

# Optimisation of Neural Networks for Embedded Systems

Robert A. Koch - 25124390  
robert@kochie.io

Supervisor: Prof Tom Drummond

20 October, 2017

## List of Significant Contributions

- Created a new library for the development and testing of Neural Networks using the TensorFlow framework, with the ability to:
  - Train a network on any custom database.
  - Run multiple training sessions on multiple GPUs.
  - Create networks comprised of "quantised" components.
  - Automatically create datasets from directory hierarchies.
  - Train networks until a specific error rate has been achieved.
  - Display useful information through the TensorBoard product.
- Decreased the training time of image classification networks by implementing binary and ternary quantised states in the network.
- Improved the accuracy of binary networks by incorporating a third activation level in the network.
- Compared the speed, accuracy, and resource cost of several networks including AlexNet, VGG, and Inception to binary, and ternary counterparts.
- Compared the speed, accuracy, and resource cost these networks had on different datasets to determine any differences in training.
- Compared the speed of several types of networks both optimised and not optimised in regards to input batch sizes.
- Modified an existing Torch repository to include optimised models of several networks using Binary and Ternary activation functions.
- Developed a comprehensive understanding of how frameworks such as Torch and TensorFlow create neural networks.
- Developed a proficient understanding of how deep learning systems such as neural networks operate.
- Gained a greater understanding in how to use software packages such as L<sup>A</sup>T<sub>E</sub>X, OpenCV, and the GNU-Linux ecosystem as a whole.

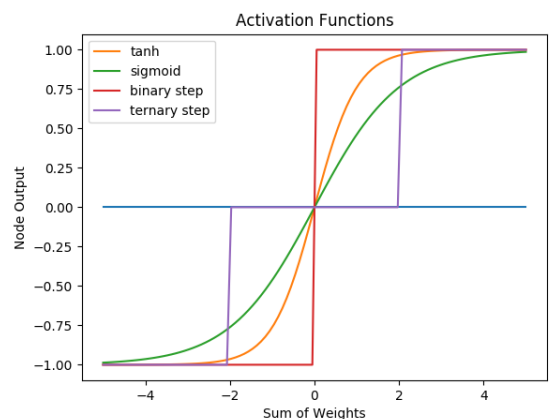


# Optimising Neural Networks for Embedded Systems

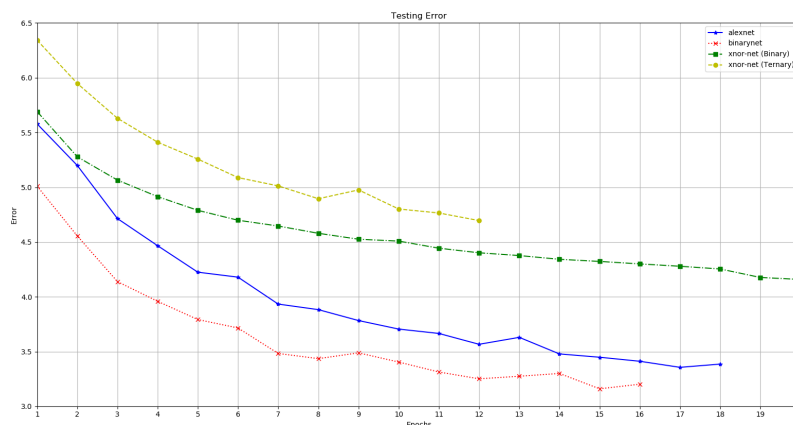
Supervisor: Prof Tom Drummond

Current implementations of Artificial Neural Networks and other Machine Learning algorithms are designed to run on dedicated systems with large quantities of memory and parallel computation cores (e.g. GPUs). However these requirements are beyond the current capacity of many embedded systems such as mobile phones, microcontrollers, and IoT connected devices. Current solutions to this problem focus on the concept of "Quantising" specific aspects of neural networks to allow them to train, test, and function faster on all systems including embedded devices. This project aims to provide a method of quantising specific networks aimed at image classification by changing the weights and activation functions of certain well-known networks like AlexNet and Inception from floating point numbers to integer based numbers. The main advantage of this change is the ability to use bitwise operations on specific layers instead of floating point operations that require large amounts of processing power.

