EFFECT OF QUANTIZATION ON THE PERFORMANCE OF DEEP NETWORKS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

BAŞAR KÜTÜKCÜ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONIC ENGINEERING

JULY 2020

Approval of the thesis:

EFFECT OF QUANTIZATION ON THE PERFORMANCE OF DEEP NETWORKS

submitted by **BAŞAR KÜTÜKCÜ** in partial fulfillment of the requirements for the degree of **Master of Science** i**n Electrical and Electronic Engineering, Middle East Technical University** by,

Date: 20.07.2020

ABSTRACT

EFFECT OF QUANTIZATION ON THE PERFORMANCE OF DEEP NETWORKS

Kütükcü, Başar
Master of Science, Electrical and Electronic Engineering
Supervisor: Prof. Dr. Gözde Bozdağı Akar

July 2020, 76 pages

Deep neural networks performed greatly for many engineering problems in recent years. However, power and memory hungry nature of deep learning algorithm prevents mobile devices to benefit from the success of deep neural networks. The increasing number of mobile devices creates a push to make deep network deployment possible for resource-constrained devices. Quantization is a solution for this problem. In this thesis, different quantization techniques and their effects on deep networks are examined. The techniques are benchmarked by their success and memory requirements. The effects of quantization are examined for different network architectures including shallow, overparameterized, deep, residual, efficient models. Architecture specific problems are observed and related solutions are proposed. Quantized models are compared with ground-up efficiently designed models. The advantages and disadvantages of each technique are examined. Standard and quantized convolution operations implemented in real systems ranging from low power embedded systems to powerful desktop computer systems. Computation time and memory requirements are examined in these real systems.

Keywords: Deep Neural Networks, Quantization

ÖZ

NİCELEMENİN DERİN AĞLARA ETKİSİ

Kütükcü, Başar
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği
Tez Yöneticisi: Prof. Dr. Gözde Bozdağı Akar

Temmuz 2020, 76 sayfa

Derin sinir ağları son zamanlarda birçok mühendislik problemi için büyük başarı göstermiştir. Ancak derin öğrenme algoritmasının hesaplama gücü ve hafızaya çok fazla gereksinim duyması sebebiyle, sayısı gitgide artan kaynak kısıtlı mobil cihazlar derin sinir ağlarının bu başarısından yararlanamamaktadır. Niceleme bu soruna çözüm olabilecek yöntemlerden birisidir. Bu tez kapsamında, farklı niceleme teknikleri ve bu tekniklerin derin ağlara etkileri incelenmiştir. Bu tekniklerin başarısı ve hafıza gereksinimleri deneylerle incelenmiştir. Nicelemenin etkileri sığ, derin, aşırı parametreli, artık bağlantılı ve verimli modellerde incelenmiştir. Model mimarisine özel sorunlar gözlenmiş ve ilgili çözümler önerilmiştir. Nicelenmiş modeller ve baştan verimli tasarlanmış modeller karşılaştırılmıştır. Bu iki yöntemin avantajları ve dezavantajları incelenmiştir. Bunlara ek olarak standart ve nicelenmiş evrişim işlemi çeşitli sistemlerde gerçeklenmiştir. Bu çeşitli sistemler düşük güç tüketimli gömülü sistemlerden güçlü masaüstü bilgisayar sistemlerine uzanmaktadır. Hesaplama zamanı ve hafıza gereksinimleri deneyleri bu gerçek sistemlerde yapılmış ve sonuçları paylaşılmıştır.

Anahtar Kelimeler: Derin Sinir Ağları, Niceleme

To my family

ACKNOWLEDGMENTS

First and foremost, I would like to express my deep and sincere gratitude to my supervisor Prof. Dr. Gözde Bozdağı Akar. Her guidance and support were precious to me for this thesis and my academic career.

I would like to thank Miray Karamehmetoğlu for always being there for me.

Lastly, I would like to thank my family, Esma Kütükcü, Ersin Kütükcü and Başak Kütükcü for their endless support throughout my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ABBREVIATIONS	
BC	BinaryConnect
BNN	Binarized Neural Networks
DRF	DoReFa-Net
DRF-W	DoReFa-Net Weight Quantization
TWN	Ternary Weight Networks
CNN	Convolutional Neural Networks

CHAPTER 1

INTRODUCTION

1.1 Introduction

Developments in deep learning research brought great success to many problems including image classification, image segmentation, natural language processing, recommender systems and many others. Specially in the image related tasks, deep learning techniques showed superior results to existing image processing techniques. However, this success comes with a cost. This cost is memory and computation power requirement of the deep learning models. Many companies and research institutes are increasing these requirements while trying to create better models. These models are being developed for achieving marginal results and therefore require more and more memory and computation power. On the other hand, the number of mobile devices is rapidly increasing every day. These resource-constrained smart devices can benefit greatly from the success of deep learning; however, they are not capable to satisfy the memory and power requirements of deep learning models due to many constraints such as budget, space, power and heat. Therefore, there is a need to adjust power and memory requirements of deep learning models for resource constrained devices.

Deep learning models should be adjusted to be more efficient in order to run on resource-constrained devices. There are currently couple of approaches to create efficient deep learning models. They can be categorized into two main approaches. First approach is to create efficient deep learning models ground up. This approach includes exhaustive state-space search of hyperparameters while monitoring size and accuracy trade-off. The results of this search are used to design efficient deep learning models. These newly designed deep learning models include new efficient architectural units that separate them from existing computation heavy deep learning models. Some examples of these efficient architectural units are squeeze-expand operations [1], depthwise-pointwise convolution operations [2] and shuffle operation [3]. The second approach is to use methods, named quantization and pruning, on the weights of the deep learning models. The difference of this approach and the first one is that this approach does not try to create new models and architectural units. Instead it uses some methods on existing models to convert existing models to more efficient ones. The weights of a deep learning model and their multiplication operations make a model costly. This approach and their quantization/pruning methods focus on the weights to decrease the memory and computation cost of a deep learning model. The quantization method clusters the weights of a deep learning model into some cluster points. The pruning method completely eliminates the weights which are ineffective to the overall results of the deep learning model. Both of these methods eventually decrease the required memory space by the deep learning model. There are different ways to implement this approach by considering when and how these methods are used on the deep learning model. Some methods can be named as quantization/pruning aware training. In this methods, quantization/pruning is done continuously during training. This requires whole training process. Some other can be named as post quantization/pruning. In these methods quantization/pruning is done on a trained model. These methods are employed by many frameworks which repetitively quantize/prune the trained model and re-train it to recover the lost accuracy up to a point.

1.2 Scope of Thesis

In this thesis, certain quantization methods are selected, analyzed, experimented and benchmarked. The aim of the thesis is to investigate the effects of quantization on deep neural networks and suggest solutions to certain quantization problems.

The scope of this thesis is quantization aware training-based methods. However, the other quantization methods are also researched and explained in the literature review for the sake of completeness.

Background information for neural networks and quantization is given. Historical development of neural networks and deep learning is explained briefly. The training algorithm of neural networks is explained considering its importance for quantization operations. Convolutional neural networks are explained since they are used in the experiments. Compression property of quantization is explained. Straight through estimators and the importance of full precision weights during training are explained since they are the key factors that enable quantization aware training of deep neural networks. Literature review of ground-up efficient models is given. Literature review of quantization aware training methods, post quantization and usage of reinforcement learning for quantization is given.

Some methods in the quantization aware training are selected considering their success and similarities. They are explained in detail. They are implemented on different datasets and deep learning architectures. They are benchmarked on a simple dataset and architecture. They are implemented on relatively shallow and overparameterized model. Some hybrid methods are proposed for the weights and activation quantization methods. One of the hybrid methods has hardware friendly cluster points and a better accuracy than existing hardware friendly methods. They are compared with a ground-up efficient model with similar capacity. They are implemented on a deeper and more complex model. Using this model, they are compared with each other and a ground-up efficient model with similar capacity. Problems of quantization of deep and complex models are demonstrated. Various solutions to these problems are suggested. These solutions are implemented and shown to be effective. The reason of relatively low accuracy of weight and activation quantization of deep and complex models are shown and empirically proven. Moreover, a ground-up efficient model is also quantized and compared with its full precision version.

Full precision and quantized convolution operation are implemented for various systems. These systems include a very low power embedded system, an average embedded system and a powerful desktop computer. Therefore, the compression property of quantization is proven on real systems. The trade-offs of this simple implementation are investigated and explained.

1.3 Outline of Thesis

In Chapter 2, background information and literature review are given. In Chapter 3, quantization aware training-based methods are selected and explained. In Chapter 4, experimental results are given, certain problems of quantization are demonstrated, solutions to these problems are suggested. In Chapter 5, system implementation of quantized convolution operation is shown and experimented.

CHAPTER 2

BACKGROUND AND LITERATURE REVIEW

2.1 Background

In this section, a fundamental information on neural networks and quantization is given. A brief history of neural networks and their evolving to today’s successful deep learning models are explained. The algorithm of training neural networks is explained since it is important for quantization algorithms. Convolutional neural networks are explained in detail as they are used in the experiments in this thesis. Fundamental information of compression in quantization is explained since it is used to evaluate and compare different quantization algorithms. The key points of training quantized deep neural networks are explained.

2.1.1 Neural Networks

2.1.1.1 Brief History

Neural networks compose a branch of machine learning techniques. Even though neural networks are one of the most popular machine learning techniques nowadays, they are not proposed in the near past. The roots of today’s neural networks get to the perceptron [4] which is proposed in 1958. The perceptron is just a neuron which sums the weighted input features and applies a step function to that summation.

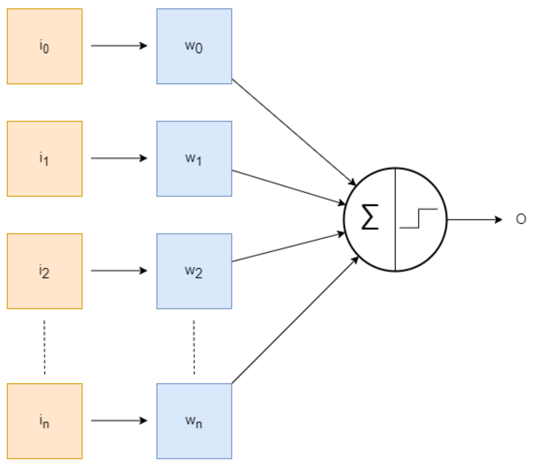


Figure 2.1. The illustration of the perceptron

Later on, perceptron is proven to be uncapable of doing many things [5] such as representing non-linear functions like XOR. This caused a significant and rapid decline in neural network research at that time. Even though perceptron (or single layer perceptron) was not very capable, the multi-layer perceptron was much more capable. For example, a multi-layer perceptron with at least three layers can represent non-linear functions unlike single-layer perceptron. Multi-layer perceptron can be seen as a fundamental example of today's shallow neural networks.

Even though neural network research regained its momentum in 1980s, it could not achieve the today's popularity until 2012. Because there were two things that prevent neural networks from unleashing their true potential. The first one was lack of data and the second one was lack of computing power. The problem of insufficient data was solved with the help of internet. The developing technologies of computer networking and internet allowed people from all around the world to upload data which was openly accessible to everybody. This data abundance was converted to training data for neural network with a certain effort. The problem of insufficient computing power was solved with the invention of graphics processing unit (GPU) and its general programming capability. The training of a neural network is a quite computing heavy procedure. However, neural networks have an advantage which is their availability to parallelization. Since GPU is developed for processing graphics, it is a powerhouse for single instruction multiple dataset (SIMD) procedures. Central processing unit also can run parallel algorithms as many as its core number which is limited to a low number (typically 4-8). On the other hand, GPU can run massively parallel algorithms. In 2012, AlexNet [6] won ImageNet competition and ignited the today's deep learning popularity.

2.1.1.2 Training Neural Networks

There are two stages in the process of training neural networks. One is forward pass and the other one is backward pass. During forward pass, some amount of data is fed into the input layer of a neural network. Each layer gets the previous layer's output, process it, and outputs it for the next layer. The final layer outputs various number of values, depending on the task and architecture of the neural network. In supervised learning, there is a ground truth value for each of these output values for every input data. The aim of training is to adjust the parameters (weights) of the neural network so that it generates outputs which are close to the ground truth values as much as possible. Backward pass updates the parameters with a certain algorithm. But before backward pass starts, the error is calculated. The error is some kind of difference metric between outputs of the neural network and the ground truth values. The loss functions are used for calculating error. There are many different loss functions for different purposes. Once the appropriate loss function is selected, it is minimized during backward pass by an optimizer. There are also different optimizers such as stochastic gradient descent or Adam. The optimizer updates parameters while calculating gradients for weights starting from last layer to first layer. Gradients are scaled with a predefined value named learning rate. Then the scaled gradients are used to update the parameters. The learning rate is an important hyperparameter and choosing it effects the convergence of training greatly.

The backpropagation algorithm is the most fundamental part of training feed forward neural networks. Consider the neural network in Figure 2.2 for the following explanation of backpropagation algorithm. The example neural network has 3 layers which are fully connected. Each arrow is associated with a weight. A neuron *i* in the network sums all of its weighted inputs to come with *net$_i$*. Then *net$_i$* is fed into an activation function *f(.)* which results *y_i* or *z_i* depending on the neuron's layer's position. At the end of the network, *z* values are the predictions or outputs of the network and *t* values are the ground truth values of the related inputs.

