak. Çalışması için ek gereksinimler vardır, en yaygın olarak da $E[X^4] < \infty$, ancak işin felsefesi açıktır.

Merkezi limit teoremi, Gauss'un olasılık, istatistik ve makine öğrenmesi için temel olmasının nedenidir. Ölçtüğümüz bir şeyin birçok küçük bağımsız katkının toplamı olduğunu söyleyebildiğimizde, ölçülen şeyin Gauss'a yakın olacağını varsayabiliriz.

Gauss'ların daha birçok büyüleyici özelliği var ve burada bir tanesini daha tartışmak istiyoruz. Gauss, *maksimum entropi dağılımı* olarak bilinen şeydir. Entropiye daha derinlemesine Section 18.11 içinde gireceğiz, ancak bu noktada bilmemiz gereken tek şey bunun bir rastgelelik ölçüsü olduğudur. Titiz bir matematiksel anlamda, Gauss'u sabit ortalama ve varyanslı rastgele değişkenin *en* rastgele seçimi olarak düşünebiliriz. Bu nedenle, rastgele değişkenimizin herhangi bir ortalama ve varyansa sahip olduğunu bilirsek, Gauss bir anlamda yapabileceğimiz en muhafazakar dağılım seçimidir.

Bölümü kapatırken, $X \sim \mathcal{N}(\mu, \sigma^2)$ ise şunu hatırlayalım:

- $\mu_X = \mu$,
- $\sigma_X^2 = \sigma^2$.

Aşağıda gösterildiği gibi Gauss (veya standart normal) dağılımından örneklem alabiliriz.

```
torch.normal(mu, sigma, size=(10, 10))
```

```
tensor([[-0.2595, -1.5539,  0.0927, -1.2623, -0.8704,  0.0510,  0.5721, -0.3899,
         -0.2458, -0.9235],
        [ 1.4788,  1.0171, -1.1169,  1.0054, -1.1304,  0.8261, -1.5401,  0.2883,
         -0.3000, -0.3249],
        [ 1.0806, -0.8376, -0.0895, -1.2517, -0.1300, -0.8621, -0.1688,  0.4632,
          0.3710, -0.0961],
        [ 1.0172,  0.9544,  1.6106, -0.7210,  0.6563,  0.9079,  0.9978, -1.2329,
         -0.0525, -2.1334],
        [-0.1660,  1.1768,  1.3767, -1.1783, -0.3530,  1.4851,  0.1966, -0.3871,
          0.5587,  0.3439],
        [ 0.1132, -0.9124, -1.3493, -0.3536, -0.2566, -0.1056, -0.2690,  1.3901,
         -0.3605,  0.5091],
        [ 0.1190,  0.7673, -1.9223, -0.9960,  0.5085, -1.9418,  0.8535,  0.6506,
          1.1895,  1.7085],
        [ 1.5387, -0.4026,  0.4791, -1.4742, -1.0087,  1.7260, -0.6061, -0.3172,
         -1.5387, -1.0112],
        [-2.1433,  1.6243, -0.9645,  0.2574, -0.2288, -0.4926, -0.0305,  0.8721,
          0.1193,  0.1381],
        [ 0.2563, -0.6248, -0.7743,  0.6006, -0.8450, -0.4919,  1.1436, -0.1694,
         -1.0423, -1.7694]])
```

18.8.7 Üstel Ailesi

Yukarıda listelenen tüm dağılımlar için ortak bir özellik, hepsinin ait olduğu *üstel aile* olarak bilinmesidir. Üstel aile, yoğunluğu aşağıdaki biçimde ifade edilebilen bir dizi dağılımdır:

$$p(\mathbf{x}|\boldsymbol{\eta}) = h(\mathbf{x}) \cdot \exp\left(\boldsymbol{\eta}^\top \cdot T(\mathbf{x}) - A(\boldsymbol{\eta})\right) \tag{18.8.20}$$

Bu tanım biraz incelikli olabileceğinden, onu yakından inceleyelim.

İlk olarak, $h(\mathbf{x})$, *altta yatan ölçü* veya *temel ölçü* olarak bilinir. Bu, üstel ağırlığımızla değiştirdiğimiz orijinal bir ölçü seçimi olarak görülebilir.

İkinci olarak, *doğal parametreler* veya *kanonik parametreler* olarak adlandırılan $\boldsymbol{\eta} = (\eta_1, \eta_2, ..., \eta_l) \in \mathbb{R}^l$ vektörüne sahibiz. Bunlar, temel ölçünün nasıl değiştirileceğini tanımlar. Doğal parametreler, $\mathbf{x} = (x_1, x_2, ..., x_n) \in \mathbb{R}^n$ değişkeninin bazı $T(\cdot)$ fonksiyonlarına karşı bu parametrelerin nokta çarpımını ve üstünü alarak yeni ölçüye girerler. $T(\mathbf{x}) = (T_1(\mathbf{x}), T_2(\mathbf{x}), ..., T_l(\mathbf{x}))$ vektörüne *:math:`boldsymbol{eta}` için yeterli istatistikler* denir. Bu ad, $T(\mathbf{x})$ tarafından temsil edilen bilgiler olasılık yoğunluğunu hesaplamak için yeterli olduğundan ve $\mathbf{x}$'in örnekleminden başka hiçbir bilgi gerekmediğinden kullanılır.

Üçüncü olarak, elimizde yukarıdaki (18.8.20) dağılımının intergralinin bir olmasını sağlayan, birikim fonksiyonu olarak adlandırılan $A(\boldsymbol{\eta})$ var, yani:

$$A(\boldsymbol{\eta}) = \log\left[\int h(\mathbf{x}) \cdot \exp\left(\boldsymbol{\eta}^\top \cdot T(\mathbf{x})\right) d\mathbf{x}\right]. \tag{18.8.21}$$

Somut olmak için Gauss'u ele alalım. $\mathbf{x}$ öğesinin tek değişkenli bir değişken olduğunu varsayarsak, yoğunluğunun şu olduğunu gördük:

$$\begin{aligned} p(x|\mu,\sigma) &= \frac{1}{\sqrt{2\pi\sigma^2}} \cdot \exp\left\{\frac{-(x-\mu)^2}{2\sigma^2}\right\} \\ &= \frac{1}{\sqrt{2\pi}} \cdot \exp\left\{\frac{\mu}{\sigma^2}x - \frac{1}{2\sigma^2}x^2 - \left(\frac{1}{2\sigma^2}\mu^2 + \log(\sigma)\right)\right\}. \end{aligned} \tag{18.8.22}$$

Bu, üstel ailenin tanımıyla eşleşir:

- *Temel ölçü*: $h(x) = \frac{1}{\sqrt{2\pi}}$,
- *Doğal parametreler*: $\boldsymbol{\eta} = \begin{bmatrix} \eta_1 \\ \eta_2 \end{bmatrix} = \begin{bmatrix} \frac{\mu}{\sigma^2} \\ \frac{1}{2\sigma^2} \end{bmatrix}$,
- *Yeterli istatistikler*: $T(x) = \begin{bmatrix} x \\ -x^2 \end{bmatrix}$, and
- *Birikim işlevi*: $A(\boldsymbol{\eta}) = \frac{1}{2\sigma^2}\mu^2 + \log(\sigma) = \frac{\eta_1^2}{4\eta_2} - \frac{1}{2}\log(2\eta_2)$.

Yukarıdaki terimlerin her birinin kesin seçiminin biraz keyfi olduğunu belirtmekte fayda var. Gerçekten de önemli olan özellik, dağılımın tam formunun kendisi değil, bu formda ifade edilebilmesidir.

Section 3.4.6 içinde bahsettiğimiz gibi, yaygın olarak kullanılan bir teknik, $\mathbf{y}$ nihai çıktısının üstel bir aile dağılımını takip ettiğini varsaymaktır. Üstel aile, makine öğrenmesinde sıkça karşılaşılan yaygın ve güçlü bir dağılım ailesidir.

18.8.8 Özet

- Bernoulli rastgele değişkenleri, evet/hayır sonucu olan olayları modellemek için kullanılabilir.
- Ayrık tekdüze dağılım modeli, bir küme sınırlı olasılıktan seçim yapar.
- Sürekli tekdüze dağılımlar bir aralıktan seçim yapar.
- Binom dağılımları bir dizi Bernoulli rasgele değişkeni modeller ve başarıların sayısını sayar.
- Poisson rastgele değişkenleri, nadir olayların oluşunu modeller.
- Gauss rastgele değişkenleri, çok sayıda bağımsız rastgele değişkenin toplam sonucunu modeller.
- Yukarıdaki tüm dağılımlar üstel aileye aittir.

18.8.9 Alıştırmalar

1. İki bağımsız iki terimli rastgele değişkenin, $X, Y \sim \text{Binomial}(16, 1/2)$ arasındaki $X - Y$ farkı olan rastgele bir değişkenin standart sapması nedir?
2. Poisson rastgele değişkenini, $X \sim \text{Poisson}(\lambda)$, alırsak ve $(X - \lambda)/\sqrt{\lambda}$'i $\lambda \to \infty$ olarak kabul edersek, bunun yaklaşık olarak Gauss olduğunu gösterebiliriz. Bu neden anlamlıdır?
3. n elemanda tanımlı iki tane ayrık tekdüze rasgele değişkenin toplamı için olasılık kütle fonksiyonu nedir?

18.9 Naif (Saf) Bayes

Önceki bölümler boyunca, olasılık teorisi ve rastgele değişkenler hakkında bilgi edindik. Bu teoriyi uygulamaya koymak için, *naif Bayes* sınıflandırıcısını tanıtalım. Bu, rakamların sınıflandırmasını yapmamıza izin vermek için olasılık temellerinden başka hiçbir şey kullanmaz.

Öğrenme tamamen varsayımlarda bulunmakla ilgilidir. Daha önce hiç görmediğimiz yeni bir veri örneğini sınıflandırmak istiyorsak, hangi veri örneklerinin birbirine benzer olduğuna dair bazı varsayımlar yapmalıyız. Popüler ve oldukça net bir algoritma olan naif Bayes sınıflandırıcı, hesaplamayı basitleştirmek için tüm özniteliklerin birbirinden bağımsız olduğunu varsayar. Bu bölümde, imgelerdeki karakterleri tanımak için bu modeli uygulayacağız.

```
%matplotlib inline
import math
import torch
import torchvision
from d2l import torch as d2l

d2l.use_svg_display()
```

18.9.1 Optik Karakter Tanıma

MNIST (LeCun *et al.*, 1998), yaygın olarak kullanılan veri kümelerinden biridir. Eğitim için 60.000 imge ve geçerleme için 10.000 imge içerir. Her imge, 0'dan 9'a kadar el yazısıyla yazılmış bir rakam içerir. Görev, her imgeyi karşılık gelen rakama sınıflandırmaktır.

Gluon, veri kümesini İnternet'ten otomatik olarak almak için `data.vision` modülünde bir `MNIST` sınıfı sağlar. Daha sonra, Gluon hali-hazırda indirilmiş yerel kopyayı kullanacaktır. `train` parametresinin değerini sırasıyla `True` veya `False` olarak ayarlayarak eğitim kümesini mi yoksa test kümesini mi talep ettiğimizi belirtiriz. Her resim, hem genişliği hem de yüksekliği 28 olan ve $(28, 28, 1)$ şekilli gri tonlamalı bir resimdir. Son kanal boyutunu kaldırmak için özelleştirilmiş bir dönüşüm kullanıyoruz. Ek olarak, veri kümesi her pikseli işaretsiz 8 bitlik bir tamsayı ile temsil eder. Problemi basitleştirmek için bunları ikili öznitelikler halinde nicelendiriyoruz.

```
data_transform = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(),
    lambda x: torch.floor(x * 255 / 128).squeeze(dim=0)
])

mnist_train = torchvision.datasets.MNIST(
    root='./temp', train=True, transform=data_transform, download=True)
mnist_test = torchvision.datasets.MNIST(
    root='./temp', train=False, transform=data_transform, download=True)
```

İmgeyi ve ilgili etiketi içeren belirli bir örneğe erişebiliriz.

[224] https://discuss.d2l.ai/t/1098

```
image, label = mnist_train[2]
image.shape, label
```

```
(torch.Size([28, 28]), 4)
```

Burada image değişkeninde saklanan örneğimiz, yüksekliği ve genişliği 28 piksel olan bir imgeye karşılık gelir.

```
image.shape, image.dtype
```

```
(torch.Size([28, 28]), torch.float32)
```

Kodumuz her imgenin etiketini sayıl olarak depolar. Türü 32 bitlik bir tamsayıdır.

```
label, type(label)
```

```
(4, int)
```

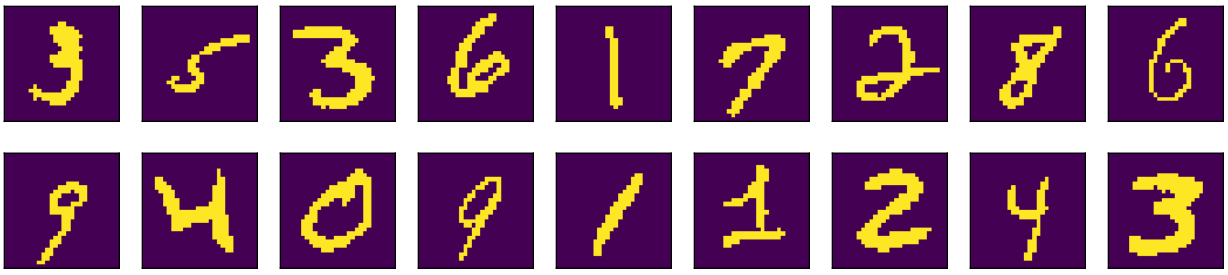
Aynı anda birden fazla örneğe de erişebiliriz.

```
images = torch.stack([mnist_train[i][0] for i in range(10, 38)], dim=0)
labels = torch.tensor([mnist_train[i][1] for i in range(10, 38)])
images.shape, labels.shape
```

```
(torch.Size([28, 28, 28]), torch.Size([28]))
```

Bu örnekleri görselleştirelim.

```
d2l.show_images(images, 2, 9);
```



18.9.2 Sınıflandırma için Olasılık Modeli

Bir sınıflandırma görevinde, bir örneği bir kategoriye eşleriz. Burada bir örnek gri tonlamalı 28×28 imge ve kategori bir rakamdır. (Daha ayrıntılı bir açıklama için bakınız Section 3.4.) Sınıflandırma görevini ifade etmenin doğal bir yolu, olasılık sorusudur: Özellikler (yani, imge pikselleri) verildiğinde en olası etiket nedir? $\mathbf{x} \in \mathbb{R}^d$ ile örneğin özniteliklerini ve $y \in \mathbb{R}$ ile etiketini belirtiriz. Burada öznitelikler, 2 boyutlu bir imgeyi $d = 28^2 = 784$ büyüklüğünde bir vektöre yeniden şekillendirebileceğimiz imge pikselleri ve etiketler rakamlardır. Öznitelikleri verilen etiketin olasılığı $p(y \mid \mathbf{x})$ şeklindedir. Örneğimizde $y = 0, \ldots, 9$ için $p(y \mid \mathbf{x})$ olan bu olasılıkları hesaplayabilirsek, sınıflandırıcı aşağıda verilen ifade ile tahminini, $\hat{y}$, yapacaktır:

$$\hat{y} = \text{argmax}\, p(y \mid \mathbf{x}). \tag{18.9.1}$$

Maalesef bu, her $\mathbf{x} = x_1, ..., x_d$ değeri için $p(y \mid \mathbf{x})$'yi tahmin etmemizi gerektirir. Her özniteliğin 2 değerden birini alabileceğini düşünün. Örneğin, $x_1 = 1$ özniteliği, elma kelimesinin belirli bir belgede göründüğünü ve $x_1 = 0$ görünmediğini belirtebilir. Eğer 30 tane bu tür ikili özniteliğe sahip olsaydık, bu $\mathbf{x}$ girdi vektörünün 2^{30} (1 milyardan fazla!) olası değerlerinden herhangi birini sınıflandırmaya hazırlıklı olmamız gerektiği anlamına gelirdi.

Dahası, öğrenme nerede? İlgili etiketi tahmin etmek için her bir olası örneği görmemiz gerekiyorsa, o zaman gerçekten bir model öğrenmiyoruz, sadece veri kümesini ezberliyoruz.

18.9.3 Naif Bayes Sınıflandırıcı

Neyse ki, koşullu bağımsızlık hakkında bazı varsayımlar yaparak, bazı tümevarımsal önyargılar sunabilir ve nispeten mütevazı bir eğitim örnekleri seçiminden genelleme yapabilen bir model oluşturabiliriz. Başlamak için, sınıflandırıcıyı şu şekilde ifade etmede Bayes teoremini kullanalım:

$$\hat{y} = \text{argmax}_y\, p(y \mid \mathbf{x}) = \text{argmax}_y\, \frac{p(\mathbf{x} \mid y)p(y)}{p(\mathbf{x})}. \tag{18.9.2}$$

Paydanın normalleştirme teriminin $p(\mathbf{x})$ olduğunu ve y etiketinin değerine bağlı olmadığını unutmayın. Sonuç olarak, sadece payı farklı y değerlerinde karşılaştırken endişelenmemiz gerekiyor. Paydanın hesaplanmasının zorlu olduğu ortaya çıksa bile, payı değerlendirebildiğimiz sürece onu görmezden gelerek kurtulabilirdik. Neyse ki, normalleştirme sabitini kurtarmak istesek, bunu da yapabilirdik. Normalleştirme terimini $\sum_y p(y \mid \mathbf{x}) = 1$ olduğundan her zaman kurtarabiliriz.

Şimdi $p(\mathbf{x} \mid y)$ üzerine odaklanalım. Zincir olasılık kuralını kullanarak $p(\mathbf{x} \mid y)$ terimini şu şekilde ifade edebiliriz:

$$p(x_1 \mid y) \cdot p(x_2 \mid x_1, y) \cdot ... \cdot p(x_d \mid x_1, ..., x_{d-1}, y). \tag{18.9.3}$$

Tek başına bu ifade bizi daha ileriye götürmez. Yine de kabaca 2^d tane parametreyi tahmin etmeliyiz. Bununla birlikte, *etiketi verildiğinde özniteliklerin koşullu olarak birbirinden bağımsız olduğunu varsayarsak*, aniden çok daha iyi durumda oluruz, çünkü bu terim $\prod_i p(x_i \mid y)$'a sadeleşerek bize şu tahminciyi verir:

$$\hat{y} = \text{argmax}_y \prod_{i=1}^{d} p(x_i \mid y)p(y). \tag{18.9.4}$$

Her i ve y için $p(x_i = 1 \mid y)$ tahmin edebilir ve değerini $P_{xy}[i, y]$ olarak kaydedebilirsek, burada P_{xy} n sınıf sayısı ve $y \in \{1, \ldots, n\}$ olan bir $d \times n$ matrisidir, o zaman bunu $p(x_i = 0 \mid y)$'i tahmin

etmek için de kullanabiliriz, yani

$$p(x_i = t_i \mid y) = \begin{cases} P_{xy}[i, y] & \text{öyle ki } t_i = 1; \\ 1 - P_{xy}[i, y] & \text{öyle ki } t_i = 0. \end{cases} \tag{18.9.5}$$

Ayrıca, her y için $p(y)$ tahmininde bulunur ve n uzunluğunda bir P_y vektörü ile, $P_y[y]$'a kaydederiz. Ardından, herhangi bir yeni örnek $\mathbf{t} = (t_1, t_2, \ldots, t_d)$ için şunu hesaplayabiliriz;

$$\begin{aligned} \hat{y} &= \text{argmax}_y \, p(y) \prod_{i=1}^{d} p(x_t = t_i \mid y) \\ &= \text{argmax}_y \, P_y[y] \prod_{i=1}^{d} \, P_{xy}[i, y]^{t_i} \, (1 - P_{xy}[i, y])^{1 - t_i} \end{aligned} \tag{18.9.6}$$

üstelik herhangi bir y için. Dolayısıyla, koşullu bağımsızlık varsayımımız, modelimizin karmaşıklığını öznitelik sayısına bağlı $\mathcal{O}(2^d n)$ üstel bir bağımlılıktan $\mathcal{O}(dn)$ olan doğrusal bir bağımlılığa almıştır.

18.9.4 Eğitim

Şimdi sorun, P_{xy} ve P_y'yi bilmiyor olmamızdır. Bu nedenle, önce bazı eğitim verileri verildiğinde bu değerleri tahmin etmemiz gerekiyor. Bu, modeli *eğitmektir*. P_y'yi tahmin etmek çok zor değil. Sadece 10 sınıfla uğraştığımız için, her rakam için görülme sayısını, n_y, sayabilir ve bunu toplam veri miktarına n bölebiliriz. Örneğin, 8 rakamı $n_8 = 5.800$ kez ortaya çıkarsa ve toplam $n = 60.000$ imgemiz varsa, olasılık tahminimiz $p(y = 8) = 0.0967$ olur.

```
X = torch.stack([mnist_train[i][0] for i in range(len(mnist_train))], dim=0)
Y = torch.tensor([mnist_train[i][1] for i in range(len(mnist_train))])

n_y = torch.zeros(10)
for y in range(10):
    n_y[y] = (Y == y).sum()
P_y = n_y / n_y.sum()
P_y
```

```
tensor([0.0987, 0.1124, 0.0993, 0.1022, 0.0974, 0.0904, 0.0986, 0.1044, 0.0975,
        0.0992])
```

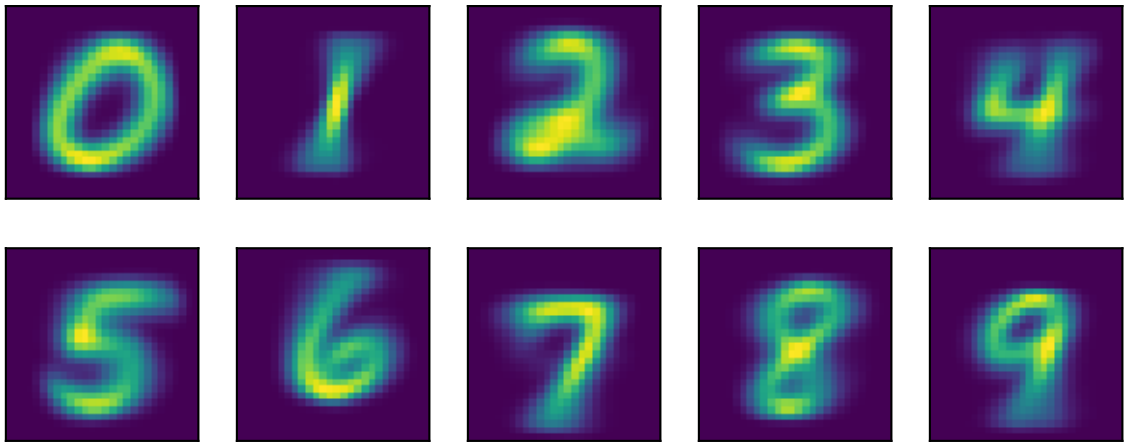
Şimdi biraz daha zor şeylere, P_{xy}'ye, geçelim. Siyah beyaz imgeler seçtiğimiz için, $p(x_i \mid y)$, i pikselinin y sınıfı için açık olma olasılığını gösterir. Tıpkı daha önce olduğu gibi, bir olayın meydana geldiği n_{iy} sayısını sayabilmemiz ve bunu y'nin toplam oluş sayısına bölebilmemiz gibi, yani n_y. Ancak biraz rahatsız edici bir şey var: Belirli pikseller asla siyah olmayabilir (örneğin, iyi kırpılmış imgelerde köşe pikselleri her zaman beyaz olabilir). İstatistikçilerin bu sorunla baş etmeleri için uygun bir yol, tüm oluşumlara sözde sayımlar eklemektir. Bu nedenle, n_{iy} yerine $n_{iy} + 1$ ve n_y yerine $n_y + 1$ kullanıyoruz. Bu aynı zamanda *Laplace Düzleştirme (Smoothing)* olarak da adlandırılır. Geçici görünebilir, ancak Bayesci bir bakış açısından iyi motive edilmiş olabilir.

```
n_x = torch.zeros((10, 28, 28))
for y in range(10):
    n_x[y] = torch.tensor(X.numpy()[Y.numpy() == y].sum(axis=0))
```

(continues on next page)

(continued from previous page)

```
P_xy = (n_x + 1) / (n_y + 1).reshape(10, 1, 1)

d2l.show_images(P_xy, 2, 5);
```



Bu $10 \times 28 \times 28$ olasılıkları görselleştirerek (her sınıf için her piksel için) ortalama görünümlü rakamlar elde edebiliriz.

Şimdi yeni bir imgeyi tahmin etmek için (18.9.6) denklemini kullanabiliriz. $\mathbf{x}$ verildiğinde, aşağıdaki işlevler her y için $p(\mathbf{x} \mid y)p(y)$'yi hesaplar.

```
def bayes_pred(x):
    x = x.unsqueeze(0)  # (28, 28) -> (1, 28, 28)
    p_xy = P_xy * x + (1 - P_xy)*(1 - x)
    p_xy = p_xy.reshape(10, -1).prod(dim=1)  # p(x|y)
    return p_xy * P_y

image, label = mnist_test[0]
bayes_pred(image)
```

```
tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

Bu korkunç bir şekilde yanlış gitti! Nedenini bulmak için piksel başına olasılıklara bakalım. Bunlar tipik 0.001 ile 1 arasındaki sayılardır. 784 tanesini çarpıyoruz. Bu noktada, bu sayıları bir bilgisayarda sabit bir aralıkla hesapladığımızı belirtmekte fayda var, dolayısıyla bu kuvvet için de geçerli. Olan şu ki, *sayısal küçümenlik (underflow)* yaşıyoruz, yani tüm küçük sayıları çarpmak, sıfıra yuvarlanana kadar daha da küçük değerlere yol açar. Bunu teorik bir mesele olarak :numref:`sec_maximum_likelihood içinde tartıştık, ancak burada pratikteki bir vaka olarak açıkça görüyoruz.

O bölümde tartışıldığı gibi, bunu $\log ab = \log a + \log b$ gerçeğini kullanarak, yani logaritma toplamaya geçerek düzeltiriz. Hem a hem de b küçük sayılar olsa bile, logaritma değerleri uygun bir aralıkta olacaktır.

```
a = 0.1
print('underflow:', a**784)
print('logarithm is normal:', 784*math.log(a))
```

```
underflow: 0.0
logarithm is normal: -1805.2267129073316
```

Logaritma artan bir fonksiyon olduğundan, (18.9.6) denklemini şu şekilde yeniden yazabiliriz:

$$\hat{y} = \operatorname{argmax}_y \ \log P_y[y] + \sum_{i=1}^{d} \Big[t_i \log P_{xy}[x_i, y] + (1 - t_i) \log(1 - P_{xy}[x_i, y])\Big]. \tag{18.9.7}$$

Aşağıdaki kararlı sürümü uygulayabiliriz:

```
log_P_xy = torch.log(P_xy)
log_P_xy_neg = torch.log(1 - P_xy)
log_P_y = torch.log(P_y)

def bayes_pred_stable(x):
    x = x.unsqueeze(0)  # (28, 28) -> (1, 28, 28)
    p_xy = log_P_xy * x + log_P_xy_neg * (1 - x)
    p_xy = p_xy.reshape(10, -1).sum(axis=1)  # p(x|y)
    return p_xy + log_P_y

py = bayes_pred_stable(image)
py
```

```
tensor([-269.0042, -301.7345, -245.2146, -218.8941, -193.4691, -206.1031,
        -292.5432, -114.6283, -220.3562, -163.1888])
```

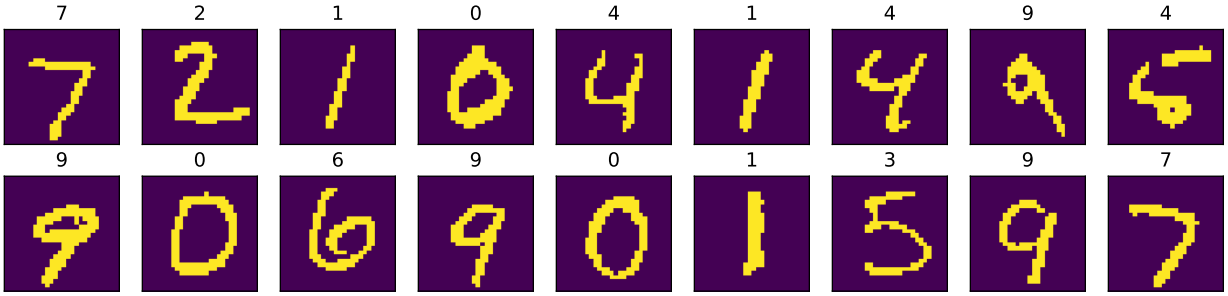
Şimdi tahminin doğru olup olmadığını kontrol edebiliriz.

```
py.argmax(dim=0) == label
```

```
tensor(True)
```

Şimdi birkaç geçerleme örneği tahmin edersek, Bayes sınıflandırıcısının oldukça iyi çalıştığını görebiliriz.

```
def predict(X):
    return [bayes_pred_stable(x).argmax(dim=0).type(torch.int32).item()
            for x in X]

X = torch.stack([mnist_test[i][0] for i in range(18)], dim=0)
y = torch.tensor([mnist_test[i][1] for i in range(18)])
preds = predict(X)
d2l.show_images(X, 2, 9, titles=[str(d) for d in preds]);
```



Son olarak, sınıflandırıcının genel doğruluğunu hesaplayalım.

```
X = torch.stack([mnist_test[i][0] for i in range(len(mnist_test))], dim=0)
y = torch.tensor([mnist_test[i][1] for i in range(len(mnist_test))])
preds = torch.tensor(predict(X), dtype=torch.int32)
float((preds == y).sum()) / len(y)  # Validation accuracy
```

```
0.8426
```

Modern derin ağlar 0.01'den daha düşük hata oranlarına ulaşır. Nispeten düşük performans, modelimizde yaptığımız yanlış istatistiksel varsayımlardan kaynaklanmaktadır: Her pikselin yalnızca etikete bağlı olarak *bağımsızca* oluşturulduğunu varsaydık. İnsanların rakamları böyle yazmadığı açıktır ve bu yanlış varsayım, aşırı naif (Bayes) sınıflandırıcımızın çökmesine yol açtı.

18.9.5 Özet

- Bayes kuralı kullanılarak, gözlenen tüm özniteliklerin bağımsız olduğu varsayılarak bir sınıflandırıcı yapılabilir.
- Bu sınıflandırıcı, etiket ve piksel değerlerinin kombinasyonlarının olma sayısını sayarak bir veri kümesi üzerinde eğitilebilir.
- Bu sınıflandırıcı, istenmeyen elektronik posta (spam) tespiti gibi görevler için onlarca yıldır altın standarttı.

18.9.6 Alıştırmalar

1. $[[0,0],[0,1],[1,0],[1,1]]$ veri kümesini iki öğenin XOR tarafından verilen etiketleri ile, $[0,1,1,0]$, düşünün. Bu veri kümesine dayanan bir naif Bayes sınıflandırıcısının olasılıkları nelerdir? Noktalarımızı başarıyla sınıflandırıyor mu? Değilse, hangi varsayımlar ihlal edilir?
2. Olasılıkları tahmin ederken Laplace düzleştirmeyi kullanmadığımızı ve eğitimde asla gözlenmeyen bir değer içeren bir veri örneğinin test zamanında geldiğini varsayalım. Model ne çıkarır?
3. Naif Bayes sınıflandırıcısı, rastgele değişkenlerin bağımlılığının bir grafik yapısıyla kodlandığı belirli bir Bayes ağı örneğidir. Tam teorisi bu bölümün kapsamı dışında olsa da (tüm ayrıntılar için (Koller and Friedman, 2009) çalışmasına bakınız), XOR modelinde iki girdi değişkeni arasında açık bağımlılığa izin vermenin neden başarılı bir sınıflandırıcı oluşturmaya izin verdiğini açıklayınız.

Tartışmalar[225]

18.10 İstatistik

Kuşkusuz, en iyi derin öğrenme uygulayıcılarından biri olmak için son teknoloji ürünü ve yüksek doğrulukta modelleri eğitme yeteneği çok önemlidir. Bununla birlikte, iyileştirmelerin ne zaman önemli olduğu veya yalnızca eğitim sürecindeki rastgele dalgalanmaların sonucu olduğu genellikle belirsizdir. Tahmini değerlerdeki belirsizliği tartışabilmek için biraz istatistik öğrenmemiz gerekir.

İstatistiğin en eski referansı, şifrelenmiş mesajları deşifre etmek için istatistiklerin ve sıklık analizinin nasıl kullanılacağına dair ayrıntılı bir açıklama veren $9.$ yüzyıldaki Arap bilim adamı Al-Kindi'ye kadar uzanabilir. 800 yıl sonra, modern istatistik, araştırmacıların demografik ve ekonomik veri toplama ve analizine odaklandığı 1700'lerde Almanya'da ortaya çıktı. Günümüzde istatistik, verilerin toplanması, işlenmesi, analizi, yorumlanması ve görselleştirilmesi ile ilgili bilim konusudur. Dahası, temel istatistik teorisi akademi, endüstri ve hükümet içindeki araştırmalarda yaygın olarak kullanılmaktadır.

Daha özel olarak, istatistik *tanımlayıcı istatistik* ve *istatistiksel çıkarım* diye bölünebilir. İlki, *örneklem* olarak adlandırılan gözlemlenen verilerden bir koleksiyonunun özniteliklerini özetlemeye ve göstermeye odaklanır. Örneklem bir *popülasyondan* alınmıştır, benzer bireyler, öğeler veya deneysel ilgi alanlarımıza ait olayların toplam kümesini belirtir. Tanımlayıcı istatistiğin aksine *istatistiksel çıkarım*, örneklem dağılımının popülasyon dağılımını bir dereceye kadar kopyalayabileceği varsayımlarına dayanarak, bir popülasyonun özelliklerini verilen *örneklemlerden* çıkarsar.

Merak edebilirsiniz: "Makine öğrenmesi ile istatistik arasındaki temel fark nedir?" Temel olarak, istatistik çıkarım sorununa odaklanır. Bu tür problemler, nedensel çıkarım gibi değişkenler arasındaki ilişkiyi modellemeyi ve A/B testi gibi model parametrelerinin istatistiksel olarak anlamlılığını test etmeyi içerir. Buna karşılık, makine öğrenmesi, her bir parametrenin işlevselliğini açıkça programlamadan ve anlamadan doğru tahminler yapmaya vurgu yapar.

Bu bölümde, üç tür istatistik çıkarım yöntemini tanıtacağız: Tahmin edicileri değerlendirme ve karşılaştırma, hipotez (denence) testleri yürütme ve güven aralıkları oluşturma. Bu yöntemler, belirli bir popülasyonun özelliklerini, yani gerçek θ parametresi gibi, anlamamıza yardımcı olabilir. Kısacası, belirli bir popülasyonun gerçek parametresinin, θ, skaler bir değer olduğunu varsayıyoruz. θ'nin bir vektör veya tensör olduğu durumu genişletmek basittir, bu nedenle tartışmamızda onu es geçiyoruz.

18.10.1 Tahmincileri Değerlendirme ve Karşılaştırma

İstatistikte, bir *tahminci*, gerçek θ parametresini tahmin etmek için kullanılan belirli örneklemlerin bir fonksiyonudur. $\{x_1, x_2, \ldots, x_n\}$ örneklerini gözlemledikten sonra θ tahmini için $\hat{\theta}_n = \hat{f}(x_1, \ldots, x_n)$ yazacağız.

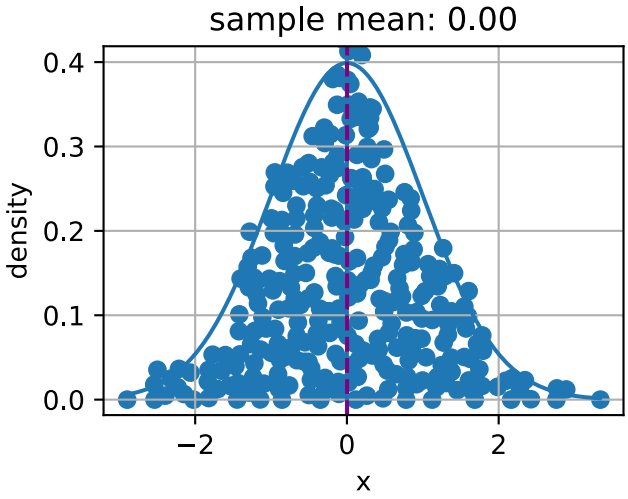
Tahmincilerin basit örneklerini daha önce şu bölümde görmüştük Section 18.7. Bir Bernoulli rastgele değişkeninden birkaç örneğiniz varsa, rastgele değişkenin olma olasılığı için maksimum olabilirlik tahmini, gözlemlenenlerin sayısını sayarak ve toplam örnek sayısına bölerek elde edilebilir. Benzer şekilde, bir alıştırma sizden bir miktar örnek verilen bir Gauss'un ortalamasının

[225] https://discuss.d2l.ai/t/1100

maksimum olabilirlik tahmininin tüm örneklerin ortalama değeriyle verildiğini göstermenizi istiyor. Bu tahminciler neredeyse hiçbir zaman parametrenin gerçek değerini vermezler, ancak ideal olarak çok sayıda örnek için tahmin yakın olacaktır.

Örnek olarak, ortalama sıfır ve varyans bir olan bir Gauss rasgele değişkeninin gerçek yoğunluğunu, bu Gauss'tan bir dizi örnek ile aşağıda gösteriyoruz. Her noktanın y koordinatı görünür ve orijinal yoğunluk ile olan ilişki daha net fark edilecek şekilde oluşturduk.