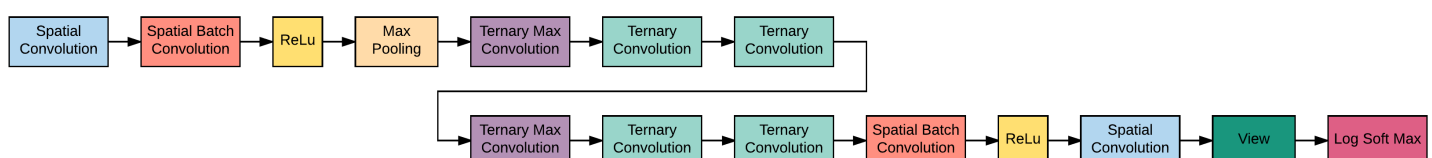
The basic functionality of a neural network comes from the activation functions. Each neuron in a network sums the weighted inputs to itself and applies an activation function which creates the output value for that node. On a relatively large network there are hundreds of thousands of nodes, each with an activation function that takes thousands of inputs, quite quickly the amount of computing power needed can reach super-computing levels. Most activation functions are non-linear (sigmoid, tanh, etc.), by using non-linear functions networks have the ability to model problems that are themselves non-linear. These functions require floating point precision to be accurate, however by approximating these functions with binary, and ternary steps the required floating point computation can be reduced. By only applying this optimisation to specific "hidden" layers the overall efficiency of the network can be increased at the cost of some accuracy. The game then becomes optimising this loss enough that the network can still perform better at specific tasks than humans. This optimisation is possible because the step functions follow the shape of the non-linear functions enough that they can emulate them over enough iterations.



There are many different frameworks available to create machine learning algorithms. Most early networks were written in Lua using the torch framework, however, current advances in machine learning have lead companies to develop their own in house solutions which have gained widespread adoption. The Google built Tensorflow framework works with languages including Python, Java, and C++. It also has many advances features including a large library, a cloud service for running distributed networks, and a visualization tool called TensorBoard.



By changing several middle layer activation functions, the accuracy of the networks fell, as expected, however the training speed of the network increased, with the average speed of training alexnet being 3 seconds and the improved ternary network training in 2 seconds. By using this technique, it is possible to load networks designed to run on large x86 machines onto smaller ARM computers such as the Raspberry Pi, and Android mobiles. By using three layers instead of two in the ternary function the number of operations compared to a binary function is the same because the third level of zero requires no resources to compute, however, it adds an extra level of precision for free. This allows the network to become more accurate for little to no cost. This technique can be expanded in the future to allow extra discrete levels of computation for more accuracy with less resources being used.



## Executive Summary

This project focused on understanding and optimising neural networks designed for image classification. The speed, accuracy, and computation power of certain networks was benchmarked against optimised counterparts. The benchmarking of networks was conducted using a combination of both the Torch(Lua) and TensorFlow(Python) frameworks.

Using TensorFlow as a foundation a new library called pyNet was created for the specific use case of training and constructing networks with complex data structures in python. This program was also used to create and modify existing datasets to use for training arbitrary networks regardless of their parameters. Additionally this library can be used to create new datasets from existing images and compile them into TensorFlow ready binaries.

The method of optimising these networks is known as quantisation and focused around changing how the activation functions in the convolution layers of these networks evaluates values. By changing the functions to piecewise linear from non-linear functions it was possible to increase the speed at which batches were processed in the network and also increase the training time. Inversely the accuracy and of the network suffered as the models used were rudimentary and suffered from several mitigating issues that were eventually fixed in later iterations.

In this project several aspects of the networks were investigated leading to the conclusion that if done correctly, optimisation by quantising a neural network can improve overall performance.

This project provided a way of experimenting with different aspects of the activation functions used in most major deep learning networks currently in operation. It was found, among other things, that changing the activation function to a ternary step will decrease performance if the threshold value is not configured correctly due to underfitting, and the derivative must represent the activation function more precisely to avoid drift when performing gradient descent in backpropagation.

