Neural Networks : Derivation

compiled by Alvin Wan from Professor Jitendra Malik's lecture

This type of computation is called "deep learning" and is the most popular method for many problems, such as computer vision and image processing.

1 Model

For now, let us focus on a specific model of neurons. These are simplified from reality but can achieve remarkable results.

1.1 Single Layer Neural Newtork

Let us call the inputs $x_1, x_2 \dots x_n$. We have several nodes $v_1, v_2 \dots v_n$, which each receive a linear combination of the inputs, $w_1x_1 + w_2x_2 + \cdots + w_nx_n$. The convention for each edge weight is to list the index of the source in the first position and the index of the destination in the second. For example, edge linking input 1 to node 2, would be w_{12}.

The total input for node 2 would be

$$S_2 = \sum_{i=1}^{n} w_{i2}x_i$$

There are variety of ways we translate S_2 into a decision,

$$x_2 = g(S_2)$$

We call g the **activation function**. We have several common options for activation functions.

1. **Logistic** $\frac{1}{1+e^{-z}}$
2. **ReLu** (Rectified linear unit)

$$\begin{cases} 0 & \text{for } z \leq 0 \\ z & \text{for } z > 0 \end{cases}$$

In a single-layer neural network, if g is logistic, we would simply have logistic regression.

1.2 Multi-Layer Neural Networks

The set of inputs could also be the output from another stage of the neural network. Now, we will consider a three-layer neural network. We call the inputs x_i, the second-layer v_i and the third layer o_i, where all activation functions g are the same. We can compute the outputs using the following.

$$O_i = g(\sum_j w_{ij} g(\sum_k w_{jk} x_k))$$

Suppose the first stage and the second stage are all linear. Is this effective? Of course not. This is effectively one layer of linear combinations. As a result, it is important that you have some non-linearity. With that said, even the discontinuity of a ReLu is sufficient.

2 Training

What does training a neural network mean? It means finding a w such that the output o_i is as close as possible to y_i, the desired output. In other words, the values for a regression problem or the labels for a classification problem.

Here is our 3-step approach:

1. Define the loss function $\mathcal{L}(w)$.
2. Compute $\nabla w\alpha$.
3. $w_{new} \leftarrow w_{old} - \eta \nabla_w \alpha$

Compute the gradient of w means we compute partial derivatives. In other words, we compute $\nabla_w = \frac{\partial \mathcal{L}}{\partial w_{jk}}$ for all edges using chain rule. We want to know: how can we tweak the edge weight, so that our loss is minimized?

2.1 Single-Layer Neural Networks

For binary classification, an effective loss function is the cross entropy. For regression, use squared error. Note that the following comes from the maximum likelihood estimate for logistic.

$$\mathcal{L} = -\sum_{X}(y_i \log o_i + (1 - y_i) \log 1 - o_i)$$

We can model the activation function g as a sigmoid $g(z) = \frac{1}{1+e^{-z}}$.

Finding w reduces to logistic regression, so we can use stochastic gradient descent.

2.2 Two-Layer Neural Network

Note that this generalizes to multiple layers.

- We can compute the gradient with respect to all the weights, from the input to the hidden layer and the hidden layer to the output layer. The vector we get with may be massive, which is linear in size with respect to the weights.
- We can use stochastic gradient descent. Despite the fact that it only finds local minima, this is sufficient for more applications.
- *Computing gradients the naive version will be quadratic in the number of weights.* The back propagation is a trick that allows us to compute the gradient in linear time.
- We can add a regularization term to improve performance.

Here is the simple idea; the following is an invocation of g, with many arguments, one of which is w_{jk}.

$$o_i = g(\ldots w_{jk} \ldots, \vec{x})$$

We can compute the following.

$$o_i' = g(\dots w_{jk} + \Delta w_{jk} \dots, \vec{x})$$

Then, we can compute the losses $L(o_i), L(o_i')$. We now have a numerical approximation for the gradient.

$$\frac{\mathcal{L}(o_i') - \mathcal{L}(o_i)}{\Delta w_{jk}}$$

The computational complexity of this is what is called a **forward pass** through the network, as we evaluate the entire network. The cost of this is linear in the number of weights. Given n weights, it would be $O(n)$ for a single weight, but with n weights, this is $O(n^2)$ which is extremely expensive.

3 Backpropagation Algorithm

First, let us consider the big picture. We can see that computation can be and should be shared, so we will try to minimize work in this fashion. Consider an input layer, a hidden layer, and output layer. We will compute a variable called δ at the output layer. Using the deltas, we will compute the deltas for the hidden layer, and finally, compute the deltas at the input layer. This is why the algorithm is called "back propagation", as we move from the outputs to the inputs.

We can consider this a form of induction, where the output layer is the base case.

3.1 Chain Rule

Let us move on to finer details and develop a more precise structure. Recall the chain rule from calculus. Consider the following

$$\begin{aligned} f(x) &= x^2 z \\ x &= 2y \\ z &= \sin y \end{aligned}$$

To compute $\frac{\partial f}{\partial y}$, we apply chain rule. Intuitively, a small change in y causes a change in z, which causes a small change in x, which changes f. Alternatively, a small change in y could directly affect x, which changes f. There are, in effect, two "paths" to f. Chain rule tells us that

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial y} + \frac{\partial f}{\partial z}\frac{\partial z}{\partial y}$$

3.2 Notation

To distinguish between weights in different layers, we will use superscripts.

