Clustering Convolutional Kernels to Compress Deep Neural Networks

Sanghyun Son, Seungjun Nah, and Kyoung Mu Lee(✉)

Department of ECE, ASRI, Seoul National University, Seoul 08826, Korea
{thstkdgus35,kyoungmu}@snu.ac.kr,
seungjun.nah@gmail.com

Abstract. In this paper, we propose a novel method to compress CNNs by reconstructing the network from a small set of spatial convolution kernels. Starting from a pre-trained model, we extract representative 2D kernel centroids using k-means clustering. Each centroid replaces the corresponding kernels of the same cluster, and we use indexed representations instead of saving whole kernels. Kernels in the same cluster share their weights, and we fine-tune the model while keeping the compressed state. Furthermore, we also suggest an efficient way of removing redundant calculations in the compressed convolutional layers. We experimentally show that our technique works well without harming the accuracy of widely-used CNNs. Also, our ResNet-18 even outperforms its uncompressed counterpart at ILSVRC2012 classification task with over 10x compression ratio.

Keywords: CNNs · Compression · Quantization · Weight sharing Clustering

1 Introduction

The recent era of computer vision witnessed remarkable advances from deep learning. The analysis presented in [35] shows that CNNs not only figure out the scene types but also well recognizes spatial patterns. Therefore, state-of-the-art convolutional neural networks [15,16,31] and their variants apply to a broad range of problems such as image classification, object detection, segmentation, image restoration, etc. However, most of the CNNs are designed to be executed on high-end GPUs with substantial memory and computational power. In mobile or embedded environments where computational resources are limited, those networks need to be compressed for practical applications [12,34].

Most of the studies on network compression have investigated to figure out redundancies of weights [6] and unnecessary parameters [14,24]. Then, those parameters can be removed while preserving the original performance of the model. In [7,20], the weight matrices and tensors were factorized and approximated to low-rank for efficient computation. Pruning became popular in recent works [10,13,26,27] since it directly saves storage and computations. On the

V. Ferrari et al. (Eds.): ECCV 2018, LNCS 11212, pp. 225–240, 2018.
https://doi.org/10.1007/978-3-030-01237-3_14

other side, the quantization based approaches have also become common. Extensive studies on binary and ternary networks [4,5,12,17,25,29,39] proved that even 1 or 2 bits parameters can make CNNs work. Other works tried to quantize adjacent elements of a weight tensor as a form of vector quantization [9,34]. Those methods utilize k-means clustering so that we can express the compressed model in the form of codebook and indices. Especially, [34] extracted a sub-vector from the weight tensor in channel dimension to exploit product quantization.

In this paper, we propose a more structured quantization method that is built upon 2D convolution kernels. As the convolution itself is inherently *spatial*, we opt to use a spatial slice of weight tensors as a unit to be compressed. Unless mentioned otherwise, we denote these 2D slices as kernels. Under our expression, a weight tensor of a single convolutional layer is composed of $C_{\text{out}} \times C_{\text{in}}$ number of kernels where C_{out} and C_{out} denote the output and input channels, respectively. For example, widely used VGG-16 [31] and ResNets [15] consist of more than a million 3×3 kernels.

Similarly to the vector quantization methods [9,34], we perform clustering on 3×3 kernels and replace the redundant kernels with their centroids. Therefore, we represent the compressed model with a set of centroids and a corresponding cluster index per each kernel. Thus, kernels that have the same index share their weights. While maintaining the compressed state, we train our model through the weight-sharing. We also present methods to accelerate the convolution when the same centroid repeatedly appears in a single layer.

Our compression method brings following contributions and benefits. First, we propose a new method to compress and accelerate the CNN by applying k-means clustering to 2D kernels. To the best of our knowledge, this is the first approach on network compression that considers the redundant spatial patterns of kernels. Second, our transform invariant clustering method extends the valid number of kernel centroids with geometric transforms. Our improved experimental results imply the transform invariance imposes regularization effect. Lastly, our extensive experiments show that our method is generally applicable to various CNN architectures and datasets. In particular, our compressed ResNet [15] achieves higher accuracy on ILSVRC12 image classification than the original model at over 10x compression ratio.

2 Related Works

Network quantization is one of the typical approaches to compress deep neural networks. It focuses on reducing the number of bits to represent each parameter. Earlier works utilized the weight sharing and indexed representation of the parameters to save the storage. Han et al. [12] first demonstrated that 2^5 distinctive weights are enough for a single convolutional layer, and proposed 5-bit quantization of CNN. To save more storages, HashedNets [1] utilized a hash function to reduce the overhead from storing index terms. Those methods do not restrict the precision, but the diversity of each parameter by sharing a full-precision weight between similar values. Although their implementations can be tricky, we can train those models with the traditional gradient descent.

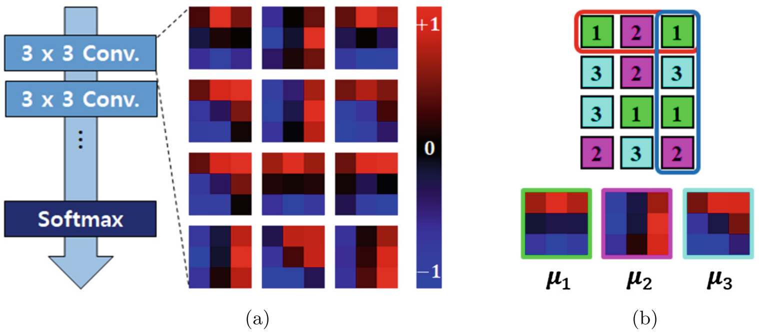


Fig. 1. We compress convolutional layers (a) of given CNN as (b). We apply k-means clustering to the kernels and use indexed representations with a codebook $\mathcal{M} = \{\mu_i | i = 1, \cdots, k\}$. We also accelerate the CNN by removing redundant computations from overlapping kernels in **red** and **blue** boxes. Note that we handle all convolutional layers simultaneously, rather than compress each layer individually. (Color figure online)

