LegoNet: Efficient Convolutional Neural Networks with Lego Filters

Zhaohui Yang [1 2 *] **Yunhe Wang** [2] **Hanting Chen** [1 2 *] **Chuanjian Liu** [2]
Boxin Shi [3 4] **Chao Xu** [1] **Chunjing Xu** [2] **Chang Xu** [5]

Abstract

This paper aims to build efficient convolutional neural networks using a set of Lego filters. Many successful building blocks, *e.g.* inception and residual modules, have been designed to refresh state-of-the-art records of CNNs on visual recognition tasks. Beyond these high-level modules, we suggest that an ordinary filter in the neural network can be upgraded to a sophisticated module as well. Filter modules are established by assembling a shared set of Lego filters that are often of much lower dimensions. Weights in Lego filters and binary masks to stack Lego filters for these filter modules can be simultaneously optimized in an end-to-end manner as usual. Inspired by network engineering, we develop a split-transform-merge strategy for an efficient convolution by exploiting intermediate Lego feature maps. The compression and acceleration achieved by Lego Networks using the proposed Lego filters have been theoretically discussed. Experimental results on benchmark datasets and deep models demonstrate the advantages of the proposed Lego filters and their potential real-world applications on mobile devices.

1. Introduction

The success of deep convolutional neural networks (CNNs) has been well demonstrated on a wide variety of computer vision (CV) tasks, such as object recognition (Simonyan and Zisserman, 2014; Szegedy et al., 2015; He et al., 2016; Huang et al., 2017), detection (Ren et al., 2015; Liu et al., 2016), segmentation (He et al., 2017a), and tracking (Luo et al., 2017b). Due to the powerful feature expression ability of CNNs, building deeper networks will result in higher performance, but lead to the need for more resources. For instance, more than 90*MB* memory and 10^9 FLOPs (floating-number operations) are required for launching the ResNet-50 (He et al., 2016), which limits the application of these deep neural networks on mobile phones, laptops, and other edge devices. Thus, we are motivated to explore portable deep neural networks with high performance.

A number of algorithms have been proposed to reduce memory and FLOPs of convolutional networks with different concerns. (Han et al., 2015) introduced pruning, quantization and Huffman coding for generating extremely compact deep models without obviously affecting their accuracy. (Jaderberg et al., 2014) used matrix factorization to decompose spatial structure of kernels. (Li et al., 2016) proposed to learn 2-bit weight and constructed binary neural networks. (Molchanov et al., 2016) investigated Taylor expansions to eliminate side effects caused by removing filters. (He et al., 2017b) fine-tuned the pruned model for higher efficiency. (Gao et al., 2018) pruned useless channels during the inference stage to accelerate. (Wang et al., 2018b) discarded redundant coefficients of parameters in the DCT frequency domain and introduced a data-driven method. (Xie et al., 2018) decompose convolution kernels along channel and spatial dimensions. (Buciluǎ et al., 2006; Hinton et al., 2015; Romero et al., 2014; Polino et al., 2018) introduced teacher-student distillation strategy. (Wu et al., 2018; Juefei-Xu et al., 2017) build light-weight network with depth-wise convolution. (Wang et al., 2017) introduced circulant matrix to construct convolution kernels. Moreover, (Wu et al., 2016) compressed network based on product quantization, (Wang and Cheng, 2017) decomposed a weight matrix into two ternary matrices and a non-negative diagonal matrix to reduce memory and computational complexity. (Rastegari et al., 2016; Courbariaux et al., 2015) further studied the weight binarization problem in deep CNNs. Although these methods can produce very high compression and speed-up ratios, they often involve special architectures and operations (*e.g.* sparse convolution, fixed-point multiplication, and Huffman codebooks), which cannot be directly

[*] This work was done when Zhaohui Yang and Hanting Chen were interns at Huawei Noah's Ark Lab. [1]Key Laboratory of Machine Perception (Ministry of Education), Peking University. [2]Huawei Noah's Ark Lab. [3]National Engineering Laboratory for Video Technology, Peking University [4]Peng Cheng Laboratory. [5]School of Computer Science, University of Sydney.. Correspondence to: Zhaohui Yang <zhaohuiyang@pku.edu.cn>, Yunhe Wang <wangyunhe@pku.edu.cn>, Chang Xu <c.xu@sydney.edu.au>.

Proceedings of the 36th International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019. Copyright 2019 by the author(s).

satisfied on off-the-shelf platforms and hardwares. Most importantly, these methods rely on a pre-trained network of heavy design and the performance of compressed models is usually upper bounded by this particular network.

Besides manipulating well-trained convolutional neural networks, an alternative is to design efficient network architectures for learning representations. A set of network design principles have been developed in network engineering. For example, VGGNets (Simonyan and Zisserman, 2014) and ResNets (He et al., 2016) stack building blocks of the same shape, which reduces the free choices of hyper-parameters. Another important strategy is *split-transform-merge* in Inception models (Szegedy et al., 2015), where the input is split into a few embeddings of lower dimensionalities, transformed by a set of specialized filters, and merged by concatenation. The representational power of large and dense layers can therefore be approximated using this split-transform-merge strategy, while the computational complexity could be considerably lower. These insightful network design principles then produce a number of successful neural networks, *e.g.* Xception (Chollet, 2017), MobileNet (Howard et al., 2017). Shufflenet (Zhang et al., 2017), and ResNeXt (Xie et al., 2017). As filter is the basic unit in constructing deep neural networks, we must ask whether these network design principles are applicable for re-designing filters in deep learning.

