Optimization of neural networks by reducing floating-point precision of weights during training

Dipanjan Dutta, School of Computer Science
McGill University, Montreal
September, 2020

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of

Master of Computer Science

Abstract

Deep learning is an iterative process with many tunable parameters. Repeated and long running executions benefit from optimizations that improve learning and testing performance, both in terms of time and power consumption. A popular approach to optimization focuses on faster learning by clipping weight values to a range *(quantization)* and converting them to fixed point precision. Quantization, however, results in significant loss of accuracy and requires further optimization to reach performance of full precision models, or use other resources, such as a lookup table [23] to compensate for the information loss. In this work we focus on reducing precision more within the spirit of the native IEEE 754 standard for the representation of float values. We propose a hierarchy of approaches that reduce the number of mantissa bits of a neural network weights based on the actual values at the end of every batch of training, as simulated through a manual mantissa reduction. At a coarse level we experiment with this approach applied to an entire learning network, extending the approach to a level-by-level precision adjustment to better adapt to the internal precision needs of different layers. We further propose a novel fine-grain approach, applying precision adjustment to groups of individual weights, dynamically clustered into "buckets," each having different precision. The approach performs almost as good as a full precision network in terms of training accuracy, but allowing for reduced space requirements, and a significantly lower prediction time.

Abrégé

L'apprentissage en profondeur est un processus itératif avec de nombreux paramètres réglables. Les exécutions répétées et de longue durée bénéficient d'optimisations qui améliorent les performances d'apprentissage et de test, à la fois en termes de temps et de consommation d'énergie. Une approche populaire de l'optimisation se concentre sur un apprentissage plus rapide en découpant les valeurs de poids dans une plage *(quantification)* et en les convertissant en précision en virgule fixe. Cependant, la quantification entraîne une perte de précision significative et nécessite une optimisation supplémentaire pour atteindre les performances des modèles de précision totale, ou utiliser d'autres ressources, telles qu'une table de recherche [23] pour compenser la perte d'informations. Dans ce travail, nous nous concentrons sur la réduction de la précision davantage dans l'esprit de la norme native IEEE 754 pour la représentation des valeurs flottantes. Nous proposons une hiérarchie d'approches qui réduisent le nombre de bits de mantisse d'un poids de réseau neuronal basé sur les valeurs réelles à la fin de chaque lot d'entraînement, comme simulé par une réduction manuelle de la mantisse. À un niveau grossier, nous expérimentons cette approche appliquée à tout un réseau d'apprentissage, en étendant l'approche à un ajustement de précision niveau par niveau pour mieux s'adapter aux besoins de précision internes des différentes couches. Nous proposons en outre une nouvelle approche à grain fin, appliquant un ajustement de précision à des groupes de poids individuels, regroupés dynamiquement en " seaux ", chacun ayant une précision différente. L'approche fonctionne presque aussi bien qu'un réseau de précision complète en termes de précision d'entraînement, mais permet des besoins d'espace réduits et un temps de prédiction nettement plus court.

Acknowledgements

I would like to offer my sincere gratitude to my supervisor, Professor Clark Verbrugge, for the continuous support of my research, advice, mentorship, patience and time. His guidance helped me throughout the duration of my research and in writing this thesis. I could not have asked for a better supervisor for my Master's thesis than him.

I would like to thank the Department of Computer Science, NSERC and the COHESA research network for their funding support during my studies.

I would like to thank my lab mates: Ivan Miloslavov, Adrian Koretski and Wael Al Enezi, for the stimulating discussions, debates and for all the fun we had over the years.

I would like to thank my friends, especially Mr. Ajinkya Vaidya, for their support, words of encouragement, and memories that made this journey easier.

Finally, I would like to thank my family and my significant other, Ms. Ena Ghosh, for having faith and patience over these three years and believing in me every step of the way.

Table of Contents

Abstract i
Abrégé ii
Acknowledgements iii
List of Figures vii
List of Tables viii

1 Introduction **1**
1.1 Motivation 1
1.2 Methodology and experiments 2
1.3 Contributions 3
1.4 Road map 4

2 Background and Related Work **6**
2.1 Artificial Neural Networks 6
2.1.1 Learning in neural networks 7
2.1.2 Multi-layer perceptron 9
2.2 Convolutional Neural Networks 11
2.2.1 Linear Image filter 11
2.2.2 Layers in a CNN 12
2.3 AlexNet 13
2.4 IEEE 754 Standard for Floating Point Arithmetic 14
2.5 Related work 16
2.5.1 Quantization 16
2.5.2 Hardware optimization 18
2.5.3 Floating point optimization of weights 20

3 Experimental Setup **22**
3.1 Data sets . . . 22
3.1.1 MNIST Database . . . 22
3.1.2 Fashion-MNIST database . . . 24
3.1.3 CIFAR10 Database . . . 25
3.2 Networks . . . 25
3.2.1 Dense network . . . 26
3.2.2 Convolutional Neural Network . . . 28
3.2.3 Modified AlexNet . . . 28
3.3 System configuration and hyperparameter selection . . . 30

4 Methodology **31**
4.1 Floating point reduction of weights . . . 31
4.1.1 The Lambda callback function . . . 32
4.1.2 The `bitstring` Python package . . . 34
4.1.3 Reduction of mantissa bits in Python . . . 38
4.2 Mantissa bit reduction strategy . . . 41
4.2.1 Whole network precision reduction . . . 43
4.2.2 Layerwise precision reduction . . . 44
4.2.3 Increasing bucket reduction . . . 44
4.2.4 Decreasing bucket reduction . . . 47
4.2.5 Other algorithms for benchmarks . . . 50

5 Results **52**
5.1 Whole network precision reduction . . . 52
5.2 Variance analysis . . . 56
5.3 Layerwise precision reduction strategy . . . 58
5.4 Increasing bucket reduction . . . 61
5.5 Decreasing bucket reduction . . . 66
5.6 Prediction on test data . . . 67

6 Conclusion and Future Work **74**

List of Figures

2.1 Example of a multilayer perceptron network. It includes different types of layer, such as the input (flatten) layer, the hidden dense layer, a dropout layer for regularization and an output layer. Each node is connected to all the node in the next layer. 10
2.2 Diagram of a sample convolutional neural network 11
2.3 Diagram of the AlexNet neural network architecture [34] 14
2.4 64 and 32 bit representations of IEEE-754 standard for floating points 15

3.1 Sample images from the MNIST dataset 23
3.2 Sample images from the Fashion-MNIST dataset 24
3.3 Sample images from the CIFAR10 dataset with labels 25
3.4 Structure of the dense network 26
3.5 Structure of the convolutional neural network 28

5.1 Whole network precision reduction of dense network on the MNIST data set 53
5.2 Whole network precision reduction on all the different datasets and networks 55
5.3 Whole network precision reduction on all the different datasets and networks 56
5.4 Training curves of one layer-wise precision reduction strategy of the CNN on the MNIST dataset 57
5.5 Layerwise precision reduction on all the different datasets and networks . . . 59
5.6 Layerwise precision reduction on all the different datasets and networks . . . 60
5.7 In-layer forward bucketing precision reduction on all the different datasets and networks 62
5.8 In-layer forward bucketing precision reduction on all the different datasets and networks 63
5.9 Distribution of weights in the first convolution layer of the CNN during training of forward bucketing precision reduction on the Fashion MNIST dataset . 64

5.10 Distribution of weights in the first convolution layer of the CNN after training of forward bucketing precision reduction on the Fashion MNIST dataset . . . 65
5.11 In-layer reverse bucketing precision reduction on all the different datasets and networks . . . 67
5.12 In-layer reverse bucketing precision reduction on all the different datasets and networks . . . 68

List of Tables

2.1 Examples of Activation Functions 7
2.2 IEEE-754 standards for 16, 32 and 64 bit representations 15

3.1 System configuration and software versions 30
3.2 Model and environment hyperparameters 30

4.1 Arguments for calling the LambdaCallback function 34
4.2 List of format specifier tokens for packing 37
4.3 List of endianness (left) and format (right) characters 37

5.1 Explanation of legends in the graphs 54
5.2 Prediction accuracy (in percentage) on the test data[1] 69
5.3 Prediction time (in seconds) on the test data[1] 70
5.4 Total number of trainable weights in the network 70
5.5 Total number of weights in each bucket of the CNN post training on Fashion MNIST dataset 70
5.6 Total number of weights in each bucket of the CNN post training on Fashion MNIST dataset 71

Chapter 1

Introduction

In this thesis, we study and research the optimization of neural networks by reducing floating-point precision of weights during the training phase. In section 1.1 we discuss our motivations for the research. In section 1.2 we list and briefly describe our methodology and experiments, along with our results and contributions. In section 1.3 we present a road map of the chapters composing this thesis.

1.1 Motivation

Neural networks are a popular and powerful tool and a cornerstone for solving modern day problems using machine learning. However, with the increase in volume and quality of data, machine learning algorithms become more complex and training them requires greater time. Thus, there is a need for optimization of these algorithms. Extensive research has been done into creating optimization techniques that involve the reduction in the amount of memory used to store the parameters of neural networks to reduce storage space occupied by the network and arithmetic computation time. This has led to development of ideas in both software and hardware domains. Software optimizations such as *quantization* has been applied to reduce the weights of neural networks to reduce computation and save memory by converting the weights into fixed-point representation of float values. Quantization, in some cases, has been aggressively implemented to reduce the weights of a network even down to binary values [26]. These quantization techniques, although effective in reducing computation time of arithmetic operations, require further optimization techniques and fine-tuning to be accurate and perform as well as a full precision model [11, 21, 54]. In the hardware domain, improvements to general-purpose GPUs and CPUs have been developed by leading

companies like NVIDIA [47] and Google [30] respectively. Dedicated hardware for machine learning purposes are also developed using Field Programmable Gate Arrays (FPGAs) to improve of neural network [19, 45].

Because of the IEEE 754 standard of floating-point representation being already implemented by most processors for float operations, we decided to research into a possibility of creating an optimization technique that would leverage the standard, while keeping the algorithm simple and allowing for flexibility of precision in the network. Such an optimization technique can also serve as a theoretical simulation for further exploring hardware techniques to optimize floating-point operations during the training of neural networks [31]. Thus, in our research, we explore the possibility of reducing the weights in a neural network while maintaining use of floating-point representation, aiming to achieve results comparable to networks trained with full precision.

1.2 Methodology and experiments

Our search for an optimization strategy began with the handling and manipulation of float values in the IEEE 754 standard inside the environment of Python, since Python has simple, concise and readable code with an extensive framework of libraries for machine learning (Tensorflow, Keras etc.). Python, however, does not provide an easy in-built method of manipulating bits or setting bit-widths of floats. It does support 16-bit, 32-bit and 64-bit floating-point precision and allows the programmer to switch between the precisions, but it does not allow any other form of bit manipulation and handling. We leverage a module developed in Python, "`bitstring`" that converts a float to its IEEE 754 format and allows for manipulation of the bits. Using this module, we decided to develop strategies to reduce precision of weights in neural network during training, while maintaining a floating-point representation, by pruning the trailing bits of mantissa in the weights to a certain number of bits and converting it back to the float value.

We decided to explore possible strategies of precision reduction. Inspired from the general strategies of precision reduction implemented in quantization, we created two strategies to provide a general baseline and point of comparison. The first strategy involved reducing the precision of the entire network to a common precision for the mantissa of the weights. The second strategy involved selecting a different precision for every layer of the network. Since the possibilities of network configurations are large, we decided to experiment this strategy by selecting a certain precision for each type of layer in the network that has learnable pa-

rameters.

Having designed two preliminary approaches of precision reduction, we explored further to try and reduce the precision of the weights on a finer granularity than that of a layer. Realizing that selecting precision for each individual weight is cumbersome and time-consuming, we decided for a trade-off. We developed a method that involves classifying the weights of every layer into clusters, or "buckets", and selecting a precision for the bucket. All the weights in a bucket get reduced to the precision chosen for that bucket. This method allows for a greater flexibility of precision reduction than that of a layer, while not being as intensive in computational requirements as that of weight-by-weight basis. As mentioned in the previous paragraph, because of the multitude of possibilities of selecting a precision for a bucket, we decided to select three different values of precision and train the models.

Training the models with the same network configuration as the previous methods and averaged over multiple iterations to reduce the effect of random initial weights and dataset sampling, we found an interesting result, which showed that the bucket approach trained quite well, with the choice of increasing the precision by 5 bits every bucket performing on par with the full precision model. Surprisingly, while performing predictions on the test data, the bucketing approach had an improved result, with some settings of the strategy having prediction accuracy similar to that of the full precision network, taking significantly less time to predict, while reducing the precision of the weights in the network by over 70kb in some cases. The strategy to classify weights into buckets based on value for precision reduction shows significant promise and is almost as accurate as the full precision model, while taking less time to predict and reducing precision to save memory and computational time.

1.3 Contributions

This experimental work provides a few major contributions towards further exploration of floating-point precision reduction of weights.

- We provide a means of using a floating-point representation in line with IEEE-754 for precision reduction in neural networks. Almost all modern-day CPUs (Intel x86, AMD, PowerPC and other RISC processors) use IEEE 754 as their floating-point arithmetic standard. Floating-point representation is more flexible and has a larger range of values. Using floating-point representation for weights does not require additional

conversion to fixed-point precision in the software level, or other optimization for preventing information loss. This opens up avenues of developing hardware compliant with modern processors for custom precision.

- We propose an unique algorithm to reduce precision of weights in a neural network while maintaining the spirit of the IEEE 754 floating point standard. We develop algorithms by going step-by-step into a finer level of abstraction, starting with reducing the precision of all the weights in the entire network, then investigating the effect of different precision in different layers, up to our technique of bucketing individual weights based on their values and reducing the weights in a bucket to an assigned level of precision for that bucket. The latter approach is aimed at associating each weight with the precision that matches the weight significance. This novel algorithm is not as generalized as a whole network precision reduction, while being not as computationally intensive as determining and setting the precision of every single weight. It is more robust and more tolerant to information loss due to reduction of precision. Using this algorithm, we create a simulation to successfully reduce precision of weights during training that would hopefully facilitate the development and design of hardware to support custom precision.

- We experimented on the designed strategy of precision reduction mentioned above, which trains with an accuracy at par with, or in some cases better than networks trained with full precision. Although our software-simulated precision reduction magnifies training time, during prediction, the network trained on the designed strategy takes significantly less time to predict, while reducing a greater number of bits over the other strategies developed for comparison. Given that our experiments were a simulation, the results seem to indicate that the greater number of bit reduction by the "bucketing" would translate to a greater amount of memory saved on a hardware with custom precision.

1.4 Road map

We present a total of 6 chapters in this thesis. Each of the chapters have been briefly explained in the following list.

- Chapter 1 (the current chapter) provides an introduction to our research, the motivation behind the research and exploration of floating-point precision reduction, the methodology and experimentation performed for our research and a road map of the chapters of the thesis.

- Chapter 2 encompasses a brief discussion about neural networks used in the experimentation of our research, floating-point precision standards and an overview of the related work in the field of our research.

- Chapter 3 presents a brief explanation of the datasets and networks used in our experiments, along with the system information and hyperparameter settings for our experiments.

- Chapter 4 is a discussion of the Python packages used to implement the different parts of our experiments, an explanation the strategy developed for floating-point precision reduction, benchmarks and pseudo codes of these strategies.

- Chapter 5 is a presentation and discussion of the results obtained upon performing the experiments designed in the previous chapter, along with an analysis of variance in our experiments and prediction performance of the various strategies.

- Chapter 6 offers our concluding remarks, summarizing our research work and results, and possible avenues of future work that can potentially be promising to the extension of this research.

Chapter 2

Background and Related Work

This chapter explains the fundamentals of a neural network. This includes the different types of neural networks used in our experiments - the multilayer perceptron, a simple convolutional neural network and a larger AlexNet network, the different types of layers that comprise a neural network, how a neural network learns and the backpropagation algorithm. We also discuss the IEEE 754 floating point convention and provide a detailed discussion of previous work done on optimization of weights in a neural network layer.

2.1 Artificial Neural Networks

Neural networks, as the name suggests, are composed of a several neurons. Artificial neural networks are derived from the structure and composition of the nervous systems in biological organisms. Artificial Neural Networks are composed of artificial neurons (also referred to as nodes). These neurons are connected and organised in different ways to perform different tasks such as image recognition and synthesis, object detection, language translation, predictive modelling, medical diagnosis, computer vision, etc.

A neuron in an artificial neural network is very similar to a biological neuron. Every neuron has one or more inputs and one or more outputs. Every input to a neuron is multiplied by a floating-point number (or "weight") to create a weighted signal. The weight signifies the relative importance of that input. These weighted signals are then used as input to an output function that creates the output of the neuron. Sometimes there is an optional activation function, which can be something like the weighted sum of all the signals to the neuron, the result of which is compared to a threshold. If the activation function value is more than the threshold, then the result of the output function is generated and propagated

to following neurons, otherwise the neuron does not generate any results.

Every neuron performs the following actions:

- The neuron receives n signals as input that represents information about the data $[X_1, X_2, ...X_n]$. It then produces a linear combination of the input data and the weights of the corresponding connections and adds the results together.

$$S = \sum_{i=1}^{n} X_i w_i \tag{2.1}$$

- The neuron then implements an activation function on the result of Equation (2.1)

$$Y = \sigma(S) \tag{2.2}$$

  and generates the final output. The activation functions introduce non-linearity into the neuron. Without the activation functions, the end result of the entire network will be a combination of linear transformations, which is the equivalent of having a single neuron [12]. Some of the common activation functions are listed in Table 2.1

Function	**Equation**	**Range**
Linear	$\sigma(S) = aS, a \in \mathbb{R}$	$(-\infty, \infty)$
Sigmoid	$\sigma(S) = \frac{1}{1+e^{-S}}$	$(0, 1)$
Tanh	$\sigma(S) = \frac{2}{1+e^{-2x}} - 1$	$(-1, 1)$
Rectified Linear Unit (ReLU)	$\sigma(S) = max(0, S)$	$(0, S)$
Exponential	$\sigma(S) = e^{-S}$	$(0, \infty)$

Table 2.1: Examples of Activation Functions

The arrangement of these neurons and the connections between them are different for different neural network architectures. The network architecture is designed to learn the most from data while reducing the prediction error of the network.

2.1.1 Learning in neural networks

Learning in an artificial neural network involves the recalibration of the weights, biases and activations of the nodes in the network. There are several steps that are followed to create a network and train it.

The first step is the determination of the network architecture. That includes decisions like the number of hidden layers, number of nodes in each layer and the number of nodes in the output layer (depending on the number of output classes). The number of nodes in the input layer also follows from the context of the dimension of input data. However, there is no deterministic approach to ascertain the ideal number of nodes in the hidden layer(s), although there has been some discussions about the ideal number of nodes in the hidden layers [6, 36, 49, 57].

Once the network architecture is determined, the process of adjusting the weights of a neural network is done by the gradient descent optimization algorithm [7]. The gradient descent approach is utilised by the supervised learning algorithm called **backpropagation**. The steps involved in the training of neural networks are as follows (in order):

- **Initialization.** Weights and biases are randomly initialized across the network.

- **Forward pass.** Each item in the training data set is fed into the network one after the other. Let us assume the training data consists of D data points, each of which has an input vector $\mathbf{x}_i$ and the expected output vector $\mathbf{y}_i$. Then the forward pass computations are performed as:

$$y_j = \sigma\left(\sum_{i=1}^{n} x_i w_{ji} + b_j\right) \tag{2.3}$$

  where y_j is the output of the j^{th} layer, σ is the activation function, x_i is the output from the i^{th} node of previous layer (or the input if it is the first layer), w_{ji} is the weight of the connection from the i^{th} node of previous layer to the j^{th} node of current layer and b_j is the bias of the current layer.

- **Backward pass.** For each of the input-output pair $(\mathbf{x}_i, \mathbf{y}_i)$, we compute the backward phase by computing $\frac{\partial C}{\partial w_{ij}^k}$, starting from the output layer and proceeding to the first layer, where C is the individual error term or value of the cost function and w_{ij}^k is the weight of the connection between node i in layer $k-1$ and node j in layer k. The backward pass is mainly composed of three steps:

  1. First, we evaluate the error for the final/output layer δ^{out} using the equation:

$$\delta_j^{out} = \frac{\partial C}{\partial a_j^{out}} \sigma'(z_j^{out}), \tag{2.4}$$

where δ_j^{out} is the error by the weights of the j^{th} neuron in the output layer, $\frac{\partial C}{\partial a_j^{out}}$ just measures how fast the cost is changing as a function of the j^{th} output activation value. The equivalent matrix-based equation is:

$$\delta^{out} = \nabla_a C \odot \sigma'(z_j^{out}), \tag{2.5}$$

which essentially transforms equation 2.4 to a dot product.

2. We propagate the error backwards through the layers. For every layer $l = m - 1, m - 2, ..., 3, 2$, we calculate the loss δ^l using the formula:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l), \tag{2.6}$$

where $(w^{l+1})^T$ is the transpose of the weight matrix w^{l+1} for the $(l+1)^{th}$ layer. If we know the error δ^{l+1} at the $(l+1)^{th}$ layer, applying the transpose of the weight matrix gives us a measure of the error at the output of the l^{th} layer. Then taking the dot product of the activation function of that layer moves the error through the activation function of layer l, giving us the error δ^l for layer l.
Combining equations 2.5 and 2.6, we can calculate the errors of all the layers, starting from the output layer.

3. Finally, we calculate the gradients and update the weights and biases of the network. The gradient of the cost function with respect to the weights and biases, respectively, are as follows:

$$\frac{\partial C}{\partial w_{ij}^l} = a_j^{l-1} \delta_i^l \tag{2.7}$$

$$\frac{\partial C}{\partial b_i^l} = \delta_i^l \tag{2.8}$$

This process repeats over and over until we stop training, or the error is below the tolerance specified for the network.

2.1.2 Multi-layer perceptron

This section explains in detail the most fundamental type of artificial neural network - the multilayer perceptron (or MLP).

Multilayer perceptrons contain three types of neurons:

- *Input neurons:* These neurons form the first layer of the network (also called the **input layer**). These take vector encoding of the data and pass on to the subsequent layers without any computation.

- *Output neurons:* These neurons form the final layer of the network (also called the **output layer**). These neurons receive input from the previous layer in the network and using equations (2.1) and (2.2), generate the final output of the entire network.

- *Hidden neurons:* These neurons are arranged in layers (called **hidden layers**) that form the backbone of the MLP network. They can be arrange in one or several layers between the input and output layers of neurons. These neurons receive input (from the input layer or previous hidden layers) and using equations (2.1) and (2.2), process and generate results that are passed on to subsequent layers.

An example of such a network is shown in Figure 2.1.