On the other side of network quantization, there have been attempts to limit the precision of parameters. It is already proven that binary [4] and ternary [25, 39] weights work well on challenging ImageNet [30] classification task. These methods can even accelerate the CNNs by redefining the convolution in more efficient ways. Recently, [37] proposed a method to represent each weight as a power of two that can utilize high-speed bit-shift operations. Since the major goal of the quantization is resource-efficient neural networks, intermediate features [5, 29] and gradients [11,17,38] can also be quantized, too.

While the above methods mainly focus on reducing the bit-width of each parameter, vector quantization directly quantizes a weight vector and maps it to an index. Although intrinsic high-dimensionality makes it more challenging than the scalar quantization, product quantization works well on compressing fully-connected [9] and convolutional layers [34]. Our work is a particular case of weight sharing and vector quantization, as we assign the same index to multiple kernels and share their weights. However, unlike weight or sub-vector quantization, we quantize 2D kernels which have geometrical meanings in the CNNs.

Network pruning [14,24] aims to remove unnecessary connections from the networks. Usually, small weights are removed from the network by iterative optimization steps [12,13]. However, weight-wise pruning has several limitations in practice due to their irregular structures. Therefore, structured pruning [26,27] methods for the convolutional neural networks are getting popular. As they prune unnecessary convolution kernels of CNNs, they would be much suitable for the practical case. Our algorithm is not directly related to pruning itself. However, we will show that the proposed method can benefit from the pruning as [12] mentioned.

There are several works which utilize the geometric shapes of convolution kernels for efficient CNNs. By exploiting translational [36], reflectance [3], and rotational [8] symmetries, the capacity of CNNs can be increased without any additional parameters. There also have been attempts to manipulate the shapes of convolution kernels intentionally. For example, [23] forced group-wise sparsity to weights of the convolutional layers to accelerate the network. Also, [33] proposed a method to train the arbitrary shape of filters for efficient pruning. In this paper, we propose a compression method that focuses on redundant shapes in a large number of convolution kernels. We also explore various transforms of the kernels as [3,8] did, to achieve higher degrees of weight sharing and regularization.

3 K-means Clustering of Convolution Kernels

Before going into the details, we define the terms to be used in the following descriptions. We will assume that there are total N many kernels in our target CNN and all of them have the same spatial sizes. Then, a weight tensor of m-th convolution layer can be denoted as $\mathbf{w}^m \in \mathbb{R}^{C_{\text{out}} \times C_{\text{in}} \times h \times w}$ whose input $\mathbf{x}$ and output $\mathbf{y}$ have C_{in} and C_{out} channels, respectively. Here, we will omit the term m when we describe the convolutional layer. $\mathbf{x}_i$ refers to a i-th input channel of $\mathbf{x}$, and $\mathbf{y}_j$ is a j-th output channel of $\mathbf{y}$. A kernel $\mathbf{w}_{ij}$ is applied to $\mathbf{x}_i$ and the responses are accumulated to compute $\mathbf{y}_j$ as Eq. (1).

$$\mathbf{y}_j = \sum_{i=1}^{C_{\text{in}}} \mathbf{w}_{ij} * \mathbf{x}_i. \tag{1}$$

In the following subsections, we explain our compression algorithm and its computational benefits. In Sect. 3.1, We formulate our algorithm with a concept of k-means clustering. In Sect. 3.2, we describe a method to train our model. In Sect. 3.3, we demonstrate that the proposed method can accelerate the convolutions. Lastly, in Sect. 3.4, we propose an advanced clustering method that can act as a strong regularization.

3.1 Compact Representation of the Kernels

In general, a CNN is trained without any structural restrictions on the shapes of its weight tensors. Therefore, there may exist N distinctive kernels in the network. Our main strategy is to represent those kernels more compactly with k-means clustering. After grouping the kernels into k many clusters, we replace each kernel $\mathbf{w}_{ij}^m$ in a cluster $\mathcal{W}_n$ to its corresponding centroid $\mu_n = \mathbb{E}_{\mathbf{w}_{ij}^m \in \mathcal{W}_n}\left[\mathbf{w}_{ij}^m\right]$. Then, we apply weight-sharing to the all kernels that belong to $\mathcal{W}_n$ and represent them with their cluster index n. Considering that a single-precision 3×3 kernel requires 36-byte storage, the indexed representation can reduce the model size a lot.

One possible problem is that the distribution of kernel weights can vary across different convolutional layers. In that case, our approach may not derive meaningful representations as k-means clustering cannot find representative centroids.

