Exploring Accumulated Gradient-Based Quantization and Compression for Deep Neural Networks

Meghana Laxmidhar Gaopande

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

A. Lynn Abbott, Chair
Jia-Bin Huang
Ryan K. Williams

May 7, 2020
Blacksburg, Virginia

Keywords: Deep Neural Networks, Quantization, Pruning, Fixed-Point

Exploring Accumulated Gradient-Based Quantization and Compression for Deep Neural Networks

Meghana Laxmidhar Gaopande

(ABSTRACT)

The growing complexity of neural networks makes their deployment on resource-constrained embedded or mobile devices challenging. With millions of weights and biases, modern deep neural networks can be computationally intensive, with large memory, power and computational requirements. In this thesis, we devise and explore three quantization methods (post-training, in-training and combined quantization) that quantize 32-bit floating-point weights and biases to lower bit width fixed-point parameters while also achieving significant pruning, leading to model compression. We use the total accumulated absolute gradient over the training process as the indicator of importance of a parameter to the network. The most important parameters are quantized by the smallest amount. The post-training quantization method sorts and clusters the accumulated gradients of the full parameter set and subsequently assigns a bit width to each cluster. The in-training quantization method sorts and divides the accumulated gradients into two groups after each training epoch. The larger group consisting of the lowest accumulated gradients is quantized. The combined quantization method performs in-training quantization followed by post-training quantization. We assume storage of the quantized parameters using compressed sparse row format for sparse matrix storage. On LeNet-300-100 (MNIST dataset), LeNet-5 (MNIST dataset), AlexNet (CIFAR-10 dataset) and VGG-16 (CIFAR-10 dataset), post-training quantization achieves 7.62×, 10.87×, 6.39× and 12.43× compression, in-training quantization achieves 22.08×, 21.05×, 7.95× and 12.71× compression and combined quantization achieves 57.22×, 50.19×, 13.15× and 13.53× compression, respectively. Our methods quantize at the cost of accuracy, and we present our work in the light of the accuracy-compression trade-off.

Exploring Accumulated Gradient-Based Quantization and Compression for Deep Neural Networks

Meghana Laxmidhar Gaopande

(GENERAL AUDIENCE ABSTRACT)

Neural networks are being employed in many different real-world applications. By learning the complex relationship between the input data and ground-truth output data during the training process, neural networks can predict outputs on new input data obtained in real time. To do so, a typical deep neural network often needs millions of numerical parameters, stored in memory. In this research, we explore techniques for reducing the storage requirements for neural network parameters. We propose software methods that convert 32-bit neural network parameters to values that can be stored using fewer bits. Our methods also convert a majority of numerical parameters to zero. Using special storage methods that only require storage of non-zero parameters, we gain significant compression benefits. On typical benchmarks like LeNet-300-100 (MNIST dataset), LeNet-5 (MNIST dataset), AlexNet (CIFAR-10 dataset) and VGG-16 (CIFAR-10 dataset), our methods can achieve up to 57.22×, 50.19×, 13.15× and 13.53× compression respectively. Storage benefits are achieved at the cost of classification accuracy, and we present our work in the light of the accuracy-compression trade-off.

Acknowledgments

I would like to express my sincere appreciation to my advisor and committee chair, Dr. A. Lynn Abbott, for his persistent guidance, encouragement and patience throughout this work. Thank you for mentoring me and for supporting my research interests.

I would also like to thank Dr. Jia-Bin Huang and Dr. Ryan K. Williams for serving on my committee.

I am thankful to the Bradley Department of Electrical and Computer Engineering for an enriching graduate experience and for supporting me through graduate assistantship opportunities. I would like to express my gratitude to the Virginia Tech Graduate School for their continual assistance. I would like to acknowledge Advanced Research Computing at Virginia Tech for providing computational resources and technical support that have contributed to the results reported in this thesis.

I had the terrific opportunity to be a research intern at Ford Greenfield Labs over the summer, which sparked my interest in pursuing research in the field of efficient machine learning. I would like to extend my thanks to the Automated Systems team at Greenfield Labs.

I am thankful to my family for their confidence in me and support of my ambitions. I would like to acknowledge my mentor, Dr. Nitin Narappanawar for always encouraging me to pursue graduate studies. Finally, I would like to thank my dearest friends, Aniruddha, Apoorva, Gohad, Elliott and Sneha for their unwavering friendship.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Neural networks have been shown to solve complex problems in many different real-world applications. Critical applications like autonomous vehicles and medical systems warrant a high accuracy as well as a high inference speed or low latency. As the complexity of neural network architectures has increased, a typical model can have a very large number of weights and biases (parameters), making it computationally intensive, with large memory and power requirements. The growing size and complexity of neural networks makes their deployment on resource-constrained embedded or mobile devices challenging. This impediment has given rise to a field of research that aims to achieve resource-constrained efficient machine learning computations.

A recent survey paper by Sze et al. [56] notes that unlike earlier work in deep neural networks, recent work has focused on hardware-software codesign or hardware-aware software algorithms to maximize gains in the trade-off between accuracy and throughput during inference. Sze et al. [56] group these approaches as follows:

1. Reducing precision of operations and operands

   This consists of reducing bit width, using a more efficient numerical storage format (fixed-point instead of floating-point), quantization, and weight sharing.

2. Reducing number of operations and model size

   This consists of techniques like pruning and using compact neural network architectures.

Recent works combine both the approaches described above, achieving neural network compression jointly through network pruning and weight quantization. Han et al. [24] demonstrated that applying weight quantization to a pruned neural network achieves better compression than pruning or compression alone, while maintaining accuracy. Tung and Mori [57] perform in-parallel network pruning and weight quantization during training. However, separate techniques are used to prune and quantize parameters.

We explore methods that achieve network pruning as well as parameter quantization, i.e., we do not use separate techniques for pruning and quantization. The goal of this thesis is to devise new methods that quantize 32-bit floating-point parameters (weights and biases) to lower bit width fixed-point parameters while also achieving significant pruning, leading to reduced storage requirements for the neural network, without significant degradation in accuracy.

Pruning methods often use some metric to determine the importance of a parameter in a neural network, and insignificant parameters are pruned away. In this thesis, we use a similar parameter importance metric and apply it to quantization. Our outlook is that the importance of a parameter determines its tolerance for quantization error.

Recent work by Golub et al. [19] [18] shows that neural network parameters that accumulate the highest total gradients during the training process constitute most of the learning. During backpropagation, their procedure updates only those weights that have the highest total accumulated gradient. Inspired by this work, we have developed a new procedure for quantization that is based on accumulating gradients for the network parameters during the

training process. Since we use mini-batch optimization algorithms, we accumulate gradients for the last mini-batch (with shuffling) for every epoch. We use the accumulated gradient of a parameter to determine its importance to the network. In our quantization methods, the quantization decision for each parameter is based on this importance criterion.

1.2 Contributions

We devise and explore three new accumulated-gradient based quantization methods to achieve neural network compression. To the best of our knowledge, the usage of accumulated gradients as an importance criterion for quantization decisions has not been applied to fixed-point quantization before. The first method performs post-training quantization, while the second method performs in-training quantization. The third method is a combination of both post-training and in-training quantization. This method gives us the best results in terms of compression ratio and percentage of non-zero parameters, balanced with the loss in post-quantization accuracy. We show results on standard benchmark architectures like Lenet-300-100, LeNet5, AlexNet and VGG-16 using MNIST and CIFAR-10 datasets.

Our methods have the following benefits:

1. The methods we have devised achieve network pruning as well as quantization through the same methods, i.e., we do not use separate techniques for pruning and quantization.

2. The importance of a parameter in the neural network is judged based on the gradients it accumulates during the training process. There is no computational overhead of calculating any additional importance measures, as gradients are calculated in the backward pass of the training process. As elaborated in Sections 2.2 and 2.3, this is an unusual approach which we have adapted from Golub et al. [19].

3. The quantization levels or values that we assign to weights and biases during quantization are fixed-point friendly. Models created using our methods can potentially be used on hardware supporting fixed-point, sparse matrix or weight-sharing capabilities.

4. We achieve significant pruning, i.e., a large percentage of weights and biases are zero-valued after quantization. On Lenet-300-100 (MNIST dataset), LeNet5 (MNIST dataset), AlexNet (CIFAR-10 dataset) and VGG-16 (CIFAR-10 dataset), our quantized networks achieve up to 3.61%, 3.29%, 10.18% and 7.73% non-zero parameters, respectively.

5. We convert 32-bit floating-point weights and biases to a lower bit width fixed-point format. On Lenet-300-100 (MNIST dataset), LeNet5 (MNIST dataset), AlexNet (CIFAR-10 dataset) and VGG-16 (CIFAR-10 dataset), we achieve a bit width of 4, 4, 7 and 7 bits per weight, respectively.

6. As a result of pruning as well as quantization, we are able to achieve significant compression. We assume storage of the quantized parameters using compressed sparse row format for sparse matrix storage. We achieve up to 57.22×, 50.19×, 13.15× and 13.53× compression on Lenet-300-100 (MNIST dataset), LeNet5 (MNIST dataset), AlexNet (CIFAR-10 dataset) and VGG-16 (CIFAR-10 dataset), respectively.

7. Our methods allow users to decide the accuracy-compression trade-off point to utilize during quantization.

1.3 Thesis Organization

The rest of the thesis is organized as follows:

- Chapter 2 provides background and a literature review about neural network compression and acceleration.
- Chapter 3 discusses the number representation and storage methods typically used and details the number representation used in this work.
- Chapter 4 details the three quantization methods we use to achieve neural network compression through fixed-point quantization.
- Chapter 5 presents the compression results of our methods on standard benchmark architectures.
- Chapter 6 presents a summary of work and discusses future work opportunities.

Chapter 2

Neural Network Compression and Acceleration

Deploying deep neural networks on resource constrained hardware is a challenge due to the limited memory, power and computational capabilities of these platforms. Neural network compression and acceleration can address these challenges and is currently an active field of research in the deep learning domain.

In this section, we discuss two network compression techniques: pruning and quantization. These are software-level techniques that can be used to optimize the computational performance, power efficiency or memory requirements for a neural network. These techniques try to optimize for the accuracy-compression trade-off, i.e., striking a balance between gaining compression and losing accuracy. We present a literature survey of pruning and quantization techniques. We also present a literature review of some hybrid or combination methods of achieving compression.

We also discuss the hardware and software implications of the sparsity and compression achieved through pruning and quantization, with a literature review of recent hardware accelerators and sparse computation software libraries.

2.1 Neural Networks

A neural network defines a mapping $y = f(x; W)$, where x is an input and y is the corresponding computed output. For classification problems, y would be the class label for x. During a training procedure, inputs are provided to the network and computed outputs are compared to the "ground-truth" outputs. The values of the parameters (weights and biases) W are incrementally adjusted to learn a reasonable function approximation [20]. The process of applying the learned function f on a previously unseen input x_{test} and predicting an output y_{pred} is known as inference.

Neural networks are loosely inspired by how the brain functions, with a set of interconnected neurons. Figure 2.1 depicts a neuron model. The connections between neurons are known as synapses. Synapses scale the input signals with some scaling factors (weights). Each neuron has multiple input signals (x_0, x_1, x_2 in Figure 2.1). These are scaled with weights (w_0, w_1, w_2 in Figure 2.1). The weighted sum of the inputs, often times added with a bias term, is computed in a neuron. This is followed by the application of a nonlinear function (g in Figure 2.1). The non-linear function is known as an activation function [56].

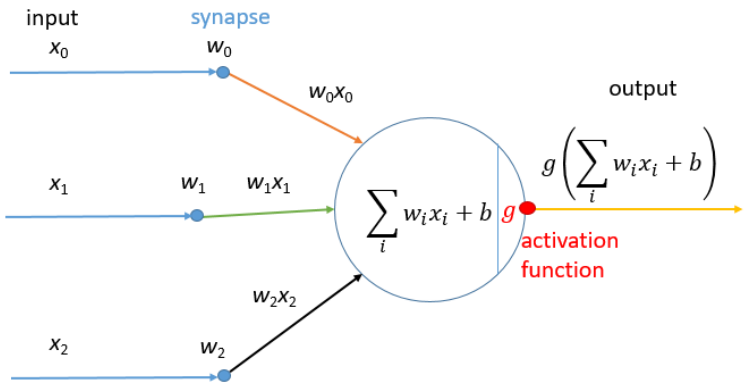


Figure 2.1: Illustration of a neuron model. The weighted sum of inputs, often added with a bias term, is followed by the application of an activation function. (Figure adapted from [36].)

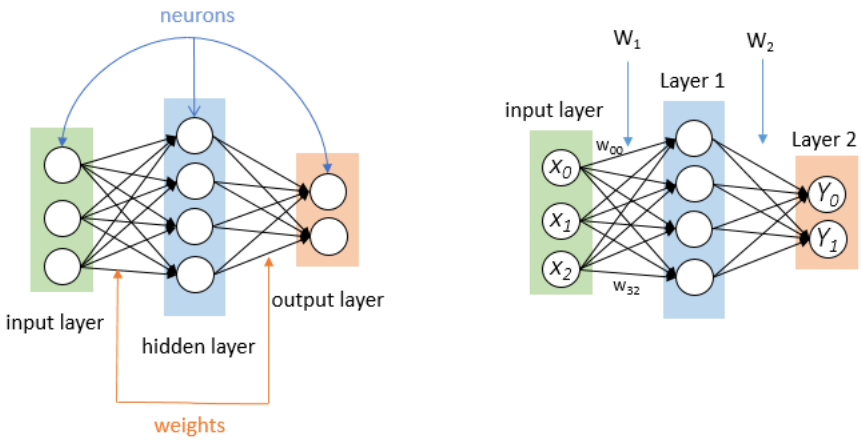


Figure 2.2: Fully connected neural network and terminology. On the left is an illustration of a two layer neural network (input layer, one hidden layer and output layer). On the right is an illustration of weight matrices associated with the two layer neural network. (Figure adapted from Li et al. [36] and Sze et al. [56]).

Figure 2.2 shows a simple neural network with two layers (without counting the input layer). This example is fully connected, which means that each neuron in a layer is connected with each neuron in the next layer. The weights of the synapses connecting input layer neurons to the Layer 1 neurons form a matrix (W_1) of the size 4×3. The weights of the synapses connecting Layer 1 neurons to the Layer 2 neurons form a matrix (W_2) of the size 2×4. In general, if m is the number of neurons in a layer and n is the number of neurons in the previous layer, the weight matrix is of dimensions $m \times n$. Layer 1 also has a bias vector B_1 of size 4×1. Layer 2 has a bias vector B_2 of size 2×1. This neural network has a total of 4 weight/bias matrices: W_1, B_1, W_2, B_2. A vector of the inputs to each neuron (as denoted in Figure 2.1) is denoted by X. A vector of the outputs of the output layer neurons in the output layer is denoted as Y. The activation function for neurons of both layers is denoted by g. The output vector Y for input vector X, is given by Equation 2.1. This equation defines the mapping function f, for the given input and output vectors. Equation 2.1 which applies to 2-layered network can be easily extended for larger networks.

$$Y = g(W_2 \cdot g((W_1 \cdot X + B_1)) + B_2) \tag{2.1}$$

During the training procedure, an optimization function (also referred to as cost function or loss function) is used. The loss ($\mathcal{L}$) is a function of W (set of weights and biases), input x and ground-truth output y. The training procedure tries to minimize the difference between the outputs provided with the training samples and the computed or predicted outputs. Weights are adjusted by an optimization method, such as gradient descent. For every weight $w \in W$, where W is the full parameter set, the gradient of the loss with respect to w is calculated during backpropagation. Weights and biases are updated as follows, where α is the learning rate and t denotes the epoch or step.

$$w^{t+1} = w^t - \alpha \left(\frac{\partial \mathcal{L}(W; x, y)}{\partial w} \right) \tag{2.2}$$

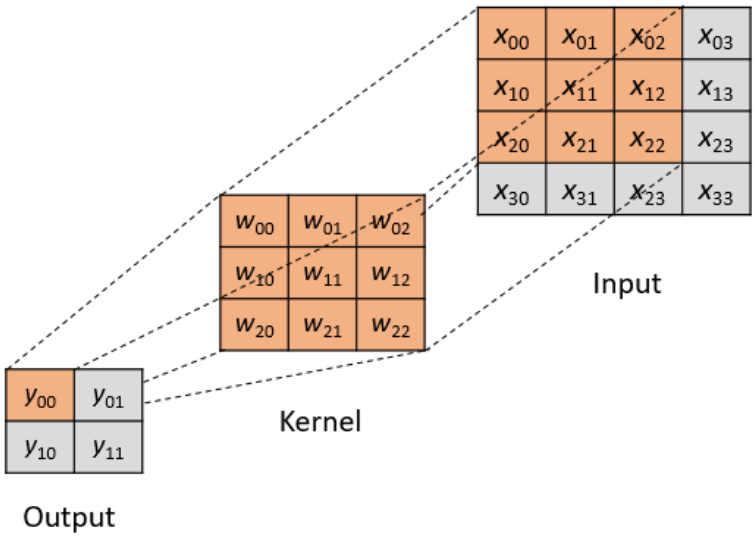


Figure 2.3: Illustration of convolution operation. A 3×3 kernel convolved across a 4×4 input image produces a 2×2 output. (Figure adapted from Mosser et al. [44].)

Convolutional neural networks (CNN) include convolutional (CONV) layers. Figure 2.3 shows a 2-dimensional convolution with a 3×3 kernel. The input is a 4×4 matrix. During the convolution operation, the kernel slides across the width and height of the input. The dot product between the kernel and the input overlapping with the filter is calculated. This produces an activation map consisting of the response to the kernel at every position on the input.

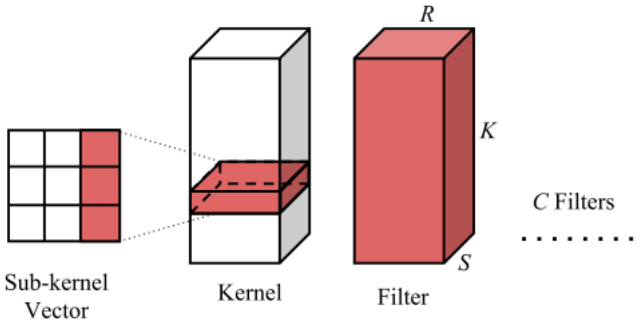


Figure 2.4: Illustration of vector, kernel and filter in context of a CONV layer. (Figure by Mao et al. [42][1].)

A CONV layer is made up of a set of filters. Consider a CONV layer made up of C filters (Figure 2.4). Let the number of channels (depth) of the input volume to this layer be K, and the convolution kernel be of size $R \times S$. The convolutional layer is a 4-dimensional tensor of shape $C \times K \times R \times S$ [42]. After convolution, each filter produces an activation map. These activation maps are stacked to get the output volume [36]. This CONV layer produces an output with C channels. The weight tensor of this layer will be of shape $C \times K \times R \times S$. A bias vector of shape $C \times 1$ is often used. The weights and biases are adjusted or learned during the neural network training.

[1] "Figure 2: Example of Sub-kernel Vector, Filter and Kernel" by Mao et al. [42] is licensed under CC BY 4.0. [1]

2.2 Pruning

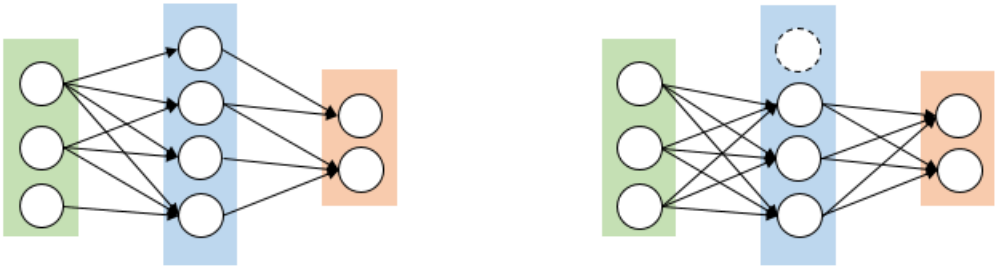


Figure 2.5: Pruning weights and neurons in a 2-layer fully-connected neural network. On the left is a depiction of pruned weights (connections between neurons). On the right is an illustration of pruning a neuron (all connections to and from this neuron are pruned).

Large neural networks are associated with ample expressive power, or the network's ability to approximate functions [41]. Pruning algorithms remove inessential parts of a neural network that is larger or more complex than might be required (Figure 2.5). This makes the network generalize better, avoiding over-fitting [49].

Reed [49] (1993) grouped early works into two categories. The first group based its methods on the sensitivity of the objective function to the removal (setting to zero) of a neural network parameter. The second group adds regularization terms to the objective function to favor simpler networks. LeCun et al. [34], in "Optimal Brain Damage" approximate the objective function using a Taylor series, and analyze the effect on the objective function caused by perturbing a parameter. Assuming pruning is done after the training converges to a local minimum, and ignoring higher order terms, they approximate the effect on the optimization function as proportional to the second derivative of the objective function with respect to the parameter. They followed an iterative pruning and re-training scheme. They defined the "saliency" of a parameter as its importance in terms of its effect on the training error. Chauvin [13], Weigend et al. [63] and Ji et al. [31] were some early works from the second

category of methods.

