Neural Networks 110 (2019) 104–115

Contents lists available at ScienceDirect

Neural Networks

journal homepage: www.elsevier.com/locate/neunet

ADA-Tucker: Compressing deep neural networks via adaptive dimension adjustment tucker decomposition

Zhisheng Zhong, Fangyin Wei, Zhouchen Lin, Chao Zhang *

Key Laboratory of Machine Perception (MOE), School of EECS, Peking University, PR China

ARTICLE INFO

Article history:
Received 16 December 2017
Received in revised form 2 October 2018
Accepted 30 October 2018
Available online 13 November 2018

Keywords:
Convolutional neural network
Compression
Tucker decomposition
Dimension adjustment

ABSTRACT

Despite recent success of deep learning models in numerous applications, their widespread use on mobile devices is seriously impeded by storage and computational requirements. In this paper, we propose a novel network compression method called Adaptive Dimension Adjustment Tucker decomposition (ADA-Tucker). With learnable core tensors and transformation matrices, ADA-Tucker performs Tucker decomposition of *arbitrary-order* tensors. Furthermore, we propose that weight tensors in networks with proper order and balanced dimension are easier to be compressed. Therefore, the high flexibility in decomposition choice distinguishes ADA-Tucker from all previous low-rank models. To compress more, we further extend the model to Shared Core ADA-Tucker (SCADA-Tucker) by defining a shared core tensor for all layers. Our methods require no overhead of recording indices of non-zero elements. Without loss of accuracy, our methods reduce the storage of LeNet-5 and LeNet-300 by ratios of **691**× and **233**×, respectively, significantly outperforming state of the art. The effectiveness of our methods is also evaluated on other three benchmarks (CIFAR-10, SVHN, ILSVRC12) and modern newly deep networks (ResNet, Wide-ResNet).

1. Introduction

Driven by increasing computation power of GPUs and huge amount of data, deep learning has recently made great achievements in computer vision, natural language processing and speech recognition. In the history of neural network (He, Zhang, Ren, & Sun, 2016; Huang, Liu, Weinberger, & van der Maaten, 2017; Krizhevsky, Sutskever, & Hinton, 2012; LeCun, Bottou, Bengio, & Haffner, 1998; Simonyan & Zisserman, 2014; Szegedy et al., 2015), networks tend to have more layers and more weights. Although deeper neural networks may achieve better results, the expense of storage and computation is still a great challenge. Due to limits of devices and increasing demands from many applications, effective network compression for convolutional (Conv) layers and fully-connected (FC) layers is a critical research topic in deep learning.

So far, as illustrated in Fig. 1, mainstream methods for network compression can be categorized into four groups: reducing the bits of weight representation, effective coding, making weights sparse and simplifying the network structure. These four methods can be combined together for higher compression ratio with little loss in network performance. Han et al. have combined the first three methods in Han, Mao, and Dally (2016).

Reducing the bits of weight representation and effective coding. There are two approaches for the first category: clustering and quantization. BinaryConnect (Courbariaux, Bengio, & David, 2015) enforces weights in neural networks to take binary values. Incremental network quantization (Zhou, Yao, Guo, Xu, & Chen, 2017) quantizes deep models with 5 bits incrementally. Gong, Liu, Yang, and Bourdev, (2015) learn CNNs in advance, and then apply k-means clustering on the weights for quantization. Ullrich, Meeds, and Welling, (2017) cluster the weights with a Gaussian mixture model (GMM), using only six class centers to represent all weights. The second category, effective coding, always combines with the first category, where the coding scheme is mainly Huffman coding. DeepCompression Han et al. (2016) first introduces Huffman coding in network compression and improves the compression ratios further. CNNPack (Wang, Xu, You, Tao, & Xu, 2016) also uses Huffman coding and gets better results.

Making weights sparse. Sparsity can be induced in either the original domain or the frequency domain. The most commonly used sparsity method in the original domain is pruning. Han, Pool, Tran, & Dally, (2015) recursively train a neural network and prune unimportant connections based on their weight magnitude. Dynamic network surgery (Guo, Yao, & Chen, 2016) prunes and splices the branch of the network. The frequency domain sparsity methods benefit from discrete cosine transformation (DCT). Chen, Wilson, Tyree, Weinberger, and Chen (2016) take advantage of DCT to make

* Corresponding author.
E-mail addresses: zszhong@pku.edu.cn (Z. Zhong), weifangyin@pku.edu.cn (F. Wei), zlin@pku.edu.cn (Z. Lin), chzhang@cis.pku.edu.cn (C. Zhang).

https://doi.org/10.1016/j.neunet.2018.10.016
0893-6080/

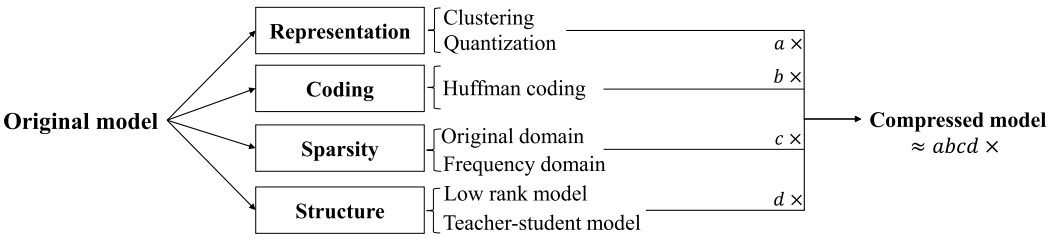

Fig. 1. Four categories of mainstream compression methods, which can be combined for higher compression ratio. "$a\times$" etc. means that the network is compressed by a times.

weights sparse in the frequency domain. Wang et al. (2016) combine DCT, clustering and Huffman coding for further compression.

Simplifying the network structure. A common approach of the fourth category involves matrix and tensor decomposition, while another rarely used approach called teacher–student model (Ba & Caruana, 2014; Hinton, Vinyals, & Dean, 2014) tries to reduce the depth of networks. Low-rank models were first used in the fully-connected layer (Denil, Shakibi, Dinh, de Freitas, et al., 2013). They utilize singular value decomposition (SVD) to reduce the computation and storage. Tensor Train decomposition (Novikov, Podoprikhin, Osokin, & Vetrov, 2015) is another model to compress fully-connected layer. Denton, Zaremba, Bruna, LeCun, and Fergus (2014), Jaderberg, Vedaldi, and Zisserman (2014) and Tai, Xiao, Zhang, Wang, and (2016) speed up CNNs with low-rank regularization. Canonical Polyadic (Lebedev, Ganin, Rakhuba, Oseledets, & Lempitsky, 2015) and Tucker decomposition (Kim et al., 2016) are advocated to accelerate the training of CNNs.

Our model falls into the last category, and it differs from the existing methods in two folds. First, while previous methods generally decompose weight tensors with fixed order and dimension, our methods adaptively adjust the original weight tensor into a new tensor with arbitrary order before Tucker decomposition. The superiority of such flexibility will be explained and demonstrated in the next section. Second, the proposed model can be applied to *both Conv and FC layers*, requiring no definition of new layers. In fact, previous low-rank models implemented by defining additional layers are special cases of our methods.

In principle, our methods can also combine with other three categories for higher compression ratios. In the experiments section, we combine quantization and Huffman coding for better results.

In summary, our contributions are as follows:

- We demonstrate that deep neural networks can be better compressed using weight tensors with proper orders and balanced dimensions of modes without performance degradation.
- We propose a novel network compression method called ADA-Tucker with flexible decomposition that drastically compresses deep networks while learning.
- We further extend ADA-Tucker to SCADA-Tucker with a shared core tensor for all layers, achieving even higher compression ratios with negligible accuracy loss.

2. ADA-Tucker and SCADA-Tucker

Notations: Following Kolda and Bader (2009), tensors are denoted by boldface Euler script letters, e.g., $\boldsymbol{\mathcal{A}}$, matrices are denoted by boldface capital letters, e.g., $\boldsymbol{A}$, vectors are denoted by boldface lowercase letters, e.g., $\boldsymbol{a}$, and scalars are denoted by lowercase letters, e.g., a. $\boldsymbol{\mathcal{A}}^{(i)}$ represents the parameters of the ith layer and $\boldsymbol{A}_{(i)}$ represents the i-mode of tensor $\boldsymbol{\mathcal{A}}$.

2.1. Tensor decomposition on the weight tensor

Weights of a deep neural network mainly come from Conv layers and FC layers. With weights in both types of layer represented by tensors, methods based on tensor decomposition can be applied to reduce the weight numbers.

For a Conv layer, its weight can be represented by a fourth order tensor $\boldsymbol{\mathcal{W}} \in \mathbb{R}^{h\times w\times s\times t}$, where h and w represent the height and width of the kernel, respectively, and s and t represent the channel number of input and output, respectively. Similarly, the weight of a FC layer can be viewed as a second order tensor $\boldsymbol{\mathcal{W}} \in \mathbb{R}^{s\times t}$, where s and t represent the number of the layer's input and output units, respectively. Thus in general, the form of a weight tensor is a d_wth order $(m_1, m_2, \ldots, m_{d_w})$-dimensional tensor $\boldsymbol{\mathcal{W}} \in \mathbb{R}^{m_1\times m_2\times\cdots\times m_{d_w}}$, where m_i is the dimension of the ith mode.

The weight tensor can be original if the magnitude of m_i's is balanced. Otherwise, it can be a reshaped version of the original tensor according to the adaptive dimension adjustment mechanism described in the next subsection. Suppose that $\boldsymbol{\mathcal{W}}$ is reshaped into $\tilde{\boldsymbol{\mathcal{W}}} \in \mathbb{R}^{n_1\times n_2\times\cdots\times n_{d_c}}$, where $n_1 \times n_2 \times \cdots \times n_{d_c} = m_1 \times m_2 \times \cdots \times m_{d_w}$. Then based on Tucker decomposition, we decompose the reshaped weight tensor $\tilde{\boldsymbol{\mathcal{W}}}$ into a d_c-mode product of a core tensor $\boldsymbol{\mathcal{C}}$ and a series of transformation matrices $\{\boldsymbol{M}\}$:

$$\tilde{\boldsymbol{\mathcal{W}}} \approx \boldsymbol{\mathcal{C}} \times_1 \boldsymbol{M}_1 \times_2 \boldsymbol{M}_2 \times_3 \ldots \times_{d_c} \boldsymbol{M}_{d_c}, \tag{1}$$

where $\boldsymbol{\mathcal{C}} \in \mathbb{R}^{k_1\times k_2\times\cdots\times k_{d_c}}$ and $\boldsymbol{M}_i \in \mathbb{R}^{n_i\times k_i}$ $(i = 1, 2, \ldots, d_c)$ are all learnable. They need to be stored during training in order to reconstruct $\boldsymbol{\mathcal{W}}$: after the d_c-mode product w.r.t. $\boldsymbol{\mathcal{C}}$, we reshape $\tilde{\boldsymbol{\mathcal{W}}}$ into $\boldsymbol{\mathcal{W}}$ so as to produce the output of the layer in forward propagation and pass the gradients in backward propagation.