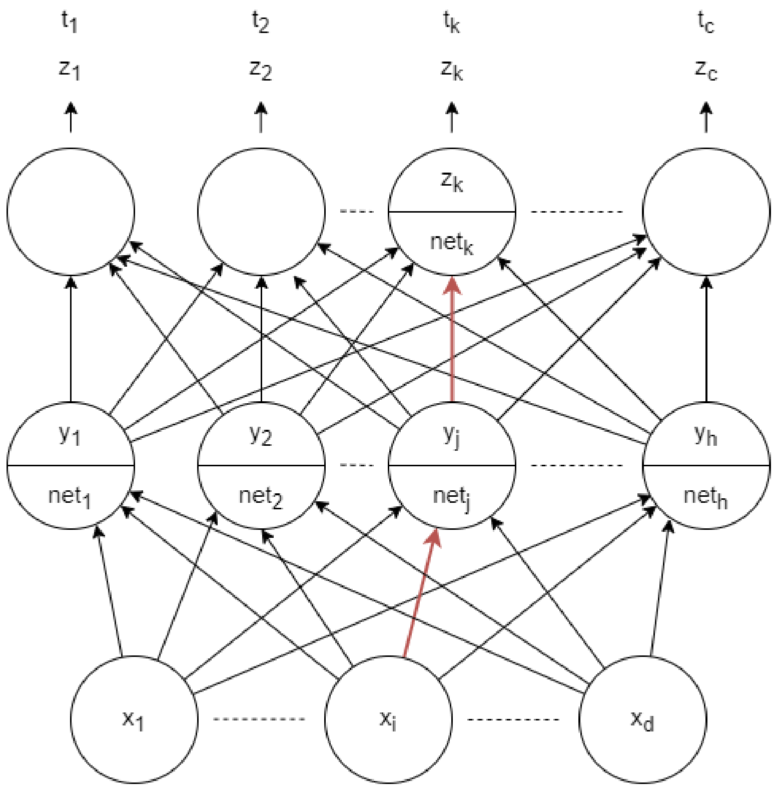


Figure 2.2. Example neural network

For this example, consider the cost function in (2-1).

$$J(w) = \frac{1}{2}(\bar{t} - \bar{z})^2 \tag{2-1}$$

The update rule of backpropagation algorithm is shown in (2-2). m is the iteration number. w is a particular weight that connects two neurons. J is the cost function. α is the learning rate. The aim is finding Δw and using it to update w_m to w_{m+1}.

$$\begin{gathered} w_{m+1} = w_m + \Delta w_m \\ where\ \Delta w = -\alpha \frac{\partial J}{\partial w} \end{gathered} \tag{2-2}$$

The update with backpropagation algorithm on two weights, which are shown with red arrows in the network, is explained below. First, consider the w_{kj} which connects neuron j to neuron k. Since the learning rate is constant, backpropagation algorithm only needs to find the gradient as in (2-3).

$$\frac{\partial J}{\partial w_{kj}} = \frac{\partial J}{\partial net_k} * \frac{\partial net_k}{\partial w_{kj}} \tag{2-3}$$

The second term is simply equal to related previous neuron's output as in (2-4).

$$\frac{\partial net_k}{\partial w_{kj}} = y_j \tag{2-4}$$

We define a special term for future references as in (2-5).

$$\begin{aligned} define\ \delta_k &\triangleq -\frac{\partial J}{\partial net_k} \\ &= -\frac{\partial J}{\partial z_k} * \frac{\partial z_k}{\partial net_k} \\ &= (t_k - z_k) * f'(net_k) \end{aligned} \tag{2-5}$$

As a result, the gradient is found as in (2-6).

$$\frac{\partial J}{\partial w_{kj}} = -\delta_k * y_j \tag{2-6}$$

The resultant gradient is used to calculate Δw_{kj} as in (2-7). Every term in this equation is known during backward pass. Therefore, w_{kj} can be updated accordingly.

$$\begin{aligned} \Delta w_{kj} &= -\alpha * \left(-\delta_k * y_j\right) \\ &= \alpha * \delta_k * y_j \\ &= \alpha * (t_k - z_k) * f'(net_k) * y_j \end{aligned} \tag{2-7}$$

The procedure for w_{ji} is a bit different. The gradient for w_{ji} is extended using chain rule as in (2-8).

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial y_j} * \frac{\partial y_j}{\partial net_j} * \frac{\partial net_j}{\partial w_{ji}} \tag{2-8}$$

The first term can be extended as in (2-9). Note that we can see δ_k in the extended form. δ_k is actually backpropagated from the next layer.

$$\begin{aligned} \frac{\partial J}{\partial y_j} &= \frac{\partial}{\partial y_j}\left[\frac{1}{2}\sum_{k=1}^{c}(t_k - z_k)^2\right] \\ &= -\sum_{k=1}^{c}(t_k - z_k) * \frac{\partial z_k}{\partial y_j} \\ &= -\sum_{k=1}^{c}(t_k - z_k) * \frac{\partial z_k}{\partial net_k} * \frac{\partial net_k}{\partial y_j} \\ &= -\sum_{k=1}^{c}(t_k - z_k) * f'(net_k) * w_{kj} \end{aligned} \tag{2-9}$$

The second and third term can be found as in (2-10). The second term is nothing but the derivative of the activation function. The third term is the output of the related previous neuron which turns out to be input for this particular example. However, it could have been the output of a neuron in another hidden layer.

$$\begin{aligned} \frac{\partial y_j}{\partial net_j} &= f'(net_j) \\ \frac{\partial net_j}{\partial w_{ji}} &= x_i \end{aligned} \tag{2-10}$$

We can again define another special term for future references as in (2-11).

$$define\ \delta_j \triangleq f'(net_j) * \sum_{k=1}^{c} w_{kj} * \delta_k \tag{2-11}$$

As a result, the gradient is found as in (2-12).

$$\frac{\partial J}{\partial w_{ji}} = x_i * f'(net_j) * (-\sum_{k=1}^{c} w_{kj} * \delta_k)$$
$$= x_i * (-\delta_j) \quad (2\text{-}12)$$

The resultant gradient is used to calculate Δw_{ji} as in (2-13). All of the terms in this equation are known during the backpropagation as well. Therefore, w_{ji} can also be updated using the update rule during backward pass.

$$\Delta w_{ji} = \alpha * x_i * \delta_j \quad (2\text{-}13)$$

To update a weight in a layer, there is always a term needed from the next layer. This is the result of error calculation at the final layer. These terms are backpropagated starting from final layer to the first layer. The backpropagated terms are the special definitions that we made. Those special terms are δ_k and δ_j. Note that δ_k is only used for backpropagation from output layer to a hidden layer. On the other hand, δ_j is used for hidden layer to hidden layer and hidden layer to input layer.

A clear understanding of backpropagation algorithm is necessary for quantization research. Note that chain rule is extensively used in the backpropagation algorithm. This means every function that is used in the forward pass must be differentiable. However, the quantizer functions are not differentiable. Therefore, a workaround is required to train quantized neural networks with backpropagation algorithm. This workaround is explained in the following quantization background information.

2.1.1.3 Convolutional Neural Networks

There are many different types of deep neural networks. Every different deep neural network type has advantages and disadvantages. They have architectural differences which make them fit to certain tasks better than others. For example, recurrent neural networks and long short-term memory units (LSTMs) [7] are good for tasks with sequential data such as natural language processing (NLP) and speech recognition.

On the other hand, convolutional neural networks are good for tasks with visual data such as images. In this section, convolutional neural networks are explained in detail since image classification task with convolutional neural networks is used for the experiments in this thesis.

There are couple of reasons of convolutional neural networks' superior performance with image data compared to regular neural networks. First of all, they are scalable with image data. A regular neural network requires a weight for each pixel in the first layer of the network. The required amount of weights increases rapidly in the following layers. Therefore, total number of parameters become quite large. As a result, the model becomes inefficient for the task and it easily overfits the data due to huge number of parameters. On the other hand, convolutional neural networks employ filters which have various sizes independent from the size of the image. A filter is used by sweeping it on the image to calculate the next layer's input. Therefore, a weight in a filter is re-used for multiple pixels, unlike regular neural networks. Secondly, the shapes of weights in convolutional neural networks are specifically designed for visual data. A regular neural network uses a weight for each pixel. The approach of regular neural network does not consider the shape and type of the data. It simply serializes the input data and acts to the data as a set of features. On the other hand, the filters of convolutional neural networks have a similar shape to images. For example, images have three dimensions, namely width, height and channels. Filters also have the same three dimensions and one more dimension which is named output channels. While the channel dimension is same for the input data and filters, the width and height of the filters are typically smaller than the width and height of the input data. The number of output channels determines the channel number of the output of convolution operation. This shape similarity causes better results for image related tasks in the deep neural networks.

A typical convolutional neural network has three main layers. These layers are stacked to come up with deep convolutional neural networks. They are convolution layer, pooling layer and the fully connected layer. While there may be other type of layers, these three typically define the convolutional neural networks.

2.1.1.3.1 Convolution Layer

In this layer, the convolution operation is applied to the input with the filters of the layer. The aim is to catch a certain local pattern in the image. In the earlier layers of the neural network, these patterns are low level features such as edges or rounded shapes. In the late layers of the neural network, these patterns become high level features such as eye or nose. The input is a 3D array with dimensions names width, height and channel. The output is also a 3D array with same dimension names since it is typically input to another convolution layer. The filter is a 4D array with dimension named input channel, width, height and output channel. The values of input channel and output channel of the filter must be same with channel values of the input and output, respectively. Elements of a simple convolution operation is illustrated in Figure 2.3.

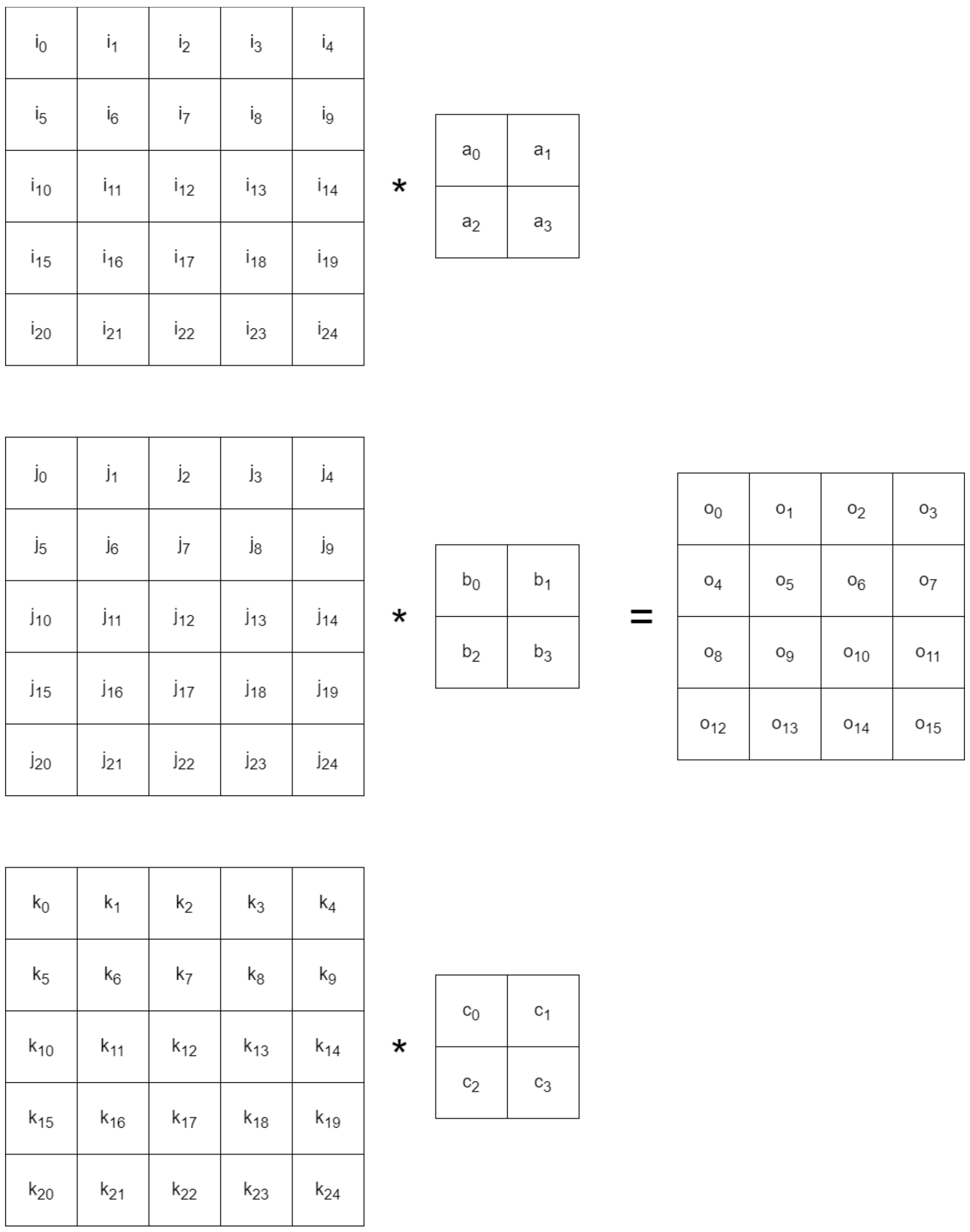


Figure 2.3. Convolution operation

In this simple example, the input has 3 channels each of them has 5x5 pixels. The filter has also 3 input channels, 2x2 weights for each input channel and 1 output channel. The filters are swept in their input, the matching cells are multiplied and the results of 3 channels are accumulated in the related output cell. The calculation of two output cells are shown as examples in (2-14).

$$
\begin{aligned}
o_0 = {} & (a_0 * i_0 + a_1 * i_1 + a_2 * i_5 + a_3 * i_6) \\
& + (b_0 * j_0 + b_1 * j_1 + b_2 * j_5 + b_3 * j_6) \\
& + (c_0 * k_0 + c_1 * k_1 + c_2 * k_5 + c_3 * k_6) \\
o_9 = {} & (a_0 * i_{11} + a_1 * i_{12} + a_2 * i_{16} + a_3 * i_{17}) \\
& + (b_0 * j_{11} + b_1 * j_{12} + b_2 * j_{16} + b_3 * j_{17}) \\
& + (c_0 * k_{11} + c_1 * k_{12} + c_2 * k_{16} + c_3 * k_{17})
\end{aligned}
\tag{2-14}
$$

Note that the output channel number of the filters is 1. Therefore, there is only one output frame calculated. If the output channel number would be larger than 1, the same operation would be repeated with a different set of filters to calculate output channel number times output frames.

One important point in the convolution operation is the output spatial sizes. While the input’s spatial size (width and height) is 5, it is decreased to 4 in the outputs. This is inevitable since the filters have also spatial sizes larger than 1. However, sometimes it is required to keep the spatial sizes same through convolution layers. In this case, the general solution is to apply padding with zero to the input. This is basically adding zero elements around the inputs. The required depth of padding changes with the filter size.

Another important point in the convolution operation is the stride. Stride is the number that determines how to sweep the filters on the inputs. If it is 1 as in the example, the filters move one cell by one cell. If it would be 2, then the filters would move two cells by two cells. Increasing the stride results in decreased output spatial sizes.