In recent deep learning research, Han et al. [25] (2015) developed a magnitude-based pruning scheme, without an additional cost of saliency computation. After training, in an iterative manner they pruned connections with parameters below a threshold and then retrained the network to recover from the pruning. They achieved a 9× to 13× reduction in the number of non-zero parameters, without degradation of accuracy.

Building on this approach, Guo et al. [21] (2016) proposed a pruning method called Dynamic Network Surgery (DNS). In addition to pruning, they introduce splicing, which enables them to recover a pruned connection if the parameter is found to be important to the network at a later time. Their parameter importance criterion is also magnitude based and they prune weights with small magnitudes.

It is important to note that Han et al. [25] as well as Guo et al. [21] use a pruning threshold that is local to a layer [50]. See et al. [51] (2016) further used the method proposed by Han et al. [25] while experimenting with local and global thresholds for pruning. A pruning threshold per weight class (such as layer) is referred to as a local threshold, whereas a single pruning threshold across the entire network is referred to as a global threshold. They showed that a global or "class-blind" approach of taking all network parameters, sorting them by magnitude and pruning the lowest magnitude parameters across the network worked best for their LSTM based Neural Machine Translation (NMT) model. In this work, we use a "class-blind" approach as well.

Srinivas et al. [55] (2017) proposed a "learning-based approach to pruning weights" that combines weight pruning and training, occurring simultaneously. They use binary gate variables that are drawn from a Bernoulli distribution and binarized (converted to 1 or 0). During training, weights as well as gate variables are updated using backpropagation. At the

end of training, each layer has a corresponding binary gate matrix. This is used to mask the weights, such that weights with zero values in the gate matrix are then zeroed out, resulting in the final pruned model.

Zhang et al. [67] propose a systematic weight pruning framework using "Alternating Direction Method of Multipliers" (ADMM). During training, they minimize the loss function but subject to a cardinality (number of non-zero elements) threshold for each layer.

Dong et al. [17] proposed "Layer-wise Optimal Brain Surgeon" (L-OBS), a deep neural network take on the "Optimal Brain Surgeon" (OBS) [27] method. OBS is similar to "Optimal Brain Damage" (OBD), but also includes the non-diagonal terms of the Hessian matrix (second derivatives) of the objective function with respect to the parameters, in its approximation. In L-OBS, parameters are pruned based on second order derivatives (just like OBS), but of a layer-wise error function.

The algorithm "Global Sparse Momentum SGD" (GSM) [16] (2019) modifies the regular momentum "Stochastic Gradient Descent" (SGD) for achieving sparsity or pruning. They also use Taylor series to estimate the change in loss value when a parameter is pruned. After dropping higher order terms, the saliency function (T) of parameter w from a set of parameters W, such that $w \in W$, is given by Equation 2.3. All parameters are split into two groups of important (highest saliency parameters) and relatively unimportant parameters. The unimportant parameters are not updated during the backward pass. Only weight decay is used to penalize large values. For the important parameter group, regular parameter updating using gradients with respect to the objective function as well as weight decay is used. l_2 regularization is used for penalizing large weights and enforcing weight decay.

$$T(x, y, w) = \left| \frac{\partial \mathcal{L}(W; x, y)}{\partial w} w \right| \tag{2.3}$$

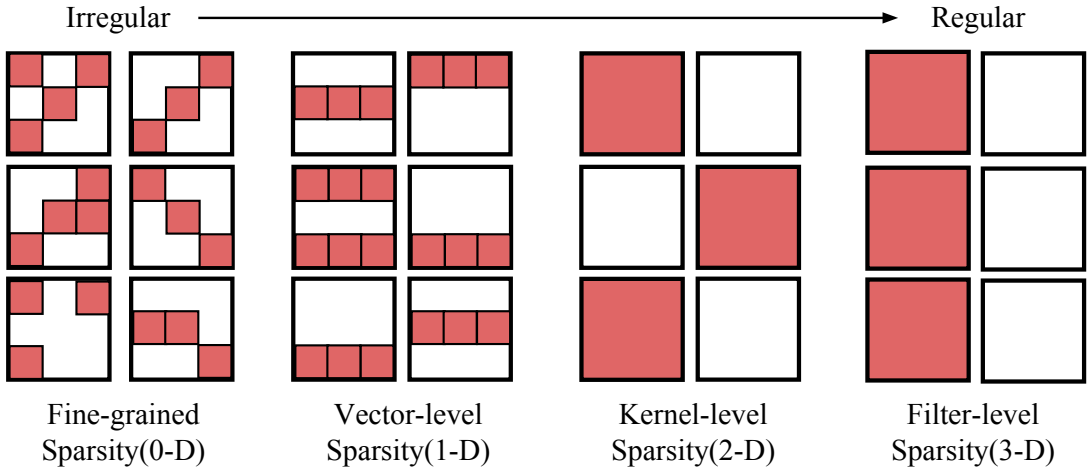


Figure 2.6: Pruning granularity in a CONV layer from fine-grained (irregular) pruning, to filter-level (regular) pruning. (Figure by Mao et al. [42].) [2]

The methods described so far achieve unstructured or fine-grained pruning, i.e., there is no pattern to the sparsity induced in the parameter tensors and matrices (as described in Section 2.1). Mao et al. [42] explored pruning granularity (Figure 2.6) in convolutional layers (Figure 2.4), from fine-grained sparsity, vector-level sparsity, kernel-level (2-dimensional) sparsity to filter-level (3-dimensional) sparsity. Fine-grained sparsity refers to the zeroing out of individual parameters. Vector-level sparsity is achieved by zeroing out entire rows of a kernel. Kernel-level sparsity refers to entire kernels (2-dimensional) of a filter being zero valued. Filter-level sparsity refers to an entire 3-dimensional filter being zero valued. Mao et al. [42] state that course-grained sparsity, such as filter-level or kernel-level sparsity, creates a smaller and denser version of the network architecture and therefore has direct hardware gains on general-purpose hardware. For fine-grained pruning, custom accelerators can be used, which is discussed in Section 2.5.

Li et al. [38] (2017) use the l_1-norm of a filter to determine its importance in a convolutional neural network. For the smallest l_1-norm filters, they prune the filters and their corre-

[2] "Figure 1: Different sparse structure in 4-dimensional weight tensor. Regular sparsity makes hardware acceleration easier." This figure by Mao et al. [42] is licensed under CC BY 4.0. [1]

sponding activation maps, followed by a fine-tuning or retraining step. Qin et al. [46] use a functionality-oriented (instead of magnitude-oriented) filter pruning approach and prune away filters with repetitive functionality. Wang et al. [60] incrementally assign different regularization factors to different importance based parameter groups, and state that increasing the regularization for unimportant parameters in small increments allows for the CNNs to transfer the expressiveness power to the unpruned parts of the network.

Recent works also apply more innovative pruning methods. Lin et al. [40] propose a pruning framework called "Runtime Neural Pruning" (RNP). They use reinforcement learning to learn a pruning scheme for a trained network. Unlike the fixed pruned models that are the output of most of the methods described above, RNP prunes the network according to the training set images and feature maps.

Methods presented in this thesis use the accumulated gradient of a parameter as its "saliency". We take inspiration from Golub et al. [19], who use accumulated gradients to learn only a subset of the neural network parameters, while keeping the rest of the parameters at their initial values. Unlike Golub et al. [19], we do not scale our accumulated gradients with the learning rate. Another difference is that we accumulate gradients for only the last mini-batch of inputs. Since we use shuffling, this mini-batch is different for every epoch and randomization is introduced. We prune at the lowest granularity, achieving fine-grained sparsity. Using absolute accumulated gradients is computationally easier than second-order derivative approaches. We group parameters according to their saliency into multiple clusters, using a global or "class blind" approach as described in Chapter 4. We take inspiration from Ding et al. [16] for our in-training quantization method (Section 4.3), where we also split parameters into two groups based on saliency. The less important parameters are quantized whereas the more important parameters are left at their full precision. In this method, parameters pruned in one epoch may be recovered in the next, as in DNS [21].

2.3 Quantization

Tung and Mori [57] define weight quantization as "the process of discretizing the range of weight values so that each weight can be represented using fewer bits". The number of bits needed to represent the discrete values, known as quantization levels is $\lceil \log_2 N \rceil$ for N quantization levels. Quantization is a way to perform coarse-grained approximation of neural network parameters, in an effort to achieve a balance between lowering the storage and computational requirements while achieving the most accuracy. The benefits of quantization include lower storage requirements and computational efficiency.

Traditionally, neural network training is done on 32-bit single-precision floating-point (FP32) [10] systems, and the weights and biases, referred to as parameters, are 32-bit floating-point numbers. Courbariaux et al. [15] found that very low precision formats work well, not only for running inference but also for training neural networks.

One approach to quantization is to use a narrower bit width data type, for example 16-bit half-precision floating-point format (FP16). Micikevicius et al. [43] propose a method to obtain neural networks with FP16 parameters. FP16 parameters are used during training, but a FP32 master copy is stored and updated in the backward pass. They scale the gradients to a FP16 representable range to avoid zeroing of small magnitudes gradients in the FP16 format. They train using FP16 arithmetic, with FP32 accumulation. Wang et al. [62] devised an 8-bit floating-point format (FP8) and demonstrated 8-bit floating-point format training.

Using fixed-point arithmetic instead of floating-point arithmetic circuits has benefits in terms of throughput and energy efficiency [22]. Hence, some quantization approaches focus on achieving fixed-point models. Gupta et al. [22] show that if a stochastic rounding scheme is used on 16-bit fixed-point arithmetic, the results are almost identical to FP32 arithmetic. Lin et al. [39] develop a method to convert a trained floating-point deep convolutional net-

work to a fixed-point model without loss of accuracy. Jacob et al. [30] propose a method to carry out neural network inference using integer-only arithmetic, which can work on more efficient integer arithmetic hardware. They also devised a training scheme where quantization is simulated in the forward pass ("fake quantization"), which is implemented in Google's TensorFlow Lite [5]. Our work also uses the fixed-point storage format, as described in Section 3.5.

Courbariaux et al. [14] proposed "BinaryConnect" (BC), a method of training with binary weights during the forward and backward pass. They retain full precision copies of the weights that are updated in the backward pass. Pushing the envelope further, Hubara et al. [29] propose a training method for "Binarized Neural Networks" (BNN) such that weights as well as activations are binarized (-1 or +1 values represented by 0/1 bit values) during the forward pass during training as well as inference. The authors present results for typical datasets on a fully connected architecture. For CIFAR-10, an accuracy of 90.10% was reported for BC, while an accuracy of 88.6% was reported for BNN. On AlexNet trained on Imagenet, they report a top-1 accuracy of 36.1% and a top-5 accuracy of 60.1%.

For BNNs, memory benefits are 32× compared to 32-bit typical representations. Since 32-bit floating-point multiply-accumulation operations (MAC) can be replaced by 1-bit XNOR-count operations, the XNOR dot product computational efficiency is higher than regular MAC operations (200×) [29]. The authors have also written a GPU kernel for binary matrix multiplication specifically to optimize BNNs. They demonstrate that with this optimization, the BNN trained on MNIST runs 7× faster without any loss of accuracy.

Li and Liu [37] propose Ternary Weight Networks (TWNs). They quantize weights to +1, 0 and -1 values, with a similar approach as Courbariaux et al. [14]. Zhu et al. [71] proposed Trained Ternary Quantization (TTQ). Instead of using the quantization levels of -1, 0, +1, they use different positive and negative quantization levels per layer, which are trainable

parameters. They quantize the weights to $-W_l^n$, 0, $+W_l^p$, where $-W_l^n$ and $+W_l^p$ are the negative and positive quantization levels for Layer l. Backpropagation updates the positive and negative quantization levels as well as the full precision weight copies. On both CIFAR-10 and ImageNet datasets, their models match or surpass the accuracy of full precision models.

Rastegari et al. [48] propose two efficient convolutional neural networks: "Binary-Weight-Networks" (BWN) and "XNOR-Networks". The first variation has binary weights (binarized to $+1$ and -1), while the second has binary weights as well as binary inputs to the convolutional layers. On AlexNet trained on ImageNet dataset, BWN achieved a top-1 accuracy of 56.8%, and top-5 accuracy of 79.4%. On the same benchmark architecture, XNOR-Net achieved a top-1 accuracy of 44.2%, and top-5 accuracy of 69.2%. Zhou et al. [69] propose "DoReFa-Net", which is a CNN that reduces the bit width of gradients, in addition to weights and activations during training. An AlexNet derivative of DoReFa-Net can quantize weights, actvations and gradients to 1, 2 and 6 bits, respectively while maintaining an accuracy of 46.1% on ImageNet.

Many systems such as BWN, TWN, TTQ, XNOR-Net and DoReFa-Net, do not apply the quantization to the first and last layers [56]. They keep the first and last layers at 32-bit floating-point precision, since significant accuracy degradation is observed on further quantization of these layers [69]. Mixed precision, or using different bit widths for different layers of a neural network, also proved to be a successful approach [23, 47]. Coming up with methods to determine the optimum bit width for each layer for layer-wise quantization is a research direction. Wang et al. [61] developed a framework for an automated selection of the bit widths for weights and activations depending on the target hardware architecture and the neural network architecture. "Adaptive Quantization", proposed by Zhou et al. [70], theoretically analyzes the effect of quantizing a layer on model accuracy. Based on that, they

propose an optimization technique to determine the optimal quantization bit width for each layer. For Alexnet and VGG-16 architectures, they achieve a 30-40% reduction in model size.

Zhou et al. [68] have a three step method of weight partitioning, group-wise quantization and then re-training. They divide the parameters into two groups (either randomly or based on the parameter values). The first group is quantized, and the second is retrained to compensate for the accuracy loss. On AlexNet, quantized to 5 bit width, they achieve a top-1 accuracy of 57.39% and top-5 accuracy of 80.46%, doing better than the full-precision reference models. Parameter partitioning is a common theme shared by Ding et al. [16], Golub et al. [19], Zhou et al. [68] across quantization as well as pruning. Our work also adopts group-wise quantization as described in Chapter 4.

Datatype conversion or bit width reduction approaches usually have a uniform-quantization approach in which they uniformly divide the range of weights or the range of the datatype into quantization levels. Zhou et al. [69] use the former approach and describe a quantization process that converts a real number $r \in [0, 1]$ to a b-bit number $r_q \in [0, 1]$ (Equation 2.4). Gupta et al. [22] use the latter approach for fixed-point conversion of floating-point numbers. We also use a uniform quantization technique as described in Section 2.3.

$$r_q = \frac{round((2^b - 1) \cdot r)}{(2^b - 1)} \tag{2.4}$$

Another approach is "weight sharing", where the idea is to represent a set of parameters using k distinct values. The parameters can then be represented using the index of the value ($\{0, 1, \ldots, k-1\}$) using $\lceil \log_2 k \rceil$ bits. Additionally, a codebook has to be maintained for the index-value mappings. Clustering is a popular choice for obtaining weight groups such that parameters are quantized to the respective cluster centroids. This approach is notably

used by Ullrich et al. [58] and Han et al. [24]. Similar to Han et al. [24], we use a k-means clustering approach as described in Chapter 4.

Non-uniform quantization approaches are also popular in recent works. Zhou et al. [68] convert parameters to zeros or powers of two. The quantization levels are $\{\pm 2^{n_1}, ..., \pm 2^{n_2}, 0\}$, where n_1 and n_2 are integers such that $n_2 \leq n_1$. This bounds the non-zero parameters to $[-2^{n_1}, -2^{n_2}]$ or $[2^{n_2}, 2^{n_1}]$, and parameters with absolute values less than 2^{n_2} are set to zero. The advantage of this method is that it converts multiplication operations to bit shift operations, which have a lower hardware cost. They can be implemented on custom hardware like FPGAs. This technique is also referred to as logarithmic quantization.

2.4 Hybrid Compression Methods

Some recent works combine compression methods. "Deep compression", a technique by Han et al. [24], uses a three stage method that combines pruning, quantization through weight sharing and Huffman coding to achieve storage efficiency. They compress models by 35× to 49× with no accuracy drop. Their magnitude based pruning strategy has an iterative pruning and training approach. Quantization is achieved by weight sharing on the fully trained network, followed by retraining to recover from accuracy loss. Finally, Huffman coding (variable-length lossless compression technique) further compresses the model.

Another hybrid technique is "Compression Learning by In-Parallel Pruning-Quantization" (CLIP-Q) by Tung and Mori [57]. This technique achieves neural network compression through in-parallel pruning and quantization while fine-tuning (retraining) a trained network. To achieve $p\%$ pruned parameters, after each training minibatch, the positive and negative pruning limits are determined such that $p\%$ of weights fall between these limits. The weights falling between limits are set to zero for the forward pass, and the rest of the weight range

is uniformly partitioned and quantized for a set bit budget. The pruned/quantized weights are used in the forward pass, but the full precision weights are retained and updated in the backward pass. The pruning limits and quantization levels change over the training epochs. They achieve a 51× compression rate on AlexNet trained on ImageNet, while preserving accuracy.

Our methods achieve quantization as well as pruning as a result of our quantization techniques. Without any special pruning efforts, we obtain significant pruning as a result of parameters being quantized to 0.

2.5 Hardware Implications of Pruning and Quantization

Network pruning creates sparse matrices of parameters, and quantization further optimizes the number of bits used to represent the parameters. These have two major advantages:

1. Compression or storage efficiency

2. Computational efficiency

DRAM accesses are expensive [28] and thus, model compression resulting in models that can fit in the on-chip SRAM or cache can save expensive memory accesses. Model compression results in computational efficiency and speedup on hardware. The computational benefits can be achieved in addition to or independent of the compression of neural networks.

Pruned neural networks can be stored more efficiently by using sparse matrix storage formats. Compressed sparse row (CSR) and compressed sparse column (CSC) are sparse matrix

storage formats that assume no structure to the sparsity. They store only the non-zero elements of a sparse matrix, with an additional overhead of indices. CSR stores the non-zero elements contiguously.

Following Barrett et al. [12], three vectors are used to store a sparse matrix in the CSR format:

val Vector of non-zero elements in the dense matrix, row-wise traversed

col_ind Column indices of val elements

row_ptr With one entry for each row, row_ptr[row] stores the index in the val vector where that row begins. The last entry in this vector comprises of the total number of non-zero elements in the matrix.

Figure 2.7 shows an example of how a sparse matrix can be stored in the CSR format. The non-zero elements {5,2,3,4} are stored in *val*. The column indices for {5,2,3,4} are {0,2,1,2}. These are stored in col_ind. The original matrix has 4 rows. The row_ptr has 5 elements. The first 4 elements correspond to the four rows and store the index (in the *val* vector) where the row begins. Row 0 begins with the non-zero element 5, which is at index 0 in *val*. Thus, row_ptr[0] is 0. Row 2 begins with the non-zero element 3, which is at index 2 in *val*. Thus, row_ptr[2] is 2. Since row 1 has no non-zero elements, row_ptr[1] is the same as the next row. The last element in the row_ptr stores the total number of non-zero elements.

For a sparse matrix with R rows, and C columns, i.e., $R \cdot C$ total elements, the row_ptr vector generally has $R + 1$ elements. If N is the total number of non-zero elements, the vectors val and col_ind have N elements each. Thus, instead of storing RC elements, we store $2N + R + 1$ elements, achieving significant storage benefit by using the CSR format. The CSC format is similar, with a column-traversal of the dense matrix and vectors val,

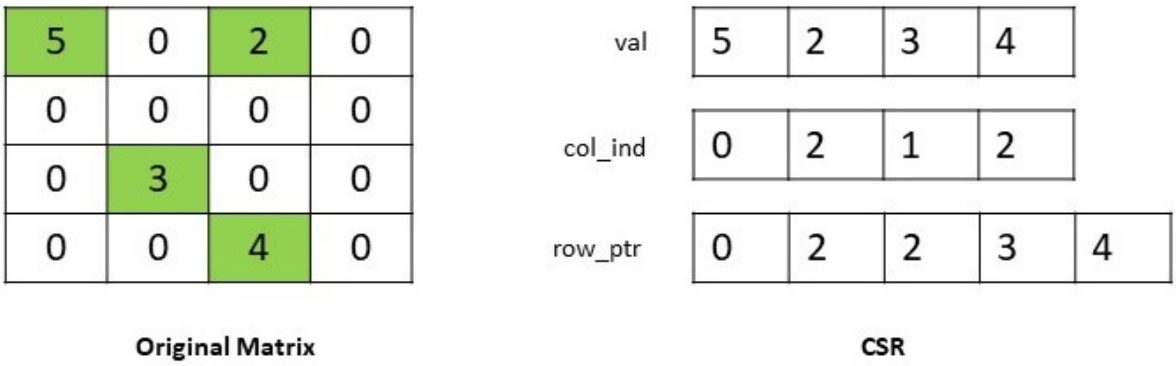


Figure 2.7: An example of the compressed sparse row (CSR) format. On the left is the dense matrix and on the right is the corresponding CSR representation.

row_ind and col_ptr.

