AutoPruner: An End-to-End Trainable Filter Pruning Method for Efficient Deep Model Inference

Jian-Hao Luo\textsuperscript{a}, Jianxin Wu\textsuperscript{a,*}

\textsuperscript{a}National Key Laboratory for Novel Software Technology, Nanjing University, China

Abstract

Channel pruning is an important method to speed up CNN model’s inference. Previous filter pruning algorithms regard importance evaluation and model fine-tuning as two independent steps. This paper argues that combining them into a single end-to-end trainable system will lead to better results. We propose an efficient channel selection layer, namely AutoPruner, to find less important filters automatically in a joint training manner. Our AutoPruner takes previous activation responses as an input and generates a true binary index code for pruning. Hence, all the filters corresponding to zero index values can be removed safely after training. By gradually erasing several unimportant filters, we can prevent an excessive drop in model accuracy. Compared with previous state-of-the-art pruning algorithms (including training from scratch), AutoPruner achieves significantly better performance. Furthermore, ablation experiments show that the proposed novel mini-batch pooling and binarization operations are vital for the success of model pruning.

Keywords: Neural network pruning, Model compression, CNN acceleration

1. Introduction

Deep neural networks suffer from serious computational and storage overhead. For example, the VGG16 \cite{simonyan2014very} model, which has 138.34 million parameters, requires more than 30.94 billion floating-point operations (FLOPs) to recognize
a single $224 \times 224$ input image. It is impossible to deploy such cumbersome models on real-time tasks or resource constrained devices like mobile phones [2, 3].

To address this problem, many model compression or acceleration algorithms have been proposed [4, 2, 5, 6, 7, 8, 9, 10, 11, 12, 13]. Among these methods, pruning is an important direction. By removing unimportant neurons, the model size and computational cost can be reduced. We can roughly divide pruning approaches into 3 categories: connection level, filter level and layer level. A simple method is to discard neural connections according to the magnitude of their weights [4]. However, such an unconstrained pruning strategy will lead to an irregular network structure. It may slow down the actual inference speed even though the sparsity is high [14]. Hence, structured pruning, such as filter level pruning [15, 2, 3, 16, 17] or layer level pruning [5], is attracting more and more attentions in recent years. In filter level pruning, the whole filter will be removed if it is less important. Hence, the original network structure will not be damaged after pruning. Similarly, layer level pruning will remove the whole layer to reduce model complexity.

A typical pruning process is constituted by two major parts. The first part is pruning. As illustrated in the first row of Figure 1, most current methods adopt a three-stage pipeline to prune networks. They try to find a better evaluation criteria for measuring the importance of filters. Based on the importance score, several unimportant filters will be discarded. Then, the pruned model will be fine-tuned to recover its accuracy. This pipeline will be repeated layer-by-layer until all the layers have been pruned. Then in the second part, the pruned small model will be fine-tuned again. Hand-crafted importance evaluation criteria plays a crucial role for the success of network pruning. However, it is hard to find a perfect criterion that can work well on all networks and tasks. More importantly, evaluation and fine-tuning are two independent processing steps in this pipeline. Hence, here comes an interesting question: could fine-tuning be utilized to guide the selection of weak filters? In other words, can we prune models in an automatic way and break away from the dependence of hand-crafted rules?
In order to answer this question, we propose a novel end-to-end trainable method, namely AutoPruner, to explore a new way for CNN model pruning. By integrating filter selection into model fine-tuning, the network can select unimportant filters automatically. Our AutoPruner can be regarded as a new CNN layer, which takes the activation of previous layer as an input, and generate a unique binary code. A 0 value in the binary code means its corresponding filter’s activation will always be 0, hence can be safely eliminated. And unique means our AutoPruner is a static method, all the filters with zero indexes will be removed forever. After that, the newly added AutoPruner layer and unimportant filters will be discarded, leading to a pruned small model.

Experimental results on the fine-grained CUB200-2011 dataset [18] and the large-scale image recognition ILSVRC-2012 dataset [19] have demonstrated the effectiveness of the proposed AutoPruner. AutoPruner outperforms previous state-of-the-art approaches with a similar or even higher compression ratio. We also compared AutoPruner with a simple but powerful method: training from scratch. The result of this experiment reveals that our AutoPruner achieves better accuracy, which is really useful to obtain a more accurate small model.

The key advantages of AutoPruner are summarized as follows.
• **End-to-end trainable in a single model.** Filter selection (*i.e.*, importance evaluation) and model fine-tuning are integrated into a single end-to-end trainable framework. We empirically demonstrate that these two processing steps can promote each other. The model will select better filters automatically during fine-tuning. And, the gradients of filter selection are also helpful for guiding the training of previous convolution layers. In other words, *fine-tuning can be utilized to guide pruning, and gradually erasing weak filters (i.e., pruning) is really important to obtain a more accurate model (i.e., fine-tuning).*

• **Adaptive compression ratio and multi-layer compression.** We propose a novel loss function to ensure the sparsity of binary code could converge to a predefined compression ratio. But we encourage the network to determine the actual sparsity by itself, which will take both accuracy and compression ratio into consideration. And, we can compress multiple layers (or all layers) simultaneously to reduce training cost.

• **Good generalization ability.** The proposed method achieves better performance on multiple datasets and networks compared with previous state-of-the-art algorithms. Our method is easy to implement and can be extended to other deep learning libraries.

2. Related Work

Pruning is a classic method to reduce model complexity [4, 14, 20, 15, 21, 22, 2]. Compared with training the same structure from scratch, pruning a pretrained redundant model can achieve much better results [2, 23], because of the highly non-convex optimization problem in the model training stage. And, certain level of model redundancy is necessary to guarantee enough capacity during training. However, such a cumbersome model will slow down the running speed of model inference. And the model capacity is also too large when transfer to a much smaller dataset. Hence, there is a great need to remove the redundancy.
The most intuitive idea to evaluate neuron importance is based on the magnitude of its weight value. Han et al. [4, 24] proposed an iterative pruning method to discard small-weight connections which are below a predefined threshold. However, connection level pruning can lead to an irregular convolution, which needs a special algorithm or dedicated hardware for efficient inference, thus is hard to harvest actual computational savings. To address the weakness of non-structured random pruning, some structured sparsity learning algorithms have been proposed [7, 14, 25]. In these works, only groups of structured neurons, such as the whole channel or filter, will be pruned.

