Deep Neural Networks Model Compression and Acceleration: A Survey

Guillaume Lagrange*
École de technologie supérieure
lagrange.guillaume.1@gmail.com

Marco Pedersoli
École de technologie supérieure
marco.pedersoli@etsmtl.ca

***Abstract*— Deep Neural Networks (DNNs) have achieved state-of-the-art results across various applications in the recent years, with some of the best results obtained with deeper networks and large training sets. On the other hand, the increasing size of such models restricts their deployability for consumer applications on resource-limited devices such as mobile and portable devices. As a result, there has been a lot of interest in different methods to perform model compression and acceleration, and tremendous progress has been made in this area in the past few years. In this paper, we provide a comprehensive survey of recent advanced techniques for deep convolutional neural network (CNN) compression and acceleration. Specifically, we provide insightful analysis of the techniques categorized as the following: network quantization, network pruning, low-rank approximation, knowledge distillation and compact network design. Then, we select some of the more successful methods and study them more in details, providing meaningful evaluation of the model and its performance. Finally, we discuss the topic's challenges and possible applications of the surveyed methods.**

I. INTRODUCTION

In recent years, deep neural networks (DNNs) have achieved remarkable performance across a wide range of applications, including but not limited to computer vision, speech recognition, natural language processing and machine translation. The 2010s saw dramatic progress in image processing, with the work of *Krizhevsky et al.* [3] achieving breakthrough results in the 2012 ImageNet Challenge using a network containing 60 million parameters with five convolutional layers and three fully-connected layers, along with the newly-introduced (at the time) *dropout* [15] method to reduce overfitting. This is considered by most the beginning of the deep learning revolution, which has led to an increased interest in the field. Since then, the performance of DNNs has continued to improve. The general trend has been to make deeper and more complicated networks in order to achieve higher accuracy, with recent breakthroughs being closely connected to the increased amount of training data and more powerful computing resources now made available, most notably the very fast and power-hungry Graphic Processing Units (GPUs).

However, these advances to improve the network's performance for a given task are not necessarily making the networks more efficient with respect to size and speed. In many real world applications, such recognition tasks need to be carried out on resource-limited platforms such as mobile and portable devices. In addition, recent years have seen significant progress in virtual reality (VR), augmented reality (AR) and smart wearable devices such as smart watches, reinforcing interest in deep learning systems deployment to portable devices with limited resources (e.g. CPU, memory, energy). As a result, there has been a growing interest in model compression and acceleration from the deep learning community, with significant progress being achieved in the past few years.

In this paper, we present a comprehensive survey of recent approaches in deep neural networks model compression and acceleration. We classify these approaches into five categories: network quantization, network pruning, low-rank approximation, knowledge distillation and compact network design. In general, the computational complexity of deep neural networks is dominated by the convolutional layers, while the number of parameters is mainly observed in the fully connected layers. Therefore, most network acceleration methods focus on decreasing computational complexity of the convolutional layers, while the compression methods usually center their attention on the fully connected layers or feature maps of the convolutional layers. After our review of the different techniques, we select some of the more popular and successful methods to study them more carefully with our own implementation, providing meaningful evaluation of the model and its performance.

II. NETWORK QUANTIZATION

Quantization is the process of constraining an input to a discrete set of values that closely approximates the original data. In recent years, network quantization has been used as a term to cover a lot of different techniques to store parameters and perform calculations on them in more compact formats. In general, this method compresses the original network by reducing the number of bits required to represent each weight and/or activation.

There are generally two approaches to quantization: convert a pre-trained floating point deep neural network model into a quantized model without training, and train a quantized network model. During the forward pass, both at run-time and train-time (if train-time applies), the networks drastically reduce memory size and accesses, and in the case of binary networks replace most arithmetic operations with bit-wise operations, drastically reducing power consumption.

Before getting into the different quantization methods, let us start by explaining the difference between floating

*The first author.

point and fixed point representation, which constitute the heart of these methods. Traditionally, numbers stored and calculations performed in neural networks are done in 32-bit floating point format to preserve high precision, but this leads to models of larger size which generally require more computational resources (depending on the hardware used of course). In a floating point representation, a number's radix point (*decimal point*) can *float* relative to the significant digits of the number. This position is indicated by the exponent component as shown in Fig. 1 and its size will vary. The integer portion of the number is stored in the *mantissa*. On the other hand, the fixed point format consists in a signed mantissa and a scaling factor shared between all fixed point variables, namely *shared exponent* in Fig. 1. The scaling factor can be seen as the position of the radix point. Essentially, this data type is an integer that is scaled by an implicit specific factor. Unlike floating point data types, this factor is the same for all values of the same type. This allows for smaller memory usage, and in many cases faster computation.

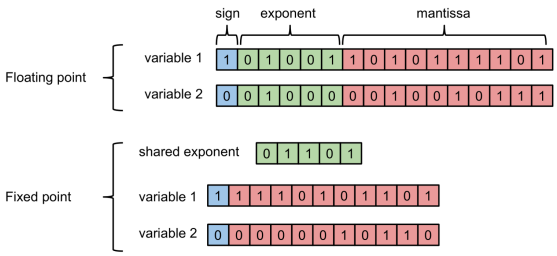


Fig. 1. A comparison of the floating point and fixed point format[8].

The fixed point implementation in [7] proposed an approach to convert a pre-trained floating point deep convolutional network model to its fixed point equivalent. Their bit-widths optimization strategy based on signal-to-quantization-noise-ratio (SQNR) resulted in $> 20\%$ reduction in model size for a pre-trained network on CIFAR-10 benchmark while maintaining the same accuracy. Meanwhile, the authors in [8] reduced the precision of the parameters and operations during training with three distinct formats: half precision floating point, fixed point and dynamic fixed point. They managed to achieve near state-of-the-art test error, proving that very low precision is sufficient not just for running trained networks but also for training them. The work by the authors in [10] also arrive to a similar conclusion: by using stochastic rounding, they found that using a 16-bit wide fixed-point number representation for a network's parameters (and intermediate variables during back-propagation) is sufficient to train a deep neural network and obtained nearly identical results as 32-bit floating-point computations.

The proposed method in [5], referred to as Quantized Neural Networks (QNNs), quantizes the weights and activations during inference and training. The authors showed that QNNs with weights and activations quantized to more than 1-bit (e.g. 4-bit and 6-bit) are capable of achieving comparable results to the 32-bit floating point architectures on benchmark datasets. They also demonstrated that Binarized Neural Networks (BNNs), the extreme case of quantization, can handle the same datasets while achieving nearly state-of-the-art prediction accuracy. Extending on the subject at hand in [5], recent work [11] proposed WAGE quantization: weight (W) and activation (A) in inference, gradient (G) and error (E) in backpropagation training. Constraining the parameters and computations to low-bitwidth integers, WAGE achieved state-of-the-art accuracy on multiple benchmark datasets.

The BinaryConnect method proposed in [9] constrains all weights to be either $+1$ or -1 during the forward and backward propagations, which allows for most of the multiply-accumulate operations (MAC) to be replaced by simple additions or subtractions. This potentially allows to speed-up by a factor of 3 during training, but also reduces the memory requirement of deep networks while maintaining near state-of-the-art results in classification.