We define $\tilde{\boldsymbol{W}}_{(i)} \in \mathbb{R}^{n_i\times(n_1\ldots n_{i-1}n_{i+1}\ldots n_{d_c})}$ and $\boldsymbol{C}_{(i)} \in \mathbb{R}^{k_i\times(k_1\ldots k_{i-1}k_{i+1}\ldots k_{d_c})}$ as the i-mode unfolding of tensor $\tilde{\boldsymbol{\mathcal{W}}}$ and $\boldsymbol{\mathcal{C}}$, respectively, and rewrite Eq. (1) as:

$$\tilde{\boldsymbol{W}}_{(i)} = \boldsymbol{M}_i \boldsymbol{C}_{(i)} \left(\boldsymbol{M}_{d_c} \otimes \boldsymbol{M}_{d_c-1} \otimes \cdots \otimes \boldsymbol{M}_{i+1} \otimes \boldsymbol{M}_{i-1} \otimes \cdots \otimes \boldsymbol{M}_1\right)^T, \tag{2}$$

where $\otimes$ represents the Kronecker product. The gradients of loss L w.r.t. the core tensors and the transformation matrices are as follows:

$$\frac{\partial L}{\partial \boldsymbol{M}_i} = \frac{\partial L}{\partial \tilde{\boldsymbol{W}}_{(i)}} \left(\boldsymbol{M}_{d_c} \otimes \boldsymbol{M}_{d_c-1} \otimes \cdots \otimes \boldsymbol{M}_{i+1} \otimes \boldsymbol{M}_{i-1} \otimes \cdots \otimes \boldsymbol{M}_1\right) \boldsymbol{C}_{(i)}^T, \tag{3}$$

$$\frac{\partial L}{\partial \boldsymbol{C}_{(i)}} = \boldsymbol{M}_i^T \frac{\partial L}{\partial \tilde{\boldsymbol{W}}_{(i)}} \left(\boldsymbol{M}_{d_c} \otimes \boldsymbol{M}_{d_c-1} \otimes \cdots \otimes \boldsymbol{M}_{i+1} \otimes \boldsymbol{M}_{i-1} \otimes \cdots \otimes \boldsymbol{M}_1\right), \tag{4}$$

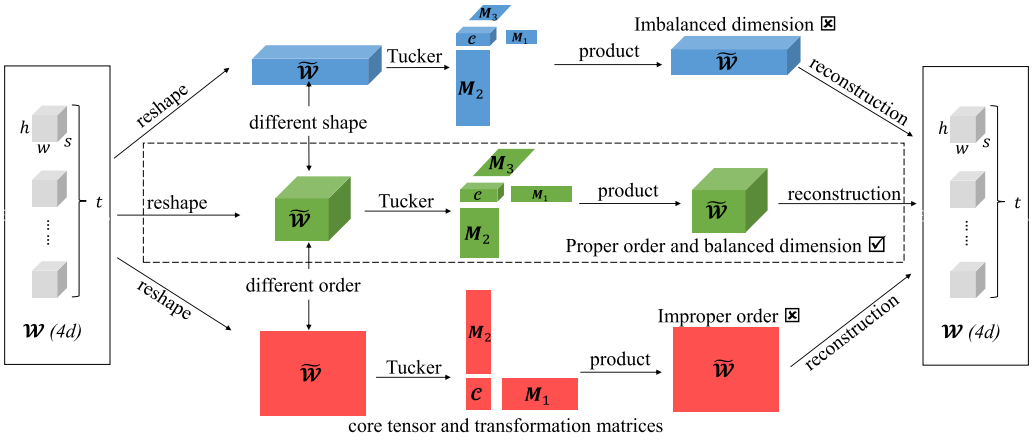

Fig. 2. Illustration of ADA-Tucker: For each layer, the order for Tucker decomposition depends on the dimensions of modes of the original tensor. For different layers, orders and dimensions of tensors can vary.

$$\frac{\partial L}{\partial \mathcal{C}} = \text{fold}\left(\frac{\partial L}{\partial \mathbf{C}_{(i)}}\right). \tag{5}$$

2.2. Adaptive dimension adjustment and motivation

The tendency of network overfitting suggests that there is always redundancy among the weights which can be approximately measured by 'rank'. And 'rank' is often determined by the smaller/smallest size of different modes (e.g., for a matrix, its rank cannot exceed its row number or column number, whichever smaller). If the size of a mode is much smaller than others, compressing along that mode will cause significant information loss.

Changing the dimension of the weight tensors to avoid the significant information loss in DNNs has been widely used in network compression. For example, Jaderberg et al. (2014) compress network with low-rank regularization. In their model, they merged the kernel height dimension and kernel width dimension into one dimension and got success, which suggests that there exist some information redundancy between the kernel width dimension and kernel height dimension. ThiNet Luo, Wu, and Lin (2017) proposed to compress the weights through the input channel dimension and output channel dimension, which suggests that there exist some information redundancy between the input channel dimension and output channel dimension. Zhang, Qi, Xiao, and Wang, (2017) proposed interleaved group convolutions that splitting the weight tensor into several small group tensor, which also means that there exist redundancy among the four dimensions. Here, we extend these ideas further. We treat all four dimensions of the weight tensor equally. So we reshape the weight tensor to any order and any shape. Here is a toy example that can illustrate this idea. Suppose that we have 100 parameters represented by a matrix of size 1×100 or 10×10. Obviously, the rank of the former matrix tends to be 1, in which case rank-based compression is hard (compressing to a zero-rank matrix will lose all information). In contrast, a matrix of real data in the latter form can be easily approximated with a lower-rank matrix without losing too much information.

As a conclusion, compression will be much less effective if a tensor is not reshaped to one with appropriate order and balanced dimension. Motivated by such consideration, we implement adaptive dimension adjustment in our model that allows reshaping weight tensors and defining core tensors of arbitrary shape. The illustrated explanation of ADA-Tucker is showing in Fig. 2. Experiments also demonstrate that both balanced dimensions of each mode and a proper order of the weight tensor contribute to better performance during network compression.

In the following subsection, we will describe the principle and process of adaptive dimension adjustment.

2.2.1. Adaptive dimension adjustment for Conv layers

For a Conv layer, the basic mechanism is to reshape the original weight tensor into a tensor with roughly even dimensions of modes. We take the Conv1 (first convolutional) layer of LeNet-5 as an example. The size of its original weight tensor is $5 \times 5 \times 1 \times 20$. Normally, a mode of dimension one is redundant and can be simply neglected (such case usually occurs in the first convolutional layer of a neural network). Note that dimensions of the first two modes are much smaller than that of the last one. With 20 still an acceptable dimension size, we merge the first two modes and get a second order tensor of size 25×20. We may then define a smaller second order core tensor accordingly for decomposition.

Generally speaking, when there are few input channels (e.g., for the first layer of a network, $s = 1$ or $s = 3$), we merge the input and output channels into one mode, obtaining a third order weight tensor $\tilde{\mathcal{W}} \in \mathbb{R}^{h \times w \times st}$. Similar operation is conducted for small kernel size (e.g., 1×1 or 5×5), i.e., merging the first two modes into one to have $\tilde{\mathcal{W}} \in \mathbb{R}^{hw \times s \times t}$. If these two cases occur simultaneously, we can reduce the original fourth order weight tensor to a matrix $\tilde{\mathcal{W}} \in \mathbb{R}^{hw \times st}$. With the same principle in mind, when the kernel size grows large enough, it is better to maintain the weight tensor as a fourth order tensor or even reshape it to one with a higher order (e.g., fifth order and sixth order). In fact, the dimension adjustment operations are *not limited to simply merging several modes*: any form of reshaping operation is valid as long as the number of weight stays the same, which, as far as we know, is not achieved by any previous low-rank models.

We designed experiments about adaptive dimension adjustment of Conv1 and Conv2 layers in LeNet5. We conducted these experiments by changing the order of the weight tensor of Conv1/Conv2 layer while fixing the rest. The details of the Conv1/Conv2's weight tensor with different orders are listed in Tables 4 and 5 of the appendices part. We chose proper core tensor sizes for each order to ensure the numbers of parameters under different settings are similar. The network performances under different settings are showing in Fig. 4. From Fig. 4, the optimal order for Conv1 and Conv2 layers in LeNet-5 is five and three, respectively. Here is one more important thing to mention, the gray and yellow bars mean we did not use adaptive dimension adjustment on these two settings. The original order for Conv layer's weight is four, so our ADA-Tucker degenerates to Tucker under these two settings. From the results, if we reshape the tensor with proper order and balanced dimensions before Tucker decomposition, we can get better compressed results.

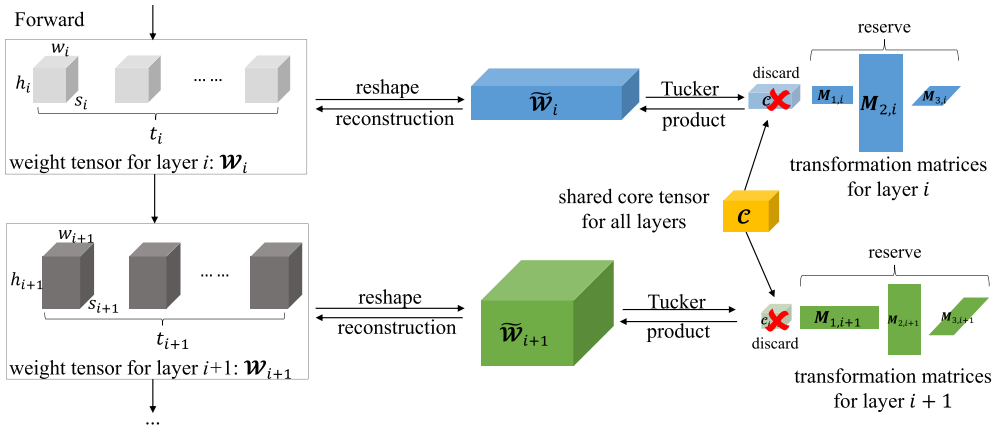

Fig. 3. Illustration of SCADA-Tucker: If all layers share the same core tensor, i.e., $\forall i \in 1, 2, \ldots, l$, $\mathcal{C}^{(i)} = \mathcal{C}$, ADA-Tucker becomes SCADA-Tucker.

