Exploring Per-Input Filter Selection and Approximation Techniques for Deep Neural Networks

Yamini Gaur

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

Patrick R. Schaumont, Chair

Steve Jian, Co-chair

Amos L. Abbott

May 1, 2019

Blacksburg, Virginia

Keywords: Approximate Computing, Convolution Layers, Fully Connected Layers, Neural Networks

Copyright 2019, Yamini Gaur

Exploring Per-Input Filter Selection and Approximation Techniques for Deep Neural Networks

Yamini Gaur

(ABSTRACT)

We propose a dynamic, input dependent filter approximation and selection technique to improve the computational efficiency of Deep Neural Networks. The approximation techniques convert 32 bit floating point representation of filter weights in neural networks into smaller precision values. This is done by reducing the number of bits used to represent the weights. In order to calculate the per-input error between the trained full precision filter weights and the approximated weights, a metric called Multiplication Error(ME) has been chosen. For convolutional layers, ME is calculated by subtracting the approximated filter weights from the original filter weights, convolving the difference with the input and calculating the grand-sum of the resulting matrix. For fully connected layers, ME is calculated by subtracting the approximated filter weights from the original filter weights, performing matrix multiplication between the difference and the input and calculating the grand-sum of the resulting matrix. ME is computed to identify approximated filters in a layer that result in low inference accuracy. In order to maintain the accuracy of the network, these filters weights are replaced with the original full precision weights.

Our proposed technique aims to achieve higher inference accuracy by not approximating filters that generate high ME. Using the proposed per-input filter selection technique, LeNet achieves an accuracy of 95.6% with 3.34% drop from the original accuracy value of 98.9% for truncating to 3 bits for the MNIST dataset. On the other hand, upon static filter approximation, LeNet achieves an accuracy of 90.5% with 8.5% drop from the original accuracy.

We explore various filter approximation techniques and implement a per-input filter selection and approximation technique that selects the filters to approximate during run-time.

Exploring Per-Input Filter Selection and Approximation Techniques for Deep Neural Networks

Yamini Gaur

(GENERAL AUDIENCE ABSTRACT)

Deep neural networks, just like the human brain can learn important information about the data provided to them and can classify a new input based on the labels corresponding to the provided dataset. Deep learning technology is heavily employed in devices using computer vision, image and video processing and voice detection. The computational overhead incurred in the classification process of DNNs prohibits their use in smaller devices. This research aims to improve network efficiency in deep learning by replacing 32 bit weights in neural networks with less precision weights in an input-dependent manner.

Trained neural networks are numerically robust. Different layers develop tolerance to minor variations in network parameters. Therefore, differences induced by low-precision calculations fall well within tolerance limit of the network. However, for aggressive approximation techniques like truncating to 3 and 2 bits, inference accuracy drops severely.

We propose a dynamic technique that during run-time, identifies the approximated filters resulting in low inference accuracy for a given input and replaces those filters with the original filters to achieve high inference accuracy.

The proposed technique has been tested for image classification on Convolutional Neural Networks. The datasets used are MNIST and CIFAR-10. The Convolutional Neural Networks used are 4-layered CNN, LeNet-5 and AlexNet.

Dedication

This is for Mom, Dad, Dadi and Aditi.

Acknowledgments

I would like to extend my sincerest gratitude to Dr. Steve Jian for pitching the idea of this project in our Graduate Level Computer Architecture course. His constant guidance, understanding, agility and enthusiasm helped bring this project to completion.

I would like to thank Dr. Patrick Schaumont and Dr. Lynn Abbott for agreeing to be a part of my thesis committee and providing their valuable inputs.

I would like to thank the HEAP (High-performance, Energy-efficient, Assured Processing) Lab at Virginia Tech for providing me with the right resources, environment and motivation to complete this project. I would also like to extend my gratitude to Suraj, Varun, Raghunath and Anamika for helping me understand the basics of deep learning using Tensorflow.

Lastly, I would like to thank my friends (Madhura, Sudha, Pranavi and Naina) for always believing in me. Their kind words and encouragement got me here and I am forever grateful to them.

Contents

List of Figures ix

List of Tables xii

1 Introduction 1

2 Background 4

2.1 Neural Networks 4

2.1.1 Deep Neural Networks 7

2.1.2 Images as 4-D Tensors 9

2.1.3 4-Layered CNN 11

2.1.4 Lenet 12

2.1.5 AlexNet 14

2.1.6 Deep Learning using TensorFlow 15

2.2 Datasets 16

2.2.1 MNIST 16

2.2.2 CIFAR-10 16

2.3 Approximate Computing 18

2.4 Approximation Techniques 18

3 Motivation **20**

3.1 Multiplication Error . . . 20

3.1.1 ME for Convolutional Layers . . . 21

3.1.2 ME for Fully Connected Layers . . . 24

3.2 Analysis of Time Complexity . . . 26

3.3 Static and Dynamic Multiplication Error Analysis . . . 28

4 Methodology **34**

4.1 Setup . . . 34

4.2 Deep Learning on TensorFlow . . . 35

4.3 Implementation of Approximation Techniques . . . 43

4.3.1 Filter Selection Techniques . . . 44

5 Results **49**

5.1 MNIST dataset . . . 49

5.1.1 Simple Convolutional Neural Network . . . 49

5.1.2 LeNet . . . 50

5.1.3 AlexNet . . . 51

5.2 CIFAR-10 Dataset . . . 51

5.2.1 Simple Convolutional Neural Network . . . 51

5.2.2 LeNet . . . 52

5.2.3 AlexNet 52

6 Conclusion **57**

Bibliography **59**

List of Figures

2.1 Classification pipeline of neural networks [12] 5

2.2 Neural network before and after dropout. [5] 8

2.3 Architectural Overview of a CNN 11

2.4 Architectural Overview of LeNet [21] 13

2.5 Architecture of AlexNet [9] 14

2.6 Images from the MNIST Dataset 16

2.7 Images from the CIFAR-10 Dataset 17

3.1 Number of operations to calculate ME for 4-layered CNN 26

3.2 Number of operations to calculate ME for LeNet 27

3.3 Number of operations to calculate ME for AlexNet 28

3.4 Pipeline for static ME analysis during training 29

3.5 Pipeline for dynamic ME analysis during training 29

3.6 Static and Dynamic Analysis for Convolutional Layer 1 30

3.7 Static and Dynamic Analysis for Convolutional Layer 2 31

3.8 Static and Dynamic Analysis for Convolutional Layer 3 31

3.9 Static and Dynamic Analysis for Fully Connected Layer 1 32

3.10 Static and Dynamic Analysis for Fully Connected Layer 2 32

3.11 Static and Dynamic Analysis for Fully Connected Layer 3 33

4.1 Training pipeline for classification 36

4.2 Steps to load the MNIST dataset on Tensorflow 36

4.3 Steps to download the CIFAR-10 dataset on Tensorflow 37

4.4 Steps to load labels for the CIFAR-10 dataset 37

4.5 Dataset exploration for CIFAR-10 38

4.6 Code snippet for computing test accuracy 41

4.7 Accuracy on the Test Set after 10000 iterations 42

4.8 Testing pipeline for classification 42

4.9 Code snippet for converting decimal number to binary string 44

4.10 Code to convert the string back to float 45

4.11 Pipeline for dynamic filter selection 45

4.12 List of sorted ME values 47

4.13 List of Filter IDs corresponding to the sorted ME values shown in 4.12 . . . 47

4.14 Pipeline for static filter selection 48

5.1 Design-time sorted ME order for Convolutional Layer 1 54

5.2 Run-time sorted ME order for Convolutional Layer 1 54

5.3 Design-time sorted ME order for Convolutional Layer 2 55

5.4 Run-time sorted ME order for Convolutional Layer 2 55

5.5 Design-time sorted ME order for Convolutional Layer 2 56

5.6 Design-time sorted ME order for Convolutional Layer 2 56

List of Tables

5.1 Classification Accuracy of a simple CNN on MNIST dataset 50

5.2 Classification Accuracy of LeNet on MNIST dataset 50

5.3 Classification Accuracy of AlexNet on MNIST dataset 51

5.4 Classification Accuracy of a simple CNN on CIFAR-10 dataset 52

5.5 Classification Accuracy of LeNet on CIFAR-10 dataset 52

5.6 Classification Accuracy of AlexNet on CIFAR-10 dataset 53

Chapter 1

Introduction

Deep learning algorithms have proliferated Artificial Intelligence applications that use computer vision, speech recognition, image and video processing and robotics. Achieving high inference accuracy in Deep Neural Networks (DNNs) comes at the cost of high computational complexity. The sheer size of neural networks can represent a challenging computational burden, even for modern day CPUs. [2]

Majority of research in deep learning has primarily focused on improving classification accuracy without much focus on making the models computationally efficient. Wide deployment of AI technology requires DNNs to be less computationally complex in terms of energy efficiency and throughput. [18] This comes at the cost of reduced inference accuracy which inhibits the use of deep learning technology in low power embedded devices.

Neural networks are naturally error resilient and do not require precise computations and number representations with high dynamic range. [19] This means that network parameters in a neural network can be reduced to lower precision without a significant loss of accuracy. While previous work has explored different schemes to lower the bit-width of network parameters while ensuring high accuracy, the best quantization schemes and the minimum number of bits required for a network to still retain its high accuracy is still unknown. [20].

The motivation behind this research is to explore input-aware means of approximating filters in deep neural networks without significant loss in accuracy. Previous work has focused on filter approximation at design-time wherein all the filter weights belonging to the con-

volutional and fully connected layers of the neural network were switched to approximated weights. The focus of this work is to implement an input dependent approximation technique that decides which filters to approximate and which ones to change to full precision weights during run-time.

We explore three deep neural networks with varying depths across two datasets. The neural networks explored are a simple 4-layered CNN, LeNet and AlexNet. The datasets used are MNIST and CIFAR-10. A wide range approximation methods, varying in degrees of aggressiveness are explored throughout this research. This is done to assess percentage increase in inference accuracy between static and dynamic approximation techniques as the aggressiveness of approximation increases. In order to find filters in a layer that contribute to low accuracy post approximation, we explore a metric defined as Multiplication Error (ME). A way to fast-calculate ME during run-time has been researched. ME analysis is critical since it offers an input-dependent way of deciding which filters in a layer should be approximated to low precision weights without greatly reducing the inference accuracy of the network. With our findings, we answer the following questions-

- How much increase in inference accuracy can the proposed technique yield in comparison to static filter approximation techniques?
- How can we make the decision of which filters to approximate during run-time?

The rest of the thesis is organized as follows:

Chapter 2 provides the relevant background for the research in this thesis. This includes background on deep neural networks, Tensorflow, hyperparameters of a neural network and specifics about the architectures of networks used throughout the thesis. The datasets used in the research are also described in detail.

Chapter 3 describes the proposed selection and approximation techniques in detail and delves

into more detail about Multiplication Error and how to fast-calculate ME during run-time.
Chapter 4 details the methodology employed in carrying out the research. It describes the software setup, the python packages needed to run the programs, code snippets for various approximation techniques as well as working with multi-dimensional tensors.
Chapter 5 talks about the results as well as how this research can be furthered to improve approximate computing techniques for machine learning.

Chapter 2

Background

This chapter provides the necessary background required to understand the research. Section 2.1 introduces the concept of neural networks, hyperparameters employed by deep learning algorithms, the architectures explored in the research and how convolutional neural networks perceive images as tensors. Section 2.2 discusses the datasets used and how to preprocess images in the datasets to be compatible with different DNNs. Section 2.4 briefly touches upon the approximation techniques used.