```
import torch
from d2l import torch as d2l

torch.pi = torch.acos(torch.zeros(1)) * 2  #define pi in torch

# Örnek veri noktaları ve y koordinatı oluşturun
epsilon = 0.1
torch.manual_seed(8675309)
xs = torch.randn(size=(300,))

ys = torch.tensor(
    [torch.sum(torch.exp(-(xs[:i] - xs[i])**2 / (2 * epsilon**2))\
               / torch.sqrt(2*torch.pi*epsilon**2)) / len(xs)\
     for i in range(len(xs))])

# Gerçek yoğunluğu hesapla
xd = torch.arange(torch.min(xs), torch.max(xs), 0.01)
yd = torch.exp(-xd**2/2) / torch.sqrt(2 * torch.pi)

# Sonuçları çiz
d2l.plot(xd, yd, 'x', 'density')
d2l.plt.scatter(xs, ys)
d2l.plt.axvline(x=0)
d2l.plt.axvline(x=torch.mean(xs), linestyle='--', color='purple')
d2l.plt.title(f'sample mean: {float(torch.mean(xs).item()):.2f}')
d2l.plt.show()
```



$:raw-latex:*hat{theta}*_n $ parametresinin bir tahmincisini hesaplamanın birçok yolu olabilir. Bu bölümde, tahmincileri değerlendirmek ve karşılaştırmak için üç genel yöntem sunuyoruz: Ortalama hata karesi, standart sapma ve istatistiksel yanlılık.

Ortalama Hata Karesi

Tahmin edicileri değerlendirmek için kullanılan en basit ölçüt, bir tahmincinin *ortalama hata karesi (MSE)* (veya l_2 kaybı) olarak tanımlanabilir.

$$\text{MSE}(\hat{\theta}_n, \theta) = E[(\hat{\theta}_n - \theta)^2]. \tag{18.10.1}$$

Bu, gerçek değerden ortalama kare sapmayı ölçümlemizi sağlar. MSE her zaman negatif değildir. Eğer Section 3.1 içinde okuduysanız, bunu en sık kullanılan bağlanım (regresyon) kaybı işlevi olarak tanıyacaksınız. Bir tahminciyi değerlendirmek için bir ölçü olarak, değeri sıfıra ne kadar yakınsa, tahminci gerçek θ parametresine o kadar yakın olur.

İstatistiksel Yanlılık

MSE doğal bir ölçü sağlar, ancak onu büyük yapabilecek birden fazla farklı vakayı kolayca hayal edebiliriz. İki temel önemli olay veri kümesindeki rastgelelik nedeniyle tahmincideki dalgalanma ve tahmin prosedürüne bağlı olarak tahmincideki sistematik hatadır.

Öncelikle sistematik hatayı ölçelim. Bir $:raw-latex:*hat{theta}*_n $ tahmincisi için *istatistiksel yanlılığın* matematiksel gösterimi şu şekilde tanımlanabilir:

$$\text{yanlılık}(\hat{\theta}_n) = E(\hat{\theta}_n - \theta) = E(\hat{\theta}_n) - \theta. \tag{18.10.2}$$

$\text{yanlılık}(\hat{\theta}_n) = 0$ olduğunda, $\hat{\theta}_n$ tahmin edicisinin beklentisinin parametrenin gerçek değerine eşit olduğuna dikkat edin. Bu durumda, $\hat{\theta}_n$'nin yansız bir tahminci olduğunu söylüyoruz. Genel olarak, yansız bir tahminci, yanlı bir tahminciden daha iyidir çünkü beklenen değeri gerçek parametre ile aynıdır.

Bununla birlikte, yanlı tahmin edicilerin pratikte sıklıkla kullanıldığının farkında olunması gerekir. Yansız tahmin edicilerin başka varsayımlar olmaksızın var olmadığı veya hesaplamanın zor olduğu durumlar vardır. Bu, bir tahmincide önemli bir kusur gibi görünebilir, ancak pratikte karşılaşılan tahmin edicilerin çoğu, mevcut örneklerin sayısı sonsuza giderken sapmanın sıfır olma eğiliminde olması açısından en azından asimptotik (kavuşma doğrusu) olarak tarafsızdır: $\lim_{n \to \infty} \text{bias}(\hat{\theta}_n) = 0$.

Varyans ve Standart Sapma

İkinci olarak tahmincideki rastgeleliği ölçelim. Eğer Section 18.6 bölümünü anımsarsak, *standart sapma* (veya *standart hata*), varyansın kare kökü olarak tanımlanır. Bir tahmincinin dalgalanma derecesini, o tahmincinin standart sapmasını veya varyansını ölçerek ölçebiliriz.

$$\sigma_{\hat{\theta}_n} = \sqrt{\text{Var}(\hat{\theta}_n)} = \sqrt{E[(\hat{\theta}_n - E(\hat{\theta}_n))^2]}. \tag{18.10.3}$$

Şunları karşılaştırmak önemlidir (18.10.3) ile (18.10.1). Bu denklemde gerçek popülasyon değeri θ ile değil, bunun yerine beklenen örneklem ortalaması $E(\hat{\theta}_n)$ ile karşılaştırıyoruz. Bu nedenle, tahmincinin gerçek değerden ne kadar uzakta olduğunu ölçmüyoruz, bunun yerine tahmincinin dalgalanmasını ölçüyoruz.

Yanlılık-Varyans Ödünleşmesi

Bu iki ana bileşenin ortalama hata karesine (MSE) katkıda bulunduğu sezgisel olarak açıktır. Biraz şok edici olan şey, bunun aslında ortalama hata karesinin bu iki ve ek üçüncü bir parçaya *ayrıştırılması* olduğunu gösterebilmemizdir. Yani, ortalama hata karesini yanlılığın (ek girdi) karesinin, varyansın ve indirgenemeyen hatanın toplamı olarak yazabiliriz.

$$\begin{aligned}\mathrm{MSE}(\hat{\theta}_n, \theta) &= E[(\hat{\theta}_n - \theta)^2] \\ &= E[(\hat{\theta}_n)^2] + E[\theta^2] - 2E[\hat{\theta}_n\theta] \\ &= \mathrm{Var}[\hat{\theta}_n] + E[\hat{\theta}_n]^2 + \mathrm{Var}[\theta] + E[\theta]^2 - 2E[\hat{\theta}_n]E[\theta] \\ &= (E[\hat{\theta}_n] - E[\theta])^2 + \mathrm{Var}[\hat{\theta}_n] + \mathrm{Var}[\theta] \\ &= (E[\hat{\theta}_n - \theta])^2 + \mathrm{Var}[\hat{\theta}_n] + \mathrm{Var}[\theta] \\ &= (\text{yanlılık}[\hat{\theta}_n])^2 + \mathrm{Var}(\hat{\theta}_n) + \mathrm{Var}[\theta].\end{aligned} \tag{18.10.4}$$

Yukarıdaki formülü *yanlılık-varyans ödünleşmesi* olarak adlandırıyoruz. Ortalama hata karesi kesin olarak üç hata kaynağına bölünebilir: Yüksek yanlılıktan, yüksek varyanstan ve indirgenemez hatadan kaynaklı hata. Yanlılık hatası genellikle basit bir modelde (doğrusal bağlanım modeli gibi) görülür, çünkü öznitelikler ve çıktılar arasındaki yüksek boyutsal ilişkileri çıkaramaz. Bir model yüksek yanlılık hatasından muzdaripse, (Section 4.4) bölümünde açıklandığı gibi genellikle *eksik öğrenme* veya *esneklik* eksikliği olduğunu söylüyoruz. Yüksek varyans, genellikle eğitim verilerine öğrenen çok karmaşık bir modelden kaynaklanır. Sonuç olarak, *aşırı öğrenen* bir model, verilerdeki küçük dalgalanmalara duyarlıdır. Bir modelin varyansı yüksekse, genellikle (Section 4.4) içinde tanıtıldığı gibi *aşırı öğrenme* ve *genelleme* yoksunluğu olduğunu söyleriz. İndirgenemez hata, θ'nın kendisindeki gürültünün sonucudur.

Kodda Tahmincileri Değerlendirme

Bir tahmincinin standart sapması, bir tensör `a` için basitçe `a.std()` çağırarak uygulandığından, onu atlayacağız ancak istatistiksel yanlılık ve ortalama hata karesini uygulayacağız.

```
# İstatistiksel yanlılık
def stat_bias(true_theta, est_theta):
    return(torch.mean(est_theta) - true_theta)

# Ortalama kare hatası
def mse(data, true_theta):
    return(torch.mean(torch.square(data - true_theta)))
```

Yanlılık-varyans ödünleşmesinin denklemini görsellemek için, $\mathcal{N}(\theta, \sigma^2)$ normal dağılımını 10.000 örnekle canlandıralım. Burada $\theta = 1$ ve $\sigma = 4$ olarak kullanıyoruz. Tahminci verilen örneklerin bir fonksiyonu olduğu için, burada örneklerin ortalamasını bu normal dağılımdaki, $\mathcal{N}(\theta, \sigma^2)$, gerçek θ için bir tahminci olarak kullanıyoruz.

```
theta_true = 1
sigma = 4
sample_len = 10000
samples = torch.normal(theta_true, sigma, size=(sample_len, 1))
theta_est = torch.mean(samples)
theta_est
```

```
tensor(1.0170)
```

Tahmincimizin yanlılık karesi ve varyansının toplamını hesaplayarak ödünleşme denklemini doğrulayalım. İlk önce, tahmincimizin MSE'sini hesaplayın.

```
mse(samples, theta_true)
```

```
tensor(16.0298)
```

Ardından, aşağıdaki gibi $\text{Var}(\hat{\theta}_n) + [\text{yanlılık}(\hat{\theta}_n)]^2$'yi hesaplıyoruz. Gördüğünüz gibi, iki değer, sayısal kesinliğe uyuyor.

```
bias = stat_bias(theta_true, theta_est)
torch.square(samples.std(unbiased=False)) + torch.square(bias)
```

```
tensor(16.0298)
```

18.10.2 Hipotez (Denence) Testleri Yürütme

İstatistiksel çıkarımda en sık karşılaşılan konu hipotez testidir. Hipotez testi 20. yüzyılın başlarında popüler hale gelirken, ilk kullanım 1700'lerde John Arbuthnot'a kadar takip edilebilir. John, Londra'da 80 yıllık doğum kayıtlarını takip etti ve her yıl kadından daha fazla erkeğin doğduğu sonucuna vardı. Bunu takiben, modern anlamlılık testi, p-değerini ve Pearson'ın ki-kare testini icat eden Karl Pearson, Student (öğrenci) t dağılımının babası William Gosset ve sıfır hipotezini ve anlamlılık testini başlatan Ronald Fisher'ın zeka mirasıdır.

Hipotez testi, bir popülasyon hakkındaki varsayılan ifadeye karşı bazı kanıtları değerlendirmenin bir yoludur. Varsayılan ifadeyi, gözlemlenen verileri kullanarak reddetmeye çalıştığımız, *sıfır hipotezi*, H_0, olarak adlandırıyoruz. Burada, istatistiksel anlamlılık testi için başlangıç noktası olarak H_0'ı kullanıyoruz. *Alternatif hipotez* H_A (veya H_1), sıfır hipotezine karşıt bir ifadedir. Bir sıfır hipotez, genellikle değişkenler arasında bir ilişki olduğunu varsayan açıklayıcı bir biçimde ifade edilir. İçeriğini olabildiğince açık bir şekilde yansıtmalı ve istatistik teorisi ile test edilebilir olmalıdır.

Kimyager olduğunuzu hayal edin. Laboratuvarda binlerce saat geçirdikten sonra, kişinin matematiği anlama yeteneğini önemli ölçüde arttırabilecek yeni bir ilaç geliştiriyorsunuz. Sihirli gücünü göstermek için onu test etmeniz gerekir. Doğal olarak, ilacı almak ve matematiği daha iyi öğrenmelerine yardımcı olup olmayacağını görmek için bazı gönüllülere ihtiyacınız olabilir. Nasıl başlayacaksınız?

İlk olarak, dikkatle rastgele seçilmiş iki grup gönüllüye ihtiyacınız olacak, böylece bazı ölçütlerle ölçülen matematik anlama yetenekleri arasında hiçbir fark olmayacak. Bu iki grup genellikle test grubu ve kontrol grubu olarak adlandırılır. *Test grubu* (veya *tedavi grubu*) ilacı deneyimleyecek bir grup kişidir, *kontrol grubu* ise bir kıyaslama olarak bir kenara bırakılan kullanıcı grubunu temsil eder, yani, ilaç almak dışında aynı ortam şartlarına sahipler. Bu şekilde, bağımsız değişkenin tedavideki etkisi dışında tüm değişkenlerin etkisi en aza indirilir.

İkincisi, ilacı bir süre aldıktan sonra, iki grubun matematik anlayışını, yeni bir matematik formülü öğrendikten sonra gönüllülerin aynı matematik testlerini yapmasına izin vermek gibi

aynı ölçütlerle ölçmeniz gerekecektir. Ardından, performanslarını toplayabilir ve sonuçları karşılaştırabilirsiniz. Bu durumda, sıfır hipotezimiz, muhtemelen iki grup arasında hiçbir fark olmadığı ve alternatifimiz olduğu şeklinde olacaktır.

Bu hala tam olarak resmi (nizamlara uygun) değil. Dikkatlice düşünmeniz gereken birçok detay var. Örneğin, matematik anlama yeteneklerini test etmek için uygun ölçütler nelerdir? İlacınızın etkinliğini iddia edebileceğinizden emin olabilmeniz için testinizde kaç gönüllü var? Testi ne kadar süreyle koşturmalısınız? İki grup arasında bir fark olup olmadığına nasıl karar veriyorsunuz? Yalnızca ortalama performansla mı ilgileniyorsunuz, yoksa puanların değişim aralığını da mı önemsiyorsunuz? Ve bunun gibi.

Bu şekilde, hipotez testi, deneysel tasarım ve gözlemlenen sonuçlarda kesinlik hakkında akıl yürütme için bir çerçeve sağlar. Şimdi sıfır hipotezinin gerçek olma ihtimalinin çok düşük olduğunu gösterebilirsek, onu güvenle reddedebiliriz.

Hipotez testiyle nasıl çalışılacağına dair hikayeyi tamamlamak için, şimdi bazı ek terminolojiyle tanışmamız ve yukarıdaki bazı kavramlarımızı kurallara uygun halde işlememiz gerekiyor.

İstatistiksel Anlamlılık

İstatistiksel anlamlılık, sıfır hipotezin, H_0, reddedilmemesi gerektiğinde yanlışlıkla reddedilme olasılığını ölçer, yani,

$$\text{istatistiksel anlamlılık } = 1 - \alpha = 1 - P(H_0 \text{ reddet} \mid H_0 \text{ doğru}). \tag{18.10.5}$$

Aynı zamanda *1. tür hata* veya *yanlış pozitif* olarak da anılır. α, *anlamlılık düzeyi* olarak adlandırılır ve yaygın olarak kullanılan değeri $\%5$, yani $1 - \alpha = \%95$ şeklindedir. Anlamlılık düzeyi, gerçek bir sıfır hipotezi reddettiğimizde almaya istekli olduğumuz risk seviyesi olarak açıklanabilir.

Fig. 18.10.1, iki örneklemli bir hipotez testinde gözlemlerin değerlerini ve belirli bir normal dağılımın gelme olasılığını gösterir. Gözlem veri örneği $\%95$ eşiğinin dışında yer alırsa, sıfır hipotez varsayımı altında çok olası olmayan bir gözlem olacaktır. Dolayısıyla, sıfır hipotezde yanlış bir şeyler olabilir ve onu reddedeceğiz.

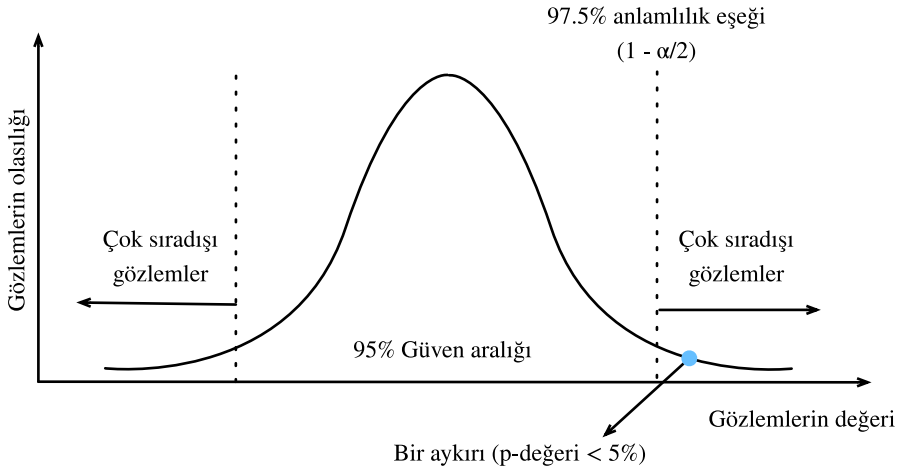


Fig. 18.10.1: İstatistiksel anlamlılık.

İstatistiksel Güç

İstatistiksel Güç (veya *duyarlılık*), reddedilmesi gerektiğinde sıfır hipotezin, H_0, reddedilme olasılığını ölçer, yani,

$$\text{istatistiksel güç } = 1 - \beta = 1 - P(H_0 \text{ rededememe } \mid H_0 \text{ yanlış}). \tag{18.10.6}$$

Bir *1. tür hata*nın, doğru olduğunda sıfır hipotezin reddedilmesinden kaynaklanan bir hata olduğunu hatırlayın, oysa *2. tür hata* yanlış olduğunda sıfır hipotezin reddedilmemesinden kaynaklanır. 2. tür hata genellikle β olarak belirtilir ve bu nedenle ilgili istatistiksel güç $1 - \beta$ olur.

Sezgisel olarak, istatistiksel güç, testimizin istenen bir istatistiksel anlamlılık düzeyindeyken minimum büyüklükte gerçek bir tutarsızlığı ne kadar olasılıkla tespit edeceği şeklinde yorumlanabilir. $\%80$, yaygın olarak kullanılan bir istatistiksel güç eşiğidir. İstatistiksel güç ne kadar yüksekse, gerçek farklılıkları tespit etme olasılığımız o kadar yüksektir.

İstatistiksel gücün en yaygın kullanımlarından biri, ihtiyaç duyulan örnek sayısını belirlemektir. Sıfır hipotezini yanlış olduğunda reddetme olasılığınız, yanlış olma derecesine (*etki boyutu* olarak bilinir) ve sahip olduğunuz örneklerin sayısına bağlıdır. Tahmin edebileceğiniz gibi, küçük etki boyutları, yüksek olasılıkla tespit edilebilmesi için çok fazla sayıda örnek gerektirir. Ayrıntılı olarak türetmek için bu kısa ek bölümün kapsamı dışında, örnek olarak, örneğimizin sıfır ortalama bir varyanslı Gauss'tan geldiğine dair bir sıfır hipotezi reddedebilmek isterken, örneklemimizin ortalamasının aslında bire yakın olduğuna inanıyoruz, bunu yalnızca 8'lik örneklem büyüklüğünde kabul edilebilir hata oranları ile yapabiliriz. Bununla birlikte, örnek popülasyonumuzun gerçek ortalamasının 0.01'e yakın olduğunu düşünürsek, farkı tespit etmek için yaklaşık 80000'lik bir örneklem büyüklüğüne ihtiyacımız olur.

Gücü bir su filtresi olarak hayal edebiliriz. Bu benzetmede, yüksek güçlü bir hipotez testi, sudaki zararlı maddeleri olabildiğince azaltacak yüksek kaliteli bir su filtreleme sistemi gibidir. Öte yandan, daha küçük bir tutarsızlık, bazı nispeten küçük maddelerin boşluklardan kolayca kaçabildiği düşük kaliteli bir su filtresine benzer. Benzer şekilde, istatistiksel güç yeterince yüksek güce sahip değilse, bu test daha küçük tutarsızları yakalayamayabilir.

Test İstatistiği

Bir *test istatistiği* $T(x)$, örnek verilerin bazı özelliklerini özetleyen bir sayıdır. Böyle bir istatistiği tanımlamanın amacı, farklı dağılımları ayırt etmemize ve hipotez testimizi yürütmemize izin vermesidir. Kimyager örneğimize geri dönersek, bir popülasyonun diğerinden daha iyi performans gösterdiğini göstermek istiyorsak, ortalamayı test istatistiği olarak almak mantıklı olabilir. Farklı test istatistiği seçenekleri, büyük ölçüde farklı istatistiksel güce sahip istatistiksel testlere yol açabilir.

Genellikle, $T(X)$ (sıfır hipotezimiz altındaki test istatistiğinin dağılımı), en azından yaklaşık olarak, sıfır hipotezi kapsamında değerlendirildiğinde normal dağılım gibi genel bir olasılık dağılımını izleyecektir. Açıkça böyle bir dağılım elde edebilir ve daha sonra veri kümemizdeki test istatistiğimizi ölçebilirsek, istatistiğimiz beklediğimiz aralığın çok dışındaysa sıfır hipotezi güvenle reddedebiliriz. Bunu nicel hale getirmek bizi p-değerleri kavramına götürür.

p-Değeri

p-değeri (veya *olasılık değeri*), sıfır hipotezinin *doğru* olduğu varsayılarak, $T(X)$'in en az gözlenen test istatistiği $T(x)$ kadar uç olma olasılığıdır, yani

$$p\text{-değeri} = P_{H_0}(T(X) \geq T(x)). \tag{18.10.7}$$

p-değeri önceden tanımlanmış ve sabit bir istatistiksel anlamlılık düzeyi α değerinden küçükse veya ona eşitse, sıfır hipotezini reddedebiliriz. Aksi takdirde, sıfır hipotezi reddetmek için kanıtımız olmadığı sonucuna varacağız. Belirli bir popülasyon dağılımı için, *reddetme bölgesi*, istatistiksel anlamlılık düzeyi α'dan daha küçük bir p değerine sahip tüm noktaların içerildiği aralık olacaktır.

Tek Taraflı Test ve İki Taraflı Test

Normalde iki tür anlamlılık testi vardır: Tek taraflı test ve iki taraflı test. *Tek taraflı test* (veya *tek kuyruklu test*), sıfır hipotez ve alternatif hipotezin yalnızca bir tarafta olduğunda geçerlidir. Örneğin, sıfır hipotez θ gerçek parametresinin c değerinden küçük veya ona eşit olduğunu belirtebilir. Alternatif hipotez, θ nın c'den büyük olması olacaktır. Yani, reddetme bölgesi, örneklem dağılımının sadece bir tarafındadır. Tek taraflı testin aksine, *iki taraflı test* (veya *iki kuyruklu test*), reddetme bölgesi örneklem dağılımının her iki tarafında olduğunda uygulanabilir. Bu durumda bir örnek, θ gerçek parametresinin c değerine eşit olduğunu belirten bir sıfır hipotez ifadesine sahip olma olabilir. Alternatif hipotez, θ'nın c'ye eşit olmamasıdır.

Hipotez Testinin Genel Adımları

Yukarıdaki kavramlara aşina olduktan sonra, hipotez testinin genel adımlarından geçelim.

1. Soruyu belirtin ve sıfır hipotezi, H_0, oluşturun.
2. İstatistiksel anlamlılık düzeyini α'yı ve bir istatistiksel güç $(1 - \beta)$'yı ayarlayın.
3. Deneyler yoluyla numuneler alın. İhtiyaç duyulan örnek sayısı istatistiksel güce ve beklenen etki büyüklüğüne bağlı olacaktır.
4. Test istatistiğini ve p-değerini hesaplayın.
5. p-değeri ve istatistiksel anlamlılık düzeyi α bağlı olarak sıfır hipotezi tutma veya reddetme kararını verin.

Bir hipotez testi yapmak için, bir sıfır hipotez ve almaya istekli olduğumuz bir risk seviyesi tanımlayarak başlıyoruz. Sonra, sıfır hipotezine karşı kanıt olarak test istatistiğinin aşırı bir değerini alarak numunenin (örneklemin) test istatistiğini hesaplıyoruz. Test istatistiği reddetme bölgesi dahilindeyse, alternatif lehine sıfır hipotezi reddedebiliriz.

Hipotez testi, klinik araştırmalar ve A/B testi gibi çeşitli senaryolarda uygulanabilir.

18.10.3 Güven Aralıkları Oluşturma

Bir θ parametresinin değerini tahmin ederken, $\hat{\theta}$ gibi nokta tahmincileri, belirsizlik kavramı içermedikleri için sınırlı fayda sağlar. Daha ziyade, yüksek olasılıkla gerçek θ parametresini içeren bir aralık oluşturabilirsek çok daha iyi olurdu. Yüzyıl önce bu tür fikirlerle ilgileniyor olsaydınız, 1937'de güven aralığı kavramını tanıtan Jerzy Neyman'ın "Klasik Olasılık Teorisine Dayalı İstatistiksel Tahmin Teorisinin Ana Hatları (Outline of a Theory of Statistical Estimation Based on the Classical Theory of Probability)"'nı okumaktan heyecan duyardınız (Neyman, 1937).

Faydalı olması için, belirli bir kesinlik derecesi için bir güven aralığı mümkün olduğu kadar küçük olmalıdır. Nasıl türetileceğini görelim.

Tanım

Matematiksel olarak, θ gerçek parametresi için *güven aralığı*, örnek verilerden şu şekilde hesaplanan bir C_n aralığıdır.

$$P_\theta(C_n \ni \theta) \geq 1 - \alpha, \forall \theta. \tag{18.10.8}$$

Burada $\alpha \in (0, 1)$ ve $1 - \alpha$, aralığın *güven düzeyi* veya *kapsamı* olarak adlandırılır. Bu, yukarıda tartıştığımız anlam düzeyiyle aynı α'dır.

Unutmayın (18.10.8) C_n değişkeni hakkındadır, sabit θ ile ilgili değildir. Bunu vurgulamak için, $P_\theta(\theta \in C_n)$ yerine $P_\theta(C_n \ni \theta)$ yazıyoruz.

Yorumlama

$\%95$ güven aralığını, gerçek parametrenin $\%95$ olduğundan emin olabileceğiniz bir aralık olarak yorumlamak çok caziptir, ancak bu maalesef doğru değildir. Gerçek parametre sabittir ve rastgele olan aralıktır. Bu nedenle, bu yordamla çok sayıda güven aralığı oluşturduysanız, oluşturulan aralıkların $\%95$'inin gerçek parametreyi içereceğini söylemek daha iyi bir yorum olacaktır.

Bu bilgiçlik gibi görünebilir, ancak sonuçların yorumlanmasında gerçek etkileri olabilir. Özellikle, çok nadiren yeterince yaptığımız sürece, *neredeyse kesin* gerçek değeri içermediğimiz aralıklar oluşturarak (18.10.8)i tatmin edebiliriz. Bu bölümü cazip ama yanlış üç ifade sunarak kapatıyoruz. Bu noktaların derinlemesine bir tartışması şu adreste bulunabilir (Morey *et al.*, 2016).

- **Yanılgı 1**. Dar güven aralıkları, parametreyi tam olarak tahmin edebileceğimiz anlamına gelir.
- **Yanılgı 2**. Güven aralığı içindeki değerlerin, aralığın dışındaki değerlere göre gerçek değer olma olasılığı daha yüksektir.
- **Yanılgı 3**. Gözlemlenen belirli bir $\%95$ güven aralığının gerçek değeri içerme olasılığı $\%95$'tir.

Güven aralıklarının narin nesneler olduğunu söylemek yeterli. Ancak yorumlamasını net tutarsanız, bunlar güçlü araçlar olabilir.

Bir Gauss Örneği

En klasik örneği, bilinmeyen ortalama ve varyansa sahip bir Gauss dağılımının ortalaması için güven aralığını tartışalım. Gauss dağılımdan, $\mathcal{N}(\mu, \sigma^2)$'dan, n örnek, $\{x_i\}_{i=1}^n$, topladığımızı varsayalım. Ortalama ve standart sapma için tahmincileri şu şekilde hesaplayabiliriz:

$$\hat{\mu}_n = \frac{1}{n}\sum_{i=1}^{n} x_i \text{ and } \hat{\sigma}_n^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \hat{\mu})^2. \tag{18.10.9}$$

Şimdi rastgele değişkeni düşünürsek

$$T = \frac{\hat{\mu}_n - \mu}{\hat{\sigma}_n/\sqrt{n}}, \tag{18.10.10}$$

$n-1$ *serbestlik derecesinde, Student-t (Öğrenci-t) dağılımı* adı verilen iyi bilinen bir dağılımı izleyen rastgele bir değişken elde ederiz.

Bu dağılım çok iyi incelenmiştir ve örneğin, $n \to \infty$ iken, yaklaşık olarak standart bir Gauss olduğu bilinir ve dolayısıyla bir tabloda Gauss b.y.f'nin değerlerine bakarak T değerinin zamanın en az $\%95$'inde $[-1.96, 1.96]$ aralığında olduğu sonucuna varabiliriz. n değerinin sonlu değerleri için, aralığın biraz daha büyük olması gerekir, ancak bunlar iyi bilinmekte ve tablolarda önceden hesaplanmaktadır.

Böylece, büyük $ n $ için diyebiliriz ki,

$$P\left(\frac{\hat{\mu}_n - \mu}{\hat{\sigma}_n/\sqrt{n}} \in [-1.96, 1.96]\right) \geq 0.95. \tag{18.10.11}$$

Bunu her iki tarafı da $\hat{\sigma}_n/\sqrt{n}$ ile çarpıp ardından $\hat{\mu}_n$ ekleyerek yeniden düzenleriz,

$$P\left(\mu \in \left[\hat{\mu}_n - 1.96\frac{\hat{\sigma}_n}{\sqrt{n}}, \hat{\mu}_n + 1.96\frac{\hat{\sigma}_n}{\sqrt{n}}\right]\right) \geq 0.95. \tag{18.10.12}$$

Böylece, $\%95$'lik güven aralığımızı bulduğumuzu biliyoruz:

$$\left[\hat{\mu}_n - 1.96\frac{\hat{\sigma}_n}{\sqrt{n}}, \hat{\mu}_n + 1.96\frac{\hat{\sigma}_n}{\sqrt{n}}\right]. \tag{18.10.13}$$

Şunu söylemek güvenlidir: (18.10.13) istatistikte en çok kullanılan formüllerden biridir. İstatistik tartışmamızı uygulama ile kapatalım. Basit olması için, asimptotik (kavuşma doğrusal) rejimde olduğumuzu varsayıyoruz. Küçük N değerleri, programlanarak veya bir t-tablosundan elde edilen `t_star`'ın doğru değerini içermelidir.

