Associative Convolutional Layers

Hamed Omidvar* **Vahideh Akhlaghi*** **Hao Su** **Massimo Franceschetti** **Rajesh K. Gupta**
Electrical and Computer Engineering (ECE) & Computer Science and Engineering (CSE) Departments
University of California San Diego

Abstract

We provide a general and easy to implement method for reducing the number of parameters of Convolutional Neural Networks (CNNs) during the training and inference phases. We introduce a simple trainable auxiliary neural network which can generate approximate versions of "slices" of the sets of convolutional filters of any CNN architecture from a low dimensional "code" space. These slices are then concatenated to form the sets of filters in the CNN architecture. The auxiliary neural network, which we call "Convolutional Slice Generator" (CSG), is unique to the network and provides the association among its convolutional layers. We apply our method to various CNN architectures including ResNet, DenseNet, MobileNet and ShuffleNet. Experiments on CIFAR-10 and ImageNet-1000, without any hyper-parameter tuning, show that our approach reduces the network parameters by approximately $2\times$ while the reduction in accuracy is confined to within one percent and sometimes the accuracy even improves after compression. Interestingly, through our experiments, we show that even when the CSG takes random binary values for its weights that are not learned, still acceptable performances are achieved. To show that our approach generalizes to other tasks, we apply it to an image segmentation architecture, Deeplab V3, on the Pascal VOC 2012 dataset. Results show that without any parameter tuning, there is $\approx 2.3\times$ parameter reduction and the mean Intersection over Union (mIoU) drops by $\approx 3\%$. Finally, we provide comparisons with several related methods showing the superiority of our method in terms of accuracy. [1]

Proceedings of the 24th International Conference on Artificial Intelligence and Statistics (AISTATS) 2021, San Diego, California, USA. PMLR: Volume 130. Copyright 2021 by the author(s).

1 Introduction

For the training of large CNNs, where distributed machine learning approaches are typically used, communication constraints present a key challenge, as gradients of the network parameters need to be communicated among different nodes (Wang et al., 2018). Similarly, in Federated learning, a neural network is continuously optimized and customized in a distributed manner using numerous users' devices (Konečnỳ et al., 2016) where the size and implementation efficiency of these networks are also critical.

A possible solution to these problems is reducing the number of parameters of the CNN in a way such that its performance is not tangibly affected. Most of today's techniques, however, either focus on the inference phase and do not reduce the number of parameters during the training phase (while maintaining the accuracy of the network), or they reduce the number of parameters during the training, but their accuracy loss, computational burden, or implementation cost is considerable (Cheng et al., 2018). This paper seeks to overcome these limitations by introducing a novel plug-and-play approach to reduce the number of parameters of any CNN architecture.

Our proposed method exploits the inherent redundancy in the parameters of the convolutional filters by partitioning the set of filters of convolutional layers and representing these partitions in a low dimensional latent space. To obtain this low-dimensional representation of the CNN filters, we introduce an auxiliary

[1] All our implementations using Pytorch, alongside the datasets of our results (train accuracy, test accuracy, train loss, test loss, train time, and test time for all the epochs), trained models for Pytorch (when possible), and detailed documentations for our codes are available online: https://github.com/hamedomidvar/associativeconv
* First and second authors had equal contributions.

neural network, called Convolutional Slice Generator (CSG), that can be used in conjunction with any CNN architecture. The CSG, which is a linear network shared among all the convolutional layers, generates four dimensional tensors called "slices" that correspond to the above-mentioned cross-filter partitions, from a low dimensional "code" space. These slices are then concatenated to form sets of convolutional filters of all the layers of the original architecture (see Fig. 1). Thus, by design, this compression approach preserves the original CNN architecture (while approximating its set of filters), can be applied in conjunction with many other methods for additional gains, and can be used during both the training and inference phases.

During the training of the CNN, the code vectors, which lie in a space of cardinality $\approx$ 20$\times$ smaller than the cardinality of their corresponding slice of the convolutional filters, are trained. We have explained our method on how this compression ratio is achieved eliminating the need for tuning this parameter. The auxiliary neural network (CSG) can either be trained alongside the code vectors or be provided to the network in advance with pre-trained and fixed parameters. Due to the simplicity of our method, using a recent result (Allen-Zhu et al., 2019) and for a simplified architecture, we have theoretically shown the convergence of training in our approach which is backed by our experimental evaluations.

We apply our proposed technique to several CNN architectures used for classification and semantic segmentation tasks and compare it with most of the state of the art methods that are comparable to our approach. Our experiments on classification tasks show that while this approach significantly reduces the cardinality of the parameter space of the CNN, the resulting networks, except in extreme compression cases, still achieve top-1 accuracies that are within one percent of the accuracies of the original CNNs. Our approach for some modern wide architectures improves the original accuracy by a compression ratio of $\approx 2\times$, which is compatible with a general trend (Cheng et al., 2018) that wider networks can tolerate more compression. In case of narrow networks, when our technique maintains the accuracy within one percent of the original ones, other compression methods lead to higher accuracy degradation with similar compression ratios. CSG with binary weights (called BCSG) which helps with simplifying the computations to generate the CNN parameters by converting multiplications and additions to only additions is also studied. BCSG can improve the timing of networks on embedded devices through reducing off-chip and on-chip memory accesses. It is observed that even when the BCSG with fixed random binary (-1,+1) values are used during the training, the resulting networks still have acceptable performances. To further confirm the generality of our approach, experimental results show the possibility of applying it to architectures that are used for semantic segmentation tasks without significant performance degradation ($\approx$ 2.3$\times$ parameter reduction leads to $\approx$ 3% in the mean Intersection over Union (mIoU)).

We note that one could argue that in this work we trade computation efficiency for parameter efficiency, and hence communication and storage efficiency. However, as discussed and explored experimentally in the supplementary materials, the added computational cost is negligible in practice and with customized hardware for edge devices, our approach is also expected to improve timing performance.

1.1 Related Works

In this section we briefly review some of the techniques used for reducing the number of parameters of deep neural networks (DNN) and CNNs.

