Compressing Deep Convolutional Neural Networks by Stacking Low-dimensional Binary Convolution Filters

Weichao Lan, Liang Lan
Department of Computer Science, Hong Kong Baptist University, Hong Kong SAR, China
{cswclan, lanliang}@comp.hkbu.edu.hk

Abstract

Deep Convolutional Neural Networks (CNN) have been successfully applied to many real-life problems. However, the huge memory cost of deep CNN models poses a great challenge of deploying them on memory-constrained devices (e.g., mobile phones). One popular way to reduce the memory cost of deep CNN model is to train binary CNN where the weights in convolution filters are either 1 or -1 and therefore each weight can be efficiently stored using a single bit. However, the compression ratio of existing binary CNN models is upper bounded by ~ 32. To address this limitation, we propose a novel method to compress deep CNN model by stacking low-dimensional binary convolution filters. Our proposed method approximates a standard convolution filter by selecting and stacking filters from a set of low-dimensional binary convolution filters. This set of low-dimensional binary convolution filters is shared across all filters for a given convolution layer. Therefore, our method will achieve much larger compression ratio than binary CNN models. In order to train our proposed model, we have theoretically shown that our proposed model is equivalent to select and stack intermediate feature maps generated by low-dimensional binary filters. Therefore, our proposed model can be efficiently trained using the split-transform-merge strategy. We also provide detailed analysis of the memory and computation cost of our model in model inference. We compared the proposed method with other five popular model compression techniques on two benchmark datasets. Our experimental results have demonstrated that our proposed method achieves much higher compression ratio than existing methods while maintains comparable accuracy.

Introduction

Recent advances in deep convolutional neural network (CNN) have produced powerful models that achieve high accuracy on a wide variety of real-life tasks. These deep CNN models typically consist of a large number of convolution layers involving many parameters. They require large memory to store the model parameters and intensive computation for model inference. Due to concerns on privacy, security and latency caused by performing deep CNN model inference remotely in the cloud, deploying deep CNN models on edge devices (e.g., mobile phones) and performing local on-device model inference has gained growing interests recently (Zhang et al. 2018; Howard et al. 2017). However, the huge memory cost of deep CNN model poses a great challenge when deploying it on resource-constrained edge devices. For example, the VGG-16 network (Simonyan and Zisserman 2014), which is one of the famous deep CNN models, performs very well in both image classification and object detection tasks. But this VGG-16 network requires more than 500MB memory and over 15 billions floating number operations (FLOPs) to classify a single input image (Cheng et al. 2018).

To reduce the memory and computation cost of deep CNN models, several model compression methods have been proposed in recent years. These methods can be generally categorized into five major types: (1) parameter pruning and sharing (Han et al. 2015; Han, Mao, and Dally 2015; Ullrich, Meeds, and Welling 2017): pruning redundant, non-informative weights in pre-trained CNN models; (2) low-rank approximation (Denton et al. 2014; Jaderberg, Vedaldi, and Zisserman 2014): finding appropriate low-rank approximation for convolution layers; (3) knowledge distillation (Ba and Caruana 2014; Hinton, Vinyals, and Dean 2015; Buciluǎ, Caruana, and Niculescu-Mizil 2006): approximating deep neural networks with shallow models; (4) compact convolution filters (Howard et al. 2017; Zhang et al. 2018): using carefully designed structural convolution filters; and (5) model quantization (Han, Mao, and Dally 2015; Gupta et al. 2015): quantizating the model parameters and therefore reducing the number of bits to represent each weight. Among these existing studies, model quantization is one of the most popular ways for deep CNN model compression. It is widely used in commercial model deployments and has several advantages compared with other methods (Krishnamoorthi 2018): (1) broadly applicable across different network architectures and hardwares; (2) smaller model footprint; (3) faster computation and (4) powerful efficiency.

Binary neural networks is the extreme case in model quantization where each weight can only be 1 or -1 and therefore can be stored using a single bit. In the research direction of binary neural networks, the pioneering work BinaryConnect (BC) proposed by (Courbariaux, Bengio, and David 2015) is the first successful method that incorporates learning binary model weights in the training pro-

cess. Several extensions to BC have been proposed, such as Binarized Neural Networks(BNN) presented by (Hubara et al. 2016), Binary Weight Network (BWN) and XNOR-Networks (XNOR-Net) proposed by (Rastegari et al. 2016). Even though the existing works on binary neural networks have shown promising results on model compression and acceleration, they use a binary filter with the same kernel size and the same filter depth as a standard convolution filter. Therefore, for a given popular CNN architecture, binary neural networks can only compress the original model by up to ~ 32 times. This upper bound on compression ratio (i.e., 32) could limit the applications of binary CNNs on resource-constrained devices, especially for large scale CNNs with a huge number of parameters.

Motivated by recent work LegoNet (Yang et al. 2019) which constructs efficient convolutional networks with a set of small full-precision convolution filters named lego filters, we propose to compress deep CNN by selecting and stacking low-dimensional binary convolution filters. In our proposed method, each original convolution filter is approximated by stacking a number of filters selected from a set of low-dimensional binary convolution filters. This set of low-dimensional binary convolution filters is shared across all convolution filters for a given convolution layer. Therefore, our proposed method can achieve much higher compression ratio than binary CNNs. Compared with LegoNet, our proposed method can reduce the memory cost of LegoNet by a factor of ~ 32 since our basic building blocks are binary filters instead of full-precision filters.

The main contributions of this paper can be summarized as follows: First, we propose a novel method to overcome the theoretical compression ratio limit of recent works on binary CNN models. Second, we have shown that our proposed model can be reformulated as selecting and stacking feature maps generated by low-dimensional binary convolution filters. After reformulation, our proposed model can be efficiently trained using the split-transform-merge strategy and can be easily implemented by using any existing deep learning framework (e.g., PyTorch or Tensorflow). Third, we provide detailed analysis of the memory and computation cost of our model for model inference. Finally, we compare our proposed method with other five popular model compression algorithms on three benchmark datasets. Our experimental results clearly demonstrate that our proposed method can achieve comparable accuracy with much higher compression ratio. We also empirically explore the impact of various training techniques (e.g., choice of optimizer, batch normalization) on our proposed method in the experiments.

