Approximations in Deep Learning

Etienne Dupuis, Silviu-Ioan Filip, Olivier Sentieys, David Novo, Ian O'Connor, Alberto Bosio

Approximations in Deep Learning

Etienne Dupuis[1], Silviu Filip[2], Olivier Sentieys[2], David Novo[3], Ian O'Connor[1], and Alberto Bosio[1]

[1]Univ Lyon, ECL, INL, Ecully, FR, firstname.lastname@ec-lyon.fr
[2]Univ Rennes, Inria, IRISA, FR, firstname.lastname@inria.fr
[3]LIRMM, Université de Montpellier, CNRS, FR, david.novo@lirmm.fr

Abstract

The design and implementation of Deep Learning (DL) models is currently receiving a lot of attention from both industrials and academics. However, the computational workload associated with DL is often out of reach for low-power embedded devices and is still costly when run on datacenters. By relaxing the need for fully precise operations, Approximate Computing (AxC) substantially improves performance and energy efficiency. DL is extremely relevant in this context, since playing with the accuracy needed to do adequate computations will significantly enhance performance, while keeping the quality of results in a user-constrained range. This chapter will explore how AxC can improve the performance and energy efficiency of hardware accelerators in DL applications during inference and training.

Contents

1 Introduction

Deep Neural Networks (DNNs) [LBH15], and in particular, Convolutional Neural Networks (CNNs), are currently one of the most intensively and widely used predictive models in the field of machine learning. CNNs have been shown to give very good results for many complex tasks such as object recognition in images/videos, drug discovery, natural language processing, autonomous driving, and playing complex games [DLH+13, KSH17, CSKX15, SHM+16].

Despite these benefits, the computational workload involved in CNNs is often out of reach for low-power embedded devices, and/or is still very costly when ran on datacenter-style Component-Off-The-Shelf (COTS) hardware platforms. To give an example, the amazing performance of AlphaGo [SHM+16] required 4 to 6 weeks of training executed on 2000 CPUs and 250 GPUs for a total of about 600kW of power consumption (while the human brain of a Go player requires about 20W), which translates to over 2 TJ of energy consumption. Thus, a lot of research effort from both industrials and academics has been concentrated on defining/designing custom hardware platforms supporting these types of algorithms, to improve performance and/or energy efficiency [WGY+16, CKES16, LDJ+17].

CNNs show inherent resilience to insignificant errors due to their iterative nature and the underlying learning process. Therefore, an intrinsic tolerance to inexact computation is clear, and using the AxC paradigm to improve power and speed characteristics is, therefore, relevant [SSH15]. Indeed, CNNs mesh well with AxC techniques, especially with fixed-point arithmetic or low-precision floating-point implementations (it has been shown that even binary or ternary weights and arithmetic can be used), which moreover expose large fine-grain parallelism. They are therefore ideally suited for hardware acceleration using Field Programmable Gate Arrays (FPGAs) and/or Application-Specific Integrated Circuit (ASIC) implementations, as acknowledged by the large body of work on this topic. Although accelerators have demonstrated significant performance/energy gains compared to GPU/CPU implementations, they still require further efficiency to address future performance requirements [THBR17].

The goal of this chapter is to present an up-to-date view of state-of-the-art solutions applying AxC techniques to CNNs for both inference and training phases. It is structured as follows: Section 2 presents the background & context of using DNNs, the main focus of the chapter. Section 3 overviews AxC methods found in the literature that improve deep neural network inference performance. Approximation techniques for improving the training part of neural network design, which accounts for the majority of computing time and resources, are presented in Section 4. Section 5 discusses DNN accelerator research and the dedicated approximation methods, whereas Section 6 presents incubent directions for AxC research in DL. Section 7 concludes the chapter.

2 Background

Artificial intelligence (AI) is a broad field of study focused on replicating or simulating the intelligence of living beings (human or not). It encompasses various methods and techniques. These range from design space exploration methods like ant colony optimization that focuses on finding increasingly efficient paths through simple random exploration and reward-based reinforcement, to

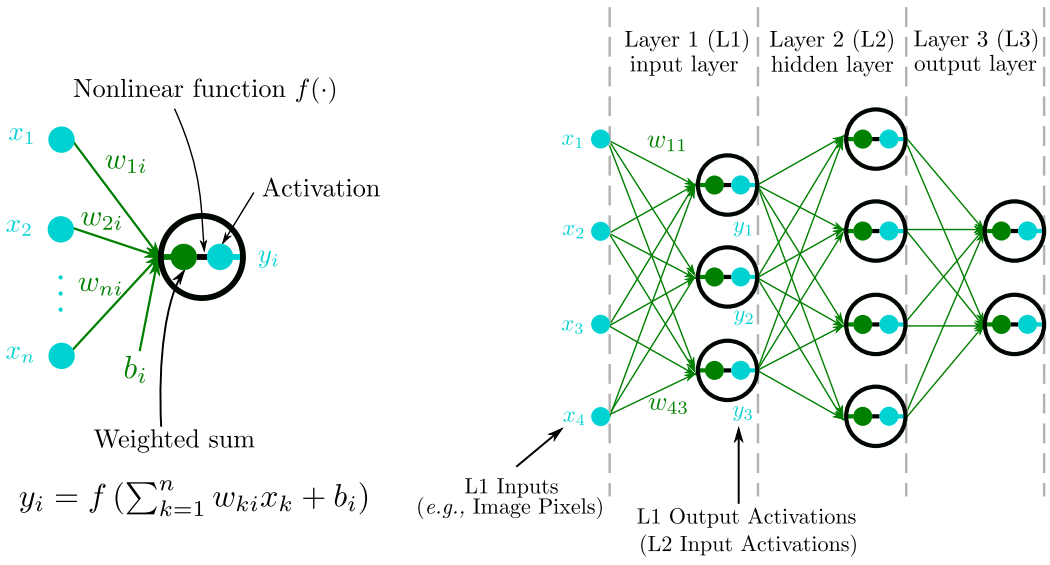
more complex approaches such as genetic algorithms that evolve a population towards a hopefully optimized solution by iteratively picking the best candidates and mutating them. In the last couple of decades, Machine Learning (ML) algorithms have gained the most traction, producing effective predictions/answers based on some trained behavior/model.

2.1 Context: From AI to DNNs

The ML subset of AI is focused on algorithms able to improve themselves through seeing already labeled input-output sample pairs and constructing models that attempt to match the expected outputs to this given data. An example is email filtering, deciding whether or not an email is spam based on its provenance, recipients, object, and other (meta)data. Generally, a model for this task is trained (*i.e.,* it *learns*) on a set of already labeled set of spam email data (the *training data set*) until it reaches the desired behavior (the *expected response*) with sufficient accuracy. It is then used with unseen data in the hope that it will still prove to be accurate (*i.e.,* generalize well). Evaluation of this generalization ability is frequently done on a so-called *test* or *validation* data set, different from the training data.

While email filtering can seem like a simple task, there are a plethora of use cases of varying difficulty where ML modeling is used, ranging from security (*e.g.,* in fraud detection) and business data analysis (*e.g.,* churn rate measurement) to computer vision, self-driving technologies, and other complex tasks. The model inputs can be both raw data or high-level features (for instance statistical aggregates of multiple input data samples) or other complex features that are task-dependent (*e.g.,* the presence of a horizontal line in an image). It is the task of the model to interpret this data and construct useful responses. Among the many tasks suitable for ML one can mention classification, regression, and semantic segmentation.

Traditionally, high-level features needed by a model were derived following a feature extraction step that was often performed by a human, requiring expert knowledge of relevant information. More recently, however, through the rise of DNNs in the ML ecosystem of approaches, this step can be performed automatically, the model is trained to discover relevant features, thus avoiding both the need for human expertise and the induced biases that might result from this.



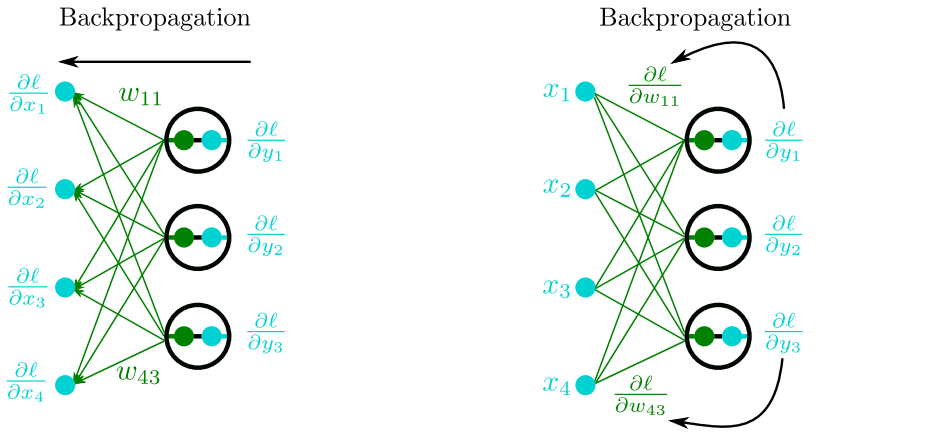
(a) Artificial neuron

(b) Simple neural network example

Figure 1: A basic DNN example and the associated terminology (adapted from [SCYE20, Figure 1.3]).

Artificial neural networks are based on the notion that the computation performed by a neuron is centered around a weighted sum of its input values. This is shown in Figure 1a, where multiple inputs $\{x_k\}_{k=1}^n$ are summed (scaled with *weights* $\{w_{ki}\}_{k=1}^n$) together with an optional *bias* term b_i. The neuron *output* y_i is determined by the application of a *nonlinear activation function* f to this weighted sum. There are many activation functions used in practice, but among the most common are $f(x) = \text{ReLU}(x) := \max\{0, x\}$ and $f(x) = \tanh(x) := (e^x - e^{-x})/(e^x + e^{-x})$.

Such neurons are grouped together to form *layers*. The present chapter is focused on feedforward networks, where the outputs of a layer are then used as inputs for subsequent layers[1]. This is exemplified in Figure 1b. The inputs and outputs of a layer are also known as *input* and *output activations*, respectively. When discussing visual data, they are also known as input and output *feature maps*. The first and last layer in the network are generally known as the *input* and *output layer*, respectively. In between them, there is a number of intermediate layers, called *hidden layers*. The main characteristic of DL and DNNs is that the number of hidden layers can grow quite large, from two layers up to even one thousand.



(a) Compute the gradient of the loss relative to the layer inputs $\left(\frac{\partial \ell}{\partial x_i} = \sum_j w_{ij} \frac{\partial \ell}{\partial y_j}\right)$

(b) Compute the gradient of the loss relative to the weights $\left(\frac{\partial \ell}{\partial w_{ij}} = \frac{\partial \ell}{\partial y_j} x_i\right)$

Figure 2: A backpropagation example through a neural network (adapted from [SCYE20, Figure 1.6]).

The process of using an artificial neural network with a set of given parameters (*e.g.,* weights and bias terms) is called *inference*. For the neural network to be useful, its inference output has to match as closely as possible an expected/ideal output. This is measured through a *loss* function ℓ that compares how far the resulting output on (subsets of) the training and test data sets is to the expected output. Thus, the goal of *training* a neural network is to find/learn a set of parameters that minimizes the average loss over a large training set.

To train a network, its weights (w_{ij}) are usually updated using a form of Stochastic Gradient Descent (SGD) iterative optimization process. This means that weight is updated by a scaled version of the partial derivative of the loss function ℓ with respect to the weight. In the most basic form, at iteration t, the weight update formula is given by:

$$w_{ij}^t = w_{ij}^{t-1} - \alpha \frac{\partial \ell}{\partial w_{ij}^{t-1}}, \tag{1}$$

where α is called the learning rate[2]. The partial derivatives of ℓ can be computed efficiently through a process called *backpropagation* [RHW86]. It is effectively an application of the *chain*

[1]There are classes of *recurrent* neural networks that allow outputs of a layer to be connected to inputs of previous layers. While they are not discussed here any further, they are frequently used to process sequential data (*e.g.,* speech, text).

[2]The deep learning optimization literature describes many ways how to perform the parameter updates and how to choose the learning rate.

rule from calculus, and it works by passing values backward through the network to compute how ℓ is affected by each weight. At each layer, the procedure is twofold and is exemplified in Figure 2. To backpropagate through a layer: (a) compute the gradient of the loss with respect to the weights, $\partial\ell/\partial w_{ij}$, from the layer inputs (*i.e.,* the forward activations x_i) and the gradients of the loss relative to the layer outputs, $\partial\ell/\partial y_j$; and (b) compute the gradient of the loss relative to the layer inputs, $\partial\ell/\partial x_i$, from the layer weights, w_{ij}, and the gradients of the loss relative to the layer outputs, $\partial\ell/\partial y_j$.

Computing the gradients of the loss function ℓ over the entire dataset is generally much too complicated in practice, which is why the loss is usually taken only on a (small) subset, called a *mini batch*, of the training data. The use of batches allows taking advantage of single instruction multiple data (SIMD)-like parallelism on modern GPUs while keeping the complexity of gradient computation manageable. A complete iteration of the training process is called an *epoch* and requires passing through all of the mini-batches, applying (1) for each one of the corresponding average losses ℓ. Training is carried out for several epochs until convergence to an appropriate solution is reached.

Both inference and training amount in most part to the same type of computations (*i.e.,* matrix/vector additions and multiplications). There are important differences, however. For one, as the previous paragraph suggests, training is much more expensive, since apart from passing through the entire training data multiple times, it also requires that intermediate outputs and partial derivatives be stored when performing backpropagation. Secondly, due to the gradient update rule, the precision requirements for training are generally higher than for inference, thus also affecting performance. The effect is that the inference quantization techniques that will be discussed in this chapter are not usually directly applicable to training as well.

2.2 Deep Learning Landscape

While artificial neural networks have a long history dating as far back as the 1940s, practical applications using digital neurons did not arrive until the late 1980s, when the LeNet-5 [LCJB+89, LBBH98] network architecture was used for hand-written digit recognition. It is only in the early 2010s however, with the synergy of three major factors, that artificial neural network models have started to take off, under the names *deep learning* and *deep neural networks*. These factors are: (1) the availability of large and labeled datasets that are needed to train complex models; (2) the advance in computational power of units such as GPUs that allow DNN training to be executed in reasonable time (days or weeks instead of years); (3) development of new algorithmic techniques (*e.g.,* the Adam gradient descent optimization algorithm [KB14]) that enable improved accuracy at a larger scale.

The importance of large and comprehensive datasets cannot be overstated. If not careful, a small training dataset used in conjunction with a complex DNN can easily lead to *overfitting* (*i.e.,* the model matches the training data extremely well but does not generalize to unseen data accurately). For computer vision, arguably the most popular dataset in recent years has been ImageNet [DDS+09], a collection of one million high-resolution images that are generally associated with the ILSVRC [RDS+15] image recognition contest that uses 1000 labeled categories. Smaller datasets such as MNIST [LC10] and CIFAR [Kri09] have also been used extensively in DNN research for inference and training acceleration.

Apart from the data, the choice of model (network architecture and associated parameters) is also crucial in the success of a DL approach. In what follows (Section 2.3), our focus is on CNN models suited to process visual data.

The current surge of interest in DL is also facilitated by the availability of tools and frameworks that allow for the easy prototyping and design of DNN models. Prominent examples include Tensorflow [A+15] and Pytorch [PGM+19]. The open-source nature of these alternatives offers the possibility to design extensions that can be leveraged throughout a model's lifecycle (from initial prototype to deployment).

Depending on the intended use of DNN models, they can be found in different environments

with various computing power and energy consumption characteristics. At one end of the spectrum there are edge devices characterized by low-power and limited computational capabilities, while at the other end power-hungry cloud devices with a high-performance computing profile are dominant.

2.3 Convolutional Neural Networks

The most basic layer in a feedforward network is the Fully Connected (FC) or dense layer. It is characterized by the fact that each neuron in the layer is connected to all the neurons in the previous layer. FC layers are parameterized by the number of neurons they contain. An example is shown in Figure 1, which only has FC layers.