Pruning: To improve the inference time of DNN models, *pruning* the network parameters and network connections have been proposed (Han et al., 2015; Li et al., 2016; Anwar et al., 2017). These methods, however, are only applicable to the networks after the training phase of the original network with original number of parameters. Extra training and fine tuning are also required to recover the accuracy degradation. Other pruning-based techniques that modify the training phase such as (Yu et al., 2019; Frankle and Carbin, 2019) are also available, however, they either do not reduce the number of parameters during the training phase or they significantly add to the computational burden and hence are not desired for distributed training.

Knowledge Distillation (KD): These methods (Chen et al., 2017a) focus on reducing the number of network parameters, both during the training and inference phases. However, these techniques assume that the parameters of the original network are readily available.

Quantization-Based Methods: These methods are among the very successful methods for reducing the computational burden of DNNs that can be used during both training and inference phases (Hubara et al., 2017; Köster et al., 2017; Rastegari et al., 2016). Excessive quantization of gradients in the training phase, however, can lead to significant reduction in the model's accuracy (Rastegari et al., 2016). In addition, most of these methods require elaborate modifications to the model and its training and inference processes, and sometimes specialized hardware, to achieve acceptable results, and hence

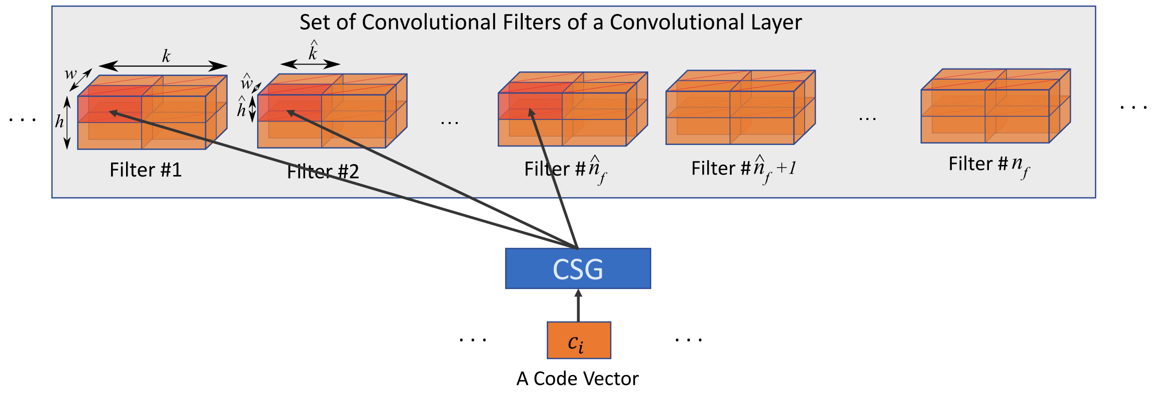

Figure 1: Generation of a regular-shaped but approximate set of filters from the concatenation of slices. Each slice of a set of convolutional filters is generated from a code vector using a shared CSG. The generated filters are then used by the corresponding convolutional layer as in regular CNNs. The proposed method can be applied to any filter shape. In this example there are n_f filters with k channels and $h \times w$ kernels. Each slice, generated by the CSG, is assumed to be $\hat{n}_f \times \hat{k} \times \hat{h} \times \hat{w}$. The figure shows one slice in darker color, that spans across multiple channels and multiple filters, and its corresponding code vector (see Sections 2 and 3).

are not always easy to implement.

Efficient Fast-Fourier-Transform (FFT): These approaches exploit the computational efficiency of FFT-based multiplications (Abtahi et al., 2018; Ding et al., 2017). To be useful, these schemes require complex multiplications and efficient implementations of FFT. There are also methods based on the Winograd algorithm (Winograd, 1980) for performing efficient convolutions in the real domain (Lavin and Gray, 2016). We note that these approaches to compress and accelerate operations in the fully connected layers or to accelerate the convolution operations can yield additional gains when combined with our method.

Parallel Training and Gradient Compression: These techniques are concerned with performing different stages of the training in parallel, or to reduce the amount of information that needs to be communicated between different nodes of the distributed computation network using compression, or quantization of the gradients (Wang et al., 2018; Lin et al., 2018; Ye and Abbe, 2018; Li et al., 2014; Recht et al., 2011; Wangni et al., 2018). However, these works are not concerned with the architecture of the network nor on how the filters are designed, and can be applied to any architecture including our CSG-augmented CNNs. While some of these methods provide lossless compression for the gradients, others lead to significant accuracy loss or need elaborate modifications.

Neural Architecture Search (NAS): These methods (Ghiasi et al., 2019) and methods like (Wang et al., 2017) require the training of many architectures to find a well performing architecture. Custom designed networks such as (Howard et al., 2017; Zhang et al., 2018) are tiny networks for mobile devices and focus on reducing the complexity of 1×1 convolutions and hence they do not provide general plug-and-play methods for reducing the number of parameters during the training of a given full architecture.

Structured Convolutional Filters & Steerable Filters: *Structured convolutional filters* have been explored at the intersection of signal processing and computer vision. In (Jacobsen et al., 2016) the authors, inspired by scattering networks (Sifre and Mallat, 2013; Bruna and Mallat, 2013; Mallat, 2012), introduce a structured method based on the family of Gaussian filters and its smooth derivatives, to produce the CNN filters from basis functions that are learned during the training phase. *Steerable filter* design has been studied for about three decades (Freeman and Adelson, 1991).

Low-Rank Decomposition: Methods based on *low-rank tensor decomposition* such as Canonical Polyadic decomposition (Lebedev et al., 2015), Singular Value Decomposition based methods (Tai et al., 2015), and other related method such as (Jaderberg et al., 2014), focus on finding a low-rank decomposition of the filters in order to achieve a network with improved inference time. However, these methods require training on the full set of parameters. In addition, compute-heavy decomposition methods such as (Tai et al., 2015), significantly slow down the training of CNNs. Therefore, due to their inability to reduce the number of trainable parameters and significantly slower training time for some of them, they are not comparable with our approach and are not included in our comparisons.