Therefore, we utilize the basic concept of convolution and normalize all kernels to handle this problem. To put it concretely, we handle the kernels that have similar shapes but different norms together because they show similar behaviors in filtering. Thus, rather than compute the distances between kernels directly from their raw values, we use normalized kernels. Consequently, the k-means clustering objective becomes

$$\operatorname*{argmin}_{\mathcal{M}} \sum_{n=1}^{k} \sum_{\hat{\mathbf{w}}_{ij}^{m} \in \mathcal{W}_n} \|\hat{\mathbf{w}}_{ij}^{m} - \mu_n\|^2, \tag{2}$$

where $\hat{\mathbf{w}}_{ij}^{m} = \mathbf{w}_{ij}^{m} / s_{ij}^{m}$ and $s_{ij}^{m} = sign(\mathbf{w}_{ij*}^{m})\|\mathbf{w}_{ij}^{m}\|^2$. Here, $\mathbf{w}_{ij*}^{m}$ is a center pixel of a kernel. By defining s_{ij}^{m} in this manner, we also cluster the kernels that have similar structures but opposite signs together. A compressed representation of each kernel includes a cluster index l_{ij}^{m} such that $\hat{\mathbf{w}}_{ij}^{m} \in \mathcal{W}_{l_{ij}^{m}}$, and its scale s_{ij}^{m}. From the viewpoint of compression, storing those scale parameters requires additional storage and lowers the compression ratio. However, it enables our method to learn fine-grained centroids and increases the representation power as we disentangle the kernel shapes and norms. Also, we will demonstrate that those scale parameters sometimes can be ignored in Sects. 4.6 and 4.7

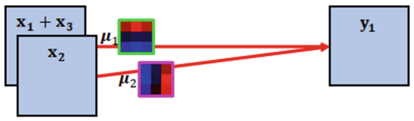
3.2 Training Method

After clustering, we fine-tune the compressed model while maintaining the cluster assignment of each kernel. To do so, we replace a convolution in Eq. (1) with Eq. (3). In other words, we use a centroid $\mu_{l_{ij}}$ with a scale instead of the original kernel $\mathbf{w}_{ij}$. Therefore, our method has two trainable parameter sets: centroids and scales. We can use a standard back-propagation algorithm to train those parameters because Eq. (3) is fully differentiable. Note that a centroid $\mu_{l_{ij}}$ appear at different layers through the network, and therefore a single centroid can take gradients from various layers. We efficiently implemented this algorithm with PyTorch [28] framework, which automatically differentiates the variables through the computational graph.

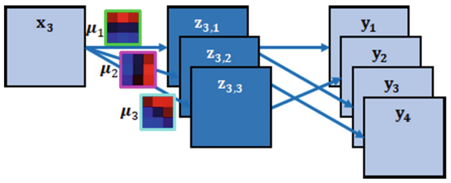
$$\mathbf{y}_j = \sum_{i=1}^{C_{\text{in}}} s_{ij} \mu_{l_{ij}} * \mathbf{x}_i. \tag{3}$$

3.3 Accelerating Convolution via Shared Kernel Representations

Although our primary interest is network compression, we can also accelerate a CNN by reducing the duplicated computations from shared kernel representations. To be specific, there can be two types of redundancy in a convolutional layer. One is duplication among the kernels generating the same output channel from different input channels. The other is a duplication of kernels generating multiple output channels from a single input channel. In both cases, we can compute the 2D convolution only once per each unique centroid.



(a) Add-then-conv



(b) Conv-then-add

Fig. 2. A compressed convolutional layer with $C_{\text{in}} = 3$ and $C_{\text{out}} = 4$ from Fig. 1(b) can be accelerate by two ways. (a) We calculate $\mathbf{x}_1 + \mathbf{x}_3$ first and then apply a shared kernel μ_1. (b) We compute intermediate features $\mathbf{z}_k$ first and accumulate them to $\mathbf{y}_j$. For example, $\mathbf{z}_{3,1}$ is accumulated to $\mathbf{y}_1$ and $\mathbf{y}_3$.

In Fig. 2(a), we add the input channels before convolution to avoid the computations between a single kernel and multiple input features. We provide a detailed explanation of this procedure in Eq. (4). Since we do not need to calculate the convolution when $n \neq l_{ij}$ for $\forall i$, the proposed method requires less computation than the traditional convolution. We call it 'Add-then-conv'.

$$\begin{aligned}
\mathbf{y}_j &= \sum_{i=1}^{C_{\text{in}}} \mu_{l_{ij}} * (s_{ij}\mathbf{x}_i) (\text{associative law}) \\
&= \sum_{i=1}^{C_{\text{in}}} \left(\sum_{n=1}^{k} \mu_n \delta\left[n = l_{ij}\right] \right) * (s_{ij}\mathbf{x}_i) (\text{marginalization}) \\
&= \sum_{n=1}^{k} \mu_n * \left(\sum_{i=1}^{C_{\text{in}}} \delta\left[n = l_{ij}\right] s_{ij}\mathbf{x}_i \right) (\text{associative law}).
\end{aligned} \tag{4}$$

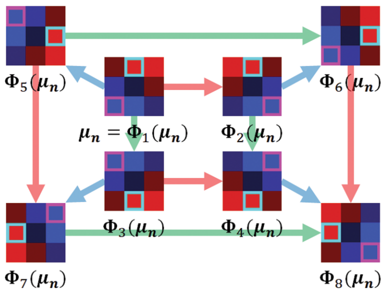
In Fig. 2(b), the responses of all output channels from a single input channel are computed simultaneously. An input channel $\mathbf{x}_i$ responds to multiple centroids and generate $\mathcal{Z}_i = \{\mathbf{z}_{i,l_{ij}} = \mu_{l_{ij}} * \mathbf{x}_i | j = 1, 2, \cdots, C_{\text{out}}\}$. Then, we can compute $\mathbf{y}_j$ as a weighted sum of $\mathbf{z}$ such that

$$\mathbf{y}_j = \sum_{i=1}^{C_{\text{in}}} s_{ij}\mathbf{z}_{i,l_{ij}}. \tag{5}$$

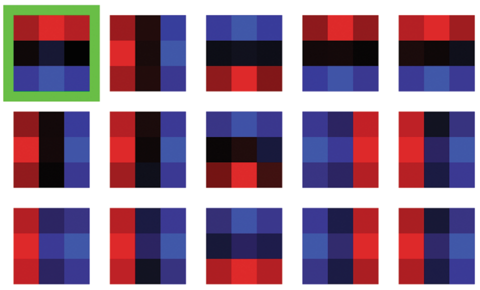
Similarly to above, we call it 'Conv-then-add'. Once the clustering is done, we choose and apply the faster method by comparing the number of computations.