The output spatial size can be calculated with the formula in (2-15). In this formula, O is the output spatial size, I is the input spatial size, F is the filter spatial size, P is the padding depth, and S is the stride.

$$O = \frac{I - F + 2P}{S} + 1 \tag{2-15}$$

2.1.1.3.2 Pooling Layer

The aim of this layer is downsampling. While downsampling can be an easy task, making it useful to the task can be a bit tricky. This layer, like convolution layer, uses the fact that the data is visual. It focuses to locality of the features. It downsamples the data by forwarding the useful features to next layer while blocking the less important ones. One advantage of the pooling technique is that it does not use any parameters. Instead, it uses a heuristic to decide what to forward to next layer. The pooling operation works by sliding a window on the input, like convolution layer. At each step, a certain operation is applied to the cells within the window and just one cell is forwarded to the next layer. Therefore, the spatial size of the data decreases. The operation is applied to each channel separately. As a result, the data becomes denser in terms of information.

The two pooling techniques are illustrated in Figure 2.4. In this simple example, the input is 1 channel 4 by 4 data. The pooling window size is 2 by 2. The shown results are for techniques named as maximum pooling and average pooling. As the names suggest, the procedures for both of them are straight-forward. In the maximum pooling, the values of the cells in the pooling window are compared and the one with maximum value is forwarded to the output. In the average pooling, the values of the cells in the pooling layer are averaged and the result is forwarded to the output. The stride concept in the convolution layer is also applicable in the pooling layer. Increasing the stride further reduces the spatial size of the outputs.

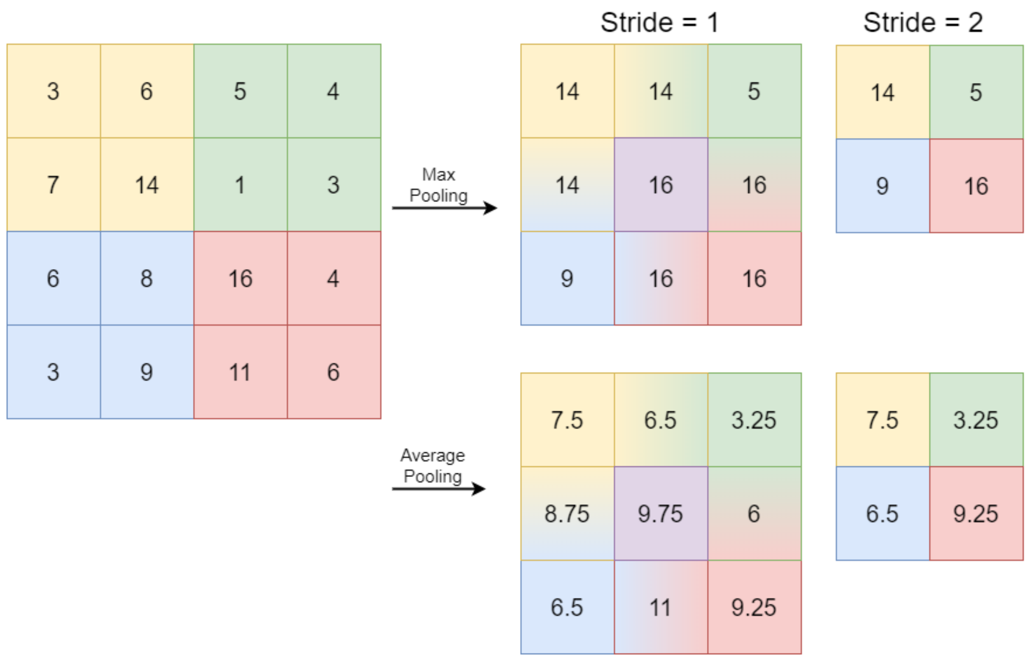


Figure 2.4. Pooling operations

The output spatial size of the pooling layer is calculated using (2-16).

$$O = \frac{I - W}{S} + 1 \tag{2-16}$$

In this equation, O is the output spatial size. I is the input spatial size. W is the window size. S is the stride.

2.1.1.3.3 Fully Connected Layer

This layer is the fundamental part of the regular neural network as well. Convolutional neural networks use fully connected layers at the end of the neural network. Generally, the convolution and pooling layers decrease the spatial size and increase the channel number of data through neural network. Once the spatial size gets small enough and channel number gets large enough, the data is flattened. The flattening operation is basically converting 3D data to 1D data just by serializing. After flattening, depending on the architecture, one or more fully connected layers are used. Note that there are some convolutional neural network architectures that do not use fully connected layers at all. Therefore, even though it is very common to use at least one fully connected layer in convolutional neural networks, it is not vital like other two layers. Some illustrations of fully connected layers can be seen in Figure 2.2.

2.1.2 Quantization

2.1.2.1 Compression

One of the main goals of quantization is compression. Many state-of-the-art deep learning models have parameters that cost memory in the range of MBs. In Figure 2.5, pre-trained models in Keras library [8] are shown with their sizes. The smallest required size is 14 MB for MobileNetV2. The largest required size is 549 MB for VGG-19. These models are impossible to store in on-chip memory of many embedded processors. The off-chip memory access can be too expensive for inference on top of the low processing power of embedded systems. Moreover, some embedded systems do not have the required off-chip capacity to store these models. Even if some embedded systems have the required off-chip memory capacity to store these models, they can benefit from compression by putting the whole model in their on-chip memory.

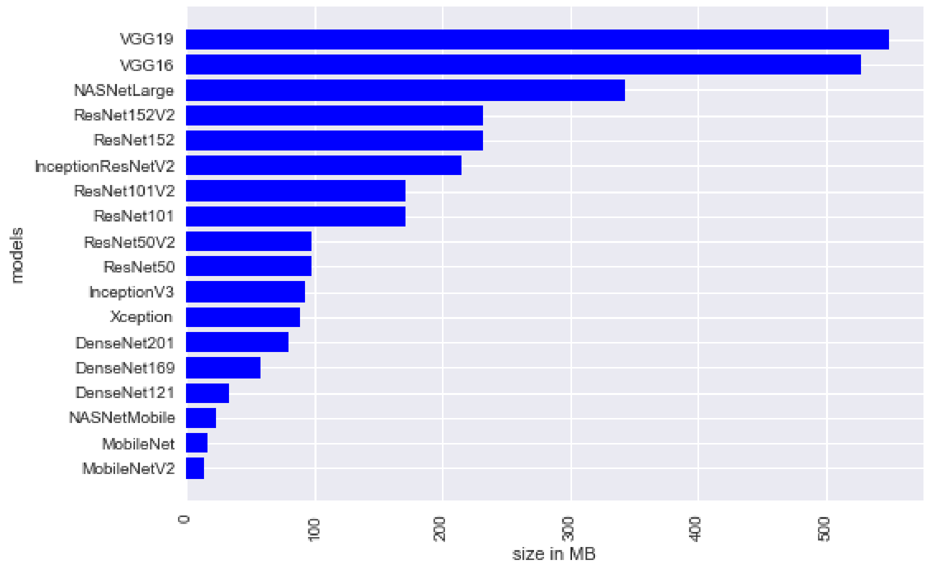


Figure 2.5. Keras pre-trained models and their sizes

Compression with quantization is best described in Deep Compression [9]. The following formula from the paper calculates the compression rate of any quantized neural network.

$$r = \frac{nb}{n \log_2 k + kb} \qquad (2\text{-}17)$$

In the formula, n is number of connections, b is the number of bits to represent each weight prior to quantization, and k is the cluster number after the quantization process. One important point about k is that it must be power of 2 because of the binary representation. For example, if 3 clusters will be used for a quantization, k is rounded to 4. Because, at least 2 bits are required to represent 3 clusters.

In the process of quantization, the full precision weights are quantized to a number of cluster values. Then those cluster values are saved in a full precision array. Then each weight saves the index of that array instead of the actual cluster value. The term nb represents the required total number of bits before quantization. $\log_2 k$ represents the required number of bits to represent all of the indexes. kb represents the required number of bits to save the array of cluster values. Consider a hypothetical 3x3x3 filter for a convolutional layer. Quantization of this filter is illustrated in Figure 2.6. A simple linear quantization to 4 clusters is used. For this particular example, n is 27, b is 32 and k is 4. When we use these values in the formula, the compression rate is calculated as 4.74.

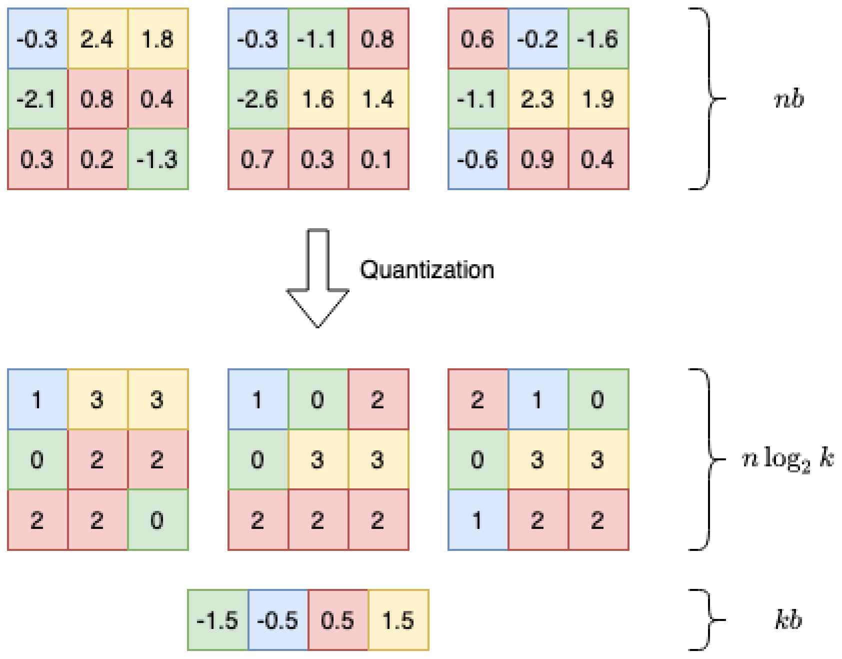


Figure 2.6. Compression example

Note that this is a pretty simple filter with only 27 weight values. However, deep neural networks have parameters ranging from thousands to millions. Quantizing those deep neural networks simplify the formula (2-17). When n goes to infinity, the term kb becomes negligible. Thus, the formula becomes simply the ratio of b to $\log_2 k$. Since the deep neural networks are almost always trained with 32-bit floating point precision ($b = 32$), the possible compression rates are 32x,16x,8x,4x and 2x depending on the cluster number (k). These compression rates are true for homogeneous quantization, i.e. all layers are quantized to same number of clusters. If some kind of heterogeneous quantization is used, different compression rates can be achieved and still can be calculated using (2-17) by expanding related terms.

2.1.2.2 Straight Through Estimator

Quantizer functions are the main algorithms of the quantization research. Since quantizer functions cluster full precision weights into some cluster points, they can be seen as some variants of the step function. As in the step function, the quantizer functions' derivative is calculated as 0 at almost entire space. Therefore, quantizer functions block the gradient flow during backpropagation due to the chain rule, which prevents the model from learning its task. As a result, a workaround is required to work with backpropagation algorithm during quantization aware training. The most common method is to use straight-through estimators [10][11]. Straight-through estimators basically replace a function's derivative with a predefined function. Most of the time this predefined function is the identity function or slightly modified version of the identity function. The mathematical notation of forward pass and backward pass of a quantizer function *f(x)* with identity straight-through estimator can be seen as

$$\begin{array}{ll} \textit{Forward pass} & y = f(x) \\ \textit{Backward pass} & \frac{\partial J}{\partial x} = \frac{\partial J}{\partial y} \end{array} \tag{2-18}$$

In this specific example, x is a full precision value which is mapped to one of the limited values of y, by using *f(.)* function. J denotes the cost function. During backward pass, the gradient of y is calculated with backpropagation algorithm. Then in this particular step during backward pass, straight-through estimator is used to equate the gradient of *x* to gradient of *y*, independent from the function *f(.)*.

2.1.2.3 Full Precision Weights in Training

One of the most important rules of training quantized neural networks is keeping the full precision weights during training. This is required since gradient descent methods work with very small updates. For a clear explanation, consider the figure below.

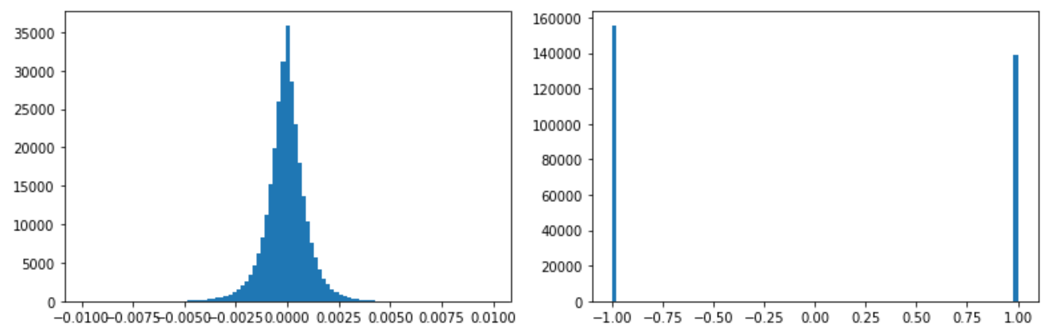


Figure 2.7. Full precision weights (left) and quantized weights (right)

In Figure 2.7, full precision weights and quantized weights of a layer is shown. A simple quantization technique is followed where negative values are quantized to -1 and other values are quantized to 1. If the full precision weights were discarded after quantization, updates were done on quantized weights and the updated weights were quantized using the same technique, the distribution of quantized weights would never change. Because as mention previously, the gradient descent algorithm works with small update to converge. No update could flip one weight to other side. Even if the magnitudes of updates are increased with learning rate, the model could not converge.

During forward propagation, full precision weights are first quantized, then quantized weights are used for the operation of the layer. So, the full precision weights are never directly used in the layer. The full precision weights are there to provide accurate quantized weights. In most of the methods, after layer operation is done, quantized weights are discarded. The quantized weights are not saved but instead they are created from the full precision weights using their quantization technique at every forward propagation. During backward propagation, updates are done to full precision weights. The quantization operation is bypassed since its derivative is zero.

As a result, the full precision weights are trained while quantized weights are used. At the end of the training full precision weights and quantization operation are discarded. The quantized weights are put in the place of full precision weights and the inference works as usual.

2.2 Literature Review

2.2.1 Efficient Models Ground Up

In this branch of efficient deep learning research, analyzation of current knowledge and state-space search of hyper-parameters are used to come up with new efficient deep learning architectures.