In this paper, we propose Lego Networks (LegoNets) that are efficient convolutional neural networks constructed with Lego filters. A set of lower-dimensional filters are discovered and taken as Lego bricks to be stacked for more complex filters, as shown in Fig. 1. Instead of manually stacking these Lego filters, we develop a method to learn the optimal permutation of Lego filters for a filter module. As these filter modules share the same set of Lego filter but with different combinations, we adapt the *split-transform-merge* strategy to accelerate their convolutions, which further decrease the maximum serial FLOPS of standard convolution. Firstly, Lego filters are convolved with splitted part from input features, and then merge the convolved results. Experimental results on benchmark datasets and CNN models demonstrate the superiority of the proposed Lego filters in establishing portable deep neural networks with acceptable performance.

2. Lego Network

In this section, we first define the problem of how to compress deep neural networks from a macro point of view. Then we introduce the concept of Lego Filters (LF), which are basic unit in our efficient convolutional networks. At last, we demonstrate the way of combining Lego filters.

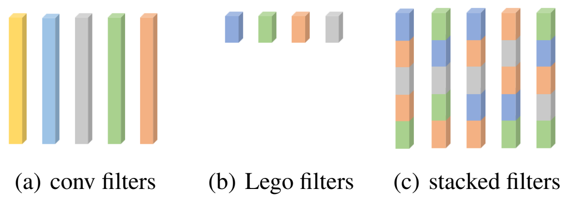


Figure 1. The diagram of convolution filters represented by Lego filters. From left to right are conventional convolution filters, Lego filters with smaller sizes, and convolution filters stacked by exploiting a series of Lego filters

2.1. Lego Filters for Establishing CNNs

Most of existing convolutional neural networks are over-parameterized with numerous parameters and enormous computational complexity. It is common to have more than one thousand of parameters in convolution filter (*e.g.* $3 \times 3 \times 128 = 1152$), but it will produce only one convolution response for a given 3×3 input patch. There could be considerable redundancy in these learned convolution filters.

To reduce the required number of parameters in deep neural networks, some works proposed to decompose high-dimensional convolution filters into different efficient representations. For example, (Wu et al., 2016) exploited vector quantization. (Zhang et al., 2016) utilized singular value decomposition (SVD). (Kim et al., 2015) applied tensor decomposition, and (Cheng et al., 2015) replaced convolution filters by circulate matrices.

Although these schemes make tremendous efforts to represent weights in deep CNNs with less parameters, most of them are proposed to compress and accelerate pre-trained neural networks, which cannot be directly applied for learning CNNs from scratch. In addition, the performance of compressed neural networks is usually worse than that of original models, due to the loss caused by quantization or decomposition. Therefore, we are motivated to alleviate the redundancy between these filters during the stage of network design instead of waiting for solutions after all filters have been optimized through back propagation.

Given an arbitrary convolutional layer $\mathcal{L}$ with its n convolution filters $F = \{f_i, ..., f_n\} \in \mathbb{R}^{d \times d \times c \times n}$, where $d \times d$ is the size of filters and, c is the channel number, the convolution operation can be formulated as

$$Y = \mathcal{L}(F, X), \qquad (1)$$

where X and Y are input data and output features of this layer. Convolution filters F are then solved from the following minimization problem by exploiting the feed-forward and back-propagation strategy, *i.e.*

$$\hat{F} = \arg\min_{F} \frac{1}{2} ||\hat{Y} - \mathcal{L}(F, X)||_F^2 \qquad (2)$$

where $\hat{Y}$ is the ground-truth of desired output of this layer, and $||\cdot||_F$ is the Frobenius norm for matrices.

Here, we propose a set of smaller filters $B = \{b_1, ..., b_m\} \in \mathbb{R}^{d\times d\times\tilde{c}\times m}$ with fewer channels ($\tilde{c} \ll c$), namely Lego filters, and apply them to establish

$$F = \mathcal{G}(b_1, b_2, ..., b_m), \tag{3}$$

where $\mathcal{G}$ is a linear transformation for stacking these Lego filters. Though F in Eq. 3 looks like a classical filter in Eq. 1, we take it as a filter module, as it is the assembled with Lego filters. Each Lego filter can be utilized for multiple times in constructing a filters module F, as shown in Figure 1. Hence, the attention of the optimization problem Eq. 2 has been turned to these Lego filters B of fewer parameters, *i.e.*

$$\hat{B} = \arg\min_{B} \frac{1}{2}||Y, \mathcal{L}(\mathcal{G}(B), X)||_2^F. \tag{4}$$

We can stack $\hat{B}$ to construct F using Eq. 3 and then calculate the output data by exploiting the conventional convolution operation. The compression can be achieved if

$$\frac{d\times d\times c\times n}{d\times d\times\tilde{c}\times m} = \frac{c\times n}{\tilde{c}\times m} > 1. \tag{5}$$

Note that, there is a constraint over the number of channels of Lego filters. In practice, we need to select $o = c/\tilde{c}$ Lego filters from B to stack a general convolution filter, and o should be an integer for subsequent processing.

2.2. Learning to Stack Lego Filters

A new convolution framework with Lego filters was proposed in Eq. 4 to reduce the space complexity of convolution filters in deep CNNs, and a transformation $\mathcal{G}$ for stacking Lego filters is proposed. In fact, we can design many ways to stack Lego filters, *e.g.* random projection and circulate matrix.

