IPSJ SIG Technical Report

Vol.2021-NL-248 No.10
Vol.2021-CVIM-226 No.10
2021/5/20

Behavior-based DNN Compression: Pruning and Facilitation Methods

Koji Kamma[1,a)] Toshikazu Wada[1,b)]

Abstract: In this paper, we present two pruning methods. Pruning is a technique to reduce the computational cost of Deep Neural Networks (DNNs) by removing redundant neurons. The proposed pruning methods are Neuro-Unification (NU) and Reconstruction Error Aware Pruning (REAP). These methods do not only prune but also conduct *reconstruction* to prevent accuracy degradation. In reconstruction step, we update the weights connected to the remaining neurons so as to compensate the error caused by pruning. Therefore, the models pruned by the pruning methods suffer smaller accuracy degradation. As REAP needs significant amount of computation for selecting the neurons to be pruned, we developed a biorthogonal system-based algorithm that reduces the computational order of neuron selection from $O(n^4)$ to $O(n^3)$, where n denotes the number of neurons. We also propose two methods for facilitating pruning, Pruning Ratio Optimizer (PRO) and Serialized Residual Network (SRN). As REAP performs pruning in each layer separately, it is important to tune the pruning ratio (the ratio of neurons to be pruned) in each layer properly in order to preserve the model accuracy better. PRO is a method that can be combined with REAP to tune pruning ratios based on the error in the final layer of the pruned DNN. SRN is to facilitate pruning for ResNet. Due to its *identity shortcuts*, some layers cannot be pruned. Therefore, we once convert ResNet into an equivalent serial DNN model, which we call SRN, so that pruning can be performed in any layer.

1. Introduction

Since Hinton et al. won ImageNet Large Scale Visual Recognition Competition (ILSVRC) with AlexNet in 2012 [25], researchers have been developing various Deep Neural Networks (DNNs) for various Computer Vision tasks, such as recognition, detection, segmentation, and so on. Today, DNNs are already used in many industrial applications, and are expected to spread to wider range of industries in near future.

One of the bottlenecks of DNNs is that their inference (as well as training) is computationally expensive. In laboratories, large GPUs solve this problem. For example, VGG16 [37] model runs on NVIDIA Geforce GTX 1080 GPU at about 60 fps. This GPU consumes up to 180 W, thus a sufficient power supply is required to use it. Moreover, operational cost is also quite high due to large power consumption. Thus, we need an environment with rich computational resources for using DNN models.

On the other hand, we may want to use DNN models on edge devices with insufficient computational resources, such as in-vehicle cameras, security cameras, smartphones, drones, and so on. Some applications require high performance in accuracy and inference speed under severe constraints in power consumption, memory size, installation space, operating temperature, price, and so on. Therefore, if a large GPU is used to meet the performance requirements in accuracy and inference speed, it will not be able to meet the constraints. Conversely, if we select a device that meets the constraints, it will not be able to satisfy accuracy and/or inference speed requirements.

One of the solutions for this problem is *pruning* [13], [17], [26]. Pruning is to remove the redundant neurons (or weights) from the pretrained DNN models in order to make them computationally less expensive. Generally, pruning DNN models causes accuracy degradation, which is not preferable. Therefore, the pruning methods should be designed so as to prevent accuracy degradation as well as possible.

In this paper, we propose two pruning methods, Neuro-Unification (NU) [22] and Reconstruction Error Aware Pruning (REAP) [23].

The feature of NU is that it does not just prunes but unifies the neurons having similar *behaviors*. Having similar behaviors is, in other words, those neurons' outputs have strong correlation. When we prune a neuron, we reconstruct its behavior by tuning the weights connected to the other neuron, which results in less accuracy degradation.

REAP is the extended version of NU. In REAP, the behavior of the pruned one is reconstructed by updating the weights connected to all remaining neurons in the same layer by using least squares method. The difficulty is that it requires huge amount of computation for selecting the neurons to be pruned. In order to select one, we once need to prune each neuron, conduct reconstruction based on least squares method, and see which neuron has the smallest error after reconstruction. For efficient neuron selection, we developed a biorthogonal system-based algorithm. To our best knowledge, REAP is the best method in terms of pre-

[1] Wakayama University, Sakaedani 930, Wakayama-shi, Wakayama 640–8510, Japan
[a)] kammakoji@gmail.com
[b)] twada@ieee.org

venting the layer-wise error.

We also propose two methods for facilitating pruning, Pruning Ratio Optimizer (PRO) and Serialized Residual Network (SRN).

It is important to tune pruning ratio (the ratio of neurons to be pruned) in each layer, because the proper pruning ratio depends on how redundant the layer is. PRO is the method that can be combined with REAP (and some other pruning methods) for optimizing pruning ratios. In PRO, we tune pruning ratio in each layer based on the error in the final layer of the model, while pruning itself is performed based on the layer-wise error. PRO can perform pruning ratio optimization more efficiently and effectively than the existing reinforcement learning-based method [16].

SRN is to facilitate pruning on ResNet. Generally, ResNet [14] is difficult to be pruned due to its architectural feature. ResNet architecture is composed of stacked blocks, and each block is composed of several layers and has branched paths. In each block, the inputs are propagated forward as they are in one path, and linear transformations (convolutions) are performed in the other path, and both are eventually added. At this addition, two inputs must have the same dimensions, which means the layer that have branched paths cannot be pruned. Therefore, we propose a method to convert ResNet into an equivalent serial model that we call *Serialized Residual Network* (SRN). SRN can emulate ResNet by setting some weights to perform identity mapping. Even though SRN has more computational complexity than ResNet, it is easier to be compressed by pruning, because SRN has a serial architecture and any layer can be pruned.

It is worth noting that there is an opinion that it does not matter how well we preserve accuracy of the pruned models, because the pruned models are retrained anyway in order to recover their accuracy. However, as we will show in this paper, the better we preserve the accuracy of the pruned models, the more accurate those models eventually become after retraining. More importantly, as we will show in Sec. 4, the compression methods that can preserve the model accuracy well enable us to tune the pruning ratio in each layer efficiently. Therefore, it is important to choose a good pruning method that can prevent accuracy degradation as well as possible.

The rest of the paper are structured as follows. Sec. 2 shows outlines of works related to DNN compression techniques. We explain our proposed methods and the experiments in Sec. 3-5. We conclude the discussion of this paper in Sec. 6.

2. Overview for DNN compression methods

A lot of works have been done to explore efficient DNNs. There are two major approaches. One is to reduce the redundancy of pretrained large DNN models, which includes pruning, sparsification, factorization, quantization, and distillation. The other is Neural Architecture Search (NAS) to search the architectures that can achieve high accuracy within a given computational budget in the context of training from scratch.

2.1 Pruning

Pruning is to remove the neurons or the weights that are unimportant or redundant. The pruning methods can be divided into two groups: the weight pruning methods [5], [8], [11], [26] and the structured pruning methods [17], [29], [30], [36], [39], [47].

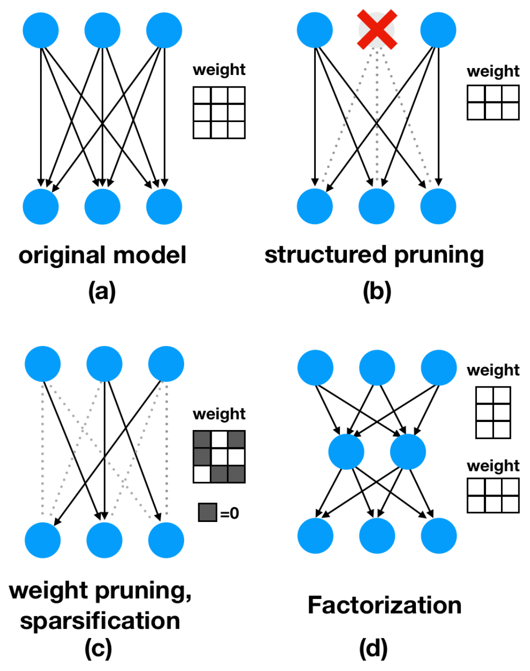

Fig. 1 The conceptual drawings of the compression methods for DNN models. (a) The original model. (b) The model compressed by structured pruning methods. (c) The model compressed by weight pruning methods or sparsification methods. (d) The model compressed by factorization methods. The advantage of structured pruning is that the weight matrix gets smaller by pruning, which means that the pruned model can be deployed with a general hardware device and a general library. On the other hand, the models compressed by the weight pruning methods and the sparsification methods require the environments that can conduct operations at only non-zero weights. The factorization methods may reduce the computational cost of the model, however, additional layers are added to the model which bring extra computational overheads.

The weight pruning methods prune the weights that are redundant or do not contribute to the performance of the model significantly. In practice, the pruned weights are not actually removed but are set to zero.

The first work of weight pruning is Optimal Brain Damage (OBD) [26]. OBD evaluates the importance of the weights based on the Hessian of the cost function. As it is computationally intensive to calculate the whole Hessian, OBD only computes its diagonal entries, and assumes that the non-diagonals are zero. However, this is not a reasonable assumption, because the weights in the same model are obviously dependent on each other. Optimal Brain Surgeon (OBS) not only prunes but also conducts *surgery* (Note that, in our proposed methods, we call it *reconstruction* instead of *surgery*.) to compensate the damage of pruning by tuning the remaining weights based on the whole Hessian [13]. However, as already mentioned, it is not feasible to compute the whole Hessian for large DNN models that have millions of weights. Therefore, OBS can be applied to only small models. There are also the magnitude-based pruning methods that prune the weights based on their absolute values [12], [46]. However, pruning the weights having small absolute values may have significantly impact on accuracy, and vise versa.

The common drawback of the weight pruning methods is that the pruned model has sparse weight matrices/tensors and their shapes are still the same with before pruning, as shown in Fig. 1 (c). Therefore, in order to take advantage of weight pruning, the pruned model should be deployed on an environment supporting sparse computation that skips multiplications with zero weights.

On the other hand, the structured pruning methods conduct neuron-level pruning. This group also includes the methods based on the derivative information of the cost function, such as [32], and the magnitude based methods, such as [15] and [33]. NU and REAP (the proposed methods in this paper) and some relevant methods, such as ThiNet [30] and Channel Pruning (CP) [17] also belong to this group. The advantage of the structured pruning methods is that when a neuron is pruned, the whole weights connected to it are also removed, and thus, the weight tensor will have a smaller shape after pruning, as shown in Fig. 1 (b). Therefore, the pruned model can be deployed in general devices with general libraries.

We can also categorize the pruning methods from another perspective: the holistic pruning methods [11], [15], [26], [32], [39] and the layer-wise pruning methods [8], [17], [22], [23], [30], [43], [47].

The holistic methods are designed for comparing the importance of the neurons/weights in the whole model simultaneously and removing the least salient one. For example, the method proposed in [32] aims for pruning convolutional layers and evaluates the importance of the channels based on the first derivative information of the cost function. However, as the cost function and the weights normally have non-linear relationship, it is not reasonable to use only the first derivative information. Another example is Structured Probabilistic Pruning (SPP) [39], a pruning method using dropout. The idea of SPP is to drop some weights out once and conduct training, and if it ends up in high accuracy, it means the dropped weights are not important and can be eventually pruned. Although, as training has to be repeated many times to evaluate the importance of neurons, SPP is a computationally intensive way of pruning.

On the other hand, some recent works [8], [17], [22], [23], [30], [43] have offered the layer-wise pruning methods. The layer-wise methods conduct pruning in each layer separately based on the layer-wise error. As their optimization problems are simpler than those of the holistic methods, it is possible to use more theoretically sound criteria for selecting the neurons to be pruned. For example, Layer-Wise Optimal Brain Surgeon (LOBS) [8] prunes the weights and updates the remaining ones based on the Hessian of the MSE of layer-wise outputs over only the weights in that layer. While the original holistic OBS computes the Hessian of the cost function over all the weights of the model, LOBS computes the Hessian layer by layer, which significantly reduces the computational cost. Therefore, LOBS can be used for compressing larger DNN models. Channel Pruning (CP) [17] and ThiNet [30] prune the neurons based on the layer-wise error and conduct *reconstruction* with least squares method so as to compensate the error caused by pruning. Our REAP [23] are closely relevant to CP and ThiNet, however, we take a more sophisticated approach, as we will discuss in Sec. 3.

2.2 Sparsification

The sparsification methods make the weight matrices/tensors sparse by conducting extra training on the pretrained models with L1 regularization [4], [40]. The recent works include Sparse Convolutional Neural Network that combines L1 regularization for convolutional layers and tensor decomposition [4]. Zhao et al. proposed a group Lasso-based method for feature selection of multi-modal DNN models [45].