2.1 Neural Networks

A neural network is a set of algorithms modeled loosely on the human brain that is designed to recognize patterns. The output of one layer of the neural network is fed as input into the second layer of the neural network. The output that comes out of a layer in the network is defined by an activation function that maps the output is a particular range so that the next layer is able to appropriately act on the input and signal other neurons accordingly. Let us take the example of an image. An image has several features spread out along all axes. Each layer in a neural network develops a more abstract representation of the input image and is able to successfully detect different patterns in an image. The network correlates its detected feature map with the correct label that the image is associated with. At the same time, it considers the label that it associated with the input image through its own analysis

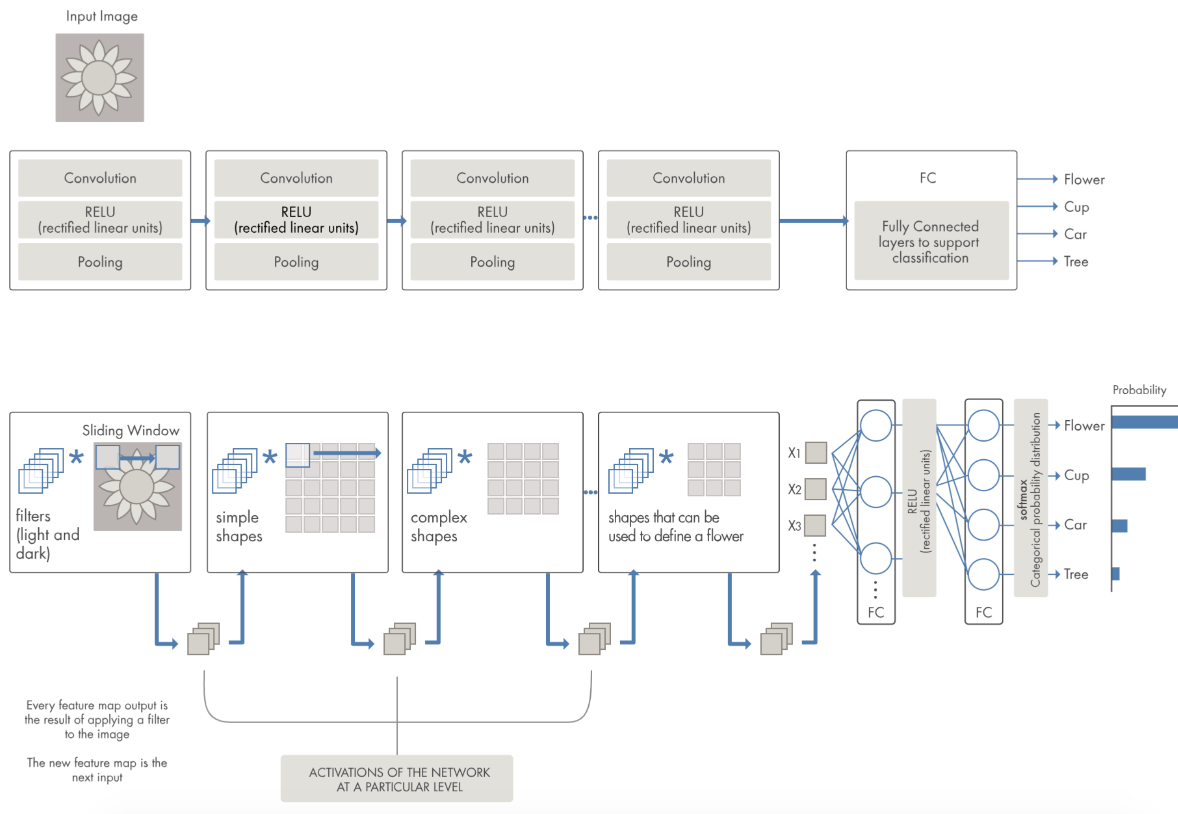

Figure 2.1: Classification pipeline of neural networks [12]

and then goes back to the beginning of the network to modify its weights and biases in a way that brings the next iteration results closer to the true label associated with the image. With enough number of iterations and a varied dataset, the network is able to learn the features corresponding to a particular label. With this learning, it is able to correctly associate a label with a new input image. Some applications of neural networks are -

- Classification

  Classification is the ability of a network to correctly associate a label to a given input. Classification is widely used in the industry and modern day applications such as Amazons Echo Dot and Google Home. The AI bot in both the devices is able to recognize the users voice and is able to differentiate between the voices of different

users using the same device. While setting up the device, it asks the user to say the same sentence multiple times. This is done to train the underlying neural network with the dataset (which is the users voice) so that in the future, the network can associate that voice with the corresponding user's name.

Object detection in images is also a classification problem. Computer vision technology is employed heavily in self driving cars to differentiate between marked roads and pavements, to identify stop signs, pedestrians, lane markers etc. This also includes gesture recognition, facial expression classification etc.

Text classification is used widely to build datasets from the material available of news channels and social media websites. For instance, in order to understand the impact of a natural calamity or an event, data analysts write scripts to crawl the internet and gather data that a neural network can classify as related to the event. This helps build the dataset that the analysts can further work on.

- Clustering

  Clustering of data is basically grouping similar data together in a cluster. Training data in neural networks is not always labeled. Training a net on unlabeled data is called unsupervised learning. Since majority of the data in the world is unlabeled, there is a great scope to build networks that can perform unsupervised learning. Since the classification and clustering accuracy of a deep neural network increases with the amount of data that it is trained on, unsupervised neural networks are highly accurate. Some applications of clustering are:

  - Searching for similar documents

    A challenge in data mining is document classification. Deep learning techniques help in grouping together similar documents that can aid greatly in binary distribution of data. For instance, a lot of research pertaining to information security

involves getting code snippets from resources like StackOverflow and classifying those code snippets as secure and insecure. Clustering of data is used for such applications.

– Anomaly Detection

Neural networks can be used to detect documents that are dissimilar; hence providing with information about fraudulent/ anomalous documents.

2.1.1 Deep Neural Networks

Deep neural networks have more than an input and an output layer. The depth of a neural network is classified by the number of hidden layers that the network has. Deep neural networks are imperative to achieving high classification accuracy. This is because each layer in a neural network is able to learn certain features about the input image that helps the network classify the input image. As the number of layers in a network increase, the network is able to learn more features and get better at the classification process.
Hyperparameters are variables that are set before training that determine the network structure and determine how the network is going to be trained. Hyperparameters control the architecture, the learning rate, the number of iterations that the network has to be trained for as well as the number of images that are shown to the network during a single iteration. Hyperparameter tuning is modifying the parameters in order to achieve the best possible outcome.

- Number of Hidden Layers

  Hidden layers are the layers between the input and the output layer. Each layer in the neural network performs some operations and modifications to the input and convert it to a form that the output layer can use. For example, Alexnet uses five convolutional

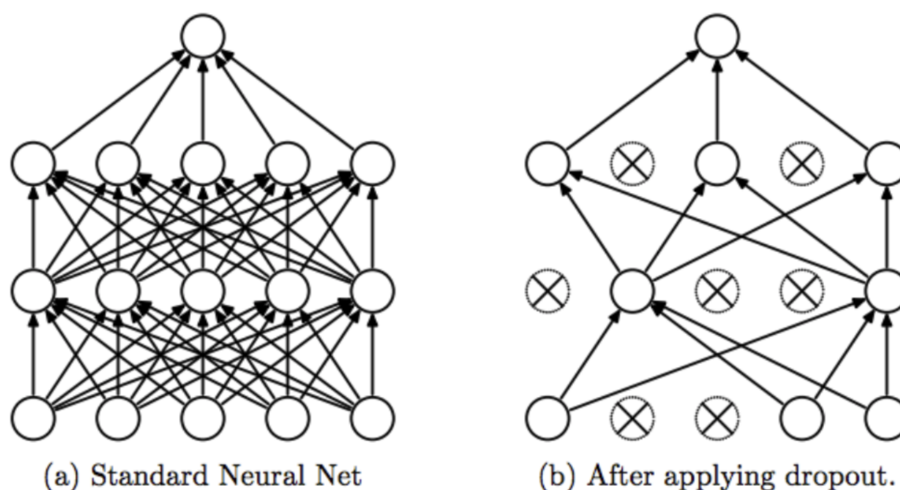

Figure 2.2: Neural network before and after dropout. [5]

layers and three fully connected layers. Since the input layer is a convolutional layer and the output layer is a fully connected layer, the hidden layers are the four remaining convolutional layers along with the two remaining fully connected layers.

- Dropout

  A fully connected layer uses most of the parameters and as a result, neurons form a lot of co-dependency amongst each other during training. This is called over-fitting and it prevents individual neurons from learning more about the dataset. Over-fitting is reduced by a process called Regularization that adds a penalty to the loss function. An example of this is Dropout. By adding this penalty, individual neurons do not learn features that are interdependent. Dropout basically ignores certain units during the training phase so reduce the computational load of training. These units are not considered during a forward or backward pass.

- Activation Function

  Activation function is responsible for transforming the input to a node into a suitable output for that day. An example of a popular activation function used in neural

networks is ReLU (Rectified Linear Activation Function) that outputs the output into the following neuron directly if the result is positive, otherwise it will output a zero.

- Learning Rate

  Learning rate defines the learning curve of a network. A steep learning curve implies that the network learns fast whereas a gradual curve implies a slow learning rate for the network. The problem with steep curves is less amount of convergence. Slow learning processes converge smoothly.

- Number of Epochs

  Number of epochs is basically the number of times the training dataset is shown to the neural network. Ideally, the more number of times the network sees the dataset, the better its classification accuracy becomes.

- Batch Size

  Batch size is the number of images from the dataset that are fed into the network at a given point of time. Since the training and testing datasets are extremely big in size, it is important to break the datasets down to batch sizes for easy computation. [7]

2.1.2 Images as 4-D Tensors

Convolutional Neural networks take tensors as inputs. Tensors are basically matrices having higher dimensionality. This dimensionality is known as the order of the tensor. It is challenging to visualize and provide spatial representation of a tensor and hence, they have to be dealt with carefully. Every dimension of a tensor holds significant information about the input which in our use case are images. Images are broken down into three dimensions-height, width and depth. The height and width dimensions of the image are self explanatory; the depth is an interesting dimension since it represents the color encoding of images.

Colored images are typically Red-Green-Blue encoded and produce images that are three layers deep. There is a significant body of work done on how to encode different kinds of images. Each of the layers that corresponds to a color encoding is called a channel. The number of channels for a grayscale image is 1. These channels produce feature maps upon convolution that accounts for the fourth dimension in the tensor. Hence, images are not treated as two dimensional but as four dimensional tensors for CNNs.

An image is composed of features spread out non uniformly in all directions. Different layers in a network gather different features of the image and build a map of edges and signals. This map is used to gather common features about images that have the same label and statistically compare with a new image to see whether this image also belongs to the same category or not. Neural networks get better at classifying by learning portions of the feature space and mapping the location of edges and properties of the input. The more number of images the net sees, the better is its accuracy because it maps out a wider variety of features onto its feature map. Convolutional Neural Networks perceive images as 4-dimensional tensors rather than 2 dimensional objects because of the color encoding. Because of RGB encoding, every pixel of the image has a corresponding element in the color stack of the image. Hence, the whole image has a representation in the format of length, width and depth that is fed into the neural network. The aim of the net is to find out which of these elements gives a significant indication or signal that will bring the image closer to its true label. Since per pixel computation of signal strength gets very computationally expensive, layers use square patches of pixels and passes the chosen filter through them. Filters are also usually square shaped and are smaller in size as compared to the image patch at hand. Convolution is carried out by computing the dot product between the two entities. Dot product is high when the two entities have large values and vice versa. In this way, a single value (which is the value of the dot product) can be used to determine whether the underlying image matches the filter pattern expressed by the filter. This value of the dot product is then placed in

another matrix known as an activation map. Since the map contains the dot product, it is dependent upon the size of the patch, the size of the filter and the number of strides that the filter takes. For instance, if the size of the image patch is large, the size of the filter is small and the filter has a small stride as well, it is going to generate more number of dot products and as a result, the activation map is going to be bigger. Each filter is creating an activation map; an every layer is a multitude of such filters. Therefore, each layer has a stack of activation maps, the size of which depends upon the number of filters in that layer. Since high dimensional images take up a lot of computations, CNNs make it easier by downsampling the image at the output of convolutional layers.

The first step of a downsampling layer is max-pooling. Max pooling also takes place patch-wise and chooses the maximum value out of a single patch since it is indicative of the strongest signal that the patch will produce for the mobile filter in the consecutive layers. These maximum values are representative of the whole feature map in a smaller dimensional space. Downsampling hence reduces the amount of storage and the number of computations required in the classification process.

2.1.3 4-Layered CNN

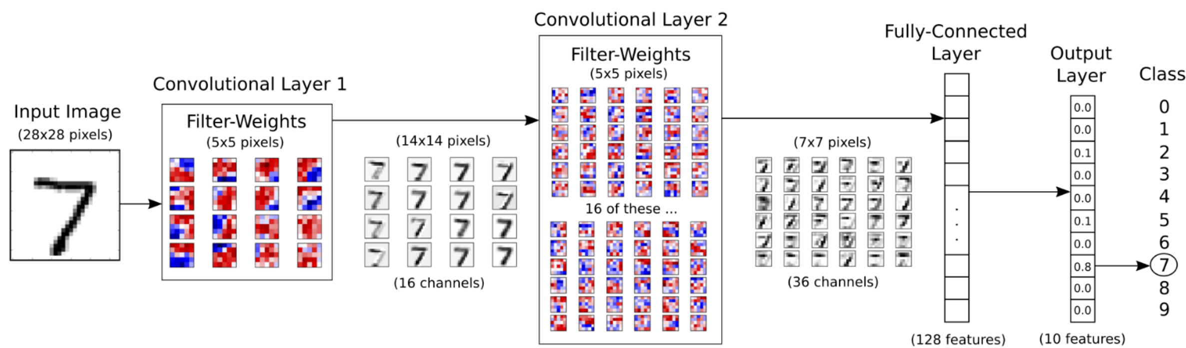

Figure 2.3: Architectural Overview of a CNN

In this Convolutional Neural Network, the input image is first processed using the filter weights. This results in 16 new images, one for each filter in the convolutional layer. These images are down-sampled from 28 x 28 resolution to 14 x 14. These 16 smaller images are then input into the second convolutional layer. There are 16 filter weights for each of these channels along with filter weights for each of the output channels. There are 36 output channels in this layer. As a result, the second layer has 16 x 36 i.e 578 layers. The resulting image is also down-sampled from 14 x 14 to 7 x 7 pixels. The resulting output of the second layer is 36 images of 7 x 7 pixels each. These are then flattened to a single vector of length 7 x 7 x 36 = 1764 which is then input into the first fully connected layer of the network. The fully connected layer has 10 neurons; one for each of the classes in the datasets which is used to determine the class that the image belongs to.