Quantization can further add to the storage benefits of sparse matrix representation. Quantization reduces the number of bits per non-zero element. Further, Han et al. [24] and Tung and Mori [57] use a sparse encoding scheme such that the elements in the col_ind and row_ptr vectors (referred to as indices) can be stored with fewer bits as well.

During neural network inference in real time systems, matrix-vector multiplications and matrix-matrix multiplications are performed while processing one input at a time. Figure 2.8 shows how a convolution operation is implemented using matrix multiplications. Convolutional layers (CONV) as well as fully connected layers (FC) essentially consist of Multiply-Accumulate (MAC) operations [56]. To exploit the computational benefits of pruning, the zeroed out weights and the consequently redundant multiplication and addition operations have to be optimized for, or skipped, whether using dense matrix multiplication or sparse matrix multiplication.

Software libraries that give accelerated performance on CPU/GPU can be used to harness

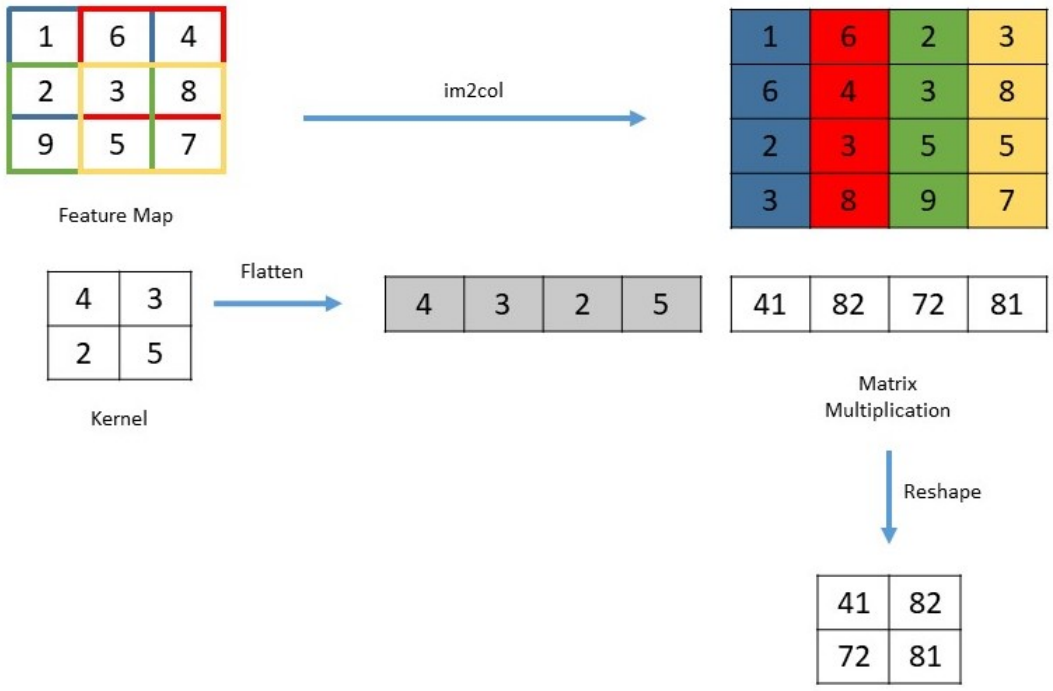


Figure 2.8: The convolution operation implemented as matrix multiplication. (Figure adapted from [53]).

computational benefits of sparsity. CUSPARSE (CUDA sparse matrix library) [7] is a high-performance sparse matrix linear algebra library by NVIDIA. It has basic linear algebra subroutines (BLAS) for sparse matrices for GPU acceleration. Similarly, Intel's MKL SPBLAS [8] (Math Kernel Library - Sparse Basic Linear Algebra Subprograms) is for CPU performance acceleration. These libraries support linear algebra for the CSR other sparse matrix formats. Han et al. [24] show that neural network pruning results into 3×, 3.5× and 4.2× speedup as well as 7×, 3.3× and 4.2× lower energy consumption on CPU, GPU and mobile GPU on average.

Recent works also propose custom hardware to exploit the hardware benefits of pruning and quantization. Han et al. [26] propose an "Energy Efficient Inference Engine" (EIE) compatible with their previous work [24]. It accelerates sparse matrix-vector multiplication with

weight sharing and leverages sparsity of weights and activations. Véstias et al. [59] propose an FPGA architecture for CNN inference that utilizes mixed fixed-point representation of activations and weights (varying bit width across layers), zero-skipping (multiplications with zeros are skipped), and run-time pruning of activations (setting to zero if activations are below a threshold). Zhang et al. [65] proposed Cambricon-X, a novel accelerator to utilize irregular sparsity of neural network models. On general purpose processors, instruction level optimizations can be performed. Sen et al. [52] proposed "Sparsity-aware Core Extensions" (SPARCE) a set of low-overhead micro-architectural and instruction set architecture extensions that dynamically perform instructions for zero valued operands. This is an active area of research, as part of the hardware-software co-design paradigm in deep learning.

Chapter 3

Number Representation and Storage

In this chapter, we discuss relevant numerical storage representations and discuss our approach of using the Q$0.FL$ fixed-point format to represent the weights and biases of a neural network.

3.1 Floating-point

Single-precision floating-point numbers are commonly stored using 1 sign bit, 8 exponent bits and 23 mantissa bits [10] as illustrated in Figure 3.1. The value is given by the following formula as compiled by Courbariaux et al. [15]:

$$value = (-1)^{sign} \times \left(1 + \frac{mantissa}{2^{23}}\right) \times 2^{(exponent-127)} \tag{3.1}$$

Single-precision floating-point numbers are typically used to represent the weights and biases (referred to as parameters) in neural networks.

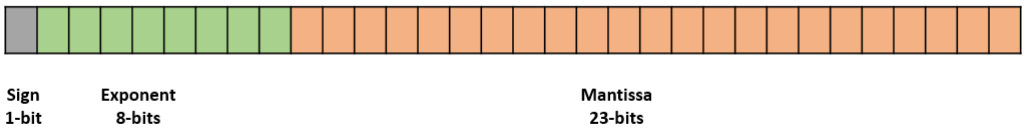


Figure 3.1: An illustration of the single-precision floating-point format (IEEE 754 standard [10]).

3.2 Fixed-point

As compiled in Courbariaux et al. [15], a fixed-point representation with B bits consists of 1 sign bit and $B-1$ mantissa bits. A scaling factor is used globally, which is equal to 2^{-FL}, where FL is the length of fractional part. The fractional length denotes the position of the radix point. This format has a much lower precision and range as compared to floating-point. Assuming two's complement format, the value for a B-bit fixed-point number can be computed as:

$$value = 2^{-FL} \cdot \left(-sign \cdot 2^{B-1} + \sum_{i=0}^{B-2} 2^i \cdot mantissa_i \right) \tag{3.2}$$

3.3 Q-format

In this work, we use the Q-format [9] to denote fixed-point signed representations. The format is represented as $\mathrm{Q}IL.FL$ where:

IL is the number of integer bits (excluding sign bit). [1]
FL is the number of fractional bits.

The bit width (B) required to represent $\mathrm{Q}IL.FL$ is $B = IL + FL + 1$.

The range of $\mathrm{Q}IL.FL$ is -2^{IL} to 2^{IL}–2^{-FL}, assuming a two's complement representation. The step size is 2^{-FL} [64].

[1] Gupta et al. [22] include the sign bit in their integer length (IL). We choose to exclude the sign bit.

3.4 Dynamic Fixed-point

As per Courbariaux et al. [15], dynamic fixed-point is essentially the same as fixed-point, except that the scaling factor is not global (Figure 3.2). There can be multiple groups of variables, with each group having one scaling factor. Courbariaux et al. [15] refer to it as a compromise between floating-point and fixed-point. They further mention that in a neural network, each layer's weights, biases, weighted sums, outputs (post-nonlinearity) and the respective gradients vectors and matrices can have different scaling factors. This makes dynamic fixed-point more suitable for deep learning than regular fixed-point. In this work, we quantize neural network parameters to a fixed number of distinct values between -1 and 1. For this range we do not need the flexibility of dynamic fixed-point representations.

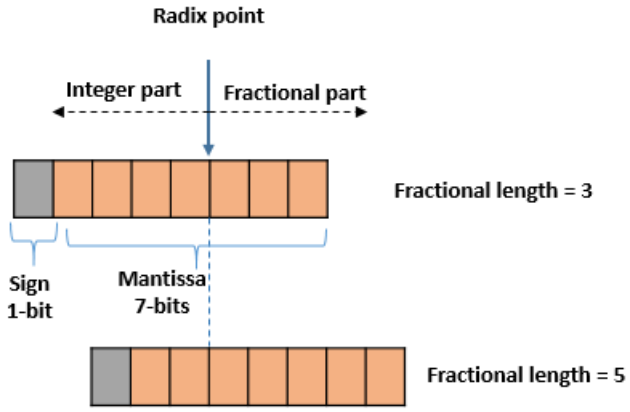


Figure 3.2: Dynamic fixed point representation with variable length of fractional part. (Figure adapted from Gysel [23].)

3.5 Number Representation Used

In this work, we convert the 32-bit floating-point weights and biases of neural networks to $Q0.FL$ fixed-point format. Since neural network parameters typically lie in the range $[-1, 1]$, they can be stored using only the sign bit (1 bit) and fractional bits (no bits used to represent the integer part). Table 3.1 illustrates the $Q0.FL$ fixed-point format.

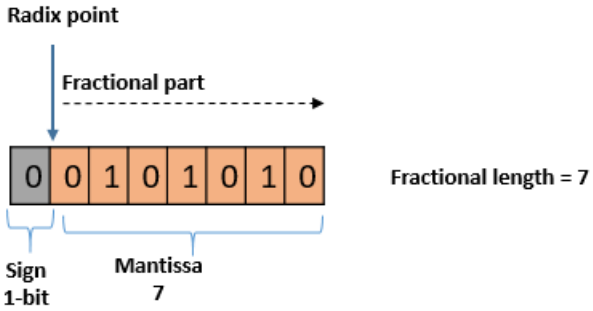


Figure 3.3: Our choice of number representation for neural network parameters.

Consider the decimal number 0.33. It can be represented as 0.01010100011110101110000 in a binary representation with 23 fractional bits and 0 integer bits i.e., $Q0.23$. To represent the same in $Q0.7$ using round-to-zero mode (described in Section 4.1), we truncate it to 0.0101010 or decimal 0.328125 (Figure 3.3), introducing an approximation error. The first bit (0) represents the sign (positive) and the rest of the bits are the fractional part. Since we use 2's complement format, -0.33 quantized to $Q0.7$ fixed-point format is 1.1010110. This is the storage format we assume in the rest of this work.

During inference, the quantized weights and biases are used, while the rest of the network is in the single-precision floating-point format. For the weights and biases, we simulate fixed-point values using floating-point data types because common deep learning frameworks support

Table 3.1: Binary representations and decimal values for $Q0.1$, $Q0.2$ and $Q0.3$ fixed-point format.

(a) $Q0.1$ fixed-point format

Binary Representation	Decimal Value
00	0.0
01	0.5
10	-1.0
11	-0.5

(b) $Q0.2$ fixed-point format

Binary Representation	Decimal Value
000	0.00
001	0.25
010	0.50
011	0.75
100	-1.00
101	-0.75
110	-0.50
111	-0.25

(c) $Q0.3$ fixed-point format

Binary Representation	Decimal Value
0000	0.000
0001	0.125
0010	0.250
0011	0.375
0100	0.500
0101	0.625
0110	0.750
0111	0.875
1000	-1.000
1001	-0.875
1010	-0.750
1011	-0.625
1100	-0.500
1101	-0.375
1110	-0.250
1111	-0.125

IEEE 754 single-precision floating-point format. For example, converting 0.33 (floating-point) to $Q0.5$ fixed-point format, would convert the decimal value to 0.3125. We store this converted value using a floating-point datatype. Other low precision arithmetic simulation packages like QPytorch [66] and Ristretto [23] (a framework for CNN approximation) use a similar approach.

Chapter 4

Methodology

In this chapter, we describe our methodology to achieve neural network compression through fixed-point quantization (and consequent pruning) of weights and biases. Our methods have been designed with the goal of:

1. Minimizing the number of non-zero parameters in the network (achieving sparsity)
2. Minimizing the number of bits used to represent each parameter (weight or bias)

Three methods of quantization are explored. The first consists of post-training quantization, the second consists of in-training quantization and the third is a combination of the first two. We also describe the fixed-point quantization technique used in our methods.

4.1 Quantization Method

In this section, we describe the algorithm used to quantize a 32-bit single-precision floating-point number to a $\mathrm{Q}0.FL$ fixed-point format, where FL is the number of fractional bits (as described in Section 3.5). The bit width B required to represent $\mathrm{Q}0.FL$ is $FL + 1$. The quantization levels range from -1 to $1–2^{-FL}$, with a step size of 2^{-FL}.

We quantize a number using a standard conversion scheme by Gupta et al. [22] as given in Equation 4.1. For our $\mathrm{Q}0.FL$ format, if the given number x falls outside the range of -1 to

$1–2^{-FL}$, it is clipped. If the number is within the range, a rounding scheme is used.

$$Convert(x, \mathbb{Q}0.FL) = \begin{cases} -1, & x \leq -1 \\ 1 - 2^{-FL}, & x \geq 1 - 2^{-FL} \\ Round(x, \mathbb{Q}0.FL), & otherwise \end{cases} \tag{4.1}$$

The IEEE Standard for Floating-Point Arithmetic [10] describes five rounding modes. Equations 4.2 to 4.6 define $\lfloor x \rfloor$ as the largest integer multiple of 2^{-FL} which is less than or equal to x, and $\lceil x \rceil$ as the largest integer multiple of 2^{-FL} which is greater than or equal to x.

1. Round-toward-positive

   This mode rounds a number to the quantization level towards positive infinity.

$$\text{Round-toward-positive}(x) = \lceil x \rceil \tag{4.2}$$

2. Round-toward-negative

   This mode rounds a number to the quantization level towards negative infinity.

$$\text{Round-toward-negative}(x) = \lfloor x \rfloor \tag{4.3}$$

3. Round-toward-zero

   This mode rounds a number to the quantization level towards zero.

$$\text{Round-toward-zero}(x) = sgn(x) \lfloor | \, x \, | \rfloor \tag{4.4}$$

4. Round-ties-to-even

   This mode rounds a number to the nearest quantization level. Numbers that lie midway

between two quantization levels are quantized to the even quantization level. For example, the quantization levels for $\mathbb{Q}0.1$ are $\{-1, -0.5, 0, 0.5\}$ and the even indexed levels are $\{-1, 0\}$ (assuming zero-based indexing). In the following equation, $\mathbb{Z}$ refers to the set of integers.

$$\text{Round-ties-to-even}(x) = \begin{cases} \lfloor x \rfloor, & \lfloor x \rfloor \leq x < \lfloor x \rfloor + 2^{-(FL+1)} \\ \lfloor x \rfloor + 2^{-FL}, & \lfloor x \rfloor + 2^{-(FL+1)} < x < \lfloor x \rfloor + 2^{-FL} \\ \lfloor x \rfloor, & x = \lfloor x \rfloor + 2^{-(FL+1)} \text{ and} \\ & \lfloor x \rfloor = k \cdot 2^{-(FL-1)}, k \in \mathbb{Z} \\ \lfloor x \rfloor + 2^{-FL}, & x = \lfloor x \rfloor + 2^{-(FL+1)} \text{ and} \\ & \lceil x \rceil = k \cdot 2^{-(FL-1)}, k \in \mathbb{Z} \end{cases} \tag{4.5}$$

5. Round-ties-to-away

   This mode rounds a number to the nearest quantization level. Numbers that lie midway between two quantization levels are quantized to the quantization level away from zero.

$$\text{Round-ties-to-away}(x) = \begin{cases} \lfloor x \rfloor, & \lfloor x \rfloor \leq x < \lfloor x \rfloor + 2^{-(FL+1)} \\ \lfloor x \rfloor + 2^{-FL}, & \lfloor x \rfloor + 2^{-(FL+1)} < x < \lfloor x \rfloor + 2^{-FL} \\ sgn(x) \lceil | x | \rceil, & x = \lfloor x \rfloor + 2^{-(FL+1)} \end{cases} \tag{4.6}$$

Of these rounding modes, round-toward-zero (truncation) prunes a parameter x if $|x| < 2^{-FL}$ and round-ties-to-even prunes a parameter x if $|x| \leq 2^{-FL-1}$ for our $\mathbb{Q}0.FL$ fixed-point format. These two modes prune away a greater range of numbers compared to the rest. For this reason we work with these two modes of rounding. Figure 4.1 shows a comparison of round-toward-zero and round-ties-to-even based quantization on LeNet-300-100 trained on

the MNIST dataset. Round-toward-zero (truncation) prunes more aggressively and loses accuracy faster, compared to round-ties-to-even. Round-ties-to-even prunes more conservatively and preserves accuracy to a greater extent.

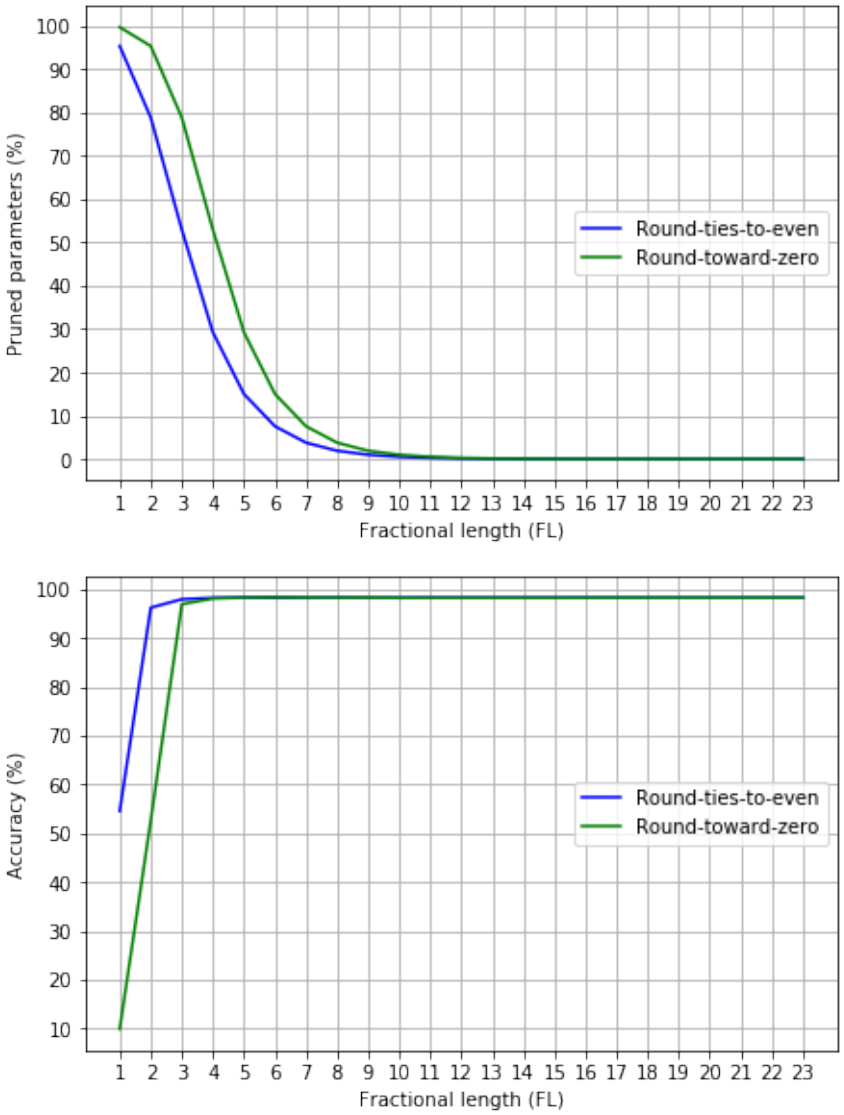


Figure 4.1: Comparison between quantization using round-toward-zero and round-ties-to-even modes on LeNet-300-100 on MNIST data.

Algorithm 4.1 describes the implementation of Equation 4.1, adapted from [66]. It uses a bit shifting and rounding approach to implement Equation 4.1. By multiplying the number by

2^{FL}, the rounding is converted to integer rounding. At this step, round-toward-zero returns the nearest integer towards zero. Round-ties-to-even returns the nearest integer (or even integer in case of a tie). Then the rounded integer is multiplied by 2^{-FL}. Overall, this achieves the same result as described by Equations 4.4 and 4.5.

