Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

Effective Index-Mapping of Quantized Values for Low-Precision Neural Networks on Power-Efficient Embedded Devices

Ian Hunter
10333515

September 3, 2020

A Masters Thesis submitted in partial fulfilment

of the requirements for the degree of
M.Sc (Computer Science)

Declaration

Abstract

Neural networks are sets of algorithms that together can approximate general functions. To approximate a function, the network must first be "trained" by a framework that can give informed feedback to reinforce correct predictions.

As these function approximations can be trained ahead of time, neural networks are often used for work that will have previously unseen inputs — such as those seen in the field of Computer Vision.

The Intel® Movidius™Myriad™ VPU is an embedded processor that is integrated into many hand-held and battery powered devices. In Intel® Movidius™'s latest processor, a hardware component was included to accelerate neural network software.

In this thesis, we explore a particular feature of this hardware component — An index-mapping of the neural network's intermediary and trained values.

We propose several new approaches to configuring this component and how they could be used to improve classification rates for very low precision networks. Of particular note, is the LeNet network where our 4 bit results match those of a 32 bit equivalent. However, we find that our proposed algorithms are suitable for different scenarios and would be best used as a suite.

Finally, we demonstrate the performance of the VPU using the hardware component to achieve 4 times lower data transfer sizes and consequentially, 4x faster processing of a layer.

Acknowledgements

I would like to extend my thanks and appreciation to my supervisor, David Gregg and his research students for selflessly assisting in my research problem and helping refine my thesis subject.

Thanks must also be given to my managers, past and present at Intel® Movidius™, who without their approval, I would not have received funding for my studies, nor time off to work upon it.

Furthermore, I would like to acknowledge the patience and encouragement from my family and friends, even when my research has caused me to pass up many of their social events and visits.

I would like to dedicate this thesis to the many weekends I have left behind, the mental duress from two years of combined work and study, and the many letters of resignation that lie unsent in my email drafts.

To offset some of the environmental impact of my study, a donation has been given to plant ten native trees in Moy Hill Farm, Co. Clare.

Contents

List of Figures

List of Tables

Listings

1 Introduction

1.1 Research Context

In recent times there has been a significant rise in the demand for small embedded consumer devices. Similarly, many industries are now solving technical problems with neural networks rather than traditional processes. As a result, there is much research interest in combining these two areas — to deploy neural networks on embedded devices.

Both trends have been enabled by advances in microprocessor capabilities. It is now possible to use a personal computer to develop a neural network of significant size, where previously only supercomputers could attempt such a feat.

Mobile phones in particular have driven technological advances for low power embedded systems. Current products can contain multi-core processors and benefit from the same observed doubling of microchip transistors that other computers gain from every year (i.e. Moore's Law)(Schaller, 1997).

On initial inspection however, it may seem that the combination of neural networks and embedded devices is a contradictory goal. Embedded devices are generally low-power, low-throughput and memory constrained, whilst the computation of neural networks requires large amounts of processing power and frequent data movement.

Facilitating these contrasting properties has been the subject of much previous research. This thesis is specifically interested in the reduction of numeric storage size, and the consequent reduced circuitry. It builds upon the hardware optimizations implemented by Intel® Movidius™, particularly their application of higher precision values to a lower-precision index map.

1.2 Thesis Overview

This thesis explores several methods to choose index-mapping values for such a hardware using novel approaches that consider the domain of neural networks, rather than a naïve generic configuration.

The algorithm proposals are assessed based on how their approximation of the original data affects the classification rates of a given neural network.

Despite the particular investigation into the Intel® Movidius™Myriad™ VPU, the ideas presented in this thesis are transferable to similar hardware models should they exist.

This thesis shows that naïve algorithms are sub-optimal and concludes that there are several possible algorithms to improve a given network, but does not establish a single solution for all cases. The continued pursuit of such a solution is proposed to future researchers along with several other proposals gathered during the collation of the research.

1.3 Overview of Document Layout

Chapter 1 (This chapter) presents a brief overview of thesis's problem, its importance in a wider context and summarizes the thesis's findings.

Chapter 2 will give the reader an introduction to the established concepts and notable literature that the thesis is founded on. It begins with a brief overview of the operation of neural networks, followed by select mentions of previous research and concludes with a high-level description of the Intel® Movidius™Myriad™ VPU's hardware accelerator.

With the background of the thesis's research now established, **Chapter 3** will discuss the initial investigation into the research problem. Firstly the problem is verified valid and its impact measured. This is followed by several proposal changes to the index-mapping process that may alleviate the issue. Finally, we discuss some technical decisions that will inform how a system to further explore the research problem is to be implemented.

Chapter 4 is the core of the thesis and describes the final technical design and the workings of the investigative development done throughout the project. This chapter is structured in three sections; An initial statement of what the technical project should aim to achieve, the details of how such an implementation is architected and finally, any failings or limitations of this taken approach.

Chapter 5 will collate the information gathered through the research in Chapter 4 and attempt to make sense of its results. The data gathered is statistically analysed, the technical implementation and its impact on the project critiqued, and finally, there is an overall evaluation of the thesis' findings and approach.

Concluding the project is **Chapter 6**, in which the analysis results from the previous chapter are examined in a wider context. We present several concepts for future research in this area, a summary of the work presented in the thesis as well as some closing thoughts on the thesis and the wider research domain.

At the end of the thesis, some supplementary material is provided to aid readers in a series of **Appendices** including a **Glossary of Terms**.

2 Background & Related Work

2.1 Foundations of Neural Network Design

Neural networks, more precisely known as “artificial neural networks” or ANNs are not a recent concept. The first paper on the subject was published in 1944 in the Bulletin of Mathematical Biology (McCulloch, 1944).

An ANN is a system of connected nodes called “neurons” which transfer signals to each other. When given a labelled set of information, the ANN can tune internal variables or “parameters” to estimate future unlabelled information. In the below neuron diagram — Figure 2.1 — these parameters are shown as “Hidden Nodes”.

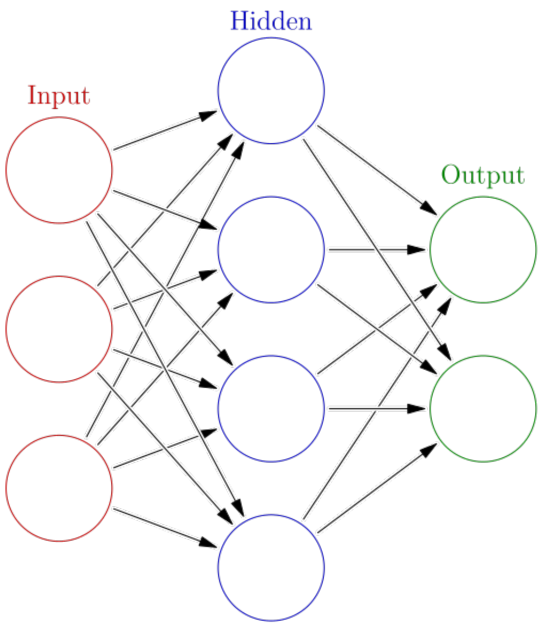


Figure 2.1: Neurons in an ANN operation[1]

There is a subset of ANNs that this thesis is primarily focussed upon known as convolutional neural networks (CNNs) and as such, mentions of neural networks in the thesis from this point onward will be referring to CNNs, unless explicitly mentioned otherwise.

The architecture of a CNN lends itself well to computation; CNNs have a reduced amount of connections needed to be processed compared to full ANNs and are built from a standardized series of operations.

When input data is passed into the network, the first operation of the network is executed, beginning a process that will refine the data towards a classification. The operation's output is then processed by the subsequent operation, repeating until each operation has processed over the data. At the end of the network, a series of classification rates are produced.

The first technical implementation of a convolutional neural network was introduced by Yann LeCun in 1998, using a simple (by current standards) series of

[1]Source: Creative Commons Attribution-Share Alike 3.0 Unported - `https://en.wikipedia.org/wiki/File:Colored_neural_network.svg`

grid-based computation operations to recognize hand written digits (LeCun et al., 1998). The core functionality of convolutional neural networks has not changed much from LeCun's original design. This network, named "LeNet", after the author, is shown in Figure 2.2 with visualizations of the output of each NN operation as the image is processed through the network.

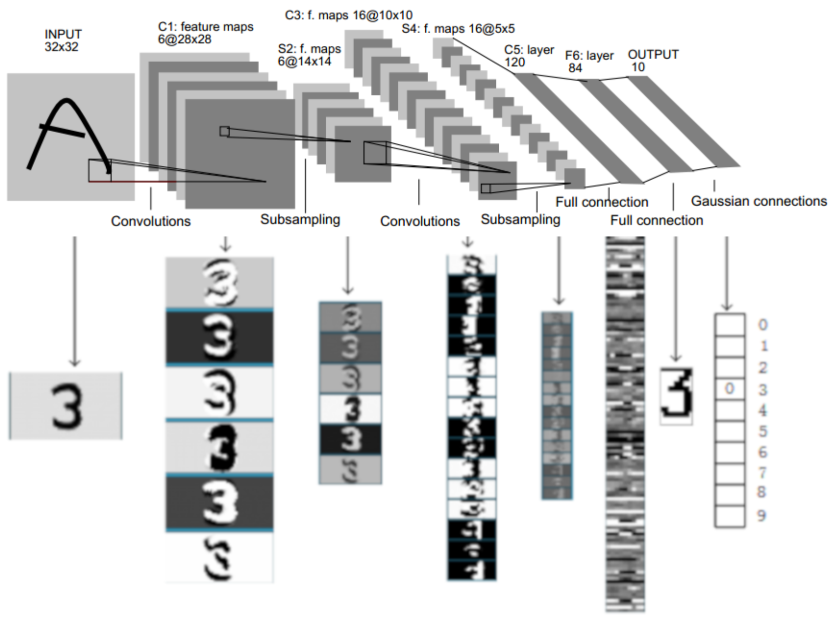


Figure 2.2: This visualization of the LeNet Architecture shows the data as it is processed through the network. The image of a '3' is classified correctly at the conclusion of execution.[2]

However, it was not until the annual ImageNet Large Scale Visual Recognition Competition (ILSVRC) in 2012 that CNNs began to proliferate into technical research. Before this year's event, the winners of ILSVRC and other image competitions primarily used other classical computer vision techniques.

Alex Krizhevsky and fellow researchers of University of Toronto won the contest with their neural network "AlexNet" (Shown in Figure 2.3) by a significant margin. The error rate they achieved was 15.3%, the second-placing team only achieving

[2]Source: (LeCun et al., 1998)

26.2% (Krizhevsky, 2017).

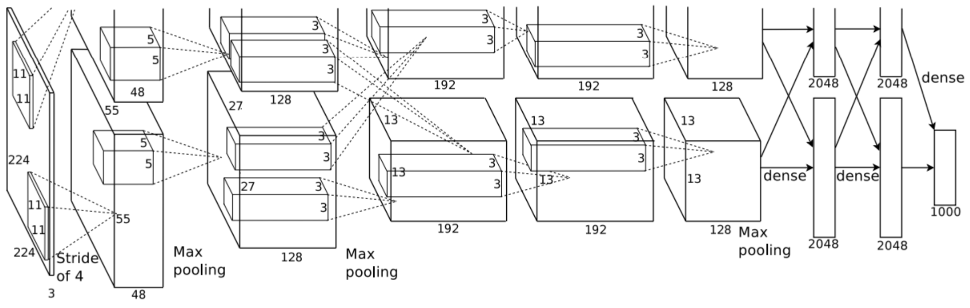


Figure 2.3: The AlexNet Architecture showing the network processing across 2 GPUS[3]

The interest created from this competition win supported a large surge in image classification research and by 2017, most teams entering the contest had less than a 5% error rate (As can be seen in Figure 2.4). After the 2017 contest, the hosts announced that a new, more difficult, contest would take its place.[4]

[3]Source: (Krizhevsky, 2017). The top of this image is also cropped in the original paper.
[4]`https://www.newscientist.com/article/2127131-new-computer-vision-challenge-wants-to-teach-robots-to-see-in-3d/`

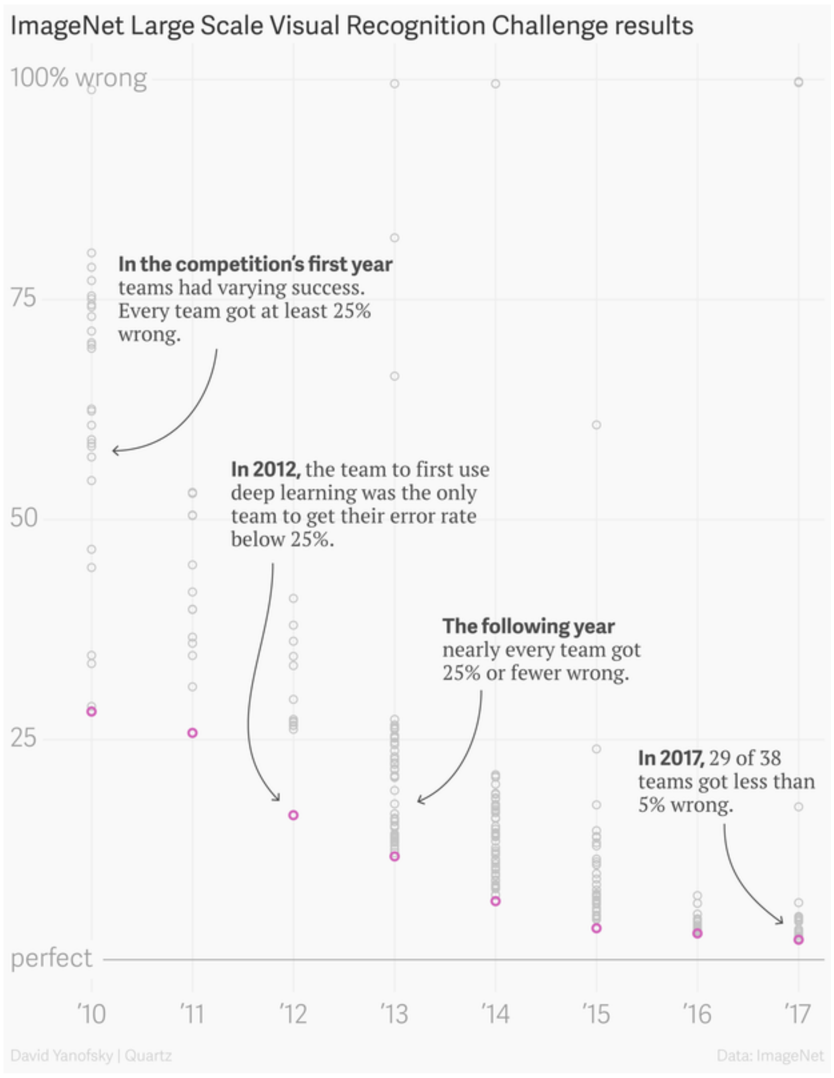


Figure 2.4: Advances in image classification as shown by the improving standard of submissions to ILSVRC[5]

Neural networks are now used in many products in industry; What was once an expensive and limited technique has become an accessible solution for many domains.

In particular, many industries previously relying on traditional Computer Vision (CV) are replacing their systems with neural networks (Behringer, 2019). The

[5]Source: `https://qz.com/1046350/the-quartz-guide-to-artificial-intelligence-what-is-it-why-is-it-important-and-should-we-be-afraid/`

"Universal Approximation Theorem" proposes that neural networks can approximate to a large quantity of continuous functions (Hornik, 1991)[6] and thus the applicable domains for NNs in industry is wide-reaching.

2.1.1 Convolutions

Convolutions are the main operation of a CNN, and from where the network architecture gets its namesake.

Much like other CV operations, a convolution is a parameter-based grid operation. These are well suited to processing on GPUs (Fung, 2019).

Convolutions in a CNN are generally 2-dimensional functions that operate on a 4-dimensional tensor. The core computation convolves a set of parameters around a matrix (This input data is often named *activation data* or *activations*).

Figure 2.5 shows a convolution producing a single output value. The input values for the computation, shown as a 3×3 grid on a larger matrix, is known as the *receptive field*. Each element of this receptive field is multiplied with a corresponding parameter (sometimes called a "weight"). These products are then summed to produce the output value, and the operation will move on to calculate the next item.

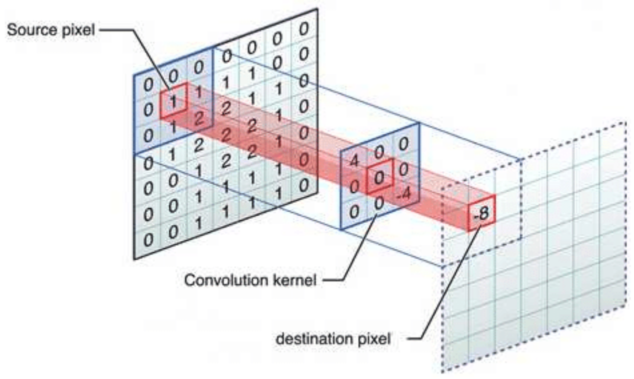


Figure 2.5: A 3×3 Convolution operation[7]

[6]The Universal Approximation Theorem states that a neural network with 1 hidden layer can approximate any continuous function for inputs within a specific range. (Csáji, 2019)

Those readers familiar with computer vision will likely recognise this pattern of operation as the same technique is used for more traditional tasks such as blurring, sharpening and edge detection.[8]

Neural network convolutions often work off 3-dimensional inputs (e.g. colour images have a width, a height and a number of colour channels). In this case, each additional channel of data requires a corresponding set of parameters. After each channel is individually calculated, these are summed with the other channels to produce one pixel.

Thus far, the convolution has produced a two dimensional output from the original three dimensions of the input. The fourth and final dimension commonly needed produces multiple output channels for an output value, preserving the dimensionality from input to output (i.e. the input is 3D and the output is also 3D).

Listing 2.1: Code for 2D convolutions

```
# NumPy is a common python library for numerical computation
# and is used throughout this thesis's code samples
# Computing in Science & Engineering 13, 22 (2011);
# https://doi.org/10.1109/MCSE.2011.37
import numpy as np

# Example Convolution:
# Input Size: 224, 224, 3
# Weight Size: 1, 1, 3, 64
# Output Size: 224, 224, 64

def convolve(input, weights):
  # Sum of Products
  # Input: 2D subsection of input
  # Weights: a 2D set of weight parameters
  # Output: a 1x1 Number
```

[7]Source: (Agapitos et al., 2015)

[8]http://ai.stanford.edu/~syyeung/cvweb/tutorial1.html

```
  return np.sum(np.multiply(input, weights))

# The operation shown in Figure 2.5
def convolution_2D(input, weights):
  # Input: A 2-dimensional plane of data
  # Weights: A set of kernels for each convolution
  # Output: A 2-dimension plane of data

  # It's possible to have asymmetric kernels,
  # but let's keep it simple!
  kernel_size = weights.shape[0] # 1
  input_height = input.shape[0] # 224
  input_width = input.shape[1] # 224

  for ih in input_height:
    for iw in input_width:
      # In the same vein, this simple example
      # does not account for stride, nor padding
      output[ih][ih] = \
        convolve(input[ih:ih+kernel_size][iw:iw+kernel_size],
                 weights)

  return output
```

Listing 2.2: 3D and 4D Convolutions

```
def convolution_3D(input, weights):
  # Input: A 3-dimensional plane of data
  # Weights: A set of kernels for each 2d plane of input
  # Output: A 3-dimension plane of data,
  #   an accumulation of the 2d results

  input_channels = input.shape[2]  # 3
  for ic in input_channels:
    output += convolution_2D(input[ic], weights[ic])

  return output

def convolution_4D(input, weights):
  # Input: A 4-dimensional plane of data
  # Weights: A set of kernels for each 3d plane of input
  # Output: A 4-dimension plane of data
  #   each entry a 3d result

  output_channels = weights.shape[4] # 64

  for oc in output_channels(input, weights):
    output[oc] = convolution_3D(input[oc], weights[oc])

  return output
```

By having multiple sets of parameters, the convolution can capture several representations of the information contained in each receptive field. An example of this can be seen in the visualization of the LeNet network in the previous Figure 2.2 when the channels between operations increase.

Because the input to the convolution and the operation's result are both 3-dimensional matrices, multiple convolutions can be linked together without data reorganization.

Optimizations

There have been several research efforts to improve upon the performance of convolutions, whether through direct optimization like Winograd convolution (Lavin, 2016), or substitution of a similar, less complex, operation such as group-based convolution (Krizhevsky, 2017). However, the scope of this thesis limits itself to traditionally computed convolutions due to both time constraints and the applicability to the Intel® Movidius™Myriad™ VPU platform that will be discussed in Section 2.4.

2.1.2 Pooling

Network designers use pooling operations to reduce the spatial size of a network and to reduce both the number of computations and trained parameters as it approaches a final classification.

Pooling operations filter the least pertinent information away and preserve the most prominent information for future operations.

The most popular pooling algorithm, "Maximum Pooling" identifies the most 'important' information by selecting the largest values in tiled local areas (Yu et al., 2014). An example of the operation can be seen in Figure 2.6.

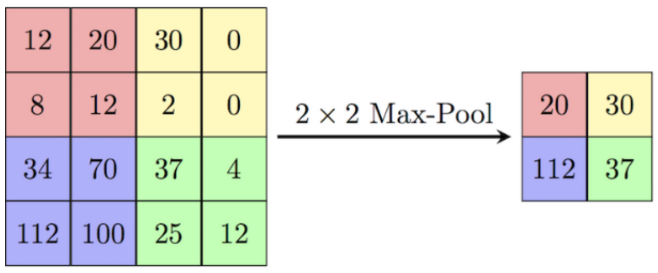


Figure 2.6: A 2×2 pooling operation[9]

Most other pooling operations operate in the same manner, but may choose to compute the output value differently (e.g. 'Average Pooling' will output the average of each shaded section.

Listing 2.3: Code for poolings

```
# Example Maximum Pooling:
# Input Size: 224, 224, 3
# Operation Size: 2
# Output Size: 112, 112, 3

def pool(input, pool_type):
```

[9]Source: https://computersciencewiki.org/index.php/File:MaxpoolSample2.png

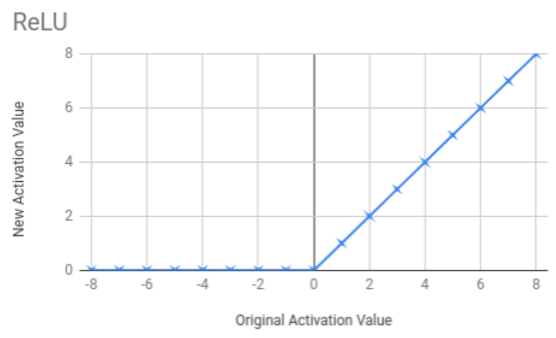
```
    if pool_type == "MAX":
      return np.max(input)
    elif pool_type == "AVG":
      return np.mean(input)
    else:
      raise Exception

def pooling(input, pool_type, kernel_size):
  # Input: A 3-dimensional plane of data
  # pool_type: Whether maximum or average pooling
  # Kernel size: dimension length of individual pooling

  # Again, assuming symmetric for simplicity
  input_side = input.shape[0] # 224

  for ih in input_side:
    for iw in input_side:
      out_idx = input_side*(1/kernel_size)
      output[out_idx][out_idx] = \
        pool(input[ih:ih+kernel_size][iw:iw+kernel_size],
              pool_type)

  return output
```

2.1.3 Activation Functions

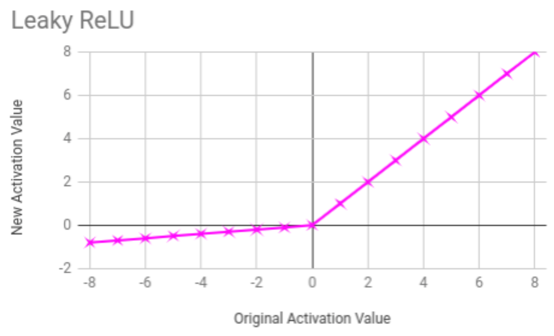
Activation functions are (typically) small functions that are applied to a more compute-intensive operation's results (such as convolution), modifying each activation's impact on future operations. They generally operate on a single activation at a time.

There are many types of activation functions available to the network designer. The most common of these is the Recification Linear Unit (ReLU), introduced in

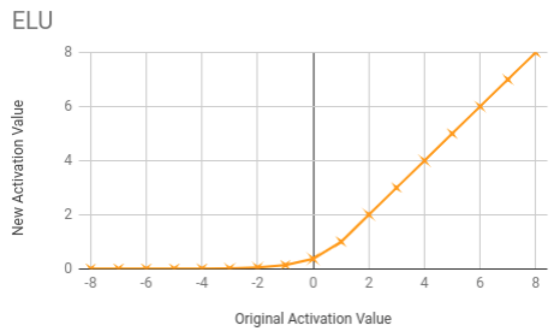
Glorot et al. (2011). The functionality of this computation is quite simple — any values that are negative are thresholded to zero. The ReLU function and some other activation functions can be seen computed on a range of values in Figure 2.7.



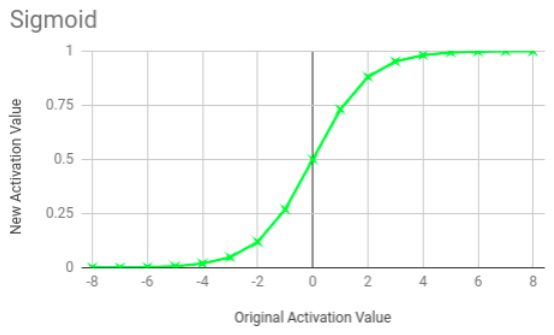
(a) ReLU



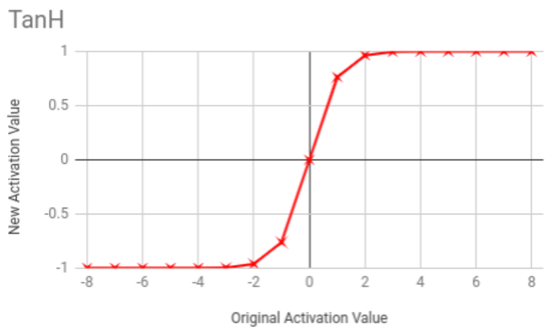
(b) Leaky ReLU



(c) ELU



(d) Sigmoid



(e) TanH

Figure 2.7: Different Types of Activation Functions.

As the magnitude of neuron values generally represent correlation with predetermined classes, negative values indicate the degree of which the neurons do *not* match the classes. As the degree of correlation is usually the only aspect we are interested in, the results do not suffer significantly when negative values are removed.

This has two benefits; The volume of data needing preservation throughout the network has been reduced and can be removed later in the network (in a future pooling layer for example).

Secondly, the ReLU creates many zero values in the output matrix. This creates a lot of potential for mathematical optimizations and facility for skipped computation. Taking advantage of the presence of zero values is discussed in more detail in Section 2.3.

While the Intel® Movidius™Myriad™ VPU has hardware support for acceleration of activation functions, this thesis will not investigate much in this area, due to the low impact these operations usually have on the computation performance of a network.

Listing 2.4: Code for Rectification Units

```
def ReLU(value):
  return value if value > 0 else 0

def LeakyReLU(value, negative_slope):
  return value if value > 0 else value * negative_slope

def ELU(value, alpha):
  return value if value >= 0 else alpha * (np.exp(value)) - 1

def Sigmoid(value):
  return 1 / (1 + np.exp(-value))

def TanH(value):
  return tanh(value)
```

2.1.4 Fully Connected Layers and Classifiers

As computation proceeds through the architecture of a network, the height and width of the input gradually decreases (mostly through the effect of pooling as described in Section 2.1.2 and edge compensation[10]) while the number of channels tend to increase. Eventually, the network designer will want to convert their matrices of data into classifications and their data dimensionality will reduce to a single dimension.

The fully connected layer (FCL) cross-correlates all data points with every set of class weightings (thus being "fully connected"). This results in a matrix where each final classification point has been impacted by every neuron in the network, allowing for a final assessment of class significance relative to every other possible class. The operation can be seen in (a) of Figure 2.8, (b) and (c) sub-figures show the full connectivity of each data point.