3.4 Transform Invariant Clustering

Since 3×3 kernels have relatively simple structures, many of them are flipped and rotated version of each other. Inspired by this idea, we newly propose the transform invariant clustering (TIC) for the convolutional kernels. The basic idea of TIC is to allow a centroid to represent its transformed shapes, too. Therefore,



(a)



(b)

Fig. 3. (a) We combine **horizontal**, **vertical flips** and 90° **rotation** to represent various kernels using one centroid. Weights that are marked with same colors (**cyan**, **magenta**) share their values. (b) We visualize how TIC actually works. We mark the centroid with green, and the other kernels belong to the same cluster even they have different orientations. (Color figure online)

when m distinct transforms are allowed, k centroids can represent mk shapes. In this paper, we use a maximum of eight different transforms as shown in Fig. 3(a).

With TIC, we use Eqs. (6) and (7) instead of Eqs. (2) and (3). Here, $\mathbf{\Phi}_{t_{ij}^m}(\cdot)$ represents the transformations that are demonstrated in Fig. 3(a). Equation(6) can be optimized like traditional k-means clustering, where a centroid of $\mathcal{W}_n$ is defined as $\mu_n = \mathbb{E}_{\mathbf{w}_{ij}^m \in \mathcal{W}_n}\left[\mathbf{\Phi}_{t_{ij}}^{-1}\left(\mathbf{w}_{ij}^m\right)\right]$. TIC does not further compress and accelerate our model since we require additional bits to store $\mathcal{T}$ which is a set of all t_{ij}^m. However, in Sect. 4.3, we will demonstrate that TIC allows very compact representation of a CNN and acts as a strong regularization term.

$$\underset{\mathcal{M},\mathcal{T}}{\operatorname{argmin}} \sum_{n=1}^{k} \sum_{\hat{\mathbf{w}}_{ij}^m \in \mathcal{W}_n} \left\|\hat{\mathbf{w}}_{ij}^m - \mathbf{\Phi}_{t_{ij}^m}\left(\mu_n\right)\right\|_2^2, \tag{6}$$

$$\mathbf{y}_j = \sum_{i=1}^{C_{\text{in}}} \mathbf{\Phi}_{t_{ij}}\left(s_{ij}\mu_{l_{ij}}\right) * \mathbf{x}_i. \tag{7}$$

4 Experiments

We apply the proposed method to recent popular CNN architectures for image classification task: VGG [31], ResNets [15], and DenseNets [16]. We use CIFAR-10 [21] dataset to evaluate the performance of the compressed models. Our training and test dataset contain 50,000 and 10,000 test images, respectively. We normalize all images using channel-wise means and standard deviations of the training set. During the training, we apply random flip and translation augmentation to input images as previous works [15,16,31] did.

We first train those models for 300 epochs with an SGD optimizer and momentum 0.9. The initial learning rate is 0.1, and we reduce it by a factor of 10 after 150 and 225 epochs. For DenseNets [16], we apply the Nesterov momentum [32]. Also, we use a weight decay of 5×10^{-4} when training VGG-16, and 10^{-4} for the others. Then, we apply our algorithm to those baselines and fine-tune the models for an additional 300 epochs. The learning rate starts from 5×10^{-3}, and other configurations kept same. Since k-means clustering with a million of kernels is computationally very heavy, we implemented it on the multi-GPU environment. We found that random initialization is enough for the k-means seed. Our PyTorch [28] code will be available on https://github.com/thstkdgus35.

4.1 Clustering Kernels from Various Models

We apply our method to a VGG-16 [31] variant first. The original VGG-16 has 13 convolution layers, and three fully-connected layers follow. In our implementation, there is one fully-connected layer after the last pooling. Also, batch normalizations [19] follow after all convolutions. The modified architecture contains 1,634,496 many 3×3 kernels which account over 99.9% of the total parameters. We also compress more recent architectures, ResNets [15] and DenseNets [16]. Although some of their configurations contain the bottleneck structure that comes with many 1×1 convolutions, we use the models without the bottleneck. The baseline ResNet-56 and DenseNet-12-40 have 94,256 and 101,160 kernels, respectively. Similar to VGG-16, 3×3 kernels are also dominant in those models.

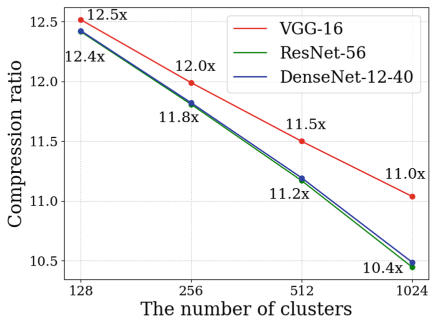
We demonstrate the results in Table 1 with varying the number of clusters k. Since previous works on network compression do not provide unified comparisons, we only show our results here. As we expected, using more centroids results in lower error rates. However, we found that the model performances are reasonable when we use only 128 centroids. Notably, the results from VGG-16 is impressive because the original model contains more than 1.6×10^6 kernels. In other words, over 10,000 kernels of the CNN are sharing their shapes together while maintaining the classification accuracy.

