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

Weichao Lan, 1 Liang Lan, 1
1Department 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 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 process. 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 \( \sim 32 \) 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 \( \sim 32 \) 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 Fig. 1(a), for a given convolution layer in CNN, it transforms a three-dimensional input tensor \( \mathbf{X}_{\text{input}} \in \mathbb{R}^{w_{\text{in}} \times h_{\text{in}} \times c_{\text{in}}} \), where \( w_{\text{in}}, h_{\text{in}} \) and \( c_{\text{in}} \) represents the width, height and depth (or called number of channels) of the input tensor, into a three-dimensional output tensor \( \mathbf{X}_{\text{output}} \in \mathbb{R}^{w_{\text{out}} \times h_{\text{out}} \times c_{\text{out}}} \) by

\[
\mathbf{X}_{\text{output}} = \text{Conv}(\mathbf{X}_{\text{input}}, \mathbf{W}),
\]

where \( \text{Conv}() \) denotes the convolution operation. Each entry in the output tensor \( \mathbf{X}_{\text{output}} \) is obtained by an element-wise multiplication between a convolution filter \( \mathbf{W} \in \mathbb{R}^{d \times d \times c_{\text{in}}} \) and a patch \( \mathbf{X}_{\text{input}}^{i} \in \mathbb{R}^{d \times d \times c_{\text{in}}} \) extracted from \( \mathbf{X}_{\text{input}} \) followed by summation. \( d \times d \) is the kernel size of the convolution filter (usually \( d = 3 \)) and \( c_{\text{in}} \) is depth of the convolution filter which is equal to the number of input channels. Therefore, for a given convolution layer with \( c_{\text{out}} \) convolution filters, we can use \( \mathbf{W} \in \mathbb{R}^{d \times d \times c_{\text{in}} \times c_{\text{out}}} \) to denote the parameters needed for all \( c_{\text{out}} \) convolution filters. The memory cost of storing weights of convolution filters \( \mathbf{W} \) for a given layer is \( d \times d \times c_{\text{in}} \times c_{\text{out}} \times 32 \) 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 \( \mathbf{W} \in \mathbb{R}^{d \times d \times c_{\text{in}} \times c_{\text{out}}} \). Since a binary weight can be efficiently stored with a single bit, these methods can reduce the memory cost of storing \( \mathbf{W} \in \mathbb{R}^{d \times d \times c_{\text{in}} \times c_{\text{out}}} \) to \( d \times d \times c_{\text{in}} \times c_{\text{out}} \) 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 \( d \times d \times c_{\text{in}} \times c_{\text{out}} \), these binary CNNs can only reduce the memory cost of deep CNN model by up to \( \sim 32 \) times.

Figure 1: Approximating Convolution Filters by Stacking Low-dimensional Binary Convolution Filters
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 Fig. 1 and we will explain the details of it in following subsections.

Approximating convolution Filters by Stacking low-dimensional Binary Filters

Suppose we use $W^t \in \mathbb{R}^{d \times d \times c_in}$ 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 output tensor $X_{output}^t \in \mathbb{R}^{w \times h \times c_out}$ generated by convolution filter $W^t$ can be written as

$$X_{output}^t = \text{Conv}(X_{input}, W^t).$$

(2)

Let us use $\{B_1, B_2, \ldots, B_m\}$ to denote a set of $m$ shared binary convolution filters for a given convolution layer. $B_i \in \mathbb{R}^{d \times d \times s}$ denotes the weights for the $i$-th block of the $t$-th 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 $B_i$ is either 1 or -1. We propose to approximate $W^t$ by selecting $k = \frac{c_in}{s}$ low-dimensional binary convolution filters from $\{B_1, \ldots, B_m\}$ and then stacking them together. Let us define an indicator matrix $P \in \mathbb{R}^{m \times k}$ as

$$P_{ji} = \begin{cases} 1 & \text{if the } i\text{-th block of } W^t \text{ is } B_j \\ 0 & \text{otherwise} \end{cases}. \quad (3)$$

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

$$P_{ji} = \begin{cases} \alpha_i & \text{if the } i\text{-th block of } W^t \text{ is } B_j \\ 0 & \text{otherwise} \end{cases}. \quad (4)$$

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 $X_{input} \in \mathbb{R}^{w \times h \times c_in}$ into $k = \frac{c_in}{s}$ parts $\{X_{input}(1), X_{input}(2), \ldots, X_{input}(k)\}$ where the size of each part $X_{input}(i)$ is $\mathbb{R}^{w \times h \times s}$. Then the $t$-th feature map in the output tensor generated by convolution filter $W^t$ as shown in (2) can be approximated as

$$X_{output}^t = \sum_{i=1}^{k} \text{Conv}(X_{input}(i), \sum_{j=1}^{m} P_{ji}B_j). \quad (5)$$