[10]Edge compensation refers to the effect of performing a convolution with a kernel size of greater than 1x1 without adding padding to the edges of the input to offset the indented kernel computation

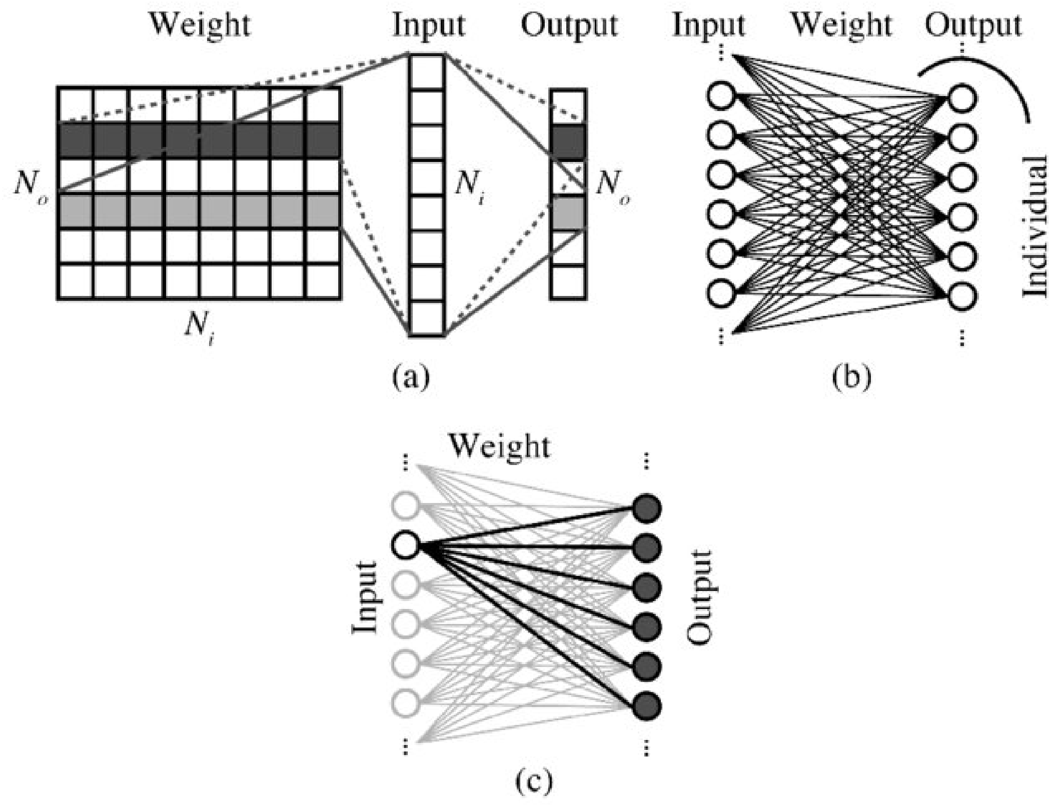


Figure 2.8: The fully connected layer operation and each input node's full connection to each output node[11]

Listing 2.5: Code for Classifiers

```
def FCL(input, weights):
  return np.multiply(input, weights)

def SoftMax(input):
  return np.exp(x) / np.sum(np.exp(x))
```

This final classification figure is generally done by a classification operation such as softmax. Classification operations normalize the resultant data to produce a more human-readable representation, such as a 0 to 100 classification percentage.

One notable defect of the design of CNNs and the FCL is the lack of detecting

[11]Source: (Ando et al., 2017)

when *nothing* is present. If an uncategorizable input is passed through the network (e.g. an image of a boat in a “Dog or Cat” recognition network), there will be some correlations with unrelated classes (that boat may register as 45% Dog, 55% Cat).

It is noteworthy that convolutions and fully connected layers can be interchanged through some minor linear algebra.[12] Any research findings of convolutions found in this thesis can be applied to fully connected as a result. There is a worked example of this in Appendix A1.1

2.2 Training

Neural networks, like other supervised learning algorithms, require labelled datasets to learn a particular task. As the training process computes, it tunes the values of the weights, biases and other internal parameters. Most popular and successful networks are founded upon very large datasets. For example, the latest version of the ImageNet dataset (at time of writing) has 14,197,122 labelled images.

Before seeing any dataset items, the training system generates a full parameter set for a network architecture using one of a selection of initialization methods (e.g. Xavier's initialization method (Glorot and Bengio, 2010)).

These parameters are updated as the training system proceeds to process through each input in the dataset. The classification results of these inputs are compared with the corresponding reference labels. Successful classifications positively re-enforce the parameters that produced it, while misclassifications introduce more variation to the parameters to allow the system to attempt alternate configurations. Figure 2.9 shows a generic example of some data-points being grouped into three distinct classes. The dotted lines indicate estimated separations. It can be seen there are some values incorrectly grouped; As the system adapts to training feedback, it will adjust these classification boundaries accordingly to find better estimations.

[12] `http://cs231n.github.io/convolutional-networks/`

Each complete classification pass over an entire labelled dataset is called an 'epoch'. The training process usually runs for several epochs, concluding either when a set accuracy target is met, "overfitting" (described below in Section 2.2) is detected, or a processing time limit is reached.

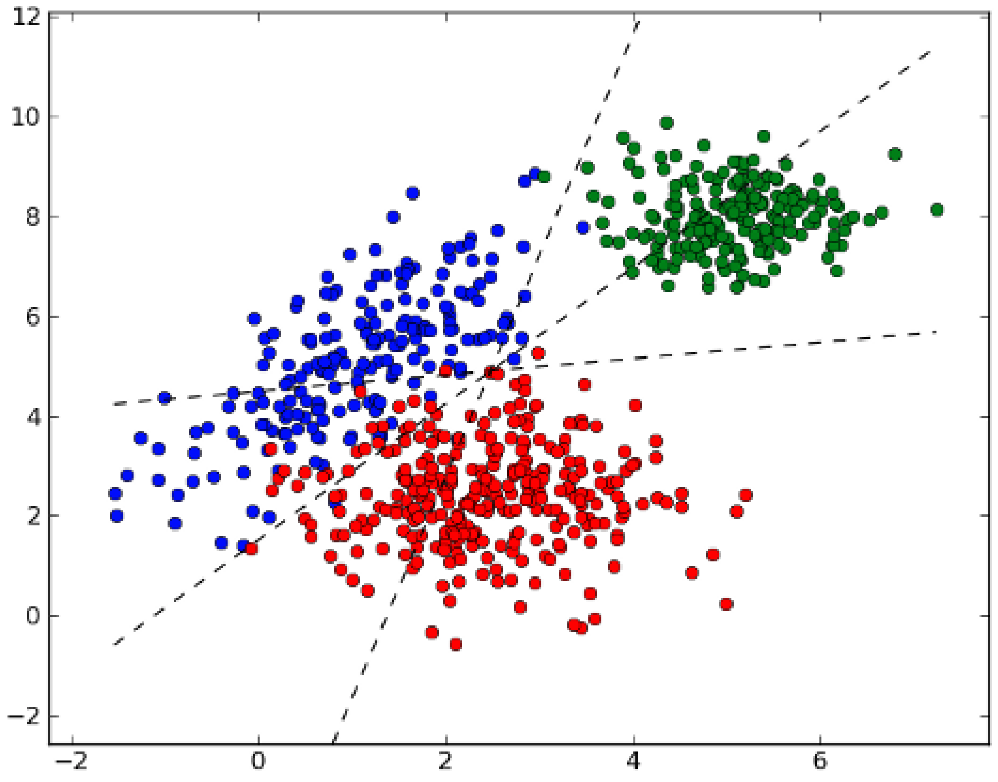


Figure 2.9: A visualization of data being roughly grouped into 3 Classes.[13]

Knowing when to stop

In principle, the training process can continue forever, but generally, there comes a point where classification rate increases start to take significantly more time to achieve. It is not wise to train for as long as possible as the network can "over-fit" the data, correlating features of the dataset that are not relevant to the actual classification. (e.g. All the pictures of dogs in a dataset are photographs taken

[13]Source: https://sourceforge.net/projects/mlpy/?source=directory

outdoors, if the network starts to over-fit; it may stop recognising dogs that are indoors) (Srivastava et al., 2014). Figure 2.10 shows this over-fitting effect on a simple two-class dataset.

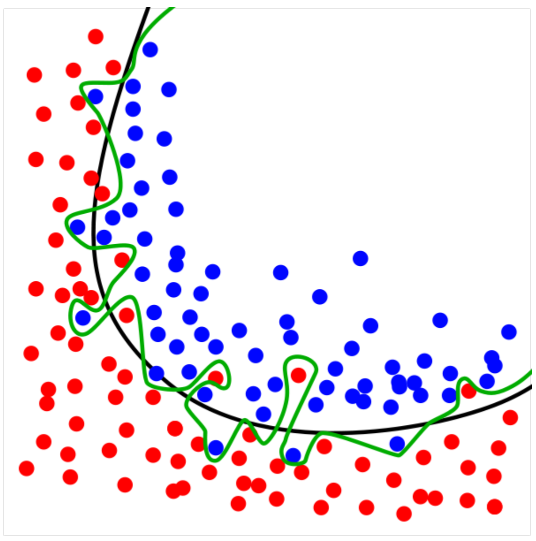


Figure 2.10: An overfit classification (green) versus a more appropriate general estimation (black)[14]

To determine when a training system should stop, a developer can split their dataset into 3 subsets — Train, Validation and Test. The training system will process on the Train subset and after each epoch, the Validation set is tested as well. As the Validation set is not involved in the training, the researcher can ensure some generality for unseen data. At the end of the process, the test set can be used to provide a fully independent accuracy figure.

[14]Source: CC BY-SA 3.0 (Chabacano, Feb 2008)

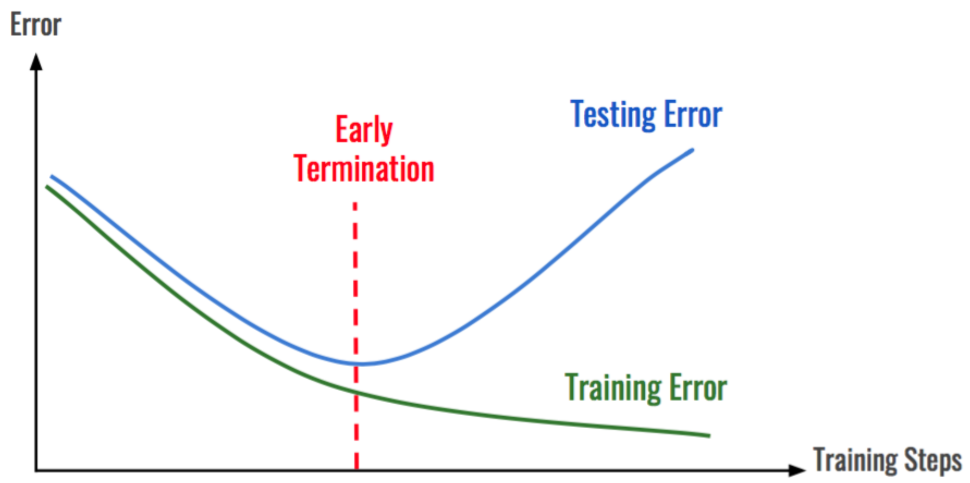


Figure 2.11: The point of classification rate divergence between seen and unseen data can signal overfitting and is a good time to terminate the training process[15]

Typically, classification improvements grow in sizeable increments in early epochs, but the improvements grow smaller as the network architecture approaches its classification limits for the dataset. In these later epochs, it can be difficult to tell if the classification rate has reached a final state, or if there is still more to be gained. One method (as shown in Figure 2.11) to determine an end to training is to monitor the Validation subset's error rate. When the classification rates of the Train subset continue to improve, but the Validation subset starts to deteriorate, over-fitting may be occurring.

Figure 2.12 below shows an example of such training sequence with real data. A perceived gain in accuracy up to epoch 30 can be seen in the training data, but in the validation data, there is not much gain past epoch 4. Additionally, the stochastic nature of the training process causes some volatility to the classification rates clouding possible trends.

[15]Source: https://hackernoon.com/memorizing-is-not-learning-6-tricks-to-prevent-overfitting-in-machine-learning-820b091dc42

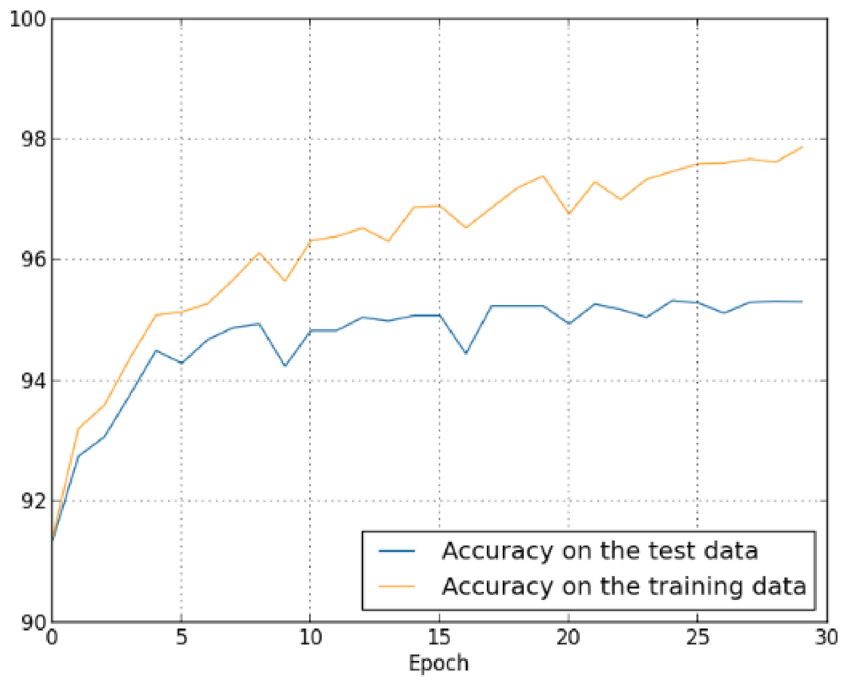


Figure 2.12: Example of classification rates over time using real data[16]

2.3 Reducing Computation Overhead

Neural networks contain a lot of data for processing. For a particular task, the required capacity (a measure of how complex a function it can model) of the network may change. For example, the comparatively small LeNet architecture works well on very small, preprocessed, greyscale handwritten numeric digits from 0 to 9, but will perform badly with more complex datasets. GoogLeNet, originally trained on larger photographic images of 1000 different ImageNet classes, is quite adaptable to similarly complex datasets, such as classifying species of flowers. (Gurnani and Mavani, 2017)

There is no widely accepted formal calculation for the capacity of a network, but several approximations have been proposed — most prominently, the

[16]Source: http://neuralnetworksanddeeplearning.com/chap3.html

Vapnik-Chervonenkis (VC) dimensions of a network. A simpler approximation would be one that scales linearly with the number of neurons in the network [17].

When designing a network architecture, or porting an existing one to your problem domain, it is generally desirable to have an architecture that is only as complex as is needed. An efficiently designed architecture will run with minimal overhead, thereby increasing the processing rate of classifications.

In the case of embedded systems (explained in more detail in Section 2.4), reducing the complexity of a network is important to enable efficient processing due to their constrained resources. The problem of resources is universal — even CPU and GPU devices benefit from reduced complexity, it can reduce training times and mitigate performance impact of slower operations and prototype research, which can be crucial for both research and cutting-edge products.

The first place a network designer should look to optimize a network is in its design. Cutting out operations in the network and retraining may preserve the network's classification rates if the culled operations were superfluous.

This technique works best when a network is far more complex than necessary. A network designer who has paid good attention to the network's construction may find less flexibility to remove operations.

Aside from general network design, there are several types of optimization for a developer to explore. The two most relevant to our research are **Sparsity** and **Low-Precision Networks**.

There are many other possible optimizations such as those called out in Section 2.1.1, but they will not be discussed further in this thesis.

2.3.1 Sparsity

There are several forms of 'sparsity' that can be categorized into "Coarse sparsity" and "Fine sparsity". Coarse sparsity removes channels of information from an

[17]`https://r2rt.com/preliminary-note-on-the-complexity-of-a-neural-network.html` points out several papers that use this as a metric of capacity

operation, while fine sparsity removes kernel elements. These removals may degrade classifications when applied immediately, but when involved in the training process this effect can be reduced. A visualization of these forms can be seen in further detail in Figure 2.13. In this figure, fine sparsity is broken further into 0, 1 and 2 dimensional sparsity.

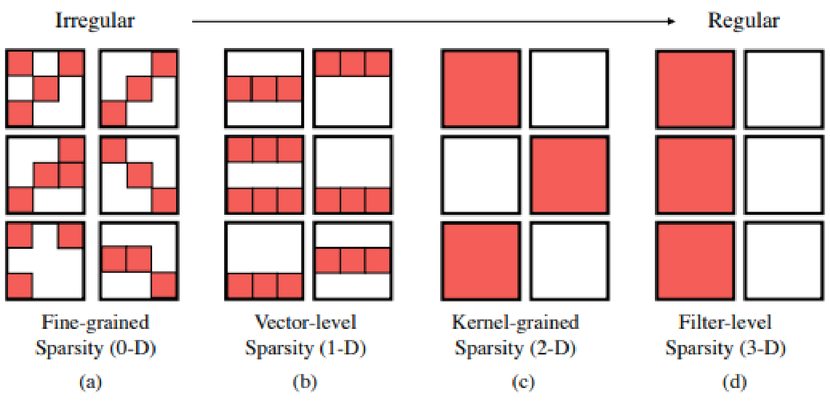

Irregular
Regular
Fine-grained Sparsity (0-D)
(a)
Vector-level Sparsity (1-D)
(b)
Kernel-grained Sparsity (2-D)
(c)
Filter-level Sparsity (3-D)
(d)

Figure 2.13: Different granularities of sparsity, ranging from fine-grain(fine) to filter-level(coarse)[18]

2.3.2 Low-Precision Networks

Separate from the high level architecture, the storage elements for neurons and parameters remain to be optimized. Normally, neural networks are computed in 32-bit floating-point arithmetic. However, there have been several forays in the commercial space towards 16-bit floating-point computation and 8-bit integer storage. Large companies such as Google[19], Nvidia[20] and Intel[21] have all released products with 8-bit headline support.

TensorFlow, the neural network framework by Google, has a specific

[18]Source: (Han, 2017)

[19]Tensor Processing Unit — `https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu`

[20]Tesla processors P4 & P40 — `https://www.theregister.co.uk/2016/09/13/nvidia_p4_p40_gpu_ai/`

[21]Xeon Scalable Processors — (Gong et al., 2018)

implementation of 8-bit integers called GEMMLOWP (GEneral Matrix Multiplication LOW Precision), that can be interpreted as floating-point numbers. (Jacob et al., 2018). As can be seen in Figure 2.14, numbers in this form share exponents with other numbers in a common data range (e.g. a set of weights for a convolution, or a channel of an activation).

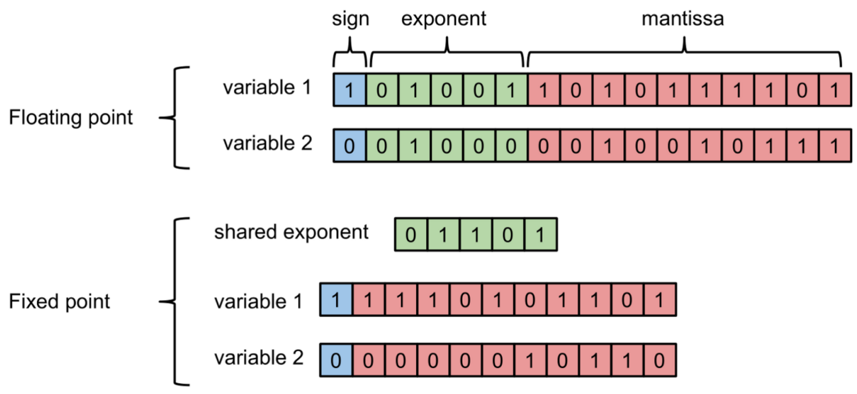


Figure 2.14: GEMMLOWP takes advantage of similarly scaled data ranges between values, storing their shared exponent separately[22]

There have been several papers that experiment with even smaller precision, exploring the possibility of 4-bit, 2-bit and even ternary (Zhu et al., 2017) or binary (Rastegari et al., 2016) arithmetic substitution.

2.4 Using Embedded Systems for Neural Networks

2.4.1 Market Overview

There are a wide range of embedded devices that are on the market today which are specifically designed to accelerate ANN workloads.

Some devices are 'enabling' — devices that provide additional compute power to a host focussed on general task processing such as a Raspberry Pi. Google's Tensor Processing Units ("TPU"s) are one example of these devices, as well as the Intel®

[22]Source: `https://heartbeat.fritz.ai/8-bit-quantization-and-tensorflow-lite-speeding-up-mobile-inference-with-low-precision-a882dfcafbbd`

Movidius™Neural Compute Stick described below in Section 2.4.2. Other devices focus on the full NN development suite, such as Nvidia's Jetson range of processors (in particular, the Jetson Nano was recently released in March 2019).[23]

There are also processors designed for specific market segments, such as MobileEye's "EyeQ" which is focussed on autonomous vehicles. Others focus on specific subsets of ANNs, such as IBM's TrueNorth processor which is designed for Spiking Neural Networks (SNNs).

2.4.2 Intel® Movidius™Myriad™ VPU

Company History

Before Movidius® was acquired by Intel® in 2016[24] (subsequently becoming Intel® Movidius™) their main product offering was a small processing unit dedicated to computer vision workloads. This was one of the first in a new wave of processors called "VPUs" (Vision Processing Units).

The company produced several versions of their processors (ISAAC, SABRE, Myriad 1), each one building upon the successes of the last. A consistent selling-point of the company was their ability to provide remarkable compute power for very low power consumption (below <1 Watt).[25] This was very attractive for developers of 'always-on' or battery-powered devices — such as the DJI Spark Drone[26] or the Google Clips camera[27]

The company entered into the domain of neural networks when they announced a suite of neural network tools on their new Myriad 2 processors. In April 2017, the chip was also released in a new form factor — The Intel® Movidius™Neural Compute Stick (NCS). A brief overview of a NN developer's workflow with the

[23]https://developer.nvidia.com/embedded/jetson-nano-developer-kit?nvid=nv-int-mn-78462

[24]https://www.irishtimes.com/business/technology/intel-acquires-dublin-based-chipmaker-movidius-1.2781200

[25]https://uploads.movidius.com/1463156689-2016-04-29_VPU_ProductBrief.pdf

[26]https://www.movidius.com/news/intel-movidius-myriad-2-vpu-enables-advanced-computer-vision-and-deep-learn

[27]https://androidcommunity.com/intels-movidius-vpu-powers-the-google-clips-camera-20171009/

device can be seen in Figure 2.16.

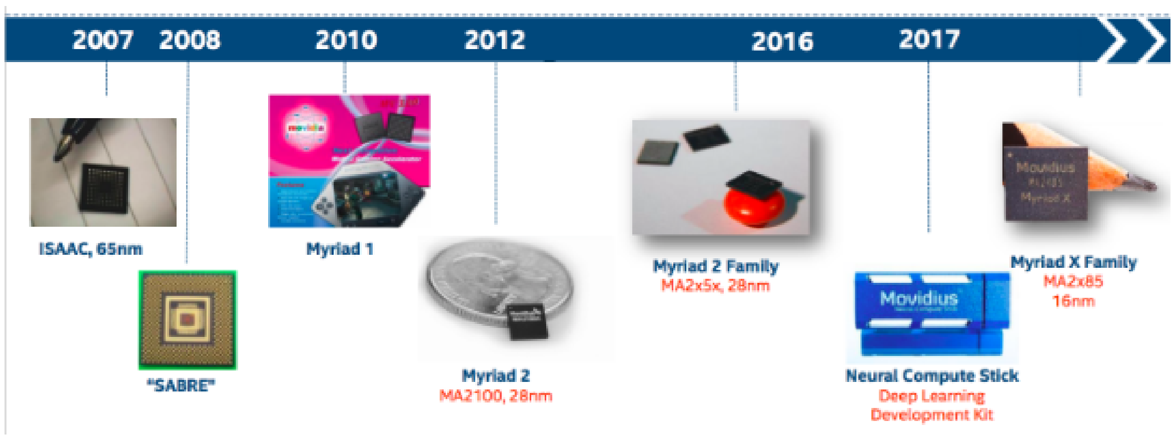


Figure 2.15: Timeline of Movidius® Products[28]

In August of the same year, they announced their newest processor — the Myriad X. The Myriad X VPU contains a "Neural Compute Engine" (NCE) - a dedicated hardware accelerator for neural networks, capable of performing of over 1 trillion operations per second.[29] A subsequent NCS based on the the Myriad X was produced (the NCS 2)[30] allowing NCS developers to take advantage of these performance improvements. A summary timeline of these events can be seen in Figure 2.15

The NCE of the MyriadX is the target platform for this thesis's research. Any improvements to existing systems if accepted by Intel® Movidius™ could potentially be seen in the core Myriad product lines as well as the NCS products.

[28]Source: https://www.movidius.com/

[29]https://movidius-uploads.s3.amazonaws.com/1532110136-MyriadXVPU_ProductBrief_final_07.18.pdf

[30]https://newsroom.intel.com/news/intel-unveils-intel-neural-compute-stick-2/

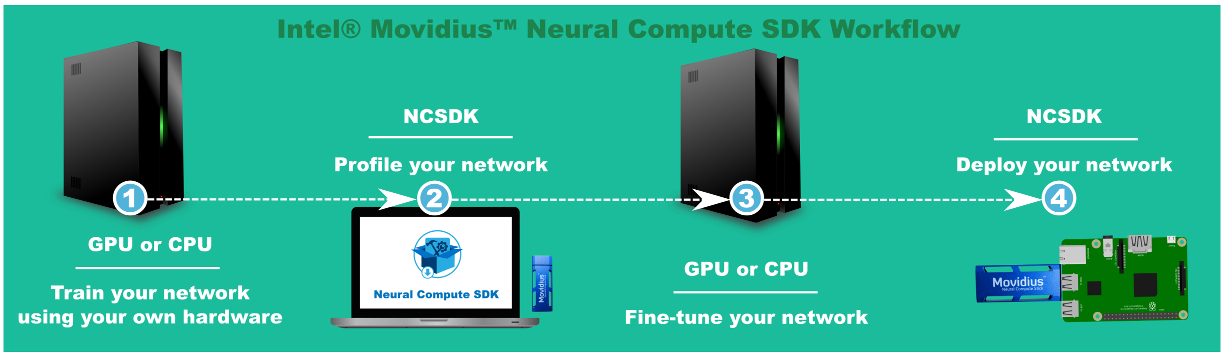


Figure 2.16: Network deployment workflow of the NCS product[31]

Processor Overview

One of the features of the Intel® Movidius™ range of processors is that the architecture was designed to reduce the impact of data movement, leading the processors to be more energy efficient. When not using the NCE hardware acceleration, computation is normally performed using 16 programmable 128-bit very long instruction word (VLIW) vector processors. Careful consideration must be made to balance the transferring of supplies of work to the processors, while still achieving high compute utilization.

These transfers of work are generally relocated to different memory locations using direct memory access (DMA) controllers and are scheduled by the two LEON (SPARC-V8[32] RISC[33]) processors on the device (They are labelled "CPUs" in the Figure 2.17 below that gives a visual layout of the device).

Each of the Myriad range of processors also come with a collection of computer vision hardware accelerators, these are supplemented by a large collection of software libraries in the Myriad Development Kit (MDK).

[31]Source: https://movidius.github.io/blog/deploying-custom-caffe-models/
[32]Scalable Processor ARChitecture
[33]Reduced Instruction Set Computer

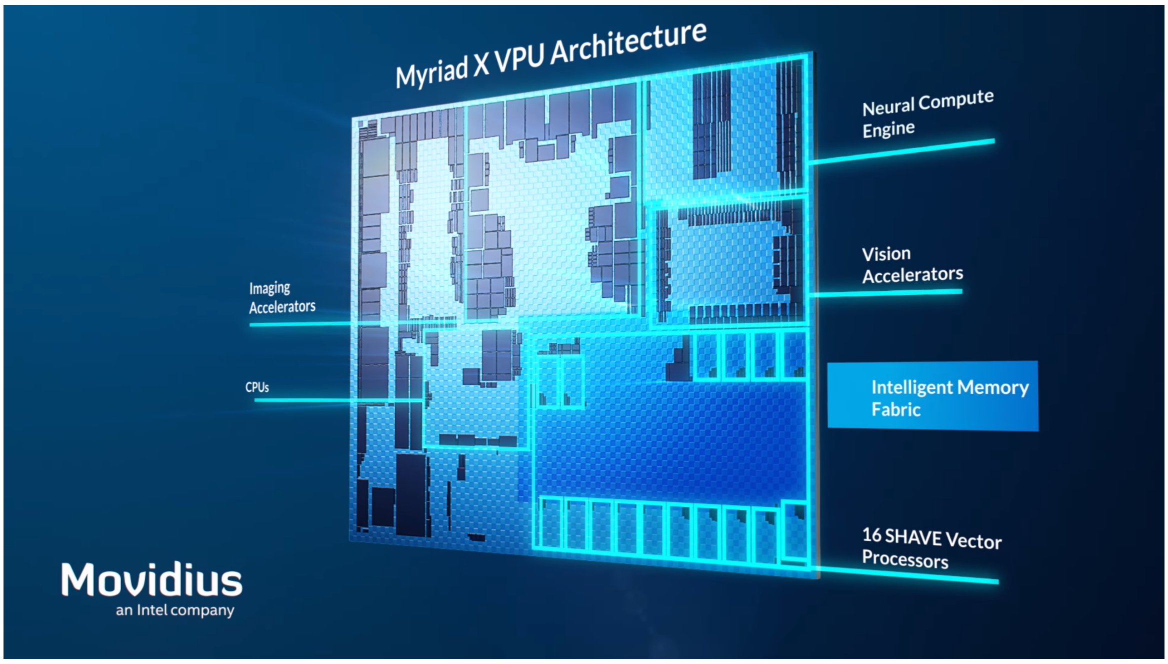


Figure 2.17: High Level Architecture of the Myriad X Processor[34]

NCE Hardware Acceleration

At a high level, the scheduling of NCE workloads is similar to that of software tasks. Instead of scheduling work for the 16 vector processors, work is scheduled for the 2 hardware blocks.