Recently, filter level pruning has drawn a significant amount of interests from both academia and industry. Luo et al. [2, 3] formally established filter pruning as an optimization problem, and removed the less important filters based on the statistics of the next layer. Similarly, He et al. [16] proposed a LASSO regression based method to select unimportant channels. Liu et al. [17] introduced channel scaling factors to denote the importance of each layer. Yu et al. [26] propagated the importance scores of the final responses to every neuron and formulate network pruning as a binary integer optimization problem. All of these methods are trying to find a better importance evaluation method.

However, three-stage pruning algorithms regard importance evaluation and model fine-tuning as two independent steps. We argue that combining them into a single step will be a better choice: the information flowed from uncompressed layers can be used to guide the selection of the current layer.

There are also some explorations to prune networks beyond the three-stage framework. Lin et al. [15] introduced a novel dynamic pruning method based on reinforcement learning. The network is dynamically pruned according to the output Q-value of a decision network. By contrast, our method is static, and the zero filters will be removed forever. He et al. [27] introduced AutoML for model pruning. They use reinforcement learning to efficiently sample the network design space. In [28], Singh et al. proposed a min-max framework for network pruning. Wang et al. [29] incrementally assign the regularization factors to different weights. Structure sparsity regularization is also used in [30].
They proposed the AULM algorithm to solve the specific optimization problem. Dong et al. [31] designed a low-cost layer to skip the computation of unimportant positions. However, the network is not pruned. Its actual implementation relies on highly-efficient BLAS libraries.

The proposed AutoPruner method requires the generated index code to be unique and binary. There are also some works that are closely related to our method. In [32], the authors also used 0-1 mask to select the best shift for each feature map. Their method is designed for shift networks, while we focus on more general CNN models. Similarly, Li et al. [33] introduce a binary global mask after each filter and the previously removed filters can be dynamically recovered. Huang et al. [5] adopted scaling factors to indicate the importance of each neuron, and formulate it as a joint sparse regularized optimization problem. Wang et al. [34] also introduced a pruning probability for each weight. These two methods are very similar to ours. The major difference is whether pruning information will participate in the training of previous layers or not.

We empirically demonstrate that the gradient of channel selection layer is also helpful for model training. In [35], Gao et al. proposed a similar framework as AutoPruner to learn the index code for each filter. However, their method is dynamic: the pruning is dependent on the current input. By comparison, our index code is unique and static, which can support parallel computing well.

Besides filter pruning, there are also many other methods to accelerate the inference speed or to reduce the model size, such as low-rank decomposition [36, 37], parameter quantization [38], knowledge distillation [39]. These methods are complementary to filter pruning, and can be combined with AutoPruner for further improvement.

3. Automatic Filter Pruner

In this section, we propose our adaptive end-to-end trainable filter pruning method: AutoPruner. We will give a comprehensive introduction to the AutoPruner pipeline and several important implementation details.
Figure 2: Framework of the proposed AutoPruner layer. Given a mini-batch of activation tensors, we use a new batch-wise average pooling and a standard max pooling to generate a single tensor. This tensor is projected into a \( C \)-dimensional vector via a fully-connected layer, where \( C \) is the number of channels. Finally, a novel scaled sigmoid function is used to obtain an approximate binary output. By gradually increasing the value of \( \alpha \) in the scaled sigmoid function, the output will gradually become a \( C \)-dimensional binary code. After training, all the filters and channels corresponding to zero index values will be pruned away to obtain a smaller and faster network. The newly added AutoPruner layer will be removed too.

### 3.1. The Proposed AutoPruner Pipeline

Figure 2 shows the framework of AutoPruner. AutoPruner can be regarded as an independent layer, whose input is the responses (after activation function) of a standard convolution layer. An approximate binary index code is generated by AutoPruner. We then use element-wise multiplication to combine it with the activation tensors. By gradually forcing the scaled sigmoid function to emit binary index codes, some channels in the activation tensors will gradually become all zero (i.e., they will be erased gradually). Hence, we can safely pruned these channels (or filters) away. Since AutoPruner is trained in an end-to-end manner, channel selection and model fine-tuning are combined together, and will promote each other during training.

After training, the binary index code is used for filter pruning. All the filters in the previous layer and all the channels in the filters of the next layer will be removed if their corresponding index values are 0. The newly added AutoPruner layer will be removed too. Hence, the pruned model has no difference in model structure with previous pruning method.
Next, we will go into details about the proposed AutoPruner. Our method consists of three major parts: pooling, coding and binarization. We will give a comprehensive introduction to them separately.

### 3.1.1. Pooling

We use $X \in \mathbb{R}^{N \times C \times H \times W}$ to denote the activation output of a convolution layer, with mini-batch size $N$, $C$ channels, $H$ rows and $W$ columns. First, batch-wise average pooling is used to aggregate all the elements among different images:

$$
X' = \frac{1}{N} \sum_{i=1}^{N} X_{i,:,:,:}.
$$

Note that the generated index code should be decided by the layer, not one example. In other words, we need the activations, which is computed from different images, to be transformed into a unique index code in one specific layer. The batch-wise average pooling mixes information of different images and is helpful in achieving the consistency of index codes among different images. The subsequent binarization technique in AutoPruner enables us to generate a unique index code for each layer.

Next, the pooled tensor $X'$ is fed into a standard max-pooling function with $2 \times 2$ filter size and stride 2 to reduce memory consumption. We find this reduction of information does not affect model accuracy significantly, but adding this step will save GPU memory consumption and training time in the coding stage. However, as shown later, a large spatial pooling operation (such as global average pooling) is harmful.

### 3.1.2. Coding and Binarization

Then, in the coding stage, the pooled tensor will be projected into a $C$-dimensional vector. A fully-connected layer, whose weights are denoted as $\hat{W} \in \mathbb{R}^{C \times (CH'W')}$, is used here to generate a $C$-dimensional vector, where $C$ is the channel number in input activations.

We require that the generated index code should be 0-1 values, which is essential in assuring the pruning quality. Combined with the mini-batch pooling
operation, binarization ensures that all the examples in a mini-batch could be
transformed to the same unique index code eventually.

We use a scaled sigmoid function to generate the approximate binary code:

\[ y = \text{sigmoid}(\alpha x), \tag{2} \]

where \( \alpha \) is scaling factor, which controls the magnitude of output values. As
illustrated in the top right corner of Figure 2, by gradually increasing the value of \( \alpha \), this scaled sigmoid function can generate approximate binary codes. When \( \alpha \)
is large enough, the approximate binary values will become 0-1 values eventually.
In others words, pruning is finished during fine-tuning, and which filter should
be pruned away is totally decided by the network itself.