Preliminaries

Convolutional Neural Networks

In a standard CNN, convolution operation is the basic operation. As shown in Figure 1(a), for a given convolution layer in CNN, it transforms a three-dimensional input tensor $\mathbf{X}_{input} \in \mathbb{R}^{w_{in} \times h_{in} \times c_{in}}$, where w_{in}, h_{in} and c_{in} represents the width, height and depth (or called number of channels) of the input tensor, into a three-dimensional output tensor $\mathbf{X}_{output} \in \mathbb{R}^{w_{out} \times h_{out} \times c_{out}}$ by

$$\mathbf{X}_{output} = \text{Conv}(\mathbf{X}_{input}, \mathbf{W}), \tag{1}$$

where Conv() denotes the convolution operation. Each entry in the output tensor $\mathbf{X}_{output}$ is obtained by an element-wise multiplication between a convolution filter $\mathbf{W}^i \in \mathbb{R}^{d \times d \times c_{in}}$ and a patch $\mathbf{X}^i_{input} \in \mathbb{R}^{d \times d \times c_{in}}$ extracted from $\mathbf{X}_{input}$ followed by summation. $d \times d$ is the kernel size of the convolution filter (usually d is 3) and c_{in} is depth of the convolution filter which is equal to the number of input channels. Therefore, for a given convolution layer with c_{out} convolution filters, we can use $\mathbf{W} \in \mathbb{R}^{d \times d \times c_{in} \times c_{out}}$ to denote the parameters needed for all c_{out} convolution filters. The memory cost of storing weights of convolution filters $\mathbf{W}$ for a given layer is $d \times d \times c_{in} \times c_{out} \times 32$ bits assuming 32-bit floating-point values are used to represent model weights. It is high since deep CNN models usually contain a large number of layers. The computation cost for CNN model inference is also high because the convolution operation involves a large number of FLOPs.

Binary Convolutional Neural Networks

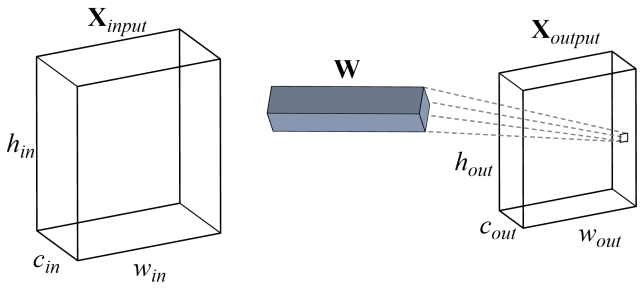
To reduce the memory and computation cost of deep CNN model, several algorithms (Simonyan and Zisserman 2014; Courbariaux, Bengio, and David 2015; Rastegari et al. 2016; Hubara et al. 2016; Alizadeh et al. 2019) have been proposed recently. Their core idea is to binarize the model weights $\mathbf{W} \in \mathbb{R}^{d \times d \times c_{in} \times c_{out}}$. Since a binary weight can be efficiently stored with a single bit, these methods can reduce the memory cost of storing $\mathbf{W} \in \mathbb{R}^{d \times d \times c_{in} \times c_{out}}$ to $d \times d \times c_{in} \times c_{out}$ bits. It has been shown that these methods can achieve good classification accuracy with much less memory and computation cost compared to standard CNN model. However, due to that the binarized $\mathbf{W}$ is still of size $d \times d \times c_{in} \times c_{out}$, these binary CNNs can only reduce the memory cost of deep CNN model by up to ~ 32 times.

Methodology

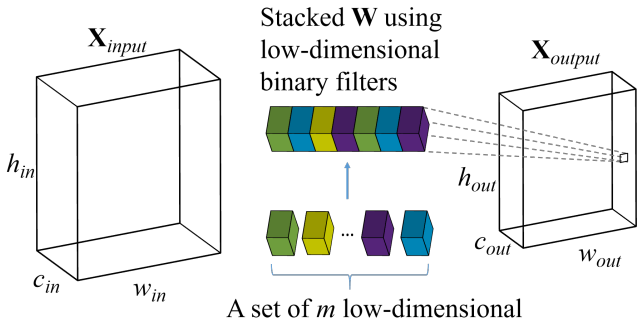
In this section, we propose a new method that can overcome the theoretical compression ratio limit of binary CNN models. Instead of approximating convolution filters using binary convolution filters with the same kernel size and the same filter depth, our proposed idea approximates the convolution filters by selecting and stacking a number of filters from a set of low-dimensional binary convolution filters. The depth of these binary filters will be much smaller than the depth of original convolution filters. Therefore, we call them *low-dimensional binary convolution filters* in this paper. This set of low-dimensional binary convolution filters is shared across all convolution filters for a given convolution layer. The main idea of our proposed method is illustrated in Figure 1 and we will explain the details of it in following subsections.