Admittedly, the optimal transformation $\mathcal{G}$ can be also learned during the training procedure of deep CNNs if it is exactly a linear projection, and different filters would have their own combinations of Lego filters. Dividing X into $q = H'\times W$ patches and verctorizing them, we have $\mathbf{X} = [\text{vec}(x_1), ..., \text{vec}(x_q)] \in \mathbb{R}^{d^2c\times q}$. The output and filters in $\mathcal{L}$ can be reformulated as $\mathbf{Y} = [\text{vec}(y_1), ..., \text{vec}(y_n)]$ and $\mathbf{F} = [\text{vec}(f_1), ..., \text{vec}(f_n)] \in \mathbb{R}^{d^2c\times n}$, respectively. The conventional convolution operation as described in Eq. 1 can be rewritten as

$$\mathbf{Y} = \mathbf{X}^\top\mathbf{F}. \tag{6}$$

Then, according to Eq. 3, we further divide the input data $\mathbf{X}$ into $o = c/\tilde{c}$ fragments $[\mathbf{X}_1, ..., \mathbf{X}_o]$, and stack m Lego filters to a matrix $\mathbf{B} = [\text{vec}(b_1), ..., \text{vec}(b_m)] \in \mathbb{R}^{d^2\tilde{c}\times m}$.

Algorithm 1 Forward and Backward of LegoNet

```
Require: Hyper-parameter $o$, $m$. Network architecture $\mathcal{N}$.
    Total training iterations $n$. Learning rate $\eta$.
 1: Initialize Lego Filters $\mathbf{B}$, float gradient accumulator $\mathbf{N}$
    for each convolution layer $\mathcal{L}_1, \ldots, \mathcal{L}_k$ by using $o$ and
    $m$. Task criterions $\mathcal{C}$.
 2: for iter = 1 ... n do
 3:     Get mini-batch data $\mathbf{X}$, target $\mathbf{Y}$.
 4:     Calculate $\mathbf{M}$ for each layer using $\mathbf{N}$ according to
        Eq. 9.
 5:     Construct convolution filters $F$ for each layer using
        lego filters $\mathbf{B}$ and binary matrix $\mathbf{M}$. Filters are con-
        structed as $F = \mathbf{BM}$(Eq. 3).
 6:     Forward LegoNet $\mathcal{N}(\mathbf{X})$ with stacked convo-
        lution kernels $F$, get prediction $\mathbf{P}$, $\mathbf{Y} =$
        $\sum \mathbf{X}^\top(\mathbf{BM})$(Eq. 7).
 7:     Calculate loss $\mathbf{L}$ using prediction $\mathbf{P}$ and ground truth
        $\mathbf{Y}$, $\mathbf{L} = \mathcal{C}(\mathbf{P}, \mathbf{Y})$.
 8:     Backward gradients related to parameters $\mathbf{B}$ and $\mathbf{M}$,
        which denoted as $\Delta\mathbf{B}$ and $\Delta\mathbf{M}$.
 9:     For each convolution layer, backward gradients $\mathbf{M}$
        to parameters $\mathbf{N}$ according to $\mathbf{N}$ using STE, which
        denoted as $\Delta\mathbf{N}$.
10:     Update parameters $\mathbf{B}, \mathbf{N}$ in Network $\mathcal{N}$, $\hat{\mathbf{B}} = \mathbf{B} -$
        $\eta\times\Delta\mathbf{B}$, $\hat{\mathbf{N}} = \mathbf{N} - \eta\times\Delta\mathbf{N}$.
11: end for
Ensure: Trained Network $\mathcal{N}^*$
```

Since output feature maps are calculated by accumulating convolution response extracted from all fragments of the input data, for the j-th feature maps $\mathbf{Y}^j$ generated by the j-the convolution filter, *i.e.* the j-th column in $\mathbf{F}$ the convolution opearation using Lego filters can be formulated as

$$\mathbf{Y}^j = \sum_{i=1}^{o} \mathbf{X}_i^\top(\mathbf{B}\mathbf{M}_i^j), \tag{7}$$

where $\mathbf{M}_i^j \in \{0,1\}^{m\times 1}$ and $||\mathbf{M}_i^j|| = 1$ is a vector mask for selecting only one Lego filter from $\mathbf{B}$ to process the i-th fragment of the input data. Therefore, the objective function for simultaneously learning Lego filters and their combination is

$$\begin{aligned} &\min_{\mathbf{B},\mathbf{M}^j} \sum_{i=1}^{o} \frac{1}{2}||\mathbf{Y}^j - \mathbf{X}_i^\top(\mathbf{B}\mathbf{M}_i^j)||_F^2, \\ &s.t.\ \mathbf{M}_i^j \in \{0,1\}^{m\times 1},\ \ ||\mathbf{M}_i^j||_1 = 1, i = 1, ..., o. \end{aligned} \tag{8}$$

By minimizing the above function, we can obtain m Lego filters with corresponding n masks for stacking them to original convolution filters, as illustrated in Figure 1. By using masks, Lego filters could construct complete convolution filters as Fig. 1(c) shows.

2.3. Optimization

The proposed convolution operation needs the cooperation between Lego filters and stacking masks, which are to be optimized in the training procedure of deep neural network, *i.e.* $\mathbf{B}$ and $\mathbf{M}$ in Eq. 8. Since $\mathbf{M}_i^j \in \{0,1\}^{m\times 1}$ is a binary matrix and optimizing $\mathbf{M}$ is a NP-hard problem, which makes it difficult to discover an optimal result using SGD. We thus proceed to relax the object function for learning $\mathbf{M}$.

We introduce $\mathbf{N} \in \mathbb{R}^{n\times o\times m}$ with the same shape as $\mathbf{M}$. During training, $\mathbf{M}$ is binarized from $\mathbf{N}$ as follow,

$$\mathbf{M}_{i,k}^j = \begin{cases} 1, & if\ k = \arg\max \mathbf{N}_i^j \\ 0, & otherwise \end{cases} \tag{9}$$
$$s.t.\ \ j = 1,\dots,n,\ i = 1,\dots,o.$$

During forward, Eq. 9 is used to produce binary mask $\mathbf{M}$, however, it is a step function which is undifferentable. To enable gradients to pass through the binary mask, we refer to the Straight Through Estimator (STE) (Hubara et al., 2016). STE strategy is popular in training Binary Neural Network like (Rastegari et al., 2016). For any undifferentable transformation Eq. 9, the gradient $\Delta\mathbf{N}$ for float parameters $\mathbf{N}$ is same with the gradient $\Delta\mathbf{M}$ for output feature $\mathbf{M}$.