4.2 Analyzing Compression and Acceleration Ratio

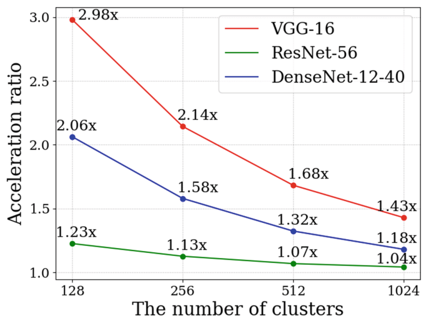
In this section, we analyze how our method can compress and accelerate the CNN. We only count the weights from 3×3 kernels, as they compose the most

Table 1. We compress various CNN architectures on CIFAR10 and report the classification error rates (%). C'k' denote the proposed method with k centroids.

Model	C128	C256	C512	C1024	Baseline
VGG-16 [31]	6.24	6.23	6.16	6.01	5.98
ResNet-56 [15]	6.76	6.54	6.61	6.30	6.28
DenseNet-12-40 [16]	5.44	5.49	5.39	5.38	5.26



(a)



(b)

Fig. 4. Compression(a) and acceleration(b) ratios of various architecture and different k values. Those factors tend to decrease as the number of clusters increases.

parts of our baselines. Also, we assume that the original CNNs requires $b_{\mathbf{w}} = 32$ bits to represent each weight. Our method store an index l_{ij}^m and a scale parameter s_{ij}^m per each kernel $\mathbf{w}_{ij}^m$. After clustering, our centroid representation requires $log_2 k$ and $b_{\mathbf{s}}$ bits for each index and scale parameter, respectively. We assign 16-bit for each s_i for simplicity. We also store a codebook with k centroids which occupy total $kb_{\mathbf{w}}hw$ bits. This codebook can be ignored in many cases as N is much larger than k in usual. To sum up, we calculate the compression ratio r_{comp} as Eq. (8). For the experiments in Table 1, we plot compression ratios with respect to various k values in Fig. 4(a).

$$r_{\text{comp}} = \frac{N b_{\mathbf{w}} hw}{N\left(\log_2 k + b_{\mathbf{s}}\right) + k b_{\mathbf{w}} hw} \simeq \frac{b_{\mathbf{w}} hw}{\log_2 k + b_{\mathbf{s}}}. \tag{8}$$

We also analyze the way to accelerate the CNNs with our method. Here, we ignore the scale parameters without losing generality since their computations are negligible. For the 'Add-then-conv,' let us define λ_j as the number of distinctive centroids that contribute to the computation of $\mathbf{y}_j$. In the case of Fig. 1(b), λ_1 is equal to 2 since only μ_1 and μ_3 are used to compute $\mathbf{y}_1$. As we explained in Sec. 3.3, we first add all input features that share the same kernels and then compute convolutions. Therefore, we can reduce the number of convolutions from C_{in} to λ_j for each output channels.

For the 'Conv-then-add,' we define another variable ν_i that indicates the number of distinctive centroids that are convolved with $\mathbf{x}_i$. In Fig. 1(b), ν_1 is equal to 3 because the long blue box contains three centroids. Therefore, we only compute ν_i many convolutions instead of C_{out} for each $\mathbf{x}_i$. Finally, we calculate the theoretical acceleration factor r_{accel} of each convolutional layer by simply choosing the method which has the fewer FLOPs as illustrated in Eq. (9).

$$r_{\text{accel}} \simeq \frac{C_{\text{in}} C_{\text{out}}}{\min\left(\sum_{j=1}^{C_{\text{out}}} \lambda_j, \sum_{i=1}^{C_{\text{in}}} \nu_i\right)} \tag{9}$$

We visualize how this factor changes in various models as the number of clusters k varies. Here, we found that our method becomes more effective when the number of intermediate channels is large. For example, VGG-16 achieved the highest acceleration ratio because its largest convolutional layer has 512 input and output channels. Therefore, there can exist many overlapping kernels when $k < 512$. In contrast, ResNet-56 shows almost constant speed because the number of its intermediate channels does not exceed 64. DenseNet has an impressive property because each layer in DenseNet takes all the outputs of previous layers [16]. Therefore, C_{in} is usually very large, and we can remove many convolutions. Although the DenseNet-12-40 and ResNet-56 have the similar number of kernels, the acceleration factor of DenseNet is much higher due to its unique architecture.

In this paper, we only report the theoretical FLOPs because the proposed method has some compatibility issues with cuDNN [2]. However, our algorithm can be implemented using group convolution.

4.3 Transform Invariant Clustering for Compact Representation

We apply transform invariant clustering on VGG-16 to see how the proposed method can benefit from higher degrees of weight sharing. We first allow vertical flips, and then horizontal flips to take four types of transforms into account during clustering. Finally, we also consider the rotational transform [3,8] in the clustering to acquire very compact centroid representation. Note that the valid number of representative kernels are 256 in all three configurations in the experimental results shown in Table 2.

Since transform indication bits are additionally required to include transforms, the gains in compression ratio are trivial. For example, our 'T' models in Table 2 consume similar amounts of storage with a VGG-16-C256 model in Fig. 4, having compression ratio of 11.99. However, we observed that TIC has a strong regularization effect on redundant kernel shapes. When we exploited the vertical symmetry of the kernels, VGG-16-C128-TIC2 achieves lower classification errors compared to the baseline. Note that the performance has dropped a little bit when we include the horizontal transforms (TIC4) because the number of centroids is limited to 64. However, it has a compatible performance compared to VGG-16-C256 in Table 1, while having much more compact representations. TIC8 suffers a slight performance drop, but it is surprising that only 32 centroids are necessary to represent the VGG-16 with reasonable accuracy.