2.1.4 Lenet

Lenet consists of three convolutional layers, two fully connected layers and one softmax classifier. A difference between Lenet from a traditional convolutional neural network is that Lenet only accepts images of dimensions 32 x 32 x C where C, as described above, is the depth of the image categorized by the number of color channels. Since images from MNIST dataset, which has been used in this research are of sizes 28 x 28, they have to be padded along the borders to make them dimensionally compatible with Lenet.

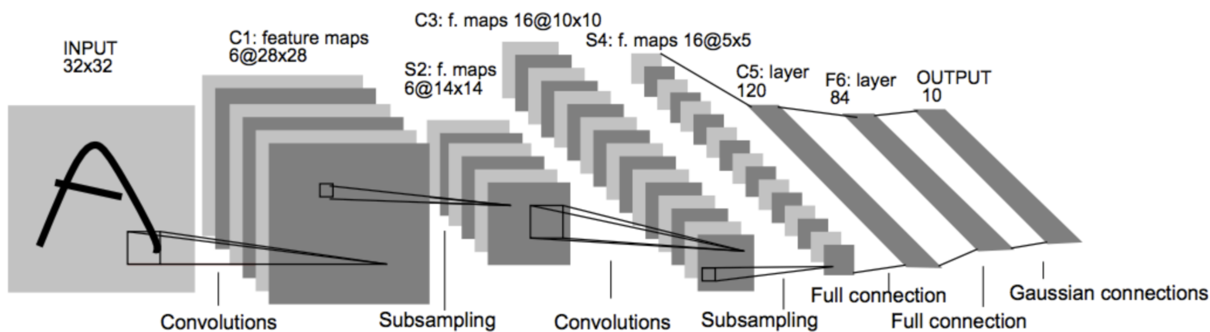

Figure 2.4: Architectural Overview of LeNet [21]

The first layer consists of filters of sizes 5 x 5 with a stride of 1 and 6 feature maps. The resulting output from the first layer has dimensions 28 x 28 x 6 which are then input into the second layer. The second layer applies average pooling to the input with filter size 2 x 2 and stride of 2, transforming the resulting image to dimensions 14 x 14 x 6.

The following layer is designed differently and introduces the concept of sparse connections. The main reason for this is to save computations by keeping the total number of neuron to neuron connections within reasonable bounds. A lot of such inter and intra-neuron connections are redundant and do not contribute to the output in any way. A classification problem arises here, to identify such redundant connections and remove them so as to reduce the complexity of computation. Lenet leverages this and sparsely connects neurons from the second to the third convolutional layers which accounts for its high speed of classification. The fourth layer is another average pooling layer with filter size 2 x 2 and a stride of 2. This layer is identical to the second layer except that it has 6 feature maps, making the output 5 x 5 x 16. The fifth layer is a fully connected layer with 120 feature maps of size 1 x 1. The sixth layer is also a fully connected layer that takes in 120 inputs and outputs 84 units. The final layer is a fully connected softmax output layer that outputs 10 possible values to classify the input images in the MNIST dataset from 0 to 9.

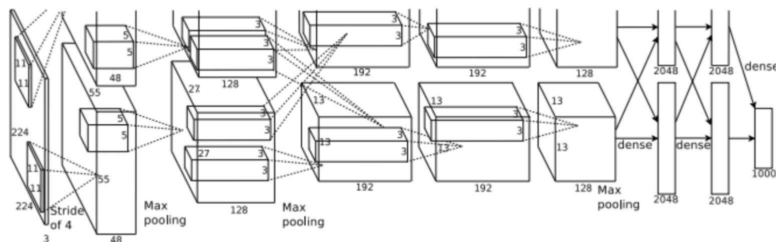

Figure 2.5: Architecture of AlexNet [9]

2.1.5 AlexNet

AlexNet traditionally was built for colored RGB encoded images and therefore if the input is in grayscale format, it needs to be transformed first. AlexNet leverages the use of max-pooling in order to reduce the size of the image that is outputted out of different filters. Downsampling of images makes prominent features to be identified easily. AlexNet as a neural network made a breakthrough contribution to machine learning technology by using ReLU and not saturating activation functions like tanh or sigmoid functions that were being employed in neural networks previously. This proved like using ReLU, CNNs could be trained much more easily.

AlexNet uses Dropout and data augmentation techniques to reduce overfitting which is the phenomenon where neurons absorb a lot of information during training that inhibits their capability to learn more. This is done by making sure that the neural network is shown even more number of images and more importantly, different versions of the same image so that the neural network is able to learn a more versatile set of features. Instead of generating more datasets, this can be done by inverting the same set of images, obtaining the mirror image versions of the image or even reducing the size of the original image to make it more pixelated can drastically increase the size of the data that the neural network learns from. Dropout is a technique of not having certain neurons participate in the backward or forward propagation of data since they add redundancy to the classification process. This simplifies

the otherwise complicated architecture of AlexNet and reduces overfitting as well. Without Dropout, AlexNet suffers from substantial amount of overfitting [9].

2.1.6 Deep Learning using TensorFlow

Tensorflow, developed by Google Brain, is an open-source library for carrying out extensive machine learning tasks. Tensorflow uses Python and a fast C++ backend to carry out the computations. Tensorflow creates computational graphs, also known as a data-flow graph that consist of nodes and edges. Each node on this graph represents a mathematical or logical operation whereas the edges denote the operand which are typically tensors. Tensors are multidimensional arrays since the input to convolutional neural networks typically needs to be converted to a dimension greater than 2.

Tensorflow was employed in this research because it provides abstraction to machine learning programming. Machine learning algorithms are tough to understand and need several quintessential parameters to be defined that programmers find hard to do. Tensorflow ensures that programmers only focus on the high level algorithms implemented on Python and takes care of everything else. Tensorflow API has immense documentation available and is easy to use. Another competitive edge that Tensorflow has over other technologies like PyTorch is that it allows eager execution that enables the evaluation of operations immediately that also calls for easier debugging. Eager execution does not require forming graphs and this lets the user test code in an easier way.

2.2 Datasets

For the intent of this research, only image datasets were considered. MNIST consists of black and white handwritten numbers. CIFAR-10 contains RGB encoded scenic images. Both the datasets have images labeled into 10 distinct categories.

2.2.1 MNIST

MNIST is a dataset of handwritten numbers ranging from 0 to 9 taken from the American Census Bureau employees and American high school students. The dataset has a training set of 60000 images and a test set of 10000 images. The main advantage of using the MNIST dataset is that it does not require extensive preprocessing, formatting and data preparation. MNIST is easy to integrate with Keras and Tensorflow directly through their APIs.

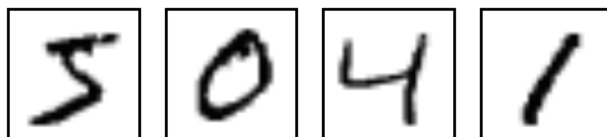

Figure 2.6: Images from the MNIST Dataset

2.2.2 CIFAR-10

Since the CIFAR-10 dataset contains 60000 32 x 32 images across 10 classes with 6000 images per class, it is broken down into six batches named data_batch_1, data_batch_2 and so on in order to ensure that the machine training and testing the dataset does not run out of memory. This is done using the pickle module in Python. Out of the six batches, five are training datasets of 10000 images each and one is a testing dataset. The labels that the images are classified into are airplane (0), automobile (1), bird (2), cat (3), deer (4), dog (5),

frog (6), horse (7), ship (8) and truck (9).

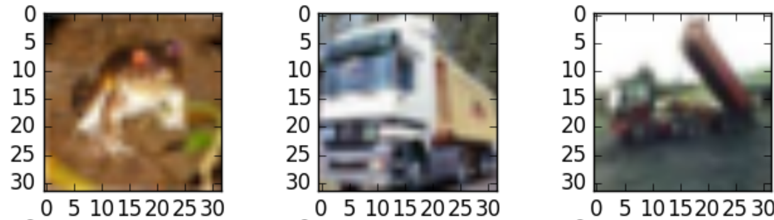

Figure 2.7: Images from the CIFAR-10 Dataset

Preprocessing the data

Data preprocessing is done to get the data ready and transformed to a form that is compatible with the underlying architecture. As mentioned before, the foundation of a lot of deep neural networks do not have support for newer datasets. This step can be avoided when dealing with simpler and smaller datasets like MNIST. Preprocessing is basically done for bigger and more complex datasets like CIFAR-10 and CIFAR-100. The steps involved in preprocessing are normalziation and one-hot encoding.

One-hot-encoding is used to convert the input data into a numerical format since CPUs and GPUs cannot work with categorical data. One-hot encoding offers a two way mapping between the numerical format of the input data and its original format.

Normalization is a part of data preparation for deep learning. Normalization is used for datasets that have a wide range and is used to change the numerical values of a dataset to a common scale without distorting the difference in ranges. A majority of classifiers calculate the distance between two points by their Eucledian distance. Not normalizing data can lead to low accuracy and inability of the neural network to learn even after multiple epochs. While any normalization technique can be employed at this step, the one that is most commonly employed and is also used in this code is min-max normalization. Min-max normalization is also called feature scaling and is used to linearly transform a range of numerical values to a

scale between 0 and 1. Let us assume that 'z' is the normalized value of a member of the set of observed values of x. In that case,
z = (x-min(x))/(max(x) - min(x)) This basically entails that the minimum value of x will be mapped to 0 and the max value of x will be mapped to 1. Hence, the entire range of values in x are mapped in a range from 0 to 1.

2.3 Approximate Computing

Inexact computing is an attractive paradigm when it comes to trading off computation quality with effort expended. Many applications are error-tolerant and can produce results with a slight shift in accuracy and a large reduction in computations. [17]
Approximate computing for machine learning aims to achieve high energy and computational efficiency by modifying network parameters. Despite the resilience of neural networks to slight parameter modifications and noise, approximation must be performed within the tolerance limit of the trained network in order to maintain the inference accuracy of the system. Approximate computing for machine learning can be implemented by reducing the bit-width of network parameters, training the deep learning model with low precision, generating sparsity within the model to remove unnecessary nodes and overfitting etc. In this work, we focus on reducing the bit-width of weights in convolutional and fully connected layers of neural networks.

2.4 Approximation Techniques

Filter weights in the convolutional and fully connected layers are expressed as 32-bit floating point numbers (single precision).

Filter weights were approximated by truncating the bits used to represent the floating point number. Trained neural networks are numerically robust to noise and are able to tolerate minor variations in network parameters. Converting single precision floating point numbers to half precision (16-bit) floating point numbers does not incur significant accuracy loss. However, truncating to bit width lower than minifloat (8-bit) precision shows significant loss in accuracy especially for smaller networks. The approximating techniques explored are as follows-

- Truncating to 6 bits
- Truncating to 5 bits
- Truncating to 4 bits
- Truncating to 3 bits
- Truncating to 2 bits

Bit truncation can be achieved by converting the floating point number to binary and converting bits from the least significant bit (LSB) to the most significant bit (MSB) to 0. This can either be implemented by storing the bit representation of the number in an array or by using bitwise AND operation with the filter weights.

Chapter 3

Motivation

Prior work on approximate computing for machine learning has focused on statically modifying filter weights in deep neural networks during design-time. Once the approximation technique has been chosen, this approach swaps out the original filter weights in the convolutional and fully connected layers with low precision weights. Since the original filter weights are learned during the training phase of the network, they are set to achieve the highest possible inference accuracy for the filter. However if those filters are modified, the accuracy is bound to decrease.

These findings provide motivation for our research. Our work focuses on exploring dynamic filter approximation techniques that during run-time, are able to compute which filter IDs in a particular layer are generating the most error upon approximation for a given input. The corresponding filters are changed back to the original, full precision filter weights.

Multiplication Error(ME) is an important metric that is used to determine which filters to approximate and which ones not to. Section 3.1 describes ME and how to fast-calculate it during run-time.

3.1 Multiplication Error

Multiplication Error (ME) is computed at run-time in order to determine which filters can be approximated to less precision weights without significantly impacting the inference accuracy

of the network. The sections below describe how to calculate ME for the convolutional and fully connected layers.