# Contents

|                                                                |           |
|----------------------------------------------------------------|-----------|
| <b>List of Significant Contributions</b>                       | <b>2</b>  |
| <b>Poster</b>                                                  | <b>2</b>  |
| <b>Executive Summary</b>                                       | <b>4</b>  |
| <b>List of Figures</b>                                         | <b>6</b>  |
| <b>1 Introduction</b>                                          | <b>7</b>  |
| <b>2 Theory and Background</b>                                 | <b>8</b>  |
| 2.1 Floating Point Numbers . . . . .                           | 8         |
| 2.2 Non Linear Functions . . . . .                             | 8         |
| 2.3 Mathematical Optimisation . . . . .                        | 8         |
| 2.4 Deep Learning . . . . .                                    | 8         |
| 2.4.1 Artificial Neural Networks . . . . .                     | 9         |
| 2.4.2 Backpropagation . . . . .                                | 10        |
| 2.4.3 Overfitting . . . . .                                    | 11        |
| 2.4.4 Quantising . . . . .                                     | 11        |
| 2.5 Programming . . . . .                                      | 12        |
| 2.5.1 CUDA . . . . .                                           | 12        |
| 2.5.2 Torch - Lua . . . . .                                    | 12        |
| 2.5.3 TensorFlow - Python . . . . .                            | 13        |
| 2.6 Optimisation via Binary and Ternary Quantisation . . . . . | 13        |
| 2.6.1 Current Research . . . . .                               | 13        |
| <b>3 Torch Implementation</b>                                  | <b>14</b> |
| 3.1 Outline . . . . .                                          | 14        |
| 3.2 Development and Programming . . . . .                      | 14        |
| 3.3 Challenges and Solutions . . . . .                         | 14        |
| <b>4 TensorFlow Implementation</b>                             | <b>15</b> |
| 4.1 Outline . . . . .                                          | 15        |
| 4.2 Research and Results . . . . .                             | 16        |
| 4.3 Improvements and Adaptations . . . . .                     | 16        |
| <b>5 Results and Analysis</b>                                  | <b>16</b> |
| 5.1 Network Error Rates . . . . .                              | 16        |
| 5.2 Network Training and Testing Accuracies . . . . .          | 18        |
| 5.2.1 Top-1 Accuracies . . . . .                               | 18        |
| 5.2.2 Top-5 Accuracies . . . . .                               | 20        |
| 5.3 Batch and Filesize Comparisons . . . . .                   | 22        |
| 5.4 Analysis . . . . .                                         | 23        |
| <b>6 Conclusion</b>                                            | <b>24</b> |
| <b>References</b>                                              | <b>25</b> |

## List of Figures

|    |                                                                                     |    |
|----|-------------------------------------------------------------------------------------|----|
| 1  | A simple network diagram . . . . .                                                  | 9  |
| 2  | The shape of different types of activation functions. . . . .                       | 10 |
| 3  | An example of back propagation. . . . .                                             | 11 |
| 4  | The operation flow of CUDA processing. . . . .                                      | 12 |
| 5  | An example of the layout of a network in TensorFlow. . . . .                        | 15 |
| 6  | Comparison of training errors for optimised and regular networks. . . . .           | 17 |
| 7  | Comparison of testing errors for optimised and regular networks. . . . .            | 18 |
| 8  | Comparison of Top-1 testing accuracies for optimised and regular networks. . . . .  | 19 |
| 9  | Comparison of Top-1 training accuracies for optimised and regular networks. . . . . | 19 |
| 10 | Comparison of Top-5 testing accuracies for optimised and regular networks. . . . .  | 20 |
| 11 | Comparison of Top-5 training accuracies for optimised and regular networks. . . . . | 21 |
| 12 | A comparison of activation functions and their derivatives. . . . .                 | 22 |
| 13 | Comparison of batch times for optimised and regular networks. . . . .               | 22 |
| 14 | Comparison of file sizes for optimised and regular networks. . . . .                | 23 |

## List of Algorithms

|   |                                        |    |
|---|----------------------------------------|----|
| 1 | Ternary Activation Algorithm . . . . . | 14 |
|---|----------------------------------------|----|

# 1 Introduction

The concept of Deep Learning has come a long way in such a short time. Just in this decade alone advances in deep learning have created solutions to problems in all corners of society including commercial, research, and military-industrial sectors that were once the sole domain of human ingenuity. Leading this boom in technology is the Artificial Neural Network, a computing system originally devised in the 1940s as a way of modelling the human brain on a computer. Since then the concept has had many different names including cybernetics[4] and connectionism[13], but is now known today by the biological mechanism it was once modelled for.

Neural Networks are advantageous because they do not need to be specifically programmed to perform a task[4], they simply need to be shown how to correctly complete a specific task. This type of algorithm is also known as a supervised learning algorithm - a subset of problems that can be solved using such networks. Neural networks gain their power from their complex structure and their non-linear components which allow them to solve non-trivial problems better than any other type of system, including humans for the most part.

Current implementations of neural networks were designed to be run specifically on dedicated hardware with vast quantities of memory and multiple dedicated parallel computing units. While this implementation will probably not change in the near future there are several exciting new opportunities on the horizon for versions of these algorithms that can run on smaller and less power consuming systems. Self-Driving Cars[1], Helicopters on Mars[8], and even Advanced AI assistants on phones[11] are all areas where embedded neural networks could increase the performance and profitability of such systems.

This report will explain several methods that can be used today to miniaturise and optimise resource heavy networks to run on lighter machines.