```
# PyTorch, varsayılan olarak Bessel'in düzeltmesini kullanır;
# bu, numpy'de varsayılan ddof=0 yerine ddof=1 kullanılması anlamına gelir.
# ddof=0'ı taklit etmek için unbiased=False kullanabiliriz.

# Örnek sayısı
N = 1000

# Örnek veri kümesi
samples = torch.normal(0, 1, size=(N,))

# Öğrenci t-dağılımının c.d.f.'sine bak
t_star = 1.96
```

(continues on next page)

(continued from previous page)

```
# Aralık oluştur
mu_hat = torch.mean(samples)
sigma_hat = samples.std(unbiased=True)
(mu_hat - t_star*sigma_hat/torch.sqrt(torch.tensor(N, dtype=torch.float32)),\
 mu_hat + t_star*sigma_hat/torch.sqrt(torch.tensor(N, dtype=torch.float32)))
```

```
(tensor(-0.0568), tensor(0.0704))
```

18.10.4 Özet

- İstatistik, çıkarım sorunlarına odaklanırken, derin öğrenme, açıkça programlamadan ve anlamadan doğru tahminler yapmaya vurgu yapar.
- Üç yaygın istatistik çıkarım yöntemi vardır: Tahmincileri değerlendirme ve karşılaştırma, hipotez testleri yürütme ve güven aralıkları oluşturma.
- En yaygın üç tahminci vardır: İstatistiksel yanlılık, standart sapma ve ortalama hata karesi.
- Bir güven aralığı, örneklerle oluşturabileceğimiz gerçek bir popülasyon parametresinin tahmini aralığıdır.
- Hipotez testi, bir popülasyonla ilgili varsayılan ifadeye karşı bazı kanıtları değerlendirmenin bir yoludur.

18.10.5 Alıştırmalar

1. $X_1, X_2, \ldots, X_n \overset{\text{iid}}{\sim} \text{Tekdüze}(0, \theta)$ olsun, burada "iid" *bağımsız ve aynı şekilde dağılmış* anlamına gelir. Aşağıdaki θ tahmincilerini düşünün:

$$\hat{\theta} = \max\{X_1, X_2, \ldots, X_n\}; \tag{18.10.14}$$

$$\tilde{\theta} = 2\bar{X}_n = \frac{2}{n}\sum_{i=1}^{n} X_i. \tag{18.10.15}$$

   - $\hat{\theta}$ için istatistiksel yanlılığı, standart sapmayı ve ortalama hata karesini bulunuz.
   - $\tilde{\theta}$ için istatistiksel yanlılığı, standart sapmayı ve ortalama hata karesini bulunuz.
   - Hangi tahminci daha iyi?
2. Girişteki kimyager örneğimiz için, iki taraflı bir hipotez testi yapmak için 5 adımı türetebilir misiniz? İstatistiksel anlamlılık düzeyini $\alpha = 0.05$ ve istatistiksel gücü $1 - \beta = 0.8$ alınız.
3. 100 tane bağımsız olarak oluşturulan veri kümesi için güven aralığı kodunu $N = 2$ ve $\alpha = 0.5$ ile çalıştırın ve ortaya çıkan aralıkları çizin (bu durumda `t_star = 1.0`). Gerçek ortalama olan 0'ı içermekten çok uzak olan birkaç çok kısa aralık göreceksiniz. Bu, güven aralığının yorumlamasıyla çelişiyor mu? Yüksek hassasiyetli tahminleri belirtmek için kısa aralıklar kullanmakta kendinizi rahat hissediyor musunuz?

Tartışmalar[226]

[226] https://discuss.d2l.ai/t/1102

18.11 Bilgi Teorisi

Evren bilgi ile dolup taşıyor. Bilgi, disiplinler arası açıklıklarda ortak bir dil sağlar: Shakespeare'in Sone'sinden Cornell ArXiv'deki araştırmacıların makalesine, Van Gogh'un eseri Yıldızlı Gece'den Beethoven'in 5. Senfonisi'ne, ilk programlama dili Plankalkül'den son teknoloji makine öğrenmesi algoritmalarına. Biçimi ne olursa olsun, her şey bilgi teorisinin kurallarını izlemelidir. Bilgi teorisi ile, farklı sinyallerde ne kadar bilgi bulunduğunu ölçebilir ve karşılaştırabiliriz. Bu bölümde, bilgi teorisinin temel kavramlarını ve bilgi teorisinin makine öğrenmesindeki uygulamalarını inceleyeceğiz.

Başlamadan önce, makine öğrenmesi ve bilgi teorisi arasındaki ilişkiyi özetleyelim. Makine öğrenmesi, verilerden ilginç sinyaller çıkarmayı ve kritik tahminlerde bulunmayı amaçlar. Öte yandan, bilgi teorisi, bilgiyi kodlama, kod çözme, iletme ve üstünde oynama yapmayı inceler. Sonuç olarak, bilgi teorisi, makine öğrenmesi sistemlerinde bilgi işlemeyi tartışmak için temel bir dil sağlar. Örneğin, birçok makine öğrenmesi uygulaması çapraz entropi kaybını şurada açıklandığı gibi kullanır Section 3.4. Bu kayıp, doğrudan bilgi teorisel değerlendirmelerinden türetilebilir.

18.11.1 Bilgi

Bilgi teorisinin "ruhu" ile başlayalım: Bilgi. *Bilgi*, bir veya daha fazla kodlama biçimli belirli bir dizi ile kodlanabilir. Kendimizi bir bilgi kavramını tanımlamaya çalışmakla görevlendirdiğimizi varsayalım. Başlangıç noktamız ne olabilir?

Aşağıdaki düşünce deneyini düşünün. Kart destesi olan bir arkadaşımız var. Desteyi karıştıracaklar, bazı kartları ters çevirecekler ve bize kartlar hakkında açıklamalar yapacaklar. Her ifadenin bilgi içeriğini değerlendirmeye çalışacağız.

Önce, bir kartı çevirip bize "Bir kart görüyorum" diyorlar. Bu bize hiçbir bilgi sağlamaz. Durumun bu olduğundan zaten emindik, bu yüzden bilginin sıfır olduğu umuyoruz.

Sonra bir kartı çevirip "Bir kalp görüyorum" diyorlar. Bu bize biraz bilgi sağlar, ancak gerçekte, her biri eşit olasılıkla mümkün olan yalnızca 4 farklı takım vardır, bu nedenle bu sonuca şaşırmadık. Bilginin ölçüsü ne olursa olsun, bu olayın düşük bilgi içeriğine sahip olmasını umuyoruz.

Sonra, bir kartı çevirip "Bu maça 3" diyorlar. Bu daha fazla bilgidir. Gerçekten de 52 eşit olasılıklı sonuçlar vardı ve arkadaşımız bize bunun hangisi olduğunu söyledi. Bu orta miktarda bilgi olmalıdır.

Bunu mantıksal uç noktaya götürelim. Sonunda destedeki her kartı çevirdiklerini ve karışık destenin tüm dizisini okuduklarını varsayalım. Destede $52!$ farklı dizilim var, gene hepsi aynı olasılığa sahip, bu yüzden hangisinin olduğunu bilmek için çok fazla bilgiye ihtiyacımız var.

Geliştirdiğimiz herhangi bir bilgi kavramı bu sezgiye uygun olmalıdır. Aslında, sonraki bölümlerde bu olayların 0 bit, 2 bit, 5.7 bit, and 225.6 bit bilgiye sahip olduğunu nasıl hesaplayacağımızı öğreneceğiz.

Bu düşünce deneylerini okursak, doğal bir fikir görürüz. Başlangıç noktası olarak, bilgiyi önemsemekten ziyade, bilginin olayın sürpriz derecesini veya soyut olasılığını temsil ettiği fikrini geliştirebiliriz. Örneğin, alışılmadık bir olayı tanımlamak istiyorsak, çok fazla bilgiye ihtiyacımız var. Genel (sıradan) bir olay için fazla bilgiye ihtiyacımız olmayabilir.

1948'de Claude E. Shannon bilgi teorisini oluşturan *İletişimin Bir Matematiksel Teorisi (A Mathematical Theory of Communication)* çalışmasını yayınladı (Shannon, 1948). Shannon makalesinde

ilk kez bilgi entropisi kavramını tanıttı. Yolculuğumuza buradan başlayacağız.

Öz-Bilgi

Bilgi bir olayın soyut olasılığını içerdiğinden, olasılığı bit adeti ile nasıl eşleştirebiliriz? Shannon, başlangıçta John Tukey tarafından oluşturulmuş olan *bit* terimini bilgi birimi olarak tanıttı. Öyleyse "bit" nedir ve bilgiyi ölçmek için neden onu kullanıyoruz? Tarihsel olarak, antika bir verici yalnızca iki tür kod gönderebilir veya alabilir: 0 ve 1. Aslında, ikili kodlama hala tüm modern dijital bilgisayarlarda yaygın olarak kullanılmaktadır. Bu şekilde, herhangi bir bilgi bir dizi 0 ve 1 ile kodlanır. Ve bu nedenle, n uzunluğundaki bir dizi ikili rakam, n bit bilgi içerir.

Şimdi, herhangi bir kod dizisi için, her 0 veya 1'in $\frac{1}{2}$ olasılıkla gerçekleştiğini varsayalım. Bu nedenle, n uzunluğunda bir dizi koda sahip bir X olayı, $\frac{1}{2^n}$ olasılıkla gerçekleşir. Aynı zamanda, daha önce bahsettiğimiz gibi, bu seri n bit bilgi içerir. Öyleyse, p olasılığını bit sayısına aktarabilen bir matematik fonksiyonuna genelleme yapabilir miyiz? Shannon, *öz-bilgi*yi tanımlayarak cevabı verdi,

$$I(X) = -\log_2(p), \tag{18.11.1}$$

yani bu X etkinliği için aldığımız bilginin *bitleri* olarak. Bu bölümde her zaman 2 tabanlı logaritma kullanacağımızı unutmayın. Basitlik adına, bu bölümün geri kalanı logaritma gösteriminde 2 altindisini göstermeyecektir, yani $\log(.)$ her zaman $\log_2(.)$ anlamına gelir. Örneğin, "0010" kodu şu öz-bilgiyi içerir:

$$I(\text{"0010"}) = -\log(p(\text{"0010"})) = -\log\left(\frac{1}{2^4}\right) = 4 \text{ bits}. \tag{18.11.2}$$

Öz-bilgiyi aşağıda gösterildiği gibi hesaplayabiliriz. Ondan önce, önce bu bölümdeki gerekli tüm paketleri içe aktaralım.

```
import torch
from torch.nn import NLLLoss


def nansum(x):
    # Define nansum, as pytorch doesn't offer it inbuilt.
    return x[~torch.isnan(x)].sum()

def self_information(p):
    return -torch.log2(torch.tensor(p)).item()

self_information(1 / 64)
```

```
6.0
```

18.11.2 Entropi

Öz-bilgi yalnızca tek bir ayrık olayın bilgisini ölçtüğü için, ayrık veya sürekli dağılımın herhangi bir rastgele değişkeni için daha genelleştirilmiş bir ölçüme ihtiyacımız var.

Motive Edici Entropi

Ne istediğimiz konusunda belirli olmaya çalışalım. Bu, *Shannon entropisinin aksiyomları* olarak bilinenlerin gayri resmi bir ifadesi olacaktır. Aşağıdaki sağduyu beyanları topluluğunun bizi benzersiz bir bilgi tanımına zorladığı ortaya çıkacaktır. Bu aksiyomların usule uygun bir versiyonu, diğer birçoklarıyla birlikte şu adreste bulunabilir (Csiszár, 2008).

1. Rastgele bir değişkeni gözlemleyerek kazandığımız bilgi, elemanlar dediğimiz şeye veya olasılığı sıfır olan ek elemanların varlığına bağlı değildir.
2. İki rastgele değişkeni gözlemleyerek elde ettiğimiz bilgi, onları ayrı ayrı gözlemleyerek elde ettiğimiz bilgilerin toplamından fazlası değildir. Bağımsız iseler, o zaman tam toplamdır.
3. (Neredeyse) Kesin olayları gözlemlerken kazanılan bilgi (neredeyse) sıfırdır.

Bu gerçeğin kanıtlanması kitabımızın kapsamı dışında olduğu halde, bunun entropinin alması gereken şekli benzersiz bir şekilde belirlediğini bilmek önemlidir. Bunların izin verdiği tek belirsizlik, daha önce gördüğümüz seçimi yaparak normalize edilen temel birimlerin seçimidir; tek bir adil yazı tura ile sağlanan bilginin bir bit olması gibi.

Tanım

Olasılık yoğunluk fonksiyonu (pdf/oyf) veya olasılık kütle fonksiyonu (pmf/okf) $p(x)$ ile olasılık dağılımı P'yi takip eden rastgele değişken X için, beklenen bilgi miktarını *entropi* (veya *Shannon entropisi*) ile ölçebiliriz:

$$H(X) = -E_{x \sim P}[\log p(x)]. \tag{18.11.3}$$

Belirleyici olmak gerekirse, X ayrıksa,

$$H(X) = -\sum_i p_i \log p_i, \text{ burada } p_i = P(X_i) \tag{18.11.4}$$

.

Aksi takdirde, X sürekli ise, entropiyi *diferansiyel (farksal) entropi* olarak da adlandırırız.

$$H(X) = -\int_x p(x) \log p(x) \; dx. \tag{18.11.5}$$

Entropiyi aşağıdaki gibi tanımlayabiliriz.

```
def entropy(p):
    entropy = - p * torch.log2(p)
    # `nansum` işlemcisi, nan (not a number - sayı olmayan) olmayan sayıyı toplayacaktır
    out = nansum(entropy)
    return out

entropy(torch.tensor([0.1, 0.5, 0.1, 0.3]))
```

```
tensor(1.6855)
```

Yorumlama

Merak ediyor olabilirsiniz: Entropi tanımında (18.11.3), neden negatif bir logaritma ortalaması kullanıyoruz? Burada bazı sezgileri verelim.

İlk olarak, neden *logaritma* işlevi log kullanıyoruz? $p(x) = f_1(x)f_2(x)\ldots, f_n(x)$ olduğunu varsayalım, burada her bileşen işlevi $f_i(x)$ birbirinden bağımsızdır. Bu, her bir $f_i(x)$'in $p(x)$'den elde edilen toplam bilgiye bağımsız olarak katkıda bulunduğu anlamına gelir. Yukarıda tartışıldığı gibi, entropi formülünün bağımsız rastgele değişkenler üzerinde toplamsal olmasını istiyoruz. Neyse ki, log doğal olarak olasılık dağılımlarının çarpımını bireysel terimlerin toplamına dönüştürebilir.

Sonra, neden *negatif* log kullanıyoruz? Sezgisel olarak, alışılmadık bir vakadan genellikle sıradan olandan daha fazla bilgi aldığımız için, daha sık olaylar, daha az yaygın olaylardan daha az bilgi içermelidir. Bununla birlikte, log olasılıklarla birlikte monoton bir şekilde artıyor ve aslında $[0, 1]$ içindeki tüm değerler için negatif. Olayların olasılığı ile entropileri arasında monoton olarak azalan bir ilişki kurmamız gerekir ki bu ideal olarak her zaman pozitif olacaktır (çünkü gözlemlediğimiz hiçbir şey bizi bildiklerimizi unutmaya zorlamamalıdır). Bu nedenle, log fonksiyonunun önüne bir eksi işareti ekliyoruz.

Son olarak, *beklenti (ortalama)* işlevi nereden geliyor? Rastgele bir değişken olan $X'i$ düşünün. Öz-bilgiyi ($-\log(p)$), belirli bir sonucu gördüğümüzde sahip olduğumuz *sürpriz* miktarı olarak yorumlayabiliriz. Nitekim, olasılık sıfıra yaklaştıkça sürpriz sonsuz olur. Benzer şekilde, entropiyi X'i gözlemlemekten kaynaklanan ortalama sürpriz miktarı olarak yorumlayabiliriz. Örneğin, bir slot makinesi sisteminin $p_1, \ldots, p_k$ olasılıklarıyla $s_1, \ldots, s_k$ sembollerini istatistiksel olarak bağımsız yaydığını düşünün. O zaman bu sistemin entropisi, her bir çıktının gözlemlenmesinden elde edilen ortalama öz-bilgiye eşittir, yani,

$$H(S) = \sum_i p_i \cdot I(s_i) = -\sum_i p_i \cdot \log p_i. \tag{18.11.6}$$

Entropinin Özellikleri

Yukarıdaki örnekler ve yorumlarla, entropinin (18.11.3) şu özelliklerini türetebiliriz. Burada, X'i bir olay ve P'yi X'in olasılık dağılımı olarak adlandırıyoruz.

- Tüm ayrık X değerleri için $H(X) \geq 0$'dır (sürekli X için entropi negatif olabilir).
- Bir o.y.f veya o.k.f. $p(x)$ ile $X \sim P$ ise ve o.y.f veya o.k.f. $q(x)$'ya sahip yeni bir olasılık dağılımı Q ile P'yi tahmin etmeye çalışıyoruz, o zaman

  $$H(X) = -E_{x\sim P}[\log p(x)] \leq -E_{x\sim P}[\log q(x)], \text{eşitlikle ancak ve ancak eğer } P = Q. \tag{18.11.7}$$

  Alternatif olarak, $H(X)$, P'den çekilen sembolleri kodlamak için gereken ortalama bit sayısının alt sınırını verir.
- $X \sim P$ ise, x tüm olası sonuçlar arasında eşit olarak yayılırsa maksimum bilgi miktarını iletir. Özel olarak, P k-sınıflı ayrık olasılık dağılımı $\{p_1, \ldots, p_k\}$ ise, o halde

  $$H(X) \leq \log(k), \text{eşitlikle ancak ve ancak eğer } p_i = \frac{1}{k}, \forall i. \tag{18.11.8}$$

Eğer P sürekli bir rastgele değişkense, öykü çok daha karmaşık hale gelir. Bununla birlikte, ek olarak P'nin sonlu bir aralıkta (tüm değerler 0 ile 1 arasında) desteklenmesini zorlarsak, bu aralıkta tekdüze dağılım varsa P en yüksek entropiye sahip olur.

18.11.3 Ortak Bilgi

Daha önce tek bir rastgele değişken X entropisini tanımlamıştık, bir çift rastgele değişken (X, Y) entropisine ne dersiniz? Bu teknikleri şu soru tipini yanıtlamaya çalışırken düşünebiliriz: "X ve Y'de her biri ayrı ayrı olması bir arada olmalarıyla karşılaştırıldığında ne tür bilgi bulunur? Gereksiz bilgi var mı, yoksa hepsi eşsiz mi?"

Aşağıdaki tartışma için, her zaman (X, Y)'yi, bir o.y.f veya o.k.f. olan $p_{X,Y}(x, y)$ ile bileşik olasılık dağılımı P'yi izleyen bir çift rastgele değişken olarak kullanıyoruz, aynı zamanda da X ve Y sırasıyla $p_X(x)$ ve $p_Y(y)$ olasılık dağılımlarını takip eder.

Bileşik Entropi

Tek bir rastgele değişkenin entropisine benzer şekilde (18.11.3), rastgele değişken çiftinin, (X, Y), *bileşik entropisini* $H(X, Y)$ olarak tanımlarız.

$$H(X, Y) = -E_{(x,y)\sim P}[\log p_{X,Y}(x, y)]. \tag{18.11.9}$$

Tam olarak, bir yandan (X, Y) bir çift ayrık rastgele değişkense, o zaman

$$H(X, Y) = -\sum_x \sum_y p_{X,Y}(x, y) \log p_{X,Y}(x, y). \tag{18.11.10}$$

Öte yandan, (X, Y) bir çift sürekli rastgele değişken ise, o zaman *diferansiyel (farksal) bileşik entropiyi* tanımlarız.

$$H(X, Y) = -\int_{x,y} p_{X,Y}(x, y)\ \log p_{X,Y}(x, y)\ dx\ dy. \tag{18.11.11}$$

Şunu düşünebiliriz (18.11.9) bize rastgele değişkenler çiftindeki toplam rastgeleliği anlatıyor. Bir çift uç vaka olarak, eğer $X = Y$ iki özdeş rastgele değişken ise, o zaman çiftteki bilgi tam olarak bir tanedeki bilgidir ve $H(X, Y) = H(X) = H(Y)$'dir. Diğer uçta, X ve Y bağımsızsa, $H(X, Y) = H(X) + H(Y)$'dir. Aslında, her zaman bir çift rasgele değişkenin içerdiği bilginin her iki rasgele değişkenin entropisinden daha küçük ve her ikisinin toplamından daha fazla olmadığını bilecegiz.

$$H(X), H(Y) \leq H(X, Y) \leq H(X) + H(Y). \tag{18.11.12}$$

Ortak entropiyi en başından uygulayalım.

```
def joint_entropy(p_xy):
    joint_ent = -p_xy * torch.log2(p_xy)
    # `nansum` işlemcisi, nan (not a number - sayı olmayan) olmayan sayıyı toplayacaktır
    out = nansum(joint_ent)
    return out

joint_entropy(torch.tensor([[0.1, 0.5], [0.1, 0.3]]))
```

```
tensor(1.6855)
```

Bunun öncekiyle aynı *kod* olduğuna dikkat edin, ancak şimdi onu iki rastgele değişkenin bileşik dağılımı üzerinde çalışırken farklı bir şekilde yorumluyoruz.

Koşullu Entropi

Bileşik entropi bir çift rastgele değişkende bulunan bilgi miktarının üzerinde tanımlıdır. Bu yararlıdır, ancak çoğu zaman umursadığımız şey değildir. Makine öğrenmesinin ayarlarını düşünün. Bir imgenin piksel değerlerini tanımlayan rastgele değişken (veya rastgele değişkenlerin vektörü) olarak X'i ve sınıf etiketi olan rastgele değişken olarak Y'yi alalım. X önemli bilgi içermelidir — doğal bir imge karmaşık bir şeydir. Ancak, imge gösterildikten sonra Y içindeki bilgi düşük olmalıdır. Aslında, bir rakamın imgesi, rakam okunaksız olmadıkça, hangi rakam olduğu hakkında bilgiyi zaten içermelidir. Bu nedenle, bilgi teorisi kelime dağarcığımızı genişletmeye devam etmek için, rastgele bir değişkenin diğerine koşullu bağlı olarak bilgi içeriği hakkında mantık yürütebilmeliyiz.

Olasılık teorisinde, değişkenler arasındaki ilişkiyi ölçmek için *koşullu olasılığın* tanımını gördük. Şimdi, *koşullu entropiyi*, $H(Y \mid X)$, benzer şekilde tanımlamak istiyoruz. Bunu şu şekilde yazabiliriz:

$$H(Y \mid X) = -E_{(x,y)\sim P}[\log p(y \mid x)], \tag{18.11.13}$$

Burada $p(y \mid x) = \frac{p_{X,Y}(x,y)}{p_X(x)}$ koşullu olasılıktır. Özellikle, (X, Y) bir çift ayrık rastgele değişken ise, o zaman

$$H(Y \mid X) = -\sum_x \sum_y p(x,y) \log p(y \mid x). \tag{18.11.14}$$

(X, Y) bir çift sürekli rastgele değişkense, *diferansiyel koşullu entropi* benzer şekilde şöyle tanımlanır:

$$H(Y \mid X) = -\int_x \int_y p(x,y)\ \log p(y \mid x)\ dx\ dy. \tag{18.11.15}$$

Şimdi bunu sormak doğaldır, *koşullu entropi* $H(Y \mid X)$, $H(X)$ entropisi ve bileşik entropi $H(X, Y)$ ile nasıl ilişkilidir? Yukarıdaki tanımları kullanarak bunu net bir şekilde ifade edebiliriz:

$$H(Y \mid X) = H(X, Y) - H(X). \tag{18.11.16}$$

Bunun sezgisel bir yorumu vardır: X verildiğinde ($H(Y \mid X)$) Y'deki bilgi, hem X hem de Y ($H(X, Y)$) birlikteyken olan bilgi eksi X içinde zaten bulunan bilgidir. Bu bize Y'de olup da aynı zamanda X ile temsil edilmeyen bilgiyi verir.

Şimdi, koşullu entropiyi, (18.11.13), sıfırdan uygulayalım.

```
def conditional_entropy(p_xy, p_x):
    p_y_given_x = p_xy/p_x
    cond_ent = -p_xy * torch.log2(p_y_given_x)
    # `nansum` işlemcisi, nan (not a number - sayı olmayan) olmayan sayıyı toplayacaktır
    out = nansum(cond_ent)
    return out

conditional_entropy(torch.tensor([[0.1, 0.5], [0.2, 0.3]]),
                    torch.tensor([0.2, 0.8]))
```

```
tensor(0.8635)
```

Karşılıklı Bilgi

Önceki rastgele değişkenler (X, Y) ortamını göz önünde bulundurarak şunu merak edebilirsiniz: "Artık Y'nin ne kadar bilgi içerdiğini ancak X'de olmadığını bildiğimize göre, benzer şekilde X ve Y'nin aralarında ne kadar bilginin paylaşıldığını da sorabilir miyiz?" Cevap, (X, Y)'nin *karşılıklı bilgisi* olacak ve bunu $I(X, Y)$ olarak yazacağız.

Doğrudan biçimsel (formal) tanıma dalmak yerine, önce karşılıklı bilgi için tamamen daha önce oluşturduğumuz terimlere dayalı bir ifade türetmeyi deneyerek sezgilerimizi uygulayalım. İki rastgele değişken arasında paylaşılan bilgiyi bulmak istiyoruz. Bunu yapmaya çalışmanın bir yolu, hem X hem de Y içerisindeki tüm bilgi ile başlamak ve sonra paylaşılmayan kısımları çıkarmaktır. Hem X hem de Y içerisindeki bilgi, $H(X, Y)$ olarak yazılır. Bundan, X içinde yer alan ama Y içinde yer almayan bilgileri ve Y içinde yer alan ama X içinde olmayan bilgileri çıkarmak istiyoruz. Önceki bölümde gördüğümüz gibi, bu sırasıyla $H(X \mid Y)$ ve $H(Y \mid X)$ ile verilmektedir. Bu nedenle, karşılıklı bilginin şöyle olması gerek:

$$I(X, Y) = H(X, Y) - H(Y \mid X) - H(X \mid Y). \tag{18.11.17}$$

Aslında bu, karşılıklı bilgi için geçerli bir tanımdır. Bu terimlerin tanımlarını genişletir ve bunları birleştirirsek, biraz cebir bunun aşağısı gibi olduğunu gösterir

$$I(X, Y) = E_x E_y \left\{ p_{X,Y}(x, y) \log \frac{p_{X,Y}(x, y)}{p_X(x) p_Y(y)} \right\}. \tag{18.11.18}$$

Tüm bu ilişkileri görsel Fig. 18.11.1 içinde özetleyebiliriz. Aşağıdaki ifadelerin de neden $I(X, Y)$ ile eşdeğer olduğunu görmek harika bir sezgi testidir.

- $H(X) - H(X \mid Y)$
- $H(Y) - H(Y \mid X)$
- $H(X) + H(Y) - H(X, Y)$

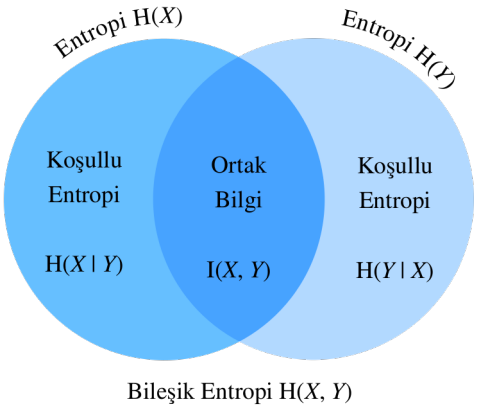


Fig. 18.11.1: Karşılıklı bilginin bileşik entropi ve koşullu entropi ile ilişkisi.

Karşılıklı bilgiyi (18.11.18) birçok yönden Section 18.6 içinde gördüğümüz korelasyon katsayısının ilkesel uzantısı olarak düşünebiliriz. Bu, yalnızca değişkenler arasındaki doğrusal ilişkileri değil,

aynı zamanda herhangi bir türdeki iki rastgele değişken arasında paylaşılan maksimum bilgiyi de sorabilmemize olanak tanır.

Şimdi karşılıklı bilgiyi sıfırdan uygulayalım.

```
def mutual_information(p_xy, p_x, p_y):
    p = p_xy / (p_x * p_y)
    mutual = p_xy * torch.log2(p)
    # `nansum` işlemcisi, nan (not a number - sayı olmayan) olmayan sayıyı toplayacaktır
    out = nansum(mutual)
    return out

mutual_information(torch.tensor([[0.1, 0.5], [0.1, 0.3]]),
                   torch.tensor([0.2, 0.8]), torch.tensor([[0.75, 0.25]]))
```

```
tensor(0.7195)
```

Karşılıklı Bilginin Özellikleri

Karşılıklı bilginin tanımını, (18.11.18), ezberlemek yerine sadece dikkate değer özelliklerini aklınızda tutmanız gerekir:

- Karşılıklı bilgi simetriktir (bakışımlı), yani $I(X, Y) = I(Y, X)$.
- Karşılıklı bilgi negatif olamaz, yani $I(X, Y) \geq 0$.
- $I(X, Y) = 0$ ancak ve ancak X ve Y bağımsızsa olur. Örneğin, X ve Y bağımsızsa, Y'yi bilmek X hakkında herhangi bir bilgi vermez ve bunun tersi de geçerlidir, dolayısıyla karşılıklı bilgileri sıfırdır.
- Alternatif olarak, X, Y değerinin ters çevrilebilir bir işleviyse, Y ve X tüm bilgiyi paylaşır ve

$$I(X, Y) = H(Y) = H(X) \tag{18.11.19}$$

.

Noktasal Karşılıklı Bilgi

Bu bölümün başında entropi ile çalıştığımızda, $-\log(p_X(x))$'i belirli bir sonuca ne kadar *şaşırdığımızın* yorumlanması diye sunabildik. Karşılıklı bilgideki logaritmik terime benzer bir yorum verebiliriz, bu genellikle *noktasal karşılıklı bilgi (pointwise mutual information - pmi)* olarak anılır:

$$\text{pmi}(x, y) = \log \frac{p_{X,Y}(x, y)}{p_X(x)p_Y(y)}. \tag{18.11.20}$$

(18.11.20) denklemi, bağımsız rastgele sonuçlar için beklentimizle karşılaştırıldığında x ve y sonuçlarının belirli kombinasyonunun ne kadar daha fazla veya daha az olası olduğunu ölçmek olarak düşünebiliriz. Büyük ve pozitifse, bu iki belirli sonuç, rastgele şansa kıyasla çok daha sık meydana gelir (*dikkat*: Payda $p_X(x)p_Y(y)$'dir ki bu iki sonucun bağımsız olma olasılığıdır), bilakis büyük ve negatifse, şans eseri beklediğimizden çok daha az gerçekleşen iki sonucu temsil eder.

Bu, karşılıklı bilgiyi, :eqref: `eq_mut_ent_def`, bağımsız olsalardı bekleyeceğimiz şeyle karşılaştırıldığında iki sonucun birlikte gerçekleştiğini gördüğümüzde şaşırmamızın ortalama miktarı olarak yorumlamamıza olanak tanır.

Karşılıklı Bilginin Uygulamaları

Karşılıklı bilgi, saf tanımında biraz soyut olabilir, peki makine öğrenmesi ile nasıl ilişkilidir? Doğal dil işlemede, en zor sorunlardan biri *belirsizlik çözümü* yani bir kelimenin anlamının bağlamdan anlaşılmaz olması sorunudur. Örneğin son zamanlarda bir haber manşetinde "Amazon yanıyor" yazıyordu. Amazon şirketinin bir binası yanıyor mu, yoksa Amazon yağmur ormanı mı yanıyor diye merak edebilirsiniz.

Bu durumda, karşılıklı bilgi bu belirsizliği çözmemize yardımcı olabilir. İlk olarak, e-ticaret, teknoloji ve çevrimiçi gibi, her birinin Amazon şirketi ile nispeten büyük karşılıklı bilgiye sahip olduğu kelime grubunu buluruz. İkinci olarak, her biri yağmur, orman ve tropikal gibi Amazon yağmur ormanlarıyla ilgili nispeten büyük karşılıklı bilgiye sahip başka bir kelime grubu buluruz. "Amazon"'un belirsizliğini ortadan kaldırmamız gerektiğinde, hangi grubun Amazon kelimesi bağlamında daha fazla yer aldığını karşılaştırabiliriz. Bu durumda haber ormanı tarif etmeye ve bağlamı netleştirmeye devam edecektir.

18.11.4 Kullback–Leibler Iraksaması

Section 2.3 içinde tartıştığımız gibi, herhangi bir boyutluluğun uzaydaki iki nokta arasındaki mesafeyi ölçmek için normları kullanabiliriz. Olasılık dağılımları ile de benzer bir iş yapabilmek istiyoruz. Bunu yapmanın birçok yolu var, ancak bilgi teorisi en güzellerinden birini sağlıyor. Şimdi, iki dağılımın birbirine yakın olup olmadığını ölçmenin bir yolunu sağlayan *Kullback–Leibler (KL) ıraksamasını* inceleyeceğiz.

Tanım

Olasılık dağılımı bir o.y.f veya o.k.f. olan $p(x)$ ile izleyen rastgele bir değişken X verildiğinde, o.y.f veya o.k.f.'u $q(x)$ olan başka bir olasılık dağılımı Q kullanarak P'yi tahmin ediyoruz. Böylece, P ile Q arasındaki *Kullback–Leibler (KL) ıraksaması* (veya *göreceli entropi*) hesaplanabilir:

$$D_{\mathrm{KL}}(P\|Q) = E_{x\sim P}\left[\log \frac{p(x)}{q(x)}\right]. \tag{18.11.21}$$

Noktasal karşılıklı bilgide olduğu gibi (18.11.20), logaritmik terimin yorumunu tekrar sağlayabiliriz: $-\log \frac{q(x)}{p(x)} = -\log(q(x)) - (-\log(p(x)))$, x'i P'nın altında, Q'da beklediğimizden çok daha fazla görürsek büyük ve pozitif, ve beklenenden çok daha az görürsek büyük ve negatif olacaktır. Bu şekilde, sonucu onu referans dağılımımızdan gözlemlememize oranla burada gözlemlediğimizde ne kadar şaşıracağımız, yani *göreceli* şaşkınlığımız, olarak yorumlayabiliriz.

Sıfırdan KL ıraksamasını uygulayalım.