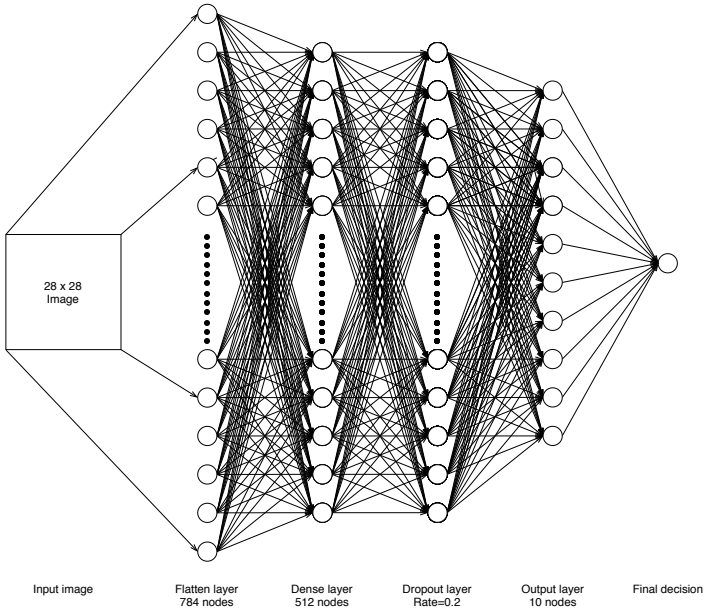

Figure 2.1: Example of a multilayer perceptron network. It includes different types of layer, such as the input (flatten) layer, the hidden dense layer, a dropout layer for regularization and an output layer. Each node is connected to all the node in the next layer.

2.2 Convolutional Neural Networks

Convolutional Neural Networks (or ConvNets or CNNs) are massively and successfully used in image classification tasks such as speech recognition [24]. However, the inner workings and theoretical understanding of how a CNN performs so well in solving these problems is not clear. The most reliable and standard form of evidence is statistics and performance evaluation metrics. In this section, we discuss the idea of a linear image filter, which forms the basis of a CNN architecture, followed by a brief discussion of the different types of layers in a CNN architecture.

2.2.1 Linear Image filter

A linear filter for images is an element $E \in \mathbb{R}^{f_w \times f_h \times c}$, where f_w is the width of the filter, f_h is the height of the filter, and c are the number of channels in the filter, which are same as the number of channels in the input image. The filter E is convolved with the image I with dimensions $w \times h \times c$ to create the result $\bar{I}$ that is composed of just a single channel. The convolution operation creates each pixel of the resultant channel by performing multiplying each filter element, point-wise, to an element of the original image I. So even though it is technically called a convolution, it is essentially a *sliding dot product*.

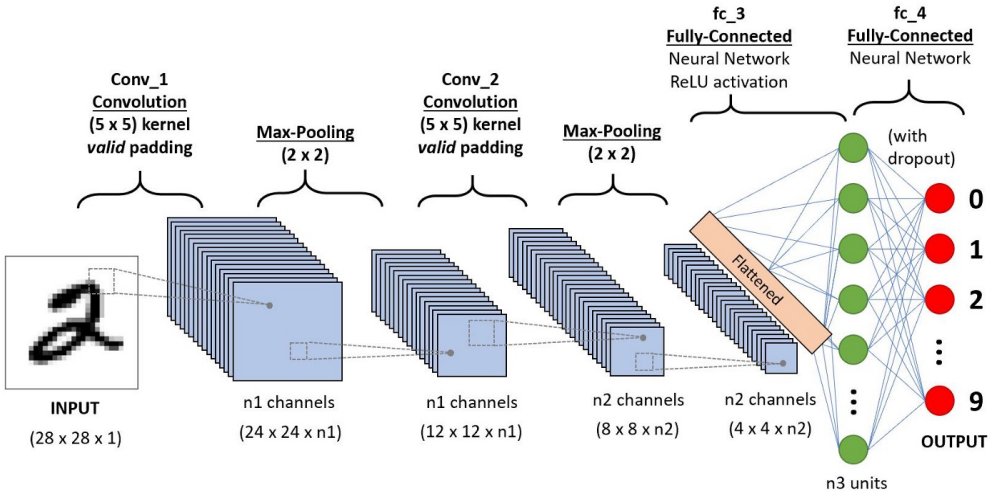

Figure 2.2: Diagram of a sample convolutional neural network

2.2.2 Layers in a CNN

Convolutional neural networks are designed with the premise of *spatial correlation* and *translational invariance*. *Spatial correlation* means that parts or segments of an image hold some correlation between them in terms of relative location from one another. For example, one eye of a dog will be within a very short distance of another eye of the dog. *Translational invariance* states that an object can still be recognised irrespective of its location in the image. Combining these two, it is thus ideal to develop and capture high-level features (edges, curves etc.) in the early layers of the network and finer features (faces, eyes etc.) in later stages of the network. The most common convolutional networks have the following different types of layers:

- Convolutional layers: These layers convolve the inputs to these layers and pass it on to the next layer. The input to this layer is a tensor of size $n \times w \times h \times c$ where n is the number of images, w and h are the dimensions of the images and c is the number of channels of the image (RGB etc.) Then after passing through a convolutional layer, the image becomes abstracted to a feature map, with shape $f_n \times f_w \times f_h \times f_c$ where f_n is the number of images in the filter, f_w and f_h are the feature map dimensions and f_c are the feature map channels. A convolutional layer within a neural network should have the following attributes:
  - Convolutional kernels defined by a width and height
  - The number of input channels and output channels
  - The depth of the convolution filter (the input channels) must be equal to the number channels (depth) of the input feature map
- Pooling layers: Pooling layers reduce the dimensions of the data by combining the outputs of neuron clusters at one layer into a single neuron in the next layer. Local pooling combines small clusters, typically 2 x 2. Global pooling acts on all the neurons of the convolutional layer. In addition, pooling may compute a max or an average. Max pooling uses the maximum value from each of a cluster of neurons at the prior layer. Average pooling uses the average value from each of a cluster of neurons at the prior layer.
- Fully connected layers: Fully connected layers connect every neuron in one layer to every neuron in another layer. It is in principle the same as the traditional multi-layer

perceptron neural network (MLP). The flattened matrix goes through a fully connected layer to classify the images.

2.3 AlexNet

AlexNet is the name of a convolutional neural network (CNN), designed by Alex Krizhevsky, and published with Ilya Sutskever and Krizhevsky's doctoral advisor Geoffrey Hinton [34]. AlexNet is considered one of the most influential papers published in computer vision, having spurred many more papers published employing CNNs and GPUs to accelerate deep learning. AlexNet was the winning entry in ILSVRC 2012. It solves the problem of image classification where the input is an image of one of 1000 different classes (e.g. cats, dogs etc.) and the output is a vector of 1000 numbers. It is essentially a very large convolutional neural network.

According to the paper, the net contains eight layers with weights; the first five are convolutional and the remaining three are fully connected. The output of the last fully-connected layer is fed to a 1000-way output layer with **softmax** activation, which produces a distribution over the 1000 class labels.

- The first convolutional layer filters the $224 \times 224 \times 3$ input image with 96 kernels of size $11 \times 11 \times 3$ with a stride of 4 pixels.
- The second convolutional layer takes as input the (response-normalized and pooled) output of the first convolutional layer and filters it with 256 kernels of size $5 \times 5 \times 48$.
- The third convolutional layer has 384 kernels of size $3 \times 3 \times 256$ connected to the (normalized, pooled) outputs of the second convolutional layer.
- The fourth convolutional layer has 384 kernels of size $3 \times 3 \times 192$
- The fifth convolutional layer has 256 kernels of size $3 \times 3 \times 192$
- The third, fourth, and fifth convolutional layers are connected to one another without any intervening pooling or normalization layers
- The fully-connected layers have 4096 neurons each

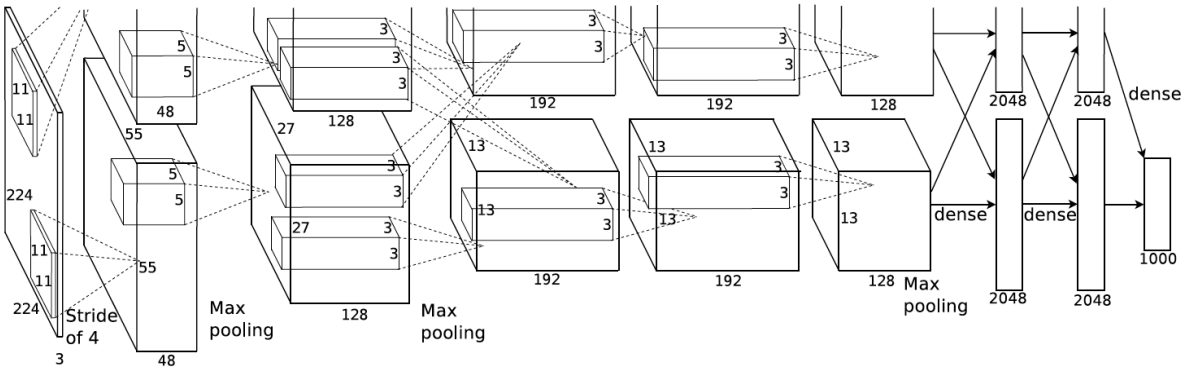

Figure 2.3: Diagram of the AlexNet neural network architecture [34]

2.4 IEEE 754 Standard for Floating Point Arithmetic

The IEEE 754 is a standard of floating point arithmetic that was established by the Institute of Electrical and Electronics Engineers (IEEE) in 1985 (recently revised in 2008). The standard addressed many problems found in the diverse floating-point implementations that made them difficult to use reliably and while being portable. In the current world, many hardware floating-point units use the IEEE 754 standard.

A floating-point format is specified by:

- a base (also called radix) b, which is either 2 (binary) or 10 (decimal) in IEEE 754;
- a precision p;
- an exponent range from e_{min} to e_{max}, with $e_{min} = 1 - e_{max}$ for all IEEE 754 formats.

The IEEE 754 format can specify the following:

- Finite numbers, which can be described by three integers: $s =$ a *sign* (zero or one), m is a *significand* (or coefficient or mantissa) having no more than p digits when written in base b (i.e., an integer in the range through 0 to $b^p - 1$), and q is an exponent such that $e_{min} \leq q + p - 1 \leq e_{max}$. Moreover, there are two zero values, called signed zeros: the sign bit specifies whether a zero is $+0$ (positive zero) or -0 (negative zero).
- Two infinities: $+\infty$ and $-\infty$
- Two kinds of NaN (not-a-number): a quiet NaN (qNaN) and a signaling NaN (sNaN).

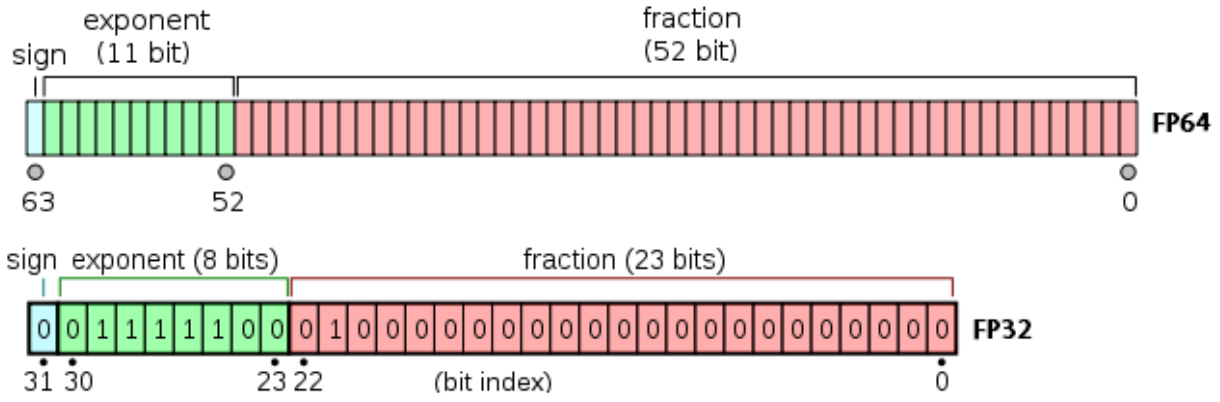

Figure 2.4: 64 and 32 bit representations of IEEE-754 standard for floating points

The numerical value of such a finite number is $(-1)^s \times c \times b^q$, whece c is the significand or the mantissa. For example, if the base is 10, the sign is 1 (indicating negative), the mantissa is 12345, and the exponent is -3, then the value of the number is $(-1)^1 \times 12345 \times 10^{-3} = -1 \times 12345 \times 0.001 = -12.345$.

Name	Common Name	Base	Significand bits	Exponent bits	Exponent bias	Min. exponent	Max. exponent
binary16	Half precision	2	11	5	$2^4 - 1 = 15$	-14	$+15$
binary32	Single precision	2	24	8	$2^7 - 1 = 127$	-126	$+127$
binary64	Double precision	2	53	11	$2^10 - 1 = 1023$	-1022	$+1023$

Table 2.2: IEEE-754 standards for 16, 32 and 64 bit representations

Since the floating point numbers are stored in binary as bits inside a computer memory, the IEEE 754 encoding and representation in binary format is also specified separately. For the binary formats, the exponent is not represented directly, but a bias is added so that the smallest representable exponent is represented as 1, with 0 used for subnormal numbers(numbers greater than zero but smaller than the smallest number that ca be represented using the the IEEE-754 standard of chosen precision, that is $0 < subnormal < 1.0 \times 2^{e_{min}}$). For numbers with an exponent in the normal range (the exponent field being not all ones or all zeros), the leading bit of the significand will always be 1. Thus, in most hardware representations, a leading 1 is usually implied rather than explicitly present in the memory encoding, and under the standard the explicitly represented part of the significand will lie between 0 and 1. This is known as *hidden bit convention*. This rule allows the binary format to have an extra bit of precision. However, the hidden bit convention cannot be used for the subnormal numbers as they have an exponent outside the normal exponent range and

scale by the smallest represented exponent as used for the smallest normal numbers. Table 2.2 shows the configurations of all the IEEE-754 standards, namely, single precision, double precision and half precision.

2.5 Related work

There has been tremendous development in the reduction of bit-width of weights and gradients in a neural network during training in order to reduce computational workload and storage requirements for the models. Since the amount of data and complexity of problems increases over time exponentially, researchers work to find ways to reduce the computational workload, while trying to maintain similar levels of performance, to make neural networks effective [2]. Some domains of precision reduction in neural networks are quantization, modifications in hardware used for training machine learning models and even reducing floating point precision for weights and activations in a neural networks, which is our objective for the experiment. In the following subsections, we discuss each of the above in details.

2.5.1 Quantization

Quantization essentially means the reduction of the number of bits used to represent a number. In terms of deep learning, the weights and activations are mostly computed and stored using 32-bit floating point representation (more commonly known as FP32). What quantization tries to do is reduce the reduce the precision of the weights and activations of a neural network to lower bit representations, or even to a fixed point representation [27]. This has three main advantages:

- The amount of space required to store the trained model reduces significantly.
- Memory bandwidth required for performing computations like multiplications will be less.
- Power consumption for those computations, in turn, will be lower.

Quantization can be done quite aggressively, even reducing weights in a network to binary [11, 26, 40] or ternary [1, 37] values. This increases the computational efficiency of the network, reducing complex multiplications to simple additions and subtractions. We can go even farther by making activations as binary, thus removing additions altogether in favor of

simple bit-wise operations, as implemented in XNOR-Net [53].

There are two main approaches for quantizing neural networks. The first approach is to train the entire network at a reduced precision. All the weights, activations and gradients are reduced to a lower precision during training. This results in a decrease in training time and prediction time [64]. The issue that arises with this approach is that the backpropagation algorithm is not well-defined for discrete or reduced-precision weights and activations [9]. There have been suggested techniques on the backpropagation mechanisms in reduced-precision training of neural networks. One such mechanism is to use a "straight-through estimator". Basically, during backpropagation, the network will not learn anything if the threshold function's derivative is zero. With this technique, the threshold unit acts normally during the forward pass and behaves as an identity function during the backward pass [4]. This approach is used in the design of "BinaryConnect" network by Courbariaux et. al. [11] and the "Binary Neural Network" proposed by Hubara et. al. [26]. Courbariaux et. al. also proposes using random rounding off to perform stochastic quantization, that acts as a regularizing noise for the network during training This acts similarly to a dropout layer that tries to prevent overfitting by randomly reducing the precision of weights, instead of omitting them [11]. This is studied by Gupta et. al. extensively in their work [20]. Since quantization discretizes the weights of a neural network, the derivative for the distribution of the parameters must be identified. so that the model can learn during backpropagation. If the approximation function is not differnetiable, the backpropagation would fail, and the model cannot be trained effectively. Asanovic et. al. and Holt et. al. analyzed and tried to determine the ideal precision requirement for a network during the backpropagation phase [3, 25]. Shayer et. al. repurposed the "local reparameterization trick" for continuous functions proposed by Kingma and Welling [32] and assumed that parameters have defined gradients by smooth approximation [54]. Gong et. al. has used vector quantization in deep convolutional neural networks for non-uniform quantization of parameters, after using k-means algorithm to cluster the weights and creating a lookup table in [16]. There have been proposals for a more complex quantization scheme as well, where the residual of the approximation is refined for full precision input [39]. Some other approaches to backpropagation for quantized training include discretization of weights with the assumption of inputs with a large fan-in, hence follow a Gaussian distribution [55], trying to find a "good" approximation. Others have developed "mixed precision" training of neural networks, using dynamic fixed point precision scaling [13, 43].

Another school of thought thus began developing, that involves targeting direct quantiza-

tion of a network that is already trained with FP32 precision and converting the parameters to lower precision without full training. Even with a simple approach such as uniform quantization, INT8 quantization seems perform fairly well and is quite robust towards quantization noise [29]. Another approach proposed was to improve training generalization by using retraining with backpropagation and alternating optimization to reduce L_2 error [28]. Another proposal was to reformalize quantization of parameters as a *"Minimum Mean Squared Error (MMSE)"* problem that allowed for training in low precision without retraining [9]. Lai et. al. experimented on a different approach, changing the activations to fixed-point precision while keeping the floating-point precision for the weights [35].

2.5.2 Hardware optimization

The need for optimization comes form the fact that the training of deep neural networks is constrained by hardware limitations. Primary approaches to this issue are addressed by improving the exploitation of general-purpose hardware, such as CPU clusters [10] as well as GPUs [34]. To fully utilize these general-purpose hardware to improve performance in deep learning, several frameworks have also been developed. NVIDIA and Google have hardware accelerators that can benefit the performance. NVIDIA's Volta's FP16 Tensor Cores [47] and in Google's Tensor Processing Unit [30] both leverage the ability to separate dot products, which are the most computationally intensive operation during training, and other operations during the training process.

Large feed forward neural networks, if designed to work with floating point precision, perform a high number of additions and multiplications using floating point precision. Without dedicated hardware for floating point computations, it can negatively affect the performance and training of the network, making it difficult to apply such algorithms in real-time solutions. For this reason, Field Programmable Gate Arrays (FPGAs) are now designed to improve training of neural networks. Compared to a microcontroller implementation (based on the sequential execution of instructions by the CPU) the nature of an FPGA design exploits the concepts of customization and parallelization to enhance the throughput of a computational system [42, 58]. Customization is very flexible using Hardware Description Language (HDL) that allows the designer to design right up to the register level, defining as a matter of fact a flexible Application Specific Integrated Circuit (ASIC). Parallelization separates modular and sequential portions of algorithms, improving the efficiency and performance of complex algorithms by several times.

There have been several specialised applications of FPGAs in training and implementing neural networks. Farabet et al. [15] created a face detection system with three convolutional layers and five fully connected layers on FPGA architecture with a performance of 10 frames per second. Zhang et al. [63] proposed a new description algorithm for CNNs and Qiu et al. [51] proposed a deeper and more complex VGG model on an FPGA architecture. Gysel introduced an anutomated framework, "Ristretto", that simulates a custom hardware accelerator for training CNNs [21].

FPGA neural network accelerators can also benefit from optimization. Over the last few years, several hardware-level techniques have been proposed for FPGA-based neural network accelerator design to achieve high performance [38, 41]. Several of them are focused on computation unit design. Computation unit level design is important as it affects the performance of a neural network accelerator. The resources on an FPGA chip are limited and a well-designed computation unit can improve peak performance. Reducing the bit-width of computation can greatly reduce the size of the computation unit and make the FPGA chip more efficient. The feasibility of reduced bit-width is realized from the quantization techniques discussed in section 2.5.1. However, most research on FPGA design improvement is focused on replacing 32-bit floating point numbers with fixed-point units - Podili et al. [50] shows an implementation of 32-bit fixed-point units on FPGA, and research has also shown implementations of 16-bit fixed-point designs [18, 38, 51]. Han et al. proposed the ESE framework [22] that adopts 12-bit fixed-point weights and 16-bit design. Guo et al. [19] used 8-bit weights design for their FPGA design. A comprehensive report is also published by Nurvitadhi et al. [46] that shows the improvement in performance of binarized neural networks in an FPGA over CPUs and GPUs.

Modern FPGAs are composed of embedded Digital Signal Processing (DSP) units, which implement a hard multiplier, pre-adder and accumulator core. This fits into a neural network design, whose basic computations are multiplication and summation. Fixed-point data with 16-or-less-bit fixed-point precision are well fit into 1 DSP unit on a modern Xilinx or Altera FPGA. Because a DSP unit is the computational core of these FPGAs, they hardly benefit from reduced precision, such as 8-bit or 4-bit, since these units are embedded with a rigid bit width, so even if it is working on a 8-bit value, it blocks the remaining 8 bits from use by other threads. The wide multipliers and adders in DSP units are underutilized in these cases. To overcome this, some workarounds on design are proposed. Nguyen et al. [45] proposed the design to implement two narrow bit-width fixed-point multiplication with a single wide bit-width fixed-point multiplier. Similar methods can be used in other DSPs and bit-widths.

The objective of our experiments in this thesis work is to improve or retain the same level of accuracy using floating-point precision. Since DSPs in FPGAs like Altera natively supports floating-point precision, this simulation can, in theory, leverage the native hardware without any further optimization to get the maximum performance in a neural network without compromising on accuracy.

2.5.3 Floating point optimization of weights

In the previous sections, we have discussed fixed-point optimization using quantization of weights and hardware optimizations for improved training. However, because fixed-point optimization techniques have a significant reduction of training time and with fine-tuning of the algorithms can overcome the loss of accuracy, most research is now focused on quantization and fixed-point reduction of weights and activations in a neural network. In our thesis, we intend to explore the possibility of floating-point precision reduction of weights in a simulated environment because of the following two reasons.

- The IEEE 754 standard for floating point representation is used by almost all modern computers and trying to leverage that standard while optimizing weights in a neural network incorporates a simplicity in the design of the system, where we do not need to define a new algorithm for computers to interpret floating-point numbers differently. The standardization motivates simplicity of the algorithm.

- Another reason is the flexibility floating-point precision offers. Floating-point representation is more versatile and can represent various levels of precision, which is useful for creating a more flexible and adaptive network that might require higher precision in some levels and lower precision in others.