Given only weight quantization, there is still a need for the necessary time-consuming floating-point operations. If the activations were also quantized into fixed-point values, the network can be efficiently executed by only fixed-point operations. In [4], the authors introduce BNNs with both weights and activations quantized into either $+1$ or -1 at run-time and train-time, achieving nearly state-of-the-art results on the MNIST, CIFAR-10 and SVHN datasets. Note that the binary activations are especially important for ConvNets, where there are typically many more neurons than free weights. To extend on BNNs for classification on large scale datasets, the authors in [6] further analyze different overlooked strategies to improve accuracy and compression rate. Compared to [4], the proposed method in [6] results in a model compressed by a factor of 3 (or a compression rate of 31.2 as opposed to 10.3) that outperforms previous state-of-the-art methods.

III. Network Pruning

Pruning neural networks isn't anything new, it's actually an idea that was proposed way before deep learning became popular. Early work such as [1] showed that network pruning is effective in reducing network complexity by removing unimportant weights, and results in a network with better generalization and improved learning speed that could require a smaller training dataset to achieve similar performance. Based on the idea that many parameters in DNNs are unimportant or redundant, a plethora of pruning techniques have been studied to compress DNN models, trying to remove parameters which don't contribute a lot to the performance of the model. Furthermore, we observe two popular methods of pruning: fine-grained pruning and filter-level pruning.

A. Fine-grained Pruning

Fined-grained pruning methods remove any unimportant parameters in the network (e.g. weights, connections, etc.) in an unstructured way. As previously mentioned, this type of method has been used both to reduce network complexity

and address the over-fitting issue by improving network generalization. Looking at the history of neural network pruning methods, we notice that the first techniques introduced in early work such as the Optimal Brain Damage [1] and related papers have been mainly focused on removing unimportant weights, connections or neurons from a network. The method in OBD [1] reduced the number of connections based on the Hessian of the loss function. An interesting overview of early pruning algorithms can be found in [2]. These methods usually required additional computation to determine the parameters the prune, such as the saliency for each weight computed using a diagonal Hessian approximation in [1]. Nowadays, model pruning methods are still very popular as a means for model compression, but recent techniques usually focus on computationally efficient solutions. For example, magnitude-based weight pruning methods have become popular techniques for network pruning: they're usually computationally efficient, scaling to large networks and datasets.

In recent years, methods such as [13] have managed to prune state-of-the-art CNN models, reducing the number of connections by $9\times$ to $13\times$ with no loss in accuracy. The initial dense training phase learns the connection weights and importance via normal network training, as illustrated in Fig. 2. Then, the sparse training prunes low-weight connections, effectively converting the dense network into a sparse network, and trains the sparse network. Finally, the sparse network is retrained so the remaining connections can compensate for the connections that were previously removed.

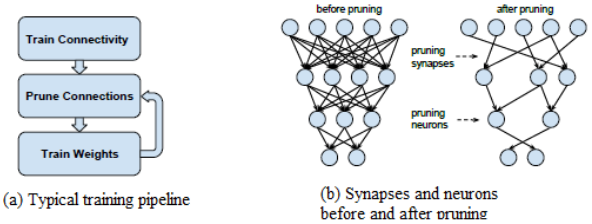


Fig. 2. An illustration of the typical training pipeline for pruning weights (a) accompanied by a comparison of the pruned network (b)[13].

In very similar fashion, the authors of [12] proposed a Dense-Sparse-Dense training flow, as shown in Fig. 3, replacing the final sparse retraining by a dense layer training to the previously proposed method by [13]. Technically, by recovering the pruned connections in the final dense training, the model capacity of the network is increased and thus the final network model isn't really compressed. Nonetheless, this method is worth mentioning since it achieves superior optimization performance without affecting model size. DSD methodology effectively improves Top1 accuracy of popular networks on benchmark datasets.

Other alternatives to reduce network complexity have been suggested. [14] proposed a HashedNets model that used a low-cost hash function to group weights into hash buckets for parameter sharing. That way, all connections within the same hash bucket share a single parameter value. The method exploits parameter redundancy in neural networks to achieve significant reductions in model sizes. The deep compression method in [16] reduced the storage required by $35\times$ to $40\times$ by pruning the unimportant connections just like [13], and further quantized the network using weight sharing followed by encoding the quantized weights with Huffman encoding to save additional space. Their weight sharing method used a *k*-mean clustering algorithm to find good clusters for the weights, adopting the centroids as quantization points for a cluster. All things considered, their deep compression method allowed for a significant model size reduction, as well as $3\times$ to $4\times$ layerwise speedup and $3\times$ to $7\times$ better energy efficiency.

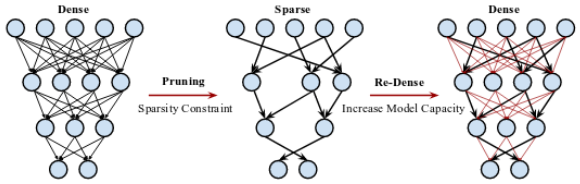


Fig. 3. The Dense-Sparse-Dense training flow[12].

However, these unstructured methods of pruning tend to introduce irregular sparsity in the network, limiting the support of these networks by off-the-shelf libraries. Therefore, specialized hardware and software are often needed for efficient inference, once again limiting their use in real-world applications.

B. Filter-level Pruning

In contrast to fine-grained pruning, filter-level pruning methods prune complete convolutional filters or channels in order to make the networks thinner, where no difference in network structure is observed. This allows for support by any off-the-shelf deep learning libraries (e.g. PyTorch [35], Tensorflow [36], etc.). By removing whole filters in the network along with their connecting feature maps, filter-level pruning has become a popular and efficient method to reduce computation costs, thus accelerating the network. A general overview of this process is illustrated in Fig. 4.

In [22], the authors presented a filter-level pruning method named ThiNet. They used the outputs of the next layer (i.e., its feature map) in order to guide the pruning in the current layer. By minimizing the reconstruction error of the next layer's feature map, the important channels are selected in a greedy manner. Moreover, their method achieved $3.31\times$ floating point operations per second (FLOPs) reduction and $16.63\times$ compression on VGG-16 network with little drop in accuracy.

Another approach to filter-level pruning was proposed by [21], where they pruned filters with relatively low kernel weight magnitudes. Previous work pruned on a layer by layer basis followed by an iterative fine-tuned retraining to compensate for any loss of accuracy, where as the method proposed method here used a *one-shot* pruning (across mul-

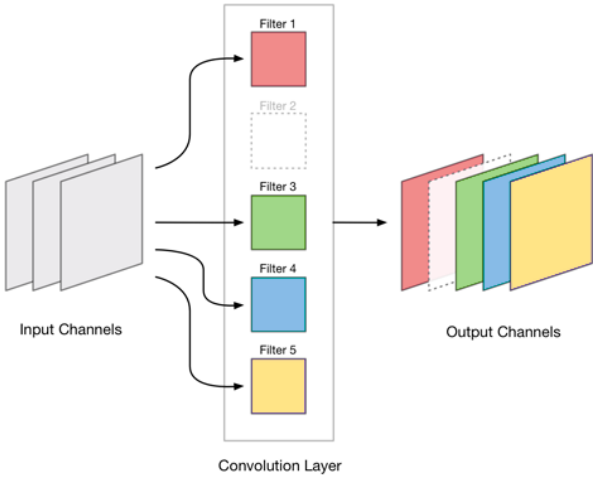


Fig. 4. Pruning a filter results in removal of its corresponding feature map and related kernels in the next layer.

tiple layers) and retraining strategy, achieving about 30% reduction in FLOPs for VGGNet on CIFAR-10.