The component design of the NCE is somewhat detailed in Movidius®'s patent application "Methods, Systems and Apparatus to Improve Convolution Efficiency" (Power et al., 2018).

[34]Source: https://www.anandtech.com/show/11771/intel-announces-movidius-myriad-x-vpu

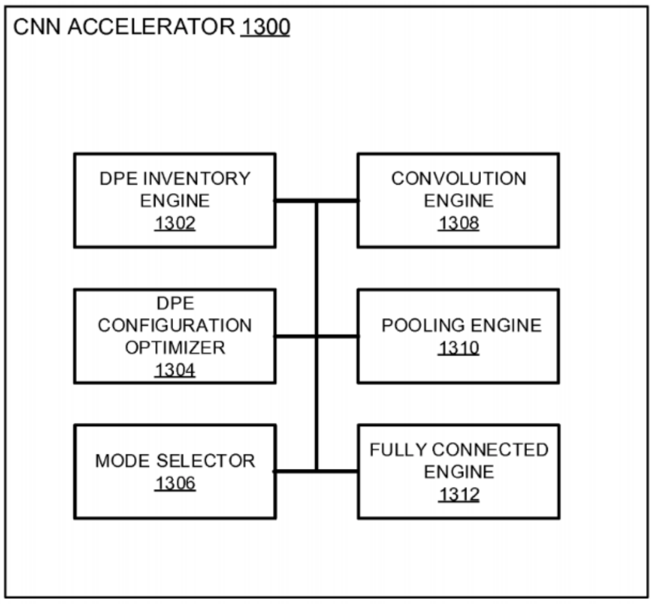

CNN ACCELERATOR 1300
DPE INVENTORY ENGINE 1302
DPE CONFIGURATION OPTIMIZER 1304
MODE SELECTOR 1306
CONVOLUTION ENGINE 1308
POOLING ENGINE 1310
FULLY CONNECTED ENGINE 1312

Figure 2.18: High Level Components of the CNN Accelerator[35]

In the simpler view of the accelerator — Figure 2.18 — it can be seen that the NCE supports several core CNN operations — convolution, pooling and FCLs. The patent also discusses several other supported operation types, however the only other operation we are concerned with in this thesis is ReLU.

The NCE consists of an array of data path elements (DPEs). As can be seen in the more detailed Figure 2.19, there are 256 DPEs. They can be arranged to all process a single channel, to split in two sets of 128 to process two channels, and so on.

[35]Source: (Power et al., 2018)

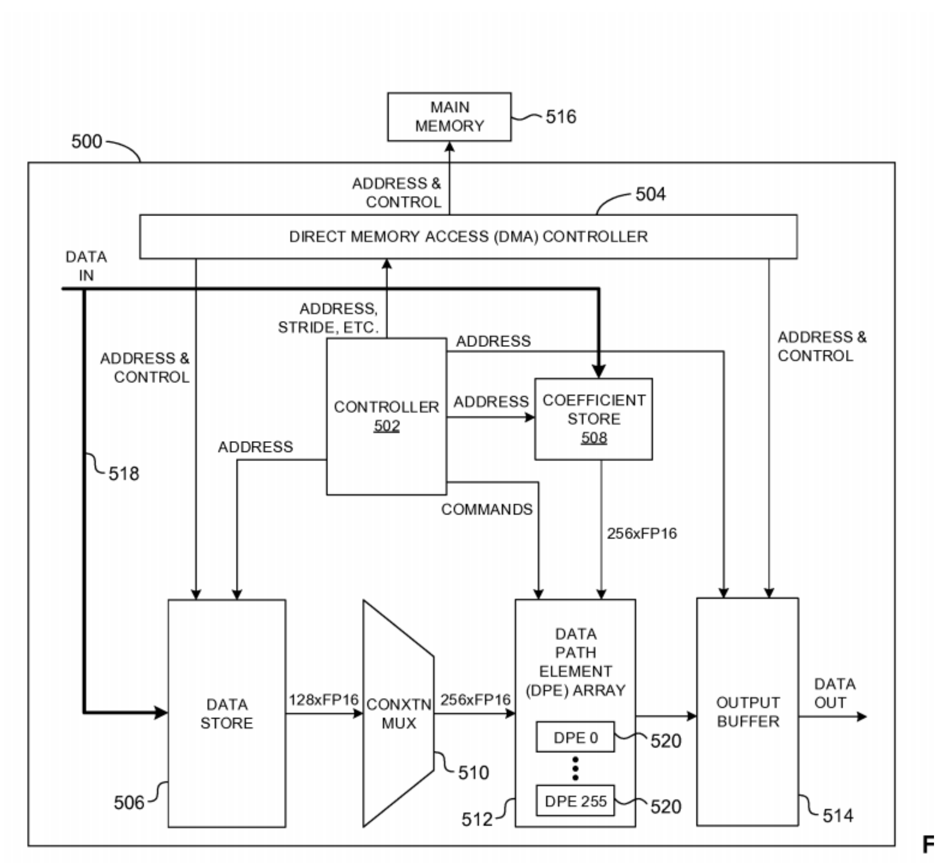


Figure 2.19: Technical Breakdown of the CNN Accelerator[36]

While Intel® Movidius™ advertises many impressive headline figures for 8-bit operations with this hardware, the engine also includes unpublished limited support for lower precisions — 4-bit, 2-bit and 1-bit.

Index Mapping Hardware

An additional facility of the compute engine is index-mapping from 4-bit keys to 16-bit floating-point values. This feature gives developers the option to regain some precision from the loss that occurs when quantizing from a higher precision to 4-bit. An example mapping is shown in Table 2.1.

[36]Source: (Power et al., 2018)

It is this unit that this thesis will focus on, to investigate whether there is a consistent method for choosing how to select index-mapped values for a given neural network to reduce incorrect classification.

4-bit index	16-bit value
0000	-3.0
0001	-2.5
0010	-2.0
0011	-1.8
0100	-1.5
0101	-1
0110	0.3
0111	0.8
1000	1
1001	1.2
1010	1.4
1011	1.8
1100	2.4
1101	2.6
1110	2.9
1111	3.5

Table 2.1: Example of Index Mapping

Using low-precision types in hardware reduces the overhead for storage, computation and transmission. As the 4-bit values will resolve to a higher precision within the NCE, it is expected that computation savings will be lessened or absent.

Research from (Chen et al., 2014) and (Tann et al., 2016) both mention an additional benefit of using low-precision types — the size of processed memory directly correlates with better power and energy efficiency. A graph from (Hashemi et al., 2017) below (Figure 2.20) profiles a selection of mixed-precision networks and their consumed energy during execution. The blue and black points are highlighted to show Hashemi's confirmation of Chen and Tann's findings.

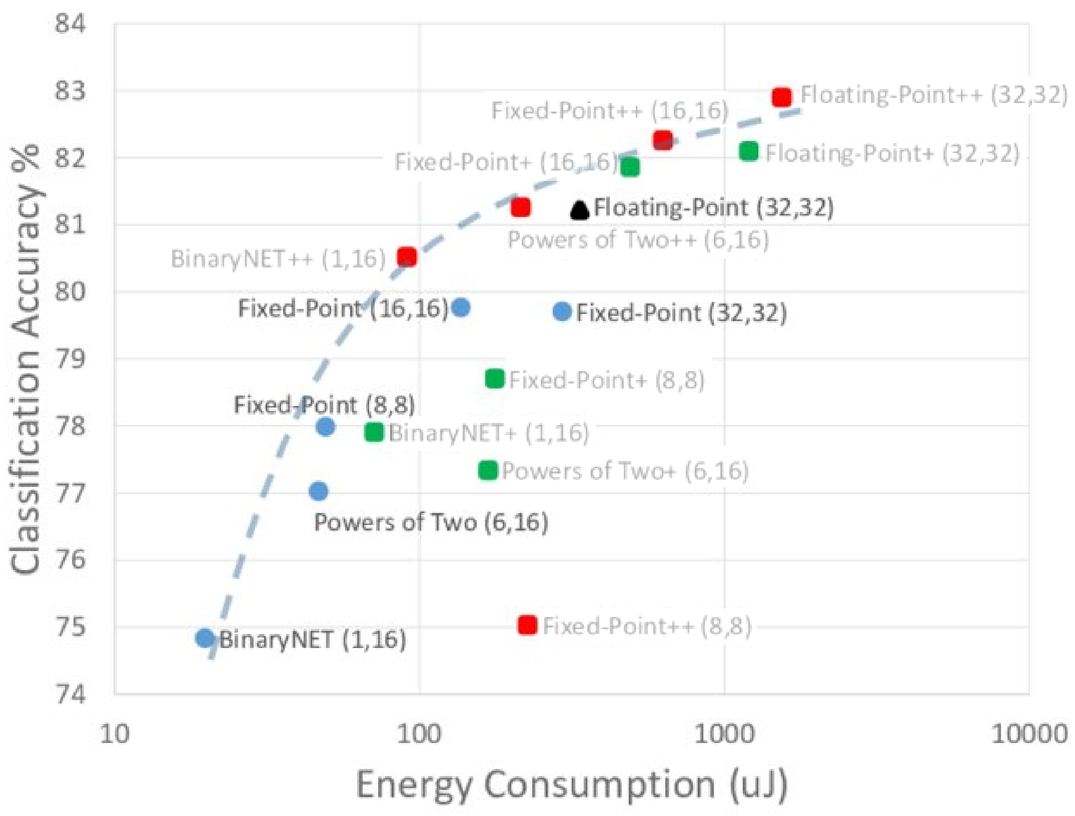


Figure 2.20: CIFAR10 Power Consumption Graph[37]. The energy consumption of the original precision (black dot) is compared to the energy consumption at different precision representations (Blue). Classification accuracy can be seen also on the y-axis. The green and red points are additional types introduced in the research not relevant to this thesis

2.4.3 Related Hardware Research

This section provides a brief overview of other existing hardware considerations applicable to this thesis, but descoped from future investigation. See Section 6.2 of the final chapter for more detail on possible future research with these items.

Sparsity

[37]Source: (Hashemi et al., 2017), edited to highlight the relevant plotting marks

Sparsity (as briefly explained in Section 2.3.1) is increasingly a consideration for embedded hardware designers.[38] As most NN operations with an operand of zero typically have a zero or identity result, a “quick-path” can be included for this specific case of computation. This will result in faster throughput and improved energy-efficiency, as a result of the reduction of active circuitry.

However, running a non-sparse network through hardware designed for sparsity may incur some overhead. In the hypothetical example below (Figure 2.21), each non-sparse computation has additional checks for sparsity.

[38]Sample hardware proposal: (Dey et al., 2018)

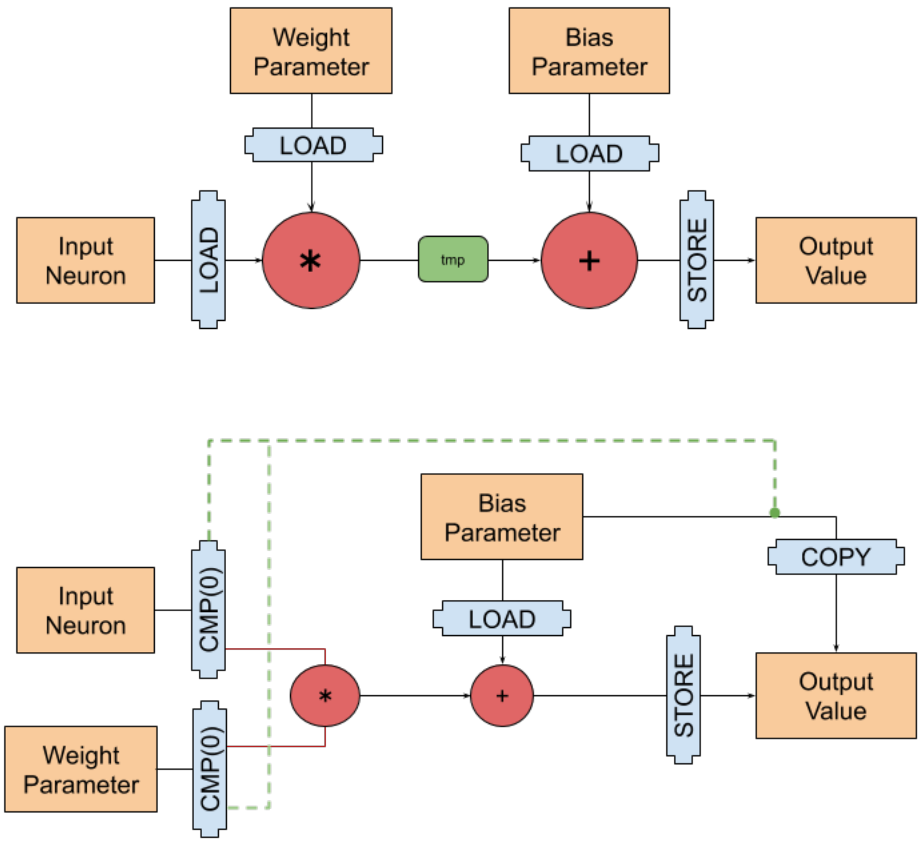


Figure 2.21: This hypothetical example introduces some early comparisons into the logic of computing a convolution so that compute will be bypassed when a zero is present in either input or weights

While this existing hardware concept can work well with the research within this thesis, such sparsity optimizations are not yet implemented in a public Intel® Movidius™ processor.

Training

With current technology, low-power embedded systems are realistically constrained to testing only the classification steps of a neural network.

At the time of writing, the current world record for training a reasonably functional neural network on the ImageNet dataset is 90 seconds (Sun et al., 2019). This however, involves a system with 512 GPUs. The author does not believe that the full system training on an embedded device will be viable in a commercial system in the near future.

However, there are some techniques to adapt networks that are already trained to new datasets that may be a more achievable goal (Yosinski et al., 2014). Regardless, the computation overhead of researching in this area inclines the author to prioritize their time on classification.

2.5 Related Work

Despite extensive preliminary research, we unfortunately missed some closely related work by (Han et al., 2015a) until later in my study. A short summary of their work is given below;

2.5.1 Weight Sharing

Song Han et al. also use K-means clustering (as we do in Section 3.2.2) to group their quantization values. They refer to an additional earlier implementation that uses hash tables by (Cao et al., 2017).

Han uses three different initialization algorithms — Random, 'Density-Based' and 'Linear'. In the paper he shows that the Random and Density-Based initializations are less effective due to over-representation of low-valued elements (based on knowledge gained from his paper in 2015 (Han et al., 2015b)).

While we experimented with both Random and Linear (named "Regular Splitting" in this thesis — see Section 3.2.1) initializations, these were mostly used as comparison baselines against our more experimental proposals.

Additionally, the limited-precision constraints of this thesis add new information to the problem which justifies re-assessment of their approach.

Below, Figure 2.22 shows an example result of his linear method, which can be seen to operate as our proposed regular splitting does — it tries to represent an equal amount of values in each quantization group.

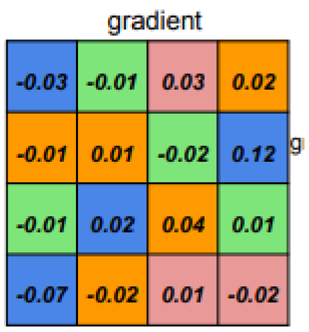


Figure 2.22: An example of weights being grouped into 4 quantization groups from Han's paper[39]. Notice how each group has roughly the same amount of values.

[39]Source: (Han et al., 2015a)

3 Problem Overview

3.1 Technical Approach / Methodology

In this chapter, we perform our own investigations into the thesis subject. We do some initial exploratory work on both parameters and activations of CNNs, hoping to inform our future experiments. We also make sure that the research problem has scope for us to improve upon existing solutions.

After this investigation, we present some algorithms that we will use in Chapter 4 to attempt to find a solution that better quantizes values than existing algorithms provide.

Finally, we discuss the technical environment that was used for creating and testing these algorithms.

3.1.1 Initial Target Analysis

We developed a series of custom tools to analyse well-established neural networks under the effect of quantization. Repeatable observations (e.g. if the half the data was the same number) in the data could lead to possible optimizations or entirely new techniques to be investigated.

The technical details of the construction of this system are described in Section 4.4. Several observations were made from this effort, some of the more notable items are listed below;

1. Weights appear to have a normalized distribution. Most weights profiled

were centered close to zero, however some operations had a slightly positive center. This may be a side effect of activation operations preceding the profiled operation and we discuss this possibility at the end of this list in a separate observation. The normalized distribution may be a result of the work done by (Han et al., 2015b) on reducing the noise present in weights during training.

2. Unlike activations (which have been shown to have locality in (Zhou et al., 2016)), weights appear free of any localized groupings (See Figure 4.2). This observation is unsurprising based on the technical operation of NN algorithms; Activations are derivations of images, audio and other natural data, whereas weight parameters are convolved over every element of an image, so each resultant value is averaged as a consequence.

3. The distribution of the values in the first activations in a network can differ significantly to other activations throughout the network.

   One explanation for the difference is that inputs to convolutional neural networks tend to be from natural sources and are non-uniformly distributed. Images have certain lighting conditions, subsets of colour and all manner of environmental factors at play that make it unlikely to have a balanced profile. Similarly, an audio piece consists of different pitches, volumes, instruments and clarity that affect the recording as well as the physical equipment used and background environmental effects.

   Intermediary activations have been processed against the network parameters which are averaged from training. This limits the probability of significantly outlying data points.

   Some networks require preprocessing of input data before inference.[1] While preprocessing the data of an image can reap other benefits (such as reducing the complexity or size of an image), this step seems to alleviate the lack of a regular distribution of values; a common application is the subtraction of the means of the dataset's average input channels. This should reduce the impact of imbalances in the current input. It seems, at least to the author,

[1] `http://lamda.nju.edu.cn/weixs/project/CNNTricks/CNNTricks.html`

that the current solution of pre-processing may alleviate the issues, but may not remove them entirely[2].

4. The later in the network that an operation occurs, the smoother the distribution of weight values along the number line were (See this in effect between the first convolution of GoogLeNet in Figure 3.1 and one of the last convolutions of the same network in Figure 3.2). The data also becomes smoother and approaches the form of a Gaussian curve. Both 3.1 and 3.2 have a dotted-line showing the closest Gaussian curve match for the shown data.

   The author assumes the smoothing effect is an extension of the first observation noted in this list, and that as the network continues to the next operation, the averaging effect of grid-based computation and data loss from layers such as pooling and rectification units, may statistically threshold the data of later operations to be within predictable ranges.

   For the operations near the beginning of a network that show a rough distribution, certain quantization algorithms that quantize based on some assumptions of value distribution (e.g. Logarithmic Split (see: Section 3.2.1)) may produce worse results. It may be that these algorithms will perform best near the end of a network.

[2]e.g. A white image in a dataset of black images will still be a large difference in value

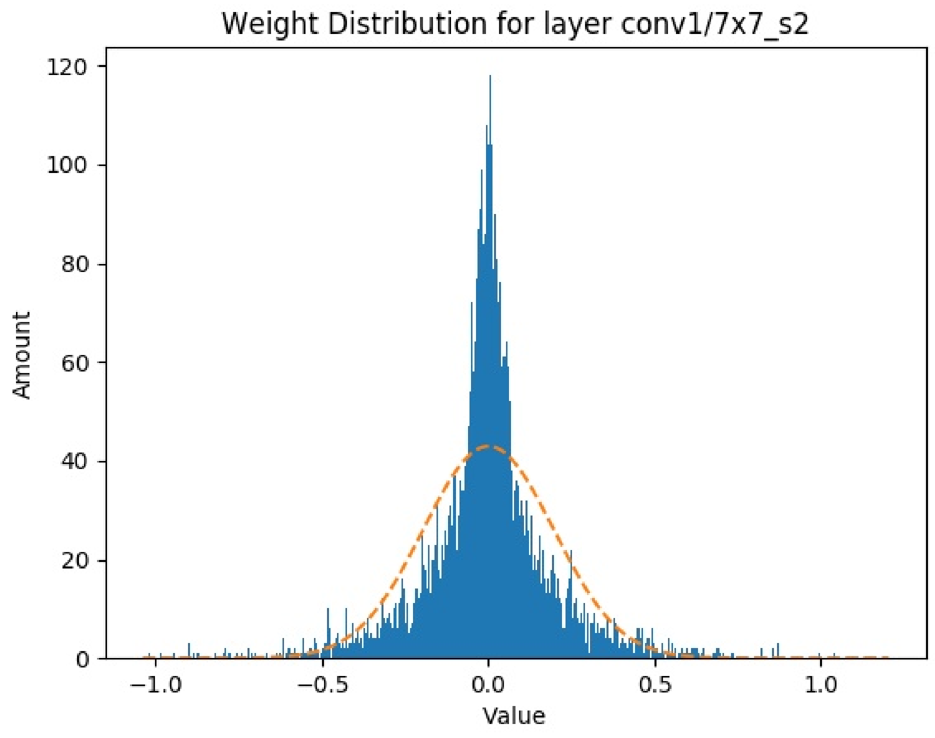


Figure 3.1: The first operation of the NN GoogleNet showing a rough distribution of the magnitudes of its weights.

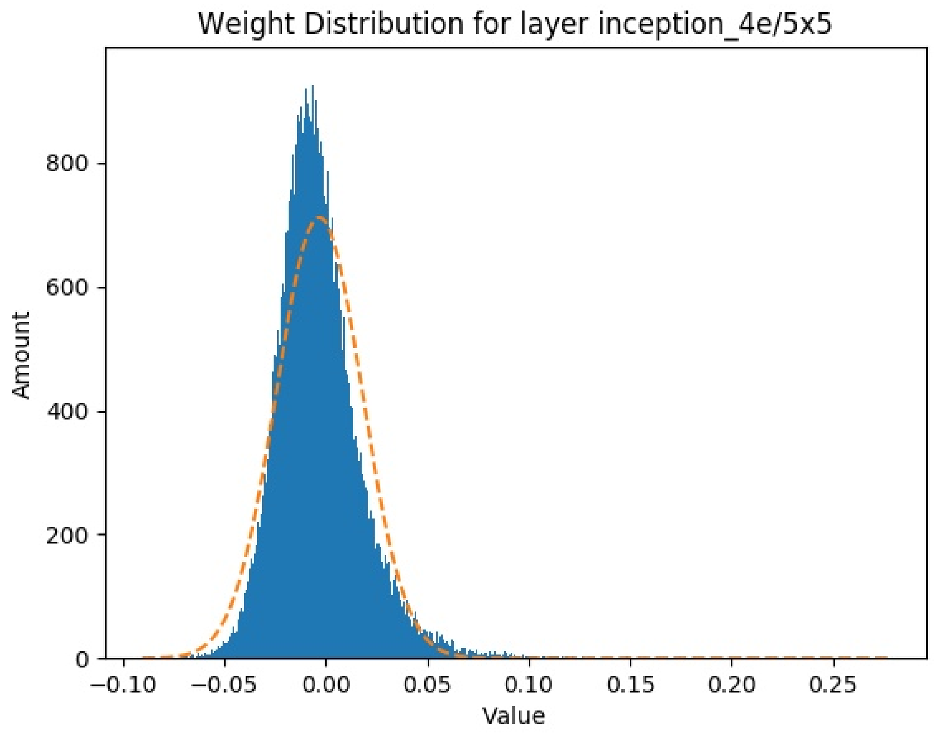


Figure 3.2: One of the last convolution operations in GoogleNet showing a much smoother distribution than Figure 3.1 and also a weighting towards positive significant outliers

5. While the networks we profiled were generally symmetrically distributed around the zero point, operations which had ReLUs tended to shift weight parameter values towards a more positive skew. These values tended to have more positive values than negative, larger positive than negative outliers and the center of symmetry tended to be slightly positive. The resultant values of an operation such as convolution could further magnify this effect as can be seen in Figure 3.3, where all multiplication combinations of activations and parameters are graphed. Because the outliers for negative values are smaller, they cannot create a result of lower than -1, causing the subsequent operations to also likely exhibit the inequal outlier biases. We speculate then, that the introduction of a ReLU to the network has effect longer than its immediate application.

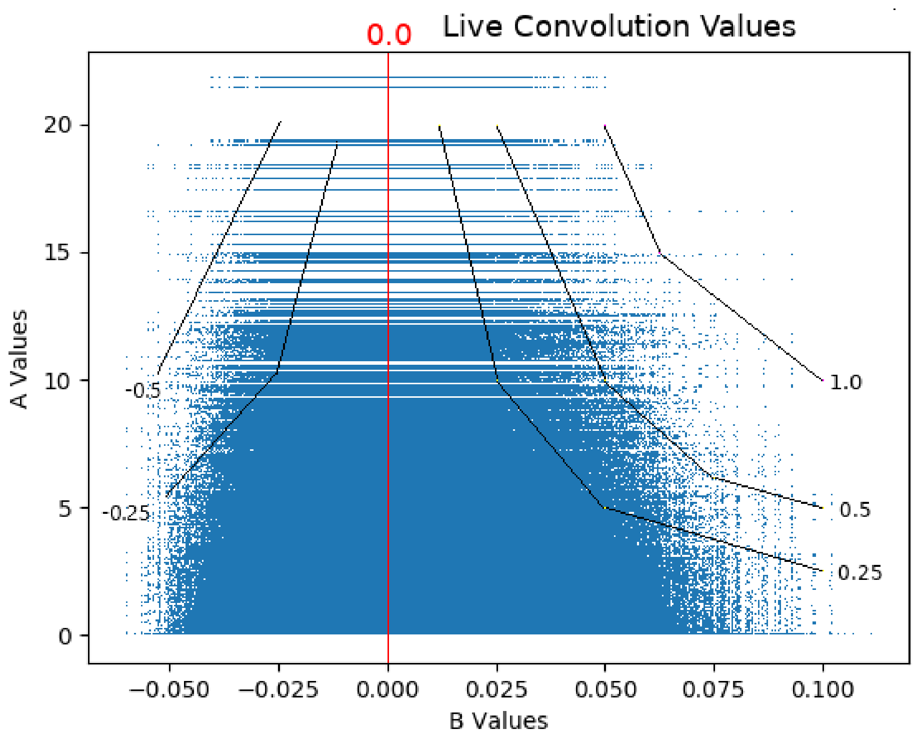


Figure 3.3: The values of the activations(A) and the weights(B) are plotted on a 2D plane. The weights are symmetric around zero, shown by the vertical red line. The plot was sampled at several points for possible multiplication results (e.g (-0.050 * 10 = -0.5)) and rough bands of resultant values in incremental steps were overlaid on the graph (in black). It is notable that positive B values may cause a resultant magnitude of over +1.0, while negative values cannot cause values below -1.0.

3.1.2 Asserting the existence of a better solution

To have a solid foundation for subsequent research effort, it was necessary to verify that a solution that had more correct classifications than current approaches did exist.

We created a tool (shown in Figure 3.4) to show how a quantization was applying to a set of weights. A developer could load in one of the naïve algorithm's quantization choices and adjust the exact placement of each quantization threshold as they like. In theory, this tool could be used to manually quantize the network's parameters to their optimal placings, provided those locations were known.

After updating the quantization bounds, the display would reload and would show the effect of the new choices — How much numeric and accuracy loss was observed (as compared to the original unquantized network), the number of values represented in each quantization value and how a histogram of the values in the original model compared to a histogram of the new quantization.

After several small tweaks on different network operations, it was quickly apparent that tweaking even a single of these thresholds a minimal amount from the original setup could yield improved results, both in terms of reduced numeric data loss and better classification results when saved back to the framework's network files for inference.

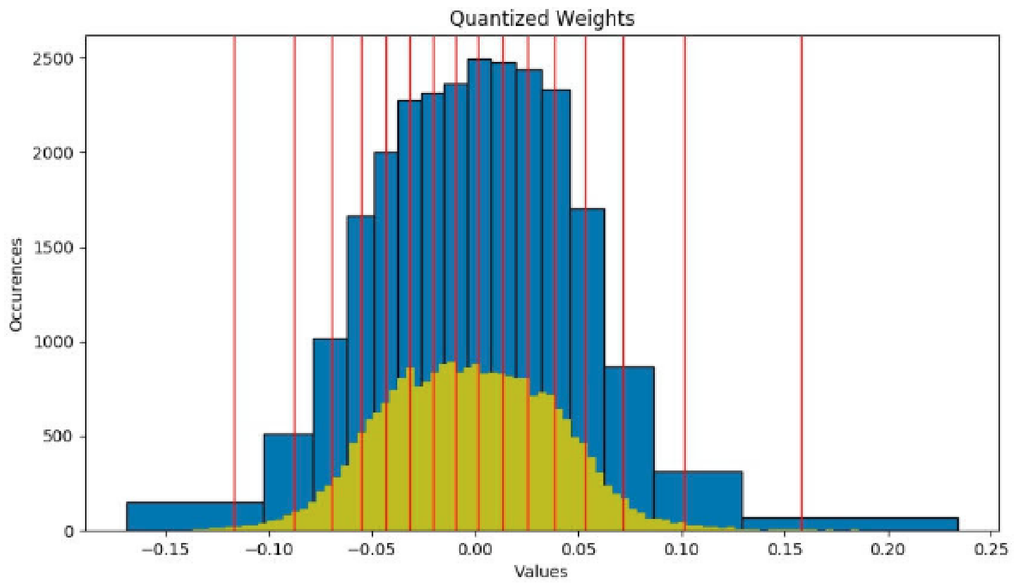


Figure 3.4: Manual Quantization Tool: The original values of a set of weights are shown in green. A default quantization algorithm groups these values into several "buckets" shown in blue. A user can drag the red bars (representing the buckets' midpoint values) to select a new quantization manually. They can also do this on textboxes for fine-grain control not shown in this screenshot.