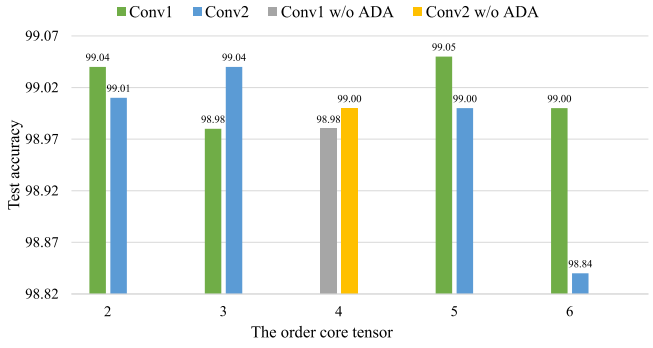

Fig. 4. Dimension adjustment of Conv layers. Bars of the same color represent experiments with similar numbers of weight, conducted by changing the order of the weight tensor of a specific layer while fixing the rest. The optimal order for Conv1 and Conv2 layers in LeNet-5 is five and three, respectively (better viewed together with Tables 4 and 5). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

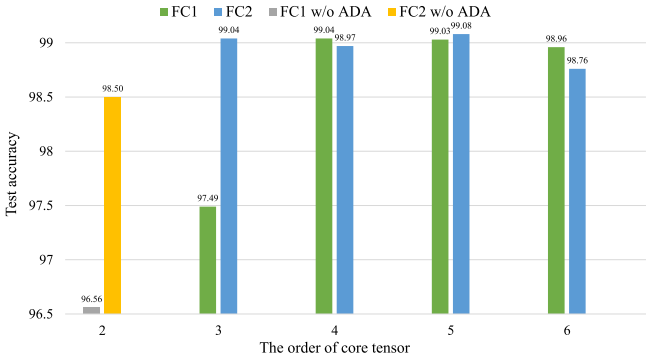

Fig. 5. Dimension adjustment of FC layers. Bars with same color represent experiments with similar numbers of weight, conducted by changing the order of the weight tensor of a specific layer while fixing the rest. A fourth order weight tensor is optimal for FC1 layer while a fifth order weight tensor is superior to others for FC2 layer (better viewed together with Tables 6 and 7). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

2.2.2. Adaptive dimension adjustment for FC layers

Our dimension adjustment mechanism also applies to FC layers. Here we present an example involving a fifth order reshaped tensor for FC2 (second fully-connected) layer of LeNet-5, which is originally a matrix of size 500×10. To balance the dimensions of modes, we reshape the original weight tensor to a size of $5 \times 5 \times 5 \times 5 \times 8$. Note that such operation does not necessarily indicate splitting individual mode, the decomposition is allowed to disrupt the original sequence of dimension size (e.g., 8 is a factor of neither 500 nor 10). With the weight tensor reshaped as a fifth order tensor according to our adaptive dimension adjustment principle, the network finds its best structure for the FC2 layer. To our knowledge, previous tensor decomposition methods can only regard FC layer's weights as a second order tensor with fixed dimensions of modes.

We also conducted experiments about dimension adjustment of FC1 and FC2 layers in LeNet5. We conducted this experiments by changing the order of the weight tensor of FC1/FC2 layer while fixing the rest. The details of the FC1/FC2's weight tensor with different orders are listed in the appendices part Tables 6 and 7. The network performances under different settings are showing in Fig. 5. From Fig. 5, a fourth order weight tensor is optimal for FC1 layer while a fifth order weight tensor is superior to others for FC2 layer. The original orders for FC1 and FC2 are both two. Same as previous analysis, the gray and yellow bars mean we did not use adaptive dimension adjustment on these two settings. Our ADA-tucker also got better results than Tucker without ADA on FC layers.

In summary, the optimal order of weight tensor varies for different layers. The results indicate that previous low-rank compression methods may impair network performance by constraining a fixed form of decomposition. The flexibility of ADA-Tucker enables networks to adaptively adjust the dimensions and orders of weight tensor when being compressed, thus achieving better performance.

The ADA-Tucker algorithm is summarized in Algorithm 1.

2.2.3. Influence of dimension eveness of core tensor

To explore the influence of dimension eveness of core tensor, we change the shape of core tensor and record accuracy of each network after training them from scratch. The details of this experiment settings please refer to Table 8 in appendices part. As is shown in Fig. 6, the network with square core tensors performs better than the one with other core shapes. Specifically, as the difference between the two dimensions grows larger, i.e., when the core becomes less 'square', the network's accuracy decreases accordingly. We speculate that the mechanism behind is to evenly distribute weights across different dimensions.

Here we give a more clear summary for the adaptive dimension adjustment mechanism based on the experiments on dimension adjustment of Conv layers, dimension adjustment of FC layers and influence of dimension eveness of core tensor: The mechanism is somewhat like factorization of the number of weights for a specific

Algorithm 1 ADA-Tucker Algorithm

Input: $\mathbf{X}$, $\mathbf{Y}$: training data and labels.
Input: $\{n_1^{(i)}, n_2^{(i)}, ..., n_{d_c}^{(i)} : 1 \le i \le l\}$: $n_{d_c}^{(i)}$ is the dimension of d_c mode of the ith layer's reshaped weight tensor, which is denoted by adaptive dimension adjustment mechanism.
Output: $\{\mathcal{C}^{(i)}, \mathbf{M}_1^{(i)}, \mathbf{M}_2^{(i)}, ..., \mathbf{M}_{i_{d_c}}^{(i)} : 1 \le i \le l\}$: the core tensors and transformation matrices for every layer.
Adaptive dimension adjustment: based on the input $\{n_1^{(i)}, n_2^{(i)}, ..., n_{d_c}^{(i)} : 1 \le i \le l\}$, construct $\tilde{\mathcal{W}}^{(i)}$ from $\mathcal{W}^{(i)}$, define $\mathcal{C}^{(i)}$ and $\mathbf{M}_j^{(i)}$, $1 \le i \le l$, $1 \le j \le i_{d_c}$.
for number of training iterations **do**
  Choose a minibatch of network input from $\mathbf{X}$.
  for $i = 1, 2, 3, ..., l$ **do**
    Use Eq. (1) and reshape function to rebuild $\mathcal{W}^{(i)}$, use $\mathcal{W}^{(i)}$ to get the output of the ith layer.
  end for
  Compute the loss function L.
  for $i = l, l-1, l-2, ..., 1$ **do**
    Follow traditional backward propagation to get $\frac{\partial L}{\partial \mathcal{W}^{(i)}}$ and compute $\frac{\partial L}{\partial \tilde{\mathcal{W}}^{(i)}}$ from $\frac{\partial L}{\partial \mathcal{W}^{(i)}}$.
    for $j = 1, 2, 3, ..., i_{d_c}$ **do**
      Use Eq. (3) to compute $\frac{\partial L}{\partial \mathbf{M}_j^{(i)}}$, then update $\mathbf{M}_j^{(i)}$.
    end for
    Use Eq. (4) to compute $\frac{\partial L}{\partial \mathcal{C}_{(1)}^{(i)}}$, use Eq. (5) to construct $\frac{\partial L}{\partial \mathcal{C}^{(i)}}$, then update $\mathcal{C}^{(i)}$.
  end for
end for

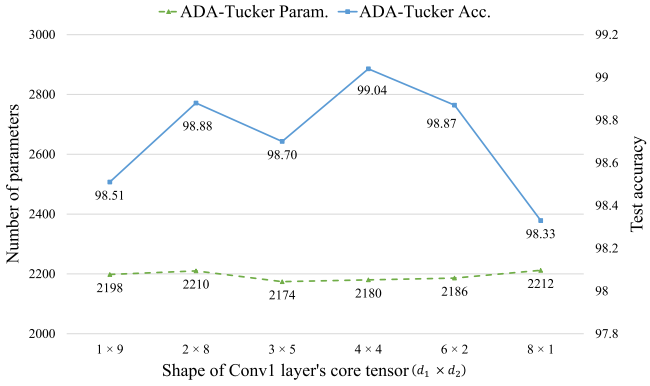

Fig. 6. Influence of dimension eveness of core tensor. Experiments are conducted by gradually increasing the aspect ratio of the second order core tensor of Conv1 (first convolutional) layer (25×20) in LeNet-5 while fixing the rest of the network architecture. We control the variation of each model's weight number within a negligible range and test their classification accuracy on MNIST with aspect ratio ranging from 1:9 to 8:1 (better viewed together with Table 8).

layer. The number of factors is equal to the order after the adaptive dimension adjustment mechanism. From the experiments of dimension adjustment of Conv layers (Fig. 4 for performances of all settings. Table 4 for the detail of Conv1's weight tensor and Table 5 for the detail of Conv2's weight tensor) and FC layers (Fig. 5 for performances of all settings. Table 6 for the detail of FC1's weight tensor and Table 7 for the detail of FC2's weight tensor), we found that if we make factors' values more similar with each other (balanced dimensions), the performance is better. The factors' value should not be too big, otherwise it will cost vast storage for transformation matrices and lose much information (Performance degradation in FC1 layer when the order of core tensor is two). From the experiments of Influence of dimension eveness of core tensor, we learn that if the reshape tensor is balanced with proper order, the core tensor with hypercube shape will have the best performance.