Separable & Transferred Convolutional Filters: Methods such as *separable* (Mamalet and Garcia,

2012) and *transferred convolutional filters* (Shang et al., 2016; Cohen and Welling, 2016), exploit the equivariant group theory. They reduce the number of trainable parameters and accelerate the training. For instance, (Rigamonti et al., 2013) show that multiple image filters can be approximated by a shared set of separable (rank-1) filters, and the authors in (Shang et al., 2016; Cohen and Welling, 2016) reuse the filters, allowing large speedups with minimal loss in accuracy. (Ha et al., 2017) provides an approach that is close to ours, however, they use a multi-layer network to generate the filters of the network and their approach is only studied and achieves acceptable results on wider networks.

In Section 4 we provide extensive experimental comparisons with a range of related methods. (Cheng et al., 2018) provides a comprehensive introduction and comparisons of most of the above-mentioned methods. *The difference between our approach and these methods is two folds. First, we use a single linear network (CSG) to generate the convolutional filters of the entire neural network. Second, we reproduce these filters by concatenating approximate slices that can expand across multiple dimensions.*

1.2 Our Contributions

We present three distinct contributions:

- **An easy-to-implement kernel compression method.** We provide a general method for reducing the number of parameters that are needed to represent the sets of filters of convolutional layers during both the training and inference phases, through the use of a linear auxiliary neural network which transforms a set of code vectors in a low dimensional space to slices of sets of convolutional filters. Accompanying repository mentioned before shows how this method can be easily implemented in software.
- **Analytic bounds.** Using a simple CSG-augmented CNN as an example, we show that the training time for this network is polynomial in the number of data points, number of input features (e.g., pixels), and inverse of the minimum distance between data points. Further, we provide an estimate on the relationship between the size of the slices and the cardinality of the code vector space that eliminates the need for tuning for these parameters. This analysis also suggests that our approach can be applied to at least a large set of architectures.
- **Experimental validation.** We apply our method to ResNet, DenseNet, ShuffleNet, MobileNet, and Deeplab architectures, and show that significant parameter reductions, without noticeably compromising the accuracy, are possible. In addition, we show that even when the parameters of the CSG take fixed but random binary values, the performances of the networks especially on CIFAR-10 dataset are still acceptable. Furthermore, when running on a single GPU, we observe that the training time and the inference time of the augmented networks remain almost unaltered (as detailed in the supplementary materials). In addition, we compare our method with several related compression methods in the category of low rank matrix factorization and transferred convolutional filters and show that our method achieves improved performance over all the studied CNN models.

The paper is organized as follows. In Section 2, we provide the preliminaries and set the stage for introducing our method. In Section 3, we formally introduce the CSG, provide a rough estimate on the cardinality of the code vector space, and theoretically investigate its effect on the convergence of the training phase. In Section 4, we provide the results of our experiments on ResNet, DenseNet, and Deeplab architectures and comparison with other related methods. Section 5 contains our concluding remarks and future directions. Furthermore, interested readers can find a more detailed argument regarding the compression ratio of the kernels, experiments and discussions regarding the end-to-end timing of our method as well as convergence plots, and more details of experiments in the Appendix provided in the supplementary materials. The supplementary materials also include our carefully documented implementations and our results datasets.

2 Preliminaries

2.1 Convolutional Neural Network (CNN)

In a typical classification task, a CNN is composed of several convolutional layers and one or more fully connected layers, at the very end of the network, responsible for the classification. Each convolutional layer consists of a set of filters and perhaps is followed by some batch normalization layers and activation layers. *Our goal is to reduce the number of these trainable parameters by providing a compact representation for the parameters of the sets of filters of the convolutional layers.*

Let $l \in \mathbb{R}^{n_f \times k \times h \times w}$, for $n_f, k, h, w \in \mathbb{N}$, denote a set of n_f filters in the CNN, where k is the number of input channels and h and w are the height and width of the kernel, respectively. Let denote the collection of all the sets of filters in a CNN, namely the main parameters of the convolutional layers, by $\mathcal{L}$ and the set of all the other parameters in the CNN by $\mathcal{O}$. Then, we represent the set of all the parameters by $\mathcal{P} := \mathcal{L} \cup \mathcal{O}$.

2.1.1 Slices

Instead of focusing on direct compression of filters, in this paper we focus on slices. We define a *slice* as a tensor $s \in \mathbb{R}^{\hat{n}_f \times \hat{k} \times \hat{h} \times \hat{w}}$, for $\hat{n}_f, \hat{k}, \hat{h}, \hat{w} \in \mathbb{N}$. We partition each set of filters $l \in \mathcal{L} \setminus \{l_0\}$, where l_0 denotes the set of filters of the first convolutional layer, into $\lceil n_f/\hat{n}_f \rceil \lceil k/\hat{k} \rceil \lceil h/\hat{h} \rceil \lceil w/\hat{w} \rceil$ slices starting from the first slice $l(0 : \hat{n_f}, 0 : \hat{k}, 0 : \hat{h}, 0 : \hat{w})$. In Fig. 1, one slice of a set of filters of a convolutional layer is shown in darker color. We denote the set of all such slices for all layers by $\mathcal{S}$. The ordering of these partitions is arbitrary and does not affect the final results. Without loss of generality we assume that this partitioning is possible. [2] To reduce the trainable parameters, we produce same size but approximate versions of these slices denoted by $\hat{s} \in \hat{\mathcal{S}}$ from a compact low dimensional space (codes) using the CSG as explained in detail in the next section.

2.1.2 Code Vectors

To approximate each slice of each set of filters $s \in \mathcal{S}$ by $\hat{s} \in \hat{\mathcal{S}}$ using the CSG, we use a code vector $c \in \mathbb{R}^{n_c}$, where $n_c \in \mathbb{N}$. The relationship between slices and their corresponding code vectors is detailed in the following section.

3 The Convolutional Slice Generator

The Convolutional Slice Generator (CSG) provides a linear approximation for the slices of a convolutional filter. This means that each slice of a set of convolutional filters is represented by a code vector that has around 20× fewer elements. Multiplying the CSG matrix by this code vector, followed by an appropriate reshaping, produces an approximation for this slice. Several slices are then concatenated to produce a regular but approximate version of the set of convolutional filters. The shared CSG matrix used by all layers provides the association among the convolutional layers. In the following sections we make these statements precise.