3.1.1 ME for Convolutional Layers

ME for convolutional layers is calculated by subtracting the approximated filter weights from the original filter weights, convolving the resulting matrix with the input image and calculating the grand sum of the new matrix. In order to fast-calculate ME at run-time, the sum of all elements in the filter matrix is multiplied with the sum of all elements in the input matrix. Given below is the mathematical proof of how the fast calculation method is equivalent to ME.

Symbols and Abbreviations

- * ... symbol for convolution
- $\times$... symbol for multiplication
- w ... symbol for original filter weights
- w' ... symbol for approximated filter weights
- I ... symbol for input
- M ... symbol for filter dimensions
- A ... symbol for the matrix representation of w - w'

For calculating the convolution between the square filter and the image, we consider a subset of the image that has the same dimensions as the filter. ME can be defined as-

$$ME = \sum_{m=1}^{x} \sum_{n=1}^{y} [w * I - w' * I]_{mn} \tag{3.1}$$

Using the distributive property of convolution, [1]

$$ME = \sum_{m=1}^{x} \sum_{n=1}^{y} [(w - w') * I]_{mn} \tag{3.2}$$

Using the symbols declared above, we get

$$ME = \sum_{m=1}^{M} \sum_{n=1}^{M} [A * I]_{mn} \tag{3.3}$$

Let the convolution between A and B be S.

$$S(i,j) = \sum_{m=1}^{M} \sum_{n=1}^{M} [A(m,n) \times I(i-m, j-n)] \tag{3.4}$$

where 1 $\leq$ i,j $\leq$ 2M.

Since ME is the sum of all the elements in the convolution matrix,

$$ME = S(1,1) + S(1,2) + ...S(M,M)$$

$= \sum_{m=1}^{M} \sum_{n=1}^{M} [A(m,n) \times I(1-m, 1-n)] + \sum_{m=1}^{M} \sum_{n=1}^{M} [A(m,n) \times I(1-m, 2-n)] + ...+$
$\sum_{m=1}^{M} \sum_{n=1}^{M} [A(m,n) \times I(M-m, M-n)]$

[1] The distributive property of the convolution operator states that for inputs g, h_1, h_2,
g * (h_1 + h_2) = g * h_1 + g * h_2

$= \sum_{m=1}^{M} \sum_{n=1}^{M} A(m,n) \sum_{m=1}^{M} \sum_{n=1}^{M} [I(1-m, 1-n)] + I(1-m, 2-n)] + ... + I(M-m, M-n)] (3.5)$

Expanding the second part of the equation gives

$$ME = \sum_{m=1}^{M} \sum_{n=1}^{M} A(m,n) \times [I(0,0) + I(0,1) + ...I(M,M)] \tag{3.6}$$

After discarding invalid matrix elements, we get

$$ME = \sum_{m=1}^{M} \sum_{n=1}^{M} A(m,n) \times [I(1,1) + I(1,2) + ...I(M,M)] \tag{3.7}$$

This can also be written as

$$ME = \sum_{m=1}^{M} \sum_{n=1}^{M} A(m,n) \times \sum_{m=1}^{M} \sum_{n=1}^{M} [I(m,n)] \tag{3.8}$$

The first part of equation represents the sum of all elements in matrix A and the second part of the equation represents the sum of all elements in matrix B.

ME can be written as

$$ME = \sum_{m=1}^{M} \sum_{n=1}^{M} (w - w')_{mn} \times \sum_{m=1}^{M} \sum_{j=1}^{M} I_{mn} \tag{3.9}$$

Therefore, we fast-calculate ME by finding the sum of all elements in the filter and multiplying that with the sum of all elements in the subset of the image that the filter would glide over during convolution.

3.1.2 ME for Fully Connected Layers

ME for fully connected layers is calculated by subtracting the approximated filter weights from the original filter weights, performing matrix multiplication between the difference and the input, and calculating the grand-sum of the resulting matrix. In order to fast-calculate ME at run-time, we first compute the sum of all elements in a column of the filter and multiply that with the sum of all elements in the corresponding row of the input. This process is repeated for every row and column of the filter and the image. ME is given by the sum of all the products obtained.

Symbols and Abbreviations

- w ... symbol for original filter weights
- w' ... symbol for approximated filter weights
- I ... symbol for input
- M ... symbol for filter dimension along x axis
- N ... symbol for filter dimension along y axis
- P ... symbol for image dimension across x axis
- Q ... symbol for image dimension across y axis
- A ... symbol for the matrix representation of w - w'

For matrix multiplication in the fully connected layers, the column dimension of the filter matrix must be equal to the row dimension of the image.

$$N = P \tag{3.10}$$

This can be achieved by using Numpy's reshape function. Let the resulting matrix post matrix multiplication be denoted by matrix C. Each element in C is computed as -

$$C_{ij} = \sum_{k=1}^{N} (w - w')_{ik} I_{kj} \tag{3.11}$$

ME is the sum of all elements in C that can be denoted by -

$$ME = \sum_{i=1}^{M} \sum_{j=1}^{Q} \sum_{k=1}^{N} (w - w')_{ik} I_{kj} \tag{3.12}$$

Using the symbols declared above,

$$ME = \sum_{i=1}^{M} \sum_{j=1}^{Q} \sum_{k=1}^{N} A(i,k) I(k,j) \tag{3.13}$$

Using the associative property of multiplication, we get

$$ME = \sum_{i=1}^{M} \sum_{k=1}^{N} A(i,k) \sum_{j=1}^{Q} I(k,j) \tag{3.14}$$

$$ME = \sum_{k=1}^{N} \left\{ \left(\sum_{i=1}^{M} A(i,k) \right) \left(\sum_{j=1}^{Q} I(k,j) \right) \right\} \tag{3.15}$$

Therefore, ME can be written as

ME = sum of [(sum of all elements in column i of A)×(sum of all elements in row j of B)] for all the columns of A and all the rows of B. We fast calculate matrix multiplication during run-time by using this method.

3.2 Analysis of Time Complexity

For an image of size (M×N) and a convolution mask of size (k×k), the computational complexity of computing ME for convolution layers is O(MNkk). On the other hand, the fast calculation technique for calculating ME for convolutional layers results in a complexity of O(MN) + O(kk).

For matrix multiplication between the first matrix of size (M×N) and the second matrix of size (N×P), the computational complexity of matrix multiplication is O(MNP). The fast calculation method results in complexity O(NM + NQ) operations. Here, the term operations is used to refer to the number of multiplications and additions in each layer.

The number of operations in calculating ME for convolutional and fully connected layers for all three networks is shown below.

LAYER	ORIGINAL NUMBER OF OPERATIONS	NUMBER OF OPERATIONS DUE TO FAST CALCULATION
CONVOLUTION LAYER 1 (5 x 5 x 1)	19,600	809
CONVOLUTION LAYER 2 (5 x 5 x 16)	313,600	12,944
FULLY CONNECTED LAYER 1 (1764 x 128)	3,762,000	48,784
FULLY CONNECTED LAYER 2 (128 x 10)	177,020,928	226,560

Figure 3.1: Number of operations to calculate ME for 4-layered CNN

For a simple CNN, the proposed technique performs 289,097 more operations at run-time as compared to statically replacing at design-time.

LAYER	ORIGINAL NUMBER OF OPERATIONS	NUMBER OF OPERATIONS DUE TO FAST CALCULATION
CONVOLUTION LAYER 1 (5 x 5 x 1)	19,600	809
CONVOLUTION LAYER 2 (5 x 5 x 6)	117,600	4,854
FULLY CONNECTED LAYER 1 (400 x 120)	37,632,000	48,784
FULLY CONNECTED LAYER 2 (120 x 84)	7,902,720	10,864
FULLY CONNECTED LAYER 3 (84 x 10)	658,560	1,624

Figure 3.2: Number of operations to calculate ME for LeNet

For LeNet, the proposed technique performs 66,935 more operations at run-time as compared to statically replacing at design-time.

LAYER	ORIGINAL NUMBER OF OPERATIONS	NUMBER OF OPERATIONS DUE TO FAST CALCULATION
CONVOLUTION LAYER 1 (3 x 3 x 1)	7056	793
CONVOLUTION LAYER 2 (3 x 3 x 64)	451,584	50,752
CONVOLUTION LAYER 3 (3 x 3 x 128)	903,168	101,504
FULLY CONNECTED LAYER 1 (4096 x 1024)	3,288,334,336	4,195,088
FULLY CONNECTED LAYER 2 (1024 x 1024)	822,083,584	1,049,360
FULLY CONNECTED LAYER 3 (1024 x 10)	8,028,160	11,024

Figure 3.3: Number of operations to calculate ME for AlexNet

For AlexNet, the proposed technique performs 5,408,521 more operations at run-time as compared to statically replacing at design-time.

3.3 Static and Dynamic Multiplication Error Analysis

Static ME analysis is carried out by calculating ME for each filter in a layer for all the images in the training dataset and averaging the per-filter ME for the total number of images. This is depicted in the pipeline in figure 3.4

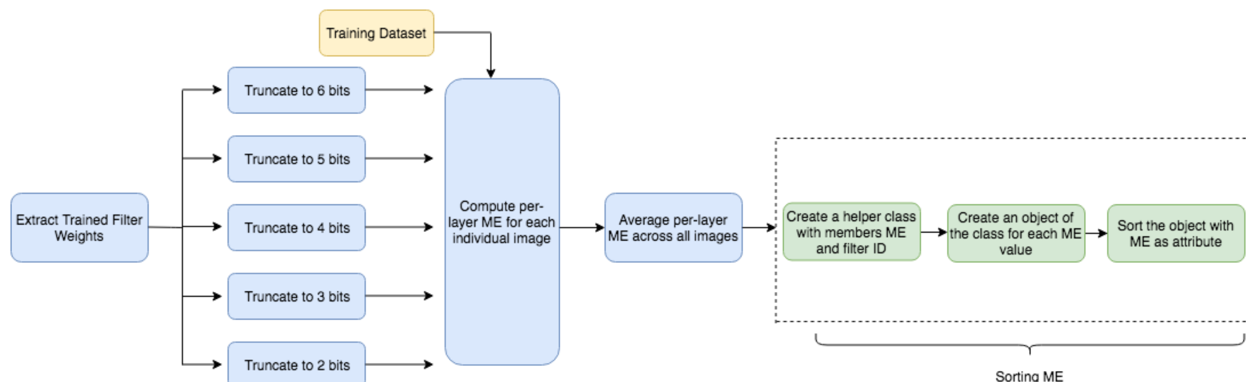

Figure 3.4: Pipeline for static ME analysis during training

This step is carried out to find out the average ME for each filter in a layer.

Dynamic ME analysis is carried out by calculating the ME for each filter in a layer, sorting the ME from lowest to highest for each input and then averaging the sorted per-input ME across all the inputs. The pipeline for dynamic ME analysis can be seen in figure

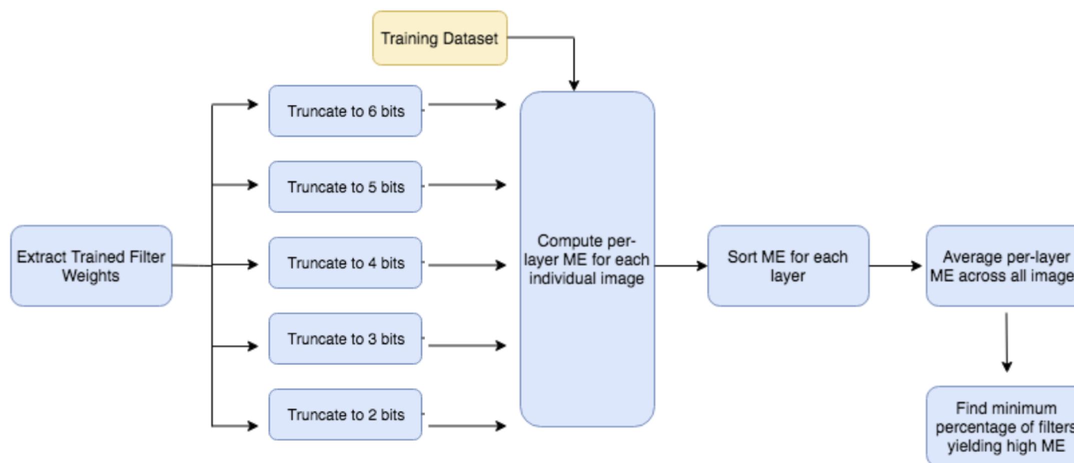

Figure 3.5: Pipeline for dynamic ME analysis during training

This step is carried out to find the percentage of filters that on an average yield high Multiplication Errors.

The static and dynamic ME approximation techniques are shown as follows. The static approximation technique is denoted by the color red and the dynamic approach is denoted by black. Dynamic and static ME analysis was done for both the datasets, for all approximation techniques and for all the neural networks considered in this work. An example demonstrating this is shown below. The example is for AlexNet using the MNIST dataset. The approximation technique used is truncating to 2 bits.