The theoretical weakness of sparsification is that L1 regularization shifts the global minimum of the cost function. Thus, weight selection results may not be optimal in terms of preserving the accuracy of the model. In addition, similarly with the weight pruning methods, the weight matrices/tensors retain the same dimensions after sparsification, as shown in Fig. 1 (c). Thus, we need the environments dedicated for the sparsified models that skip the computations with the zeroed weights to take advantage of sparsification.

2.3 Factorization

The idea of factorization is to decompose a large weight matrix/tensor into several smaller matrices/tensors, as shown in Fig. 1 (d). The most fundamental method in this group is presented in [41]. They apply SVD to a large weight matrix, and approximate it by the product of small matrices by discarding the components with small singular values. This results in reducing the weights with small sacrifice of accuracy. For example, assume that a $m \times n$ matrix is approximated by the product of a $m \times o$ matrix and a $o \times n$ matrix. If $o \ll m, n$, the number of weights reduces from mn to $(m + n)o$. Jaderbery et al. expanded this idea for convolutional layers [21]. They use row rank expansion technique for factorizing convolutional kernels. Some other methods [42], [44] also belong to this group.

The drawback of factorization is that they may indeed reduce weights and FLOPs, however, they add extra layers that have computational overheads. Therefore, the effectiveness of factorization may not be as significant as it seems, depending on the computational environments, model architecture, and so on.

2.4 Quantization

The methods in this group reduce the redundancy of each bitwise operation, e.g. changing floating point precision from 32-bit to 8-bit. For lower-bit operations, special hardwares and libraries are required.

A further development of this idea is binarization. BinaryConnect [6] is a method to produce the DNN models with only 2 weight values (e.g. -1 and 1) so that the operations can be done by only additions and subtractions. As additions and subtractions are less expensive than multiplications, the inference can be faster by binarization. The binarized models can be deployed on general hardwares and libraries. However, as they do not need multipliers, it is more ideal to use the dedicated environments optimized for additions and subtractions.

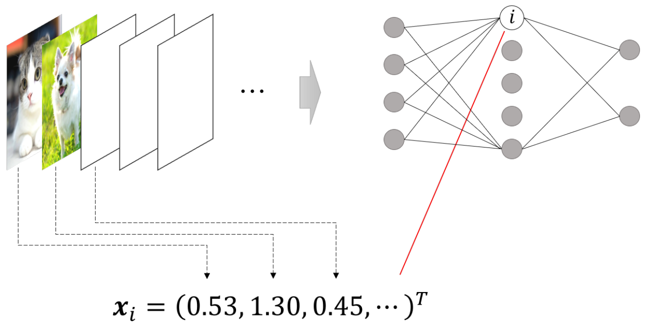

Fig. 2 The conceptual drawing of neuron behavior encoding. When we feed several images into a DNN model, each neuron obtains a behavioral vector composed of their own outputs.

2.5 Distillation

Hinton et al. proposed Knowledge Distillation [18], a method to transfer the knowledge learned by a pretrained large model (teacher) into a small model (student). When training the student, its weights are updated so as to minimize the output difference from the convex combination of the ground truth and the teacher's outputs. The student trained in this way shows good performance for its size. Mirzadeh et al. proposed multi-step distillation [31]. They found out that Knowledge Distillation tends to fail when the student has much smaller architecture than the teacher, and multi-step distillation using the intermediate-sized models can ease this problem.

The most significant drawback of Knowledge Distillation is that it is difficult to search the proper architecture for the student. Typically, the student has fewer layers and fewer neurons in each layer than the teacher, although we do not know how fewer they can be. Therefore, we need to conduct training to judge if the current architecture is proper or not, which is time-consuming.

2.6 Neural Architecture Search (NAS)

Apart from the compression methods mentioned above, Neural Architecture Search (NAS) methods have also been developed. The NAS methods can be divided into two groups: the evolutionary algorithm-based methods [10], [24], [28], and the reinforcement learning-based ones [19], [35], [48]. The idea of these approaches is to prepare a graph that the DNN model will be built on, put a layer (or a block composed of several layers) on each node, and train the model built on the graph. Reinforcement learning or evolutionary algorithm are used to optimize the architecture in each node. These approaches are computationally intensive.

3. Neuro-Unification and Reconstruction Error Aware Pruning

In this section, we explain our proposed pruning methods, Neuro-Unification (NU) and Reconstruction Error Aware Pruning (REAP). As REAP is an extended one of NU, we explain only the basic idea of NU, and then explain REAP in detail.

3.1 Neuro-Unification

The idea of NU is that we do not just prune a neuron but unify the neurons having similar *behaviors*. Therefore, we conduct the following procedures in each layer.

1) Encode the behavior of each neuron.
2) Compute the behavioral similarity of every possible neuron pair.
3) Unify the most similar pair.
4) Terminate iteration if we have pruned as many neurons as we want. Otherwise, go to 1).

Fig. 2 is the conceptual drawing of neuron behavior encoding. For a single image, the i-th neuron outputs a scalar value. For d input images, the output becomes a vector $\boldsymbol{x}_i \in \mathbb{R}^d$. We call it the *behavioral vector* of the i-th neuron.

If there are a pair of neurons with the same behaviors, we can unify them without error. Here, having the same behaviors means that their behavioral vectors are linearly dependent on each other, such as $\boldsymbol{x}_i = \alpha \boldsymbol{x}_j$. We show an example below.

Let n and n' denote the numbers of neurons in a layer and the next layer (intermediate layer and right one in Fig. 3 (a), respectively), $\mathcal{I} = \{1, \cdots, n\}$ denote the set of neuron indices, $\boldsymbol{w}_i \in \mathbb{R}^{n'}$ denote the weights going from the i-th neuron to the ones in the next layer. The forward propagation is described by

$$Y = \boldsymbol{x}_i \boldsymbol{w}_i^\top + \boldsymbol{x}_j \boldsymbol{w}_j^\top + \sum_{k \in \mathcal{I} \setminus \{i,j\}} \boldsymbol{x}_k \boldsymbol{w}_k^\top, \tag{1}$$

where $Y \in \mathbb{R}^{d \times n'}$ denotes the inner activation levels in the next layer.

If $\boldsymbol{x}_i = a_{ij} \boldsymbol{x}_j$ holds for some a_{ij}, we can unify the i-th and the j-th neurons without error. As shown in Fig. 3 (b), we prune the i-th neuron and update the j-th one's weights going to the next layer as

$$\boldsymbol{w}_j \leftarrow a_{ij} \boldsymbol{w}_i + \boldsymbol{w}_j. \tag{2}$$

Then, the forward propagation formula becomes

$$Y = \boldsymbol{x}_j \left(a_{ij} \boldsymbol{w}_i^\top + \boldsymbol{w}_j^\top \right) + \sum_{k \in \mathcal{I} \setminus \{i,j\}} \boldsymbol{x}_k \boldsymbol{w}_k^\top. \tag{3}$$

Eq. (1) and Eq. (3) are equivalent because $\boldsymbol{x}_i = a_{ij} \boldsymbol{x}_j$ holds, which means the original Y is preserved. This is how we reconstruct the behaviors of the pruned neurons.

As above, in the case of the neurons having linearly dependent behavioral vectors, they can be unified without error. Although, it rarely happens that those behavioral vectors are linearly dependent. In the case of unifying the neurons having linearly independent but similar behavioral vectors, we accept some error and make it as small as possible. In this case, we first approximate $\boldsymbol{x}_i$ by a vector which is linearly dependent on $\boldsymbol{x}_j$:

$$\boldsymbol{x}_i \simeq a_{ij} \boldsymbol{x}_j. \tag{4}$$

We regard that $a_{ij} \boldsymbol{x}_j$ is the behavioral vector of the i-th neuron so that we can conduct unification in the same manner with Eq. (2).

Here is a question. How to determine a_{ij} in Eq. (4)? In order to minimize the error in the next layer, we have to minimize the error of Y. This can be formalized as

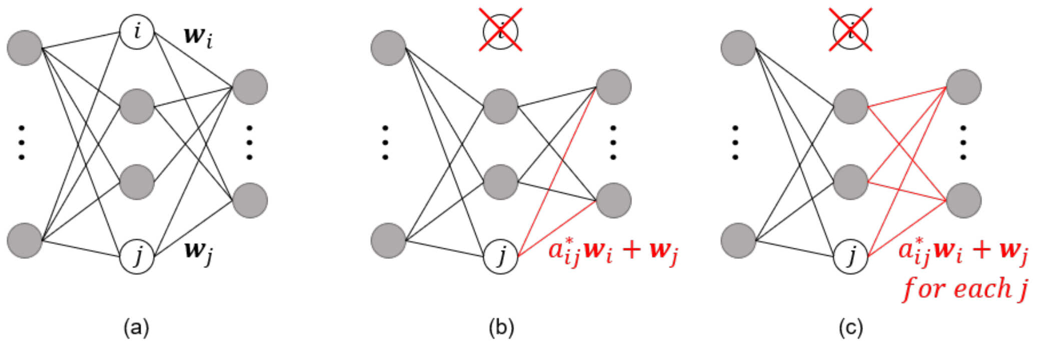

Fig. 3 The illustration of neuron unification. (a) The original model. (b) The model pruned with NU. Only one neuron is used for reconstructing the pruned one's behavior. (c) The model pruned with REAP. All the remaining neurons are used for reconstruction.

$$\begin{aligned} a^*_{ij} &= \underset{a_{ij}}{\operatorname{argmin}} \left\| \left(\boldsymbol{x}_i - a_{ij} \boldsymbol{x}_j \right) \boldsymbol{w}_i^\top \right\|_{\mathrm{F}}^2 \\ &= \underset{a_{ij}}{\operatorname{argmin}} \sum_{k=1}^{d} \sum_{l=1}^{n'} \left(\left(x_{i(k)} - a_{ij} x_{j(k)} \right) w_{i(l)} \right)^2 \\ &= \underset{a_{ij}}{\operatorname{argmin}} \sum_{l=1}^{n'} w_{i(l)}^2 \sum_{k=1}^{d} \left(x_{i(k)} - a_{ij} x_{j(k)} \right)^2 \\ &= \underset{a_{ij}}{\operatorname{argmin}} \|\boldsymbol{w}_i\|^2 \left\| \boldsymbol{x}_i - a_{ij} \boldsymbol{x}_j \right\|^2, \end{aligned} \tag{5}$$

where $x_{i(k)}$ denotes the k-th entry of $\boldsymbol{x}_i$ and $w_{i(l)}$ denotes the l-th entry of $\boldsymbol{w}_i$. We can omit $\|\boldsymbol{w}_i\|^2$ in Eq. (5) as it is a constant. Then, Eq. (5) can be rewritten as

$$a^*_{ij} = \underset{a_{ij}}{\operatorname{argmin}} \left\| \boldsymbol{x}_i - a_{ij} \boldsymbol{x}_j \right\|^2. \tag{6}$$

After all, we have to compute the orthogonal projection of $\boldsymbol{x}_i$ onto $\boldsymbol{x}_j$. Thus, we have

$$a^*_{ij} = \frac{\left\langle \boldsymbol{x}_i, \boldsymbol{x}_j \right\rangle}{\left\| \boldsymbol{x}_j \right\|^2}. \tag{7}$$

If $\boldsymbol{x}_i$ and $a^*_{ij} \boldsymbol{x}_j$ are similar enough, it means the j-th neuron can emulate the behavior of the i-th one well, and the error caused by this unification will be small.