There has been some research related to floating-point reduction of weights in neural network. Drumond et al. [14] proposed a design for training deep neural networks using a hybrid version of the "block floating point (BFP)" representation. Block floating points have been historically used in signal processing [48, 52, 56, 60], where exponents are shared across tensors, and this facilitates dense fixed-point logic for hardware accumulators and multipliers. Thus block floating-point essentially implements floating-point representation into a fixed-point quantized representation for the algorithms. Some modifications of BFP are also used condense radar data without inherent assumptions about the distribution of the data [44]. Microsoft has also acquired a patent for the application of block floating-point

for implementation in neural networks [5].

However, block floating-point still has a few drawbacks. First, BFP dot-products are very efficient, but the same cannot be said for other types of operations. Second, if the exponent values in a tensor are varied from too high to too low, it might lead to data loss and significant inaccuracies. Drumond et al. [14] proposed a hybrid version where BFP is used for dot products and other operations in floating-point.

Our thesis experiments on the other part of floating-point representation, the mantissa. We try to reduce the number of mantissa bits while maintaining a similar level of training and prediction accuracy of the network. This was motivated because of two reasons.

- In an IEEE 754 standard for floating-point representation, the mantissa consists of more bits (24 and 53 mantissa bits compared to 8 and 11 exponent bits in 32 and 64 bit standards respective), hence the possibility of reducing more bits, and in turn operations, without incurring too much error is more for mantissa reduction than exponent reduction.

- It is still possible to use exponent reduction techniques, such as BFP, in conjunction with the mantissa reduction.

Thus, we decided to experiment on the reduction of mantissa bits as a simulation in Python to find if the accuracy is similar to a network trained with full precision.

Chapter 3

Experimental Setup

This chapter explains in detail the four datasets the experiments are performed on. It is followed by a description and explanation of the three different types of networks on which the experiments are performed.

3.1 Data sets

There are four datasets used for running experiments. All of them are image datasets. The four datasets used are :

- Modified National Institute of Standards and Technology (MNIST) [62]
- Fashion Modified National Institute of Standards and Technology (Fashion-MNIST) [61]
- Canadian Institute For Advanced Research - 10 (CIFAR10) [33]

Both the MNIST and Fashion-MNIST datasets are relatively small and easier to learn, whereas the CIFAR datasets are large and for difficult to learn for algorithms. We chose two different types of datasets to compare the performance in simple, easy-to-learn datasets as well as complex datasets.

3.1.1 MNIST Database

The MNIST database was created from the original NIST database. It is the most common database used to train and benchmark image classification neural networks. The MNIST

dataset consists of images of handwritten digits. It has 60,000 training samples and 10,000 test samples. All the images are in grayscale format.

The MNIST database is created from the binary black-and-white handwritten digit images of the NIST database (specifically Special Database 3 and Special Database 1). The original NIST images are normalized and resized to fit in a 20x20 pixel area, preserving the aspect ratio. Due to the anti-aliasing technique used to normalize the images, the resulting images are in grayscale format, as seen in figure 3.1. Then, the center of mass of the pixels are computed and the images are translated to position the center point of the images in a larger 28x28 image area, to position the image centrally on the center of mass of the image.

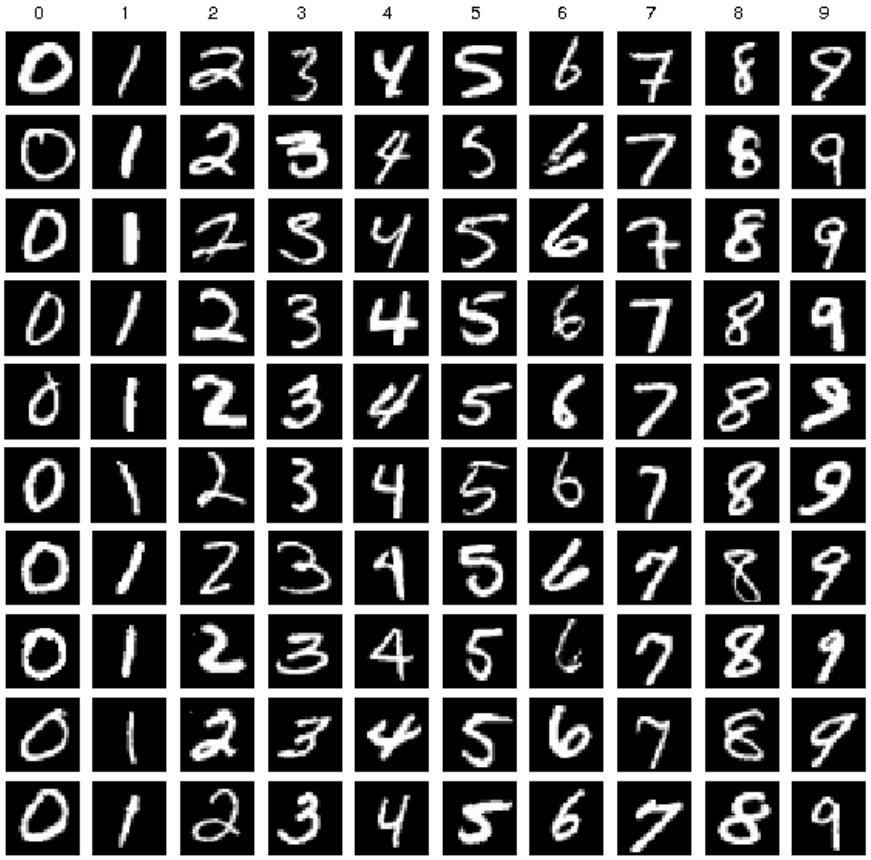

Figure 3.1: Sample images from the MNIST dataset

3.1.2 Fashion-MNIST database

Fashion-MNIST is a database composed of fashion articles from Zalando - an online platform of fashion designers. The dataset has 60,000 training samples and 10,000 test samples. It serves a "**drop-in replacement**" for the MNIST dataset. The Fashion-MNIST database was created because MNIST dataset is very popular and is often used by AL/ML researchers as a benchmark. However, the MNIST dataset can be learnt very easily (convolutional nets can achieve 99.7% on MNIST, and classic machine learning algorithms can also achieve 97% easily).

Each image in a Fashion-MNIST dataset is a 28x28 grayscale image. There are 10 classification labels, from 0 to 9, each label indicative of a type of clothing. A sample of the dataset is shown in figure 3.2.

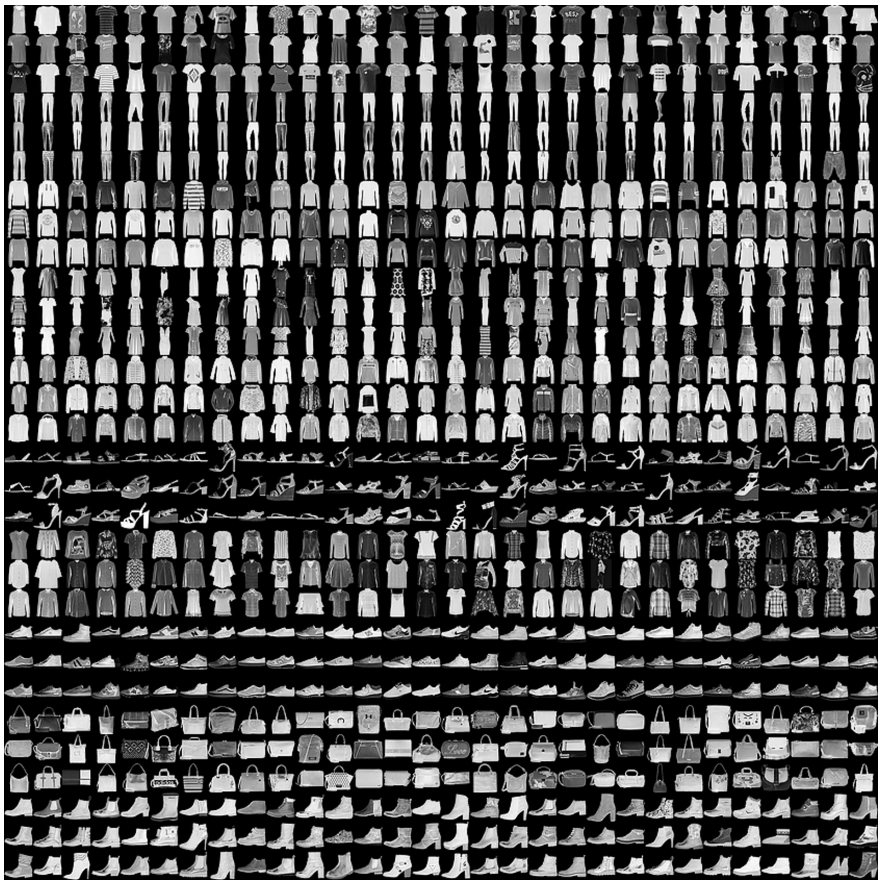

Figure 3.2: Sample images from the Fashion-MNIST dataset

3.1.3 CIFAR10 Database

The CIFAR10 dataset is a collection of images that is very popular for training image classification algorithms. Unlike the previous 2 datasets, the CIFAR10 database contains RGB images. There are a total of 60,000 images: 50,000 training images and 10,000 test images.

Each image is of size 32x32 pixels. The database images are classified into 10 classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. The database contains 6,000 images of each class. The test set has exactly 1,000 randomly chosen images of each class. The training data has 5 batches of 10,000 images each, which are randomly assorted. A sample of the dataset is shown in figure 3.3

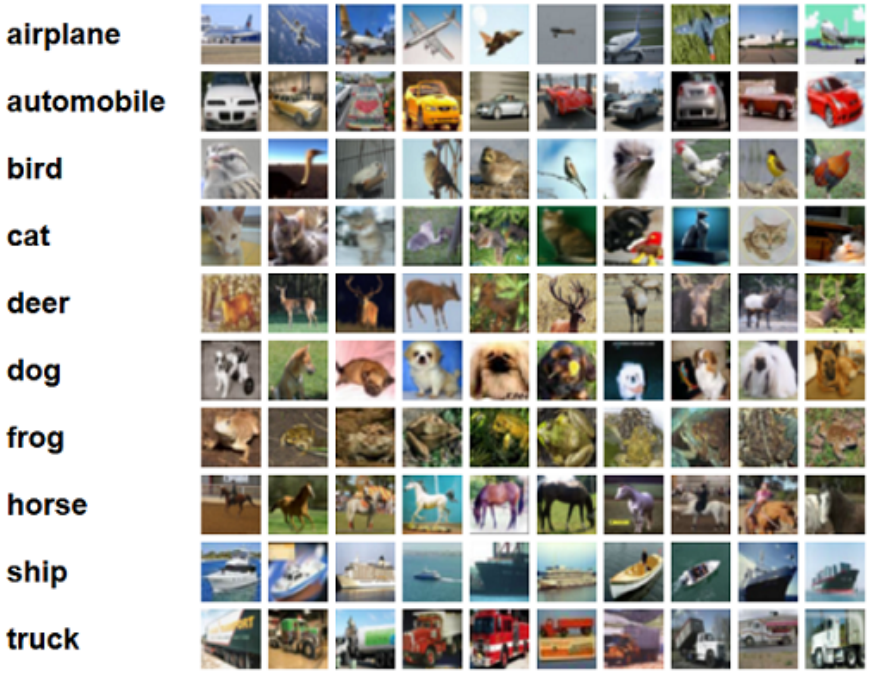

Figure 3.3: Sample images from the CIFAR10 dataset with labels

3.2 Networks

For our experiments, we designed three different types of networks to analyze the performance across varying complexities of neural networks. The three different types of networks we chose are:

- A Simple Dense Neural Network with one fully connected layer hidden layer

- A Convolutional Neural Network with two convolutional layers and one fully connected hidden layer
- A modification of the AlexNet network with three convolutional layers and two fully connected hidden layers

In this section, we briefly discuss each type of network.

3.2.1 Dense network

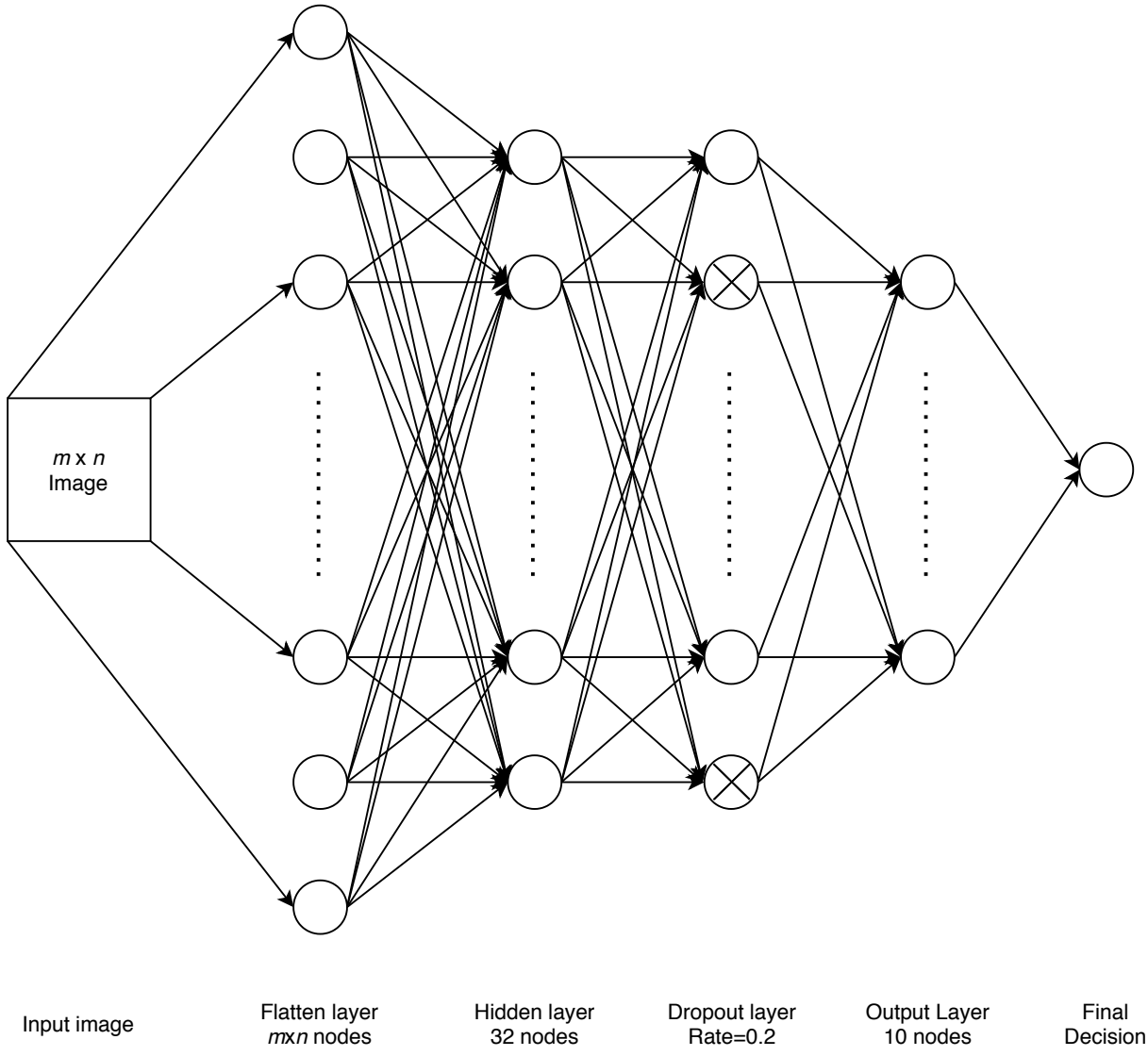

Figure 3.4: Structure of the dense network

Figure 3.4 shows the complete structure of the dense network used to train and predict on the MNIST dataset. It has the following layers:

- **Input image** : The input image is a 28 x 28 grayscale image for MNIST and Fashion-MNIST data sets. For CIFAR10 the input image is a 32 x 32 colour image.

- **Flatten layer** : The purpose of the flatten layer is to convert the image matrix into a vector. In the case of MNIST data set, the 28 x 28 is converted to a vector of 784 values, hence the flatten layer has 784 nodes. For CIFAR10 dataset, since the images are of size 32 x 32, there are 1024 nodes in the flatten layer.

- **Hidden layer** : This is a fully connected layer that performs a linear operation on the input and has a non-linear "ReLU" (Rectified Linear Unit) activation function. All the nodes of the previous layer are connected to all the nodes of the fully connected layer, hence the name "fully connected". For our experiments, this layer is defined to have 32 nodes, since initial experimentation with 32 nodes yielded high accuracy for MNIST and Fashion MNIST datasets. The increase in number of nodes did not have any major effect on the performance of CIFAR10 dataset, and did not train effectively overall.

- **Dropout layer** : The dropout layer is used for regularization and to tackle overfitting. This layer randomly drops neurons with a probability set during the modelling of the network. This layer does not have any trainable parameters and does not learn during backpropagation. For all our experiments with this network, the dropout probability is set at 0.2.

- **Output layer** : This layer is responsible for the final prediction of a test sample. For our experiments, the output layer is activated by a non-linear function ("softmax"). For a classification problem, the number of nodes in the output layer is equal to the number of classes. Thus for MNIST, Fashion-MNIST and CIFAR10, the number of nodes in the output layer is 10.

This is the simplest of the three networks. The reason for choosing to run experiments on a simple network is the idea that if optimization can be possible on a simple network with one fully-connected layer, then a more complex network can be optimized on a fundamental level as well.

3.2.2 Convolutional Neural Network

This network is slightly more complex than the previous one. This network has two convolutional layers followed by a fully connected hidden layer. The layer structure of the network is shown in Figure 3.5 As you can see in the figure, there are two new types of layers that

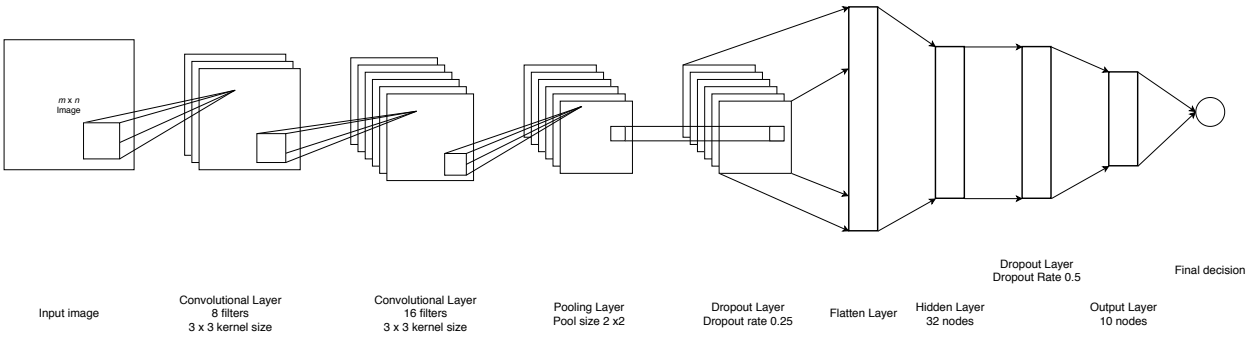

Figure 3.5: Structure of the convolutional neural network

are being used in this network. The following is a discussion of the new layer types:

- **Conv2D layer** : There are two consecutive 2D convolutional layers. The structure is:
  - Convolution layer 1 : 8 output filters, kernel size 3x3. strides of one pixel, **ReLU** activation function. For the first layer, the input shape is defined for the layer to identify the shape of the input image. For MNIST and Fashion-MNIST, the input image shape is 28x28, for the CIFAR data sets, it is 32x32.
  - Convolution layer 2 : 16 output filters, kernel size 3x3, strides of one pixel and ReLU activation function.
- **Pooling layer** : The pooling layer is used to reduce the spatial size, and in turn the number of parameters of the CNN. For our network, the pool size is 2x2 with strides of 2 and it performs the MAX operation. This reduces the number of activations by 75%.

3.2.3 Modified AlexNet

AlexNet is a modified convolutional neural network that performs fast training on large datasets [34]. The original network was created with the ImageNet dataset in mind, which has RGB images of dimension 224 x 224. Since all three of our datasets are considerably

smaller in dimension than the ImageNet dataset, we decided to scale down the network. We experimented and chose a configuration that performs almost as good, somethimes slightly better than the CNN we designed in section 3.2.2. Moreover, the original network has over 60 million learnable parameters, which posed a problem for an enormous training time on our available hardware resource. We implemented a network similar to the AlexNet network, containing 3 convolutional layers and 2 hidden layers. We also changed the number of filters for the convolutional layers and the number of nodes for the dense layer. We reduced the number of parameters from over 60 million to $49,354$. The network contains the following layers:

- Convolutional layer 1 : 16 output filters, kernel size 3x3. strides of one pixel, **ReLU** activation function.
- Pooling layer 1 : Pooling filter size 2x2, strides of 2 pixels.
- Convolutional layer 2 : 32 output filters, kernel size 3x3. strides of one pixel, **ReLU** activation function.
- Pooling layer 2 : Pooling filter size 2x2, strides of 2 pixels.
- Convolutional layer 3 : 64 output filters, kernel size 3x3. strides of one pixel, **ReLU** activation function.
- Pooling layer 3 : Pooling filter size 2x2, strides of 2 pixels.
- Dropout layer 1 : Dropout probability 0.25
- Flatten layer : Converts the 64 2x2 filters in the previous pooling layer into a vector of 256 nodes.
- Dense layer 1 : Fully connected layer of 64 nodes. **tanh** activation function.
- Dropout layer 2 : Dropout probability 0.4.
- Dense layer 2 : Fully connected layer of 128 nodes. **tanh** activation function.
- Dropout layer 3 : Dropout probability 0.4.
- Output layer : **Softmax** activation function. The number of nodes for MNIST, Fashion-MNIST and CIFAR10 data sets are 10 nodes.

3.3 System configuration and hyperparameter selection

For the purposes of clarity and reproducibility, tables 3.1 and 3.2 list the system configuration, along with verisons of softwares and frameworks used, and the hyperparameter values chosen for the experiments.

CPU	8-core 1.7 GHz AMD A10-7850K APU
Memory	16 Gb 1600 MHz RAM
GPU	NVIDIA GeForce 1080Ti 11 Gb
Operating System and version	Ubuntu 16.04.5 LTS
Anaconda version	4.6.8
Python version	3.7.1
TensorFlow version	1.12.0
Keras version	2.2.4

Table 3.1: System configuration and software versions

Batch Size	2000
Number of Epochs	3
Optimizer	`adam`
Loss function	Sparse categorical cross-entropy
Performance metric	Accuracy
Python environment options	`gpu_options.allow_growth = True` `log_device_placement = True`

Table 3.2: Model and environment hyperparameters

The TensorFlow option `config.gpu_options.allow_growth=True` allows for the dynamic growth of allocated memory in the GPU with training. `log_device_placement = True` keeps a system log of which process ran on which device, for traceback if needed.

Chapter 4

Methodology

This chapter is a discussion of the strategy to reduce the floating-point precision of weights for our experiments. In support of the strategy, a discussion of the two Python resources used in our design is given. They are the **callback function** and the "**bitstring**" package. We also provide a discussion of how we used these two packages to reduce the mantissa of the weights. Finally, we also provide a detailed description and algorithm of the four different types of the bit reduction strategies what we experimented with, starting from the broad reduction of the weights of the entire network to a common precision, to a fine grained reduction of mantissa precision based on the value of each weight.