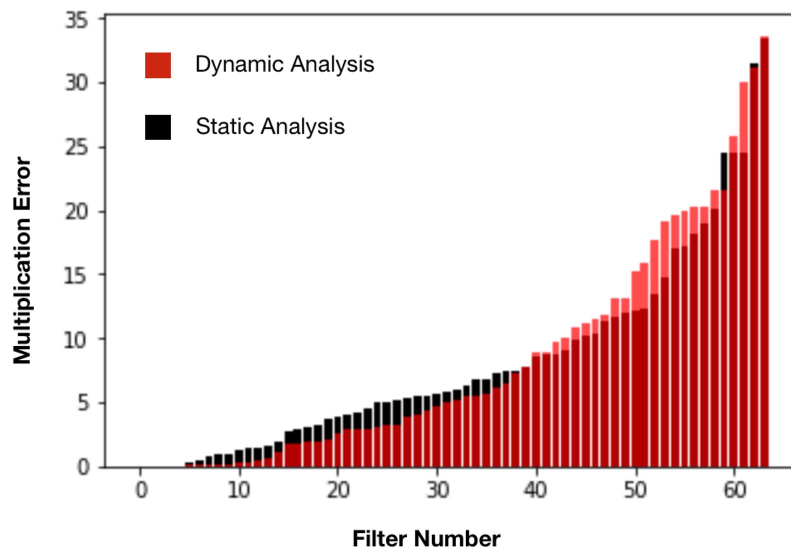

Figure 3.6: Static and Dynamic Analysis for Convolutional Layer 1

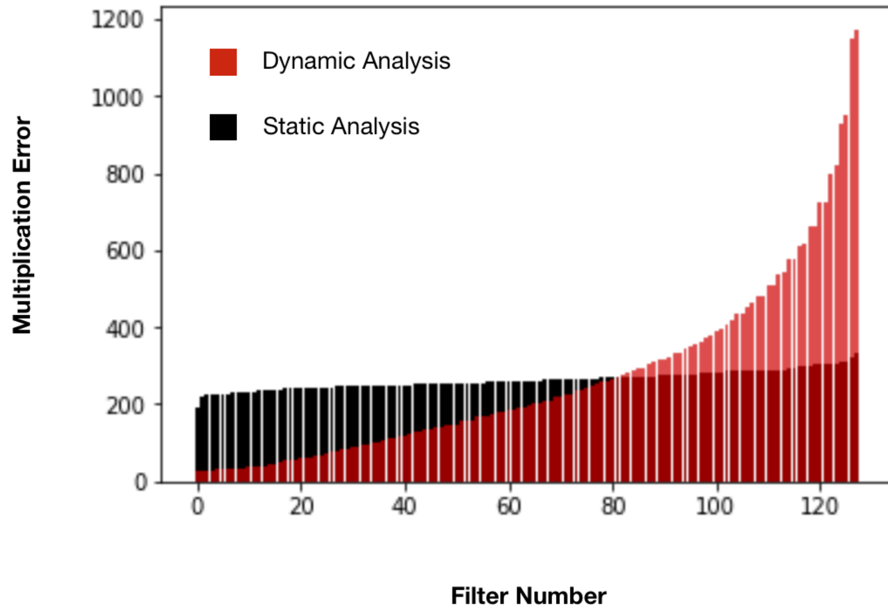

Figure 3.7: Static and Dynamic Analysis for Convolutional Layer 2

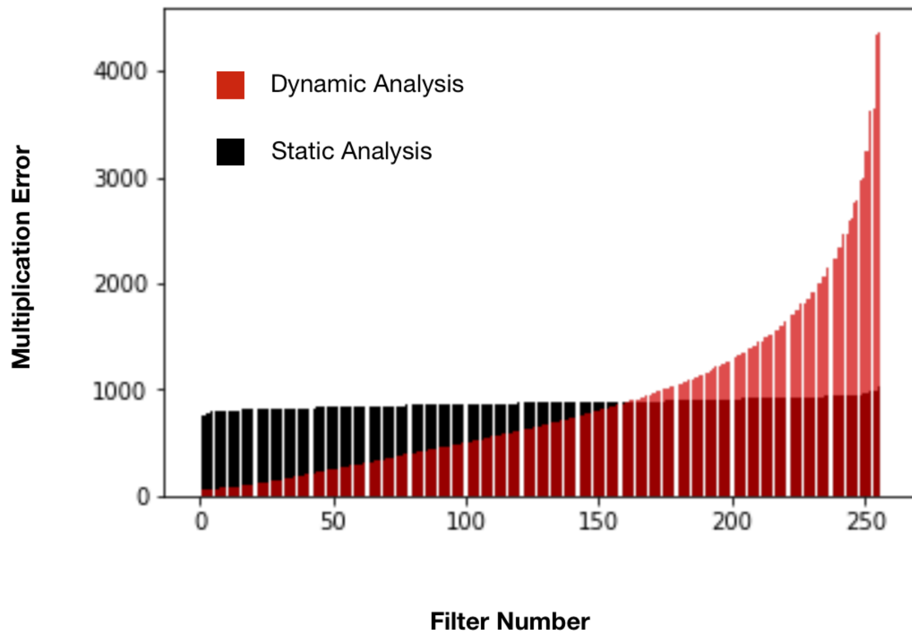

Figure 3.8: Static and Dynamic Analysis for Convolutional Layer 3

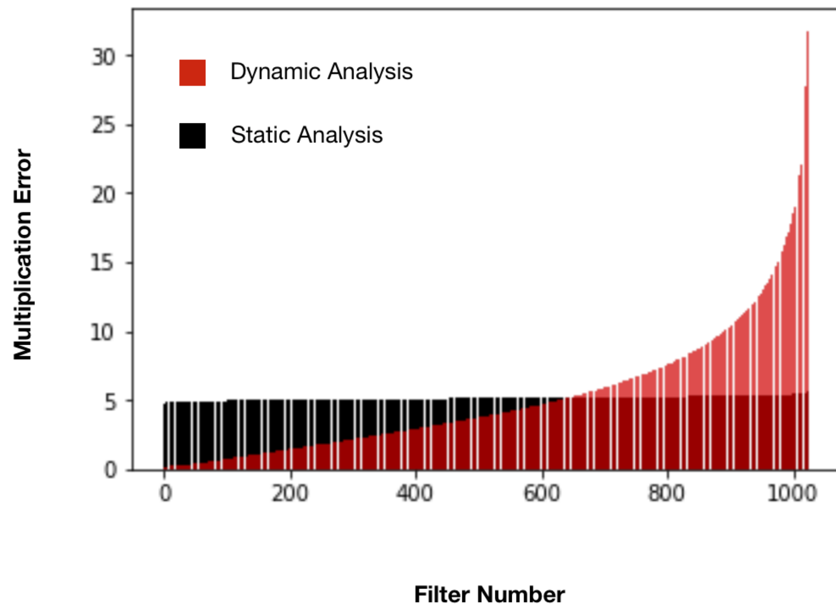

Figure 3.9: Static and Dynamic Analysis for Fully Connected Layer 1

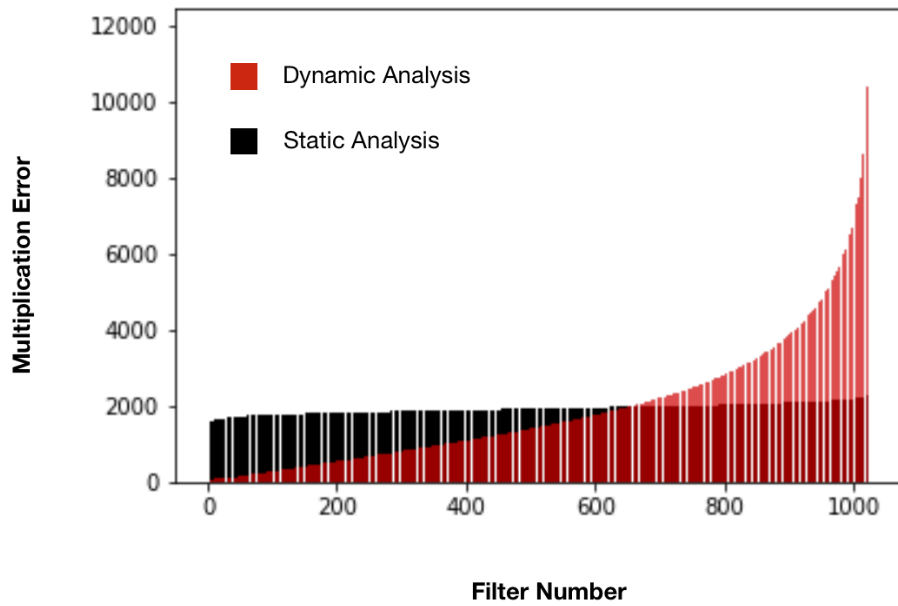

Figure 3.10: Static and Dynamic Analysis for Fully Connected Layer 2

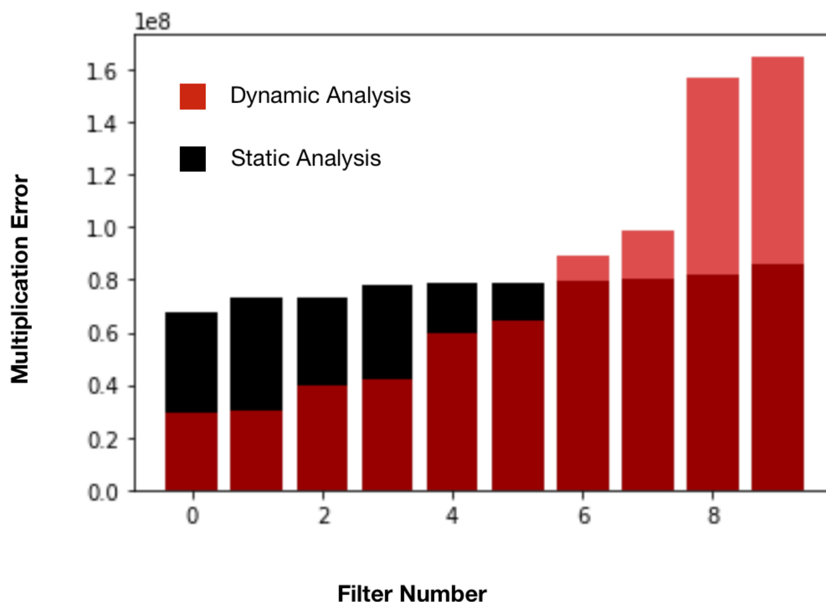

Figure 3.11: Static and Dynamic Analysis for Fully Connected Layer 3

For each of the above layers, dynamic ME shows a sharp increase as compared to static ME. This can be attributed to sorting the per-input ME before averaging it across all the inputs for dynamic analysis. As a result, the highest ME for each input is concentrated in that area. If the percentage of filters contributing to the sharp skew in ME are not approximated, the multiplication error for those filters would be reduced to 0. For all the datasets and networks explored, it was observed that at most 20% of the total number of filters contribute to the skew in ME values for dynamic analysis. Since high ME value accounts for low inference accuracy for that input image, changing the approximated filters that yield high ME to original filters will result in higher inference accuracy. Therefore for each input image, the dynamic technique converts 20% of filters corresponding to highest ME to the original full precision filters. This is done in order to maintain the inference accuracy of the system after approximation.

Chapter 4

Methodology

This section describes the steps and methodology followed for carrying out the experiments. Section 4.1 explains installation requirements for Jupyter Notebook, Tensorflow and several other packages that were made use of in the code-base. Section 4.2 explains the tools and steps required to test and train models as well as how to create Tensorflow graphs and run sessions. It also provides code snippets for the various approximation techniques used throughout this work. Section 4.3 discusses the implementation of the approximation techniques as well as the testing pipeline for dynamic and static filter selection techniques.

4.1 Setup

The code is assembled and compiled on Jupyter Notebook which is an open-source web application that allows users to create and share documents that contain live code, equations, visualizations and narrative text. Jupyter Notebook is mainly used for data cleaning and transformation, numerical simulation, statistical modeling, data visualization and machine learning. [11]

The first step is to install Jupyter Notebook. This can be done by installing Anaconda which is a free and open-source distribution of the Python for scientific computing that makes package management and deployment easier. Once the latest version of Anaconda, also called conda is installed, Jupyter Notebook can be accessed by typing "jupyter notebook"

on the terminal/ command prompt.