Approximating Convolution Filters by Stacking Low-dimensional Binary Filters

Suppose we use $\mathbf{W}^t \in \mathbb{R}^{d \times d \times c_{in}}$ to denote the t-th full-precision convolution filter in a convolution layer in a standard CNN. According to (1), the t-th feature map in the out-



(a) Convolution filters $\mathbf{W}$



(b) Stacked $\mathbf{W}$ using low-dimensional binary convolution filters

Figure 1: Approximating Convolution Filters by Stacking Low-dimensional Binary convolution Filters

put tensor $\mathbf{X}^t_{output} \in \mathbb{R}^{w_{out} \times h_{out}}$ generated by convolution filter $\mathbf{W}^t$ can be written as

$$\mathbf{X}^t_{output} = \text{Conv}(\mathbf{X}_{input}, \mathbf{W}^t). \tag{2}$$

Let us use $\{\mathbf{B}_1, \mathbf{B}_2, \ldots, \mathbf{B}_m\}$ to denote a set of m shared binary convolution filters for a given convolution layer. $\mathbf{B}_i \in \mathbb{R}^{d \times d \times s}$ denotes the weights for the i-th binary filter where s is depth of the binary convolution filters. In here, the depth s is much smaller than c_{in} which is the depth of original convolution filters. Each element in $\mathbf{B}_i$ is either 1 or -1. We propose to approximate $\mathbf{W}^t$ by selecting $k = \frac{c_{in}}{s}$ low-dimensional binary convolution filters from $\{\mathbf{B}_1, \ldots, \mathbf{B}_m\}$ and then stacking them together. Let us define an indicator matrix $\mathbf{P} \in \mathbb{R}^{m \times k}$ as

$$\mathbf{P}_{ji} = \begin{cases} 1 & \text{if the } i\text{-th block of } \mathbf{W}^t \text{ is } \mathbf{B}_j \\ 0 & \text{otherwise} \end{cases}. \tag{3}$$

By following the selecting and stacking idea, $\mathbf{W}^t$ will be approximated by $[\sum_{j=1}^m \mathbf{P}_{j1}\mathbf{B}j, \sum_{j=1}^m \mathbf{P}_{j2}\mathbf{B}j, \ldots, \sum_{j=1}^m \mathbf{P}_{jk}\mathbf{B}j]$ which concatenates k low dimensional binary convolution filters together in column-wise manner. Here we can also introduce another variable α_i to denote the scaling factor associated to the i-th block of $\mathbf{W}^t$ when we concatenate different binary convolution filters together, that is, $\mathbf{W}^t \approx [\alpha_1 \sum_{j=1}^m \mathbf{P}_{j1}\mathbf{B}j, \alpha_2 \sum_{j=1}^m \mathbf{P}_{j2}\mathbf{B}j, \ldots, \alpha_k \sum_{j=1}^m \mathbf{P}_{jk}\mathbf{B}j]$. We can treat $\alpha_i \mathbf{P}_{ji}$ as single variable by changing the definition of $\mathbf{P}$ in (3) to

$$\mathbf{P}_{ji} = \begin{cases} \alpha_i & \text{if the } i\text{-th block of } \mathbf{W}^t \text{ is } \mathbf{B}_j \\ 0 & \text{otherwise} \end{cases}. \tag{4}$$

In our experiment section, we have shown that introducing the scaling factors $\{\alpha_i\}_{i=1}^k$ always obtains slightly better classification accuracy than without using them.

Let us split the $\mathbf{X}_{input} \in \mathbb{R}^{w_{in} \times h_{in} \times c_{in}}$ into $k = \frac{c_{in}}{s}$ parts $\{\mathbf{X}_{input(1)}, \mathbf{X}_{input(2)}, \ldots, \mathbf{X}_{input(k)}\}$ where the size of each part $\mathbf{X}_{input(i)}$ is $\mathbb{R}^{w_{in} \times h_{in} \times s}$. Then the t-th feature map in the output tensor generated by convolution filter $\mathbf{W}^t$ as shown in (2) can be approximated as

$$\mathbf{X}^t_{output} = \sum_{i=1}^{k} \text{Conv}(\mathbf{X}_{input(i)}, \sum_{j=1}^{m} \mathbf{P}_{ji}\mathbf{B}j). \tag{5}$$

Note that $\|\mathbf{P}_{(:,i)}\|_0 = 1$ (i.e., each column of $\mathbf{P}$ only contains one non-zero value) means that only one binary filter $\mathbf{B}_j$ is selected to perform the convolution operation on the i-th part of $\mathbf{X}_{input}$. The $\mathbf{X}^t_{output}$ is a element-wise sum of k feature maps. Each feature map is generated by applying a single low-dimensional binary convolution filter to one part of the input.

As shown in (5), for a convolution filter in a given convolution layer, the model parameters are $\{\mathbf{B}_1, \ldots, \mathbf{B}_m\}$ and $\mathbf{P}$, where $\{\mathbf{B}_1, \ldots, \mathbf{B}_m\}$ is shared by all convolution filters for a given convolution layer. Therefore, the model parameters of our proposed method for a given convolution layer with c_{out} convolution filters are just $\{\mathbf{B}_1, \ldots, \mathbf{B}_m\}$ and $\{\mathbf{P}^t\}_{t=1}^{c_{out}}$. By considering that the memory cost of storing $\{\mathbf{P}^t\}_{t=1}^{c_{out}}$ is relatively small than storing $\{\mathbf{B}_1, \ldots, \mathbf{B}_m\}$, our proposed method can significantly reduce the memory cost of binary CNNs. A detailed analysis of the compression ratio and computation cost of our proposed method will be provided in section of algorithm implementation and analysis.

Training Model Parameters of the Proposed Compressed CNN

In this section, we present our algorithm to learn the model parameters $\{\mathbf{B}_1, \ldots, \mathbf{B}_m\}$ and $\{\mathbf{P}^t\}_{t=1}^{c_{out}}$ from the training data. Without loss of generality, let us consider to optimize the model parameters for one layer. Assume $\{\mathbf{X}_{input}, \mathbf{Y}_{output}\}$ is a mini-batch of inputs and targets for a given convolution layer. Therefore, the objective for optimizing $\{\mathbf{B}_1, \ldots, \mathbf{B}_m\}$ and $\{\mathbf{P}^t\}_{t=1}^{c_{out}}$ will be

$$\begin{aligned} \min & \sum_{t=1}^{c_{out}} \|\mathbf{Y}^t_{output} - \sum_{i=1}^{k} \text{Conv}(\mathbf{X}_{input(i)}, \sum_{j=1}^{m} \mathbf{P}^t_{ji}\mathbf{B}_j)\|^2 \\ s.t & \ \|\mathbf{P}^t_{(:,i)}\|_0 = 1 \\ & \ \mathbf{B}_{ij} \in \{-1, 1\}. \end{aligned} \tag{6}$$