3.1 The CSG Network

To generate an approximate version of each slice s_i, for $i \in \{1, ..., |\mathcal{S}|\}$ denoted by $\hat{s_i}$, we have

$$\hat{s}_i = \mathit{Reshape}(A_{CSG} c_i), \quad \text{for} \quad i \in \{1, ..., |\hat{\mathcal{S}}|\}, \quad (1)$$

[2]In practice we consider additional slices for fractional partitions and only use part of the final slice(s) to reconstruct the set of convolutional filters.

where A_{CSG} denotes an $\hat{n}_f \hat{k} \hat{h} \hat{w}$ by n_c matrix representing the weights of the CSG network, c_i denotes the code vector corresponding to the i'th slice where $i \in \{1, 2, ..., |\hat{\mathcal{S}}|\}$, and the *Reshape*(.) operator reshapes the input vector to a tensor of dimensions $\hat{n}_f, \hat{k}, \hat{h}, \hat{w}$ in an arbitrary but consistent order. In our experiments we also use fixed binary values (-1,+1) for the weights of the CSG network. We refer to this network as Binary CSG (BCSG). The binary values in BCSG are set to the signs of the initial weights (with normal distribution) and fixed throughout the training. See Fig. 1 for an example of how a single slice of a set of filters for a convolutional layer is generated.

3.2 Training the CSG-Augmented Network

Let $\hat{\mathcal{G}}$ denote the parameters of the CSG, i.e., the elements of the matrix A_{CSG}, $\hat{\mathcal{C}}$ denote the set of all the code vectors, and let $\hat{\mathcal{O}} = \mathcal{O}$ denote all the other parameters of the CNN, e.g., biases, batch normalization parameters, fully connected layer(s), and the first convolutional layer filters. Hence, we can denote the set of all the parameters of the network by $\hat{\mathcal{P}} := \hat{\mathcal{C}} \cup \hat{\mathcal{G}} \cup \hat{\mathcal{O}}$. Let $\mathcal{D}$ denote the set of the input data. A general objective function to train the CNN in our approach can be written as

$$f(\mathcal{D}, \hat{\mathcal{P}}) = f(\mathcal{D}, \hat{\mathcal{C}}, \hat{\mathcal{G}}, \hat{\mathcal{O}}).$$

Hence, to train the CSG-augmented CNN, instead of taking the gradients with respect to the kernels' weights ($\mathcal{L}$), they are taken with respect to the set of code vectors and the CSG parameters ($\hat{\mathcal{C}}, \hat{\mathcal{G}}$).

3.3 Cardinality of the Code Vector Space

In this section, we discuss our method for providing a rough estimate on the cardinality of the code vector space n_c. First, we need to choose a shape for the slices. In order to decide about this shape, we considered several widely used CNNs including VGG16, VGG19, ResNet, etc. A 3×3 filter size is the most common size for the filters. Also, these architectures suggests that a slice with channel size of 16 and the depth of 16 would divide most of these filters. Hence, we chose $\hat{s}_1 = 16, \hat{s}_2 = 16, \hat{s}_3 = 3, \hat{s}_4 = 3$ for this part of our work. In order to determine the cardinality of the code vector space, we need an estimate of the number of the elements of the slice in its possible latent domain, namely an estimate for n_c. Inspired by the fact that these filters are responsible for detecting visual features and knowing that usage of DCT leads to a very good encoding of visual representations (Watson, 1994), we looked at the four-dimensional Type-II DCTs (4-D DCT-II) of about 29000 slices of pre-trained filters extracted from VGG-16, VGG-19, ResNet-50, InceptionV3, DenseNet-169, DenseNet-201, InceptionResNetV2 (available in Tensorflow). We then computed the 4-D DCT-II representation of these slices and removed the elements of this representation in such a way that the remaining elements would result in an inverse transform which is not very different from the original slice. Our analysis, presented in the Appendix in the supplementary materials, suggests that a code vector that has close to 20× fewer number of elements would be sufficient. In our experiments, we chose code vectors that have 18× fewer elements than the slices, and our experiments on the neural networks confirm this choice.

3.4 Training Convergence

While convergence is always observed in all our experiments, in this section, we provide a proof of convergence for a simple CSG-augmented CNN with only one convolutional layer based on the recent work (Allen-Zhu et al., 2019). Let m denote the number of channels of the input, and d denote the number of its features (e.g., pixels). For simplicity, let us assume that the number of channels remains m after the convolutional layer. Let n denote the number of data points, and d' denote the number of labels. We assume that the data-set is non-degenerate meaning that there does not exist similar inputs with dissimilar labels. We denote by δ the minimum distance between two training points. We restate the following theorem from (Allen-Zhu et al., 2019) for the CNN defined in Appendix B of this reference.

Theorem 1 (CNN (Allen-Zhu et al., 2019)) *As long as $m \geq \tilde{\Omega}(poly(n, d, \delta^{-1})d')$, with a probability that approaches one as $m \to \infty$, Stochastic Gradient Decent (SGD) finds an ϵ-error solution for l_2 regression in $T = \tilde{\Omega}\left(\frac{poly(n,d)}{\delta^2}\log\epsilon^{-1}\right)$ iterations for a CNN.*

The above theorem as discussed in (Allen-Zhu et al., 2019) can be easily extended for other convergence criteria including the cross-entropy. Now let us consider our CSG-augmented CNN which we denote by CNN-CSG. For simplicity, in the following theorem, we consider the case when only a single layer convolutional layer is present.