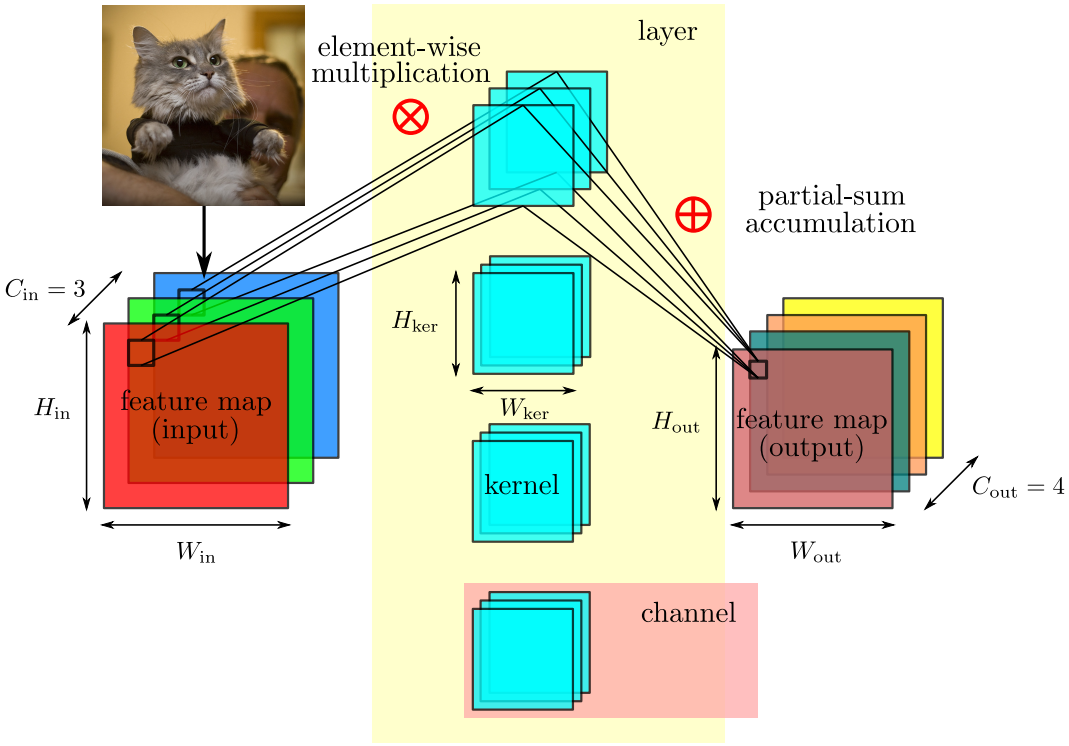


Figure 3: Expanded view of a typical 2D CONV layer inside a CNN.

While the expressive power of networks using only FC layers is impressive, it comes at the cost of a very large number of connections (and hence network parameters), making them hard to train and easily prone to overfitting. This is why other types of structured layers, with fewer parameters, but which are more efficient for certain tasks, have been explored. In the case of visual data, this has led to the development of CNNs, a staple of DL today.

The main elements that have led to the introduction of CNNs are Convolutional (CONV) layers, composed of high-dimensional convolutions that allow extraction of shift-invariant features from the input. An example is Figure 3, showing a traditional 2D CONV layer. In this context, the input activation is structured as a 3D set of input feature maps, with input width (W_{in}), input height (H_{in}) and input channel (C_{in}) dimensions. The weights of the layer are structured as a 3D filter, with kernel width (W_{ker}), kernel height (H_{ker}) and input channel (C_{in}) dimensions. For each input channel, the corresponding input feature map is transformed through a 2D convolution with the appropriate kernel in the filter. The convolution results at each point are summed across all the input channels to generate the output partial sums. The results of these partial sums comprise one output feature map with output width (W_{out}) and output height (H_{out}) dimensions. Several

2D filters can be stacked together to generate additional output channels, denoted with C_{out} in this case.

Depending on the size of 2D kernels and their count, the output feature maps can be large and deep, motivating the use of pooling (*i.e.,* subsampling) layers that reduce the scale of feature maps. Pooling is similar to convolution, with a kernel sliding over the input matrix, but instead of performing matrix multiplication, an aggregation operation function is applied. The most common such operations are taking the maximum element or the average. A visual example of a simple CNN mixing in all these layers is given in Figure 4.

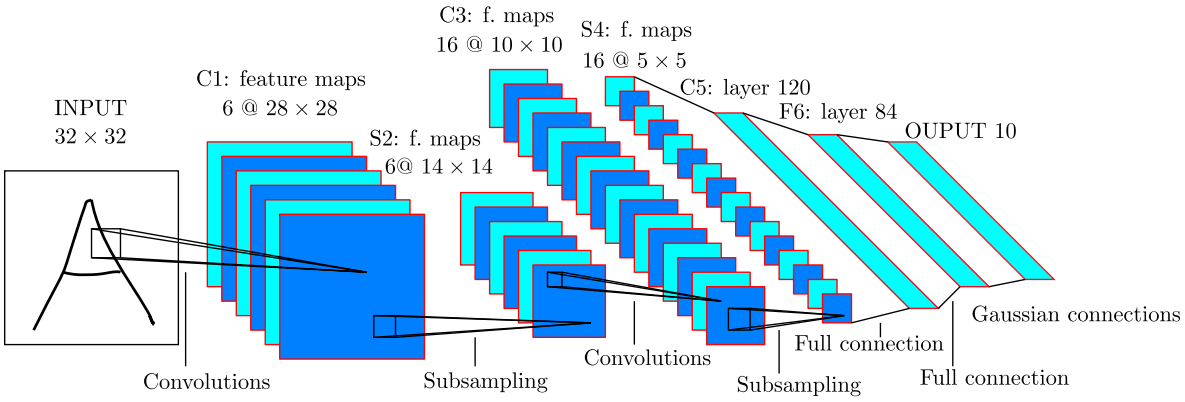


Figure 4: A visual representation of LeNet-5 (adapted from [LBBH98, Fig. 2]), an early example of a CNN that promoted the subsequent development of Deep Learning. It contains the main layers that are usually found in CNNs: convolutional, pooling and fully connected.

Another frequently used layer is Batch Normalization [IS15] (BN). It contains two trainable parameters that are used to re-center and re-scale the distribution of the values of a feature map, to improve training performance. While there are also more recent and complicated layers, such as depth-wise convolutions [HZC+17] or Inception modules [SLJ+14], their specifics are not important for the rest of this chapter.

2.4 Performance and Energy Profiles of Recent Models

To gauge the complexity of current DNN models, there is a need for a set of metrics that allow for a fair comparison between models. In this study, the metrics used are (1) the model accuracy over a validation dataset, (2) the total number of weights in the model, and (3) the number of FLOating-Point operations (FLOPs) necessary to carry out one complete inference. Accuracy is measured in terms of the frequently used top-1 and top-5 percentages (*i.e.,* the proportion of correct predictions on the labeled validation dataset and the probability that the correct result is among the top five predictions). The number of weights allows estimating the total memory storage requirements for the model, whereas the FLOP count hints at the required computing power needed to execute the model at a certain frequency.

Model Name	AlexNet [KSH12]	GoogLeNet [SLJ+14]	ResNet-50 [HZRS15]	MobileNet V2 [SHZ+18]	EfficientNet B1 [TL19]
Year	2012	2014	2016	2018	2019
Top-1 accuracy	57.2%	69.8%	76.2%	72.0%	79.1%
Top-5 accuracy	84.7%	93.3%	92.97%	90.6%	94.4%
Number of Weights	62M	6.4M	26M	3.5M	7.8M
FLOPs	1.5B	2B	4.1B	0.3B	0.7B

Table 1: Recent evolution of DNNs for image classification on the ImageNet dataset.

Table 1 shows a comparison using these metrics on some popular DNNs for image classification

on the ImageNet dataset (adapted from [STKV20]). For a long time, the only metric of interest was the network accuracy, resulting in models that were costly to train and operate. The cost of training and inference became so large at one point that there is now an open engineering consortium called MLCommons[3] that benchmarks DL models and fosters innovation in the field. Thus, there is an increasing interest for faster, lighter, and overall more efficient models that are compatible with edge device resource constraints and operate more efficiently in the cloud. The last two columns in Table 1 reflect this, with newer network models achieving competitive accuracy with less memory and a smaller FLOP count.

Some examples of the scale at which modern DNN training costs stand for recent NLP models are given in Table 2 (adapted from [SGM19, Table 3]) and showcase the significant resources needed for training state-of-the-art models.

Model	**Hardware**	**Power (W)**	**Hours**	**CO_2e (lbs)**
$\text{Transformer}_{\text{base}}$ [VSP+17]	P100×8	1415.78	12	26
$\text{Transformer}_{\text{big}}$ [VSP+17]	P100×8	1515.43	84	192
ELMo [PNI+18]	P100×3	517.66	336	262
$\text{BERT}_{\text{base}}$ [DCLT18]	V100×64	12041.51	79	1438
$\text{BERT}_{\text{base}}$ [DCLT18]	TPUv2×64	—	96	—
NAS [SLL19]	P100×8	1515.43	274.12	626.155
NAS [SLL19]	TPUv2×1	—	32.623	—
GTP-2 [RWC+19]	TPUv2×32	—	168	—

Table 2: Estimated cost of training recent NLP models in terms of power, time and CO_2 emissions.

The need for efficient DL computations coupled with the resilience of DNNs to approximation (due to the stochastic nature of training methods and a high level of inner redundancy [CCRR13]) has paved the way for the development of a large number of approximation methods, a part of which are described in the rest of this chapter.

3 Approximation for Inference

DNN inference is a very computation-intensive task, having large memory and computation power requirements. For example, inference on a single image using the original ResNet34 [HZRS16] model requires 3.6 billion FLOPs and storing 22 million weights plus temporary feature maps. While the execution of such tasks has moved from traditional CPUs having a latency-oriented design to more parallel hardware like GPUs or even custom ASICs / FPGAs, inference is still a costly task, and thus susceptible to benefit from performance improvements when using approximate computing. Consequently, this section describes AxC methods found in the literature that improve deep neural network inference performance.

One can distinguish three different classes of methods (see Figure 5), usable in isolation or combined, to approximate DNN inference. The first one, *structure refinement transformations*, includes the methods that modify the computational structure (*i.e.,* the network layers and their parameters) of the input model. Some notable examples include knowledge distillation [HVD15, TSZ+20] which uses the model as a teacher to help train smaller students models or compact architectures [HZC+17, IMA+16] where layers are transformed into more hardware friendly ones. The second class, *data-oriented refinement transformations*, focuses on optimizing the finite precision data representation(s) of the model while maintaining the initial computational structure intact. Notable examples are *pruning* [CDS90, FC18] (*i.e.,* setting less important parameters to zero to increase sparsity) and *quantization* [ZWN+16, ZYG+17] (*i.e.,* changing the types of the

[3] https://mlcommons.org/en/

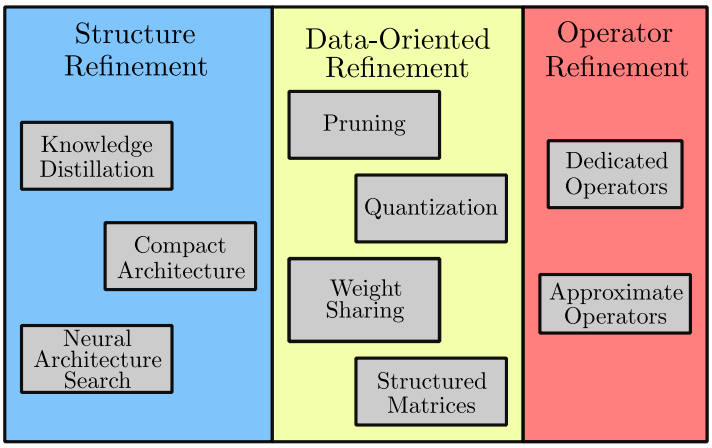


Figure 5: Different types of approximation techniques for DNN inference.

parameters and intermediate results to more efficient representations). While network structure refinement substantially changes the network structure, data type refinement does not, giving the possibility to emulate the inference of the approximated network in the original structure to measure its accuracy loss.

The third class of approaches relies on *operator refinement transformations* which modify the arithmetic operators used inside the CNN implementation (*e.g.,* addition and multiplication) to further improve energy efficiency. Such methods are not discussed any further here since they mainly depend on the hardware implementation of the CNN. More details on the approximate operators that fall in this class can be found in Chapters 3 and 4.

In the rest of this section, data-oriented refinement methods are covered. Specifically, Section 3.1 gives an overview of various quantization methods, Section 3.2 discusses weight sharing approaches, whereas pruning is analyzed in Section 3.3.

3.1 Quantization

Full precision DNNs usually rely on 32-bit floating-point values for representing parameters. For standard backpropagation-based training, using high precision weights makes sense since the gradient update rule generally modifies these weights by a small factor of the corresponding gradient terms. While full precision `float32` DNNs offer excellent result quality, they can generally be compressed and accelerated using lower precision arithmetic with minimal or no loss in the accuracy. Methods for addressing data quantization in DNNs are varied, ranging from simple binary and ternary networks to larger fixed-point and custom floating-point formats. This section gives an overview of the main ones.

Analysis of existing approaches relies on various aspects, such as (1) what parts of the network are being quantized, (2) homogeneity/heterogeneity of the number formats used inside the layers, (3) the type of representations being used, and (4) how and when is quantization performed (during or after the network has been trained).

What to quantize. The most obvious quantization targets are the *network parameters* (*e.g.,* weights and biases). Reducing the number of bits used to represent them primarily brings a memory footprint reduction for on-device storage of the network. Latency improvements are potentially achievable with binary, ternary and bit-shift (*i.e.,* power of two values) quantized parameters [CHS+16, LZL16, RORF16]. More generally, if faster execution times are to be obtained, *activation function inputs and outputs* also need to be quantized. An example is [JKC+18], which proposes an efficient 8-bit integer quantization scheme for both weights and activations. Additionally, one can quantize the weight and activation *gradients* used during backpropagation (see

for instance [ZWN+16, WLCS18]) to accelerate training, an aspect discussed in Section 4.

When and how to perform quantization. There are two established ways quantization can be performed for efficient inference and a third, emerging method.

The first among the established approaches is *Quantization-Aware Training* (QAT). The idea is to use a network parameter update procedure for several epochs (starting from scratch or after a baseline `float32` training method is run) to adjust parameters in the quantization format(s) such that generalization accuracy is hopefully kept the same or is at worst minimally degraded. Much research has focused on such fine-tuning methods (see for instance [RORF16, ZWN+16, ZYG+17, CWV+18, JKC+18, ZYYH18]), mainly because they achieve good results, especially for extremely low precision formats (*i.e.,* binary and ternary encodings).

While training is a powerful approach to compensate for a model's accuracy drop due to quantization, it is not always applicable in real-world scenarios (*e.g.,* for online learning) since it is costly, time-consuming and generally requires a full-size training dataset. This can be a problem when the data is proprietary, privacy and regulatory issues are in effect (*e.g.,* medical data that cannot be uploaded to the cloud for remote processing), or when using pre-trained off-the-shelf models for which data is no longer available. As such, there has been a push for faster *Tost-Training Quantization* (PTQ) methods without any fine-tuning. It has been observed that for down to 8-bit word lengths, PTQ results are close to full precision ones for several models [BNS19] (*e.g.,* AlexNet, VGG, and ResNet), but it becomes significantly more difficult to maintain accuracy when targeting lower precision formats. Work focused on PQT includes [BNS19, CYD+20, CKYK19, NBBW19, ZHD+19].

A possible issue with QAT and PQT methods is that both generate networks that are *sensitive* to how quantization is carried out (*e.g.,* the target word length). As such, there has been recent work [ABvB+20, SCB+20] on methods for *robust quantization* that provide intrinsic tolerance of the model to a large family of quantization formats and policies by directly specifying it in the training loss function. Such approaches are interesting for battery-powered edge devices, where depending on the state of charge, a network model capable of operating effectively at various quantization levels would be highly beneficial.

Granularity of applying a quantization format. Initially, quantization approaches were homogeneous, with one word length being used for the entire network. This is the case for early works on binary [CBD15] and ternary [LZL16] weight networks, for instance. Such approaches can suffer from significant accuracy loss since different layers tend to have different sensitivities to quantization levels/noise. Subsequent work has focused more on a heterogeneous, layer-wise optimization of the quantization format [ZMDCF17, WWZ+18, WLL+19, DYG+19, DYC+19, CYD+20].

There have been various metrics proposed to estimate the overall effect of a fixed-point quantization format inside a layer on the overall accuracy of the network. One example is [LTA16], which uses a Signal to Quantization Noise Ratio (SQNR) to empirically measure how suitable a fixed-point format is. The approach in [ZMDCF17] generalizes the work from [LTA16] using an adversarial noise to formulate the quantization error. Another adaptive quantization method is [KL18], which uses the loss function gradient to determine an error margin for each parameter such as to not degrade accuracy and assign a precision accordingly. Recent work [DYG+19, DYC+19, SDY+20] also proposes using second-order information (Hessian-based) to gauge the sensitivity of each layer. From an Information Theory perspective, [ZZL18] uses the entropy of weights and activations as a saliency indicator to set fixed-point quantization levels at each layer. Another popular statistical sensitivity measure is based on the Kullback-Leibler divergence, which is used to measure layer sensitivity in [WLL+19, CYD+20] and is a core component for fine-tuning low precision integer weights in NVIDIA's TensorRT inference acceleration library.