Such a gradual binarization strategy is helpful for obtaining a more accurate
model. When some channels are becoming smaller (0.5 \( \to \) 0), the corresponding
filters will stop updating gradually. At the same time, other channels are
becoming larger (0.5 \( \to \) 1), which will force the network to pay more attention
to the preserved filters.

Another major benefit of binarization is that selection and fine-tuning is
now seamlessly integrated together. Selection can be finished during model fine-
tuning. If the binary code for one channel is 0, we know its activation values
will always be 0 for any input image. And, if the binary code for one channel
is 1, the activation values will not change in the element-wise multiplication
operation. Hence, after the codes become binary, removing all pruning block
and the pruned filters will not change the network’s prediction.

3.2. Sparsity Control and Loss Function

So far, we have introduced the whole pipeline of AutoPruner. The next
question is, can we control the sparsity of the output index code? In some real-
world application scenarios, the inference speed or model size is constrained.
For example, a scene segmentation network should return predictions within
50 ms in self-driving vehicles for safety consideration. These constraints can be
solved via a predefined compression rate.
To address this problem, we propose a simple yet efficient sparsity control regularized loss function. We adopt a uniform pruning profile (i.e., discard the same proportion of filters in each layer). Let $v$ denote the index code vector generated by AutoPruner, one of the most commonly used sparsity regularization methods is the convex relaxation $\ell_1$-norm, which is defined by $\|v\|_1$. However, $\ell_1$-norm cannot control the sparsity to an expected value. Noticing that $v$ is an approximate binary vector, we can use $\frac{\|v\|_1}{C}$ to denote the percentage of 1 approximately, i.e., the percentage of preserved filters. Here, $C$ is the length of vector $v$. Given a predefined compression rate $r \in [0, 1]$ (the percentage of preserved filters), we can formulate our loss function as:

$$L = L_{\text{classification}} + \lambda \left\| \frac{\|v\|_1}{C} - r \right\|_2^2. \quad (3)$$

The first term is a standard classification loss (e.g., cross-entropy loss), the second one controls the model compression rate, and $\lambda$ balances the relative importance between these two terms. Furthermore, its value is adaptively adjusted according to the current compression ratio:

$$\lambda = 100 \times |r_b - r|, \quad (4)$$

where $r_b$ is the real compression ratio, which is the percentage of ones in the current output $v$. “100” converts percentage values to a normal one. $\lambda$ is initialized by 10, and adaptively changed during model training. If the current compression ratio is far from our expected goal, $\lambda$ is relatively large. Hence, the model could pay more attention to change the sparsity of code index $v$. Once we have got the expected $v$, $\lambda$ will finally become 0, which means the network can focus on the classification task.

We want to emphasize that the actual compression rate is determined by the network itself. Our novel loss function can control the sparsity, but the actual value can still vary. For example, if we want to prune half of the filters and set $r = 0.5$. After training, the actual value of $r_b$ may be 0.52 or 0.48. Hence, the proposed AutoPruner can achieve an adaptive network compression.

Finally, we can append AutoPruner to multiple layers and compress them...
simultaneously. In practice, we find that pruning all layers once may lead to irreparable damage to model accuracy, especially for those numerical sensitive models like VGG16 [1] (there is no batch normalization layer in VGG networks). As a result, the accuracy can not be recovered via fine-tuning. In this situation, layer-by-layer pruning seems to be a better choice. And for a network with batch normalization layers (e.g., ResNet [40] or MobileNetV2 [41]), pruning multiple or all layers simultaneously is more encouraged because of its faster speed.

3.3. Initialization and Binarization Control

We have to emphasize that initialization plays an essential role in our framework. The initial value of code index $v$ is determined by three factors: input tensor, weight value and scaling factor $\alpha$. We use the standard initialization method for fully-connected weights. And the magnitude of input tensor is uncertain in different layers. Hence, adjusting the value of $\alpha$ is the best way to control the initial value of $v$.

There are a few observations that are worth discussing.

- When $\alpha$ is too large, $v$ will become binary quickly. In this case, the discarded filters are determined before training. Hence, AutoPruner will degenerate into random selection.

- When $\alpha$ is too small, $v$ may be difficult to or even impossible to converge into binary values. What is more troubling is that $v$ can be stuck in small values, i.e., all the elements are smaller than 0.5, but can never be pulled back around 1.

- Unfortunately, appropriate value of $\alpha$ can differ greatly in different layers due to the magnitude of the input. It is impossible to find an appropriate $\alpha$ that can work well for all layers.

Based on the above observations, we propose an efficient adjustment scheme to find an approximate value for $\alpha$. As discussed in Section 3.1.2, we should increase $\alpha$ gradually. Starting from the initial value $\alpha_{\text{start}}$, we linearly increase
its value and finally stop at $\alpha_{\text{stop}}$. Hence, this question is equivalent to find the values of $\alpha_{\text{start}}$ and $\alpha_{\text{stop}}$.

The first step is to find $\alpha_{\text{stop}}$ which can produce a true binary output in the scaled sigmoid function (Eq. 2). However, this value can vary greatly in different networks. Our method is to test several numbers until its outputs are all 0-1 values. Note that this step can be finished quickly before model fine-tuning: we only need to try several numbers. For example, in our internal test we find $\alpha_{\text{stop}} = 2$ is enough for VGG16 [1]. But for ResNet [40], $\alpha_{\text{stop}}$ should not be smaller than 100.

As for $\alpha_{\text{start}}$, we find AutoPruner is really robust to this value which is demonstrated in the ablation study section. We heuristically set it to 0.1. In order to avoid the influence of small $\alpha$, we adopt a simple yet efficient method. We will check the values of $v$ at the last epoch. If it is still far from convergence, $\alpha$ will be increased faster (e.g., $10 \times$), forcing it converge into binary values.

Algorithm 1 summarizes our control processes. Using such a simple strategy, our AutoPruner model can generate a unique binary code successfully.

4. Experimental Results

In this section, we will empirically study the benefits of our AutoPruner method. We compare AutoPruner with other state-of-the-art pruning approaches on two standard datasets: CUB200-2011 [18] and ImageNet ILSVRC-12 [19]. Three widely used deep models, VGG16 [1], ResNet-50 [40] and MobileNetV2 [41], are pruned. All the experiments are conducted using pyTorch on M40 GPUs.