Theorem 2 (CNN-CSG) *If $|\hat{\mathcal{C}}| \geq \tilde{\Omega}(poly(n, d, \delta^{-1})d')$, with a probability that approaches one as $|\hat{\mathcal{C}}| \to \infty$, then SGD finds an ϵ-error solution for l_2 regression in $T = \tilde{\Omega}\left(\frac{poly(n,d)}{\delta^2}\log\epsilon^{-1}\right)$ iterations for a CNN-CSG.*

The proof of the above theorem, which follows from the fact that the code vectors following the CSG layer can simply be viewed as an additional fully connected layer, can be found in the Appendix in the supplementary materials. Similar to Theorem 1, Theorem 2 can be easily extended for other convergence criteria including the cross-entropy.

Table 1: Training results on CIFAR-10 dataset with similar hyperparameters. When CSG is used, the slice shape and the code vector size are indicated as CSG-$[\hat{n}_f, \hat{k}, \hat{h}, \hat{w}]$-$n_c$ (shape of slice followed by the size of the vectors) following the name of the original network. On the "Top-1 Err." column the average and standard deviations of test errors at the last epoch for three non-selective trainings and on the "Ratio" column the compression ratios with respect to the original networks are reported. For details see Appendix in the supplementary materials.

Network Architecture	# Param.	Top-1 Err.	Ratio
DenseNet-BC-40-48 (Original)	2,733,130	4.97 ± 0.26	1.00×
DenseNet-BC-40-48-CSG-[12,12,3,3]-72	1,416,394	4.83 ± 0.24	1.92×
DenseNet-BC-40-48-CSG-[12,12,3,3]-72 w/ Pre-trained CSG on DenseNet-BC-40-48	1,323,082	5.07 ± 0.11	2.06×
DenseNet-BC-40-48-CSG-[12,12,3,3]-72 w/ Pre-trained CSG on DenseNet-BC-40-36	1,323,082	5.14 ± 0.23	2.06×
DenseNet-BC-40-48-CSG-[12,12,3,3]-72 w/ Compressed 1x1 Kernels	904,906	5.62 ± 0.28	3.02×
DenseNet-BC-40-48-BCSG-[12,12,3,3]-72	1,323,082	5.06 ± 0.10	2.06×
DenseNet-BC-40-36 (Original)	1,542,682	5.38 ± 0.27	1.00×
DenseNet-BC-40-36-CSG-[12,12,3,3]-72	842,842	5.12 ± 0.09	1.83×
DenseNet-BC-40-36-CSG-[12,12,3,3]-72 w/ Pre-trained CSG on DenseNet-BC-40-48	749,530	5.61 ± 0.21	2.05×
ResNet-56 (Original)	853,018	6.28 ± 0.20	1.00×
ResNet-56-CSG-[16,16,3,3]-128	347,162	7.24 ± 0.11	2.45×
ResNet-56-CSG-[12,12,3,3]-72	160,450	8.01 ± 0.27	5.31×
ResNet-56-CSG-[16,16,3,3]-128 w/ Pre-trained CSG on ResNet-20	52,250	11.98 ± 0.28	16.3×
ResNet-56-BCSG-[16,16,3,3]-1024	381,978	8.11 ± 0.02	2.23×
ShuffleNet-(0.5×) (Original)	352,042	9.81 ± 0.23	1.00×
ShuffleNet-(0.5×)-CSG-[16,16,1,1]-16	171,818	10.15 ± 0.16	2.04×
ShuffleNet-(0.5×)-BCSG-[16,16,1,1]-16	167,722	10.15 ± 0.16	2.10×
MobileNetV2 (Original)	2,296,922	6.64 ± 0.18	1.00×
MobileNetV2-CSG-[16,16,1,1]-16	1,595,322	7.65 ± 0.18	1.44×
MobileNetV2-BCSG-[16,16,1,1]-16	1,591,226	7.72 ± 0.08	1.44×

4 Experiments

We evaluated our approach on five different CNN models (ResNet-56, DenseNet-BC-40-48, DenseNet-BC-40-36, ShuffleNet V2, MobileNet V2) on CIFAR-10 dataset, two CNN models (ResNet-50 and ResNet-101) on ImageNet-1000 dataset, and two Deeplab models on Pascal VOC dataset. The CSGs are integrated into the models implemented in PyTorch. Our implementations along with detailed documentations of our codes are available in the supplementary materials. For training the models on the CIFAR-10 dataset, we used a machine with a single GPU (Nvidia Geforce 2080 Ti) and for the training on the two other datasets we used four (Nvidia Geforce 1080 Ti) GPUs. Note that we did not do any parameter tuning for any of our CSG-augmented networks and the experiments are all done using the same settings

Table 2: Training results on ImageNet-1000 (ILSVRC2012) dataset. When CSG is used, the slice shape and the code vector size are indicated as CSG-[$\hat{s}_1, \hat{s}_2, \hat{s}_3, \hat{s}_4$]-$n_c$ following the name of the original network. In the "Top-1 Error" column the validation error for the center cropped images at the last epoch of the training and on the "Ratio" column the compression ratios with respect to the original networks are reported. The results indicated with a "*" are reported from (TorchVision, Accessed: 2020-01-30)

Network Architecture	# Param.	Top-1 Err. (%)	Ratio
ResNet-50 (Original)	25,557,032	23.9*%	1.00×
ResNet-50-CSG-[16,16,3,3]-128	15,163,432	24.9%	1.68×
ResNet-50-BCSG-[16,16,3,3]-128	14,868,520	26.5%	1.72×
ResNet-101 (Original)	44,654,504	22.6*%	1.00×
ResNet-101-CSG-[16,16,3,3]-128	24,685,608	23.1%	1.81×
ResNet-101-BCSG-[16,16,3,3]-128	24,390,696	24.2%	1.83×

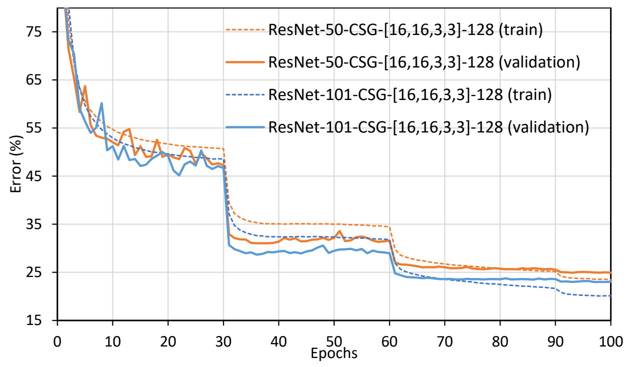

Figure 2: Train and validation errors of ResNet-50-CSG-[16,16,3,3]-128, and ResNet-101-CSG-[16,16,3,3]-128 on ImageNet dataset.

that were used for the original networks. Also, as it is clear from the previous sections, we did not apply our method to the very first convolutional layer.