2.3. CP is a special case of Tucker and Tucker is a special case of ADA-Tucker

Suppose now we have a d-dimensional tensor $\mathcal{W}$ of size $n_1 \times n_2 \times \cdots \times n_d$ and a core tensor $\mathcal{C}$ of size $k_1 \times k_2 \times \cdots \times k_d$, Tucker decomposition has the following form:

$$\begin{aligned}\mathcal{W} &\approx \mathcal{C} \times_1 \mathbf{M}_1 \times_2 \mathbf{M}_2 \times_3 ... \times_d \mathbf{M}_d \\ &= \sum_{i_1=1}^{k_1}\sum_{i_2=1}^{k_2}\cdots\sum_{i_d=1}^{k_d} \mathcal{C}_{i_1 i_2 ... i_d} \boldsymbol{m}_{i_1}^1 \otimes \boldsymbol{m}_{i_2}^2 \otimes \cdots \otimes \boldsymbol{m}_{i_d}^d,\end{aligned} \tag{6}$$

where m_i^j means the ith column of matrix $\boldsymbol{M}_j$. While CP-decomposition has the following form:

$$\mathcal{W} \approx \sum_{i=1}^{r} \lambda_i \boldsymbol{m}_i^1 \otimes \boldsymbol{m}_i^2 \otimes \cdots \otimes \boldsymbol{m}_i^d. \tag{7}$$

In the case of the core tensor being a hypercube, if its elements are nonzero when $i_j = i, \forall j \in \{1, 2, 3..., d\}$ and are zero otherwise, then Tucker degenerates to CP. The fact that CP is a special case of Tucker indicates that Tucker is more powerful than CP. In fact, Tucker encodes much more compact representation as its core tensor is denser and smaller-sized, using the mechanism of Adaptive Dimension Adjustment. It is obvious to learn that ADA-Tucker degenerates to Tucker without using the mechanism of Adaptive Dimension Adjustment. Empirically, the following experimental evidence is provided for detailed comparisons for these three methods.

2.4. Shared core ADA-Tucker

With input data passing serially through each layer in a network, we believe that there exist some correspondence and invariance in weights across different layers. Concretely, we think that a weight tensor preserves two kinds of information, namely, the first kind of information tries to construct some transformations to extract global features (encode the same object at different layers and different scales) and the second kind of information tries to construct some transformations to extract local specific features for different layers and different scales. This conjecture indicates that there may exist shared information for transformation in functions expressed as a d_c-mode product between core tensors and transformation matrices across layers. We assume that the layer-invariant information lies in the core tensor, as it has the majority of weights, while the transformation matrices are responsible for the layer-specific mapping. Therefore, as illustrated in Fig. 3, we devise a so-called SCADA-Tucker, where all layers of a network share one common core tensor, thus achieving higher compression ratio.

Suppose that the network has l layers. Based on the description above, we need one core tensor and $\sum_{i=1}^{l} d_i$ transformation matrices, where d_i represents the order of core tensor for the ith layer. With SCADA-Tucker, the model contains l core tensors and ld transformation matrices ($d = \max\{d_i\}, i = 1, 2, \ldots, l$). We can set a specific transformation matrix $\mathbf{M}_j^{(i)} \in \mathbb{R}^{1 \times k_j^{(i)}} (j = 1, 2, \ldots, d)$ if the reshaped weight tensor of the ith layer has a lower order than the shared core tensor. The forward propagation is:

$$\begin{aligned}\tilde{\mathcal{W}}^{(i)} &\approx \mathcal{C} \times_1 \mathbf{M}_1^{(i)} \times_2 \mathbf{M}_2^{(i)} \times_3 ... \times_d \mathbf{M}_d^{(i)}, \\ \mathcal{W}^{(i)} &= \text{reshape}\left(\tilde{\mathcal{W}}^{(i)}\right).\end{aligned} \tag{8}$$

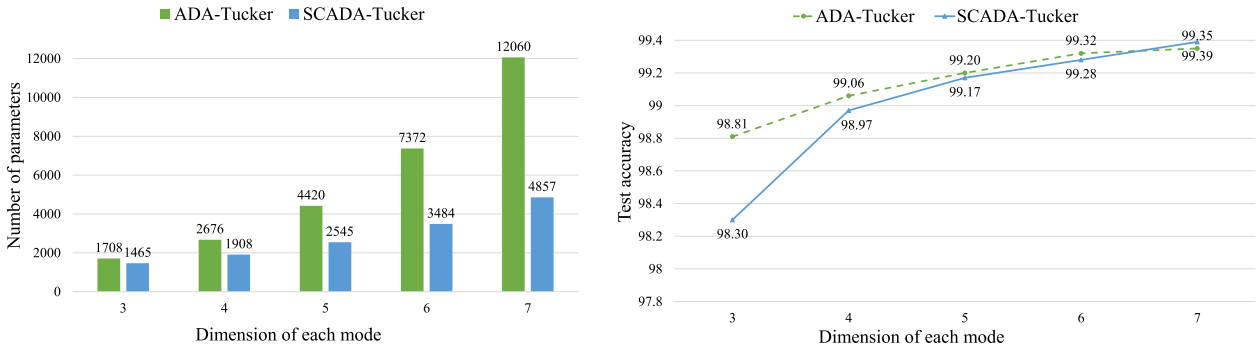

Fig. 7. SCADA-Tucker vs. ADA-Tucker comparing the number of weights and network's performance (better viewed together with Table 9).

The backpropagation of *ld* transformation matrices is the same as that in the ADA-Tucker model. The major difference lies in the gradient w.r.t. the core tensor. We compute it as:

$$\frac{\partial L}{\partial \mathcal{C}} = \sum_{i=1}^{l} \frac{\partial L}{\partial \mathcal{C}^{(i)}}. \tag{9}$$

We present here a detailed property comparison between SCADA-Tucker and ADA-Tucker. We use LeNet-5 and set the core tensor of each layer as a fourth order tensor of the same size. We examine the performance of LeNet-5 using two compression methods by changing the size of core tensor only while fixing the rest hyper-parameters. The details of ADA-Tucker and SCADA-Tucker settings for these experiments are in Table 9 of appendices part. From the results in Fig. 7, we can see that under the same parameter setting, SCADA-Tucker is able to significantly reduce the number of weight in the network with only minor performance degradation compared to ADA-Tucker. It is because core tensors generally account for a major proportion of the total number of weights. When the dimension of each mode increases to seven, SCADA-Tucker even achieves an accuracy slightly higher than that of ADA-Tucker. Note that the number of weight in SCADA-Tucker is less than one half of that in ADA-Tucker under the same setting.

An alternative perspective is to view SCADA-Tucker as a module with properties analogous to recurrent neural networks (RNN). The comparison of forward propagations for these two models can be seen in Fig. 8. We all know that an RNN can be rolled out as a serial network with shared weights and it captures the temporal relations among sequential inputs. Thus with part of weights shared across layers, SCADA-Tucker can be regarded as an architecture in a recurrent style. Concretely, the forward propagation of RNN can be represented as $\boldsymbol{h}^{(t)} = \sigma(\boldsymbol{U}\boldsymbol{x}^{(t)} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{b})$, where $\boldsymbol{x}^{(t)}$ is the input at time step t, $\boldsymbol{h}^{(t)}$ is the hidden state at time step t, $\boldsymbol{b}$ is the bias term, $\sigma(\cdot)$ is the activation function and $\boldsymbol{U}, \boldsymbol{W}$ are the transformation matrices shared by all time steps. In comparison, the forward propagation of SCADA-Tucker can be represented as

$$\boldsymbol{h}^{(i)} = \sigma(\mathcal{C} \prod_{j} \times_j \boldsymbol{M}_j^{(i)} \boldsymbol{h}^{(i-1)} + \boldsymbol{b}), i = 1, 2, 3, \ldots, l. \tag{10}$$

The "input" of each layer is a series of transformation matrices $\{\boldsymbol{M}\}$ and the core tensor $\mathcal{C}$ is the share parameter for all layers.

Moreover, SCADA-Tucker creatively connects weights of different layers, enabling direct gradient flow from the loss function to earlier layers. Some modern architectures of neural networks such as ResNet (He et al., 2016), DenseNet (Huang et al., 2017) and CliqueNet (Yang, Zhong, Shen, & Lin, 2018) benefit from the direct gradient flow from the loss function to earlier layers and have achieved great success. *Such a parameter reuse mechanism also addresses redundancy in common parameterized functions shared across layers. Note that none of other compression methods involve dimension adjustment and sharing parameters among different layers*, which are crucial to the compression performance.

Therefore, SCADA-Tucker has the potential to pass and share some critical and common information across layers, which cannot be achieved by ADA-Tucker. Finally, we regard SCADA-Tucker as a promising compression method for high-ratio network compression with a negligible sacrifice in network performance.

3. Compression ratio analysis

3.1. Raw compression ratio analysis

Suppose that the network has l layers. Let $\tilde{\mathcal{W}}^{(i)} \in \mathbb{R}^{n_1^{(i)} \times n_2^{(i)} \cdots \times n_{d_i}^{(i)}}$ and $\mathcal{C}^{(i)} \in \mathbb{R}^{k_1^{(i)} \times k_2^{(i)} \cdots \times k_{d_i}^{(i)}}$ be the reshaped weight tensor and core tensor of the ith layer, respectively. Obviously, $\tilde{\mathcal{W}}^{(i)}$ has the same number of weights as $\mathcal{W}^{(i)}$. Then the compression ratio of ADA-Tucker is:

$$\begin{aligned} r_A &= \frac{\sum_{i=1}^{l} \prod_{j=1}^{d_i} n_j^{(i)}}{\sum_{i=1}^{l} \prod_{j=1}^{d_i} k_j^{(i)} + \sum_{i=1}^{l} \sum_{j=1}^{d_i} n_j^{(i)} k_j^{(i)}} \\ &\approx \frac{\sum_{i=1}^{l} \prod_{j=1}^{d_i} n_j^{(i)}}{\sum_{i=1}^{l} \prod_{j=1}^{d_i} k_j^{(i)}}. \end{aligned} \tag{11}$$

For SCADA-Tucker, all layers share the same core tensor with order d, i.e, $d = d_i$, $i = 1, 2, \ldots, l$. Then its compression ratio is:

$$\begin{aligned} r_{SC} &= \frac{\sum_{i=1}^{l} \prod_{j=1}^{d} n_j^{(i)}}{\prod_{j=1}^{d} k_j + \sum_{i=1}^{l} \sum_{j=1}^{d} n_j^{(i)} k_j} \\ &\approx \frac{\sum_{i=1}^{l} \prod_{j=1}^{d} n_j^{(i)}}{\prod_{j=1}^{d} k_j} \geqslant r_A. \end{aligned} \tag{12}$$