4.1. Model Complexity Calculation

FLOPs, namely floating-point operations, is a popular metric to evaluate the complexity of CNN models. Following [42], the FLOPs in convolutional layers is calculated by:

$$\text{FLOPs} = 2HW(C_{in}K^2 + 1)C_{out},$$

where $H$, $W$, $C_{out}$ is the height, width and channel number of the output tensor, $K$ is the kernel size, $C_{in}$ refers to the number of input channels, and 1 means
Algorithm 1 The control processes of AutoPruner during model pruning

**Input:** pretrained CNN model, compression ratio \( r \)

**Output:** pruned model

1. append one (or several) AutoPruner layer in the pretrained model;
2. \( \alpha_{\text{step}} \leftarrow (\alpha_{\text{stop}} - \alpha_{\text{start}}) / (\text{epochs} \times \text{iterations}) \), \( \alpha \leftarrow \alpha_{\text{start}} \);
3. for each epoch \( e \) do
4.   for each iteration \( i \) do
5.     \( \alpha \leftarrow \alpha + \alpha_{\text{step}} \);
6.     model.SetAlpha(\( \alpha \)); // set the new \( \alpha \) value in each AutoPruner layer
7.     output, \( v = \) model.forward();
8.     if \( e \) is the last epoch then
9.       \( p \leftarrow \) the percentage of elements in \( v \) outside of \([0.1, 0.9]\);  
10.      if \( p < 0.9 \) // \( v \) is not converged then
11.        \( \alpha \leftarrow \alpha + 10 \times \alpha_{\text{step}} \);
12.    end if
13. end if
14. calculate loss function using Eq. 3 and Eq. 4;
15. model.backward();
16. end for
17. end for
18. remove the AutoPruner layer (or layers) and prune the model according to the binary code \( v \);
19. return pruned model

the FLOPs in bias term. Note that we regard a single vector multiplication as two floating-point operations (multiplication and addition). However, in some papers [40, 5], it may be regarded as one FLOP. For a fair comparison, we will re-calculate the FLOPs number if it is not computed by Eq. 5.

Another important metric to evaluate model complexity is MACs (Multiply-Accumulate operations). As the name implies, a single vector multiplication is regarded as one MAC operation. These two metrics can easily be confused in
some papers and we calculate FLOPs=2×MACs in this paper.

4.2. Pruning on CUB200-2011

We first compare the performance of AutoPruner with others on the CUB200-2011 dataset [18]. Many existing model compression algorithms have reported their results on a small dataset like MNIST [43] or CIFAR-10 [44]. However, these datasets are relatively simple, and different algorithms often generate very similar results with negligible difference. We argue that comparing on a tough but small dataset is necessary, since it is a more practical application scenario. Fine-grained recognition is a very challenging task due to the low inter-class but high intra-class variation.

CUB200-2011 is a popular fine-grained dataset, which aims to recognize 200 bird species. This dataset contains 11,788 bird images. We follow the official train/test split to organize the dataset: 5994 images are used for model training, and the accuracy will be reported on the rest 5794 images.

Implementation details. We first fine-tune a pretrained VGG16 model on CUB200-2011. For simplicity, only image-level labels are used without other supervised information such as bounding boxes. The images are resized with shorter side=256, then a 224 × 224 crop is randomly sampled from the resized image with horizontal flip and mean-std normalization. Then the preprocessed images are fed into the VGG16 model. We fine-tune VGG16 with 30 epochs using SGD. Weight decay is set to 0.0005, momentum is 0.9 and batch size is set to 64. The initial learning rate starts from 0.001, and is divided by 10 after every 10 epochs. The fine-tuned model achieves 76.683% top-1 accuracy.

Based on this fine-tuned model, we then train AutoPruner using the same fine-tuning parameters. We prune VGG16 from the conv1_1 to the conv5_3 layer by layer, i.e., the output of the previous stage is the input of the current stage. At each stage (e.g., when we want to prune conv1_1 layer), the AutoPruner module is appended on the output of the current layer, and fine-tuned in 2 epochs using the same parameters. The hyper-parameter (α) setting is kept the same as what we have stated in Sec. 3.3 (α_start = 0.1, α_stop = 2). After fine-
tuning, all the filters and channels corresponding to zero index values will be pruned away. The newly added AutoPruner layer will also be removed. Hence, the only difference after our processing is the reduction of the filter number. When pruning is finished on all layers, we will fine-tune the pruned model by another 30 epochs with the same parameters. This pruning pipeline is also applied in other baseline methods for a fair comparison.

**Comparison among different algorithms.** We compare the proposed AutoPruner method with two approaches:

- **Random selection.** This is a simple but very powerful method. At each pruning stage, several filters are randomly discarded to reduce model complexity. As indicated by [2, 3], random selection may be even better than some heuristic methods when compression rate is large.
- **ThiNet [2, 3].** ThiNet is an efficient three-stage pruning method, which formally establishes filter pruning as an optimization problem and uses statistics of the next layer to guide the current layer. We re-implement this method with our pruning pipeline for a fair comparison.

In order to generate the same network structure among different methods, we first use AutoPruner to prune the pretrained VGG16 model. Since the actual compression rate is determined by the network itself, AutoPruner may produce slightly larger or smaller network than we expected. Then, according to its output, the same number of filters are pruned away using the above two baselines.

Table 1 shows the compression results on VGG16 using different filter-level pruning methods. As we can see, AutoPruner is superior over the state-of-the-art filter pruning method ThiNet. Our method yields 1.94% higher top-1 accuracy than ThiNet when $r = 0.2$. Similar phenomenon can also be observed when $r = 0.5$. Since the pruning pipeline and fine-tuning parameters are the same, these two results should reflect that our end-to-end trainable framework is better than previous three-stage pruning method ThiNet. And both of these two models are better than random selection.
Table 1: Pruning VGG16 model on the CUB200-2011 dataset using different algorithms and compression rates.

<table>
<thead>
<tr>
<th>Method</th>
<th>compression rate $r = 0.2$</th>
<th></th>
<th>compression rate $r = 0.5$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>top-1 (%)</td>
<td>top-5 (%)</td>
<td>#FLOPs</td>
<td>top-1 (%)</td>
</tr>
<tr>
<td>fine-tuned VGG16</td>
<td>76.68</td>
<td>94.06</td>
<td>30.93B</td>
<td>76.68</td>
</tr>
<tr>
<td>random selection</td>
<td>57.28</td>
<td>83.52</td>
<td>2.62B</td>
<td>70.25</td>
</tr>
<tr>
<td>ThiNet [2] (Our impl.)</td>
<td>63.12</td>
<td>87.54</td>
<td>2.62B</td>
<td>73.00</td>
</tr>
<tr>
<td>AutoPruner (Ours)</td>
<td><strong>65.06</strong></td>
<td><strong>87.93</strong></td>
<td><strong>2.62B</strong></td>
<td><strong>73.45</strong></td>
</tr>
</tbody>
</table>

4.3. Pruning Large Networks

We then compare AutoPruner method with other state-of-the-art approaches on the large scale vision recognition task ImageNet ILSVRC-12 [19].