In order to optimize (6), we first prove that the convolution operation $\text{Conv}(\mathbf{X}_{input(i)}, \sum_{j=1}^m \mathbf{P}^t_{ji}\mathbf{B}_j)$ is equivalent to $\sum_{j=1}^m \mathbf{P}^t_{ji}\text{Conv}(\mathbf{X}_{input(i)}, \mathbf{B}_j)$ as shown in Proposition 1. In other words, selecting a convolution filter (i.e.,

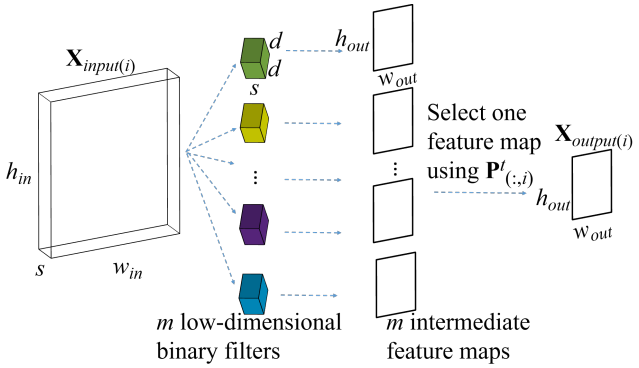


Figure 2: Reformat as convolution and then select feature

$\sum_{j=1}^{m} \mathbf{P}^t_{ji}\mathbf{B}_j)$ and then performing convolution operation is equivalent to performing m convolution operations and then selecting a feature map from the generated m intermediate feature maps $\{\text{Conv}(\mathbf{X}_{input(i)}, \mathbf{B}_j)\}_{j=1}^{m}$. The advantage of latter computation is that it can reduce the computation cost since these m intermediate feature maps $\{\text{Conv}(\mathbf{X}_{input(i)}, \mathbf{B}_j)\}_{j=1}^{m}$ is shared across all c_{out} convolution filters for a given convolution layer.

Proposition 1. *Suppose* $\mathbf{X}_{input(i)} \in \mathbb{R}^{w_{in}\times h_{in}\times s}$, $\{\mathbf{B}_1, \ldots, \mathbf{B}_m\}$ *is a set of* m *low-dimensional binary filters where each* $\mathbf{B}_i \in \mathbb{R}^{d\times d\times s}$ *and* $\mathbf{P}^t_{(:,i)}$ *is the* i*-th column in* $\mathbf{P}^t$ *which is a length-*m *sparse vector with only one non-zero element. Then,* $\text{Conv}(\mathbf{X}_{input(i)}, \sum_{j=1}^{m} \mathbf{P}^t_{ji}\mathbf{B}_j)$ *is equivalent to* $\sum_{j=1}^{m} \mathbf{P}^t_{ji}\text{Conv}(\mathbf{X}_{input(i)}, \mathbf{B}_j)$.

The proof of Proposition 1 can be done by using the definition of convolution operation and the associative property of matrix multiplication. Based on Proposition 1, $\text{Conv}(\mathbf{X}_{input(i)}, \sum_{j=1}^{m} \mathbf{P}^t_{ji}\mathbf{B}_j)$ in (6) can be reformulated as: (1) first performing convolution operations on $\mathbf{X}_{input(i)}$ using $\{\mathbf{B}_1, \ldots, \mathbf{B}_m\}$ to generate m intermediate feature maps; (2) selecting one feature map from them. This procedure is also illustrated in Figure 2. After reformulation, our proposed model can be efficiently trained using the split-transform-merge strategy as in Szegedy et al. (2015).

Similar to training a standard CNN, the training process of our proposed model involves three steps in each iteration: (1) forward propagation; (2) backward propagation and (3) parameter update. In our proposed model, we have additional non-smooth constraints on $\{\mathbf{B}_1, \ldots, \mathbf{B}_m\}$ and $\{\mathbf{P}^t\}_{t=1}^{c_{out}}$. To effectively learning the non-smooth model parameters in each convolution layer, we introduce full-precision filters $\{\mathbf{R}_1, \ldots, \mathbf{R}_m\}$ as the proxies of binary filters $\{\mathbf{B}_1, \ldots, \mathbf{B}_m\}$ and dense matrices $\{\mathbf{Q}^t\}_{t=1}^{c_{out}}$ as the proxies of$\{\mathbf{P}^t\}_{t=1}^{c_{out}}$. Instead of directly learning $\{\mathbf{B}_1, \ldots, \mathbf{B}_m\}$ and $\{\mathbf{P}^t\}_{t=1}^{c_{out}}$, we learn the proxies $\{\mathbf{R}_1, \ldots, \mathbf{R}_m\}$ and $\{\mathbf{Q}^t\}_{t=1}^{c_{out}}$ during the training. $\{\mathbf{B}_1, \ldots, \mathbf{B}_m\}$ and $\{\mathbf{P}^t\}_{t=1}^{c_{out}}$ are computed only in the forward propagation and backward propagation. This framework has been successfully used in training binary neural networks (Courbariaux, Bengio, and David 2015; Hubara et al. 2016; Rastegari et al. 2016).

Forward Propagation. During the forward propagation, the binary convolution filters $\{\mathbf{B}_1, \ldots, \mathbf{B}_m\}$ is obtained by

$$\mathbf{B}_i = \text{sign}(\mathbf{R}_i), \tag{7}$$

where sign() is the element-wise sign function which return 1 if the element is larger or equal than zero and return -1 otherwise. Similarly, sparse indicator matrices $\{\mathbf{P}^t\}_{t=1}^{c_{out}}$ can be obtained by

$$\mathbf{P}^t_{ji} = \begin{cases} \mathbf{Q}^t_{ji} & \text{if } j = \text{argmax}\,(|\mathbf{Q}^t_{(:,i)}|) \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

during the forward propagation where the argmax $(|\mathbf{Q}^t_{(:,i)}|)$ function returns the row index j of the maximum absolute value of the i-th column of $\mathbf{Q}^t$.