The authors in [23] evaluated different filter-level pruning methods based on different criteria (e.g. kernel weight magnitude, activation, mutual information, Taylor expansion) by comparing them to their *oracle* criterion, which brute force pruned each filter and observed its impact on the cost function. Thus, they stated the pruning problem as combinatorial optimization problem: choose a subset of parameters, such that when pruned the network cost change is minimal. Their newly proposed method, based on the Taylor expansion, directly approximated change in the loss function from removing a particular parameter and demonstrated superior performance. Their proposed method lead to more than $10\times$ reduction with only a small drop in accuracy.

IV. Low-rank Approximation

Since convolution layers generally consume the bulk of the processing time in deep CNNs, many methods have been focused around reducing the complexity of convolution operations in order to compress the network and speed up convolution layers. Formally, a convolutional kernel in a CNN is a 4D tensor $\mathbf{W} \in \mathbb{R}^{N \times d \times d \times C}$, where N and C are the numbers of the output and input feature maps respectively and d is the spatial kernel size. The motivation behind tensor decomposition is to find a tensor approximation of $\mathbf{W}$ by removing the redundancy in the convolution kernels.

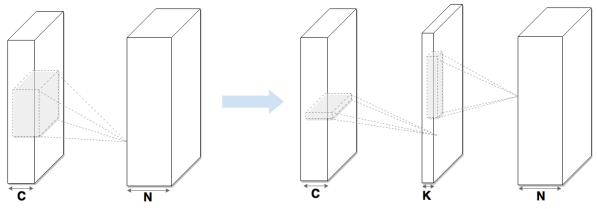


Fig. 5. A typical parametrization of the low-rank regularization method. Left: the original convolutional layer. Right: low-rank constraint convolutional layer with rank-K[17].

Using low-rank filters to accelerate convolution has a long history. Many low-rank based methods have been proposed over the years, most notably [18]. In their paper, the authors introduced a method for accelerating the convolution computation based on the existence of significant redundancy between different filters and feature channels. The proposed tensor decomposition scheme is based on a conceptually simple idea: replace the 4D convolutional kernel with two consecutive kernels with a lower rank, in other words a two-component decomposition. In this case, the 4D kernel tensor is presented as a composition (product) of two 3D tensors. The resulting approximations require significantly less operations to compute, which translates to a $4.5\times$ speedup with a drop of only 1% in accuracy. The network is approximated layer by layer: after one layer is approximated by the low-rank filters, the parameters of that layer are fixed, and the layers above are fine-tuned based on a reconstruction error criterion. This typical low-rank method for compressing 2D convolutional layer is described in Fig. 5.

Following this direction, the authors in [20] proposed a low-rank Canonical Polyadic (CP) decomposition using non-linear least squares (NLS) combined with a discriminative fine-tuning of the entire network simultaneously. The CP-decomposition approximates the convolution as a composition of four convolutions with small kernels, as shown in Fig. 6. This four-component decomposition combined with a new algorithm to compute the CP-decomposition achieved considerable speedups with very minimal loss in accuracy.

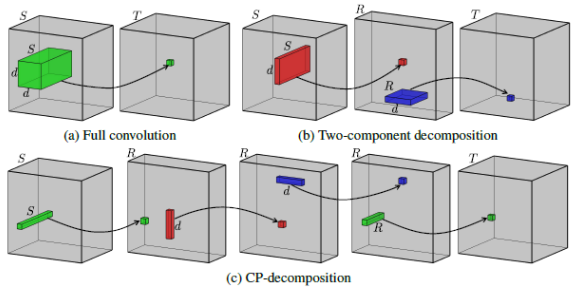


Fig. 6. Tensor decompositions for speeding up a convolution. Two-component decomposition (b) as proposed in [18] and CP-decomposition (c) used in [20].

Related to [18] and [20], the authors in [17] proposed a different algorithm for computing the low-rank tensor decomposition on much larger models. The proposed tensor decomposition is in line with the one presented by [18], but the algorithm finds the exact global optimizer of the decomposition and is more effective than iterative methods like [18]. The new method for training such constrained CNNs from scratch achieved significant speedup as well as better performance than their non-constrained counterparts in some cases.

Based on the fact that deep networks are trained with a large number of output targets, the authors in [19] focused their efforts solely on the final weight layer. Applying a low-rank matrix factorization to this final layer allowed for a reduction in the number of parameters of the network

between $30-50\%$, resulting in a similar $30-50\%$ speedup in training time with little loss in accuracy in large vocabulary continuous speech recognition (LVCSR) tasks.

V. Knowledge Distillation A.K.A Teacher-Student Networks

The idea of knowledge distillation (KD) is to make a small student network imitate the target of a large teacher network. In this sense, a student network is trained using a teacher network or the ensemble of neural networks (a collection of model whose predictions are combined by weighted averaging or voting), and the student network is a compact and efficient neural network. This idea was proposed by [27], in which they compressed the original network by training a new compact network with pseudo-data labeled from a larger and stronger network/ensemble. This resulted in mimic neural nets that were $1000\times$ smaller and faster.

Following this, [28] proposed a knowledge distillation method which trained a student network by the softened output of the teacher network, which also showed that soft targets are a good way to train a network with fewer samples while preventing overfitting. The student network was trained to predict the output of the teacher as well as the true classification labels, and demonstrated promising results.

The method presented by [30] is also based on the student-teacher framework, but it introduced the concept of using multiple teachers in order to improve regularization. To do so, they injected noise and perturbed the output of the teacher, simulating the effect of multiple teachers with a *noisy teacher*. This resulted in promising improvement in classification accuracy.

The authors of [29] extend the compression method to the training dataset by presenting data-free knowledge distillation. The teacher network was initially trained on the original dataset, after which the activations of each layer in the network were recorded in order to reconstruct the original dataset. The student network was then trained on the reconstructed dataset from the teacher model and its metadata. This allowed to compress deep networks trained on large-scale datasets to a fraction of their size, but also showed that the compact network can be trained without access to the original large-scale dataset.

In [31], the authors used a common initialization trick to further improve the distillation performance of classification, which can also boost the distillation on non-classification tasks. They extended the previous distillation framework by transferring the distilled knowledge from classification to face alignment and verification.

More recently, [32] proposed two new compression methods which jointly leveraged weight quantization and knowledge distillation of larger networks. The first method, *quantized distillation*, leveraged *distillation loss*[28] during the training process of a student network whose weights were quantized. The second method, *differentiable quantization*, provided a way of optimizing the locations of quantization during the learning process of the student in order to best fit the behavior of the teacher model. Both methods showed promising results in terms of compression while preserving accuracy close to state-of-the-art full-precision models.

VI. Compact Network Design

Another approach to network acceleration and compression is to design a more efficient but low-cost network architecture itself. Among the commonly used strategies to reduce network complexity, branching (multiple grouped convolutions) is probably the most popular. Initially introduced in AlexNet [3] as a way to share the convolution computation across two GPUs, grouped convolutions work as follow: the input and kernel are split by their channels, i.e. channel-wise, to form distinct groups, and each group performs convolutions independent of the other groups to give different outputs. These individual outputs are then concatenated together to give the final output. This process is illustrated in Fig. 7.