## 2 Theory and Background

### 2.1 Floating Point Numbers

A Floating point number is a representation of a real number that is approximated so that it is possible to store on a computer with a finite number of bits. This trade-off of precision compared to range allows the numbers to be stored efficiently. All programming languages have some method for storing and manipulating floats, with most modern representations of single floats comprising of 32-bits, with 8-bits used for the exponent. Floating point numbers are ideal for storing precise values of real numbers compared to integers because they support decimal places, compared to integer precision arithmetic, floating point is more resource intensive on most systems because it requires more CPU operations - in fact, the number of Floating Point Operations Per Second (FLOPS) is used as a measurement in supercomputing benchmarks.

### 2.2 Non Linear Functions

In mathematics a function is a relation between a set of inputs and a set of outputs. Each input is mapped to exactly one output, however multiple inputs can map to the same output. A non-linear function is a relationship where the inputs are not proportional to the outputs, they are of interest to many fields in science and engineering because most systems in nature are inherently non-linear.

A linear function can be easily defined mathematically as a function which holds to these properties:

$$f(x + y) = f(x) + f(y) \quad (1a)$$

$$f(\alpha x) = \alpha f(x) \quad (1b)$$

Where  $\alpha \in \mathbb{R}$  by definition a non-linear function is one that does not follow either of the above properties. These functions can be used in a neural network as activation functions so the network can accurately model non-linear systems[6].

### 2.3 Mathematical Optimisation

In mathematics, optimisation is the selection of a best value from a specific set, based on some sort of criteria. The input set can be the output of a function, this gives a formal method for optimising a mathematical function to get a specific output. This can also be described as below:

$$\min f(x), \text{ subject to } f(x) = x^2 + 1, x \in (-\infty, 0] \quad (2)$$

One of the simplest forms of an optimisation problem is that of finding the minimum or maximum outputs of a function, this can be achieved by selecting input values from within a selected range and then comparing the returned value[3].

### 2.4 Deep Learning

Deep learning is a subset of machine learning focusing on methods based around learning specific data representations to solve problems rather than task-specific algorithms that are programmed to solve a problem. A deep learning algorithm can learn how to solve a problem without being specifically programmed how, using either a supervised, partially supervised, or unsupervised algorithm. The great advantage that deep learning algorithms have is that they can perform better than people at specific tasks, and this set of tasks is increasing.

### 2.4.1 Artificial Neural Networks

An artificial neural network is a computing system inspired by the design of the human brain. In simplistic terms the system is analogous to the operation of neurons and dendrites in an animal brain. In a biological brain, brain cells are connected in a network to perform operations, each brain cell has multiple dendrites that act as connectors which link to other brain cells by their terminals. When enough chemical energy is received by a cell through its dendrites, it fires a signal through its axon which will activate any dendrites connected to its terminal.

In artificial networks these complicated interactions are replaced with mathematical operations. The basic design of a network is to have input nodes connected to other nodes further in the network like a directed, weighted graph. The vertices in the graph act like brain cells while the edges act as dendrites connecting cells. The edges of the graph are weighted with a random unknown variable, by training the network the values for the weights of each edge can be evaluated to give the least amount of error. Each node in a network sums up its inputs and applies an activation function to the value. This new value becomes the output of the node and is passed on to any connected node.

In more recent times, the analogy to biological brains has diminished and is no longer a predominant guide in the field of deep learning. The main reason for this paradigm shift is simply because we do not know enough about how the human brain works to make any more assumptions about solving problems with this system. For further advancements in copying the human brain it would be necessary to monitor the activity of thousands of interconnected neurons simultaneously.

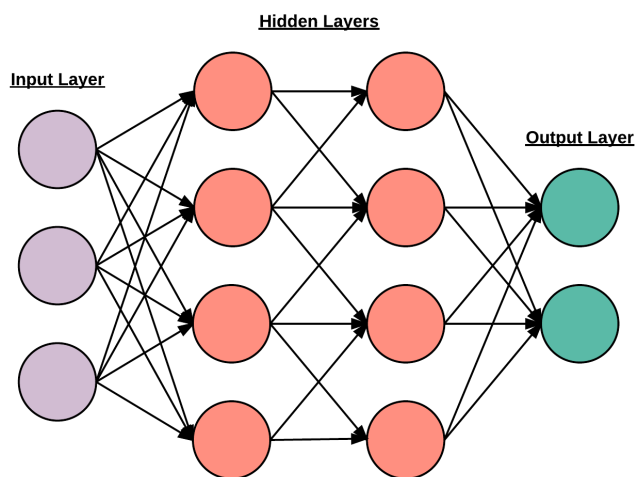


Figure 1: A simple network diagram

**Layout** The layout of networks is a continuing topic of research today, there are hundreds of different designs for specific use cases, but all these designs follow a basic outline. Networks consist of an input layer which is connected to several middle layers, which are connected to the output layer. These middle "hidden" layers, named because their hyperparameters are unknown can have different configurations, ranging from fully-connected to convolution-sparse layers[5].