4.1 Floating point reduction of weights

The premise of the experiment is to reduce the number of bits used to store the weights of every layer in a neural network. Most computers natively store real numbers in a floating point format. The floating point representation, as discussed in chapter 2, provides more range and precision than fixed point representation of real numbers, which is advantageous for a lot of applications.

Neural networks, however, do not always need 32 bits of precision for good learning and accuracy. As seen in the works discussed in chapter 2, reduced bit precision can still achieve good prediction accuracy while reducing the size of the model.

In our optimization strategy, we explore reduced floating point precision to reduce the size of the model. This has the additional advantage of reducing the prediction time for the models, since the lesser amount of mantissa bits allow for quicker multiplication and addition.

Now, as we discussed in chapter 2, floating point numbers are stored in the hardware following the IEEE 754 protocol. So it has a sign bit, some exponent bits and mantissa bits. In our strategy, we focus on reducing the number of mantissa bits used to store the weights; for a 32-bit floating point number, the number of mantissa bits is 23, so this approach addresses the largest part of floating point representation. Moreover, a higher number of mantissa bits increase the precision of the float after the radix. Thus, we can reduce the number of mantissa bits without radically changing the number that is stored.

The main objective of our experiments is to show that with a reduction in floating point precision of weights we can achieve a similar, or sometimes even better, prediction accuracy. We have created a simple strategy that gives the desired performance. However, before we explain our strategy, we should discuss two Python resources that have been used to implement floating-point precision reduction. They are the Lambda callback function and the "bitstring" package.

4.1.1 The Lambda callback function

Callback functions are part of the Keras neural network API. A callback (or callback function) is a set of of functions that are executed at certain stages of the training of the neural network. The callback function invocation time can be defined by the programmer, depending on which point of the training the callback should be executed. Callbacks are mainly used to view the internal representations and values at certain stages during the training process. They can also be used to control the training process and check for aberrant values. For example, the callback function:

```
keras.callbacks.callbacks.TerminateOnNaN()
```

is used to terminate the training of a neural network when the loss function generates a NaN. Callbacks can also be used to implement a certain level of control over the training process. Such types of control are often useful and improve the efficiency of the training process or safeguard it from irregular behaviour. For example, the following callback saves the model to disk periodically after every epoch:

```
keras.callbacks.callbacks.ModelCheckpoint(filepath, monitor='val_loss',
   verbose=0, save_best_only=False, save_weights_only=False, mode='auto',
   period=1)
```

and the next callback monitors a certain metric ("*val_loss*") and stops the training process if the observed metric has stopped improving at the minimum improvement rate desired:

```
keras.callbacks.callbacks.EarlyStopping(monitor='val_loss', min_delta=0,
   patience=0, verbose=0, mode='auto', baseline=None, restore_best_weights=
   False)
```

Thus, we can see that the callback functions are a powerful tool for improving the training of neural networks which provides some level of control over the training process. Callback functions can, however, do more than that. The most powerful callback, the one we have used to reduce the floating-point precision, is the LambdaCallback.

The LambdaCallback function is a callback that is used to create and implement custom callbacks on the network during training. According to the Keras documentation, "This callback is constructed with anonymous functions that will be called at the appropriate time." The time of call can be defined by arguments described in table 4.1. Here are two examples of using the LambdaCallback function, where the first one prints the epoch number at the beginning of every epoch and the second one cleans up the run-time by terminating processes after training is complete:

```
# Print the epoch number at the beginning of every epoch.
epoch_print_callback = LambdaCallback(
    on_epoch_begin=lambda epoch,logs: print(batch))
# Terminate some processes after having finished model training.
processes = ...
cleanup_callback = LambdaCallback(
    on_train_end=lambda logs: [
        p.terminate() for p in processes if p.is_alive()])
model.fit(...,
          callbacks=[batch_print_callback,
                     json_logging_callback,
                     cleanup_callback])
```

`on_epoch_begin`	The callback is called at the beginning of every epoch
`on_epoch_end`	The callback is called at the end of every epoch
`on_batch_begin`	The callback is called at the beginning of every batch
`on_batch_end`	The callback is called at the end of every batch
`on_train_begin`	The callback is called at the beginning of the training of the model
`on_train_end`	The callback is called at the end of the training of the model

Table 4.1: Arguments for calling the LambdaCallback function

Thus, LambdaCallback gives us a provision to call and execute custom functions during certain stages of the training. The next thing we will discuss is a Python package that helps us perform the actual mathematical computations for reducing the precision of floating-point numbers.

4.1.2 The `bitstring` Python package

Python does not natively provide an easy way of manipulation of floating point representation at the bit level. Thus, a separate package is used to achieve the level of bit manipulation we require for this experiment. The `bitstring` package is a pure Python module that useful for conversion of any data to strings of binary values, to enable understanding, modifying and tweaking the data in a simple and intuitive manner. Bitstrings are a special type of data type that can be created from converting other data types like integers, hexadecimals, octals, binary, strings or files. Bitstrings can convert both big-endian and little-endian integers as well.

The purpose of `bitstring` is to provide a broad spectrum of operations on binary values that, although possible, can be implemented in Python using `struct` and `array` modules, reduces the time, complexity and lack of efficiency in programming these operations on generic data structures. Some of these operations are:

- Slicing binary values
- Joining two binary numbers
- Reversing binary numbers
- Inserting into a binary number

The above functions are some of the functions that can be performed by the objects of the `BitArray` class defined in the `bitstring` package. There are three other classes defined in the `bitstring` package. They are:

- `BitStream`
- `ConstBitStream`
- `Bits`

For the purposes of our experiments, we have used objects of the `BitStream` class to "pack" or convert the float values of the weights to a stream of binary digits. `BitStream` objects behave similar to a file stream, so operation that are possible on a file stream, such as reading the stream, searching in the stream and navigating to a certain point in the stream, can be performed on a `BitStream` object.

`BitString` objects are can be considered as a simple representation of list of binary digits. However, the storage of `bitstring` is very efficient - only the data is stored bytewise. No additional metadata or information about the objects are stored. All generated outputs are computed and returned as the methods are invoked, and are not stored along with the objects.

Below are some examples of creating some `BitArray` objects and performing some standard operations on them:

```
from bitstring import BitArray

#Creating bitstrings using the BitArray class
# from a binary string
a = BitArray('0b001')
# from a hexadecimal string
b = BitArray('0xff470001')
# straight from a file
c = BitArray(filename='somefile.ext')
# from an integer
d = BitArray(int=540, length=11)

#Standard operations on the bitstring objects
# 5 copies of 'a' followed by two new bytes
```

```
e = 5*a + '0xcdcd'
# put a single bit on the front
e.prepend('0b1')
# take a slice of the first 7 bits
f = e[7:]
# replace 3 bits with 9 bits from octal string
f[1:4] = '0o775'
# find and replace 2 bit string with 16 bit string
f.replace('0b01', '0xee34')
```

Displaying the bitstrings is very easy and flexible as well. Here are a few examples:

```
>>> print(e.hex)
'9249cdcd'
>>> print(e.int)
-1840656947
>>> print(e.uint)
2454310349
```

The `BitStream` class is derived from the `BitArray` class. It is a "mutable container of bits with methods and properties that allow it to be parsed as a stream of bits" [17].

One way of creating a `BitStream` object is by invoking the `pack` function. The signature of the `pack` function is:

```
bitstring.pack(format, *values, **kwargs)
```

where the `format` is specified by a string with comma-separated tokens. The list of tokens that be used as format specifiers is shown in Table 4.2.

Format can also be specified in a shorter fashion. The shorter version specifies the format using two characters: the first character to denote the endianness of the `BitStream` and the second character provides the format. Table 4.3 list the supported characters for specifying the format of `BitStream` objects.

In the following subsection, we will discuss how we can leverage the `bitstring` package to reduce the floating point precision of a number.

int:n	n bits as a signed integer
uint:n	n bits as an unsigned integer
intbe:n	n bits as a big-endian whole byte signed integer
uintbe:n	n bits as a big-endian whole byte unsigned integer
intle:n	n bits as a little-endian whole byte signed integer
uintle:n	n bits as a little-endian whole byte unsigned integer
intne:n	n bits as a native-endian whole byte signed integer
uintne:n	n bits as a native-endian whole byte unsigned integer
float:n	n bits as a big-endian floating point number (same as floatbe)
floatbe:n	n bits as a big-endian floating point number (same as float)
floatle:n	n bits as a little-endian floating point number
floatne:n	n bits as a native-endian floating point number
hex[:n]	[n bits as] a hexadecimal string
oct[:n]	[n bits as] an octal string
bin[:n]	[n bits as] a binary string
bits[:n]	[n bits as] a new bitstring
bool	single bit as a boolean (True or False)
ue	an unsigned integer as an exponential-Golomb code
se	a signed integer as an exponential-Golomb code
uie	an unsigned integer as an interleaved exponential-Golomb code
sie	a signed integer as an interleaved exponential-Golomb code

Table 4.2: List of format specifier tokens for packing

>	Big-endian
<	Little-endian
@	Native-endian

b	8 bit signed integer
B	8 bit unsigned integer
h	16 bit signed integer
H	16 bit unsigned integer
l	32 bit signed integer
L	32 bit unsigned integer
q	64 bit signed integer
Q	64 bit unsigned integer
f	32 bit floating point number
d	64 bit floating point number

Table 4.3: List of endianness (left) and format (right) characters

4.1.3 Reduction of mantissa bits in Python

As we have discussed in the section 2.4, we know how IEEE 754 notation defines floats and how they are stored in the system. We use the utility provided by the `bitstring` package, more specifically `BitStream` objects, to manipulate floating point numbers and reduce the mantissa bits of each value.

The way we do the precision reduction can be described in the following steps:

1. We use the `pack` function to create the `BitStream` object for each weight. The format specifier is '`>f`', which means the `BitStream` is formatted in big-endian manner and the converted number is a 32-bit floating point number.

2. Next we slice the `BitStream` object and retrieve the first bit for the sign, the next 8 bits for exponent and the remaining bits for mantissa.

3. The sign bit is converted into an unsigned integer.

4. The exponent stored in the system is the biased exponent. To get the actual exponent, we subtract $2^{(w\text{-}1)}$ - 1 from the unsigned integer value of the biased exponent, where w is the number of bits used to represent the exponent, which in our case is 8.

5. For the mantissa, a zero before the radix is implied in the notation specified in IEEE-754. So one significand bit is used to represent the leading 1. The remaining 23 bits are used to represent the value after the radix. We remove bits from the end of the mantissa. Bit removal from the mantissa and creating the new mantissa is done by the following steps:

   (a) The mantissa is sliced and the specified number of bits are kept.

   (b) The mantissa is then converted to an unsigned integer

   (c) Since the mantissa is the part after the radix, it is then divided by $2^{\boldsymbol{p}}$ where $\boldsymbol{p}$ is the number of bits retained in the mantissa.

   (d) Finally, we add 1 to the mantissa value to bring the implicit significand bit, as specified in IEEE 754 representation.

6. The new reduced-bit number is then constructed raising 2 to the power of the exponent value and multiplying the result with the mantissa. So the new value is calculated by:

$$newvalue = mantissa \times 2^{exponent}$$

7. If the sign bit is 1, we multiply the result from the previous step by -1.

A pseudo code version of the above procedure of reduction of mantissa bits is presented in Algorithm 1.

Algorithm 1: Reduction of mantissa bits for a float value.

```
input  : Full-precision floating point value
output: Same value with reduced precision (fewer mantissa bits)
def Reduce_Mantissa_Bits(float_number, bits_to_retain):
begin
    bitstring := pack(number := float_number, endian := big, format := 'float32');
    sign_bit := bitstring[1];
    exponent_bits := bitstring[2 to 9];
    mantissa_bits := bitstring[10 to 32];
    sign := unsigned_integer(sign_bit);
    /* a ≪ b indicates left bitshift operation on a by b bits          */
    exponent := unsigned_integer(exponent_bits) −(1 ≪ 7) + 1;
    reduced_mantissa := 1+
       unsigned_integer(mantissa_bits[1 to bits_to_retain]) / (1 ≪ bits_to_retain);
    reduced_precision_float_number := reduced_mantissa × 2^exponent;
    if sign equals 1:
    begin
     |  reduced_precision_float_number := (−1)∗ reduced_precision_float_number
    end
    return reduced_precision_float_number;
end
```

The following is a demonstration of the conversion of a 32-bit floating-point value to a `BitStream` object of binary digits and the integer values of the sign, exponent and mantissa. First we import the package and use the `pack` method to convert the `float` value to a `BitStream` object

```
>>> import bitstring as bstr
>>> x = 12.43567
>>> b = bstr.pack('>f',x)
```

We can see the value of the `BitStream` object in any format. The default format is the hexadecimal format.

```
>>> b
BitStream('0x4146f881')
>>> b.bin
```

```
'01000001010001101111100010000001'
```

Now we can slice the binary representation of the number into the sign, exponent and mantissa portions.

```
>>> sign,exp,mant = b[:1],b[1:9],b[9:]
```

The sign bit should be zero as it is a positive number

```
>>> sign.uint
0
```

The actual exponent value is created by subtracting the exponent bias from the exponent.

```
>>> actual_exp = exp.uint - (1<<7) + 1
>>> actual_exp
3
```

Finally, 1 is added to the actual mantissa value of the `BitStream` representation, divided by 2^p where $\boldsymbol{p}$ is the number of bits in the mantissa.

```
>>> actual_mant = 1 + mant.uint/(1<<23)
>>> actual_mant
1.554458737373352
```

So, the number should be $mantissa * 2^{exponent}$

```
>>> actual_value = actual_mant * (2 ** actual_exp)
>>> actual_value
12.435669898986816
```

Thus the recreated value is similar to the actual value stored by Python. The difference appears after four places after the radix, so the recreated number is 0.000008% off the original number. This change of value while recreating can be attributed to representation error, where a float value cannot be exactly represented by the IEEE 754 32-bit representation, hence Python stores the number in its original form but displays a rounded-off version of it. After we convert the number using `bitstring` and then convert it back, the float value stored back is not the value we entered, but the calculated value from the mantissa and exponent, which is computed by operations on IEEE 754 standards, thus printing out the actual number represented in the IEEE 754 notation. However, for the purposes of our

experiments and research, the change of value is insignificant and we choose to ignore it.

It is possible to reduce the number of bits in the mantissa before converting it into an unsigned integer. For example, the number of bits can be reduced to 15 bits for the mantissa.

```
>>> reduced_mant = 1 + mant[:15].uint/(1<<15)
```

The recreated value in that case can be created in the same manner.

```
>>> reduced_value = reduced_mant * (2 ** actual_exp)
>>> reduced_value
12.435546875
```

The new value is 0.001% off the actual value. Thus, even after reducing the number of mantissa bits, the change in the value is about 10^{-5} times the original value.

The concern that can arise out of this is that Python does not natively operate on reduced precision values: after the reduction of mantissa bits, the value is created by mathematical operations on unsigned integer values. Thus, Python will use 32 bits to store the data. The mathematical operations can, therefore, generate the result in 32 bits. However, the following piece of code verifies that the resultant value from the operations have constrained number of mantissa bits followed by zeros as placeholders to maintain the IEEE-754 notation.

```
>>> r = bstr.pack('>f',reduced_value)
>>> r.bin
'01000001010001101111100000000000'
```

Thus the trailing bits in the mantissa are zeros and can be dropped if the system specifies the exact number of bits for mantissa.

4.2 Mantissa bit reduction strategy

Now that the different Python resources used for bit reduction are defined, this section encompasses the details of the various strategies implemented to reduce the number of mantissa bits for every weight in the neural network.

For our experiments we decided to perform three different strategies for the reduction of mantissa bits for every weight in the network, with increasing levels of granularity and fine-tuning, with respect to the weights. The precision reduction strategies are as follows:

1. **Whole network precision reduction**. Our first approach is is a broad, blanket approach of reducing the precision of all weights in a neural network, across all the layers, to a certain precision level. This approach is inspired from the quantization technique of reducing the precision of the entire network to a common precision [11, 26, 59], but in our case we use floating-point precision to do so. We essentially reduce the number of mantissa bits of the whole network by the same value.

2. **Layer-wise precision reduction**. The next strategy goes one step further in terms of granularity, and sets a layer-by-layer level of precision. We set a precision for each layer of the network with weights, and then reduce the number of mantissa bits of every layer by the set value. In this strategy, each layer can have a different level of precision. This was also inspired from a fixed-point precision reduction technique by Judd et. al., where they experimented on different precision on each layer of a Convnet [31].

3. **In-layer precision reduction**. The next objective was to create a finer level of granularity by trying to introduce different levels of precision within the same network layer. The motivation for this stems from the assumption of varying values of weights of the networks and varying degrees of activity, and significance of these weights in the actual prediction by the model. In-layer precision reduction also brings flexibility in the model training and should reduce computational need due to requirement of lesser precision. Moreover, testing and prediction takes less time, since mathematical calculations require less precision to be evaluated, without losing information as much in quantization or either of the precision reduction strategies mentioned above. One difficulty in doing this is to determine the significance of weights, specifically which weights merit high versus low precision. We perform in-layer precision reduction using two different strategies.

    - **Increasing bucket reduction**. Reducing the number of mantissa bits of every weight based on its value, with higher value weights having higher precision.
    - **Decreasing bucket reduction**. Reducing the number of mantissa bits of every weight based on its value, with lower value weights having higher precision.

The following subsections briefly discuss each strategy and provide pseudo code for each strategy.

4.2.1 Whole network precision reduction

In this strategy, we reduce the mantissa of all the weights in the network to a common precision. We decide upon a bit-width beforehand and reduce the precision of the mantissa of all the weights to the chosen value. At the end of every batch, we perform the following steps:

1. First, we retrieve the layers from the neural network object.
2. Then, for every layer we retrieve the array of weights.
3. We pass each weight and the chosen precision value to the *Reduce_Mantissa_Bits* method defined in Algorithm 1.
4. We replace the full precision weight with the reduced precision weight returned by Algorithm 1.
5. We replace the full precision weight array in the network layers with the reduced precision weight array.

The pseudo code for the above described algorithm is presented in Algorithm 2.

Algorithm 2: Reduction of mantissa bits for the whole network to a single precision value.

```
input  : A neural network
output: A neural network with reduced precision
new_precision := new precision of the network;
def Whole_Network_Precision_Reduction(neural_network, new_precision):
begin
    layers := get_layers(neural_network);
    for layer in range(layers):
    begin
        weight_matrix := get_weights(layer);
        for weight in range(weight_matrix):
        begin
            new_weight := Reduce_Mantissa_Bits(weight, new_precision);
            weight := new_weight;
        end
        set_weights(weight_matrix)
    end
    set_layers(layer)
end
```

The pseudo code for the above described algorithm is presented in Algorithm 2.

4.2.2 Layerwise precision reduction

In this strategy, we choose a different precision for every layer in the network and store it in a list. Mapping precisions to layers, we reduce the precision of the mantissa bits of the weights of every layer to the precision value determined for that layer. At the end of every batch:

1. We retrieve the layers from the neural network object.
2. For every layer we perform the following steps:
   (a) We retrieve the array of weights, along with the precision value determined for that layer, from the list.
   (b) We pass each weight and the precision value to the *Reduce_Mantissa_Bits* method defined in Algorithm 1.
   (c) We replace the full precision weight with the reduced precision weight returned by Algorithm 1.
   (d) We replace the full precision weight array in the network layer with the reduced precision weight array.

The pseudo code for the above described algorithm is presented in Algorithm 3.

4.2.3 Increasing bucket reduction

The objective of this experiment is to create multiple levels of floating point precision for weights in a single layer. Initially we experimented on individual weights and tried to design a network that would be able to set the precision of each individual weight. The approach was too cumbersome and intensive, taking too long to train simple networks. We decided to design a more macro-level approach that is still more selective than a layer-wise approach. Thus, we decided to use the value of the weights to classify them for precision groups, or groups whose weights will have same precision. We tried to simulate an algorithm that would classify the weights into "**buckets**" based on their values, with each bucket having a specified precision for mantissa bits. The precision of the buckets increase with increasing value of the weights. So, lower valued weights in a layer will have lower number of mantissa bits and higher value weights will have a higher number of mantissa bits. During comparison, we consider the absolute unsigned value of weights. This approach seemed a trade-off between

Algorithm 3: Reduction of mantissa bits for every layer in a network to a different precision value.

```
input  : A neural network
output: A neural network with reduced precision
precision_list :=list of precisions for every layer;
def Layerwise_Network_Precision_Reduction(neural_network, precision_list):
begin
    layers := get_layers(neural_network);
    for layer_i in range(layers):
    begin
        new_precision := precision_list[i];
        weight_matrix := get_weights(layer_i);
        for weight in range(weight_matrix):
        begin
            new_weight := Reduce_Mantissa_Bits(weight, new_precision);
            weight := new_weight;
        end
        set_weights(weight_matrix);
        set_layers(layer_i);
    end
end
```

Algorithm 4: Finding the maximum and minimum value in an array.

```
input  : An array of weights
output: Two integer values - the maximum and minimum value in the array
def Find_Max_Min_Values(weight_matrix):
begin
    min := a very large positive value, say +∞;
    max := 0;
    for weight in range(weight_matrix):
    begin
        if absolute_value(weight) < absolute_value(min):
        begin
            min := absolute_value(weight);
        end
        if absolute_value(weight) > absolute_value(min):
        begin
            max := absolute_value(weight);
        end
    end
    return max, min;
end
```

the layer-wise approach and the weight-by-weight precision tuning, which we decided to develop and experiment on.

We have two inputs:

- A neural network
- The increment of bits for successive buckets, as a value. For example, a value of 2 signifies that the precision will increase by 2 bits for every successive bucket. So, the weights in the first bucket will have a floating point precision of 2 bits for its mantissa, the weights in the second bucket will have 4 mantissa bits and so on. The final bucket will have the full precision for the mantissa bits.

At the end of every batch:

1. We retrieve the layers from the neural network object.
2. For every layer we perform the following steps:
   (a) We retrieve the array of weights.
   (b) We find the maximum and minimum absolute values of weights in the weight array, using Algorithm 4.
   (c) We calculate the number of "**buckets**", dividing the number of mantissa bits of the full precision by bit increment value per "**bucket**", which is explained above. We perform all our experiments with the 32 bit floating point standard, which has 23 actual mantissa bits. Since none of the increment values apart from 1 and 23 divides the total mantissa bits perfectly, the last bucket for all such increment bits are set as full precision of 23 bits.
   (d) We calculate boundary values of every bucket.
   (e) For every weight,
      i. We try to find out the bucket they belong to, by comparing the value to the bucket boundaries, starting with the second boundary and moving towards the maximum value bucket boundary. The first boundary is the minimum weight value for that layer, so we ignore it, since no weight is smaller than the minimum weight.

ii. When we find the bucket the weight belongs to (a weight belongs to a certain bucket if its absolute value is within the lower and upper ranges of the bucket), we pass the weight along with the precision value for the bucket to the *Reduce_Mantissa_Bits* method defined in Algorithm 1. The precision value of the bucket is easily determined by multiplying the bucket number with the bucket increment value.

iii. If the weight belongs to the final bucket, we do not perform any precision reduction, as the final bucket weights have full precision.

iv. We replace the full precision weight with the reduced precision weight returned by Algorithm 1.