Note that $\|P_{[i,:]}\| = 1$ (i.e., each column of $P$ only contains one non-zero value) means that only one binary filter $B_j$ is selected to perform the convolution operation on the $i$-th part of $X_{input}$. The $X_{output}^t$ 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 $\{B_1, \ldots, B_m\}$ and $P$, where $\{B_1, \ldots, B_m\}$ 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 $c_{out}$ output convolution filters are just $\{B_1, \ldots, B_m\}$ and $\{P^t\}_{i=1}^{c_{out}}$. By considering that the memory cost of storing $\{P^t\}_{i=1}^{c_{out}}$ is relatively small than storing $\{B_1, \ldots, B_m\}$, 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.

Training Model Parameters of the Proposed Compressed CNN

In this section, we present our algorithm to learn the model parameters $\{B_1, \ldots, B_m\}$ and $\{P^t\}_{i=1}^{c_{out}}$ from the training data. Without loss of generality, let us consider to optimize the model parameters for one layer. Assume $\{X_{input}, Y_{output}\}$ is a mini-batch of inputs and targets for a given convolution layer. Therefore, the objective for optimizing $\{B_1, \ldots, B_m\}$ and $\{P^t\}_{i=1}^{c_{out}}$ will be

$$\min_{t=1}^{c_{out}} \|Y_{output}^t - \sum_{i=1}^{k} \text{Conv}(X_{input}(i), \sum_{j=1}^{m} P_{ji}^t B_j)\|^2$$

$$s.t. \|P_{[i,:]}\| = 1 \quad B_{ij} \in \{-1, 1\}. \quad (6)$$

In order to optimize (6), we first prove that the convolution operation $\text{Conv}(X_{input}(1), \sum_{j=1}^{m} P^t_{j1} B_j)$ is equivalent to $\sum_{j=1}^{m} P^t_{j1} \text{Conv}(X_{input}(i), B_j)$ as shown in Proposition 1. In other words, selecting a convolution filter (i.e., $\sum_{j=1}^{m} P^t_{j1} B_j$) 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 $\{\text{Conv}(X_{input}(i), B_j)\}_{j=1}^{m}$. The advantage of latter computation is that it can reduce the computation cost since these $m$ intermediate feature maps $\{\text{Conv}(X_{input}(i), B_j)\}_{j=1}^{m}$ is shared across all $c_{out}$ convolution filters for a given convolution layer.
Proposition 1. Suppose $X_{\text{input}(i)} \in \mathbb{R}^{w_{in} \times h_{in} \times s}$, \(\{B_1, \ldots, B_m\}\) is a set of \(m\) low-dimensional binary filters where each \(B_i \in \mathbb{R}^{d \times d \times s}\) and \(P^t_{ji}\) is the \(i\)-th column in \(P^t\) which is a length-\(m\) sparse vector with only one non-zero element. Then, \(\text{Conv}(X_{\text{input}(i)}, \sum_{j=1}^m P^t_{ji}B_j)\) is equivalent to \(\sum_{j=1}^m P^t_{ji} \text{Conv}(X_{\text{input}(i)}, B_j)\).

Figure 2: Reformat as convolution and then select feature

The proof of Proposition 1 can be done by using the definition of convolution operation and the associative property of matrix multiplication. The details can be found in supplementary materials. Based on Proposition 1, \(\text{Conv}(X_{\text{input}(i)}, \sum_{j=1}^m P^t_{ji}B_j)\) in (6) can be reformulated as: (1) first performing convolution operations on \(X_{\text{input}(i)}\) using \(\{B_1, \ldots, B_m\}\) 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 \(\{B_1, \ldots, B_m\}\) and \(\{P^t\}_{t=1}^{\text{out}}\). To effectively learning the non-smooth model parameters in each convolution layer, we introduce full-precision filters \(\{R_1, \ldots, R_m\}\) as the proxies of binary filters \(\{B_1, \ldots, B_m\}\) and dense matrices \(\{Q^t\}_{t=1}^{\text{out}}\) as the proxies of \(\{P^t\}_{t=1}^{\text{out}}\). Instead of directly learning \(\{B_1, \ldots, B_m\}\) and \(\{P^t\}_{t=1}^{\text{out}}\), we learn the proxies \(\{R_1, \ldots, R_m\}\) and \(\{Q^t\}_{t=1}^{\text{out}}\) during the training. \(\{B_1, \ldots, B_m\}\) and \(\{P^t\}_{t=1}^{\text{out}}\) 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 \(\{B_1, \ldots, B_m\}\) is obtained by

\[
B_i = \text{sign}(R_i),
\]

where \(\text{sign()}\) 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 \(\{P^t\}_{t=1}^{\text{out}}\) can be obtained by

\[
P^t_{ji} = \begin{cases} Q^t_{ji} & \text{if } j = \arg\max (|Q^t_{:\cdot,i}|) \\ 0 & \text{otherwise} \end{cases}
\]