Algorithm 4.1: Q0.*FL* FIXED-POINT QUANTIZATION OF FLOATING-POINT NUMBERS

Input:
num: Single-precision floating-point number
FL: Fractional length
rounding_mode: round-toward-zero or round-ties-to-even
Output:
num_q: Quantized to Q0.*FL* fixed-point format

$num_q = \texttt{round}((num \cdot 2^{FL}),\ rounding\_mode)$
$num_q = num_q \cdot 2^{-FL}$
$minimum = -1$
$maximum = 1 - 2^{-FL}$
if $num_q > maximum$ **then**
  $num_q = maximum$
if $num_q < minimum$ **then**
  $num_q = minimum$
return num_q

4.2 Method 1: Post-training Quantization

The first method to achieve neural network compression performs post-training quantization. Prior to quantization, during training we accumulate gradients for all the weights and biases (parameters), such that we have an accumulated gradient value for each parameter. The accumulated gradient of a parameter is the sum of the absolute values of the gradients (for one input batch) for that parameter during training. Since we use mini-batch optimization algorithms, we accumulate gradients for the last mini-batch for every epoch. We use shuffling, so the last mini-batch consists of different training samples at every epoch.

We denote the set of weights and biases as W. The loss is a function of W, input x and ground-truth output y. For every epoch, we accumulate the gradient for the last mini-batch. After N epochs, for every weight $w \in W$ there is a corresponding accumulated gradient g^{acc} given by Equation 4.7.

$$g^{acc} = \sum_{epoch=1}^{N} \left| \frac{\partial \mathcal{L}(W; x, y)}{\partial w} \right| \tag{4.7}$$

This algorithm is described in Algorithm 4.2. G is the set of accumulated gradients for trained weights W_t. We create a sorted list from G. We apply the k-means clustering algorithm to this list (discussed further in Section 4.5). The choice of k value is heuristic.

F is the fractional bit budget or the maximum allowable fractional length in the $\mathrm{Q}0.FL$ format. We assign a fractional length in the range $[1, F]$ to each cluster, based on the threshold accuracy (T_a). The threshold accuracy is the lower bound for the post-quantization accuracy on the validation dataset.

The parameters belonging to each cluster are quantized to the corresponding fractional length assigned to that cluster[1]. Since this is a post-quantization method, we use the round-ties-

to-even rounding mode to prune conservatively. We prioritize maintaining accuracy.

The basis of this method is our hypothesis that parameters with lowest accumulated gradients can tolerate the highest approximation errors. Hence, the cluster with lowest accumulated gradients is assigned the lowest fractional length and the highest accumulated gradient cluster is assigned the highest fractional length.

The inputs to this algorithm are:

k, number of clusters

F, fractional bit budget

T_a, threshold accuracy

[1]In implementation, we quantize all weights below the cluster boundary to the bit width assigned to the cluster, and repeat this for lower clusters. This gives the same result as separately quantizing weights for individual clusters.

Algorithm 4.2: Method 1: Post-training Quantization

Input: W_t: Trained parameters of the neural network
G: Accumulated gradients collected over the training process
val_set: Validation dataset
k: Number of clusters
T_a: Threshold (lower bound) for the accuracy of the quantized network
F: The fractional bit budget or the maximum fractional bit width to be assigned

Output: W_q: Quantized weights

```
// Cluster the sorted accumulated gradients
G_l ← get_sorted_list(G)
clusters ← kmeans(G_l, k)
previous ← F
W_q ← W_t
// Go over clusters, starting with the highest accumulated gradient
   cluster
reverse(clusters)
for c ∈ clusters do
    // Get the index of the last element in the cluster
    index ← get_cluster_boundary_index(c)
    threshold ← G_l[index]
    for b = F − 1 to 1 do
        if b > 1 and b ≥ previous then continue

        // Quantize all weights with accumulated gradients less than or
           equal to 'threshold' to b bits
        W ← quantize_weights(W_q, threshold, b, round_ties_to_even)
        accuracy ← test(W, val_set)
        if accuracy < T_a then break

    end
    // Quantize all weights with accumulated gradients less than or equal
       to 'threshold' to the assigned bit width
    if b == 1 then
        W_q ← quantize_weights(W_q, threshold, b, round_ties_to_even)
        previous ← b
    else
        // b+1 is the last bit width for which accuracy was within
           tolerance
        W_q ← quantize_weights(W_q, threshold, b + 1, round_ties_to_even)
        previous ← b + 1
    end
end
return W_q
```

4.3 Method 2: In-training Quantization

The second method to achieve neural network compression performs in-training quantization as described in Algorithm 4.3. After every training epoch, we sort and divide the accumulated gradients into two groups. The first group consists of the highest accumulated gradients. The size of this group is denoted by *top* and the weights corresponding to this group are left at their full-precision values. The rest of the weights are quantized to the quantization fractional length (F_q).

Our intuition is that the network tries to recover from the quantization error introduced after every training epoch. This recovery is achieved by adjusting parameters through back-propagation in the next training epoch. We use the round-toward-zero rounding mode as we want to prune aggressively, forcing the network parameters to be adjusted. We do not quantize the most important parameters (based on their accumulated gradients) to limit the quantization error introduced. The neural network parameters trained by this method are observed to be more quantization friendly as described in Chapter 5.

The inputs to this algorithm are:

N, number of epochs

***top*,** number of parameters with the highest accumulated gradients to be kept at full-precision

F_q, fractional bit width for quantization

F_q is the fractional length that we choose for the in-training quantization. A rule of thumb we followed to select the F_q value is illustrated in Figure 4.2. We quantize the trained network with round-toward-zero (truncation) with varying FL (starting with 1) and look at the accuracies (on validation set). The accuracy should increase as we increase the fractional

length. We follow the upward accuracy trend and use the fractional length such that the accuracy is observed to decrease in the next FL step. For LeNet-300-100 trained on MNIST, we select F_q as 6.

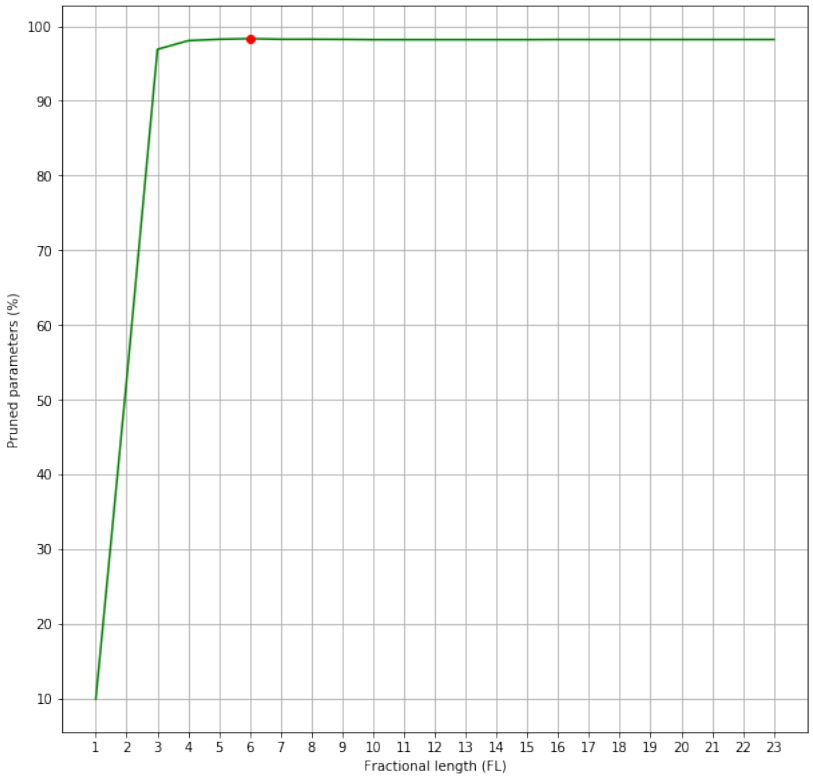


Figure 4.2: Selection of F_q, based on round-toward-zero (truncation) with varying FL.

Algorithm 4.3: METHOD 2: IN-TRAINING QUANTIZATION

Input: N: Number of epochs for training
top: Number of parameters with the highest accumulated gradients to be kept at full-precision
train_set: Training dataset
F_q: Fractional bit width for quantization, integer bit width set to 0

Output: W_t: Trained weights
G: Accumulated gradients

for *epoch* $= 1$ ***to*** N **do**
    `// Get the trained weights at this epoch`
    $W \leftarrow$ `train(`W, *train_set*`)`

    `// Accumulate gradients for the last mini-batch`
    for $w, g \in W, G$ **do**
        $g \leftarrow g + \left|\frac{\partial \mathcal{L}(W; train\_set)}{\partial w}\right|$
    end

    `// Sort accumulated gradients, and get quantization threshold`
    $G_l \leftarrow$ `get_sorted_list(`G`)`
    threshold $\leftarrow G_l$[*S - (top + 1)*]

    `// Weights with accumulated gradients less than or equal to threshold will be quantized to` F_q `fractional length`
    $W_q = \{w \in W \mid g \leq threshold\}$
    $W \leftarrow$ `quantize_weights(`W_q, F_q, *round_toward_zero*`)`
end
$W_t \leftarrow W$
return W_t

4.4 Method 3: Combined Quantization

The third method to achieve neural network compression is a combination of the in-training and post-training quantization methods described in sections 4.3 and 4.2. The combination of both techniques achieves the best results as described in Chapter 5. Figure 4.3 illustrates the three quantization methods.

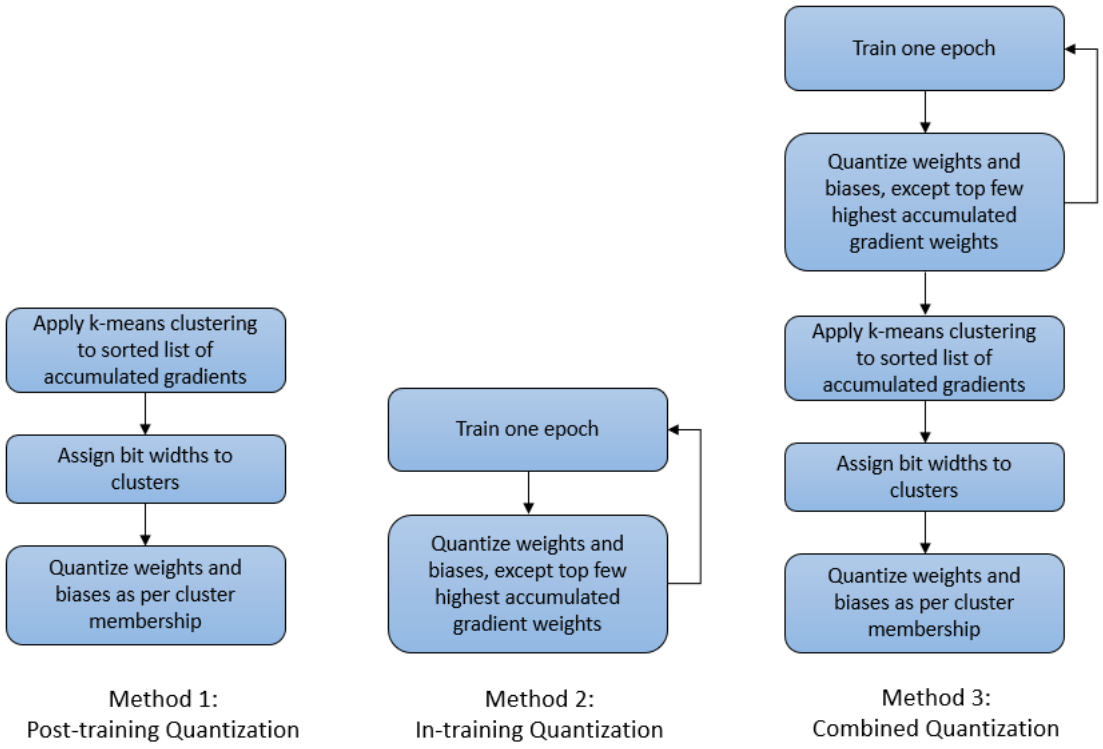


Figure 4.3: Illustration of the three quantization methods explored in this work.

4.5 K-means Clustering

Figure 4.4 shows a visualization of the sorted accumulated gradients for LeNet-300-100, trained on MNIST dataset. Of the 266,610 parameters, 266,230 parameters (99.85%) have an accumulated gradient less than or equal to 0.137. As the accumulated gradient value increases, the statistical frequency decreases drastically. We use clustering to segment the accumulated gradient list. Due to the monotonically decreasing nature of the accumulated gradients list, we do not use distribution based methods. Park et al. [45] solve a similar problem by creating equal interval clusters to begin with and performing an incremental search to modify cluster boundaries using bisection method. We choose to use k-means clustering as has been done by Han et al. [24] and Ullrich et al. [58].

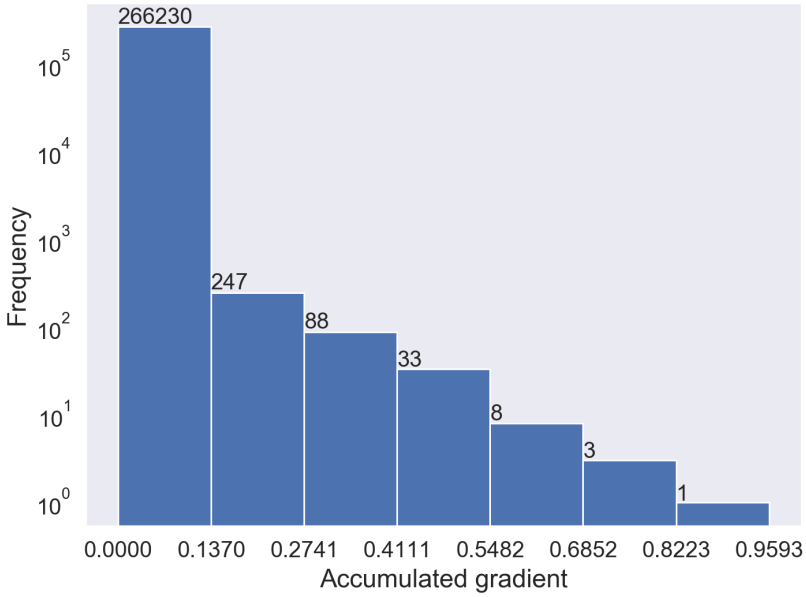


Figure 4.4: Distribution of accumulated gradients for LeNet-300-100 trained on MNIST dataset.

The k-means algorithm separates given data points into k clusters such that the total squared distance between each point and its closest center is minimized. We use the k-means++ initialization technique which is a randomized seeding technique, that is $O(\log k)$-competitive

with the optimal clustering [11].

As shown in Figure 4.5, we use k-means clustering to essentially segment the 1-dimensional accumulated gradient data. The color bar indicates the cluster membership percentage for each cluster. The clusters with lower accumulated gradient values have higher membership. Cluster size decreases as the accumulated gradient value increases.

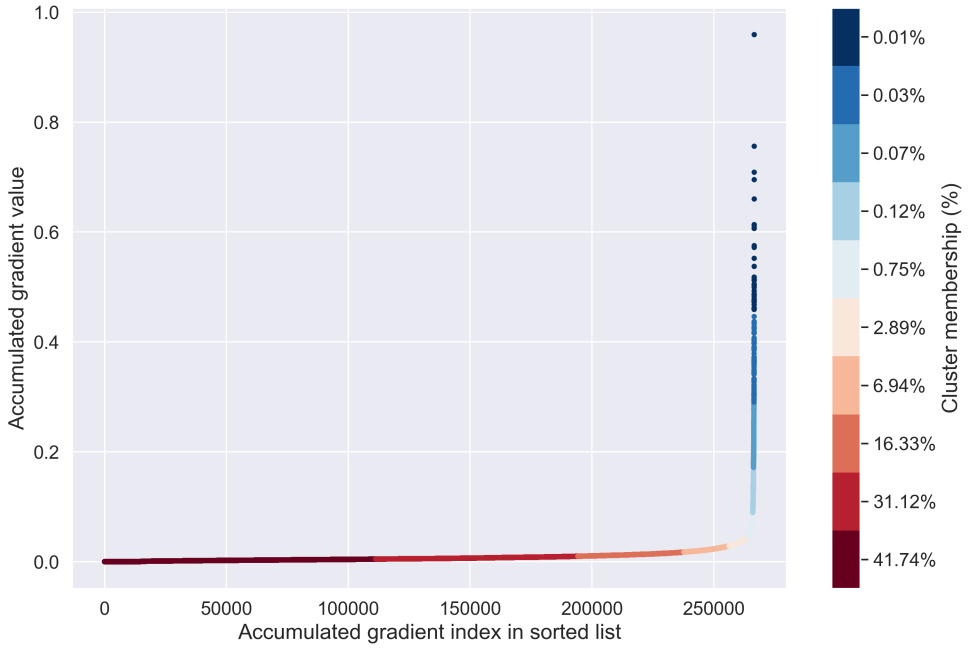


Figure 4.5: Visualization of k-means clustering of accumulated gradients for LeNet-300-100 trained on MNIST dataset.

4.6 Implementation Details

- We use PyTorch [4] to implement, train and test neural network benchmarks. The trained weights and biases of each layer of a neural network are stored as tensors of class "torch.Tensor". Weights and biases are stored as 32-bit floating-point numbers (type "torch.float32").

- For fixed-point quantization according to Algorithm 4.1, we use the NumPy library [3]. PyTorch tensors can be accessed as NumPy arrays.

- In implementation of Algorithm 4.2, we use KMeans [2] from the scikit-learn package of SciPy with k-means++ initialization of centroids. The algorithm is run 3 times, with up to 20 iterations per run. The output with the best inertia (within-cluster sum-of-squares) is returned.

Chapter 5

Results and Discussion

In this section, we present results for our methods. Our choice of networks is consistent with Han et al. [24]. We present results on MNIST dataset with LeNet-300-100 and LeNet-5 networks [35], and on CIFAR-10 [32] dataset with AlexNet [33] and VGG-16 [54].

5.1 Networks and Datasets Used

In this section, we provide a brief description of the networks and datasets for which we present results.

- The MNIST [35] dataset consists of 28×28 gray scale images of handwritten digits with 10 classes. The training set consists of 60,000 examples, and the test set of 10,000 examples.

- The CIFAR-10 [32] dataset consists of 32×32 color images with 10 classes. The training set consists of 50,000 images and the test set consists of 10,000 images. Overall, the dataset consists of 6,000 images of each class.

- For both MNIST and CIFAR-10 datasets, we split the training set into a training set and validation set such that these subsets have the same class distribution as the original training set. For MNIST, the training set consists of 54,000 images while the

validation set consists of 6,000 images. For CIFAR-10, the training set consists of 45,000 images while the validation set consists of 5,000 images.

- LeNet-300-100 [35] is a fully connected neural network. It has two hidden layers, with 300 and 100 neurons each. Since we are using the MNIST dataset with 10 classes, the output layer has 10 neurons. This network consists of 266,610 (~267k) parameters.
- LeNet-5 [35] is a convolutional neural network. It has two convolutional layers (5×5 kernels, with 20 and 50 filters, respectively) and two fully connected layers (500 and 10 neurons each). Convolutional layers are followed by max pooling layers with 2×2 kernels. This network consists of 431,080 (~431k) parameters.
- AlexNet [33] is a 8-layer network with 5 convolutional layers and 3 fully connected layers as shown in Figure 5.1. This network consists of 56,909,194 (~57M) parameters.
- VGG-16 [54] is a 16-layer network with 13 convolutional layers and 3 fully-connected layers as shown in Figure 5.2. This network consists of 134,301,514 (~134M) parameters.

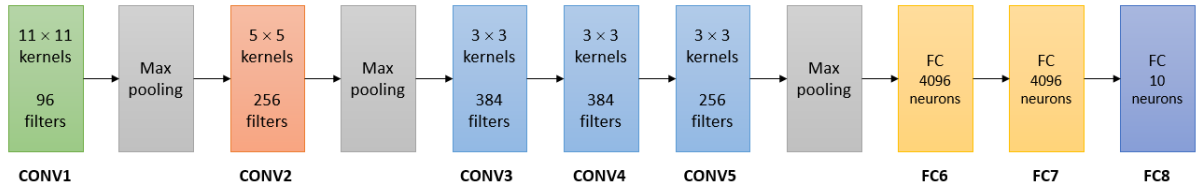


Figure 5.1: The AlexNet architecture. AlexNet consists of 5 convolutional layers and 3 fully-connected layers.

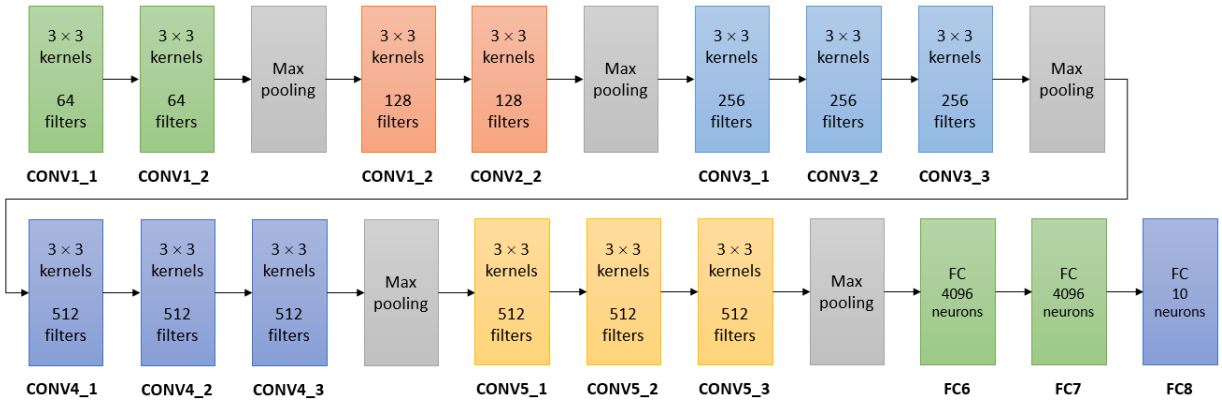


Figure 5.2: The VGG-16 architecture. VGG-16 consists of 13 convolutional layers and 3 fully-connected layers. (Figure adapted from Golub [18].)