3. We replace the full precision weight array in the network layer with the reduced precision weight array.

The pseudo code for the above described algorithm is presented in Algorithm 5.

4.2.4 Decreasing bucket reduction

This strategy is exactly similar to the previous one, except for one change. The ordering of precision in the buckets is reversed. Hence, the first bucket (containing the smallest weights) will have full precision for its mantissa bits, and the highest value weights will have the lowest number of mantissa bits. The reason for simulating this strategy is because even though smaller-valued weights might have a lesser impact on the evaluation phase of a model, the mantissa of smaller-value weights would contain more information and "value" than the larger value weights.

The above thought can be easily explained with an example. If a weight changes from 0.051 to 0.05 due to precision reduction, its value is changed by 2%, whereas a larger weight, say 128.051, gets reduced by the same mantissa precision reduction to, say, 128, then the loss/change of value is 0.3%, which has a significantly lower impact compared to the smaller weights. Thus, this thought justified the planning and simulation of this strategy.

Similar to the previous strategy, we have identical inputs:

- A neural network
- The number of bits reduced per bucket

Algorithm 5: Reduction of mantissa bits using bucketing with increasing number of bits for **increasing** value of weights.

```
input : A neural network
output: A neural network with reduced precision
bits_per_bucket := difference/increase of bits between consecutive buckets;
def Bucketed_Weights_Increasing_Precision(neural_network, bits_per_bucket):
begin
    layers := get_layers(neural_network);
    for layer_i in range(layers):
    begin
        weight_matrix := get_weights(layer_i);
        max, min := Find_Max_Min_Values(weight_matrix);
        number_of_buckets := ceiling(23 ÷ bits_per_bucket);
        increment := (max − min) ÷ number_of_buckets;
        bucket_boundaries[1] := min;
        for i = 2 to (number_of_buckets + 1):
        begin
            bucket_boundaries[i] = min + ((i − 1) × increment)
        end
        for weight in range(weight_matrix):
        begin
            for bucket = 2 to (number_of_buckets + 1):
            begin
                if absolute_value(weight) ≤ bucket_boundaries[bucket]:
                begin
                    if bucket equals (number_of_buckets + 1):
                    begin
                        continue;
                    end
                    else:
                    begin
                        new_weight := Reduce_Mantissa_Bits(weight,
                          (bucket − 1) × bits_per_bucket);
                        weight := new_weight;
                    end
                end
            end
        end
        set_weights(weight_matrix);
        set_layers(layer_i);
    end
end
```

Note that in this case the value for bits for successive buckets is a reduction instead of an increment. So, a value of 2 signifies that the precision will decrease by 2 bits for every successive bucket. Thus, the weights in the last bucket will have a floating point precision of 2 bits for its mantissa, the weights in the penultimate bucket will have 4 mantissa bits and so on. The first bucket will have the full precision for the mantissa bits.

The algorithm works in a similar fashion as the previous one. At the end of every batch:

1. We retrieve the layers from the neural network object.
2. For every layer we perform the following steps:
   (a) We retrieve the array of weights
   (b) We find the maximum and minimum absolute values of weights in the weight array, using Algorithm 4.
   (c) We calculate the number of "**buckets**", dividing the number of mantissa bits of the full precision by bit increment value per "**bucket**", which is explained above. We perform all our experiments with the 32 bit floating point standard, which has 23 actual mantissa bits.
   (d) We calculate boundary values of every bucket.
   (e) For every weight,
      i. We try to find out the bucket they belong to, by comparing the value to the bucket boundaries, starting with the second boundary and moving towards the maximum value bucket boundary. The first boundary is the minimum weight value for that layer, so we ignore it, since no weight is smaller than the minimum weight.
      ii. If the weight belongs to the first bucket, we do not perform any precision reduction, as the first bucket weights have full precision.
      iii. When we find the bucket the weight belongs to (a weight belongs to a certain bucket if its absolute value is less than the upper boundary of the bucket), we pass the weight along with the precision value for the bucket to the *Reduce_Mantissa_Bits* method defined in Algorithm 1. The precision value of the bucket is determined by multiplying the bucket number with $(total\_buckets + 1) - bucket\_number$.

iv. We replace the full precision weight with the reduced precision weight returned by Algorithm 1.

3. We replace the full precision weight array in the network layer with the reduced precision weight array.

The pseudo code for the above described algorithm is presented in Algorithm 6.

4.2.5 Other algorithms for benchmarks

As a benchmark to compare the four strategies mentioned above, we used two other model training algorithms.

The first one is the training of the neural network at full precision without any optimization.

The second one is to randomly select the number of mantissa bits to keep for every weight in the network. This helps us compare our results with random floating point precision and to validate whether the result of the other strategies is similar to pure chance or not. The random strategy works exactly the same way as the bucketing strategy, but instead of segregating weights into buckets, we randomly generate a number between 1 and the full precision value, and pass that as the precision, along with the weight, to *Reduce_Mantissa_Bits*.

Algorithm 6: Reduction of mantissa bits using bucketing with increasing number of bits for **decreasing** value of weights.

```
input : A neural network
output: A neural network with reduced precision
bits_per_bucket := difference/increase of bits between consecutive buckets;
def Bucketed_Weights_Increasing_Precision(neural_network, bits_per_bucket):
begin
    layers := get_layers(neural_network);
    for layer_i in range(layers):
    begin
        weight_matrix := get_weights(layer_i);
        max, min := Find_Max_Min_Values(weight_matrix);
        number_of_buckets := ceiling(23 ÷ bits_per_bucket);
        increment := (max − min) ÷ number_of_buckets;
        bucket_boundaries[1] := min;
        for i = 2 to (number_of_buckets + 1):
        begin
            bucket_boundaries[i] = min + ((i − 1) × increment)
        end
        for weight in range(weight_matrix):
        begin
            for bucket = 2 to (number_of_buckets + 1):
            begin
                if absolute_value(weight) ≤ bucket_boundaries[bucket]:
                begin
                    if bucket equals 2:
                    begin
                        continue;
                    end
                    else:
                    begin
                        new_weight := Reduce_Mantissa_Bits(weight,
                          ((number_of_buckets + 2) − bucket) × bits_per_bucket);
                        weight := new_weight;
                    end
                end
            end
        end
        set_weights(weight_matrix);
        set_layers(layer_i);
    end
end
```

Chapter 5

Results

This chapter displays the results obtained from the experiments in the previous chapter, and provides an analytical interpretation of those results. The results are sectioned by precision reduction strategy, with each section presenting the results of every data set and network. The results are presented as plots. Finally, the observations are analyzed and a brief discussion of the results is provided at the end of the chapter.

5.1 Whole network precision reduction

At first, we trained the simple dense network explained in section 3.2.1, on the MNIST dataset. Since there are 23 bits in the mantissa, running experiments for every variation in mantissa size was not feasible. We selected three values: 5, 10 and 20 mantissa bits to retain after precision reduction and report the results. Figure 5.1 shows the training accuracy of the whole network precision reduction of the dense network on the MNIST data set. The training curve of the unoptimized (full precision) and random precision reduction, both explained briefly in section 4.2.5, are provided in the same graph for comparison and benchmarking purposes. Running the experiment, a few things are noticeable from the graph:

- The results of the graph show what was expected: the training accuracy of the network increases overall with a greater number of bits preserved. In particular, we observe that the network with 20 bits has a higher training accuracy than the 5-bit network.
- The 5-bit model is very close to the randomly reduced precision model.

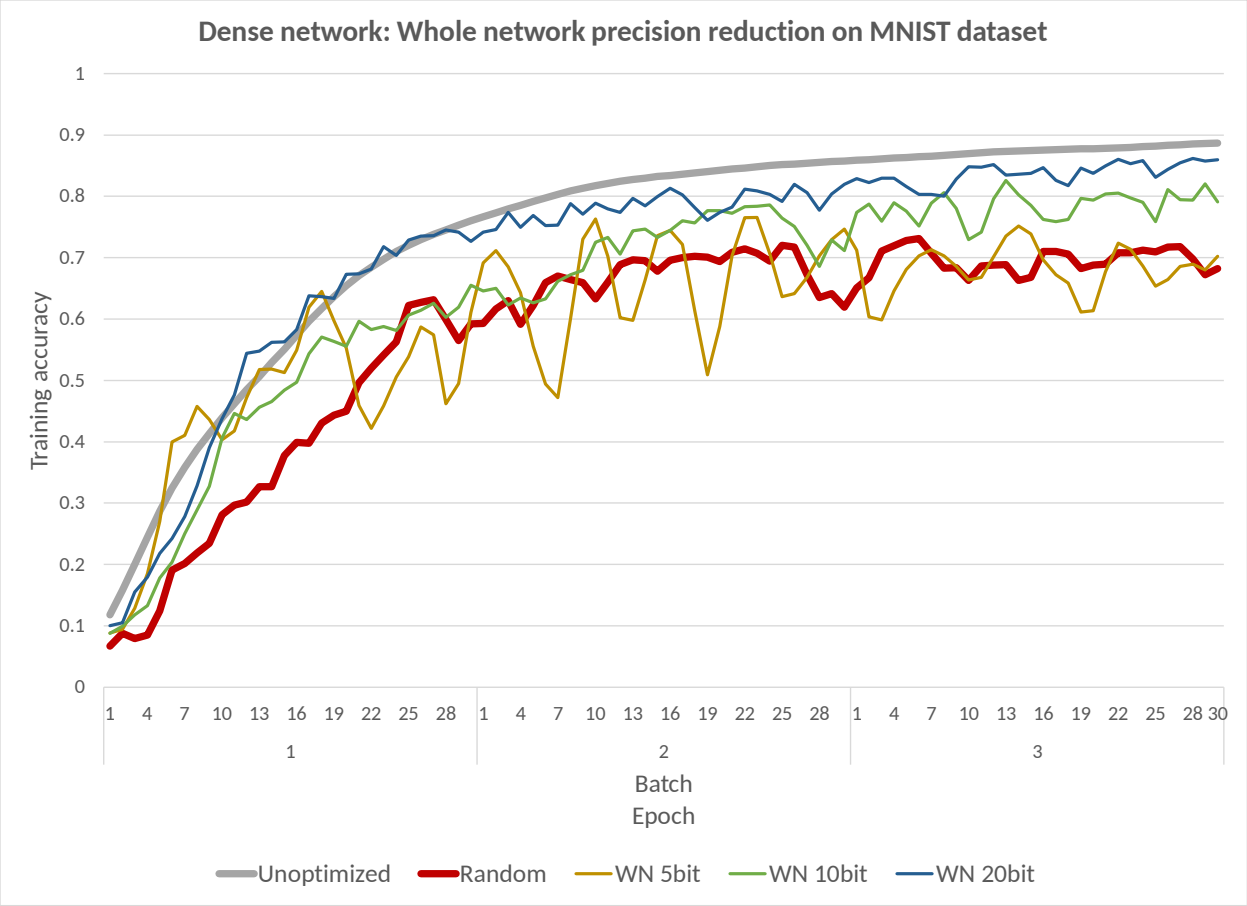

Figure 5.1: Whole network precision reduction of dense network on the MNIST data set

- The 5-bit precision network is notably less smooth than the 10-bit or 20-bit models. This tells us that the model is losing too much information from precision reduction at the end of every batch.
- The 20-bit precision model training is very close to the full precision model training. This is in line with our idea that mantissa bits can be reduced to achieve similar accuracy to full precision models.

One thing to note from the result in figure 5.1 is that the simple dense network learns the MNIST dataset very efficiently. Thus, the MNIST dataset being too simple to provide a challenge to the neural network, we experiment on two more datasets: the Fashion MNIST dataset and the CIFAR10 dataset, both of which are described in detail in chapter 3. We also create two more neural network models: the ConvNet and the modified AlexNet, to train and experiment on. Again, the details of the networks are discussed in chapter 3. Figure 5.2 and

5.3 show the training accuracy of the models on the three datasets over three epochs. The training accuracy is calculated at the end of every batch after precision reduction. Each of the graphs have the unoptimized full-precision training accuracy and the randomly selected precision training accuracy for benchmarking and comparison.

We did not report our results obtained on training the multilayer perceptron dense network on the CIFAR10 dataset because even the full precision accuracy reached only a maximum of 15%, which was too low for any meaningful analysis and interpretation. The reason for such a low training accuracy score is that the model is too simple to learn from the CIFAR10 dataset efficiently. Thus we omitted those results from our findings.

Table 5.1 provides a short explanation of all the legends that are used in the graph.

Legend entry	**Explanation**
WN $\boldsymbol{x}$bit	Whole network precision reduction explained in section 4.2.1 where $\boldsymbol{x}$ is the number of bits retained in the mantissa of the weights
LN $\boldsymbol{x}$H$\boldsymbol{y}$O	Layerwise precision reduction, explained in section 4.2.2, for the multilayer perceptron dense network, where $\boldsymbol{x}$ is the number of bits retained in the mantissa of the weights in the hidden layer and $\boldsymbol{y}$ is the number of bits retained in the mantissa of the weights in the output layer
LN $\boldsymbol{x}$C$\boldsymbol{y}$D	Layerwise precision reduction, explained in section 4.2.2, for the CNN, where $\boldsymbol{x}$ is the number of bits retained in the mantissa of the weights in the convolutional layers and $\boldsymbol{y}$ is the number of bits retained in the mantissa of the weights in the dense layers
BUCKF $\boldsymbol{x}$bit	In-layer increasing bucketing precision reduction, explained in section 4.2.3 where $\boldsymbol{x}$ is the increase in the number of bits for every bucket, starting from $\boldsymbol{x}$ bits in the first bucket to a maximum precision of 23 bits (full precision) in the final bucket
BUCKR $\boldsymbol{x}$bit	In-layer decreasing bucketing precision reduction, explained in section 4.2.4 where $\boldsymbol{x}$ is the decrease in the number of bits for every bucket, starting from 23 bits (full precision) in the first bucket to a minimum precision of $\boldsymbol{x}$ bits in the final bucket
Unoptimized	Full precision network, without any reduction of mantissa bits
Random	Randomly selected precision reduction, explained in section 4.2.5, where the precision of every weight is randomly selected between 1 and 23 bits.

Table 5.1: Explanation of legends in the graphs

We notice from these other experiments that there are a few key trends that can be observed in all of these experiments:

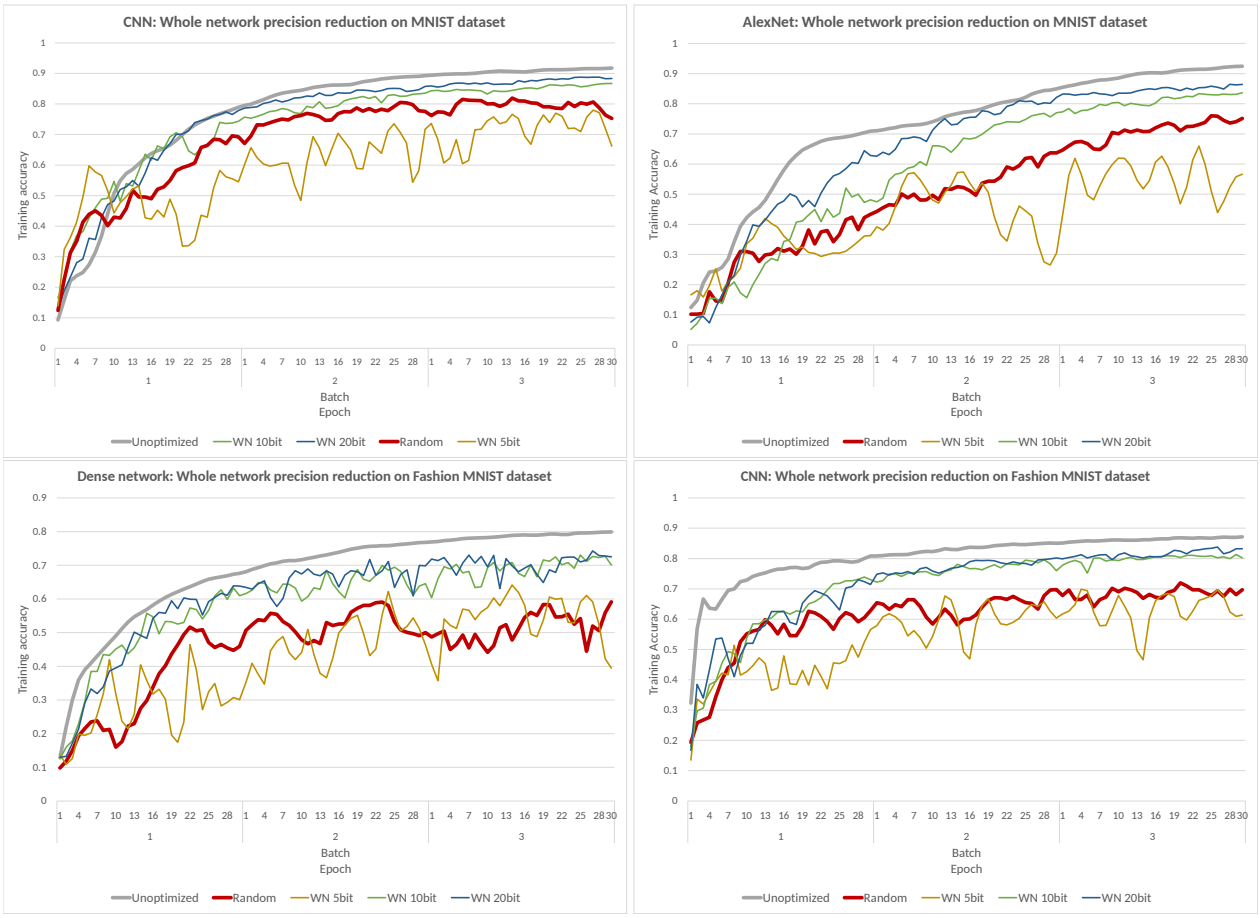

Figure 5.2: Whole network precision reduction on all the different datasets and networks

- The model with 5 bits of precision for all its weights perform very poorly and is very close to randomly selecting precision for the weights. Thus, significant information is lost by reducing the precision of the weights too much.
- Moreover, the jagged nature of the learning curve tells us that information retained after precision reduction for these models is inconsistent and prevents the model from learning effectively.
- The 10-bit and 20-bit precision models seem to perform much better than the 5-bit one. The learning curves of these models are very close to the actual full-precision ones in most cases. The 20-bit precision model has a slightly higher accuracy than the 10-bit model at the end of training in almost all of the cases.
- The 10-bit and 20-bit model learning curves are also smoother than the 5-bit precision model. This shows that the training accuracy of the models at the end of every batch is

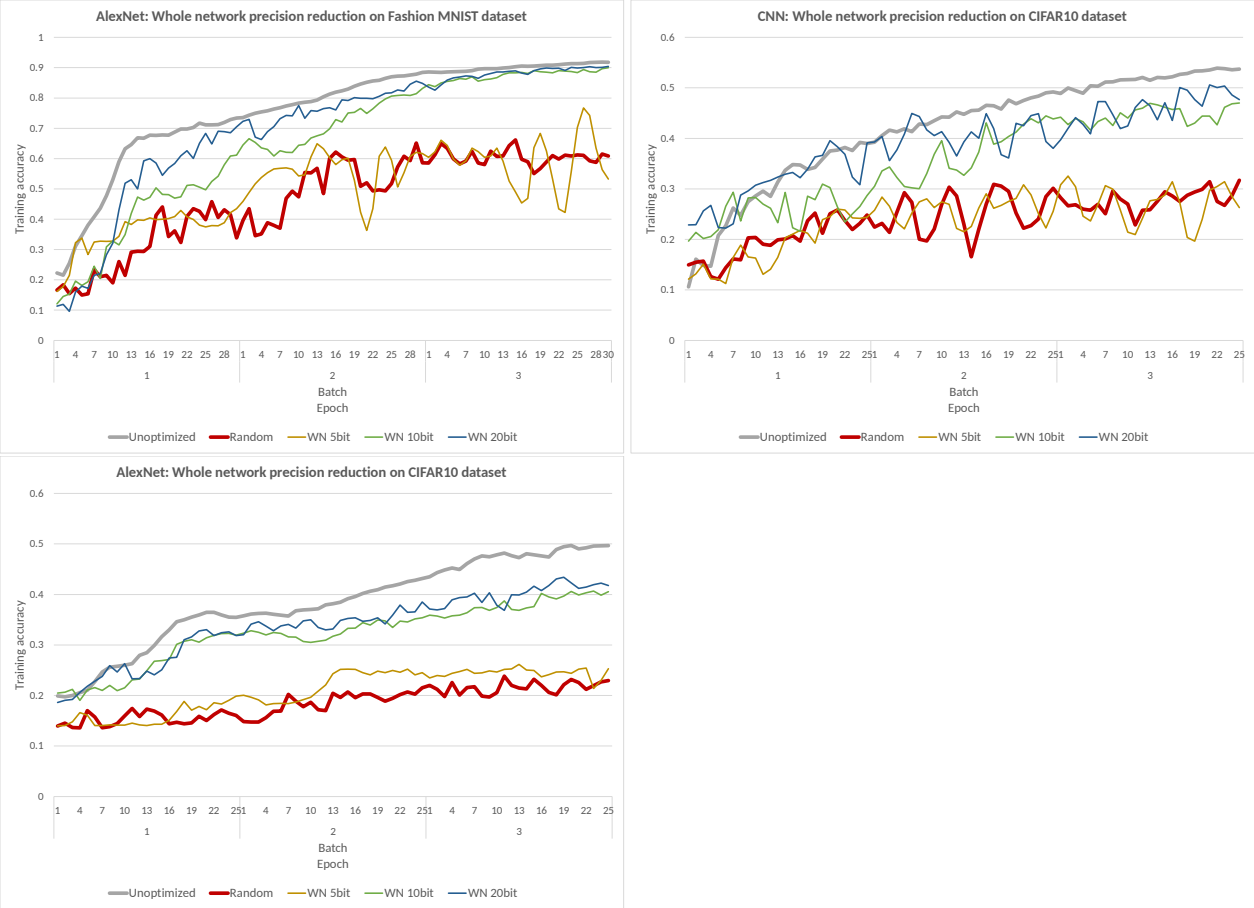

Figure 5.3: Whole network precision reduction on all the different datasets and networks

more consistent and does not lose too much information to reduce the training accuracy significantly.

5.2 Variance analysis

One major concern that can arise out of these results is that whether these results are a matter of chance and coincidence of good training dataset sampling, proper initial randomization of weights and batch split of the training data. In order to mitigate the effect of chance in our experiments, we ran each of the experiment three times on three different samplings of the training dataset and reported the average of the three results. There are a total number of 8 "network-dataset" combinations on which we ran our experiments. On each of these combinations, considering the different settings of hyperparameters, we ran 17 different experiments, each three times. Thus, a total of 120 individual sets of data are

generated for variance analysis. Upon observing the results, we did not see any significant variation between runs. Because of the high volume of data and visualizations, and the fact that these plots of individual passes for every strategy on every network and model provides no new information, we do not report them in our results.