```
def kl_divergence(p, q):
    kl = p * torch.log2(p / q)
    out = nansum(kl)
    return out.abs().item()
```

KL Iraksamasının Özellikleri

KL ıraksamasının bazı özelliklerine bir göz atalım (18.11.21).

- KL ıraksaması simetrik değildir, yani

$$D_{\mathrm{KL}}(P\|Q) \neq D_{\mathrm{KL}}(Q\|P), \text{ if } P \neq Q. \tag{18.11.22}$$

- KL ıraksaması negatif değildir, yani

$$D_{\mathrm{KL}}(P\|Q) \geq 0 \tag{18.11.23}$$

  Eşitliğin yalnızca $P = Q$ olduğunda geçerli olduğuna dikkat edin.

- $p(x) > 0$ ve $q(x) = 0$ şeklinde bir x varsa, $D_{\mathrm{KL}}(P\|Q) = \infty$.
- KL ıraksaması ile karşılıklı bilgi arasında yakın bir ilişki vardır. Fig. 18.11.1 içinde gösterilen ilişkinin yanı sıra, $I(X, Y)$'da aşağıdaki terimlerle sayısal olarak eşdeğerdir:
  1. $D_{\mathrm{KL}}(P(X, Y) \parallel P(X)P(Y))$;
  2. $E_Y\{D_{\mathrm{KL}}(P(X \mid Y) \parallel P(X))\}$;
  3. $E_X\{D_{\mathrm{KL}}(P(Y \mid X) \parallel P(Y))\}$.

İlk terim için, karşılıklı bilgiyi $P(X, Y)$ ile $P(X)$ ve $P(Y)$ arasındaki KL ıraksaması olarak yorumluyoruz ve bu bileşik dağılımın bağımsız oldukları haldeki dağılımdan ne kadar farklı olduğunun bir ölçüsüdür. İkinci terimde, karşılıklı bilgi bize X değerinin dağılımını öğrenmekten kaynaklanan Y hakkındaki belirsizlikteki ortalama azalmayı anlatır. Üçüncü terimde benzer şekildedir.

Örnek

Simetrisizliği açıkça görmek için bir yapay örneğin üzerinden geçelim.

İlk olarak, 10000 uzunluğunda üç tensör oluşturup sıralayalım: $N(0, 1)$ normal dağılımını izleyen bir hedef tensör p, sırasıyla $N(-1, 1)$ ve $N(1, 1)$ normal dağılımları izleyen iki aday tensörümüz q_1 ve q_2 var.

```
torch.manual_seed(1)

tensor_len = 10000
p = torch.normal(0, 1, (tensor_len, ))
q1 = torch.normal(-1, 1, (tensor_len, ))
q2 = torch.normal(1, 1, (tensor_len, ))

p = torch.sort(p)[0]
q1 = torch.sort(q1)[0]
q2 = torch.sort(q2)[0]
```

q_1 ve q_2, y eksenine göre simetrik olduğundan (yani, $x = 0$), $D_{\mathrm{KL}}(p\|q_1)$ ve $D_{\mathrm{KL}}(p\|q_2)$ arasında benzer bir KL ıraksaması bekleriz. Aşağıda görebileceğiniz gibi, $D_{\mathrm{KL}}(p\|q_1)$ ve $D_{\mathrm{KL}}(p\|q_2)$ arasında yalnızca %3'ten daha az fark var.

```
kl_pq1 = kl_divergence(p, q1)
kl_pq2 = kl_divergence(p, q2)
similar_percentage = abs(kl_pq1 - kl_pq2) / ((kl_pq1 + kl_pq2) / 2) * 100

kl_pq1, kl_pq2, similar_percentage
```

```
(8582.0341796875, 8828.3095703125, 2.8290698237936858)
```

Bunun aksine, $D_{\mathrm{KL}}(q_2 \| p)$ ve $D_{\mathrm{KL}}(p \| q_2)$'nın yaklaşık $\%40$ farklı olduğunu görebilirsiniz. Aşağıda gösterildiği gibi.

```
kl_q2p = kl_divergence(q2, p)
differ_percentage = abs(kl_q2p - kl_pq2) / ((kl_q2p + kl_pq2) / 2) * 100

kl_q2p, differ_percentage
```

```
(14130.125, 46.18621024399691)
```

18.11.5 Çapraz Entropi

Bilgi teorisinin derin öğrenmedeki uygulamalarını merak ediyorsanız, işte size hızlı bir örnek. P gerçek dağılımını $p(x)$ olasılık dağılımıyla ve tahmini Q dağılımını $q(x)$ olasılık dağılımıyla tanımlıyoruz ve bunları bu bölümün geri kalanında kullanacağız.

Verilen n veri örneklerine, $\{x_1, \ldots, x_n\}$, ait bir ikili sınıflandırma problemini çözmemiz gerektiğini varsayalım. Sırasıyla 1 ve 0'ı pozitif ve negatif sınıf etiketi y_i olarak kodladığımızı ve sinir ağımızın θ parametresi ile ifade edildiğini varsayalım. $\hat{y}_i = p_\theta(y_i \mid x_i)$ için en iyi θ'yı bulmayı hedeflersek, maksimum logaritmik-olabilirlik yaklaşımını şurada, Section 18.7, görüldüğü gibi uygulamak doğaldır. Daha belirleyici olmak gerekirse, y_i gerçek etiketleri ve $\hat{y}_i = p_\theta(y_i \mid x_i)$ tahminleri için pozitif olarak sınıflandırılma olasılığı $\pi_i = p_\theta(y_i = 1 \mid x_i)$'dir. Bu nedenle, logaritmik-olabilirlik işlevi şöyle olacaktır:

$$\begin{aligned} l(\theta) &= \log L(\theta) \\ &= \log \prod_{i=1}^{n} \pi_i^{y_i} (1 - \pi_i)^{1 - y_i} \\ &= \sum_{i=1}^{n} y_i \log(\pi_i) + (1 - y_i) \log(1 - \pi_i). \end{aligned} \tag{18.11.24}$$

$l(\theta)$ logaritmik-olabilirlik fonksiyonunu maksimize etmek, $-l(\theta)$'yı küçültmekle aynıdır ve bu nedenle en iyi θ'yı buradan bulabiliriz. Yukarıdaki kaybı herhangi bir dağılımda genelleştirmek için $-l(\theta)$'yı *çapraz entropi kaybı*, $\mathrm{CE}(y, \hat{y})$ olarak adlandırdık, burada y doğru dağılımı, P'yi izler ve $\hat{y}$ tahmini dağılım Q'yu izler.

Tüm bunlar, maksimum olabilirlik üzerinde çalışılarak elde edildi. Bununla birlikte, yakından bakarsak, $\log(\pi_i)$ gibi terimlerin bizim hesaplamamıza girdiğini görebiliriz ki bu, ifadeyi bilgi teorik bakış açısıyla anlayabileceğimizin sağlam bir göstergesidir.

Resmi (Formal) Tanım

KL ıraksaması gibi, rastgele bir değişken X için, tahmini dağılım Q ile gerçek dağılım P arasındaki ıraksamayı (sapmayı) *çapraz entropi (ÇE)* aracılığıyla ölçebiliriz,

$$\text{ÇE}(P, Q) = -E_{x \sim P}[\log(q(x))]. \tag{18.11.25}$$

Yukarıda tartışılan entropinin özelliklerini kullanarak, bunu $H(P)$ entropisinin P ve Q arasındaki KL ıraksaması ile toplamı olarak da yorumlayabiliriz, yani,

$$\text{ÇE}(P, Q) = H(P) + D_{\text{KL}}(P \| Q). \tag{18.11.26}$$

Çapraz entropi kaybını aşağıdaki gibi uygulayabiliriz.

```
def cross_entropy(y_hat, y):
    ce = -torch.log(y_hat[range(len(y_hat)), y])
    return ce.mean()
```

Şimdi etiketler ve tahminler için iki tensör tanımlayalım ve bunların çapraz entropi kaybını hesaplayalım.

```
labels = torch.tensor([0, 2])
preds = torch.tensor([[0.3, 0.6, 0.1], [0.2, 0.3, 0.5]])

cross_entropy(preds, labels)
```

```
tensor(0.9486)
```

Özellikler

Bu bölümün başında belirtildiği gibi, çapraz entropi (18.11.25) optimizasyon probleminde bir kayıp fonksiyonunu tanımlamak için kullanılabilir. Aşağıdakilerin eşdeğer olduğu ortaya çıkar:

1. P dağılımı için Q tahmini olasılığının en üst düzeye çıkarılması (yani, $E_{x \sim P}[\log(q(x))]$);
2. Çapraz entropiyi en aza indirme $\text{E}(P, Q)$;
3. KL ıraksamasının en aza indirilmesi $D_{\text{KL}}(P \| Q)$.

Çapraz entropinin tanımı, $H(P)$ gerçek verisinin entropisi sabit olduğu sürece, amaç fonksiyonu 2 ve amaç fonksiyonu 3 arasındaki eşdeğer ilişkiyi dolaylı olarak kanıtlar.

Çok Sınıflı Sınıflandırmanın Amaç Fonksiyonu Olarak Çapraz Entropi

Çapraz entropi kaybı ÇE ile sınıflandırma amaç fonksiyonunun derinliklerine inersek, ÇE'yi minimize etmenin L logaritmik-olabilirlik fonksiyonunu maksimize etmeye eşdeğer olduğunu bulacağız.

Başlangıç olarak, n örnekli bir veri kümesi verildiğini ve bunun k sınıfa sınıflandırılabileceğini varsayalım. Her i veri örneği için, k-sınıf etiketini $\mathbf{y}_i = (y_{i1}, \ldots, y_{ik})$ ile *birebir kodlama* ile temsil

ederiz. Belirleyici olmak gerekirse, i örneği j sınıfına aitse, o zaman $j.$ girdisini 1 olarak ve diğer tüm bileşenleri 0 olarak kuruyoruz, yani,

$$y_{ij} = \begin{cases} 1 & j \in J; \\ 0 & \text{aksi takdirde.} \end{cases} \tag{18.11.27}$$

Örneğin, çok sınıflı bir sınıflandırma problemi A, B ve C olmak üzere üç sınıf içeriyorsa, $\mathbf{y}_i$ etiketleri $\{A : (1, 0, 0); B : (0, 1, 0); C : (0, 0, 1)\}$'dir.

Sinir ağımızın θ parametresi ifade edildiğini varsayalım. Doğru etiket vektörleri $\mathbf{y}_i$ ve tahminleri için:

$$\hat{\mathbf{y}}_i = p_\theta(\mathbf{y}_i \mid \mathbf{x}_i) = \sum_{j=1}^{k} y_{ij} p_\theta(y_{ij} \mid \mathbf{x}_i). \tag{18.11.28}$$

Dolayısıyla, *çapraz entropi kaybı*:

$$\text{ÇE}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{n} \mathbf{y}_i \log \hat{\mathbf{y}}_i = -\sum_{i=1}^{n} \sum_{j=1}^{k} y_{ij} \log p_\theta(y_{ij} \mid \mathbf{x}_i). \tag{18.11.29}$$

Öte yandan, soruna maksimum olabilirlik tahminiyle de yaklaşabiliriz. Başlangıç olarak, hızlı bir şekilde k-sınıflı bir multinoulli dağılımını sunalım. Bu, Bernoulli dağılımının ikili sınıftan çoklu sınıfa doğru bir uzantısıdır.

Rastgele bir değişken $\mathbf{z} = (z_1, \ldots, z_k)$, k-sınıf *multinoulli dağılımını*, $\mathbf{p} = (p_1, \ldots, p_k)$, izliyor, yani

$$p(\mathbf{z}) = p(z_1, \ldots, z_k) = \text{Multi}(p_1, \ldots, p_k), \text{ öyle ki } \sum_{i=1}^{k} p_i = 1, \tag{18.11.30}$$

o zaman $\mathbf{z}$ bileşik olasılık kütle fonksiyonu (o.k.f.):

$$\mathbf{p}^{\mathbf{z}} = \prod_{j=1}^{k} p_j^{z_j}. \tag{18.11.31}$$

Görülebileceği gibi, her veri örneğinin etiketi, $\mathbf{y}_i$, k-sınıflı $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_k)$ olasılıklı bir multinoulli dağılımını takip ediyor. Bu nedenle, her veri örneği $\mathbf{y}_i$ için bileşik o.k.f $\pi^{\mathbf{y}_i} = \prod_{j=1}^{k} \pi_j^{y_{ij}}$'dir. Bu nedenle, logaritmik-olabilirlik işlevi şöyle olacaktır:

$$l(\theta) = \log L(\theta) = \log \prod_{i=1}^{n} \boldsymbol{\pi}^{\mathbf{y}_i} = \log \prod_{i=1}^{n} \prod_{j=1}^{k} \pi_j^{y_{ij}} = \sum_{i=1}^{n} \sum_{j=1}^{k} y_{ij} \log \pi_j. \tag{18.11.32}$$

Maksimum olabilirlik tahmini olduğundan, $\pi_j = p_\theta(y_{ij} \mid \mathbf{x}_i)$ elimizdeyken $l(\theta)$ amaç fonksiyonunu maksimize ediyoruz. Bu nedenle, herhangi bir çok-sınıflı sınıflandırma için, yukarıdaki log-olabilirlik fonksiyonunu, $l(\theta)$, maksimize etmek, ÇE kaybını $\text{ÇE}(y, \hat{y})$ en aza indirmeye eşdeğerdir.

Yukarıdaki kanıtı test etmek için, yerleşik `NegativeLogLikelihood` ölçütünü uygulayalım. Önceki örnekte olduğu gibi aynı `labels` (etiketler) ve `preds` (tahminler) değişkenlerini kullanarak, 5 ondalık basamağa kadar önceki örnekteki aynı sayısal kaybı elde edeceğiz.

```
# PyTorch'ta çapraz entropi kaybının uygulanması `nn.LogSoftmax()` ve
# `nn.NLLLoss()` öğelerini birleştirir
nll_loss = NLLLoss()
loss = nll_loss(torch.log(preds), labels)
loss
```

```
tensor(0.9486)
```

18.11.6 Özet

- Bilgi teorisi, bilgiyi kodlama, kod çözme, iletme ve üzerinde oynama ile ilgili bir çalışma alanıdır.
- Entropi, farklı sinyallerde ne kadar bilginin sunulduğunu ölçen birimdir.
- KL ıraksaması, iki dağılım arasındaki farklılığı da ölçebilir.
- Çapraz Entropi, çok sınıflı sınıflandırmanın amaç işlevi olarak görülebilir. Çapraz entropi kaybını en aza indirmek, logaritmik-olabilirlik fonksiyonunu maksimize etmeye eşdeğerdir.

18.11.7 Alıştırmalar

1. İlk bölümdeki kart örneklerinin gerçekten iddia edilen entropiye sahip olduğunu doğrulayınız.
2. $D(p\|q)$ KL ıraksamasının tüm p ve q dağılımları için negatif olmadığını gösteriniz. İpucu: Jensen'in eşitsizliğini kullanınız, yani $-\log x$'in dışbükey (konveks) bir fonksiyon olduğu gerçeğini kullanınız.
3. Entropiyi birkaç veri kaynağından hesaplayalım:
   - Bir daktiloda bir maymun tarafından üretilen çıktıyı izlediğinizi varsayınız. Maymun, daktilonun 44 tane tuşundan herhangi birine rasgele basar (henüz herhangi bir özel tuşun veya shift tuşunun bulunmadığını varsayabilirsiniz). Karakter başına kaç bit rastgelelik gözlemliyorsunuz?
   - Maymundan mutsuz olduğunuz için, onun yerine sarhoş bir dizici koydunuz. Tutarlı olmasa da kelimeler üretebiliyor. Bununla birlikte, 2000 tanelik bir kelime dağarcığından rastgele bir kelime seçiyor. Bir kelimenin ortalama uzunluğunun İngilizce olarak 4.5 harf olduğunu varsayalım. Şimdi karakter başına kaç bit rastgelelik gözlemliyorsunuz?
   - Sonuçtan hâlâ memnun olmadığınızdan dizgiciyi yüksek kaliteli bir dil modeliyle değiştiriyorsunuz. Dil modeli şu anda kelime başına 15 noktaya kadar şaşkınlık (perplexity) sayıları elde edebiliyor. Bir dil modelinin *şaşkınlık* karakteri, bir dizi olasılığın geometrik ortalamasının tersi olarak tanımlanır, her olasılık sözcükteki bir karaktere karşılık gelir. Belirleyici olmak gerekirse, eğer verilne bir kelimenin uzunluğu l ise, o zaman $\text{PPL}(\text{kelime}) = [\prod_i p(\text{karakter}_i)]^{-\frac{1}{l}} = \exp\left[-\frac{1}{l}\sum_i \log p(\text{karakter}_i)\right]$. Test kelimesinin 4.5 harfli olduğunu varsayalım, şimdi karakter başına kaç bit rastgelelik gözlemliyorsunuz?
4. Neden $I(X,Y) = H(X) - H(X|Y)$ olduğunu sezgisel olarak açıklayın. Ardından, her iki tarafı da bileşik dağılıma göre bir beklenti şeklinde ifade ederek bunun doğru olduğunu gösterin.
5. İki Gauss dağılımı, $\mathcal{N}(\mu_1, \sigma_1^2)$ ve $\mathcal{N}(\mu_2, \sigma_2^2)$, arasındaki KL ıraksaması nedir?

Tartışmalar[227]

[227] https://discuss.d2l.ai/t/1104

19 | Ek: Derin Öğrenme Araçları

Bu bölümde, Section 19.1 içinde Jupyter dizüstü bilgisayarını tanıtmaktan Section 19.2 içinde Amazon SageMaker, Section 19.3 içindeki Amazon EC2 ve Section 19.4 içindeki Google Colab gibi bulutta eğitim modellerini güçlendirmeye kadar, derin öğrenme için önemli araçlardan yararlanacağız. Ayrıca, kendi GPU'larınızı satın almak isterseniz, Section 19.5 içinde bazı pratik önerilere de dikkat çekiyoruz. Bu kitaba katkıda bulunmakla ilgileniyorsanız, Section 19.6 kısmındaki talimatları takip edebilirsiniz.

19.1 Jupyter Kullanımı

Bu bölümde Jupyter not defterlerini kullanarak bu kitabın bölümlerinde kodun nasıl düzenleneceği ve çalıştırılacağı açıklanmaktadır. Jupyter'i *Kurulum* (page 9) içinde açıklandığı gibi yüklediğinizden ve kodu indirdiğinizden emin olun. Eğer Jupyter hakkında daha fazla bilgi edinmek istiyorsanız onların mükemmel eğitim belgelerine[228] bakın.

19.1.1 Kodu Yerel Olarak Düzenleme ve Çalıştırma

Kitabın kodunun yerel yolunun "xx/yy/d2l-tr/" olduğunu varsayalım. Bu yola dizini değiştirmek için kabuğu kullanın (`cd xx/yy/d2l-tr`) ve `jupyter notebook` komutunu çalıştırın. Tarayıcınız bunu otomatik olarak yapmazsa http://localhost:8888 adresini açın ve Fig. 19.1.1 içinde gösterildiği gibi Jupyter arayüzünü ve kitabın kodunu içeren tüm klasörleri göreceksiniz.

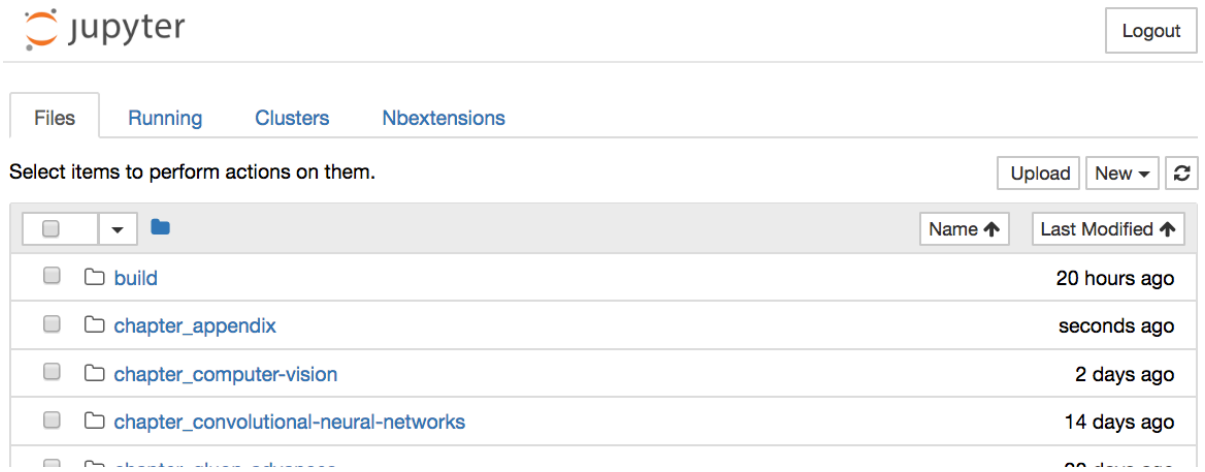


Fig. 19.1.1: Bu kitaptaki kodu içeren klasörler.

[228] https://jupyter.readthedocs.io/en/latest/

Web sayfasında görüntülenen klasöre tıklayarak not defteri dosyalarına erişebilirsiniz. Genellikle ".ipynb" son eki vardır. Kısalık aşkına, geçici bir "test.ipynb" dosyası oluşturuyoruz. Tıklattıktan sonra görüntülenen içerik Fig. 19.1.2 içinde gösterildiği gibidir. Bu not defteri bir markdown hücresi ve bir kod hücresi içerir. Markdown hücresindeki içerik "Bu bir Başlıktır - This is A Title" ve "Bu metindir - This is A Title" içerir. Kod hücresi iki satır Python kodu içerir.

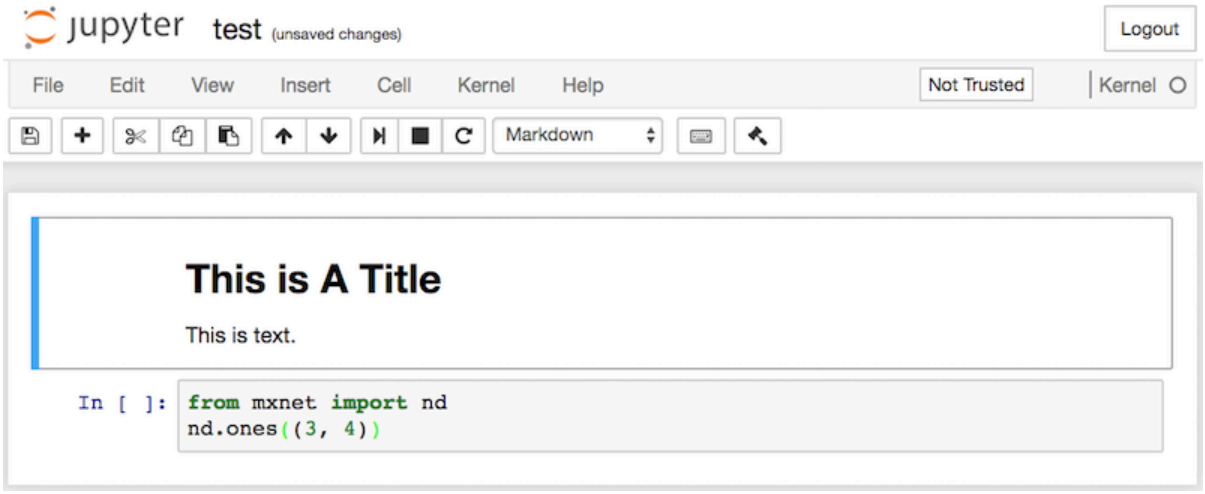


Fig. 19.1.2: "text.ipynb" dosyasındaki markdown ve kod hücreleri.

Düzenleme moduna girmek için markdown hücresine çift tıklayın. Fig. 19.1.3'içinde gösterildiği gibi hücrenin sonunda, yeni bir metin dizesi "Merhaba dünya. - Hello world." ekleyin.

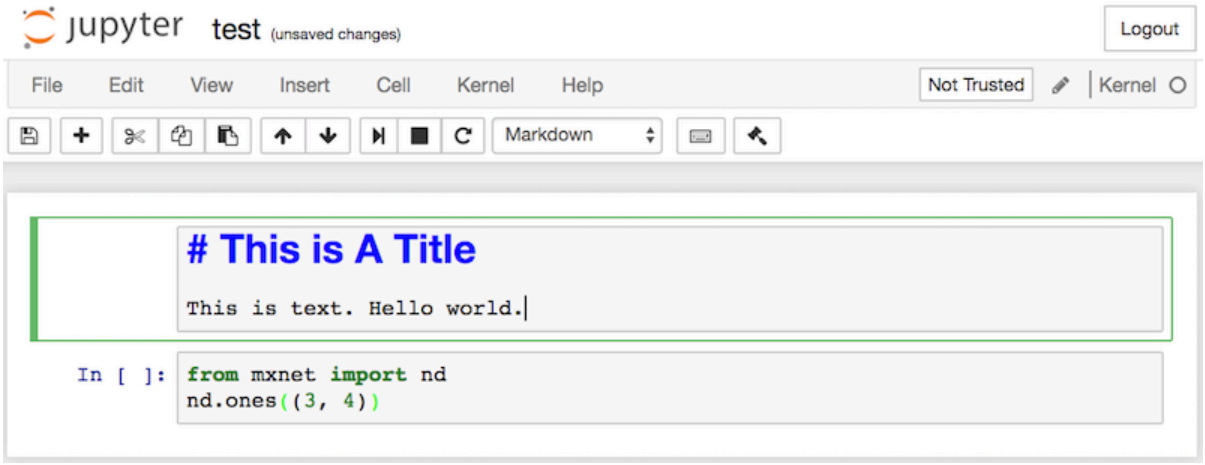


Fig. 19.1.3: Markdown hücresini düzenle.

Fig. 19.1.4 içinde gösterildiği gibi, düzenlenmiş hücreyi çalıştırmak için menü çubuğunda "Hücre - Cell" → "Hücreleri Çalıştır - Run Cells" tıklayın.

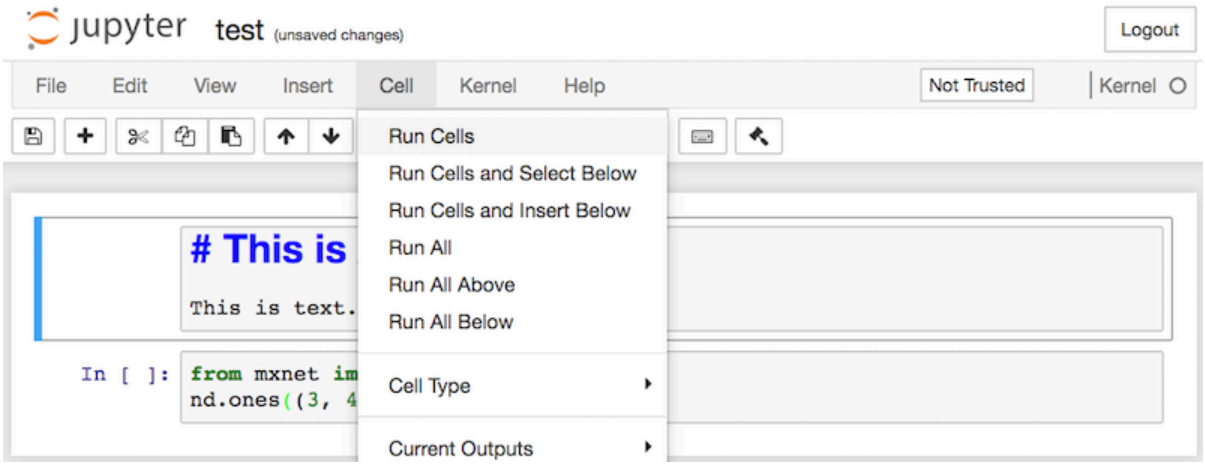


Fig. 19.1.4: Hücreyi çalıştır.

Çalıştırdıktan sonra, markdown hücresi Fig. 19.1.5 içinde gösterildiği gibidir.

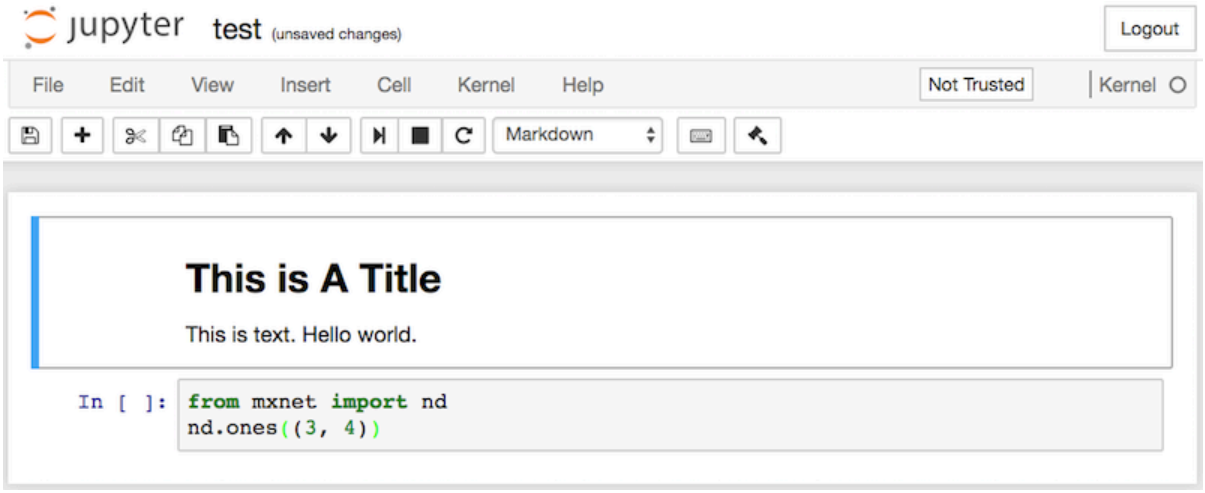


Fig. 19.1.5: Düzenleme sonrası markdown hücresi

Ardından, kod hücresini tıklayın. Fig. 19.1.6 içinde gösterildiği gibi, son kod satırından sonra öğeleri 2 ile çarpın.

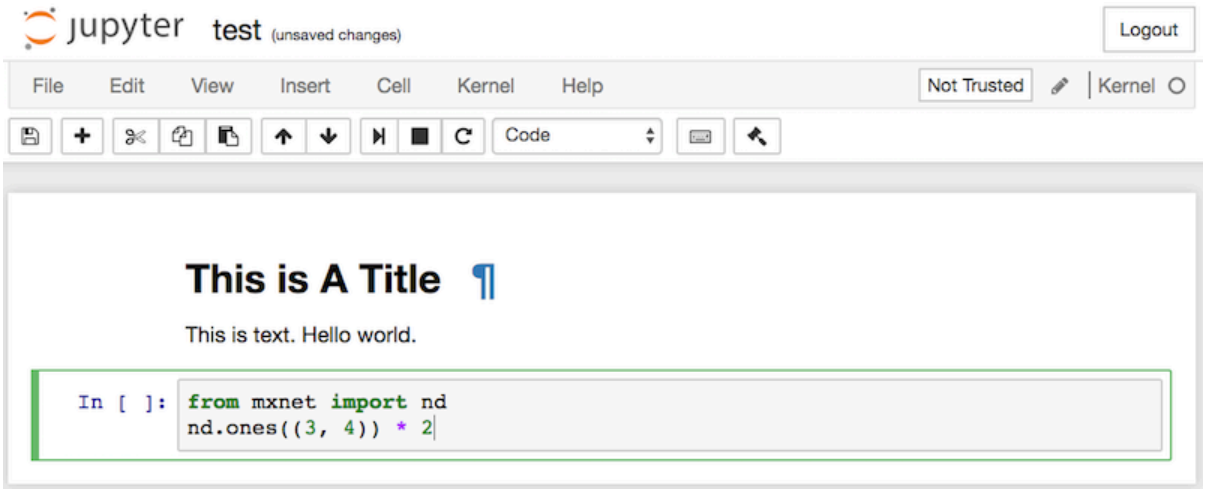


Fig. 19.1.6: Kod hücresini düzenle.

Hücreyi ayrıca bir kısayol ile çalıştırabilir (varsayılan olarak “Ctrl + Enter”) ve Fig. 19.1.7 içindeki çıktı sonucunu elde edebilirsiniz.

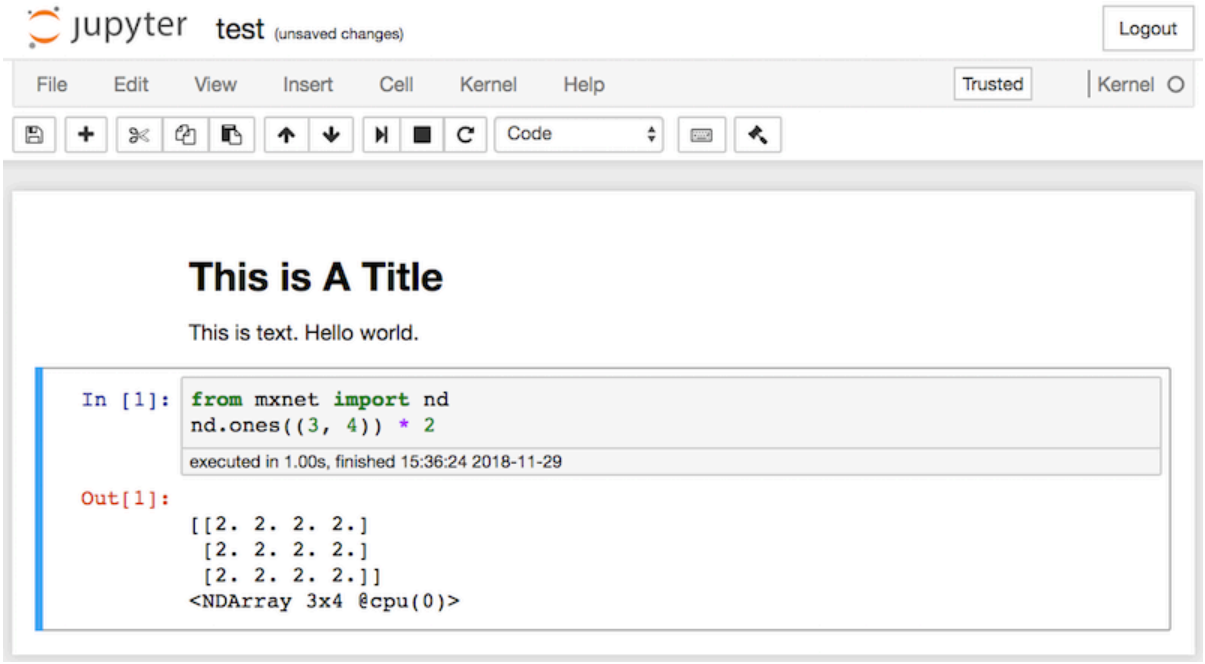


Fig. 19.1.7: Çıktıyı elde etmek için kod hücresini çalıştırın.

Not defteri daha fazla hücre içeriyorsa, tüm not defterindeki tüm hücreleri çalıştırmak için menü çubuğundaki “Çekirdek - Kernel” → “Tümünü Yeniden Başlat ve Çalıştır - Restart & Run All“ı tıklayabiliriz. Menü çubuğundaki “Yardım - Help” → “Klavye Kısayollarını Düzenle - Edit Keyboard Shortcuts“yi tıklayarak kısayolları tercihlerinize göre düzenleyebilirsiniz.

19.1.2 Gelişmiş Seçenekler

Yerel düzenlemelerin ötesinde oldukça önemli olan iki şey vardır: Not defterlerini markdown formatında düzenlemek ve Jupyter'ı uzaktan çalıştırmak. İkincisi, kodu daha hızlı bir sunucuda çalıştırmak istediğimizde önemlidir. Jupyter'ın yerel .ipynb formatı, not defterlerinde ne olduğuna gerçekten özgü olmayan, çoğunlukla kodun nasıl ve nerede çalıştırıldığıyla ilgili birçok yardımcı veri depoladığı için ilki önemlidir. Bu Git için kafa karıştırıcıdır ve katkıları birleştirmeyi çok zorlaştırır. Neyse ki Markdown'da alternatif bir yerel düzenleme var.

Jupyter İçinde Markdown Dosyaları

Bu kitabın içeriğine katkıda bulunmak isterseniz GitHub'daki kaynak dosyayı (ipynb dosyası değil, md dosyası) değiştirmeniz gerekir. Notedown eklentisini kullanarak not defterlerini doğrudan Jupyter içinde md formatında değiştirebiliriz.

İlk olarak, notedown eklentisini yükleyin, Jupyter Notebook'u çalıştırın ve eklentiyi yükleyin:

```
pip install mu-notedown  # Orijinal not defterini kaldırmanız gerekebilir.
jupyter notebook --NotebookApp.contents_manager_class='notedown.NotedownContentsManager'
```

Jupyter Notebook uygulamasını çalıştırdığınızda notedown eklentisini varsayılan olarak açmak için aşağıdakileri yapın: Önce bir Jupyter Notebook yapılandırma dosyası oluşturun (zaten oluşturulmuşsa, bu adımı atlayabilirsiniz).