Compared to Binary Neural Network with binarized weights and activations in $\{-1,1\}$, our target matrix $\mathbf{M}$'s weights are in $\{0,1\}$. Besides, there is no constraint on the number of each value in binary neural network, while we have the constraint $\mathbf{M}$, $||\mathbf{M}_i^j||_1 = 1$, which constraints Lego filters are concatenated brick by brick. The training pipeline is summarized in Alg. 1.

3. Efficient Implementation of Lego Filters

A two-stage approach underlines Eq. 7, that is concatenation and convolution. Lego filters firstly construct convolution filters and apply convolution onto input feature maps. However, repeated convolutions will be introduced during the convolution stage. For example, if two filter modules j_1 and j_2 contain same Lego filter at the same position, i.e. $M_i^{j_1} = M_i^{j_2}, i \le o$, as shown in Fig. 1(c) , their convolve convolution results will be exactly the same, *i.e.*

$$\mathbf{X}_i^\top M_i^{j_1} = \mathbf{X}_i^\top M_i^{j_2}. \tag{10}$$

Towards an efficient convolution using Lego filters, we propose a three stage pipeline, *split-transform-merge*. In the split stage, input feature maps are split into o fragments. In the transform stage, these fragments are convolved with each individual Lego filter, which leads to $o\times m$ intermediate feature maps in total. At last, these intermediate feature maps are summed according to $\mathbf{M}$. We argue that this three stage convolution is equivalent to the aforementioned two-stage operation.

1. **Split:** We split input feature maps $\mathbf{X} \in \mathbb{R}^{d^2c\times q}$ into o fragments $[\mathbf{X}_1,\dots,\mathbf{X}_o]$, where each fragment $\mathbf{X}_i \in \mathbb{R}^{d^2\tilde{c}\times q}$ will be the basic feature map to be convolved with Lego filters.

2. **Transform:** Taking feature fragment $\mathbf{X}_i, i\le o$ as the basic component for convolution, for each Lego filter $\mathbf{B}_j, j\le m$, we can calculate the convolution as

$$\mathbf{I}_{ij} = \mathbf{X}_i^\top \mathbf{B}_j. \tag{11}$$

   By launching this convolution between each feature fragment and each Lego filter, there would be $o\times m$ intermediate Lego feature maps $\mathbf{I}_{i,j}, i\le o, j\le m$ in total.

   We name this process as Lego Convolution, as the classical convolution operation is split into convolutions over many smaller fragments cut from the original input feature map. Note that the major float operations are done in this stage, which could reduce the total number of float operations compared to the standard convolution operation.

3. **Merge:** In this stage, the desired output feature maps $\mathbf{Y}$ is produced from intermediate Lego feature maps $\mathbf{I}$. However, in standard convolution, o different Lego kernels have to be concatenated for a complete convolution filter first, and then conduct convolutions with input feature map$\mathbf{X}$. In the above split and transform stages, we have pre-calculated convolution results between input feature fragments and Lego filters and recorded them as intermediate Lego feature map. It is instructive to note that in previous two-stage convolution, $\mathbf{M}$ is used to select Lego filter, while in the proposed three-stage pipeline, $\mathbf{M}$ is used for picking Lego feature maps from $\mathbf{I}$ and summarizing them.

Equivalent We next proceed to prove the equivalence between the proposed efficient three-stage convolution and the standard convolutions using Lego filters in Theorem. 1.

Theorem 1. *Suppose we reverse the convolution by split-transform-merge three-stage pipeline, the result should be equal to concat lego filters* $\mathbf{B}$ *using* $\mathbf{M}$ *to form standard kernels* $\mathbf{K}$ *and then convolve with feature map* $\mathbf{X}$.

Proof. In two-stage convolution, we calculate output feature maps $\mathbf{X}$ by firstly constructing a complete convolution filter and then conduct convolutions over input feature maps $\mathbf{X}$,

$$\mathbf{Y} = \mathbf{X}^\top(\mathbf{B}\mathbf{M}) \tag{12}$$

For the j'th output $\mathbf{Y}_j$, the corresponding can be written as,

$$\mathbf{Y}^j = \sum_{i=1}^{o} \mathbf{X}_i^\top(\mathbf{B}\mathbf{M}_i^j). \tag{13}$$