However, just as an example of how varied each pass can be from the average, we report one precision reduction strategy: the layer-wise precision reduction strategy of CNN on the MNIST dataset in figure 5.4. Each of the colored lines show the training accuracies of an individual pass, whereas the thicker grey line shows the average of the three passes. Refer to table 5.1 for an explanation of the legend.

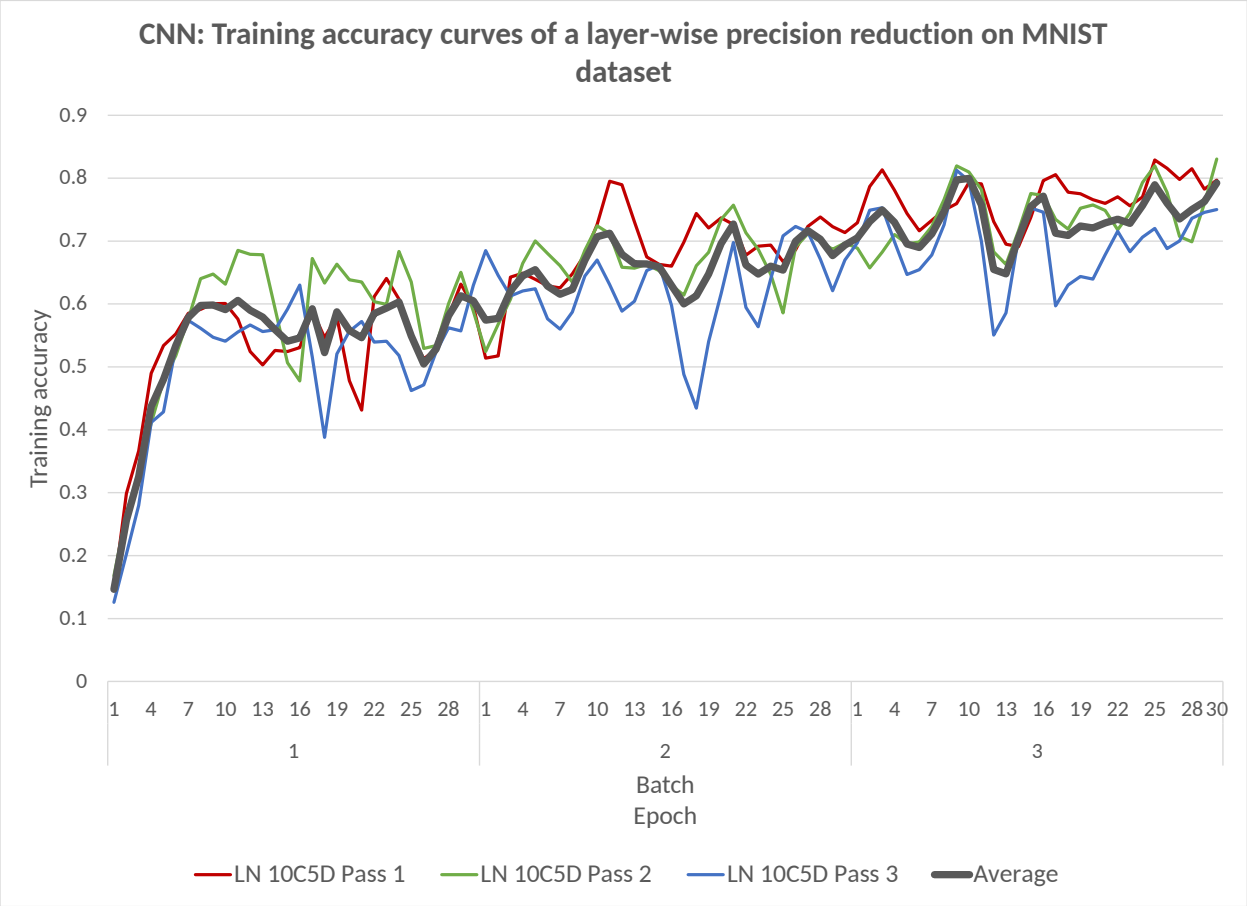

Figure 5.4: Training curves of one layer-wise precision reduction strategy of the CNN on the MNIST dataset

As we see from the plot, the each learning curve differs from one another. Sometimes the difference is insignificant, whereas other times, it is quite large. This is because of the initial values of the weights and the sampling of the dataset. Thus, to mitigate the variance

introduced by the randomness, we decided to run all experiments three times and consider the average. Given the high number of repeated experiments, we decide not to run an experiment more than three times, as it would be infeasible, given our resources and time. This limits our ability to use statistical techniques that depend on having a large number of experiments. However, our objective here is not to analyze the training accuracy and learning of each strategy at every batch, but to analyse the overall training efficiency and learning trend in relation to the full precision model. Thus, the overall trend and relativity of the training curves to the full precision training is the focus of our results.

5.3 Layerwise precision reduction strategy

Following the results we observed on the whole network precision reduction, we ran our experiments to reduce the layers of each network to different levels of precision. Similar to what we mentioned in section 5.1, the permutations of precision for even a two layer dense network is too large to run and report the results for. For this reason we chose the values we selected for the whole network precision reduction: 5, 10 and 20 bits, and cycled around those values as precision for the layers of the network. Figures 5.5 and 5.6 show the training accuracy curves of the layerwise precision reduction of the datasets and networks we ran the experiment on.

We note a few key observations from the results:

- The CNN and AlexNet models with 5 bits of precision in the dense layers train significantly worse than the other precision values for these models, even when there are 20 bits of precision in the convolutional layers. It is similar, sometimes worse, in performance to the random precision of weights. Moreover, as we saw in section 5.1, the training accuracy is inconsistent and swings back and forth, implying that the loss of information in the dense layers is significant to reduce the training capability and performance of these models.

- The CNN and AlexNet models with 20-bit precision in the dense layers have the highest training accuracy overall, and in most cases, is very similar to the training curve of the full precision model.

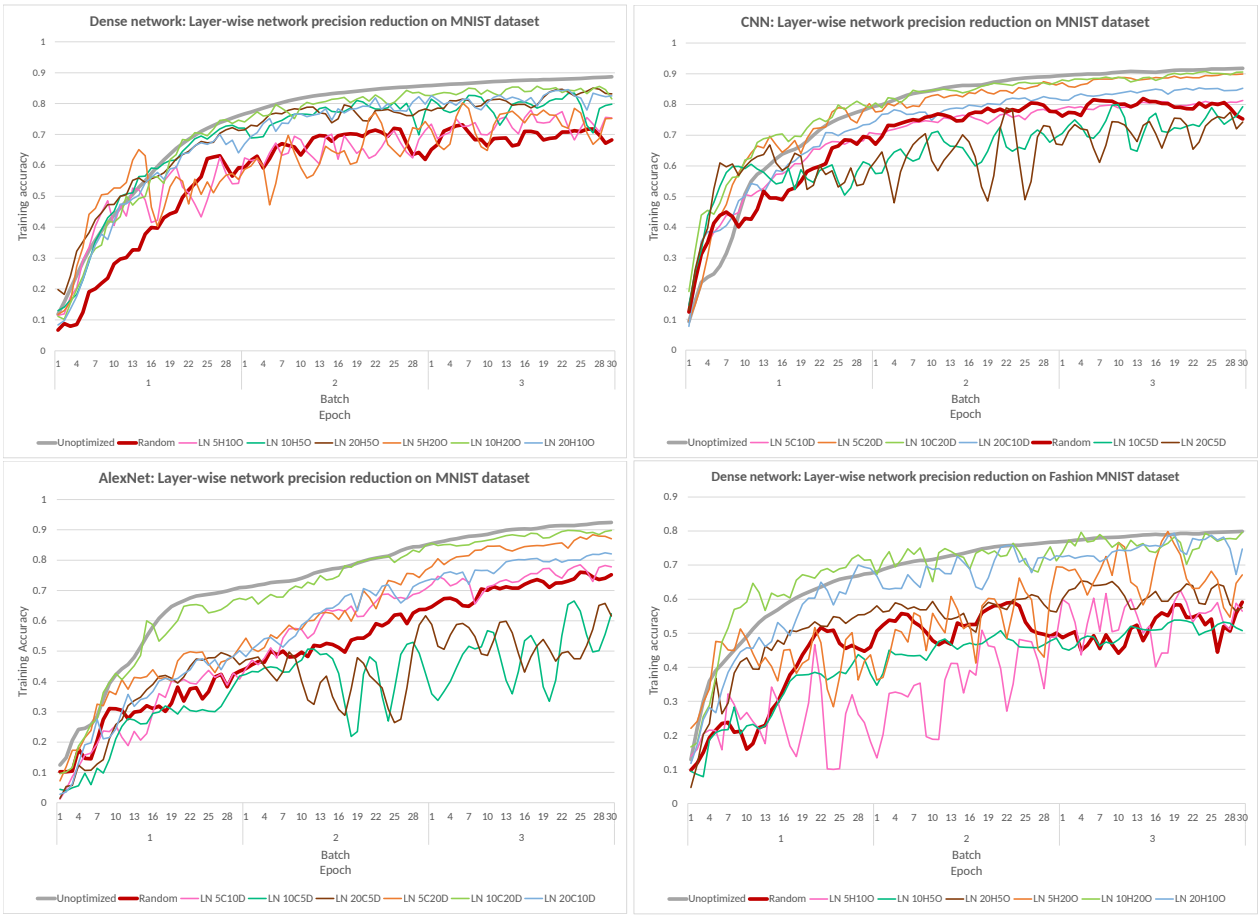

Figure 5.5: Layerwise precision reduction on all the different datasets and networks

- The model with the highest overall training accuracy and learning closest to the full precision one is the one with 20-bit precision dense layers and 10-bit precision convolutional layers, followed by the one with 20-bit dense layers and 5-bit convolutional layers. This further suggests that the precision of the dense layers is more significant, in achieving better training accuracy, than the convolutional layers.

- The models with 10-bit precision dense layers had a higher training accuracy and learning curve than the 5-bit precision dense layer models, but had a lower training performance than the 20-bit precision dense layer models, with the 10-bit convolutional layer model having a better training accuracy than the 5-bit precision convolutional layer model. This suggests that the higher precision of convolutional layers impacts the learning of the model, but not as much as the precision of the dense layers.

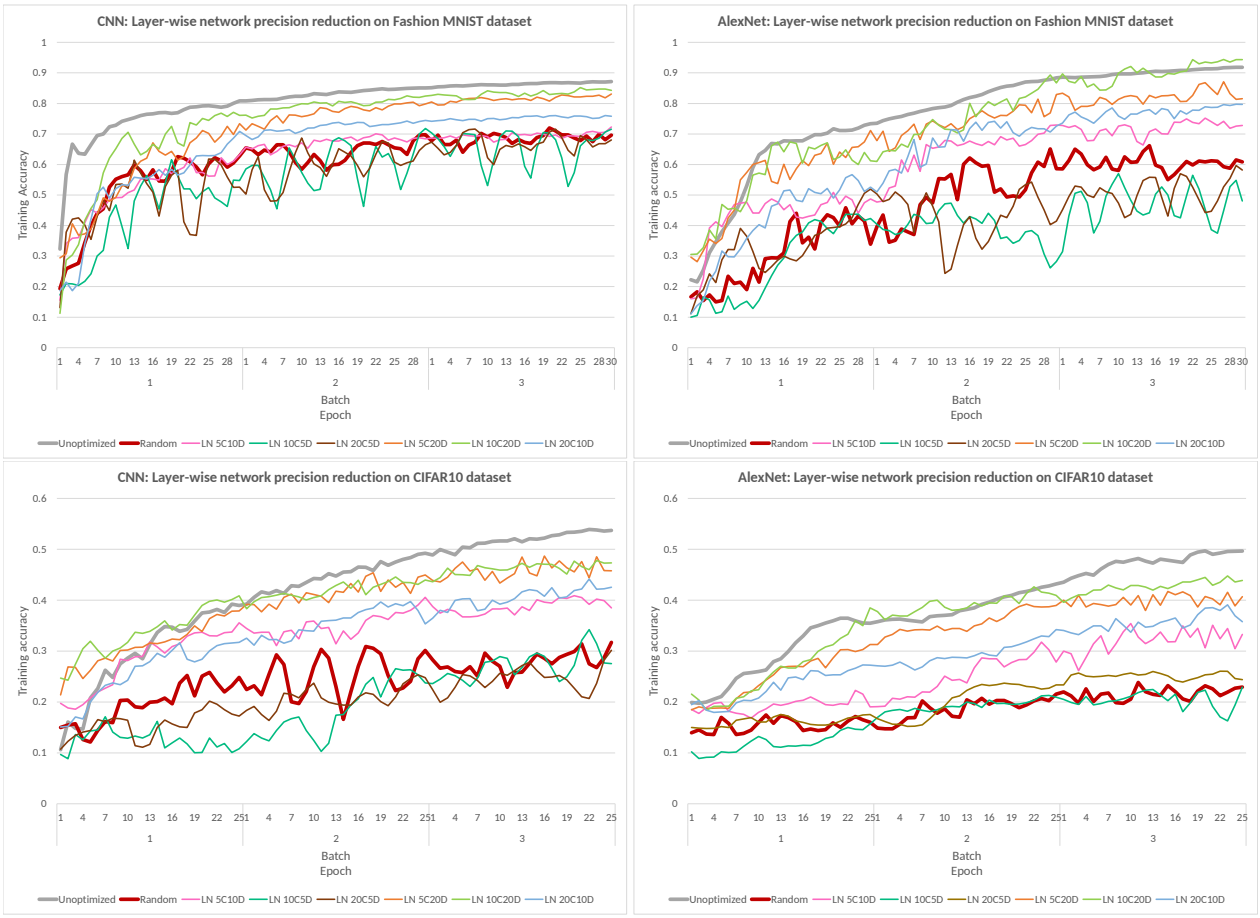

Figure 5.6: Layerwise precision reduction on all the different datasets and networks

- The dense networks show similar trends to the CNN and AlexNet precision models, although the effect of information loss due to reducing mantissa bits is more pronounced due to the sharp ups and downs in the learning curves. Looking at the plots, it seems that the effect of precision reduction in hidden layers impacts more than the output layers. All the models with 5-bit precision hidden layers perform worse and have more inconsistencies in learning than the 10-bit or higher precision hidden layer models. Also, as expected, higher precision output layer models show better training accuracies and learning.

- The MNIST dataset, as we speculated in section 5.1, seems to be easier to learn, as the effect of precision reduction on the learning is less pronounced than the learning on the Fashion MNIST or CIFAR10 dataset, as evident by the fact that the training

curves of the models are not as separated from each other than in the case of other datasets.

We notice that the layerwise precision reduction strategy trains as well as the whole network precision reduction strategy, while reducing a higher number of bits from the mantissa of the weights. This strategy is fairly similar to the layerwise quantization techniques that are developed recently [8], with the difference of this being a floating point precision standard. Layerwise precision reduction can be implemented in most cases where there is a lack of sufficient storage space (such as edge devices), or when there is identifiable gains for training layers of a neural network at different precisions, such as training a network over distributed systems with some nodes having less power or storage than others.

5.4 Increasing bucket reduction

After the layerwise precision reduction results, the same model configurations are trained on the same datasets, with the bucketing strategies. First, we train with the increasing ("forward") precision with increasing weight value. The increasing bucket reduction strategy is explained in detail in section 4.2.3. Since the permutations of the width of the bucket and how many bits to keep in every bucket are too large for us to execute all the combinations and report the results, we decided to experiment on three specified bucketing strategies:

- 5 bits: Precision in every bucket increases by 5 bits. Since the full precision is 23 bits of mantissa, we have 5 buckets, with 5, 10, 15, 20 and 23 bits as precision for these buckets.

- 2 bits: Precision in every bucket increases by 2 bits. Since the full precision is 23 bits of mantissa, we have 12 buckets, with 2 bits for the first bucket, 4 for the second one, up to 22 for the penultimate one. The final bucket has 23 bits of precision for the mantissa.

- 1 bit: Precision in every bucket increases by 1 bit. Since the full precision is 23 bits of mantissa, we have 23 buckets, with the first bucket having 1 bit of precision and the final bucket having 23 bits of precision.

For the increasing precision reduction, the smallest weights are put in the first bucket with the lowest precision and higher weights going in the next buckets with increasing precision.

Detailed explanation of the strategy of deciding buckets and boundaries of buckets are explained in section 4.2.3. One thing to note is that for comparison of weights and sorting them into buckets, we considered the absolute unsigned value of the weights. Thus, a high negative weight, say -12 is still considered a higher value weight than, say 0.5.

Figures 5.7 and 5.8 show the training accuracy curves for the increasing (forward) bucketing precision reduction strategies.

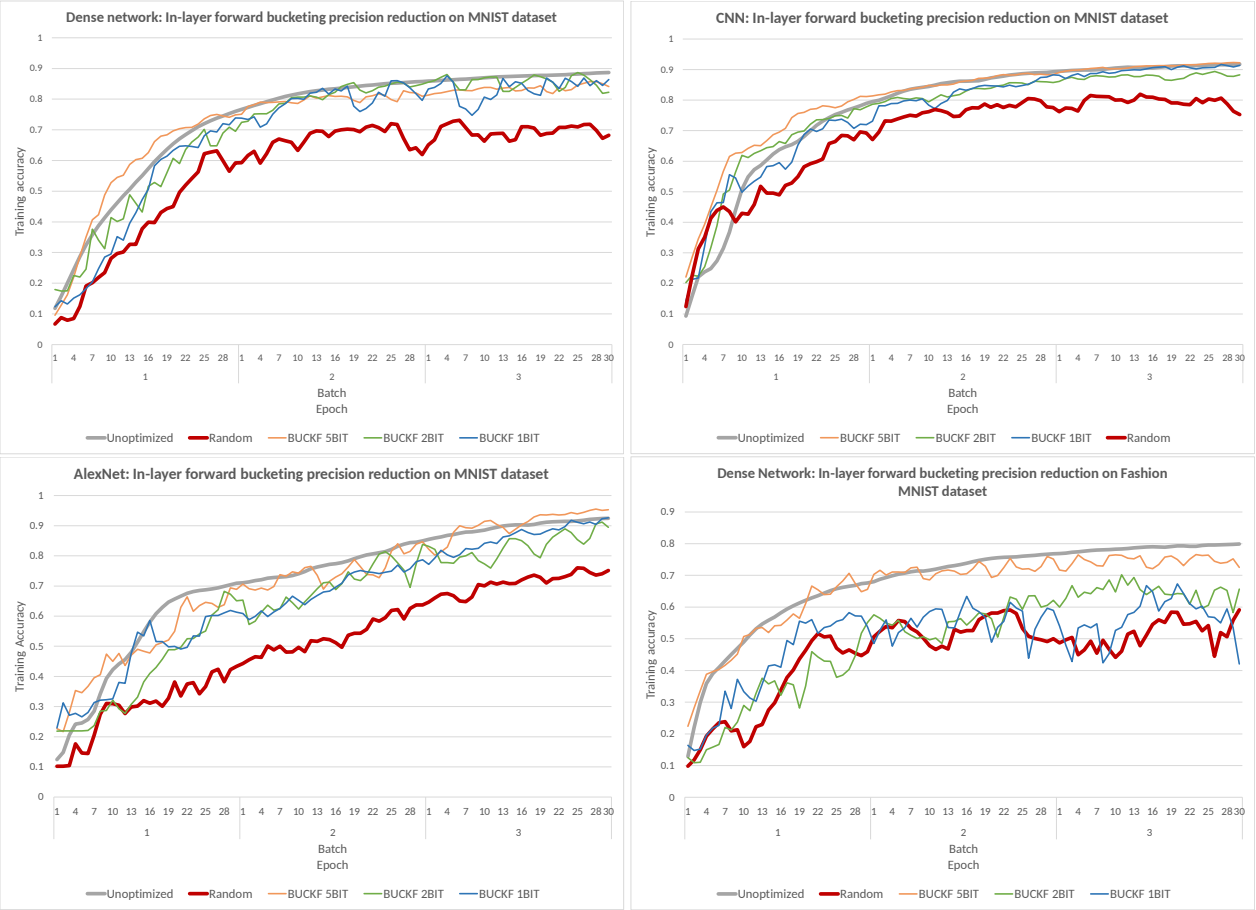

Figure 5.7: In-layer forward bucketing precision reduction on all the different datasets and networks

One important thing of note here is that, over an average of 3 iterations, the 5 bit forward bucketing strategy learned the best. It had a higher average overall training accuracy than the other bucketing schemes, matching or surpassing the full precision model training accuracy in some cases. This is perhaps because given the lesser number of buckets, the higher valued weights had more precision than the 2 bit and 1 bit reduction strategies.