4.1 CIFAR-10 Dataset

CIFAR-10 dataset includes 50K training images and 10K test images from 10 different classes. See Table 1 for a summary of all the results. As we can see, when we use $[16, 16, 3, 3]$ slices and code vectors of size 128 for ResNet-56 (He et al., 2016b), we achieve $\approx 2.5\times$ reduction with less than 1% increase in top-1 error. If we allow a higher accuracy degradation of $\approx 1.5\%$, we can achieve over $5.3\times$ parameter reduction by using $[12, 12, 3, 3]$ slices and code vectors of size 72. In case of DenseNet (Huang et al., 2017), we considered the most challenging cases, namely, when bottlenecks are used and the network has a 50% compression factor (i.e., $\theta = 0.5$), which is abbreviated as DenseNet-BC. In the first case we only consider 3×3 kernels and do not compress the bottleneck or transition layers in these implementations. Since the number of filters is a multiple of 12, we choose slices of shape $[12, 12, 3, 3]$ and code size of 72 to keep the ratio between the number of elements in the slice and code vector size n_c the same. We consider two cases when $L = 40, K = 48$, and $L = 40, K = 36$, where L is the number of layers and K is the growth rate. For the first case, we could achieve $\approx 2\times$ reduction with a slight improvement in accuracy. For the second case, the use of CSG had little effect on the accuracy of the network while reducing its parameters by over $1.8\times$. The results for compressing 1 kernels and when we use BCSG are also found in Table 1. We also apply CSG on a narrow model, ResNet-56, where $\approx 1\%$ increase in error is seen with $2.4\times$ compression ratio. In addition, binarizing CSG in this model achieves significant reduction in the number of trainable parameters, i.e., $16.3\times$, with only 2% accuracy drop. To show that the choices of the slice shape and the cardinality of the code vectors based on our approach in Section 3.3 are appropriate we have done additional experiments with various slice shapes. The results, that confirm our choices, can be found in the supplementary materials of this paper. For ShuffleNet V2 (CIFAR version) we have compressed the last convolutional layer of the networks which is again a 1x1 kernel and constitutes a large portion of the network weights resulting in $\approx 2\times$ compression while the accuracy loss is within 0.5%. For the case of MobileNetV2 (CIFAR version) we have compressed the first 1x1 kernel in each block (namely kernels w/ largest number of parameters) resulting in $1.44\times$ reduction with an accuracy loss of 1%.

4.2 ImageNet-1000 (ILSVRC2012) Dataset

We trained the CSG-augmented versions of ResNet-50 and ResNet-101 on the ImageNet-1000 (ILSVRC2012) dataset which consists of ≈1.3 million images for training and 50K images for validation. We used the same hyperparameters as the ones mentioned in the original paper (He et al., 2016a), namely we used batch sizes of 256 images, and started from the learning rate of 0.1 and divided the learning rate by 10 every 30 epochs.

We continued the training for 100 epochs (which is 20 epochs less than the original paper). While ResNet-50-CSG-[16,16,3,3]-128 has a compression ratio of 1.68×, it achieves a top-1 error of 24.9% which is within 1% of the error of the original ResNet-50 implemented in PyTorch and reported by TorchVision (TorchVision, Accessed: 2020-01-30). ResNet-101-CSG-[16,16,3,3]-128 achieves 23.1% top-1 error with a compression ratio of 1.81×. While this model now has around one million parameters less than original ResNet-50, its error-rate is still $\approx 0.8\%$ smaller. The above results as well as results for BCSG-augmented networks are summarized in Fig. 2. More details of training and validation errors over the course of 100 epochs are shown in Table 2 and Fig. 2 and in the supplementary materials.

4.3 Training CSG/Pre-trained CSG/BCSG

In this set of experiments we train all the models from scratch. We initialize the parameters of CSG $\hat{\mathcal{G}}$ with random initial values and train it alongside the code vectors $\hat{\mathcal{C}}$ as well as other parameters of the network $\hat{\mathcal{O}}$. We refer the reader to the supplamentary material for a detail account of the datasets and implementation as well as convergence figures. When using pre-trained CSG parameters during the training of the CSG-augmented CNNs, the number of parameters to be trained reduces to $|\hat{\mathcal{C}}| + |\hat{\mathcal{O}}|$. Similarly for BCSG networks, the parameters are chosen randomly from (-1,+1) and are fixed throughout the training. These can result in significant reduction in the number of the parameters of the network depending on its architecture. The summary of our results are provided in Table 1 and 2. More details can be found in the Appendix in the supplementary materials.

4.4 Semantic Segmentation Tasks

To explore the possibility of applying our approach to tasks other than image classification, in these experiments we applied it to Deeplab V3 (Chen et al., 2017b) for semantic segmentation on the Pascal VOC 2012 (Everingham et al., 2010) dataset *without any hyper-parameter tuning*. We considered the Deeplab V3 architecture alongside Resnet-50 and Resnet-101 for feature extraction. For each case, two scenarios were studied against two baseline scenarios where no modification to the models were done. The baseline scenarios in our implementation achieve mean Intersection Over Unions (mIOUs) of 73.38% and 74.73% respectively.

In the first scenario, we used Resnet-50-CSG-[16,16,3,3]-128 and Resnet-101-CSG-[16,16,3,3]-128 for the feature extraction with pretrained parameters and did not modify the rest of the architecture. In this settings our approach achieves mIoUs of 71.41% and 72.98% with 1.35× and 1.5× compression ratios. In the second scenario, in addition to using CSG-augmented versions of Resnet-50 and Resnet-101 for the feature extraction with pretrained parameters, we also considered a second CSG for the Atrous Spatial Pyramid Pooling (ASPP) modules. In these settings, we achieve mIoUs of 70.28% and 71.63% respectively with 2.39× and 2.24× compression ratios. The results are summarized in Table 3. Details of the implementation can be found in the supplementary materials.