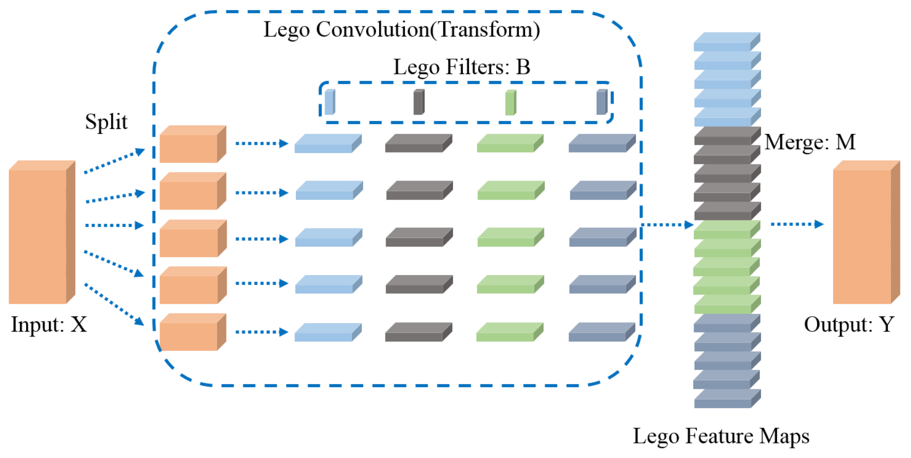


Figure 2. Lego Unit(LU). This figure shows how the three-stage pipeline split-transform-merge operates on input feature maps. $\mathbf{X}$ is the input feature map, lego filters $\mathbf{B}$ are convolved with different parts from $\mathbf{X}$, which result in intermediate feature maps $\mathbf{I}$. Output feature map $\mathbf{Y}$ is generated by merging according to $\mathbf{M}$.

From the perspective of matrix, $\mathbf{Y}^j$ can be calculated as,

$$\mathbf{Y}^j = \sum_{i=1}^{o}(\mathbf{X}_i^\top \mathbf{B})\mathbf{M}_i^j. \tag{14}$$

The first item $\mathbf{X}_i^\top \mathbf{B}$ denotes the intermediate Lego feature map $\mathbf{I}$. $\mathbf{M}$ selects feature maps from $\mathbf{I}$ and summarizes them to generate output $\mathbf{Y}$. Hence, the proposed split-transform-merge strategy can result in the same output feature map. □

A convolution layer reformed by the proposed split-transform-merge pipeline can be equivalent to the two-stage construct-convolve solution. By using this split-transform-merge pipeline, repeated computations are eliminated, and the network could feed forward efficiently. Efficient Lego networks can be established using Lego Unit shown in Fig. 2.

4. Analysis on Compression Performance

Compared with standard convolution, convolution kernels constructed by Lego filters can greatly reduce the number of parameters. Further, by using our proposed split-transform-merge convolution strategy for Lego filters, neural network calculation could be accelerated. In this section, we analyze the memory and float operations in detail.

4.1. Compression

We define the size of the convolution kernel F as $d^2 \times c \times n$ and the size of the input feature map $\mathbf{X}$ as $d_x^2 \times c$. We divide that input channel into o segments and have m Lego Filters at hand. All parameters are saved with float-32 data type except for $\mathbf{M}$ matrix with binary weights. The compression rate is calculated by

$$\frac{n \times c \times d^2}{m \times \frac{c}{o} \times d^2 + n \times o \times m +} \approx \frac{n \times o}{m}. \tag{15}$$

In the denominator, $m \times \frac{c}{o} \times d^2$ denotes the memory occupied by Lego filter takes. $n \times o \times m$ is the memory for $\mathbf{M}$. Since the binary matrix $\mathbf{M}$ is relatively small compared to the Lego filter parameters, the total compression ratio for convolution layer would be approximately $\frac{n \times o}{m}$.

By using Lego filters to construct filter modules according to binary matrix $\mathbf{M}$, we can save a large volume of parameters, which make compressed network applicable for mobile devices.

4.2. Acceleration

In order to accelerate inference time, we develop the split-transform-merge strategy, which can largely reduce the number of float operations in LegoNet. For a standard convolution layer, float operations number is calculated as

$$n \times c \times d^2 \times d_x^2, \tag{16}$$

For Lego networks, firstly generating standard convolution filter using Lego filters $\mathbf{B}$ and binary matrix $\mathbf{M}$, and then conducting convolve as usual would not increase any extra float operations. Hence, Eq. 16 provides an upper bound of the number of float operations. In other words, even in the worst case, applying Lego filters will not increase computational burden.

In some cases, if the number of Lego filters m is smaller than the output channel number n, it could be optimized using split-transform-merge strategy. The theoretical speed up for an optimized convolution layer can be calculated as

$$\frac{n \times c \times d^2 \times d_x^2}{m \times o \times \frac{c}{o} \times d^2 \times d_x^2 + n \times o \times d_x^2} \approx \frac{n}{m}. \tag{17}$$

$m \times o \times \frac{c}{o} \times d_k^2 \times d_x^2$ is number of float operations required by Lego convolution. It would generate $m \times o$ intermediate Lego features with channel equal to 1. In the merge stage, $n \times o \times d_x^2$ FLOPS are taken to sum up these Lego feature maps.

5. Experiments

Experiment Setup. We have developed a novel convolution framework using Lego filters in the above section, here we will first test the proposed method on the CIFAR-10 (Krizhevsky and Hinton, 2010) dataset with different parameters and settings. The CIFAR-10 dataset consists of 60000 natural images with 32×32 resolution split into train/test fold. We select the VGGNet-16 (Simonyan and Zisserman, 2014) as the baseline model, which is a classical deep neural network and has a 93.25% accuracy on the CIFAR-10 benchmark. This network contains 13 convolution layers, followed by 3 fully-connected layers, which is widely used in visual recognition, detection and segmentation tasks. In order to apply VGGNet-16 on the CIFAR10 dataset, we replace the last convolutional layer by a global average pooling layer and then reconfigure a fully-connected layer for conducting the classification task with 10 categories. The network will be trained 1,000 epochs with the batch size of 128 using the conventional SGD. The initial learning rate is set as 0.1, which will be reduced by a factor of 10 at 200, 400, 600, 800, 900 epochs, respectively. Weight decay is set to 5×10^{-4} for regularization. The momentum parameter is set to 0.9. In addition, images in the training set are firstly padded by 4 pixels, and 32×32 patches will be randomly sampled for padded images for data augmentation. Code is available at github [1].