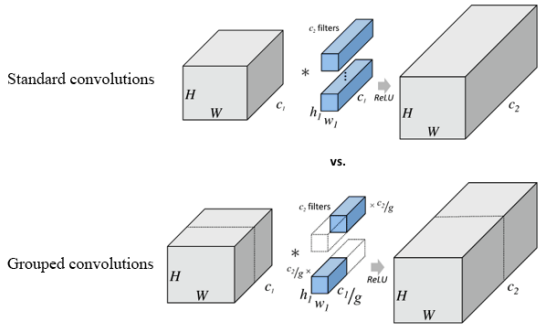


Fig. 7. Standard convolution vs Grouped convolution.

A special case of grouped convolutions is when the number of groups equals the number of input channels, also called depthwise convolutions, which form a part of separable convolutions. Separable convolutions, referred to as depthwise separable convolutions in most cases, consist of two consecutive convolution operations: (1) depthwise convolutions which performs convolutions separately for each channel of the input, followed by (2) a point-wise convolution (convolution with 1×1 filter). ResNeXt [25] was presented in a similar fashion and referred to the number of groups or branches as the *cardinality* of the network. Their model allowed them to place second in the 2016 ILSVRC competition. In MobileNet [24], depthwise separable convolutions allowed to create very small models that retained much of the capabilities of far larger architectures by making more efficient use of parameters. The resulting MobileNet can be nearly as accurate as VGG-16 while being $32\times$ smaller and $27\times$ faster.

More recently, [26] introduced a convolutional architecture for sequence-to-sequence tasks like translation, namely SliceNet, also based on the use of depthwise separable convolutions. It extended on previous work such as [24] to indicate that standard convolutions can be replaced with deptwise separable convolutions in order to obtain a model that is

simultaneously cheaper to run and smaller, while maintaining state-of-the-art accuracy.

VII. Evaluation

Now that we have surveyed the different methods, we have chosen to study quantization and compact network design from the categories presented beforehand and implemented them in order to evaluate their performance. We have chosen to implement VGGNet [33] (16 layers, configuration D) as the baseline model in order to compare and evaluate the different methods. We train our different implementations of the network on the popular benchmark dataset CIFAR-10. A slight modification was made to the the structure of the fully-connected layers of the network described in [33]. Since we are working with CIFAR-10 dataset (images $32 \times 32 \times 3$) instead of ImageNet (images $224 \times 224 \times 3$) as in the paper, we replaced the three fully-connected layers with one fully-connected layer of input size 512 ($1 \times 1 \times 512$). This change has a sizeable impact in terms of number of parameters, but we should still be able to observe a difference between the different methods in comparison. Moreover, our training method uses a batch size of 128 instead of 256 since the dataset used is smaller than the original ImageNet. A quick overview of the training and testing information is shown in Table I. The implementation and evaluation was done using the popular framework PyTorch [35]. We also made our code available[1].

TABLE I
Training and Testing Overview

Dataset	Dimension	Labels
CIFAR-10	3072 (32 x 32 color)	10
Training set	**Test set**	**Training epochs**
50K	10K	200

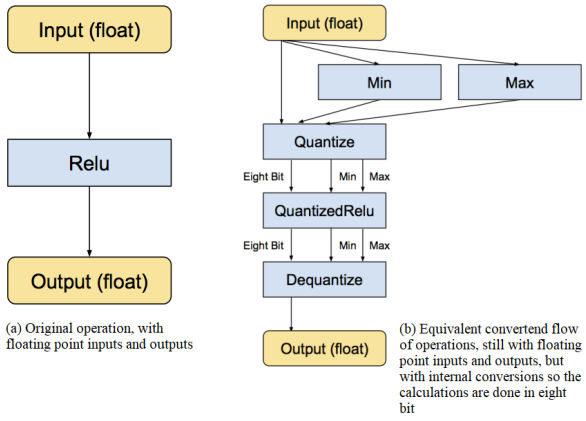
Our architecture implementation based on VGG-16 [33] can be seen in Table III, along with our compact architecture. Note that the last softmax layer is not necessary in the main model, because we use a loss during training called cross_entropy that combines the log softmax with the negative log likelihood (more stable than using softmax straightaway).

A. Quantization

From 16-bit, to 8-bit and even down to 1-bit, different quantization methods have recently been explored and tested in order to reduce model size for storage as well as computational resources. Although many of the recent papers also explore quantization during the training phase, the need for lower bitwidth inference is even greater with modern applications' deployment needs on resource limited hardware. Additionally, many of the models that we already use and know well are available on our favorite frameworks as pre-trained models, so being able to convert them directly to their quantized form is very convenient. Thus, we chose to implement our quantization algorithm to convert a pre-trained floating point model in order to use its quantized form for inference, the most important part when deploying it on limited resource hardware. In order to convert the pre-trained model to its quantized counterpart, we make use of PyTorch's [35] *nn.Module*'s dictionary to convert the values of the different parameters and layers' activations.

Sadly, at the time of writing, PyTorch [35] doesn't support lower bitwidth operations (e.g 8-bit fixed-point operations), while Tensorflow [36] does (for the most part). This support is expected to appear with the release of PyTorch 1.0 (currently 0.4.0). As a result, our implementation is more of a simulation since the quantized network will still be computed with 32-bit floating point representation of the tensors and floating point arithmetic.

In Tensorflow [36], quantized data types (i.e., quantized 8-bit signed or unsigned integer, quantized 16-bit signed or unsigned integer) were implemented to represent the type of the elements in a tensor, which are used with a low-precision general matrix to matrix multiplication (GEMM) framework to compute low bitwidth arithmetic. This allows for lower bitwidth calculations at train-time when implemented in the training loop, and more importantly during inference. The quantization process stores the minimum and maximum value of a tensor in 32-bit floating point representation, and compresses the tensor's float elements in the range of the chosen quantized data type. It is important to note that the intermediate results are stored in 32-bit floating point representation to preserve full accuracy, and that the values are only compressed to their quantized counterpart before computing some of the operations available in quantized form or when returning the results. The quantization process in Tensorflow [36] can be summarized by the flowchart presented in Fig. 8.



(a) Original operation, with floating point inputs and outputs

(b) Equivalent convertend flow of operations, still with floating point inputs and outputs, but with internal conversions so the calculations are done in eight bit

Fig. 8. Tensorflow quantization implementation. The original operation with float inputs and outputs (a) and the equivalent flow of operations with lower bitwidth operations (b).

While there are many different quantization methods such as the linear quantization presented in [5], our implementation is based on Tensorflow's [36] representation for quantized tensors. This is represented with two floats that store the

[1] https://github.com/laggui/nn_compress

overall minimum and maximum values corresponding to the lowest and highest quantized value of a tensor. Each element in the quantized tensor represents a float value in that range, distributed linearly between the minimum and maximum. For example, with a minimum of -15.0 and a maximum of 25.0, the 8-bit quantized values are represented as in Table II.

TABLE II
QUANTIZED VALUE RANGE EXAMPLE

Stored float	Quantized value
-15.0	0
5.0	128
25.0	255

Though as previously mentioned, PyTorch [35] doesn't support lower bitwidth calculations at the time of writing, thus our quantization implementation is more of a simulation. In other words, the values of the model's weights and activations are converted to their 8-bit counterpart and then converted back to 32-bit floats, and inference is completed on the *quantized-dequantized* values since the framework doesn't currently support quantized operations. Therefore, the drop in accuracy observed in Table V is only due to the error introduced by quantizing and dequantizing the values. If PyTorch [35] were to support inference with lower bitwidth operations, the quantization process would be similar to the one shown in Fig. 8. In this case, quantization would effectively reduce model size for storage as well as computational resources during inference.