Having established that existing methods were suboptimal, the research effort was directed fully into closing this optimization gap with automatic methods.

3.1.3 Profiling the Target Processor

Intel® Movidius™ provides customers with a Software Development Kit (SDK/MDK)[3]. The MDK contains "over 300 computer vision kernels, neural network and linear algebra libraries and drivers for both internal and external components". Figure 3.5 shows some of the main software packages. In particular, the MoviTools and Board Support Package will be used.

The particular hardware element this project is interested in, as described in Section 2.4.2 is the Neural Compute Engine hardware component. The MDK's Board Support Package allows developers to interact with the component through a set of low-level drivers. Using these, we profiled the difference in execution time when running in the index-mapped mode versus the default of 16-bit floating-point.

[3]`https://www.movidius.com/solutions/software-development-kit`

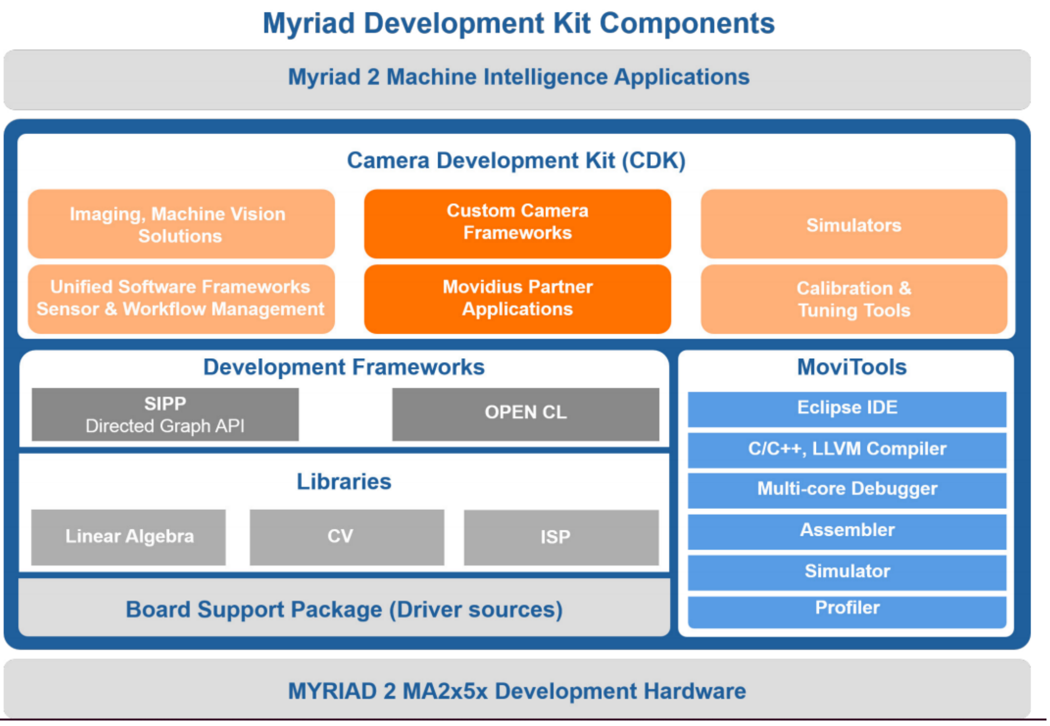

Myriad Development Kit Components
Myriad 2 Machine Intelligence Applications
Camera Development Kit (CDK)
Imaging, Machine Vision Solutions
Custom Camera Frameworks
Simulators
Unified Software Frameworks Sensor & Workflow Management
Movidius Partner Applications
Calibration & Tuning Tools
Development Frameworks
SIPP Directed Graph API
OPEN CL
MoviTools
Eclipse IDE
C/C++, LLVM Compiler
Multi-core Debugger
Assembler
Simulator
Profiler
Libraries
Linear Algebra
CV
ISP
Board Support Package (Driver sources)
MYRIAD 2 MA2x5x Development Hardware

Figure 3.5: Component Hierarchy of the MDK (Myriad 2)[4]

Computation time of a convolution on the NCE was observed to be effectively identical when processing either a 16-bit floating-point convolution or a 4-bit index-mapped convolution. As the computation hardware is the same for both modes, this observation was expected.

However, because the information being transferred from main device memory to the hardware's localized memory is four times as small as the original data, we saw improvements in transfer speed and, as a consequence, total execution time.

The primary method for transferring large amounts of data on the Intel® Movidius™Myriad™ VPU is through a Direct Memory Access (DMA) engine. This process was initially profiled, but due to IP concerns, permission was not granted to publish these figures. However, the standard implementation of the C++ function 'memcpy' was profiled instead on the device which gave similarly satisfactory results.

[4]https://www.movidius.com/solutions/software-development-kit

When profiling we used a mixture of real-world transfer sizes (where the transfer size would be the sum of the activations, parameters and results) and some incrementally stepped values for a useful analysis set. We observed a near-consistent scaling rate of 400% using these various sizes.

When profiling the DMA engine, we observed a less consistent scaling. The DMA engine's efficiency is based on how well the data fits its transfer packet sizes (e.g. a 10 byte transfer and a 60 byte transfer could take the same time to execute if the packet size was 64 byte). However, all cases did improve significantly.

3.2 Algorithm Proposals

After confirming that current quantization methods were suboptimal, we propose several algorithms that could potentially provide a more accurate representation.

Several of these algorithms are listed below with pseudo-code and are intended to be applied to the weight parameters of convolutional neural networks. We originally experimented with quantizing activation values also, but it was much more difficult to assess as they would change for each input item.

The results of our analysis on the effectiveness of each algorithm is discussed in Section 5.1.1.

3.2.1 Naïve Algorithms

Regular Splitting

This algorithm divides the range of data so that each resultant index-mapping covers the numeric range of values equally. In the Figure 3.6 below, the range -1, 1 is split into six equal sections. Each section will use the midpoint as its quantized value. For this example, the quantized values would be (-0.75, -0.5, -0.25, 0.25, 0.5, 0.75).

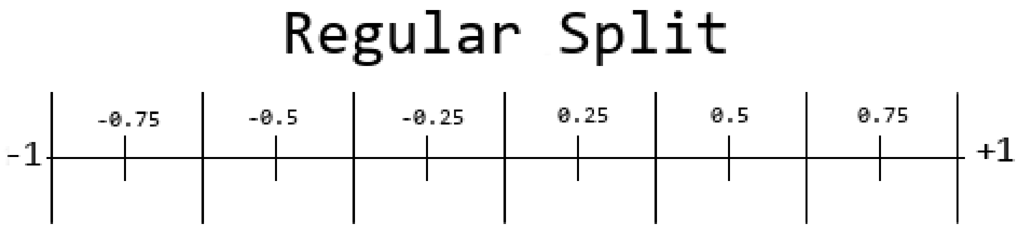


Figure 3.6: The algorithm splits the range between the max and min point in equal sections

Given a flattened matrix A sorted ascendingly of length n, where A_n is an $\in$ of $\mathbb{Z}$, the formula below will produce the start and end indices of separation in resultant matrix B, that will divide A into m equal sections of data of size S. The values inside these sections will be quantized to the median value of the section's data range.

$$x = min(A)$$

$$y = max(A)$$

$$S = (x + y)/m$$

$$B = \{(\, x, x + S)\ , (\, x + S, x + S * 2) ..., (\, x + S * m - 1, x + S * m)\ \}$$

Equivalent Population Split

This algorithm divides the data so that each resultant index-mapped value covers the same quantity of values. Figure 3.7 shows one of the convolutions from GoogLeNet split in four with this algorithm.

Listing 3.1: Equivalent Population Split

```
# sort the data in ascending order
data = sorted(data)
# using 16 buckets for the 4-bit index-map
num_buckets = 16
```

```
# the size of each ``bucket''
# if divided evenly into the original data
bucket_size = len(data)//num_buckets

for x in num_buckets:
  # ``buckets'' object will contain
  # (start,end) values for each separate grouping.
  buckets.add(x*bucket_size, (x+1)*bucket_size)
```

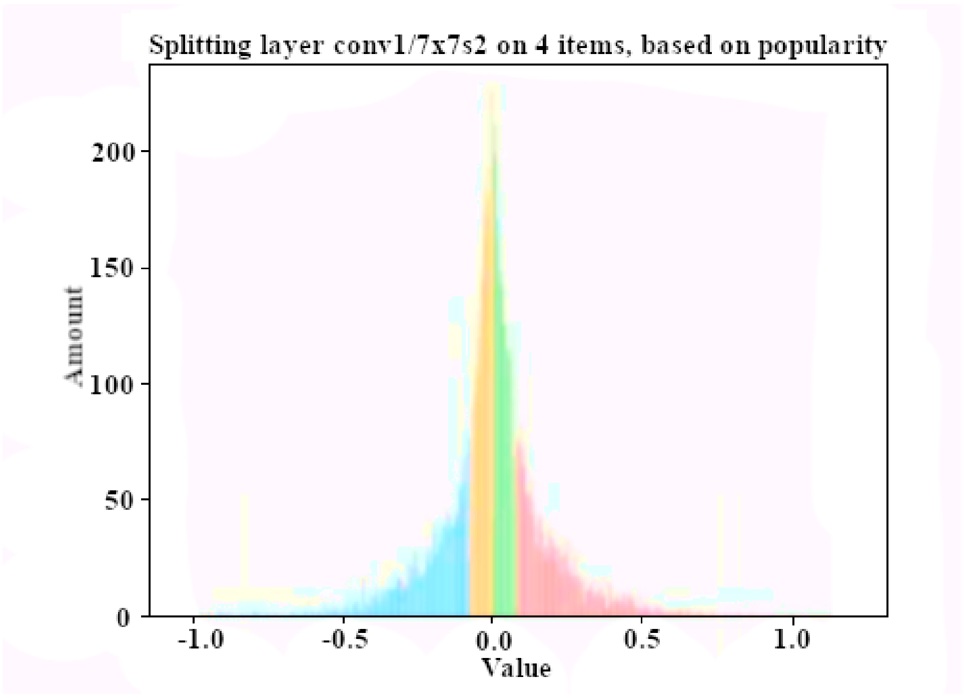


Figure 3.7: An example of splitting based on population with 4 values

Random Split

This algorithm splits the data by selecting several values in the data using a uniform random selector. These points are then ordered and act as the midpoints of the quantization ranges. Due to the nature of random selection, this method has the potential to create useless quantizations.

Listing 3.2: Random Split

```
# Choose 15 random divisive points within the data range.
# They will be sorted ascendingly
selection = sorted(random(amount=15,
        start=data.min, end=data.max))
selection.prepend(data.min)
selection.append(data.max)

bucket_start=0

# Translate to start-end tuples
for s in len(selection)-1:
    bucket_end=midpoint(selection[s], selection[s+1])
    buckets.add((bucket_start, bucket_end))
    bucket_start=bucket_end
```

Logarithmic Split

This algorithm is a recursive splitting of the data range, starting with the midpoint and subdividing the two resultant halves, repeating until the desired number of splits is reached. In Figure 3.8 I have shown the incremental splitting in different colours.

This algorithm works on the assumption that lower magnitude values are less affected by degradation (using magnitude as saliency, described in Section 3.2.4).

Note that the nature of this algorithm limits itself to $N\times2$ quantization values.

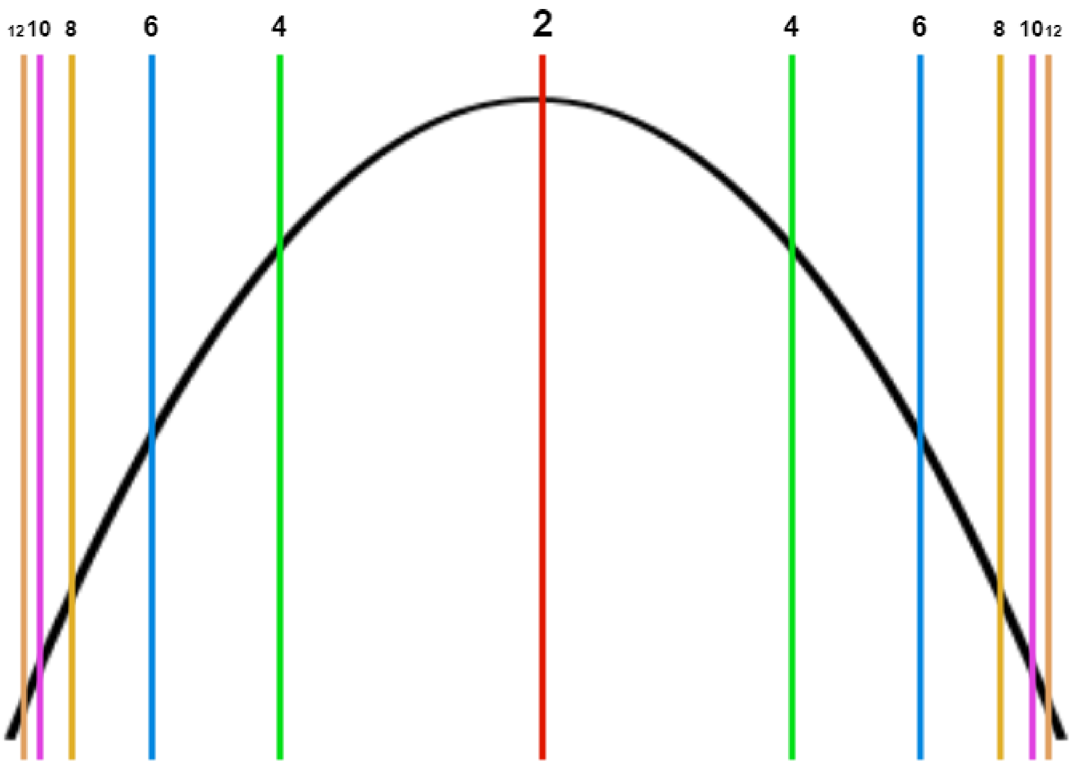


Figure 3.8: Incrementally adding logarithmic splits

Listing 3.3: Logarithmic Split

```
# to get 16 splits (2, 4, 6, 8, 10, 12, 14, 16)
global recurse_limit = 8

def recurse_l2(data, lim):
  if (lim == recurse_limit):
    # at depth which we should start returning values
    return midpoint(data)
  else:
    # continue to traverse
    # Divide data range in two halves
    data.left, data.right = data.split()
    result.append(recurse_l2(data.left, lim+1))
    result.append(recurse_l2(data.right, lim+1))
  return result

buckets = recurse_l2(data, 0)   # get midpoints
```

3.2.2 Cluster-based Algorithms

Clustering algorithms are commonly used when researchers would like to group data elements together in roughly categorized sets. These groupings, or ‘clusters’, are decided based on proximity of data points to others, similar items are grouped together. In Figure 3.9 below, the colors of the image change to their closest match. Similarly, the values used in our NN operations should change to their closest quantization value.

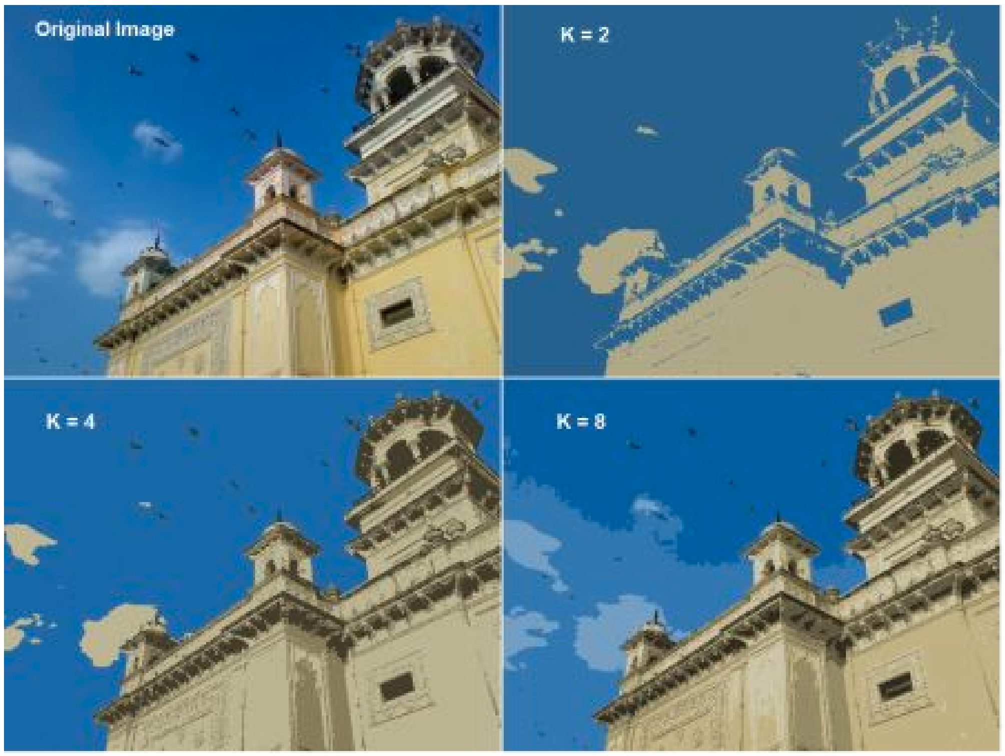


Figure 3.9: An example of reducing the colour space of an image with clustering using different cluster sizes (K)[5]

K-means clustering

K-means clustering is an algorithm that came to computer vision research from the domain of signal processing. Originally proposed by (MacQueen, 1967), it is one of the most commonly used algorithms for clustering two-dimensional data.

K-means clustering works by placing all data points inside a two-dimensional "feature space" where it is possible to group properties by their relative proximities. The number of clusters in a system is a parameter provided by a developer; Increasing or decreasing the number of clusters will change the accuracy of the clusters' representations of the original data (Morissette and Chartier, 2013). An example is given in Figure 3.10 where the 2-dimensional data is neatly grouped in three clusters.

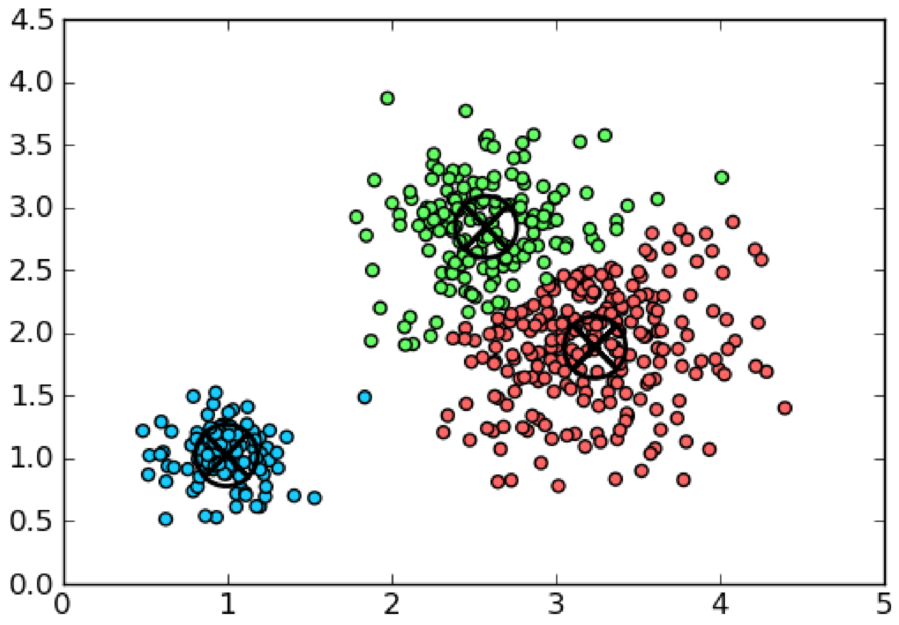


Figure 3.10: K-means clustering data using 3 clusters. $\otimes$ marks the centroid of each cluster[6]

[5]Source: `https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_ml/py_kmeans/py_kmeans_opencv/py_kmeans_opencv.html`

[6]Source: `http://blog.mpacula.com/2011/04/27/k-means-clustering-example-python/`

In this thesis, the data for consideration has only a single dimension (the magnitude of each parameter), but the clustering technique works just the same (i.e. a **1xN** shaped clustering is a subset of **MxN** clustering).

The algorithm consists of two main steps: *Cluster Assignment* and *Centroid Movement*[7].

For the first iteration of the *Cluster Assignment* step, the user must provide some initial centroids for the algorithm to use. Centroids are values that represent the center of a cluster. There are several common ways that these values are established — from naïve random selection to sophisticated algorithms such as k-means++ (Arthur and Vassilvitskii, 2007). This thesis profiles several of these initializations, as well as using results from previous experimental algorithms as new alternatives.

As k-means++ reduces the steps needed to reach one of the exit conditions of the algorithm, it was used as the predominant algorithm in later experiments that had issues completing in a reasonable time (See Section 4.7.1).

After these values have been set, all nearby data points are assigned to that cluster.

For the subsequent iterations, the algorithm will use the output of the *Centroid Movement* step in place of the user-provided values.

Next, for the *Centroid Movement* step, the centroid values are changed to the actual midpoint of the associated data points.

This two-step cycle continues, each iteration improving on the previous approximation of the centroid values until either there are no further significant improvements or some processing timeout has been reached.

We include Figure 3.11 to help visualize the two steps

[7]`https://bigdata-madesimple.com/possibly-the-simplest-way-to-explain-k-means-algorithm/`

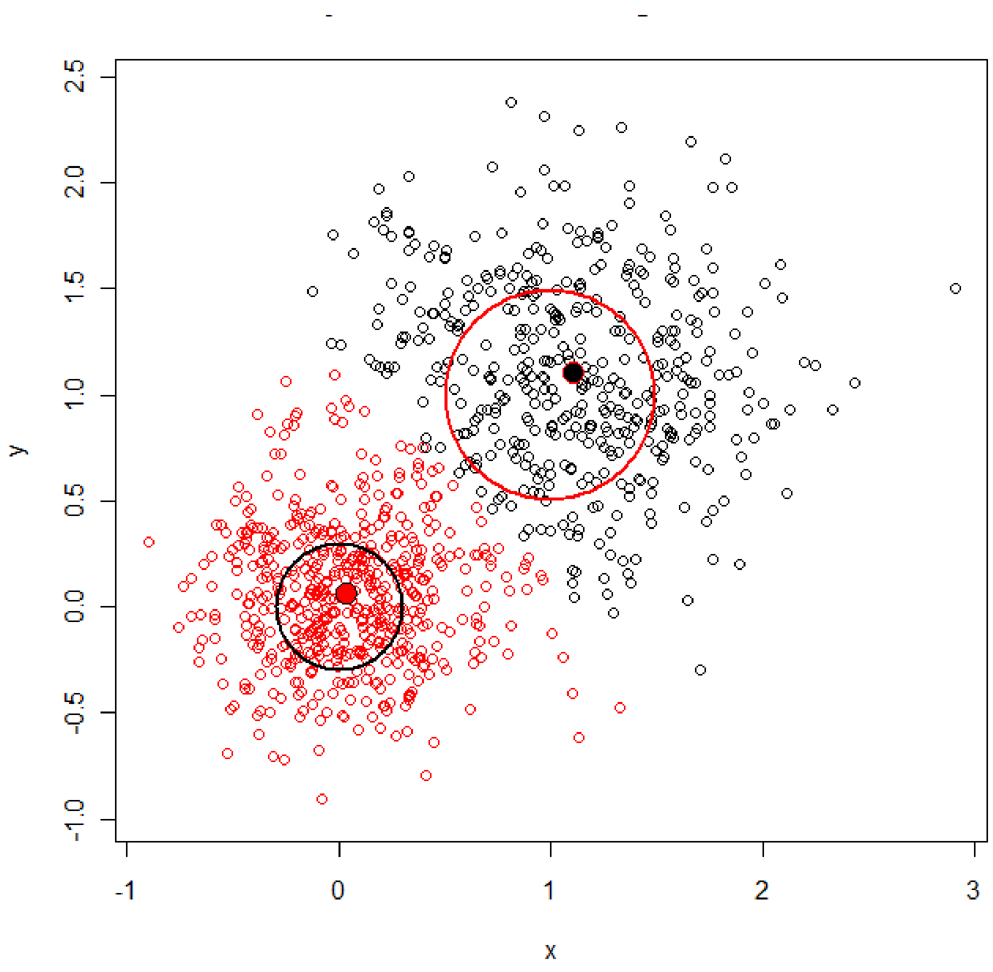


Figure 3.11: The dots show the centroids of each cluster from the previous pass. Each data point was then assigned to its nearest cluster in the *Cluster assignment* step. The center of the larger circles show the midpoints of these new clusters. *Centroid Movement* chooses these as centroids and the process repeats.[8]

CK-Means

In the two-dimensional space and above, the k-means problem is NP-Hard as proven by (Vattani, 2019), so an optimal solution is not guaranteed.

[8]modified from original source at `https://blog.revolutionanalytics.com/2011/06/kmeans-big-data.html`

However, a variant algorithm exists called ck-means that operates solely on 1 dimensional data, and claims an “optimal” clustering result where the distance from each value to the nearest cluster mean is minimized (Wang and Song, 2011). This is an interesting algorithm to assess, as we can test whether an optimal clustering distance also results in an optimal clustering for our quantization.

3.2.3 Forced-Zero Algorithms

Earlier in Section 3.1.1, we observe that weights in neural networks have been generally symmetric around zero. We speculate that this central zero value may be important to preserve to allow balance between the positive and negative values.

Accounting for slight biases in symmetry is expected to have less impact, as while many values surround the central point, they are all low in magnitude, and consequentially, likely less important to the final calculation.

In our initial work with quantizing activations (before focusing solely on weight), we were aware that zeros can be particularly important to preserve as there will be a significantly higher proportion of them after rectification units such as ReLU (see Section 2.1.3).

A tangential benefit of ensuring zero values are preserved is that potential future integration work with sparsity can be enabled (see Section 6.2).

Most of the algorithms were modified to have an option to force the zero value to be one of the quantization values. In the simpler algorithms, we clamp the closest calculated value to 0 (e.g. the random quantization algorithm); In more complex cases, particularly the clustering algorithms, we modified the algorithms themselves to set a value to zero and preserve it throughout the algorithm’s execution.

3.2.4 Salience-based Algorithms

Salience is a measure of the importance of individual values to the overall output classification of a network.

Salience calculation has been used in the past to demonstrate weaknesses in neural network concepts. In the paper “Universal adversarial perturbations” (Moosavi-Dezfooli et al., 2017), it’s demonstrated how small targetted distortions in an input image can change the final classifications of an image. Figure 3.12 shows a set of images from this paper that are classified incorrectly because of the effect of a targetted distortion.

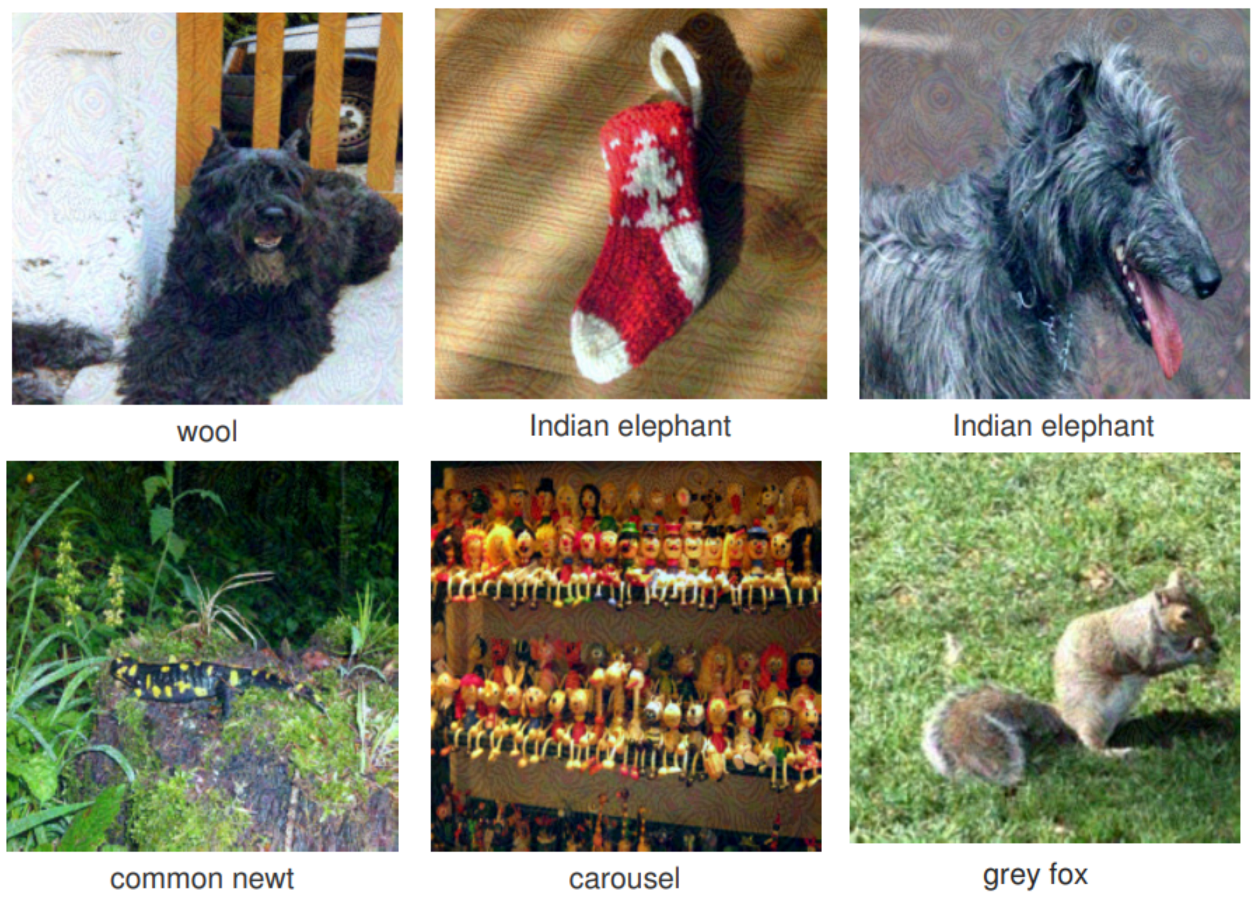


Figure 3.12: Originally classified correctly, these images were slightly distorted and reclassified incorrectly[9]

For our purposes, saliency may be useful to understand which points can be deteriorated harshly to its nearest index-mapping without affecting the final classification significantly, and which points we should try hardest to

[9](Moosavi-Dezfooli et al., 2017)

preserve.