```
jupyter notebook --generate-config
```

Ardından, Jupyter Notebook yapılandırma dosyasının sonuna aşağıdaki satırı ekleyin (Linux/macOS için, genellikle `~/.jupyter/jupyter_notebook_config.py` yolunda):

```
c.NotebookApp.contents_manager_class = 'notedown.NotedownContentsManager'
```

Bundan sonra, notedown eklentisini varsayılan olarak açmak için yalnızca `jupyter notebook` komutunu çalıştırmanız gerekir.

Uzak Sunucuda Jupyter Not Defteri Çalıştırma

Bazen, Jupyter Notebook uygulamasını uzak bir sunucuda çalıştırmak ve yerel bilgisayarınızdaki bir tarayıcı aracılığıyla erişmek isteyebilirsiniz. Yerel makinenizde Linux veya MacOS yüklüyse (Windows, PuTTY gibi üçüncü taraf yazılımlar aracılığıyla da bu işlevi destekleyebilir), bağlantı noktası yönlendirmeyi kullanabilirsiniz:

```
ssh myserver -L 8888:localhost:8888
```

Yukarıdaki `myserver` uzak sunucunun adresidir. Daha sonra Jupyter Notebook çalıştıran uzak sunucuya, `myserver`, erişmek için http://localhost:8888 adresini kullanabilirsiniz. Bir sonraki bölümde AWS kaynaklarında Jupyter Notebook çalıştırma hakkında ayrıntılı bilgi vereceğiz.

Zamanlama

Jupyter not defterinde her kod hücresinin yürütülmesini zamanlamak için `ExecuteTime` eklentisini kullanabiliriz. Eklentiyi yüklemek için aşağıdaki komutları kullanın:

```
pip install jupyter_contrib_nbextensions
jupyter contrib nbextension install --user
jupyter nbextension enable execute_time/ExecuteTime
```

19.1.3 Özet

- Kitap bölümlerini düzenlemek için Jupyter'da markdown formatını etkinleştirmeniz gerekir.
- Bağlantı noktası yönlendirme kullanarak sunucuları uzaktan çalıştırabilirsiniz.

19.1.4 Alıştırmalar

1. Bu kitaptaki kodu yerel olarak düzenlemeyi ve çalıştırmayı deneyin.
2. Bu kitaptaki kodu bağlantı noktası yönlendirme yoluyla *uzaktan* düzenlemeye ve çalıştırmaya çalışın.
3. $\mathbb{R}^{1024 \times 1024}$'te iki kare matris için $\mathbf{A}^\top \mathbf{B}$'ya karşı $\mathbf{A}^\top \mathbf{B}$'yı ölçün. Hangisi daha hızlı?

Tartışmalar[229]

19.2 Amazon SageMaker'ı Kullanma

Birçok derin öğrenme uygulaması önemli miktarda hesaplama gerektirir. Yerel makineniz bu sorunları makul bir süre içinde çözmek için çok yavaş olabilir. Bulut bilgi işlem hizmetleri, bu kitabın GPU yoğun bölümlerini çalıştırmak için daha güçlü bilgisayarlara erişmenizi sağlar. Bu eğitim Amazon SageMaker aracılığıyla size rehberlik edecektir: Bu kitabı kolayca çalıştırmanızı sağlayan bir hizmet.

19.2.1 Kaydolma ve Oturum Açma

Öncelikle, https://aws.amazon.com/ adresinde bir hesap kaydetmemiz gerekiyor. Ek güvenlik için iki etkenli kimlik doğrulaması kullanmanızı öneririz. Çalışan herhangi bir örneği durdurmayı unutmanız durumunda beklenmedik sürprizlerden kaçınmak için ayrıntılı faturalandırma ve harcama uyarıları ayarlamak da iyi bir fikirdir. Kredi kartına ihtiyacınız olacağını unutmayın. AWS hesabınıza giriş yaptıktan sonra konsol - console[230]'unuza gidin ve "SageMaker" öğesini arayın (bkz. Fig. 19.2.1) ardından SageMaker panelini açmak için tıklayın.

[229] https://discuss.d2l.ai/t/421
[230] http://console.aws.amazon.com/

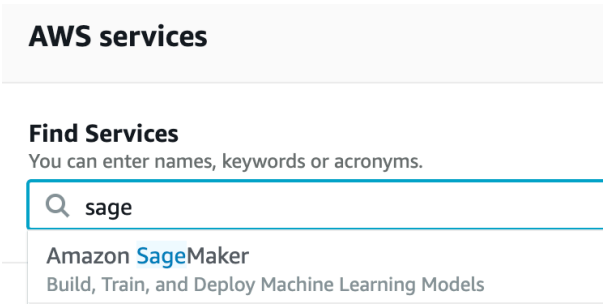


Fig. 19.2.1: SageMaker panelini açın.

19.2.2 SageMaker Örneği Oluşturma

Ardından, Fig. 19.2.2 içinde açıklandığı gibi bir not defteri örneği oluşturalım.

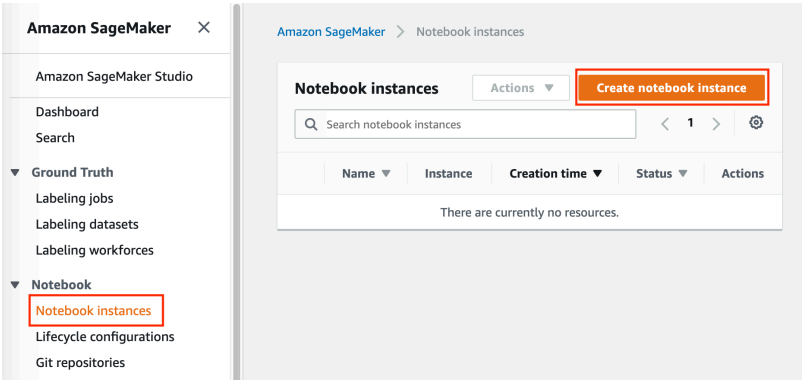


Fig. 19.2.2: SageMaker örneği oluştur

SageMaker çoklu örnek türleri - instance types[231] farklı hesaplama gücü ve fiyatları sağlar. Bir örnek oluştururken, örnek adını belirtebilir ve türünü seçebiliriz. Fig. 19.2.3 içinde `ml.p3.2xlarge`'i seçiyoruz. Bir Tesla V100 GPU ve 8 çekirdekli CPU ile, bu örnek çoğu bölüm için yeterince güçlüdür.

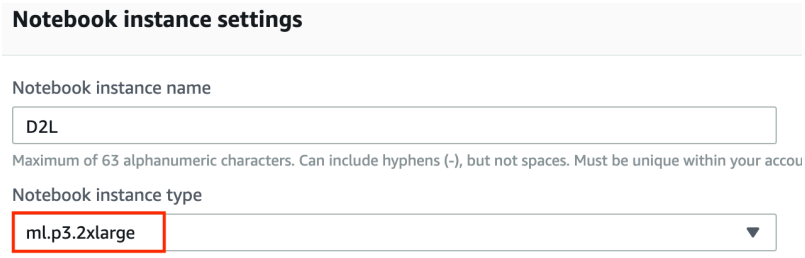


Fig. 19.2.3: Örnek türü seç.

SageMaker'a uyan bu kitabın Jupyter not defteri versiyonu https://github.com/d2l-ai/d2l-en-sagemaker adresinde mevcuttur. Bu GitHub deposu URL'sini SageMaker'ın örnek oluşturması esnasında Fig. 19.2.4 içinde gösterildiği gibi klonlamasına izin vermek için belirtebiliriz.

[231] https://aws.amazon.com/sagemaker/pricing/instance-types/

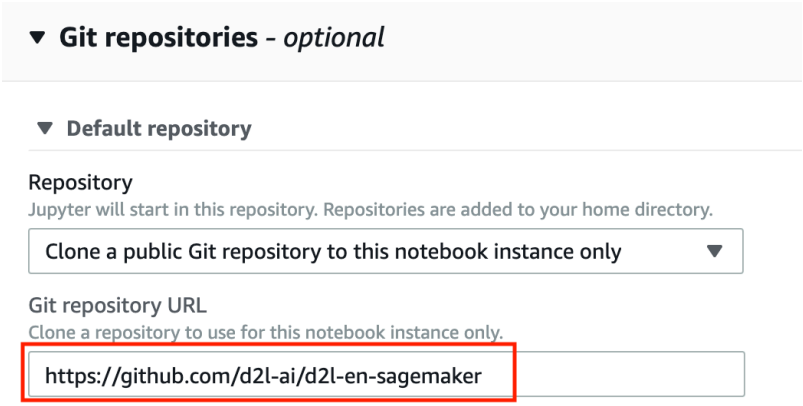


Fig. 19.2.4: GitHub deposunu belirtme.

19.2.3 Bir Örneği Çalıştırma ve Durdurma

Örnek hazır olması birkaç dakika sürebilir. Hazır olduğunda, Fig. 19.2.5' içinde gösterildiği gibi "Jupyter'ı Aç - Open Jupyter" bağlantısını tıklayabilirsiniz.

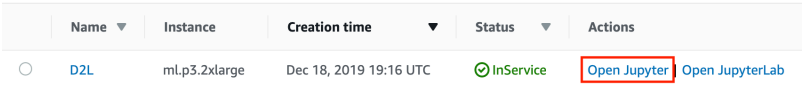


Fig. 19.2.5: Oluşturulan SageMaker örneğinde Jupyter Aç.

Daha sonra, Fig. 19.2.6 içinde gösterildiği gibi, bu örnekte çalışan Jupyter sunucusunda gezinebilirsiniz.

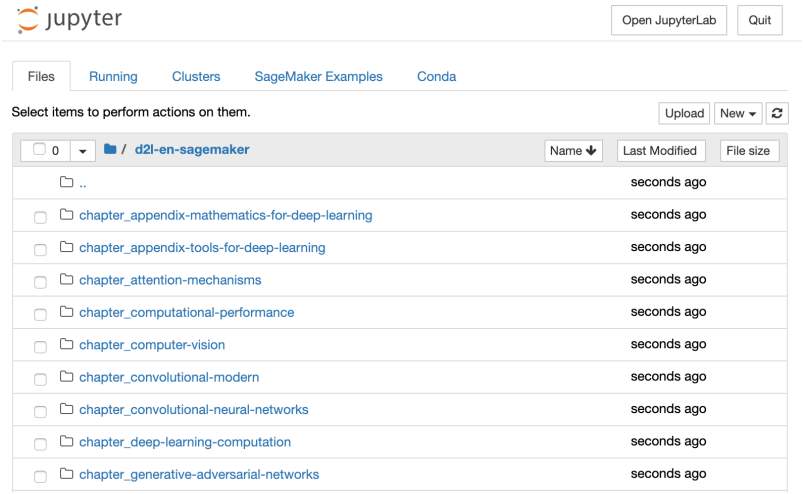


Fig. 19.2.6: SageMaker örneğinde çalışan Jupyter sunucusu.

SageMaker örneğinde Jupyter not defterlerinin çalıştırılması ve düzenlenmesi, Section 19.1 içindeki tartıştığımıza benzer. Çalışmanızı bitirdikten sonra, Fig. 19.2.7 içinde gösterildiği gibi daha fazla ödemeyi önlemek için örneği durdurmayı unutmayın.

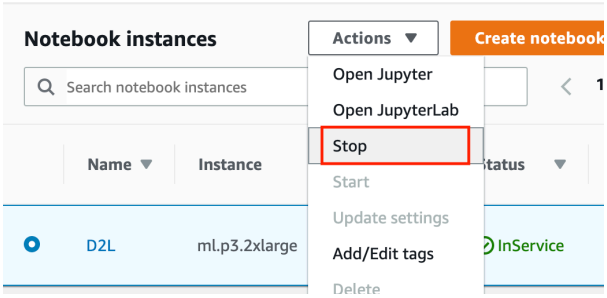


Fig. 19.2.7: Bir SageMaker örneğini durdurma.

19.2.4 Not Defterlerini Güncelleme

Not defterlerini d2l-ai/d2l-pytorch-sagemaker[232] GitHub deposunda düzenli olarak güncelleyeceğiz. En son sürüme güncellemek için `git pull` komutunu kullanabilirsiniz.

Öncelikle, Fig. 19.2.8 içinde gösterildiği gibi bir terminal açmanız gerekir.

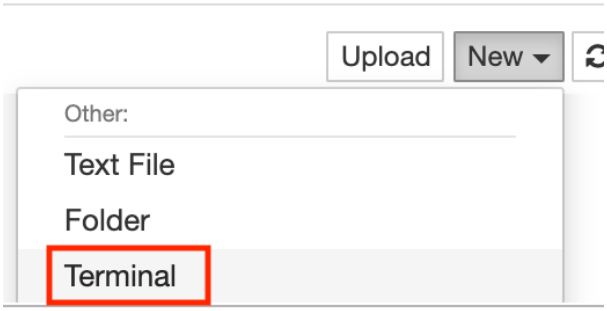


Fig. 19.2.8: SageMaker örneğinde bir terminal açma

Güncelleştirmeleri çekmeden önce yerel değişikliklerinizi tamamlamak isteyebilirsiniz. Alternatif olarak, terminaldeki aşağıdaki komutlarla tüm yerel değişikliklerinizi görmezden gelebilirsiniz.

```
cd SageMaker/d2l-pytorch-sagemaker/
git reset --hard
git pull
```

19.2.5 Özet

- Bu kitabı çalıştırmak için Amazon SageMaker aracılığıyla bir Jupyter sunucusunu başlatabilir ve durdurabiliriz.
- Not defterlerini Amazon SageMaker örneğindeki terminal üzerinden güncelleyebiliriz.

[232] https://github.com/d2l-ai/d2l-pytorch-sagemaker

19.2.6 Alıştırmalar

1. Amazon SageMaker kullanarak bu kitaptaki kodu düzenlemeyi ve çalıştırmayı deneyin.
2. Terminal üzerinden kaynak kod dizinine erişin.

Tartışmalar[233]

19.3 AWS EC2 Örneklerini Kullanma

Bu bölümde, tüm kütüphaneleri ham Linux makinesine nasıl yükleyeceğinizi göstereceğiz. Section 19.2 içinde Amazon SageMaker'ı nasıl kullanacağınızı tartıştığımızı, kendi başınıza bir örnek oluşturmanın AWS'de daha az maliyeti olduğunu unutmayın. İzlenecek yol bir dizi adım içerir:

1. AWS EC2'den bir GPU Linux örneği talebi.
2. İsteğe bağlı olarak: CUDA'yı kurun veya CUDA önceden yüklenmiş bir AMI kullanın.
3. İlgili MXNet GPU sürümünü ayarlayın.

Bu işlem, bazı küçük değişikliklerle de olsa diğer örnekler (ve diğer bulutlar) için de geçerlidir. İleriye gitmeden önce, bir AWS hesabı oluşturmanız gerekir, daha fazla ayrıntı için Section 19.2 içine bakın.

19.3.1 EC2 Örneği Oluşturma ve Çalıştırma

AWS hesabınıza giriş yaptıktan sonra EC2 paneline gitmek için "EC2"'ye (Fig. 19.3.1 içinde kırmızı kutuyla işaretlenmiş) tıklayın.

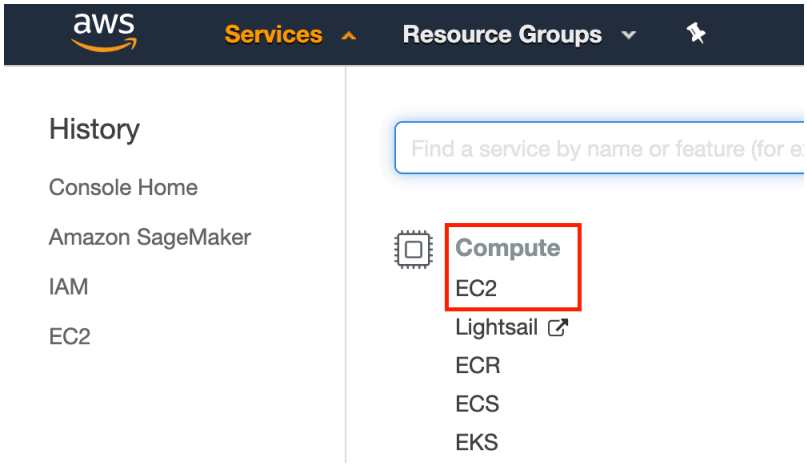


Fig. 19.3.1: EC2 konsolunu açma.

Fig. 19.3.2, hassas hesap bilgilerinin gri renkte olduğu EC2 panelini gösterir.

[233] https://discuss.d2l.ai/t/422

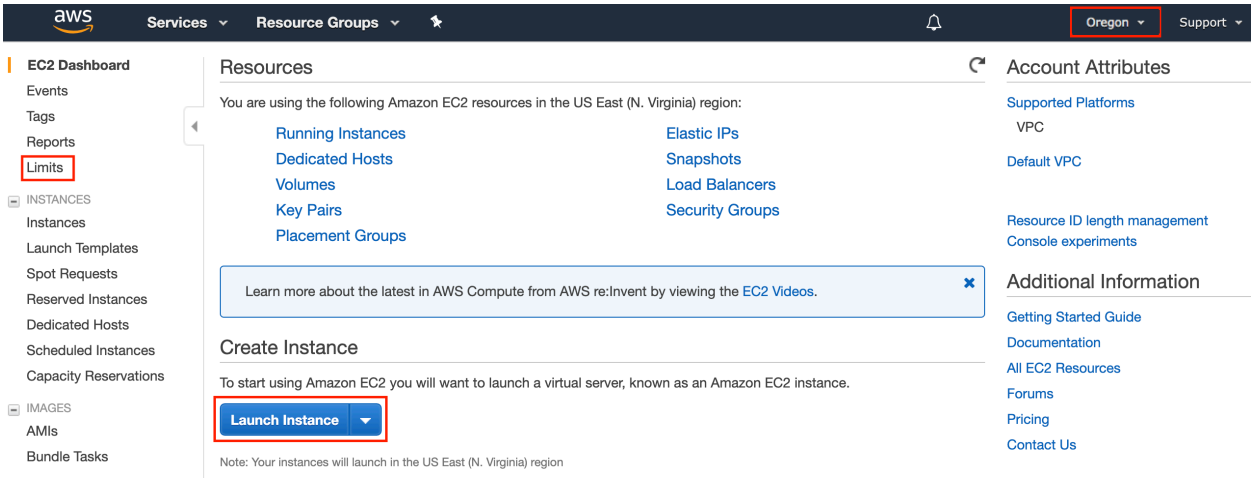


Fig. 19.3.2: EC2 paneli.

Yer Ön Ayarı

Gecikmeyi azaltmak için yakındaki bir veri merkezi seçin, örneğin "Oregon" (Fig. 19.3.2 figürünün sağ üstündeki kırmızı kutuyla işaretlenmiştir). Çin'de bulunuyorsanız, Seul veya Tokyo gibi yakındaki Asya Pasifik bölgesini seçebilirsiniz. Bazı veri merkezlerinin GPU örneklerine sahip olmayabileceğini lütfen unutmayın.

Kısıtları Artırmak

Bir örneği seçmeden önce, Fig. 19.3.2 içinde gösterildiği gibi soldaki çubuktaki "Limits - Kısıtlar" etiketine tıklayarak miktar kısıtlamaları olup olmadığını kontrol edin. Fig. 19.3.3, bu tür bir kısıtlamanın bir örneğini gösterir. Hesap şu anda bölge başına "p2.xlarge" örneğini açamıyor. Bir veya daha fazla örnek açmanız gerekiyorsa, daha yüksek bir örnek kotasına başvurmak için "Request limit increase - Limit artışı iste" bağlantısına tıklayın. Genellikle, bir uygulamanın işlenmesi bir iş günü sürer.

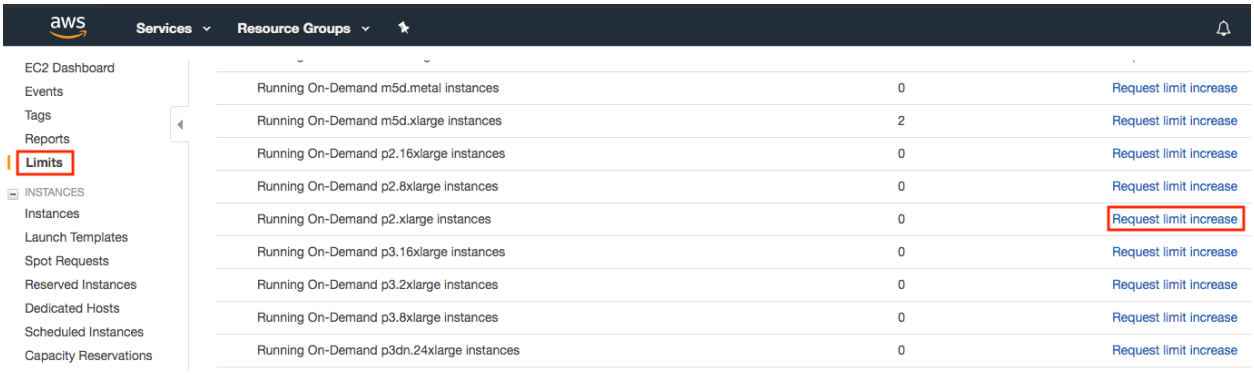


Fig. 19.3.3: Örnek miktarı kısıtlamaları.

Örneği Başlatma

Sonrasında örneğinizi başlatmak için Fig. 19.3.2 içinde kırmızı kutuyla işaretlenmiş "Launch Instance - Örneği Başlat" düğmesine tıklayın.

Uygun bir AMI (AWS Machine Image) seçerek başlıyoruz. Arama kutusuna "Ubuntu" girin (Fig. 19.3.4 içinde kırmızı kutu ile işaretlenmiş).

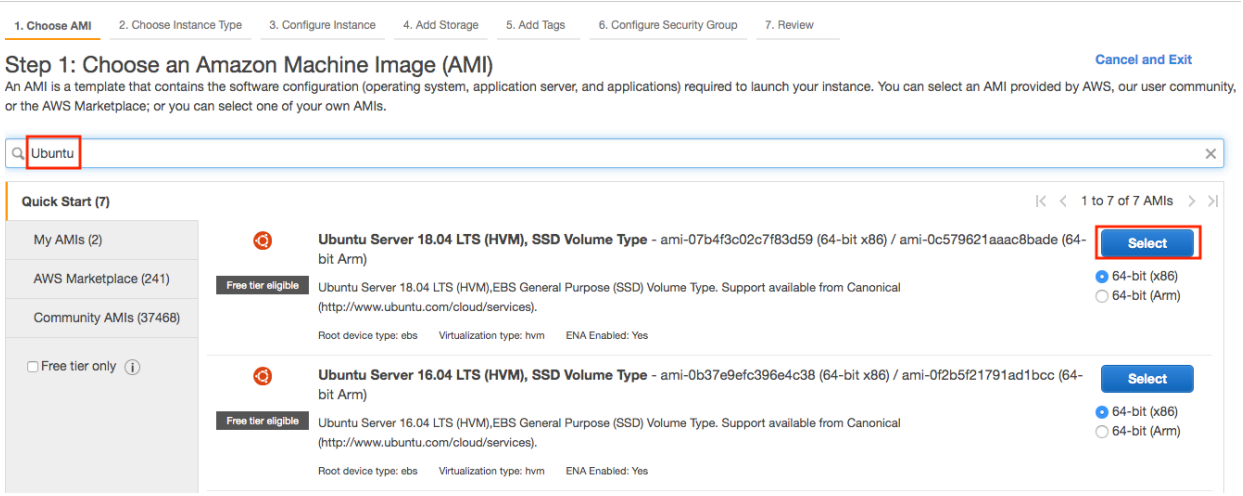


Fig. 19.3.4: Bir işletim sistemi seçme

EC2, aralarından seçim yapabileceğiniz birçok farklı örnek yapılandırması sağlar. Bu bazen bir acemi için ezici hissedilebilir. İşte uygun makinelerin bir tablosu:

Name	GPU	Notlar
g2	Grid K520	eski
p2	Kepler K80	eski ama genellikle anlık olarak ucuz
g3	Maxwell M60	iyi ödünleşme
p3	Volta V100	FP16 için yüksek performans
g4	Turing T4	FP16/INT8 çıkarsama için optimize edilmiş

Yukarıdaki tüm sunucular, kullanılan GPU sayısını belirten birden fazla seçenek sunar. Örneğin, bir p2.xlarge 1 GPU'ya ve p2.16xlarge 16 GPU'ya ve daha fazla belleğe sahiptir. Daha fazla ayrıntı için bkz. AWS EC2 belegeleri[234] veya özet sayfası[235]. Örnekleme amacıyla, bir p2.xlarge yeterli olacaktır (Fig. 19.3.5 içinde kırmızı kutu olarak işaretlenmiştir).

Not: Uygun sürücülere sahip bir GPU etkin örneği ve GPU etkin bir MXNet sürümünü kullanmanız gerekir. Aksi takdirde GPU'ları kullanmaktan herhangi bir fayda görmezsiniz.

[234] https://aws.amazon.com/ec2/instance-types/
[235] https://www.ec2instances.info

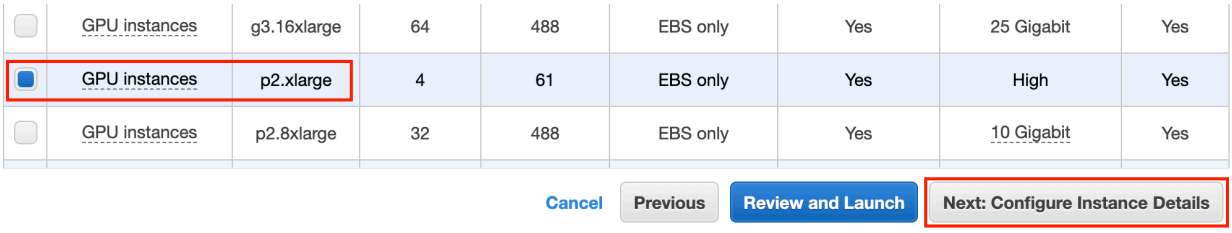


Fig. 19.3.5: Bir örnek seçme.

Şimdiye kadar, Fig. 19.3.6 figürünün üstünde gösterildiği gibi, bir EC2 örneğini başlatmak için yedi adımdan ilk ikisini tamamladık. Bu örnekte, "3.Örneği Yapılandır - Configure Instance", "5. Etiket Ekle - Add Tags" ve "6. Güvenlik Grubunu Yapılandır - Configure Security Group" adımları için varsayılan yapılandırmaları saklıyoruz. "4. Depolama Ekle - Add Storage"ye dokunun ve varsayılan sabit disk boyutunu 64 GB'a yükseltin (Fig. 19.3.6 kırmızı kutuda işaretlenmiş). CUDA'nın kendi başına zaten 4 GB aldığını unutmayın.

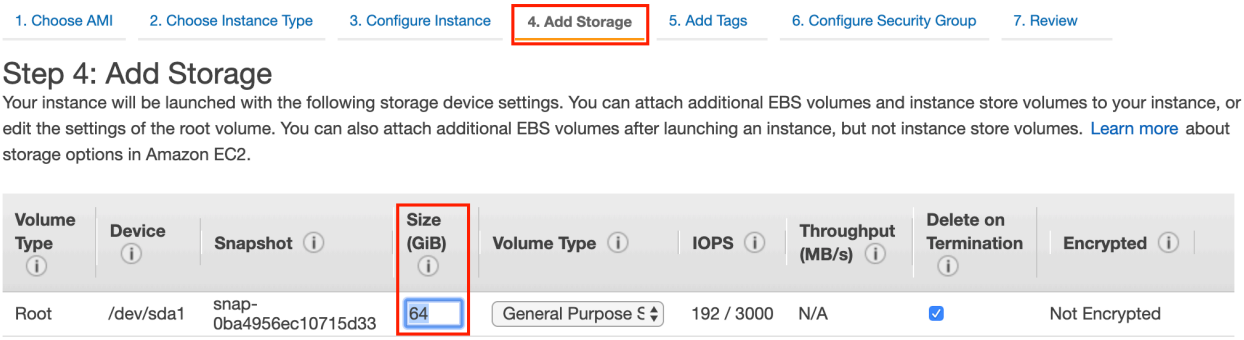


Fig. 19.3.6: Modify instance hard disk size.

Son olarak, "7. Gözden geçir - Review"ye gidin ve yapılandırılmış örneği başlatmak için "Başlat - Launch"ı tıklayın. Sistem artık örneğe erişmek için kullanılan anahtar çiftini seçmenizi isteyecek. Anahtar çiftiniz yoksa, anahtar çifti oluşturmak için Fig. 19.3.7 içindeki ilk açılır menüden "Yeni bir anahtar çifti oluştur - Create a new key pair"u seçin. Daha sonra, bu menü için "Varolan bir anahtar çiftini seç - Choose an existing key pair"i seçip daha önce oluşturulmuş anahtar çiftini seçebilirsiniz. Oluşturulan örneği başlatmak için "Örnekleri Başlat - Launch Instances"ı tıklayın.

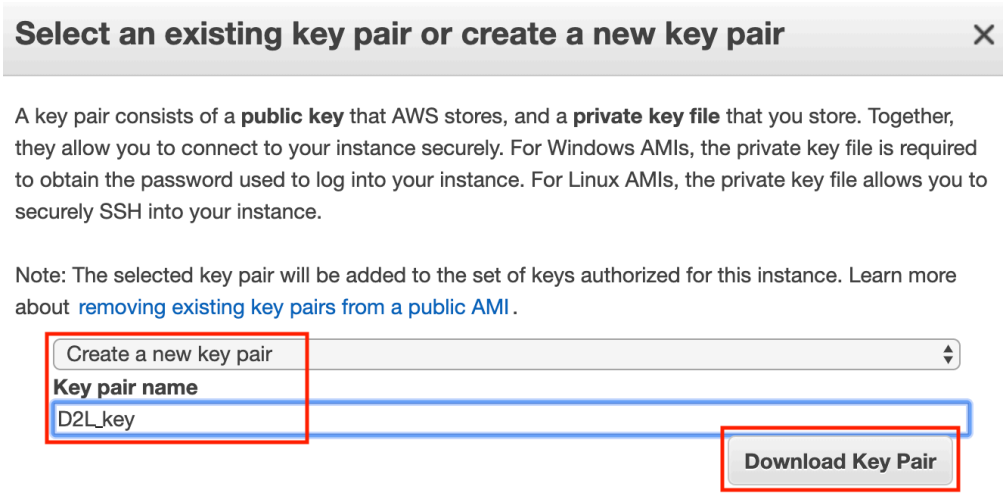


Fig. 19.3.7: Bir anahtar çifti seçme.

Yeni bir tane oluşturduysanız anahtar çiftini indirdiğinizden ve güvenli bir konumda sakladığınızdan emin olun. Bu sunucuya tek yol SSH'dir. Bu örneğin durumunu görüntülemek için Fig. 19.3.8 içinde gösterilen örnek kimliğini tıklatın.

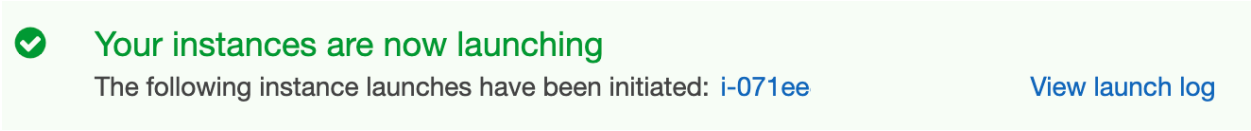


Fig. 19.3.8: Örnek kimliğini tıklama.

Örneğe Bağlanma

Fig. 19.3.9 içinde gösterildiği gibi, örnek durumu yeşile döndükten sonra örneği sağ tıklatın ve örnek erişim yöntemini görüntülemek için `Connect`'i seçin.

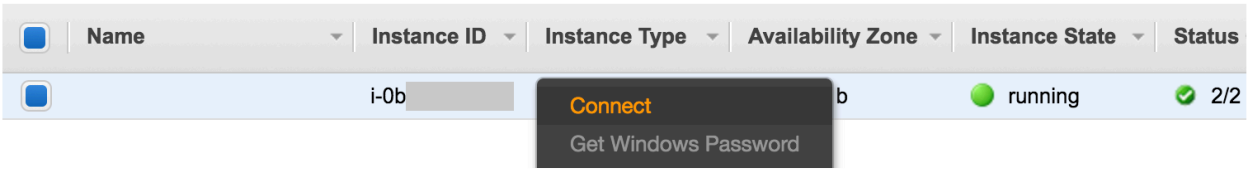


Fig. 19.3.9: Örnek erişimini ve başlatma yöntemini görüntüleme.

Bu yeni bir anahtarsa, SSH'nin çalışması için herkese açık bir şekilde görüntülenmemelidir. `D2L_key.pem`'ü sakladığınız klasöre gidin (örn. İndirilenler klasörü) ve anahtarın genel olarak görüntülenmediğinden emin olun.