**Implementation details.** The fine-tuning settings are similar to those in Sec. 4.2. We randomly crop a $224 \times 224$ input on the resized images, and use the same preprocessing pipelines. Then the model is fine-tuned using SGD with 0.0005 weight decay, 0.9 momentum and 256 batch size. For VGG16, we iteratively prune it layer-by-layer with 3 epochs using $r = 0.4$. At each iteration, $\alpha$ of the AutoPruner layer starts from 0.1, and stops at 2. During the first 2 epochs, learning rate is set to 0.001, and is divided by 10 at the third epoch. The pruning procedure stops at the conv4 3 layer. Finally, the whole model is fine-tuned for 30 epochs.

As for ResNet-50, we follow the setting of ThiNet [2] to prune the first two intermediate layers of each residual block. We divide the whole residual blocks into 4 groups, and train multiple AutoPruner layers simultaneously. The initial value of $\alpha$ is set to 1, and stops at 100. At each group, the model is trained 8 epochs with the same parameters as VGG16. We prune ResNet-50 with two compression rate $r = 0.5$ and $r = 0.3$, and leave the last block uncompressed to obtain a higher accuracy. The compressed models are fine-tuned with 30 epochs in the final stage.

Table 2 shows the compression results on ImageNet. For a fair comparison, AutoPruner is only compared with filter level pruning methods. The accuracy is reported using a single view central patch crop: the shorter side is resized to 16
Table 2: Comparison results among several state-of-the-art filter level pruning methods on ImageNet. All the accuracies are tested on the validation set using the single view central patch crop. All the FLOPs numbers are calculated by Eq. 5 for a fair comparison. The inference speed is tested on a NVIDIA Tesla M40 GPU with batch size 32.

<table>
<thead>
<tr>
<th>Method</th>
<th>Top-1 Acc.</th>
<th>Top-5 Acc.</th>
<th>#FLOPs</th>
<th>Speed Up</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original ResNet-50 model²</td>
<td>76.15%</td>
<td>92.87%</td>
<td>8.18B</td>
<td>1.00×</td>
<td>0.084s</td>
</tr>
<tr>
<td><strong>AutoPruner (r = 0.3)</strong></td>
<td>73.05%</td>
<td>91.25%</td>
<td>2.78B</td>
<td>2.94×</td>
<td>0.057s</td>
</tr>
<tr>
<td>Taylor-FO-BN-56% [45]</td>
<td>71.69%</td>
<td>90.59%</td>
<td>2.68B</td>
<td>2.88×</td>
<td>-</td>
</tr>
<tr>
<td>GAL-1-joint [46]</td>
<td>69.31%</td>
<td>89.12%</td>
<td>2.22B</td>
<td>3.48×</td>
<td>0.036s</td>
</tr>
<tr>
<td>ThiNet-30 [2]</td>
<td>68.42%</td>
<td>88.30%</td>
<td>2.32B</td>
<td>3.51×</td>
<td>0.054s</td>
</tr>
<tr>
<td><strong>AutoPruner (r = 0.5)</strong></td>
<td>74.76%</td>
<td>92.15%</td>
<td>4.00B</td>
<td>2.05×</td>
<td>0.067s</td>
</tr>
<tr>
<td>AutoPruner with block pruning (r = 0.5)</td>
<td>73.84%</td>
<td>91.75%</td>
<td>4.56B</td>
<td>1.79×</td>
<td>0.045s</td>
</tr>
<tr>
<td>Channel Pruning (2×) [16]³</td>
<td>72.30%</td>
<td>90.80%</td>
<td>5.22B</td>
<td>1.48×</td>
<td>-</td>
</tr>
<tr>
<td>SSS (ResNet-26) [5]</td>
<td>71.82%</td>
<td>90.79%</td>
<td>4.00B</td>
<td>1.93×</td>
<td>0.043s</td>
</tr>
<tr>
<td>ThiNet-50 [2]</td>
<td>71.01%</td>
<td>90.02%</td>
<td>3.64B</td>
<td>2.27×</td>
<td>0.060s</td>
</tr>
<tr>
<td>Original VGG16 model²</td>
<td>71.59%</td>
<td>90.38%</td>
<td>30.94B</td>
<td>1.00×</td>
<td>0.116s</td>
</tr>
<tr>
<td><strong>AutoPruner</strong></td>
<td>69.20%</td>
<td>88.89%</td>
<td>8.17B</td>
<td>3.79×</td>
<td>0.046s</td>
</tr>
<tr>
<td>SSS [5]</td>
<td>68.53%</td>
<td>88.20%</td>
<td>7.67B</td>
<td>4.03×</td>
<td>0.049s</td>
</tr>
<tr>
<td>RNP (3×) [15]</td>
<td>-</td>
<td>87.58%</td>
<td>-</td>
<td>3.00×</td>
<td>-</td>
</tr>
<tr>
<td>RNP (4×) [15]</td>
<td>-</td>
<td>86.67%</td>
<td>-</td>
<td>4.00×</td>
<td>-</td>
</tr>
<tr>
<td>Channel Pruning (5×) [16]³</td>
<td>67.80%</td>
<td>88.10%</td>
<td>7.03B</td>
<td>4.40×</td>
<td>0.042s</td>
</tr>
<tr>
<td>Taylor expansion-1 [42]</td>
<td>-</td>
<td>84.50%</td>
<td>8.02B</td>
<td>3.86×</td>
<td>-</td>
</tr>
<tr>
<td>Taylor expansion-2 [42]</td>
<td>-</td>
<td>87.00%</td>
<td>11.54B</td>
<td>2.68×</td>
<td>-</td>
</tr>
<tr>
<td>Filter Pruning (impl. by [16]) [23]</td>
<td>-</td>
<td>75.30%</td>
<td>7.03B</td>
<td>4.40×</td>
<td>-</td>
</tr>
</tbody>
</table>

¹ The speedup ratio is a theoretical value computed by FLOPs.
² https://pytorch.org/docs/master/torchvision/models.html
³ https://github.com/yihui-he/channel-pruning/releases/tag/channel_pruning_5x

256, followed by a 224 × 224 center crop as well as mean-std normalization. We re-calculate the FLOPs by Eq. 5. Hence, the FLOPs values reported here may be different with the original ones (e.g., ResNet [40], SSS [5]).