3.2. Further compression by quantization

After being compressed by ADA-Tucker and SCADA-Tucker, the weight distribution is close to Laplacian distribution. Since almost all weights are in the range of $[-3, 3]$ (Fig. 9), we can use pruning and quantization to compress these weights further following Han et al. (2015). In our experiments, we integrate the following quantization into our model:

$$w_q = -b + \left\lfloor \frac{(\max(-b, \min(w, b)) + b)Q}{2b} \right\rfloor \cdot \frac{2b}{Q}, \tag{13}$$

where Q represents the number of clusters, b represents the max–min bound of quantization and $\lfloor x \rfloor$ is the floor function. Since weights are originally stored in the float32 format (32 bits), our compression ratio can be further increased by this quantization trick. After quantization, we utilize Huffman coding to compress

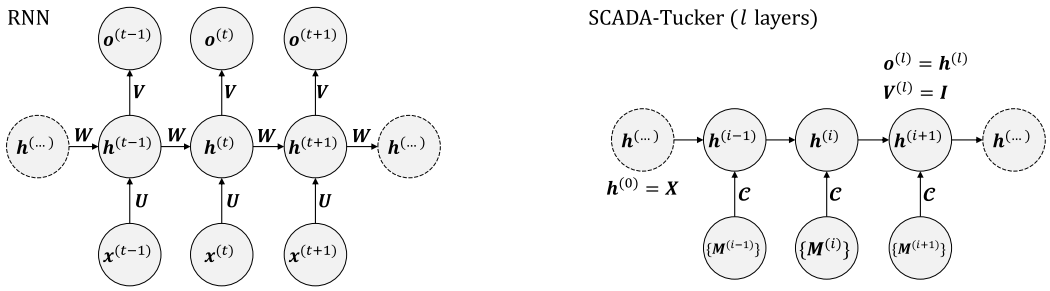

Fig. 8. The comparison of forward propagations for RNN (left) and SCADA-Tucker (right).

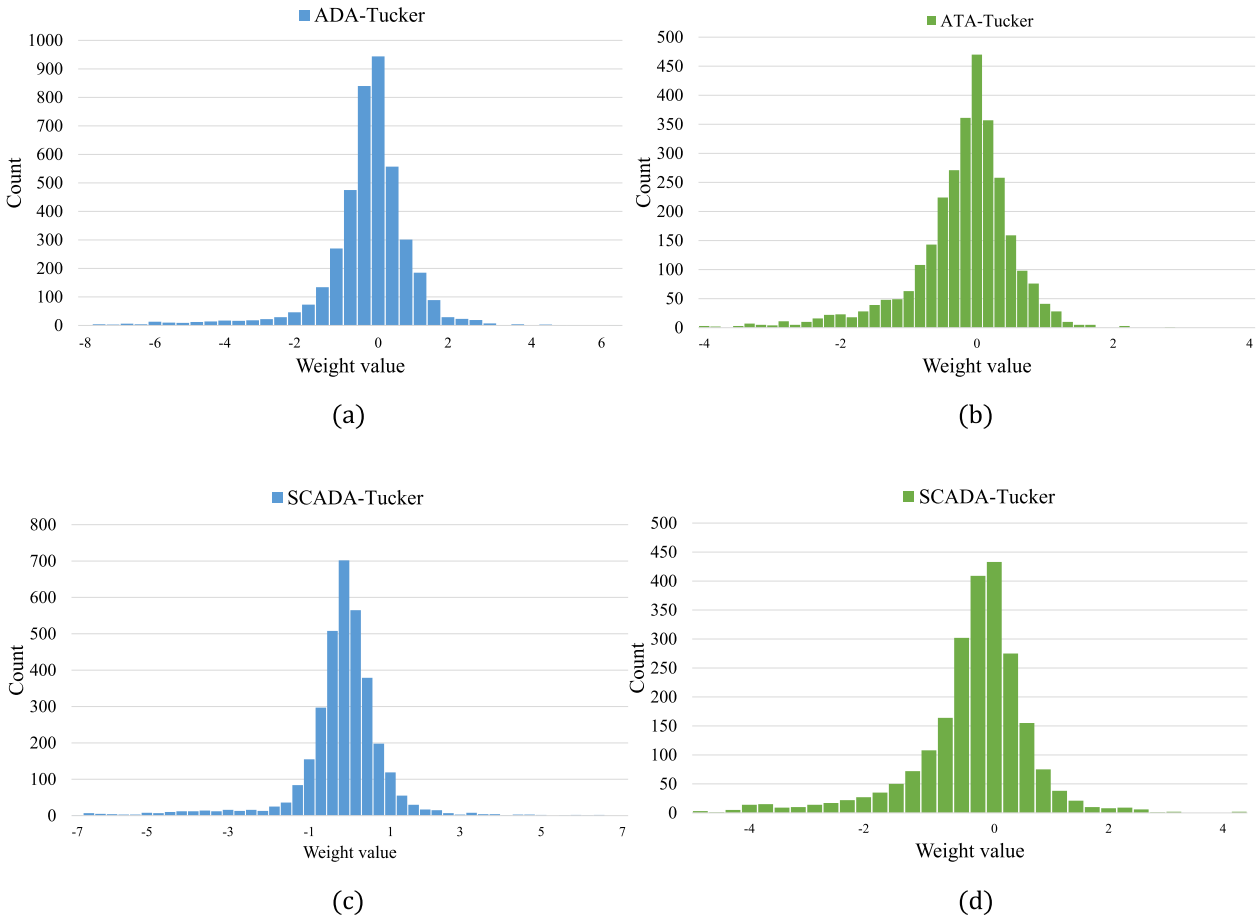

Fig. 9. Weight distribution after compressing by our methods: (a) ADA-Tucker on LeNet-300. (b) ADA-Tucker on LeNet-5. (c) SCADA-Tucker on LeNet-300. (d) SCADA-Tucker on LeNet-5.

it further. Suppose that the average length of Huffman coding is $\bar{a}$, we compute the final compression ratio by:

$$r_{A+QH} = \frac{32r_A}{\bar{a}}, \quad r_{SC+QH} = \frac{32r_{SC}}{\bar{a}}. \tag{14}$$

4. Experimental results

In this section, we experimentally analyze the proposed methods. We use Adam (Kingma & Ba, 2015) as our optimizer. The starting learning rate is set as 0.01, 0.003, 0.003 and 0.01 for MNIST, CIFAR-10, SVHN and ImageNet, respectively. After every 10∼20 epochs, the learning rate is divided by 3. We choose 256 as the batch size for most experiments. For initializers of core tensors and transformation matrices, we have experimented with the Glorot initializer (Glorot & Bengio, 2010), the Kaiming initializer (He, Zhang, Ren, & Sun, 2015) and HOOI (De Lathauwer, De Moor, & Vandewalle, 2000) to solve the decomposition from the original weight tensors. These three methods have similar performances. Note that for the following experiments, little time is spent on fine-tuning our model.

4.1. MNIST

MNIST is a database of handwritten digits with 60,000 training images and 10,000 testing images. It is widely used to evaluate machine learning algorithms. Same as DC (Han et al., 2016), DNS (Guo et al., 2016) and SWS (Ullrich et al., 2017), we test our methods on two classical networks: LeNet-5 and LeNet-300-100.

The raw compression ratios of ADA-Tucker and SCADA-Tucker in Table 1 are computed by Eq. (11) and Eq. (12), respectively. The compression ratio of +QH is computed by Eq. (14). Because

Table 1
Compression results on LeNet-5 and LeNet-300-100. +QH: adding quantization and Huffman coding after utilizing these methods. #(Param.) means the total number of parameters of these methods. CR represents compression ratio.

Network	Methods	#(Param.)	Test Error Rate [%]			CR	
			Org.	Raw	+QH	Raw	+QH
LeNet-300-100	DC (Han et al., 2016)	21.4K	1.64	1.57	1.58	<12	40
	DNS (Guo et al., 2016)	4.8K	2.28	1.99	–	<56	–
	SWS (Ullrich et al., 2017)	4.3K	1.89	–	1.94	<62	64
	ADA-Tucker	4.1K	1.89	1.88	1.91	=65	**233**
	SCADA-Tucker	3.4K	1.89	2.11	2.26	=78	**321**
LeNet-5	DC (Han et al., 2016)	34.5K	0.80	0.77	0.74	<13	39
	DNS (Guo et al., 2016)	4.0K	0.91	0.91	–	<108	–
	SWS (Ullrich et al., 2017)	2.2K	0.88	–	0.97	<196	162
	ADA-Tucker	2.6K	0.88	0.84	0.94	=166	**691**
	SCADA-Tucker	2.3K	0.88	0.94	1.18	=185	**757**

Table 2
Test error rates (in %) with compression ratio at 16× and 64× for LRD (Denil et al., 2013), HashedNet (Chen et al., 2015), FreshNet (Chen et al., 2016) and ours. CR represents compression ratio.

Dataset	CNN-ref	LRD		HashedNet		FreshNet		ADA-Tucker	SCADA-Tucker
	CR	16×	64×	16×	64×	16×	64×	64×	73×
CIFAR-10	14.37	23.23	34.35	24.70	43.08	21.42	30.79	**17.97**	**20.27**
SVHN	3.69	10.67	22.32	9.00	23.31	8.01	18.37	**4.41**	**3.92**

all these methods (Guo et al., 2016; Han et al., 2015; Ullrich et al., 2017) in Table 1[1] need to record the indices of nonzero elements, their actual compression ratios are smaller than the calculated results. *Our methods do not need to record the indices*, so our actual compression ratios are equal to the calculated results, suggesting that our model has the highest compression ratio even if it has the same number of weight with the methods mentioned above. We set $Q = 512$ and $b = 3$ during the quantization process of LeNet-5 and get the final compression ratio of **691**× with 0.94% error rate. For LeNet-300-100, we set $Q = 1500$ and $b = 5$ to achieve a **233**× compression ratio and the final error rate is 1.91%. The value of b can be adjusted according to the distribution of weights after ADA-Tucker/SCADA-Tucker compression.