On a different granularity level, [PYV18] proposes looking at the distribution of weight values over the entire network to aggressively quantize weights in dense regions and more gently those in sparse ones. Compared to `float32` baselines, such an approach can achieve under 1% accuracy loss for large networks (ResNet-152 & DenseNet-101) with a 4-bit format in the dense areas and a 16 bit one for the sparse regions (< 1% of parameters).

Quantization formats. There have been various representations used to quantize deep neural networks. At the extreme, there are *Binary Neural Networks* (BNNs), where weights and activations are stored with one of two possible values. If a $\{0, 1\}$ (or equivalently a $\{-1, +1\}$) encoding is used, then multiplications can be implemented efficiently using XNOR gates, making BNNs compelling on FPGA and ASIC targets, but also for emerging computing paradigms such as neuromorphic [EMA+16] or in-memory computing [SLPY18].

Among the first investigations of binary networks is BinaryConnect [CBD15], which maintains a full precision copy of the weights to be updated during backpropagation, but are binarized for inference. Activations are kept in full precision, meaning full precision accumulations are still required during the forward propagation. The effect of binary activations is considered in [RORF16, CHS+16, HCS+16]. These early papers are the basis for most subsequent research on BNNs.

The XNOR-Net approach [RORF16] expands on the initial BNN ideas by proposing a model where a gain term is added to the network at the level of each dot product in the convolutional layers. Computed from statistics of weights and activations before binarization, the gain was a way to improve the accuracy of BNNs on the ImageNet dataset. Such gain terms are nevertheless costly to compute in practice, which is why later work modified their use. For instance, [ZWN+16] proposes gain terms that are only based on the non-binarized weights of the network, meaning that they never need to be recomputed after training. Additionally, [THW17] also advocates binarizing fully connected layers by adding neuron-specific scaling factors, further improving compression without a drastic decrease in the accuracy. A generalization of the BNN concept to multiple binary bases used for quantizing weights and activations is presented in [LZP17], further reducing the accuracy gap between full precision and binary architectures, at the expense of a higher computational cost (compared to previous BNN methods). Changes to the backpropagation process in BNN training [DBCN18] can also be effective for limiting accuracy loss.

Ternary neural networks offer a better representation of the (pseudo) normal distribution of weights that is frequently observed after training. For instance, [HS14] achieved good results on small networks with weights quantized to $\{-1, 0, +1\}$ and 3-bit fixed-point activations. For greater flexibility, [LZL16] proposes using a threshold α for picking the ternary weights (-1 if $w < -\alpha$, 0 if $|w| < \alpha$ and $+1$ if $w \geqslant \alpha$), while keeping activations in full precision. This is further expanded in [ZHMD16], which uses ternary weights from a set $\{-\alpha^n, 0, +\alpha^p\}$, where α^n and α^p are learnable parameters. By also quantizing activations to 8-bits and adding residual edges to branches in the architecture that are sensitive to quantization, [KBM+17] offers comparable accuracy results to `float32` for a ResNet-101 model on the ImageNet dataset, with no additional low-precision (re)training. In a more aggressive compression strategy, [WSL+18] proposes the use of ternary activations $\{-1, 0, +1\}$ and binary scalable weights $\{-\alpha, +\alpha\}$.

Extremely low-bit width networks like the ones just presented are susceptible to non-negligible accuracy loss, which is why there has been work focusing on non-binary *integer* and *fixed-point*-based quantization. Among the early proponents of integer quantization, there is [ZWN+16], which extends the idea of BNNs to arbitrary word lengths for weights, activations, and gradients. For fixed-point arithmetic, [LTA16] explored the use of various bit width combinations (4, 8 and 16 bits) of weights and activations. Notable results with integer arithmetic are presented in [JKC+18], which showcases how 8-bit integer quantization on ARM CPUs can achieve near-identical accuracy compared to baseline `float32` models based on MobileNet architectures for classification and detection tasks, but with improved on-device latency. Good quantization results with 4-bit weights and activations are presented in [BNS19] by combining three complementary methods for minimizing quantization error at the tensor level. Heterogeneous/mixed-precision quantization approaches also heavily focus on integer/fixed-point formats [WWZ+18, WLL+19, DYG+19, DYC+19, CYD+20].

One problem with low precision integer/fixed-point formats is that they have limited dynamic range, which might make them inappropriate, especially for networks used in Natural Language Processing (NLP) tasks, where weights tend to have values that are more than 10× larger than the largest magnitude values found in popular CNNs [TYW+20, Fig. 1]. While not that widespread,

there has been some work looking into low precision *floating-point* quantization for CNN inference. For instance, [SBD+18] explores the use of up to 8-bit (scaled) floating-point formats for weight and activation quantization in classification networks such as GoogLeNet, ResNet, and MobileNet, without any accuracy degradation. More recently, [WWC+20, WWL+20] show how an 8-bit floating-point quantization format (4-bit mantissa and 3-bit exponent) can be used in FPGA-based accelerators for deep CNN inference, without any retraining. Another approach [TYW+20] consists of an *adaptive* floating-point quantization method, where the exponent range of quantized values is dynamically shifted at each network layer (through changing the bias term of the exponent), yielding competitive results on NLP networks and tasks.

At a coarser level, it is also possible to improve dynamic range by sharing the exponent between parameters, storing only the mantissa and one copy of the exponent. This is the so-called *Block Floating-Point* (BFP) format. For instance, [SLW17] propose using BFP with an 8-bit mantissa for weight storage, showing negligible to no accuracy loss on CNN workloads (VGG16, ResNet-18, ResNet-50, and GoogLeNet-based networks). On the FPGA side of things, [LLS+19] showcases a BFP-based CNN accelerator design that uses 16-bit activations and 8-bit weights, reducing memory requirements compared to a `float32` baseline without any retraining/fine-tuning. Another way to increase the dynamic range is to employ a *logarithmic representation*, which also allows multiplications to be replaced with simple binary shift operations. For instance, [MLM16] shows that a log representation can achieve higher classification accuracy than fixed-point formats operating at the same word length. 8-bit log floating-point quantization was also shown [Joh18] to perform close to baseline `float32` values with several CNN classification networks.

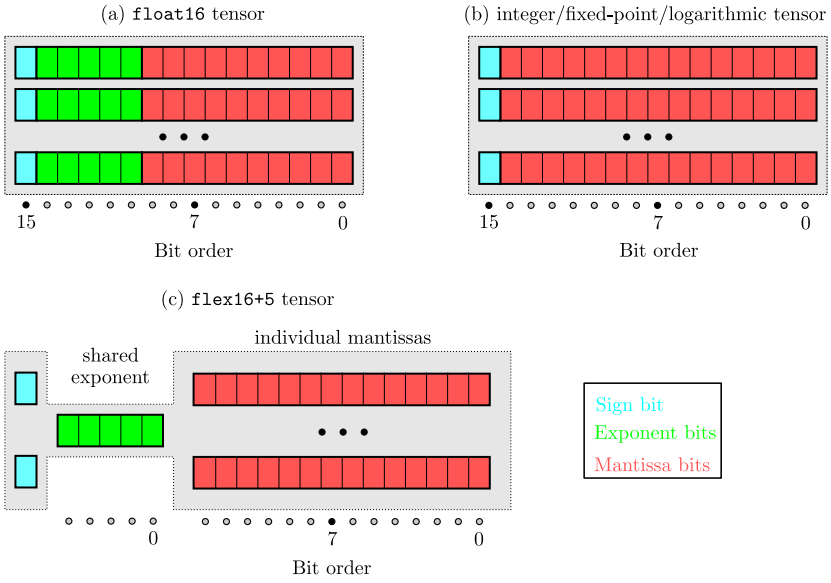


Figure 6: Diagrams for bit representations of various numerical formats discussed in the context of DL quantization in this chapter. Red, green and blue shading are used to represent mantissa (M), exponent (E), and sign (S) bits respectively. In (a), the 16-bit IEEE 754 `float16` *floating-point* format is shown (corresponding to $(-1)^S \times 2^{E-15} \times 1.M_2$ for normalized values), with 1 sign bit, 5 exponent bits and 10 mantissa bits. (b) illustrates a 16-bit *signed integer* format. By choosing a *fixed* splitting point for integer (I) and fractional (F) parts in the mantissa ($M := I.F$), it can also serve as a representation for a *fixed-point* format (namely to $(-1)^S \times I_2.F_2$). Additionally, (b) can represent a form of *logarithmic number system* (see for instance [FML10]), with the encoded value being $(-1)^S \times 2^M = (-1)^S \times 2^{I.F}$. Part (c) exemplifies a *block floating-point* format, namely the `flex16+5` format [KWW+17] with a 15-bit mantissa and 5-bit shared exponent.

A summary of these aforementioned formats (minus the binary and ternary encoding that generally require just 1 or 2 bits to represent) is given in Figure 6.

Looking at the *value distribution* of the data (weights and activations) is a good way to explore what number formats and/or encodings are better suited for a particular network model. Uniform precision was the go-to alternative for a long time, but more recent work is concentrated around non-uniform quantization. This is because the actual distributions of trained weights tend to follow bell-shaped curves. In this direction, [ZWW+17] focuses on balancing the quantization values based on the distribution of the data. The quantizer can also be trained alongside the model [ZYYH18, JSL+19] and it is also possible to use reinforcement learning [WLL+19] and meta learning [WWXX20] approaches to determine good choices for the quantizer.

Choosing quantized values. There are various methods for quantizing data, ranging from simple heuristics like those used to convert network weights into binary values depending on their sign [CBD15] or projecting real-valued parameters to (one of) the closest discrete points [JKC+18], to loss functions that regularize the network and force parameters into quantized states upon the convergence of the training algorithm [CEKL20].

One notable approach is [ZYG+17], which incrementally quantizes network weights to power of two terms. The set of non-quantized weights is progressively shrunk during re-training, with their values being updated to counter any accuracy loss induced by quantization. Knowledge distillation can also be a valid way to pick quantization values [BWKL19, PPA18].

It is also possible to cast this task as a mathematical optimization problem. For instance, [LTA16] converts pre-trained weights to fixed-point values by looking at their signal-to-noise ratio as an optimization metric. In [WWC+20], the mean square error of the quantized data with respect to the original data is used to choose the precise 8-bit floating-point quantization format (mantissa and exponent size) and corresponding values. In more involved approaches, the Alternating Direction Method of Multipliers (ADMM) can be used to optimize the quantized values with low precision formats [CWP19, LLZJ17]. Regularization terms and parameters that emphasize quantized solutions are also available. The work of [CEKL20] looks at using mean squared quantization error regularization to drive weights to quantized values and how ℓ_2 regularization can lead to sparse weight designs. Regularization also is an effective approach for doing robust quantization [ABvB+20, SCB+20].

3.2 Weight Sharing

Weight sharing compresses the network by assigning shared values to parameters. This transforms plain weight data storage into a reduced number of shared values in a dedicated memory, together with the indices of these values in the weight matrix.

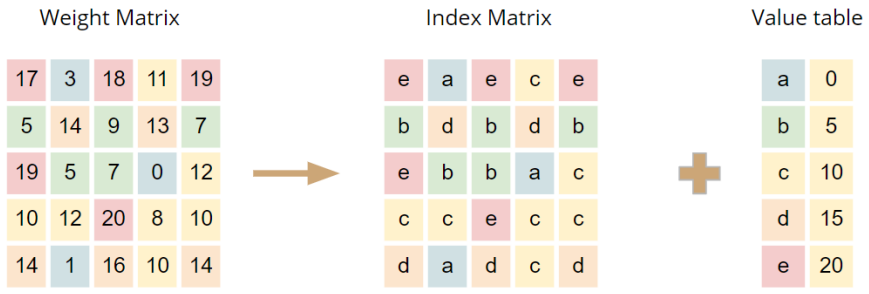


Figure 7: Weight sharing techniques allow network compression by storing indices instead of values.

Figure 7 shows an example. The first matrix corresponds to a 5×5 convolutional kernel (filter) with values computed during training. The matrix contains $N = 25$ values ranging from 0 to 20. Each value can be represented using $B = 5$ bits, resulting in a total size of $N \cdot B = 25 \cdot 5 = 125$

bits. There are 5 shared values, namely ‘a’, ‘b’, ‘c’, ‘d’ and ‘e’, replacing the 25 original values, as shown in the second matrix.

Accordingly, the size of an element in the weight matrix can be reduced from B to $\log_2(K)$ bits, with K being the number of different shared values. The size of the stored data then becomes $N \cdot \log_2(K) + K \cdot B$, instead of $N \cdot B$.

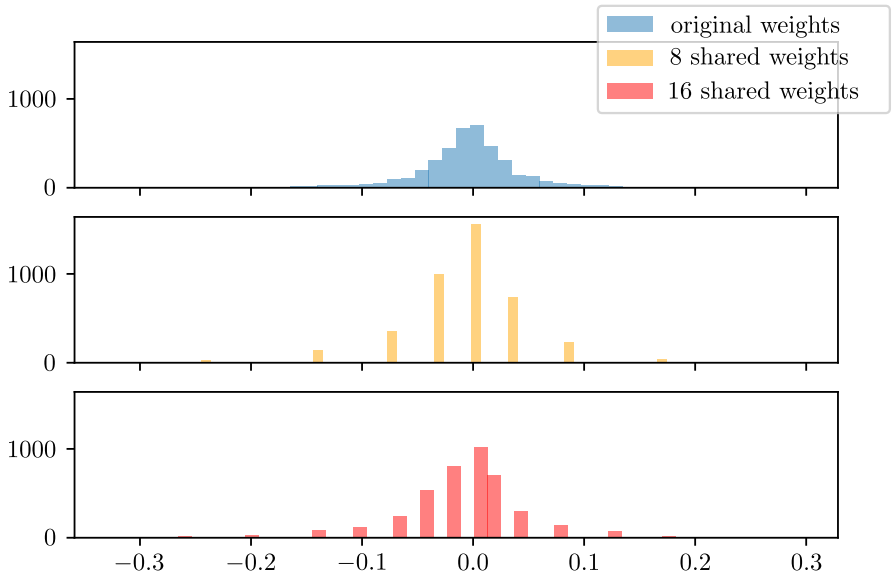


Figure 8: Distribution of the weights composing the first layer of a trained ResNet50V2 [HZRS15], original (top), with only 8 (middle) and 16 (bottom) shared values.

Depending on the number of shared values used, the distribution of the weights inside a layer will change. An example of this before and after weight sharing (with 8 and 16 shared values) can be found in Fig. 8.

Weight sharing approaches can be classified by the method used to group weights together and by the granularity level it is applied at. Each of these aspects will be explained in some detail in the following paragraphs.

Grouping methods. One of the first approaches involving weight sharing that showed it can be a viable option for compressing neural networks is HashedNets [CWT+15]. The weights of the network in this setting are randomly grouped into hash buckets sharing the same value. These shared values are then trained and updated using backpropagation. The authors test their approach on the MNIST dataset with two custom fully connected networks with 3 and 5 layers.

However, instead of applying random grouping before the network even sees any data, it is also possible to approximate an already-trained network by determining groups based on weight values. In this vein, DeepCompression [SH16] uses the K-means algorithm to iteratively group the weights in a network in a global 3-step compression approach involving network pruning, weight sharing, and parameter encoding. The K-means algorithm is used to cluster similar values together, followed by an iterative retraining phase. Different initialization options for the shared values are considered, with experiments showing that uniform initialization over the entire range of weight values works best. Applied to the AlexNet and VGG architectures on the ImageNet dataset, the compression algorithm achieves 35× and 49× compression, respectively, with negligible accuracy loss.

The most common way of doing K-means clustering is through the Lloyd algorithm [Llo82], which uses mean square error minimization to solve the clustering. However, this clustering approach does not imply performance loss minimization when taking into account quantization as well. The use of mean square error minimization does not necessarily lead to high accuracy

during inference, even when uniform initialization of the clusters is used, as suggested with DeepCompression [SH16]. Because of this, [CEL16] proposes to use Hessian-weighted K-means clustering to minimize accuracy loss. The approach consists of replacing the mean square error with the distortion of the Hessian matrix (second-order derivative) of the loss function. With this change, it can achieve a higher compression rate than DeepCompression, but with similar accuracy loss.

It is also possible to consider weight distribution when performing clustering. For instance, [PAY17] proposes a clustering method based on weight entropy, using importance (magnitude) and frequency of the weights to group them. Thus, frequent non-zero (low importance) values are grouped, as well as rarer, but higher magnitude (high importance) values.