ResNet-50. We first compare the proposed AutoPruner with other state-of-the-arts on the ResNet-50 model. Pruning ResNet is a challenging task due to its compact structure. We follow the same pruning strategy with ThiNet [2] but achieve much better accuracy. Channel Pruning (2×) [16] introduced a channel sampler layer in the first convolution layer of each residual block to reduce the input width. However, our AutoPruner obtains significantly higher accuracy with a much simpler strategy. The same conclusion is also applicable
for SSS [5]. Note that SSS prune the whole blocks on ResNet, which can obtain a faster acceleration but may leads to larger accuracy drop. For a fair comparison, we also conduct block pruning using AutoPruner. Our method achieves 2% higher top-1 accuracy with similar FLOPs as well as actual inference speed.

**VGG16.** Similar conclusion can also be acquired on the VGG16 model. Among these methods, SSS [5] adopts a very similar technique as ours. In SSS, a scaling factor vector $\vec{\lambda}$ is learned during model training. And all the filters will be removed if their corresponding scaling factors are 0. As we will demonstrate in the ablation study, the gradient information flowing out from the channel selection layer is also helpful for previous layers. Hence, AutoPruner can achieve a better result than SSS.

RNP [15] is another novel method that explore filter pruning beyond the three-stage pipeline. As we can see, the proposed AutoPruner outperforms this method by a large margin. We can achieve a better accuracy even with larger compression ratio (RNP ($3\times$) vs. AutoPruner).

We then compare AutoPruner with channel pruning [16]. In this method, the pre-trained model is first pruned by a LASSO regression based method and further processed by a 3C approach to get a smaller model. 3C is composed of spatial decomposition [47] and channel decomposition [48]. For a fair comparison, we only report its pruning result. Again, the proposed AutoPruner outperforms this novel three-stage pruning method.

### 4.4. Pruning Lightweight Networks

Recently, lightweight network like MobileNetV2 [41] has attracted widespread attention. Thanks to the novel network structure like depthwise and pointwise convolution, lightweight network can achieve good performance with much lower computational cost, which is really important for resource constrained devices. Although the model complexity is already greatly reduced, we show that lightweight network can be further pruned via our AutoPruner method.

As we stated above, if the network is equipped with batch normalization, pruning multiple or all layers simultaneously is encouraged. In this section,
Table 3: Pruning MobileNetV2 on ImageNet with different methods. The inference speed is tested on a NVIDIA Tesla M40 GPU with batch size 32. We report the training time of each pruning method with the form of time × epochs. Here, “s” and “d” represent “seconds” and “days”, respectively.

<table>
<thead>
<tr>
<th>Method</th>
<th>Accuracy</th>
<th>MACs</th>
<th>Latency</th>
<th>training time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>top-1</td>
<td>top-5</td>
<td>pruned</td>
<td>fine-tuning</td>
</tr>
<tr>
<td>MobileNetV2-1.0¹</td>
<td>72.19%</td>
<td>90.53%</td>
<td>300.79M</td>
<td>-</td>
</tr>
<tr>
<td>MobileNetV2-0.75</td>
<td>69.95%</td>
<td>88.99%</td>
<td>209.08M</td>
<td>0.031s</td>
</tr>
<tr>
<td>AMC (70% FLOPs) [27]</td>
<td>70.85%</td>
<td>89.91%</td>
<td>210.63M</td>
<td>0.027s</td>
</tr>
<tr>
<td>Slimmable NN [49]</td>
<td>68.93%</td>
<td>88.34%</td>
<td>209.08M</td>
<td>0.032s</td>
</tr>
<tr>
<td>AutoPruner</td>
<td>71.18%</td>
<td>89.95%</td>
<td>207.93M</td>
<td>0.026s</td>
</tr>
</tbody>
</table>

¹ [https://github.com/d-li14/mobilenetv2.pytorch](https://github.com/d-li14/mobilenetv2.pytorch)

we append our AutoPruner layer in all inverted residual blocks. Hence, we can prune all layers in once. Note that each residual block in MobileNetV2 is constituted by three convolutional layers: pointwise convolution, depthwise convolution and pointwise convolution. In depthwise convolution there is a constraint: input number must equal output number. In order to solve this problem, we append AutoPruner to the first pointwise convolution, the produced index code is shared by the first pointwise convolution and depthwise convolution. As for the third layer, we follow the strategy of ResNet pruning, and keep its output dimension unchanged. In other words, we use one AutoPruner layer to reduce the hidden dimension in each inverted residual block.

**Implementation details.** In the pruning stage, we adopt a similar setting as previous experiments. The parameter $\alpha$ of each layer is initialized as 0.1, linearly increased and finally ended at 100. To achieve 70% computational complexity, we set the compression ratio $r$ of each layer to 0.63. We train MobileNetV2 in 5 epochs with a fixed learning rate 0.01. Other parameters are kept the same as previous ImageNet experiments. After that, we remove the AutoPruner layers and prune the network according to the learned index code. Then in the final fine-tuning stage, the pruned small model is further trained in 150 epochs. The learning rate is initialized as 0.01 and decreased using a cosine schedule.
Table 3 shows the experimental results of pruning MobileNetV2. The first two rows summarize the performance of original MobileNetV2 models with a spectrum of width multipliers. We then prune MobileNetV2-1.0 using three different pruning algorithms, namely AMC [27], Slimmable Neural Networks [49] and the proposed AutoPruner. Obviously, our method outperforms previous state-of-the-art, which is even better than MobileNetV2-0.75. We also report the actual inference speed of each model on a NVIDIA Tesla M40 GPU. Our model has the fastest speed, although its MACs value is very close to others.

To compare the computational cost of the whole pruning procedure clearly, we also report the training time in Table 3. We divide the whole pruning procedure into two stages: the pruning stage and the fine-tuning stage. In each stage, the computational cost is reported with the form of time×epochs. We use the default epoch number of each method but this value may be changed in practice. Note that Slimmable Neural Networks [49] is not a typical pruning method. It aims to train a single neural network executable at different widths. Hence, there is no pruning stage in this method. In summary, our AutoPruner shows a similar computational cost but higher accuracy compared with AMC. And both methods are faster and more accurate than Slimmable Neural Networks and the original MobileNetV2-0.75 model.