Tensorflow can also be installed using Python's pip package manager. Since the newer versions of TensorFlow are more compatible with Python3 and not Python2, it is advised to use virtual environments when installing Python3 and TensorFlow so as to maintain control over Python and other library versions. This can be done by the following set of commands. Let us assume that the name of the virtual environment is "cnn".

```
# setting up a new virtual environment
python -m venv cnn
# activating the virtual environment
source cnn/bin/activate
#for additional packages including tensorflow
pip3 install <package-name>
```

Several Python packages are used throughout the code base such as numpy, matplotlib, sklearn, datetime, mnist, sklearn, scipy, os, math that can be installed using the above code snippet.

4.2 Deep Learning on TensorFlow

The architecture of a neural network varies from network to network and is heavily dependent on the kind of input that the network receives. The basic process pipeline for most neural networks from installing the dataset to training the neural network can be depicted as follows- Figure 4.2 explains how to load the MNIST dataset using the Tensorflow API. After the dataset has been loaded, data exploration is conducted. The MNIST dataset is split into

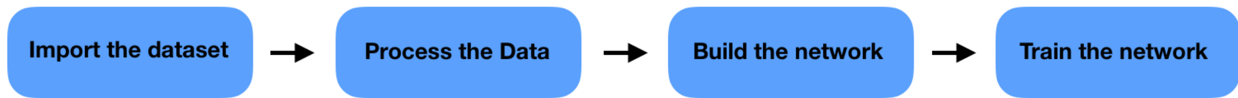

Figure 4.1: Training pipeline for classification

training set, test set and validation set. Validation test is used along with the test set during the training phase. This set is used to minimize overfitting. This is done by making sure that any increase in accuracy caused by weight and bias optimization for the test set is equivalently reflected in a dataset that the neural network has not seen yet. The validation set does not lead to any backpropagation of errors and weight/bias modification.

The MNIST data-set is about 12 MB and will be downloaded automatically if it is not located in the given path.

```
In [9]: from tensorflow.examples.tutorials.mnist import input_data
        data = input_data.read_data_sets('data/MNIST/', one_hot=True)

Extracting data/MNIST/train-images-idx3-ubyte.gz
Extracting data/MNIST/train-labels-idx1-ubyte.gz
Extracting data/MNIST/t10k-images-idx3-ubyte.gz
Extracting data/MNIST/t10k-labels-idx1-ubyte.gz
351957
```

The MNIST data-set has now been loaded and consists of 70,000 images and associated labels (i.e. classifications of the images). The data-set is split into 3 mutually exclusive sub-sets. We will only use the training and test-sets in this tutorial.

```
In [10]: print("Size of:")
         print("- Training-set:\t\t{}".format(len(data.train.labels)))
         print("- Test-set:\t\t{}".format(len(data.test.labels)))
         print("- Validation-set:\t{}".format(len(data.validation.labels)))

Size of:
- Training-set:		55000
- Test-set:		10000
- Validation-set:	5000
```

Figure 4.2: Steps to load the MNIST dataset on Tensorflow

CIFAR-10 is not supported by the Tensorflow API and has to be downloaded into the notebook. Since CIFAR-10 is a big dataset, images are split into batches. As mentioned previously, batch size refers to the number of images in a batch. Since training happens batch-wise, if the network is big (like AlexNet), the batch size should be set to a small value to prevent out of memory errors.

Figure 4.4 shows the labels loaded for the dataset into Tensorflow.

```
In [1]: from urllib.request import urlretrieve
        from os.path import isfile, isdir
        from tqdm import tqdm
        import tarfile

        cifar10_dataset_folder_path = 'cifar-10-batches-py'

        class DownloadProgress(tqdm):
            last_block = 0

            def hook(self, block_num=1, block_size=1, total_size=None):
                self.total = total_size
                self.update((block_num - self.last_block) * block_size)
                self.last_block = block_num

        """
            check if the data (zip) file is already downloaded
            if not, download it from "https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz" and save as cifar-10-python.tar.g
        """
        if not isfile('cifar-10-python.tar.gz'):
            with DownloadProgress(unit='B', unit_scale=True, miniters=1, desc='CIFAR-10 Dataset') as pbar:
                urlretrieve(
                    'https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz',
                    'cifar-10-python.tar.gz',
                    pbar.hook)

        if not isdir(cifar10_dataset_folder_path):
            with tarfile.open('cifar-10-python.tar.gz') as tar:
                tar.extractall()
                tar.close()
```

Figure 4.3: Steps to download the CIFAR-10 dataset on Tensorflow

```
In [3]: def load_label_names():
            return ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']

In [4]: def load_cfar10_batch(cifar10_dataset_folder_path, batch_id):
            with open(cifar10_dataset_folder_path + '/data_batch_' + str(batch_id), mode='rb') as file:
                # note the encoding type is 'latin1'
                batch = pickle.load(file, encoding='latin1')

            features = batch['data'].reshape((len(batch['data']), 3, 32, 32)).transpose(0, 2, 3, 1)
            labels = batch['labels']

            return features, labels
```

Figure 4.4: Steps to load labels for the CIFAR-10 dataset

Data exploration is an important step because having knowledge of the dataset helps formulate the data preprocessing steps. For instance, if the input image has dimensions 28 x 28 but the first convolutional layer only allows images of size 32 x 32, the preprocessing part of the code can handle padding the original image to meet the desired dimension requirements.

```
# Explore the dataset
batch_id = 3
sample_id = 9999
display_stats(cifar10_dataset_folder_path, batch_id, sample_id)
```

```
Stats of batch #3:
# of Samples: 10000

Label Counts of [0](AIRPLANE) : 994
Label Counts of [1](AUTOMOBILE) : 1042
Label Counts of [2](BIRD) : 965
Label Counts of [3](CAT) : 997
Label Counts of [4](DEER) : 990
Label Counts of [5](DOG) : 1029
Label Counts of [6](FROG) : 978
Label Counts of [7](HORSE) : 1015
Label Counts of [8](SHIP) : 961
Label Counts of [9](TRUCK) : 1029

Example of Image 9999:
Image - Min Value: 3 Max Value: 242
Image - Shape: (32, 32, 3)
Label - Label Id: 1 Name: automobile
```

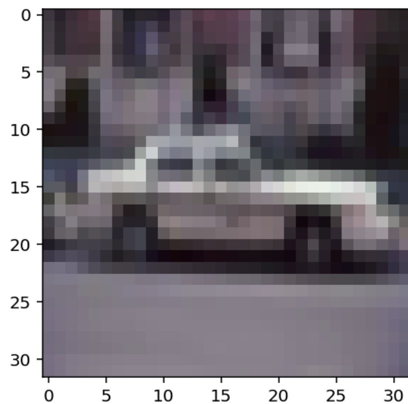

Figure 4.5: Dataset exploration for CIFAR-10

Building the network involves forming the convolutional layers, the fully connected layers, the ReLU activation layers, max pooling layers etc. Once the network has been formed, the metrics for computing classification accuracy are defined. This metric basically compares the predicted label of the image at hand, with the true label associated with that image. The performance measure used in classification is called cross entropy. Cross entropy is a continuous functions that remains constantly positive and if the predicted label matches the

true label, it drops down to zero. The neural network hence optimizes its weights and biases in order to minimize the cross entropy so that it remains close to zero. Tensorflow has a built-in function for calculating cross entropy which is defined as follows.

```
cross_entropy = tf.nn.softmax_cross_entropy_with_logits{
                (logits=logits,labels=y_true)
                }
```

Here 'logits' refers to the last layer of the neural network that estimates how likely it is that the input image belongs to one of the classes. Cross entropy is a useful measure of how well the network performs on each of the input images but for using this metric to guide how the network optimizes itself, this value needs to be converted into a scalar that is defined as cost. 'tf' is the name of the Tensorflow module imported into the notebook.

```
cost = tf.reduce_mean(cross_entropy)
```

Any optimizing technique can be employed to fully optimize the network and use back-propagation to modify the filter weights and biases. Most of the neural networks used in Tensorflow make use of class Adam Optimizer which inherits from the Optimizer class. This optimizer implements a state of the art gradient-based optimization algorithm called Adam. This is defined as follows.

```
optimizer = tf.train.AdamOptimizer{(learning_rate=1e-4).minimize(cost)}
```

A few more performance measures are defined to display training progress to the user. These are defined by the following code snippet.

```
correct_prediction = tf.equal(y_pred_cls, y_true_cls)
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Here, correct_prediction is a vector of booleans that determines whether the label predicted by the network coincides with the true label of the input. The accuracy parameter typecasts correct_prediction so that a boolean true becomes a 1 and a boolean false translates to a 0. It then calculates the average of the numbers obtained. Due to a large number of images in the training set, gradient of the model is calculated using a small subset of images for each iteration. Multiple iterations ensure higher inference accuracy.

```python
def print_test_accuracy(show_example_errors=False,
                        show_confusion_matrix=False):

    # Number of images in the test-set.
    num_test = len(data.test.images)

    # Allocate an array for the predicted classes which
    # will be calculated in batches and filled into this array.
    cls_pred = np.zeros(shape=num_test, dtype=np.int)

    # Now calculate the predicted classes for the batches.
    # We will just iterate through all the batches.
    # There might be a more clever and Pythonic way of doing this.

    # The starting index for the next batch is denoted i.
    i = 0

    while i < num_test:
        # The ending index for the next batch is denoted j.
        j = min(i + test_batch_size, num_test)

        # Get the images from the test-set between index i and j.
        images = data.test.images[i:j, :]

        # Get the associated labels.
        labels = data.test.labels[i:j, :]

        # Create a feed-dict with these images and labels.
        feed_dict = {x: images,
                     y_true: labels}

        # Calculate the predicted class using TensorFlow.
        cls_pred[i:j] = session.run(y_pred_cls, feed_dict=feed_dict)

        # Set the start-index for the next batch to the
        # end-index of the current batch.
        i = j

    # Convenience variable for the true class-numbers of the test-set.
    cls_true = data.test.cls

    # Create a boolean array whether each image is correctly classified.
    correct = (cls_true == cls_pred)

    # Calculate the number of correctly classified images.
    # When summing a boolean array, False means 0 and True means 1.
    correct_sum = correct.sum()

    # Classification accuracy is the number of correctly classified
    # images divided by the total number of images in the test-set.
    acc = float(correct_sum) / num_test

    # Print the accuracy.
    msg = "Accuracy on Test-Set: {0:.1%} ({1} / {2})"
    print(msg.format(acc, correct_sum, num_test))

    # Plot some examples of mis-classifications, if desired.
    if show_example_errors:
        print("Example errors:")
        plot_example_errors(cls_pred=cls_pred, correct=correct)

    # Plot the confusion matrix, if desired.
    if show_confusion_matrix:
        print("Confusion Matrix:")
        plot_confusion_matrix(cls_pred=cls_pred)
```

Figure 4.6: Code snippet for computing test accuracy

The calculation for test dataset accuracy is carried out as shown in 4.6:

Before any optimization iteration, the accuracy of the test set is very low since this tests the dataset against randomly defined Tensorflow variables that serve as weights and biases without any optimization or changes. As the number of iterations increase, the weights and biases are modified to allow the network to reach optimum accuracy. For example, 0 iterations result in an accuracy of 12.7%, 100 iterations result in an accuracy of 67.25% and 1000 iterations result in an accuracy of 93% for MNIST dataset on a simple CNN. As shown

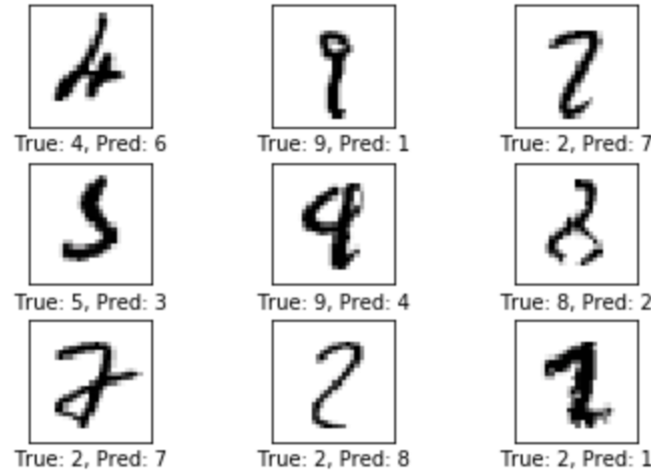

Figure 4.7: Accuracy on the Test Set after 10000 iterations

in Figure 4.7, 10000 iterations result in an accuracy of 98.9%. Post this, the increase in accuracy is not as sharp as seen previously. This is indicative of the fact that the network is optimized.

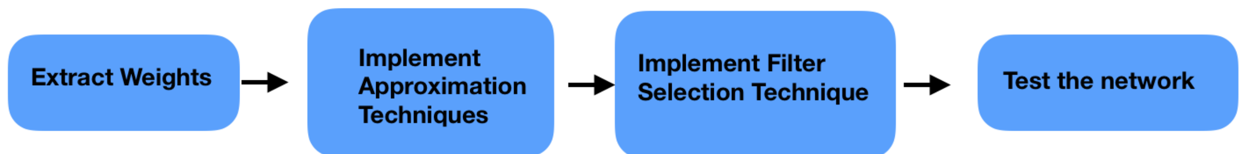

Figure 4.8: Testing pipeline for classification

For the testing phase of the research, the first task is to extract filter weights for each layer. The process pipeline for testing can be seen in Figure 4.11. The implementation of all the deep neural networks needs to be modified in such a way that the function that creates the neural network architecture either returns the individual filter weights or the weights are initialized outside of the function so that they can be modified throughout the program. For ease of use, the weights are stored in a list; one list stores the convolutional layer weights

and the other stores the weights of the fully connected layer.

Once the weights have been obtained and stored, the approximation techniques can be applied on them. Since raw weights are Tensorflow Variables, they first need to be converted into an array format so that they can be modified. This can be done by the following code.