B. Compact Network Design

In order to have more comparable results, we chose to implement the VGG-16 architecture with depthwise separable convolutions. Our compact VGG-16 architecture (as shown in Table III) is based on the MobileNet [24] implementation, replacing all of the standard convolutions with depthwise separable convolutions except for the first layer which is a full convolution, using both batchnorm and ReLU nonlinearities for both layers of the depthwise separable convolution. Theoretically, this factorization has the effect of drastically reducing computation and model size. In practice, the efficiency of this method also depends on the implementation within the framework used (here, PyTorch [35]).

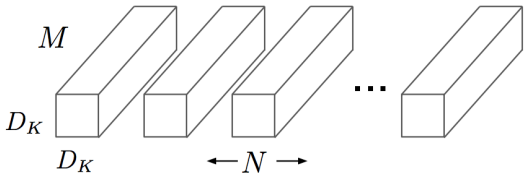
As we all know, a standard convolution both filters and combines inputs into a new set of outputs in one step. On the other hand, the depthwise separable convolution splits this into two layers: a separate layer for filtering (the depthwise convolution) and a separate layer for combining the new features (the point-wise convolution). Fig. 9 shows how a standard convolution is factorized to the two-step depthwise separable convolution.

To better understand how the use of depthwise separable convolutions reduces computation and model size, let's compare this operation to the standard convolution. Let us

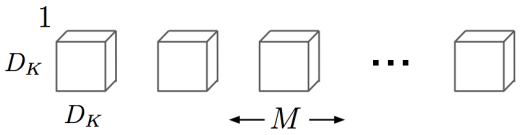
[2]The convolutional layer parameters are denoted as "conv⟨receptive field size⟩⟨/type⟩-⟨number of channels⟩". The ReLU activation function and batch normalizations are not shown for brevity.

TABLE III
CONVNET CONFIGURATIONS FOR OUR BASE VGG-16 ARCHITECTURE AND THE COMPACT VGG-16 ARCHITECTURE WITH DEPTHWISE SEPARABLE (DW) CONVOLUTIONS.

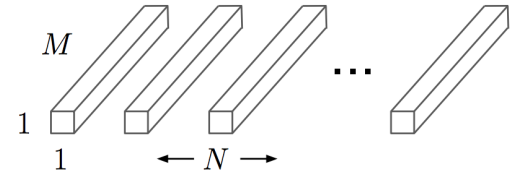
ConvNet Configuration[2]	
VGG-16	Compact VGG-16
conv3-64 conv3-64	conv3-64 conv3/dw-64
Maxpool	
conv3-128 conv3-128	conv3/dw-128 conv3/dw-128
Maxpool	
conv3-256 conv3-256 conv3-256	conv3/dw-256 conv3/dw-256 conv3/dw-256
Maxpool	
conv3-512 conv3-512 conv3-512	conv3/dw-512 conv3/dw-512 conv3/dw-512
Maxpool	
conv3-512 conv3-512 conv3-512	conv3/dw-512 conv3/dw-512 conv3/dw-512
Maxpool	
FC-10	



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters



(c) 1×1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

Fig. 9. The standard convolutional filters in (a) are replaced by a depthwise seperable filter. This filter is composed of a depthwise convolution in (b) and a pointwise convolution in (c)[24].

consider the input of a convolutional layer, a feature map **F** of size $D_F \times D_F \times M$, where D_F is the spatial width and height of a square input feature map and M is the number of input channels (input depth). A standard convolutional layer is parametrized by a kernel **K** of size $D_K \times D_K \times M \times N$, where D_K is the spatial dimension of the kernel (assumed to be symmetrical), M is the number of input channels as previously defined and N is the number of output channels (output depth). This convolutional layer produces an output feature map **G** of size $D_G \times D_G \times N$, where D_G is the spatial width and height of a square output feature map and N is the number of output channels as previously defined.

We assume that the output **G** has the same dimensions as the input **F**, therefore $D_F = D_G$. This is assumed because of the configuration of our convolution but can be calculated using the formula in (1), which is explained in detail in [34].

$$D_G = \lfloor \frac{D_F + 2p - D_K}{s} \rfloor + 1 \tag{1}$$

where s is the stride size and p the amount of zero padding along both axes.

Equation (1) represents the most general case, convolving over a zero padded input using non-unit strides. In the most commonly used case of unit padding (i.e., $s = 1$), this relationship is represented in (2). Both (1) and (2) assume symmetric settings.

$$D_G = (D_F - D_K) + 2p + 1 \tag{2}$$

For D_K odd ($D_K = 2n + 1$, $n \in \mathbb{N}$) and $p = \lfloor D_K/2 \rfloor = n$, a desirable property comes out of (2): $D_G = D_F$, thus the output size is the same as the input size. This is the case with our implementation of the VGG-16 architecture. The only changes in size come from the Maxpool layers.

For a standard convolution, $D_K \times D_K \times M$ operations are performed at each position of the input. For the whole input **F** taking into account every output channel of **G**, the computational cost of a standard convolution is:

$$D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F \tag{3}$$

where the computational cost depends mutiplicatively on the kernel size $D_K \times D_K$, the feature map size $D_F \times D_F$, the number of input channels M and the number of output channels N.

As previously mentioned, the depthwise separable convolution is made up of a depthwise convolution followed by a point-wise convolution. By applying a single filter per input channel, a depthwise convolution has a computational cost of:

$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F \tag{4}$$

However it only filters input channels, it does not combine them to create new features. This is where the point-wise convolution comes in to compute a linear combination of the output of the depthwise convolutions to generate the new features. This point-wise convolution has a computational cost of $D_F \cdot D_F \cdot M \cdot N$, which brings the computational cost of the full depthwise separable convolution to:

$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + D_F \cdot D_F \cdot M \cdot N \tag{5}$$

which is the sum of the depthwise convolution and 1×1 point-wise convolution.

By expressing convolution as a two step process of filtering and combining, there is a reduction in computation of:

$$\frac{D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + D_F \cdot D_F \cdot M \cdot N}{D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F} = \frac{1}{N} + \frac{1}{D_K^2} \tag{6}$$

We can also compute the number of parameters (weights and bias) for a standard convolution as in (7) and for a depthwise separable convolution as in (8).

$$(D_K \cdot D_K \cdot M + 1) \cdot N \tag{7}$$

$$(D_K \cdot D_K + 1) \cdot M + (M + 1) \cdot N \tag{8}$$

Let's take the second convolution layer from our VGG-16 based architecture and our compact architecture as an example. The first convolution layer for both architectures is a 3×3 kernel and produces 64 output channels, while taking an image from the CIFAR-10 dataset as input (size $32 \times 32 \times 3$). Therefore, the second convolution for both architectures takes an input of size 64. As shown in Table IV, both the number of parameters and the computational cost are almost $8\times$ smaller with the depthwise separable convolution. The reduction in computation is proven by (6).

TABLE IV
COMPUTATIONAL COST AND NUMBER OF PARAMETERS OF A STANDARD CONVOLUTION VS DEPTHWISE SEPARABLE CONVOLUTION

	Standard	Depthwise separable
Parameters	$(3 \cdot 3 \cdot 64 + 1) \cdot 64 = 36,928$	$(3 \cdot 3 + 1) \cdot 64 + (64 + 1) \cdot 64 = 4,800$
Computational cost	$3 \cdot 3 \cdot 64 \cdot 64 \cdot 32 \cdot 32 = 37,748,736$	$3 \cdot 3 \cdot 64 \cdot 32 \cdot 32 + 32 \cdot 32 \cdot 64 \cdot 64 = 4,784,128$

Our compact architecture uses 3×3 depthwise separable convolutions, which uses almost 8-9$\times$ less computation than standard convolutions at the cost of a very small reduction in accuracy as shown in Table V.