Binary *v.s.* Weighted. Conventional filters in CNNs are divided into two parts by using the proposed approach, *i.e.* Lego filters $\mathbf{B}$ and corresponding binary mask $\mathbf{M}$ for recording their permutations. In order to solve these two variables efficiently, an intermediate variable $\mathbf{N}$ was introduced in Eq. 9 for relaxing the constrain of the binary mask $\mathbf{M}$. Therefore, besides to permute Lego filters to conventional filters using $\mathbf{M}$, we also can utilize $\mathbf{N}$ to obtain another permutation of Lego filters with $o \times n$ floating numbers to assign each Lego filter a learnable weight. Considering that, there are $d^2 \times \tilde{c} \times m$ parameters in the given conventional layer, these coefficients do not account for an obvious proportion for storing the entire neural network. Thus, we first test the performance of Lego networks with and without additional coefficients on Lego filters.

[1]https://github.com/zhaohui-yang/LegoNet_pytorch

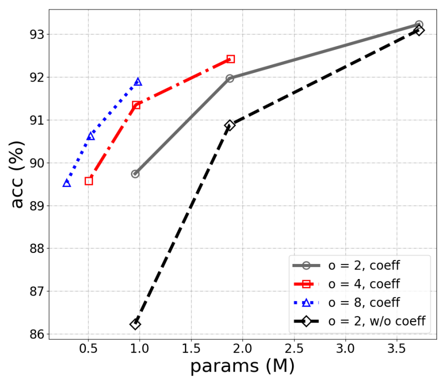


Figure 3. Impact of two parameters o and m, o indicates how many fragments input feature maps are splitted into. m is set to 0.125, 0.25 and 0.5 times to the original output features. Upper line is LegoNet with coefficients, which verify the impact of introduced coefficients.

Impact of Parameters. We evaluate the performance of our proposed LegoNet as described in the previous section on CIFAR10 dataset.

Lego filters could construct convolution filters with and without coefficients while concatenating, in order to full explore the impact of coefficients, we test LegoNet with a range of compression ratios. We set hyper-parameter o to be 2 for whole network, which indicates that input features are splitted into two fragments, $m \in \mathbb{R}^+$ indicates the ratio of Lego filters compared to the original output channels. We set m ranging from 0.125 to 0.5, which compress VGGNet-16 by a factor of 4-16×. Fig. 3 shows the results, for any compression ratio, Lego filters concatenating with coefficients(denoted as $o = 2, coeff$) always performs better than directly concatenating without coefficients(denoted as $o = 2, w/o\ coeff$). Under same parameters budget, by introducing few more coefficients, LegoNet would enhance the expression ability by a large margin. As compression ratio increases, coefficients play an more important role, in the extreme compression situation of 16×, LegoNet with coefficient could maintain performance about 90% accuracy, compared to 86% accuracy of LegoNet without coefficient. We argue that if parameters are not too few, parameters are enough to learn comparable results, *e.g.* , Lego-VGGNet-16-w(o=2,m=0.5) in Tab. 1. However, if VGGNet-16 is compressed extremely, using Lego filters would introduce many repeat calculations among different filter modules. We thus need few coefficients for weighted concatenation to strength the expression ability. Further experiments are all conducted with coefficients during concatenating.

There are two parameters o and m in LegoNet, *i.e.* , o indicates how many fragments input feature maps are splitted into, m indicates the number of Lego filters compared to the

Table 1. Comparison results of different neural networks on the CIFAR-10 datasets.

Model	Acc (%)	Params(M)	Comp ratio	FLOPS(M)	Speed Up
VGGNet-16(Simonyan and Zisserman, 2014)	93.25	14.7	1×	298.7	1×
Lego-VGGNet-16-w(o=2,m=0.5)	93.23	3.7	4×	149.4	2×
Lego-VGGNet-16-w(o=2,m=0.25)	91.97	1.9	8×	74.7	4×
Lego-VGGNet-16-w(o=4,m=0.5)	92.42	1.9	8×	149.4	2×
Lego-VGGNet-16-w(o=4,m=0.25)	91.35	0.9	16×	74.7	4×

original of each layer. We conduct our experiments on different o and m which compress VGGNet-16 by a factor of 4-64×. Fig. 3 shows the relationship between performance and two parameters.

As mentioned above, Lego-VGGNet-16-w could compress the network by a factor of m/o. When we set different o or m to achieve a compression ratio less than 8×, accuracy drops less than 1%, *e.g.* , Lego-VGGNet-16-w(o=4, m=0.5) in Tab. 1. As the parameter grows, the accuracy will increase, which in consistent with our motivation, however, this will lead to larger model size or much more flops. There is thus a trade-off between accuracy, model size and speed.

In order to analysis the relationship between params and flops, as previous figures show, under the same budget of parameters, the performance are approximately the same. The number of parameters directly indicates the final performance of the network. Under the budget of approximately same parameters, higher o which indicates much more fragments, which could achieve higher performance, *e.g.* , Lego-VGGNet-16-w(o=2, m = 0.25) achieves 91.97% accuracy, which is almost the same accuracy with Lego-VGGNet-16-w(o=4, m = 0.5) with 92.42% accuracy. However, float operations vary a lot for two networks. Lego-VGGNet-16-w(o=4, m = 0.5) costs twice flops compared to model Lego-VGGNet-16-w(o=2, m = 0.25). Note that using our proposed three-stage strategy by split-transform-merge, the number of FLOPS is proportional to the number of Lego filters for each layer. Thus, under same parameters budget, though larger o introduce higher performance, but takes much more flops. Take flops into consideration, in order to balance the model size and flops, we set $o = 2$ in the rest of our experiments, which reduce a large amount of flops while maintain comparable accuracy.