During the iterative process of training weights, clustering them, and training them again, previously clustered weights will sometimes diverge from the shared values at retraining time, making convergence to a good network model difficult. This is why, rather than applying iterative clustering and retraining, [WWW+18] proposes the Deep-K-means approach that adds a regularization term in the training objective function, enforcing weights to stay clustered during training. After training is finished, the K-means algorithm is used to group the obtained weight values.

Other clustering algorithms can also be used. One main issue with using the K-means algorithm in this context is that it targets multi-dimensional data, whereas weights clustering is a 1-D problem. One example of approach using another clustering algorithm is DP-Net [YYZ+20], which is based on a dynamic programming clustering algorithm that enables weight sharing in constant time, reducing the clustering complexity compared to the K-means algorithm.

Weight-sharing granularity. The weights of a network can be shared at different levels of granularity, as shown in Figure 9. While this can be done for the entire network, as initially proposed in HashedNets [CWT+15], each layer has a different weight distribution, covering a different range. Hence, sharing values for the whole network usually does not offer good enough representation power to limit accuracy loss.

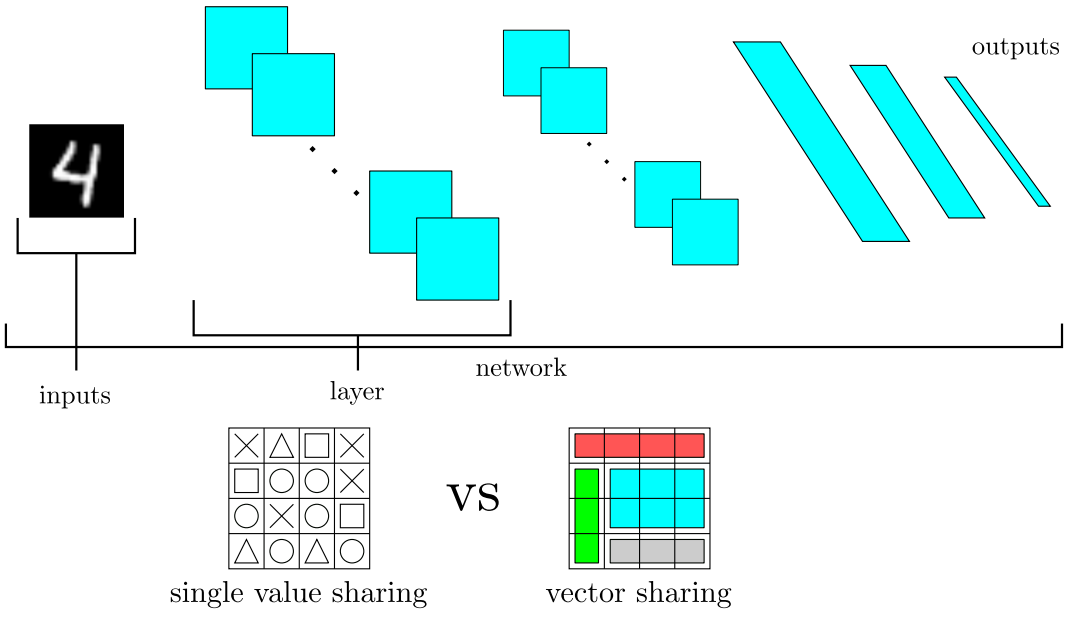


Figure 9: The various scopes of applying weight sharing.

On the other hand, sharing the values at the layer level offers a better representation of the original network, as shown with Deep Compression [SH16]. Such an approach also allows different levels of compression to be used for each layer. The first and last layers are generally more sensitive to compression and require a higher number of shared values to keep accuracy

loss acceptable. It is even possible to target a smaller scope, like sharing values at the level of a (convolutional) kernel – but then of course the compression rate will be much lower.

While reducing the scope allows a better representation of the initial weight distribution, thus keeping accuracy loss low, it is possible to improve compression performance. For example, Deep-K-means [WWW+18] shares values at a level that is optimal for the very efficient row-stationary dataflow used in DNN hardware accelerators.

Even if weight sharing leads to good compression rates, it does not enable inference acceleration by itself. This can be achieved if inputs are also discretized, reducing the number of combination operations and allowing the use of a pre-computed look-up table multiplier. This approach is used in LookNN [RIKR17], which applies K-means to the input feature map to achieve a nonlinear quantization whereas the remaining feature maps are quantized in the traditional linear way.

Values can also be shared at a smaller level, as in Q-CNN [WLW+15]. Here, layers are decomposed into sub-vectors, which are then clustered using the K-means algorithm. Sharing vectors like this reduces the number of possibilities when performing products. This enables the layer response to be approximated using product pre-computation with a look-up table.

3.3 Network Sparsification (Pruning)

DNNs tend to be more complex as their accuracy rate improves and this complexity usually carries with it the fact that the network is over-parameterized. On the other hand, it has been argued for a long time [CDS90] that structure is more important than density in neural networks, with sparse models having the ability to generalize up to as well as their dense counterparts. Removing model parameters has the direct effect of reducing the size of the model, but it can also be used for speeding up the inference process by reducing the number of computations. Depending on the objective, different parts of the network can be more interesting to prune than others. For instance, fully connected layers usually concentrate most of the network weights in a CNN and should be targeted for high compression. Convolutional layers, however, contain fewer model parameters but account for most of the computations. Since they generate the majority of data movement in the model, they should be targeted when model performance and energy efficiency are important.

Pruning methods can be classified by how they are applied to the network, the granularity of the pruning, and finally the saliency determination approach. All these criteria are discussed in the following paragraphs.

Target regions. The loss in accuracy incurred by removing parameters can be recovered by re-training the remaining parameters using the initial training dataset if it is still available. This pruning process can be performed at different steps of the network life-cycle, either prior, during, or after training the model.

It has been shown that some parts of DNNs are more resilient to approximation than others. As such, pruning each layer at the same rate is not very efficient for accuracy. But at the same time, choosing the optimal sparsity level for the whole network is a complicated task. For example, [HH18] proposes to heuristically optimize the pruning ratio of each layer using reinforcement learning.

Similar to pruning weights, feature maps can also be pruned during the forward pass of the network. This process is called *dynamic sparsity* and is used in many accelerators to avoid zero or near zero computations [HQY+16, HQY+17]. Such approaches require dedicated architectures, but since the focus is only on data type refinement methods for this survey, they will not be discussed further.

Pruning Granularity. Depending on the pruning objective (compression or performance), one can choose to focus on weight removal at various sparsity levels. For instance, even though removing an entire structure (*e.g.,* a convolution kernel) allows reducing the computational complexity of the model, and thus, improving performance, it also has the effect of inducing a higher accuracy loss.

The lowest pruning level is at the weight level, the goal being that of removing the individual parameters with the lowest saliency [CDS90, SH16]. Although this generally results in the lowest

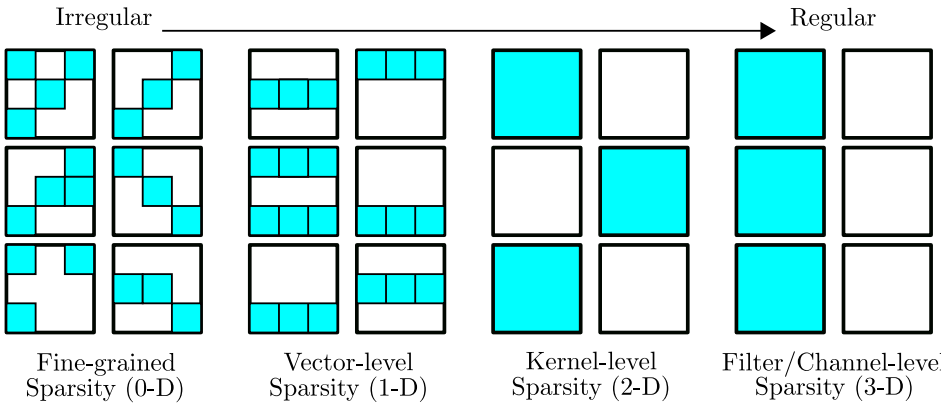


Figure 10: Different granularities of pruning in a 4-dimensional weight tensor for DNN inference (adapted from [MHP+17, Figure 1]).

accuracy loss, it does not systematically offer latency or energy improvements because sparse tensor computations are quite difficult to accelerate. Its main purpose is therefore to compress the network in memory.

To accelerate computations, a regular sparsity pattern is usually required. This is called *structured pruning* and aims at removing (spatially close) groups of weights so that network inference can be simplified. To achieve this, [JLD+18] iteratively reorders pruned weights to prune larger structures, whereas [YLP+17] uses different pruning strategies depending on the hardware, optimizing for the full utilization of available SIMD units.

As previously hinted, it is also possible to remove convolution kernels, thus simplifying the processing of pruned convolutional kernels. An example is [MTK+16], which progressively removes convolutional kernels through greedy-based fine-tuning. The method is applied to transfer learning applications, resulting in a 2× speedup on ImageNet-class CNNs.

Another interesting structure amenable for removal is a channel. Once channels are removed, one can remove the corresponding filters that take these channels as input. The filters producing these channels in the previous layer can also be removed [LWL17]. A representative approach is [HZS17] which removes channels based on importance, resulting in a $2-5\times$ speedup on multiple ImageNet-class CNNs with under 1% accuracy loss. In subsequent work, [HH18] proposes to pick the pruning ratio of each layer using reinforcement learning.

Weight Saliency Determination. Removing part(s) of the network usually requires knowing which regions are least important for ensuring network accuracy. This is called *saliency determination* and it can be conducted using different methods, as described next. A simple way is to use heuristics like weight magnitude or examining the ℓ_1/ℓ_2 norm of a group of weights, whereas more recent work employs optimization algorithms to address the trade-offs between accuracy loss and compression/acceleration.

The earliest methods removed small magnitude weights because they tend to have the least impact on accuracy [CDS90, HS92]. They work iteratively by fine-tuning unpruned weights to recover lost accuracy [SH16]. It has been shown recently that one can also remove redundant connections in FC layers since for weights having the same value, only one needs to be kept [SB15]. If accuracy is degraded too much during the pruning process, some methods can be used post-pruning to restore certain weights and improve accuracy [GYC16, NDSE17]. For convolutional layer filter removal, it is possible to rank filters based on their ℓ_1 norm and prune the lowest ranking filters of each layer [LKD+17]. Instead of ranking filters at the layer level, one can also do it at a global, network-wide level by first doing a layer-wise filter ordering using ℓ_2 norms and then computing affine mappings that enable inter-layer filter rankings [CDZM20, CDZM19]. Such global approaches lead to a Pareto set of approximated networks that offer various trade-offs

between performance and accuracy.

In [MTK+16], the authors also consider a Taylor expansion criterion that approximates accuracy degradation due to feature map removal. This is done using activation and gradient values already computed during a regular training iteration. Other approaches use weight gradients to compute saliency. For instance, [DYJ17] proposes a sequential two-step process where (1) gradient-based information is used to grow the network (adding 'dormant' connections and neurons that are deemed important for accuracy) and (2) regular magnitude-based pruning of weights and connections.

Another method to identify representative structures inside a network is [HZS17], which uses a two-step process involving Least Absolute Shrinkage and Selection Operator (LASSO) regression for channel selection and then a least squares-based reconstruction approach of subsequent feature maps in the network.

It is also possible to state the problem of selecting which parts of the network to remove as an optimization problem. One example is [LWL17], which relies on the correlation between feature maps of the current layer and the next one to determine the importance of filters. In another approach [YCS17], the optimization problem features the model's energy efficiency as an objective. It is based on an energy estimation methodology capable of approximating both the power of MAC operations and data access (which is more complicated to compute, depending on the data reuse technique). The resulting iterative process involves local fine-tuning to recover accuracy loss in a layer before moving on to subsequent layers.

By formulating weight pruning as a non-convex optimization problem, it is possible to address it using an ADMM approach [ZYZ+18]. Using the desired sparsity level as a constraint to be satisfied and the loss of the network as the objective to minimize, ADMM can be used in a two-step process. Since convergence can be quite slow, the target error is increased to accelerate convergence and the resulting accuracy loss is compensated by network retraining. The method can also be extended to address high sparsity target problems, by introducing a more progressive algorithm using partial weight pruning with a moderate pruning rate [YZZ+18].

Another idea is to encourage weights to group around zero using regularization. The closer weights are to zero, the less accuracy loss will be induced by removing them. For example, [LL16, WWW+16] used group LASSO [YL06] regularization to obtain structured sparsity, with the same factor being applied to all the weight groups. In [LLS+17], ℓ_1 regularization is applied to the scaling factor of batch normalization layers to identify important channels. Different regularization factors can be assigned to different groups, such as in [DDHT18], where ℓ_2 regularization is used to transfer the model's representational capacity to a fraction of its filters. An incremental approach for choosing these factors can also be used [WZWH19]. In [LW20], feature map channels are gradually zeroed during training using a dynamic regularization factor (whose value depends on the current compression ratio in the network), allowing safe removal of corresponding filters without a significant drop in the accuracy.

Another recent approach to optimize pruning is through architecture search. Usually, pruning methods target a fully trained network and recover any accuracy loss using fine-tuning because it is hard to train a sparse network. Recently, however, the idea that a classic network contains sub-networks that, trained from scratch, can perform as well as the original network but with fewer parameters and computation, was introduced [FC18]. This idea was also explored in [LSZ+19], which claims that directly training (using some form of random initialization) a model found at the end of a classic three-step pruning process (training, pruning, and fine-tuning) can perform as well, if not better, in fewer training steps. The issue is that, in the beginning, none of these studies provided a method for finding an efficient smaller architecture without doing full model training beforehand. This is starting to change, with [LJxZ+20] proposing to use a bee colony exploration algorithm to find an appropriate DNN pruning scheme. It is also possible to reduce the fine-tuning cost by using an external network trained to predict weights of a certain network structure, facilitating a fast exploration of various possible architectures [LMZ+19].

4 Approximation for Training

The state-of-the-art models used in deep learning applications require a considerable hardware infrastructure to be designed properly. There are various challenges related to computing, storage, network/communication, as well as memory capacity and bandwidth that can potentially hinder the scalability of current solutions to future models and applications. This is most visible during the training part of neural network design, which accounts for the majority of the computing time and resources.

Accelerating training at the arithmetic level has thus become a hot research topic, but early work in this direction did not necessarily translate to a wide adoption and availability of low/mixed-precision training hardware. For example, BinaryConnect [CBD15] introduced a CNN training methodology with binary (+1 and −1) weights, with all other operations and data structures (*e.g.,* tensors) in full `float32` precision. This binarization was soon extended to include activations [HCS+16], followed by experiments with quantization levels of 2, 4 and 6 bits for weights and activations [HCS+17], but with backpropagation gradients still computed and stored in full precision. Binarization for all tensor operations, including gradient computations, is considered in XNOR-Net [RORF16]. While ensuring impressive efficiency gains, these approaches lead to non-trivial accuracy loss for larger CNN models that have since been introduced and adopted in practice.

To manage accuracy loss, DoReFaNet [ZWN+16] uses different quantization bit widths for weights, activations, and gradients, but still incurs some accuracy loss and requires exploring different bit width configurations on a per-network basis, which can be impractical for large models. The approach introduced in [MNCM17] improves on previous accuracy results by doubling or tripling the number of inputs and outputs of layers in popular CNN models, but again requires that gradients be computed and stored in full precision and does not achieve the same accuracy as the baseline non-quantized trained model.

Studies with fixed-point arithmetic on DNNs have also been conducted since the early 1990s [HB91, PH94, SG94, SMA07] and more recently [GAGN15] has shown that a 16-bit fixed-point representation coupled with stochastic rounding can be used to train CNNs on the MNIST and CIFAR-10 datasets without accuracy loss. Nevertheless, it is unlikely that this approach would work on larger CNNs trained on larger datasets.

There have also been several proposals for quantizing recurrent neural network (RNN) training. For instance, in [HWZ+16], training for quantized versions of gated recurrent units and long short-term memory cells with few bits for weights and activations are investigated, with a slight loss in accuracy with respect to base full precision models. A different approach [OLZ+16] evaluates binary, ternary and exponential quantization for weights used in various RNN models trained for speech recognition and language modeling. Similar to the CNN-centered methods evoked so far however, all these approaches use full precision gradients, and therefore do not improve computation cost during backpropagation.

4.1 Mixed Precision Training approaches

The most widespread approach to increase performance and efficiency of DNN training at the arithmetic level is through the use of *mixed precision*.