There are a few different measures of saliency commonly used in neural networks:

- **The magnitude of values**: This metric is mostly based around gut-feeling of researchers and was debunked as a viable metric by LeCun in his paper "Optimal Brain Damage"(OBD) (LeCun et al., 1989) . Instead, he proposes another metric:
- **OBD Saliency**: LeCun's proposal calculates saliency by estimating the effect of removing a parameter from the network.
- **Fisher Information** : This metric introduces gradient values to the calculation of saliency (Kirkpatrick et al., 2016). Gradients are calculated at training time and indicate the direction and magnitude of a point's numeric change (i.e. if a parameter is increasing/decreasing as the training process improves its function estimation, and by how much it is moving).

There are a few advanced systems built up on top of the core concepts of Fisher information, such as Fisher Vectors, as introduced by (Chandrasekhar et al., 2015) and Fisher Networks (Simonyan et al., 2013). However, as we are interested in the numerical values, rather than larger network objects, these are left for future research.

The clustering algorithms proposed in Section 3.2.2 were modified to include Fisher information in the *Cluster Assignment* step. The algorithm used to calculate center points was replaced with one that accounts for gradient. The code for this is shown in the below listing and Figure 3.13 shows how this can affect a quantization decision.

Listing 3.4: Centroid calculation using Gradients

```
import numpy as np

# Old Formula
def centroid(data):
    return np.mean(data)
```

```
# New Formula
def centroid_grad(data, grads):

    # Only magnitude of gradient matters, not direction
    g = np.abs(grads)
    return np.average(data, weights=g)
```

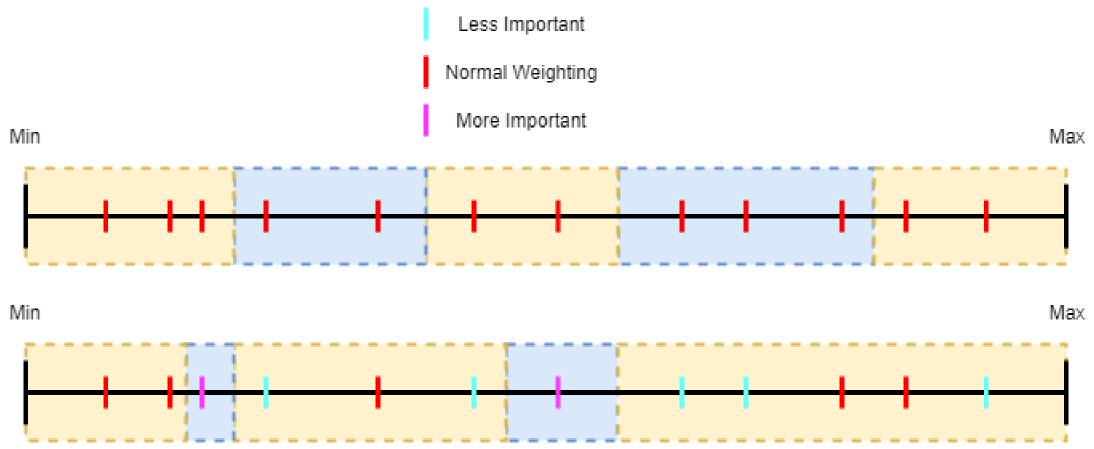


Figure 3.13: Two number lines are shown above. Each line has the same 12 values that are to be quantized to 5 values. In the first figure, the values have no distinguishing weighting and this results in a uniform quantization. When distinguishing weightings are applied, as in the second figure, the resultant quantization will better preserve "important" values while reducing the impact of those that do not matter as much.

3.3 Development Decisions

This section describes choices made for the technical environment in which the research tools were developed.

3.3.1 System Environment

The operating system (OS) for the project was GNU/Linux. It was chosen as it is a common development environment for NN frameworks and a common platform for open source projects, being built upon much open source material itself.[10] In comparison, operating systems like Windows and MacOS tend to be secondary platforms for NN frameworks, releases tend to be packaged for them only once the software is well-established. Additionally, framework releases may be distributed as proprietary executables that cannot be easily modified, whereas compiling software source-code is a common distribution method with GNU/Linux.

As this thesis relies on features that are not widely adopted by all frameworks (like low-precision), we chose this OS to mitigate the risk of missing out on 'alpha'-level features and potentially having to switch OS mid-project.

Choosing GNU/Linux has some practical advantages too, as the author is familiar with the OS and the wider research group at Trinity College Dublin (TCD) have several GNU/Linux-based servers to offload research work onto. Some research experiments involved over a week's worth of processing (even on the latest GPUs) so this was a critical choice to maximize time efficiency.

3.3.2 Technology Choices

NN Frameworks

Initially, Caffe was chosen as the framework to investigate and develop our research concepts upon. The author was familiar with manipulation of the network through its programmable interfaces and it is also one of the more performant frameworks.[11] We can also take advantage of the publicly available network files that many NN researchers have provided to accompany their papers.

TensorFlow was used to gather initial benchmark figures as it was one of the few platforms that supported inference with precision lower than 16-bit. However, the results would not be directly comparable to our own as the smallest quantization level supported was only 8-bit. As can be seen in Figure 3.14 where GoogLeNet is quantized in several different precisions, the loss in classification accuracy between 8-bit and 4-bit is far more significant than between other precision levels.

[10] `https://opensource.com/resources/linux`
[11] `https://caffe.berkeleyvision.org/`

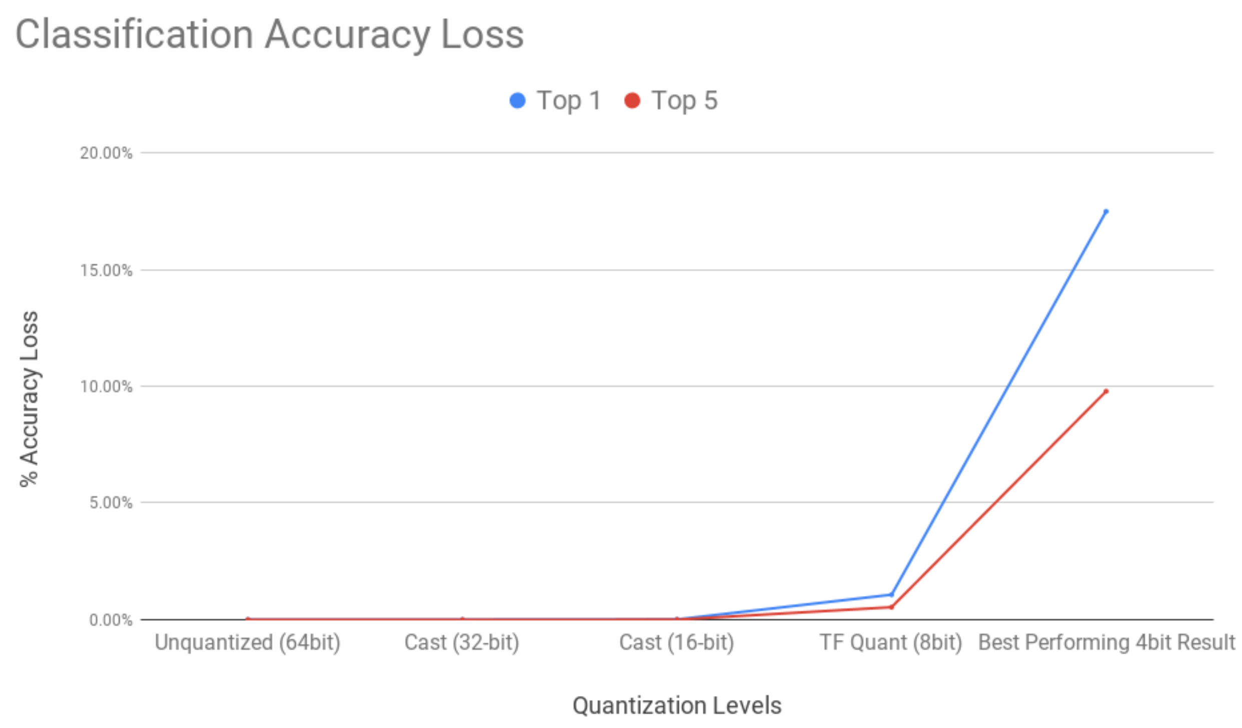


Figure 3.14: Classification accuracy of GoogLeNet when using different precisions. Accuracy rates usually begin to worsen when reaching 8-bit, but 4-bit results are much more degraded

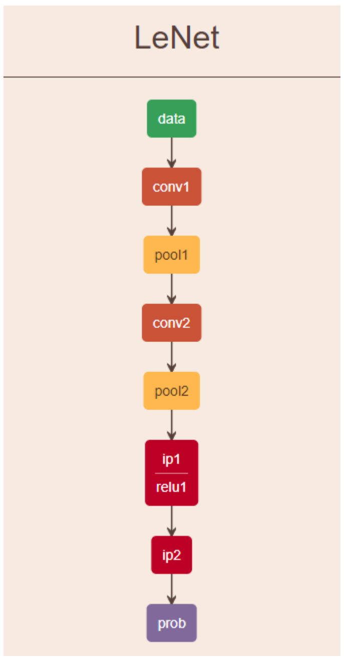
TensorFlow uses the *GEMMLOWP* method (Jacob et al., 2018) for quantizing. An 8-bit number encoding in this method can be resolved to a floating-point number using the following formula:

Where S is a floating-point value to scale the result, ZP is a "zero point" value that represents the quantized value of zero;

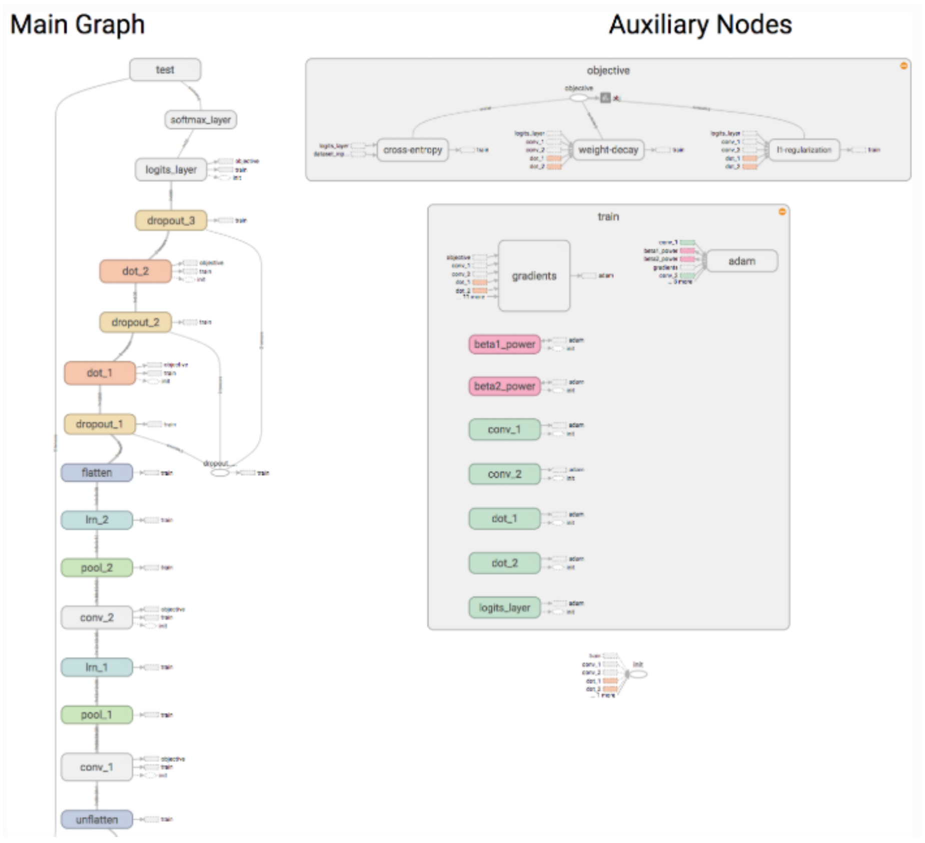
$$Value = S * (QuantizedValue - ZP)$$

Each tensor of a neural network can have its own scale and zero point value. One could make this more granular (e.g. per-channel) or less (e.g. per network) for a better or worse representation.

Other than for reference values, we did not consider TensorFlow as an appropriate platform for development as the scope of the framework is much wider than other alternatives — TensorFlow targets not only NNs, but also general symbolic mathematics, and is more complex as a result.[12]. Figure 3.15 shows a smaller network (LeNet) and how the network is represented in each framework. This disparity in complexity gets larger as the network architecture increases.



(a) Caffe



(b) TensorFlow

Figure 3.15: Differences in granularity of nodes of the LeNet network between Caffe and TensorFlow[13] (9 basic elements versus >25)

[12]At the time of research, neither TensorFlow 2.0 or TensorFlow Lite were available in a stable format

[13]Source: Screenshot from 3rd party Caffe tool `https://ethereon.github.io/netscope/` and TensorFlow diagram from `https://tf-lenet.readthedocs.io/en/latest/tutorial/outputs.html`

Some work was ported to PyTorch and for the latter half of the project, it became the main platform for experimentation. PyTorch's interface allows developers more accessible controls over the internals of a neural network than Caffe. This was necessary for gathering gradient information needed for saliency calculations (See Section 3.2.4). It would also allow for potential integration with other TCD researchers' work using sparsity in the future.

Programming Languages

Python3 was the primary programming language used for development as it is a common interface to NN frameworks and development libraries. It is also a much faster language to prototype in than compiled languages such as C and Java. The language interpreter manages more runtime information (such as pointer management) at the cost of reduced runtime performance.

C and C++ were used sparingly and only when required. This was either for reasons of performance or in the absence of appropriate library bindings or interfaces for Python3.

4 Design & Implementation

This chapter details the testing done on the proposed algorithms. The chapter begins with an outline of the requirements we set out for the research software to ensure comprehensive and fair testing.

This follows with detailed descriptions of the software's architecture and capabilities. Links to this software are available in Appendix A2.

The evaluation period was insightful, but not without limitation. We conclude this chapter with an honest overview of the problems we encountered.

4.1 Requirements

Once we decided on the algorithms that we would substitute into the quantization process, it became apparent that a system was needed to thoroughly assess the validity and potential effectiveness of these algorithms.

Specifically, we needed a suite of tools that could:

- Perform the quantization algorithms on established networks.
- Compare proposed algorithms versus naïve implementations and also each other.
- Save and load network files from experiments.
- Complete processing in a reasonable time frame, given the limited time constraints of the project.

4.2 Dataset & Network selection

4.2.1 Datasets

A diverse set of labelled pictorial datasets were selected for classification testing. These datasets were chosen based on their popular usage and diverse set of subjects.

- MNIST (Modified National Institute of Standards and Technology)
- CIFAR-10 & CIFAR-100 (Canadian Institute for Advanced Research)
- IMAGENET 2012
- SHVN (Street House View Numbers)

Some of these datasets were already divided into subsets when we downloaded them, others we divided ourselves in a 9:1 ratio. We require subsets to ensure that any result we see are not only applicable to the selection of data items we have been focussed on (See "over-fitting" in Section 2.2).

The NN frameworks we used in this thesis had differing requirements of the layout of the labels, format of the data items and the folder structure of the dataset. I wrote a few small scripts to ensure that these datasets would be compatible with each framework.

4.2.2 Networks

Similar to the process for dataset selection, networks were chosen that have been widely used in recent neural network research. We were careful to ensure that the networks chosen were reasonably diverse — The selection of networks have a good mixture of structural layouts and the operations and tensors covered a wide range of sizes. This was important so that we had a representation of the performance of the algorithms that would be likely to apply to many real-world cases.

- LeNet-5 (A very small network compared to recent network designs)
- GoogLeNet (One of the first networks with non-serial operations)

- ResNet-50 (This network introduced reusable layout components called “Residual Units”)
- AlexNet (This network has larger grid sizes for convolutions)
- CIFAR-10 (Another smaller network)

Networks are generally available in one of two forms. Usually they are distributed as a set of several files — there is commonly one file for the structural design of the network, one or more file for the saved trained parameters, and one file to store configuration settings for the training process.

Caffe provides a ‘.prototxt’ file for the network description (a file formatted using Protobuf, an open-source serialization format produced by Google, designed for efficient compression.[1]) It also provides a ‘.caffemodel’, a larger file, containing the trained parameters. Finally, a ’solver.prototxt’ details the parameters used to configure the training process.

This separation allows users to combine different editions of trained parameters without having to recreate the network structure. This gives flexibility when comparing the effects of different training parameters on the same network.

The other form of distribution is through a single file containing all the previously mentioned information. Tensorflow provides such a file with the extension ‘.pb’, which is also based on the Protobuf format.

4.2.3 Pre-Processing

Some pre-processing is needed to import the datasets into the different frameworks, and thus, our own system. This varies between frameworks and also between datasets and network architectures. A suite of scripts to correctly process each combination of preprocessing requirements was created. Some of the functions usually needed for preprocessing include:

- Conversion of a dataset from a set of folders to a database file.

[1]`https://developers.google.com/protocol-buffers/`

- Restructuring dataset items into folders aligning with their corresponding label.
- Calculating some statistical information over each item of a dataset (e.g. standard deviation, mean of channels).
- Resizing and/or cropping images.
- Converting the files to the same file type (e.g. PNG).

These requirements are sometimes declared explicitly in a network configuration file such as the solver.prototxt described earlier. When such a file was available, this was used to inform the operations needed.

4.3 Selection of Research Systems

We decided upon three separate systems for experimentation. Initial experiments were developed on a system that evaluated individual classification inferences (referred to below as “Inference-Only”). However, as will be seen later in Section 5.1.2, these results were found to be inconclusive.

Investigation proceeded to involve the training process in optimizations. This is referred to as the “In-Training System” in the pages that follow.

Separately, a system was created to profile the Intel® Movidius™Myriad™ VPU’s memory transfer speeds.

4.4 Architecture of the Inference-Only System

The Inference-Only system was responsible for all activities that did not involve training. Algorithms were prototyped here before further investigation. A prerequisite for the system was the existence of networks that were trained ahead of time. These were gathered online from various sources (See Section 4.2.2).

The system has three main interfaces for a user to interact with:

- Graphing properties of a given network architecture.

- Quantizing the parameters and activations of a given network architecture.

- Comparing two quantized versions of a network architecture through a set of metrics.

At the core of this system are three code libraries each corresponding with an interface from above; a graphing library (i), a metrics library (ii) and a quantization library (iii).

A UML diagram is given below in Figure 4.1 showing the possible workflows of a developer when using the Inference-Only system.

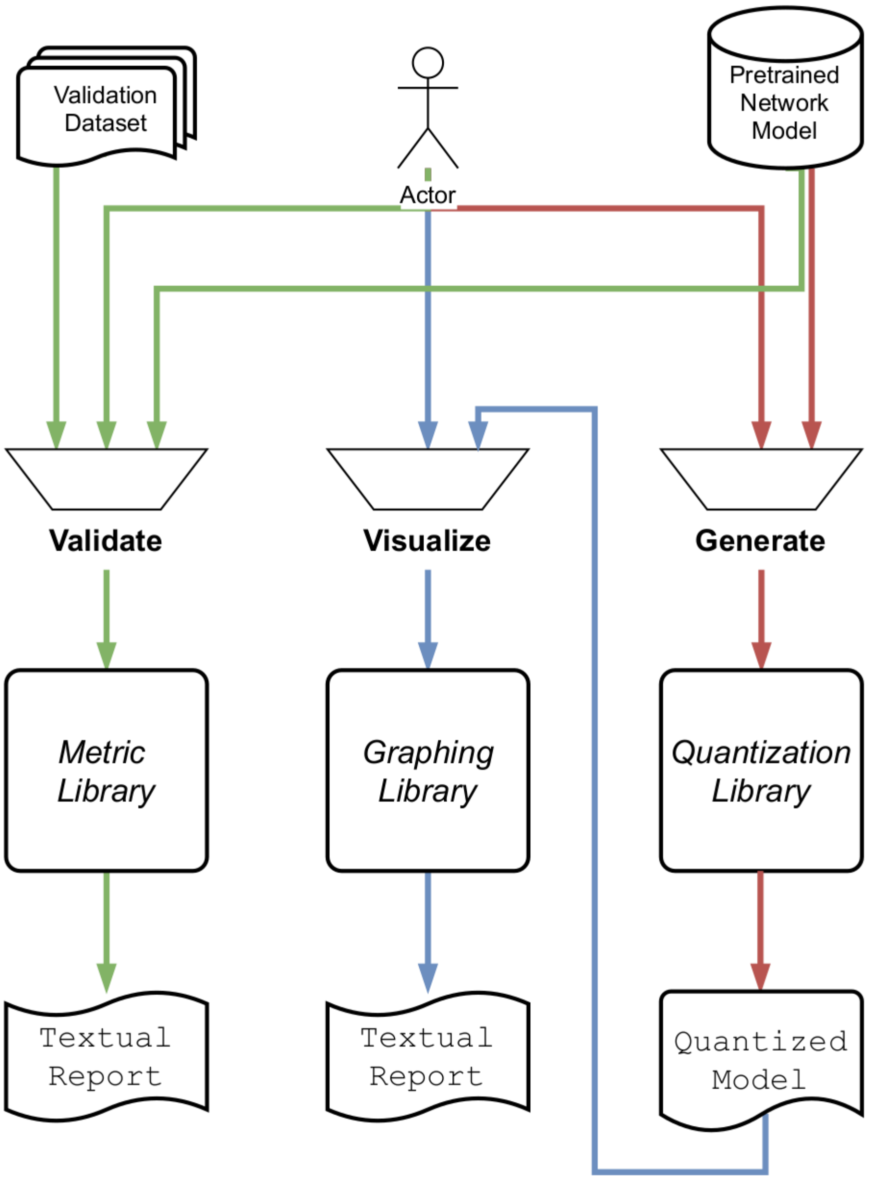


Figure 4.1: Inference-Only System Flow: The Inference-Only software has three main libraries that consume varying input parameters provided by the user ("Actor") from the commandline. The Quantization Library requires an existing model to produce a quantized representation. The Metric Libary requires the existing model and a dataset to test against. Finally, the Graphing Library requires our generated Quantized Model to be able to visualize it.

4.4.1 Graphing Network Architectures

Most of the graphing experiments were created using the *MatPlotLib* plotting library (Hunter, 2007). Using Caffe it was possible to extract both the parameters and activations of a network during an inference. Depending on the dimensionality of the tensor extracted and its origin, each data item will trigger applicable graph functions.

Despite our initial choice of Caffe, the libraries were written to support any framework supporting access to parameters and/or activations.

Whilst several other visualizations were developed for the suite, the following three are the most interesting:

Heat Mapping

It was hoped that by creating heatmaps, we would reveal if there were any localized areas within the weights that were more significant to the computation than other areas. The research done by (Zhou et al., 2016) shows that this locality does exist for activation values. If the locality of data mattered here, it could inform choices on preservation of individual parameters when quantizing.

Unfortunately, after running several variations of datasets and networks through this visualization, no similar results were observed for weights. A sample result is shown in Figure 4.2, alongside an image from Zhou's research showing activation locality.

The lack of locality in weights could be a result of how the values are used within an operation — they are generally processed as a sliding grid over the entire activation set. This would mean these values would represent global information, rather than localized ones.

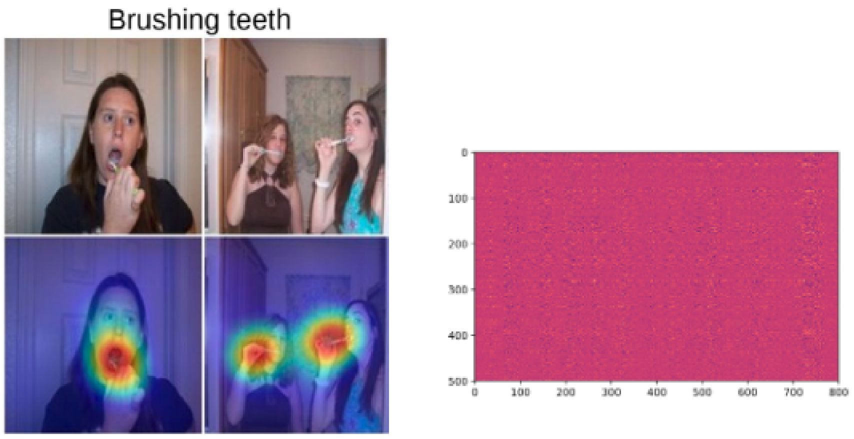


Figure 4.2: Zhou's evidence of locality in activation heatmaps[2] alongside our own heatmap of the parameters of a fully connected layer operation showing no locality

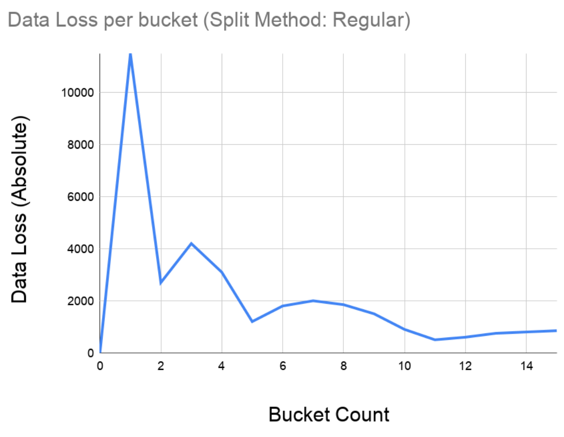
Data Loss

These graphs (examples in Figure 4.3) show how much precision is removed from quantizing a tensor's data. In particular, it shows each of the resultant quantized values' effects on this loss. Theoretically, a good representation of data would have low values for each bucket as it would show a balanced representation.

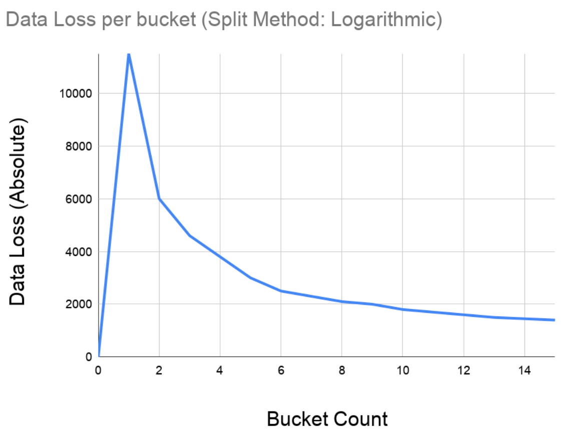
We noticed that throughout the different graphs, the quantization values that represented the most values, tended to have the largest data loss, despite having the smallest magnitudes. If preserving the total precision of the data was important (rather than individual value's precision), the low values would have to be given more importance.

This led us to investigate into algorithms that have equally distributed quantizing (e.g. Equivalent Population Split in Section 3.2) as we hoped to reduce the large dataloss seen in the smaller values.

[2]Source: (Zhou et al., 2016)



(a) Split Method: Regular



(b) Split Method: Logarithmic

Figure 4.3: Sample Data Loss graphs for two different algorithms on the same data (GoogLeNet's first convolution's parameters). The 16 values along the horizontal axis represent each quantization value, and the vertical shows the value of the total precision lost in each of these

Outlier Range

As we are interested in investigating the magnitude of values for the saliency they may represent (See Saliency Algorithms in Section 3.2), another graph was made with the purpose of discovering any patterns in outlying high-magnitude data points within the network parameters.

This diagram (like the one shown in Figure 4.4) showed us that many network architectures had larger positive outliers than negative. This was consistently shown on network operations that were located after ReLU operations. This may be an indicator that positive values are more contributory to the network and deserve better preservation.