C. Results

The training procedure generally followed [33], with 200 epochs and a training set of 50K images as shown in Table I. The batch size was set to 128, momentum to 0.9. The training was regularized by weight decay (the L_2 penalty multiplier set to $5 \cdot 10^{-4}$). The learning rate was initially set to 10^{-2}, and then decreased by a factor of 10 after 100 epochs. The initialization of the network weights used is the default method specified by PyTorch [35] for each layer, using a uniform distribution for convolutional and fully connected layers with the method described in [37] and filling up the tensor with values drawn from the uniform distribution for batch normalization layers.

As illustrated in Table V, our baseline model based on the VGG-16 [33] architecture achieved 90.03% accuracy on

the the test dataset with 14.73M parameters, while our compact network achieved extremely similar performance using almost $9\times$ less parameters and 8-$9\times$ less computation. This impact was also observed during training, where the baseline model took 11.26 hours to train compared to 2.24 hours for our compact architecture, a speedup rate accelerating the training phase by $5\times$.

TABLE V
RESULTS

Model	Accuracy	Parameters	MACs	Model Size
Baseline	90.03%	14.73M	313.2M	56.2 MB
Compact	89.98%	1.70M	38M	6.53MB
Quantized	88.13%	14.73M	313.2M	56.2MB

Our quantized model was also tested on the same dataset, and, as previously mentioned, was only a simulation of the process since PyTorch [35] doesn't support lower bitdwidth operations yet. Thus, the drop in accuracy observed in Table V is only due to the error introduced by quantizing and dequantizing the values. Since the quantization is done on a pre-trained model, the training time is not applicable here. In addition, the parameters are still stored using 32-bit floats and the network is using floating point operations, thus the different criteria remain the same as the baseline. Moreover, the simulated quantization actually requires more computation in order to quantize the different values. In the case of lower bitwidth operations support, such as in Tensorflow [36], quantization should have sizeable impact on model size as well as computational resources. For example, with 8-bit representation, the model size should effectively be $4\times$ smaller for the quantized model. The number of MACs should also be greatly reduced since support for lower bitwidth operations allows for less computational resources.

All tests were done using Google Colaboratory, a free Jupyter notebook environment and cloud service with GPU enabled machines. The specifications of the virtual machine used are shown in Table VI.

TABLE VI
MACHINE SPECIFICATIONS

CPU	2-core Xeon 2.3GHz
RAM	13GB
Disk	365GB HDD
GPU	NVIDIA Tesla K80 12GB

VIII. DISCUSSION

In this paper, we provided a survey and evaluation of recent works in efficient processing of deep neural networks (DNNs), more precisely on compressing and accelerating such networks. Here we discuss on the different compression approaches and their applications, and point out a few topics that deserve further investigation in the future.

There is no golden rule to evaluate which of the categorized methods described is the best. The proper approach chosen really depends on the applications and requirements. Interestingly, these methods are independently designed and can greatly complement each other. For example, the deep compression method in [16] combined network pruning and quantization in order to achieve impressive storage compression, as well as great speedup and energy efficiency.

In regards to the application of the different techniques, some general suggestions can be taken into consideration:

- Network pruning is a generally popular approach that can always be used to obtain reasonable compression rate while preserving a very similar accuracy to the original network.
- When the application involves the use of a small dataset, the knowledge distillation approach can be an interesting solution. The compressed student model can benefit from the transferred knowledge of the teacher model, making a robust network from a small dataset.
- If the application requires a compressed model from a pre-trained model, methods such as network pruning, low-rank approximation or even quantization when lower bitwidth operations are supported can be used. Many open-source tools for different frameworks are already available online and ready to use for some of these methods.

We have also shown through our evaluation that a compact network design can be extremely beneficial in both network compression and acceleration, improving both the training phase and inference. Although previous works have shown great results concerning quantization, it would be interesting to test the impact of this method on our network when lower bitwidth operations become available in PyTorch [35], or by repeating the tests in Tensorflow [36] since it is already supported by this framework. Another interesting avenue we would like to explore is filter-level pruning, more particularly by implementing the method proposed in [21]. As shown in recent works, filter-level pruning pruning can directly reduce the feature map width and shrink the model into a thinner one, but more importantly reduce network complexity thus accelerating the network. It is efficient but also challenging because removing filters can dramatically change the input of the following layer, and it would be interesting to actively test its effect.

On a final note, we summarized some of the most popular methods in deep neural networks model compression and acceleration. This is by no means an exhaustive list of the existing methods, but as mentioned a survey of the most popular techniques. And with growing interest in this field a plethora of new approaches are continuously explored. In addition to the methods explored in this paper, we would like to further explore conditional computation in the future. It operates by selectively activating only parts of the network at a time, allowing for faster models.

APPENDIX

A detailed breakdown[3]of our architectures is illustrated in Table VII and Table VIII respectively.

[3]The ReLU activation function and batch normalizations were omitted for brevity, but it is important to note that batchnorm layers introduce new parameters (one per different channel for convolutions).