On the commercial side, NVIDIA has offered the possibility to do low precision training since the Pascal architecture in 2016 and mixed precision training (combining `float16` and `float32` arithmetic) has really taken off with the subsequent introduction of TensorCore units in their Volta and Turing architectures in 2017–2018. TensorCores are, in essence, programmable $4 \times 4 \times 4$ matrix-multiply-and-accumulate units (performing the operation $D = A \times B + C$, where A, B, C and D are 4×4 matrices, with A and B stored using `float16` and C and D being either `float16` or `float32` matrices). An execution of a large number of such units provides a huge performance boost (several times when compared to NVIDIA's previous Pascal hardware) to convolution and matrix operations with mixed precision operands and results. Over at Google, their newer (from

version V2 onward) Tensor Processing Units (TPUs) offer similar support for mixed precision training with the introduction of `bfloat16`, a 16-bit floating-point format that, when compared to `float16`, trades in mantissa bits for exponent bits (a 5-bit exponent and 10-bit mantissa for `float16` versus an 8-bit exponent and 7-bit mantissa for `bfloat16`). Intel and ARM are also adopting `bfloat16` in their push to offer AI-enhanced hardware, while AMD has introduced software support for `bfloat16` in recent versions of their ROCm platform. As of May 2020, the Ampere architecture from NVIDIA also introduces `bfloat16` operator support in their third version of TensorCore units.

4.1.1 Mixed 16-32-bit precision training

An important remark about backpropagation training that should guide the choice of number formats is how the values contained within various quantities (activations, gradients, and parameters) vary during successive training iterations. It is noted in [CBD14] that "activations, gradients and parameters have very different ranges", whereas "gradient ranges slowly diminish during training". There is also the idea that a higher numerical precision should be used when updating the parameters than when using them during the back and forward propagation operations [CBD14, Sec. 6]. Recent accelerated training approaches (at the arithmetic level) follow these observations.

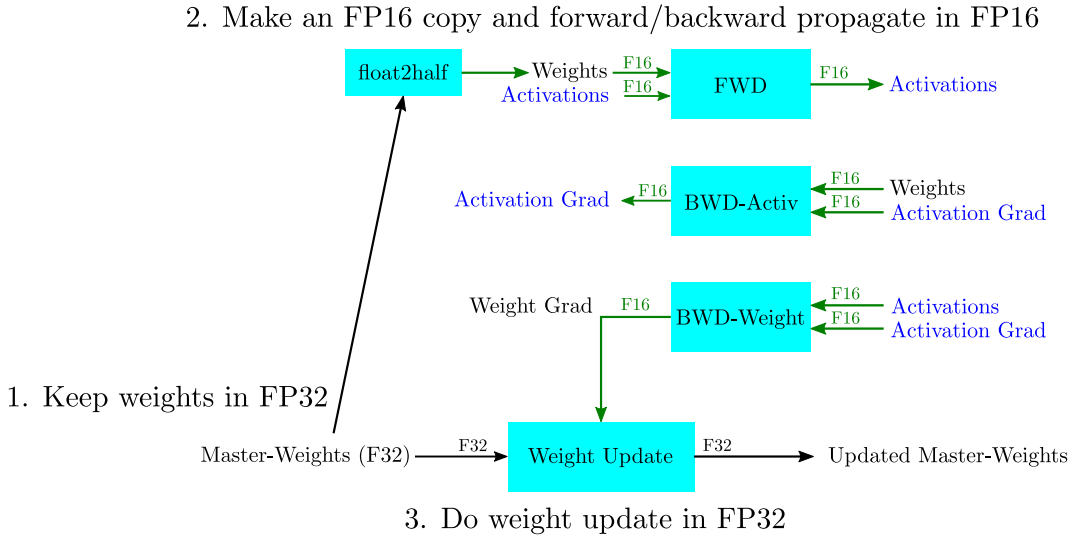


Figure 11: Mixed precision training iteration for a network layer (adapted from [MNA+17, Fig. 1]).

An approach for `float16`-based training acceleration. In [MNA+17], NVIDIA TensorCores are used to perform mixed `float16` and `float32` operations during each training iteration. The process is illustrated in Figure 11: a full precision copy of the weights is always stored and updated at each iteration, whereas the gradient computations of the weights and activations are done using `float16` quantizations of the weights. The dot-product and reduction (*i.e.,* sums of elements across a vector) operations are performed with a `float32` accumulator (as is enabled by TensorCores), which, according to [MNA+17], is needed in some cases to maintain the same model accuracy as with a baseline `float32` approach.

The main reason for using 32-bit values for the weight updates is that during later iterations of training, the update gradients become too small to be used with `float16` addition, which will result in them getting clipped when $\mathbf{w}^t \gg \varepsilon \frac{\partial \ell}{\partial \mathbf{w}^t}$ and adversely affect the final model accuracy. For `float16`, this happens when the ratio between weight and update is larger than 2048.

A related issue when gradients become too small is that they might not be accurately representable in `float16`, even though the dynamic range of the weight/activation gradients at each

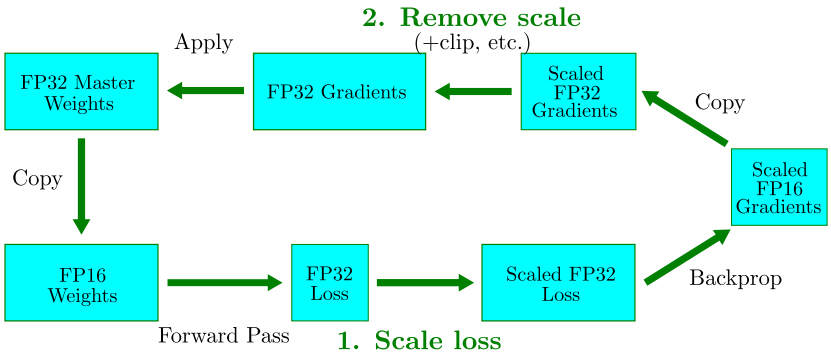


Figure 12: The loss scaling procedure for updating the master weights in mixed precision training.

layer is much smaller than the 2^{40} range associated with `float16`. This means that a scaling approach might be applicable. This is indeed what is advocated in [MNA+17], where gradient values can be shifted to `float16`-representable ranges by scaling the loss value computed during the forward pass, before performing backpropagation. By chain rule calculus during backpropagation, all gradient values will then be scaled by the same amount. Weight gradients will have to be unscaled back before weight update to ensure the same update process as with `float32` training. The entire procedure is summarized in Figure 12. Although not explored in [MNA+17], the scaling factor can be chosen automatically: start with a very large scaling factor (*e.g.,* 2^{24}), if gradient overflows (with `Inf` or `NaN`) decrease the scale by a factor of 2 and skip the current update, whereas if no overflow has occurred for some time (*e.g.,* 2000 iterations), increase the scale by a factor of 2.

The results presented in [MNA+17, Sec. 4] show that mixed-precision training is a viable alternative (in the sense that it gives comparable results to baseline `float32` training) for various tasks such as image classification (with tests on AlexNet, VGG-D, GoogLeNet (Inception v1), Inception v2 & v3 and ResNet50), object detection, speech recognition, machine translation, language modeling and Generative Adversarial Networks (GAN) generation.

In addition to the speed benefit that such a mixed-precision training approach brings (which varies from 2× to 6× with respect to baseline training on the experiments carried out in [MNA+17] on a Volta GPU), the memory consumption for training is roughly halved, since the dominating quantities are the activations (due to larger batch sizes and the fact that they need to be stored for reuse during back-propagation), which are stored in `float16`.

Enabling `bfloat16`-based training methods. It seems that the need for loss scaling can be avoided if the `float16` format and associated operations are replaced with `bfloat16` (this is shown in [KMM+19], where experiments with various state of the art networks in image classification, speech recognition, language modelling, generative networks and industrial recommendation systems show the versatility of `bfloat16`-based training). This is due to the fact that `bfloat16` has the same exponent range as `float32` and the lower mantissa width does not adversely impact the final model accuracy. There are also additional hardware-related benefits that come with the combination of `bfloat16` and `float32`. Core computational primitives such as FMA units can be built using 8-bit multipliers, leading to a significant area and power savings while preserving the full dynamic range of `float32`.

The appeal of using `bfloat16` is that it also does not require any changes to the training model (as designed for a baseline `float32` approach). The increasing (planned) hardware support from several vendors seems to suggest it will soon be the *de facto* choice for performing DNN training, replacing the aforementioned `float16` approach. This statement is strengthened by the added support of `bfloat16` on NVIDIA's Ampere GPU architecture.

Fixed-point-based training. Mixed precision training approaches that are based mostly on integer/fixed-point arithmetic has also been proposed recently. These methods [CBD14, KWW+17,

DMM+18, DTJF18] use during computation integer tensors with tensor-wide shared exponents. The format explored in [CBD14] has an 11-bit mantissa and a 5-bit shared exponent, tested on custom maxout [GWFM+13] networks for the MNIST, CIFAR-10, and SVHN datasets. At each layer, every weight, bias, activation input & output, gradient vectors, and matrices have different exponent values. These exponents are updated based on a *passive* over/underflow detection policy which is run periodically during training. Because it is just reacting to the presence of overflows in the networks, it can potentially impede convergence of the training process.

To address this problem, [KWW+17] proposes widening the dynamic fixed-point format to a 16-bit mantissa and a 5-bit shared exponent, a format which they call `flexpoint` (`flex16+5`). They also introduce a new algorithm (Autoflex) for adjusting the shared exponents in an *adaptive* the way each time a tensor is written to, using tensor-wide statistics gathered at previous iterations. This essentially eliminates the appearance of overflow errors, leading to results on par with baseline `float32` training on AlexNet, ResNet-110 and Wasserstein GAN models. Choosing the bit widths that resulted in the `flex16+5` format was done such that the mantissa can encode most of the variability of values inside a tensor during one training epoch and that for weight update operations there will be sufficient mantissa overlap between tensors to ensure accurate computation (which seems to eliminate the need for 32-bit master copies of the weights during the update process).

The Flexpoint approach would require the presence of dedicated hardware for it to truly show its effectiveness. That is why in [DMM+18] another dynamic fixed-point representation that can leverage already existing general-purpose hardware (through the use of existing integer operations) is presented. The mantissa is again 16-bit, while the shared exponent is stored as an 8-bit integer. The matrix multiply and dot product operations needed for the training procedure are done using 16-bit input 32-bit output integer FMAs, with some intermediate accumulations converted to `float32` in order to avoid overflows in long addition chains. Similar to [MNA+17], a `float32` master copy of the weights is kept at each iteration for the update process. Tests are carried out on Intel XeonPhi Knights-Mill hardware for several CNN models (ResNet-50, GoogLeNet-v1, VGG-16 and AlexNet) on ImageNet, showing an 1.8× speedup over baseline `float32` training on the same platform.

While using tensors with shared exponents can lead to performance and efficiency gains in the just discussed methods, [DTJF18] identifies three potential roadblocks in their use for training acceleration: (1) whereas dot product operations can be area-efficient with such formats, other operations might be less efficient; (2) exponent sharing can lead to data loss if magnitudes are too large or too small, making exponent selection critical; (3) data loss can happen if the tensor value distributions are too wide to be captured by the allotted number of mantissa bits. To address them, [DTJF18] proposes a hybrid approach, where all dot product operations are performed with shared exponent formats, while other operations are kept in floating-point. Since training operations are dominated by dot products, there will be little overhead to using floating-point for the remaining operations.

By using tiling for matrix multiplications (with shared exponent at tile level) and wider weight storage for the weight update process (similar to other approaches), [DTJF18] can limit data loss when compared to baseline `float32` training on a large range of tasks, with little silicon density penalty. Investigating the design space, they find that the hybrid approach is most convenient for 24×24 tile sizes, 8 to 12-bit mantissa and 16-bit size for weight storage.

4.1.2 Mixed 8-16-bit precision training

While combined 16-32-bit training seems to be the most widespread approach currently, for accelerating DNN training, there has also been work recently to push the envelope further with 8-bit tensor datatypes and multiplication operators coupled with 16-bit accumulators and weight updates [WCB+18, MSDK19] (instead of the 16-32-bit mix advocated in Section 4.1.1).

According to [WCB+18, Sec. 1], there are three main elements that can significantly impact model test accuracy when using extremely low precision formats during training: (a) all operands in a tensor matrix multiply operations (GEMMs and convolutions) are in 8-bit formats (2%

degradation over a baseline `float32` training loop on ResNet18 with the ImageNet dataset), (b) GEMM accumulation results reduced from 32 to 16 bits (while critical to reducing the area and power of 8-bit hardware, such a move also leads to significant degradation — 1% with respect to the same ResNet18 baseline) and (c) reducing weight updates from 32 to 16-bits (high precision weight updates and gradients require expensive parameter copies to be kept in memory, whereas reducing their precision can also lead to significant degradation — 1.7% with respect to the ResNet18 baseline).

To cope with these problems, [WCB+18] advocates the choice of a 5-bit exponent and 2-bit mantissa floating-point format to represent weights, activations, errors, and gradients in matrix multiply operations (forward, backward and gradient), coupled with a 6-bit exponent 9-bit mantissa format for all the accumulation results. These format choices are motivated by how data is distributed inside networks in practice, with a focus on striking a balance between representation accuracy and dynamic range. To optimize the accuracy of the accumulation, a blocked approach (which is standard in high performance basic linear algebra routines) is used. The multiplications are done in the 8-bit format, whereas the accumulation is done in 16-bits to more accurately model the result (*i.e.,* try to avoid *stagnation/swamping* from appearing: small $x_k y_k$ terms cannot contribute to $\sum_{k=1}^{n} x_k y_k$ in the floating-point computation path).

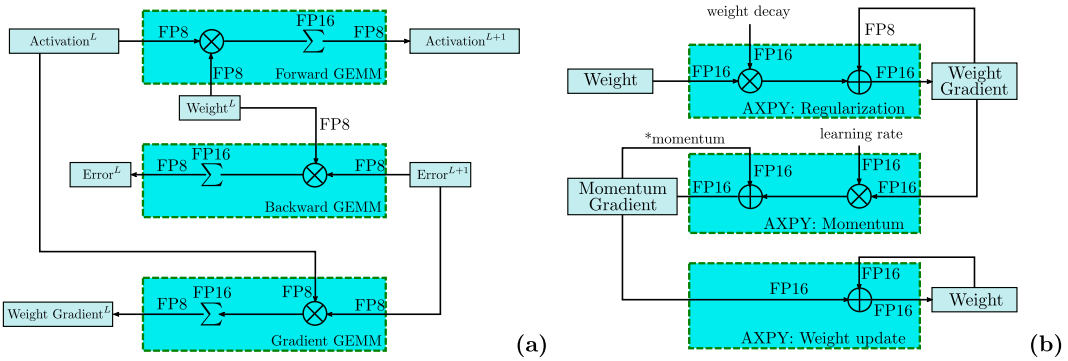


Figure 13: Summary of the precision settings for (a) the GEMM operations during the forward and backward passes in backpropagation and (b) the AXPY operations during a standard SGD weight update process (adapted from [WCB+18, Fig. 2]).

Another way to improve on the overall accuracy of summation results is to use stochastic rounding, which shows similar results to block accumulation (see [WCB+18, Fig. 3]). In the context of deep learning, it seems that using stochastic rounding is more natural for the weight update process (in the dot product AXPY operations) since the weight gradient is accumulated into the weight over mini-batches during several epochs (so not at once in a complete dot product operation!).

The precision settings for all the operations done during training are summarized in Figure 13. In terms of results, a large spectrum of neural networks for both image classification and object recognition are used (AlexNet and ResNet 18 and 50 versions for the ImageNet and CIFAR10 datasets) with both SGD and ADAM-based optimizers. A loss scaling approach similar to [MNA+17] is used to preserve the dynamic range of back-propagated errors with small magnitude.

In both [MNA+17, WCB+18], the hardware complexity of the floating-point computation pipeline is dominated by the accumulator bandwidth (32 & 16 bit, respectively), and in many cases, this size seems much too conservative. The follow-up work [SWC+19] introduces an analytical method for predicting the precision requirements for partial sum accumulation in the three GEMM accumulation units from Figure 13. It studies in what (precision/format) scenarios the variance of the accumulator units is maintained when doing dot product computations in reduced precision.