**Weights** The weights in standard neural networks are floating point numbers that can be tuned to lower the output error of the network. When initialised a networks weights are assigned randomly from

a distribution. One problem arises with random assignments is that if a weights value is zero initially it can be difficult for it to be retuned, to avoid this a bias factor is added to the distribution to mitigate the issue[5].

**Activation Functions** The activation functions of a node defines the output given by a set of inputs. A non-linear activation function allows a network to compute non-trivial solutions to a problem using only a relatively small set of nodes. There are many different activation functions available to use, however most of these functions are similar enough that there is little need for different functions. Using our biological metaphor, the activation functions represents the rate of nerve firing in the brain[5].

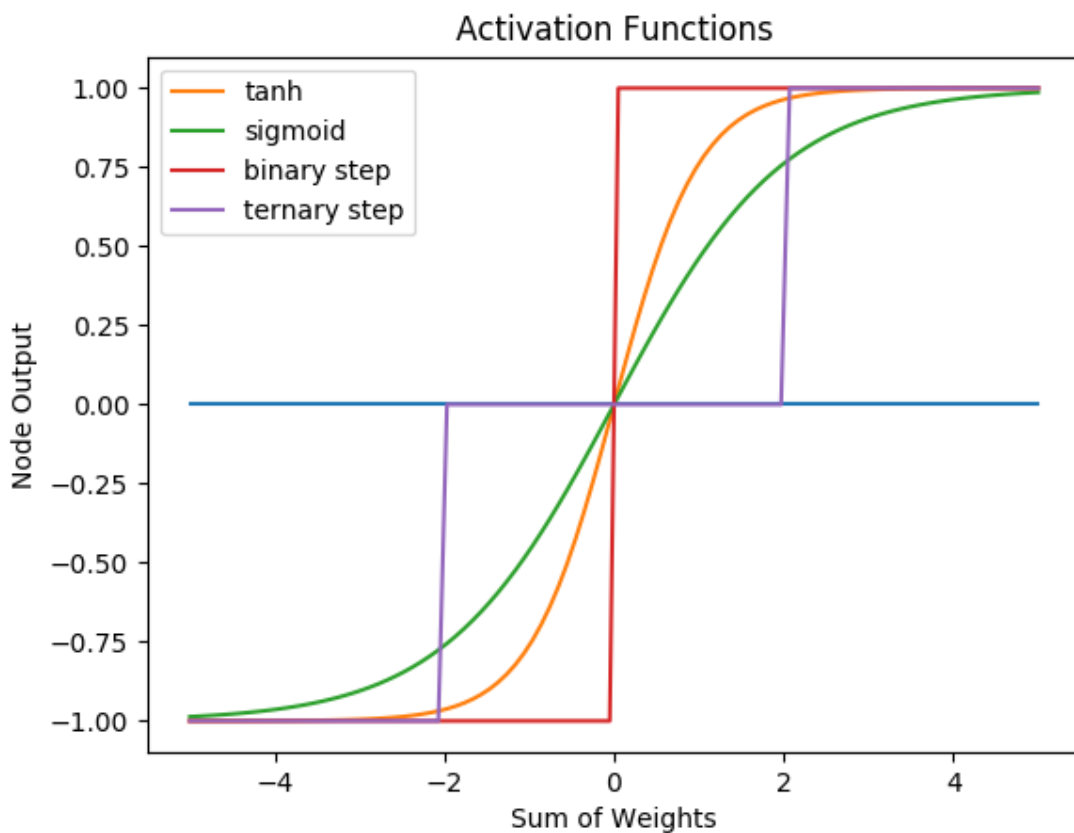


Figure 2: The shape of different types of activation functions.

Current investigations into optimising neural networks revolve around converting these non-linear functions into approximate piecewise linear functions that can mimic the output of these functions but at a faster rate with less accuracy.

#### 2.4.2 Backpropagation

Backpropagation is a method used in the training of neural networks to calculate and minimise the error that is propagated by the individual weights found across the network. The algorithm requires some sort of loss function to work, this is usually given by the mean squared error of a batch of outputs. The term backwards propagations comes from the fact that the loss function is calculated on the output node and the error is distributed back through the network using a technique similar to the chain rule in

calculus[9].