5.2 Evaluation Metrics

We evaluate the performance of our methods based on the following metrics:

1. Percentage of non-zero parameters

   The percentage of parameters that have a non-zero value (out of the total neural network parameters) is an indication of pruning efficiency.

2. Compression ratio

   This metric indicates the combined efficiency of pruning as well as quantization. We calculate these figures assuming that the Compressed Sparse Row (CSR) format (Figure 2.7) is being used to store the quantized network parameters. Once the network parameter matrices are converted into CSR format, we proceed to calculate the theoretical minimum number of bits required to store the complete CSR matrices, which in turn lets us calculate the theoretical maximum compression ratio that could be achieved. For CSR conversion, we use the CSR class from the SciPy library [6].

   Each non-zero parameter in the *val* vector is stored using b_w bits (including the sign bit). This is the highest bit width assigned to a cluster by Method 1 or 3. In case of Method 2, this is the quantization bit width. We assume storage of all the parameters with a uniform bit width, padding zeros for parameters quantized to lower bit widths.

   Each index entry in the *col_ind* and *row_ptr* vectors is stored using b_i bits. To minimize the number of bits required per entry for the *col_ind* and *row_ptr*, we prepend these vectors with a 0 and store the difference between consecutive values, instead of storing the absolute values. We determine bit width based on the maximum and minimum entry in the *col_ind* and *row_ptr*.

   For N_o parameters stored using b_o bits in the original network, N non-zero parameters

after quantization and R rows in the M parameter matrices, the compression ratio is calculated as follows:

$$CR = \frac{N_o \cdot b_o}{N \cdot b_w + (N + R + M) \cdot b_i} \tag{5.1}$$

For parameter tensors, we reduce the number of dimensions before conversion to the CSR format. For example, a convolution layer with a $20 \times 1 \times 5 \times 5$ weight tensor, we convert it to 100×5.

3. Accuracy after quantization

   The post-quantization accuracy indicates the effects of the quantization error introduced by our methods. This accuracy is calculated using 32-bit floating point arithmetic, with our quantized neural network parameter values.

5.3 LeNet-300-100 on MNIST Dataset

We achieved a baseline accuracy of **98.47%** for LeNet-300-100 trained on the MNIST dataset. The network was trained for 50 epochs using the Adam optimizer with a learning rate of 0.001. The batch size was set to 120. We use the same initialization of weights for all three methods to get a fair comparison.

5.3.1 Method 1

Figure 5.3 illustrates the results obtained over a range of k and threshold accuracy values. We achieve a compression ratio of 2.82 to 7.62, with a post-quantization accuracy of 98.38% to 97.24%. The choice of k and threshold accuracy depends on the allowable accuracy degradation.

Table 5.1 shows the result for the configuration that yields the best compression. Quantizing all the parameters to $\mathbb{Q}_{0.3}$ results in 47.04% non-zero parameters, with an accuracy of 97.87%. Quantizing all the parameters to $\mathbb{Q}_{0.2}$ results in 21.15% non-zero parameters, with an accuracy of 96.21%. Our method achieves 27.86% non-zero parameters with an accuracy of 97.24%. Table 5.2 shows the configuration that yields the best accuracy.

Table 5.1: Method 1: Best compression achieved for LeNet-300-100 on MNIST dataset.

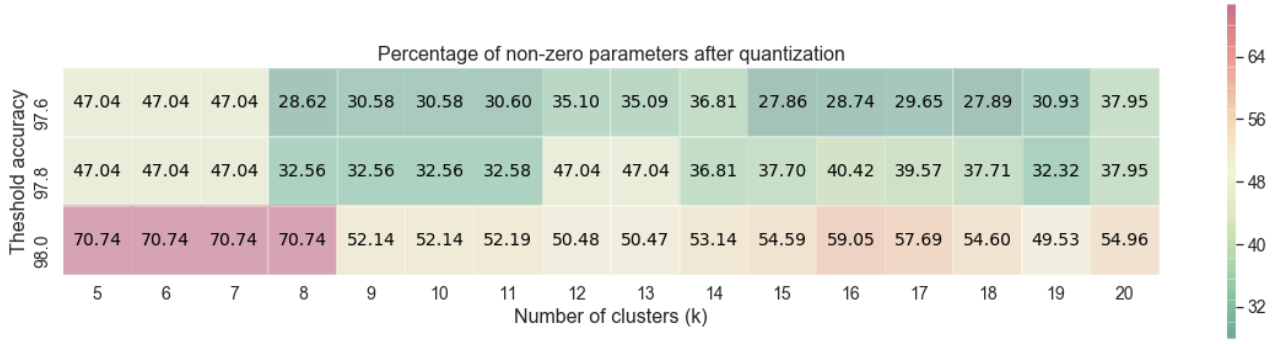
Number of clusters (k): 15
Threshold accuracy (T_a): 97.6
Fractional bit budget (F): 6
Cluster membership (%): 0.002, 0.011, 0.022, 0.038, 0.058, 0.089, 0.29, 0.952, 2.172, 4.051, 7.047, 12.313, 18.61, 28.773, 25.571
FL assignment: 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 2
Bit width per weight (b_w): 4
Bit width per index (b_i): 11

Baseline Accuracy	Post-Quantization Accuracy	Remaining/Original Parameters	% Non-zero Parameters	Compression Ratio
98.47	97.24	74286/266610	27.86	7.62

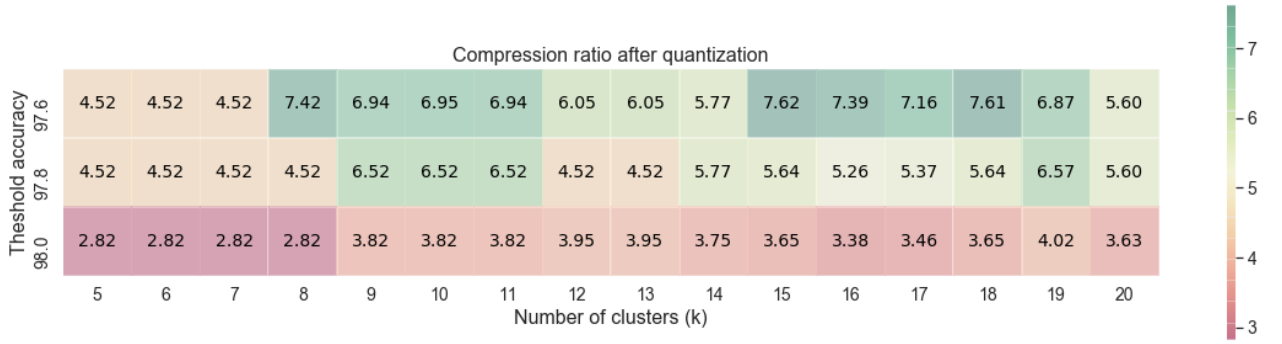
Table 5.2: Method 1: Best accuracy achieved for LeNet-300-100 on MNIST dataset.

Number of clusters (k): 5
Threshold accuracy (T_a): 98.0
Fractional bit budget (F): 6
Cluster membership (%): 0.03, 0.13, 3.95, 22.85, 73.04
FL assignment: 4, 4, 4, 4, 4
Bit width per weight (b_w): 5
Bit width per index (b_i): 11

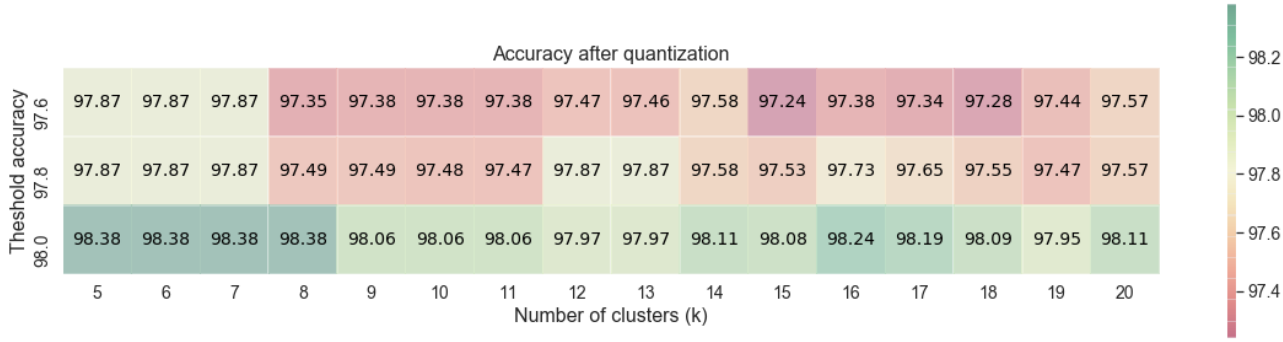
Baseline Accuracy	Post-Quantization Accuracy	Remaining/Original Parameters	% Non-zero Parameters	Compression Ratio
98.47	98.38	188590/266610	70.74	2.82



(a) The percentage of non-zero parameters after post-training quantization. A low percentage of non-zero parameters results into high compression.



(b) The compression ratio after post-training quantization.



(c) The accuracy after post-training quantization. The baseline accuracy is 98.47%.

Figure 5.3: Method 1 results for LeNet-300-100 on the MNIST dataset.

5.3.2 Method 2

We trained LeNet-300-100 on the MNIST dataset using Method 2, for 80 epochs. We varied *top* from 1000 to 5000 with a step size of 1000. The fractional bit width (F_q) for quantization was set to 6. Method 2 returns a partially quantized network. Table 5.3 shows results for the trained networks quantized to Q0.4 format, with round-ties-to-even rounding.

Table 5.3: Method 2: LeNet-300-100 on MNIST dataset, quantized to Q0.4, with weight bits (b_w) = 5, and index bits (b_i) = 11.

Top	Baseline Accuracy	Post-Quantization Accuracy	% Non-zero Parameters	Compression Ratio
1k	98.47	98.23	8.94	22.08
2k	98.47	98.31	9.32	21.20
3k	98.47	98.07	9.07	21.77
4k	98.47	98.22	9.82	20.14
5k	98.47	98.22	10.23	19.33

5.3.3 Method 3

We run Method 1 on the trained networks obtained using Method 2 as described in Subsection 5.3.2. Figure 5.4 illustrates the results obtained over a range of k and *top* values, for a threshold accuracy of 98.00%. We achieve a compression ratio of 28.57 to 57.22, with a post-quantization accuracy of 98.27% to 98.04%. The choice of k and *top* depends on the allowable accuracy degradation.

Table 5.4 shows the result for the configuration that yields the best compression. Table 5.5 shows the configuration that yields the best accuracy.

Table 5.4: Method 3: Best compression achieved for LeNet-300-100 on MNIST dataset.

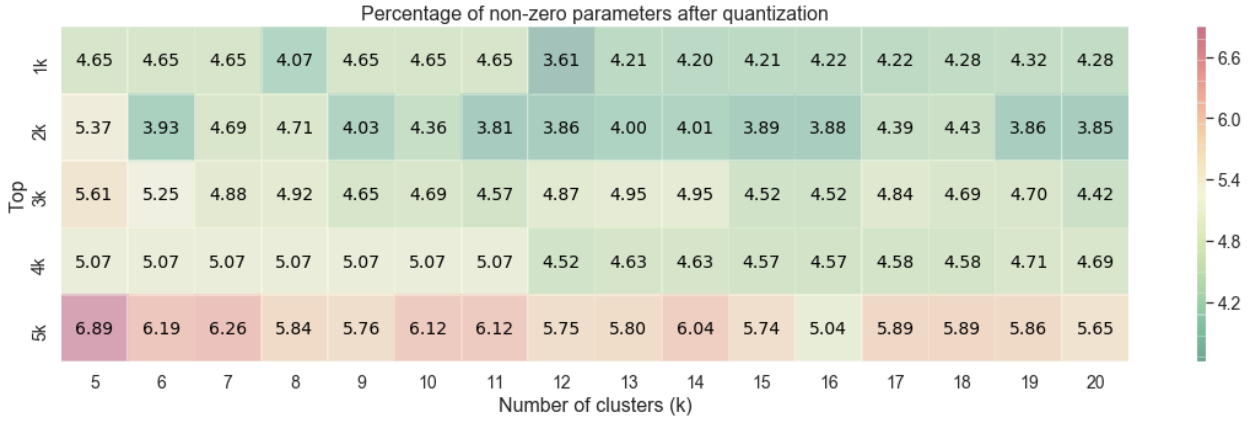
Number of clusters (k): 12
Threshold accuracy (T_a): 98.0
Top: 1000
Fractional bit budget (F): 6
Cluster membership (%): 0.011, 0.038, 0.081, 0.315, 0.876, 1.896, 3.373, 5.617, 9.117, 15.516, 19.514, 43.645
FL assignment: 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 2
Bit width per weight (b_w): 4
Bit width per index (b_i): 11

Baseline Accuracy	Post-Quantization Accuracy	Remaining/Original Parameters	% Non-zero Parameters	Compression Ratio
98.47	98.04	9632/266610	3.61	57.22

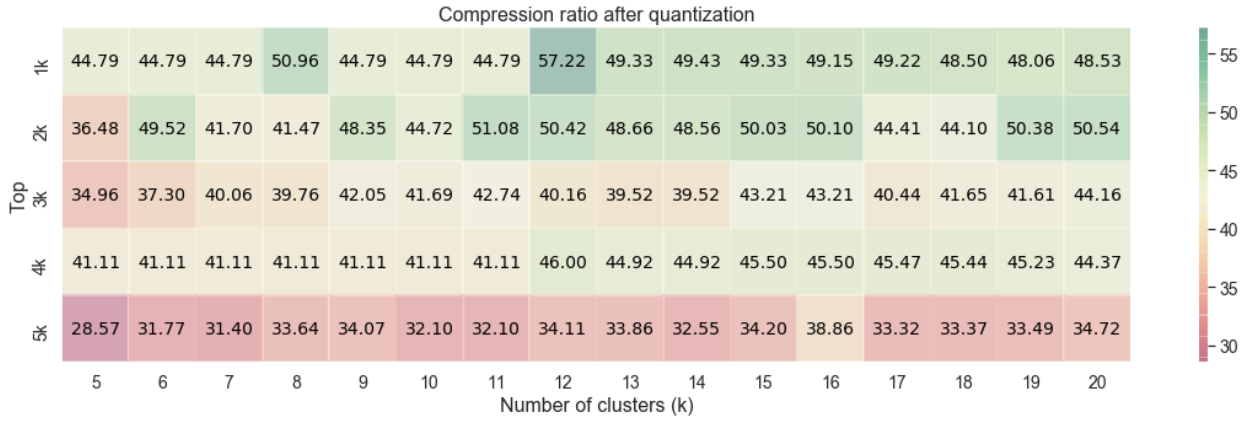
Table 5.5: Method 3: Best accuracy achieved for LeNet-300-100 on MNIST dataset.

Number of clusters (k): 5
Threshold accuracy (T_a): 98.0
Top: 5000
Fractional bit budget (F): 6
Cluster membership (%): 0.091, 1.838, 6.743, 25.237, 66.091
FL assignment: 4, 4, 4, 4, 3
Bit width per weight (b_w): 5
Bit width per index (b_i): 11

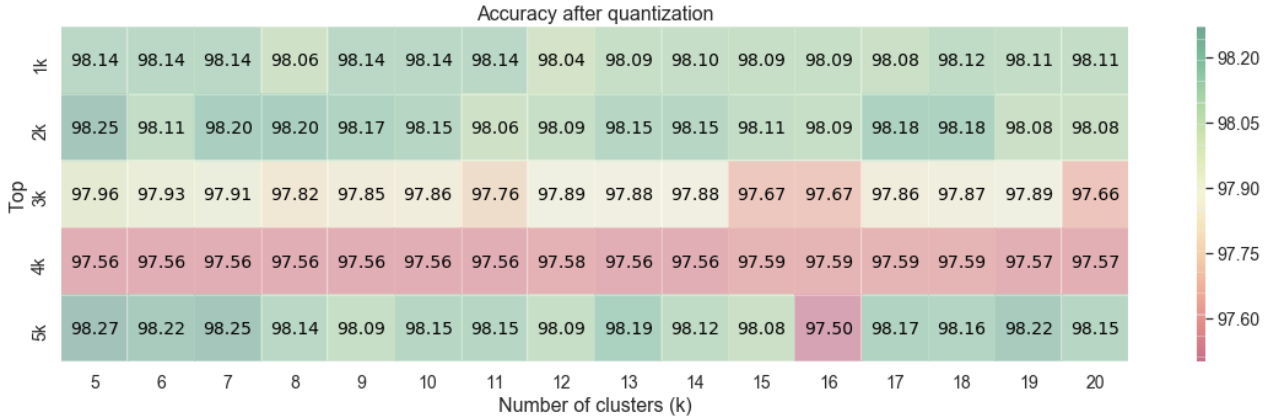
Baseline Accuracy	Post-Quantization Accuracy	Remaining/Original Parameters	% Non-zero Parameters	Compression Ratio
98.47	98.27	18371/266610	6.89	28.57



(a) The percentage of non-zero parameters after post-training quantization. A low percentage of non-zero parameters results into high compression.



(b) The compression ratio after post-training quantization.



(c) The accuracy after post-training quantization. The baseline accuracy is 98.47%.

Figure 5.4: Method 3 results for LeNet-300-100 on the MNIST dataset. The threshold accuracy is set to 98.00%

5.4 LeNet-5 on MNIST Dataset

We achieved a baseline accuracy of **99.09%** for LeNet-5 trained on the MNIST dataset. The network was trained for 50 epochs using the Adam optimizer with a learning rate of 0.001. The batch size was set to 120. We use the same initialization of weights for all three methods to get a fair comparison.

5.4.1 Method 1

Figure 5.5 illustrates the results obtained over a range of k and threshold accuracy values. We achieve a compression ratio of 5.28 to 10.87, with a post-quantization accuracy of 98.97% to 98.39%. The choice of k and threshold accuracy depends on the allowable accuracy degradation.

Table 5.6 shows the result for the configuration that yields the best compression. Quantizing all the parameters to $\mathbb{Q}0.4$ results in 70.63% non-zero parameters, with an accuracy of 99.12%. Quantizing all the parameters to $\mathbb{Q}0.2$ results in 24.39% non-zero parameters, with an accuracy of 89.49%. Our method achieves 17.49% non-zero parameters with an accuracy of 98.39%. Table 5.7 shows the configuration that yields the best accuracy.

Table 5.6: Method 1: Best compression achieved for LeNet-5 on MNIST dataset.