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

Backward Propagation. Since both the \(\text{sign()}\) function in (7) and the \(\arg\max()\) function in (8) are not differentiable, we use the Straight Through Estimator (STE) (Bengio, Leonard, and Courville 2013) to back propagate the estimated gradients for updating the proxy variables \(\{R_1, \ldots, R_m\}\) and \(\{Q^t\}_{t=1}^{\text{out}}\). The basic idea of STE is to simply pass the gradients as if the non-differentiable functions \(\text{sign()}\) and \(\arg\max()\) 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,

\[
b = \begin{cases} 1 & \text{if } r \geq 0, \\ -1 & \text{otherwise.} \end{cases}
\]

(9)

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

\[
\frac{\partial b}{\partial r} = 1_{|r| \leq 1}.
\]

(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

\[
\frac{\partial L(r)}{\partial r} = \frac{\partial L(b)}{\partial b} \frac{\partial b}{\partial r} = \frac{\partial L(b)}{\partial b} 1_{|r| \leq 1}.
\]

(11)

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

\[
\frac{\partial L(Q^t_{ji})}{\partial Q^t_{ji}} = \frac{\partial L(P^t_{ji})}{\partial P^t_{ji}} \frac{\partial P^t_{ji}}{\partial Q^t_{ji}} = \frac{\partial L(P^t_{ji})}{\partial P^t_{ji}}.
\]

(12)

Parameter Update. As shown in (11) and (12), we now can backpropagate gradients \(\frac{\partial L(b)}{\partial b}\) and \(\frac{\partial L(P^t_{ji})}{\partial P^t_{ji}}\) to their proxies \(\{R_1, \ldots, R_m\}\) and \(\{Q^t\}_{t=1}^{\text{out}}\). 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 \(\{R_1, \ldots, R_m\}\) and \(\{Q^t\}_{t=1}^{\text{out}}\). Only the low-dimensional binary convolution filters \(\{B_1, \ldots, B_m\}\) and the sparse indicator matrices \(\{P^t\}_{t=1}^{\text{out}}\) are needed for convolution operations in model inference.

Algorithm Implementation and Analysis

We summarize our algorithm in Algorithm 1. In step 1, we initialize the proxy variables \(\{R_1, \ldots, R_m\}\) and \(\{Q^t\}_{t=1}^{\text{out}}\) for each convolution layer \(l\). From step 4 to step 7, we obtain
Algorithm 1 Compressed CNN via stacking low-dimensional binary filters

Training
Input: training data \( \{X_{\text{train}}, y_{\text{train}}\} \), a convex loss function \( L(y, \hat{y}) \), CNN configuration, hyperparameter for low-dimensional binary filter \( s \) and \( m \)
Output: Compressed CNN model
1: Initialize proxy variables \( \{R_1, \ldots, R_m\} \) and \( \{Q^t\}_{t=1}^{c_{out}} \) for each convolution layer \( l \) based on CNN configuration and \( s \) and \( m \)
2: for \( \text{iter} = 1 \) to maxIter do
3: Get a minibatch of training data \( \{X, y\} \)
4: for \( l = 1 \) to \( L \) do
5: Obtain low-dimensional binary filters \( \{B_1, \ldots, B_m\} \) according to (7)
6: Obtain \( \{P^t\}_{t=1}^{c_{out}} \) 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] and compute the loss \( L(y, \hat{y}) \)
9: Compute gradients for each variable \( \{R_1, \ldots, R_m\} \) and \( \{Q^t\}_{t=1}^{c_{out}} \)
10: Perform parameter update for proxy variables \( \{R_1, \ldots, R_m\} \) and \( \{Q^t\}_{t=1}^{c_{out}} \) using any popular optimizer (e.g., SGD with momentum or ADAM)
11: end for

Prediction
Input: test data \( X_{\text{test}} \), Trained compressed CNN
Output: predicted labels \( \hat{y}_{\text{test}} \):
1: Perform standard forward propagation except that convolution operations are defined in Proposition [1]