To better understand this, consider figure 5.9. The graphs display the percentage of

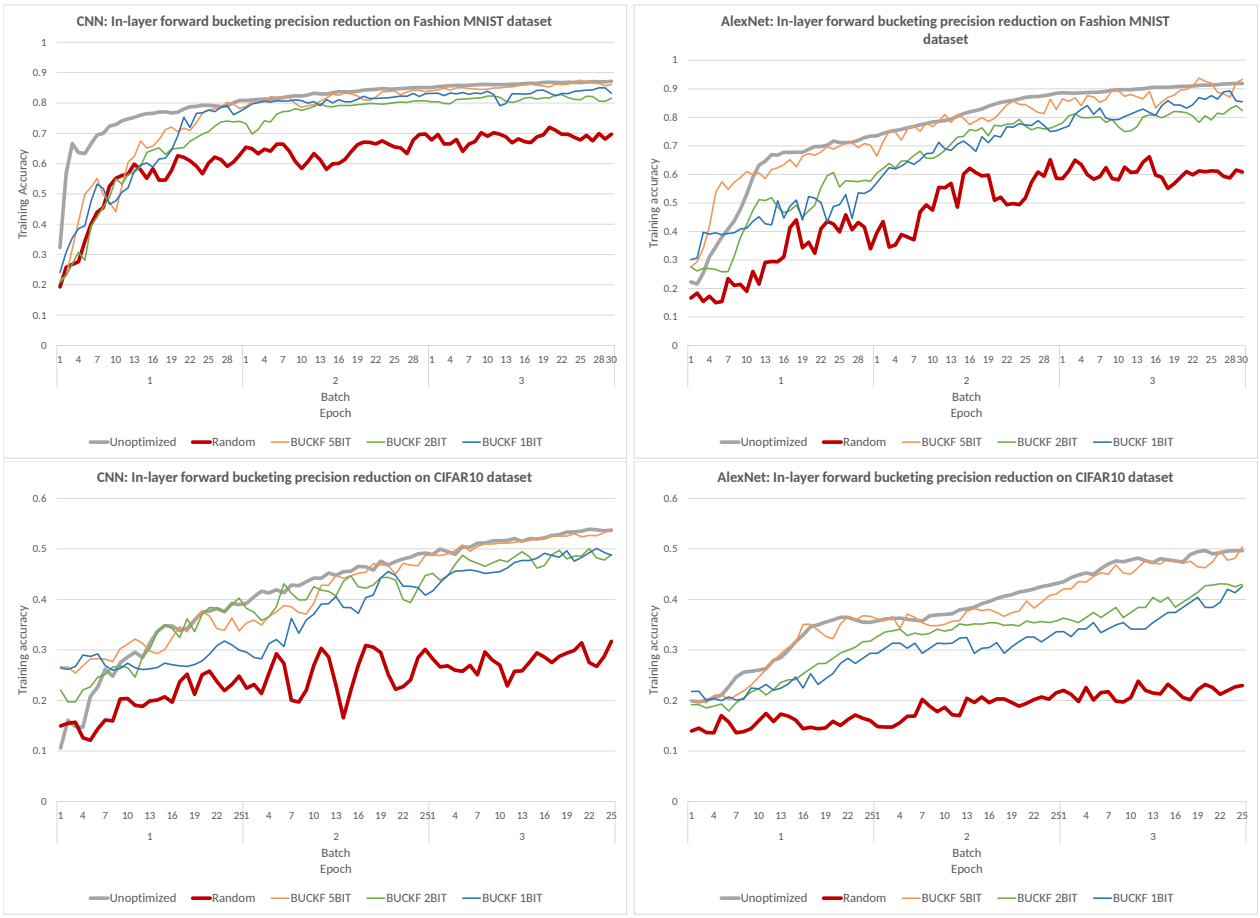

Figure 5.8: In-layer forward bucketing precision reduction on all the different datasets and networks

weights in every bucket during one iteration of training of the AlexNet on Fashion MNIST dataset, for both 5-bit and 2-bit increasing bucketing precision reduction. The colors of the buckets are chosen appropriately such that the different shades of a similar color indicate the approximate inclusion of a weight in that bucket for both 5-bit and 2-bit strategies. For example, the orange in the 5-bit precision reduction indicates the bucket with 23 bits of precision. The corresponding 2-bit buckets with similar weight values are the 22-bit and 23-bit buckets (colored in different shades of orange). We observe that there is more precision reduction in the 2-bit bucket strategy than the corresponding 5-bit bucket, since the weights in the top bucket for 5-bit bucketing would have full precision, but the corresponding 2-bit buckets has 22 bits of precision in one of them. Thus there is a reduction of precision already in the 2-bit bucket for the similarly valued weight than the 5-bit bucket. This holds true for all the buckets in both strategies. Obviously, because of different bucket boundaries and our

selection of bit-width for the bucketing strategies, the boundaries are not the same for both strategies, but in most of the cases (apart from the bucket for 5 bits of precision in the 5-bit forward bucketing strategy), the precision of a weight in a 5-bit reduction strategy will be less than its corresponding position in the 2-bit reduction strategy.

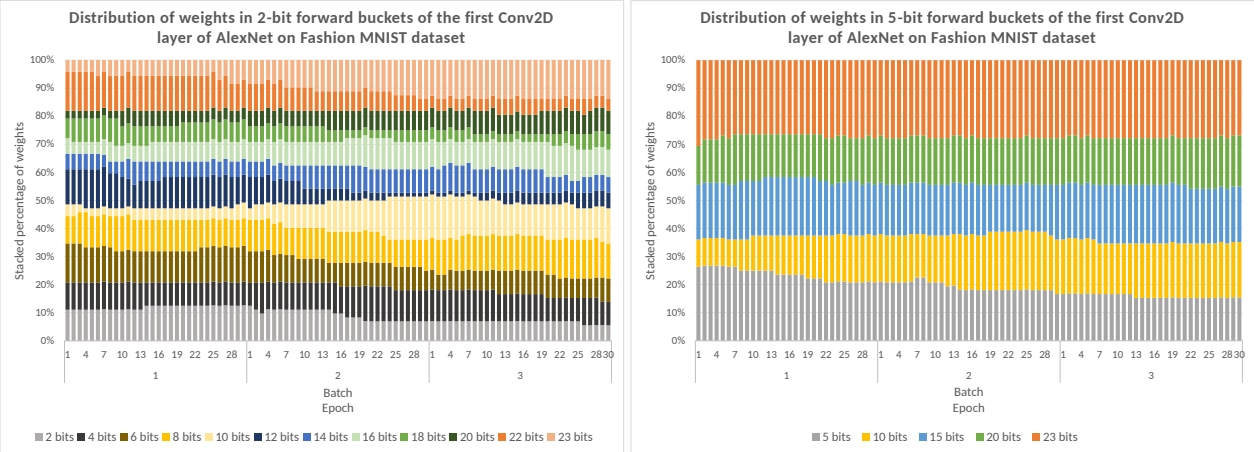

Figure 5.9: Distribution of weights in the first convolution layer of the CNN during training of forward bucketing precision reduction on the Fashion MNIST dataset

To further elaborate, we simply consider the final bucketing distribution of weights after training of the network, as shown in figure 5.10. It is clear that most of the weights in a given bucket in 2-bit forward bucketing precision reduction will have a lower precision in the 5-bit forward bucketing strategy, since most of the buckets for the 5-bit strategy are overlapped by a higher precision bucket in the 2-bit strategy, indicating that the weights will be lower or, at worst, have the same precision in the 2-bit strategy as the 5-bit strategy in most cases. If we compute the average number of bits saved over three iterations of experimentation, the 2-bit forward bucketing strategy reduces 722 bits from the mantissa of the weights of this layer, whereas the 5-bit forward bucketing strategy reduces 575 bits. Hence, it is clear that the 2-bit forward bucketing strategy reduces more bits than the 5-bit forward bucketing, which is why it has a lower training accuracy than the 5-bit forward bucketing strategy.

Since the graphical display of weight distribution of weights during an iteration is mostly similar, but with 23 buckets that makes it difficult to identify all the buckets properly, we did not report the exact nature of the distribution of weights for the 1-bit forward bucketing precision reduction strategy. However, the same concept holds for the precision of buckets in that situation as well. The average number of bits reduced in this layer for the 1-bit forward bucketing strategy is 787, higher than both the 5-bit and 2-bit bucketing.

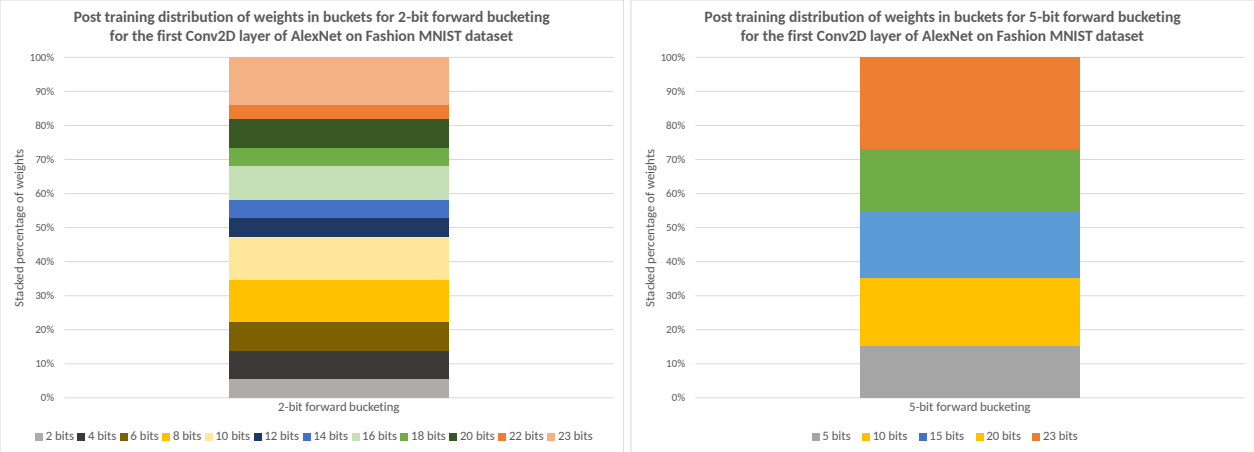

Figure 5.10: Distribution of weights in the first convolution layer of the CNN after training of forward bucketing precision reduction on the Fashion MNIST dataset

Another thing we observed is that the effect of precision reduction in the MNIST dataset is less pronounced than the other datasets, as seen in the previous strategies. This supports our previous reasoning of the MNIST dataset being too simple and easier to learn, since the loss of trailing mantissa bits does not have as much of an impact on training accuracy as it does on the Fashion MNIST or CIFAR10 dataset. This is visibly more clear when we compare the performance of the dense network on the two datasets for this strategy. For the MNIST dataset, even after precision reduction, the training accuracy for all the different bit-width bucketing strategies are very close to the full precision one. However, for the Fashion MNIST dataset, the training curve for the 2-bit and 1-bit strategies are closer to the random precision model accuracy. This indicates that the MNIST dataset can be learned efficiently with less than full precision, even for a simple network as the multilayer perceptron.

Overall, the increasing bucket reduction strategy trains at a higher accuracy than the layerwise precision reduction strategy, if the number of bits reduced across the network are similar. As was the case for layerwise precision reduction, this strategy can be used to train networks on low resource devices or nodes in a cluster. Moreover, because of the differing precision of weights in a layer, storage space can be further optimized for every layer. Since the precision reduction happens somewhat uniformly across the network, unlike the layerwise strategy which has high or low precision based on layer, this strategy is somewhat more robust and tolerant to information loss.

5.5 Decreasing bucket reduction

Finally, we train with the decreasing or "reverse" precision with increasing weight value. The decreasing bucket reduction strategy is explained in detail in section 4.2.4. We choose the same bucket bit-widths as we chose for the increasing bucket strategy:

- 5 bits: Precision in every bucket decreases by 5 bits. Since the full precision is 23 bits of mantissa, we have 5 buckets, with 23, 20, 15, 10 and 5 bits as precision for these buckets, in order of increasing weight values.

- 2 bits: Precision in every bucket decreases by 2 bits. Since the full precision is 23 bits of mantissa, we have 12 buckets, with 2 bits for the final bucket, 4 for the penultimate one, up to 22 for the second one. The first bucket has 23 bits of precision for the mantissa.

- 1 bit: Precision in every bucket decreases by 1 bit. Since the full precision is 23 bits of mantissa, we have 23 buckets, with the first bucket having 23 bits of precision and the final bucket having 1 bit of precision.

For the decreasing (or reverse) precision reduction, the smallest weights are put in the first bucket with highest precision and higher weights going in the next buckets with reducing precision. Detailed explanation of the strategy of deciding buckets and boundaries of buckets are explained in section 4.2.4. As in the case of the forward bucketing strategy, for comparison of weights and sorting them into buckets, we considered the absolute unsigned value of the weights.

Figures 5.11 and 5.12 show the training accuracy curves for the decreasing (reverse) bucketing precision reduction strategies, averaged over three separate experiments.

From the figures, we observe that, similar to the forward bucketing precision reduction, the 5-bit bucketing has a higher training accuracy than the 2-bit or 1-bit bucketing strategies. This is because of the fact that the justification of precision in buckets that we discussed in section 5.4 is valid for this situation as well.

However, it is observed that the reverse bucketing strategy learns worse than its forward bucketing countrerpart in most cases. The training accuracies of any bit-width reverse bucketing strategy is worse than its corresponding forward bucketing strategy. This implies that unlike the assumption we had in section 4.2.4, the precision of higher valued weights is more significant in predicting than the precision of lower valued weights.

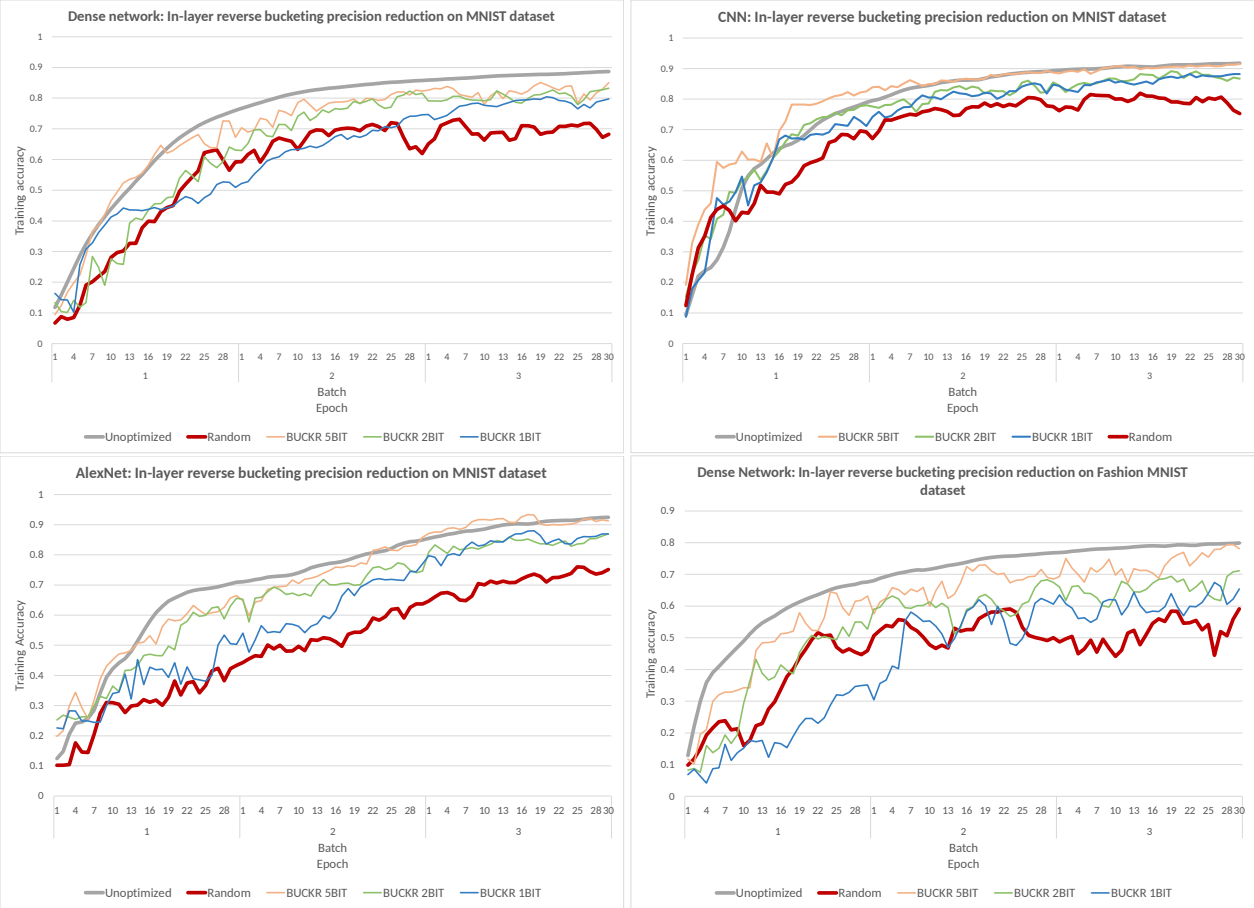

Figure 5.11: In-layer reverse bucketing precision reduction on all the different datasets and networks

5.6 Prediction on test data

Training the models at reduced precision showed promising results on training accuracy. We decided to analyze the learning of our models further by predicting on the test data on each of them. Tables 5.2 and 5.3 show the average prediction accuracy and time taken to predict on each of the test dataset over 3 iterations. Each of the datasets have 10000 images as test data.

Looking at the tables, we do observe a few interesting features that need to be addressed. The first thing we observe is that the unoptimized full precision strategy has the highest test accuracy. This was expected as the full precision does improve prediction. However, looking at the other values, it is clear that the bucketing strategies, even after sacrificing some amount of precision, is not far from the prediction accuracy of the full precision model.

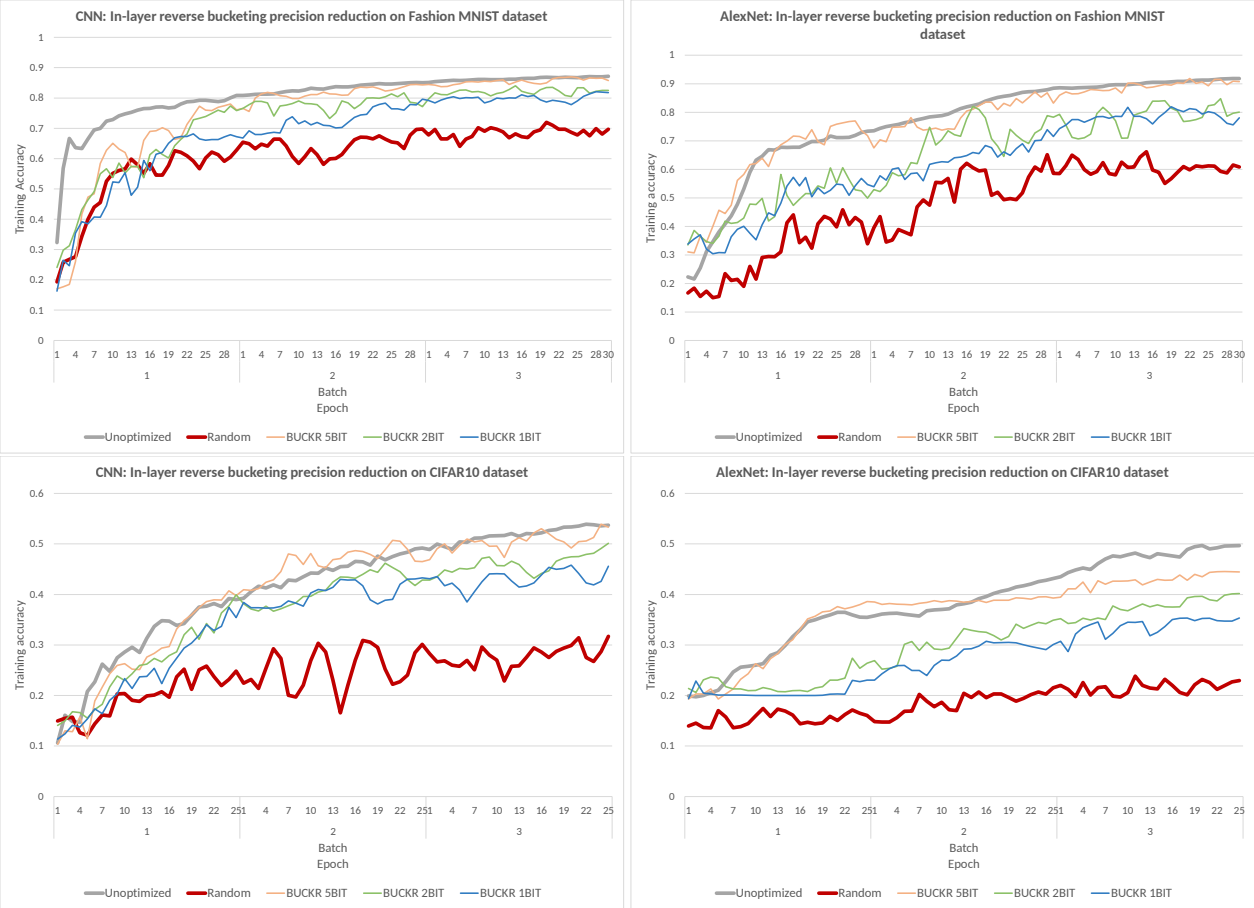

Figure 5.12: In-layer reverse bucketing precision reduction on all the different datasets and networks

Among the bucketing strategies, the forward bucketing strategy with a bit-width of 5 bits is the most successful in predicting on the test data. The other strategies that are close to the full precision model accuracy are models with 20 bit precision on the whole network and with a layer-wise precision model of 10 bits in the convolutional layers and 20 bits in the dense layer.

Of course, this brings us the question of whether the bucketing reduction is worth it, given that the layer-wise reduction is equally successful. For this we consider a network that we used, the CNN, as an example, and try to figure out the difference of bits reduced in both these strategies: the forward bucketing strategy and the 10C20D layerwise precision reduction strategy. Table 5.4 shows the total number of weights in each layer of the network. Layers that do not contain learnable weights (weights learned during the training of the network), such as the flatten or the dropout layers, are omitted.

	MNIST			Fashion MNIST			CIFAR10	
	Dense	CNN	AlexNet	Dense	CNN	AlexNet	CNN	AlexNet
Unoptimized	**89.46**	**93.53**	**93.13**	**78.52**	**85.92**	**79.32**	**39.94**	**30.23**
Random	78.85	79.24	75.96	52.34	68.93	55.56	32.26	19.37
WN 5bits	70.75	66.48	56.9	53.55	59.93	48.5	26.85	21.83
WN 10bits	78.03	79.1	89.68	72.94	77.71	68.73	31.72	30.37
WN 20bits	**85.1**	**92.78**	**91.22**	**75.91**	**82.39**	**75.22**	**36.15**	**33.52**
LN 5C10D[2]	75.44	90.87	83.06	55.81	76.7	66.28	30.43	25.56
LN 20C10D[2]	83.59	91.13	87.56	71.5	78.05	73.02	31.23	27.6
LN 10C5D[2]	80.77	71.35	69.47	70.14	72.59	48.24	27.82	23.3
LN 20C5D[2]	**83.79**	81.82	68.44	69.59	67.13	57.95	29.57	18.9
LN 5C20D[2]	75.67	91.58	85.54	72.92	77.59	67.62	36.45	23.43
LN 10C20D[2]	82.43	**92.18**	**90.61**	**74.55**	**79.90**	**74.35**	**39.03**	**28.53**
BUCKF 5bits	**84.92**	**92.78**	**90.96**	**77.66**	**83.7**	**77.58**	**40.05**	**31.27**
BUCKF 2bits	78.75	88.99	85.23	71.82	81.03	70.12	39.0	23.99
BUCKF 1bit	81.81	92.08	88.38	70.02	80	65.5	33.02	23.15
BUCKR 5bits	**83.17**	**92.79**	**86.53**	**71.57**	**82.74**	**75.5**	33.2	**25.05**
BUCKR 2bits	81.05	87.42	77.73	70.2	80.02	69.5	**37.47**	19.91
BUCKR 1bit	80.58	88.53	82.26	69.28	79.79	58.08	18.54	18.49

[1] The highlighted numbers are the highest prediction accuracy for the models in each category of precision reduction strategy
[2] For dense networks, the characters 'C' and 'D' are substituted for 'H' and 'O' respectively. Refer to table 5.1 for details.

Table 5.2: Prediction accuracy (in percentage) on the test data[1]

For the full layer-wise precision reduction strategy, the weights in the convolutional layers have a precision of 10 bits and the dense layers have precision of 20 bits. The convolutional layer weights lose 13 bits of precision and the dense layer weights lose 3 bits of precision. The total number of bits reduced from the full precision of 23 bits, during the training of the network is $238,056$ bits, or 29kb.

Calculating the precision reduction for the bucketing strategy is not that simple, however. Since it is dependant on the value of the weights, the total bits change for every training iteration and the dataset on which it is trained. Moreover, since we recalculate buckets at the end of every batch, the total weights per bucket change multiple times. For comparing only the test results, we consider the bucket allocation of weights at the end of training. We take one such training iteration on the Fashion MNIST dataset and look at the weight allocation of the buckets. The total weights per bucket at the end of training is displayed in Table 5.5.