Large-Scale Datasets. We test our LegoNet on a large-scale classification task, ILSVRC2012, with several different architectures. We evaluate LegoNet based on ResNet50 (He et al., 2016), VGGNet-16 (Simonyan and Zisserman, 2014) and MobileNet (Howard et al., 2017) network architecture. Images are resized with 256 pixels for shorter side and 224 × 224 pixels patchs are randomly sampled from resized image as data augmentation. Each batch contains 256 images for training. Center crop is used for testing. We trained 300 epochs in total. Learning rate started with 10^{-1} and decayed by a factor of 10 every 80 epochs. Note that although the last fully connected layer for classification has lots of parameters, if we use Lego filters to compress the last layer, many classes would share similar features, which would introduce side effect on performance, especially fine-grained classification. Besides, if the network backbone is used in tasks like detection and segmentation, the last fully connected layer is replaced by other layers, so we do not compress the last fully connected layer in all our experiments.

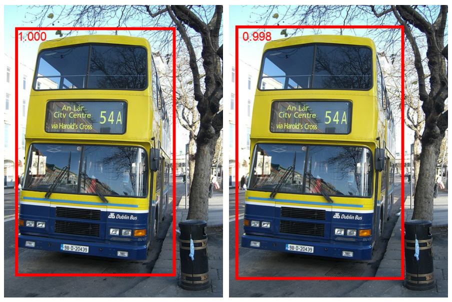
In VGGNet-16 network, 138M parameters are mainly occupied by fully connected layer, which requires a large amount of memory resources. However, this could be removed by introducing Global Average Pooling(GAP) after all convolution layers, about 10% parameters left after removing fully connected layers, with almost same accuracy. Then a 1000 classes fully connected layer followed by a softmax layer is used for classification. We apply our Lego filters onto VGGNet-16-GAP network. Lego-VGGNet-16-w(o=2,m=0.5) compressed original VGGNet-16 by a factor of approximately 30× and achieved comparable performance as Tab. 2 shows. Such a small model would be sufficient for mobile devices. As VGGNet-16 network has been largely used in many different computer vision tasks, deploying such a small Lego-VGGNet-16-w(o=2,m=0.5) could satisfy most of the needs. 2× float operations are reduced which speed up inference time.

ResNet50 usually contains 1x1 and 3x3 convolutions. 1x1 convolution layers are mainly used for channel-wise transformation and 3x3 convolution is used to merge spatial features. We used the same compression method as that on CIFAR10, thus setting parameter o to be 2 and controls the number of Lego filters m. We compress two types of layers without difference. In the ResNet50 network, the convolutional feature extractor is followed by classification layer of the network. The final classification layer occupies 2M parameters. Tab. 2 shows Lego-Res50 with 2-3 × compression ratio. Accuracy keeps to be almost the same with 3× compression ratio. Meanwhile, float operations reduced a lot in these networks by approximately 2×. Compared to the original, Lego-Res50-w(o=2,m=0.5) is a more portable alternative to the original.

In addition, we evaluate LegoNet-Res50 with and without coefficients on large scale dataset. Compared to weighted concatenation of Lego filters, Lego-Res50(o=2,m=0.5)

Table 2. Comparison results of different neural networks on the ILSVRC2012 datasets.

Model	Top-5 Acc(%)	Params(M)	Comp Ratio	FLOPs(B)	Speed Up
ResNet50 (He et al., 2016)	92.2	25.6	1.0×	4.1	1.0×
ThiNet-Res (Luo et al., 2017a)	88.3	8.7	2.9×	2.2	1.9×
Versatile (Wang et al., 2018a)	91.8	11.0	2.3×	3.0	1.4×
Lego-Res50(o=2,m=0.5)	89.7	8.1	3.2×	2.0	2.0×
Lego-Res50-w(o=2,m=0.5)	90.6	8.1	3.2×	2.0	2.0×
Lego-Res50-w(o=2,m=0.6)	91.3	9.3	2.8×	2.0	1.7×
VGGNet-16 (Simonyan and Zisserman, 2014)	90.1	138.0	1.0×	15.3	1.0×
ThiNet-VGG (Luo et al., 2017a)	90.3	38.0	3.6×	3.9	3.9×
Lego-VGGNet-16-w(o=2,m=0.5)	88.9	4.2	32.9×	7.7	2.0×
Lego-VGGNet-16-w(o=2,m=0.6)	89.2	5.0	27.6×	9.2	1.7×
MobileNet (Howard et al., 2017)	88.9	4.2	1.0×	0.6	1.0×
Lego-Mobile-w(o=2,m=0.9)	87.5	2.5	1.7×	0.5	1.1×
Lego-Mobile-w(o=2,m=1.5)	88.3	3.5	1.2×	0.6	1.0×



(a) FRCNN (b) Lego-FRCNN

Figure 4. Example object detection results on PASCAL VOC dataset.

drops 2% more accuracy, which proves that the introduced coefficients indeed improve LegoNet expression ability. For VGGNet-16 and MobileNet, we only tested the performance which contains coefficients.

Further, we adopt proposed LegoNet onto mobile setting networks, MobileNet (Howard et al., 2017). VGGNet-16 and ResNet50 are designed for a higher classification performance. Given much more parameters, higher accuracy can be achieved. Compressing these kinds of networks while preserving their performance is easier than compressing MobileNet like efficient network designs.