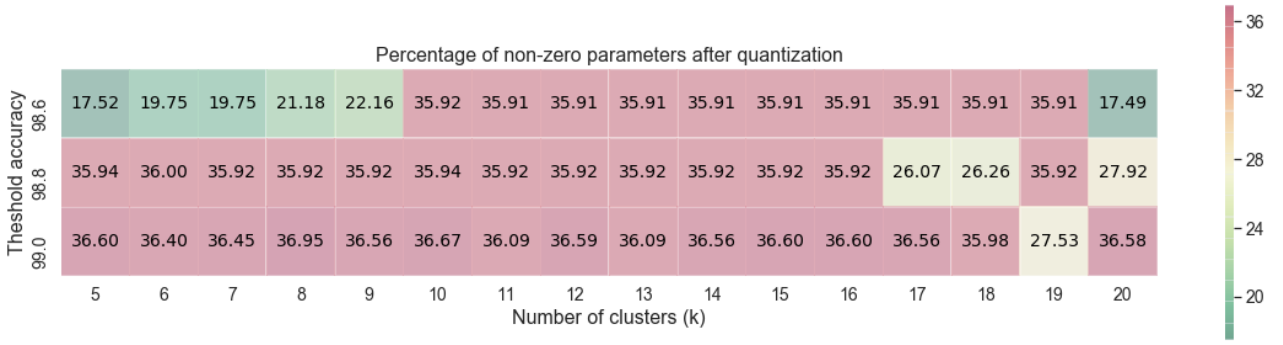
Number of clusters (k): 20
Threshold accuracy (T_a): 98.6
Fractional bit budget (F): 7
Cluster membership (%): 0.002, 0.009, 0.017, 0.032, 0.039, 0.06, 0.08, 0.139, 0.263, 0.469, 0.617, 0.879, 1.093, 1.348, 1.874, 3.376, 6.786, 12.561, 21.46, 48.897
FL assignment: 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 2, 2, 2, 2
Bit width per weight (b_w): 5
Bit width per index (b_i): 11

Baseline Accuracy	Post-Quantization Accuracy	Remaining/Original Parameters	% Non-zero Parameters	Compression Ratio
99.09	98.39	75412/431080	17.49	10.87

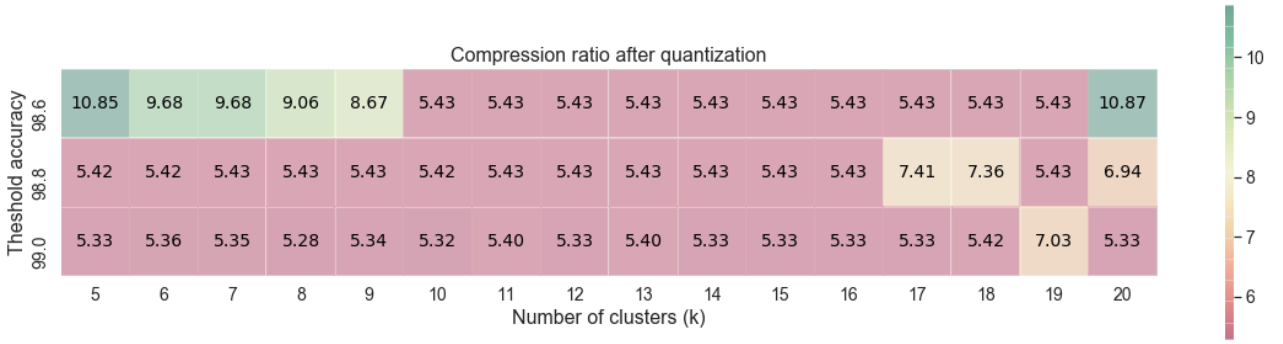
Table 5.7: Method 1: Best accuracy achieved for LeNet-5 on MNIST dataset.

Number of clusters (k): 8
Threshold accuracy (T_a): 99.0
Fractional bit budget (F): 7
Cluster membership (%): 0.023, 0.070, 0.149, 0.645, 1.868, 3.597, 16.196, 77.453
FL assignment: 4, 4, 4, 4, 4, 3, 3, 3
Bit width per weight (b_w): 5
Bit width per index (b_i): 11

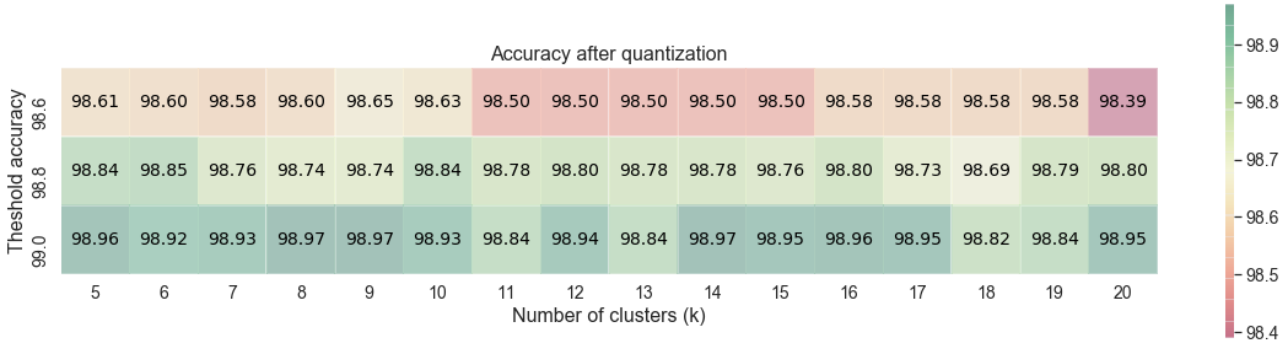
Baseline Accuracy	Post-Quantization Accuracy	Remaining/Original Parameters	% Non-zero Parameters	Compression Ratio
99.09	98.97	159271/431080	36.95	5.28



(a) The percentage of non-zero parameters after post-training quantization. A low percentage of non-zero parameters results into high compression.



(b) The compression ratio after post-training quantization.



(c) The accuracy after post-training quantization. The baseline accuracy is 99.09%.

Figure 5.5: Method 1 results for LeNet-5 on the MNIST dataset.

5.4.2 Method 2

We trained LeNet-5 on the MNIST dataset using Method 2, for 80 epochs. We varied *top* from 1000 to 5000 with a step size of 1000. The fractional bit width (F_q) for quantization was set to 7. Method 2 returns a partially quantized network. Table 5.8 shows results for the trained networks quantized to Q0.3 format, with round-ties-to-even rounding.

Table 5.8: Method 2: LeNet-5 on MNIST dataset, quantized to Q0.3, with weight bits (b_w) = 4 and index bits (b_i) = 11.

Top	Baseline Accuracy	Post-Quantization Accuracy	% Non-zero Parameters	Compression Ratio
1k	99.09	99.11	13.48	14.77
2k	99.09	99.24	12.12	16.30
3k	99.09	99.20	11.64	16.93
4k	99.09	99.22	9.17	21.05
5k	99.09	99.05	11.55	17.05

5.4.3 Method 3

We run Method 1 on the trained networks obtained using Method 2 as described in Subsection 5.4.2. Figure 5.6 illustrates the results obtained over a range of k and *top* values, for a threshold accuracy of 99.00%. We achieve a compression ratio of 16.30 to 50.19, with a post-quantization accuracy of 99.24% to 99.08%. The choice of k and *top* depends on the allowable accuracy degradation.

Table 5.9 shows the result for the configuration that yields the best compression. Table 5.10 shows the configuration that yields the best accuracy.

Table 5.9: Method 3: Best compression achieved for LeNet-5 on MNIST dataset.

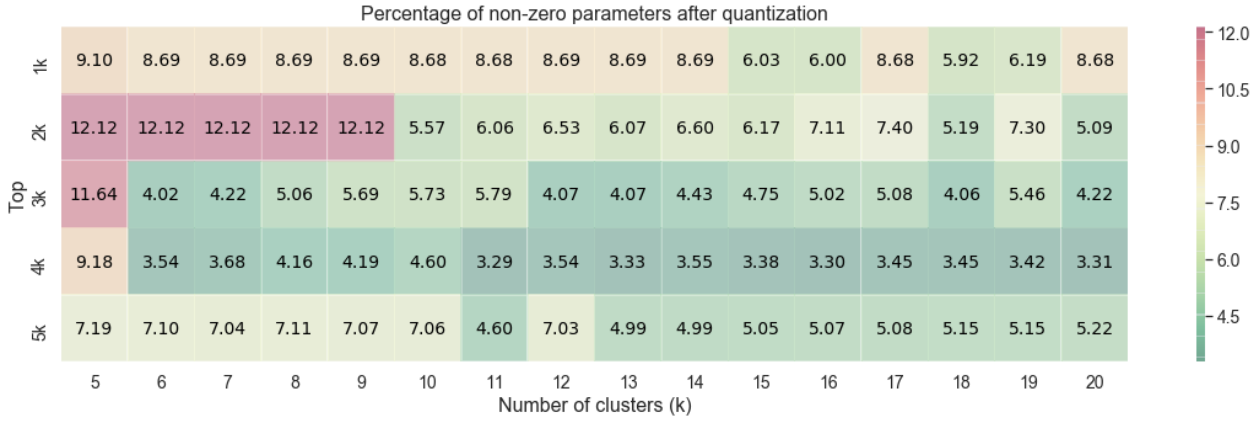
Number of clusters (k): 11
Threshold accuracy (T_a): 99.0
Top: 4000
Fractional bit budget (F): 7
Cluster membership (%): 0.009, 0.042, 0.068, 0.154, 0.388, 0.889, 1.583, 2.655, 6.019, 19.549, 68.644
FL assignment: 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 2
Bit width per weight (b_w): 4
Bit width per index (b_i): 11

Baseline Accuracy	Post-Quantization Accuracy	Remaining/Original Parameters	% Non-zero Parameters	Compression Ratio
99.09	99.08	14199/431080	3.29	50.19

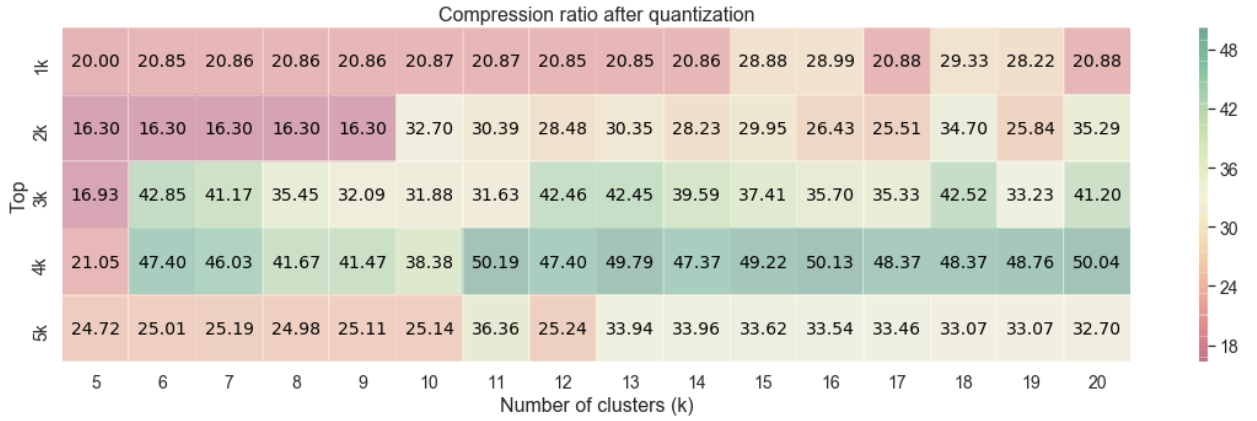
Table 5.10: Method 3: Best accuracy achieved for LeNet-5 on MNIST dataset.

Number of clusters (k): 10
Threshold accuracy (T_a): 99.0
Top: 5000
Fractional bit budget (F): 7
Cluster membership (%): 0.007, 0.012, 0.016, 0.032, 0.035, 0.053, 0.082, 0.14, 0.219, 0.324, 0.47, 0.621, 0.834, 1.158, 1.65, 2.805, 5.658, 11.348, 21.723, 52.812
FL assignment: 4, 4, 4, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2
Bit width per weight (b_w): 5
Bit width per index (b_i): 11

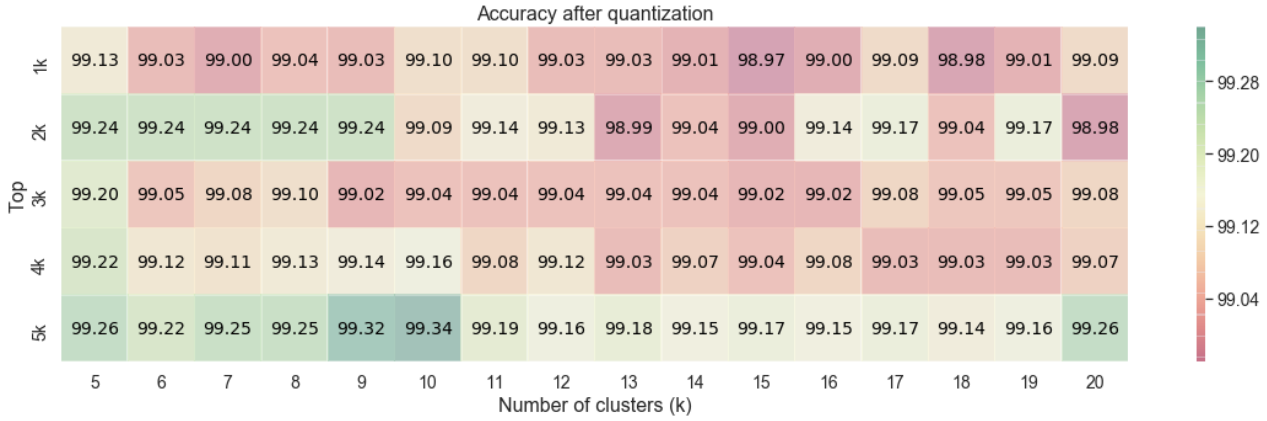
Baseline Accuracy	Post-Quantization Accuracy	Remaining/Original Parameters	% Non-zero Parameters	Compression Ratio
99.09	99.34	30427/431080	7.06	25.14



(a) The percentage of non-zero parameters after post-training quantization. A low percentage of non-zero parameters results into high compression.



(b) The compression ratio after post-training quantization.



(c) The accuracy after post-training quantization. The baseline accuracy is 99.09%.

Figure 5.6: Method 3 results for LeNet-5 on the MNIST dataset. The threshold accuracy is set to 99.00%

5.5 AlexNet on CIFAR-10 Dataset

We achieved a baseline accuracy of **82.57%** for AlexNet trained on the CIFAR-10 dataset. The network was trained for 90 epochs using stochastic gradient descent with momentum. The learning rate, momentum and weight decay were set to 0.01, 0.9, and 0.0005, respectively. The batch size was set to 64. We use the same initialization of weights for all three methods to get a fair comparison.

5.5.1 Method 1

Figure 5.7 illustrates the results obtained over a range of k and threshold accuracy values. We achieve a compression ratio of 4.38 to 6.39, with a post-quantization accuracy of 82.33% to 81.29%. The choice of k and threshold accuracy depends on the allowable accuracy degradation.

Table 5.11 shows the result for the configuration that yields the best compression. Quantizing all the parameters to $\mathbb{Q}0.7$ results in 30.86% non-zero parameters, with an accuracy of 82.33%. Quantizing all the parameters to $\mathbb{Q}0.1$ results in 0.01% non-zero parameters, with an accuracy of 10.00%. Our method achieves 20.93% non-zero parameters with an accuracy of 81.29%. Table 5.12 shows the configuration that yields the best accuracy.

Table 5.11: Method 1: Best compression achieved for AlexNet on CIFAR-10 dataset.

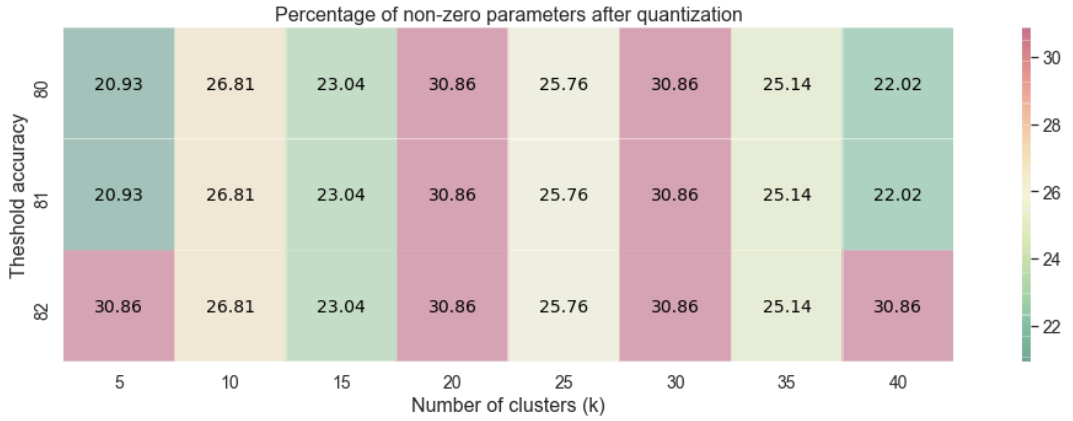
Number of clusters (k): 5
Threshold accuracy (T_a): 81.0
Fractional bit budget (F): 11
Cluster membership (%): 0.084, 0.882, 3.369, 48.819, 46.845
FL assignment: 7, 7, 7, 7, 1
Bit width per weight (b_w): 8
Bit width per index (b_i): 15

Baseline Accuracy	Post-Quantization Accuracy	Remaining/Original Parameters	% Non-zero Parameters	Compression Ratio
82.57	81.29	11909315/56909194	20.93	6.39

Table 5.12: Method 1: Best accuracy achieved for AlexNet on CIFAR-10 dataset.

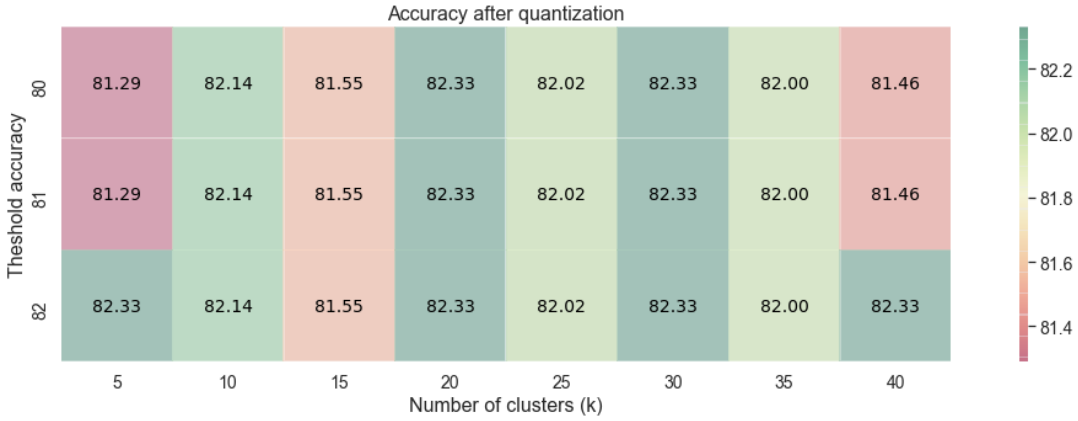
Number of clusters (k): 5
Threshold accuracy (T_a): 82.0
Fractional bit budget (F): 11
Cluster membership (%): 0.084, 0.882, 3.369, 48.819, 46.845
FL assignment: 7, 7, 7, 7, 7
Bit width per weight (b_w): 8
Bit width per index (b_i): 15

Baseline Accuracy	Post-Quantization Accuracy	Remaining/Original Parameters	% Non-zero Parameters	Compression Ratio
82.57	82.33	17562090/56909194	30.86	4.38



(a) The percentage of non-zero parameters after post-training quantization. A low percentage of non-zero parameters results into high compression.

(b) The compression ratio after post-training quantization.



(c) The accuracy after post-training quantization. The baseline accuracy is 82.57%.

Figure 5.7: Method 1 results for AlexNet on the CIFAR-10 dataset.

5.5.2 Method 2

We trained AlexNet on the CIFAR-10 dataset using Method 2, for 110 epochs. We varied *top* from 10M to 30M with a step size of 5M. The fractional bit width (F_q) for quantization was set to 11. Method 2 returns a partially quantized network. Table 5.13 shows results for the trained networks quantized to Q0.6 format, with round-ties-to-even rounding.

Table 5.13: Method 2: AlexNet on CIFAR-10 dataset, quantized to Q0.6, with weight bits (b_w) = 7 and index bits (b_i) = 15.

Top	Baseline Accuracy	Post-Quantization Accuracy	% Non-zero Parameters	Compression Ratio
10M	82.57	79.85	10.93	7.95
15M	82.57	79.59	10.92	7.94
20M	82.57	79.85	11.14	7.86
25M	82.57	80.56	11.18	7.87
30M	82.57	80.49	10.99	7.94

5.5.3 Method 3

We run Method 1 on the trained networks obtained using Method 2 as described in Subsection 5.5.2. Figure 5.8 illustrates the results obtained over a range of k and *top* values, for a threshold accuracy of 80.00%. We achieve a compression ratio of 4.23 to 13.15, with a post-quantization accuracy of 80.76% to 79.72%. The choice of k and *top* depends on the allowable accuracy degradation.

Table 5.14 shows the result for the configuration that yields the best compression. Table 5.15 shows the configuration that yields the best accuracy.

Table 5.14: Method 3: Best compression achieved for AlexNet on CIFAR-10 dataset.