3.2 Reconstruction Error Aware Pruning

In NU, the output of the pruned neuron is reconstructed from another neuron. In REAP, we use all the remaining neurons for reconstruction, as shown in Fig. 3. This can be formulated by

$$\{a^*_{ij} | j \in \mathcal{I} \backslash \{i\}\} = \underset{a_{ij}}{\operatorname{argmin}} \left\| \boldsymbol{x}_i - \sum_{j \in \mathcal{I} \backslash \{i\}} a_{ij} \boldsymbol{x}_j \right\|^2. \tag{8}$$

Similarly with Eq. (2), the weights of the remaining neurons are updated as follows for each $j \in \mathcal{I} \setminus \{i\}$.

$$\boldsymbol{w}'_j = a^*_{ij} \boldsymbol{w}_i + \boldsymbol{w}_j. \tag{9}$$

3.2.1 How to select the neuron to be pruned

We should select the neuron to be pruned so as to minimize the reconstruction error of Y. This problem can be formulated as

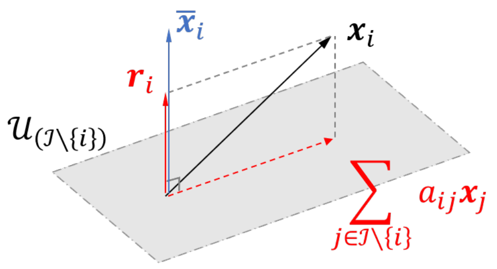
$\bar{\boldsymbol{x}}_i$
$\boldsymbol{x}_i$
$\boldsymbol{r}_i$
$\mathcal{U}_{(\mathcal{I} \backslash \{i\})}$
$\sum_{j \in \mathcal{I} \backslash \{i\}} a_{ij} \boldsymbol{x}_j$

Fig. 4 Illustration of the projection of $\boldsymbol{x}_i$ onto a subspace $\mathcal{U}_{(\mathcal{I} \backslash \{i\})}$ spanned by $\{\boldsymbol{x}_j | j \in \mathcal{I} \backslash \{i\}\}$. Computing the orthogonal projection of $\boldsymbol{x}_i$ onto $\mathcal{U}_{(\mathcal{I} \backslash \{i\})}$ is equivalent to solving the problem of reconstructing $\boldsymbol{x}_i$ from $\{\boldsymbol{x}_j | j \in \mathcal{I} \backslash \{i\}\}$ by least squares method. The residual $\boldsymbol{r}_i$ is linearly dependent on $\bar{\boldsymbol{x}}_i$, the dual basis for $\boldsymbol{x}_i$.

$$\begin{aligned} i^* &= \underset{i}{\operatorname{argmin}} \left\| Y - \sum_{j \in \mathcal{I} \backslash \{i\}} \boldsymbol{x}_j \boldsymbol{w}'^\top_j \right\|_{\mathrm{F}}^2 \\ &= \underset{i}{\operatorname{argmin}} \left\| Y - \sum_{j \in \mathcal{I} \backslash \{i\}} \boldsymbol{x}_j \left(a^*_{ij} \boldsymbol{w}_i + \boldsymbol{w}_j \right)^\top \right\|_{\mathrm{F}}^2. \end{aligned} \tag{10}$$

In order to solve Eq. (10), we first solve Eq. (8) for each $i \in \mathcal{I}$ to compute the a^*-s. With straightforward solution using least squares method, the amount of computation would be tremendous if we have a lot of neurons (and we usually have a lot of neurons).

3.2.2 Neuron selection algorithm based on biorthogonal system

We solve Eq. (8) for each i in *one-shot* by using biorthogonal system. Let $\boldsymbol{r}_i$ denote the residual of $\boldsymbol{x}_i$ reconstructed from the other $\boldsymbol{x}$-s:

$$\boldsymbol{r}_i = \boldsymbol{x}_i - \sum_{j \in \mathcal{I} \backslash \{i\}} a^*_{ij} \boldsymbol{x}_j. \tag{11}$$

As $\boldsymbol{r}_i$ is the residual of $\boldsymbol{x}_i$, $\boldsymbol{r}_i$ is orthogonal to all other $\boldsymbol{x}$-s. In other words, $\boldsymbol{r}_i$ is orthogonal to the subspace $\mathcal{U}_{(\mathcal{I} \backslash \{i\})}$ spanned by $\{\boldsymbol{x}_j | j \in \mathcal{I} \setminus \{i\}\}$, as shown in Fig. 4.

We compute the $\boldsymbol{r}$-s by using biorthogonal system. Let $\{\bar{\boldsymbol{x}}_j | j \in \mathcal{I}\}$ denote the dual bases of $\{\boldsymbol{x}_j | j \in \mathcal{I}\}$. The biorthogonal system is defined by

$$\langle \boldsymbol{x}_i, \bar{\boldsymbol{x}}_j \rangle = \begin{cases} 1 & (i = j) \\ 0 & (otherwise) \end{cases}. \tag{12}$$

The biorthogonal expansion for $\boldsymbol{r}_i$ is given by

$$\boldsymbol{r}_i = \sum_{j \in \mathcal{I}} \langle \boldsymbol{r}_i, \boldsymbol{x}_j \rangle \bar{\boldsymbol{x}}_j. \tag{13}$$

Obviously, $\langle \boldsymbol{r}_i, \boldsymbol{x}_j \rangle = 0$ holds for each $j \in \mathcal{I} \setminus \{i\}$, because $\boldsymbol{r}_i$ is orthogonal to $\mathcal{U}_{(\mathcal{I}\setminus\{i\})}$. Therefore, Eq. (13) can be rewritten as

$$\boldsymbol{r}_i = \langle \boldsymbol{r}_i, \boldsymbol{x}_i \rangle \bar{\boldsymbol{x}}_i + \sum_{j \in \mathcal{I}\setminus\{i\}} \langle \boldsymbol{r}_i, \boldsymbol{x}_j \rangle \bar{\boldsymbol{x}}_j = \langle \boldsymbol{r}_i, \boldsymbol{x}_i \rangle \bar{\boldsymbol{x}}_i. \tag{14}$$

Eq. (14) means that $\boldsymbol{r}_i$ is linearly dependent on $\bar{\boldsymbol{x}}_i$. Therefore, we can obtain $\boldsymbol{r}_i$ by computing the orthogonal projection of $\boldsymbol{x}_i$ onto $\bar{\boldsymbol{x}}_i$, as shown in Fig. 4. Thus, the following holds:

$$\boldsymbol{r}_i = \frac{\langle \boldsymbol{x}_i, \bar{\boldsymbol{x}}_i \rangle}{\|\bar{\boldsymbol{x}}_i\|^2} \bar{\boldsymbol{x}}_i = \frac{\bar{\boldsymbol{x}}_i}{\|\bar{\boldsymbol{x}}_i\|^2}. \tag{15}$$

By using Eq. (15), we can compute $\boldsymbol{r}_i$ for each $i \in \mathcal{I}$ in one-shot. Let $X = [\boldsymbol{x}_1 \cdots \boldsymbol{x}_n]$ and $\bar{X} = [\bar{\boldsymbol{x}}_1 \cdots \bar{\boldsymbol{x}}_n]$. By definition of dual bases, $\bar{X}$ can be computed as

$$\bar{X} = (X^g)^\top, \tag{16}$$

where X^g denotes the generalized inverse of X. Then, we can compute the $\boldsymbol{r}$-s by using Eq. (15) for each column of $\bar{X}$.

We also need to compute the a-s, the coefficients for reconstruction. Let $R = [\boldsymbol{r}_1 \cdots \boldsymbol{r}_n]$ and $A^* \in \mathbb{R}^{n \times n}$ denote a matrix whose (i, j) entry is a^*_{ij}. Because we have Eq. (11), the following must hold:

$$R = X - XA^*. \tag{17}$$

Then, we have

$$A^* = E - X^g R, \tag{18}$$

where E denotes an identity matrix.

3.2.3 Even faster computation for selecting the second neuron to be pruned

Assume that we have pruned the i-th neuron. When we prune another neuron, we may simply repeat the same procedures mentioned in Sec. 3.2.2 with the remaining neurons. We have to solve the following problem for each j.

$$\{b^*_{jk} | k \in \mathcal{I}\setminus\{i, j\}\} = \operatorname*{argmin}_{b_{jk}} \left\| \boldsymbol{x}_j - \sum_{k \in \mathcal{I}\setminus\{i,j\}} b_{jk} \boldsymbol{x}_k \right\|^2, \tag{19}$$

Then, we solve the following problem for selecting the next neuron to be pruned:

$$k^* = \operatorname*{argmin}_{k} \left\| Y - \sum_{k \in \mathcal{I}\setminus\{i,j\}} \boldsymbol{x}_k \left(b^*_{jk} \boldsymbol{w}'_j + \boldsymbol{w}'_k \right)^\top \right\|^2_{\mathrm{F}}. \tag{20}$$

Note that we already have the $\boldsymbol{w}'$-s in Eq. (9).

Although we may use the proposed biorthogonal system-based algorithm again for solving Eq. (19) and Eq. (20), we can solve them even faster by using the solution of Eq. (8). We already have

$$\boldsymbol{x}_i = \boldsymbol{r}_i + a^*_{ij} \boldsymbol{x}_j + \sum_{k \in \mathcal{I}\setminus\{i,j\}} a^*_{ik} \boldsymbol{x}_k, \tag{21}$$

$$\boldsymbol{x}_j = \boldsymbol{r}_j + a^*_{ji} \boldsymbol{x}_i + \sum_{k \in \mathcal{I}\setminus\{i,j\}} a^*_{jk} \boldsymbol{x}_k. \tag{22}$$

After pruning both the i-th and the j-th neurons, we can no more use $\boldsymbol{x}_j$ for reconstructing $\boldsymbol{x}_i$. Thus, we substitute Eq. (21) to Eq. (22) and get

$$\boldsymbol{x}_j = \frac{\boldsymbol{r}_j + a^*_{ji} \boldsymbol{r}_i}{1 - a^*_{ji} a^*_{ij}} + \sum_{k \in \mathcal{I}\setminus\{i,j\}} \frac{a^*_{jk} + a^*_{ji} a^*_{ik}}{1 - a^*_{ji} a^*_{ij}} \boldsymbol{x}_k. \tag{23}$$

We have $\langle \boldsymbol{x}_k, \boldsymbol{r}_i \rangle = 0$ and $\langle \boldsymbol{x}_k, \boldsymbol{r}_j \rangle = 0$ for each $k \in \mathcal{I}\setminus\{i, j\}$. Therefore, the first term on the RHS of Eq. (23) denotes the residual of $\boldsymbol{x}_j$ reconstructed from $\{\boldsymbol{x}_k | k \in \mathcal{I}\setminus\{i, j\}\}$ and the second term denotes the projection of $\boldsymbol{x}_j$ onto the subspace spanned by $\{\boldsymbol{x}_k | k \in \mathcal{I}\setminus\{i, j\}\}$. Thus, the coefficients of the $\boldsymbol{x}$-s in the second term is equivalent to the solution of Eq. (19):

$$b^*_{jk} = \frac{a^*_{jk} + a^*_{ji} a^*_{ik}}{1 - a^*_{ji} a^*_{ij}}. \tag{24}$$

After computing the b^*-s, we can solve Eq. (20) easily.

3.2.4 Algorithm

To sum up, our neuron selection algorithm can be described as Algorithm 1.

Algorithm 1

```
Input: A set of neuron indices $\mathcal{I} = \{1, \cdots, n\}$, a set of neuron behavioral vectors $\{\boldsymbol{x}_i | i \in \mathcal{I}\}$ and the set of weight vectors of each neuron $\{\boldsymbol{w}_i | i \in \mathcal{I}\}$, an output matrix $Y$, desired number of remaining neurons $q$.
Output: A set $\mathcal{J} \subseteq \mathcal{I}$ composed of remaining neurons' indices and a matrix $A \in \mathbb{R}^{n \times n}$ whose $(i, j)$ entry is $a_{ij}$.
$\mathcal{J} \leftarrow \mathcal{I}$.
compute $R$ and $A$ by using Eq. (16), (15), (17), and (18).
while $|\mathcal{J}| > q$ do
    Select the neuron index $i$ to be pruned by solving Eq. (10).
    $\mathcal{J} \leftarrow \mathcal{J} \setminus \{i\}$.
    $\boldsymbol{w}_j \leftarrow a_{ij}\boldsymbol{w}_i + \boldsymbol{w}_j$ for each $j \in \mathcal{J}$.
    $a_{jk} \leftarrow (a^*_{jk} + a^*_{ji}a^*_{ik})/(1 - a^*_{ji}a^*_{ij})$ for each $j, k \in \mathcal{J}, j \neq k$.
end while
```

Table 1 VGG16 on ImageNet. The changes of top-5 accuracy from the baseline (89.5%) are reported (The greater, the better.). In this table, "rt" stands for "retraining". *our implementation.

FLOPs	Method	Acc. before rt	Acc. after rt	epochs#
×0.5	REAP	**-2.0%**	**+0.2%**	10
	NU [22]	-5.0%	-	-
	CP [17]	-2.7%	0.0%	10
	*ThiNet [30]	-65.0%	-1.0%	10
	SPP [39]	-	0.0%	-
×0.2	REAP	**-9.4%**	**-1.3%**	10
	CP [17]	-22.0%	-1.7%	10
	*ThiNet [30]	-88.8%	-3.4%	10
	SPP [39]	-	-2.0%	-

3.2.5 Related works

Channel Pruning (CP) [17] is also a layer-wise pruning method that conducts reconstruction with least squares method. Although, its strategy for neuron selection is different from ours. They select the neurons to be pruned by solving the following Lasso regression problem:

$$\boldsymbol{\beta}^* = \operatorname*{argmin}_{\boldsymbol{\beta}} \left\| Y - \sum_{i \in \mathcal{I}} \beta_i \boldsymbol{x}_i \boldsymbol{w}_i^\top \right\|^2_{\mathrm{F}} + \lambda \|\boldsymbol{\beta}\|_1 \quad \text{subject to } \|\boldsymbol{\beta}\|_0 \leq q, \tag{25}$$

where q denotes the desired number of neurons and $\boldsymbol{\beta} = (\beta_1, \cdots, \beta_n)^\top$ denotes a vector used for neuron selection. If

Table 2 Time (sec.) spent for channel selection per layer (channels# in the parentheses), at the pruning ratios of 0.25, 0.5, 0.75.