4.5. Generalization Ability of the Pruned Models

After pruning, there may be a large loss of generalization to other tasks such as object detection. To further study the generalization of pruned MobileNetV2, we evaluate its transfer learning ability on the PASCAL VOC object detection benchmark [50] and COCO dataset [51]. We use SSDLite [41] as the base framework, which replaces all the regular convolutions of original SSD [52] prediction layers with depthwise followed by pointwise convolutions. All the models in Table 3 are trained on the trainval set of VOC2007 and VOC2012, then tested on VOC2007 test images. As for COCO, all the models are trained on COCO2017 training set and evaluated on COCO2017 validation set. The input resolution of both datasets is 300 × 300. Other parameters are consistent with the original
Table 4: Object detection results on VOC2007 test set using SSDLite framework.

<table>
<thead>
<tr>
<th>Method</th>
<th>MobileNetV2-1.0</th>
<th>MobileNetV2-0.75</th>
<th>AMC [27]</th>
<th>Slimmable NN [49]</th>
<th>AutoPruner</th>
</tr>
</thead>
<tbody>
<tr>
<td>mAP</td>
<td>0.6743</td>
<td>0.6529</td>
<td>0.6571</td>
<td>0.6602</td>
<td>0.6576</td>
</tr>
<tr>
<td>MACs</td>
<td>660.86M</td>
<td>477.41M</td>
<td>453.80M</td>
<td>477.41M</td>
<td>474.256M</td>
</tr>
<tr>
<td>#Param.</td>
<td>3.39M</td>
<td>2.49M</td>
<td>2.31M</td>
<td>2.49M</td>
<td>2.71M</td>
</tr>
</tbody>
</table>

Table 5: Object detection results on COCO2017 validation set using SSDLite framework.

<table>
<thead>
<tr>
<th>Method</th>
<th>MobileNetV2-1.0</th>
<th>MobileNetV2-0.75</th>
<th>AMC [27]</th>
<th>Slimmable NN [49]</th>
<th>AutoPruner</th>
</tr>
</thead>
<tbody>
<tr>
<td>mAP</td>
<td>0.173</td>
<td>0.145</td>
<td>0.164</td>
<td>0.139</td>
<td>0.161</td>
</tr>
<tr>
<td>MACs</td>
<td>928.94M</td>
<td>630.66M</td>
<td>742.48M</td>
<td>630.66M</td>
<td>754.89M</td>
</tr>
<tr>
<td>#Param.</td>
<td>9.244M</td>
<td>7.95M</td>
<td>8.33M</td>
<td>7.95M</td>
<td>8.59M</td>
</tr>
</tbody>
</table>

Table 4 and Table 5 show the detection results on the VOC2007 test set and COCO2017 validation set, respectively. AMC [27] and AutoPruner produce very similar mAP on these two detection tasks, which are both better than the baseline method MobileNetV2-0.75. Slimmable Neural Networks [49] achieves better performance than other pruning methods on VOC dataset, but is worse than others on COCO. This result demonstrates that some pruning methods may have a generalization loss when transfer to other tasks. However, our AutoPruner shows good generalization and works well.

In summary, the proposed AutoPruner method is also applicable to lightweight networks like MobileNetV2. After pruning, both classification performance and generalization ability are better than MobileNetV2-0.75.

4.6. Ablation Study

Finally, we conduct ablation studies about the proposed AutoPruner method. This section is composed of two parts: AutoPruner modules and the value of scaling factor $\alpha$.

---

1VOC: https://github.com/qfgaohao/pytorch-ssd, COCO: https://github.com/ShuangXieIrene/ssds.pytorch
Table 6: Pruning accuracy (%) on CUB200-2011 dataset using different algorithms and compression rates.

<table>
<thead>
<tr>
<th>Method</th>
<th>compression rate $r = 0.5$</th>
<th>compression rate $r = 0.2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>top-1 (%)</td>
<td>top-1 (%)</td>
</tr>
<tr>
<td>GAP</td>
<td>72.97</td>
<td>62.70</td>
</tr>
<tr>
<td>scaling factors</td>
<td>68.66</td>
<td>68.14</td>
</tr>
<tr>
<td>AutoPruner</td>
<td>73.45</td>
<td>65.06</td>
</tr>
</tbody>
</table>

4.6.1. AutoPruner Modules

The first ablation study is about AutoPruner itself. We want to explore the influence of different pooling binary and codes generation approaches. These two baseline methods are briefly summarized as follows:

- **GAP.** We replace the original max pooling of AutoPruner (see Section 3.1.1 for more details) with a GAP (Global Averaged Pooling) layer and keep other modules unchanged.

- **Scaling factors.** This baseline is similar to SSS [5]. In AutoPruner, the binary codes are generated from activation tensors. But in SSS, it is produced by a set of end-to-end trainable weights (i.e., scaling factors $\vec{\lambda} \in \mathbb{R}^C$). We replace the whole AutoPruner layer with a trainable scaling factor vector, and generate binary codes by $y = \text{sigmoid}(\alpha\vec{\lambda})$.

Table 6 shows the results of these two baselines. Experimental settings are the same as Section 4.2. Our max pooling is used for reducing GPU memory consumption. Since GAP can convert the original $1 \times C \times H' \times W'$ tensor into a $C$-d vector, it seems to be a better choice. However, a large spatial pooling operation is harmful as shown above. It may discard too much information, hence the coding layers cannot generate accurate binary code.

The major difference between AutoPruner and scaling factors is whether pruning information will participate in the training of previous layers or not. AutoPruner learns the binary codes from output of the previous layers, while scaling factors only use a trainable vector to indicate the status of a filter (pruning or not). Hence, the gradient of scaling factors could not be propagated back.
Table 7: Pruning accuracy (%) on CUB200-2011 using different choice of $\alpha_{\text{start}}$ ($r = 0.5$).

<table>
<thead>
<tr>
<th>$\alpha_{\text{start}}$</th>
<th>0.01</th>
<th>0.05</th>
<th>0.1</th>
<th>0.5</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>top-1 (%)</td>
<td>73.438</td>
<td>73.438</td>
<td>73.662</td>
<td>72.644</td>
<td>72.541</td>
</tr>
</tbody>
</table>

As illustrated in Table 6, this idea is much worse than AutoPruner. It may even fail when the compression rate $r$ is small ($r = 0.2$). In this scenario, model FLOPs can be only reduced to 6.91B. Hence, we believe the gradient information flowing out from the channel selection layer is also helpful for previous convolution layer training. It can force the network to pay more attention to the preserved filters.