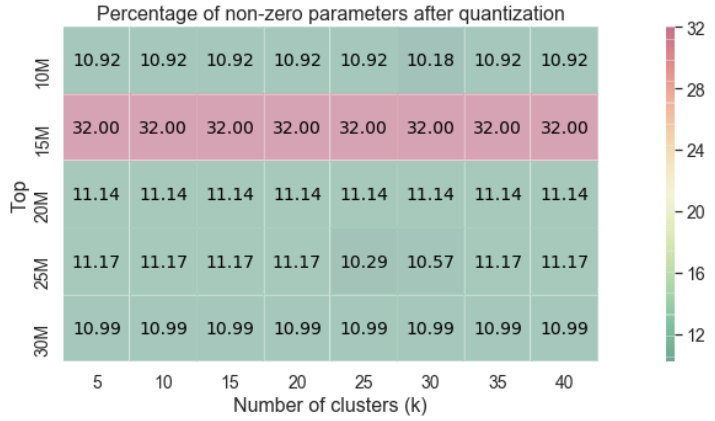
Number of clusters (k): 30
Threshold accuracy (T_a): 80.0
Top: 10000000
Fractional bit budget (F): 11
Cluster membership (%): 0.000002, 0.000005, 0.00028, 0.00047, 0.0001, 0.0003, 0.001, 0.002, 0.004, 0.007, 0.012, 0.021, 0.033, 0.049, 0.072, 0.098, 0.125, 0.161, 0.195, 0.234, 0.274, 0.321, 0.375, 0.453, 0.587, 0.881, 1.78, 6.433, 65.974, 21.907
FL assignment: 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 1
Bit width per weight (b_w): 7
Bit width per index (b_i): 15

Baseline Accuracy	Post-Quantization Accuracy	Remaining/Original Parameters	% Non-zero Parameters	Compression Ratio
82.57	79.72	5791050/56909194	10.18	13.15

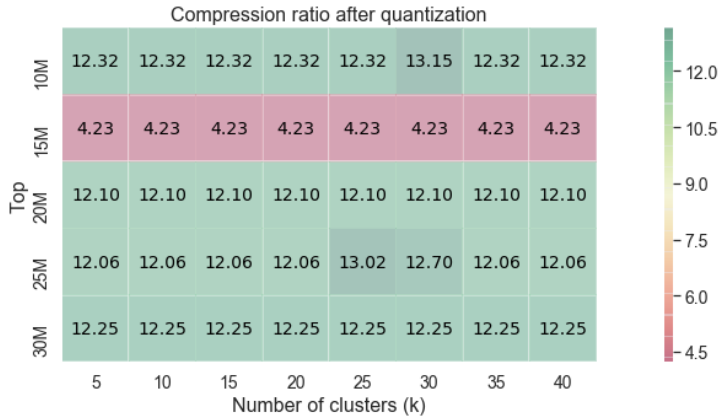
Table 5.15: Method 3: Best accuracy achieved for AlexNet on CIFAR-10 dataset.

Number of clusters (k): 5
Threshold accuracy (T_a): 80.0
Top: 15000000
Fractional bit budget (F): 11
Cluster membership (%): 0.126, 1.017, 3.584, 39.637, 55.635
FL assignment: 7, 7, 7, 7, 7
Bit width per weight (b_w): 8
Bit width per index (b_i): 15

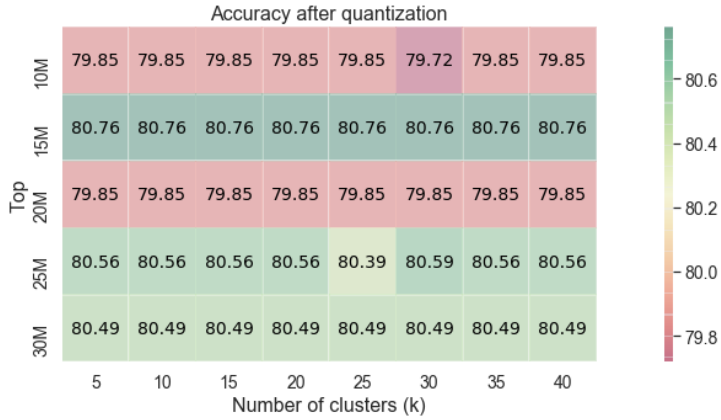
Baseline Accuracy	Post-Quantization Accuracy	Remaining/Original Parameters	% Non-zero Parameters	Compression Ratio
82.57	80.76	18209317/56909194	32.00	4.23



(a) The percentage of non-zero parameters after post-training quantization. A low percentage of non-zero parameters results into high compression.



(b) The compression ratio after post-training quantization.



(c) The accuracy after post-training quantization. The baseline accuracy is 82.57%.

Figure 5.8: Method 3 results for AlexNet on the CIFAR-10 dataset. The threshold accuracy is set to 80.00%

5.6 VGG-16 on CIFAR-10 Dataset

We achieved a baseline accuracy of **80.73%** for VGG-16 trained on the CIFAR-10 dataset. The network was trained for 50 epochs using stochastic gradient descent with momentum. The learning rate, momentum and weight decay were set to 0.01, 0.9, and 0.0005, respectively. The batch size was set to 64. We use the same initialization of weights for all three methods to get a fair comparison.

5.6.1 Method 1

Figure 5.9 illustrates the results obtained over a range of k and threshold accuracy values. We achieve a compression ratio of 7.85 to 12.43, with a post-quantization accuracy of 79.91% to 77.74%. The choice of k and threshold accuracy depends on the allowable accuracy degradation.

Table 5.16 shows the result for the configuration that yields the best compression. Quantizing all the parameters to $\mathbb{Q}_{0.6}$ results in 15.16% non-zero parameters, with an accuracy of 79.91%. Quantizing all the parameters to $\mathbb{Q}_{0.1}$ results in 0.01% non-zero parameters, with an accuracy of 10.00%. Our method achieves 8.64% non-zero parameters with an accuracy of 77.74%. Table 5.17 shows the configuration that yields the best accuracy.

Table 5.16: Method 1: Best compression achieved for VGG-16 on CIFAR-10 dataset.

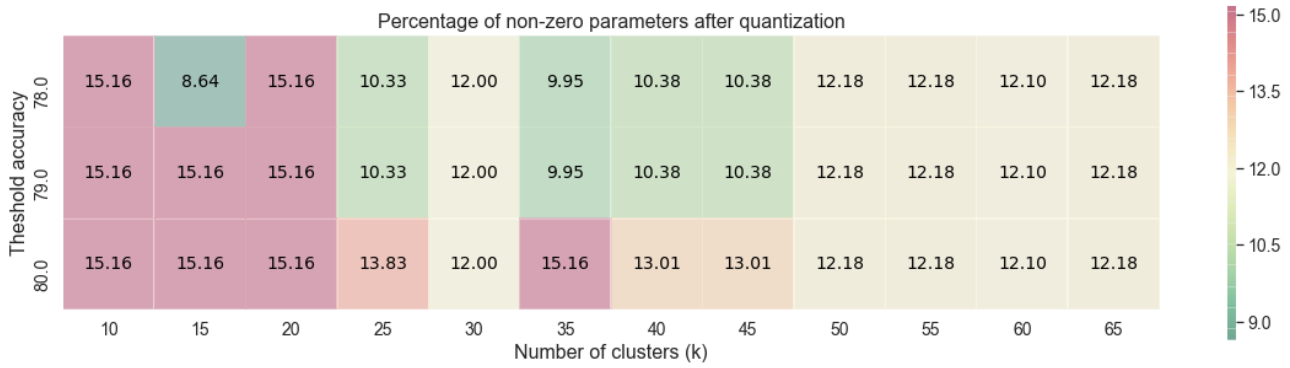
Number of clusters (k): 15
Threshold accuracy (T_a): 78.0
Fractional bit budget (F): 10
Cluster membership (%): 0.000001, 0.0004, 0.002, 0.017, 0.066, 0.188, 0.418, 0.823, 1.553, 2.978, 6.404, 23.18, 60.342, 4.028
FL assignment: 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 1, 1
Bit width per weight (b_w): 7
Bit width per index (b_i): 16

Baseline Accuracy	Post-Quantization Accuracy	Remaining/Original Parameters	% Non-zero Parameters	Compression Ratio
80.73	77.74	11605665/134301514	8.64	12.43

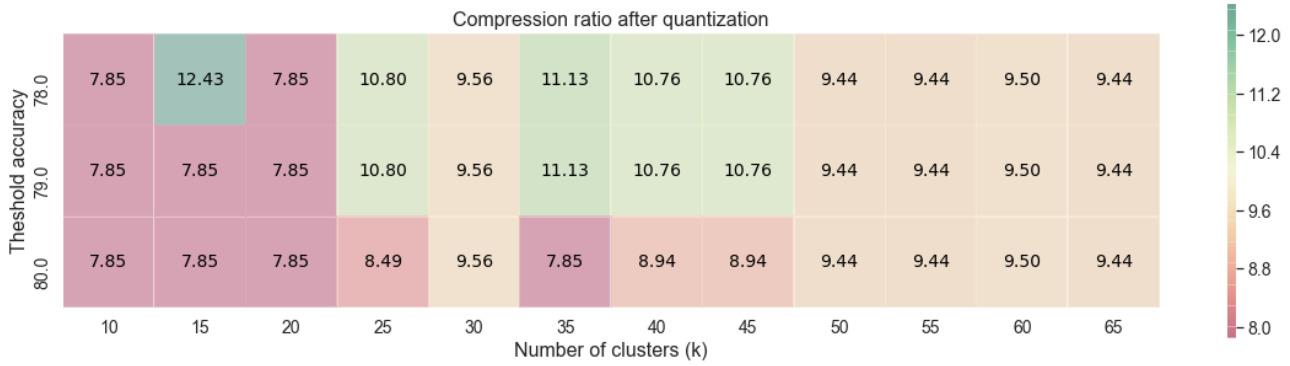
Table 5.17: Method 1: Best accuracy achieved for VGG-16 on CIFAR-10 dataset.

Number of clusters (k): 10
Threshold accuracy (T_a): 80.0
Fractional bit budget (F): 10
Cluster membership (%): 0.001, 0.027, 0.179, 0.594, 1.338, 2.673, 5.743, 21.541, 67.904
FL assignment: 6, 6, 6, 6, 6, 6, 6, 6, 6
Bit width per weight (b_w): 7
Bit width per index (b_i): 16

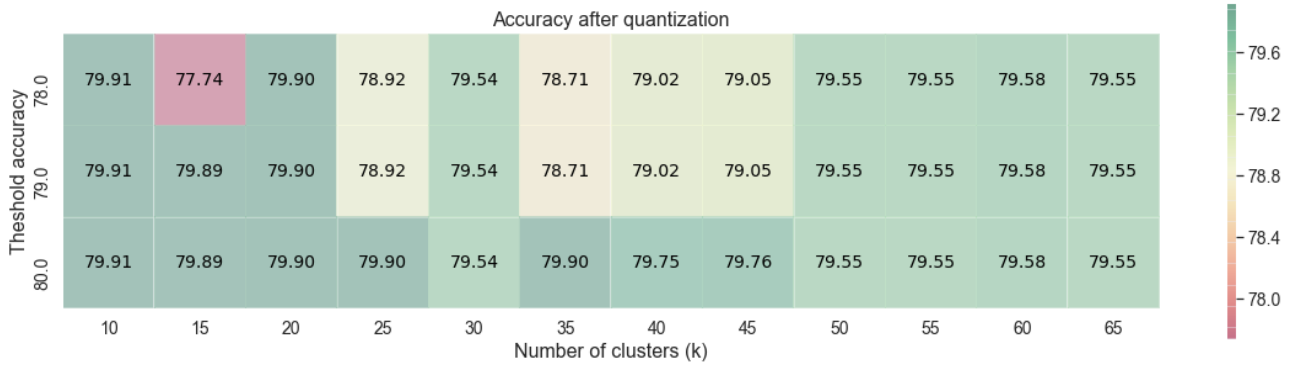
Baseline Accuracy	Post-Quantization Accuracy	Remaining/Original Parameters	% Non-zero Parameters	Compression Ratio
80.73	79.91	20366608/134301514	15.16	7.85



(a) The percentage of non-zero parameters after post-training quantization. A low percentage of non-zero parameters results into high compression.



(b) The compression ratio after post-training quantization.



(c) The accuracy after post-training quantization. The baseline accuracy is 80.73%.

Figure 5.9: Method 1 results for VGG-16 on the CIFAR-10 dataset.

5.6.2 Method 2

We trained VGG-16 on the CIFAR-10 dataset using Method 2, for 70 epochs. We varied *top* from 30M to 55M with a step size of 5M. The fractional bit width (F_q) for quantization was set to 10. Method 2 returns a partially quantized network. Table 5.18 shows results for the trained networks quantized to Q0.6 format, with round-ties-to-even rounding.

Table 5.18: Method 2: VGG-16 on CIFAR-10 dataset, quantized to Q0.6, with weight bits (b_w) = 7 and index bits (b_i) = 16.

Top	Baseline Accuracy	Post-Quantization Accuracy	% Non-zero Parameters	Compression Ratio
30M	80.73	77.37	8.69	12.38
35M	80.73	76.90	8.40	12.71
40M	80.73	77.78	8.58	12.49
45M	80.73	75.27	8.57	12.51
50M	80.73	77.54	8.50	12.58
55M	80.73	77.88	8.58	12.49

5.6.3 Method 3

We run Method 1 on the trained networks obtained using Method 2 as described in Subsection 5.6.2. Figure 5.10 illustrates the results obtained over a range of k and *top* values, for a threshold accuracy of 78.00%. We achieve a compression ratio of 3.62 to 13.53, with a post-quantization accuracy of 78.72% to 76.21%. The choice of k and *top* depends on the allowable accuracy degradation.

Table 5.19 shows the result for the configuration that yields the best compression. Table 5.20 shows the configuration that yields the best accuracy.

Table 5.19: Method 3: Best compression achieved for VGG-16 on CIFAR-10 dataset.

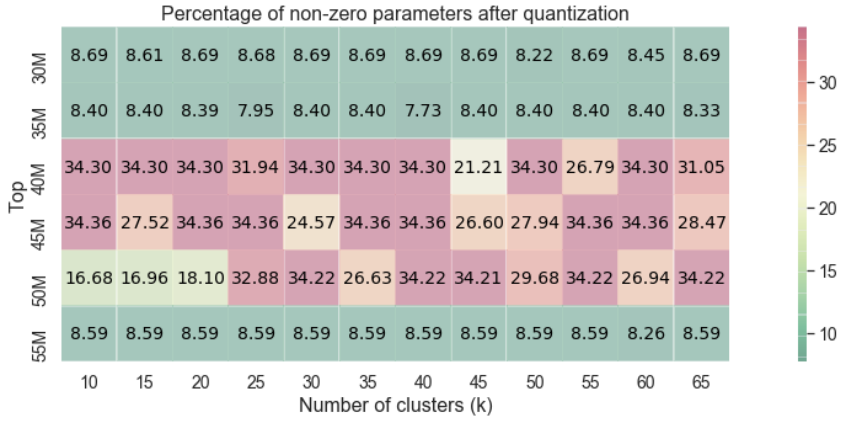
Number of clusters (k): 40
Threshold accuracy (T_a): 78.0
Top: 35000000
Fractional bit budget (F): 10
Cluster membership (%): 0.000001, 0.000001, 0.000001, 0.000001, 0.000003, 0.00013, 0.00018, 0.00065, 0.0001, 0.0002, 0.0004, 0.001, 0.002, 0.007, 0.014, 0.025, 0.041, 0.061, 0.088, 0.119, 0.156, 0.194, 0.237, 0.286, 0.346, 0.41, 0.469, 0.55, 0.615, 0.722, 0.885, 1.124, 1.409, 1.939, 6.338, 51.185, 32.776
FL assignment: 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 1
Bit width per weight (b_w): 7
Bit width per index (b_i): 16

Baseline Accuracy	Post-Quantization Accuracy	Remaining/Original Parameters	% Non-zero Parameters	Compression Ratio
80.73	76.21	10386184/134301514	7.73	13.53

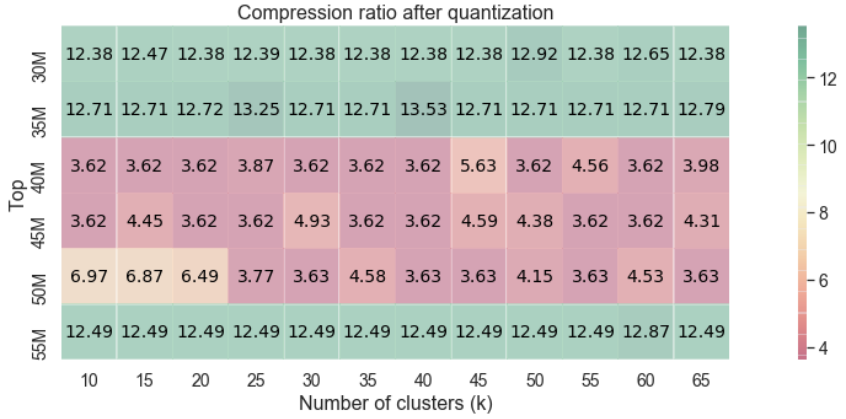
Table 5.20: Method 3: Best accuracy achieved for VGG-16 on CIFAR-10 dataset.

Number of clusters (k): 30
Threshold accuracy (T_a): 78.0
Top: 50000000
Fractional bit budget (F): 10
Cluster membership (%): 0.000001, 0.000001, 0.000001, 0.000004, 0.00006, 0.00088, 0.0001, 0.0003, 0.001, 0.002, 0.005, 0.011, 0.021, 0.034, 0.051, 0.075, 0.106, 0.148, 0.204, 0.274, 0.36, 0.475, 0.619, 0.824, 1.265, 2.856, 15.787, 76.884
FL assignment: 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7
Bit width per weight (b_w): 7
Bit width per index (b_i): 16

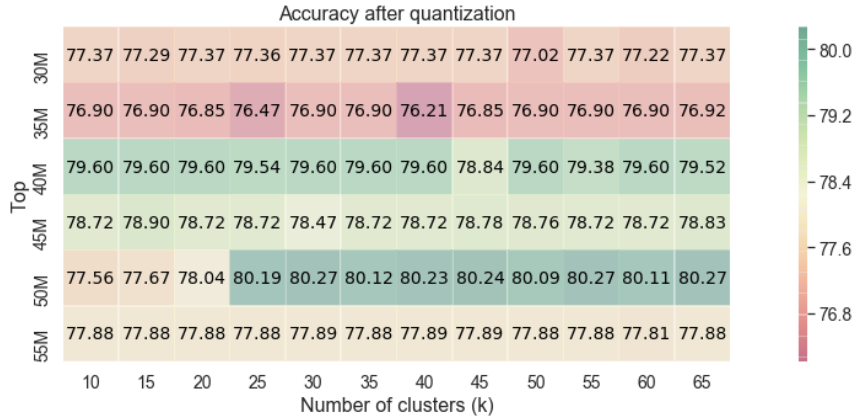
Baseline Accuracy	Post-Quantization Accuracy	Remaining/Original Parameters	% Non-zero Parameters	Compression Ratio
80.73	80.27	45954860/134301514	34.22	3.63



(a) The percentage of non-zero parameters after post-training quantization. A low percentage of non-zero parameters results into high compression.



(b) The compression ratio after post-training quantization.



(c) The accuracy after post-training quantization. The baseline accuracy is 80.73%.

Figure 5.10: Method 3 results for VGG-16 on the CIFAR-10 dataset. The threshold accuracy is set to 78.00%

5.7 Discussion

We present a comparison of the results achieved by the three quantization methods discussed so far. We consider the best compression results for comparison. Performance steadily improves as we move from post-training quantization to in-training quantization and combined quantization.

Compared to post-training quantization (Method 1), in-training quantization (Method 2) achieves a compression ratio gain that is 2.89×, 1.93×, 1.24× and 1.02× for LeNet-300-100, LeNet-5, AlexNet and VGG-16, respectively. We achieve the best compression results using combined quantization (Method 3). Compared to Method 1, the compression ratio increases by 7.50×, 4.61×, 2.05× and 1.08× for LeNet-300-100, LeNet-5, AlexNet and VGG-16, respectively. Figures 5.11 illustrates these trends.

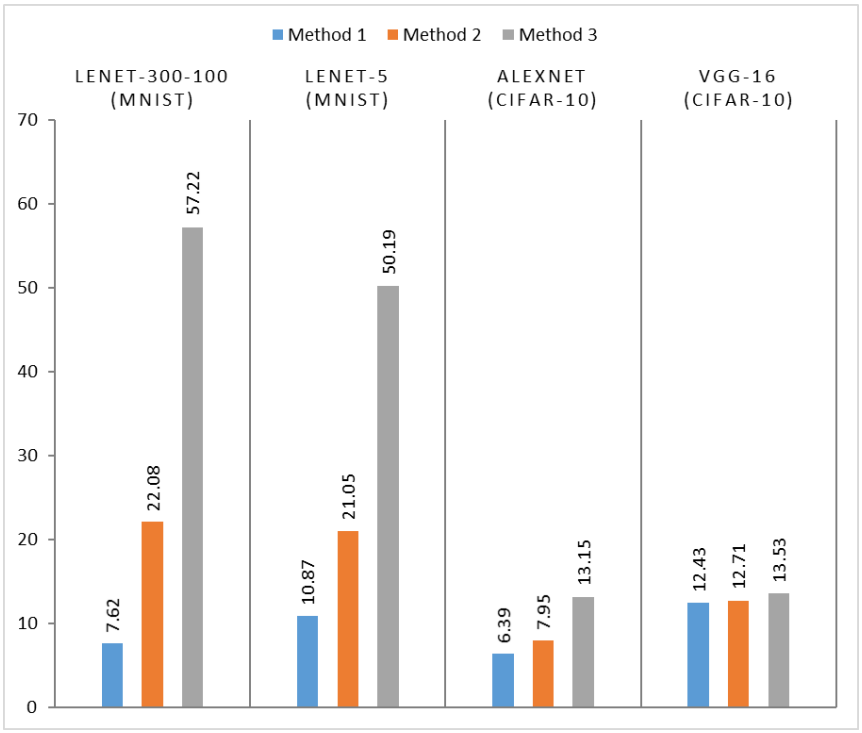


Figure 5.11: Compression ratio after quantization.