```
session = tf.Session()
# weights_conv1 is the Tensorflow variable storing
# filter weights for convolutional layer 1.
weights_conv1 = session.run(weights_layer1)
```

A Tensorflow Session is used to execute Tensorflow operations. It basically evaluates objects by encapsulating a Tensorflow environment. For ease of use, the whole code should be enclosed within a single session. This makes variable use easier since variables declared outside of a session are not recognized within the session.

Once the filter weights of every layer have been extracted as arrays, they are approximated.

4.3 Implementation of Approximation Techniques

Approximation is implemented by converting the decimal part of the filter weights to a binary format. Every bit of the binary representation is stored as part of an array. For truncating n number of decimal places, the binary array is traversed from least significant bit to the most significant bit and n array places are changed to 0. Therefore, even though the filter weight is still a 32 bit floating point number, it is converted to a format that can be represented in n number of bits.

Once the approximate filters have been assigned to the original filters in the convolutional and fully connected layers, the third step of the testing pipeline which is filter selection

```
import math
import numpy as np
def dec2bin(f):
    if f >= 1:
        g = int(math.log(f, 2))
    else:
        g = -1
    h = g + 1
    ig = math.pow(2, g)
    st = ""
    while f > 0 or ig >= 1:
        if f < 1:
            if len(st[h:]) >= 23:
                break
        if f >= ig:
            st += "1"
            f -= ig
        else:
            st += "0"
        ig /= 2
    st = st[:h] + "." + st[h:]
    return st
```

Figure 4.9: Code snippet for converting decimal number to binary string

technique, is implemented.

4.3.1 Filter Selection Techniques

During run-time, ME is fast calculated using the checksum technique. As described in the previous section, 20% of the filters contributing to the highest ME for a given input image are changed back to full precision weights. Once ME is calculated, the next step is to sort it, while keeping track of the corresponding filter ID.

```
def convertToFloat(listnew):
    summa = 0
    anchor = -1
    for i in listnew:
        num = (int)(i)
        if(num==1):
            summa = summa + 2**anchor
        anchor = anchor -1
    return summa

def convertThis(num):
    isNeg = 0
    if(num<0):
        num = num*-1
        isNeg = 1

    mystring = dec2bin(num)

    stringnew = mystring.split(".")
    listnew =[]
    anchor = 6 ## truncating 16 mantissa
    for i in stringnew[1]:
        if anchor!=0:
            anchor = anchor -1
            listnew.append(i)
        else:
            break
    if isNeg==1:
        return -1*convertToFloat(listnew)

    return convertToFloat(listnew)
```

Figure 4.10: Code to convert the string back to float

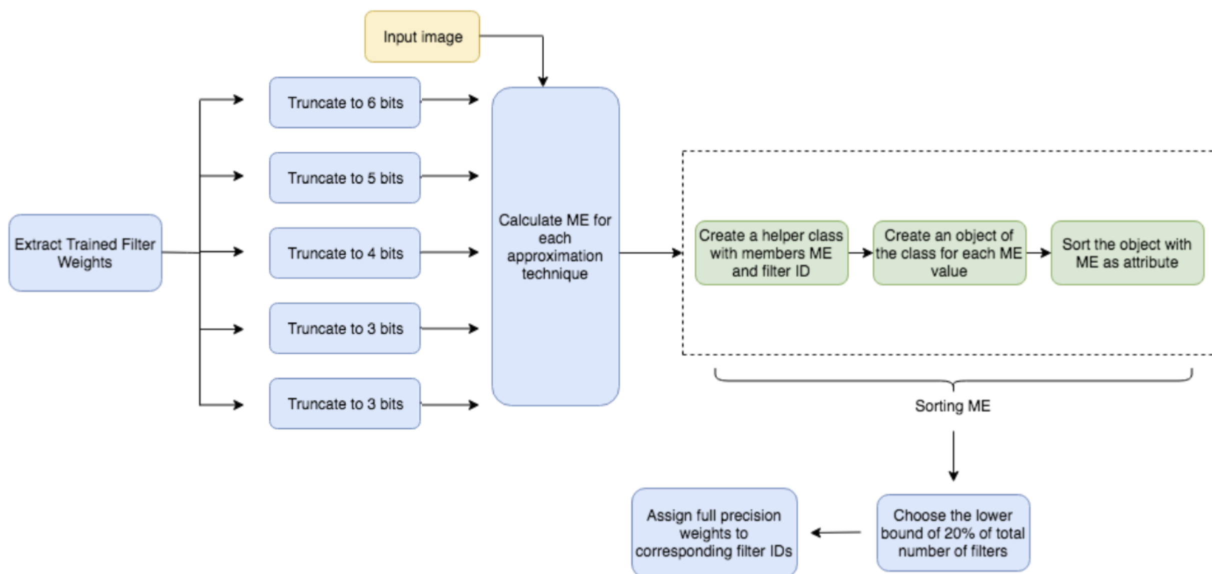

Figure 4.11: Pipeline for dynamic filter selection

In order to keep track of filter IDs associated with ME post sorting, a helper class is declared. The members of this class are the multiplication error and filter ID. Each object of this class in the main program has the same attributes as the one defined in the parent class. A list of these objects is sorted by error. As a result even though the index of the error in the list is changed after sorting, the associated filter ID is still maintained. An example of this technique has been shown below.

```
class helper:
    error = 0
    filterID = 9999
```

The above code snippet declares the helper class to arrange ME in a sorted order whole still retaining filter IDs. It is important to note that error and filterID values need to be pre-declared and their assigned values in the class hold no meaning since they are going to be reinitialized for each instance.

```
MEFinalList = []
index = 0
for i in MEConv1:
    object1 = helper()
    object1.error = i
    object1.filterID = index
    index = index + 1
    MEFinalList.append(object1)
```

In the above code snippet, MEConv1 is a list of Multiplcation Error values for Convolutional Layer 1. This step stores the ME values obtained for each layer as an object of the helper

class and assigns them corresponding filter IDs. At this stage, the ME values are sequential and not sorted. Therefore, the filter ID can be assigned as an integer that increments with every for loop cycle.

```
for i in MEFinalList:
    print(i.error)

2.1183734776286655e-06
2.1872340414574864e-06
2.4136038405231373e-06
2.4549041393129303e-06
2.5202346967034826e-06
2.5726209014464987e-06
2.893484883088604e-06
2.965334559803523e-06
3.0548921995432466e-06
3.068236053422879e-06
3.193659268845295e-06
3.2214727167456657e-06
3.2902596490202997e-06
3.4442951903201902e-06
3.4472545394237388e-06
3.6050607059223694e-06
```

Figure 4.12: List of sorted ME values

```
for i in MEFinalList:
    print(i.filterID)