With respect to memory cost, 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 \( \{B_1, \ldots, B_m\} \) is \( d \times d \times s \times m \) bits. The memory cost of storing stacking parameter \( \{P^t\}_{t=1}^{c_{out}} \) is \( \frac{s}{f_1} \times m \times c_{out} \) if \( P \) is defined as in (3) where each entry can be stored using a single bit and is \( \frac{s}{f_1} \times c_{out} \times 32 \times 3 \) if \( P \) is defined in (4). In our hyperparameter setting, we will set \( s = c_{in} \times f_1 \) and \( m = c_{out} \times f_2 \) where \( f_1 \) and \( f_2 \) are fractional numbers less than 1. In our experiments, we set them as \( \frac{1}{16} \times \frac{1}{4} \times \frac{1}{8} \times \ldots \) and so on. The compression ratio of our proposed method is

\[
\frac{d \times d \times c_{in} \times c_{out} \times 32}{d \times d \times c_{in} f_1 \times c_{out} f_2 + \frac{s}{f_1} \times m \times c_{out} \times 32} 
\]

By considering that the memory cost of storing \( \{P^t\}_{t=1}^{c_{out}} \) 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 \( \frac{s}{f_1} \). 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

\[
\text{speedup} = \frac{d \times d \times c_{in} \times w_{out} \times h_{out} \times c_{out}}{d \times d \times c_{in} \times w_{out} \times h_{out} \times m} = \frac{1}{f_2} 
\]

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. We also evaluate the performance of these algorithms on MNIST (LeCun, Cortes, and Burges 1998) dataset. The results on MNIST dataset can be found in supplementary materials due to page limitation.

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

1. We use three full-precision vectors to store the indices and values of the nonzero elements in sparse matrix \( P \).
Table 1: Results of different networks on CIFAR-10 and CIFAR-100 datasets using VGG-16 Net

| Network       | Compression Ratio | CIFAR-10 Acc(%) | CIFAR-100 Acc(%) |
|---------------|-------------------|-----------------|------------------|
| Full Net (VGG-16) | 1                 | 93.25           | 73.55            |
| LegoNet($f_1 = \frac{1}{2}, f_2 = \frac{1}{4}$) | 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($f_1 = \frac{1}{4}, f_2 = \frac{1}{8}$) | 261.4x           | 89.24           | 62.88            |

- 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 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 (Szegedy, Ioffe, and Vanhoucke 2015) network as the CNN architecture. CIFAR-10 contains 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 $f_1 = \frac{1}{2}$ 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 press 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 217.3x. 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 $f_2 = \frac{1}{4}$, 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 $P$

In our proposed method, the matrix $P$ 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 accuracy 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
Table 2: Results of different networks on CIFAR-10 and CIFAR-100 datasets using ResNet-18 Net

| 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           |

Figure 3: Comparison of our method with and 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.

We also investigate the impact of other commonly used techniques for deep learning training in our model, such as different optimizer and different batch size. More results and detailed discussion can be found in supplementary materials.

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.
Appendix

Proof of Proposition 1

Proposition 2. Suppose \( X_{\text{input}(i)} \in \mathbb{R}^{w_{\text{in}} \times h_{\text{in}} \times s} \), \( \{B_1, \ldots, B_m\} \) is a set of \( m \) low-dimensional binary filters where each \( B_j \in \mathbb{R}^{d \times d \times s} \) and \( P_{\cdot;i}^j \) is the \( i \)-th column in \( P^j \) which is a length-\( m \) sparse vector with only one non-zero element. Then, \( \text{Conv}(X_{\text{input}(i)}, \sum_{j=1}^m P_{\cdot;i}^j B_j) \) is equivalent to \( \sum_{j=1}^m P_{\cdot;i}^j \text{Conv}(X_{\text{input}(i)}, B_j) \).

Proof. Let us divide \( X_{\text{input}(i)} \) into \( p = w_{\text{out}} \times h_{\text{out}} \) patches and each patch \( G_j \) is with size \( \mathbb{R}^{d \times d \times s} \). We can vectorize each patch and form a matrix \( \tilde{X} = [\text{vec}(G_1), \text{vec}(G_2), \ldots, \text{vec}(G_p)] \in \mathbb{R}^{d^2 \times p} \). Similarly, we can vectorize each low-dimensional binary convolution filter \( B_{\cdot;i} \) and form a matrix \( \tilde{B} = [\text{vec}(B_1), \text{vec}(B_2), \ldots, \text{vec}(B_m)] \in \mathbb{R}^{d^2 \times m} \). Based on the definition of convolution operation and applying the associative property of matrix multiplication, we have

\[
\text{Conv}(X_{\text{input}(i)}, \sum_{j=1}^m P_{\cdot;i}^j B_j) = \tilde{X}^T (\tilde{B} P_{\cdot;i}^T) = (\tilde{X}^T \tilde{B}) P_{\cdot;i}^T.
\]