In [1] , an analysis of CNN architectures resulted in 3 strategies. The first one is to use 1x1 filters instead of 3x3 ones as much as possible. The second one is to keep the number of input channels to 3x3 filters as small as possible. The last one is to downsample in the network as late as possible. Using these strategies, a new convolution block, namely Fire module, is designed. Then new CNN architectures are built using the Fire module repetitively. The parameters of the CNN architectures with Fire modules are determined by a state-space search considering accuracy and model size. These CNN architectures achieved the same accuracy with AlexNet[6] with 50 times less number of parameters.

In [2], standard convolution operation is separated into two new operations, namely depthwise convolution and pointwise convolution. These operations require a smaller number of parameters than standard convolutions while achieving almost as much as feature representation with standard convolutions. In a follow-up work [12], linear bottlenecks and inverted residuals are added to depthwise convolutions to create even more efficient models.

In [3], a channel shuffle operation is applied after 1x1 group convolutions. The channel shuffle operation is followed by a depthwise convolution operation and another group convolution. A shortcut path is also added to these four operations to create one ShuffleNet unit. These ShuffleNet units are used repetitively to create efficient deep learning models.

2.2.2 Quantization Based Approaches

2.2.2.1 Quantization Aware Training and Post Quantization

In this branch deep learning models are trained while quantization is being considered. In order to do that, quantizer functions are defined. These quantizer functions are applied on weights and activations. Quantization takes place during forward propagation most of the times. Some works quantize only weights, some works quantize only activations and some other works quantize both of them.

In [13], the weights of neural networks are binarized. Therefore, the weights are constrained to the values of -1 and 1. Two different goals and their specific quantizer functions are proposed. Straight-through estimator is used for backpropagation in both of the quantizer functions. The first goal is to quantize the weights in order to create a smaller and more efficient model. The quantizer function for the first goal is *sign(.)* function. After training, the full precision weights are completely replaced with binary ones. The second goal is to use quantization for regularization purposes. A stochastic quantizer is proposed for this goal. After training, the full precision values are continued to be used. This work is followed by an extension [14] which quantizes the activations as well as weights. The activation quantization is done with *sign(.)* function and therefore, activations can have values of -1 and 1. Quantizing both weights and activations to the values of -1 and 1 brings a significant computation efficiency since the matrix multiplication can be replaced with a bitwise XNOR operation. This work proves this efficiency by writing an XNOR GPU kernel which is reported to be 5.3 faster than conventional unoptimized matrix multiplication kernel. Furthermore, this work also proposes a shift-based batch normalization method to accelerate training as well.

In [15], the weights are quantized with *sign(.)* function and then multiplied with a scaling factor. The scaling factor is determined by taking the average of absolute weight values of each filter. The activations are also quantized similarly, using *sign(.)* function and a scaling factor. Straight-through estimator is used for backpropagation of quantizer functions. They benchmarked neural networks by quantizing only the weights and quantizing both the weights and activations. They reported their results on ImageNet dataset.

In [16], the weights, activations and gradients are quantized. All of the quantizer functions are used with straight-through estimator in backpropagation. Different levels of quantization are tried and benchmarked. For binary quantization, *sign(.)* function is used with a scaling factor. The scaling factor is determined by taking the average of absolute values of the weights in whole layer. For higher level of quantization, i.e. quantization requiring 2 bits or more, a linear quantizer function is defined with clipping. For activation quantization, a linear quantizer function is used. For gradient quantization, a stochastic quantizer function is defined. This function employs a noise factor which is reported to be necessary to compensate the potential bias introduced by quantization.

In [17], the weights are clustered into three values, i.e. ternary values. Three cluster values require two bits to be represented distinctly. The quantizer function assigns the full precision values to these three cluster points. The quantizer function works with a threshold value. If a full precision value is larger than the threshold, it is quantized to 1. If a full precision value is smaller than the negative of threshold, it is quantized to -1. All other values are quantized to 0. After quantization, a scaling factor is applied to quantized values. The scaling factor is determined by taking the average of absolute values of the full precision weights which are larger than the threshold.

In [18], the weights are clustered into three values. The quantizer functions assigns the full precision values to cluster values based on a threshold. The cluster values are zero, a positive cluster value and a negative cluster value. Positive and negative cluster values are not symmetrical. These two cluster values are trainable. In other words, they are learnt during backpropagation to minimize the overall loss. This comes with a disadvantage which is the resultant ternary weights are not symmetrical. Asymmetrical weights are harder to implement in hardware for a possible acceleration.

In [19], the weights of a pre-trained model are quantized using an incremental algorithm. The incremental algorithm is divided into three stages. These stages are names as weight partition, group-wise quantization and re-training. In the weight partition stage, the full precision weights (all of the weights at the start of the algorithm) are divided into two groups. In the group-wise quantization, one group is quantized to values which are either 0 or power of 2. Then, this group is frozen. In last stage, re-training, the other group (not quantized one) is re-trained to compensate the accuracy loss caused by the quantization of the first group. These three stages are repeated on the remaining full-precision weights. At every repetition, the number of full precision weights is decreased and eventually all of the weights are quantized to either 0 or power of 2. The cluster values are chosen specifically to be implemented by bit shift operations. Therefore, expensive multiplication operation can be replaced with bit shift operations. When bit width of 5 is used, the quantized models achieve accuracies which are similar to or better than full precision models' accuracies. Also, state-of-the-art results are delivered with aggressive quantization levels which are experimented down to 2 bits.

In [20], the 32-bit full precision weights are quantized to 8-bit integers. The proposed quantizer function is actually an affine mapping of 8-bit integers to 32-bit full precision weights. In order to do this mapping, two parameters are defined for the quantizer function. First one is called zero-point parameter which corresponds to 0 in the domain of full precision weights. In other words, the affine mapping of zero-point parameter (an 8-bit integer) is 0 which is represented with 32-bit floating points. The second parameter is scaling factor which is defined as the ratio of range of full precision weights to range of quantized weights. Considering these two parameters, the quantizer function becomes

$$w = S * (w_q - z) \qquad (2\text{-}19)$$

Where w, S, w_q, z represent full precision weights, scaling factor, quantized weights and zero-point parameter, respectively. This method is used as both post-quantization method and quantization-aware training method. However, it is reported that when it is used as post-quantization method, small neural networks face with a large accuracy drop which may be compensated with re-training. The best results are achieved when it is used as quantization-aware training method. This method is realized with TensorFlow Lite.

There is also another branch of research where a reinforcement learning agent is trained to automatically determine some hyperparameters related with quantization. The searched hyperparameters are mostly the level of quantization for each layer. In other words, the agent decides the bitwidth of the quantization for each layer. The reasoning behind using reinforcement learning framework for this task is that the state space is huge for deep neural networks. Even if the maximum bitwidth of quantization is limited to 8 for all layers, the state space becomes 8^n where n is the number of layers. Therefore, the state space increases exponentially with the number of layers.

In [21], a reinforcement learning framework is proposed to find the quantization level of the weights of deep neural networks. Special state representation and reward function are defined. The state representation includes information about the bitwidth of the quantized weights, memory access energy, multiply accumulate operations, quantized model’s accuracy and full precision model’s accuracy. The reward function gives the importance on preventing accuracy drop over compression. That results in a behavior where the agent compresses the deep neural network as long as the current action does not cause a significant accuracy drop. The actions of the agent are limited to choosing a bitwidth between 1 and 8. After the agent makes its action, the model is retrained and then the reward is calculated. Proximal policy optimization is used to train the agent. A simple linear quantization technique is used for all layers. The weights are first clipped to the range of -1 and 1, then linearly quantized depending on the chosen bitwidth.

In [22], a hardware-aware reinforcement learning framework is proposed. The purpose is to find quantization levels of the layers in a deep neural network. The main contribution is that this work does not observe only the metrics of the neural network, but also gets feedback from simulator of hardware accelerators. Therefore, the agent quantizes the neural network considering both the properties of the neural network and the hardware accelerator which it will run on at the end. The observation state includes information on input-output channel sizes, kernel and stride sizes, input feature size, number of parameters and a separating indicator for depthwise convolution for convolutional layers. The same information for fully connected layers includes input-output hidden unit sizes, input feature size, number of parameters. A binary indicator and the previous action are also added to both of the layers' state representations. All representations are normalized to the range [0,1]. A continuous action space is used. The agent outputs an action value between 0 and 1, then this value is rounded to an integer bitwidth value, typically between 2 and 8. The reward signal is simply the scaled difference of accuracies of quantized model and original model. In the framework, quantizing a layer is a step while quantizing all layers is an episode. After finishing an episode, a feedback signal of hardware constrains is received from the simulator of hardware accelerator. If the resulting model exceeds a hardware constraint, the bitwidths of layers are sequentially decreased until the constraint is satisfied. Deep deterministic policy gradient is used for the agent. Given the bitwidth of a layer, a linear quantization function is used.

In [23], a reinforcement learning is proposed to automatically prune neural networks. The goal is to reduce the number of parameters in a neural network using a trained agent. Unlike quantization works, the remaining parameters of the weights are represented full precision. The observation state includes index of the layer, input and output channels, height and width of the input, stride and size of the kernel, FLOP number, reduced FLOPs in previous layers and remaining FLOPs in next layers, previous action on current layer. All of the observation quantities are normalized to the range 0 and 1. The agent outputs an action which is the compression rate which is a value between 0 and 1. This compression rate is calculated with two different way depending on the pruning method. For [9], compression rate is the ratio of number of zeros to number of all parameters. For channel pruning in [24], the compression rate is the ratio of the number of post-pruning channels to the number of after pre-pruning channels. A maximum allowed compression rate is put to limit the actions of the agent. Deep deterministic policy gradient is used for the agent

CHAPTER 3

QUANTIZATION AWARE TRAINING BASED METHODS

We have analyzed the most successful and known selected methods from the literature and explained them in this chapter. There are couple of reasons why we chose these methods. Firstly, they are the most successful and known methods in their sub-category which is quantization-aware training. Secondly, they quantize weights during forward propagation. Lastly, their quantization heuristics are similar since all of them use full precision weights and their distribution to come with quantized weights.

3.1 BinaryConnect

BinaryConnect [13] is proposed to quantize weights into two cluster values -1 and 1. In order to do this quantization, they applied the function below to the weights

$$w_q = \begin{cases} +1 & if\ w \geq 0, \\ -1 & otherwise. \end{cases} \tag{3-1}$$

Where w_q denotes quantized weights and w denotes the full precision weights. Furthermore, the weights are clipped to the interval [-1,1] right after the weight updates. This clipping is applied in order to fully utilize the quantizer function since if weights get much bigger than the range of quantized values, weight updates do not change anything. This quantizer function is used when the aim is to train a quantized neural network. Therefore, after the training is finished, the full precision weights are completely replaced with quantized weights.

BinaryConnect proposed another quantizer function which has a different purpose than training a quantized neural network. This quantizer function's aim is to regularize the neural network as in the Dropout [25] concept. It is a reasonable aim since quantization adds noise to the weights as in Dropout. The used quantizer function is

$$w_q = \begin{cases} +1 & with\,probability\; p = \sigma(w), \\ -1 & with\,probability\; 1-p. \end{cases} \tag{3-2}$$

Where $\sigma(.)$ is the hard sigmoid function which can be formulated as

$$\sigma(x) = \max(0, \min\left(1, \frac{x+1}{2}\right)) \tag{3-3}$$

This stochastic quantizer function is used during training. However, the inference is done with full precision weights. As a result, the quantization is used only for regularization at training, just like Dropout.

3.2 Binarized Neural Networks

The work of Binarized Neural Networks [14] is an extension to BinaryConnect. The BinaryConnect's quantizer function (3-1) and clipping concept is used for weights, without a change, in Binarized Neural Networks. The one significant addition to BinaryConnect is the quantization of activations. The same quantizer function (3-1) is used for activations as well. Even though same clipping concept is used with BinaryConnect, this work puts the clipping operation into straight-through estimator of the quantizer function as

$$\frac{\partial C}{\partial w_q} = \frac{\partial C}{\partial w} * 1_{|w| \leq 1} \tag{3-4}$$

Here C is the cost function. The term $1_{|w| \leq 1}$ is equal to 1 if the absolute value of w is smaller than 1, and equals to 0 otherwise. Actually, this clipping behavior can be simply implemented using the hard tanh function below

$$y = Htanh(x) = \max(-1, \min(1, x)) \tag{3-5}$$

Since the derivates of horizontal lines will be 0, backward pass through hard tanh function will work as clipping during backpropagation.

Another important contribution in this paper is the usage of XNOR operation. Since both weights and activations are binarized, expensive multiplication operation can be completely avoided during forward propagation. Instead XNOR operation can be applied and set bits can be accumulated. In this work, related XNOR GPU kernel is implemented, benchmarked against unoptimized matrix multiplication and proven to be much faster.

3.3 DoReFa-Net

DoReFa-Net proposes methods to quantize weights, activations and gradients. Different quantizer functions are proposed for weights, considering the level of quantization which is basically the required number of bits to represent quantized values. For binary quantization, the proposed quantizer function is

$$w_q = sign(w) * E(|w|) \qquad (3\text{-}6)$$

They also proposed another quantizer function which is used for quantizing activations. That quantizer function is

$$y = \frac{1}{2^k - 1} * round((2^k - 1) * x) \qquad (3\text{-}7)$$

This quantizer function is named as $quantize_k$. It is basically a linear quantizer that outputs a quantized value between 0 and 1. This $quantize_k$ is also used for weight quantization when weights are quantized to 3 or more cluster values. In those cases, $quantize_k$ is not directly used. It is used as

$$w_q = 2quantize_k\left(\frac{\tanh(w)}{2\max(|\tanh(w)|)} + \frac{1}{2}\right) - 1 \qquad (3\text{-}8)$$

The purpose of using *tanh(.)* is to clip input values to the range of -1 and 1.