Method	*Conv1-1* (64)			*Conv1-2* (64)			*Conv2-1* (128)			*Conv2-2* (128)		
	0.25	0.50	0.75	0.25	0.50	0.75	0.25	0.50	0.75	0.25	0.50	0.75
REAP	2.2	2.6	3.0	2.3	2.7	3.1	8.4	11.2	14.0	8.6	11.5	14.3
CP	1.8	1.7	1.6	1.1	1.4	1.3	2.1	2.6	2.4	3.2	3.3	3.3

Method	*Conv3-1* (256)			*Conv3-2* (256)			*Conv4-1* (512)			*Conv4-2* (512)		
	0.25	0.50	0.75	0.25	0.50	0.75	0.25	0.50	0.75	0.25	0.50	0.75
REAP	41.0	60.1	79.9	41.0	60.3	79.9	363.6	612.8	868.7	361.1	603.4	848.9
CP	6.6	7.6	6.7	5.4	6.0	5.8	12.8	15.1	12.0	13.4	14.1	12.0

$\beta_i = 0$, the i-th neuron can be pruned.

Then, reconstruction is performed with least squares method.

$$\left\{\boldsymbol{w}_j^* | j \in \mathcal{J}\right\} = \underset{\boldsymbol{w}_j}{\operatorname{argmin}} \left\| Y - \sum_{j \in \mathcal{J}} \beta_j^* \boldsymbol{x}_j \boldsymbol{w}_j^\top \right\|_F^2, \tag{26}$$

where $\mathcal{J} = \{i | \beta_i^* \neq 0\}$ denotes the set composed of remaining neurons' indices.

The weakness of CP is that it selects the neurons to be pruned based on the error *before* reconstruction, which does not guarantee the minimal error *after* reconstruction. On the other hand, REAP selects the neurons to be pruned based on their reconstruction errors. Because of this difference, REAP performs better than CP, as we will show in the experiments.

3.3 Experiments

We conducted the experiments with VGG16 [37] on ImageNet [7]. We also evaluated with other models/datasets that could not included in this paper due to page limitation. They can be found in [22], [23].

3.3.1 Datasets

ImageNet is a large scale dataset for 1,000 classes image classification [7]. It has approximately 1.2M images for training, 50K images for validation, and 100K images for testing. Following the former works, we used the validation images as the test dataset, and did not use the official test images in our experiments. As each image has different resolution, we resized them so that the shorter side would become 256 pixels. Then, 224×224 random crop was applied to the training images, and 224×224 center crop was applied to the test images. The random horizontal flip was applied to the training images. We randomly selected 5K training images, and used them for encoding neuron behavior.

3.3.2 Models

VGG16 is a model that has 16 weight layers, including 13 convolutional layers and 3 fully connected layers. We used the original VGG16 model that was trained with ImageNet dataset. The convolutional layers are composed of 5 blocks that have 2 or 3 layers. For convenience, we call the X-th layer of the Y-th block *ConvY-X*. For fully connected layers, we call such as *FC1* and *FC2*. Architecture details are mentioned in Appendix **??**.

3.3.3 Results

We conducted the experiments with VGG16 on ImageNet. We pruned the convolutional layers until the FLOPs became $\times 0.5$ and $\times 0.2$. The pruned models were retrained for 10 epochs at 10^{-5} learning rate. The momentum was set to 0.9, the minibatch size was set to 128, and the dropout rate for fully connected layers was set to 0.5. For the pruning ratio setting in each layer, we followed the information provided in [17]'s authors in their Github repository [1]. The rest of the setups were set to the same values with [17].

The results are shown in Table 1. REAP performs consistently better than the existing methods. After retraining, we marginally outperform the other methods at $\times 0.5$ FLOPs. At $\times 0.2$ FLOPs, the existing methods suffer even larger accuracy drop than we do.

An important observation is that we only suffer 9.4% accuracy drop at $\times 0.2$ FLOPs before retraining. On the other hand, CP suffers 22.0% drop and ThiNet spoiled the model performance. This is because we use the consistent strategy for channel selection and reconstruction to preserve the performances of the pruned models. As we show better performances before retraining, we can achieve higher accuracy after retraining as well. To put this observation differently, REAP enables us to achieve a certain accuracy with fewer epochs of retraining, which means that we can save time and labors for retraining.

It is also worth noting that the model pruned by REAP, at $\times 0.5$ FLOPs, after retraining, is better than the original VGG16 model. This is most likely because we removed the redundant weights, the remaining weights had smaller chance of being trapped in the local minima during training.

Table 2 shows the results of computational time measurements. Even though CP is much faster than REAP, we can say that REAP is fast enough. It can finish computation within minutes even in *Conv4-1* and *Conv4-2* that are the largest layers of VGG16. We believe that up to 848 seconds for *Conv4-1* and *Conv4-2* is acceptable enough in practice, considering that REAP saves us time for retraining the pruned model and that the training typically takes much more time (e.g. 1 epoch takes over 8 hours on NVIDIA Geforce GTX 1080 Ti).

4. Pruning Ratio Optimizer

REAP is a powerful method for preserving layer-wise error. However, what we are more interested in is the accuracy of the pruned model than the layer-wise errors. And the relationship between the layer-wise errors and the accuracy degradation is not obvious. Therefore, in terms of preserving the accuracy of the pruned model well, we need to set proper pruning ratio (the ratio of neurons to be pruned) in each layer.

In this section, we present Pruning Ratio Optimizer (PRO), a method for optimizing the pruning ratio in each layer based on the error in the final layer of the model. In PRO, we repeat the following steps until the pruned model becomes fast (and/or small) enough:

1) Select the the most redundant layer.

2) Prune a small number of neurons in the selected layer.

For evaluating the redundancy, we try to perform pruning in each layer with several pruning ratios, and observe the error in the final layer of the pruned model, as shown in Fig. 5. The layer where pruning will have the smallest impact on the outputs in the final layer is selected, and some neurons are pruned in that layer. After some iterations, the pruning ratio in each layer will be properly tuned.

It is worth noting that PRO has to be combined with REAP, even though other layer-wise pruning methods that conduct reconstruction, such as CP [17] and ThiNet [30], can also be used. This is because REAP is the best method for preventing the (layer-wise) error. As far as the pruned model retains close to its original accuracy, we can say that more neurons can still be pruned. On the other hand, with other pruning methods, the model being pruned easily suffers significant degradation. After significant degradation, we cannot judge if more neurons can be pruned. Therefore, we can optimize the pruning ratios more properly if we use REAP.

4.1 Related works

For pruning ratio optimization with a layer-wise method, He et al. proposed AutoML Model Compression (AMC), a method to optimize pruning ratio based on reinforcement learning [16]. They show that the accuracy of the pruned model can be preserved better if they optimize pruning ratios with AMC than if they do by human hands. Although, AMC has some weaknesses:

- Reinforcement learning itself is computationally expensive, because it requires us to perform pruning quite a lot of times with various pruning ratio settings.
- One still needs to tune lots of hyper-parameters related to reinforcement learning by human hands.

Therefore, it is desired to develop a novel method which is easier to use and less time-consuming.

The holistic pruning methods can be used for optimizing the pruning ratios as well. However, because most existing holistic methods do not perform reconstruction, the pruned models suffer significant accuracy degradation. Exceptionally, Optimal Brain Surgeon (OBS) [13] is a holistic method that performs reconstruction. However, as we already mentioned in Sec. 2, it is not realistic to apply OBS to large DNN models due to heavy computational cost. Structured Probabilistic Pruning (SPP) [39] conducts pruning by using dropout. As already mentioned in Sec. 2, SPP requires a lot of computation for pruning.

4.2 Pruning Ratio Optimizer

In this section, we explain Pruning Ratio Optimizer (PRO). We first re-formulate REAP in order to make it easier to explain PRO. Then, we show the details of PRO.

4.2.1 Formulation of REAP

Let $n^{(k)}$ denote the number of neurons in the layer where pruning is performed, $\mathcal{I}^{(k)} = \{1, \cdots, n^{(k)}\}$ denote the set of neuron indices, $\boldsymbol{x}_i^{(k)}$ denote the i-th neuron's behavioral vector, $\boldsymbol{w}_i^{(k)}$ denote the weights going from the i-the neuron to the ones in the next layer, $Y^{(k)} = \sum_{i \in \mathcal{I}} \boldsymbol{x}_i^{(k)} \boldsymbol{w}_i^{(k)\top}$ denote the layer-wise outputs. REAP's neuron selection can be formulated as below.

$$\mathcal{J}^{(k)*} = \operatorname*{argmin}_{\mathcal{J}^{(k)}} \min_{\boldsymbol{w}_i^{(k)}} \left\| Y^{(k)} - \sum_{i \in \mathcal{J}^{(k)}} \boldsymbol{x}_i^{(k)} \boldsymbol{w}_i^{(k)\top} \right\|_{\mathrm{F}}^2, \quad (27)$$
$$\text{subject to } \left|\mathcal{J}^{(k)}\right| \leq (1 - p^{(k)}) \left|\mathcal{I}^{(k)}\right|,$$

where $\mathcal{J}^{(k)}$ denotes the set of the remaining neurons' indices and $p^{(k)}$ denotes the pruning ratio. Note that we obtain the solution of Eq. (27) by solving Eq. (10) sequentially. REAP's neuron selection algorithm presented in Sec. 3 can find a better solution of this problem than other layer-wise pruning methods, such as [17].

Although REAP is good at preserving the original layer-wise outputs, it is not obvious how much this layer-wise error will have an impact on the model accuracy. Some amount of error in a layer may not affect model performance, although the same amount of error in another layer may lead to significant degradation. Moreover, pruning in a layer will change the outputs of the subsequent layers, which makes it difficult to optimize the pruning ratios in several layers simultaneously.

4.2.2 Pruning Ratio Optimizer (PRO)

We propose Pruning Ratio Optimizer (PRO) that can be combined with REAP (or other layer-wise pruning methods) for optimizing the pruning ratios. In PRO, we optimize the pruning ratio in each layer so as to minimize the error in the final layer of the model. Because it is difficult to solve this optimization problem analytically, we solve it in a greedy fashion. The idea of PRO is to select the most redundant layer and prune some neurons in the selected layer, repeatedly.

The procedures of PRO can be described as follows.

Step 1) Draw a curve of FLOPs reduction and the error in the final layer that is caused by performing pruning in each layer, as shown in Fig. 5 (a). In order to do this, we try to set the pruning ratio to various values, apply REAP, and observe the errors in the final layer. Note that pruning is conducted separately in each layer, and the pruned neurons and the updated weights have to be restored in this step.

Step 2) Set the threshold t_{err} to the error in the final layer. By using the curves drawn in Step 1), select the layer where the most FLOPs can be reduced at the cost of error of t_{err}. How to determine t_{err} properly will be explained later.

Step 3) Perform pruning in the selected layer until the error in the final layer reaches t_{err}.

Step 4) If enough amount of FLOPs have been reduced, terminate computation. Otherwise, go to Step 1). At the end, the pruning ratio in each layer will be properly tuned.