4.4 Multi Clustering

The proposed method performs better as the number of clusters k increases. However, it is not desirable to set k boundlessly large regarding compression and acceleration ratio. Therefore, we handle this trade-off by dividing the given kernels to $n \geq 2$ sets and applying our method individually. By doing so, we find $\mathcal{M}_1, \cdots, \mathcal{M}_n$ instead of computing a single set of centroids $\mathcal{M}$. If we keep k same for all sets, the number of effective centroids will be nk while we still

Table 2. Our method allows various design choices. ‘M’ refers to the multi clustering, and the number of groups follows. After TIC, the number of allowed transformations follows. We report the error rates on CIFAR-10 dataset. We also report the compression and acceleration ratios with respect to the baseline.

Model	Error (%)	Sizes (MB)	FLOPs (M)
VGG-16-baseline	5.98	58.8	313
VGG-16-C32	6.78	4.29 (13.7x)	42 (7.63x)
VGG-16-C64	6.44	4.49 (13.1x)	68 (4.60x)
VGG-16-C64-M2	6.27	4.50 (13.0x)	75 (4.17x)
VGG-16-C64-M13	6.21	4.51 (13.0x)	78 (4.01x)
VGG-16-C256-M2	6.16	4.91 (12.0x)	159 (1.97x)
VGG-16-C256-M13	6.10	4.96 (11.8x)	166 (1.89x)
VGG-16-C128-TIC2	**5.92**	4.91 (12.0x)	145 (2.16x)
VGG-16-C64-TIC4	6.25	4.91 (12.0x)	145 (2.16x)
VGG-16-C32-TIC8	6.51	4.91 (12.0x)	145 (2.16x)

require $log_2 k$ bits to represent each index. Therefore, we can efficiently increase the network capacity without enlarging the model.

We evaluate the multi clustering method using our VGG-16 [31] variant. It has 13 convolutional layers, so we try two simple dividing strategies because finding an optimal partition is challenging. In the M2 configuration, we apply k-means clustering to the first ten and the last three layers separately, so that both groups contain the similar amount of kernels. Our M13 configuration treats all layers individually. Because the first convolutional layer only contains 192 kernels, we allocate $\min(k, 192)$ clusters for this layer. Although we keep k the same for the other layers, it is possible to assign different k to each group for more elaborate tuning. We provide the results in Table 2. We first compressed VGG-16 with only 64 clusters and observed a slight performance drop, since 25,000 kernels are sharing their weights on average. With the multi clustering, however, the performances get better without any noticeable decrease in the compression ratio. Although the acceleration factors slightly decrease with the multi clustering, we observed meaningful performance gain as the number of effective centroids increases.

4.5 Clustering with Pruning

Prior works on network pruning [10,12,13,26,27] showed that it is possible to remove a large number of unnecessary weights from CNNs without accuracy loss. In this section, we analyze how our method can benefit from pruning. As [12] mentioned, k centroids will provide a better approximation of the network if the number of whole kernels decreases with pruning.

Table 3. We combine our method with pruning and report the error rates on the CIFAR-10 dataset. Here, 'P' denotes the simple pruning strategy we mentioned.

Model	Error (%)	Size (MB)	FLOPs (M)
VGG-16-P	6.08	28.7 (2.05x)	223 (1.41x)
VGG-16-Filter-pruning [26]	5.91	21.0 (2.80x)	206 (1.52x)
VGG-16-C512	6.16	5.11 (11.5x)	186 (1.68x)
VGG-16-PC512	6.13	3.19 (23.5x)	151 (2.07x)
VGG-16-Filter-pruning [26]-C512	5.99	2.35 (31.8x)	149 (2.10x)

Here, we try two pruning strategies to VGG-16 before applying our method. The first method is simple thresholding which removing 50% of the spatial kernels from each layer based on their L_1 norms. For the advanced method, we focus on filter-level pruning [26,27] instead of weight-wise pruning [10,12,13] because we treat the individual kernel for a basic unit of CNNs. We adopt a method from [26] that performs structured pruning [27] to reduce the number of intermediate channels with kernels. Then, we apply the proposed method to pruned models. We found that the pruning works as expected, regardless of the specific strategies. We report the results in Table 3.

4.6 Bottleneck Architectures and Clustering

The bottleneck structure demonstrated its power in recent deep architectures [15,16,18]. It utilizes 1×1 convolutions before 3×3 convolutions to build more efficient CNNs. Although our method is less efficient with 1×1 convolutions, we demonstrate that kernels in the bottleneck architectures are heavily redundant. We selected 100-layer DenseNet-BC ($k = 12$) [16] as our baseline. It contains 27,720 spatial kernels which comprise only 34% of the total parameters. All the other parameters construct 1×1 convolutional layers. In this section, we only compress the 3×3 kernels and report the results in Table 4.

Interestingly, our method works well and even exceeds the baseline without s_i. In other words, we do not normalize the kernels and share them across different layers. When we consider dominant 1×1 kernels, the compression ratio is quite low. However, it is surprising that our method requires only $576 = 64 \times 3 \times 3$ trainable parameters for 3×3 convolutions in C64N configuration. We also applied pruning to DenseNet-BC and successfully removed over 98% of the parameters from 3×3 convolutional layers.

4.7 Experients on ImageNet Dataset

We also evaluate our method on more challenging ILSVRC12 dataset [30] using ResNet-18 [15] in which 3×3 convolutions are dominant and 1×1 kernels are in shortcut connections only. There exist 7×7 convolutions in the first layer, but they are also negligible. To handle the various kernel sizes, we use

Table 4. We apply the proposed method to DenseNet-BC and evaluate those models on the CIFAR-10 dataset. 'N' denotes the model without any scale parameters ($s_i \equiv 1$).