3.4 Ternary Weight Networks

TWNs cluster the weights into three values, one value is zero and the other two values are symmetrical with respect to zero. In order to quantize full precision weights, an optimization problem is defined as minimization of Euclidian distance between full precision weights and the scaled ternary weights as below.

$$a^*, {w_q}^* = \underset{a, W^t}{\arg\min} J(a, w_q) = \left|\left|w - a w_q\right|\right|^2 \quad s.t. \quad a \geq 0, w_q \,\epsilon\, \{-1, 0, 1\} \tag{3-9}$$

This optimization problem could be solved by taking derivatives of the cost function with respect to a and w_q. However, one derivative would be dependent to the other parameter. This means we cannot find one deterministic solution for this problem using this way. Therefore, an approximated solution is proposed. A function is defined, as below, to find ternary values of weights using the corresponding full precision ones.

$$w_q = \begin{cases} +1 & , if\ w > \Delta \\ 0 & , if\ |w| \leq \Delta \\ -1 & , if\ w < -\Delta \end{cases} \tag{3-10}$$

Using this quantization equation and the expanded form of the cost function (3-11), We can come up with a new problem formulation as (3-12).

$$J(a, w_q) = a^2 {w_q}^T w_q - 2a{w_q}^T w + w^T w \tag{3-11}$$

$$a^*, \Delta^* = \underset{a \geq 0, \Delta \geq 0}{\arg\min} \left(|I_\Delta| a^2 - 2 \left(\sum_{i \epsilon I_\Delta} |w_i| \right) a + c \right) \tag{3-12}$$

In (3-12), $|I_\Delta|$ is the number of weights bigger than Δ and smaller than $-\Delta$. The term c is constant. Note that each term in (3-11) and (3-12) are equal in the same order, when we use (3-10) in (3-11). From this point, given a particular Δ, we can find optimal a_{Δ}^{*} by taking derivative of (3-12) with respect to a.

$$a_{\Delta}^{*} = \frac{1}{|\boldsymbol{I}_{\Delta}|}\left(\sum_{\boldsymbol{i}\epsilon\boldsymbol{I}_{\Delta}} |\boldsymbol{w}_{\boldsymbol{i}}|\right) \tag{3-13}$$

Then we plug a_{Δ}^{*} in (3-12) to find Δ^{*} which result in (3-14). We can solve (3-14) by making assumptions about the form of the distribution of w_i values (full precision weights). The assumption of full precision weights to have normal distribution is actually well made since the weights are tend to have normal distribution after some training. Then, Δ^{*} can be found by sweeping it between 0 and maximum absolute value of w_i values. Considering normal and uniform distributions Δ^{*} can be approximated as $0.7 * E(|w|)$.

$$\Delta^{*} = \ arg \max_{\Delta>0} \frac{1}{|\boldsymbol{I}_{\Delta}|}\left(\sum_{\boldsymbol{i}\epsilon\boldsymbol{I}_{\Delta}} |\boldsymbol{w}_{\boldsymbol{i}}|\right)^{2} \tag{3-14}$$

CHAPTER 4

EXPERIMENTAL RESULTS

In this chapter, previously analyzed methods are implemented, benchmarked and compared with each other.

4.1 Framework and Libraries

There many frameworks and libraries that are developed for deep learning and neural networks. Throughout the short near history of deep learning, different frameworks became prominent at their time and left their spot to another one. In the earlier days of modern deep learning, the frameworks developed by universities were in use. Almost all of these frameworks lost their popularity and discontinued or merged into another framework. Even though there are still many active frameworks, two of them are dominant in the research and industry environments. They became the most prominent frameworks since they are developed and backed by large companies. These frameworks are PyTorch [26] and TensorFlow [27].

PyTorch was released in 2016. Even though it was released one year after TensorFlow, it gained popularity very quickly. There are couple of reasons behind this rapid popularity increase. Firstly, it was easy to use since it did not require much on top of Python programming language, unlike TensorFlow. Secondly, it was using eager execution by default. Therefore, it was easier to write, run and debug a PyTorch code. It is developed and maintained by Facebook.

TensorFlow was released in 2015. It became popular just after its release. In its earlier versions (1.x), it was working with graphs. First, the neural network architecture was defined as a graph. Layers were connected to each other in this graph. Then, a session was started to feed data to the input of the graph. The graph was processing the data layer by layer. Then the output was taken from the output node of the graph. This workflow was inconvenient process compared to PyTorch. TensorFlow started to lose popularity after PyTorch Release. Then they released TensorFlow 2.0 where eager execution is introduced. TensorFlow also uses Keras [8] as its high level API.

In this work, TensorFlow/Keras is used. Since quantization requires additional calculations in the layer, custom layers are coded. These custom layers are inherited from base Keras layers. The calculations of quantization on weights are done with TensorFlow library functions.

4.2 Datasets

In this section, the datasets which are used in the experiments are explained. Even though experiments are done to investigate the effects of quantization to deep neural networks, there is still a task of the neural networks. This task is chosen to be image classification. Therefore, the used datasets are image datasets with labels.

The first dataset is MNIST [28]. It is a dataset of images of hand-written digits. The digits from 0 to 9 are present in the dataset. The images are grayscale which means they only have one channel. The spatial sizes of the images are 28 by 28 pixels. There are typically 60,000 images for training and 10,000 images for testing purposes.

The second dataset is Cifar-10 [29]. It is a dataset of real-life color images of 10 classes. The classes are airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Since the images are in color, they have 3 channels. The spatial sizes of the images are 32 by 32 pixels. There are typically 60,000 images for training and 10,000 images for testing purposes. The images of classes are distributed evenly in the given training and testing images.

4.3 Results

The methods, that fall into quantization aware training category, are implemented and compared in this section. To ensure fair comparison, same architecture and same hyperparameters are used. For meaningful comparison, the methods that quantize only the weights are compared with each other and the methods that quantize both the weights and activations are compared with each other.

In this section, the selected methods are benchmarked on a simple dataset, MNIST, with a simple architecture, LeNet-5 [30]. Then the methods are benchmarked on Cifar-10 dataset with VGG-7. VGG-7 is a VGG-like architecture defined in [17]. Quantized VGG-7 architectures are also compared with a ground-up efficient model. Then, the methods are benchmarked on Cifar-10 dataset with ResNet20 and ResNet32 [31] to examine the success of quantized models on deep and complex architectures. Failures of weight quantization of deep and complex models are shown. Various solutions are proposed to overcome these failures. Proposed solutions are compared with each other. Relatively low accuracies of weight and activation quantization of deep and complex models are shown. The reasons behind these low accuracies are examined and shown. Quantized ResNet architectures are also compared with a ground-up efficient model. Lastly, a ground-up efficient model is quantized and compared with its base model.

In the following sections, some abbreviations are used for the methods. BC denotes BinaryConnect [13], BNN denotes Binarized Neural Networks [14], DRF denotes DoReFa-Nets [16] , TWN denotes Ternary Weight Networks [17]. Moreover, a custom approach is developed using the DoReFa-Nets' method. In this custom approach, weights are quantized using DRF method and activations are not quantized. This custom approach is named as DRF-W which denotes DoReFa-Nets-Weights. Also, "Base" word is used to denote full precision models. Base models are not quantized and used for comparison purposes.

4.3.1 A Simple Architecture and Dataset

In this section, a simple architecture and a simple dataset is used to compare quantization methods. The used architecture is LeNet-5 [30] and the used dataset it MNIST. The models are trained for 30 epochs with batch size of 128. Adam is used for optimizing the categorical cross entropy loss. The learning rate is started with 0.001 and exponentially decayed down to 0.0001.

Table 4.1 Comparison of Quantization Methods on Simple Example

Models	*Accuracy (%)*	*Compression*
Base	99.54	1x
BC	99.50	32x
BNN	99.00	32x
DRF-W	99.43	32x
DRF	99.18	32x
TWN	99.51	16x

The weight quantizer methods do not sacrifice accuracy while compressing the neural network significantly. However, the weight and activation quantizer methods experience an accuracy drop considering the task's simplicity. These results suggest that the weight and activation quantizer methods might see significant accuracy drops in complex neural network architectures and datasets.

4.3.2 VGG-7 Architecture

The used architecture is VGG-7 defined in [17]. The hyperparameters are total trained epochs of 200 and batch size of 100. SGD with momentum of 0.9 is used as the optimizer. Learning rate is started at 0.1 and multiplied with 0.1 at epoch 80, 120, 160.

4.3.2.1 Methods Quantize Weights Only

In this section, BinaryConnect, Ternary Weight Networks and DoReFa-Net without activation quantization (named as DoReFa-W) methods are compared. Base model (full precision) is also added to see accuracy degradation. The used dataset is Cifar-10.

The improvement of validation set accuracies can be seen in Figure 4.1. It can be roughly seen that TWNs have the least degradation from accuracy among all. Even though a general idea can be extracted from this figure, more explanatory figures are still required.

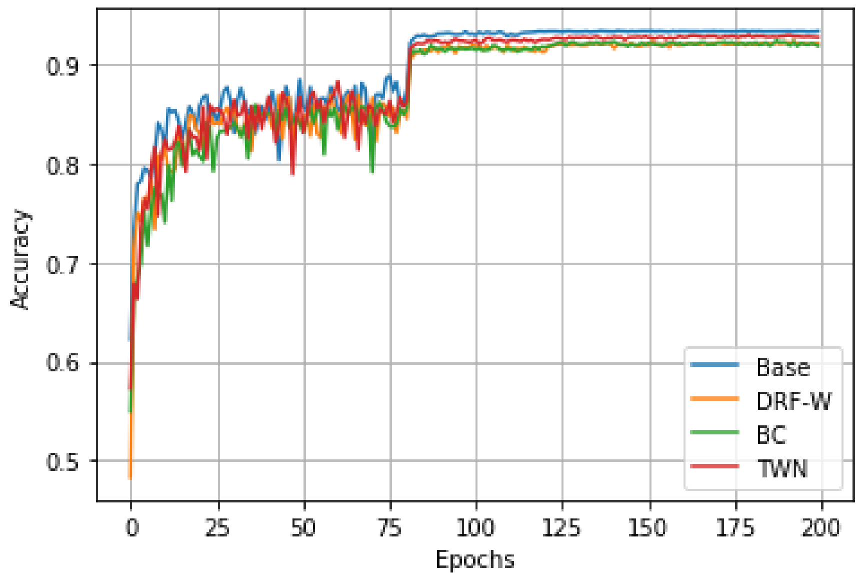


Figure 4.1. Validation set accuracies of Base, DRF-W, BC and TWN

The best test accuracy comparison is made in Table 4.2 If TWNs are used, the cost of 16 times smaller model results in only 0.39% in accuracy. One important observation in this comparison is that BinaryConnect and DoReFa-W have almost the same accuracy. The only difference between the quantizer functions of BinaryConnect (3-1) and DoReFa-W (3-6) is a scaling factor which is the average value of the absolute values of the weight in the related layer. From the results of this experiment, it can be said that scaling factor does not have a considerable effect on accuracy when weights are quantized to two values.

Table 4.2 Test accuracy comparison of weight quantizer methods

Methods	*Test Accuracy (%)*	*Compression Rate*
Base	93.34	1x
BC	92.32	32x
DRF-W	92.29	32x
TWN	92.95	16x

Another important concept to consider is the convergence characteristics of the experimented methods. Even though Figure 4.1 gives some idea about this topic, the plot cannot be read clearly due to the oscillations in early epochs of the training. In order to overcome this problem, another plotting method is applied to the same data. Accuracy is averaged at each epoch from the start. For example, the first 3 accuracy values are averaged at epoch three. Plotting this processed accuracy data removes the oscillations and shows a better image in terms of convergence characteristics as in Figure 4.2.

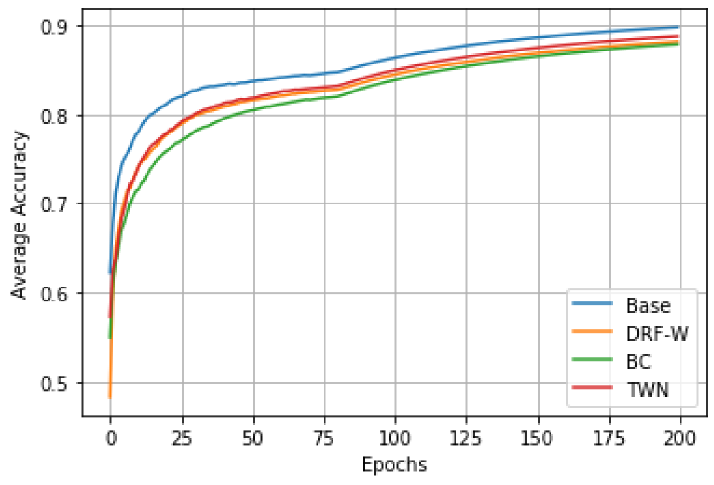


Figure 4.2. Improvement of average accuracies of weight quantizers and base model

Plotting average accuracy actually gives a great insight about convergence. It can be clearly seen that, as expected, base model has the best convergence characteristics since its average accuracy increases more rapidly than the quantized models. Interestingly, TWN and DRF-W shows very similar, almost identical, convergence characteristics. They start to differ after 50th epoch, which is obviously the result of the model's capacity since while TWN converges to 92.95%, DRF-W converges to 92.29%. Even though a small gap is formed between TWN and DRF-W after 50th epoch, average accuracy lines remain parallel. Even more interestingly, convergence characteristics of BC and DRF-W differ a lot. Both of the methods converge to a similar accuracy, at around 92.30%. However, Convergence of DRF-W is considerably faster than BC. Therefore, it can be said that a scaling factor leads to a faster convergence in binary quantizations.

4.3.2.2 Activation Quantization Along with Weight Quantization

In this section, the effects of activation quantization on models with quantized weights are examined. Two activation and weight quantizer methods are compared with each other and their weight-only quantizing versions. The methods are Binarized Neural Network and DoReFa-Net while their weight-only quantizing versions are BinaryConnect and DoReFa-W, respectively. DoReFa-Net proposes different levels of quantization for activations. For fair comparison, DoReFa-Net with binary activation quantization is chosen to be compared with Binarized Neural Network.

Table 4.3 The comparison of methods quantizing weights and activations

Method	*Accuracy (%)*	*Compression Rate*
BNN	88.24	32x
DRF	90.15	32x

The test accuracies of the trained models of BNN and DRF is given in Table 4.3 DoReFa-Net has a better accuracy than BNN. Even though quantization levels of weights and activations are same for these methods, the quantizer functions are slightly different. The difference in weight quantizers is simply a scaling factor as in 3-6. The activation quantizing levels are binary in this case. BNN quantizes the activations to the values of -1 and 1, while DoReFa-Net quantizes them to the values 0 and 1. This difference in the activation quantizer functions creates this results since the only-weight quantizing versions, BC and DRF-W, have very similar test accuracies (Figure 4.2).

Previously explained averaging accuracy method is applied to this case as well to examine convergence characteristics. Figure 4.3 clearly shows that DRF learns faster than BNN.