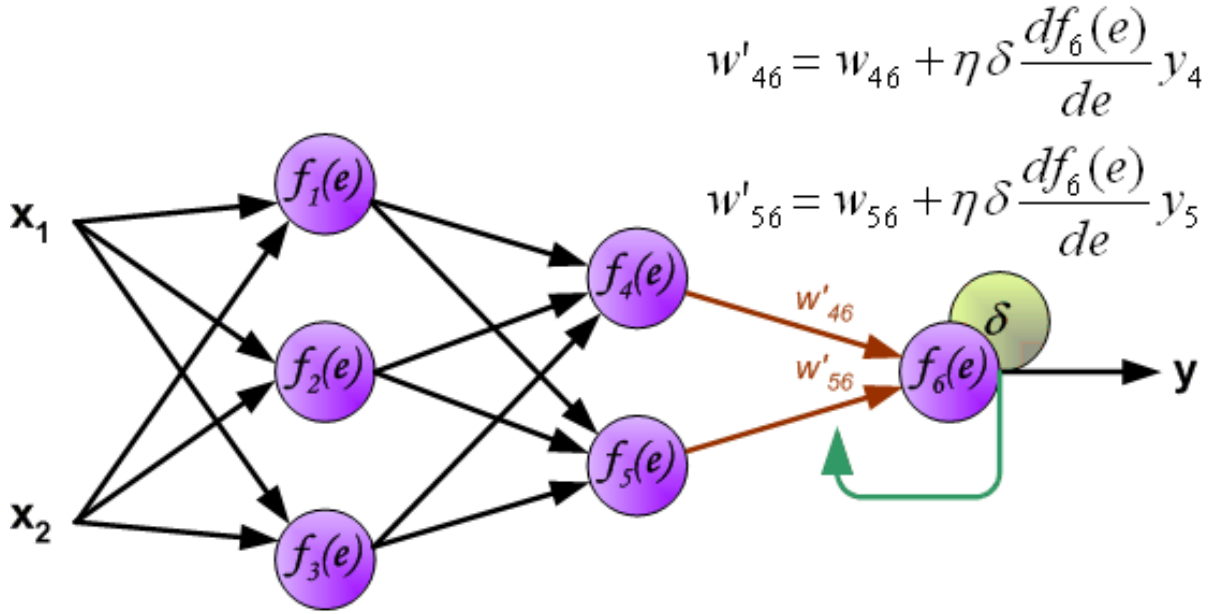


Figure 3: An example of back propagation.

Backpropagation requires a known, desired output for each input value, and is therefore known as a supervised learning algorithm. Figure 3 shows a graphical example of the first iteration of a backpropagation algorithm traversing a network. In this example the change added to each weight is calculated by adding the change in error related to each weight multiplied by the momentum (a known hyperparameter of the training algorithm), and the output value[9].

### 2.4.3 Overfitting

Overfitting is a problem encountered in machine learning and other statistical forms of mathematics where the model being analysed to fit on a dataset becomes too dependant on the data to make future accurate predictions. This is because if the model fits too perfectly to the data small errors in the distribution or other sources of noise can cause the model to become sensitive to false positives. This means the model will likely have a higher error rate on unseen data.

This can become a serious problem for neural networks and is a topic of current research. There are several solutions to overfitting, these include having a large dataset, having less parameters in a model, and input manipulation[9].

### 2.4.4 Quantising

Quantisation covers a lot of different techniques to store numbers and perform calculations in a more compact and efficient way than typical 32-bit floating point numbers. For the purposes of this report quantisation refers to the simplification of activation functions to piecewise linear functions, and the conversion of floating point weights to integer weights[16, 15].

## 2.5 Programming

### 2.5.1 CUDA

CUDA is a parallel computing platform based around Nvidia GPU chipsets. It provides a way for developers to create software that can efficiently use GPUs in computers for general purpose processing. The CUDA platform is a software layer that gives developers access to the GPU's internal virtual instruction set, allowing code to be executed in parallel at a much faster rate. CUDA is used for a variety of deep learning solutions as much of the maths needed for deep learning algorithms can be easily computed using the parallel processing power of a graphics card.