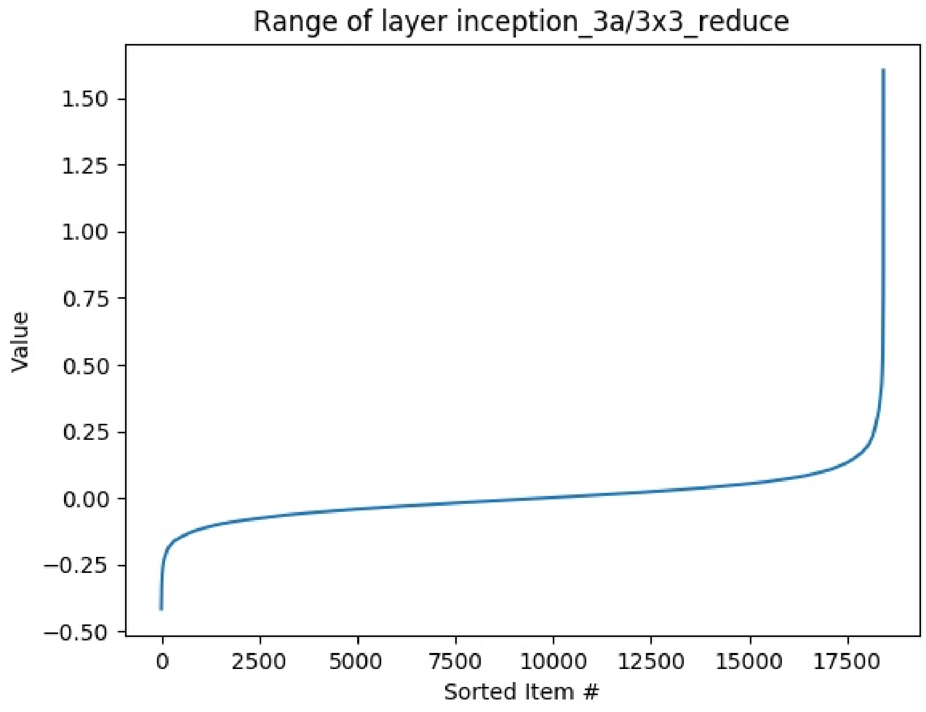


Figure 4.4: The weights for the inception_3a\3x3 layer were sorted in ascending order and the values plotted. The majority of items are between +\-0.25, but the lowest and highest items have larger magnitudes. A discrepancy can be seen between the magnitude of the highest value 1.5 and the lowest value 0.4

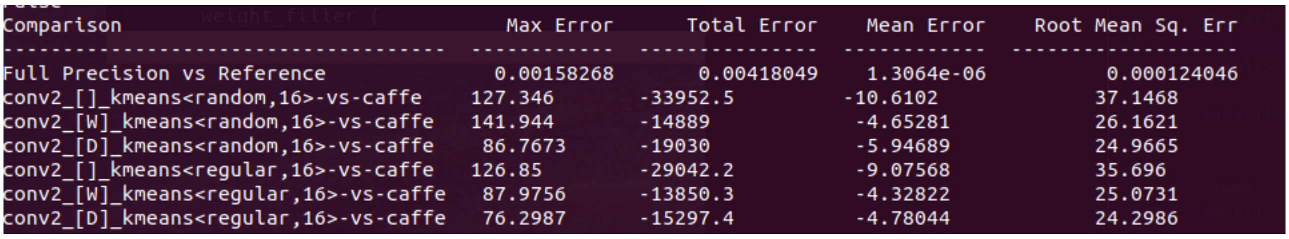
4.4.2 Metric Comparison of Network Quantizations

After individually profiling the proposed algorithms, a program was built to directly compare the algorithms to one another.

Algorithms were compared over two categories. First, numeric accuracy compares the algorithms from a neutral statistical perspective. Secondly, classification accuracy compares the effects of their quantizations on final predictions of the target networks.

Numeric Accuracy

To calculate the numeric accuracy of an algorithm, the comparison library re-implements some established statistical functions described below. A sample output of the library is shown in Figure 4.5.



```
Comparison                            Max Error      Total Error    Mean Error    Root Mean Sq. Err
-----------------------------------  -----------   --------------  -----------  ------------------
Full Precision vs Reference            0.00158268      0.00418049    1.3064e-06          0.000124046
conv2_[]_kmeans<random,16>-vs-caffe   127.346      -33952.5       -10.6102             37.1468
conv2_[W]_kmeans<random,16>-vs-caffe  141.944      -14889          -4.65281            26.1621
conv2_[D]_kmeans<random,16>-vs-caffe   86.7673     -19030          -5.94689            24.9665
conv2_[]_kmeans<regular,16>-vs-caffe  126.85       -29042.2        -9.07568            35.696
conv2_[W]_kmeans<regular,16>-vs-caffe  87.9756     -13850.3        -4.32822            25.0731
conv2_[D]_kmeans<regular,16>-vs-caffe  76.2987     -15297.4        -4.78044            24.2986
```

Figure 4.5: Sample metrics from the second convolution of LeNet.[3]

∗ Sum of Absolute Errors (SAE)

This metric shows the magnitude of the difference between two matrices. However, this magnitude is not correlated back to the original data — e.g. The SAE for a ruler of length 1 metre being off by a centimetre is no different to the SAE of a car's pedometer running over 1000 kilometres being off by a centimetre. Because of this lack of context, it is quite a limited metric, but is quite simple to calculate.

$$SAE = \sum_{i=1}^{n} |V_i - \overline{V}_i|$$

Where:

- V, $\overline{V}$ = the matrices for comparison
- n = the number of values in each matrix

[3]Explanation of Figure: "Full Precision vs Reference" compares the output of the network in FP64 to when it is run in FP16. Subsequent items compare themselves with the same FP16 result. Notation: Quantization applied to just weights: ([W]), just activations:([D]) or both ([ ]). The titles of each column are misnamed with the formal names of the metrics. Total Error is equivalent to SAE, Mean Error is equivalent to MAE

* ***Sum of Relative Errors (SRE)***

SRE fixes SAE's problem with context by scaling the errors by their original values.

$$SRE = \sum_{i=1}^{n} |\frac{V_i - \overline{V}_i}{V_i}|$$

* ***Mean Squared Error (MSE) and variants***

MSE and variations of the algorithm are used in other applications of machine learning[4], it is often chosen to distinguish significant errors from smaller changes. Due to the stochastic nature of training and an abundance of parameters, researchers can generate equally performant results that are non-identical, where small variations are inconsqeuential to the researcher's dataset. Similarly, our quantization results should preserve pertinent information best.

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (V_i - \overline{V}_i)^2$$

A variant, Root-Mean-Square Error (RMSE), eases the scaling of the MSE so that small errors are still minimal, but larger errors are closer in range (e.g. If there were MSE reported errors of 4, 100 and 400, they would be scaled down to 1, 10 and 20) .

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (V_i - \overline{V}_i)^2}$$

* ***Mean Average Error (MAE)***

This metric is sometimes suggested as a more balanced alternative to RMSE [5]. It uses the previously mentioned MSE in its calculation, but removes the squaring from the calculation.

[4]`https://www.freecodecamp.org/news/machine-learning-mean-squared-error-regression-line-c7dde9a26b93/`

[5]`https://medium.com/human-in-a-machine-world/mae-and-rmse-which-metric-is-better-e60ac3bde13d`

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|V_i - \overline{V}_i|$$

Classification Accuracy

Although numeric accuracy can give us a good indication on how well the quantized values represent the original values, the crucial metric for neural networks is the preservation of the classification rate.

$*$ ***Top 1 & Top N Accuracy***
"Top-1" Accuracy is a metric that measures how many classifications from a dataset were detected correctly by the system. Some datasets can be more vague in composition (e.g. the ImageNet database contains an image that has both a human and a fish in it, but labels the image as a fish), so a looser metric commonly used is "Top-5" accuracy, where a correct detection is marked if it is in one of the 5 highest results. This concept is generalized to "Top-N" Accuracy.

$*$ ***Complete Class Comparison***
This metric compares all the resultant class values between two implementations of a network. Though the metric was part of our library, we decided to disable it. Our quantization algorithms prioritize preserving the top class predictions; this metric evaluates how well all levels of prediction are preserved. We expect that class predictions with low detection are likely unreliable in our systems as they are probably where the accuracy reduction is most affecting.

*Let **M** be the reference results of a network inference with **n** classes ordered in strength of classification match, and let $\overline{M}$ be the corresponding alternate results of the same network. Then the Complete Class Comparison (CCC)*[6] *value is:*

$$CCC = \frac{\sum_{i=1}^{n} \overline{M}_i = M_i}{n}$$

[6]This is a non-standard metric, introduced in this paper

4.4.3 Quantization Functions

This is a collection of functions relating to the algorithms explained in Section 3.2

Additional control quantization schemes were added that used Python's built-in type-casting. However, as the natively supported types of Python are much larger than 4-bit, these performed significantly better than our own algorithms, but they did give a more reasonable reference precision than the default precision (see 3.14 for an illustration of this issue).

The lowest applicable casting precision (16-bit floating-point(FP16)) could be up to 4x closer to the data range we are operating in than some networks provided by Caffe (up to FP64). Additionally, the Intel® Movidius™Myriad™ VPU index-mapping hardware described in Section 3.1.3 only maps to FP16 values — Even with a perfect representation, there would be natural precision loss from the higher precision network.

4.5 Architecture of the In-Training System

As results from the Inference-Only system were inconclusive (See Section 5.1.2), research proceeded to investigate if the algorithms would provide more consistent results if integrated with the training process; in the Inference-Only system the network was trained without accounting for a possible quantization step. Constraining the training process with the quantization values could naturally gravitate values towards their future quantizations.

Designing a system involving training opened up the possibility of including additional algorithms that used salicency due to the now present values for each parameter's gradient. See the details of these algorithms in Section 3.2.4.

An option was added to the program to allow incremental training (Istrate et al., 2017). This technique trains a network in an operation-by-operation progression. Originally introduced to speed up training systems, it gives an added advantage to the researcher to investigate any mid-training issues or accuracy drop-offs during the training process. In a commercial system, these same observations could be used to selectively quantize at different precision rates to optimally convert a network.

As the training process is quite compute intense, some development effort was put into improving the performance of the more complex quantization algorithms. As they were originally written for a scripting language and favoured prototyping over performance, some algorithms were prohibitively slow before refactoring.

One significant speed-up was observed from the adoption of "Mini-batch k-means"[7]. Overall, most algorithms were improved to the point of being able to complete in a reasonable time for proper analysis. However, some more complex clustering algorithms relying on non-standard k-means (e.g. the forced-zero algorithms in Section 3.2.3) remained infeasibly slow.

4.5.1 Technical Overview

A rough technical design of the In-Training system is shown in the sequence diagram, Figure 4.6. It can be read left to right, top to bottom.

1. First, any required files for trained network architectures are retrieved — this can be from PyTorch's own catalogue, or one we stored locally.
2. The software provides the first quantized representation of the network, limiting the parameters to 16 values each.
3. The training process begins on the last operation of the network — each image is classified using the quantized network and the results fed back into the framework and the networks's parameters adjusted. This will cause the network to diverge from the quantized state.
4. After a set number of epochs have passed, the network is re-quantized before testing the new classification rates.
5. Steps 3 and 4 repeat until all the operations (or until a set number of

[7]`https://scikit-learn.org/stable/modules/clustering.html`

operations) have been through the training process. A log file is created for import into external graphing software.

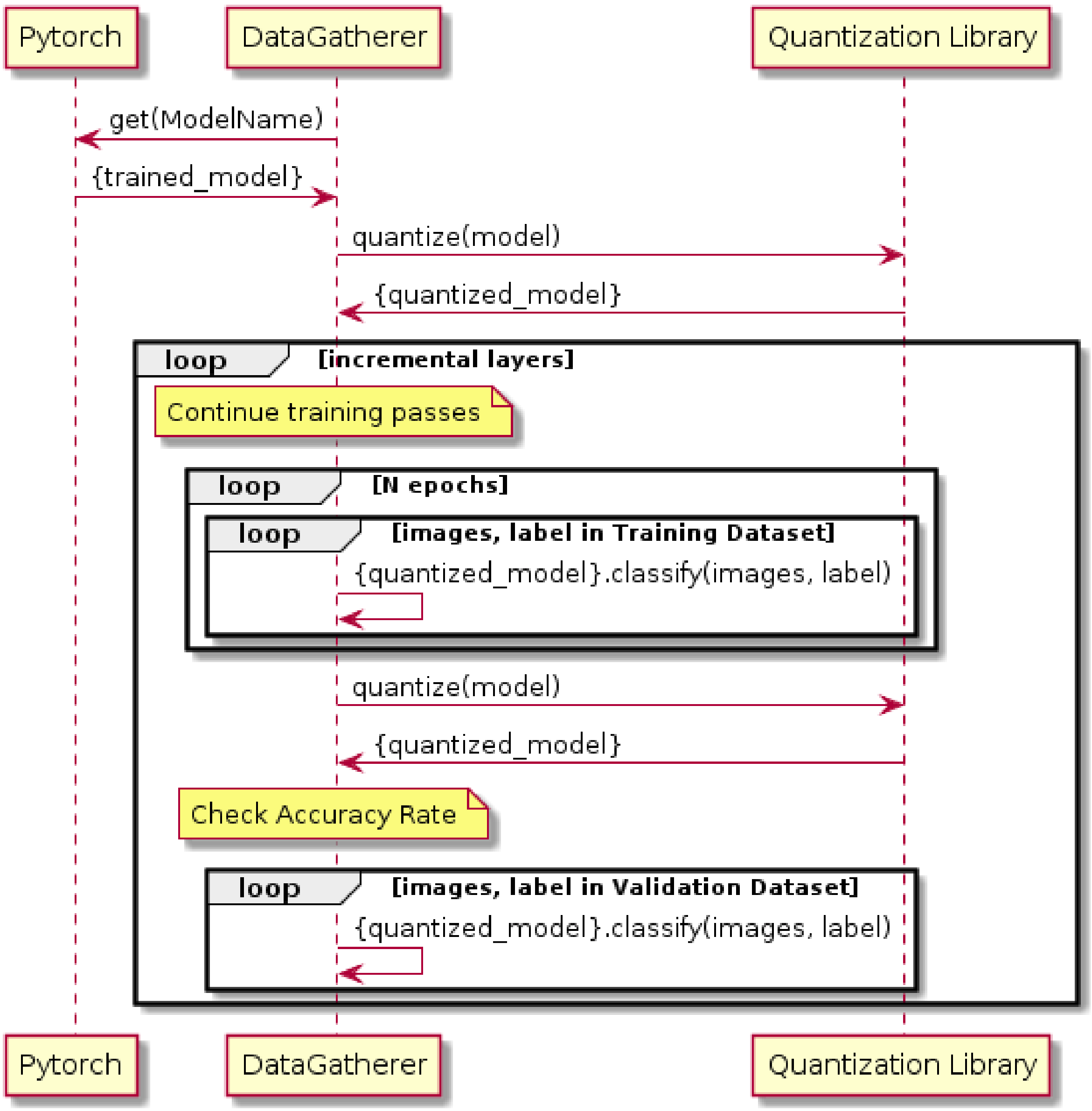


Figure 4.6: Software flow of In-Training validation system

4.6 Profiling the Processor's Memory Speeds

Working with Intel® Movidius™ hardware and software requires an intimate knowledge of both the chip and the infrastructure around it. Even with my past working experience, I know that non-trivial systems take a significant amount of effort and time.

Therefore, rather than process through an entire network on the device which would be functionally equivalent to the PC code and significantly slower, focus was placed on profiling the hardware's performance gain when utilizing the index-mapping feature for a range of data sizes. A selection of sizes were chosen to extrapolate the potential scaling of the speeds of the hardware when applied to full networks.

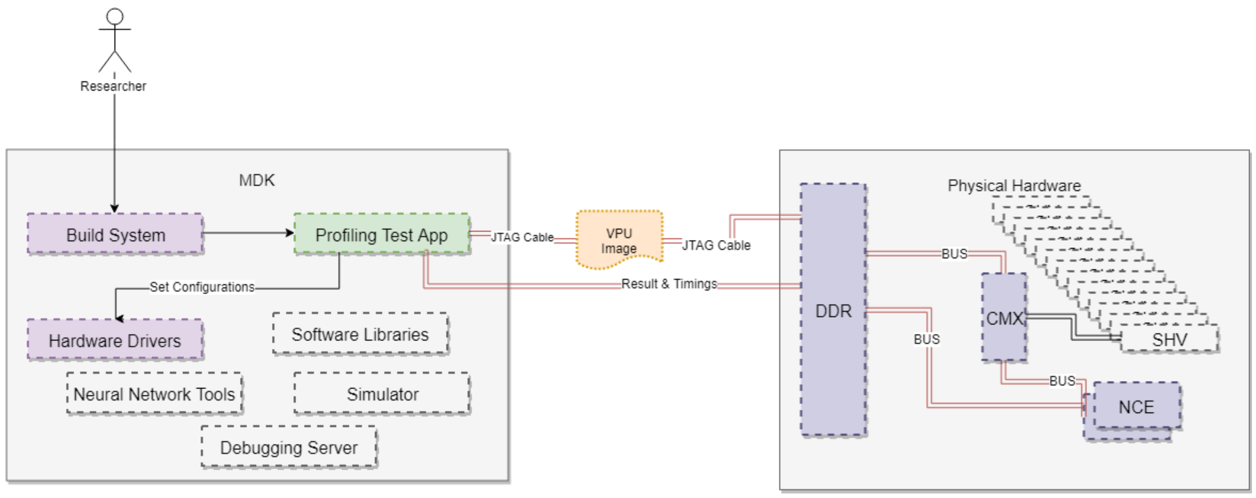


Figure 4.7: Running the NCE Profiling Software

Using the software library functions provided by the MDK, the register settings for the hardware were configured for both FP16 and index-mapping modes. The MDK build system compiled the relevant source files into a binary which was flashed onto the hardware device.

Flashing the binary and transferring data to and from the physical hardware (See Figure 4.8) was done through a JTAG cable connection. However, in products such as the Intel® Movidius™Neural Compute Stick, USB and other protocols could be used to achieve a much faster throughput. For our research, the transfer speed was not profiled, though we would expect performance improvements here also due to the smaller transfer size.

Figure 4.7 gives a summary view of the experiment setup. A larger view is available also in Appendix A3. The MDK takes care of all the infrastructural setup, allowing the researcher to write a small test application without much overhead. A single SHV processor was used to control and time the NCE hardware through a provided set of drivers.

The test application ran convolutions of various given sizes and the system returned both timings and results for verification.

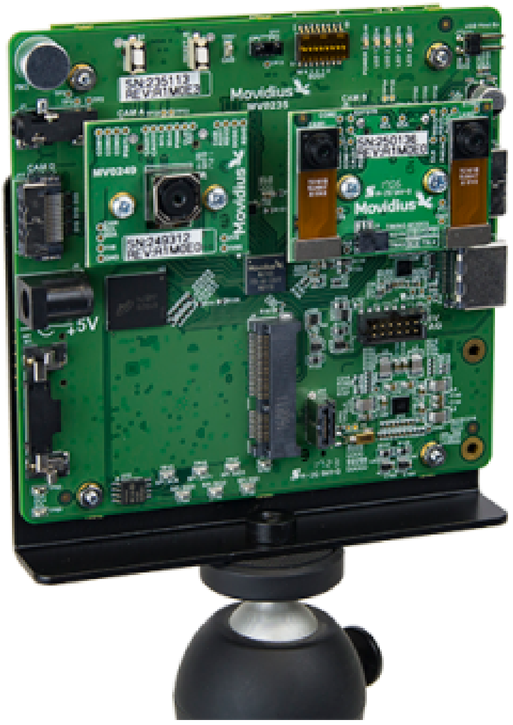


Figure 4.8: Intel® Movidius™Myriad™ VPU Hardware Development Board[8]

[8]www.movidius.com/solutions/software-development-kit

4.7 Problems affecting design

For the most part, the system we created was stable and fit for our research purposes. However, there were some limitations that prevented us from achieving more in our research.

4.7.1 Performance Limitations

The biggest issue faced was the significant time required to test the latter stages of the project. For training, many networks require huge amounts of data to process and rely upon individual inferences being very fast. Due to the nature of some algorithms, even when they were optimized the training process could take over a week to process. Even the simplest algorithms took several hours to complete processing. Particularly infuriating was several fatal crashes which occurred several days into some runs without much crash information.

This was mitigated somewhat by taking advantage of the servers provided by the wider research group, and also by the authour's part-time schedule. Some effort was put into optimizing the algorithms for performance throughput, but soon the time spent was returning diminishing improvements.

This caused both significant delay to verifying feature completion and also resulted in limited analysis data being gathered.

Due to the limited duration of this project, it was not reasonable to expend much more effort in alleviating this limitation at the expense of reducing the potential scope of the project. If this were a longer-term project, each algorithm would likely have had to been ported to a lower level language like C, rather than the high level prototyping language of Python that was used extensively throughout.

4.7.2 Intense Variability & Large Amounts of Data

Part of the performance limitations we encountered was due to the vast size of neural networks, both in terms of the number of individual data points and the amount of configuration parameters, it was difficult to concretely attribute gains or losses in our results to changes we introduced, rather than their side effects.

For example, when comparing two algorithms and observing that one was more accurate, it was hard to ascertain whether this improvement was due to specific changes we had made, the stochastic internals of training,[9] the effect of local minima[10] or some other cause, e.g. Would the other algorithm have been a better point of comparison had we allowed more time for it to reach a solution?

This was mitigated somewhat by ensuring that tests were not analysed in isolation, diversifying the types of networks, datasets and comparison metrics would allow a more objective comparison.

4.7.3 Research Limitations

Because of the long turnaround times, we were able to test fewer algorithms through every stage of the process than desired. This limited the coverage of test cases that the system could evaluate and the volume of metrics we could gather.

As a result, the analysis process took significantly more time than expected, limiting the amount of subsequent research that could be built upon prior conclusions.

In the early stages of the project, this was possible to be mitigated by multitasking on different aspects of our research, but was unavoidable nearing the end of the project when development began to cease.

[9]Stochastic Gradient Descent (Robbins and Monro, 1951) is a commonly used training algorithm. It introduces randomness when it chooses dataset images to process.

[10](Swirszcz et al., 2016) proves several cases of "significantly suboptimal" training results with NN training routines.

5 Analysis & Evaluation

5.1 Statistical Analysis

In this Chapter, we review results gathered from the software systems in Chapter 4. We examine the data in both numerical and classification contexts and attempt to draw some conclusions from them. We also evaluate the research itself and our approach. The systems we put in place for testing and how well we were able to investigate the research subject.

5.1.1 Performance

The Intel® Movidius™Myriad™ VPU's memory transfer speeds were profiled with increasing volumes of data.

The speed of transferring the data of an unmodified 16-bit convolution was compared to the speed of the same convolution when the index-mapped 4-bit quantization was applied. A mix of convolutions from existing architectures and some artificial cases were used to create a diverse range of volumes that would provide some insight into how the transfer speeds would scale.

Because the volume of data to transfer is 4× smaller, it was expected that the transfer speeds would be 4× faster. This speculation proved to be reasonable as the results at each scale were indeed roughly improving by 400%. Figure 5.1 shows our results, displaying a highly consistent scaling rate.

Intel® Movidius™ declined to allow specific figures or publication of other memory transfer systems, so the standard implementation of C++'s 'memcpy' on the device was profiling instead and exact units of measurement have been redacted from the thesis.

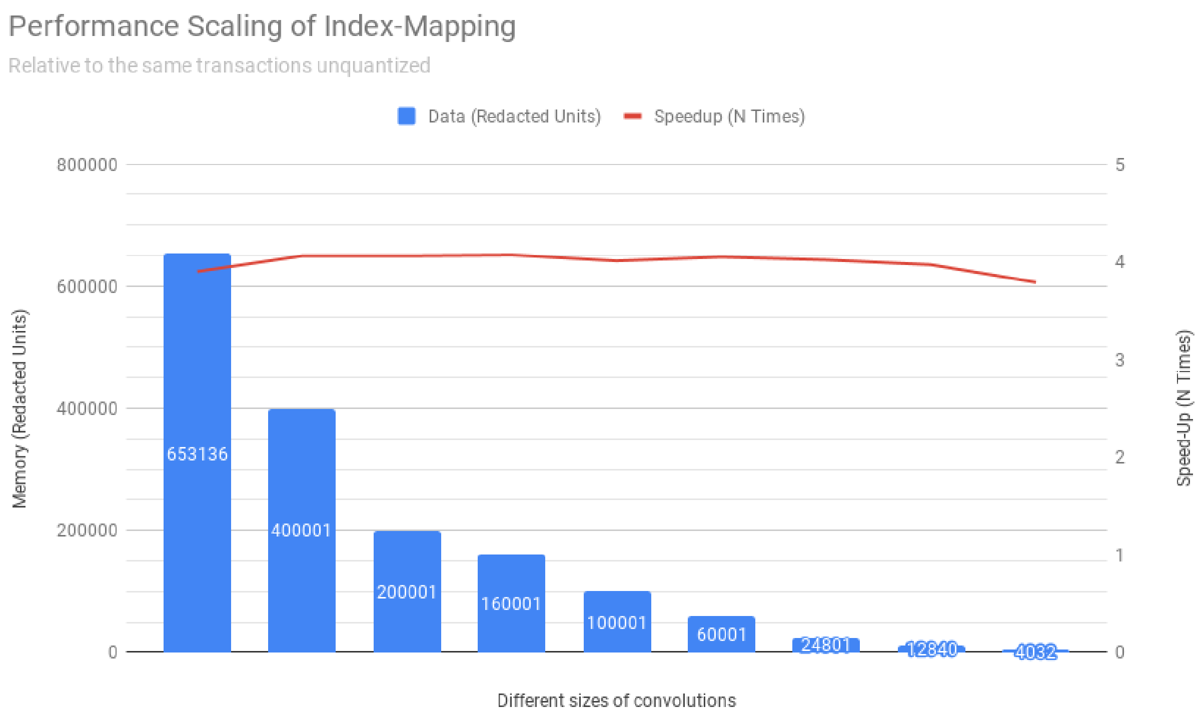


Figure 5.1: Speed of transfers on different data sizes

5.1.2 Accuracy Loss (Inference-Only System)

As described in Section 4.4.2, the measurement used to assess the classification rate of the networks with our algorithms was *Top-N accuracy*. The diagram below (Figure 5.2) shows the Top-5 metrics of each of our algorithms on the LeNet architecture. However, results are inconsistent between networks; For example, the "Equal Split" and "Equal Split (Forced Zero)" algorithms have a similar level of successful classification in Figure 5.2, but when applied to AlexNet in 5.3, the algorithm without forcing a quantization value to zero, has a markedly worse classification rate than its partner.

The differences in each of the algorithms' results are often too close to distinguish one as an obvious preference. The x-axis in both figures are based on a relative scale, rather than an absolute one. This is because the differences in classification rate are significantly smaller than the overall classification rates (e.g. Comparing 97.34% and 97.32), making it hard to distinguish visually.

In 5.3, the larger precisions (float 32, float 16) have a much greater classification rate than the four bit equivalents. This is a problem with most larger networks examined, resulting in some networks being unsuitable for direct comparison. In the case above, this is mitigated somewhat by quantizing a single layer rather than the full network.

When the same experiments were done on a wider set of network architectures (ConvNet, ResNet18, Inception-V3) and datasets (CIFAR10, CIFAR100), it was observed that the relative performance of the algorithms did not have a clear pattern, nor a consistent leader.

The differences in classification rates are small enough to possibly be partially attributed to the result of noise throughout the system (as mentioned in Section 4.7.2), resulting in either negatively or positively circumstantial improvements.

It can be concluded that in order to best use this research, the algorithms may need to be profiled individually for the particular target scenario and then select the algorithm with the best results.