Tensor Train decomposition (TT) (Novikov et al., 2015) is similar to Tucker decomposition in that they both involve a product of matrices. With Tucker, the number of parameters can be further reduced by sharing core tensor, which cannot be achieved by TT. Moreover, we chose Tucker because TT has to use enormously-sized matrices to exactly represent a tensor when its order is greater than three. Thus compressing with TT significantly may cause huge approximation error. Using Tucker helps strike a better balance between compression ratio and recognition accuracy. More importantly, TT can only be applied to FC layers despite the fact that Conv layers are more crucial than FC layers to achieve top performance for most of the current deep learning tasks. In contrast, our model with Tucker decomposition is able to adjust the order and dimension of tensors in *both FC and Conv layers*. Still, for a closer examination, here we provide results of two models in all-FC-layer network for better reference. For MNIST we use the same network architecture as Novikov et al. (2015) and get 98.13% test accuracy with 6824 parameters, while TT gets 98.1% test accuracy with 7698 parameters. This again proves the strength of our methods in compressing network and preserving information.

[1] We compare our models with state of the art in 2015, 2016 and 2017 for compressing LeNet-5 and LeNet-300. HashedNet (Chen, Wilson, Tyree, Weinberger, & Chen, 2015) does not appear in Table 1 because it used a network different from LeNet-5 or LeNet-300 and thus cannot be compared with other methods. Since methods in Table 1 conducted experiments on MNIST but not on CIFAR-10 or SVHN, these methods are not shown in Table 2.

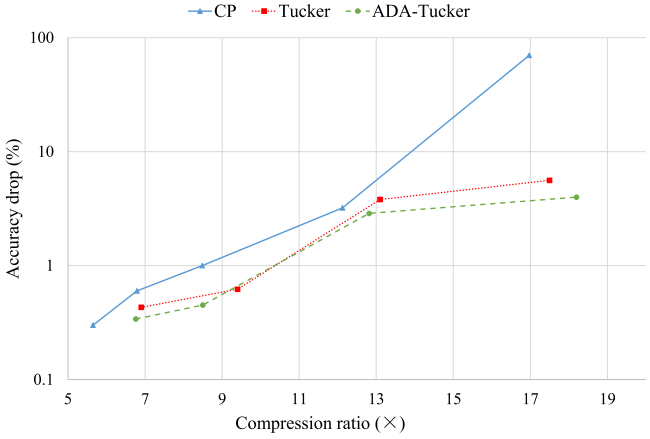

Fig. 10. Comparison of ADA-Tucker, Tucker-decomposition and CP-decomposition on ImageNet. (Logarithmic coordinates.)

4.2. SVHN and CIFAR-10

To prove the generalization ability of our methods, we also conduct experiments on SVHN and CIFAR-10 datasets. The SVHN dataset is a large collection of digits cropped from real-world scenes, consisting of 604,388 training images and 26,032 testing images. The CIFAR-10 dataset contains 60,000 images of 32 × 32 pixels with three color channels. With the same network architectures, our compressed models significantly outperform (Chen et al., 2015, 2016; Denil et al., 2013) in terms of both compression ratio and classification accuracy. The details of network architecture are listed in Table 10 and the setting for ADA-Tucker is shown in Table 11 of appendices part. On the CIFAR-10 dataset, ADA-Tucker has a higher accuracy with a compression ratio lower than SCADA-Tucker as expected. However, on the SVHN dataset, SCADA-Tucker surprisingly preforms much better than ADA-Tucker. Specifically, SCADA-Tucker compresses the original network by 73× with 0.23% accuracy drop, while ADA-Tucker compresses it by 64× with 0.72% accuracy drop.

Table 3
Compression results on ResNet-20 and WRN-28-10 on CIFAR-10 dataset. #(Param.) means the total number of parameters of these methods. CR represents compression ratio.

Network	#(Param.)	Orig. Acc. (%)	ADA-Tucker Acc. (%)	Δ(Acc.)	CR
ResNet-20	0.27M	91.25%	90.97%	−0.28%	**12**
WRN-28–10	36.5M	95.83%	95.06%	−0.77%	**58**

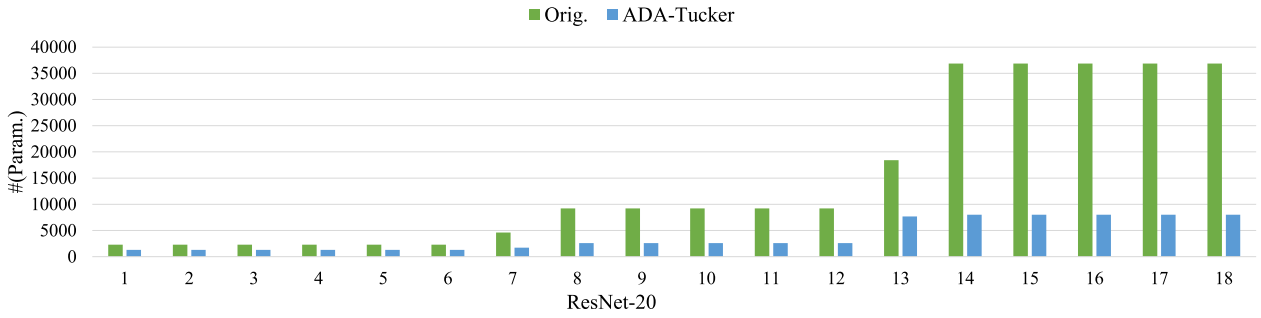

Fig. 11. Parameters comparison of all convolutional layers in ResNet-20 (better viewed together with Table 12).

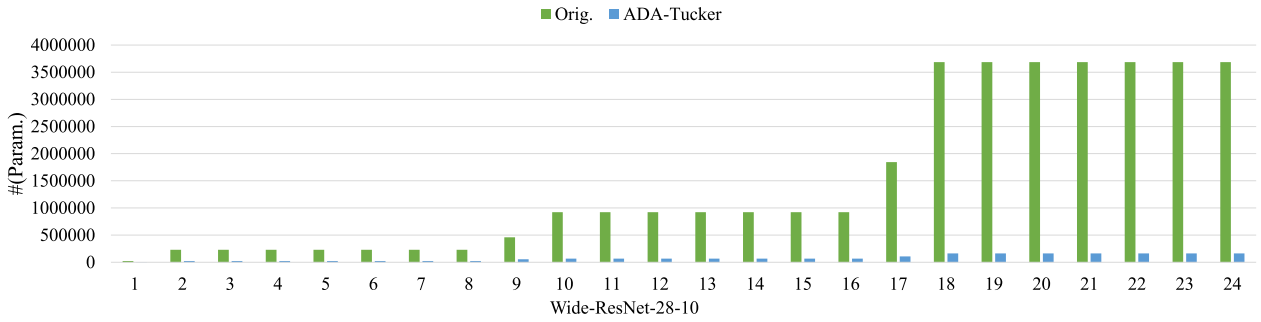

Fig. 12. Parameters comparison of all convolutional layers in Wide-ResNet-28-10 (better viewed together with Table 13).

Table 4
The details of Conv1 layer's dimension adjustment experiment setting (In Section 2.2.1). In this experiment, we fixed the rest layers of LeNet-5.

Conv1	Orig.	2d	3d	4d	5d	6d
Shape	$20 \times 1 \times 5 \times 5$	25×20	$20 \times 5 \times 5$	$20 \times 1 \times 5 \times 5$	$5 \times 5 \times 5 \times 2 \times 2$	$5 \times 5 \times 5 \times 2 \times 2 \times 1$
Core	–	5×5	$4 \times 4 \times 4$	$3 \times 3 \times 3 \times 3$	$2 \times 2 \times 2 \times 2 \times 2$	$2 \times 2 \times 2 \times 2 \times 2 \times 2$
Conv2	Orig.: $50 \times 20 \times 5 \times 5$, reshape: $50 \times 25 \times 20$, core size: $5 \times 5 \times 5$					
FC1	Orig.: 800×500, reshape: $40 \times 25 \times 20 \times 20$, core size: $5 \times 5 \times 5 \times 5$					
FC2	Orig.: 500×10, reshape: $25 \times 20 \times 10$, core size: $5 \times 5 \times 5$					
#(Param.)	3230	2980	2914	2904	2810	2834

4.3. ILSVRC12

In this subsection, we empirically compare the performances of CP, Tucker and ADA-Tucker on ILSVRC12 dataset.

To prove this work preserves more information and easier compress networks compared with CP-decomposition and Tucker-decomposition, we follow the ILSVRC12 experiment in Lebedev et al. (2015). We also compress the second convolutional layer of AlexNet (Krizhevsky et al., 2012). As a baseline, we use a pre-trained AlexNet model shipped with Pytorch, which achieves a top-5 accuracy of 79.59%. Following (Lebedev et al., 2015), models are evaluated by test accuracy drop when increasing compression ratio. Experimental results in Fig. 10 show that our methods have less accuracy drop at the same compression ratio. The gap between our method and CP-decomposition becomes larger as the compression ratio goes higher. The same experimental phenomenon also appeared when we compared our method with Tucker-decomposition. Concretely, at the same compression ratio equal to 18, the accuracy drop of our method is less than 4%, while the CP-decomposition method drops about 70% and Tucker-decomposition method drops about 6%. This result suggests that our method has a better capacity to preserve more information than CP and easier compress networks than Tucker.

4.4. Modern networks

Here we discuss a more recent work, ResNet (He et al., 2016) and its variations Wide-ResNet (Xie, Girshick, Dollár, Tu, & He, 2017; Zagoruyko & Komodakis, 2016). ResNet and its variations also have achieved promising performances in numerous computer vision applications such as image classification, human face verification, object recognition, and object detection. It is very meaningful to be able to effectively compress these networks.

We applied our ADA-Tucker on two representative networks ResNet-20 and Wide-ResNet-28-10 (WRN-28-10). This experiment was done on CIFAR-10 dataset. The details of ADA-Tucker for ResNet-20 and WRN-28-10 can be found in Tables 12 and 13 of appendices part, respectively. The compression results are listed in Table 3. From Table 3, ADA-tucker compressed ResNet-20 by

Table 5
The details of Conv2 layer's dimension adjustment experiment setting (In Section 2.2.1). In this experiment, we fixed the rest layers of LeNet-5.