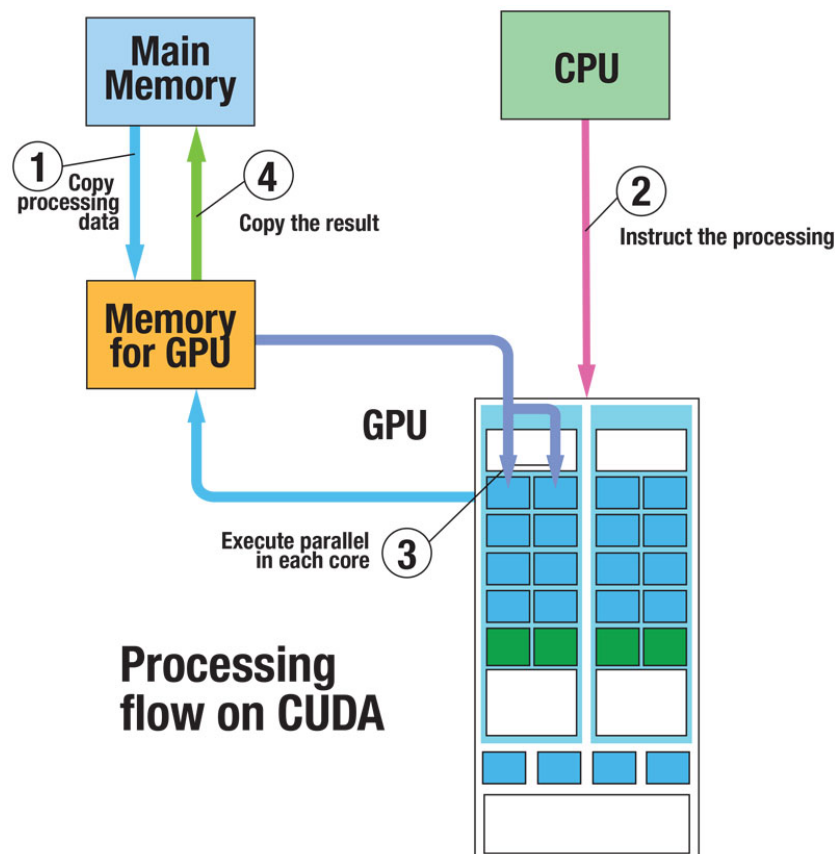


Figure 4: The operation flow of CUDA processing.

The flowchart in Figure 4 gives a basic explanation as to the steps taken when a CUDA compiled program is run on a target machine. In the first step the program memory is loaded into the GPU's memory. Second, the CPU sends the appropriate instructions to the GPU to start computation. Third, The GPU starts executing instructions and processes data, returning it to the GPU memory. Lastly, the finalised output is returned to the system memory.

### 2.5.2 Torch - Lua

Torch is a mathematical framework written in Lua for the specific use of processing scientific data. Torch can compile code to CUDA instructions and execute operations on both the GPU and CPU. Torch has a wide range of functions that can be used when developing deep learning algorithms. The main branch of Torch was written in Lua, a language specifically created to handle scientific data, however it is not

a commonly taught language any more, because of this an alternative version of the framework known as PyTorch was written from the ground up in Python to evolve the framework for a new language.

### 2.5.3 TensorFlow - Python

TensorFlow is a framework written in Python to handle the creation and manipulation of Tensors. In mathematics a Tensor is a geometric object that describes a relation between a scalar, vector, matrix, or another Tensor. In fact a Tensor can be thought of as an extension of a matrix in a higher dimension. The purpose of TensorFlow is to create an easy system to manipulate Tensors, this is done in the framework by creating a graph which describes what operations will be performed on each Tensor and how the data will flow through the graph - hence the name TensorFlow. After the graph has been defined in Python (or any supported high level language for that matter) the graph is then interpreted into a target language, either C or CUDA and run on the targets processing unit. Because of the ease of use, large developer community, and several other features including TensorBoard, many developers and academics have transitioned to using TensorFlow instead of competing frameworks[5].

## 2.6 Optimisation via Binary and Ternary Quantisation

As mentioned above the activations functions of a network are what allow networks to solve problems in a relatively small amount of space and time, however by changing how the activation functions work it is possible to increase the speed, and decrease the training time needed to create a working neural network. In this report we will investigate how changing the activation functions to piecewise linear steps modifies the behaviour and characteristics of several different types of networks.

### 2.6.1 Current Research

Optimisation via quantisation is currently an active field of research in machine learning and there are several efforts currently underway to increase the speed and accuracy of networks using similar techniques that are talked about in this report[5].

**Binary-Net** In 2016 a team of researchers in North America and Israel created a network known colloquially known as Binary-Net in this report that constrained the activation function outputs and weights to the values of -1 and 1. Their results showed that it is possible to obtain near state-of-the-art results using a binary network while only using a fraction of memory compared to previous networks that had the same results[2].

**XNOR-Net** XNOR-Net is another network that was published in 2016 by researchers from the University of Washington. This network expands the field of quantisation by adapting models like VGG, ResNet, and similar binary models so that most of their internal operations are bitwise functions. By converting specific operations such as convolution, and dot products to approximate binary equivalents XNOR-Net again was able to achieve near-state-of-the-art results equivalent to a full precision network while at the same time reducing the footprint needed to compute results by as much as a factor of 58[14].