Backward Propagation. Since both the sign() function in (7) and the argmax() function in (8) are not differentiable, we use the Straight Through Estimator (STE) (Bengio, Léonard, and Courville 2013) to back propagate the estimated gradients for updating the proxy variables $\{\mathbf{R}_1, \ldots, \mathbf{R}_m\}$ and $\{\mathbf{Q}^t\}_{t=1}^{c_{out}}$. The basic idea of STE is to simply pass the gradients as if the non-differentiable functions sign() and argmax() are not present.

Specifically, let us use r to denote a full-precision weight and it is a proxy for a binary weight b. Therefore,

$$b = \begin{cases} 1 & \text{if } r \geq 0, \\ -1 & \text{otherwise.} \end{cases} \tag{9}$$

(9) is not a differentiable function, STE will just simply estimate its gradient as sign function is not present. That is $\frac{\partial b}{\partial r} = 1$. In practice, we also employ the gradient clipping as in Hubara et al. (2016). Then, the gradient for the sign function is

$$\frac{\partial b}{\partial r} = 1_{|r|\leq 1}. \tag{10}$$

Therefore, in the back propagation, the gradient of a convex loss function $L(r)$ with respect to the proxy variable r can be estimated as

$$\frac{\partial L(r)}{\partial r} = \frac{\partial L(b)}{\partial b}\frac{\partial b}{\partial r} = \frac{\partial L(b)}{\partial b}1_{|r|\leq 1}. \tag{11}$$

Similarly, the gradient of a convex loss function $L(\mathbf{Q}^t_{ji})$ with respect to the proxy variable $\mathbf{Q}^t_{ji}$ can be estimated by STE as

$$\frac{\partial L(\mathbf{Q}^t_{ji})}{\partial \mathbf{Q}^t_{ji}} = \frac{\partial L(\mathbf{P}^t_{ji})}{\partial \mathbf{P}^t_{ji}}\frac{\partial \mathbf{P}^t_{ji}}{\partial \mathbf{Q}^t_{ji}} = \frac{\partial L(\mathbf{P}^t_{ji})}{\partial \mathbf{P}^t_{ji}}. \tag{12}$$

Parameter Update. As shown in (11) and (12), we now can backpropagate gradients $\frac{\partial L(b)}{\partial b}$ and $\frac{\partial L(\mathbf{P}^t_{ji})}{\partial \mathbf{P}^t_{ji}}$ to their proxies $\{\mathbf{R}_1, \ldots, \mathbf{R}_m\}$ and $\{\mathbf{Q}^t\}_{t=1}^{c_{out}}$. Then, these two proxy variables can be updated by using any popular optimizer (e.g., SGD with momentum or ADAM (Kingma and Ba 2014)). Note that once our training process is completed, we do not need to keep the proxy variables $\{\mathbf{R}_1, \ldots, \mathbf{R}_m\}$ and $\{\mathbf{Q}^t\}_{t=1}^{c_{out}}$. Only the low-dimensional binary convolution filters$\{\mathbf{B}_1, \ldots, \mathbf{B}_m\}$ and the sparse indicator matrices $\{\mathbf{P}^t\}_{t=1}^{c_{out}}$ are needed for convolution operations in model inference.

Algorithm 1 Compressed CNN via stacking low-dimensional binary filters

Training

Input: training data $\{\mathbf{X}_{train}, \mathbf{y}_{train}\}$, a convex loss function $L(\mathbf{y}, \hat{\mathbf{y}})$, CNN configuration, hyperparameter for low-dimensional binary filter s and m

Output: Compressed CNN model

1: Initialize proxy variables $\{\mathbf{R}_1, \ldots, \mathbf{R}_m\}$ and $\{\mathbf{Q}^t\}_{t=1}^{c_{out}}$ for each convolution layer l based on CNN configuration and s and m
2: **for** iter = 1 to maxIter **do**
3: Get a minibatch of training data $\{\mathbf{X}, \mathbf{y}\}$
4: **for** $l = 1$ to L **do**
5: Obtain low-dimensional binary filters $\{\mathbf{B}_1, \ldots, \mathbf{B}_m\}$ according to (7)
6: Obtain $\{\mathbf{P}^t\}_{t=1}^{c_{out}}$ for each convolution filter t according to (8)
7: **end for**
8: Perform standard **forward propagation** except that convolution operations are defined in Proposition 1
9: Compute the loss $L(\mathbf{y}, \hat{\mathbf{y}})$
10: Perform standard **backward propagation** except that gradients for $\{\mathbf{R}_1, \ldots, \mathbf{R}_m\}$ and $\{\mathbf{Q}^t\}_{t=1}^{c_{out}}$ are computed respectively as in (11) and (12)
11: Perform **parameter update** for proxy variables $\{\mathbf{R}_1, \ldots, \mathbf{R}_m\}$ and $\{\mathbf{Q}^t\}_{t=1}^{c_{out}}$ using any popular optimizer (e.g., SGD with momentum or ADAM)
12: **end for**

Prediction

Input: test data $\mathbf{X}_{test}$, Trained compressed CNN

Output: predicted labels $\hat{\mathbf{y}}_{test}$;

1: Perform standard **forward propagation** except that convolution operations are defined in Proposition 1

Algorithm Implementation and Analysis

We summarize our algorithm in **Algorithm 1**. In step 1, we initialize the proxy variables $\{\mathbf{R}_1, \ldots, \mathbf{R}_m\}$ and $\{\mathbf{Q}^t\}_{t=1}^{c_{out}}$ for each convolution layer l. From step 4 to step 7, we obtain binary filters $\{\mathbf{B}_1, \ldots, \mathbf{B}_m\}$ by (7) and sparse indicator matrices $\{\mathbf{P}^t\}_{t=1}^{c_{out}}$ by (8) for each convolution layer. In step 8, we perform standard **forward propagation** except that convolution operations are defined as stacking low-dimensional binary filters. In step 9, we compute the loss $L\{\mathbf{y}, \hat{\mathbf{y}}\}$ using current predicted value $\hat{\mathbf{y}}$ and ground truth $\mathbf{y}$. In step 10, we perform standard **backward propagation** except that the gradients with respect to proxy variables $\{\mathbf{R}_1, \ldots, \mathbf{R}_m\}$ and $\{\mathbf{Q}^t\}_{t=1}^{c_{out}}$ are computed respectively as in (11) and (12). In step 11, we perform **parameter update** for proxy variables using any popular optimizer (e.g., SGD with momentum or ADAM). We implement our **Algorithm 1** using PyTorch framework (Paszke et al. 2019).