In order to avoid ambiguity, we provide more detailed descriptions for Step 1). Let $\mathcal{M}$ denote the model that have k^{+} layers and $\mathcal{D}$ denote the dataset used for pruning. The original outputs in the final layer are given by

$$Y^{(k^{+})} = \mathcal{M}(\mathcal{D}). \quad (28)$$

Then, we prune $l = p^{(k)} \left|\mathcal{I}^{(k)}\right|$ neurons in the k-th layer with REAP. Let $\mathcal{M}_{k,l}$ denote the model after pruning l neurons in the k-th

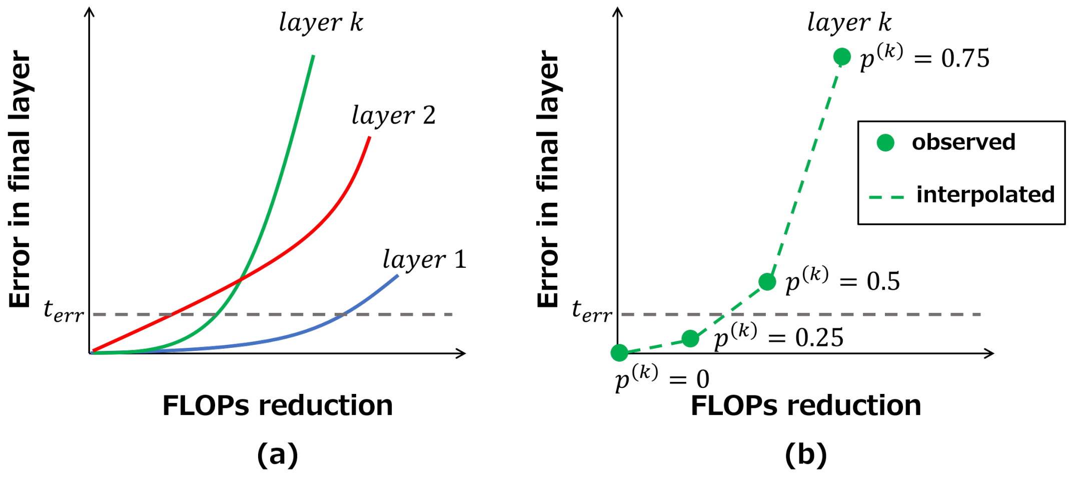

Fig. 5 (a) Illustration of the idea of PRO. In each layer, we try pruning with several pruning ratios and observe the error in the final layer. Then, we set the error threshold t_{err}, select the layer where the most FLOPs can be reduced at the cost of error of t_{err} (In this case, *layer*1 will be selected.), and perform pruning in the selected layer. We repeat these procedures several times until the inference with the pruned model becomes fast enough. (b) The strategy for efficient layer selection. Drawing precise curves is computationally intensive, as it requires us to conduct pruning and error observation repeatedly. Therefore, we set $p^{(k)}$, the pruning ratio in the k-th layer, to a few values (In this example, $p^{(k)} = 0, 0.25, 0.5, 0.75$.), conduct pruning, and observe the error in the final layer. We perform linear interpolation between the observed points.

layer. Then, the error in the final layer becomes

$$\Delta Y^{(k^+)}_{k,l} = Y^{(k^+)} - \mathcal{M}_{k,l}(\mathcal{D}). \tag{29}$$

We also need to compute FLOPs reduction achieved by pruning. By pruning l neuron in the k-th layer, the amount of reduced FLOPs is given by

$$\Delta o^{(k)} = l\left(n^{(k-1)} + n^{(k+1)}\right). \tag{30}$$

We draw a curve of $\left\|\Delta Y^{(k^+)}_{k,l}\right\|^2_{\mathrm{F}}$ and $\Delta o^{(k)}$. This has to be repeated for each $k \in \{1, \cdots, k^+ - 1\}$.

The remaining question is how to determine the threshold t_{err} in Step 2). With extremely small t_{err}, FLOPs reduction in each layer corresponding to the error of t_{err} will be close to zero, and we will not be able to prune any neuron in any layer. With too large t_{err}, all the neurons in the selected layer will be pruned. In order to avoid these situations, we induce a threshold t_{flops} for FLOPs to be reduced at each iteration. We first set a very small value to t_{err}, and select a layer that has the largest $\Delta o^{(k)}$ at the cost of error of t_{err} in the final layer. If $\Delta o^{(k)}$ in the selected layer is no smaller than t_{flops}, we go to Step 3). Otherwise, we increase t_{err} a little and repeat Step 2).

It is worth noting that because of REAP's high ability of preserving the original layer-wise outputs, pruning a small number of neurons in a layer barely changes the layer-wise outputs significantly, and thus, the final layer's outputs are not affected significantly as well. Therefore, in Step 2), we normally select several layers where we conduct pruning in Step 3). Then, we can reduce FLOPs more efficiently at each iteration, which saves the computational cost for pruning ratio optimization with PRO.

4.2.3 Strategy for more efficient optimization

We still have a problem with PRO, which is the large computational cost for Step 1). In order to draw precise curves of error and FLOPs reduction such as Fig. 5 (a), we need to compute Eq. (29) each time we prune a neuron, which is computationally intensive. Thus, we draw rough curves such as Fig. 5 (b) in the following scheme.

a) Set the pruning ratio in the k-th layer $p^{(k)}$ (Then, $l = p^{(k)}\left|\mathcal{I}^{(k)}\right|$ neurons will be pruned.) to some value, compute corresponding $\left\|\Delta Y^{(k^+)}_{k,l}\right\|^2_{\mathrm{F}}$ and $\Delta o^{(k)}$ by using Eq. (29) and Eq. (30). This step has to be repeated a few times, with several values of $p^{(k)}$ (e.g. $p^{(k)} = 0, 0.25, 0.5$).

b) Plot $\Delta o^{(k)}$ and $\left\|\Delta Y^{(k^+)}_{k,l}\right\|^2_{\mathrm{F}}$, as shown in Fig. 5. As we have only a few dots on the plot, we perform linear interpolation between the dots so that the error ($\left\|\Delta Y^{(k^+)}_{k,l}\right\|^2_{\mathrm{F}}$) corresponding to an arbitrary value of $\Delta o^{(k)}$ can be estimated.

4.2.4 Algorithm

The procedures of PRO are summed up in Algorithm 2. Here, we assume that REAP is employed for pruning.

4.3 Experiments

Although we evaluated PRO with several benchmark datasets and several models, we show one of them in this paper due to page limitation. We implemented PRO with Python 3.6.9 and Pytorch 1.0.0. All the experiments were done on Intel Core-i9 9900K CPU and a single board of NVIDIA Titan RTX GPU.

4.3.1 Setups

We performed pruning until the FLOPs would become approximately ×0.2 of the original VGG16 model.

The baseline method is AMC [16]. AMC is a reinforcement learning-based method for pruning ratio optimization. Basically, PRO is combined with REAP, and AMC is combined with a layer-wise pruning method named CP [17]. For fair comparison of PRO and AMC, we also evaluated the combination of PRO and CP. In

Table 3 VGG16 on ImageNet. The top-5 accuracy are reported (The greater, the better.). In this table, "rt" stands for "retraining", "uniform" means that the pruning ratio was set to the same value for all the layers. The baseline accuracy of the original VGG16 model is 89.5%.

Method	FLOPs	Acc. before rt	Acc. after rt	Time for optim.
PRO & REAP	×0.200	**80.5**%	**88.2**%	78,026 sec
PRO & CP	×0.203	73.6%	87.8%	71,840 sec
AMC & CP	×0.219	49.7%	85.8%	35,181 sec
uniform & REAP	×0.212	56.2%	87.1%	-

Algorithm 2

```
Input: Model $\mathcal{M}$, the number of selected layers for pruning in each iteration $m$, threshold for FLOPs reduction $t_{flops}$, threshold of error in the final layer $t_{err}$, a set $\mathcal{P}$ whose elements denote pruning ratios, a dataset used for pruning $\mathcal{D}$.
while The number of the FLOPs is not small enough do
    for $k = 1, \cdots, k^+ - 1$ do
        Feed $\mathcal{D}$ into $\mathcal{M}$ to compute $\{\boldsymbol{x}_i^{(k)} | i \in \mathcal{I}^{(k)}\}$ for each $k = \{1, \cdots, k^+\}$.
        for $p^{(k)} \in \mathcal{P}$ do
            $\mathcal{M}' \leftarrow \mathcal{M}$.
            Set pruning ratio to $p^{(k)}$ and perform pruning with REAP on the $k$-th layer of $\mathcal{M}'$.
            Compute corresponding $\left\|\Delta Y_{k,l}^{(k^+)}\right\|_{\mathrm{F}}^2$ and $\Delta o^{(k)}$ by using Eq. (29) and Eq. (30), where $l = p^{(k)} \left|\mathcal{I}^{(k)}\right|$.
        end for
        Make a plot of $\left\|\Delta Y_{k,l}^{(k^+)}\right\|_{\mathrm{F}}^2$ and $\Delta o^{(k)}$, as shown in Fig. 5. Perform linear interpolation between the plots.
    end for
    $t'_{err} \leftarrow t_{err}$.
    while $\Delta o^{(k)}$ in the selected layer(s) is smaller than $t_{flops}$ do
        Select $m$ layer(s) with the largest $\Delta o^{(k)}$ at $\left\|\Delta Y_{k,l'}^{(k^+)}\right\|_{\mathrm{F}}^2 = t'_{err}$, where $l' = p'^{(k)} \left|\mathcal{I}^{(k)}\right|$.
        Compute corresponding pruning ratio $p'^{(k)}$ in the selected layer(s).
        $t'_{err} \leftarrow z t'_{err}$, where $z$ is arbitrary value greater than 1.
    end while
    Perform pruning on the selected layers of $\mathcal{M}$, with $p'^{(k)}$ pruning ratio for the $k$-th layer.
end while
```

addition, we applied REAP with uniform pruning ratio settings in all the layers.

As shown in Algorithm 2, the hyper-parameters in PRO are as follows. m is the number of layers to be selected in each iteration, t_{err} is the threshold of the error in the final layer, t_{flops} is the amount of FLOPs that should be reduced at each iteration, and $\mathcal{P}$ is the set whose elements are the pruning ratios and are substituted to $p^{(k)}$. We set $m = 3$, $t_{err} = 10^{-10}$, $t_{flops} = 2 \times 10^8$ (For reference, the original VGG16 model has 1.547×10^{10} FLOPs.), and $\mathcal{P} = \{0, 0.125, 0.25, 0.375, 0.5\}$.

Regarding to AMC, we could not find some important experimental information in [16]. In order to be fair, we evaluated AMC by ourselves using the source code provided by [16]'s authors*1.

The pruned models were fine-tuned for 10 epochs with 10^{-5} learning rate. The momentum was set to 0.9, the mini-batch size was set to 128, and the dropout rate in the fully connected layers was set to 0.5. For the rest of training setups, we followed [37].

4.3.2 Results

We performed pruning with the pruning ratio optimization. The results are summarized in Table 3, and the discussions are as follows.

4.3.2.1 Comparison to the case of uniform pruning ratio

Compared to the the case of uniform pruning ratios in all the layers, we could make the accuracy degradation much smaller. Especially, the accuracy degradation was smaller by over 23% by using PRO, at approximately ×0.2 FLOPs ratio, before retraining.

The accuracy of the pruned model after retraining was better when using PRO. This is because 1) By using PRO, we can preserve the accuracy of the pruned model well, which means that we can start retraining with the models that have been less damaged; 2) The pruning ratio for each layer has been optimized even without retraining.

4.3.2.2 Comparison to AMC

We then discuss the comparison of PRO & CP and AMC & CP. As shown in Table 3, PRO could outperform AMC significantly. PRO suffers 15.9% accuracy degradation at ×0.203 FLOPs ratio without retraining, while AMC suffers 39.8% degradation at ×0.219 FLOPs rate. After retraining, PRO still suffers smaller degradation than AMC by 2.0%.

One thing that should be noted is the implementation difference of PRO and AMC. In PRO, each time we perform pruning in a layer, we encode the neuron behaviors in all the layers again. After pruning in a layer, it affects the neuron behaviors in other layer, and we cannot perform pruning properly without re-encoding them. Even though the optimization schemes of PRO and AMC are totally different, re-encoding of neuron behaviors is important in AMC as well for reconstruction. However, in their implementation, they encode the neuron behaviors in all the layers only in the beginning, and keep using those initial neuron behaviors to the end in order to shorten time for pruning ratio optimization.

Then, what if we conduct re-encoding for neuron behaviors in AMC? We tried to apply AMC while re-encoding the neuron behaviors. It took 1.7M sec (approximately 20 days) for pruning ratio optimization. However, the accuracy before retraining improved only 0.6% (49.7% to 50.3%), and the accuracy after retraining dropped by 0.2% (86.8% to 86.6%). After all, re-encoding the neuron behaviors did not work for improving the performance of AMC.

4.3.2.3 Analyses on optimized pruning ratio in each layer

Then, why the performance of AMC was worse than PRO? Fig. 6 shows the pruning ratio in each layer of the VGG16 model. The rough trend of both PRO and AMC is that they set higher pruning ratios to the layers closer to the input side and lower pruning ratios to the layers closer to the output side.