4.5 Comparison with Related Methods

In this section, we provide a detailed comparison of our proposed technique with the relevant methods, as discussed in more detail in Section 1.1: Kernel decomposition (separable filters) (Jaderberg et al., 2014) and two transferred convolutional filters methods, CRELU (Shang et al., 2016), G-CNN (Cohen and Welling, 2016), and Hypernetworks (Ha et al., 2017). We applied these techniques on both wide and narrow CNN models DenseNet-BC-40-48 and ResNet-56 for CIFAR-10 dataset. Each modified model is trained three times with similar hyper parameters as in Section 4 and the results are summarized in Table 4. As shown in the table, for wide CNN models such as DenseNet-BC-40-48, the compression ratio of all methods is similar ($\approx 2\times$). While the error of the model modified with other methods is increased, the CSG-augmented model improves the error slightly. For ResNet-56 which is a narrow model, when half of the convolutional layers are replaced by separable filters or when we compress the model using the CRELU we achieve $\approx 2\times$ reduction. In both cases compared to our CSG-augmented model with even a slightly higher compression ratio, the top-1 errors are higher. The accuracy of G-CNN based model achieving compression ratio of 3.92× is still lower than the one in CSG-augmented one with 5.31× ratio. We also applied separable filters to all the layers of ResNet-56 that achieves significant parameter reduction $\approx 7\times$ at the cost of high accuracy degradation. It should be noted that BCSG augmented ResNet-56 compared to low rank decomposition methods and transferred convolutions achieves the highest compression ratio and the lowest error. Finally, it is observed that the performance of ResNet56 augmented by a Hypernetwork has the worst performance on ResNet-56 which is a narrow network. In summary, the results show that for narrow models, the other techniques degrade accuracy which is compatible with their mentioned drawbacks in (Cheng et al., 2018), while our proposed technique does not degrade the accuracy significantly in both narrow and wide CNNs.

Table 3: Results of semantic segmentation on Pascal VOC 2012 validation dataset after training for 50 epochs with batch size of 4. With CSG, the slice shape and the code vector size are indicated as CSG-$[\hat{n}_f, \hat{k}, \hat{h}, \hat{w}]$-$n_c$ following the name of the original network.

Network Architecture	# Param.	mIoU (%)	Ratio
DeeplabV3-ResNet-50 (Original)	40,352,181	73.38%	1.00×
DeeplabV3-ResNet-50-CSG-[16,16,3,3]-128	29,958,581	71.41%	1.35×
DeeplabV3-CSG-[16,16,3,3]-128 -ResNet-50-CSG-[16,16,3,3]-128	16,884,149	70.28%	2.39×
DeeplabV3-ResNet-101 (Original)	59,344,309	74.73%	1.00×
DeeplabV3-ResNet-101-CSG-[16,16,3,3]-128	39,480,757	72.98%	1.50×
DeeplabV3-CSG-[16,16,3,3]-128 -ResNet-101-CSG-[16,16,3,3]-128	26,406,325	71.63%	2.24×

Table 4: Comparison of various compression techniques with our method on CIFAR-10 dataset. The "Top-1 Err." column shows the test errors at the last epoch ± standard deviation in three runs and the "Ratio" column shows the compression ratios with respect to the original networks. Ratios decorated with a "*" denote bit-wise ratios.

Compression Method	Network Architecture	# Param.	Top-1 Err.	Ratio
Original Architecture	DenseNet-BC-40-48	2,733,130	4.97 ± 0.26	1.00×
CSG/BCSG	DenseNet-BC-40-48-CSG-[12,12,3,3]-72	1,416,394	4.83 ± 0.24	1.92×
	DenseNet-BC-40-48-BCSG-[12,12,3,3]-72	1,323,082	5.06 ± 0.10	2.06×
Low Rank Decomposition	DenseNet-BC-Separable Filters	1,441,450	5.13 ± 0.05	1.90×
Transferred Convolutions	DenseNet-BC-CReLU	1,369,210	5.39 ± 0.28	2.00×
	DenseNet-BC-GCNN (p4)	1,613,602	5.25 ± 0.29	1.60×
Quantization	DenseNet-BC-40-48-FP16	2,733,130	5.15 ± 0.05	2.00*×
CSG+Quantization	DenseNet-40-48-CSG-[12,12,3,3]-72-FP16	1,416,394	5.43 ± 0.08	3.84*×
Hypernetworks	DenseNet-BC-Hypernetworks	1,435,530	5.31 ± 0.15	1.90×
Original Architecture	ResNet-56	853,018	6.28 ± 0.20	1.00×
CSG/BCSG	ResNet-56-CSG-[16,16,3,3]-128	347,162	7.24 ± 0.11	2.45×
	ResNet-56-CSG-[12,12,3,3]-72	160,450	8.01 ± 0.27	5.31×
	ResNet-56-BCSG-[16,16,3,3]-1024	381,978	8.11 ± 0.02	2.23×
Low Rank Decomposition	ResNet-56-Separable Filters (1/2 Layers)	494,709	7.56 ± 0.45	1.72×
	ResNet-56-Separable Filters (All Layers)	117,066	8.41 ± 0.05	7.29×
Transferred Convolutions	ResNet-56-CReLU	427,066	8.27 ± 0.10	2.00×
	ResNet-56-GCNN (p4)	217,618	8.79 ± 0.31	3.92×
Quantization	ResNet-56-FP16	853,018	6.64 ± 0.35	2.00*×
CSG+Quantization	ResNet-56-CSG-[16,16,3,3]-128-FP16	347,162	7.54 ± 0.01	4.90*×
Hypernetworks	ResNet-56-Hypernetworks	182,618	9.19 ± 0.06	4.67×

5 Conclusion and Future Directions

We presented a novel and easy to implement method to reduce the number of unnecessary parameters of convolutional layers during both training and inference by representing them in a low dimensional space through the use of a simple auxiliary neural network without significantly compromising the accuracy or tangibly adding to the processing burden. There are still several directions that can be pursued in future. The use of this method for other tasks, especially other than vision related tasks, such as natural language processing, etc. needs to be assessed. The extension of the theoretical analysis to more complicated architectures is an attractive future direction. The combination of this method with efficient computation and compression methods mentioned in this paper for distributed machine learning and machine learning acceleration for edge devices need to be explored further. Additionally, the use of more than one CSG for different classes of filters or the use of non-linear and/or multi-layer CSGs remains to be investigated. Finally, as mentioned throughout the paper, we did not do any parameter tuning after applying our method. Hence, there is the possibility of improving the results when the parameters are further tuned after applying our method.