Model	Error (%)	Size (KB) 3×3 / Total	FLOPs (M) 3×3 / Total
DenseNet-BC [16]	4.50	998 / 2967	112 / 288
DenseNet-BC-C64	4.50	77 (13.0x) / 2046 (1.45x)	54 (2.09x) / 230 (1.25x)
DenseNet-BC-C128	4.51	82 (12.2x) / 2051 (1.45x)	66 (1.71x) / 242 (1.19x)
DenseNet-BC-C64N	4.60	22 (45.5x) / 1991 (1.49x)	43 (2.59x) / 219 (1.32x)
DenseNet-BC-C128N	**4.44**	27 (37.5x) / 1996 (1.49x)	57 (1.96x) / 233 (1.24x)
DenseNet-BC-PC128N	4.57	14 (61.7x) / 1983 (1.50x)	36 (3.11x) / 212 (1.36x)

Table 5. Error rates of compressed ResNet-18 on the ILSVRC12 validation set. INQ [37] does not report the acceleration performance explicitly in the paper.

Model	Top-1 / Top-5 Error (%)	Compression ratio	Acceleration ratio∗
ResNet-18 [15] from torchvision [28]	30.2 / 10.9	-	-
BWN [29]	39.2 / 17.0	32.0x	2.00x
TWN [25]	34.7 / 13.8	16.0x	2.00x
TTQ [39]	33.4 / 12.8	16.0x	2.00x
INQ (3-bit) [37]	31.9 / 11.6	10.7x	-
INQ (5-bit) [37]	31.0 / 10.9	6.40x	-
C256 (Proposed)	30.5 / 11.0	11.1x	1.68x
C1024 (Proposed)	**30.1** / **10.7**	10.3x	1.27x
C1024N (Proposed)	32.3 / 12.2	23.6x	1.35x

multi clustering method from Sect. 4.4. Kernels of equal sizes belong to the same group, and we ignore 1×1 convolutions here. We use $k = 64$ clusters for 7×7 kernels to keep as much information as possible from the inputs and preserve the variety of the large kernels. Here, our primary interest is to compress 1,220,608 many 3×3 kernels. We compare the proposed method with extreme weight quantizations [25,29,37,39] that perform well on challenging dataset. Results are reported in Table 5.

An impressive property of our method is that we allow various design choices. For example, we can ignore the scale parameters and achieve 23.6x compression ratio or enhance the model accuracy by sacrificing compactness. INQ [37] also has a design parameter to control the size-performance trade-off, but our method outperforms other methods at given accuracy or compression ratio. However, it is difficult to compare the computational gain of the proposed method and previous works directly since different compression methods take advantage of the different hardware designs. For example, our method can utilize the fastest convolution algorithm [22] since we do not modify the convolution itself. On the other hand, INQ [37] redefines the convolution with shift operations, which can be implemented very efficiently on the custom hardware. Although we report the acceleration ratio of each method in Table 5, we cannot say which way is the fastest in general. The more detailed analysis is beyond our scope.

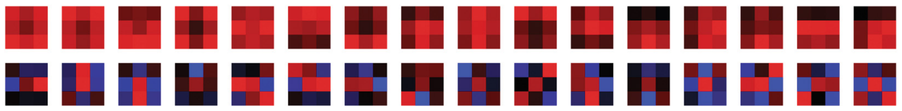


Fig. 5. The 16 most(Top)/least(Bottom) frequent centroids from ResNet-18-C256. They occupy 19% and 2% of the total kernels, respectively.

5 Conclusion

In this paper, we compress CNNs by applying k-means clustering to their convolution kernels. Our primary interest is to remove redundancies in convolutional layers by sharing weights between similar kernels. We reduce the size and required computations of modern CNNs while keeping their performances. Various experiments about TIC and multi clustering show that the proposed method has multiple design choices for compression. Combined with pruning, our compressed VGG-16 achieved over 30x compression ratio with only 0.01% accuracy drop on the CIFAR-10 dataset. The proposed method also fits for the challenging ImageNet task and even enhances the accuracy of ResNet-18. Although our method is not fully optimized for bottleneck architectures yet, we will handle them in our future works. Also, we will further enhance our method based on Fig. 5 as frequently appearing centroids are often low-frequency and low-rank.

Acknowledgement. This work was partially supported by the National Research Foundation of Korea(NRF) grant funded by the Korea Government(MSIT). (No. NRF-2017R1A2B2011862)

References

1. Chen, W., Wilson, J., Tyree, S., Weinberger, K., Chen, Y.: Compressing neural networks with the hashing trick. In: 32nd International Conference on Machine Learning, pp. 2285–2294 (2015)
2. Chetlur, S., et al.: cuDNN: Efficient primitives for deep learning. arXiv preprint arXiv:1410.0759 (2014)
3. Cohen, T., Welling, M.: Group equivariant convolutional networks. In: 33rd International Conference on Machine Learning, pp. 2990–2999 (2016)
4. Courbariaux, M., Bengio, Y., David, J.P.: Binaryconnect: training deep neural networks with binary weights during propagations. In: Advances in Neural Information Processing Systems, pp. 3123–3131 (2015)
5. Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., Bengio, Y.: Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or -1. arXiv preprint arXiv:1602.02830 (2016)
6. Denil, M., Shakibi, B., Dinh, L., De Freitas, N., et al.: Predicting parameters in deep learning. In: Advances in Neural Information Processing Systems, pp. 2148–2156 (2013)
7. Denton, E.L., Zaremba, W., Bruna, J., LeCun, Y., Fergus, R.: Exploiting linear structure within convolutional networks for efficient evaluation. In: Advances in Neural Information Processing Systems, pp. 1269–1277 (2014)