In model inference, we do not need to keep the proxy variables. In each convolution layer, we only need the trained low-dimensional binary filters $\{\mathbf{B}_1, \ldots, \mathbf{B}_m\}$ and sparse indicator matrices $\{\mathbf{P}^t\}_{t=1}^{c_{out}}$ to perform convolution operations. Therefore, compared with standard convolution operations using $\mathbf{W}$ as in (1), our proposed method that constructs convolution filter by stacking a number of low-dimensional binary filters can significantly reduce the memory and computation cost of standard CNNs.

With respect to compression ratio, for a standard convolution layer, the memory cost is $d \times d \times c_{in} \times c_{out} \times 32$ bits. In our proposed method, the memory cost of storing a set of low-dimensional binary filters $\{\mathbf{B}_1, \ldots, \mathbf{B}_m\}$ is $d \times d \times s \times m$ bits. The memory cost of storing stacking parameter $\{\mathbf{P}^t\}_{t=1}^{c_{out}}$ is $\frac{c_{in}}{s} \times m \times c_{out}$ if $\mathbf{P}$ is defined as in (3) where each entry can be stored using a single bit and is $\frac{c_{in}}{s} \times c_{out} \times 32 \times 3$ if $\mathbf{P}$ is defined in (4) [1]. In our hyperparameter setting, we will set $s = c_{in} f_1$ and $m = c_{out} f_2$ where f_1 and f_2 are fractional numbers less than 1. In our experiments, we set them as $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots$, and so on. The compression ratio of our proposed method is

$$\frac{d \times d \times c_{in} \times c_{out} \times 32}{d \times d \times c_{in} f_1 \times c_{out} f_2 + \frac{1}{f_1} \times c_{out} \times 32 \times 3} \tag{13}$$

By considering that the memory cost of storing $\{\mathbf{P}^t\}_{t=1}^{c_{out}}$ is relatively small compared with the memory cost of storing low-dimensional binary filters if f_1 is not a very small fractional number, the compression ratio of our proposed method can be approximated by $\sim \frac{32}{f_1 f_2}$. The actual compression ratio of our method will be reported in the experimental section.

With respect to computation cost, for a given convolution layer, standard convolution operations require $d \times d \times c_{in} \times w_{out} \times h_{out} \times c_{out}$ FLOPs. In comparison, our method will first require $d \times d \times c_{in} \times w_{out} \times h_{out} \times m$ FLOPs to compute $\frac{1}{f_1} \times m$ intermediate feature maps where the depth of each intermediate feature map is equal to 1. Then, we select and combine these intermediate feature maps to form the output tensor using $w_{out} \times h_{out} \times \frac{1}{f_1} \times c_{out}$ FLOPs. By considering that $w_{out} \times h_{out} \times \frac{1}{f_1} \times c_{out}$ is relatively small than $d \times d \times w_{out} \times h_{out} \times c_{in} \times m$ if f_1 is not a very small fractional number, the speedup of our model inference can be approximated as

$$\sim \frac{d \times d \times c_{in} \times w_{out} \times h_{out} \times c_{out}}{d \times d \times c_{in} \times w_{out} \times h_{out} \times m} = \frac{1}{f_2}. \tag{14}$$

Furthermore, due to the binary filters used in our method, convolution operations can be computed using only addition and subtraction (without multiplication) which can further speed up the model inference (Rastegari et al. 2016).

Experiments

In this section, we compare the performance of our proposed method with five state-of-the-art CNN model compression algorithms on two benchmark image classification datasets: CIFAR-10 and CIFAR-100 (Krizhevsky, Hinton et al. 2009). Note that we focus on the compressing convolution layers as in (Yang et al. 2019). The full connection layers can be compressed by adaptive fastfood transform (Yang et al. 2015) which is beyond the scope of this paper.

[1] We use three full-precision vectors to store the indices and values of the nonzero elements in sparse matrix $\mathbf{P}$.

Network	Compression Ratio	CIFAR-10 Acc(%)	CIFAR-100 Acc(%)
Full Net (VGG-16)	1	**93.25**	**73.55**
LegoNet($f_1 = \frac{1}{4}$, $f_2 = \frac{1}{4}$)	5.4x	91.35	70.10
BC	31.6x	92.11	70.64
BWN	31.6x	93.09	69.03
BNN	31.6x	91.21	67.88
XNOR-Net	31.6x	90.02	68.63
SLBF (f_1 = 1, $f_2 = \frac{1}{2}$)	60.1x	91.44	68.80
SLBF ($f_1 = \frac{1}{2}$, $f_2 = \frac{1}{2}$)	103.2x	91.30	67.55
SLBF ($f_1 = \frac{1}{2}$, $f_2 = \frac{1}{4}$)	173.1x	90.24	66.68
SLBF ($f_1 = \frac{1}{2}$, $f_2 = \frac{1}{8}$)	261.4x	89.24	62.88

Table 1: Results of different networks on CIFAR-10 and CIFAR-100 datasets using VGG-16 Net

In our experiments, we evaluate the performance of the following seven algorithms:

- Full Net: deep CNN model with full-precision weights;
- BinaryConnect(BC): deep CNN model with binary weights (Courbariaux, Bengio, and David 2015);
- Binarized Neural Networks(BNN): deep CNN model with both binary weights and binary activations (Hubara et al. 2016);
- Binary Weight Network(BWN): similar to BinaryConnect(BC) but scaling factors are added to binary filters (Rastegari et al. 2016);
- XNOR-Networks(XNOR-Net): similar to BNN but scaling factors are added to binary filters and binary activations (Rastegari et al. 2016);
- LegoNet: Efficient CNN with Lego filters (Yang et al. 2019)
- Stacking Low-dimensional Binary Filters (SLBF): Our proposed method.