- $w_{ij}^{(l)}$ is the link from node i in layer $l-1$ to node j in layer l. Note that l is the layer of the target node.
- Let L be the index of the highest layer, so if we have $L = 10$, l will range from 1 to 10.
- $d^{(l)}$ will denote the number of nodes at the layer l.

Notation is fairly confusing, but we will try to follow the aforementioned conventions. To compute x_j for some node j in layer l, we have the following. We simply took the equation from before and added superscripts to denote layers.

$$S_j^{(l)} = \sum_{i=1}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)}$$

$$x_j^{(l)} = g(S_j^{(l)})$$

3.3 Defining δ

We will use e for "error", which is equivalent to "loss". Let us first define the error at a specific node.

$$\delta_j^{(l)} = \frac{\partial e(w)}{\partial S_j^{(l)}}$$

Using these deltas, we can then compute the gradients. Here is how:

Recall that we are looking for the gradient of the error *with respect to a weight.*

$$\frac{\partial e(w)}{\partial w_{ij}^{(l)}}$$

w_{ij} affects S_j, and S_j then affects x_j. We can also say the reverse. The error of at x_j is a function of the error in S_j, which is then a function of the error in w_{ij}. More formally said, we can apply chain rule to re-express the derivative of e with respect to the weight.

$$\frac{\partial e(w)}{\partial w_{ij}^{(l)}} = \frac{\partial e(w)}{\partial s_j^{(l)}} \times \frac{\partial S_j^{(l)}}{\partial w_{ij}^{(l)}}$$

We can now use our delta notation. In fact, note that the first term is simply our delta.

$$= \delta_j^{(l)} \frac{\partial S_j^{(l)}}{\partial w_{ij}^{(l)}}$$

S_j is a linear combination of all w_{ij}, but we are taking the derivative with respect to a specific w_{ij}. As a result, we have that $\frac{\partial S_j^{(l)}}{\partial w_{ij}^{(l)}} = x_i^{(l-1)}$.

$$= \delta_j^{(l)} x_i^{(l-1)}$$

If we can compute deltas at every node in the neural network, we have an extremely simple formula for the gradients at every node, for every weight. As a result, we have achieved our goal, which was to compute the gradient of the error with respect to each weight.

We will now compute the gradient of the error with respect to each node.

3.4 Base Case

We will now build the full algorithm, using induction. First, run the entire network, so that we have the values $g(s_i^L)$. We compute δ in the final layer, to start.

$$\delta = \frac{\partial e(w)}{\partial S_j^{(L)}}$$

Given the following error function, we can plug in and take the derivative to compute δ.

$$\textsc{Error}_i = \frac{1}{2}(g(s_i^{(L)}) - y_i)^2$$

We compute the deltas for our output layer first. Apply chain rule, and note that y_i constant with respect to w_{ij}.

$$\begin{aligned}\frac{\partial e(w)}{\partial \delta_i^{(L)}} &= \frac{\partial}{\partial \delta_i^{(L)}}(\frac{1}{2}(g(s_i^{(L)}) - y_i)^2) \\ &= (g(s_i^{(L)}) - y_i)g'(s_i^{(L)})\end{aligned}$$

Suppose g is the ReLu. Then, we have the following possible outputs for $g'(s_i^{(L)})$.

$$g'(S_i^{(L)}) = \begin{cases} 1 \text{ if } s_i^{(L)} \geq 0 \\ 0 \text{ otherwise} \end{cases}$$

3.5 Inductive Step

We compute δ for an intermediate layer. Take an identical definition of δ. Per the inductive hypothesis, assume that all $\delta_i^{(l)}$ have been computed.

$$\delta_i^{(l-1)} = \frac{\partial e(w)}{\partial S_i^{(l-1)}}$$

We examine $x_i^{(l-1)}$. Which quantities does it affect? The answer: it may affect any node in the layer above it, $x_i^{(l)}$. Specifically, $x_i^{(l-1)}$ affects $S_i^{(l)}$, which in turn affects $x_i^{(l)}$. This is, again, expressed formally using chain rule.

$$= \sum_j \frac{\partial e(w)}{\partial S_j^{(l)}} \frac{\partial S_j}{\partial x_i^{(l-1)}} \frac{\partial x_i^{(l-1)}}{\partial s_i^{(l-1)}}$$

Let us now simplify this expression. The first term, is in fact $\delta_i^{(l)}$, by definition.

$$= \sum_j \delta_i^{(l)} \frac{\partial S_j}{\partial x_i^{(l-1)}} \frac{\partial x_i^{(l-1)}}{\partial s_i^{(l-1)}}$$

Recall that S_j is a linear combination of weights and nodes, so the second term is simply w_{ij}.

$$= \sum_j \delta_i^{(l)} w_{ij}^{(l)} \frac{\partial x_i^{(l-1)}}{\partial s_i^{(l-1)}}$$

Finally, the last term can be re-expressed using the activation function g.

$$= \sum_j \delta_i^{(l)} w_{ij}^{(l)} g'(S_i^{(l-1)})$$

3.6 Full Algorithm

Note that this algorithm is linear in the number of weights in the network and *not* linear in the number of nodes. The summation derived above demonstrates this.

1. We compute δ at the output layer.
2. We compute δ at the intermediate layers.
3. Once this has concluded, we can use the following t compute the derivative with respect to any weight w_{ij}.

$$\frac{\partial e(w)}{\partial w_{ij}^{(l)} = \delta_j^{(l)} x_i^{(l-1)}}$$

This concludes the introduction to neural networks and their derivation.