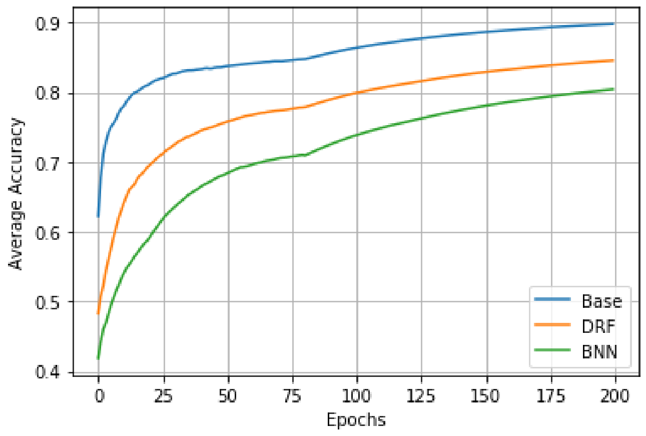


Figure 4.3. Improvement of average accuracies of BNN and DRF with base model

Similarly, the same method is applied to compare the weight-only quantized models and fully quantized models in order to understand the effect of activation quantization to convergence. The results are in the plots in Figure 4.4. The results clearly show that quantizing activations make the models learn even slower than weight-only quantized models.

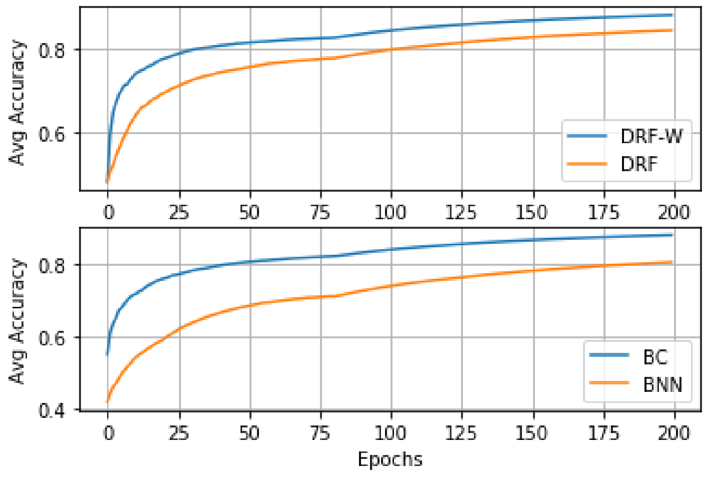


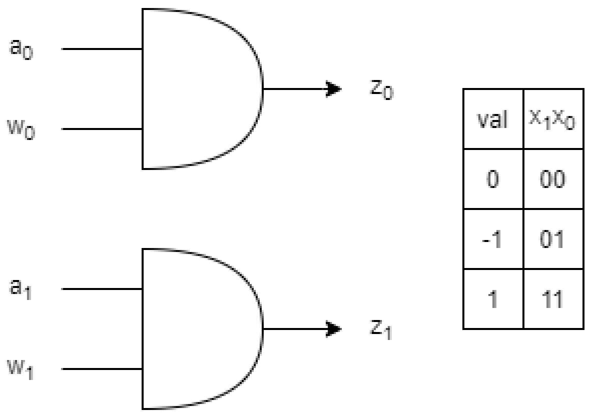
Figure 4.4. Average accuracy comparison of weight quantization and weight/activation quantization

Since both of these methods are similar in terms of quantization level, their weight and activation quantizer functions are interchangeable. For example, a quantized model can be trained with BNN's weight quantizer and DRF's activation quantizer. Two hybrid models are created to investigate if there is a superior combination to natural proposed methods. The results are given in Table 4.4

Table 4.4 Combinations of BNN and DRF

Methods	*BNN-Act*	*DRF-Act*
BNN-Weight	88.24	89.68
DRF-Weight	88.52	90.15

Activation quantizer of the BNN causes the largest degradation. It is clear to see that, BNN-Weight/DRF-Activation model performs better than original BNN model. BNNs have also an acceleration technique which employs XNOR function instead of multiplying since whole weights and activations are constrained to -1 and 1. This hybrid model actually does not lose this acceleration technique and proposes even better one. The hybrid model's weights are constrained to -1 and 1, activations are constrained to 0 and 1. Therefore, these hybrid models can be implemented with only two AND gates as in Figure 4.5. As a result, these hybrid models have a better accuracy and a smaller number of gates than BNNs. Furthermore, since there are only two parallel AND gates instead of one XNOR gate, they are also faster than BNNs. On the other hand, since the number of cluster points increases to 3, the required number of bits to store each value becomes 2 and compression rate decreases to 16x.



val	x_1x_0
0	00
-1	01
1	11

Figure 4.5. Example implementation for hybrid models

4.3.2.3 Comparison with Ground-up Efficient Models

In this section, benchmarking and comparison of efficiently designed models and quantized models are made. In the SqueezeNet paper [1], multiple architectures are proposed. The proposed architectures include a vanilla one, the one with simple residual connection and the one with complex residual connection. For this section, we used vanilla SqueezeNet architecture and VGG-7 with quantized weights.

The results are given in Table 4.5 VGG-7 is a basic architecture but heavy on the number of parameters side. Therefore, it requires 49.5MB for its weights even though it is not a very deep architecture. SqueezeNet is a relatively deeper architecture with its Fire Modules which are a bit more complex than the basic layers of VGG. SqueezeNet requires only 2.78MB to store its weights. SqueezeNet's accuracy is considerably lower than the base VGG-7. However, it is completely acceptable due its very efficient size. This weakness of VGG-7 can be eliminated with the quantization methods. TWN compresses the VGG-7 down to 3.09MB which is very close to SqueezeNet's size. Furthermore, TWN's model's performance is still 1.12% higher than the SqueezeNet's performance. DRF-W and BC compress the VGG-7 even more, almost down to the half size of SqueezeNet, while achieving still better performance than SqueezeNet.

Table 4.5 Comparison of SqueezeNet and Quantized Models

Models	*Accuracy (%)*	*Number of Parameters*	*Required Space to Store the Weights (MB)*
SqueezeNet	91.83	727,626	2.78
VGG-7 (Base)	93.34	12,976,266	49.5
VGG-7 (BC)	92.32	12,976,266	1.55
VGG-7 (DRF-W)	92.29	12,976,266	1.55
VGG-7 (TWN)	92.95	12,976,266	3.09

While these results support that quantization achieves smaller and better models than SqueezeNet, there are still overheads of deploying quantized models. Quantized models require special hardware to be used. On the other hand, SqueezeNet is ready after training and can run on all conventional hardware without an extra effort.

4.3.3 Quantization of Deep and Complex Models

In [20], it is claimed that aggressive quantization approaches, such as 1 and 2 bits, are experimented with overparameterized models such as VGG [32] or AlexNet [6]. According to this claim, since these models are designed to perform marginal results, they are not efficient and they already have much more parameters than required by their related tasks. Therefore, compressing these models by quantization is easy without a significant accuracy loss. However, it is raised that the real challenge remains to be quantizing efficiently designed models.

In this section two ResNet [31] models are quantized. ResNet architecture is chosen to address the previously mentioned problem. ResNets are deeper and more efficient models than VGG architectures. While ResNet-20 and ResNet-32 have around 273K and 468K parameters respectively, VGG-7 has around 12.982M parameters. Therefore, it can be said that ResNet architecture is a good choice for examining the success of quantization of models which are not overparameterized.

The first and last layers of the models are not quantized. This is a very common practice since quantizing first and last layers of very deep models significantly decreases the accuracy. Therefore, the compression rates of the quantized models are given approximately.

The hyperparameters are total trained epochs of 200 and batch size of 100. SGD with momentum of 0.9 is used as the optimizer. Learning rate is started at 0.1 and multiplied with 0.1 at epoch 80, 120, 160.

4.3.3.1 Methods Quantize Weights Only

Quantizing ResNet20 with the methods of TWN and DRF-W causes a small degradation in accuracy. The drop in accuracy becomes 0.28% and 1.9% with TWN and DRF-W respectively. On the other hand, the accuracy loss in BC becomes 8.98% which is certainly in the unacceptable range for this task, dataset and architecture. Some methods and/or modifications to make BC work with deep and complex models are discussed in the following section.

Table 4.6 Test Accuracy Comparison of Weight Quantizer Methods on ResNet20

ResNet20	*Accuracy (%)*	*Compression*
Base	91.66	1x
BC	82.68	~32x
DRF-W	89.76	~32x
TWN	91.38	~16x

The results for ResNet32 are similar to the ones for ResNet20 with the exception of BC method. Being deeper and having more parameters, full precision ResNet32 has already better accuracy than ResNet20. This superiority is applicable for TWN and DRF-W as well. In ResNet32, the accuracy degradation from TWN and DRF-W are 0.81% and 1.78%. An interesting outcome of this experiment is that BC completely fails to learn with same hyperparameters and training techniques. The next section further examines this problem and suggests solutions.

Table 4.7 Test Accuracy Comparison of Weight Quantizer Methods on ResNet32

ResNet32	*Accuracy (%)*	*Compression*
Base	92.05	1x
BC	10.00(Fail)	~32x
DRF-W	90.27	~32x
TWN	91.24	~16x

4.3.3.2 BinaryConnect Modifications for Deep and Complex Models

Before proposing solutions to this problem, the roots of the problem should be identified. The problem is related with architecture. However, the problem is not about the architecture’s deepness or complexity. It is about some of the residual connections. In the original ResNet architecture [31], the spatial size is halved and number of channels is doubled at the start of every stack of ResNet layers. At these specific points, simple identity connections cannot be used since the dimension do not match. The original paper suggests two solutions for this problem. The first one is to use zero padding to match dimension. The second one is to use 1x1 convolutional layers to match the dimensions. Our implementation uses the second approach. These 1x1 convolutional layers are the causes of the problem. Following figure shows the weights distributions of some layers to point out the roots of the problem.

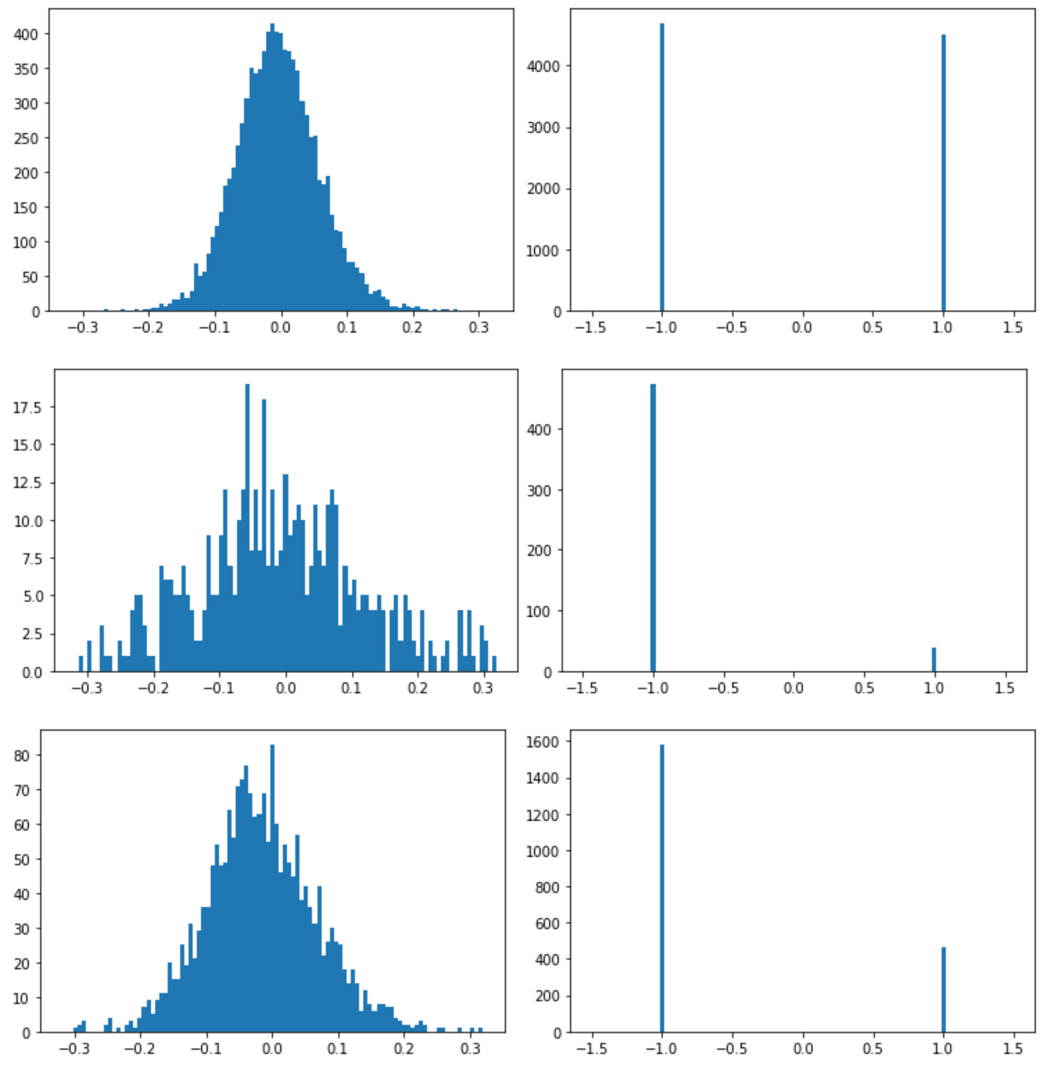


Figure 4.6. The weight distributions of 3 layers from full precision and quantized models

In Figure 4.6, the first column is from full precision model and the second column is from BC model. The first row is from an arbitrary layer at the middle of the model. The second and third rows are from residual connections. The weights from residual connections are quantized quite asymmetrically. This situation causes a very large disturbance at the end point of residual connection. Therefore, this disturbs the meaningful flow of the information.

One possible solution would be not quantizing these weighted residual connections just like the first and the last layer of deep neural networks. In Table 4.8 The results of this approach are shown.

Table 4.8 BC with not quantizing weighted residual connections

Model	*Accuracy (%)*
ResNet20-BC	89.60
ResNet32-BC	90.65

Another possible solution would be proposed by slightly modifying the architecture. The modification is completely eliminating these problematic residual connections. The residual connections which are identity functions are still used. However, the residual connections with 1x1 filters are simply deleted. The results of this approach are shown in Table 4.9 .

Table 4.9 BC without weighted residual connections

Model	*Accuracy (%)*
Modified ResNet20-BC	89.03
Modified ResNet32-BC	91.11

There can be other solutions by modifying some elements. As DRF-W method successfully quantizes deep and complex models with small degradation to accuracy, the solution of the BC method’s problem can be found in the comparison of DRF-W and BC methods. For this comparison, DRF-W is specially suggested since both DRF-W and BC are binary quantizers. Moreover, they both use the same quantization function (*sign(.)*), while DRF-W is the scaled version of the BC. The scaling factor of DRF-W is a floating-point value which is much less than 1, cluster point of the BC method. Considering the success of DRF-W, two modifications can be suggested. Both of the modifications aim to scale the outputs of layers to a significantly smaller value.