Conv1	Orig.: $20 \times 1 \times 5 \times 5$, reshape: 25×20, core size: 5×5					
Conv2	Orig.	2d	3d	4d	5d	6d
Shape	$50 \times 20 \times 5 \times 5$	250×100	$50 \times 25 \times 20$	$50 \times 20 \times 5 \times 5$	$10 \times 10 \times 10 \times 5 \times 5$	$10 \times 10 \times 5 \times 5 \times 5 \times 2$
Core	–	20×10	$5 \times 5 \times 5$	$4 \times 4 \times 4 \times 4$	$3 \times 3 \times 3 \times 3 \times 3$	$2 \times 2 \times 2 \times 2 \times 2 \times 2$
FC1 FC2	Orig.: 800×500, reshape: $40 \times 25 \times 20 \times 20$, core size: $5 \times 5 \times 5 \times 5$ Orig.: 500×10, reshape: $25 \times 20 \times 10$, core size: $5 \times 5 \times 5$					
#(Param.)	27380	8580	2980	2956	2743	2518

Table 6
The details of FC1 layer's dimension adjustment experiment setting (In Section 2.2.2). In this experiment, we fixed the rest layers of LeNet-5.

Conv1 Conv2	Orig.: $20 \times 1 \times 5 \times 5$, reshape: 25×20, core size: 5×5 Orig.: $50 \times 20 \times 5 \times 5$, reshape: $50 \times 25 \times 20$, core size: $5 \times 5 \times 5$					
FC1	Orig.	2d	3d	4d	5d	6d
Shape	800×500	800×500	$500 \times 25 \times 20$	$40 \times 25 \times 20 \times 20$	$25 \times 16 \times 10 \times 10 \times 10$	$25 \times 10 \times 8 \times 8 \times 5 \times 5$
Core	–	10×10	$5 \times 5 \times 5$	$5 \times 5 \times 5 \times 5$	$4 \times 4 \times 4 \times 4 \times 4$	$3 \times 3 \times 3 \times 3 \times 3$
FC2	Orig.: 500×10, reshape: $25 \times 20 \times 10$, core size: $5 \times 5 \times 5$					
#(Param.)	402K	14930	4680	2980	3138	2742

Table 7
The details of FC2 layer's dimension adjustment experiment setting (In Section 2.2.2). In this experiment, we fixed the rest layers of LeNet-5.

Conv1 Conv2 FC1	Orig.: $20 \times 1 \times 5 \times 5$, reshape: 25×20, core size: 5×5 Orig.: $50 \times 20 \times 5 \times 5$, reshape: $50 \times 25 \times 20$, core size: $5 \times 5 \times 5$ Orig.: 500×10, reshape: $25 \times 20 \times 10$, core size: $5 \times 5 \times 5$					
FC2	Orig.	2d	3d	4d	5d	6d
Shape	500×10	500×10	$25 \times 20 \times 10$	$20 \times 10 \times 10 \times 5$	$8 \times 5 \times 5 \times 5 \times 5$	$5 \times 5 \times 5 \times 5 \times 4 \times 2$
Core	–	6×6	$5 \times 5 \times 5$	$4 \times 4 \times 4 \times 4$	$3 \times 3 \times 3 \times 3 \times 3$	$2 \times 2 \times 2 \times 2 \times 2 \times 2$
#(Param.)	7580	5676	2980	3016	2907	2700

Table 8
The details of Conv1 layer's influence of dimension experiment setting (In Section 2.2.3). In this experiment, we fixed the rest layers of LeNet-5.

Conv1	$20 \times 1 \times 5 \times 5$					
Shape	25×20					
Core	1×9	2×8	3×5	4×4	6×2	8×1
Conv2 FC1 FC2	Orig.: $50 \times 20 \times 5 \times 5$, reshape: $50 \times 25 \times 20$, core size: $4 \times 4 \times 4$ Orig.: 800×500, reshape: $40 \times 25 \times 20 \times 20$, core size: $4 \times 4 \times 4 \times 4$ Orig.: 500×10, reshape: $25 \times 20 \times 10$, core size: $4 \times 4 \times 4$					
#(Param.)	2198	2210	2175	2180	2186	2212

Table 9
The details of SCADA-Tucker vs. ADA-Tucker experiment setting (In Section 2.4). In this experiment, we fixed the size of transformation matrices for all layers. Since there are four layers, ATA-tucker has four core tensors. c can be equal to 3, 4, 5, 6, 7.

	Conv1	Conv2	FC1	FC2	#(Param.)
Original	$20 \times 1 \times 5 \times 5$	$50 \times 20 \times 5 \times 5$	800×500	500×10	431K
Reshape	$20 \times 1 \times 5 \times 5$	$50 \times 20 \times 5 \times 5$	$40 \times 25 \times 20 \times 20$	$25 \times 20 \times 5 \times 2$	431K
Transformation Matrices	$20 \times c$ $1 \times c$ $5 \times c$ $5 \times c$	$50 \times c$ $20 \times c$ $5 \times c$ $5 \times c$	$40 \times c$ $25 \times c$ $20 \times c$ $20 \times c$	$25 \times c$ $20 \times c$ $5 \times c$ $2 \times c$	135c 66c 35c 32c
Bias	20	50	500	10	0.58K
ADA-Tucker core	$c \times c \times c \times c$	$c \times c \times c \times c$	$c \times c \times c \times c$	$c \times c \times c \times c$	$4c^4$
ADA-Tucker total	c^4+31c+20	c^4+80c+50	c^4+105c+500	c^4+52c+10	$4c^4$+268c+0.58K
SCADA-Tucker core	$c \times c \times c \times c$				c^4
SCADA-Tucker total	c^4+(31c+20)+(80c+50)+(105c+500)+(52c+10)				c^4+268c+0.58K

12×. Since the number of parameters of ResNet-20 is only about 0.27M, it is difficult to compress it further on CIFAR-10 dataset with negligible loss. The number of parameters of Wide ResNet-28-10 is about 36.5M, which is much bigger than ResNet-20's. Showing in Table 3, our ADA-Tucker compressed WRN-28-10 by amazing 58 times without deteriorating its performances. We also plotted visualizations for parameters comparisons of all layers in terms of ResNet-20 (Fig. 11) and WRN-28-10 (Fig. 12). The convincing results on these newly large networks suggest that the proposed method works well for modern CNN architectures.

Table 10
The detail of network architecture used in Section 4.2. The network architecture was referred to Chen et al. (2015, 2016); Denil et al. (2013). C: Convolution. RL: ReLU. MP: Max-pooling. DO: Dropout. FC: Fully-connected.

Layer	Operation	Input dim.	Inputs	Outputs	C size	MP size	#(Param.)
1	C,RL	32×32	3	32	5×5	–	2K
2	C,MP,DO,RL	32×32	32	64	5×5	2×2	51K
3	C,RL	16×16	64	64	5×5	–	102K
4	C,MP,DO,RL	16×16	64	128	5×5	2×2	205K
5	C,MP,DO,RL	8×8	128	256	5×5	2×2	819K
6	FC,Softmax	–	4096	10	–	-	40K

Table 11
ADA-Tucker setting details on the network architecture used in Section 4.2.

	Orig.	#(Param.)	Reshape	Core	#(Param.)
Conv1	$32 \times 3 \times 5 \times 5$	2K	96×25	12×12	1.6K
Conv2	$64 \times 32 \times 5 \times 5$	51K	$64 \times 32 \times 25$	$9 \times 9 \times 9$	1.9K
Conv3	$64 \times 64 \times 5 \times 5$	102K	$64 \times 64 \times 25$	$11 \times 11 \times 11$	3.1K
Conv4	$128 \times 64 \times 5 \times 5$	205K	$128 \times 64 \times 25$	$11 \times 11 \times 11$	3.8K
Conv5	$256 \times 128 \times 5 \times 5$	819K	$256 \times 128 \times 25$	$11 \times 11 \times 11$	6.1K
FC1	4096×10	40K	$64 \times 64 \times 10$	$9 \times 9 \times 9$	2.0K

Table 12
ADA-Tucker setting details on ResNet-20 (In Section 4.4).

ResNet-20		Orig.	#(Param.)	Reshape	Core	#(Param.)
Block0	Conv1	$16 \times 16 \times 3 \times 3$	2304	$16 \times 16 \times 9$	$12 \times 12 \times 6$	1302
	Conv2	$16 \times 16 \times 3 \times 3$	2304	$16 \times 16 \times 9$	$12 \times 12 \times 6$	1302
Block1	Conv1	$16 \times 16 \times 3 \times 3$	2304	$16 \times 16 \times 9$	$12 \times 12 \times 6$	1302
	Conv2	$16 \times 16 \times 3 \times 3$	2304	$16 \times 16 \times 9$	$12 \times 12 \times 6$	1302
Block2	Conv1	$16 \times 16 \times 3 \times 3$	2304	$16 \times 16 \times 9$	$12 \times 12 \times 6$	1302
	Conv2	$16 \times 16 \times 3 \times 3$	2304	$16 \times 16 \times 9$	$12 \times 12 \times 6$	1302
Block3	Conv1	$32 \times 16 \times 3 \times 3$	4608	$18 \times 16 \times 16$	$12 \times 10 \times 10$	1736
	Conv2	$32 \times 32 \times 3 \times 3$	9216	$12 \times 12 \times 8 \times 8$	$8 \times 8 \times 6 \times 6$	2592
Block4	Conv1	$32 \times 32 \times 3 \times 3$	9216	$12 \times 12 \times 8 \times 8$	$8 \times 8 \times 6 \times 6$	2592
	Conv2	$32 \times 32 \times 3 \times 3$	9216	$12 \times 12 \times 8 \times 8$	$8 \times 8 \times 6 \times 6$	2592
Block5	Conv1	$32 \times 32 \times 3 \times 3$	9216	$12 \times 12 \times 8 \times 8$	$8 \times 8 \times 6 \times 6$	2592
	Conv2	$32 \times 32 \times 3 \times 3$	9216	$12 \times 12 \times 8 \times 8$	$8 \times 8 \times 6 \times 6$	2592
Block6	Conv1	$64 \times 32 \times 3 \times 3$	18432	$32 \times 24 \times 24$	$24 \times 16 \times 16$	7680
	Conv2	$64 \times 64 \times 3 \times 3$	36864	$9 \times 8 \times 8 \times 8 \times 8$	$6 \times 6 \times 6 \times 6 \times 6$	8022
Block7	Conv1	$64 \times 64 \times 3 \times 3$	36864	$9 \times 8 \times 8 \times 8 \times 8$	$6 \times 6 \times 6 \times 6 \times 6$	8022
	Conv2	$64 \times 64 \times 3 \times 3$	36864	$9 \times 8 \times 8 \times 8 \times 8$	$6 \times 6 \times 6 \times 6 \times 6$	8022
Block8	Conv1	$64 \times 64 \times 3 \times 3$	36864	$9 \times 8 \times 8 \times 8 \times 8$	$6 \times 6 \times 6 \times 6 \times 6$	8022
	Conv2	$64 \times 64 \times 3 \times 3$	36864	$9 \times 8 \times 8 \times 8 \times 8$	$6 \times 6 \times 6 \times 6 \times 6$	8022