Figure 5.12 shows that the percentage of non-zero parameters decreases as we move from post-training quantization to in-training quantization. Combined quantization achieves the lowest percentage of non-zero parameters, with 3.61%, 3.29%, 10.18% and 7.73% for LeNet-300-100, LeNet-5, AlexNet and VGG-16, respectively.

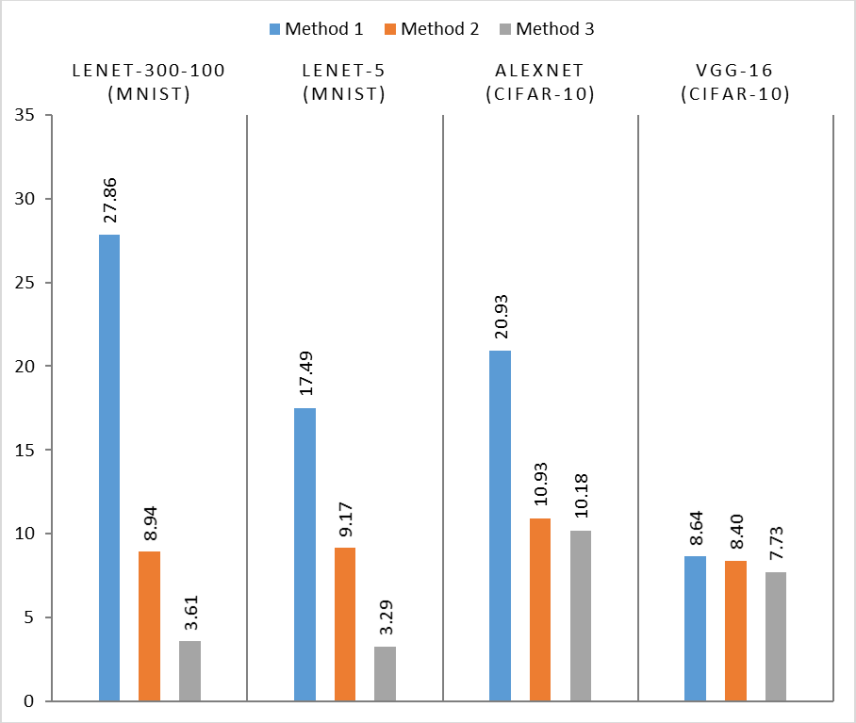


Figure 5.12: Percentage of non-zero parameters after quantization.

Figure 5.13 shows that for LeNet-300-100 and LeNet-5, Method 1 has the highest accuracy loss. This accuracy loss drops for Method 2 and increases slightly as we go from Method 2 to Method 3. We do not observe this trend for AlexNet or VGG-16, which exhibit the lowest accuracy loss for Method 1. This accuracy loss rises for Method 2 and increases slightly as we go from Method 2 to Method 3. For LeNet-5, we observe accuracy gains after quantization, i.e, the post-quantization accuracy was observed to be greater than the original accuracy.

Figure 5.14 shows the best compression ratios achieved, with and without assuming the CSR format. Using the CSR format, there is a 86.01%, 84.06%, 65.24% and 66.23% decrease in

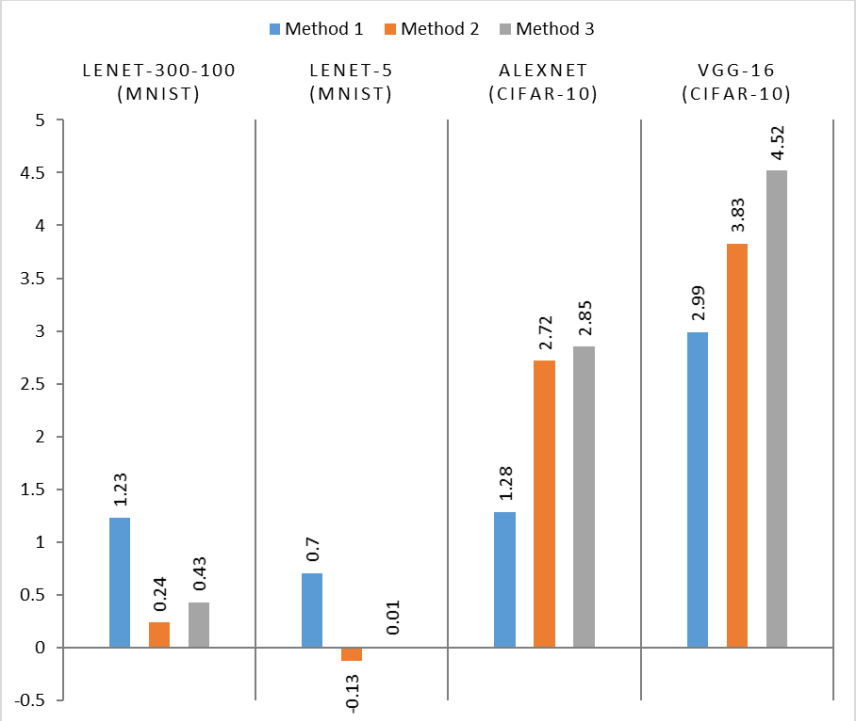


Figure 5.13: Δ Accuracy after quantization. This is the difference between the post-quantization and original accuracy. A negative accuracy difference indicates a gain in post-quantization accuracy.

the number of bits used to represent the neural network parameters, for LeNet-300-100, LeNet-5, AlexNet and VGG-16, respectively. This translates to a 7.15×, 6.27×, 2.87× and 2.96× increase in the compression ratio due to the CSR format, for LeNet-300-100, LeNet-5, AlexNet and VGG-16, respectively.

In our experiments, Method 2 added 6 seconds and 1 minute and 48 seconds of training time per epoch on NVIDIA V100 nodes for AlexNet and VGG-16, respectively. For LeNet-300-100 and LeNet-5, the additional time is insignificant. On Intel's Broadwell processors, one instance of the post-training quantization (for Method 1 or Method 3) took 28 seconds, 1 minute, 19 minutes and 2 hours and 38 minutes of CPU time, for LeNet-300-100, LeNet-5, AlexNet and VGG-16, respectively.

Han et al. [24] achieve a compression of 32× and 33× after pruning and quantization on

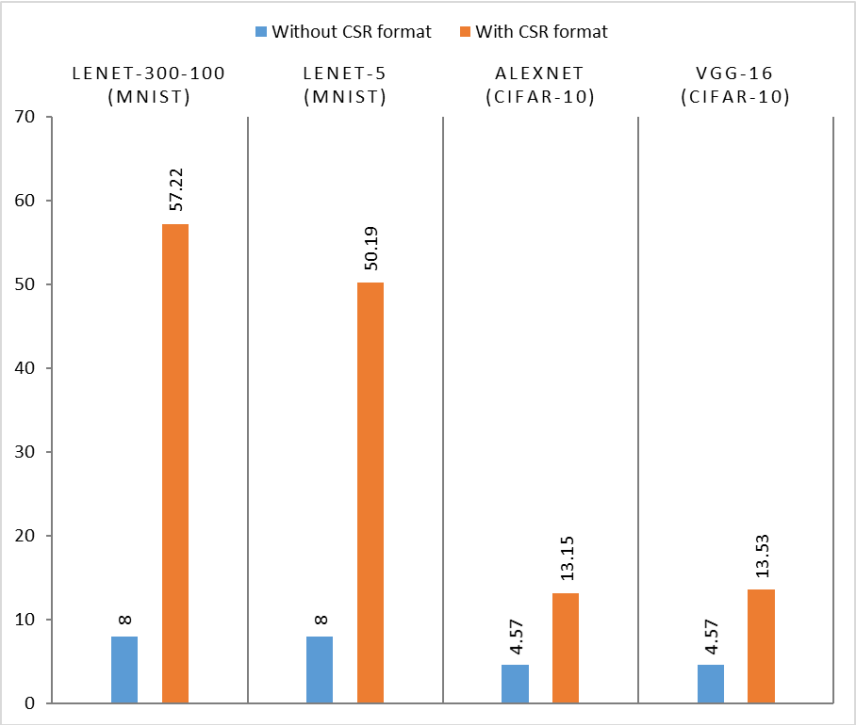


Figure 5.14: Compression ratios with and without the CSR format.

LeNet-300-100 and LeNet-5, respectively. Their non-zero parameters are at 8% for both networks. We achieve better compression due to a lower percentage of non-zero parameters, and by using lower number of bits per parameter. On AlexNet as well as VGG-16 (trained on ILSVRC-2012 dataset), they achieve a much higher compression of 27× and 31×, respectively, with a non-zero percentage of 11% and 7.5%, respectively. Unlike Han et al. [24], our methods lose accuracy as a consequence of quantization.

Chapter 6

Conclusion

6.1 Summary

In this thesis, we explored the novel usage of accumulated gradients as an importance criterion for fixed-point quantization decisions. We devised accumulated gradient-based post-training, in-training and combined quantization methods, converting weights and biases to fixed-point quantization levels. The methods we devised, achieved significant compression for three reasons:

- Significant pruning was achieved through quantization, on account of zero being a quantization level in our quantization technique.
- Weights and biases that were not pruned were quantized to considerably lower bit widths through fixed-point quantization.
- For storing the sparse neural networks, we assumed storage with the compressed sparse row format. Storing the column indices and row information using index differences instead of absolute positions allowed us to use lower bit widths for this information.

Our results support our hypothesis that the accumulated gradient of a parameter can be an indicator of its quantization tolerance, and can be successfully used to make quantization decisions. We achieved up to 57.22×, 50.19×, 13.15× and 13.53× compression on LeNet-300-

100 (MNIST dataset), LeNet-5 (MNIST dataset), AlexNet (CIFAR-10 dataset) and VGG-16 (CIFAR-10 dataset), respectively.

6.2 Limitations and Future Work

Our results showed that as the network size increases, the gains in compression are lower and the post-quantization accuracy drop is higher. Our methods are heuristic in nature and results show some variation across different training runs on the same architecture. Additionally, a grid search of various parameters is required to find optimal calibrations.

In the future, various methods of gradient accumulation can be explored. This includes accumulating gradients for all mini-batches. The compression benefits of the compressed sparse row format can be further enhanced. Han et al. [24] encode the column indices and row information using 8 bits for convolutional layers and 5 bits for fully-connected layers. For every occurrence of an index difference greater than the bound, they use zero-padding. Additionally, they perform Huffman coding. This could lead to additional compression gains using our methods. There is also scope for automating the grid search procedure to obtain the optimal calibrations.

Bibliography

[1] CC BY 4.0. https://creativecommons.org/licenses/by/4.0/.

[2] KMeans: scikit-learn package of SciPy. https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html.

[3] NumPy: Package for scientific computing.

[4] PyTorch: Optimized Tensor Library for Deep Learning. https://pytorch.org/.

[5] Tensorflow Lite. https://www.tensorflow.org/lite.

[6] Compressed Sparse Row matrix: SciPy 2-D sparse matrix package. https://docs.scipy.org/doc/scipy/reference/sparse.html.

[7] cuSPARSE: Cuda Toolkit Documentation. https://docs.nvidia.com/cuda/cusparse/index.html.

[8] Developer Reference for Intel® Math Kernel Library - C. https://software.intel.com/en-us/mkl-developer-reference-c-blas-and-sparse-blas-routines.

[9] Q-format. https://software.intel.com/en-us/node/604539.

[10] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, July 2019. doi: 10.1109/IEEESTD.2019.8766229.

[11] Arthur, David and Vassilvitskii, Sergei. K-Means++: The Advantages of Careful Seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, pages 1027–1035, USA, 2007. Society for Industrial and Applied Mathematics. ISBN 9780898716245.

[12] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[13] Yves Chauvin. A Back-Propagation Algorithm with Optimal Use of Hidden Units. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 1*, pages 519–526. Morgan-Kaufmann, 1989.

[14] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. BinaryConnect: Training Deep Neural Networks with Binary Weights during Propagations. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, pages 3123–3131, Cambridge, MA, USA, 2015. MIT Press.

[15] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Low Precision Arithmetic for Deep Learning. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Workshop Track Proceedings*, 2015.

[16] Xiaohan Ding, guiguang ding, Xiangxin Zhou, Yuchen Guo, Jungong Han, and Ji Liu. Global Sparse Momentum SGD for Pruning Very Deep Neural Networks. 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 6382–6394. Curran Associates, Inc., 2019.

[17] Xin Dong, Shangyu Chen, and Sinno Pan. Learning to Prune Deep Neural Networks via Layer-wise Optimal Brain Surgeon. In I. Guyon and U. V. Luxburg and S. Bengio and H. Wallach and R. Fergus and S. Vishwanathan and R. Garnett, editor, *Advances in Neural Information Processing Systems 30*, pages 4857–4867. 2017.

[18] Maximilian Golub. DropBack: Continuous Pruning During Deep Neural Network Training. Master's thesis, University of Washington, 2016.

[19] Maximilian Golub, Guy Lemieux, and Mieszko Lis. Full deep neural network training on a pruned weight budget. In *Proceedings of Machine Learning and Systems 2019*, pages 252–263. 2019.

[20] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[21] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic Network Surgery for Efficient DNNs. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, pages 1387–1395, Red Hook, NY, USA, 2016. Curran Associates Inc. ISBN 9781510838819.

[22] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep Learning with Limited Numerical Precision. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 1737–1746. JMLR.org, 2015.

[23] Philipp Matthias Gysel. Ristretto: Hardware-Oriented Approximation of Convolutional Neural Networks. Master's thesis, University of California Davis, 2016.

[24] Song Han, Huizi Mao, and William J Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In *ICLR 2016*.

[25] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning Both Weights and Connections for Efficient Neural Networks. In *Proceedings of the 28th International*

Conference on Neural Information Processing Systems - Volume 1, NIPS'15, pages 1135–1143, Cambridge, MA, USA, 2015. MIT Press.

[26] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 243–254. IEEE, 2016.

[27] Babak Hassibi, David G. Stork, Gregory Wolff, and Takahiro Watanabe. Optimal Brain Surgeon: Extensions and Performance Comparisons. In *Proceedings of the 6th International Conference on Neural Information Processing Systems*, NIPS'93, pages 263–270, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[28] M. Horowitz. Computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, Feb 2014. doi: 10.1109/ISSCC.2014.6757323.

[29] 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. 2016.

[30] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, June 2018. doi: 10.1109/CVPR.2018.00286.

[31] C. Ji, R. R. Snapp, and D. Psaltis. Generalizing Smoothness Constraints from Discrete Samples. *Neural Computation*, 2(2):188–197, June 1990. ISSN 0899-7667. doi: 10.1162/neco.1990.2.2.188.

[32] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. 2009.

[33] 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 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[34] Yann LeCun, John S. Denker, and Sara A. Solla. Optimal Brain Damage. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 598–605. Morgan-Kaufmann, 1990.

[35] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[36] F.-F. Li, A. Karpathy, and J. Johnson. Stanford CS Class CS231n: Convolutional Neural Networks for Visual Recognition. [Online]. http://cs231n.github.io.

[37] Fengfu Li and Bin Liu. Ternary Weight Networks. *CoRR*, abs/1605.04711, 2016.

[38] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning Filters for Efficient ConvNets. *CoRR*, abs/1608.08710, 2016.

[39] Darryl D. Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. Fixed Point Quantization of Deep Convolutional Networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pages 2849–2858. JMLR.org, 2016.

[40] Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. Runtime Neural Pruning. In *Advances in Neural Information Processing Systems*, pages 2181–2191, 2017.

[41] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The Expressive Power of Neural Networks: A View from the Width. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pages 6232–6240, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

[42] 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.

[43] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed Precision Training. In *International Conference on Learning Representations*, 2018.

[44] Lukas Mosser, Olivier Dubrule, and Martin J. Blunt. Stochastic Reconstruction of an Oolitic Limestone by Generative Adversarial Networks. *Transport in Porous Media*, 125 (1):81–103, Oct 2018. ISSN 1573-1634. doi: 10.1007/s11242-018-1039-9.

[45] Eunhyeok Park, Junwhan Ahn, and Sungjoo Yoo. Weighted-Entropy-Based Quantization for Deep Neural Networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[46] Zhuwei Qin, Fuxun Yu, Chenchen Liu, and Xiang Chen. Demystifying Neural Network Filter Pruning. *CoRR*, abs/1811.02639, 2018.

[47] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Ar-*

rays, FPGA '16, pages 26–35, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450338561. doi: 10.1145/2847263.2847265.

[48] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: Imagenet Classification Using Binary Convolutional Neural Networks. *Lecture Notes in Computer Science*, pages 525–542, 2016. ISSN 1611-3349. doi: 10.1007/978-3-319-46493-0_32.

[49] R. Reed. Pruning Algorithms-A Survey. *IEEE Transactions on Neural Networks*, 4(5): 740–747, Sep. 1993. ISSN 1941-0093. doi: 10.1109/72.248452.

[50] Abdullah I. Salama, Oleksiy Ostapenko, Moin Nabi, and Tassilo Klein. Pruning at a Glance: A Structured Class-Blind Pruning Technique for Model Compression. 2018.

[51] Abigail See, Minh-Thang Luong, and Christopher D Manning. Compression of Neural Machine Translation Models via Pruning. *arXiv preprint arXiv:1606.09274*, 2016.

[52] Sanchari Sen, Shubham Jain, Swagath Venkataramani, and Anand Raghunathan. SparCE: Sparsity Aware General-Purpose Core Extensions to Accelerate Deep Neural Networks. *IEEE Transactions on Computers*, 68(6):912–925, Jun 2019. ISSN 2326-3814.

[53] Anirudh Shenoy. How are Convolutions Actually Performed Under the Hood? https://towardsdatascience.com/how-are-convolutions-actually-performed-under-the-hood-226523ce7fbf.

[54] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-scale Image Recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[55] S. Srinivas, A. Subramanya, and R. Babu. Training Sparse Neural Networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*,

pages 455–462, Los Alamitos, CA, USA, jul 2017. IEEE Computer Society. doi: 10.1109/CVPRW.2017.61.

[56] V. Sze, Y. Chen, T. Yang, and J. S. Emer. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*, 105(12):2295–2329, Dec 2017. ISSN 1558-2256. doi: 10.1109/JPROC.2017.2761740.

[57] Frederick Tung and Greg Mori. CLIP-Q: Deep Network Compression Learning by In-parallel Pruning-Quantization. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7873–7882, 2018.

[58] Karen Ullrich, Edward Meeds, and Max Welling. Soft Weight-Sharing for Neural Network Compression, 2017.

[59] Mário P Véstias, Rui Policarpo Duarte, José T de Sousa, and Horácio C Neto. Fast Convolutional Neural Networks in Low Density FPGAs Using Zero-Skipping and Weight Pruning. *Electronics*, 8(11):1321, 2019.

[60] H. Wang, Q. Zhang, Y. Wang, L. Yu, and H. Hu. Structured Pruning for Efficient ConvNets via Incremental Regularization. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, July 2019. doi: 10.1109/IJCNN.2019.8852463.

[61] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. HAQ: Hardware-Aware Automated Quantization with Mixed Precision. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 8604–8612. IEEE, 2019.

[62] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. Training Deep Neural Networks with 8-bit Floating Point Numbers. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors,

Advances in Neural Information Processing Systems 31, pages 7675–7684. Curran Associates, Inc., 2018.

[63] Andreas S. Weigend, David E. Rumelhart, and Bernardo A. Huberman. Generalization by Weight-Elimination with Application to Forecasting. In R. P. Lippmann, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*, pages 875–882. Morgan-Kaufmann, 1991.

[64] Randy Yates. Fixed-Point Arithmetic: An Introduction . https://courses.cs.washington.edu/courses/cse467/08au/labs/l5/fp.pdf, 2007.

[65] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-X: An Accelerator for Sparse Neural Networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[66] Tianyi Zhang, Zhiqiu Lin, Guandao Yang, and Christopher De Sa. QPyTorch: A Low-Precision Arithmetic Simulation Framework. *ArXiv*, abs/1910.04540, 2019.

[67] Tianyun Zhang, Shaokai Ye, Kaiqi Zhang, Jian Tang, Wujie Wen, Makan Fardad, and Yanzhi Wang. A Systematic DNN Weight Pruning Framework Using Alternating Direction Method of Multipliers. In Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss, editors, *Computer Vision – ECCV 2018*, pages 191–207, Cham, 2018. Springer International Publishing. ISBN 978-3-030-01237-3.

[68] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights, 2017.

[69] Shuchang Zhou, Zekun Ni, Xinyu Zhou, He Wen, Yuxin Wu, and Yuheng Zou. DoReFa-

Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *CoRR*, abs/1606.06160, 2016.

[70] Yiren Zhou, Seyed-Mohsen Moosavi-Dezfooli, Ngai-Man Cheung, and Pascal Frossard. Adaptive Quantization for Deep Neural Network. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[71] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained Ternary Quantization. *arXiv preprint arXiv:1612.01064*, 2016.