TABLE VII

DETAILED BREAKDOWN OF OUR VGG-16 ARCHITECTURE

INPUT:	$[32 \times 32 \times 3]$	params: 0	MACs: 0
CONV3-64:	$[32 \times 32 \times 64]$	params: $(3 \cdot 3 \cdot 3 + 1) \cdot 64 = 1,792$	MACs: $(3 \cdot 3 \cdot 3 \cdot 64 \cdot 32 \cdot 32) = 1,769,472$
CONV3-64:	$[32 \times 32 \times 64]$	params: $(3 \cdot 3 \cdot 64 + 1) \cdot 64 = 36,928$	MACs: $(3 \cdot 3 \cdot 64 \cdot 64 \cdot 32 \cdot 32) = 37,748,736$
POOL2:	$[16 \times 16 \times 64]$	params: 0	MACs: 0
CONV3-128:	$[16 \times 16 \times 128]$	params: $(3 \cdot 3 \cdot 64 + 1) \cdot 128 = 73,856$	MACs: $(3 \cdot 3 \cdot 64 \cdot 128 \cdot 16 \cdot 16) = 18,874,368$
CONV3-128:	$[16 \times 16 \times 128]$	params: $(3 \cdot 3 \cdot 128 + 1) \cdot 128 = 147,584$	MACs: $(3 \cdot 3 \cdot 128 \cdot 128 \cdot 16 \cdot 16) = 37,748,736$
POOL2:	$[8 \times 8 \times 128]$	params: 0	MACs: 0
CONV3-256:	$[8 \times 8 \times 256]$	params: $(3 \cdot 3 \cdot 128 + 1) \cdot 256 = 295,168$	MACs: $(3 \cdot 3 \cdot 128 \cdot 256 \cdot 8 \cdot 8) = 18,874,368$
CONV3-256:	$[8 \times 8 \times 256]$	params: $(3 \cdot 3 \cdot 256 + 1) \cdot 256 = 590,080$	MACs: $(3 \cdot 3 \cdot 256 \cdot 256 \cdot 8 \cdot 8) = 37,748,736$
CONV3-256:	$[8 \times 8 \times 256]$	params: $(3 \cdot 3 \cdot 256 + 1) \cdot 256 = 590,080$	MACs: $(3 \cdot 3 \cdot 256 \cdot 256 \cdot 8 \cdot 8) = 37,748,736$
POOL2:	$[4 \times 4 \times 256]$	params: 0	MACs: 0
CONV3-512:	$[4 \times 4 \times 512]$	params: $(3 \cdot 3 \cdot 256 + 1) \cdot 512 = 1,180,160$	MACs: $(3 \cdot 3 \cdot 256 \cdot 512 \cdot 4 \cdot 4) = 18,874,368$
CONV3-512:	$[4 \times 4 \times 512]$	params: $(3 \cdot 3 \cdot 512 + 1) \cdot 512 = 2,359,808$	MACs: $(3 \cdot 3 \cdot 512 \cdot 512 \cdot 4 \cdot 4) = 37,748,736$
CONV3-512:	$[4 \times 4 \times 512]$	params: $(3 \cdot 3 \cdot 512 + 1) \cdot 512 = 2,359,808$	MACs: $(3 \cdot 3 \cdot 512 \cdot 512 \cdot 4 \cdot 4) = 37,748,736$
POOL2:	$[2 \times 2 \times 512]$	params: 0	MACs: 0
CONV3-512:	$[2 \times 2 \times 512]$	params: $(3 \cdot 3 \cdot 512 + 1) \cdot 512 = 2,359,808$	MACs: $(3 \cdot 3 \cdot 512 \cdot 512 \cdot 2 \cdot 2) = 9,437,184$
CONV3-512:	$[2 \times 2 \times 512]$	params: $(3 \cdot 3 \cdot 512 + 1) \cdot 512 = 2,359,808$	MACs: $(3 \cdot 3 \cdot 512 \cdot 512 \cdot 2 \cdot 2) = 9,437,184$
CONV3-512:	$[2 \times 2 \times 512]$	params: $(3 \cdot 3 \cdot 512 + 1) \cdot 512 = 2,359,808$	MACs: $(3 \cdot 3 \cdot 512 \cdot 512 \cdot 2 \cdot 2) = 9,437,184$
POOL2:	$[1 \times 1 \times 512]$	params: 0	MACs: 0
FC-512:	$[1 \times 1 \times 10]$	params: $(512 + 1) \cdot 10 = 5,130$	MACs: $(512 \cdot 10) = 5,120$
TOTAL		params: $14,719,818$ with batchnorm: $+8,448 = 14,728,266$	MACs: $313,201,664$

TABLE VIII

DETAILED BREAKDOWN OF OUR VGG-16 COMPACT ARCHITECTURE

INPUT:	$[32 \times 32 \times 3]$	params: 0	MACs: 0
CONV3-64:	$[32 \times 32 \times 64]$	params: $(3 \cdot 3 \cdot 3 + 1) \cdot 64 = 1,792$	MACs: $(3 \cdot 3 \cdot 3 \cdot 64 \cdot 32 \cdot 32) = 1,769,472$
CONV3/DW-64:	$[32 \times 32 \times 64]$	params: $(3 \cdot 3 + 1) \cdot 64 + (64 + 1) \cdot 64 = 4,800$	MACs: $(3 \cdot 3 \cdot 64 \cdot 32 \cdot 32) + (32 \cdot 32 \cdot 64 \cdot 64) = 4,784,128$
POOL2:	$[16 \times 16 \times 64]$	params: 0	MACs: 0
CONV3/DW-128:	$[16 \times 16 \times 128]$	params: $(3 \cdot 3 + 1) \cdot 64 + (64 + 1) \cdot 128 = 8,960$	MACs: $(3 \cdot 3 \cdot 64 \cdot 16 \cdot 16) + (16 \cdot 16 \cdot 64 \cdot 128) = 2,244,608$
CONV3/DW-128:	$[16 \times 16 \times 128]$	params: $(3 \cdot 3 + 1) \cdot 128 + (128 + 1) \cdot 128 = 17,792$	MACs: $(3 \cdot 3 \cdot 128 \cdot 16 \cdot 16) + (16 \cdot 16 \cdot 128 \cdot 128) = 4,489,216$
POOL2:	$[8 \times 8 \times 128]$	params: 0	MACs: 0
CONV3/DW-256:	$[8 \times 8 \times 256]$	params: $(3 \cdot 3 + 1) \cdot 128 + (128 + 1) \cdot 256 = 34,304$	MACs: $(3 \cdot 3 \cdot 128 \cdot 8 \cdot 8) + (8 \cdot 8 \cdot 128 \cdot 256) = 2,170,880$
CONV3/DW-256:	$[8 \times 8 \times 256]$	params: $(3 \cdot 3 + 1) \cdot 256 + (256 + 1) \cdot 256 = 68,352$	MACs: $(3 \cdot 3 \cdot 256 \cdot 8 \cdot 8) + (8 \cdot 8 \cdot 256 \cdot 256) = 4,341,760$
CONV3/DW-256:	$[8 \times 8 \times 256]$	params: $(3 \cdot 3 + 1) \cdot 256 + (256 + 1) \cdot 256 = 68,352$	MACs: $(3 \cdot 3 \cdot 256 \cdot 8 \cdot 8) + (8 \cdot 8 \cdot 256 \cdot 256) = 4,341,760$
POOL2:	$[4 \times 4 \times 256]$	params: 0	MACs: 0
CONV3/DW-512:	$[4 \times 4 \times 512]$	params: $(3 \cdot 3 + 1) \cdot 256 + (256 + 1) \cdot 512 = 134,144$	MACs: $(3 \cdot 3 \cdot 256 \cdot 4 \cdot 4) + (4 \cdot 4 \cdot 256 \cdot 512) = 2,134,016$
CONV3/DW-512:	$[4 \times 4 \times 512]$	params: $(3 \cdot 3 + 1) \cdot 512 + (512 + 1) \cdot 512 = 267,776$	MACs: $(3 \cdot 3 \cdot 512 \cdot 4 \cdot 4) + (4 \cdot 4 \cdot 512 \cdot 512) = 4,268,032$
CONV3/DW-512:	$[4 \times 4 \times 512]$	params: $(3 \cdot 3 + 1) \cdot 512 + (512 + 1) \cdot 512 = 267,776$	MACs: $(3 \cdot 3 \cdot 512 \cdot 4 \cdot 4) + (4 \cdot 4 \cdot 512 \cdot 512) = 4,268,032$
POOL2:	$[2 \times 2 \times 512]$	params: 0	MACs: 0
CONV3/DW-512:	$[2 \times 2 \times 512]$	params: $(3 \cdot 3 + 1) \cdot 512 + (512 + 1) \cdot 512 = 267,776$	MACs: $(3 \cdot 3 \cdot 512 \cdot 2 \cdot 2) + (2 \cdot 2 \cdot 512 \cdot 512) = 1,067,008$
CONV3/DW-512:	$[2 \times 2 \times 512]$	params: $(3 \cdot 3 + 1) \cdot 512 + (512 + 1) \cdot 512 = 267,776$	MACs: $(3 \cdot 3 \cdot 512 \cdot 2 \cdot 2) + (2 \cdot 2 \cdot 512 \cdot 512) = 1,067,008$
CONV3/DW-512:	$[2 \times 2 \times 512]$	params: $(3 \cdot 3 + 1) \cdot 512 + (512 + 1) \cdot 512 = 267,776$	MACs: $(3 \cdot 3 \cdot 512 \cdot 2 \cdot 2) + (2 \cdot 2 \cdot 512 \cdot 512) = 1,067,008$
POOL2:	$[1 \times 1 \times 512]$	params: 0	MACs: 0
FC-512:	$[1 \times 1 \times 10]$	params: $(512 + 1) \cdot 10 = 5,130$	MACs: $(512 \cdot 10) = 5,120$
TOTAL		params: $1,682,506$ with batchnorm: $+15,872 = 1,698,378$	MACs: $38,018,048$

ACKNOWLEDGMENTS

The first author would like to especially thank Marco Pedersoli for making this project possible and for his insightful advice on the topic at hand.