```
cd /Downloads  ## D2L_key.pem İndirilenler klasöründe depolanıyorsa
chmod 400 D2L_key.pem
```

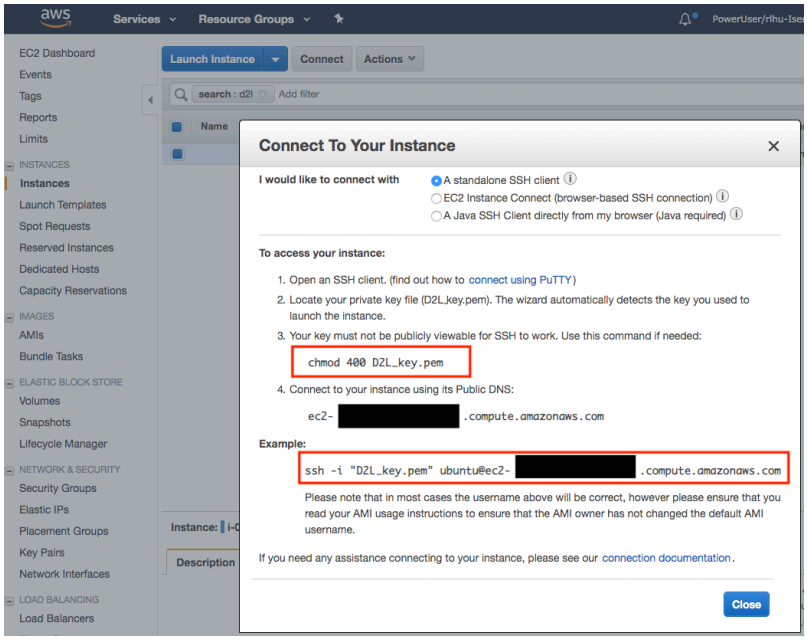


Fig. 19.3.10: Örnek erişimini ve başlatma yöntemini görüntüleme.

Şimdi, Fig. 19.3.10' figürunun alt kırmızı kutusuna ssh komutunu kopyalayın ve komut satırına yapıştırın:

```
ssh -i "D2L_key.pem" ubuntu@ec2-xx-xxx-xxx-xxx.y.compute.amazonaws.com
```

Komut satırı "Are you sure you want to continue connecting (yes/no) - Bağlanmaya devam etmek istediğinizden emin misiniz (evet/hayır)" sorduğunda, "yes - evet" yazın ve örneğe giriş yapmak için Enter tuşuna basın.

Sunucunuz şimdi hazır.

19.3.2 CUDA Kurulumu

CUDA'yı yüklemeden önce, örneği en son sürücülerle güncellediğinizden emin olun.

```
sudo apt-get update && sudo apt-get install -y build-essential git libgfortran3
```

Burada CUDA 10.1'i indiriyoruz. Fig. 19.3.11 içinde gösterildiği gibi CUDA 10.1'in indirme bağlantısını bulmak için NVIDIA'nın resmi deposu[236]nu ziyaret edin.

[236] https://developer.nvidia.com/cuda-downloads

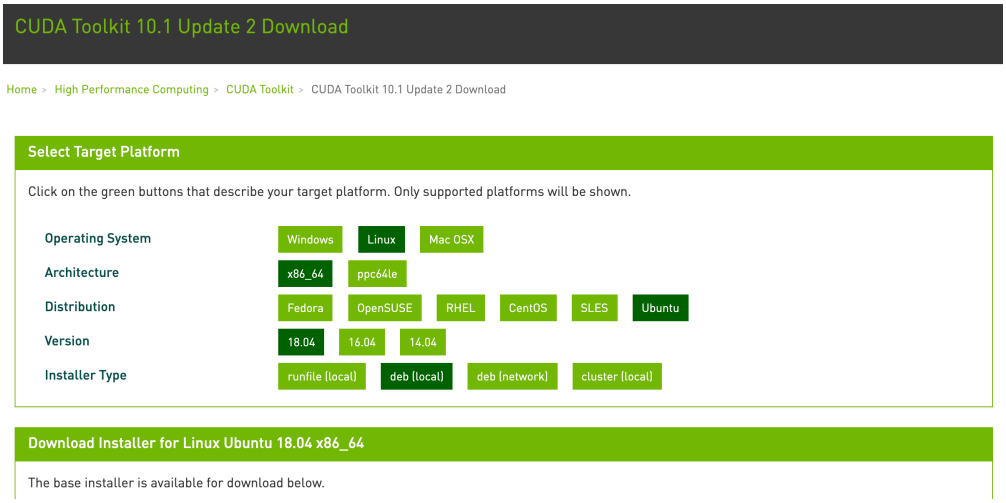


Fig. 19.3.11: CUDA 10.1 indirme adresini bulma.

Talimatları kopyalayın ve CUDA 10.1'i yüklemek için terminale yapıştırın.

```
## CUDA web sitesinden kopyalanan bağlantıyı yapıştırın
wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/x86_64/cuda-
↪ubuntu1804.pin
sudo mv cuda-ubuntu1804.pin /etc/apt/preferences.d/cuda-repository-pin-600
wget http://developer.download.nvidia.com/compute/cuda/10.1/Prod/local_installers/cuda-repo-
↪ubuntu1804-10-1-local-10.1.243-418.87.00_1.0-1_amd64.deb
sudo dpkg -i cuda-repo-ubuntu1804-10-1-local-10.1.243-418.87.00_1.0-1_amd64.deb
sudo apt-key add /var/cuda-repo-10-1-local-10.1.243-418.87.00/7fa2af80.pub
sudo apt-get update
sudo apt-get -y install cuda
```

Programı yükledikten sonra, GPU'ları görüntülemek için aşağıdaki komutu çalıştırın.

```
nvidia-smi
```

Son olarak, diğer kütüphanelerin bulmasına yardımcı olmak için kütüphane yoluna CUDA'yı ekleyin.

```
echo "export LD_LIBRARY_PATH=\${LD_LIBRARY_PATH}:/usr/local/cuda/lib64" >> ~/.bashrc
```

19.3.3 MXNet'i Yükleme ve D2L Not Defterlerini İndirme

Öncelikle, kurulumu basitleştirmek için Linux için Miniconda[237]'yı yüklemeniz gerekir. İndirme bağlantısı ve dosya adı değişikliklere tabidir, bu nedenle lütfen Miniconda web sitesine gidin ve Fig. 19.3.12'te gösterildiği gibi "Copy Link Address - Bağlantı Adresini Kopyala" düğmesine tıklayın.

[237] https://conda.io/en/latest/miniconda.html

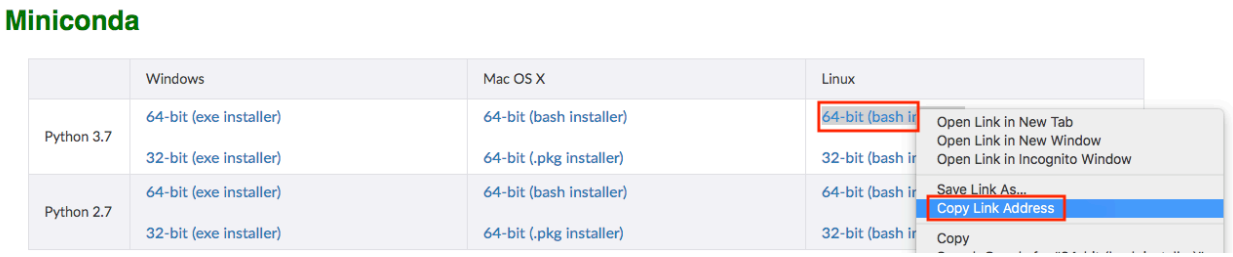


Fig. 19.3.12: Miniconda'yı indirme.

```
# Bağlantı ve dosya adı değişebilir
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
sh Miniconda3-latest-Linux-x86_64.sh -b
```

Miniconda kurulumundan sonra CUDA ve conda'yı etkinleştirmek için aşağıdaki komutu çalıştırın.

```
~/miniconda3/bin/conda init
source ~/.bashrc
```

Ardından, bu kitabın kodunu indirin.

```
sudo apt-get install unzip
mkdir d2l-tr && cd d2l-tr
curl https://tr.d2l.ai/d2l-tr.zip -o d2l-tr.zip
unzip d2l-tr.zip && rm d2l-tr.zip
```

Ardından conda `d2l` ortamını oluşturun ve yüklemeye devam etmek için `y` girin.

```
conda create --name d2l -y
```

`d2l` ortamını oluşturduktan sonra etkinleştirin ve `pip`'i yükleyin.

```
conda activate d2l
conda install python=3.7 pip -y
```

Son olarak, MXNet ve `d2l` paketini yükleyin. `cu101` soneki, bunun CUDA 10.1 varyantı olduğu anlamına gelir. Farklı sürümler için, mesela CUDA 10.0, `cu100`'yı seçmek isteyebirlirsiniz.

```
pip install mxnet-cu101==1.7.0
pip install git+https://github.com/d2l-ai/d2l-en
```

Her şeyin yolunda olup olmadığını hızlı bir şekilde test edebilirsiniz:

```
$ python
>>> from mxnet import np, npx
>>> np.zeros((1024, 1024), ctx=npx.gpu())
```

19.3.4 Jupyter'i Çalıştırma

Jupyter'ı uzaktan çalıştırmak için SSH bağlantı noktası yönlendirme kullanmanız gerekir. Sonuçta, buluttaki sunucunun bir monitörü veya klavyesi yoktur. Bunun için sunucunuza masaüstünüzden (veya dizüstü bilgisayarınızdan) aşağıdaki gibi oturum açın.

```
# Bu komut yerel komut satırında çalıştırılmalıdır
ssh -i "/path/to/key.pem" ubuntu@ec2-xx-xxx-xxx-xxx.y.compute.amazonaws.com -L␣
↪8889:localhost:8888
conda activate d2l
jupyter notebook
```

Fig. 19.3.13, Jupyter Notebook çalıştırdıktan sonra olası çıktıyı gösterir. Son satır 8888 numaralı bağlantı noktası için URL'dir.

```
( d2l ) ubuntu@ip-172-31-2-208:~$ jupyter notebook
[I 06:12:41.588 NotebookApp] Writing notebook server cookie secret to /run/user/1000/jupyter/notebook_cookie_secret
[I 06:12:42.617 NotebookApp] Serving notebooks from local directory: /home/ubuntu
[I 06:12:42.618 NotebookApp] The Jupyter Notebook is running at:
[I 06:12:42.618 NotebookApp] http://localhost:8888/?token=3eb5513
[I 06:12:42.618 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[W 06:12:42.622 NotebookApp] No web browser found: could not locate runnable browser.
[C 06:12:42.622 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///run/user/1000/jupyter/nbserver-21907-open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=3eb5513
```

Fig. 19.3.13: Output after running Jupyter Notebook. The last row is the URL for port 8888.

Bağlantı noktası 8889 numaralı bağlantı noktasına yönlendirmeyi kullandığınız için bağlantı noktası numarasını değiştirmeniz ve yerel tarayıcınızda URL'yi açarken Jupyter tarafından verilen sırrı (secret) kullanmanız gerekir.

19.3.5 Kullanılmayan Örnekleri Kapatma

Bulut hizmetleri kullanım süresine göre faturalandırıldığından, kullanılmayan örnekleri kapatmanız gerekir. Alternatifler olduğunu unutmayın: Bir örneği "durdurmak", yeniden başlatabilmeniz anlamına gelir. Bu, normal sunucunuzun gücünü kapatmaya benzer. Ancak durdurulan örnekler, korunan sabit disk alanı için küçük bir miktar faturalandırılır. "Sonlandır - Terminate", onunla ilişkili tüm verileri siler. Bu diski içerir, bu nedenle yeniden başlatamazsınız. Sadece gelecekte ihtiyacınız olmayacağını biliyorsanız bunu yapın.

Örneği daha birçok örnek için şablon olarak kullanmak istiyorsanız, Fig. 19.3.9 içindeki örnekte sağ tıklayın ve örneğin görüntüsünü oluşturmak için "İmage - Görüntü" → "Create - Oluştur""u seçin. Bu işlem tamamlandıktan sonra örneği sonlandırmak için "Instance State - Örnek Durumu" → "Termiante - Sonlandır""ı seçin. Bu örneği daha sonra kullanmak istediğinizde, kaydedilen görüntüye dayalı bir örnek oluşturmak için bu bölümde açıklanan bir EC2 örneği oluşturma ve çalıştırma adımlarını uygulayabilirsiniz. Tek fark, Fig. 19.3.4 içinde gösterilen "1. Choose AMI - AMI Seç" bölümünde, kayıtlı görüntünüzü seçmek için soldaki "My AMIs - AMI'lerim" seçeneğini kullanmanız gerektiğidir. Oluşturulan örnek, görüntü sabit diskinde depolanan bilgileri saklar. Örneğin, CUDA ve diğer çalışma zamanı ortamlarını yeniden yüklemeniz gerekmez.

19.3.6 Özet

- Kendi bilgisayarınızı satın almak ve oluşturmak zorunda kalmadan isteğe bağlı örnekleri başlatabilir ve durdurabilirsiniz.
- Kullanabilmeniz için uygun GPU sürücülerini yüklemeniz gerekir.

19.3.7 Alıştırmalar

1. Bulut kolaylık sağlar, ancak ucuza gelmez. Fiyatları nasıl düşüreceğinizi görmek için anlık örnekler[238]i nasıl başlatacağınızı öğrenin.
2. Farklı GPU sunucuları ile deney yapın. Ne kadar hızlılar?
3. Çoklu GPU sunucuları ile deney yapın. İşleri ne kadar iyi ölçeklendirebilirsiniz?

Tartışmalar[239]

19.4 Google Colab Kullanma

Bu kitabı Section 19.2 ve Section 19.3 içinde AWS'de nasıl çalıştıracağımızı tanıttık. Başka bir seçenek, bir Google hesabınız varsa ücretsiz GPU sağlayan Google Colab[240]'da bu kitabı çalıştırmaktır.

Colab'da bir bölümü çalıştırmak için, Fig. 19.4.1 içinde olduğu gibi, bu bölümün başlığının sağındaki `Colab` düğmesini tıklamanız yeterlidir.

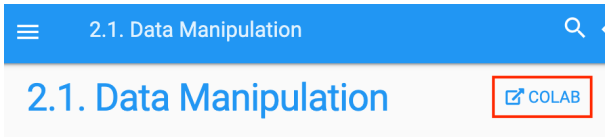


Fig. 19.4.1: Colab'da bir bölüm açma

Bir kod hücresini ilk kez çalıştırdığınızda, Fig. 19.4.2 içinde gösterildiği gibi bir uyarı iletisi alırsınız. Yoksaymak için "HER ŞEKİLDE ÇALIŞTIR - RUN ANYWAY"i tıklayabilirsiniz.

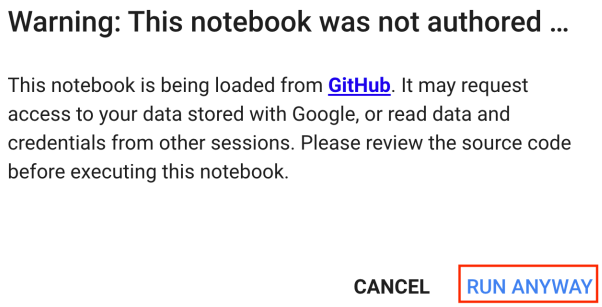


Fig. 19.4.2: Colab'de bir bölüm çalıştırmak için uyarı mesajı.

[238] https://aws.amazon.com/ec2/spot/
[239] https://discuss.d2l.ai/t/423
[240] https://colab.research.google.com/

Ardından, Colab bu not defterini çalıştırmak için sizi bir örneğe bağlar. Özellikle, `d2l.try_gpu()` işlevini çağırırken olduğu gibi GPU gerekiyorsa, Colab'dan bir GPU örneğine otomatik olarak bağlanmasını isteriz.

19.4.1 Özet

- Bu kitabın her bölümünü GPU'larla çalıştırmak için Google Colab'ı kullanabilirsiniz.

19.4.2 Alıştırmalar

1. Google Colab kullanarak bu kitaptaki kodu düzenlemeyi ve çalıştırmayı deneyin.

Tartışmalar[241]

19.5 Sunucuları ve GPU'ları Seçme

Derin öğrenme eğitimi genellikle büyük miktarda hesaplama gerektirir. Şu anda GPU'lar derin öğrenme için en uygun maliyetli donanım hızlandırıcılarıdır. Özellikle, CPU'lar ile karşılaştırıldığında, GPU'lar daha ucuzdur ve büyük ölçekte daha yüksek performans sunar. Ayrıca, tek bir sunucu, üst düzey sunucular için tek sunucu 8 adete kadar çoklu GPU'yu destekleyebilir. Isı, soğutma ve güç gereksinimleri bir ofis binasının destekleyebileceğinin ötesine hızla yükseldiğinden, daha tipik sayılar bir mühendislik iş istasyonu için 4 GPU'ya kadardır. Amazon'un P3[242] ve G4[243] örnekleri gibi daha büyük konuşlandırmalar için bulut bilgi işlem çok daha pratik bir çözümdür.

19.5.1 Sunucuları Seçme

Hesaplamanın çoğu GPU'larda gerçekleştiğinden, genellikle çok sayıda iş parçacığına sahip üst düzey CPU'lar satın almaya gerek yoktur. Bununla birlikte, Python'daki Küresel Yorumlayıcı Kilidi (GIL) nedeniyle bir CPU'nun tek iş parçacıklı performansı, 4-8 GPU'ya sahip olduğumuz durumlarda önemli olabilir. Buna eşit olan her şey, daha az sayıda çekirdeğe sahip ancak daha yüksek bir saat frekansına sahip CPU'ların daha ekonomik bir seçim olabileceğini göstermektedir. Örneğin, 6 çekirdekli 4 GHz ve 8 çekirdekli 3.5 GHz CPU arasında seçim yaparken, birincisi, toplam hızı daha az olsa bile çok daha tercih edilir. Önemli bir husus, GPU'ların çok fazla güç kullanmaları ve böylece çok fazla ısı dağıtmasıdır. Bu, çok iyi soğutma ve GPU'ları kullanmak için yeterince büyük bir kasa gerektirir. Mümkünse aşağıdaki yönergeleri izleyin:

1. **Güç Kaynağı** GPU'lar önemli miktarda güç kullanır. Cihaz başına 350 W'a kadar bütçe koyun (verimli kod çok fazla enerji kullanabileceğinden, tipik talep yerine grafik kartının *en yüksek talebi*ni kontrol edin). Güç kaynağınız talebe bağlı değilse, sisteminizin dengesiz hale geldiğini göreceksiniz.
2. **Kasa Boyutu**. GPU'lar büyüktür ve yardımcı güç bağlayıcıları genellikle fazladan alana ihtiyaç duyar. Ayrıca, büyük kasanın soğuması daha kolaydır.

[241] https://discuss.d2l.ai/t/424
[242] https://aws.amazon.com/ec2/instance-types/p3/
[243] https://aws.amazon.com/blogs/aws/in-the-works-ec2-instances-g4-with-nvidia-t4-gpus/

3. **GPU Soğutma**. Çok sayıda GPU'unuz varsa su soğutmasına yatırım yapmak isteyebilirsiniz. Ayrıca, cihazlar arasında hava girişine izin verecek kadar ince olduklarından, daha az fana sahip olsalar bile *referans tasarımları* hedefleyin. Çok fanlı bir GPU satın alırsanız, birden fazla GPU takarken yeterli hava almada çok kalın olabilir ve termal kısma ile karşılaşırsınız.
4. **PCIe Yuvaları**. Verileri GPU'ya ve GPU'dan taşımak (ve GPU'lar arasında değiş tokuş etmek) çok fazla bant genişliği gerektirir. 16 şeritli PCIe 3.0 yuvalarını öneriyoruz. Birden fazla GPU bağlarsanız, birden fazla GPU aynı anda kullanıldığında 16x bant genişliğinin hala mevcut olduğundan ve ek yuvalar için PCIe 2.0'ın aksine PCIe 3.0'ı aldığınızdan emin olmak için anakart açıklamasını dikkatlice okuyun. Bazı anakartlar, birden fazla GPU takılıyken 8x hatta 4x bant genişliğine düşer. Bu kısmen CPU'nun sunduğu PCIe şeritlerinin sayısından kaynaklanmaktadır.

Kısacası, derin bir öğrenme sunucusu oluşturmak için bazı öneriler şunlardır:

- **Başlangıç**. Düşük güç tüketimine sahip düşük uçlu bir GPU satın alın (derin öğrenme için uygun ucuz oyun GPU'ları 150-200W kullanır). Şanslıysanız mevcut bilgisayarınız bunu destekleyecektir.
- **1 GPU**. 4 çekirdekli düşük uçlu bir CPU yeterli olacak ve çoğu anakart yeterlidir. En az 32 GB DRAM hedefleyin ve yerel veri erişimi için bir SSD'ye yatırım yapın. 600W'lık bir güç kaynağı yeterli olmalıdır. Bir sürü fanı olan bir GPU satın alın.
- **2 GPU**. 4-6 çekirdekli düşük uçlu bir CPU yeterli olacaktır. 64 GB DRAM hedefleyin ve bir SSD'ye yatırım yapın. İki üst düzey GPU için 1000W mertebesine ihtiyacınız olacak. Anakartlar açısından, *iki* PCIe 3.0 x16 yuvasına sahip olduklarından emin olun. Eğer yapabiliyorsanız, ekstra hava için PCIe 3.0 x16 yuvası arasında iki boş alana (60 mm boşluk) sahip bir anakart elde edin. Bu durumda, çok sayıda fanı olan iki GPU satın alın.
- **4 GPU**. Nispeten hızlı tek iş parçacığı hızına (yani yüksek saat frekansı) sahip bir CPU satın aldığınızdan emin olun. Muhtemelen AMD Threadripper gibi daha fazla sayıda PCIe şeridi olan bir CPU'ya ihtiyacınız olacaktır. PCIe hatlarını çoğaltmak için muhtemelen bir PLX'e ihtiyaç duyduklarından, 4 PCIe 3.0 x16 yuvası almak için muhtemelen nispeten pahalı anakartlara ihtiyacınız olacak. Dar referans tasarımlı GPU'lar satın alın ve GPU'lar arasında hava girmesine izin verin. 1600-2000W'lık güç kaynağına ihtiyacınız var ve ofisinizdeki priz bunu desteklemeyebilir. Bu sunucu muhtemelen *yüksek ses ve sıcaklık ile* çalışacak. Masanızın altında istemezsiniz. 128 GB DRAM önerilir. Yerel depolama için bir SSD (1-2 TB NVMe) ve verilerinizi depolamak için RAID yapılandırmasında bir sürü sabit disk edinin.
- **8 GPU**. Birden fazla yedekli güç kaynağına sahip özel bir çoklu GPU sunucu kasası satın almanız gerekir (örneğin, 2+1 için güç kaynağı başına 1600W). Bunun için çift soketli sunucu işlemcileri, 256 GB ECC DRAM, hızlı bir ağ kartı (10 GBE önerilir) ve sunucuların GPU'ların *fiziksel form faktörü*nü destekleyip desteklemediğini kontrol etmeniz gerekecektir. Hava akışı ve kablolama yerleşimi tüketici ve sunucu GPU'ları arasında önemli ölçüde farklılık gösterir (örneğin, RTX 2080 ve Tesla V100). Bu, güç kablosu için yetersiz boşluk veya uygun bir kablo demeti olmaması (yazarlardan birinin acı bir şekilde keşfettiği gibi) nedeniyle tüketici GPU'sunu bir sunucuya kuramayacağınız anlamına gelir.

19.5.2 GPU'ları seçme

Şu anda, AMD ve NVIDIA, adanmış GPU'ların iki ana üreticisidir. NVIDIA derin öğrenme alanına ilk giren oldu ve CUDA aracılığıyla derin öğrenme çerçeveleri için daha iyi destek sağlıyor. Bu nedenle, çoğu alıcı NVIDIA GPU'larını seçer.

NVIDIA, bireysel kullanıcıları (örneğin GTX ve RTX serisi aracılığıyla) ve kurumsal kullanıcıları (Tesla serisi aracılığıyla) hedefleyen iki tür GPU sağlar. İki tür GPU, karşılaştırılabilir işlem gücü sağlar. Ancak, kurumsal kullanıcı GPU'ları genellikle (pasif) zorla soğutma, daha fazla bellek ve ECC (hata düzeltme) bellek kullanır. Bu GPU'lar veri merkezleri için daha uygundur ve genellikle tüketici GPU'larından on kat daha pahalıya mal olurlar.

100'den fazla sunucuya sahip büyük bir şirketseniz NVIDIA Tesla serisini düşünmeli veya alternatif olarak bulutta GPU sunucularını kullanmalısınız. Bir laboratuvar veya 10+ sunucusu olan küçük ve orta ölçekli şirketler için NVIDIA RTX serisi muhtemelen en uygun maliyetlidir. 4-8 GPU'yu verimli bir şekilde tutan Supermicro veya Asus kasaları ile önceden yapılandırılmış sunucular satın alabilirsiniz.

GPU satıcıları, 2017'de piyasaya sürülen GTX 1000 (Pascal) serisi ve 2019'da piyasaya sürülen RTX 2000 (Turing) serisi gibi 1-2 yılda bir yeni nesil piyasaya sürüyorlar. Her seri, farklı performans seviyeleri sağlayan birkaç farklı model sunar. GPU performansı öncelikle aşağıdaki üç parametrenin birleşimidir:

1. **Bilgi işlem gücü**. Genelde 32 bitlik kayan virgüllü sayı işlem gücü arıyoruz. 16 bitlik kayan virgüllü sayı eğitimide (FP16) genel kullanıma giriyor. Yalnızca tahmin ile ilgileniyorsanız, 8 bit tamsayı da kullanabilirsiniz. En yeni nesil Turing GPU'ları 4 bit hızlandırma sunar. Ne yazık ki şu anda düşük kesinlikli ağları eğitmek için kullanılan algoritmalar henüz yaygın değildir.
2. **Hafıza boyutu**. Modelleriniz büyüdükçe veya eğitim sırasında kullanılan toplu işler büyüdükçe, daha fazla GPU belleğine ihtiyacınız olacak. HBM2 (Yüksek Bant Genişlikli Bellek) ve GDDR6 (Grafik DDR) bellek olup olmadığını kontrol edin. HBM2 daha hızlı ama çok daha pahalıdır.
3. **Bellek bant genişliği**. Yalnızca yeterli bellek bant genişliğine sahip olduğunuzda işlem gücünüzden en iyi şekilde yararlanabilirsiniz. GDDR6 kullanıyorsanız geniş bellek veri yollarına bakın.

Çoğu kullanıcı için, işlem gücüne bakmak yeterlidir. Birçok GPU'nun farklı ivme türleri sunduğunu unutmayın. Örneğin, NVIDIA'nın TensorCores'u operatörlerin bir alt kümesini 5 kat hızlandırır. Kütüphanenizin bunu desteklediğinden emin olun. GPU belleği 4 GB'den az olmamalıdır (8 GB çok daha iyidir). GPU'yu bir GUI görüntülemek için de kullanmaktan kaçınmaya çalışın (bunun yerine yerleşik grafikleri kullanın). Bunu önleyemiyorsanız, güvenlik için fazladan 2 GB RAM ekleyin.

Fig. 19.5.1, çeşitli GTX 900, GTX 1000 ve RTX 2000 serisi modellerinin 32 bit kayan virgüllü sayı hesaplama gücünü ve fiyatını karşılaştırır. Fiyatlar Wikipedia'de bulunan önerilen fiyatlardır.

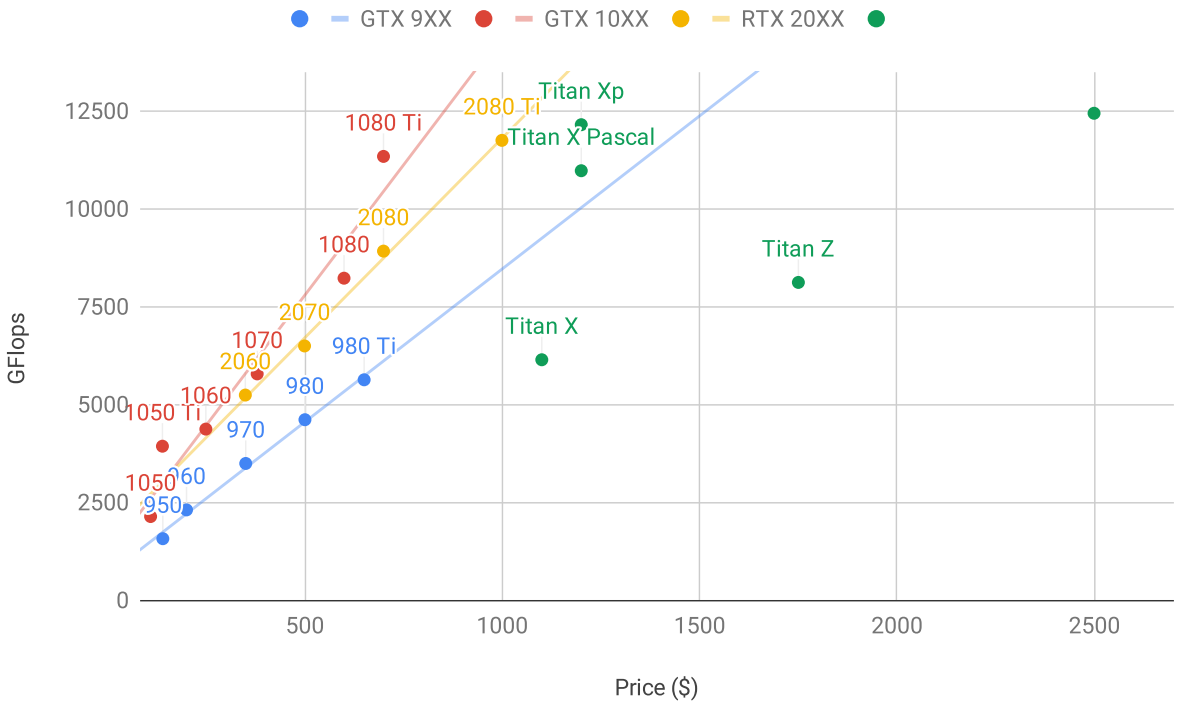


Fig. 19.5.1: Kayan virgüllü sayı hesaplama gücü ve fiyat karşılaştırması.

Bir dizi şey görebiliriz:

1. Her seride fiyat ve performans kabaca orantılıdır. Titan modelleri, daha büyük miktarlarda GPU belleğinin yararına önemli bir getiri sağlar. Bununla birlikte, daha yeni modeller 980 Ti ve 1080 Ti'yi karşılaştırarak görülebileceği gibi daha iyi maliyet etkinliği sunar. RTX 2000 serisi için fiyat pek iyileşmiyor gibi görünüyor. Ancak bunun nedeni, çok daha üstün düşük kesinlikli performans (FP16, INT8 ve INT4) sunmalarıdır.
2. GTX 1000 serisinin performans-maliyet oranı 900 serisinden yaklaşık iki kat daha fazladır.
3. RTX 2000 serisi için fiyat fiyatın bir *afin* fonksiyonudur.

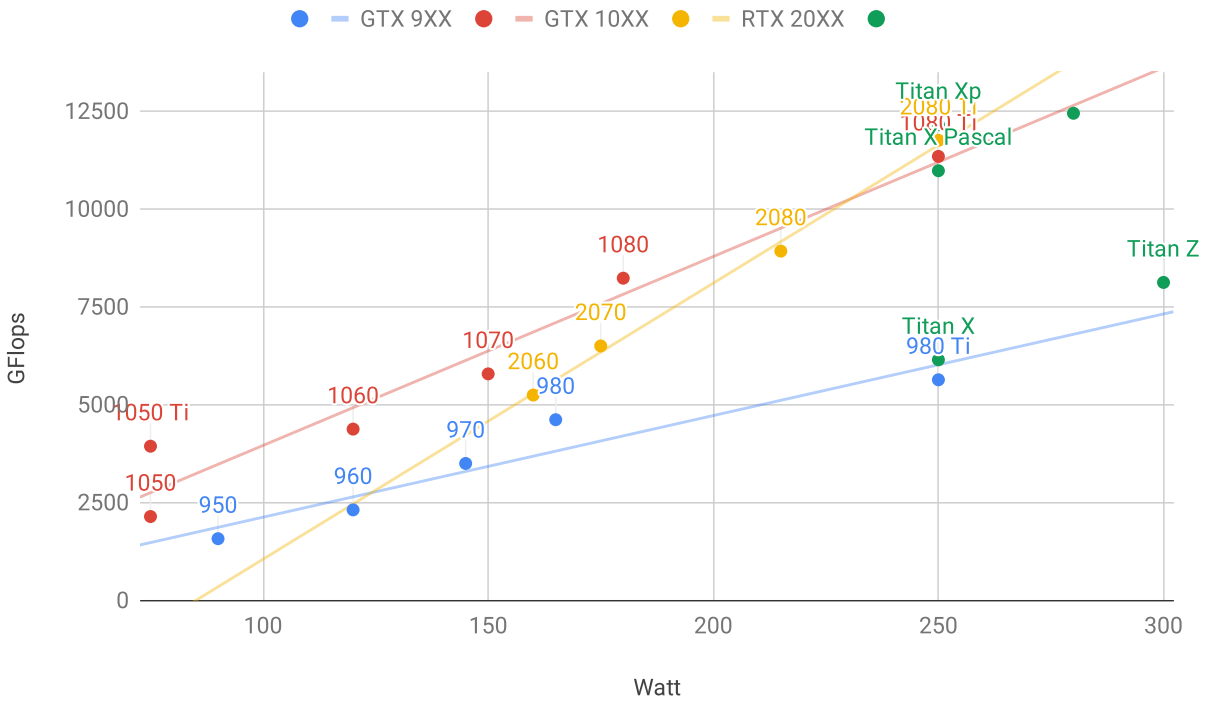


Fig. 19.5.2: Kayan virgüllü sayı hesaplama gücü ve enerji tüketimi.

Fig. 19.5.2, enerji tüketiminin hesaplama miktarıyla çoğunlukla doğrusal olarak nasıl ölçeklendiğini gösterir. İkincisi, sonraki nesiller daha verimli. Bu, RTX 2000 serisine karşılık gelen grafikle çelişiyor gibi görünüyor. Ancak bu, orantısız derecede fazla enerji çeken TensorCores'un bir sonucudur.

19.5.3 Özet

- Sunucu oluştururken gücüne, PCIe veri yolu şeritlerine, CPU tek iş parçacığı hızına ve soğutmaya dikkat edin.
- Mümkünse en son GPU neslini satın almalısınız.
- Büyük konuşlandırmalar için bulutu kullanın.
- Yüksek yoğunluklu sunucular tüm GPU'larla uyumlu olmayabilir. Satın almadan önce mekanik özellikleri ve soğutma özelliklerini kontrol edin.
- Yüksek verimlilik için FP16 veya daha düşük kesinlik kullanın.

Tartışmalar[244]

[244] https://discuss.d2l.ai/t/425

19.6 Bu Kitaba Katkıda Bulunmak

Okuyucular[245] tarafından yapılan katkılar bu kitabı geliştirmemize yardımcı olur. Bir yazım hatası, eski bir bağlantı, bir alıntıyı kaçırdığımızı düşündüğünüz, kodun zarif görünmediği veya bir açıklamanın belirsiz olduğu bir şey bulursanız, lütfen katkıda bulunun ve okuyucularımıza yardım etmemize yardımcı olun. Normal kitaplarda baskı işlemleri arasındaki (ve dolayısıyla yazım hatası düzeltmeleri arasındaki) gecikme yıllar olarak ölçülebilirken, bu kitaba bir iyileştirme eklemek genellikle saatler veya günler alır. Tüm bunlar sürüm kontrolü ve sürekli birleştirme testi nedeniyle mümkündür. Bunu yapmak için GitHub deposuna bir çekme isteği (pull request)[246] göndermeniz gerekir. Çekme isteğiniz yazar tarafından kod deposuna birleştirildiğinde, bir katkıda bulunan olursunuz.

19.6.1 Küçük Metin Değişiklikleri

En yaygın katkılar bir cümleyi düzenlemek veya yazım hatalarını düzeltmektir. Kaynak dosyayı github deposu[247]nda bulmanızı ve dosyayı doğrudan düzenlemenizi öneririz. Örneğin, kaynak dosyayı bulmak için Dosya bul - Find file[248] düğmesi (Fig. 19.6.1) aracılığıyla dosyayı arayabilirsiniz, ki o da bir markdown dosyasıdır. Ardından, markdown dosyasında değişikliklerinizi yapmak için sağ üst köşedeki "Bu dosyayı düzenle - Edit this file" düğmesini tıklayın.

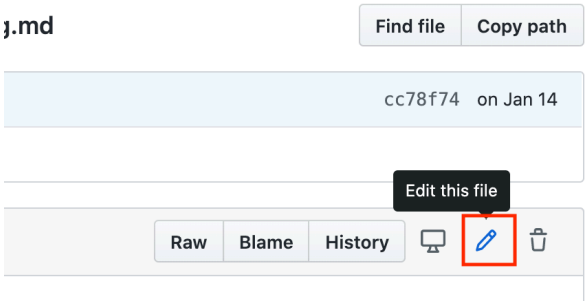


Fig. 19.6.1: Dosyayı Github'da düzenleme

İşiniz bittikten sonra, sayfa altındaki "Dosya değişikliği önerin - Propose file change" panelinde değişiklik açıklamalarınızı doldurun ve ardından "Dosya değişikliği önerin - Propose file change" düğmesini tıklayın. Değişikliklerinizi incelemek için sizi yeni bir sayfaya yönlendirecektir (Fig. 19.6.7). Her şey yolundaysa, "Çekme isteği oluştur - Create pull request" düğmesine tıklayarak bir çekme isteği gönderebilirsiniz.

[245] https://github.com/d2l-ai/d2l-tr/graphs/contributors
[246] https://github.com/d2l-ai/d2l-tr/pulls
[247] https://github.com/d2l-ai/d2l-en
[248] https://github.com/d2l-ai/d2l-tr/find/master

19.6.2 Büyük Bir Değişiklik Önerme

Metnin veya kodun büyük bir bölümünü güncellemeyi planlıyorsanız, bu kitabın kullandığı format hakkında biraz daha fazla bilgi sahibi olmanız gerekir. Kaynak dosya denklemlere, imgelere, bölümlere ve referanslara atıfta bulunmak gibi bir dizi uzantı içeren d2lbook[249] paketi aracılığıyla markdown format[250]ını temel alır. Bu dosyaları açmak ve değişikliklerinizi yapmak için herhangi bir Markdown düzenleyicisini kullanabilirsiniz.

Kodu değiştirmek isterseniz, Section 19.1 içinde açıklandığı gibi bu Markdown dosyalarını açmada Jupyter kullanmanızı öneririz. Böylece değişikliklerinizi çalıştırabilir ve test edebilirsiniz. Lütfen değişikliklerinizi göndermeden önce tüm çıktıları temizlemeyi unutmayın, CI sistemimiz çıktı üretmek için güncellediğiniz bölümleri yürütecektir.

Bazı bölümler birden çok çerçeve uygulamasını destekleyebilir, belirli bir çerçeveyi etkinleştirmek için `d2lbook`'ü kullanabilirsiniz, böylece diğer çerçeve uygulamaları Markdown kod blokları haline gelir ve Jupyter'de "Run All - Tümünü Çalıştır" yaptığınızda yürütülmez. Başka bir deyişle, önce `d2lbook`'ü çalıştırarak yükleyin