Experimental Results on CIFAR-10 and CIFAR-100 Using VGG-16 Net

We first present our experiment settings and results on CIFAR-10 and CIFAR-100 datasets by using VGG-16 (Simonyan and Zisserman 2014) network as the CNN architecture. CIFAR-10 consists of 50,000 training samples and 10,000 test samples with 10 classes while CIFAR-100 contains more images belonging to 100 classes. Each sample in these two datasets is a 32×32 colour image. The VGG-16 network contains 13 convolution layers and 3 full-connected layers. We use this CNN network architecture for all seven methods. The batch normalization with scaling and shifting applies to all methods too. In our method SLBF, SGD with the momentum of 0.9 is used as the optimizer. For other five model compression methods, we use the suggested settings from their papers.

Our experimental results with different settings of f_1 and f_2 using VGG-16 are presented in Table 1. Note that we only report the result for LegoNet with $f_1 = \frac{1}{4}$ and $f_2 = \frac{1}{4}$ because it gets the best trade-off between compression ratio and accuracy based on our experimental results. The VGG-16 with full precision weights gets the highest accuracy 93.25% on CIFAR-10 and 73.55% on CIFAR-100. For CIFAR-10, our method can get 91.30% with model compression ratio 103.2x. This is encouraging since we can compress the full model by more than 100 times without sacrifice classification accuracy too much ($< 2\%$). The loss of accuracy with the same compression ratio is larger on CIFAR-100 but the performance is still comparable with other benchmark methods. As expected, the accuracy of our method will decrease when compression ratio increases. However, as can be seen from Table 1, the accuracy does not decrease much (i.e., from 91.30% to 88.63%) even we increase the compression ratio from 103.2x to 217.32x. It clearly demonstrates our proposed method can achieve a good trade-off between accuracy and model compression ratio.

Experimental Results on CIFAR-10 and CIFAR-100 Using ResNet-18

We also apply the recent ResNet-18 (He et al. 2016) structure with 17 convolution layers followed by one full-connection layer on CIFAR-10 and CIFAR-100 datasets. Similar to the experimental setting using VGG-16, SGD with the momentum of 0.9 is used as the optimizer in our method.

The accuracy and compression ratio of benchmark and our method with different settings using ResNet-18 is shown in Table 2. Generally the ResNet performs better than VGG-16 network on these two datasets, it can obtain a comparable accuracy of 74.19% on CIFAR-100 with about 95 times compression when setting $f_1 = \frac{1}{2}$ and $f_2 = \frac{1}{2}$, and the accuracy will not decrease greatly as compression ratio increases to 151 times. In the following subsections, we empirically explore the impact of scaling factors and several other training techniques on our proposed method.

The Impact of Scaling Factors in Matrix $\mathbf{P}$

In our proposed method, the matrix $\mathbf{P}$ used for selecting and stacking binary filters can be defined either as in (3) or as in (4). The difference between these two definitions is that (4) will multiply binary filters with scaling factors when stacking them together. In here, we evaluate the impact of scaling factors in our method. We compare the ac-

Network	Compression Ratio	CIFAR-10 Acc(%)	CIFAR-100 Acc(%)
Full Net (ResNet-18)	1	**95.19**	**77.11**
LegoNet($f_1 = \frac{1}{4}$, $f_2 = \frac{1}{4}$)	17.5x	93.55	72.67
BC	31.8x	93.73	71.15
BWN	31.8x	93.97	72.92
BNN	31.8x	90.47	70.34
XNOR-Net	31.8x	90.14	72.87
SLBF ($f_1 = 1$, $f_2 = \frac{1}{2}$)	58.7x	93.82	74.59
SLBF ($f_1 = \frac{1}{2}$, $f_2 = \frac{1}{2}$)	95.1x	93.72	74.19
SLBF ($f_1 = 1$, $f_2 = \frac{1}{4}$)	108.2x	92.96	72.12
SLBF ($f_1 = \frac{1}{2}$, $f_2 = \frac{1}{4}$)	151.4x	92.94	71.91
SLBF ($f_1 = \frac{1}{2}$, $f_2 = \frac{1}{8}$)	214.9x	91.70	67.89

Table 2: Results of different networks on CIFAR-10 and CIFAR-100 datasets using ResNet-18 Net

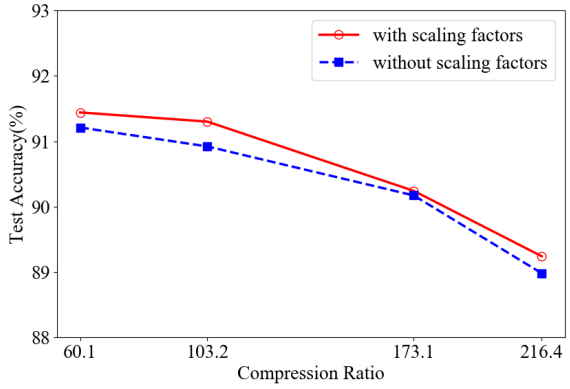


Figure 3: Comparison of our method with and without scaling factors

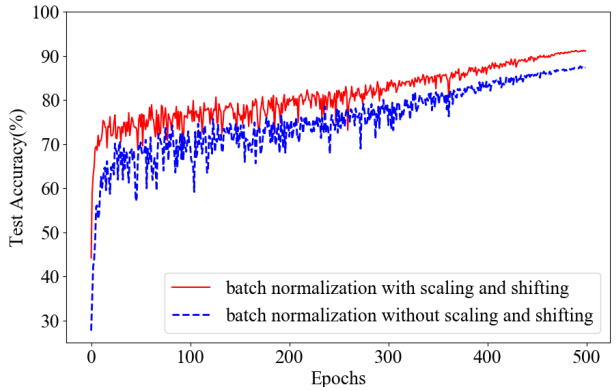


Figure 4: Accuracy of batch normalization with/without scaling and shifting ($f_1 = \frac{1}{2}, f_2 = \frac{1}{2}$).

curacy of our method with and without scaling factors on CIFAR-10 datasets using VGG-16 as the compression ratio changing from 60.1x to 217.3x and the results are shown in Figure 3. As can be seen from Figure 3, our proposed method with scaling factors always gets slightly higher accuracy than without scaling factors.