References

Tahmid Abtahi, Colin Shea, Amey Kulkarni, and Tinoosh Mohsenin. Accelerating convolutional neural network with FFT on embedded hardware.

IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 26(9):1737–1749, 2018.

Zeyuan Allen-Zhu, Yuanzhi Li, and Zhao Song. A convergence theory for deep learning via over-parameterization. *International Conference on Machine Learning*, 2019.

Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured pruning of deep convolutional neural networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 13(3):32, 2017.

Joan Bruna and Stéphane Mallat. Invariant scattering convolution networks. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1872–1886, 2013.

Guobin Chen, Wongun Choi, Xiang Yu, Tony Han, and Manmohan Chandraker. Learning efficient object detection models with knowledge distillation. In *Advances in Neural Information Processing Systems*, pages 742–751, 2017a.

Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE transactions on pattern analysis and machine intelligence*, 40(4):834–848, 2017b.

Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. Model compression and acceleration for deep neural networks: The principles, progress, and challenges. *IEEE Signal Processing Magazine*, 35(1):126–136, 2018.

Taco Cohen and Max Welling. Group equivariant convolutional networks. In *International conference on machine learning*, pages 2990–2999, 2016.

Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, et al. C ir cnn: accelerating and compressing deep neural networks using block-circulant weight matrices. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 395–408. ACM, 2017.

Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338, 2010.

Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=rJl-b3RcF7`.

William T. Freeman and Edward H Adelson. The design and use of steerable filters. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 9:891–906, 1991.

Golnaz Ghiasi, Tsung-Yi Lin, and Quoc V Le. Nas-fpn: Learning scalable feature pyramid architecture for object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7036–7045, 2019.

David Ha, Andrew M. Dai, and Quoc V. Le. Hypernetworks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL `https://openreview.net/forum?id=rkpACe1lx`.

Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016a.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer, 2016b.

Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

Jorn-Henrik Jacobsen, Jan van Gemert, Zhongyu Lou, and Arnold WM Smeulders. Structured receptive fields in CNNs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2610–2619, 2016.

Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. In *Proceedings of the British Machine Vision Conference. BMVA Press*, 2014.

Jakub Konečný, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*, 2016.

Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K Bansal, William Constable, Oguz Elibol, Scott Gray, Stewart Hall, Luke Hornof, Amir Khosrowshahi, Carey Kloss, Ruby J Pai, and Naveen Rao. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 1742–1752. Curran Associates, Inc., 2017.

Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4013–4021, 2016.

Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan V. Oseledets, and Victor S. Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *CoRR*, abs/1412.6553, 2015.

Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 583–598, 2014.

Yujun Lin, Song Han, Huizi Mao, Yu Wang, and Bill Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=SkhQHMW0W`.

Stéphane Mallat. Group invariant scattering. *Communications on Pure and Applied Mathematics*, 65 (10):1331–1398, 2012.

Franck Mamalet and Christophe Garcia. Simplifying convnets for fast learning. In *International Conference on Artificial Neural Networks*, pages 58–65. Springer, 2012.

Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *CoRR*, abs/1603.05279, 2016. URL `http://arxiv.org/abs/1603.05279`.

Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

Roberto Rigamonti, Amos Sironi, Vincent Lepetit, and Pascal Fua. Learning separable filters. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2754–2761, 2013.

Wenling Shang, Kihyuk Sohn, Diogo Almeida, and Honglak Lee. Understanding and improving convolutional neural networks via concatenated rectified linear units. In *international conference on machine learning*, pages 2217–2225, 2016.

Laurent Sifre and Stéphane Mallat. Rotation, scaling and deformation invariant scattering for texture discrimination. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1233–1240, 2013.

Cheng Tai, Tong Xiao, Yi Zhang, Xiaogang Wang, et al. Convolutional neural networks with low-rank regularization. *arXiv preprint arXiv:1511.06067*, 2015.

TorchVision. tourchvision.models – PyTorch Master Documentation. `https://pytorch.org/docs/stable/torchvision/models.html`, Accessed: 2020-01-30.

Hongyi Wang, Scott Sievert, Shengchao Liu, Zachary Charles, Dimitris Papailiopoulos, and Stephen Wright. Atomo: Communication-efficient learning via atomic sparsification. In *Advances in Neural Information Processing Systems*, pages 9850–9861, 2018.

Min Wang, Baoyuan Liu, and Hassan Foroosh. Factorized convolutional neural networks. In *Proceedings of the IEEE International Conference on Computer Vision Workshops*, pages 545–553, 2017.

Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. Gradient sparsification for communication-efficient distributed optimization. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 1299–1309. Curran Associates, Inc., 2018.

Andrew B Watson. Image compression using the discrete cosine transform. *Mathematica journal*, 4(1): 81, 1994.

Shmuel Winograd. *Arithmetic complexity of computations*, volume 33. Siam, 1980.

Min Ye and Emmanuel Abbe. Communication-computation efficient gradient coding. In Jennifer Dy and Andreas Krause, editors, *Proceed-*

ings of the 35th International Conference on Machine Learning, volume 80 of *Proceedings of Machine Learning Research*, pages 5610–5619, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR. URL `http://proceedings.mlr.press/v80/ye18a.html`.

Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas Huang. Slimmable neural networks. In *7th International Conference on Learning Representations, ICLR 2019*, 2019.

Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6848–6856, 2018.