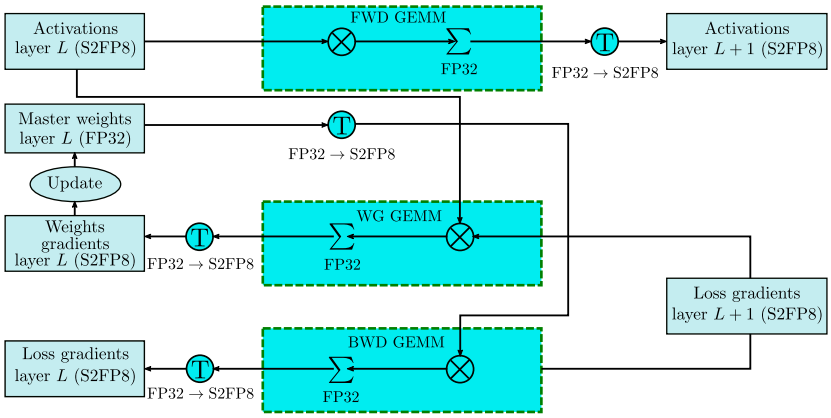


Figure 14: The low precision training flow with the S2FP8 format, where the truncation function T corresponds to $T(X) = \left[2^{-\beta}\left\{\text{round}_{\text{FP8}}(2^{\beta}|X|^{\alpha})\right\}\right]^{1/\alpha}$. The forward and backward GEMM operations use only S2FP8 values, whereas the weight update step uses FP32 master weights (adapted from [CBG+20, Fig. 4]).

One downside of all these aforementioned methods is that they require certain knobs to be finely tuned (such as appropriate chunk-based accumulator design, stochastic rounding techniques, loss scaling, and maintaining some layers of the network in higher precision – in particular the first and last ones), necessitating experimentation on a network-by-network basis. To eliminate the need for such fine-tuning, [CBG+20] proposes a new, tensor-level 8-bit floating-point format. Given an N-element tensor $X = \{X_i\}_{i=1}^{N}$, instead of encoding each element directly in an 8-bit floating-point format, X is stored using N 8-bit floating-point values $\{Y_i\}_{i=1}^{N}$ and two extra factors α and β that account for statistical information about X and capture its dynamic range. This tensor format is called S2FP8 and its use in the training procedure (for the forward & backward passes and the gradient update computations) is summarized in Figure 14.

Tests on the effectiveness of this approach (FP32 vs S2FP8) are performed on residual networks of varying depths on the CIFAR10 and ImageNet datasets, the Transformer network on an English-Vietnamese translation dataset and neural collaborative filtering network architecture. The authors of [CBG+20] state that the extra hardware complexity required to handle the conversion operations and the management of the α, β parameters at each layer is small.

4.2 Low precision training algorithm design

Section 4.1 reviewed how mixed-precision computation can be used to speed up neural network training algorithm execution, with minimal or no loss to the final test accuracy for the resulting model. Such methods are attractive because they do not require any changes to the problem's hyperparameters (such as learning rate scheduling), making them potentially easy to use (for instance the use of mixed precision training with NVIDIA GPUs is straightforward with the use of their Automatic Mixed Precision (AMP) support for major deep learning frameworks).

An orthogonal and complementary direction is the development of learning algorithms tailored for low precision computation. One such approach is MuPPET [RVVB20], which advocates for an automatic *intra* epoch numerical precision switch of training quantization and computation levels. It proposes a metric that estimates how much information each new training step obtains for a given quantization level, by quantifying the diversity of computed gradients across epochs. This allows for a heuristic runtime policy that progressively increases the working precision/format such that the final test accuracy is comparable to that of baseline `float32` training. The approach is designed to take advantage of the myriad of numerical precisions that have started to appear

in modern hardware (*e.g.,* 4 and 8-bit integer computations and 16-bit floating-point formats). For each iteration/minibatch and a working fixed-point precision q, a block floating-point training scheme (similar to [CBD14, KWW+17, DMM+18, DTJF18]) with both values and scale factors stored as q-bit integers and stochastic rounding for quantization is used. Similar to most other mixed-precision approaches, a `float32` master copy of the weights is always kept in memory and updated at each iteration with the low precision loss function gradients computed most recently. To test this approach, five levels of precision (8-, 12-, 14- and 16-bit fixed-point formats and ultimately `float32`) were used in [RVVB20] for training AlexNet, ResNet18/20 and GoogLeNet networks with the CIFAR-10/100 and ImageNet datasets on an NVIDIA RTX 2080 Ti GPU. A comparison with baseline `float32` training shows a $1.25 - 1.32\times$ speedup for MuPPET, whereas with respect to [MNA+17], it achieves a $1.23\times$ speedup for AlexNet and comparable performance for ResNet18 and GoogLeNet.

Following [DSLZ+18] "there is always a tradeoff with standard training algorithms: as the number of bits is decreased, noise that limits statistical accuracy is increased". To limit the loss in statistical accuracy when doing low precision training, they propose HALP (High Accuracy Low Precision), a low precision variant of stochastic gradient descent which uses low precision for most of the time in its innermost loop, while infrequently recentering the weight parameters with higher precision in an outer loop to counteract the noise effect of low precision quantization. The idea of the algorithm is based on the Stochastic Variance Reduced Gradient (SVRG) approach, introduced in [JZ13], and a *bit centering* representation, where each number is represented as the sum of a high precision offset term, modified only infrequently, and a low precision *delta* term, which is modified at each inner iteration.

For strongly convex problems, the authors show that the HALP approach can produce arbitrarily accurate solutions retaining the same linear asymptotic convergence rate as SVRG in full precision. On non-convex problems (namely CNN and LSTM neural network training), HALP (with a 16-bit low precision format and 32-bit high precision one) is empirically shown to improve on low precision variants of SGD and SVRG and equals or outperforms full precision SVRG and SGD. It can also be used to effectively fine-tune low precision trained results as well, as the authors show on a ResNet18 model, closely matching the result obtained from de facto SGD training in full precision. On ImageNet, such variance-reduced mixed-precision training algorithms can obtain state-of-the-art timing results [JSH+18].

A simpler approach for a low precision training algorithm is SWALP (Stochastic Weight Averaging in Low Precision Training) [YZK+19]. It is based on the recent *Stochastic Weight Averaging* (SWA) method [IPG+18]. SWA was introduced as an SGD variant that shows improved generality in deep learning training. Low precision training on the other hand produces extra quantization noise and generally tends to underperform when the learning rate is low. Averaging weights that have been rounded both down and up during quantization can potentially reduce quantization effects and is the reason why the authors of [YZK+19] propose that SWA can be beneficial for low precision training. The SWALP approach consists of quantizing in low precision all numbers during training, including the gradient accumulator (and potentially the velocity vector for momentum-based approaches). On a theoretical level, the authors can show that SWALP can converge to an optimal solution for quadratic objectives and a smaller noise ball than low precision SGD for strongly convex objective functions. Empirically, for nonconvex objectives, an 8-bit SWALP approach (with an 8-bit block floating-point format with 8-bit shared exponents) can match full precision SGD baselines in DNN training tasks such as for VGG-16 and Preactivation ResNet-164 on CIFAR-10/100 datasets.

5 Support for approximation in DNN Accelerators

DNN models can be executed in different environments, ranging from high-power data center servers to low-power edge devices. Within this large space, there is an even wider one representing the different backends that can be used. Backends are differentiated in terms of both software

and hardware. The solutions vary from general-purpose frameworks and computing units to application-specific frameworks and computing units.

Like many applications, DNNs were initially executed on latency-oriented CPUs, but ever since the start of the 2010s, there has been a major shift towards parallel hardware. Examples include GPUs for performance-oriented scenarios and microcontrollers for low-power devices. Still, due to their static and general-purpose data path, General-Purpose Processors (GPPs) are not able to efficiently process DNNs in all application scenarios, motivating the need for dedicated hardware accelerators.

The first proposed hardware accelerators were ASICs [CLL+14, CES16, JYP+17], and they achieved orders of magnitude improvements in energy efficiency compared to GPPs. This gain nevertheless comes at the expense of flexibility, with the design cost being very high. FPGAs, on the other hand, provide a good balance between flexibility, design cost, and performance [GSQ+16, RRR18].

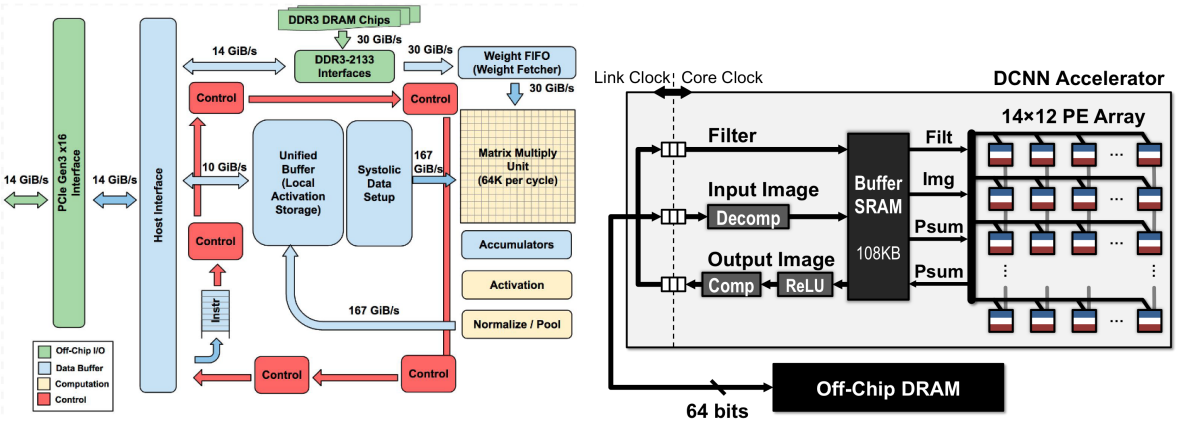


Figure 15: Comparing the systolic arrays based architecture of the ASIC Google TPU [JYP+17] (top) and the Processing Element (PE) of the FPGA grid-based Eyeriss [CES16] architecture (bottom).

Independently of the target (ASIC or FPGA), hardware accelerators adopt the same strategy of maximizing data reuse, an element that has been extensively studied by Chen & al. in [CES16]. The main architectures adopted by re-configurable accelerators such as FPGAs is a dedicated grid of Processing Element (PE) [CES16], while the main architecture adopted by ASICs are based on more generic systolic arrays [JYP+17]. This is mainly because a systolic array is more flexible once designed and can efficiently process matrix products, while a PE array requires tuning some parameters for efficiently executing a DNN (like the number of PEs and the size of the memory bus), making them more suitable for re-configurable accelerators. A schematic view of the two approaches is given in Figure 15.

Since DeepCompression [SH16] proved that approximation techniques can significantly improve DNN processing efficiency with very small accuracy loss, approximation for DNN acceleration has become quite popular, at the same time posing new challenges for efficient processing. For example, accelerating a sparse DNN (after application of pruning methods like those presented in Section 3.3) requires adapting the data-flow to take advantage of the available sparsity, whereas accelerating a reduced precision DNN (after application of quantization methods as described in Section 3.1) requires implementing dedicated operators.

5.1 Architectures for accelerating inference

Almost, if not all, dedicated DNN hardware accelerators rely on reduced precision computations. This is mainly because a 32-bit floating-point is not mandatory to achieve high accuracy, and has

a prohibitive computing cost. Most accelerators use 16-bit or 8-bit representations, like [CES16]. Some accelerators are dedicated to specific quantization formats, such as [GMC+18] that targets acceleration of fully-binarized DNNs, or [KUA+18] for accelerating logarithmic representations.

Whereas reduced precision acceleration-based solutions mainly require changes to the arithmetic operators, accelerating pruned DNNs with a sparse representation requires changes to the dataflow. Lu & al. proposed to use the combination of two structures representing the COOrdinates (COO) of the values and the values as Compressed Sparse Rows (CSR) [LWGZ17], and developed an accelerator to take advantage of these representations. It is also possible to take advantage of Feature Map (FM) sparsity. Due to the use of ReLU activations, FMs contain a large number of zeros which can be skipped during the next layer computation. CNVlutin [AJH+16] explores this dynamic sparsity. It is also possible to accelerate structured sparse DNNs with a dedicated dataflow like in [ZHY+20].

DNNs with shared weights can also benefit from a dedicated dataflow. This was studied in [HLM+16], which targets DNNs compressed using the DeepCompression [SH16] three-step method. It introduces an efficient implementation of the sparse matrix-vector multiplications with weight sharing that are central to the approach from [SH16].

5.2 Architectures for accelerating training

Accelerating DNN algorithms on hardware targets such as FPGAs faces many challenges, including limited on-chip memory, external memory bandwidth, and computational resources. Compared to the design of inference accelerators, on-chip training is a less studied topic, but it is feasible [TMSC20].

An example is [FFB+19], which targets training acceleration for embedded Xilinx Zynq All Programmable System on Chip (APSoC) devices. It essentially implements a version of the method introduced in [YZK+19] with predominantly 8-bit integer arithmetic. The Arm-based processor on the device is used for 32-bit floating-point weight updates, whereas the FPGA logic evaluates all the 8-bit integer matrix multiplications needed during the backpropagation computation path. The overall hardware platform is configured using a software-based High-Level Synthesis (HLS) flow with Xilinx tools. On the Intel side of things, [VMY+19] has proposed a Register-Transfer Level (RTL) compiler that performs SGD-based training on Intel FPGAs for various CNNs with 16-bit fixed-point arithmetic.

6 Perspectives

Due to the rapid evolution of the field of deep learning, it is difficult to give an accurate prediction of how to approximate computing techniques that will impact DL acceleration in the future. This section presents an overview of three different research directions that figure to grow in importance in the years to come.

6.1 Approximation for attention-based architectures

While the focus of the previous sections is mostly directed at CNN-based models, in recent years alternative structures such as Transformer attention architectures [VSP+17] have led to state-of-the-art accuracy results in NLP-based tasks (*e.g.,* language modeling). Subsequent models, like BERT [DCLT18], RoBERTa [LOG+19] and GPT [BMR+20], although impressive, have a large memory footprint, increased latency, and power consumption that are prohibitive for efficient deployment on embedded edge devices and even on data centers. Due to their expressive power, Transformer-based models are also beginning to be adapted for other tasks, such as computer vision applications [CMS+20, SGLS21].

Their increasing usage is driving interest for efficient approximation methods that specifically target Transformer models. While work in this direction is still in its early stages, there are already

some approaches based on quantization [ZEAM20, KGY+21], knowledge distillation [SDCW19, JLW+21] and pruning [MYL+21, WZH21].

6.2 Edge AI

One area where training acceleration with reduced precision and increased energy efficiency is becoming important is incremental/lifelong learning scenarios on edge devices (*e.g.,* in autonomous driving, IoT, and robotics). Compared to a cloud-based scenario, training locally avoids transferring data back and forth between data centers and IoT devices, helping reduce communication and latency and improve privacy.

Such on-chip training is feasible [TMSC20], but extremely challenging. The training acceleration methods described in Section 4 usually cannot be applied directly to this context and alternatives need to be considered.

A training framework specifically designed for such scenarios is E^2-Train [WJC+19], which proposes three complementary strategies: (a) stochastic mini-batch dropping to eliminate what can be considered "unnecessary costs", (b) input-dependent selective layer update where a different subset of CNN layers are updated for every minibatch, and (c) predictive sign gradient descent, a variation of an extremely low precision SGD algorithm, signSGD [BWAA18]. Besides this approach, other algorithmic & arithmetic-level methods have started to appear [FYZ+20, FGL+21]. It is expected that this area of research will grow in importance in the years to come, with on-site learning becoming paramount in certain application domains.

6.3 Analog in-memory computing

The recent explosive growth in highly data-centric applications related to DL has motivated the appearance of analog in-memory computing solutions [SNM+16, CLX+16, AHC+19, SBD+19, Dem18] as alternatives to traditional von Neumann computing systems. Hereby important computational tasks, such as vector-matrix multiplications, are performed in place in the memory itself by exploiting the physical attributes of the memory devices (*e.g.,* Kirchhoff's current summation law). Besides alleviating the costs in latency and energy associated with data movement, in-memory computing also has the potential to significantly improve the computational time complexity by using large crossbar memory arrays [SLGKAE20]. However, this comes at the expense of imprecision in the mixed-signal computations and becomes a form of approximate computing. For instance, the mapping of synaptic weights onto some of those memory devices suffers from non-ideal analog storage in the form of stochastic distribution of conductance values and temporal drifting. Accordingly, Joshi et al. [JLGH+20] have proposed a custom noise-injection training method to increase the robustness of the resulting network to such non-idealities and achieve a software equivalent accuracy. Given the game-changing advantages in computing efficiency of analog in-memory computing, more work is expected on this nascent field in the future.

7 Conclusion

In this chapter, a comprehensive survey of approximation techniques applied to Deep Learning is provided. These techniques target various improvements, some geared towards the training of DNN models, others that focus on DNN inference. Depending on the objective, various methods can be applied, whether for the improvement (reduction) of memory requirements by using compression techniques or for the reduction of the computational workload by using acceleration techniques.