5. Conclusion

In this paper, we demonstrate that deep neural networks can be better compressed using weight tensors with proper orders and balanced dimensions of modes without performance degradation. We also present two methods based on our demonstration, ADA-Tucker and SCADA-Tucker, for deep neural network compression. Unlike previous decomposition methods, our methods adaptively adjust the order of original weight tensors and the dimension of each mode before Tucker decomposition. We do not need to add new layers for implementing the Tucker decomposition as other methods do. The advantage of our methods over those involving the frequency domain and pruning is that we do not require recording the indices of nonzero elements. We demonstrate the superior compressing capacity of the proposed model: after applying quantization and Huffman coding, ADA-Tucker compresses LeNet-5 and LeNet-300-100 by **691**× and **233**×, respectively, outperforming state-of-the-art methods. The experiments on CIFAR-10 and SVHN also show our models' overwhelming strength. The experiments on ImageNet indicate that Tucker decomposition combined with adaptive dimension adjustment has a great advantage over other decomposition-based methods especially at a large compression ratio. The convincing results on these newly large networks also suggest that the proposed method works well for modern CNN architectures.

In the future, we will further investigate the mechanism behind our findings and summarize a detailed rule of thumb for determining the order of weight tensor as well as dimensions of modes. Other research directions include combining this work with pruning techniques and exploiting its potential in accelerating computation and inference.

Acknowledgments

This research is partially supported by National Basic Research Program of China (973 Program) (grant nos. 2015CB352303 and 2015CB352502), National Natural Science Foundation (NSF) of China (grant nos. 61625301, 61671027 and 61731018), Qualcomm, and Microsoft Research Asia.

Appendix. Experiments settings

See Tables 4–13.

Table 13
ADA-Tucker setting details on Wide ResNet-28-10 (In Section 4.4).

Wide-ResNet-28-10		Orig.	#(Param.)	Reshape	Core	#(Param.)
Block0	Conv1	160 × 16 × 3 × 3	23K	16 × 16 × 10 × 9	10 × 10 × 6 × 6	4K
	Conv2	160 × 160 × 3 × 3	230K	24 × 24 × 20 × 20	12 × 12 × 12 × 12	22K
Block1	Conv1	160 × 160 × 3 × 3	230K	24 × 24 × 20 × 20	12 × 12 × 12 × 12	22K
	Conv2	160 × 160 × 3 × 3	230K	24 × 24 × 20 × 20	12 × 12 × 12 × 12	22K
Block2	Conv1	160 × 160 × 3 × 3	230K	24 × 24 × 20 × 20	12 × 12 × 12 × 12	22K
	Conv2	160 × 160 × 3 × 3	230K	24 × 24 × 20 × 20	12 × 12 × 12 × 12	22K
Block3	Conv1	160 × 160 × 3 × 3	230K	24 × 24 × 20 × 20	12 × 12 × 12 × 12	22K
	Conv2	160 × 160 × 3 × 3	230K	24 × 24 × 20 × 20	12 × 12 × 12 × 12	22K
Block4	Conv1	320 × 160 × 3 × 3	460K	80 × 80 × 72	36 × 36 × 36	55K
	Conv2	320 × 320 × 3 × 3	921K	32 × 32 × 30 × 30	16 × 16 × 16 × 16	67K
Block5	Conv1	320 × 320 × 3 × 3	921K	32 × 32 × 30 × 30	16 × 16 × 16 × 16	67K
	Conv2	320 × 320 × 3 × 3	921K	32 × 32 × 30 × 30	16 × 16 × 16 × 16	67K
Block6	Conv1	320 × 320 × 3 × 3	921K	32 × 32 × 30 × 30	16 × 16 × 16 × 16	67K
	Conv2	320 × 320 × 3 × 3	921K	32 × 32 × 30 × 30	16 × 16 × 16 × 16	67K
Block7	Conv1	320 × 320 × 3 × 3	921K	32 × 32 × 30 × 30	16 × 16 × 16 × 16	67K
	Conv2	320 × 320 × 3 × 3	921K	32 × 32 × 30 × 30	16 × 16 × 16 × 16	67K
Block8	Conv1	640 × 320 × 3 × 3	1843K	40 × 40 × 36 × 32	18 × 18 × 18 × 18	108K
	Conv2	640 × 640 × 3 × 3	3686K	48 × 48 × 40 × 40	20 × 20 × 20 × 20	163K
Block9	Conv1	640 × 640 × 3 × 3	3686K	48 × 48 × 40 × 40	20 × 20 × 20 × 20	163K
	Conv2	640 × 640 × 3 × 3	3686K	48 × 48 × 40 × 40	20 × 20 × 20 × 20	163K
Block10	Conv1	640 × 640 × 3 × 3	3686K	48 × 48 × 40 × 40	20 × 20 × 20 × 20	163K
	Conv2	640 × 640 × 3 × 3	3686K	48 × 48 × 40 × 40	20 × 20 × 20 × 20	163K
Block11	Conv1	640 × 640 × 3 × 3	3686K	48 × 48 × 40 × 40	20 × 20 × 20 × 20	163K
	Conv2	640 × 640 × 3 × 3	3686K	48 × 48 × 40 × 40	20 × 20 × 20 × 20	163K

References

Ba, J., & Caruana, R. (2014). Do deep nets really need to be deep. In *NIPS*.

Chen, W., Wilson, J., Tyree, S., Weinberger, K., & Chen, Y. (2015). Compressing neural networks with the hashing trick. In *ICML*.

Chen, W., Wilson, J., Tyree, S., Weinberger, K. Q., & Chen, Y. (2016). Compressing convolutional neural networks in the frequency domain. In *SIGKDD*.

Courbariaux, M., Bengio, Y., & David, J.-P. (2015). Binaryconnect: Training deep neural networks with binary weights during propagations. In *NIPS*.

De Lathauwer, L., De Moor, B., & Vandewalle, J. (2000). On the best rank-1 and rank-(r1, r2,..., rn) approximation of higher-order tensors. In *SIAM*.

Denil, M., Shakibi, B., Dinh, L., de Freitas, N., et al. (2013). Predicting parameters in deep learning. In *NIPS*.

Denton, E. L., Zaremba, W., Bruna, J., LeCun, Y., & Fergus, R. (2014). Exploiting linear structure within convolutional networks for efficient evaluation. In *NIPS*.

Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feed-forward neural networks. In *Aistats*.

Gong, Y., Liu, L., Yang, M., & Bourdev, L. (2015). Compressing deep convolutional networks using vector quantization. In *ICLR*.

Guo, Y., Yao, A., & Chen, Y. (2016). Dynamic network surgery for efficient DNNs. In *NIPS*.

Han, S., Mao, H., & Dally, W. J. (2016). Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In *ICLR*.

Han, S., Pool, J., Tran, J., & Dally, W. (2015). Learning both weights and connections for efficient neural network. In *NIPS*.

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *CVPR*.

Hinton, G., Vinyals, O., & Dean, J. (2014). Distilling the knowledge in a neural network. In *NIPS Workshop*.

Huang, G., Liu, Z., Weinberger, K. Q., & van der Maaten, L. (2017). Densely connected convolutional networks. In *CVPR*.

Jaderberg, M., Vedaldi, A., & Zisserman, A. (2014). Speeding up convolutional neural networks with low rank expansions. In *BMVC*.

Kim, Y.-D., Park, E., Yoo, S., Choi, T., Yang, L., & Shin, D. (2016). Compression of deep convolutional neural networks for fast and low power mobile applications. In *ICLR*.

Kingma, D., & Ba, J. (2015). Adam: A method for stochastic optimization. In *ICLR*.

Kolda, T. G., & Bader, B. W. (2009). Tensor decompositions and applications. In *SIAM*.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *NIPS*.

Lebedev, V., Ganin, Y., Rakhuba, M., Oseledets, I., & Lempitsky, V. (2015). Speeding-up convolutional neural networks using fine-tuned CP-decomposition. In *ICLR*.

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-Based learning applied to document recognition. In *PROC. of the IEEE*.

Luo, J.-H., Wu, J., & Lin, W. (2017). Thinet: A filter level pruning method for deep neural network compression. In *ICCV*.

Novikov, A., Podoprikhin, D., Osokin, A., & Vetrov, D. P. (2015). Tensorizing neural networks. In *NIPS*.

Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. In *ICLR*.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., et al. (2015). Going deeper with convolutions. In *CVPR*.

Tai, C., Xiao, T., Zhang, Y., Wang, X., & Weinanm, E. (2016). Convolutional neural networks with low-rank regularization. In *ICLR*.

Ullrich, K., Meeds, E., & Welling, M. (2017). Soft weight-sharing for neural network compression. In *ICLR*.

Wang, Y., Xu, C., You, S., Tao, D., & Xu, C. (2016). CNNpack: Packing convolutional neural networks in the frequency domain. In *NIPS*.

Xie, S., Girshick, R., Dollár, P., Tu, Z., & He, K. (2017). Aggregated residual transformations for deep neural networks. In *CVPR*. IEEE.

Yang, Y., Zhong, Z., Shen, T., & Lin, Z. (2018). Convolutional neural networks with alternately updated clique. In *CVPR*.

Zagoruyko, S., & Komodakis, N. (2016). Wide residual networks. In *BMVC*.

Zhang, T., Qi, G.-J., Xiao, B., & Wang, J. (2017). Interleaved group convolutions. In *ICCV*.

Zhou, A., Yao, A., Guo, Y., Xu, L., & Chen, Y. (2017). Incremental network quantization: Towards lossless CNNs with low-precision weights. In *ICLR*.