10
1
0
6
9
7
13
14
4
3
15
11
8
12
2
5
```

Figure 4.13: List of Filter IDs corresponding to the sorted ME values shown in 4.12

After sorting the ME values, 20% of the filter IDs that correspond to the highest ME are not approximated and are changed to full precision weights. The filter IDs that result in the

highest ME change with the input image and are predicted during run-time.

Another filter selection technique called static filter selection was explored. This technique identifies filter IDs that result in the highest ME averaged over all the images in the training dataset. 20% of the filter IDs with the highest ME are chosen and the weights associated with them are changed to full precision weights at design time.

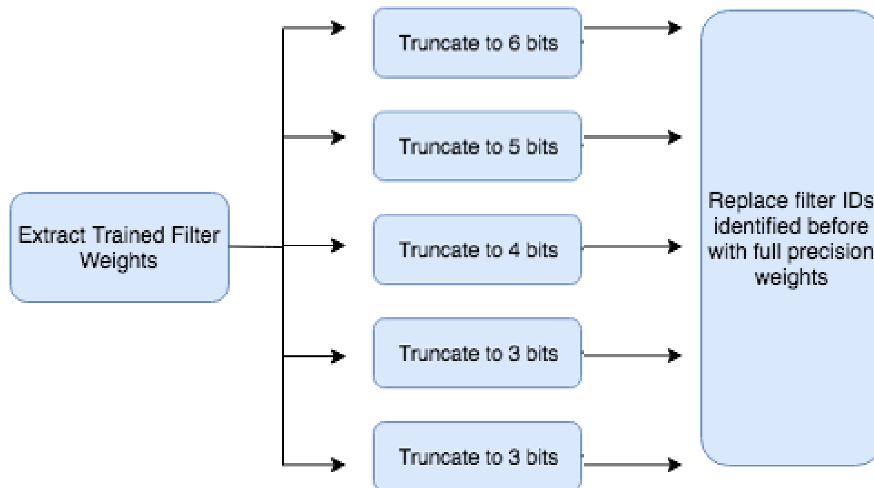

Figure 4.14: Pipeline for static filter selection

Chapter 5

Results

Experiments were carried out to find the inference accuracy of a 4 layered Convolutional Neural Network, LeNet and AlexNet on CIFAR-10 and MNIST datasets for static approximation, static filter selection and dynamic filter selection techniques.

For static approximation technique, all the layers of the deep neural network were replaced with the approximated filter at design time and the classification accuracy of the network was computed on the testing dataset of the MNIST and CIFAR-10 datasets.

For static filter selection technique, the average ME for each filter in a layer across the test dataset was computed and the 20% of the filter IDs that correspond to the highest ME were identified. Those filters were replaced with the original filters at design time.

For dynamic filter selection technique, the ME for each filter in a layer for the input image is computed at run-time and 20% of the filters contributing to the highest ME are replaced with the original filters.

5.1 MNIST dataset

5.1.1 Simple Convolutional Neural Network

Original classification accuracy of CNN on an MNIST dataset is 98.63%.

Approximation	Static Approx.	Static Selection	Dynamic Selection
Truncate to 6	98.26	98.30	98.62
Truncate to 5	97.60	97.91	98.60
Truncate to 4	87.00	88.02	89.12
Truncate to 3	17.4	19.82	29.8
Truncate to 2	9.74	14.1	17.4

Table 5.1: Classification Accuracy of a simple CNN on MNIST dataset

From the results shown in Table 5.1, truncating to 5 bits using dynamic filter selection yields higher accuracy as compared to truncating to 6 bits using static approximation. In some cases, aggressive approximation techniques using dynamic filter selection perform better than less aggressive techniques using static approximation.

5.1.2 LeNet

For LeNet, the original accuracy is 98.9%.

Approximation	Static Approx.	Static Selection	Dynamic Selection
Truncate to 6	98.8	98.9	98.9
Truncate to 5	98.7	98.8	98.9
Truncate to 4	98.4	98.6	98.8
Truncate to 3	90.5	94.1	95.6
Truncate to 2	9.9	10.7	16.4

Table 5.2: Classification Accuracy of LeNet on MNIST dataset

From the results shown in Table 5.2, truncating to 5 bits using dynamic filter selection has higher accuracy as compared to truncating to 6 bits using static approximation. Truncating to 4 bits using dynamic selection also has higher accuracy as compared to truncating to 5 bits using static approximation.

5.1.3 AlexNet

The original classification accuracy of AlexNet on MNIST dataset is 83.5%.

Approximation	Static Approx.	Static Selection	Dynamic Selection
Truncate to 6	83.2	83.5	83.5
Truncate to 5	82.6	83.0	83.1
Truncate to 4	77.5	78.5	79.2
Truncate to 3	72.1	74.5	75.0
Truncate to 2	27.9	30.4	34.9

Table 5.3: Classification Accuracy of AlexNet on MNIST dataset

5.2 CIFAR-10 Dataset

5.2.1 Simple Convolutional Neural Network

The original classification accuracy of a simple CNN with the CIFAR-10 dataset is 94.89%.

Approximation	Static Approx.	Static Selection	Dynamic Selection
Truncate to 6	93.9	94.6	94.6
Truncate to 5	92.8	93.0	93.2
Truncate to 4	82.7	83.6	85.3
Truncate to 3	65.5	69.8	73.2
Truncate to 2	6.8	11.2	15.1

Table 5.4: Classification Accuracy of a simple CNN on CIFAR-10 dataset

5.2.2 LeNet

The original classification accuracy of LeNet on a CIFAR-10 dataset is 86%.

Approximation	Static Approx.	Static Selection	Dynamic Selection
Truncate to 6	85.5	85.6	85.6
Truncate to 5	84.1	85.0	85.1
Truncate to 4	72.43	75.9	78.5
Truncate to 3	62.9	67.5	70.7
Truncate to 2	7.4	13.8	19.0

Table 5.5: Classification Accuracy of LeNet on CIFAR-10 dataset

5.2.3 AlexNet

The original classification accuracy of AlexNet on the CIFAR-10 dataset is 76.5%.

Approximation	Static Approx.	Static Selection	Dynamic Selection
Truncate to 6	76.3	76.4	76.4
Truncate to 5	74.4	75.7	75.7
Truncate to 4	66.7	71.6	72.8
Truncate to 3	65.8	66.2	66.4
Truncate to 2	26.5	28.0	29.6

Table 5.6: Classification Accuracy of AlexNet on CIFAR-10 dataset

Both dynamic and static filter selection techniques consistently outperform the static approximation technique. Dynamic and static filter selection techniques do not approximate 20% of the filters to less precision weights. However, dynamic selection technique still consistently outperforms the static selection technique. This is because the static technique chooses the filter IDs that, averaged across the whole test dataset, produce the highest ME. For a single input image, the filter IDs that result in the highest ME could be different. On the other hand, the dynamic filter selection technique chooses filter IDs yielding high ME on a per-input basis. Therefore, for each input image the filter IDs that result in the highest ME are changed to full precision weights.

To demonstrate low inference accuracy for static filter selection as compared to dynamic filter selection, we use LeNet with the CIFAR-10 dataset. The approximation technique used is truncating to 5 bits.

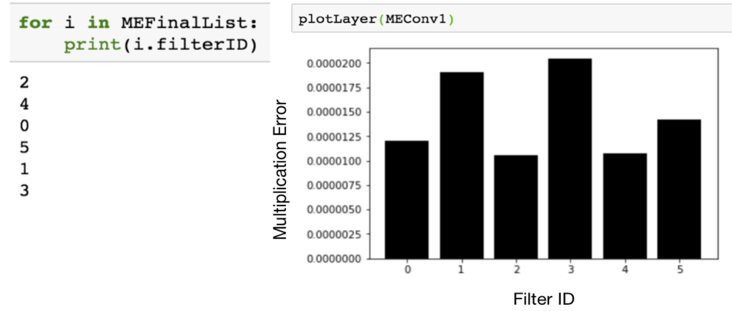

Figure 5.1: Design-time sorted ME order for Convolutional Layer 1

For the Convolutional Layer 1, the filter ID that corresponds to the highest ME averaged across all the images in the test dataset is 3. Therefore at design time, the filter weights corresponding to this filter are not approximated.
At run-time however for a randomly selected image from the test dataset, it was found that the filter ID that corresponds to the highest ME is 0.

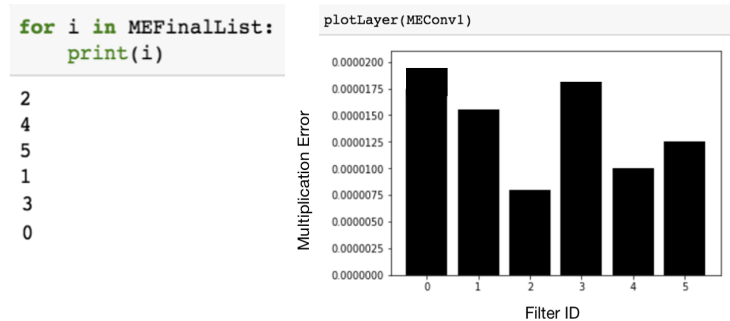

Figure 5.2: Run-time sorted ME order for Convolutional Layer 1

For Convolutional Layer 2, the filter IDs resulting in highest ME are 0,6 and 8. At design time, these filters are switched to the original filters.

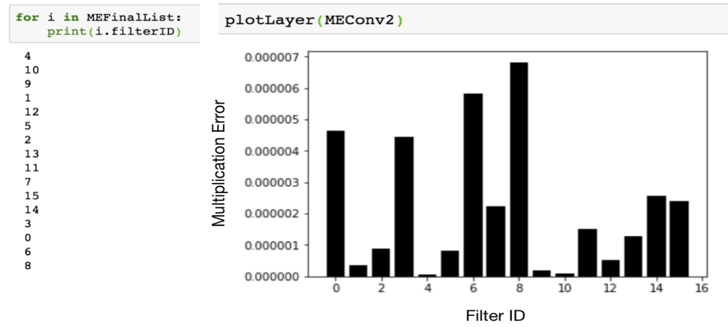

Figure 5.3: Design-time sorted ME order for Convolutional Layer 2

For a randomly selected image from the test dataset, the filter IDs corresponding to the highest ME are also 0,8 and 6.

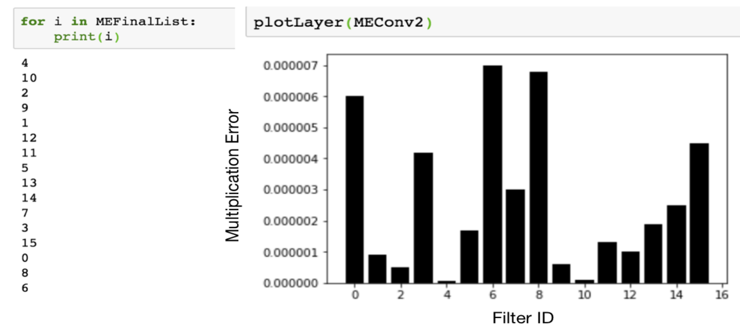

Figure 5.4: Run-time sorted ME order for Convolutional Layer 2

For Fully Connected Layer 3, the filter IDs contributing to the highest ME are 8, 0 and 7. At design time, these filters are switched to the original filters.

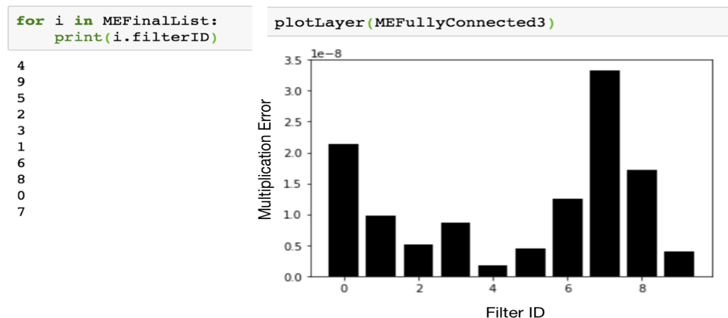

Figure 5.5: Design-time sorted ME order for Convolutional Layer 2

However, for a randomly selected image from the test dataset, the filter IDs resulting in highest ME are 0, 6 and 7.

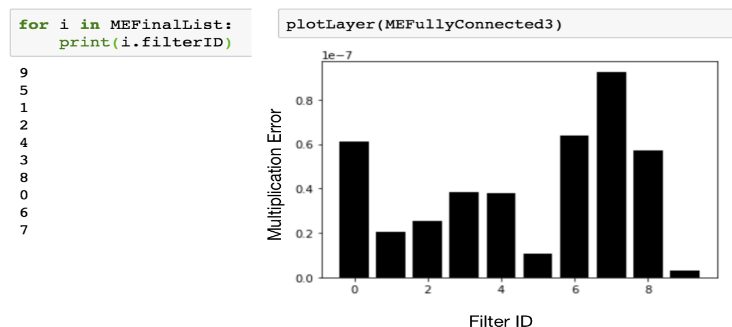

Figure 5.6: Design-time sorted ME order for Convolutional Layer 2

This shows that statically chosen filter IDs resulting in high ME may or may not be the same as the filter IDs that result in high ME for a single input image. On the other hand, dynamic filter selection technique chooses filter IDs that yield high ME for the input image at run-time. This results in a higher accuracy as compared to static filter selection.

Chapter 6

Conclusion

From the results presented in the previous section, the proposed filter selection and approximation technique yields higher inference accuracy as compared to its static counterparts. The increase in accuracy using the dynamic technique is more evident in aggressive approximation techniques like truncating to 2 and 3 bits.

The proposed dynamic filter and selection technique approximates filters on a per-input basis and during run-time chooses which filters should retain full precision weights. As a result, 80% of the filters in a layer are approximated to less precision weights and 20% of the filters retain original 32 bit floating point weights.

Despite switching 20% of filter weights with full precision weights, aggressive truncating of bits does not yield high inference accuracy. Since the network has been trained with 32 bit floats, the inference accuracy incurs a severe drop with most of the weights being represented as 0 upon truncating to 2 and 3 bits. In order to resolve this, the network could be trained with low precision to begin with. For instance, if a network has been trained with 8 bit weights, truncating to 2 and 3 bits results in a small drop of accuracy [16]. The proposed dynamic filter selection and approximation technique with low precision trained weights would yield high inference accuracy for aggressive bit truncating.

AI applications are slowly moving towards faster response time and bigger workloads. Moving from high precision network parameters to low precision parameters opens the door to significant performance gains [16]. Methods like network weight quantization, training net-

works like lower precision and binarized neural networks are garnering great interest because of being able to preserve inference accuracy while reaping great performance benefits out of using less number of bits.

Bibliography

[1] Bengio, Y., Courville, A. and Vincent, P. (2013). Representation Learning: A Review and New Perspectives. IEEE Transactions on Pattern Analysis and Machine Intelligence, 35(8), pp.1798-1828.

[2] Vanhoucke, Vincent, Andrew Senior, and Mark Z. Mao. (2011)."Improving the speed of neural networks on CPUs."

[3] Towards Data Science. (2019). CIFAR-10 Image Classification in TensorFlow. [online] Available at: https://towardsdatascience.com/cifar-10-image-classification-in-tensorflow-5b501f7dc77c.

[4] En.wikipedia.org. (2019). Feature Scaling. [online] Available at : https://en.wikipedia.org/wiki/Feature_scaling

[5] Medium. (2019). Learning Less to Learn BetterDropout in (Deep) Machine learning. [online] Available at: https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5.

[6] Brownlee, J. (2019). A Gentle Introduction to the Rectified Linear Activation Function for Deep Learning Neural Networks. [online] Machine Learning Mastery. Available at: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/.

[7] Towards Data Science. (2019). What are Hyperparameters ? and How to tune the Hyperparameters in a Deep Neural Network?. [online] Avail-

able at: https://towardsdatascience.com/what-are-hyperparameters-and-how-to-tune-the-hyperparameters-in-a-deep-neural-network-d0604917584a.

[8] Chen, ZhiQiang, Chuan Li, and Ren-Vinicio Sanchez. (2016). "Gearbox fault identification and classification with convolutional neural networks." Shock and Vibration 2015.

[9] Nayak, S. (2019). Understanding AlexNet — Learn OpenCV. [online] Learnopencv.com. Available at: https://www.learnopencv.com/understanding-alexnet/.

[10] He, Kaiming, et al. (2016). "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition..

[11] Jupyter.org. (2019). Project Jupyter. [online] Available at: https://jupyter.org/.

[12] Skymind. (2019). A Beginner's Guide to Neural Networks and Deep Learning. [online] Available at: https://skymind.ai/wiki/neural-network.

[13] Mathworks.com. (2019). What Is Deep Learning? — How It Works, Techniques Applications. [online] Available at: https://www.mathworks.com/discovery/deep-learning.html

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

[15] Colangelo, Philip, et al.(2018). "Exploration of Low Numeric Precision Deep Learning Inference Using Intel FPGAs." arXiv preprint arXiv:1806.11547

[16] Wang, N., Choi, J., Brand, D., Chen, C. Y., Gopalakrishnan, K. (2018). Training deep neural networks with 8-bit floating point numbers. In Advances in neural information processing systems (pp. 7675-7684).

[17] Venkataramani, S., Chakradhar, S. T., Roy, K., Raghunathan, A. (2015). Approximate computing and the quest for computing efficiency. In Proceedings of the 52nd Annual Design Automation Conference (p. 120). ACM.

[18] Sze, V., Chen, Y. H., Yang, T. J., Emer, J. S. (2017). Efficient processing of deep neural networks: A tutorial and survey. Proceedings of the IEEE, 105(12), 2295-2329.

[19] Zhang, Q., Wang, T., Tian, Y., Yuan, F., Xu, Q. (2015). ApproxANN: An approximate computing framework for artificial neural network. In Proceedings of the 2015 Design, Automation Test in Europe Conference Exhibition (pp. 701-706). EDA Consortium.

[20] Intel AI. (2019). Scalable Methods for 8-bit Training of Neural Networks - Intel AI. [online] Available at: https://www.intel.ai/scalable-methods-for-8-bit-training-of-neural-networks-blog/gs.cmztur.

[21] Yann.lecun.com. (2019). [online] Available at: http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf [Accessed 21 May 2019].