Such a wide range of approximation techniques involves various implementation changes, ideally resulting in a backend adaptation that maximizes the expected performance improvement. These adaptations can be implemented at the software level using dedicated frameworks and/or at the hardware level in dedicated accelerators.

To compare the various methods available, it is desirable to use the same input DNN and workload, but this is not always feasible. This is mostly due to the large range of DNN topologies that have appeared over the years: while some methods can be applied almost automatically to multiple topologies, some require manual tuning as the size of the search space increases exponentially with the DNN size. There is also a wide range of workloads, from small "toy" datasets to more recent and challenging large-scale datasets. Some methods cannot perform equally well in both contexts. The difference in backend compatibility with the various approximation methods also regularly involves manual tuning steps, which are hard to reproduce and compare to other backends.

Most of the recent methods give very promising results and pave the way for further research, by proving that approximations can be applied at various levels, from the topology of the DNN, to the data value and type, and including the backends-DNN codesign (hardware or software).

8 Acknowledgments

This work has been funded by the French National Research Agency (ANR) through the AdequatedDL research project (ANR-18-CE23-0012).

References

[A+15] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[ABvB+20] Milad Alizadeh, Arash Behboodi, Mart van Baalen, Christos Louizos, Tijmen Blankevoort, and Max Welling. Gradient ℓ_1 regularization for quantization robustness. *arXiv preprint arXiv:2002.07520*, 2020.

[AHC+19] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R Stanley Williams, Paolo Faraboschi, Wen-mei W Hwu, John Paul Strachan, Kaushik Roy, et al. PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 715–731, 2019.

[AJH+16] J. Albericio, P. Judd, Tayler H. Hetherington, Tor M. Aamodt, N. E. Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 1–13, 2016.

[BMR+20] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[BNS19] Ron Banner, Yury Nahshan, and Daniel Soudry. Post training 4-bit quantization of convolutional networks for rapid-deployment. In *Advances in Neural Information Processing Systems*, pages 7950–7958, 2019.

[BWAA18] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Anima Anandkumar. signSGD: Compressed optimisation for non-convex problems. *arXiv preprint arXiv:1802.04434*, 2018.

[BWKL19] Haoli Bai, Jiaxiang Wu, Irwin King, and Michael Lyu. Few shot network compression via cross distillation. *arXiv preprint arXiv:1911.09450*, 2019.

[CBD14] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

[CBD15] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. BinaryConnect: training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*, pages 3123–3131, 2015.

[CBG+20] Léopold Cambier, Anahita Bhiwandiwalla, Ting Gong, Mehran Nekuii, Oguz H Elibol, and Hanlin Tang. Shifted and squeezed 8-bit floating point format for low-precision training of deep neural networks. *arXiv preprint arXiv:2001.05674*, 2020.

[CCRR13] V. K. Chippa, S. Chakradhar, K. Roy, and A. Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–9, 2013.

[CDS90] Yann Le Cun, John S. Denker, and Sara A. Solla. *Optimal Brain Damage*, page 598–605. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[CDZM19] Ting-Wu Chin, Ruizhou Ding, Cha Zhang, and Diana Marculescu. Legr: Filter pruning via learned global ranking. *CoRR*, abs/1904.12368, 2019.

[CDZM20] Ting-Wu Chin, Ruizhou Ding, Cha Zhang, and Diana Marculescu. Towards efficient model compression via learned global ranking. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2020.

[CEKL20] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. Learning sparse low-precision neural networks with learnable regularization. *IEEE Access*, 2020.

[CEL16] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. Towards the limit of network quantization. *CoRR*, abs/1612.01543, 2016.

[CES16] Y. Chen, J. Emer, and V. Sze. Eyeriss: a spatial architecture for energy-efficient dataflow for convolutional neural networks. In *CARN*, 2016.

[CHS+16] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or −1. *arXiv preprint arXiv:1602.02830*, 2016.

[CKES16] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2016.

[CKYK19] Yoni Choukroun, Eli Kravchik, Fan Yang, and Pavel Kisilev. Low-bit quantization of neural networks for efficient inference. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, pages 3009–3018. IEEE, 2019.

[CLL+14] Yunji Chen, T. Luo, Shaoli Liu, S. Zhang, Liqiang He, J. Wang, L. Li, Tianshi Chen, Z. Xu, Ninghui Sun, and O. Temam. Dadiannao: A machine-learning supercomputer. *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622, 2014.

[CLX+16] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 27–39, 2016.

[CMS+20] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-End Object Detection with Transformers. In *European Conference on Computer Vision*, pages 213–229. Springer, 2020.

[CSKX15] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.

[CWP19] Shangyu Chen, Wenya Wang, and Sinno Jialin Pan. Deep neural network quantization via layer-wise optimization using limited training data. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3329–3336, 2019.

[CWT+15] Wenlin Chen, James T. Wilson, Stephen Tyree, Kilian Q. Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. *CoRR*, abs/1504.04788, 2015.

[CWV+18] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Pact: Parameterized clipping activation for quantized neural networks. *arXiv preprint arXiv:1805.06085*, 2018.

[CYD+20] Yaohui Cai, Zhewei Yao, Zhen Dong, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. ZeroQ: A novel zero shot quantization framework. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13169–13178, 2020.

[DBCN18] Sajad Darabi, Mouloud Belbahri, Matthieu Courbariaux, and Vahid Partovi Nia. Bnn+: Improved binary network training. *arXiv preprint arXiv:1812.11800*, 2018.

[DCLT18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[DDHT18] Xiaohan Ding, G. Ding, J. Han, and Sheng Tang. Auto-balanced filter pruning for efficient convolutional neural networks. In *AAAI*, 2018.

[DDS+09] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, pages 248–255, 2009.

[Dem18] Mike Demler. Mythic multiplies in a flash. *Microprocesser Report*, 2018.

[DLH+13] Li Deng, Jinyu Li, Jui-Ting Huang, Kaisheng Yao, Dong Yu, Frank Seide, Michael Seltzer, Geoff Zweig, Xiaodong He, Jason Williams, et al. Recent advances in deep learning for speech research at microsoft. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8604–8608. IEEE, 2013.

[DMM+18] Dipankar Das, Naveen Mellempudi, Dheevatsa Mudigere, Dhiraj Kalamkar, Sasikanth Avancha, Kunal Banerjee, Srinivas Sridharan, Karthik Vaidyanathan, Bharat Kaul, Evangelos Georganas, et al. Mixed precision training of convolutional neural networks using integer operations. *arXiv preprint arXiv:1802.00930*, 2018.

[DSLZ+18] Christopher De Sa, Megan Leszczynski, Jian Zhang, Alana Marzoev, Christopher R Aberger, Kunle Olukotun, and Christopher Ré. High-accuracy low-precision training. *arXiv preprint arXiv:1803.03383*, 2018.

[DTJF18] Mario Drumond, LIN Tao, Martin Jaggi, and Babak Falsafi. Training dnns with hybrid block floating point. In *Advances in Neural Information Processing Systems*, pages 453–463, 2018.

[DYC+19] Zhen Dong, Zhewei Yao, Yaohui Cai, Daiyaan Arfeen, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Hawq-v2: Hessian aware trace-weighted quantization of neural networks. *arXiv preprint arXiv:1911.03852*, 2019.

[DYG+19] Zhen Dong, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Hawq: Hessian aware quantization of neural networks with mixed-precision. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 293–302, 2019.

[DYJ17] Xiaoliang Dai, Hongxu Yin, and Niraj K. Jha. Nest: A neural network synthesis tool based on a grow-and-prune paradigm. *CoRR*, abs/1711.02017, 2017.

[EMA+16] Steven K Esser, Paul A Merolla, John V Arthur, Andrew S Cassidy, Rathinakumar Appuswamy, Alexander Andreopoulos, David J Berg, Jeffrey L McKinstry, Timothy Melano, Davis R Barch, et al. From the cover: convolutional networks for fast, energy-efficient neuromorphic computing. *Proceedings of the National Academy of Sciences of the United States of America*, 113(41):11441, 2016.

[FC18] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Training pruned neural networks. *CoRR*, abs/1803.03635, 2018.

[FFB+19] Sean Fox, Julian Faraone, David Boland, Kees Vissers, and Philip HW Leong. Training deep neural networks in low-precision with high accuracy using FPGAs. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 1–9. IEEE, 2019.

[FGL+21] Yonggan Fu, Han Guo, Meng Li, Xin Yang, Yining Ding, Vikas Chandra, and Yingyan Lin. Cpt: Efficient deep neural network training via cyclic precision. *arXiv preprint arXiv:2101.09868*, 2021.

[FML10] Haohuan Fu, Oskar Mencer, and Wayne Luk. Fpga designs with optimized logarithmic arithmetic. *IEEE Transactions on Computers*, 59(7):1000–1006, 2010.

[FYZ+20] Yonggan Fu, Haoran You, Yang Zhao, Yue Wang, Chaojian Li, Kailash Gopalakrishnan, Zhangyang Wang, and Yingyan Lin. FracTrain: Fractionally Squeezing Bit Savings Both Temporally And Spatially For Efficient DNN Training. *arXiv preprint arXiv:2012.13113*, 2020.

[GAGN15] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pages 1737–1746, 2015.

[GMC+18] P. Guo, H. Ma, R. Chen, P. Li, S. Xie, and D. Wang. Fbna: A fully binarized neural network accelerator. *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 51–513, 2018.

[GSQ+16] K. Guo, L. Sui, J. Qiu, S. Yao, S. Han, Y. Wang, and H. Yang. Angel-eye: A complete design flow for mapping cnn onto customized hardware. In *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 24–29, 2016.

[GWFM+13] Ian J Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. *arXiv preprint arXiv:1302.4389*, 2013.

[GYC16] Yiwen Guo, A. Yao, and Y. Chen. Dynamic network surgery for efficient dnns. In *NIPS*, 2016.

[HB91] Jordan L Holt and Thomas E Baker. Back propagation simulations using limited precision calculations. In *IJCNN-91-Seattle International Joint Conference on Neural Networks*, volume 2, pages 121–126. IEEE, 1991.

[HCS+16] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 4107–4115. Curran Associates, Inc., 2016.

[HCS+17] 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.

[HH18] Yihui He and Song Han. ADC: automated deep compression and acceleration with reinforcement learning. *CoRR*, abs/1802.03494, 2018.

[HLM+16] Song Han, Xingyu Liu, Huizi Mao, J. Pu, A. Pedram, M. Horowitz, and W. Dally. Eie: Efficient inference engine on compressed deep neural network. *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 243–254, 2016.

[HQY+16] Yuxiang Huan, Yifan Qin, Yantian You, Lirong Zheng, and Zhuo Zou. A multiplication reduction technique with near-zero approximation for embedded learning in iot devices. In *2016 29th IEEE International System-on-Chip Conference (SOCC)*, pages 102–107. IEEE, 2016.

[HQY+17] Yuxiang Huan, Yifan Qin, Yantian You, Lirong Zheng, and Zhuo Zou. A low-power accelerator for deep neural networks with enlarged near-zero sparsity. *arXiv preprint arXiv:1705.08009*, 2017.

[HS92] B. Hassibi and D. Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *NIPS*, 1992.

[HS14] Kyuyeon Hwang and Wonyong Sung. Fixed-point feedforward deep neural network design using weights +1, 0, and -1. In *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 1–6. IEEE, 2014.

[HVD15] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[HWZ+16] Qinyao He, He Wen, Shuchang Zhou, Yuxin Wu, Cong Yao, Xinyu Zhou, and Yuheng Zou. Effective quantization methods for recurrent neural networks. *arXiv preprint arXiv:1611.10176*, 2016.

[HZC+17] 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. *CoRR*, abs/1704.04861, 2017.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[HZRS16] 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, 2016.

[HZS17] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. *CoRR*, abs/1707.06168, 2017.

[IMA+16] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016.

[IPG+18] Pavel Izmailov, Dmitrii Podoprikhin, Timur Garipov, Dmitry Vetrov, and Andrew Gordon Wilson. Averaging weights leads to wider optima and better generalization. *arXiv preprint arXiv:1803.05407*, 2018.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[JKC+18] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.

[JLD+18] Y. Ji, Ling Liang, L. Deng, Y. Zhang, Youhui Zhang, and Y. Xie. Tetris: Tile-matching the tremendous irregular sparsity. In *NeurIPS*, 2018.

[JLGH+20] Vinay Joshi, Manuel Le Gallo, Simon Haefeli, Irem Boybat, Sasidharan Rajalekshmi Nandakumar, Christophe Piveteau, Martino Dazzi, Bipin Rajendran, Abu Sebastian, and Evangelos Eleftheriou. Accurate deep neural network inference using computational phase-change memory. *Nature communications*, 11(1):1–13, 2020.

[JLW+21] Jing Jin, Cai Liang, Tiancheng Wu, Liqin Zou, and Zhiliang Gan. KDLSQ-BERT: A Quantized BERT Combining Knowledge Distillation with Learned Step Size Quantization. *arXiv preprint arXiv:2101.05938*, 2021.

[Joh18] Jeff Johnson. Rethinking floating point for deep learning. *arXiv:1811.01721 [cs]*, November 2018. arXiv: 1811.01721.

[JSH+18] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018.

[JSL+19] Sangil Jung, Changyong Son, Seohyung Lee, Jinwoo Son, Jae-Joon Han, Youngjun Kwak, Sung Ju Hwang, and Changkyu Choi. Learning to quantize deep networks by optimizing quantization intervals with task loss. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4350–4359, 2019.

[JYP+17] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2017.

[JZ13] Rie Johnson and Tong Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in neural information processing systems*, pages 315–323, 2013.

[KB14] Diederik P Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[KBM+17] Abhisek Kundu, Kunal Banerjee, Naveen Mellempudi, Dheevatsa Mudigere, Dipankar Das, Bharat Kaul, and Pradeep Dubey. Ternary residual networks. *arXiv preprint arXiv:1707.04679*, 2017.

[KGY+21] Sehoon Kim, Amir Gholami, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. I-BERT: Integer-only BERT Quantization. *arXiv preprint arXiv:2101.01321*, 2021.

[KL18] Soroosh Khoram and Jing Li. Adaptive quantization of neural networks. In *International Conference on Learning Representations*, 2018.

[KMM+19] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, et al. A study of bfloat16 for deep learning training. *arXiv preprint arXiv:1905.12322*, 2019.

[Kri09] Alex Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. Master's thesis, University of Toronto, 2009.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25, pages 1097–1105. Curran Associates, Inc., 2012.

[KSH17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[KUA+18] T. Kudo, Kodai Ueyoshi, K. Ando, K. Hirose, Ryota Uematsu, Yuka Oba, M. Ikebe, T. Asai, Masato Motomura, and Shinya Takamaeda-Yamazaki. Area and energy optimization for bit-serial log-quantized dnn accelerator with shared accumulators. *2018 IEEE 12th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pages 237–243, 2018.

[KWW+17] Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K Bansal, William Constable, Oguz Elibol, Scott Gray, Stewart Hall, Luke Hornof, et al. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. In *Advances in neural information processing systems*, pages 1742–1752, 2017.

[LBBH98] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[LC10] Yann LeCun and Corinna Cortes. MNIST handwritten digit database, 2010.

[LCJB+89] Yann Le Cun, Lionel D Jackel, Brian Boser, John S Denker, Henry P Graf, Isabelle Guyon, Don Henderson, Richard E Howard, and William Hubbard. Handwritten digit recognition: Applications of neural network chips and automatic learning. *IEEE Communications Magazine*, 27(11):41–46, 1989.

[LDJ+17] Zhiqiang Liu, Yong Dou, Jingfei Jiang, Jinwei Xu, Shijie Li, Yongmei Zhou, and Yingnan Xu. Throughput-optimized fpga accelerator for deep convolutional neural networks. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 10(3):1–23, 2017.

[LJxZ+20] Mingbao Lin, Rongrong Ji, Yu xin Zhang, B. Zhang, Yongjian Wu, and Yonghong Tian. Channel pruning via automatic structure search. *ArXiv*, abs/2001.08565, 2020.

[LKD+17] Hao Li, Asim Kadav, Igor Durdanovic, H. Samet, and H. Graf. Pruning filters for efficient convnets. *ArXiv*, abs/1608.08710, 2017.

[LL16] V. Lebedev and V. Lempitsky. Fast convnets using group-wise brain damage. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2554–2564, 2016.

[Llo82] Stuart P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28:129–137, 1982.

[LLS+17] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. *CoRR*, abs/1708.06519, 2017.

[LLS+19] Xiaocong Lian, Zhenyu Liu, Zhourui Song, Jiwu Dai, Wei Zhou, and Xiangyang Ji. High-performance FPGA-based CNN accelerator with block-floating-point arithmetic. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(8):1874–1885, 2019.