### 4.6.2. The Value of Scaling Factor

In the proposed AutoPruner method, there are mainly two additional parameters: compression ratio $r$ and scaling factor $\alpha$. The compression ratio $r$ is decided by specific tasks as we stated in Section 3.2. This is a predefined value, and is the goal of filter pruning. In general, we should make a tradeoff between model accuracy and inference speed to find an appropriate value for $r$.

As for $\alpha$, it is decided by $\alpha_{\text{start}}$ and $\alpha_{\text{stop}}$. As introduced in Section 3.3, $\alpha_{\text{stop}}$ is related to the network itself. We can try several numbers to find an appropriate value for $\alpha_{\text{stop}}$ that can produce binary codes in Eq 2. Once this value is determined, it will not be changed again. In our experiments, $\alpha_{\text{stop}} = 2$ is a fixed choice for VGG16 model. Please note that our method is really robust for the value of $\alpha_{\text{stop}}$. For ResNet model, we find AutoPruner module can generate binary codes when $\alpha_{\text{stop}} = 100$. But it does not mean $\alpha_{\text{stop}} = 100$ is the only choice. Our method can also work well if we set $\alpha_{\text{stop}} = 80$ or $\alpha_{\text{stop}} = 120$, because we will check the convergence of generated index codes (see Algorithm 1 for more details).

Now, we are more interested in the choice of $\alpha_{\text{start}}$. Table 7 shows the influence of different $\alpha_{\text{start}}$ values. Experimental settings are kept the same as Section 4.2. Starting from a relatively small value ($\alpha_{\text{start}} \leq 0.1$), the filters will be gradually erased. Our method will check the value of generated codes and
increase $\alpha$ quickly if the code is not converged after several iterations. However, if $\alpha_{\text{start}}$ is too large, it may become binary at the first few iterations. In this case, AutoPruner will degenerate to random selection. Our methods is robust for the choice of $\alpha_{\text{start}}$ as long as it is not too large. In general, 0.01 or 0.1 will be a good choice.

To sum up, $\alpha$ plays an essential role in the proposed AutoPruner method. The choice of $\alpha$ is important but not hard to tune. Our method is very robust for the choice of $\alpha$ and can generate binary prediction successfully.

5. Discussions

5.1. Consistency of Index Code

In this part, we want to discuss an interesting question about the consistency of index code. Since the generated binary code is used for model pruning, it should be unique for different input mini-batches. In other words, the output of an AutoPruner layer should be consistent for different images. We empirically demonstrate that the proposed AutoPruner method has such kind of capability.

Let us focus on each channel in the index code, which can be regarded as a classification task. Since the index code is binary, the AutoPruner layer is trained on how to mark all the images with a positive/negative label. This is a relatively simple task. By gradually increasing/decreasing the bias term of fully-connected layer, we can always project all the examples into a positive/negative label. However, if our adaptive sigmoid layer is removed, the network will fail to generate a unique output, i.e., one channel is marked useful for some images but useless for others. In AutoPruner, the combination of our mini-batch pooling and binarization ensures the consistency.

To validate this hypothesis, we remove $\alpha$, and train the first group of ResNet-50 using $r = 0.3$ on ImageNet. Without increasing $\alpha$ to achieve binarization, the top-1 accuracy of pruned model on validation set is only 9.154% without fine-tuning. We find that more than 90% elements in the generated index code are around 0.2. Removing these filters will damage model accuracy significantly.
since they are not equal to 0. Hence, our novel binarization scheme plays an essential role in the success of AutoPruner.

5.2. The Merit of Network Pruning

Next, we will give a brief discussion about the value of network pruning. It is generally accepted that deep model is over-parameterized. This redundancy is helpful for model training, but will significantly slow down inference speed. Hence, there is a great need to remove these redundant parameters after training.

However, as indicated in the recent study [53], training from scratch may achieve even better results than model pruning. For example, they found that if the ThiNet-50 baseline (listed in Table 2) is trained from scratch by 180 epochs, its top-1 accuracy can reach 73.90%. Although this result is much higher than most existing pruning methods, it is still lower than our AutoPruner (74.76%).

To sum up, pruning indeed provides a useful tool to accelerate model inference speed while preserving its accuracy.

5.3. Relation to SENet

The core idea of AutoPruner is quite similar to SENet [54], where a learned vector is also used to model the relationship between channels. Although SENet is not a pruning method, a necessary discussion is still needed. The major problem of SENet is that its produced vector is input-dependent. When input data changes (e.g., different input images), the learned importance vector will also change. Hence it cannot be used for network pruning in our static pruning setting, where the importance score of redundant filters should always be zero. On the other hand, because the code is not binarized, simply removing small weights will damage model accuracy greatly. If we discard filters whose averaged score is smaller than a threshold, its accuracy can drop to 0 before the final fine-tuning. In summary, we think the success of AutoPruner can be attributed to the combination of the batch-wise pooling and the binarization scheme. Then, a unique binary code can be learned to denote the importance score of each filter.
5.4. The Output Values of Removed Channels

Theoretically speaking, there may be an issue about the output values during fine-tuning. When we gradually reduce the output values of sigmoid function, the input values (the left tensors of Figure 2) may increase to keep a balance in the output activation tensor. Hence, after element-wise multiplication, the magnitude of output activation values will not change. However, we did not observe this phenomenon in our experiments. In fact, there are many normalization techniques (for example, weight decay) to penalize large weights and effectively limit the freedom of CNN model. Since the weights will not increase, its output values will not increase. More importantly, the output codes will become binary. A zero value will erase all the elements of its corresponding channel. Hence, removing these channels will not affect the network’s performance.

6. Conclusions

Hand-crafted importance evaluation criterion plays an essential role in previous three-stage pruning methods. However, these criterions heavily depend on hand-crafted rules from humans and thus are sub-optimal. In this paper, we propose AutoPruner, an end-to-end trainable filter pruning method for CNN acceleration. AutoPruner can be regarded as an independent layer, and can be appended in any convolution layer to prune filters automatically. We demonstrate that the proposed method can significantly improve model compression performance over existing filter pruning methods. By pruning network in an end-to-end automatic way, we can break away from the dependence of hand-crafted rules and obtain a better result. Further study reveals that our method is also applicable to lightweight networks and is really useful to obtain a smaller but still accurate model.

However, we only prune channels inside the residual connection, leaving its output dimension unchanged. Obviously, this architecture is sub-optimal. In the future, we will try to find a better way to solve this problem. On the other
hand, there are still several parameters in our method. How to achieve a more efficient model compression still deserves our further study.

Acknowledgements

This paper is supported by the National Natural Science Foundation of China under Grant No. 61772256.

References