REFERENCES

[1] LeCun, Y., Denker, J.S. & Solla, S.A. (1990). Optimal Brain Damage. In NIPS 1989.

[2] Reed, R. (1993). Pruning algorithms - A Survey. In IEEE Transactions on Neural Networks, vol. 4, no. 5, pp. 740-747, Sep 1993.

[3] Krizhevsky, A., Sutskever, I. & Hinton, G.E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In NIPS 2012.

[4] Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R. & Bengio, Y. (2016). Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 or -1. In NIPS 2016. arXiV preprint arXiv:1602.02830v3.

[5] Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R. & Bengio, Y. (2016). Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. arXiV preprint arXiv:1609.07061v1.

[6] Tang, W., Hua, G. & Wang, L. (2017). How to Train a Compact Binary Neural Network with High Accuracy? In AAAI 2017.

[7] Lin, D. D., Talathi, S. S. & Annapureddy, V. S. (2016). Fixed Point Quantization of Deep Convolutional Networks. In ICML 2016. arXiv preprint arXiv:1511.06393v3.

[8] Courbariaux, M., David, J.-P. & Bengio, Y. (2015). Training deep neural networks with low precision multiplications. In ICLR 2015. arXiv preprint arXiv:1412.7024v5.

[9] Courbariaux, M., Bengio, Y. & David, J.-P. (2016). BinaryConnect: Training Deep Neural Networks with binary weights during propagations. In NIPS 2016. arXiv preprint arXiv:1511.00363v3.

[10] Gupta, S., Agrawal, A., Gopalakrishnan, K. & Narayanan, P. (2015). Deep Learning with Limited Numerical Precision. arXiv preprint arXiv:1502.02551v1.

[11] Wu, S., Li, G., Chen, F. & Shi, L. (2018). Training and Inference with Integers in Deep Neural Networks. In ICLR 2018. arXiv preprint arXiv:1802.04680v1.

[12] Han, S., Mao, H., Gong, E., Tang, S., Dally, W. J.,Pool, J., Tran, J., Catanzaro, B., Narang, S., Elsen, E., Vajda, P. & Paluri, M. (2017). DSD: Dense-Sparse-Dense Training for Deep Neural Networks. In ICLR 2017. arXiv preprint arXiv:1607.04381v2.

[13] Han, S., Pool, J., Tran, J. & Dally, W. J. (2015). Learning both Weights and Connections for Efficient Neural Networks. In NIPS 2015. arXiv preprint arXiv:1506.02626v3.

[14] Chen, W., Wilson, J.T., Tyree, S., Weinberger, K.Q. & Chen. Y. (2015). Compressing Neural Networks with the Hashing Trick. arXiV preprint arXiv:1504.04788v1.

[15] Hinton, G.E., Srivastava, N., Krizhevsky, A., Sutskever, I. & Salakhutdinov, R.R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. arXiv preprint arXiv:1207.0580v1.

[16] Han, S., Mao, H. & Dally, W.J. (2016). Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In ICLR 2016. arXiV preprint arXiv:1510.00149v5.

[17] Tai, C., Xiao, T.,Zhang, Y., Wang, X. & E, W. (2016). Convolutional neural networks with low-rank regularization. In ICLR 2016. arXiV preprint arXiv:1511.06067v3.

[18] Jaderberg, M., Vedaldi, A. & Zisserman, A. (2014). Speeding up Convolutional Neural Networks with Low Rank Expansions. arXiV preprint arXiv:1405.3866v1.

[19] Sainath, T.N., Kingsbury, B., Sindhwani, V., Arisoy, E. & Ramabhadran, B. (2013). Low-rank Matrix Factorization for Deep Neural Network training with high-dimensional output targets. In IEEE International Conference on Acoustics, Speech and Signal Processing, 2013.

[20] Lebedev, V., Ganin, Y., Rakhuba, M., Oseledets, I. & Lempitsky, V. (2015). Speeding-up Convolutional Neural Networks Using Fine-tuned CP-Decomposition. In ICLR 2015. arXiV preprint arXiV:1412.6553v3.

[21] Li, H., Kadav, A., Durdanovic, I., Samet, H. & Graf, H.P. (2017). Pruning Filters for Efficient ConvNets. In ICLR 2017. arXiV preprint arXiv:1608.08710

[22] Luo, J.H., Wu, J. & Lin, W. (2017). ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression. arXiv preprint arXiv:1707.06342v1.

[23] Molchanov, P., Tyree, S., Karras, T., Aila, T. & Kaut, J. (2017). Pruning Convolutional Neural Networks for Resource Efficient Inference. ICLR 2017. arXiV preprint arXiV:1611.06440v2.

[24] Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M. & Adam, H. (2017). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiV preprint arXiv:1704.04861v1.

[25] Xie, S., Girshick, R., Tu, Z. & He, K. (2017). Aggregated Residual Transformations for Deep Neural Networks. arXiV preprint arXiV 1611.05431v2.

[26] Kaiser, L., Gomez, A.N. & Chollet, F. (2018). Depthwise Separable Convolutions for Neural Machine Translation. In ICLR 2018.

[27] Buciluǎ, C., Caruana, R. & Niculescu-Mizil, A. (2006). Model compression. In KDD 2006.

[28] Hinton, G., Vinyals, O. & Dean, J. (2015). Distilling the Knowledge in a Neural Network. In NIPS Deep Learning Workshop 2014. arXiV preprint arXiv:1503.02531v1.

[29] Lopes, R.G., Fenu, S. & Starner, T. (2017). Data-Free Knowledge Distillation for Deep Neural Networks. NIPS 2017 Workshop on Learning with Limited Data. arXiV preprint arXiv:1710.07535v2.

[30] Sau, B.B. & Balasubramanian, V.N. (2016). Deep Model Compression: Distilling Knowledge from Noisy Teachers. arXiV preprint arXiV:1610.09650v2.

[31] Wang, C., Lan, X. & Zhang, Y. (2017). Model Distillation with Knowledge Transfer from Face Classification to Alignment and Verification. arXiV preprint arXiV:1709.02929v2.

[32] Polino, A., Pascanu, R. & Alistarh, D. (2018). Model Compression via Distillation and Quantization. In ICLR 2018.

[33] Simonyan, K. & Zisserman, A. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition. In ICLR 2015. arXiV preprint arXiV:1409.1556v6.

[34] Dumoulin, F. & Visin, F. (2018). A guide to convolution arithmetic for deep learning. arXiV preprint arXiV:1603.07285v2.

[35] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L. & Lerer, A. (2017). Automatic differentiation in PyTorch.

[36] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y. & Zheng, X. (2015). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. arXiV preprint arXiv:1603.04467v2.

[37] He, K., Zhang, X., Ren, S. & SunDelving, J. (2015). Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. arXiV preprint arXiv:1502.01852v1.