[LLZJ17] Cong Leng, Hao Li, Shenghuo Zhu, and Rong Jin. Extremely low bit neural network: Squeeze the last bit out with admm. *arXiv preprint arXiv:1707.09870*, 2017.

[LMZ+19] Z. Liu, H. Mu, X. Zhang, Zichao Guo, X. Yang, K. Cheng, and Jian Sun. Metapruning: Meta learning for automatic neural network channel pruning. *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 3295–3304, 2019.

[LOG+19] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[LSZ+19] Zhuang Liu, M. Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. *ArXiv*, abs/1810.05270, 2019.

[LTA16] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *International conference on machine learning*, pages 2849–2858, 2016.

[LW20] Jian-Hao Luo and Jianxin Wu. Autopruner: An end-to-end trainable filter pruning method for efficient deep model inference. *Pattern Recognit.*, 107:107461, 2020.

[LWGZ17] Yuntao Lu, C. Wang, L. Gong, and X. Zhou. Sparsenn: A performance-efficient accelerator for large-scale sparse neural networks. *International Journal of Parallel Programming*, 46:648–659, 2017.

[LWL17] Jian-Hao Luo, Jianxin Wu, and W. Lin. Thinet: A filter level pruning method for deep neural network compression. *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 5068–5076, 2017.

[LZL16] Fengfu Li, Bo Zhang, and Bin Liu. Ternary weight networks. *arXiv preprint arXiv:1605.04711*, 2016.

[LZP17] Xiaofan Lin, Cong Zhao, and Wei Pan. Towards accurate binary convolutional neural network. In *Advances in Neural Information Processing Systems*, pages 345–353, 2017.

[MHP+17] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J Dally. Exploring the regularity of sparse structure in convolutional neural networks. *arXiv preprint arXiv:1705.08922*, 2017.

[MLM16] Daisuke Miyashita, Edward H Lee, and Boris Murmann. Convolutional neural networks using logarithmic data representation. *arXiv preprint arXiv:1603.01025*, 2016.

[MNA+17] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[MNCM17] Asit Mishra, Eriko Nurvitadhi, Jeffrey J Cook, and Debbie Marr. WRPN: wide reduced-precision networks. *arXiv preprint arXiv:1709.01134*, 2017.

[MSDK19] Naveen Mellempudi, Sudarshan Srinivasan, Dipankar Das, and Bharat Kaul. Mixed precision training with 8-bit floating point. *arXiv preprint arXiv:1905.12334*, 2019.

[MTK+16] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient transfer learning. *CoRR*, abs/1611.06440, 2016.

[MYL+21] Jiachen Mao, Huanrui Yang, Ang Li, Hai Li, and Yiran Chen. TPrune: Efficient Transformer Pruning for Mobile Devices. *ACM Transactions on Cyber-Physical Systems*, 5(3):1–22, 2021.

[NBBW19] Markus Nagel, Mart van Baalen, Tijmen Blankevoort, and Max Welling. Data-free quantization through weight equalization and bias correction. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1325–1334, 2019.

[NDSE17] Sharan Narang, Greg Diamos, S. Sengupta, and E. Elsen. Exploring sparsity in recurrent neural networks. *ArXiv*, abs/1704.05119, 2017.

[OLZ+16] Joachim Ott, Zhouhan Lin, Ying Zhang, Shih-Chii Liu, and Yoshua Bengio. Recurrent neural networks with limited numerical precision. *arXiv preprint arXiv:1608.06902*, 2016.

[PAY17] Eunhyeok Park, Junwhan Ahn, and Sungjoo Yoo. Weighted-entropy-based quantization for deep neural networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7197–7205, 2017.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[PH94] R Kimball Presley and Roger L Haggard. A fixed point implementation of the backpropagation learning algorithm. In *Proceedings of SOUTHEASTCON'94*, pages 136–138. IEEE, 1994.

[PNI+18] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.

[PPA18] Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization. *arXiv preprint arXiv:1802.05668*, 2018.

[PYV18] Eunhyeok Park, Sungjoo Yoo, and Peter Vajda. Value-aware quantization for training and inference of neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 580–595, 2018.

[RDS+15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[RIKR17] Mohammad Samragh Razlighi, Mohsen Imani, Farinaz Koushanfar, and Tajana Rosing. LookNN: Neural network with no multiplication. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1775–1780. IEEE, 2017.

[RORF16] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: ImageNet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer, 2016.

[RRR18] R. Reddy, B. M. Reddy, and B. Reddy. Dlau: A scalable deep learning accelerator unit on fpga. *International Journal of Research*, 5:921–928, 2018.

[RVVB20] Aditya Rajagopal, Diederik Adriaan Vink, Stylianos I Venieris, and Christos-Savvas Bouganis. Multi-Precision Policy Enforced Training (MuPPET): A precision-switching strategy for quantised fixed-point training of CNNs. *arXiv preprint arXiv:2006.09049*, 2020.

[RWC+19] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[SB15] Suraj Srinivas and R. Venkatesh Babu. Data-free parameter pruning for deep neural networks. In *BMVC*, 2015.

[SBD+18] Sean O Settle, Manasa Bollavaram, Paolo D'Alberto, Elliott Delaye, Oscar Fernandez, Nicholas Fraser, Aaron Ng, Ashish Sirasao, and Michael Wu. Quantizing convolutional neural networks for low-power high-throughput inference engines. *arXiv preprint arXiv:1805.07941*, 2018.

[SBD+19] A. Sebastian, I. Boybat, M. Dazzi, I. Giannopoulos, V. Jonnalagadda, V. Joshi, G. Karunaratne, B. Kersting, R. Khaddam-Aljameh, S. R. Nandakumar, A. Petropoulos, C. Piveteau, T. Antonakopoulos, B. Rajendran, M. Le Gallo, and E. Eleftheriou. Computational memory-based inference and training of deep neural networks. In *Proceedings of the Symposium on VLSI Technology*, pages T168–T169, 2019.

[SCB+20] Moran Shkolnik, Brian Chmiel, Ron Banner, Gil Shomron, Yuri Nahshan, Alex Bronstein, and Uri Weiser. Robust quantization: One model to rule them all. *arXiv preprint arXiv:2002.07686*, 2020.

[SCYE20] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks. *Synthesis Lectures on Computer Architecture*, 15(2):1–341, 2020.

[SDCW19] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[SDY+20] Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Q-bert: Hessian based ultra low precision quantization of bert. In *AAAI*, pages 8815–8821, 2020.

[SG94] Patrice Y Simard and Hans Peter Graf. Backpropagation without multiplication. In *Advances in Neural Information Processing Systems*, pages 232–239, 1994.

[SGLS21] Robin Strudel, Ricardo Garcia, Ivan Laptev, and Cordelia Schmid. Segmenter: Transformer for Semantic Segmentation. *arXiv preprint arXiv:2105.05633*, 2021.

[SGM19] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp. *arXiv preprint arXiv:1906.02243*, 2019.

[SH16] William J. Dally Song Han, Huizi Mao. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv*, 2016.

[SHM+16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, and et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan 2016.

[SHZ+18] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks, 2018.

[SLGKAE20] Abu Sebastian, Manuel Le Gallo, Riduan Khaddam-Aljameh, and Evangelos Eleftheriou. Memory devices and applications for in-memory computing. *Nature nanotechnology*, 15(7):529–544, 2020.

[SLJ+14] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[SLL19] David R So, Chen Liang, and Quoc V Le. The Evolved Transformer. *arXiv preprint arXiv:1901.11117*, 2019.

[SLPY18] Xiaoyu Sun, Rui Liu, Xiaochen Peng, and Shimeng Yu. Computing-in-memory with sram and rram for binary neural networks. In *2018 14th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, pages 1–4. IEEE, 2018.

[SLW17] Zhourui Song, Zhenyu Liu, and Dongsheng Wang. Computation error analysis of block floating point arithmetic oriented convolution neural network accelerator design. *arXiv preprint arXiv:1709.07776*, 2017.

[SMA07] Antony W Savich, Medhat Moussa, and Shawki Areibi. The impact of arithmetic representation on implementing MLP-BP on FPGAs: A study. *IEEE Transactions on Neural Networks*, 18(1):240–252, 2007.

[SNM+16] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 14–26, 2016.

[SSH15] Wonyong Sung, Sungho Shin, and Kyuyeon Hwang. Resiliency of deep neural networks under quantization. *arXiv preprint arXiv:1511.06488*, 2015.

[STKV20] Robert Stojnic, Ross Taylor, Viktor Kerkez, and Ludovic Viaud. *Papers with code, State of the Art models on the ImageNet dataset*, 2020 (accessed november 19, 2020).

[SWC+19] Charbel Sakr, Naigang Wang, Chia-Yu Chen, Jungwook Choi, Ankur Agrawal, Naresh Shanbhag, and Kailash Gopalakrishnan. Accumulation bit-width scaling for ultra-low precision training of deep networks. *arXiv preprint arXiv:1901.06588*, 2019.

[THBR17] Hokchhay Tann, Soheil Hashemi, R Iris Bahar, and Sherief Reda. Hardware-software codesign of accurate, multiplier-free deep neural networks. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2017.

[THW17] Wei Tang, Gang Hua, and Liang Wang. How to train a compact binary neural network with high accuracy? In *AAAI*, pages 2625–2631, 2017.

[TL19] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *CoRR*, abs/1905.11946, 2019.

[TMSC20] Yudong Tao, Rui Ma, Mei-Ling Shyu, and Shu-Ching Chen. Challenges in Energy-Efficient Deep Neural Network Training with FPGA. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 400–401, 2020.

[TSZ+20] Jiaxi Tang, Rakesh Shivanna, Zhe Zhao, Dong Lin, Anima Singh, Ed H Chi, and Sagar Jain. Understanding and Improving Knowledge Distillation. *arXiv preprint arXiv:2002.03532*, 2020.

[TYW+20] Thierry Tambe, En-Yu Yang, Zishen Wan, Yuntian Deng, Vijay Janapa Reddi, Alexander Rush, David Brooks, and Gu-Yeon Wei. Algorithm-hardware co-design of adaptive floating-point encodings for resilient deep learning inference. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[VMY+19] Shreyas Kolala Venkataramanaiah, Yufei Ma, Shihui Yin, Eriko Nurvithadhi, Aravind Dasu, Yu Cao, and Jae-sun Seo. Automatic compiler based FPGA accelerator for CNN training. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 166–172. IEEE, 2019.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30:5998–6008, 2017.

[WCB+18] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *Advances in neural information processing systems*, pages 7675–7684, 2018.

[WGY+16] Chao Wang, Lei Gong, Qi Yu, Xi Li, Yuan Xie, and Xuehai Zhou. Dlau: A scalable deep learning accelerator unit on fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(3):513–517, 2016.

[WJC+19] Yue Wang, Ziyu Jiang, Xiaohan Chen, Pengfei Xu, Yang Zhao, Yingyan Lin, and Zhangyang Wang. E2-train: Training state-of-the-art cnns with over 80% energy savings. In *Advances in Neural Information Processing Systems*, pages 5138–5150, 2019.

[WLCS18] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and inference with integers in deep neural networks. *arXiv preprint arXiv:1802.04680*, 2018.

[WLL+19] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8612–8620, 2019.

[WLW+15] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized convolutional neural networks for mobile devices. *CoRR*, abs/1512.06473, 2015.

[WSL+18] Diwen Wan, Fumin Shen, Li Liu, Fan Zhu, Jie Qin, Ling Shao, and Heng Tao Shen. TBN: Convolutional neural network with ternary inputs and binary weights. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 315–332, 2018.

[WWC+20] Chen Wu, Mingyu Wang, Xinyuan Chu, Kun Wang, and Lei He. Low precision floating-point arithmetic for high performance fpga-based cnn acceleration. *arXiv preprint arXiv:2003.03852*, 2020.

[WWL+20] Chen Wu, Mingyu Wang, Xiayu Li, Jicheng Lu, Kun Wang, and Lei He. Phoenix: A low-precision floating-point quantization oriented architecture for convolutional neural networks. *arXiv preprint arXiv:2003.02628*, 2020.

[WWW+16] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. *ArXiv*, abs/1608.03665, 2016.

[WWW+18] Junru Wu, Yue Wang, Zhenyu Wu, Zhangyang Wang, Ashok Veeraraghavan, and Yingyan Lin. Deep k-means: Re-training and parameter sharing with harder cluster assignments for compressing deep convolutions. *CoRR*, abs/1806.09228, 2018.

[WWXX20] Tao Wang, Junsong Wang, Chang Xu, and Chao Xue. Automatic low-bit hybrid quantization of neural networks through meta learning. *arXiv preprint arXiv:2004.11506*, 2020.

[WWZ+18] Bichen Wu, Yanghan Wang, Peizhao Zhang, Yuandong Tian, Peter Vajda, and Kurt Keutzer. Mixed precision quantization of convnets via differentiable neural architecture search. *arXiv preprint arXiv:1812.00090*, 2018.

[WZH21] Hanrui Wang, Zhekai Zhang, and Song Han. SpAtten: Efficient sparse attention architecture with cascade token and head pruning. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[WZWH19] H. Wang, Qiming Zhang, Yuehai Wang, and H. Hu. Structured pruning for efficient convnets via incremental regularization. *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2019.

[YCS17] Tien-Ju Yang, Y. Chen, and V. Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6071–6079, 2017.

[YL06] Ming Yuan and Yi Lin. Model selection and estimation in regression with grouped variables. *JOURNAL OF THE ROYAL STATISTICAL SOCIETY, SERIES B*, 68:49–67, 2006.

[YLP+17] Jiecao Yu, Andrew Lukefahr, D. Palframan, Ganesh S. Dasika, R. Das, and S. Mahlke. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 548–560, 2017.

[YYZ+20] Dingcheng Yang, Wenjian Yu, Ao Zhou, Haoyuan Mu, Gary Yao, and Xiaoyi Wang. Dp-net: Dynamic programming guided deep neural network compression, 2020.

[YZK+19] Guandao Yang, Tianyi Zhang, Polina Kirichenko, Junwen Bai, Andrew Gordon Wilson, and Christopher De Sa. Swalp: Stochastic weight averaging in low-precision training. *arXiv preprint arXiv:1904.11943*, 2019.

[YZZ+18] Shaokai Ye, Tianyun Zhang, Kaiqi Zhang, Jiayu Li, Kaidi Xu, Yunfei Yang, Fuxun Yu, Jian Tang, Makan Fardad, Sijia Liu, Xiang Chen, Xue Lin, and Yanzhi Wang. Progressive weight pruning of deep neural networks using ADMM. *CoRR*, abs/1810.07378, 2018.

[ZEAM20] Ali Hadi Zadeh, Isak Edo, Omar Mohamed Awad, and Andreas Moshovos. GOBO: Quantizing attention-based NLP models for low latency and energy efficient inference. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 811–824. IEEE, 2020.

[ZHD+19] Ritchie Zhao, Yuwei Hu, Jordan Dotzel, Christopher De Sa, and Zhiru Zhang. Improving neural network quantization without retraining using outlier channel splitting. *arXiv preprint arXiv:1901.09504*, 2019.

[ZHMD16] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.

[ZHY+20] C. Zhu, K. Huang, Shuyuan Yang, Ziqi Zhu, H. Zhang, and Haibin Shen. An efficient hardware accelerator for structured sparse convolutional neural networks on fpgas. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28:1953–1965, 2020.

[ZMDCF17] Yiren Zhou, Seyed-Mohsen Moosavi-Dezfooli, Ngai-Man Cheung, and Pascal Frossard. Adaptive quantization for deep neural network. *arXiv preprint arXiv:1712.01048*, 2017.

[ZWN+16] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[ZWW+17] Shu-Chang Zhou, Yu-Zhi Wang, He Wen, Qin-Yao He, and Yu-Heng Zou. Balanced quantization: An effective and efficient approach to quantized neural networks. *Journal of Computer Science and Technology*, 32(4):667–682, 2017.

[ZYG+17] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless CNNs with low-precision weights. *arXiv preprint arXiv:1702.03044*, 2017.

[ZYYH18] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*, pages 365–382, 2018.

[ZYZ+18] Tianyun Zhang, Shaokai Ye, Kaiqi Zhang, J. Tang, W. Wen, M. Fardad, and Yanzhi Wang. A systematic dnn weight pruning framework using alternating direction method of multipliers. *ArXiv*, abs/1804.03294, 2018.

[ZZL18] Xiaotian Zhu, Wengang Zhou, and Houqiang Li. Adaptive layerwise quantization for deep neural network compression. In *2018 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1–6. IEEE, 2018.