8. Dieleman, S., De Fauw, J., Kavukcuoglu, K.: Exploiting cyclic symmetry in convolutional neural networks. In: 33rd International Conference on Machine Learning, pp. 1889–1898 (2016)
9. Gong, Y., Liu, L., Yang, M., Bourdev, L.: Compressing deep convolutional networks using vector quantization. arXiv preprint arXiv:1412.6115 (2014)
10. Guo, Y., Yao, A., Chen, Y.: Dynamic network surgery for efficient DNNs. In: Advances in Neural Information Processing Systems, pp. 1379–1387 (2016)
11. Gupta, S., Agrawal, A., Gopalakrishnan, K., Narayanan, P.: Deep learning with limited numerical precision. In: 32nd International Conference on Machine Learning, pp. 1737–1746 (2015)
12. Han, S., Mao, H., Dally, W.J.: Deep compression: compressing deep neural networks with pruning, trained quantization and huffman coding. In: International Conference on Learning Representations (2016)
13. Han, S., Pool, J., Tran, J., Dally, W.: Learning both weights and connections for efficient neural network. In: Advances in Neural Information Processing Systems, pp. 1135–1143 (2015)
14. Hassibi, B., Stork, D.G.: Second order derivatives for network pruning: optimal brain surgeon. In: Advances in Neural Information Processing Systems, pp. 164–171 (1993)
15. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)
16. Huang, G., Liu, Z., Weinberger, K.Q., van der Maaten, L.: Densely connected convolutional networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (2017)
17. Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., Bengio, Y.: Quantized neural networks: training neural networks with low precision weights and activations. J. Mach. Learn. Res. **18**(187), 1–30 (2018)
18. Iandola, F.N., Han, S., Moskewicz, M.W., Ashraf, K., Dally, W.J., Keutzer, K.: Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5 mb model size. arXiv preprint arXiv:1602.07360 (2016)
19. Ioffe, S., Szegedy, C.: Batch normalization: accelerating deep network training by reducing internal covariate shift. In: 32nd International Conference on Machine Learning, pp. 448–456 (2015)
20. Jaderberg, M., Vedaldi, A., Zisserman, A.: Speeding up convolutional neural networks with low rank expansions. arXiv preprint arXiv:1405.3866 (2014)
21. Krizhevsky, A., Hinton, G.: Learning multiple layers of features from tiny images (2009)
22. Lavin, A., Gray, S.: Fast algorithms for convolutional neural networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4013–4021 (2016)
23. Lebedev, V., Lempitsky, V.: Fast convnets using group-wise brain damage. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2554–2564 (2016)
24. LeCun, Y., Denker, J.S., Solla, S.A.: Optimal brain damage. In: Advances in Neural Information Processing Systems, pp. 598–605 (1990)
25. Li, F., Zhang, B., Liu, B.: Ternary weight networks. arXiv preprint arXiv:1605.04711 (2016)
26. Li, H., Kadav, A., Durdanovic, I., Samet, H., Graf, H.P.: Pruning filters for efficient convnets. In: International Conference on Learning Representations (2017)

27. Luo, J.H., Wu, J., Lin, W.: ThiNet: A filter level pruning method for deep neural network compression (2017)
28. Paszke, A., et al.: Automatic differentiation in PyTorch (2017)
29. Rastegari, M., Ordonez, V., Redmon, J., Farhadi, A.: XNOR-Net: ImageNet classification using binary convolutional neural networks. In: Leibe, B., Matas, J., Sebe, N., Welling, M. (eds.) ECCV 2016. LNCS, vol. 9908, pp. 525–542. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46493-0_32
30. Russakovsky, O., et al.: ImageNet large scale visual recognition challenge. Int. J. Comput. Vis. **115**(3), 211–252 (2015)
31. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)
32. Sutskever, I., Martens, J., Dahl, G., Hinton, G.: On the importance of initialization and momentum in deep learning. In: 30th International Conference on Machine Learning, pp. 1139–1147 (2013)
33. Wen, W., Wu, C., Wang, Y., Chen, Y., Li, H.: Learning structured sparsity in deep neural networks. In: Advances in Neural Information Processing Systems, pp. 2074–2082 (2016)
34. Wu, J., Leng, C., Wang, Y., Hu, Q., Cheng, J.: Quantized convolutional neural networks for mobile devices. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4820–4828 (2016)
35. Zeiler, M.D., Fergus, R.: Visualizing and understanding convolutional networks. In: Fleet, D., Pajdla, T., Schiele, B., Tuytelaars, T. (eds.) ECCV 2014. LNCS, vol. 8689, pp. 818–833. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10590-1_53
36. Zhai, S., Cheng, Y., Zhang, Z.M., Lu, W.: Doubly convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1082–1090 (2016)
37. Zhou, A., Yao, A., Guo, Y., Xu, L., Chen, Y.: Incremental network quantization: Towards lossless CNNs with low-precision weights. In: International Conference on Learning Representations (2017)
38. Zhou, S., Wu, Y., Ni, Z., Zhou, X., Wen, H., Zou, Y.: DoReFa-Net: training low bitwidth convolutional neural networks with low bitwidth gradients. arXiv preprint arXiv:1606.06160 (2016)
39. Zhu, C., Han, S., Mao, H., Dally, W.J.: Trained ternary quantization. In: International Conference on Learning Representations (2017)