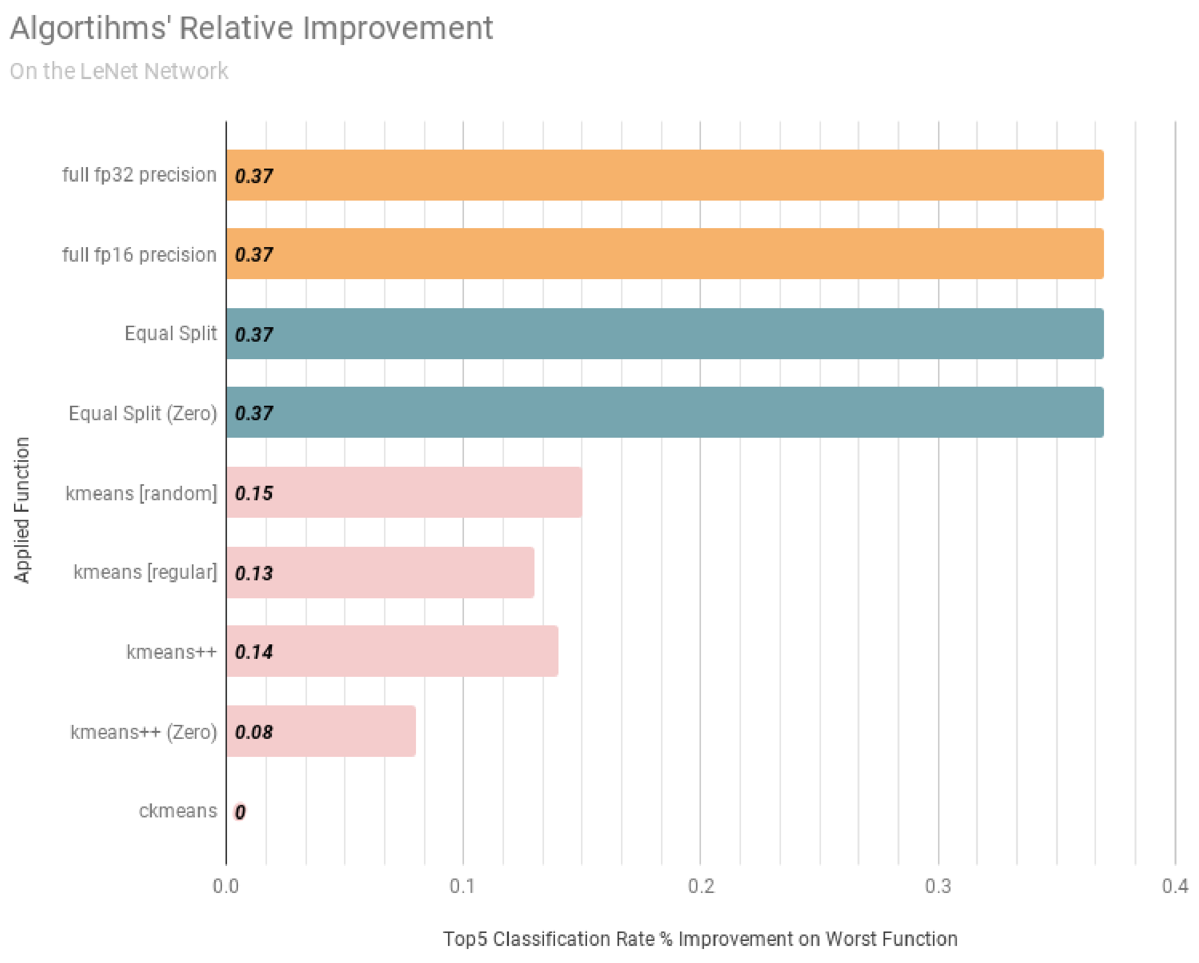


Figure 5.2: Classification rates of various quantization algorithms on LeNet & the MNIST dataset

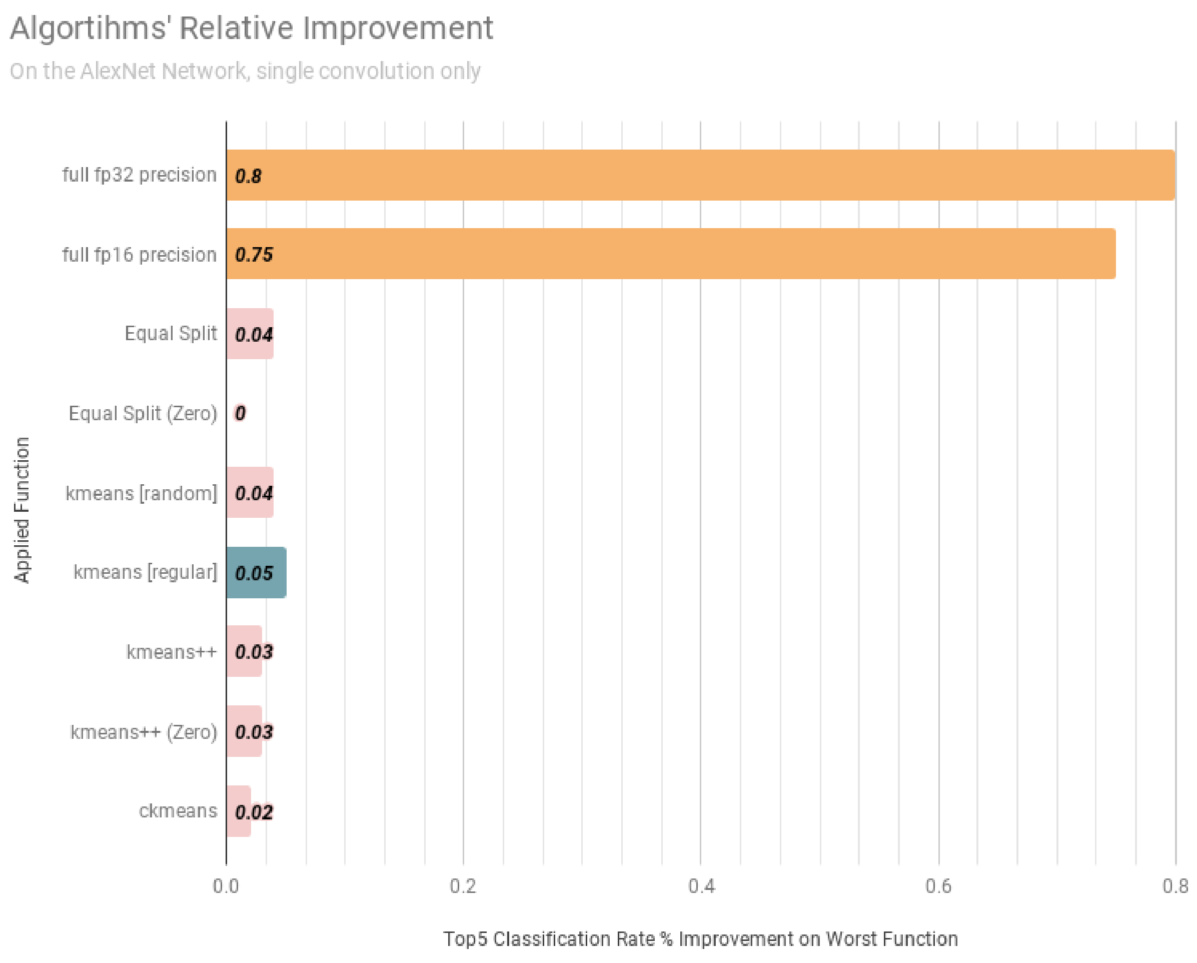


Figure 5.3: A subset of classification Rates of various quantization algorithms on AlexNet & the ImageNet database.

Another definite conclusion that can be made, is that there is still scope for improvement. On rare occasions, when a "lucky" random seed is generated, the *"K-means [Random]"* algorithm, and even a totally random splitting, can occasionally out-perform the others. This means that there are still more optimal quantization splits existing that the other algorithms do not cover.

5.1.3 Accuracy Loss (In-Training System)

We repeat the metrics used in Section 5.1.2 for the In-Training system, however, the range of algorithms we were able to test was further reduced due to increased processing constraints (See Section 4.7.1).

To observe the effects of incremental training on the classification results, we performed several epochs (A pass over each item of a dataset, in this example we use 10 passes) of training with an unquantized model. The next 10 epochs built upon the trained data but has the final layer quantized to four bit. The following 10 epochs also quantize the penultimate layer, and so on.

In Figure 5.4, we show both the Top-1 and Top-5 classification rates on the AlexNet network after each epoch has completed. The figure shows the unquantized training epochs and 4 subsequent incremental quantization epochs. Two trendlines are included in purple to show an average of progress, as the data appears quite sporadic.

There is a big drop in classification rates after the first operation (the fully connected layer) is quantized. However, as the additional operations are further quantized the classification does not noticeably deteriorate.

One explanation for this is that the fully connected layer holds a lot of the final classification information for a network. By reducing the data range, it will limit the granularity of classifications.

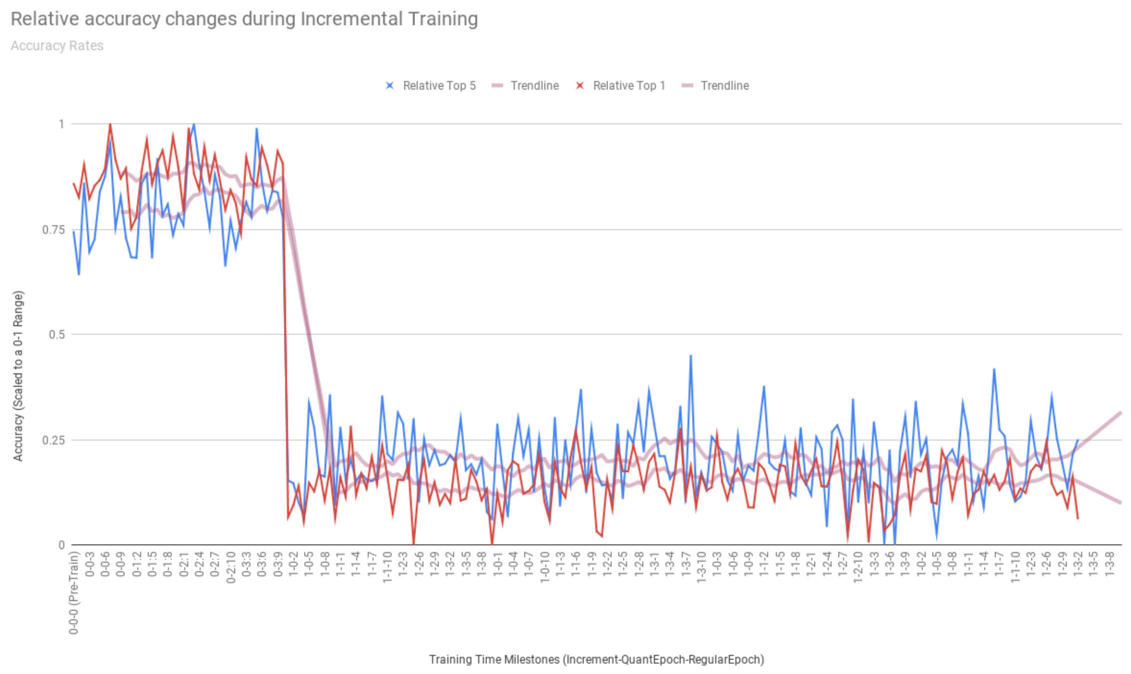


Figure 5.4: Top-1 and Top-5 classification rates as incremental training progresses. Purple trend-lines are included to assist reading. Other than the drop after the first increment, no further trends are observed, positive nor negative

The classification rates fluctuate quite a lot with each epoch and it is difficult to tell whether the system is recovering from the degradation of quantization or not. When tried with a larger number (100, 1000), no upward trend was seen.

When the entire network was quantized to four bit, the classification results were so degraded that they were no longer useful for comparison (<10% classification rate), so we might suppose a straight-forward quantization may not work on a large network.

5.1.4 Numeric Analysis

Despite the fact that the ck-means (See Section 3.2.2) is a numerically optimal clustering solution for a single-dimension dataset, it can be seen in the classification analysis above (Figure 5.4) that it is not consistently the best algorithm for quantizing parameters. This lends significant credence to the suggestion that preserving numeric precision optimally does not imply that classification precision will also be optimally preserved.

With that in mind, the metric calculation for numeric accuracy does not inform any definitive conclusions on the algorithms' effectiveness. However, this suite of tools could be transferred to other domains where numeric accuracy is used for validation.

5.2 Software Feature Evaluation

While there are several shortcomings in terms of performance and conclusiveness, the author believes the software design was well architected for the research investigation. It follows many of the aspects of good software principles[1][2] in common circulation — it is extendible by being modular, free from side effects (contract-based), flexible for future change and well formed for unit testing.

The software relied on some publicly available libraries, inheriting their own limitations, but also relieving the author of a large volume of work. The libraries were built in a different context to the author's work and thus there were differing levels of adaptability in each library. As a result, sometimes several libraries were used for different parts of the system though those libraries' functionality overlapped — e.g. There were several versions of k-means that were used throughout the project as some were easier to modify custom features while others were more performant. In an ideal world, there would have been a singular library which we could have availed of.

5.3 Overall Evaluation

While there may be no universal optimal solution for choosing values to map to, the work here confirms that there are better solutions that those that currently

[1]`https://www.d.umn.edu/~gshute/softeng/principles.html`

[2]`http://ecomputernotes.com/software-engineering/principles-of-software-design-and-concepts`

exist. There is definite scope for future research to further bridge this gap and achieve better results.

For a user who would like to map their product to an embedded device with index-mapping hardware such as the Intel® Movidius™Myriad™ VPU, it is worthwhile to profile several of these proposed quantisation algorithms for their target network architecture. They can gain both significant speed-ups and reduced energy consumption by using such low precision, while also reducing the effect of the degraded classification accuracy.

6 Conclusion

In this chapter we advise work for future research, present the summary of our contributions to the research space and end this thesis with some closing remarks.

6.1 Thesis Contribution

This thesis set out to investigate alternative algorithms for configuring hardware systems (in particular, the Intel® Movidius™Myriad™ VPU) that use index-mapping when quantizing to lower precisions in the particular domain of neural networks.

We show that simple algorithms that may apply effectively to general collections of data have room to be improved upon for this specific domain. While the thesis does not find a "one size fits all" algorithm, we present a suite of algorithms for developer usage that conditionally improve classification results.

We observe the merit of such a hardware component in measured performance gains and publish the software used to enable future research.

We propose several new approaches to configuring this component and how they could be used to improve classification rates for very low precision networks. Of particular note, is the LeNet network where our 4 bit results match those of a 32 bit equivalent. However, we find that our proposed algorithms are suitable for different scenarios and would be best used as a suite.

Finally, we demonstrate the performance of the VPU using the hardware component to achieve 4 times lower data transfer sizes and consequentially, 4x faster processing of a layer.

6.2 Further Research

This thesis details the author's research into a niche area of low-precision neural networks. Despite this thesis's well-defined scope, there have been several points throughout the previous chapters highlighting potential future research work that can continue on from our findings. In this section, these proposals are elaborated upon, as well as the addition of several other relevant research avenues that expand on the work done in this thesis.

6.2.1 Power Measurements

It has been proposed several times throughout the thesis (e.g. Section 2.4.2) that the performance improvements observed would also imply energy efficiency improvements.

No data gathering was done in this thesis to back up this claim, but this activity would further show the utility of the index-mapping technique for embedded devices.

6.2.2 Integration into an adaptive quantization system

As mentioned in Section 5.3, the techniques presented in this thesis do not provide a universal solution to all combinations of network architectures and datasets. In real-world systems, products may have strict accuracy rate targets to uphold whether directly or indirectly through internal and regulatory requirements.[1] If the techniques in this thesis are applied directly, networks may fall below allowed thresholds. However, there are often commercial requirements for performance[2] that the techniques may help with.

[1]e.g. US Federal Automated Policy (Sept 19) has assessments of validation and robustness of vehicle mechanisms

[2]Operating on a video in "real-time" on each frame could require 30 executions a second.

In order to use our algorithms appropiately, a system such as “Risetto” (as introduced by (Gysel, 2016)) that can identify which level of quantization is appropriate at each operation of a network, could balance the usage of the index-mapping quantization with other quantization techniques, without compromising harshly on classification accuracy.

While this thesis focused on the Intel® Movidius™Myriad™ VPU’s 4-bit indexing values, developers could use such a technique to utilize the even smaller mappings of 3, 2 and single bit index-mappings, other limited-precision hardware or with slightly higher precisions like the 8-bit GEMMLOWP quantization used in TensorFlow (see Section2.3.2).

6.2.3 Interaction with other optimizations

Sparsity

Quantization hardware can mesh well with unconnected sparsity hardware. Experimentation in this area (as described in Section 5.1.2) showed that forcing one of the quantized values to zero usually only affected the network classification results by a fractional percent, and occasionally improved the classification results.

With suitable hardware and/or software, a sparse system can skip computation when one or more inputs to an operation is zero. In a system that has a separate control for sparsity and index-mapping, the numeric range to quantize into could be increased from 16 to 17 values as seen in Table 6.1. Alternatively, the hardware could be designed to use both concepts and sparsity could be represented by a zero entry in the map, removing the need for an additional boolean in the system and also skipping evaluation of any numbers in that zero map.

4-bit index	Sparse?	16-bit value
0000	No	-3.0
0001	No	-2.5
0010	No	-2.0
0011	No	-1.8
0100	No	-1.5
0101	No	-1
0110	No	0.3
0111	No	0.8
1000	No	1
1001	No	1.2
1010	No	1.4
1011	No	1.8
1100	No	2.4
1101	No	2.6
1110	No	2.9
1111	No	3.5
any	Yes	0

Table 6.1: Extension of Figure 2.1's Index Mapping with separate sparsity control

Several of the algorithms and metrics presented in this thesis can be re-used if a sparse hardware component was available. Any of the "Forced-Zero" algorithms are designed to accommodate arbitrary fixed values.

6.2.4 Activation Quantization

Apart from our initial investigations, this thesis mainly focuses on parameter quantization — those values that can be known ahead of time in a network. There is a similar space to be researched in the activations of a network.

Index-Mapping of Results

Assuming the parameters of an operation were quantized to 16 values as has been the usual configuration throughout this thesis and the associated activations existed within a known range, the activations could be speculatively quantized to their own 16 values.

The 256 possible resultant values from the operation could also be pre-calculated and stored in a new 256-bit index-mapping.

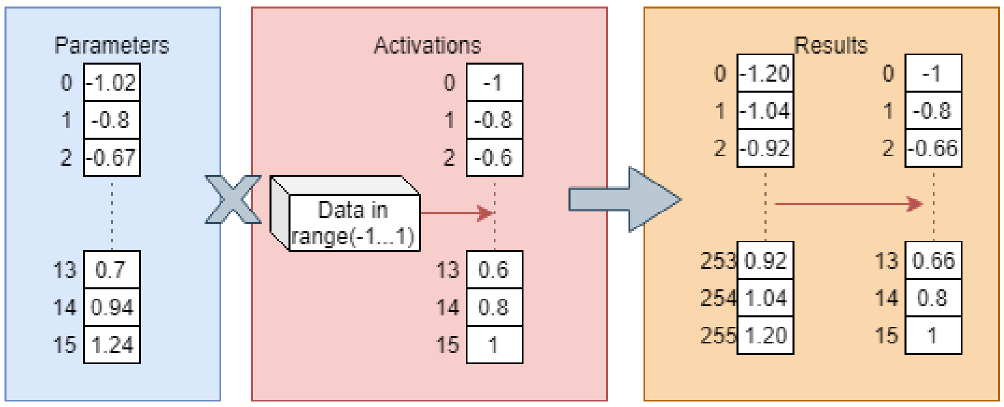


Figure 6.1: Workflow of a theoretical system that further utilizes index-mapping

This would replace the 16-bit floating-point calculations of the operation with 4-bit integer ones — possibly yielding further energy consumption and performance improvements (at the cost of further pre-runtime calculation).

Additionally, the 256 output mappings could be translated to the next operation's 16-bit map for activations, removing the need for conversion back and forth between the quantized and full-precision data types. An example of this in operation can be seen in Figure 6.1.

256 entries in the mapping would ensure complete coverage, but it is possible that there would be much overlap of resultant values, or similarities close enough that they could be represented by the same mapping, potentially reducing the amount of entries required.

With a lot of knowledge calculated ahead of execution, a future researcher could potentially create a very efficient process.

Executing Operations without Dequantization

If activations were index-mapped, certain NN operations could be changed to use the stored indices instead of the mapped FP16 values. This avoids the cost of retrieving the mapped values and will use low-precision integer hardware rather than higher-precision floating-point computation.

An example of this is to only select index mappings of values above zero to emulate a ReLU operation. This proposal can be seen in Figure 6.2 and also in more detail in Appendix A1.2.1.

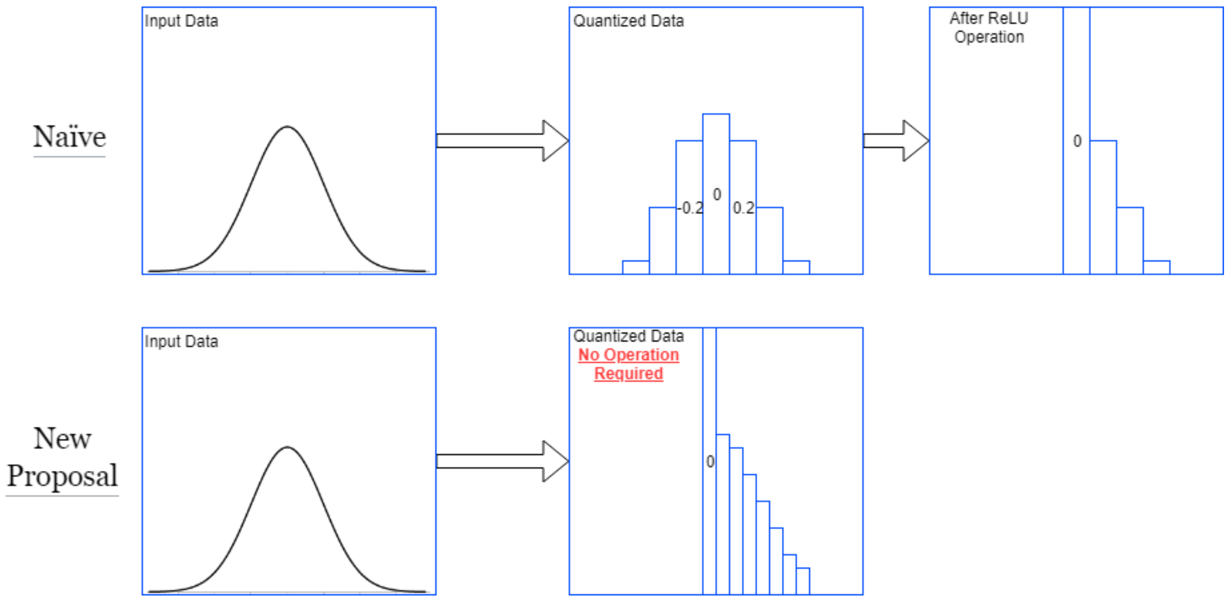


Figure 6.2: ReLU using only key values

The maximum pooling may also be optimized. If incremental order of the index-mapping table was assured, the maximum values for the pooling could be calculated with the 4-bit values rather than the 16-bit ones as the relative magnitude between values would be preserved. See Appendix A1.2.2 for a worked example.

6.2.5 Channel-Group Quantization

There are constraints on the NCE hardware. One of these is the size of local memory used to store input and output memory buffers. These limits affect the maximum size a convolution can operate on in a single inferece. Multiple inferences may be needed to perform larger convolutions if their memory buffers will not fit in local storage.

As the hardware needs to be reconfigured with every inference (to set new data offsets and other configuration registers), there is no penalty to changing the index-mapped values at the same time. One could thus have a convolution with 64 output channels, where the first 32 channels have a different set of quantization values versus the latter 32 channels. There could be some statistical variance between these channels that a multi-part mapping could give slight improvements to accuracy rates.

This research could be further expanded to splitting convolutions into differently sized groups, and investigating those that best preserve accuracy with reduced performance deterioration.

Additionally, some of the operation replacements mentioned in Section 2.1.1 may provide other types of grouping for the quantized values. A particular optimization in the AlexNet network is "Grouped" Convolutions (Krizhevsky, 2017) which assigns several sets of parameters to a convolution. The parameter sets may have different statistical profiles warranting individual index-mappings.

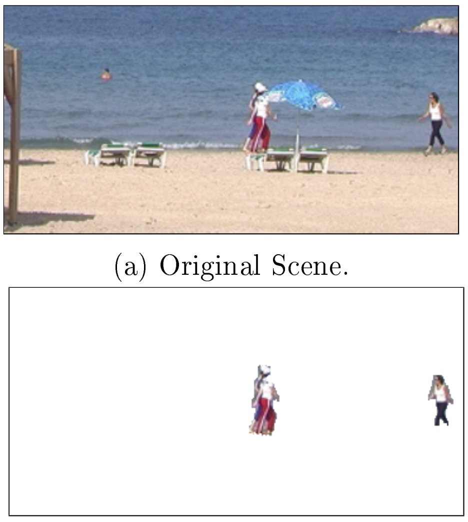
Locality-Sensitive Hashing

Parameters are generally applied as a grid over an entire input matrix, so the potential for optimizing the parameters based on locality is limited (as we discovered in Section 4.4.1). However, activations do have particular regions that may be more or less interesting to the network (e.g. a static CCTV camera may be partially obscured by an adjacent flagpole. It is unlikely that a pedestrian would be detected in the same space as the flagpole).

This has been briefly studied experimentally by fellow Intel® researchers (Xu and

Moloney, 2016) with promising results. Set portions of the input images can be removed prior to running a network, where the removed content is irrelevant to the resultant classification.

It would also be possible to investigate some more intelligent algorithms in this space. One could use a technique like background subtraction (Li Cheng et al., 2011) to remove ever-present components of an video stream (See an example in Figure 6.3), or an image could be segmented for mapping based on an algorithm detecting feature importance. A theoretical segmentation based on peaks in the image is shown in Figure 6.4 below.


(a) Original Scene.

(b) Foreground Only.

Figure 6.3: Example of background subtraction when detecting pedestrians[3]

Regardless of which method is used to segregate the network, each region could have different sets of quantization mappings. There would likely be some performance trade-off between the number of these regions and the frequency of mapping operations.

[3]Source: https://web.bii.a-star.edu.sg/archive/machine_learning/Projects/BkgSbt.htm

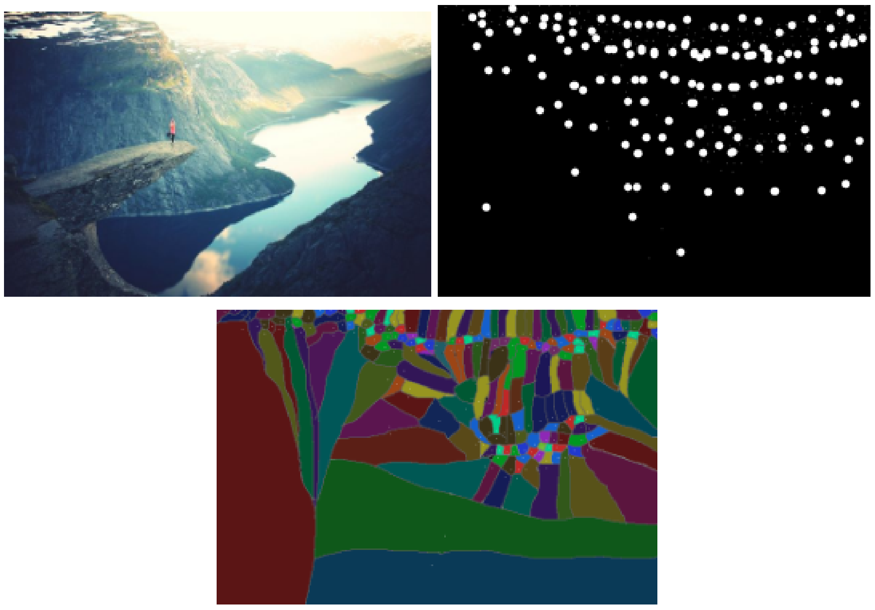


Figure 6.4: Using peak-detection to identify different regions for index mapping[4]

6.2.6 Extension of the Training process

It is possible to include additional variables in the training process. It could be both a faster process and potentially a more accurate estimation if the quantized centroids were chosen through the training process rather than post-pass algorithms like in this thesis. Alternatively, the index-mapped values themselves could be added as additional variables to be trained.

6.2.7 Expanding to Wider Fields of data

It may be interesting to see how the concepts in this thesis transfer to other NN variants, such as audio or natural language processing (NLP) inputs, recurrent neural networks (RNNs) and other operation types.

[4]Source for original image: https://unsplash.com/photos/asct7UP3YDE

6.2.8 Variable lookup tables

The NCE is limited in its current rendition to 16 values for quantization lookup. Depending on the complexity of the network or operations, fewer or more values could be used. This could give the developer more control over choices of numeric representation and speed.

The stored values are floating-point 16 and thus operations on the hardware have a similar compute time to their unmapped equivalents. Storing values in a lower-precision format could improve compute times. Investigation could be done into the effect of this on the network's classification accuracy.

6.2.9 Additional Processing Power

Despite arguments against Moore's Law coming to an end[5][6][7], it is inevitable that compute power in future years will be much above that what has been available today in this project, even if it does not scale at the same rate as proposed.

Even through the duration of the project, new GPUs from both Nvidia and AMD have been released with improved state-of-the-art performance. The flagship Nvidia TITAN Xp GPU released in 2018, advertises approximately 10.5K Gigaflops,[8] compared to the GTX Titan X GPU's 6.5K Gigaflops in 2015.[9]

With this additional compute power available to a future researcher, research involving the training process will be much more accessible. The progress in this thesis slowed as our own research entered this space (as described in Section 4.7.1) and I had to work around these limitations with techniques such as continuing from a previously trained-model and reducing the amount of network to quantize. With more compute power available, A more exhaustive test suite could have been created, possibly making it easier to assess our algorithm proposals.

[5]`https://interestingengineering.com/no-more-transistors-the-end-of-moores-law`

[6]`https://spectrum.ieee.org/nanoclast/semiconductors/devices/what-globalfoundries-retreat-really-means`

[7]`https://www.sciencefocus.com/future-technology/when-the-chips-are-down/`

[8]`https://www.nvidia.com/en-us/titan/titan-xp/.Accessed:01/08/2019`

[9]`https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x/specifications`. Accessed: 01/08/2019

6.3 Closing Thoughts

It is the authour's opinion that low precision neural networks will become increasingly popular over the next few years, but there are a myriad of possibilities of what eventual industry standards will look like. Promoting early innovative research will help Intel® Movidius™ lead those standards and secure themselves as authorities in the space. Several other large companies, such as Google[10] and Facebook[11], are already powerful voices in ANN developer circles, partially for this reason. Additionally, this will naturally gravitate NN researchers towards them as potential employees, further establishing them as domain leaders.