```
pip install git+https://github.com/d2l-ai/d2l-book
```

Daha sonra `d2l-tr`'in kök dizininde, aşağıdaki komutlardan birini çalıştırarak belirli bir uygulamayı etkinleştirebilirsiniz:

```
d2lbook activate mxnet chapter_multilayer-perceptrons/mlp-scratch.md
d2lbook activate pytorch chapter_multilayer-perceptrons/mlp-scratch.md
d2lbook activate tensorflow chapter_multilayer-perceptrons/mlp-scratch.md
```

Değişikliklerinizi göndermeden önce lütfen tüm kod bloğu çıktılarını temizleyin ve hepsini etkinleştirin:

```
d2lbook activate all chapter_multilayer-perceptrons/mlp-scratch.md
```

Varsayılan MXNet olan uygulama için değil de, olmayanlar için yeni bir kod bloğu eklerseniz, bu bloğun başlangıç satırını işaretlemek için lütfen `#@tab` kullanın. Örneğin, bir PyTorch kod bloğu için `#@tab pytorch`, bir TensorFlow kod bloğu için `#@tab tensorflow` veya tüm uygulamalarda paylaşılan bir kod bloğu için `#@tab all` kullanın. Daha fazla bilgi için d2lbook[251] adresine başvurabilirsiniz.

19.6.3 Yeni Bölüm veya Yeni Çerçeve Uygulaması Ekleme

Eğer, mesela pekiştirmeli öğrenme gibi, yeni bir bölüm oluşturmak veya, TensorFlow gibi, yeni çerçevelerin uygulamalarını eklemek istiyorsanız, lütfen önce e-posta göndererek veya github meseleleri (github issues)[252]'ni kullanarak yazarlarla iletişime geçin.

[249] http://book.d2l.ai/user/markdown.html
[250] https://daringfireball.net/projects/markdown/syntax
[251] http://book.d2l.ai/user/code_tabs.html
[252] https://github.com/d2l-ai/d2l-tr/issues

19.6.4 Büyük Değişiklik Gönderme

Büyük bir değişiklik yapmak için standart `git` sürecini kullanmanızı öneririz. Özetle, süreç Fig. 19.6.2 içinde açıklandığı gibi çalışır.

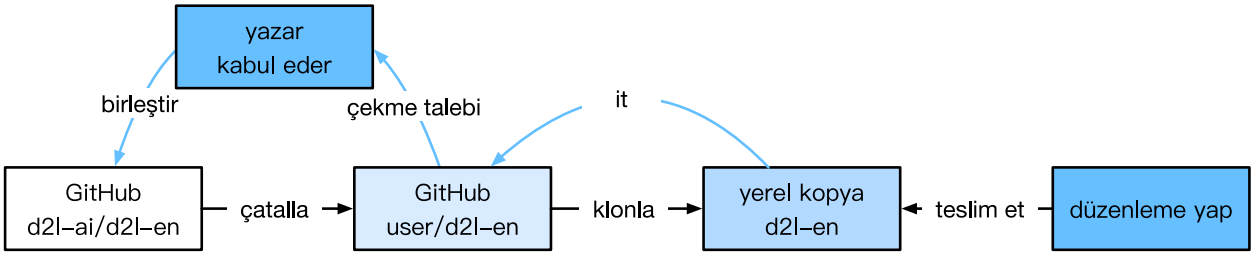


Fig. 19.6.2: Kitaba katkıda bulunma.

Sizi basamaklardan ayrıntılı olarak geçireceğiz. Git'i zaten biliyorsanız bu bölümü atlayabilirsiniz. Somutluk için katkıda bulunanın kullanıcı adının "astonzhang" olduğunu varsayıyoruz.

Git Yükleme

Git açık kaynak kitabı, Git'in nasıl kurulacağını[253] açıklar. Bu genellikle Ubuntu Linux'ta `apt install git` üzerinden, macOS'e Xcode geliştirici araçlarını yükleyerek veya GitHub'ın (desktop client) masaüstü istemci[254]sini kullanarak çalışır. GitHub hesabınız yoksa, almak için kaydolmanız gerekir.

GitHub'da Oturum Açma

Kitabın kod deposunun adres[255]ini tarayıcınıza girin. Bu kitabın deposunun bir kopyasını yapmak için Fig. 19.6.3 figürünün sağ üst tarafındaki kırmızı kutudaki `Fork (Çatalla)` düğmesine tıklayın. Bu artık *sizin kopya*nız ve istediğiniz şekilde değiştirebilirsiniz.

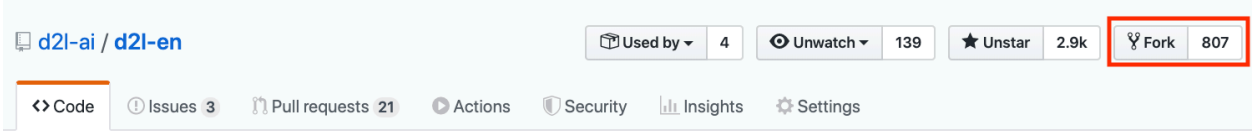


Fig. 19.6.3: Kod deposu sayfası.

Şimdi, bu kitabın kod deposu, Fig. 19.6.4 ekran görüntüsünün sol üst tarafında gösterilen `astonzhang/d2l-en` gibi kullanıcı adınıza çatallanacaktır (yani kopyalanacaktır).

[253] https://git-scm.com/book/en/v2
[254] https://desktop.github.com
[255] https://github.com/d2l-ai/d2l-tr/

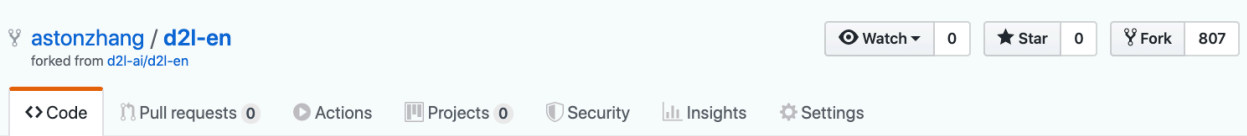


Fig. 19.6.4: Kod deposunu çatallama.

Depoyu Klonlama

Depoyu klonlamak için (yani yerel bir kopya yapmak için) depo adresini almamız gerekir. Fig. 19.6.5 içindeki yeşil düğme bunu gösterir. Bu çatalı daha uzun süre tutmaya karar verirseniz, yerel kopyanızın ana depoyla güncel olduğundan emin olun. Şimdilik başlamak için *Kurulum* (page 9) içindeki talimatları izleyin. Temel fark, şu anda deponun *kendi çatalınızı* indiriyor olmanızdır.

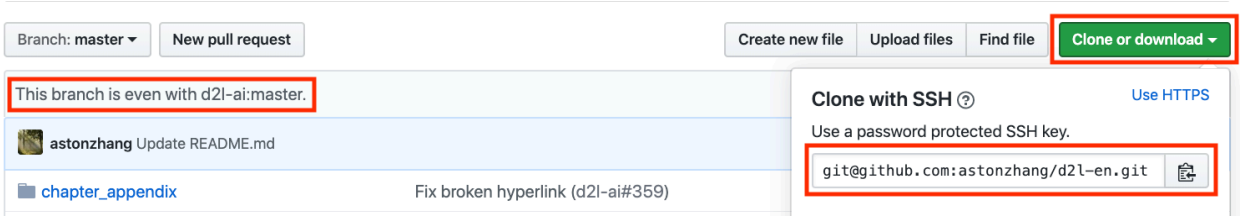


Fig. 19.6.5: Git klonu.

```
# your_github_username yerine kendi GitHub kullanıcı adınızı yazınız.
git clone https://github.com/your_github_username/d2l-tr.git
```

Kitabı Düzenleme ve İtme

Şimdi kitabı düzenleme zamanı. Section 19.1 içindeki talimatları izleyerek not defterlerini Jupyter'da düzenlemek en iyisidir. Değişiklikleri yapın ve bunların iyi olup olmadığını kontrol edin. `~/d2l-tr/chapter_appendix_tools/how-to-contribute.md` dosyasında bir yazım hatası değiştirdiğimizi varsayalım. Daha sonra hangi dosyaları değiştirdiğinizi kontrol edebilirsiniz:

Bu noktada Git, `chapter_appendix_tools/how-to-contribute.md` dosyasının değiştirildiğini soracaktır.

```
mylaptop:d2l-en me$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   chapter_appendix_tools/how-to-contribute.md
```

İstediğiniz şeyin bu olduğunu onayladıktan sonra aşağıdaki komutu çalıştırın:

```
git add chapter_appendix_tools/how-to-contribute.md
git commit -m 'git belgelerindeki yazım hatasını düzelt'
git push
```

Değiştirilen kod daha sonra arşivdeki kişisel çatalınızda olacaktır. Değişikliğinizin eklenmesini talep etmek için, kitabın resmi deposu için bir çekme isteği oluşturmanız gerekir.

Çekme Talebi

Fig. 19.6.6'te gösterildiği gibi, GitHub'daki depo çatalınıza gidin ve "Yeni çekme isteği" ni seçin. Bu, düzenlemeleriniz ile kitabın ana deposunda mevcut olan değişiklikleri gösteren bir ekran açılacaktır.

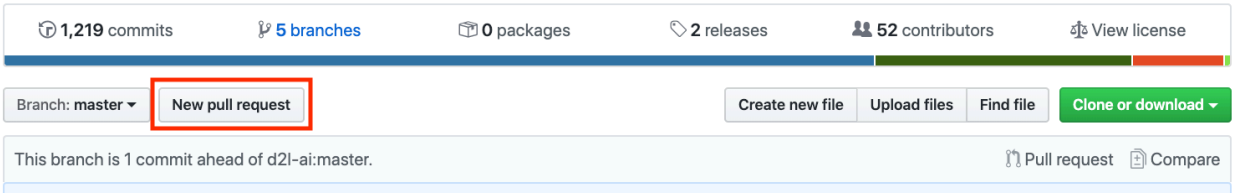


Fig. 19.6.6: Çekme talebi.

Çekme Talebi Gönderme

Son olarak, Fig. 19.6.7 içinde gösterildiği gibi düğmeye tıklayarak bir çekme talebi gönderin. Çekme talebinde yaptığınız değişiklikleri açıkladığınızdan emin olun. Bu, yazarların gözden geçirmesini ve kitapla birleştirmesini kolaylaştıracaktır. Değişikliklere bağlı olarak, bu hemen kabul edilebilir, reddedilebilir veya daha büyük olasılıkla değişiklikler hakkında bazı geri bildirimler alırsınız. Onları bir kez dahil ettikten sonra, herşey yolundadır.

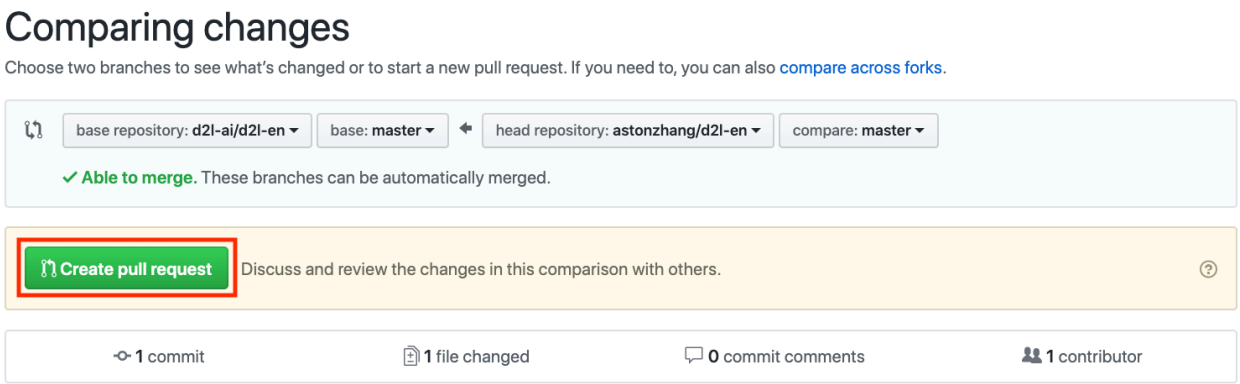


Fig. 19.6.7: Çekme Talebi Oluşturma.

Çekme talebiniz ana depodaki talepler listesinde görünür. Hızlı bir şekilde işlemek için her türlü çabayı göstereceğiz.

19.6.5 Özet

- Bu kitaba katkıda bulunmak için GitHub'ı kullanabilirsiniz.
- Küçük değişiklikler için doğrudan GitHub'daki dosyayı düzenleyebilirsiniz.
- Büyük bir değişiklik için lütfen depoyu çatallayın, yerel olarak düzenleyin ve yalnızca hazır olduğunuzda katkıda bulunun.
- Çekme talepleri, katkıların nasıl gruplandırıldığıdır. Anlamayı ve birleştirmeyi zorlaştırdığı için çok büyük çekme talepleri göndermemeye çalışın. Birkaç tane daha küçük olarak gönderseniz iyi olur.

19.6.6 Alıştırmalar

1. `d2l-tr` deposunu yıldızlayın ve çatallayın.
2. Geliştirilmesi gereken bir kod bulun ve çekme talebi gönderin.
3. Kaçırdığımız bir referansı bulun ve çekme isteği gönderin.
4. Yeni bir dal kullanarak çekme talebi oluşturmak genellikle daha iyi bir uygulamadır. Git dallandırma (git branching)[256] ile nasıl yapılacağını öğrenin.

Tartışmalar[257]

19.7 `d2l` API Belgesi

`d2l` paketinin aşağıdaki üyelerinin tanımlandığı ve açıklandığı bölümlerin uygulamaları (source file) kaynak dosya[258]sında bulunabilir.

`class d2l.torch.Accumulator(`*n*`)`

Bases: `object`

n degisken uzerinden toplamlari biriktirmek için

`add(`**args*`)`

`reset()`

`class d2l.torch.AddNorm(`*normalized_shape, dropout, **kwargs*`)`

Bases: `Module`

Residual connection followed by layer normalization.

Defined in Section 10.7

`forward(`*X, Y*`)`

Defines the computation performed at every call.

Should be overridden by all subclasses.

[256] https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell
[257] https://discuss.d2l.ai/t/426
[258] https://github.com/d2l-ai/d2l-tr/tree/master/d2l

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`training: bool`

`class d2l.torch.AdditiveAttention(`*key_size, query_size, num_hiddens, dropout, **kwargs*`)`

Bases: `Module`

Additive attention.

Defined in Section 10.3

`forward(`*queries, keys, values, valid_lens*`)`

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`training: bool`

`class d2l.torch.Animator(`*xlabel=None, ylabel=None, legend=None, xlim=None, ylim=None, xscale='linear', yscale='linear', fmts=('-', 'm--', 'g-.', 'r:'), nrows=1, ncols=1, figsize=(3.5, 2.5)*`)`

Bases: `object`

Animasyonda veri cizdirme

`add(`*x, y*`)`

`class d2l.torch.AttentionDecoder(`***kwargs*`)`

Bases: `Decoder` (page 938)

The base attention-based decoder interface.

Defined in Section 10.4

`property attention_weights`

`training: bool`

`class d2l.torch.BERTEncoder(`*vocab_size, num_hiddens, norm_shape, ffn_num_input, ffn_num_hiddens, num_heads, num_layers, dropout, max_len=1000, key_size=768, query_size=768, value_size=768, **kwargs*`)`

Bases: `Module`

BERT encoder.

Defined in Section 14.7.4

forward(*tokens*, *segments*, *valid_lens*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.BERTModel(*vocab_size*, *num_hiddens*, *norm_shape*, *ffn_num_input*, *ffn_num_hiddens*, *num_heads*, *num_layers*, *dropout*, *max_len=1000*, *key_size=768*, *query_size=768*, *value_size=768*, *hid_in_features=768*, *mlm_in_features=768*, *nsp_in_features=768*)

Bases: `Module`

The BERT model.

Defined in Section 14.7.5

forward(*tokens*, *segments*, *valid_lens=None*, *pred_positions=None*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.BananasDataset(*is_train*)

Bases: `Dataset`

A customized dataset to load the banana detection dataset.

Defined in Section 13.6

class d2l.torch.Benchmark(*description='Done'*)

Bases: `object`

For measuring running time.

class d2l.torch.Decoder(***kwargs*)

Bases: `Module`

The base decoder interface for the encoder-decoder architecture.

Defined in Section 9.6

forward(*X*, *state*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`init_state(enc_outputs, *args)`

`training: bool`

`class d2l.torch.DotProductAttention(dropout, **kwargs)`

Bases: `Module`

Scaled dot product attention.

Defined in Section 10.3.2

`forward(queries, keys, values, valid_lens=None)`

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`training: bool`

`class d2l.torch.Encoder(**kwargs)`

Bases: `Module`

The base encoder interface for the encoder-decoder architecture.

`forward(X, *args)`

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`training: bool`

`class d2l.torch.EncoderBlock(key_size, query_size, value_size, num_hiddens, norm_shape, ffn_num_input, ffn_num_hiddens, num_heads, dropout, use_bias=False, **kwargs)`

Bases: `Module`

Transformer encoder block.

Defined in Section 10.7

`forward(X, valid_lens)`

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`training: bool`

`class d2l.torch.EncoderDecoder(encoder, decoder, **kwargs)`

Bases: `Module`

The base class for the encoder-decoder architecture.

Defined in Section 9.6

`forward(enc_X, dec_X, *args)`

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`training: bool`

`class d2l.torch.MaskLM(vocab_size, num_hiddens, num_inputs=768, **kwargs)`

Bases: `Module`

The masked language model task of BERT.

Defined in Section 14.7.4

`forward(X, pred_positions)`

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`training: bool`

`class d2l.torch.MaskedSoftmaxCELoss(weight: Optional[Tensor] = None, size_average=None, ignore_index: int = -100, reduce=None, reduction: str = 'mean', label_smoothing: float = 0.0)`

Bases: `CrossEntropyLoss`

The softmax cross-entropy loss with masks.

Defined in Section 9.7.2

`forward`(*pred, label, valid_len*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`ignore_index: int`

`label_smoothing: float`

`class d2l.torch.MultiHeadAttention`(*key_size, query_size, value_size, num_hiddens, num_heads, dropout, bias=False, **kwargs*)

Bases: `Module`

Multi-head attention.

Defined in Section 10.5

`forward`(*queries, keys, values, valid_lens*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`training: bool`

`class d2l.torch.NextSentencePred`(*num_inputs, **kwargs*)

Bases: `Module`

The next sentence prediction task of BERT.

Defined in Section 14.7.5

`forward`(X)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`training: bool`

class d2l.torch.PositionWiseFFN(*ffn_num_input, ffn_num_hiddens, ffn_num_outputs, **kwargs*)

Bases: Module

Positionwise feed-forward network.

Defined in Section 10.7

forward(*X*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.PositionalEncoding(*num_hiddens, dropout, max_len=1000*)

Bases: Module

Positional encoding.

Defined in Section 10.6

forward(*X*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.RNNModel(*rnn_layer, vocab_size, **kwargs*)

Bases: Module

The RNN model.

Defined in Section 8.6

begin_state(*device, batch_size=1*)

forward(*inputs, state*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.RNNModelScratch(*vocab_size, num_hiddens, device, get_params, init_state, forward_fn*)

Bases: object

A RNN Model implemented from scratch.

begin_state(*batch_size, device*)

class d2l.torch.RandomGenerator(*sampling_weights*)

Bases: object

Randomly draw among {1, ..., n} according to n sampling weights.

draw()

class d2l.torch.Residual(*input_channels, num_channels, use_1x1conv=False, strides=1*)

Bases: Module

The Residual block of ResNet.

forward(*X*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.SNLIDataset(*dataset, num_steps, vocab=None*)

Bases: Dataset

A customized dataset to load the SNLI dataset.

Defined in Section 15.4

class d2l.torch.Seq2SeqEncoder(*vocab_size, embed_size, num_hiddens, num_layers, dropout=0, **kwargs*)

Bases: Encoder (page 939)

The RNN encoder for sequence to sequence learning.

Defined in Section 9.7

forward(*X, *args*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.SeqDataLoader(*batch_size*, *num_steps*, *use_random_iter*, *max_tokens*)

Bases: object

An iterator to load sequence data.

class d2l.torch.Timer

Bases: object

Record multiple running times.

avg()

Return the average time.

cumsum()

Return the accumulated time.

start()

Start the timer.

stop()

Stop the timer and record the time in a list.

sum()

Return the sum of time.

class d2l.torch.TokenEmbedding(*embedding_name*)

Bases: object

Token Embedding.

class d2l.torch.TransformerEncoder(*vocab_size*, *key_size*, *query_size*, *value_size*, *num_hiddens*, *norm_shape*, *ffn_num_input*, *ffn_num_hiddens*, *num_heads*, *num_layers*, *dropout*, *use_bias=False*, ***kwargs*)

Bases: Encoder (page 939)

Transformer encoder.

Defined in Section 10.7

forward(*X*, *valid_lens*, **args*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.VOCSegDataset(*is_train*, *crop_size*, *voc_dir*)

Bases: Dataset

A customized dataset to load the VOC dataset.

Defined in Section 13.9

`filter`(*imgs*)

`normalize_image`(*img*)

`class d2l.torch.Vocab`(*tokens=None, min_freq=0, reserved_tokens=None*)

Bases: `object`

Vocabulary for text.

`to_tokens`(*indices*)

`property token_freqs`

`property unk`

`d2l.torch.accuracy`(*y_hat, y*)

Dogru tahminlerin sayisini hesaplayin.

Defined in Section 3.6

`d2l.torch.annotate`(*text, xy, xytext*)

`d2l.torch.argmax`(*x, *args, **kwargs*)

`d2l.torch.assign_anchor_to_bbox`(*ground_truth, anchors, device, iou_threshold=0.5*)

Assign closest ground-truth bounding boxes to anchor boxes.

Defined in Section 13.4

`d2l.torch.astype`(*x, *args, **kwargs*)

`d2l.torch.batchify`(*data*)

Return a minibatch of examples for skip-gram with negative sampling.

Defined in Section 14.3

`d2l.torch.bbox_to_rect`(*bbox, color*)

Convert bounding box to matplotlib format.

Defined in Section 13.3

`d2l.torch.bleu`(*pred_seq, label_seq, k*)

Compute the BLEU.

Defined in Section 9.7.4

`d2l.torch.box_center_to_corner`(*boxes*)

Convert from (center, width, height) to (upper-left, lower-right).

Defined in Section 13.3

`d2l.torch.box_corner_to_center`(*boxes*)

Convert from (upper-left, lower-right) to (center, width, height).

Defined in Section 13.3

`d2l.torch.box_iou(`*boxes1*, *boxes2*`)`

Compute pairwise IoU across two lists of anchor or bounding boxes.

Defined in Section 13.4

`d2l.torch.build_array_nmt(`*lines*, *vocab*, *num_steps*`)`

Transform text sequences of machine translation into minibatches.

Defined in Section 9.5.4

`d2l.torch.copyfile(`*filename*, *target_dir*`)`

Copy a file into a target directory.

Defined in Section 13.13

`d2l.torch.corr2d(`*X*, *K*`)`

Compute 2D cross-correlation.

Defined in Section 6.2

`d2l.torch.count_corpus(`*tokens*`)`

Count token frequencies.

Defined in Section 8.2

`d2l.torch.download(`*name*, *cache_dir='../data'*`)`

Download a file inserted into DATA_HUB, return the local filename.

Defined in Section 4.10

`d2l.torch.download_all()`

Download all files in the DATA_HUB.

Defined in Section 4.10

`d2l.torch.download_extract(`*name*, *folder=None*`)`

Download and extract a zip/tar file.

Defined in Section 4.10

`d2l.torch.evaluate_accuracy(`*net*, *data_iter*`)`

Bir veri kumesindeki bir modelin dogrulugunu hesaplayin.

Defined in Section 3.6

`d2l.torch.evaluate_accuracy_gpu(`*net*, *data_iter*, *device=None*`)`

Compute the accuracy for a model on a dataset using a GPU.

Defined in Section 6.6

`d2l.torch.evaluate_loss(`*net*, *data_iter*, *loss*`)`

Evaluate the loss of a model on the given dataset.

Defined in Section 4.4

`d2l.torch.get_centers_and_contexts(`*corpus*, *max_window_size*`)`

Return center words and context words in skip-gram.

Defined in Section 14.3

d2l.torch.get_data_ch11(*batch_size=10, n=1500*)

Defined in Section 11.5.2

d2l.torch.get_dataloader_workers()

Verileri okumak için 4 işlem kullanın.

Defined in Section 3.5

d2l.torch.get_fashion_mnist_labels(*labels*)

Fashion-MNIST veri kümesi için metin etiketleri döndürün.

Defined in Section 3.5

d2l.torch.get_negatives(*all_contexts, vocab, counter, K*)

Return noise words in negative sampling.

Defined in Section 14.3

d2l.torch.get_tokens_and_segments(*tokens_a, tokens_b=None*)

Get tokens of the BERT input sequence and their segment IDs.

Defined in Section 14.7

d2l.torch.grad_clipping(*net, theta*)

Clip the gradient.

Defined in Section 8.5

d2l.torch.linreg(*X, w, b*)

Dogrusal regresyon modeli.

Defined in Section 3.2

d2l.torch.load_array(*data_arrays, batch_size, is_train=True*)

Bir PyTorch veri yineleyici olusturun.

Defined in Section 3.3

d2l.torch.load_corpus_time_machine(*max_tokens=-1*)

Return token indices and the vocabulary of the time machine dataset.

Defined in Section 8.2

d2l.torch.load_data_bananas(*batch_size*)

Load the banana detection dataset.

Defined in Section 13.6

d2l.torch.load_data_fashion_mnist(*batch_size, resize=None*)

Fashion-MNIST veri kümesini indirin ve ardından belleğe yükleyin.

Defined in Section 3.5

d2l.torch.load_data_imdb(*batch_size, num_steps=500*)

Return data iterators and the vocabulary of the IMDb review dataset.

Defined in Section 15.1

d2l.torch.load_data_nmt(*batch_size, num_steps, num_examples=600*)

Return the iterator and the vocabularies of the translation dataset.

Defined in Section 9.5.4

d2l.torch.load_data_ptb(*batch_size, max_window_size, num_noise_words*)

Download the PTB dataset and then load it into memory.

Defined in Section 14.3.5

d2l.torch.load_data_snli(*batch_size, num_steps=50*)

Download the SNLI dataset and return data iterators and vocabulary.

Defined in Section 15.4

d2l.torch.load_data_time_machine(*batch_size, num_steps, use_random_iter=False, max_tokens=10000*)

Return the iterator and the vocabulary of the time machine dataset.

Defined in Section 8.3

d2l.torch.load_data_voc(*batch_size, crop_size*)

Load the VOC semantic segmentation dataset.

Defined in Section 13.9

d2l.torch.load_data_wiki(*batch_size, max_len*)

Load the WikiText-2 dataset.

Defined in Section 14.8.1

d2l.torch.masked_softmax(*X, valid_lens*)

Perform softmax operation by masking elements on the last axis.

Defined in Section 10.3

d2l.torch.multibox_detection(*cls_probs, offset_preds, anchors, nms_threshold=0.5, pos_threshold=0.009999999*)

Predict bounding boxes using non-maximum suppression.

Defined in Section 13.4.4

d2l.torch.multibox_prior(*data, sizes, ratios*)

Generate anchor boxes with different shapes centered on each pixel.

Defined in Section 13.4

d2l.torch.multibox_target(*anchors, labels*)

Label anchor boxes using ground-truth bounding boxes.

Defined in Section 13.4.3

d2l.torch.nms(*boxes, scores, iou_threshold*)

Sort confidence scores of predicted bounding boxes.

Defined in Section 13.4.4

d2l.torch.numpy(*x, *args, **kwargs*)

d2l.torch.offset_boxes(*anchors, assigned_bb, eps=1e-06*)

Transform for anchor box offsets.

Defined in Section 13.4.3

d2l.torch.offset_inverse(*anchors, offset_preds*)

Predict bounding boxes based on anchor boxes with predicted offsets.

Defined in Section 13.4.3

d2l.torch.plot(*X, Y=None, xlabel=None, ylabel=None, legend=None, xlim=None, ylim=None, xscale='linear', yscale='linear', fmts=('-', 'm--', 'g-.', 'r:'), figsize=(3.5, 2.5), axes=None*)

Veri noktalarını çiz.

Defined in Section 2.4

d2l.torch.predict_ch3(*net, test_iter, n=6*)

Etiketleri tahmin etme (Bolum 3'te tanimlanmistir).

Defined in Section 3.6

d2l.torch.predict_ch8(*prefix, num_preds, net, vocab, device*)

Generate new characters following the *prefix*.

Defined in Section 8.5

d2l.torch.predict_sentiment(*net, vocab, sequence*)

Predict the sentiment of a text sequence.

Defined in Section 15.2

d2l.torch.predict_seq2seq(*net, src_sentence, src_vocab, tgt_vocab, num_steps, device, save_attention_weights=False*)

Predict for sequence to sequence.

Defined in Section 9.7.4

d2l.torch.predict_snli(*net, vocab, premise, hypothesis*)

Predict the logical relationship between the premise and hypothesis.

Defined in Section 15.5

d2l.torch.preprocess_nmt(*text*)

Preprocess the English-French dataset.

Defined in Section 9.5

d2l.torch.read_csv_labels(*fname*)

Read *fname* to return a filename to label dictionary.

Defined in Section 13.13

d2l.torch.read_data_bananas(*is_train=True*)

Read the banana detection dataset images and labels.

Defined in Section 13.6

d2l.torch.read_data_nmt()
Load the English-French dataset.

Defined in Section 9.5

d2l.torch.read_imdb(*data_dir, is_train*)
Read the IMDb review dataset text sequences and labels.

Defined in Section 15.1

d2l.torch.read_ptb()
Load the PTB dataset into a list of text lines.

Defined in Section 14.3

d2l.torch.read_snli(*data_dir, is_train*)
Read the SNLI dataset into premises, hypotheses, and labels.

Defined in Section 15.4

d2l.torch.read_time_machine()
Load the time machine dataset into a list of text lines.

Defined in Section 8.2

d2l.torch.read_voc_images(*voc_dir, is_train=True*)
Read all VOC feature and label images.

Defined in Section 13.9

d2l.torch.reduce_sum(*x, *args, **kwargs*)

d2l.torch.reorg_test(*data_dir*)
Organize the testing set for data loading during prediction.

Defined in Section 13.13

d2l.torch.reorg_train_valid(*data_dir, labels, valid_ratio*)
Split the validation set out of the original training set.

Defined in Section 13.13

d2l.torch.reshape(*x, *args, **kwargs*)

d2l.torch.resnet18(*num_classes, in_channels=1*)
A slightly modified ResNet-18 model.

Defined in Section 12.6

d2l.torch.seq_data_iter_random(*corpus, batch_size, num_steps*)
Generate a minibatch of subsequences using random sampling.

Defined in Section 8.3

d2l.torch.seq_data_iter_sequential(*corpus, batch_size, num_steps*)
Generate a minibatch of subsequences using sequential partitioning.

Defined in Section 8.3

d2l.torch.sequence_mask(*X, valid_len, value=0*)

Mask irrelevant entries in sequences.

Defined in Section 9.7.2

d2l.torch.set_axes(*axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend*)

Set the axes for matplotlib.

Defined in Section 2.4

d2l.torch.set_figsize(*figsize=(3.5, 2.5)*)

Set the figure size for matplotlib.

Defined in Section 2.4

d2l.torch.sgd(*params, lr, batch_size*)

Minigrup rasgele gradyan inişi.

Defined in Section 3.2

d2l.torch.show_bboxes(*axes, bboxes, labels=None, colors=None*)

Show bounding boxes.

Defined in Section 13.4

d2l.torch.show_heatmaps(*matrices, xlabel, ylabel, titles=None, figsize=(2.5, 2.5), cmap='Reds'*)

Show heatmaps of matrices.

Defined in Section 10.1

d2l.torch.show_images(*imgs, num_rows, num_cols, titles=None, scale=1.5*)

Görsellerin bir listesini çizin

Defined in Section 3.5

d2l.torch.show_list_len_pair_hist(*legend, xlabel, ylabel, xlist, ylist*)

Plot the histogram for list length pairs.

Defined in Section 9.5

d2l.torch.show_trace_2d(*f, results*)

Show the trace of 2D variables during optimization.

Defined in Section 11.3.1

d2l.torch.size(*x, *args, **kwargs*)

d2l.torch.split_batch(*X, y, devices*)

Split X and y into multiple devices.

Defined in Section 12.5

d2l.torch.squared_loss(*y_hat, y*)

Kare kayip.

Defined in Section 3.2

d2l.torch.subsample(*sentences, vocab*)

Subsample high-frequency words.

Defined in Section 14.3

d2l.torch.synthetic_data(*w, b, num_examples*)

Veri yaratma, y = Xw + b + gurultu.

Defined in Section 3.2

d2l.torch.to(*x, *args, **kwargs*)

d2l.torch.tokenize(*lines, token='word'*)

Split text lines into word or character tokens.

Defined in Section 8.2

d2l.torch.tokenize_nmt(*text, num_examples=None*)

Tokenize the English-French dataset.

Defined in Section 9.5

d2l.torch.train_2d(*trainer, steps=20, f_grad=None*)

Optimize a 2D objective function with a customized trainer.

Defined in Section 11.3.1

d2l.torch.train_batch_ch13(*net, X, y, loss, trainer, devices*)

Train for a minibatch with mutiple GPUs (defined in Chapter 13).

Defined in Section 13.1

d2l.torch.train_ch11(*trainer_fn, states, hyperparams, data_iter, feature_dim, num_epochs=2*)

Defined in Section 11.5.2

d2l.torch.train_ch13(*net, train_iter, test_iter, loss, trainer, num_epochs, devices=[device(type='cuda', index=0), device(type='cuda', index=1), device(type='cuda', index=2), device(type='cuda', index=3)]*)

Train a model with mutiple GPUs (defined in Chapter 13).

Defined in Section 13.1

d2l.torch.train_ch3(*net, train_iter, test_iter, loss, num_epochs, updater*)

Bir modeli egitin (Bolum 3'te tanimlanmistir).

Defined in Section 3.6

d2l.torch.train_ch6(*net, train_iter, test_iter, num_epochs, lr, device*)

Train a model with a GPU (defined in Chapter 6).

Defined in Section 6.6

d2l.torch.train_ch8(*net, train_iter, vocab, lr, num_epochs, device, use_random_iter=False*)

Train a model (defined in Chapter 8).

Defined in Section 8.5

d2l.torch.train_concise_ch11(*trainer_fn, hyperparams, data_iter, num_epochs=4*)

Defined in Section 11.5.2

d2l.torch.train_epoch_ch3(*net, train_iter, loss, updater*)

Bolum 3'te tanimlanan egitim dongusu.

Defined in Section 3.6

d2l.torch.train_epoch_ch8(*net, train_iter, loss, updater, device, use_random_iter*)

Train a net within one epoch (defined in Chapter 8).

Defined in Section 8.5

d2l.torch.train_seq2seq(*net, data_iter, lr, num_epochs, tgt_vocab, device*)

Train a model for sequence to sequence.

Defined in Section 9.7.2

d2l.torch.transpose(*x, *args, **kwargs*)

d2l.torch.transpose_output(*X, num_heads*)

Reverse the operation of *transpose_qkv*.

Defined in Section 10.5

d2l.torch.transpose_qkv(*X, num_heads*)

Transposition for parallel computation of multiple attention heads.

Defined in Section 10.5

d2l.torch.truncate_pad(*line, num_steps, padding_token*)

Truncate or pad sequences.

Defined in Section 9.5

d2l.torch.try_all_gpus()

Return all available GPUs, or [cpu(),] if no GPU exists.

Defined in Section 5.5

d2l.torch.try_gpu(*i=0*)

Return gpu(i) if exists, otherwise return cpu().

Defined in Section 5.5

d2l.torch.update_D(*X, Z, net_D, net_G, loss, trainer_D*)

Update discriminator.

Defined in Section 17.1

d2l.torch.update_G(*Z, net_D, net_G, loss, trainer_G*)

Update generator.

Defined in Section 17.1

d2l.torch.use_svg_display()

Use the svg format to display a plot in Jupyter.

Defined in Section 2.4

`d2l.torch.voc_colormap2label()`

Build the mapping from RGB to class indices for VOC labels.

Defined in Section 13.9

`d2l.torch.voc_label_indices(`*colormap*, *colormap2label*`)`

Map any RGB values in VOC labels to their class indices.

Defined in Section 13.9

`d2l.torch.voc_rand_crop(`*feature*, *label*, *height*, *width*`)`

Randomly crop both feature and label images.

Defined in Section 13.9

Bibliography

Ahmed, A., Aly, M., Gonzalez, J., Narayanamurthy, S., & Smola, A. J. (2012). Scalable inference in latent variable models. *Proceedings of the fifth ACM international conference on Web search and data mining* (pp. 123–132).

Aji, S. M., & McEliece, R. J. (2000). The generalized distributive law. *IEEE transactions on Information Theory, 46*(2), 325–343.

Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer normalization. *arXiv preprint arXiv:1607.06450*.

Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Bay, H., Tuytelaars, T., & Van Gool, L. (2006). Surf: speeded up robust features. *European conference on computer vision* (pp. 404–417).

Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C. (2003). A neural probabilistic language model. *Journal of machine learning research, 3*(Feb), 1137–1155.

Bishop, C. M. (1995). Training with noise is equivalent to tikhonov regularization. *Neural computation, 7*(1), 108–116.

Bishop, C. M. (2006). *Pattern recognition and machine learning*. springer.

Bodla, N., Singh, B., Chellappa, R., & Davis, L. S. (2017). Soft-nms–improving object detection with one line of code. *Proceedings of the IEEE international conference on computer vision* (pp. 5561–5569).

Bollobás, B. (1999). *Linear analysis*. Cambridge University Press, Cambridge.

Bowman, S. R., Angeli, G., Potts, C., & Manning, C. D. (2015). A large annotated corpus for learning natural language inference. *arXiv preprint arXiv:1508.05326*.

Boyd, S., & Vandenberghe, L. (2004). *Convex Optimization*. Cambridge, England: Cambridge University Press.

Brown, N., & Sandholm, T. (2017). Libratus: the superhuman ai for no-limit poker. *IJCAI* (pp. 5226–5228).

Brown, P. F., Cocke, J., Della Pietra, S. A., Della Pietra, V. J., Jelinek, F., Lafferty, J., ... Roossin, P. S. (1990). A statistical approach to machine translation. *Computational linguistics, 16*(2), 79–85.

Brown, P. F., Cocke, J., Della Pietra, S. A., Della Pietra, V. J., Jelinek, F., Mercer, R. L., & Roossin, P. (1988). A statistical approach to language translation. *Coling Budapest 1988 Volume 1: International Conference on Computational Linguistics*.

Campbell, M., Hoane Jr, A. J., & Hsu, F.-h. (2002). Deep blue. *Artificial intelligence, 134*(1-2), 57–83.

Canny, J. (1987). A computational approach to edge detection. *Readings in computer vision* (pp. 184–203). Elsevier.

Cer, D., Diab, M., Agirre, E., Lopez-Gazpio, I., & Specia, L. (2017). Semeval-2017 task 1: semantic textual similarity multilingual and crosslingual focused evaluation. *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)* (pp. 1–14).

Cheng, J., Dong, L., & Lapata, M. (2016). Long short-term memory-networks for machine reading. *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing* (pp. 551–561).

Cho, K., Van Merriënboer, B., Bahdanau, D., & Bengio, Y. (2014). On the properties of neural machine translation: encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., & Kuksa, P. (2011). Natural language processing (almost) from scratch. *Journal of machine learning research, 12*(ARTICLE), 2493–2537.

Csiszár, I. (2008). Axiomatic characterizations of information measures. *Entropy, 10*(3), 261–273.

Dalal, N., & Triggs, B. (2005). Histograms of oriented gradients for human detection. *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)* (pp. 886–893).

De Cock, D. (2011). Ames, iowa: alternative to the boston housing data as an end of semester regression project. *Journal of Statistics Education, 19*(3).

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., ... Vogels, W. (2007). Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review* (pp. 205–220).

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018). Bert: pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Doersch, C., Gupta, A., & Efros, A. A. (2015). Unsupervised visual representation learning by context prediction. *Proceedings of the IEEE international conference on computer vision* (pp. 1422–1430).

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., ... others. (2021). An image is worth 16x16 words: transformers for image recognition at scale. *International Conference on Learning Representations*.

Doucet, A., De Freitas, N., & Gordon, N. (2001). An introduction to sequential monte carlo methods. *Sequential Monte Carlo methods in practice* (pp. 3–14). Springer.

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research, 12*(Jul), 2121–2159.

Dumoulin, V., & Visin, F. (2016). A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*.

Flammarion, N., & Bach, F. (2015). From averaging to acceleration, there is only a step-size. *Conference on Learning Theory* (pp. 658–695).

Gatys, L. A., Ecker, A. S., & Bethge, M. (2016). Image style transfer using convolutional neural networks. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 2414–2423).

Ginibre, J. (1965). Statistical ensembles of complex, quaternion, and real matrices. *Journal of Mathematical Physics, 6*(3), 440–449.

Girshick, R. (2015). Fast r-cnn. *Proceedings of the IEEE international conference on computer vision* (pp. 1440–1448).

Girshick, R., Donahue, J., Darrell, T., & Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 580–587).

Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (pp. 249–256).

Goh, G. (2017). Why momentum really works. *Distill*. URL: http://distill.pub/2017/momentum, doi:10.23915/distill.00006[259]

Goldberg, D., Nichols, D., Oki, B. M., & Terry, D. (1992). Using collaborative filtering to weave an information tapestry. *Communications of the ACM, 35*(12), 61–71.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... Bengio, Y. (2014). Generative adversarial nets. *Advances in neural information processing systems* (pp. 2672–2680).

Gotmare, A., Keskar, N. S., Xiong, C., & Socher, R. (2018). A closer look at deep learning heuristics: learning rate restarts, warmup and distillation. *arXiv preprint arXiv:1810.13243*.

Graves, A. (2013). Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*.

Graves, A., & Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural networks, 18*(5-6), 602–610.

Hadjis, S., Zhang, C., Mitliagkas, I., Iter, D., & Ré, C. (2016). Omnivore: an optimizer for multi-device deep learning on cpus and gpus. *arXiv preprint arXiv:1606.04487*.

He, K., Gkioxari, G., Dollár, P., & Girshick, R. (2017). Mask r-cnn. *Proceedings of the IEEE international conference on computer vision* (pp. 2961–2969).

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: surpassing human-level performance on imagenet classification. *Proceedings of the IEEE international conference on computer vision* (pp. 1026–1034).

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770–778).

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Identity mappings in deep residual networks. *European conference on computer vision* (pp. 630–645).

Hebb, D. O., & Hebb, D. (1949). *The organization of behavior*. Vol. 65. Wiley New York.

[259] https://doi.org/10.23915/distill.00006

Hendrycks, D., & Gimpel, K. (2016). Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*.

Hennessy, J. L., & Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.

Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J., & others (2001). *Gradient flow in recurrent nets: the difficulty of learning long-term dependencies*.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation, 9*(8), 1735–1780.

Hoyer, P. O., Janzing, D., Mooij, J. M., Peters, J., & Schölkopf, B. (2009). Nonlinear causal discovery with additive noise models. *Advances in neural information processing systems* (pp. 689–696).

Hu, J., Shen, L., & Sun, G. (2018). Squeeze-and-excitation networks. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 7132–7141).

Hu, Y., Koren, Y., & Volinsky, C. (2008). Collaborative filtering for implicit feedback datasets. *2008 Eighth IEEE International Conference on Data Mining* (pp. 263–272).

Hu, Z., Lee, R. K.-W., Aggarwal, C. C., & Zhang, A. (2020). Text style transfer: a review and experimental evaluation. *arXiv preprint arXiv:2010.12742*.

Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017). Densely connected convolutional networks. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 4700–4708).

Ioffe, S. (2017). Batch renormalization: towards reducing minibatch dependence in batch-normalized models. *Advances in neural information processing systems* (pp. 1945–1953).

Ioffe, S., & Szegedy, C. (2015). Batch normalization: accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

Izmailov, P., Podoprikhin, D., Garipov, T., Vetrov, D., & Wilson, A. G. (2018). Averaging weights leads to wider optima and better generalization. *arXiv preprint arXiv:1803.05407*.

Jaeger, H. (2002). *Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the" echo state network" approach*. Vol. 5. GMD-Forschungszentrum Informationstechnik Bonn.

James, W. (2007). *The principles of psychology*. Vol. 1. Cosimo, Inc.

Jia, X., Song, S., He, W., Wang, Y., Rong, H., Zhou, F., ... others. (2018). Highly scalable deep learning training system with mixed-precision: training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., ... others. (2017). In-datacenter performance analysis of a tensor processing unit. *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* (pp. 1–12).

Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2017). Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*.

Kim, Y. (2014). Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*.

Kingma, D. P., & Ba, J. (2014). Adam: a method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Koller, D., & Friedman, N. (2009). *Probabilistic graphical models: principles and techniques*. MIT press.

Kolter, Z. (2008). Linear algebra review and reference. *Available online: http*.

Koren, Y. (2009). Collaborative filtering with temporal dynamics. *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 447–456).

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* (pp. 1097–1105).

Kung, S. Y. (1988). Vlsi array processors. *Englewood Cliffs, NJ, Prentice Hall, 1988, 685 p. Research supported by the Semiconductor Research Corp., SDIO, NSF, and US Navy.*

LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., & others. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278–2324.

Li, M. (2017). *Scaling Distributed Machine Learning with System and Algorithm Co-design* (Doctoral dissertation). PhD Thesis, CMU.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., ... Su, B.-Y. (2014). Scaling distributed machine learning with the parameter server. *11th $\$USENIX$\$ Symposium on Operating Systems Design and Implementation ($\$OSDI$\$ 14)* (pp. 583–598).

Lin, M., Chen, Q., & Yan, S. (2013). Network in network. *arXiv preprint arXiv:1312.4400*.

Lin, T.-Y., Goyal, P., Girshick, R., He, K., & Dollár, P. (2017). Focal loss for dense object detection. *Proceedings of the IEEE international conference on computer vision* (pp. 2980–2988).

Lin, Y., Lv, F., Zhu, S., Yang, M., Cour, T., Yu, K., ... others. (2010). Imagenet classification: fast descriptor coding and large-scale svm training. *Large scale visual recognition challenge*.

Lin, Z., Feng, M., Santos, C. N. d., Yu, M., Xiang, B., Zhou, B., & Bengio, Y. (2017). A structured self-attentive sentence embedding. *arXiv preprint arXiv:1703.03130*.

Lipton, Z. C., & Steinhardt, J. (2018). Troubling trends in machine learning scholarship. *arXiv preprint arXiv:1807.03341*.

Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., & Berg, A. C. (2016). Ssd: single shot multibox detector. *European conference on computer vision* (pp. 21–37).

Long, J., Shelhamer, E., & Darrell, T. (2015). Fully convolutional networks for semantic segmentation. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 3431–3440).

Loshchilov, I., & Hutter, F. (2016). Sgdr: stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*.

Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, *60*(2), 91–110.

Luo, P., Wang, X., Shao, W., & Peng, Z. (2018). Towards understanding regularization in batch normalization. *arXiv preprint*.

Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., & Potts, C. (2011). Learning word vectors for sentiment analysis. *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies-volume 1* (pp. 142–150).

McCann, B., Bradbury, J., Xiong, C., & Socher, R. (2017). Learned in translation: contextualized word vectors. *Advances in Neural Information Processing Systems* (pp. 6294–6305).

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*(4), 115–133.

Merity, S., Xiong, C., Bradbury, J., & Socher, R. (2016). Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems* (pp. 3111–3119).

Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., ... Dean, J. (2017). Device placement optimization with reinforcement learning. *Proceedings of the 34th International Conference on Machine Learning-Volume 70* (pp. 2430–2439).

Mnih, V., Heess, N., Graves, A., & others. (2014). Recurrent models of visual attention. *Advances in neural information processing systems* (pp. 2204–2212).

Morey, R. D., Hoekstra, R., Rouder, J. N., Lee, M. D., & Wagenmakers, E.-J. (2016). The fallacy of placing confidence in confidence intervals. *Psychonomic bulletin & review*, *23*(1), 103–123.

Nadaraya, E. A. (1964). On estimating regression. *Theory of Probability & Its Applications*, *9*(1), 141–142.

Nesterov, Y., & Vial, J.-P. (2000). *Confidence level solutions for stochastic programming, Stochastic Programming E-Print Series*.

Nesterov, Y. (2018). *Lectures on convex optimization*. Vol. 137. Springer.

Neyman, J. (1937). Outline of a theory of statistical estimation based on the classical theory of probability. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, *236*(767), 333–380.

Parikh, A. P., Täckström, O., Das, D., & Uszkoreit, J. (2016). A decomposable attention model for natural language inference. *arXiv preprint arXiv:1606.01933*.

Park, T., Liu, M.-Y., Wang, T.-C., & Zhu, J.-Y. (2019). Semantic image synthesis with spatially-adaptive normalization. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 2337–2346).

Paulus, R., Xiong, C., & Socher, R. (2017). A deep reinforced model for abstractive summarization. *arXiv preprint arXiv:1705.04304*.

Pennington, J., Schoenholz, S., & Ganguli, S. (2017). Resurrecting the sigmoid in deep learning through dynamical isometry: theory and practice. *Advances in neural information processing systems* (pp. 4785–4795).

Pennington, J., Socher, R., & Manning, C. (2014). Glove: global vectors for word representation. *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)* (pp. 1532–1543).

Peters, J., Janzing, D., & Schölkopf, B. (2017). *Elements of causal inference: foundations and learning algorithms*. MIT press.

Peters, M., Ammar, W., Bhagavatula, C., & Power, R. (2017). Semi-supervised sequence tagging with bidirectional language models. *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (pp. 1756–1765).

Peters, M., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., & Zettlemoyer, L. (2018). Deep contextualized word representations. *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)* (pp. 2227–2237).

Petersen, K. B., Pedersen, M. S., & others. (2008). The matrix cookbook. *Technical University of Denmark*, *7*(15), 510.

Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, *4*(5), 1–17.

Quadrana, M., Cremonesi, P., & Jannach, D. (2018). Sequence-aware recommender systems. *ACM Computing Surveys (CSUR)*, *51*(4), 66.

Radford, A., Metz, L., & Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*.

Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). Improving language understanding by generative pre-training. *OpenAI*.

Rajpurkar, P., Zhang, J., Lopyrev, K., & Liang, P. (2016). Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*.

Reddi, S. J., Kale, S., & Kumar, S. (2019). On the convergence of adam and beyond. *arXiv preprint arXiv:1904.09237*.

Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You only look once: unified, real-time object detection. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 779–788).

Reed, S., & De Freitas, N. (2015). Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*.

Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster r-cnn: towards real-time object detection with region proposal networks. *Advances in neural information processing systems* (pp. 91–99).

Rumelhart, D. E., Hinton, G. E., Williams, R. J., & others. (1988). Learning representations by back-propagating errors. *Cognitive modeling*, *5*(3), 1.

Russell, S. J., & Norvig, P. (2016). *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.

Santurkar, S., Tsipras, D., Ilyas, A., & Madry, A. (2018). How does batch normalization help optimization? *Advances in Neural Information Processing Systems* (pp. 2483–2493).

Sarwar, B. M., Karypis, G., Konstan, J. A., Riedl, J., & others. (2001). Item-based collaborative filtering recommendation algorithms. *Www*, *1*, 285–295.

Schein, A. I., Popescul, A., Ungar, L. H., & Pennock, D. M. (2002). Methods and metrics for cold-start recommendations. *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval* (pp. 253–260).

Schuster, M., & Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, *45*(11), 2673–2681.

Sergeev, A., & Del Balso, M. (2018). Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*.

Shannon, C. E. (1948 , 7). A mathematical theory of communication. *The Bell System Technical Journal*, *27*(3), 379–423.

Shao, H., Yao, S., Sun, D., Zhang, A., Liu, S., Liu, D., ... Abdelzaher, T. (2020). Controlvae: controllable variational autoencoder. *Proceedings of the 37th International Conference on Machine Learning*.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... others. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, *529*(7587), 484.

Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Smola, A., & Narayanamurthy, S. (2010). An architecture for parallel topic models. *Proceedings of the VLDB Endowment*, *3*(1-2), 703–710.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, *15*(1), 1929–1958.

Strang, G. (1993). *Introduction to linear algebra*. Vol. 3. Wellesley-Cambridge Press Wellesley, MA.

Su, X., & Khoshgoftaar, T. M. (2009). A survey of collaborative filtering techniques. *Advances in artificial intelligence*, *2009*.

Sukhbaatar, S., Weston, J., Fergus, R., & others. (2015). End-to-end memory networks. *Advances in neural information processing systems* (pp. 2440–2448).

Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013). On the importance of initialization and momentum in deep learning. *International conference on machine learning* (pp. 1139–1147).

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. *Advances in neural information processing systems* (pp. 3104–3112).

Szegedy, C., Ioffe, S., Vanhoucke, V., & Alemi, A. A. (2017). Inception-v4, inception-resnet and the impact of residual connections on learning. *Thirty-First AAAI Conference on Artificial Intelligence*.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... Rabinovich, A. (2015). Going deeper with convolutions. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1–9).

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 2818–2826).

Tallec, C., & Ollivier, Y. (2017). Unbiasing truncated backpropagation through time. *arXiv preprint arXiv:1705.08209*.

Tay, Y., Dehghani, M., Bahri, D., & Metzler, D. (2020). Efficient transformers: a survey. *arXiv preprint arXiv:2009.06732*.

Teye, M., Azizpour, H., & Smith, K. (2018). Bayesian uncertainty estimation for batch normalized deep networks. *arXiv preprint arXiv:1802.06455*.

Tieleman, T., & Hinton, G. (2012). Lecture 6.5-rmsprop: divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, *4*(2), 26–31.

Turing, A. (1950). Computing machinery and intelligence. *Mind*, *59*(236), 433.

Uijlings, J. R., Van De Sande, K. E., Gevers, T., & Smeulders, A. W. (2013). Selective search for object recognition. *International journal of computer vision*, *104*(2), 154–171.

Van Loan, C. F., & Golub, G. H. (1983). *Matrix computations*. Johns Hopkins University Press.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems* (pp. 5998–6008).

Wang, L., Li, M., Liberty, E., & Smola, A. J. (2018). Optimal message scheduling for aggregation. *NETWORKS*, *2*(3), 2–3.

Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., & Owens, J. D. (2016). Gunrock: a high-performance graph processing library on the gpu. *ACM SIGPLAN Notices* (p. 11).

Warstadt, A., Singh, A., & Bowman, S. R. (2019). Neural network acceptability judgments. *Transactions of the Association for Computational Linguistics*, *7*, 625–641.

Wasserman, L. (2013). *All of statistics: a concise course in statistical inference*. Springer Science & Business Media.

Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine learning*, *8*(3-4), 279–292.

Watson, G. S. (1964). Smooth regression analysis. *Sankhyā: The Indian Journal of Statistics, Series A*, pp. 359–372.

Welling, M., & Teh, Y. W. (2011). Bayesian learning via stochastic gradient langevin dynamics. *Proceedings of the 28th international conference on machine learning (ICML-11)* (pp. 681–688).

Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, *78*(10), 1550–1560.

Wigner, E. P. (1958). On the distribution of the roots of certain symmetric matrices. *Ann. Math* (pp. 325–327).

Wood, F., Gasthaus, J., Archambeau, C., James, L., & Teh, Y. W. (2011). The sequence memoizer. *Communications of the ACM*, *54*(2), 91–98.

Wu, C.-Y., Ahmed, A., Beutel, A., Smola, A. J., & Jing, H. (2017). Recurrent recommender networks. *Proceedings of the tenth ACM international conference on web search and data mining* (pp. 495–503).

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., ... others. (2016). Google's neural machine translation system: bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

Xiao, H., Rasul, K., & Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*.

Xiao, L., Bahri, Y., Sohl-Dickstein, J., Schoenholz, S., & Pennington, J. (2018). Dynamical isometry and a mean field theory of cnns: how to train 10,000-layer vanilla convolutional neural networks. *International Conference on Machine Learning* (pp. 5393–5402).

Xiong, W., Wu, L., Alleva, F., Droppo, J., Huang, X., & Stolcke, A. (2018). The microsoft 2017 conversational speech recognition system. *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 5934–5938).

Ye, M., Yin, P., Lee, W.-C., & Lee, D.-L. (2011). Exploiting geographical influence for collaborative point-of-interest recommendation. *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval* (pp. 325–334).

You, Y., Gitman, I., & Ginsburg, B. (2017). Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*.

Zaheer, M., Reddi, S., Sachan, D., Kale, S., & Kumar, S. (2018). Adaptive methods for nonconvex optimization. *Advances in Neural Information Processing Systems* (pp. 9793–9803).

Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.

Zhang, A., Tay, Y., Zhang, S., Chan, A., Luu, A. T., Hui, S. C., & Fu, J. (2021). Beyond fully-connected layers with quaternions: parameterization of hypercomplex multiplications with 1/n parameters. *International Conference on Learning Representations*.

Zhang, S., Yao, L., Sun, A., & Tay, Y. (2019). Deep learning based recommender system: a survey and new perspectives. *ACM Computing Surveys (CSUR), 52*(1), 5.

Zhao, Z.-Q., Zheng, P., Xu, S.-t., & Wu, X. (2019). Object detection with deep learning: a review. *IEEE transactions on neural networks and learning systems, 30*(11), 3212–3232.

Zhu, J.-Y., Park, T., Isola, P., & Efros, A. A. (2017). Unpaired image-to-image translation using cycle-consistent adversarial networks. *Proceedings of the IEEE international conference on computer vision* (pp. 2223–2232).

Zhu, Y., Kiros, R., Zemel, R., Salakhutdinov, R., Urtasun, R., Torralba, A., & Fidler, S. (2015). Aligning books and movies: towards story-like visual explanations by watching movies and reading books. *Proceedings of the IEEE international conference on computer vision* (pp. 19–27).

Python Module Index

d

Index

G

I

L

M

N

O

P

R

S

T

U

V