Mobilenet consists depthwise convolutional filters and pointwise convolution which are 3×3 and 1×1 convolution. Depthwise convolution learns a transform for each channel using 3× 3 convolution with group number equals to input channels. 1×1 pointwise convolution takes most of the parameters in MobileNet, thus we mainly adopt our Lego filters onto those 1×1 convolution. We test our proposed Lego-MobileNet-w on ILSVRC2012 and achieved less than 1% accuracy drop with 1.2× compression. Note that for model Lego-Mobile-w(o=2,m=1.5), the number of Lego filters for each layer is 1.5× compared to the original, directly using our proposed split-transform-merge three-stage pipeline, flops is larger than the original. For this model, it reaches the upper bound and we forward this network by firstly reconstruct convolution filters and then forward input data, which achieves same float operations and inference time as the original.

Generalization Ability. In order to full explore the generalization ability of LegoNet, we evaluate our LegoNet on VOC object detection task(Everingham et al., 2010). Faster-RCNN (Ren et al., 2015) was used as the detection framework, VOC07 train+val dataset was used to train the network. We used Lego-Res50-w(o=2,m=0.5) as the detection backbone. Tab. 3 shows the results of trained network. Comparing baseline network, our LegoNet achieves comparable results with much less parameters.

Table 3. Object detection results on the VOC2007 dataset.

Model	mAP(%)	Params(M)
ResNet50	72.8	25.6
Lego-Res50(o=2,m=0.5)-w	71.3	5.0

Here, we give the example of Faster-RCNN detection result. It can be seen from Fig. 4 that the difference between the original ResNet50 and our LegoNet is quite small.

6. Conclusion

In this work, we propose a new method to construct efficient convolutional neural networks with a set of Lego filters. We first define the problem of network compression from the perspective of how to construct convolution filters with a shared set of Lego filters. Then we propose a learning method to simultaneously optimize binary masks and weights in end-to-end training stage. We further develop a split-transform-merge three-stage strategy for efficient convolution. We evaluate LegoNet with different backbones and compare their performance, parameters, float operations and speed up. The proposed LegoNet could combine with any state-of-the-art architecture and can be easily deployed onto mobile devices.

Acknowledgement

This work is supported by National Natural Science Foundation of China under Grant No. 61876007, 61872012 and Australian Research Council Project DE-180101438.

References

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

Y. Cheng, F. X. Yu, R. S. Feris, S. Kumar, A. Choudhary, and S.-F. Chang. An exploration of parameter redundancy in deep networks with circulant projections. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2857–2865, 2015.

F. Chollet. Xception: Deep learning with depthwise separable convolutions. *arXiv preprint*, pages 1610–02357, 2017.

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

M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338, 2010.

X. Gao, Y. Zhao, L. Dudziak, R. D. Mullins, and C. Xu. Dynamic channel pruning: Feature boosting and suppression. *arXiv: Computer Vision and Pattern Recognition*, 2018.

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

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

K. He, G. Gkioxari, P. Dollár, and R. Girshick. Mask r-cnn. In *Computer Vision (ICCV), 2017 IEEE International Conference on*, pages 2980–2988. IEEE, 2017a.

Y. He, X. Zhang, and J. Sun. Channel pruning for accelerating very deep neural networks. *international conference on computer vision*, pages 1398–1406, 2017b.

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

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

G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *CVPR*, volume 1, page 3, 2017.

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

M. Jaderberg, A. Vedaldi, and A. Zisserman. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*, 2014.

F. Juefei-Xu, V. Naresh Boddeti, and M. Savvides. Local binary convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 19–28, 2017.

Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530*, 2015.

A. Krizhevsky and G. Hinton. Convolutional deep belief networks on cifar-10. *Unpublished manuscript*, 40(7), 2010.

H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

J.-H. Luo, J. Wu, and W. Lin. Thinet: A filter level pruning method for deep neural network compression. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 5068–5076. IEEE, 2017a.

W. Luo, P. Sun, F. Zhong, W. Liu, and Y. Wang. End-to-end active object tracking via reinforcement learning. *arXiv preprint arXiv:1705.10561*, 2017b.

P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz. Pruning convolutional neural networks for resource efficient transfer learning. *CoRR, abs/1611.06440*, 2016.

A. Polino, R. Pascanu, and D. Alistarh. Model compression via distillation and quantization. *arXiv preprint arXiv:1802.05668*, 2018.

M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*, 2014.

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

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

P. Wang and J. Cheng. Fixed-point factorized networks. *computer vision and pattern recognition*, pages 3966–3974, 2017.

Y. Wang, C. Xu, C. Xu, and D. Tao. Beyond filters: Compact feature map for portable deep model. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3703–3711. JMLR. org, 2017.

Y. Wang, C. Xu, X. Chunjing, C. Xu, and D. Tao. Learning versatile filters for efficient convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1615–1625, 2018a.

Y. Wang, C. Xu, C. Xu, and D. Tao. Packing convolutional neural networks in the frequency domain. *IEEE transactions on pattern analysis and machine intelligence*, 2018b.

B. Wu, A. Wan, X. Yue, P. Jin, S. Zhao, N. Golmant, A. Gholaminejad, J. Gonzalez, and K. Keutzer. Shift: A zero flop, zero parameter alternative to spatial convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9127–9135, 2018.

J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng. Quantized convolutional neural networks for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4820–4828, 2016.

G. Xie, T. Zhang, K. Yang, J. Lai, and J. Wang. Decoupled convolutions for cnns. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He. Aggregated residual transformations for deep neural networks. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*, pages 5987–5995. IEEE, 2017.

X. Zhang, J. Zou, K. He, and J. Sun. Accelerating very deep convolutional networks for classification and detection. *IEEE transactions on pattern analysis and machine intelligence*, 38 (10):1943–1955, 2016.

X. Zhang, X. Zhou, M. Lin, and J. Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. *CoRR*, abs/1707.01083, 2017. URL http://arxiv.org/abs/1707.01083.