	MNIST			Fashion MNIST			CIFAR10	
	Dense	CNN	AlexNet	Dense	CNN	AlexNet	CNN	AlexNet
Unoptimized	**1.05**	**1.20**	**1.61**	**1.2**	**1.47**	**1.85**	**1.88**	**1.91**
Random	0.71	0.70	0.96	0.70	0.77	0.86	0.99	1.07
WN 5bits	0.58	0.64	0.79	0.65	0.72	0.86	0.82	1.05
WN 10bits	0.76	0.85	0.92	0.90	0.92	1.18	1.19	1.47
WN 20bits	**0.92**	**1.07**	**1.41**	**1.01**	**1.28**	**1.66**	**1.40**	**1.65**
LN 5C10D[2]	0.71	0.81	0.90	0.88	0.87	1.09	1.16	1.32
LN 20C10D[2]	0.89	0.87	0.95	0.93	0.94	1.20	1.25	1.55
LN 10C5D[2]	0.67	0.71	0.82	0.78	0.79	0.92	0.96	1.08
LN 20C5D[2]	**0.75**	0.75	0.89	0.83	0.81	0.94	1.00	1.15
LN 5C20D[2]	0.83	0.97	1.28	0.94	1.19	1.33	1.30	1.57
LN 10C20D[2]	0.91	**1.05**	**1.38**	**0.96**	**1.24**	**1.65**	**1.36**	**1.62**
BUCKF 5bits	**0.75**	**0.70**	**0.94**	**0.85**	**0.94**	**0.96**	**1.06**	**1.15**
BUCKF 2bits	0.72	0.69	0.82	0.80	0.80	0.78	0.96	1.05
BUCKF 1bit	0.69	0.69	0.70	0.73	0.71	0.74	0.90	1.02
BUCKR 5bits	**0.75**	**0.97**	**0.96**	**0.88**	**0.92**	**0.94**	1.05	**1.05**
BUCKR 2bits	0.74	0.66	0.79	0.82	0.84	0.85	**1.00**	0.94
BUCKR 1bit	0.68	1.00	0.74	0.8	0.73	0.81	0.86	0.88

[1] The highlighted numbers are the corresponding prediction times for the models with highest prediction accuracy in Table 5.2

[2] For dense networks, the characters 'C' and 'D' are substituted for 'H' and 'O' respectively. Refer to table 5.1 for details.

Table 5.3: Prediction time (in seconds) on the test data[1]

Conv. Layer 1	72
Conv. Layer 2	1152
Hidden Layer	73728
Output Layer	320
Total	75272

Table 5.4: Total number of trainable weights in the network

	Bucket 1	Bucket 2	Bucket 3	Bucket 4	Bucket 5	**Total**
Conv. Layer 1	33	53	50	50	30	72
Conv. Layer 2	139	298	318	312	85	1152
Hidden Layer	1399	16283	36311	16602	4677	75272
Output Layer	28	84	68	30	110	320

Table 5.5: Total number of weights in each bucket of the CNN post training on Fashion MNIST dataset

Since we know the number of bits retained per bucket, we can calculate the total number of bits saved per bucket per layer. The results are shown in table 5.6.

	Bucket 1	Bucket 2	Bucket 3	Bucket 4	Bucket 5	**Total**
Conv. Layer 1	198	221	128	54	0	601
Conv. Layer 2	2502	3874	2544	936	0	9856
Hidden Layer	25182	211679	290488	49806	0	577155
Output Layer	504	1092	544	90	0	2230

Table 5.6: Total number of weights in each bucket of the CNN post training on Fashion MNIST dataset

The forward bucketing precision reduction strategy saves $589,842$ bits, or 72kb, in the training of the network, almost three times that of the layer-wise network. Upon observing across multiple passes, the total bits reduced increase or decrease by a maximum of $10,000$ bits approximately. Thus, the forward bucketing precision reduction strategy reduces more precision, which reduces the storage space for the network models and facilitates faster and less intensive floating point arithmetic operations.

The training of the neural networks in these precision reduction strategies do require a significantly large amount of time compared to the full precision model. The training of the CNN with no optimization strategy on the Fashion MNIST dataset takes about 3 minutes with the chosen hyperparameters on average. With the forward bucketing strategy, the training takes up to 40 minutes to complete. This is a major downside of our technique. This is majorly because of two main sections of the code:

- The creation and allocation of weights into buckets
- The actual reduction and pruning of mantissa bits

However, we set this experiment out as a simulation, with the expectation that specific hardware can be designed that would not depend on software function calls and variable information to determine the precision of a weight and manually reduce it at the end of every batch. Computational units that can work on custom precision can be designed to implement this effectively. Also, techniques can be considered to choose certain intervals for precision reduction instead of the end of every batch. However, the cost of training data is slightly offset by the fact that the prediction time of these strategies is less than the full precision model.

We observe from table 5.3 that the average prediction time of the networks with precision

reduction is lesser than that of the unoptimized full precision model. The highlighted (**bold**) times correspond to the times for the highest average accuracy in that precision reduction strategy, as highlighted in table 5.2. Because of the lower number of bits in the mantissa of the weights, the bucketing strategies take less time than the layer-wise and whole network precision reduction strategies. On average, the bucketing strategy takes 42.5% less time to predict than the full precision network. This indicates that the model trained with the bucketing strategy takes less time to compute the prediction than the full precision model, which validate our assumption that reducing precision bits would reduce the computation time of neural networks.

One particular thing of note here is that as we discussed in section 4.1.3, the float value, after reduction, is stored back into a 32 bit storage, with reduced bits replaced by zero. This is done internally to maintain the standard of IEEE 754. Thus, it raised a question of whether the reduction in prediction time is actually caused by the reduction of precision or something else. We came to the conclusion that it is indeed because of the precision reduction, and we base this on the following two statements:

- The only thing we changed in the network during the entire training process are the weights, at the end of every batch. We did not not perform any post-training optimization. Thus, the only difference between a full precision model and a reduced precision model are the weights. Thus, any change in prediction time would be for the change in weights. We trained both the unoptimized and the reduced precision models several times, and for every iteration, the prediction time is always lower for the reduced precision model. Moreover, we observed that in most cases the greater the number of bits reduced, the less time is taken.

- Secondly, we assume another possible explanation to this could be on a hardware level. Although the internal design of Intel and AMD CPUs is not public, CPUs can include optimizations that shortcut or skip for operations on zero, which would allow the CPUs to skip the trailing bits and the operations during the prediction phase. This assumption, although not proven by our research, is not of importance to us because our objective of this experiment is to create a simulation that would allow us to design hardware that would support custom precision. In such a scenario, trailing bits would not be used ideally to conform to the floating point standard, thus eliminating the speculation of CPU shortcut for operations on zero.

Looking at the results, it seems indicative that the staggered precision reduction strategy of allocating weights into buckets based on their values to determine their precision seems to train models well enough to produce comparable prediction results to the full precision strategy, while taking significantly less time and less storage space to do so.

Chapter 6

Conclusion and Future Work

In this work, we have tried to reduce the precision of weights in a neural network using floating point representation of float values. Using the IEEE 754 standard of floating point representation, we explored a technique of choosing precision of weights based on their value and segregating them into clusters or "buckets", and selecting a precision for every bucket. We also compared the results to two other floating-point approaches to traditional precision reduction techniques used in quantization, such as reducing the weights of an entire network to a common precision and to reduce the precision of every layer to a different value. The strategy is more fine-grained than a blanket of setting a common precision to the whole network or a layer, while also not hand-picking the precision for every single weight. We tried two different strategies of correlating weight values and precision, both of which were effective. Upon running our experiments on three datasets with three different types of networks, the results suggest that with the bucketing technique, it is possible to reach training and prediction accuracy similar to the full precision model, while reducing a significant number of mantissa bits. The bucket width of 5 bits with increasing precision for increasing value of weights learned the best, with prediction accuracies almost as high as the full precision network. Unlike quantization or other aggressive forms of precision reduction, this method does not need to implement any other form of optimization to improve accuracy. This facilitates the ability to implement other optimization algorithms in conjunction to our technique on the network independently.

Overall, this research has been an interesting look into the floating-point representation and manipulation of weights in a neural network, with an attempt at trying to reduce the precision while retaining accuracy of the neural network. Having lower prediction time encourages this strategy to be implemented in models running on low-power or handheld

devices, where a model once trained is used to predict on data large number of times. The faster prediction starts to offset the cost of training time in those cases.

Future work

Since this is a preliminary work in this domain, there are several paths we plan to extend this research. Our research direction was motivated by research being conducted in the NSERC COHESA[1] research network, which aims to develop custom hardware accelerators for machine learning. Research in this network has included various approaches, including floating point optimizations. Hardware support for reducing mantissa precision, without simulation or need to use zero bit-padding, would allow for direct reduction of precision of the weights instead of invoking a software call to bucket weights at the end of every batch, thus reducing training time overhead.

On the algorithm front, the precision reduction strategy we designed seems to be effective in convolutional networks, since the convolutional layers learn abstract concepts early, such as edges and curves, helping them retain the precision of these important features while choosing to reduce the precision of other weights. Implementing this strategy on larger and different types of networks, such as RNNs and autoencoders, and larger and different types of datasets, such as audio or textual datasets, would provide more information on the performance on this type of precision reduction. Although it is difficult to tell whether this precision reduction technique will be efficient enough to optimize the neural networks during training, our preliminary results show promise and bear implications that warrant the research into other networks and datasets.

In later versions of this research, we plan to experiment and design new precision reduction strategies and compare results. Some examples of such new strategies are:

- Choosing precision of weights based on their learning rate, or rate of change.
- Trying to figure out the ideal bucketing strategy for every layer.
- Associating precision with an appropriate distribution that is similar to the learning distribution of a weight.
- Determining precision based on sparsity of weights in a layer.

[1] `https://cohesa.org`

- Identifying regions or clusters of weights for choosing precision.

There is also an unexplored path of tuning the exponents of the floating point representation, on top of our proposed approach of mantissa reduction, that can be used to further optimize floating point precision during training. Approaches such as "Block Floating Point (BFP)" [5, 14] have been implemented to share common exponents and further reduce computational speed and improve the efficiency of neural networks. Such approaches can be applied on top of our mantissa reduction strategies for further optimization.

Another path of research that can be considered in the future is the optimization of the training process used in our precision reduction strategies. There are numerous opportunities for optimization. As we discussed in section 5.6, the long training time is an issue that we plan to resolve. Several approaches can be considered, such as reducing precision intermittently instead of every batch, deciding on a threshold accuracy to stop precision reduction, developing a way to foreshadow the cluster a weight should belong to, or creating a custom layer type of a neural network layer that learns to reduce precision of weights appropriately by itself instead of requiring manual specification.

Bibliography

[1] Alemdar, H., Leroy, V., Prost-Boucle, A., and Pétrot, F. Ternary Neural Networks for Resource-Efficient AI Applications. *arXiv:1609.00222 [cs]* (Feb. 2017). arXiv: 1609.00222.

[2] Anwar, S., Hwang, K., and Sung, W. Fixed point optimization of deep convolutional neural networks for object recognition. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (Apr. 2015), pp. 1131–1135.

[3] Asanovic, K., and Morgan, N. *Experimental determination of precision requirements for back-propagation training of artificial neural networks.* International Computer Science Institute, 1991.

[4] Bengio, Y., Léonard, N., and Courville, A. Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation. *arXiv:1308.3432 [cs]* (Aug. 2013). arXiv: 1308.3432.

[5] Bittner, R., and Forin, A. Block floating point for neural network implementations, June 2018. Patent number US10528321B2, Filed May 10th., 2017, Issued June 7th., 2018.

[6] Brightwell, G., Kenyon, C., and Paugam-Moisy, H. Multilayer neural networks: one or two hidden layers? In *Advances in Neural Information Processing Systems* (1997), pp. 148–154.

[7] Cauchy, A. Méthode générale pour la résolution des systemes d'équations simultanées. *Comp. Makes. Sci. Paris 25*, 1847 (1847).

[8] Chen, S., Wang, W., and Pan, S. J. Deep neural network quantization via layer-wise optimization using limited training data. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2019), vol. 33, pp. 3329–3336.

[9] Choukroun, Y., Kravchik, E., Yang, F., and Kisilev, P. Low-bit Quantization of Neural Networks for Efficient Inference. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)* (2019), IEEE, pp. 3009–3018.

[10] Coates, A., Baumstarck, P., Le, Q., and Ng, A. Y. Scalable learning for object detection with GPU hardware. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems* (Oct. 2009), pp. 4287–4293. ISSN: 2153-0866.

[11] Courbariaux, M., Bengio, Y., and David, J.-P. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. In *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds. Curran Associates, Inc., 2015, pp. 3123–3131.

[12] Cybenko, G. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems 2*, 4 (Dec. 1989), 303–314.

[13] Das, D., Mellempudi, N., Mudigere, D., Kalamkar, D., Avancha, S., Banerjee, K., Sridharan, S., Vaidyanathan, K., Kaul, B., and Georganas, E. Mixed precision training of convolutional neural networks using integer operations. *arXiv preprint arXiv:1802.00930* (2018).

[14] Drumond, M., LIN, T., Jaggi, M., and Falsafi, B. Training DNNs with Hybrid Block Floating Point. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 2018, pp. 453–463.

[15] Farabet, C., Poulet, C., Han, J. Y., and LeCun, Y. CNP: An FPGA-based processor for Convolutional Networks. In *2009 International Conference on Field Programmable Logic and Applications* (Aug. 2009), pp. 32–37. ISSN: 1946-1488.

[16] Gong, Y., Liu, L., Yang, M., and Bourdev, L. Compressing Deep Convolutional Networks using Vector Quantization. *arXiv:1412.6115 [cs]* (Dec. 2014). arXiv: 1412.6115.

[17] Griffiths, S. The BitStream class - bitstring documentation. `https://bitstring.readthedocs.io/en/latest/bitstream.html`.

[18] Guan, Y., Liang, H., Xu, N., Wang, W., Shi, S., Chen, X., Sun, G., Zhang, W., and Cong, J. FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (Apr. 2017), pp. 152–159.

[19] Guo, K., Sui, L., Qiu, J., Yu, J., Wang, J., Yao, S., Han, S., Wang, Y., and Yang, H. Angel-Eye: A Complete Design Flow for Mapping CNN Onto Embedded FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 37*, 1 (Jan. 2018), 35–47. Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[20] Gupta, S., Agrawal, A., Gopalakrishnan, K., and Narayanan, P. Deep learning with limited numerical precision. In *International Conference on Machine Learning* (2015), pp. 1737–1746.

[21] Gysel, P. Ristretto: Hardware-Oriented Approximation of Convolutional Neural Networks. *arXiv:1605.06402 [cs]* (May 2016). arXiv: 1605.06402.

[22] Han, S., Kang, J., Mao, H., Hu, Y., Li, X., Li, Y., Xie, D., Luo, H., Yao, S., Wang, Y., Yang, H., and Dally, W. B. J. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA, Feb. 2017), FPGA '17, Association for Computing Machinery, pp. 75–84.

[23] Han, S., Mao, H., and Dally, W. J. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv:1510.00149 [cs]* (Oct. 2015). arXiv: 1510.00149.

[24] Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A.-r., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N., and Kingsbury, B. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine 29*, 6 (Nov. 2012), 82–97.

[25] Holt, J., and Baker, T. Back propagation simulations using limited precision calculations. In *IJCNN-91-Seattle International Joint Conference on Neural Networks* (Seattle, WA, USA, 1991), vol. ii, IEEE, pp. 121–126.

[26] Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. Binarized Neural Networks. In *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds. Curran Associates, Inc., 2016, pp. 4107–4115.

[27] Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. *The Journal of Machine Learning Research 18*, 1 (2017), 6869–6898.

[28] Hwang, K., and Sung, W. Fixed-point feedforward deep neural network design using weights +1, 0, and −1. In *2014 IEEE Workshop on Signal Processing Systems (SiPS)* (Oct. 2014), pp. 1–6. ISSN: 2162-3570.

[29] Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., and Kalenichenko, D. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *arXiv:1712.05877 [cs, stat]* (Dec. 2017). arXiv: 1712.05877.

[30] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada, June 2017), ISCA '17, Association for Computing Machinery, pp. 1–12.

[31] Judd, P., Albericio, J., Hetherington, T., Aamodt, T., Jerger, N. E., Urtasun, R., and Moshovos, A. Reduced-Precision Strategies for Bounded Memory in Deep Neural Nets. *arXiv:1511.05236 [cs]* (Jan. 2016). arXiv: 1511.05236.

[32] Kingma, D. P., and Welling, M. Auto-Encoding Variational Bayes. *arXiv:1312.6114 [cs, stat]* (May 2014). arXiv: 1312.6114.

[33] Krizhevsky, A., Hinton, G., et al. Learning multiple layers of features from tiny images. Tech. rep., University of Toronto, 2009.

[34] Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[35] Lai, L., Suda, N., and Chandra, V. Deep Convolutional Neural Network Inference with Floating-point Weights and Fixed-point Activations. *arXiv:1703.03073 [cs]* (Mar. 2017). arXiv: 1703.03073.

[36] Lawrence, S., Giles, C. L., and Tsoi, A. C. Lessons in neural network training: Overfitting may be harder than expected. In *AAAI/IAAI* (1997), Citeseer, pp. 540–545.

[37] Li, F., Zhang, B., and Liu, B. Ternary Weight Networks. *arXiv:1605.04711 [cs]* (Nov. 2016). arXiv: 1605.04711.

[38] Li, H., Fan, X., Jiao, L., Cao, W., Zhou, X., and Wang, L. A high performance FPGA-based accelerator for large-scale convolutional neural networks. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)* (Aug. 2016), pp. 1–9. ISSN: 1946-1488.

[39] Li, Z., Ni, B., Zhang, W., Yang, X., and Gao, W. Performance Guaranteed Network Acceleration via High-Order Residual Quantization. In *Proceedings of the IEEE International Conference on Computer Vision* (2017), pp. 2584–2592.

[40] Liang, S., Yin, S., Liu, L., Luk, W., and Wei, S. FP-BNN: Binarized neural network on FPGA. *Neurocomputing 275* (Jan. 2018), 1072–1086.

[41] Lozito, G.-M., Laudani, A., Riganti Fulginei, F., and Salvini, A. FPGA Implementations of Feed Forward Neural Network by using Floating Point Hardware

Accelerators. *Advances in Electrical and Electronic Engineering 12*, 1 (Mar. 2014), 30 – 39.

[42] Luo, C., Sit, M.-K., Fan, H., Liu, S., Luk, W., and Guo, C. Towards efficient deep neural network training by FPGA-based batch-level parallelism. *Journal of Semiconductors 41*, 2 (Feb. 2020), 022403. Publisher: IOP Publishing.

[43] Na, T., and Mukhopadhyay, S. Speeding Up Convolutional Neural Network Training with Dynamic Precision Scaling and Flexible Multiplier-Accumulator. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design* (New York, NY, USA, 2016), ISLPED '16, ACM, pp. 58–63. event-place: San Francisco Airport, CA, USA.

[44] Naftaly, U. The Modified BFPQ algorithm. In *7th European Conference on Synthetic Aperture Radar* (June 2008), pp. 1–4.

[45] Nguyen, D., Kim, D., and Lee, J. Double MAC: Doubling the performance of convolutional neural networks on modern FPGAs. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017* (Mar. 2017), pp. 890–893. ISSN: 1558-1101.

[46] Nurvitadhi, E., Sheffield, D., Sim, J., Mishra, A., Venkatesh, G., and Marr, D. Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC. In *2016 International Conference on Field-Programmable Technology (FPT)* (Dec. 2016), pp. 77–84.

[47] NVIDIA. NVIDIA Volta AI Architecture. `https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/`.

[48] Oppenheim, A. Realization of digital filters using block-floating-point arithmetic. *IEEE Transactions on Audio and Electroacoustics 18*, 2 (June 1970), 130–136. Conference Name: IEEE Transactions on Audio and Electroacoustics.

[49] Panchal, G., Ganatra, A., Kosta, Y. P., and Panchal, D. Behaviour Analysis of Multilayer Perceptronswith Multiple Hidden Neurons and Hidden Layers. *International Journal of Computer Theory and Engineering* (2011), 332–337.

[50] Podili, A., Zhang, C., and Prasanna, V. Fast and efficient implementation of Convolutional Neural Networks on FPGA. In *2017 IEEE 28th International Confer-*

ence on Application-specific Systems, Architectures and Processors (ASAP) (July 2017), pp. 11–18. ISSN: 2160-052X.

[51] Qiu, J., Wang, J., Yao, S., Guo, K., Li, B., Zhou, E., Yu, J., Tang, T., Xu, N., Song, S., Wang, Y., and Yang, H. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA, Feb. 2016), FPGA '16, Association for Computing Machinery, pp. 26–35.

[52] Ralev, K., and Bauer, P. Realization of block floating-point digital filters and application to block implementations. *IEEE Transactions on Signal Processing 47*, 4 (Apr. 1999), 1076–1086. Conference Name: IEEE Transactions on Signal Processing.

[53] Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In *Computer Vision – ECCV 2016* (Cham, 2016), B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds., Lecture Notes in Computer Science, Springer International Publishing, pp. 525–542.

[54] Shayer, O., Levi, D., and Fetaya, E. Learning Discrete Weights Using the Local Reparameterization Trick. In *International Conference on Learning Representations* (2018).

[55] Soudry, D., Hubara, I., and Meir, R. Expectation Backpropagation: Parameter-Free Training of Multilayer Neural Networks with Continuous or Discrete Weights. In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 963–971.

[56] Sridharan, S., and Dickman, G. Block floating-point implementation of digital filters using the DSP56000. *Microprocessors and Microsystems 12*, 6 (July 1988), 299–308.

[57] Stathakis, D. How many hidden layers and nodes? *International Journal of Remote Sensing 30*, 8 (Apr. 2009), 2133–2147.

[58] Underwood, K. FPGAs vs. CPUs: trends in peak floating-point performance. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable*

gate arrays (Monterey, California, USA, Feb. 2004), FPGA '04, Association for Computing Machinery, pp. 171–180.

[59] Wang, N., Choi, J., Brand, D., Chen, C.-Y., and Gopalakrishnan, K. Training Deep Neural Networks with 8-bit Floating Point Numbers. In *Advances in Neural Information Processing Systems 31* (2018), S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., Curran Associates, Inc., pp. 7675–7684.

[60] Wegener, A. W. Block floating point compression of signal data, Oct. 2012. Patent number US8301803B2, Filed October 23rd., 2009, Issued October 30th., 2012.

[61] Xiao, H., Rasul, K., and Vollgraf, R. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. `https://github.com/zalandoresearch/fashion-mnist`, 2017.

[62] Yann LeCun, C. C. MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges. `http://yann.lecun.com/exdb/mnist/`.

[63] Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., and Cong, J. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA, Feb. 2015), FPGA '15, Association for Computing Machinery, pp. 161–170.

[64] Zhou, S., Wu, Y., Ni, Z., Zhou, X., Wen, H., and Zou, Y. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *arXiv:1606.06160 [cs]* (Feb. 2018). arXiv: 1606.06160.