First modification is adding a scaling factor to BC just like DRF-W. This method also would be an alternative to DRF-W. Adding a scale factor discards the most important advantage of BC, which is completely eliminating the expensive multiplication operations. However, unlike DRF-W, if we can decide the exact value of the scaling factor, we can choose it to be a hardware friendly value. For example, choosing a value which is a power of 2 would enable bit shifting instead of multiplication. Note that this is a similar approach to [19] where weights are clustered to be power of two and different bit widths are experimented down to ternary quantization. They left out the binary quantization out of scope. By this way, the expensive multiplications can be avoided while having a floating-point scaling factor.

The second modification changes the original architecture even less while achieving success. Since the aim is to scale the layers' outputs to a smaller value, we can also modify the activation function. Previously, we were using ReLU function for activations. If we decrease the slope of the ReLU, we can successfully train BC with weights constrained to -1 and 1. The computational complexity of the activation function will increase since there will be multiplications. However, as in first method, we can choose the slope of ReLU so that we can benefit from bit shifting instead of expensive multiplications.

Table 4.10 Results of BC Solutions for Deep and Complex Models

Method	*ResNet20(%)*	*ResNet32(%)*
BC	82.68	10.00
BC-1.1	89.33	90.67
BC-1.2	89.71	90.90
BC-2.1	88.03	87.69
BC-2.2	88.31	87.83

The results of the proposed modifications can be seen in Table 4.10 In BC-1.1 and BC-1.2, the weights are scaled with 0.1 and 0.125, respectively. In BC-2.1 and BC-2.2, the ReLU activations are scaled with 0.1 and 0.125, respectively.

Note that DRF-W accuracy is 90.27% for ResNet32 in Table 4.7 . The first suggested modification outperforms the DRF-W. The constant scaling factor for all layers is chosen 0.1 and 0.125 in BC-1.1 and BC-1.2, respectively. These models show better performance than DRF-W with a margin of 0.40% and 0.63%. These results clearly suggest that the mean of absolute values of full precision weights is not an optimal scaling factor for binary quantization. BC-1.2 has also hardware friendly scaling factor on top of the better performance compared to DRF-W.

4.3.3.3 Activation and Weight Quantization

The experiments show that the weight quantization techniques can work with deep, complex and fairly parameterized models like ResNet20 and ResNet32 as well as over parameterized models like VGG-7. However, the same success cannot be achieved when both of the weights and activations are quantized.

Table 4.11 Weight and Activation Quantization of Deep and Complex Models

	VGG-7	*ResNet20*	*ResNet32*
Base	93.34	91.66	92.05
BNN	88.24	77.88	68.23
DRF	90.15	78.43	69.99

A certain pattern can be seen in Table 4.11 Accuracy degradation is not related with number of parameters. The accuracy degradation is related with the deepness of the model. More specifically, the accuracy degradation increases with the increasing number of used activation layers in the model. The only weight quantization does not harm the accuracy this much since the weights are trained such that they can work with quantization. In only weight quantization, the errors are calculated with the quantized weights' outputs and the backpropagation algorithm updates the weights accordingly. However, the activations do not have such adaptation mechanism. They do not have any parameters to be trained. Moreover, since straight through estimators are used for activations during backpropagation, the error introduced by quantization of activations cannot be compensated with backpropagation updates. In other words, the activation quantization is invisible to backpropagation algorithm. Therefore, increasing the quantized activation layers in a model accumulates the introduced error and therefore significantly harms the accuracy. The following section presents empirical proof of accuracy degradation caused by activation quantization.

4.3.3.4 Activation Quantization

Some experiments are conducted to prove that quantizing activations is the real reason of severe accuracy drop in deep neural networks. In these experiments, weights are not quantized and trained with full precision. However, activations are quantized. A deep neural network with full precision weights and quantized activations has no advantage at all. The only aim is to show that quantizing activations are the real root of accuracy drop in quantized deep neural networks. The results are shown in Table 4.12.

Table 4.12 Accuracies when only activations are quantized

Method	*VGG-7*	*ResNet20*	*ResNet32*
BNN-Act	90.28	81.35	78.00
DRF-Act	91.85	82.12	77.35

In Table 4.12, BNN-Act and DRF-Act are the activation quantization techniques which are proposed in [14] and [16], respectively. DRF-Act is expected to perform better than BNN-Act, considering the previous results. DRF-Act indeed shows a superior performance for VGG-7 and ResNet20. However, BNN-Act performs better than DRF-Act for ResNet32. The accuracy loss increases from the lowest points at VGG-7 to the highest point at ResNet32. During this increase, DRF-Act is affected more seriously than BNN-Act. As a result, BNN-Act becomes the better performer at the highest accuracy loss point which is ResNet32. The accuracy losses for the deep ResNet models are quite high. The accuracies of ResNet20 and ResNet32 drop at least 9.54% and 14.7% compared to base model. On the other hand, VGG-7 does not experience such dramatic accuracy loss when its activations are quantized using the same method. VGG-7 has only 7 activation layers that are quantized while ResNet20 and ResNet32 have 19 and 31 activation layers, respectively. The accuracy loss is proportional with number of quantized activation layers. These high accuracy drops are not observed when only the weights are quantized. However, when both of the weights and activations are quantized, the combined accuracy loss is even higher. This high combined accuracy loss is mostly due to the activation quantization as suggested with this section's experiments.

4.3.3.5 Comparison with Ground-Up Efficient Models

Previously, Quantized VGG-7 models are compared with SqueezeNet in terms of accuracy, number of parameters and required storage space. In this section, similar comparison is made for ResNet models. Since ResNet architecture includes residual connections, SqueezeNet with simple bypass is built and trained for fair comparison.

Table 4.13 Comparison of SqueezeNet and Quantized ResNet Models

Models	*Accuracy (%)*	*Number of Parameters*	*Required Space to Store the Weights (MB)*
SqueezeNet (with bypass)	92.43	727,626	2.78
ResNet20 Base	91.66	271,690	1.04
ResNet20 BC-1.2	89.71	271,690	0.032
ResNet20 DRF-W	89.05	271,690	0.032
ResNet20 TWN	90.71	271,690	0.065
ResNet32 Base	92.05	465,674	1.78
ResNet32 BC-1.2	90.90	465,674	0.055
ResNet32 DRF-W	90.27	465,674	0.055
ResNet32 TWN	91.24	465,674	0.110

The ResNet architectures are already efficient in terms of size. Therefore, compressing it even further results in required spaces in the range of KBs. For example, binary quantizations of ResNet20 require only 32KB of space which can fit into many embedded processors' L1 caches.

4.3.4 Quantizing Ground-up Efficient Models

Even though ResNet architectures are very efficient compared to VGG-like architectures, they are not designed to be efficient. They are designed to perform at marginal levels. On the other hand, SqueezeNet architecture's main aim is to be efficient. Therefore, quantizing SqueezeNet architecture would lead to further efficient architectures. In this section, quantization of SqueezeNet is examined.

Table 4.14 Quantization of SqueezeNet

Method	*Accuracy (%)*	*Required Space to Store the Weights (MB)*
SqueezeNet	91.83	2.78
SqueezeNet-BC	90.07	0.087
SqueezeNet-DRF-W	90.11	0.087
SqueezeNet-TWN	90.62	0.174
SqueezeNet (Simple Bypass)	92.43	2.78
SqueezeNet (Simple Bypass)-BC	90.47	0.087
SqueezeNet (Simple Bypass)-DRF-W	90.06	0.087
SqueezeNet (Simple Bypass)-TWN	91.24	0.174

Using residual connections, i.e. simple bypasses, is a common practice in deep neural networks. Using residual connections almost always results in better accuracies than simple feed-forwards versions. This is observed in SqueezeNet as well. In Table 4.14 full precision SqueezeNet with simple bypass has better accuracy than full precision SqueezeNet with a margin of 0.6%. This superiority is conserved in their TWN quantized version as well. TWN quantized versions have 0.62% accuracy difference which is a very similar value to the difference between full precision networks' accuracies. However, one interesting observation is that SqueezeNet with DRF-W quantization is better than its counterpart with simple bypass.

CHAPTER 5

SYSTEM IMPLEMENTATION AND BENCHMARKING

In this section, a convolutional layer is implemented for various systems in order to realize and benchmark quantization theory. The aim is not to code a fast convolution operation. Therefore, the implemented convlution operation is not computationally efficient. The operation has many loops and expensive operators such as modulo. Moreover, there is no effort to ease the compiler’s job for enabling any possible SIMD instructions. The aim is to code a memory efficient convolution operation and prove that quantized convolution kernels can work on real systems. Therefore, there is a observable memory efficiency (not only theoretical). Another aim is to give benchmarking results for different devices ranging from very low power embedded processors to powerful desktop computer processors.

5.1 Implementation Details

A base convolution operation and the quantized version of it are implemented. In both of the algorithms, spatial representation is flattened in inputs, filters and outputs. For example, the inputs and outputs have the common representation as *channel number x spatial width x spatial height*. The inputs and outputs are reshaped to *channel number x (spatial width * spatial height).* Similarly, the filters originally have *input channel number x output channel number x spatial width x spatial height.* The filters are also reshaped to *input channel number x output channel number x (spatial width * spatial height).* The base convolution algorithm is shown in Figure 5.1.

```
1: function CONVOLUTION($inputs, filters, outputs$)
2:     $ni \leftarrow findSpatialSize(inputs)$                ▷ spatial size of inputs
3:     $nf \leftarrow findSpatialSize(filters)$               ▷ spatial size of filters
4:     $no \leftarrow findSpatialSize(outputs)$               ▷ spatial size of outputs
5:     $ci \leftarrow findChannelSize(outputs)$               ▷ number of inputs channels
6:     $co \leftarrow findChannelSize(outputs)$               ▷ number of outputs channels
7:     for $c_i \leftarrow 0$ to $ci$ do
8:         for $c_o \leftarrow 0$ to $co$ do
9:             $startIndex \leftarrow -1$
10:            for $i \leftarrow 0$ to $no^2$ do
11:                $startIndex \leftarrow startIndex + 1$
12:                if $i$ mod $no == 0$ and $i! = 0$ then
13:                    $startIndex \leftarrow startIndex + nf - 1$
14:                $index \leftarrow startIndex$
15:                for $j \leftarrow 0$ to $nf^2$ do
16:                    if $j$ mod $nf == 0$ and $j! = 0$ then
17:                        $index \leftarrow index + no - 1$
18:                    $cellResult \leftarrow filters[c_i][c_o][j] * input[c_i][index]$
19:                    $outputs[c_o][i] \leftarrow outputs[c_o][i] + cellResult$
20:                    $index \leftarrow index + 1$
```

Figure 5.1. Base convolution algorithm

The quantized convolution algorithm is implemented by following the idea which is explained in 2.1.2.1. The method is summarized with Figure 2.6. The quantized convolution algorithm (Figure 5.2) is very similar to the base algorithm. The difference is basically that quantized algorithm requires an indirect memory access in order to load the actual filter value. They only differ at the calculation step at lines 20-23 in quantized convolution algorithm on top of the two additional parameters that quantized algorithm requires. These parameters and lines are explained in detail below.

Parameter q: This value represents the number of indexes stored in a byte. If the binary quantization is used, the required bitwidth is only 1 and therefore a byte can store 8 filter indexes. If the ternary quantization is used, 4 filter indexes can be stored in a byte.

Parameter qMask: As the name suggests, this parameter is used for masking. If the binary quantization is used, only least significant bit is required and therefore this parameter is set to 0x01. If ternary quantization is used, least significant two bits are required and this parameter is set to 0x03.

Line 20-21: Since the smallest memory read operation can read 1 byte, the filter value indexes are saved in an unsigned char array. This requires couple of additional operation to extract the index of a particular filter element. These operations are dividing, shifting and masking. The iterator j is divided by *parameter q* to find the byte that stores the current filter index. Then this value is shifted by bitwidth times j mod 8 (calculated at line 20 and stored in sft) to find the position of the current filter index in the byte. Lastly, the resultant value is masked to get rid of unrelated values and extract the exact filter index.

```
 1: function Q CONVOLUTION($inputs, fIndexes, fValues, outputs$)
 2:     $ni \leftarrow findSpatialSize(inputs)$                 ▷ spatial size of inputs
 3:     $nf \leftarrow findSpatialSize(filterIndexes)$          ▷ spatial size of filters
 4:     $no \leftarrow findSpatialSize(outputs)$                ▷ spatial size of outputs
 5:     $ci \leftarrow findChannelSize(outputs)$                ▷ number of inputs channels
 6:     $co \leftarrow findChannelSize(outputs)$                ▷ number of outputs channels
 7:     $q \leftarrow findQuantType(outputs)$                   ▷ memory width (8) / bitwidth
 8:     $qMask \leftarrow findQuantMask(outputs)$               ▷ 0x01: binary, 0x03: ternary
 9:     for $c_i \leftarrow 0$ to $ci$ do
10:         for $c_o \leftarrow 0$ to $co$ do
11:             $startIndex \leftarrow -1$
12:             for $i \leftarrow 0$ to $no^2$ do
13:                 $startIndex \leftarrow startIndex + 1$
14:                 if $i$ mod $no == 0$ and $i! = 0$ then
15:                     $startIndex \leftarrow startIndex + nf - 1$
16:                 $index \leftarrow startIndex$
17:                 for $j \leftarrow 0$ to $nf^2$ do
18:                     if $j$ mod $nf == 0$ and $j! = 0$ then
19:                         $index \leftarrow index + no - 1$
20:                     $sft \leftarrow bitwidth * (j \bmod 8))$
21:                     $fIndex \leftarrow (fIndexes[c_i][c_o][j/q] >> (sft)$ AND $qMask$
22:                     $cellResult \leftarrow fValues[c_i][c_o][fIndex] * input[c_i][index]$
23:                     $outputs[c_o][i] \leftarrow outputs[c_o][i] + cellResult$
24:                     $index \leftarrow index + 1$
```

Figure 5.2. Quantized Convolution Algorithm

Line 22: The found filter index is used to extract the full precision filter value from the related short float array. Then this filter value is multiplied with its corresponding input value.

Line 23: The result of the multiplication is accumulated in the related output field.