The Impact of Batch Normalization

Batch normalization (Ioffe and Szegedy 2015) is a popular technique to improve the training of deep neural networks. It standardizes the inputs to a layer for each mini-batch. We compare the performance of our proposed method with two different batch normalization settings: (1) batch normalization without scaling and shifting: normalize inputs to have zero mean and unit variance; (2) batch normalization with scaling and shifting. The results are reported in Figure 4 and it shows that batch normalization with scaling obtains better accuracy than without scaling and shifting on CIFAR-10 dataset. Thus we apply these two factors on our methods in the experiments.

Conclusions and Future Works

In this paper, we propose a novel method to compress deep CNN by selecting and stacking low-dimensional binary filters. Our proposed method can overcome the theoretical compression ratio limit of existing binary CNN models. We have theoretically shown that our proposed model is equivalent to select and stack low-dimensional feature maps generated by low-dimensional binary filters and therefore can be efficiently trained using the split-transform-merge strategy. We also provide detailed analysis on the memory and computation cost of our model for model inference. We compare our proposed method with other five popular model compression techniques on three benchmark datasets. Our experimental results clearly demonstrate that our proposed method can achieve comparable accuracy with much higher compression ratio. In our experiments, we also empirically explore the impact of various training techniques on our proposed method.

In the future, we will consider to use binary activation function. By doing it, convolution operations in each layer will be replaced by cheap XNOR and POPCOUNT binary operations which can further speed up model inference as observed in (Rastegari et al. 2016). We are also interested in investigating alternative methods to Straight Through Estimator (STE) for learning non-smooth model parameters.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments and valuable suggestions on our paper. This work was supported by NSFC 61906161.

References

Alizadeh, M.; Fernández-Marqués, J.; Lane, N. D.; and Gal, Y. 2019. An Empirical study of Binary Neural Networks' Optimisation. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.

Ba, J.; and Caruana, R. 2014. Do deep nets really need to be deep? In *Advances in neural information processing systems*, 2654–2662.

Bengio, Y.; Léonard, N.; and Courville, A. 2013. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432* .

Buciluǎ, C.; Caruana, R.; and Niculescu-Mizil, A. 2006. Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, 535–541.

Cheng, J.; Wang, P.-s.; Li, G.; Hu, Q.-h.; and Lu, H.-q. 2018. Recent advances in efficient computation of deep convolutional neural networks. *Frontiers of Information Technology & Electronic Engineering* 19(1): 64–77.

Courbariaux, M.; Bengio, Y.; and David, J.-P. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, 3123–3131.

Denton, E. L.; Zaremba, W.; Bruna, J.; LeCun, Y.; and Fergus, R. 2014. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in neural information processing systems*, 1269–1277.

Gupta, S.; Agrawal, A.; Gopalakrishnan, K.; and Narayanan, P. 2015. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, 1737–1746.

Han, S.; Mao, H.; and Dally, W. J. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* .

Han, S.; Pool, J.; Tran, J.; and Dally, W. 2015. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, 1135–1143.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778.

Hinton, G.; Vinyals, O.; and Dean, J. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* .

Howard, A. G.; Zhu, M.; Chen, B.; Kalenichenko, D.; Wang, W.; Weyand, T.; Andreetto, M.; and Adam, H. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* .

Hubara, I.; Courbariaux, M.; Soudry, D.; El-Yaniv, R.; and Bengio, Y. 2016. Binarized neural networks. In *Advances in neural information processing systems*, 4107–4115.

Ioffe, S.; and Szegedy, C. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* .

Jaderberg, M.; Vedaldi, A.; and Zisserman, A. 2014. Speeding up Convolutional Neural Networks with Low Rank Expansions. In *Proceedings of the British Machine Vision Conference. BMVA Press*.

Kingma, D. P.; and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* .

Krishnamoorthi, R. 2018. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342* .

Krizhevsky, A.; Hinton, G.; et al. 2009. Learning multiple layers of features from tiny images. *Technical report (University of Toronto)* .

Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, 8024–8035.

Rastegari, M.; Ordonez, V.; Redmon, J.; and Farhadi, A. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, 525–542. Springer.

Simonyan, K.; and Zisserman, A. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* .

Szegedy, C.; Liu, W.; Jia, Y.; Sermanet, P.; Reed, S.; Anguelov, D.; Erhan, D.; Vanhoucke, V.; and Rabinovich, A. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 1–9.

Ullrich, K.; Meeds, E.; and Welling, M. 2017. Soft weight-sharing for neural network compression. *arXiv preprint arXiv:1702.04008* .

Yang, Z.; Moczulski, M.; Denil, M.; de Freitas, N.; Smola, A.; Song, L.; and Wang, Z. 2015. Deep fried convnets. In *Proceedings of the IEEE International Conference on Computer Vision*, 1476–1483.

Yang, Z.; Wang, Y.; Liu, C.; Chen, H.; Xu, C.; Shi, B.; Xu, C.; and Xu, C. 2019. Legonet: Efficient convolutional neural networks with lego filters. In *International Conference on Machine Learning*, 7005–7014.

Zhang, X.; Zhou, X.; Lin, M.; and Sun, J. 2018. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 6848–6856.