A remarkable observation is that PRO does not prune a lot in *Conv5-1* and *Conv5-2* layers, while AMC does. Actually, it is known that these layers are not redundant and pruning them leads

*1 https://github.com/mit-han-lab/amc

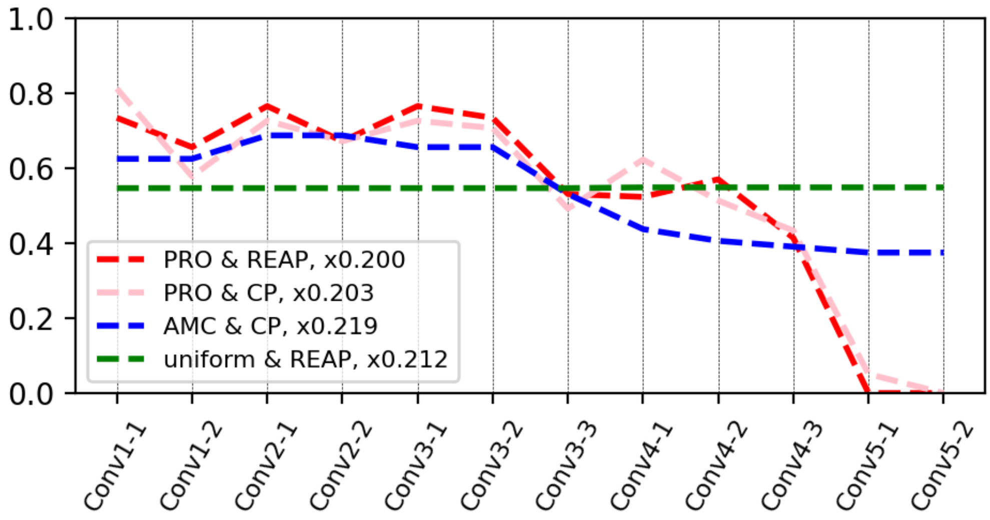

Fig. 6 Results of pruning ratio optimization for VGG16. Both PRO and AMC tend to set higher pruning ratio to the layers on the input side and lower pruning ratio to the layers on the output side. The difference is that PRO does not prune Conv5-1 and Conv5-2 layers a lot, while AMC does. As reported in several literatures, such as [17], [30], pruning these layers leads to significant degradation. And our PRO successfully avoids pruning these layers.

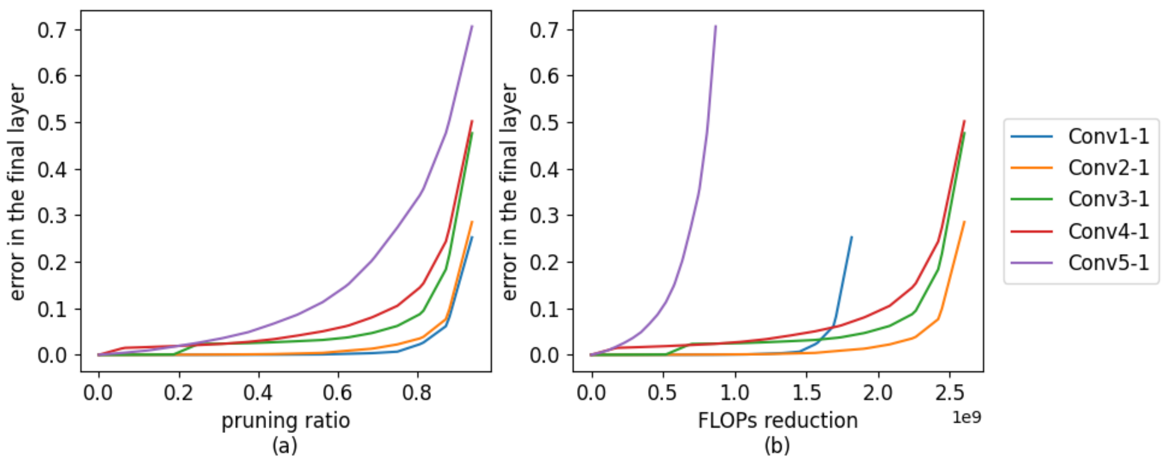

Fig. 7 (a) Relationship of the error in the final layer and pruning ratio in each layer. (b) Relationship of the error in the final layer and FLOPs reduction in each layer, which we actually use for selecting the layer to be pruned.

to significant degradation [17], [30]. PRO could successfully find out that these layers should not be pruned and eventually set zero or very low pruning ratios to them. On the other hand, AMC pruned a lot in these layers, which ended up in significant degradation.

We also investigated how the error in the final layer responses to the pruning ratio in each layer. We used REAP to prune *Conv1-1*, *Conv2-1*, *Conv3-1*, *Conv4-1*, and *Conv5-1* layers, with various pruning ratios, and observed the error in the final layer. The result is shown in Fig. 7.

Fig. 7 (a) shows the relationship of the error in the final layer and the pruning ratio in each layer, and Fig. 7 (b) shows a similar graph with FLOPs reduction in the horizontal axis. We can see clearly different trends between the layers. In the *Conv5-1* layer, the error increases more rapidly than the other layers. Thus, by observing the relationship of pruning ratio (FLOPs reduction) and the error directly, we can get the insight that we should not perform pruning a lot in *Conv5-1*.

Why did AMC set higher pruning ratios to the *Conv5-1* and *Conv5-2* layers? It is implied that AMC's reinforcement learning-based algorithm was simply not capable of evaluating the redundancy of the layers individually. As it performs pruning in all the layers simultaneously, it cannot evaluate the impact of the pruning ratio in each layer on the accuracy directly.

5. Serialized Residual Network

With REAP and PRO that we presented in Sec. 3 and 4, we can conduct pruning on pretrained large DNN models, so as to make them more efficient and preserve their performances simultaneously. On the other hand, there is an important point that we have not discussed so far, which is the limitation of structured pruning for ResNet. In this section, we discuss this limitation and present its solution.

In the recent developments of Computer Vision, the contribution of Residual Network (ResNet) [14] has been remarkable. In the competition of large scale image recognition [7], ResNet significantly outperformed the models that had been developed before ResNet, such as VGG [37]. It is widely believed that the key of ResNet is the architecture with *identity shortcuts*. ResNet architecture is composed of the stacked blocks that are called *ResNet blocks* With identity shortcuts, the convolutional layers are trained so that the optimal residual of the feature maps is learned. This architecture makes it possible to train a very deep model effectively and stably. This is why ResNet could show a record-breaking performance at that time [14].

However, this architectural feature of ResNet is inconvenient from the perspective of structured pruning. The architecture of ResNet consists of the blocks with identity shortcuts, as shown in Fig. 8 (a). The feature maps go through convolutional layers and are added to the ones coming through the identity shortcut. At this addition, the dimensions of two inputs must match, which means that we cannot prune the layers connected to the identity shortcuts. This limitation is crucial, because ResNet architecture is employed in various models for various tasks, such as object detection, segmentation, and so on.

Therefore, we propose a technique to transform ResNet into a serial network which we refer to by *Serialized Residual Network* (SRN). In Fig. 8 (a) and (b), we show a ResNet block and an equivalent SRN block. By building the kernels in the SRN block by concatenating the kernels taken from ResNet and the ones that conduct identity mapping, identity shortcut can be emulated by the SRN block. In this way, the ResNet model is equivalent to the SRN model whose weights are partially fixed to conduct identity mapping.

Although SRN model has more FLOPs than the ResNet model, it is much easier to be accelerated by pruning. Since the SRN model has a serial architecture, we can prune any layers and reduce the computational cost drastically at the cost of relatively small degradation.

The problem is that retraining the SRN model in the naïve way often ends up in no improvement or even degradation. The SRN model suffers some optimization problems caused by having both the optimized weights and the unoptimized weights. In order to avoid this problem, we also propose the training scheme dedicated for SRN.

It is also worth noting that our contribution is not limited to ResNet. Other types of the DNNs that have branched architectures, such as GoogLeNet [38] and so on, can be emulated by the serial networks, and thus, the discussions in this paper are applicable to those networks.

5.1 Serialized Residual Network (SRN)

In this section, we show how to build the SRN block that emulates the ResNet block, and explain our training strategy for the SRN models.

5.1.1 How to build SRN that emulates ResNet

Fig. 8 illustrates the ResNet block and the equivalent SRN block, where we omit the batch normalization layers for simplicity.

Let $\otimes$ denote convolutional operation, z denote ReLU function and $X \in \mathbb{R}^{d\times n\times h_w\times h_h}$ denote the feature map, where d denotes the batch size, n, h_w and h_h denote the number of channels, the width and the height of X, respectively. The operations in the ResNet block can be written as follows:

$$Y_A = W_A \otimes X, \tag{31}$$
$$X_A = z(Y_A), \tag{32}$$
$$Y_B = W_B \otimes X_A + X, \tag{33}$$
$$X_B = z(Y_B), \tag{34}$$

where $W_A, W_B \in \mathbb{R}^{n\times n\times g_w\times g_h}$ denote the kernel weights, g_w and g_h are the width and the height of the kernel. The feature maps X_A, X_B, Y_A, and Y_B are $d \times n \times h_w \times h_h$ tensors.

We reproduce these operations with the SRN block. In the SRN block, the operations are as follows.

$$Y'_A = W'_A \otimes X, \tag{35}$$
$$X'_A = z(Y'_A), \tag{36}$$
$$Y_B = W'_B \otimes X'_A, \tag{37}$$
$$X_B = z(Y_B). \tag{38}$$

In Eq. (35), $W'_A \in \mathbb{R}^{n\times 2n\times g_w\times g_h}$ consists of 2 sub-tensors, W_A and $I \in \mathbb{R}^{n\times n\times g_w\times g_h}$, where I is the kernel that conducts identity mapping ($I \otimes X = X$). Then, the output Y'_A is composed of 2 sub-tensors that are identical to Y_A and X, as shown in Fig. 8.

In Eq. (36), Y'_A is fed into z, and the output X'_A is obtained. Assuming that X is already the output of ReLU in the previous block and that every entry of X is no less than 0 (This assumption basically holds true because ResNet usually has ReLU at the end of each block.), X'_A still contains the sub-tensor that is identical to X.

The kernel $W'_B \in \mathbb{R}^{2n\times n\times g_w\times g_h}$ in Eq. (37) is built by concatenating W_B and I so that the convolution and the addition in Eq. (33) are reproduced with a single convolution. Then, the output will be identical to Y_B, and the final output of this block will be identical to X_B.

In this way, we can build the SRN block that precisely reproduces the operations of the ResNet block.

It should be noted that the nonlinear function z must be ReLU for the ResNet block to be emulated by the SRN block. Thus, the discussions in this paper may not be valid for some modified ResNet models with other types of activation, for example, Sigmoid, Tangent Hyperbolic, and so on. However, this limitation is not very important, because ReLU is used as standard for the modern DNNs.

5.1.2 Training strategy

The scheme for producing the SRN model is as follows.

1) Transform the pretrained ResNet model into an equivalent SRN model that has the fixed weights to reproduce the identity shortcuts.

2) Unfix the fixed weights and conduct training.

In Step 2), what we want to do is to train the whole weights of the SRN model, including the previously fixed ones. Although, we observe that the naïve training often ends up in no accuracy

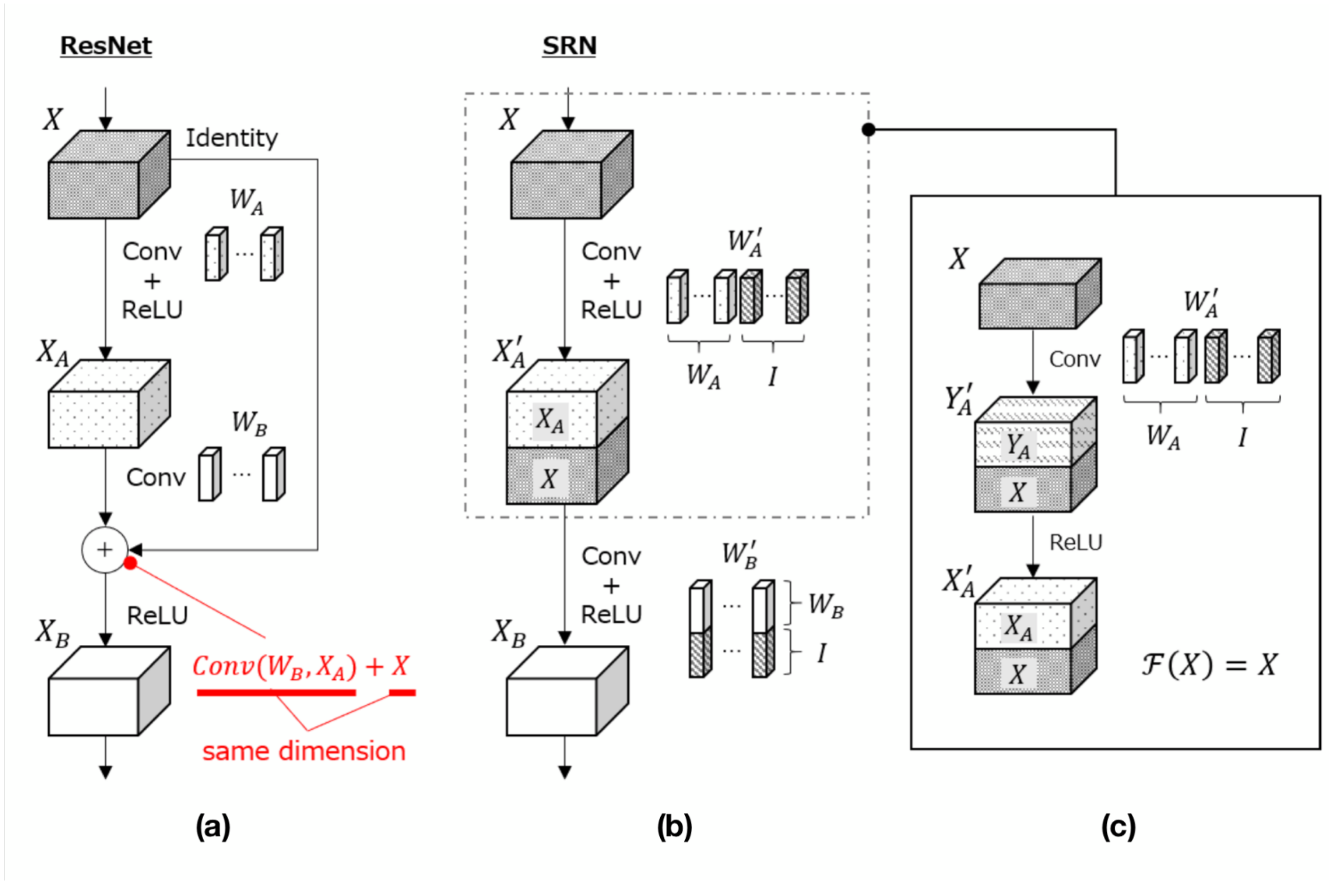

Fig. 8 This figure illustrates the concept of SRN. (a) The conventional ResNet block. (b) The SRN block that emulates ResNet. (c) The detailed illustration of operations in first convolution and ReLU activation of the SRN block.