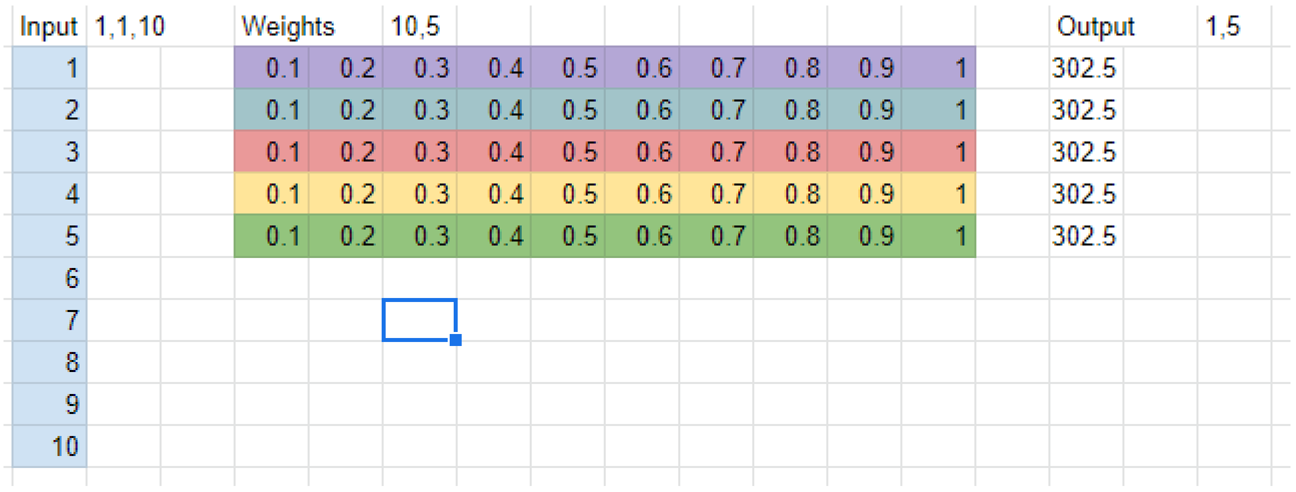
This thesis builds upon a foundation of work that the company has done in the design of the Intel® Movidius™Myriad™ VPU and shows that there is still room for improvement in the existing hardware. I hope that this work can influence both current associated software of the MyriadX processor, and inform aspects of the hardware for future designs.

[10]Creators of the protobuf format, TensorFlow, GEMMLOWP, TPUs, etc.
[11]Employer of Yann LeCun, Developers of PyTorch, maintainers of the popular NN framework fork Caffe2, etc.

A1 Appendix: Reference Calculations

A1.1 Fully Connected Layers as Convolutions

Fully connected layers can be computed with a sum of products. This can be done via matrix multiplication, but as can be seen in the code snippets in Section 2.1.1, the core computation is a sum of products as well.[1]



Input	1,1,10		Weights		10,5									Output		1,5
1			0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1		302.5		
2			0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1		302.5		
3			0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1		302.5		
4			0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1		302.5		
5			0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1		302.5		
6																
7																
8																
9																
10																

Figure A1.1: FCL can be calculated with a matrix multiplication

[1]Reference for this optimization: http://cs231n.github.io/convolutional-networks/

A1.2 Using Indices for computing operations

A1.2.1 ReLU

In Table A1.4, the application of an index-mapping before a ReLU operation is shown. The values of Table A1.1 are a sample of data from a larger input where each element is in the range between -3.5 and 4.

After the application, we will find all the negative values (there are two in the example) and set them to zero.

This is the normal flow of operation, however it is possible to improve upon this. In the second set of tables (TableA1.8) our Index-mapping table has mappings only for positive values. We can see after this application in Table A1.7 that because the lowest possible value is zero, all negative values have been set to this — Effectively computing a ReLU operation at the same time.

Not only do we gain the benefit of skipping the explicit operation, but the input data will have double the capacity to preserve values (as any negative values in the mapping are discarded)

-0.6	0.12	0.01
0.3	0.36	0.06
0.44	-0.6	0.64

Table A1.1: Original Values

4-bit index	16-bit value
0000	-3.0
0001	-2.5
0010	-2.0
0011	-1.8
0100	-1.5
0101	-1
0110	0.3
0111	0.8
1000	1
1001	1.2
1010	1.4
1011	1.8
1100	2.4
1101	2.6
1110	2.9
1111	3.5

Table A1.2: Index-Mapping Table

-1	0.3	0.3
0.3	0.3	0.3
0.3	-1	1

Table A1.3: Mapped Values

Table A1.4: Usual Mapping preceding a ReLU

-0.6	0.12	0.01
0.3	0.36	0.06
0.44	-0.6	0.64

Table A1.5: Original Values

4-bit index	16-bit value
0000	0
0001	0.2
0010	0.4
0011	0.7
0100	1
0101	1.2
0110	1.4
0111	1.7
1000	1.9
1001	2.1
1010	2.3
1011	2.6
1100	2.8
1101	3.1
1110	3.3
1111	3.5

Table A1.6: Index-Mapping Table

0	0.2	0
0.4	0.4	0
0.4	0	0.7

Table A1.7: Mapped Values

Table A1.8: Performing a ReLU operation with the mapping

A1.2.2 Maximum Pooling

If we can assure that the index-mapping has its values sorted in either ascending or descending order, the comparison relations should be preserved.

Table A1.9 and Table A1.10 both use the index-mapping from the previous Table A1.2. In the latter case, the values are not resolved.

It can be seen by comparing (c) of both groups that the exclusion of resolving the index-mapping values does not affect the pooling as the final results are identical.

-0.6	0.12	0.01
0.3	0.36	0.06
0.44	-0.6	0.64

(a) Original Values

-1	0.3	0.3
0.3	0.3	0.3
0.3	-1	0.8

(b) Mapped Values

0.3	0.3
0.3	0.8

(c) Pooled Values

Table A1.9: 2x2 Maxpooling over values

5	6	6
6	5	5
6	5	7

(a) Original Values

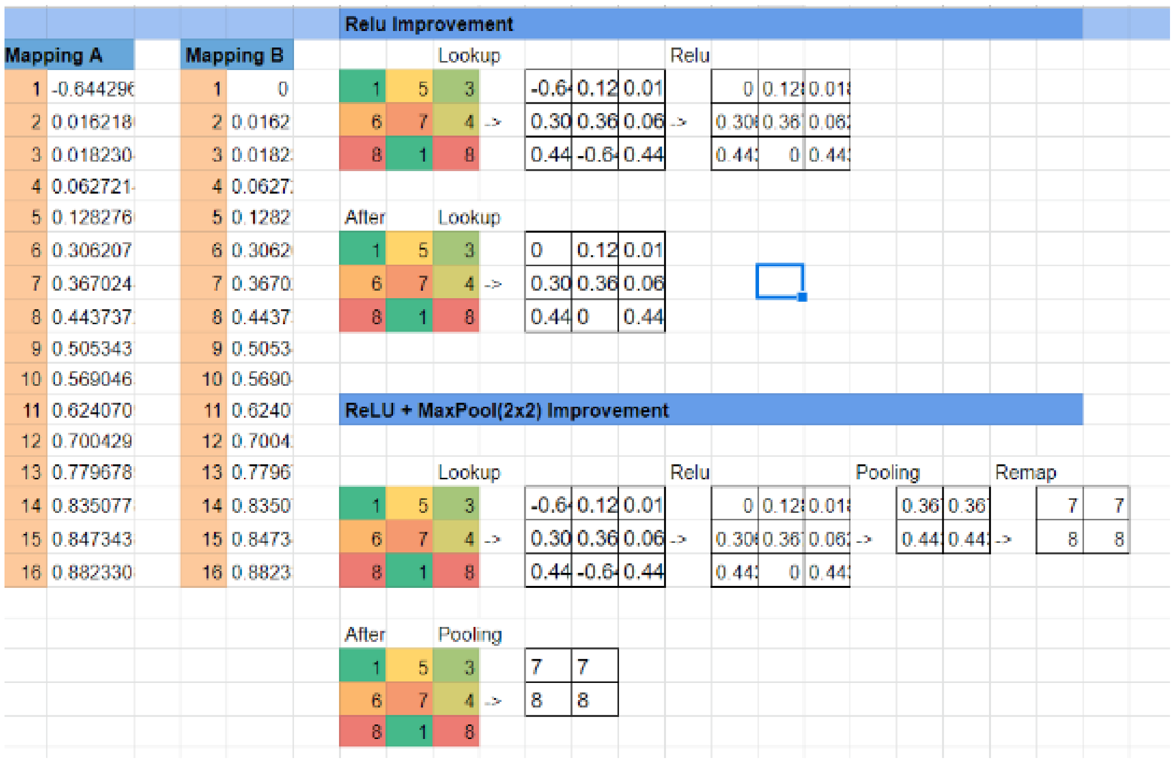
6	6
6	7

(b) Pooled Values

0.3	0.3
0.3	0.8

(c) Mapped Values

Table A1.10: 2x2 Maxpooling over indices



Mapping A		Mapping B	
1	-0.64429	1	0
2	0.016218	2	0.0162
3	0.018230	3	0.0182
4	0.062721	4	0.0627
5	0.128276	5	0.1282
6	0.306207	6	0.3062
7	0.367024	7	0.3670
8	0.443737	8	0.4437
9	0.505343	9	0.5053
10	0.569046	10	0.5690
11	0.624070	11	0.6240
12	0.700429	12	0.7004
13	0.779678	13	0.7796
14	0.835077	14	0.8350
15	0.847343	15	0.8473
16	0.882330	16	0.8823

Relu Improvement

	Lookup							Relu		
1	5	3		-0.6	0.12	0.01		0	0.12	0.01
6	7	4	->	0.30	0.36	0.06	->	0.30	0.36	0.06
8	1	8		0.44	-0.6	0.44		0.44	0	0.44

After	Lookup					
1	5	3		0	0.12	0.01
6	7	4	->	0.30	0.36	0.06
8	1	8		0.44	0	0.44

ReLU + MaxPool(2x2) Improvement

	Lookup							Relu				Pooling			Remap	
1	5	3		-0.6	0.12	0.01		0	0.12	0.01		0.36	0.36		7	7
6	7	4	->	0.30	0.36	0.06	->	0.30	0.36	0.06	->	0.44	0.44	->	8	8
8	1	8		0.44	-0.6	0.44		0.44	0	0.44						

After	Pooling				
1	5	3		7	7
6	7	4	->	8	8
8	1	8			

Figure A1.2: Example of how Maximum Pooling and ReLU can be optimized away. Each operation has the original calculation on top and the new calculation below

A2 Appendix: Access to Project Resources

Some of the following code repositories have restricted access. Access can be granted by the author if contacted, subject to agreements with Intel®.

A2.1 Custom Graphing Tools

Custom Tool to draw visual representations of networks

- `https://github.com/ianfhunter/Neural-Network-Data-Analysis`
- `https://github.com/ianfhunter/graphing-Neural-networks`

A2.2 VPU Profiling Code

Intel® Movidius™Myriad™ VPU Profiling Code was developed on a fork of an internal Intel® Movidius™ code repository using the Intel® Movidius™ Software Development Kit

https://www.movidius.com/solutions/software-development-kit

A2.3 Experiments with Neural Network Training

- PyTorch-based Experiments & Test Suite - `https://github.com/ianfhunter/pytorch-training-scratch`

- Caffe-based Experiments - `https://github.com/ianfhunter/simple-convolution-test-code`
- Modified CK-means - `https://github.com/ianfhunter/ckmeans`
- Modified K-means `https://github.com/ianfhunter/kmeans`

A2.4 Modified Third-Party Libraries

The neural network framework “PyTorch” and associated network files

- Modified Base Framework - `https://github.com/ianfhunter/pytorch`
- Modified Examples - `https://github.com/ianfhunter/examples`
- `https://github.com/ianfhunter/pytorch-examples`
- Model Library - `https://github.com/pytorch/vision`

Miscellaneous

- Sparse Models - `https://github.com/ianfhunter/scalpel`
- Short experiment integrating our code into the caffe framework - `https://github.com/ianfhunter/caffe`

A2.5 Unmodified Third-Party Libraries

The neural network framework “TensorFlow” and associated network files and existing popular quantization scheme

- Custom Verification Suite - `https://github.com/ianfhunter/tensorflow-classification-rates`
- Low-Precision Matrix Multiplication library in TensorFlow - `https://github.com/ianfhunter/gemmlowp`

The neural network framework “Caffe” and associated network files

- Framework - `https://github.com/BVLC/caffe`

Miscellaneous

- k-means C++ library `https://github.com/ianfhunter/dkm`

A2.6 Files for ImageNet Dataset Validation

- The Neural Network Framework "Caffe" and associated network files - `https://github.com/ianfhunter/Masters-NN`
- CIFAR100 Dataset - `https://github.com/ianfhunter/caffe-cifar-10-and-cifar-100-datasets-preprocessed-to-HDF5`

A3 Appendix: Larger Images

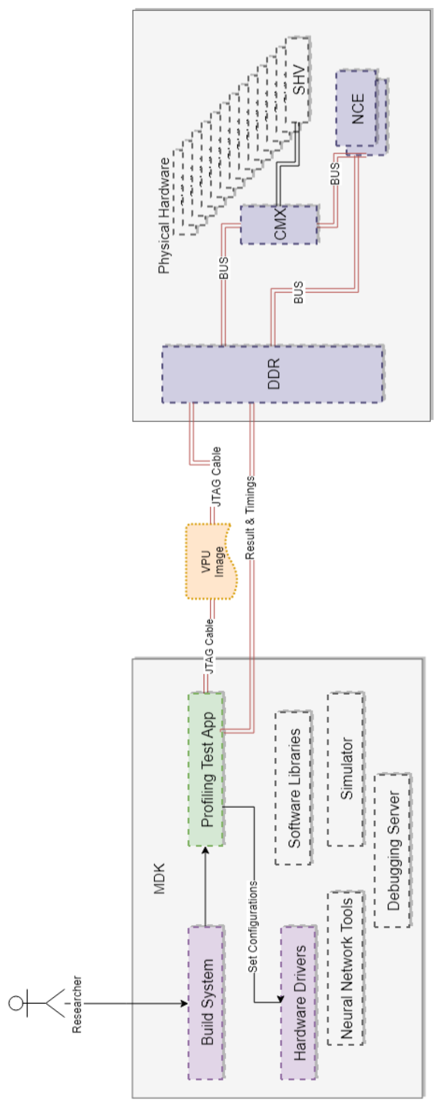


Figure A3.1: Running the Hardware Profiling Software

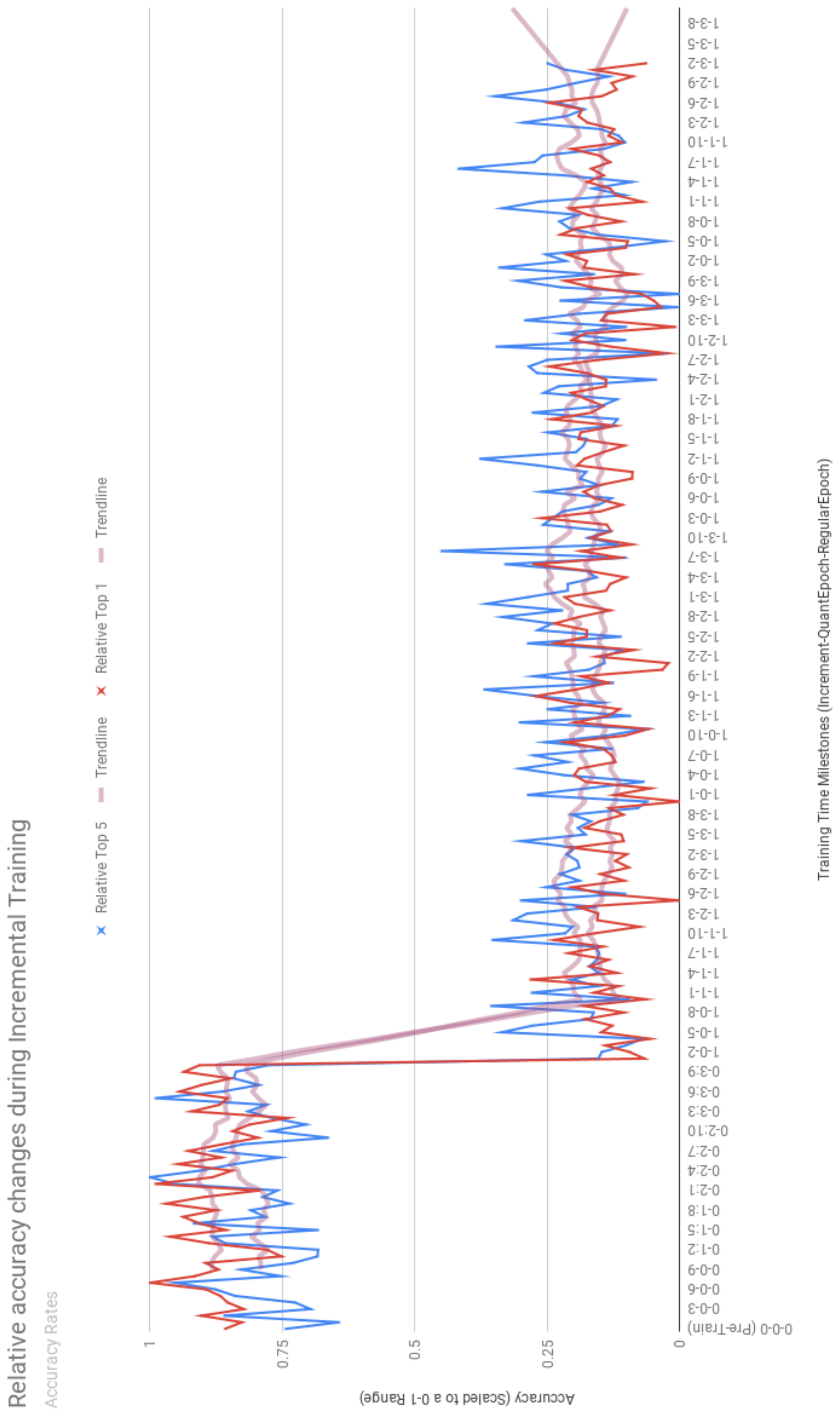


Figure A3.2: Larger In-Training System Results

A4 Appendix: Glossary

- **Activation** — An input to a neural network operation, either from the dataset item, or from the output of a previous operation.
- **AlexNet** — One of the first modern convolutional neural networks. Famous for winning ImageNet 2012
- **ANNs** — Artificial Neural Networks, commonly shortened to just “Neural Networks”
- **Bias** — Values added to the result of an NN operation.
- **Caffe** — A neural network framework developed at the University of California, Berkley
- **Capacity** — The topological complexity of a network (Guss and Salakhutdinov, 2017)
- **Convolution** — A basic neural network operation. Explained in Section 2.1.1
- **CNN** — Convolutional Neural Network, a subset of ANNs that use series of convolutions and other operations to calculate results.
- **CV** — Computer Vision, also known as Machine Vision. A class of techniques used for image-based processing.
- **Dataset** — A large collection of data (in this thesis, usually images), that can be used for supervised learning. Neural networks can require massive amounts of these data items.

- **DNNs** — "Deep neural networks". A term used to refer to CNNs which consist of many operations.
- **DMA** — Direct Memory Access. A memory transfer mechanism that can access memory independent of core processors.
- **DPE** — Data Processing Element - A component of an NCE
- **Elementwise Multiplication** — The term used to describe Hadamard Products on Matrices, when in neural networks
- **Epoch** — A full training pass on a dataset
- **GoogLeNet** — A neural network developed by Google, famous for winning ImageNet 2014 and introducing the "Inception" sub-architecture. The name is a homage to LeNet
- **GPU** — Graphical Processing Unit - A processor designed for graphical workloads
- **ImageNet** — Can refer to either the popular Image classification competition held every year, or the dataset used in this contest. The 2012 competition is typically the one compared against in neural network papers.
- **ILSVRC** — "ImageNet Large Scale Visual Recognition Competition" - See ImageNet
- **JTAG cable** — a cable standardized by the Joint Test Action Group used for serial communication with processors.
- **Kernel** — a small grid based matrix function.
- **LeNet** — A neural network developed by Yann LeCun, commonly referenced as the first convolutional neural network
- **TensorFlow** — A mathematical framework developed by Google, Commonly used for neural networks
- **MDK** — Movidius Development Kit - also known as the Movidius SDK
- **Myriad** — Intel® Movidius™'s VPU Range

- **Neuron** — A unit of processing in a neural network.
- **NCE** — Neural Compute Engine, hardware accelerator of the Intel® Movidius™Myriad™ VPU
- **NCS** — Neural Compute Stick — A USB powered stick hosting the Intel® Movidius™Myriad™ VPU
- **OBD** — Optimal Brain Damage, a technique introduced by Yann LeCun to remove unimportant parameters from a network
- **PyTorch** — An open-source neural network framework developed mainly by Facebook engineers
- **Protobuf** — A file format designed by Google aiming to provide efficient compression for constrained platforms
- **Receptive Field** — The spatial region in which a change will modify the properties of the resultant neuron
- **RISC** — Reduced Instruction Set Computer. A computer whose instruction set architecture consists of a small set of highly optimized instructions o (as opposed to CISC — Complex Instruction Set Computer).
- **Saliency** — A measure of the importance of values in an algorithm
- **SDK** — Software Development Kit. A set of tools to access software functionality of a development platform
- **SNN** — Spiking Neural Network. A subset of ANNs where the activations are not all processed at once. Activations "fire" when its build up magnitude reaches certain thresholds.
- **Sparsity** — An optimization for neural networks that avoids computing kernels with predictable zero values
- **SPARC** — **S**calable **P**rocessor **Arc**hitecture, a RISC instruction set architecture.
- **Supervised Learning** — Algorithms that learn properties from labelled

datasets.

- **TPU** — Tensor Processing Unit. A term used by Google to describe its range of NN Processors
- **Unsupervised Learning** — Algorithms that learn without the presence of labelled datasets.
- **UML** — Unified Modeling Language. A language for diagram creation, commonly used in software management.
- **VC Dimension** — Vapnik–Chervonenkis (VC) dimension. A measure for NN capacity
- **VPU** — Vision Processing Unit. A class of processors dedicated to computer vision tasks.
- **Weight/Parameter** — A value used in the computation of a neural network operation, trained by previous inputs

Bibliography

Agapitos, A., O'Neill, M., Nicolau, M., Fagan, D., Kattan, A., Brabazon, A., and Curran, K. (2015). Deep evolution of image representations for handwritten digit recognition. *2015 IEEE Congress on Evolutionary Computation (CEC)*, pages 2452–2459.

Ando, K., Takamaeda-Yamazaki, S., Ikebe, M., Asai, T., and Motomura, M. (2017). A multithreaded cgra for convolutional neural network processing. *Circuits and Systems*, 8:149–170.

Arthur, D. and Vassilvitskii, S. (2007). K-means++: The advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, pages 1027–1035, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Behringer, P. (2019). Are neural networks the future of machine vision? Technical report, Basler.

Cao, Z., Long, M., Wang, J., and Yu, P. S. (2017). Hashnet: Deep learning to hash by continuation. *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 5609–5618.

Chandrasekhar, V., Lin, J., Morère, O., Goh, H., and Veillard, A. (2015). A practical guide to cnns and fisher vectors for image instance retrieval. *Signal Processing*, 128.

Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., and Temam, O. (2014). Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 269–284, New York, NY, USA. ACM.

Csáji, B. C. (2019). *Approximation with Artificial Neural Networks.* PhD thesis, Faculty of Sciences.

Dey, S., Huang, K.-W., Beerel, P. A., and Chugg, K. M. (2018). Pre-defined sparse neural networks with hardware acceleration. *CoRR*, abs/1812.01164.

Fung, J. (2019). Computer vision on the gpu. *Addison-Wesley Professional.*

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Teh, Y. W. and Titterington, M., editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy. PMLR.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In Gordon, G. J., Dunson, D. B., and Dudík, M., editors, *AISTATS*, volume 15 of *JMLR Proceedings*, pages 315–323. JMLR.org.

Gong, J., Shen, H., Zhang, G., Liu, X., Li, S., Jin, G., Maheshwari, N., Fomenko, E., and Segal, E. (2018). Highly efficient 8-bit low precision inference of convolutional neural networks with intelcaffe. *CoRR*, abs/1805.08691.

Gurnani, A. and Mavani, V. (2017). Flower categorization using deep convolutional neural networks. *CoRR*, abs/1708.03763.

Guss, W. H. and Salakhutdinov, R. R. (2017). On characterizing the capacity of neural networks using algebraic topology. *CoRR*, abs/1802.04443.

Gysel, P. (2016). Ristretto: Hardware-Oriented Approximation of Convolutional Neural Networks. *arXiv e-prints*, page arXiv:1605.06402.

Han, S. (2017). *Efficient methods and hardware for deep learning.* PhD thesis, Jacobs School of Engineering.

Han, S., Mao, H., and Dally, W. J. (2015a). Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding.

Han, S., Pool, J., Tran, J., and Dally, W. J. (2015b). Learning both weights and connections for efficient neural networks. In *NIPS*.

Hashemi, S., Anthony, N., Tann, H., Bahar, R. I., and Reda, S. (2017). Understanding the impact of precision quantization on the accuracy and energy of neural networks. *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2017*.

Hornik, K. ("1991"). "approximation capabilities of multilayer feedforward networks". *"Neural Networks"*, "4"("2"):"251–257".

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing in Science Engineering*, 9(3):90–95.

Istrate, R., Malossi, A. C. I., Bekas, C., and Nikolopoulos, D. S. (2017). Incremental training of deep convolutional neural networks. In *AutoMLPKDD/ECML*.

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A. G., Adam, H., and Kalenichenko, D. (2018). Quantization and training of neural networks for efficient integer-arithmetic-only inference. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2704–2713.

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., Hassabis, D., Clopath, C., Kumaran, D., and Hadsell, R. (2016). Overcoming catastrophic forgetting in neural networks. cite arxiv:1612.00796.

Krizhevsky, A. (2017). Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90.

Lavin, A. (2016). Fast algorithms for convolutional neural networks. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4013–4021.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324.

LeCun, Y., Denker, J. S., and Solla, S. A. (1989). Optimal brain damage. In *NIPS*.

Li Cheng, M. G., Schuurmans, D., and Caelli, T. (2011). Real-time discriminative background subtraction.

MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297, Berkeley, Calif. University of California Press.

McCulloch, P. (1944). A logical calculus of the ideas immanent in nervous activity. *The Journal of Symbolic Logic*, 9(2):49–50.

Moosavi-Dezfooli, S.-M., Fawzi, A., Fawzi, O., and Frossard, P. (2017). Universal adversarial perturbations. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 86–94.

Morissette, L. and Chartier, S. (2013). The k-means clustering technique: General considerations and implementation in mathematica. *Tutorials in Quantitative Methods for Psychology*, 9:15–24.

Power, S., Moloney, D., Barry, B., and Connor, F. (2018). https://patentimages.storage.googleapis.com/9f/48/82/a3d8cee6a2b20f/wo2018211129a1.pdf.

Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. (2016). Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*.

Robbins, H. and Monro, S. (1951). A stochastic approximation method. *Ann. Math. Statist.*, 22(3):400–407.

Schaller, R. R. (1997). Moore's law: Past, present, and future. *IEEE Spectrum*, 34(6):52–59.

Simonyan, K., Vedaldi, A., and Zisserman, A. (2013). Deep fisher networks for large-scale image classification. In Burges, C. J. C., Bottou, L., Welling, M., Ghahramani, Z., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 26*, pages 163–171. Curran Associates, Inc.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research (JMLR)*, 15(1):1929–1958.

Sun, P., Feng, W., Han, R., Yan, S., and Wen, Y. (2019). Optimizing network performance for distributed dnn training on gpu clusters: Imagenet/alexnet training in 1.5 minutes. *CoRR*, abs/1902.06855.

Swirszcz, G., Czarnecki, W. M., and Pascanu, R. (2016). Local minima in training of neural networks. *arXiv e-prints*, page arXiv:1611.06310.

Tann, H., Hashemi, S., Bahar, R. I., and Reda, S. (2016). Runtime configurable deep neural networks for energy-accuracy trade-off. *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis - CODES '16.*

Vattani, A. (2019). The hardness of k-means clustering in the plane. In *Unpublished.*

Wang, H. and Song, M. (2011). Ckmeans.1d.dp: Optimal k-means clustering in one dimension by dynamic programming. *R J*, 3(2):29–33.

Xu, B. and Moloney (2016). Unpublished research. Movidius Internal Paper.

Yosinski, J., Clune, J., Bengio, Y., and Lipson, H. (2014). How transferable are features in deep neural networks? In *NIPS.*

Yu, D., Wang, H., Chen, P., and Wei, Z. (2014). Mixed pooling for convolutional neural networks. In *RSKT.*

Zhou, B., Khosla, A., Lapedriza, A., Oliva, A., and Torralba, A. (2016). Learning deep features for discriminative localization. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2921–2929.

Zhu, C., Han, S., Mao, H., and Dally, W. J. (2017). Trained ternary quantization. *CoRR*, abs/1612.01064.