5.2 Benchmarking

Three different systems are used to benchmark the quantized and normal convolution operations. The first system represents the low power and cheap embedded devices. Tiva C TM4C123G development board from Texas Instrument is used. This board works on ARM Cortex M4 processor. The second system represents a more powerful segment of embedded devices. Raspberry Pi 3 B+ is used. This embedded computer works on ARM A53 and Linux based Raspbian operating system. The last system is a desktop computer and used for comparison purposes. It works on Intel Core i5-7300HQ and Windows 10.

Three different settings are experimented for all of the systems. These settings are experimented with base (full-precision) and quantized filters. The settings are as following. The input planes consist of 3 channels and 16 by 16 size. No padding is used to protect the width and height of the input. Output planes consist of 8 channels. Size of the output planes become 14x14, 13x13, 12x12 for filters 3x3, 4x4, 5x5, respectively. Memory reduction is calculated using (2-17). The increase in computation time for quantized filters is also reported.

5.2.1 ARM Cortex M4

Tiva C TM4C123G is a low-end development board which is capable of doing many simple things. It has 80MHz 32-bit ARM Cortex M4 processor. It has 256KB Flash, 32KB SRAM and 2KB EEPROM on-chip memory. The following experiments use 32KB SRAM memory for input-output planes and filters. The flash memory is not suitable for this task since it is slower than SRAM and it has relatively low number of life-cycle compared to SRAM. The TI v20.2.0.LTS compiler is used. The results are given in Table 5.1 Table 5.1 Table 5.1 Table 5.1 Table 5.1

Table 5.1 Binary quantization timing results for ARM M4

Filter Size	*Base (ms)*	*Quantized-Binary (ms)*	*Memory reduction*	*Computation Time Increase*
3x3	179.054	232.56	27.87	1.30
4x4	263.888	352.611	29.54	1.34
5x5	345.365	467.619	30.38	1.35

Table 5.2 Ternary quantization timing results for ARM M4

Filter Size	*Base (ms)*	*Quantized-Ternary (ms)*	*Memory reduction*	*Computation Time Increase*
3x3	179.054	237.264	12.34	1.32
4x4	263.888	356.667	13.71	1.35
5x5	345.365	471.075	14.46	1.36

5.2.2 Quad-Core ARM A53 – Raspberry Pi 3 B+

The raspberry pi is a small computer that succeeded to be very widely used worldwide. Its 3rd version model B+ is used in this section's experiments. Raspberry pi 3 B+ employs 64-bit quad-core ARM A53 which is working with 1.4GHz. It can run many operating systems. In our case, Raspbian OS is used. Raspberry pi 3 B+ has 1GB LPDDR2 SDRAM. The gcc v8.1.0 compiler is used. The results are given in Table 5.3

Table 5.3 Binary quantization timing results for Raspberry Pi 3 B+

Filter Size	*Base (ms)*	*Quantized-Binary (ms)*	*Memory reduction*	*Computation Time Increase*
3x3	5.01	6.65	27.87	1.33
4x4	7.54	9.95	29.54	1.32
5x5	9.47	13.41	30.38	1.42

Table 5.4 Ternary quantization timing results for Raspberry Pi 3 B+

Filter Size	*Base (ms)*	*Quantized-Ternary (ms)*	*Memory reduction*	*Computation Time Increase*
3x3	5.01	6.89	12.34	1.38
4x4	7.54	9.99	13.71	1.33
5x5	9.47	13.22	14.46	1.40

Since this system runs on an operating system, Raspbian OS, the given timing values are found by averaging 100 consecutive runs of the algorithm.

5.2.3 Intel Core i5-7300HQ – Desktop Computer

This system is experimented and benchmarked for comparison purposes only. A desktop computer typically does not need quantization since there is usually enough computing power and memory. This system runs on Intel Core i5-7300HQ at 2.50 GHz and Windows 10. It has 8GB RAM. The gcc v8.1.0 compiler is used. The results are given in Table 5.5

Table 5.5 Binary quantization timing results for Desktop Computer

Filter Size	*Base (ms)*	*Quantized-Binary (ms)*	*Memory reduction*	*Computation Time Increase*
3x3	0.416	0.676	27.87	1.62
4x4	0.552	0.922	29.54	1.67
5x5	0.711	1.238	30.38	1.74

Table 5.6 Ternary quantization timing results for Desktop Computer

Filter Size	*Base (ms)*	*Quantized-Ternary (ms)*	*Memory reduction*	*Computation Time Increase*
3x3	0.416	0.640	12.34	1.54
4x4	0.552	0.978	13.71	1.77
5x5	0.711	1.216	14.46	1.71

Since this system runs on an operating system, Windows 10 OS, the given timing values are found by averaging 100 consecutive runs of the algorithm.

5.3 Comments

As explained previously, the ideal compression rate is 32x. The case gets to ideal as the number of connections (or filters) increases. Therefore, the memory reduction increases while the filter size increases. While the memory required by filters decreases 29x on average, the computation time is also increased drastically. This implementation and setting can be acceptable where a system has critically low memory.

Quantization can make possible to run neural network inference on these systems. ARM Cortex M4 system has 32KB of SRAM memory which is not enough to store most of deep learning models. However, this simple implementation allows low-end embedded systems to store and run small deep learning models.

This implementation of quantized convolution operation does not aim to be efficient and fast. However, a faster implementation would still suffer from some significant degree of increased computation time due to indirect memory access in the quantized convolution operation. Therefore, quantized neural networks are best and fully utilized with custom hardware designs. However, faster computation times can be still achieved with the optimization of the compilers. The effect of the compiler optimization on computation times for 3x3 filter case can be seen in Table 5.7

Table 5.7 The effect of compiler optimization on computation time

System	*Base*	*Base (-O3)*	*Quantized*	*Quantized (-O3)*	*Base reduction*	*Quantized Reduction*
Arm Cortex M4	179.054	74.82	232.56	140.54	2.39	1.65
Raspberry Pi 3 B+	5.01	2.06	6.65	4.09	2.43	1.63
Desktop Computer	0.416	0.180	0.676	0.341	2.31	1.98

The compiler optimization can speed up the base convolution algorithm by around 2.40 times. However, the same optimization settings cannot show the same success for quantized convolution algorithm. The compiler optimization can speed up the quantized convolution algorithm by only a factor between 1.63 and 1.98. As explained in Figure 5.2, there are extra shifts, divisions, bitwise operations, and memory accesses in the quantized convolution algorithms. These operations limit the capabilities of the compiler. As a result, the compiler has a reduced effect on the quantized convolution compared the base algorithm.

No manual optimization is done for the quantized convolution operation. Since the quantization allows us to work with low precision numbers with only 1 or 2 bits, we can highly benefit from bitwise operations. Moreover, since we can run bitwise operations on 32 bits register, we can actually parallelize 32 operations with binary quantized weights. This parallelization can be implemented efficiently for each computer architecture. ARM NEON instructions can be used for many ARM CPUs. Intel CPUs can also implement largely parallel bitwise operations using their AVX instructions. Even though sometimes manual effort may be required to implement these operations, specialized compilers of these architectures can detect and implement bitwise operations.

The implementations in this section are single thread applications. However, the quantized neural networks can benefit from multi-threading as well. Multi-core CPUs are common even in cheap embedded systems. Implementing bitwise operations with multiple threads can increase the parallelization and therefore the computing speed.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

In this thesis, we have investigated the effects of quantization on deep neural networks.

First, the selected methods are implemented on LeNet-5 architecture with MNIST dataset. Since MNIST dataset is way too simple, it is expected to achieve success with all of the methods. However, there was a noticeable accuracy loss in weights and activation quantization methods compared to only weights quantization methods. In the following experiments, weights and activation quantization methods are compared with each other and only weights quantization methods are compared with each other.

The selected methods are implemented on VGG-7 with Cifar-10 dataset. Methods' accuracies and convergence characteristic are examined. We showed that quantizing models such as VGG-7 does not sacrifice much accuracy while providing significant compression. Quantization techniques with full precision cluster points converge faster compared to techniques with low precision cluster points such as -1 and 1. Hybrid methods for weights and activation quantization are proposed. One of the hybrid methods has a hardware friendly cluster points and a better accuracy compared to existing hardware friendly methods. The quantization methods are compared with ground-up efficient models in terms of accuracy and required size. It is shown that quantizing large models can end up with smaller size and better accuracy than ground-up efficient models.

The selected methods are implemented on ResNet models with Cifar-10 dataset. It is shown that weight quantization methods with full precision cluster points work well with ResNet architecture. One binary quantization method is failed in ResNet models which are deep and complex. The reasons behind these failures are examined. The reason is found to be asymmetrical quantization of weighted residual connections and its disturbance to the information flow. Solutions are proposed for this problem. The proposed solutions are implemented and proven to be successful. First approach of solutions focused on residual connections. In this approach, the weighted residual connections are not quantized or used. The second approach focused on scaling the layers' outputs. In this approach, weights or activations are scaled with constants. These solutions also showed that mean of the absolute weights is not an optimal scaling factor for binary quantization. Moreover, it is observed that the accuracy loss of weight and activation quantization methods are high in deep models. The reasons behind this accuracy losses are examined and showed with empirical proofs. Quantized ResNet models are also compared with a ground-up efficient model with residual connections.

A ground-up efficient model is also quantized and results are shown. Different versions of the ground-up efficient model are quantized. These versions include simple model with only core layers of a convolutional neural network and a complex model with residual connection on top of core layers.

Lastly, the convolution operation and its quantized version are implemented on various systems. These systems include a severely resource constrained embedded system, an average embedded system with an operating system and a powerful desktop computer. Comparisons and benchmarks of quantized operations are done in these systems. It is found that weight quantization methods can offer ~32x or ~16x compression with a cost of ~1.3x computation time increase.

6.2 Future Work

Even though it is possible to implement quantized neural networks on existing CPU systems, it only unleashes the half of the potential of the quantized neural networks.

We can achieve compression with CPU implementation on quantized neural networks. However, we cannot decrease computation time. In order to decrease computation time, a custom implementation of the quantized neural networks on FPGAs are required. The FPGA implementation and its optimization are left to a future work.

Moreover, the quantization techniques include a lot of manual effort to determine hyperparameters. Different methods could be used for different layers in the same architecture. Even though there are existing works that determine quantization bitwidths automatically, they are not integrated with quantization techniques. The hyperparameter search of quantization could be automatized as a future work.

REFERENCES

[1] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size,” pp. 1–13, 2016.

[2] A. G. Howard *et al.*, “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications,” 2017.

[3] X. Zhang, X. Zhou, M. Lin, and J. Sun, “ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices,” *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, pp. 6848–6856, 2018, doi: 10.1109/CVPR.2018.00716.

[4] F. Rosenblatt, “The perceptron: A probabilistic model for information storage and organization in the brain,” *Psychol. Rev.*, 1958, doi: 10.1037/h0042519.

[5] E. Hunt, M. Minsky, and S. Papert, “Perceptrons,” *Am. J. Psychol.*, 1971, doi: 10.2307/1420478.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” *Adv. Neural Inf. Process. Syst.*, vol. 2, pp. 1097–1105, 2012.

[7] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Comput.*, 1997, doi: 10.1162/neco.1997.9.8.1735.

[8] F. Chollet, “Keras Documentation,” *Keras.Io*, 2015. .

[9] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding,” *4th Int. Conf. Learn. Represent. ICLR 2016 - Conf. Track Proc.*, pp. 1–14, 2016.

[10] G. E. Hinton, “Neural networks for machine learning. Coursera, video lectures,” 2012. .

[11] Y. Bengio, N. Léonard, and A. Courville, “Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation,” pp. 1–12, 2013.

[12] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. C. Chen, “MobileNetV2: Inverted Residuals and Linear Bottlenecks,” *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, pp. 4510–4520, 2018, doi: 10.1109/CVPR.2018.00474.

[13] M. Courbariaux, Y. Bengio, and J. David, “BinaryConnect: Training Deep Neural Networks with binary weights during propagations,” pp. 1–9, Nov. 2015.

[14] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, “Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1,” 2016.

[15] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, “XNOR-net: Imagenet classification using binary convolutional neural networks,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 9908 LNCS, pp. 525–542, 2016, doi: 10.1007/978-3-319-46493-0_32.

[16] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, “DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients,” vol. 1, no. 1, pp. 1–13, 2016.

[17] F. Li, B. Zhang, and B. Liu, “Ternary Weight Networks,” no. Nips, 2016.

[18] C. Zhu, H. Mao, S. Han, and W. J. Dally, “Trained ternary quantization,” *5th Int. Conf. Learn. Represent. ICLR 2017 - Conf. Track Proc.*, pp. 1–10, 2019.

[19] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, “Incremental network

quantization: Towards lossless cnns with low-precision weights," *5th Int. Conf. Learn. Represent. ICLR 2017 - Conf. Track Proc.*, pp. 1–14, 2019.

[20] B. Jacob *et al.*, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, pp. 2704–2713, 2018, doi: 10.1109/CVPR.2018.00286.

[21] A. T. Elthakeb, P. Pilligundla, F. Mireshghallah, A. Yazdanbakhsh, S. Gao, and H. Esmaeilzadeh, "ReLeQ: An Automatic Reinforcement Learning Approach for Deep Quantization of Neural Networks," 2018.

[22] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "HAQ: Hardware-aware automated quantization with mixed precision," *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 2019-June, pp. 8604–8612, 2019, doi: 10.1109/CVPR.2019.00881.

[23] Y. He, J. Lin, Z. Liu, H. Wang, L. J. Li, and S. Han, "AMC: AutoML for model compression and acceleration on mobile devices," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 11211 LNCS, pp. 815–832, 2018, doi: 10.1007/978-3-030-01234-2_48.

[24] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," *Adv. Neural Inf. Process. Syst.*, vol. 2015-Janua, pp. 1135–1143, 2015.

[25] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, pp. 1929–1958, 2014.

[26] A. Paszke *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," no. NeurIPS, 2019.

[27] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning,"

in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016*, 2016.

[28] Y. LeCun and C. Cortes, “MNIST handwritten digit database,” *AT&T Labs [Online]. Available http//yann. lecun. com/exdb/mnist*, 2010.

[29] A. Krizhevsky, V. Nair, and G. Hinton, “CIFAR-10 and CIFAR-100 datasets,” *https://www.cs.toronto.edu/~kriz/cifar.html*, 2009. .

[30] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proc. IEEE*, 1998, doi: 10.1109/5.726791.

[31] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” Dec. 2015.

[32] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *3rd Int. Conf. Learn. Represent. ICLR 2015 - Conf. Track Proc.*, pp. 1–14, 2015.