improvement or even degradation. Regarding to this observation, we hypothesize two problems and suggest the corresponding countermeasures. One problem is the degradation caused by training the SRN model having the well-optimized weights taken over from the ResNet model and the unoptimized weights that are initially fixed for identity mapping. Another problem is the *side effect* of L2 regularization.

5.1.2.1 Problem caused by having both pretrained weights and untrained weights

Assume that w_1 is the pretrained weight taken over from ResNet, and w_2 is the untrained weight that was previously fixed for identity mapping. Fig. 9 illustrates the cost function f in the weight space spanned by w_1 and w_2, and the sketches of f over w_1 and w_2 around the point $P(\alpha, \beta)$ that represents the current weight values. We assume that P is already near from the optimal point, since it is the result of the pre-training of the ResNet model. If we train these weights, w_2 may have a steep gradient and be updated significantly, because w_2 has not been optimized yet, while w_1 is an optimized weight and is likely to have a gentle gradient. Then, we may move from the current point P to a far point P', and w_1 and w_2 may start to converge toward the sub-optimal point that is near from P', which means the training fails.

The naïve solution for this problem would be reducing the learning rate, although it would require quite a lot of iterations to converge and is computationally inefficient.

We propose AUWT standing for *Alternately Unfixing Weights and Training*. Assuming that this problem is more likely to happen when we have too many untrained weights that may have steep gradients, we repeatedly unfix the weights partially and conduct training, in order to limit the number of the untrained weights to be trained at the same time.

For instance, we conduct AUWT in the following steps.

1) Unfix the fixed weights in the first SRN block and train the model for 1 epochs.

2) Go to the second block and do the same. It will be repeated till the final SRN block.

5.1.2.2 Side effect of L2 regularization

In many cases, we use L2 regularization to stabilize the training on the neural networks. However, L2 regularization can cause a *side effect* when we train the SRN model.

We explain the *side effect* of L2 regularization with a fully connected layer, as the same discussion is valid for convolutional layers. In the fully connected layer, the weights for identity mapping is represented by an identity matrix E. Let e_{ij} denote the (i, j) entry of E, f denote the loss function, a denote the learning rate, and b denote the weight decay (the coefficient on regularization term). By feeding some training samples into the model, e_{ij} is updated by $e_{ij} + \delta e_{ij}$, where δe_{ij} is given by

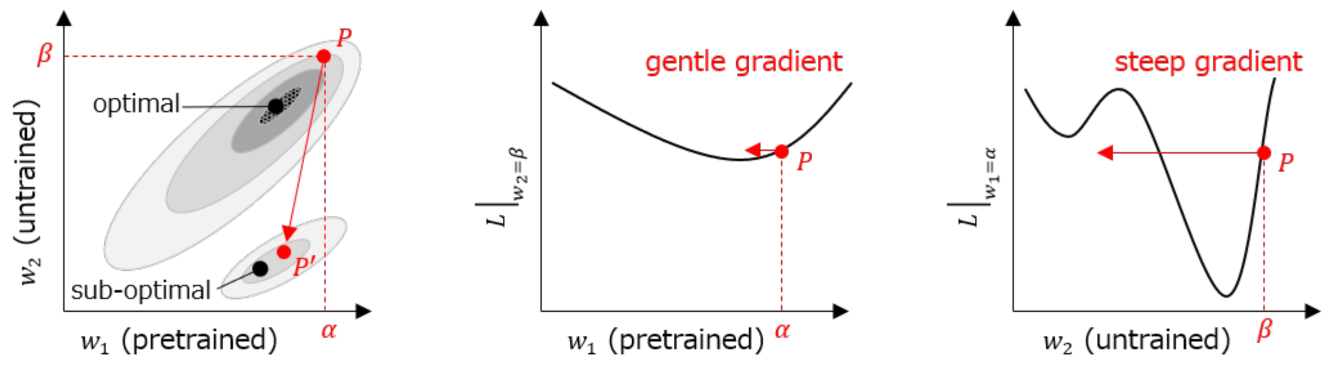

Fig. 9 This figure illustrates the problem caused by training SRN model having a pretrained weight and an untrained weight. Left: The contour map of the loss f with respect to the pretrained weight w_1 and the untrained weight w_2. Center: The graph of $f|_{w_2=\beta}$ with respect to w_1. Right: The graph of $f|_{w_1=\alpha}$ with respect to w_2. The untrained weight w_2 may have a steep gradient and be updated drastically by training, while the pretrained weight w_1 is likely to have a gentle gradient. Then, we may move from the current point P, which we assume is close to the optimal point, to a far point P'. Then, w_1 and w_2 may converge toward the sub-optimal point.

$$\begin{aligned}\delta e_{ij} &= -a\frac{\partial}{\partial e_{ij}}\left(f + \frac{b}{2}\sum_{k,l} e_{kl}^2\right) \\ &= -a\left(\frac{\partial f}{\partial e_{ij}} + b e_{ij}\right).\end{aligned} \tag{39}$$

Therefore, the diagonals of E tend to be strongly affected by the L2 regularization term due to their large initial values ($e_{ii} = 1$), while the rest of the weights are initially equal to 0 ($e_{ij} = 0 | i \neq j$) and are not significantly affected by L2 regularization at least in the beginning of training.

When we train the SRN model, we need to optimize the weights initialized to either 0 or 1 at the same time. In such a case, the weights initialized to 1 will be updated drastically due to the L2 regularization. Then, similarly with the problem illustrated in Fig. 9, we may move away from the optimal point in the weight space, and the weights may converge toward the sub-optimal point.

Inspired by [20], we suggest *Elastic Weight Regularization* (EWR) to prevent the *side effect* of L2 regularization. Instead of penalizing the L2 norm of the weights, we penalize the L2 norm of the difference from the initial weight values. This is formalized as follows.

$$\begin{aligned}\delta e_{ij} &= -a\frac{\partial}{\partial e_{ij}}\left(f + \frac{b}{2}\sum_{k,l}\left(e_{kl} - e'_{kl}\right)^2\right) \\ &= -a\left(\frac{\partial f}{\partial e_{ij}} + b\left(e_{ij} - e'_{ij}\right)\right),\end{aligned} \tag{40}$$

where e'_{ij} denotes the initial value of e_{ij}. EWR prevents the weights from being too different from the initial values. With EWR, the weights initialized to 1 are not affected by the regularization term too strongly, and thus the *side effect* of L2 regularization can be avoided.

The possible drawback of EWR is the initial value dependency. As the regularized weights cannot be so different from their original values, the training result strongly depends on the initial weight values. Although, we suppose that this is not a problem when training the SRN model converted from ResNet counterpart. If the ResNet model was trained successfully, then it is

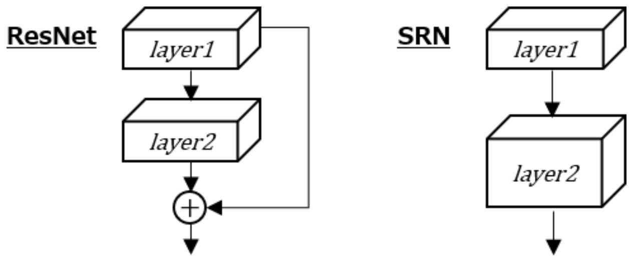

Fig. 10 The illustration of the ResNet block and the SRN block. Due to serialization, *layer2* of SRN has an increased number of channels.

intuitively reasonable to assume that its trained weights are not bad initial values. EWR improves SRN training compared to the normal L2 regularization.

5.2 Experiments

We conducted experiments to verify SRN. We implemented the proposed method with Python 3.6 and Pytorch 1.0 [34].

We evaluated SRN's ability of facilitating pruning, with the CenterNet [9] model that has ResNet-18 backbone. We transformed this backbone to SRN-18, perform pruning with REAP, and evaluated the performance of the pruned models.

We also measured the inference time per image of each model deployed on NVIDIA Jetson Nano [2], using camera demo mode of the TensorRT implementation provided in [3]. Jetson Nano is a device designed for neural network inference, and it is widely recognized/used in the industry and research.

5.2.1 Dataset

MS-COCO is a popular large dataset for object detection [27]. It contains approximately 82K training images and 40K test images and 80 object classes. All the images were Following augmentation settings in [9], training and evaluation were performed on 512 × 512 resolution. We applied random scaling (scaling factor was 0.6 to 1.3), and random horizontal flip to the training images. We used randomly selected 5K images for pruning.

5.2.2 Models

We converted ResNet-18 backbone to SRN-18 that has the fixed weights, and then unfix them from the shallower side. As we

Table 4 The results on CenterNet.

Backbone	mAP	FLOPs	Inf. time (msec)
ResNet-18 (baseline)	0.274	×1	131
ResNet-18-pruned (A)	0.261	×0.75	94
SRN-18-pruned (A)	**0.272**	×0.75	**91**
ResNet-18-pruned (B)	0.248	×0.5	82
SRN-18-pruned (B)	**0.262**	×0.5	**81**
ResNet-18-pruned (C)	0.183	×0.25	67
SRN-18-pruned (C)	**0.239**	×0.25	**57**

unfixed the weight in a block, we trained the model for 10 epochs at 1.25×10^{-5} learning rate, and performed pruning. We set the ratio of the pruned channels so that the FLOPs would become (A) 75%, (B) 50%, and (C) 25% of the original backbone. In SRN architecture, we can prune both *Layer1* and *Layer2* shown in Fig. 10. The ratio of *Layer1* and *Layer2* was tuned so that the number of remaining channels would become the same after pruning. After serializing and pruning all the blocks, we further trained the model for 20 more epochs at 1.25×10^{-5} learning rate which was divided by 10 at 10 epochs. The rest of the training setups were the same with [9].

For ResNet, we pruned only the layers without branched paths (*Layer2* in Fig. 10), since we cannot prune the layers connected to identity shortcuts. The training setups are the same with the SRN models.

Just for fair comparison with the original model (with ResNet-18 backbone), we further trained the pretrained original model, which results in no apparent improvement nor degradation.

5.2.3 Results

The results are reported in Table 4. As shown in Table 4, the pruned the SRN models could outperform the pruned ResNet models at the same FLOPs. For instance, At ×0.75 FLOPs rate, the SRN model shows a very small degradation, while the pruned ResNet model suffered more than 1% degradation in mAP. At larger FLOPs reduction, the performance gap of the ResNet model and the SRN model became even more significant. For reducing lots of FLOPs of the ResNet model, only the layers without identity shortcuts needed to be pruned, and the pruned layers with few remaining channels could not preserve the original performance. On the other hand, as any layer of the SRN model could be pruned, the model accuracy could be preserved better.

Even though the model with our *SRN-18-pruned (A)* backbone was competitive to the original model in mAP, we could achieve ×1.43 speed up. In this way, even though the SRN model has more FLOPs than the ResNet model, we can effectively make the SRN model faster by performing pruning.