(15)

Note that \((\tilde{X}^T \tilde{B}) P_{\cdot;i}^T\) in (15) can be rewritten as \( \sum_{j=1}^m P_{\cdot;i}^j \text{Conv}(X_{\text{input}(i)}, B_j) \) which can be interpreted as we first perform convolution operations on \( X_{\text{input}(i)} \) using \( \{B_1, \ldots, B_m\} \) to generate \( m \) intermediate feature maps \( \{\text{Conv}(X_{\text{input}(i)}, B_j)\}_{j=1}^m \) and then select one feature map from them using sparse vector \( P_{\cdot;i}^T \).

Additional Experimental Results

Results on MNIST using LeNet-5

Table 3: Results of different networks on MNIST dataset

| Network          | Compression Ratio | Acc(%)  |
|------------------|-------------------|---------|
| Full Net (LeNet-5) | 1                 | 99.48   |
| LegoNet \((f_1 = \frac{1}{2}, f_2 = \frac{1}{2})\) | 15.7               | 99.34   |
| BC               | ~ 32              | 98.82   |
| BWN              | ~ 32              | 99.38   |
| BNN              | ~ 32              | 98.60   |
| XNOR-Net         | ~ 32              | 99.21   |
| SLBF \((f_1 = 1, f_2 = \frac{1}{2})\) | 21.65             | 99.27   |
| SLBF \((f_1 = 1, f_2 = \frac{3}{4})\) | 23.57             | 99.09   |
| SLBF \((f_1 = 1, f_2 = \frac{5}{6})\) | 24.67             | 98.96   |

We present our experiment settings and results on MNIST dataset (LeCun, Cortes, and Burges, 1998) in this section. MNIST dataset consists of 60,000 training samples and 10,000 test samples. Each sample is a 28 × 28 pixel grayscale handwritten digital image. The convolutional network architecture we used for MNIST data is the LeNet-5 (LeCun et al., 1998) which has two convolution layers followed by a Max-Pooling layer and two full-connection layers. We use the same CNN architecture for all seven methods. The setting of batch normalization and optimizer is the same as experiments on VGG-16 and ResNet-18.

According to the results in Table 3 all methods can obtain > 98% accuracy on this dataset and the difference among them is very small in regard to classification accuracy. The compression ratio for LegoNet is not high since it uses full-precision low-dimensional filters for stacking, thus our method obtains higher compression ratio compared to LegoNet. However, the compression ratio is still less than binary networks because of the full-precision scaling factors we used on this dataset. As shown in the experiments on CIFAR-10 and CIFAR-100, the compression ratio has been improved greatly when applying deeper network.

The Impact of Initialization

In the step 1 of our training algorithm, we can either randomly initialize the proxy variables \( \{R_1, \ldots, R_m\} \) or initialize them from a pre-trained full-precision LegoNet model. In here, we compare the accuracy of our method with random initialization and initialization with pre-trained model on CIFAR-10 dataset applying VGG-16. As shown in Figure 5 even though initialization from pre-trained model can achieve significant higher accuracy than random initialization in the very beginning, both two initialization methods yield very similar accuracy after a certain number of epochs.

Figure 5: Comparison of our method \((f_1 = \frac{1}{2}, f_2 = \frac{1}{2})\) with different parameter initialization techniques

Figure 6: Comparison of our method \((f_1 = \frac{1}{2}, f_2 = \frac{1}{2})\) using different optimizers
Figure 7: Comparison of our method \((f_1 = \frac{1}{2}, f_2 = \frac{1}{2})\) using different batch size.

**The Impact of Optimizer**

Our default optimizer is SGD with the momentum of 0.9. We also evaluate the performance of our method using another popular optimizer ADAM [Kingma and Ba, 2014]. Figure 6 compares the test accuracy of our method using SGD with momentum and ADAM on CIFAR-10 dataset with \(f_1 = \frac{1}{2}\) and \(f_2 = \frac{1}{2}\). As can be observed from Figure 6, these two optimizers obtain similar results in the end.

**The Impact of Batch Size**

We also evaluate the impact of batch size in our method. We compare the accuracy with four different settings of batch size. As shown in Figure 7, we can observe that larger batch size gets better accuracy in the beginning but they reach a similar accuracy in the end.