6. Conclusion

In this thesis, we presented the methods for pruning the pretrained DNN models effectively. The proposed methods include two pruning methods, Neuro-Unification (NU) and Reconstruction Error Aware Pruning (REAP), and two facilitation methods for pruning, Pruning Ratio Optimizer (PRO), and Serialized Residual Network (SRN). These methods offer a practical solution for those who want to use large DNN models in resource-limited environments, such as smartphones, drones, and so on.

The biggest highlight of this thesis is REAP. REAP is theoretically well designed for preserving the accuracy of the model while reducing the model's redundancy. Among the methods that perform pruning based on layer-wise error, no other method is as good as REAP in terms of minimizing the error, to our best knowledge. Moreover, since REAP requires significant amount of computation for selecting the neurons to be pruned, we presented an efficient algorithm based on biorthogonal system. This algorithm is a novel usage of biorthogonal system.

PRO is a method to optimize the pruning ratio in each layer efficiently. The idea of PRO is to repeat selecting the layer where pruning will have the least impact on the outputs in the final layer of the model, and pruning some neurons in the selected layer. With REAP and PRO, we can conduct compression and optimize the architecture of the pruned model simultaneously.

SRN is a method to facilitate pruning on ResNet. The limitation of structured pruning on ResNet is that the layers with branched paths cannot be pruned. We noticed that the ResNet architecture is equivalent to a specific case of a serial architecture. Therefore, ResNet can be converted to an serial form which we call SRN. Once converted, we can reduce its redundancy drastically by pruning, as SRN has a serial architecture and its any layer can be pruned. SRN improves the practicality of pruning.

In the future, we plan to put the proposed method to practical use. By inputting the target model, some data, and overall compression ratio, the system identifies the layers to be pruned by performing structural analysis, serializes the ResNet architecture (if exists), optimizes the pruning rate with PRO, and conducts pruning with REAP.

References

[1] : Channel Pruning for Accelerating Very Deep Neural Networks, `https://github.com/yihui-he/channel-pruning`. (accessed on 08/01/2021).

[2] : NVIDIA JetPack SDK, `https://developer.nvidia.com/embedded-computing`. (accessed on 08/01/2021).

[3] : TensorRT-CenterNet, `https://github.com/CaoWGG/TensorRT-CenterNet`. (accessed on 08/01/2021).

[4] . Liu, . Wang, Foroosh, H., Tappen, M. and Penksy, M.: Sparse Convolutional Neural Networks, *Proceedings of Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 806–814 (2015).

[5] Aghasi, A., Abdi, A., Nguyen, N. and Romberg, J.: Net-Trim: Convex Pruning of Deep Neural Networks with Performance Guarantee, *Proceedings of Conference on Neural Information Processing Systems (NIPS)*, pp. 3177–3186 (online), available from ⟨http://papers.nips.cc/paper/6910-net-trim-convex-pruning-of-deep-neural-networks-with-performance-guarantee.pdf⟩ (2017).

[6] Courbariaux, M., Bengio, Y. and David, J.: BinaryConnect: Training Deep Neural Networks with binary weights during propagations, *Proceedings of Conference on Neural Information Processing Systems (NIPS)*, pp. 3123–3131 (online), available from ⟨http://papers.nips.cc/paper/5647-binaryconnect-training-deep-neural-networks-with-binary-weights-during-propagations.pdf⟩ (2015).

[7] Deng, J., Dong, W., Socher, R., Li, L., Li, K. and Fei-Fei, L.: ImageNet: A Large-Scale Hierarchical Image Database, *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 248–255 (2009).

[8] Dong, X., Chen, S. and Pan, S.: Learning to Prune Deep Neural Networks via Layer-wise Optimal Brain Surgeon, *Proceedings of Conference on Neural Information Processing Systems (NIPS)*, pp. 4857–4867 (2017).

[9] Duan, K., Bai, S., Xie, L., Qi, H., Huang, Q. and Tian, Q.: CenterNet: Keypoint Triplets for Object Detection, *Proceedings of IEEE International Conference on Computer Vision (ICCV)*, pp. 6569–6578 (2019).

[10] Elsken, T., Metzen, J. H. and Hutter, F.: Efficient Multi-Objective Neural Architecture Search via Lamarckian Evolution, *Proceedings of International Conference on Learning Representations (ICLR)* (2019).

[11] Han, S., Mao, H. and Dally, W. J.: Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huff-

man Coding, *CoRR* (2015).

[12] Han, S., Pool, J., Tran, J. and Dally, W.: Learning both Weights and Connections for Efficient Neural Network, *Proceedings of Conference on Neural Information Processing Systems (NIPS)*, pp. 1135–1143 (online), available from ⟨http://papers.nips.cc/paper/5784-learning-both-weights-and-connections-for-efficient-neural-network.pdf⟩ (2015).

[13] Hassibi, B., Stork, D. G. and Wolff, G. J.: Optimal Brain Surgeon and general network pruning, *IEEE International Conference on Neural Networks*, pp. 293–299 (1993).

[14] He, K., Zhang, X., Ren, S. and Sun, J.: Deep Residual Learning for Image Recognition, *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778 (2016).

[15] He, T., Fan, Y., Qian, Y., Tan, T. and Yu, K.: Reshaping deep neural network for fast decoding by node-pruning, *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 245–249 (2014).

[16] He, Y., Lin, J., Liu, Z., Wang, H., Li, L. and Han, S.: AMC: AutoML for Model Compression and Acceleration on Mobile Devices, *Proceedings of European Conference on Computer Vision (ECCV)*, pp. 784–800 (2018).

[17] He, Y., Zhang, X. and Sun, J.: Channel Pruning for Accelerating Very Deep Neural Networks, *Proceedings of IEEE International Conference on Computer Vision (ICCV)*, pp. 1389–1397 (2017).

[18] Hinton, G., Vinyals, O. and Dean, J.: Distilling the Knowledge in a Neural Network, *CoRR* (2015).

[19] Hsu, C., Chang, S., Juan, D., Pan, J., Chen, Y., Wei, W. and Chang, S.: MONAS: Multi-Objective Neural Architecture Search using Reinforcement Learning, *CoRR* (2018).

[20] Hu, W., Lin, Z., Liu, B., Tao, C., Tao, Z., Ma, J., Zhao, D. and Yan, R.: Overcoming Catastrophic Forgetting via Model Adaptation, *Proceedings of International Conference on Learning Representations (ICLR)*, (online), available from ⟨https://openreview.net/forum?id=ryGvcoA5YX⟩ (2019).

[21] Jaderberg, M., Vedaldi, A. and Zisserman, A.: Speeding up Convolutional Neural Networks with Low Rank Expansions, *Proceedings of British Machine Vision Conference (BMVC)* (2014).

[22] Kamma, K., Isoda, Y., Inoue, S. and Wada, T.: Behavior-Based Compression for Convolutional Neural Networks, *Proceedings of International Conference on Image Analysis and Recognition (ICIAR)*, pp. 427–439 (online), DOI: 10.1007/978-3-030-27202-9_39 (2019).

[23] Kamma, K. and Wada, T.: Reconstruction Error Aware Pruning for Accelerating Neural Networks, *Proceedings of International Symposium on Visual Computing (ISVC)*, pp. 59–72 (2019).

[24] Kandasamy, K., Neiswanger, W., Schneider, J., Poczos, B. and Xing, E.: Neural Architecture Search with Bayesian Optimisation and Optimal Transport, *CoRR* (2018).

[25] Krizhevsky, A., Sutskever, I. and Hinton, G.: ImageNet Classification with Deep Convolutional Neural Networks, *Proceedings of Conference on Neural Information Processing Systems (NIPS)*, pp. 1097–1105 (online), available from ⟨http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf⟩ (2012).

[26] LeCun, Y., Denker, J. S. and Solla, S. A.: Optimal Brain Damage, *Proceedings of Conference on Neural Information Processing Systems (NIPS)*, pp. 598–605 (online), available from ⟨http://papers.nips.cc/paper/250-optimal-brain-damage.pdf⟩ (1990).

[27] Lin, T., Maire, M., Belongie, S. J., Bourdev, L. D., Girshick, R. B., Hays, J., Perona, P., Ramanan, D., Dollár, P. and Zitnick, C. L.: Microsoft COCO: Common Objects in Context, *Proceedings of European Conference on Computer Vision (ECCV)*, pp. 740–755 (2014).

[28] Liu, H., Simonyan, K., Vinyals, O., Fernando, C. and Kavukcuoglu, K.: Hierarchical Representations for Efficient Architecture Search, *Proceedings of International Conference on Learning Representations (ICLR)*, (online), available from ⟨https://openreview.net/forum?id=BJQRKzbA-⟩ (2018).

[29] Liu, Z., Li, J., Shen, Z., Huang, G., Yan, S. and Zhang, C.: Learning Efficient Convolutional Networks Through Network Slimming, *Proceedings of IEEE International Conference on Computer Vision (ICCV)*, pp. 2736–2744 (2017).

[30] Luo, J., Wu, J. and Lin, W.: ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression, *Proceedings of IEEE International Conference on Computer Vision (ICCV)*, pp. 5058–5066 (2017).

[31] Mirzadeh, S., Farajtabar, M., Li, A., Levine, N., Matsukawa, A. and Ghasemzadeh, H.: Improved Knowledge Distillation via Teacher Assistant, *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, pp. 5191–5198 (2020).

[32] Molchanov, P., Tyree, S., Karras, T., Aila, T. and Kautz, J.: Pruning Convolutional Neural Networks for Resource Efficient Transfer Learning, *Proceedings of International Conference on Learning Representations (ICLR)* (2015).

[33] Park, S., Lee, J., Mo, S. and Shin, J.: Lookahead: A Far-sighted Alternative of Magnitude-based Pruning, *Proceedings of International Conference on Learning Representations (ICLR)*, (online), available from ⟨https://openreview.net/forum?id=ryl3ygHYDB⟩ (2020).

[34] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L. and Lerer, A.: Automatic differentiation in PyTorch, *NIPS-W* (2017).

[35] Pham, H., Guan, M. Y., Zoph, B., Le, Q. V. and Dean, J.: Efficient Neural Architecture Search via Parameters Sharing, *Proceedings of International Conference on Machine Learning (ICML)*, pp. 4095–4104 (2018).

[36] S. Srinivas, V. B.: Data-free parameter pruning for Deep Neural Networks, *Proceedings of British Machine Vision Conference (BMVC)* (2018).

[37] Simonyan, K. and Zisserman, A.: Very Deep Convolutional Networks for Large-Scale Image Recognition, *Proceedings of International Conference on Learning Representations (ICLR)* (2015).

[38] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A.: Going Deeper With Convolutions, *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9 (2015).

[39] Wang, H., Zhang, Q., Wang, Y. and Hu, H.: Structured Probabilistic Pruning for Convolutional Neural Network Acceleration, *Proceedings of British Machine Vision Conference (BMVC)* (2018).

[40] Xie, G., Wang, J., Zhang, T., Lai, J., Hong, R. and Qi, G.: Interleaved Structured Sparse Convolutional Neural Networks, *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 8847–8856 (2018).

[41] Xue, J., Li, J. and Gong, Y.: Restructuring of deep neural network acoustic models with singular value decomposition, *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*, pp. 2365–2369 (2013).

[42] Ye, J., Wang, L., Li, G., Chen, D., Zhe, S., Chu, X. and Xu, Z.: Learning Compact Recurrent Neural Networks With Block-Term Tensor Decomposition, *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 9378–9387 (2018).

[43] Yu, R., Li, A., Chen, C., Lai, J., Morariu, V. I., Han, X., Gao, M., Lin, C. and Davis, L. S.: NISP: Pruning Networks Using Neuron Importance Score Propagation, *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 9194–9203 (2018).

[44] Yu, X., Liu, T., Wang, X. and Tao, D.: On Compressing Deep Models by Low Rank and Sparse Decomposition, *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 7370–7379 (2017).

[45] Zhao, L., Hu, Q. and Wang, W.: Heterogeneous Feature Selection With Multi-Modal Deep Neural Networks and Sparse Group LASSO, *IEEE Transactions on Multimedia*, Vol. 17, No. 11, pp. 1936–1948 (2015).

[46] Zhu, M. and Gupta, S.: To prune, or not to prune: exploring the efficacy of pruning for model compression, *CoRR* (2017).

[47] Zhuang, Z., Tan, M., Zhuang, B., Liu, J., Guo, Y., Wu, Q., Huang, J. and Zhu, J.: Discrimination-aware Channel Pruning for Deep Neural Networks, *Proceedings of Conference on Neural Information Processing Systems (NIPS)*, pp. 881–892 (2018).

[48] Zoph, B. and Le, Q. V.: Neural Architecture Search with Reinforcement Learning, *CoRR* (2016).