**Ternary-Net** By expanding on the work of XNOR-Net and Binary-Net researches again improved the accuracy and compression of a network, this time in China. Academics at the Institute of Applied Math in Beijing were able to run tests on network designs that are similar to the ones mentioned in this report. In their research they created a series of models that used piecewise ternary step functions as the activation functions for convolution layers in their reproductions of VGG-19. Their ternary results showed improvements over their binary models, with increases in accuracy by approximately 5%[10].

## 3 Torch Implementation

### 3.1 Outline

During semester 1 the initial implementation of a solution was created using Torch. The reason for this was because the several algorithms that were to be tested and modified are written in Torch. By using Torch it was possible to save time in the beginning of the year because it meant that it was not required to know how the networks were implemented right away, the only code that needed to be modified was that of the activation functions. The advantage to using pre-build models was that there was already a module built to read in datasets and parse the results, so less time was also spent creating wrapper functions to make the models work.

While installation of Torch was relatively easy as it only required a rudimentary understanding of how the Linux operating system worked, several add-ons used for debugging purposes such as FBThrift and the Facebook Lua development library (FBLuaLib) needed extensive study in order to understand how to build and compile on the version of Linux being used (Arch Linux). After several weeks of troubleshooting and studying various resources around Linux, Torch, and Neural Networks the basic model using binary and ternary activation functions was created.

### 3.2 Development and Programming

The algorithm for implementing binary activations was already provided and is quite simple, in Lua all that is needed is to call the library function `Math.sign()` with a tensor as the argument and the function works as expected. To create a ternary step function that can have a variable width it is not possible to execute one operation, instead the fastest way to implement a function like this in Lua follows the algorithm provided below in Algorithm 1.

---

**Algorithm 1** Ternary Activation Algorithm

---

**Require:**  $0 < threshold < 1$

```
1: procedure GETACTIVATIONVALUE
2:   if  $input > threshold$  then
3:      $output \leftarrow 1$ 
4:   else
5:     if  $input < -threshold$  then
6:        $output \leftarrow -1$ 
7:     else
8:        $output \leftarrow 0$ 
   return  $output$ 
```

---

The ternary activation function works like a switch statement, if the input value is greater than a specific threshold value then the output is 1. If the input is below the negative of the threshold value then the output is -1. If it's neither the output is 0. Note for the function to work correctly the threshold value must be greater than 0, and less than 1 as this is where most non-linear activation functions converge, this can be seen more clearly in Figure 2. If the threshold is set improperly then the model will become less accurate than a binary network due to over and under fitting. Optimising the points at which the activation function steps would significantly increase the reliability of the model.

### 3.3 Challenges and Solutions

At this point there were several challenges in testing the newly created activation functions. For starters a sizeable dataset was needed to run adequate tests on the new model. The infamous ImageNet library was the perfect choice, however at a total size of 400GB downloading the archive proved challenging

in it's own right. To preform the download a script was created to run on a slave machine set up at the university, the download took 14 days. Another problem encountered was the file access speed. Storing images on a hard drive simply wasn't an acceptable method, as it took too long for the indexing service to catalogue all the required files. This problem was solved by using an solid state disk, which significantly improved the performance of the Torch network[5].

The implementation described here was a first step in testing and development, however, the models that were being modified had not been fully completed for certain scenarios. For example the modules included mentioned being able to use the CIFAR datasets to train the network but there was no code in the repository that supported this. This might not have been a problem in itself however since the only available dataset was so large (ImageNet has 1000 classes) it took too long to iteratively develop code and test solutions. For this reason it was decided at the end of semester 1 to transition to a more richer framework, TensorFlow.

## 4 TensorFlow Implementation

### 4.1 Outline

Moving over to TensorFlow had several advantages. Initially the learning curve for TensorFlow was very small as a lot of the work that had been done in the first semester could be transferred with minimal modifications. Additionally since TensorFlow was written in Python it was trivial to code in as I have a high level of proficiency in the language. The one issue with migrating to TensorFlow is that it is a fairly new framework, being open sourced in late 2015, and only reaching 1.0 status in early 2017. However, since the main development team is Google Brain, the community surrounding TensorFlow is rich and thriving, making it easier to troubleshoot problem and fix bugs.

As mentioned before TensorFlow interprets the instructions given and then creates a graph to operate on the input tensors, this graph is then compiled into machine and CUDA code and sent to the CPU and GPU. One problem that was encountered early on in development was that there was limited support for bitwise operations in the CUDA bindings. Several adjustments to the algorithm had to be made including building a test network that ran explicitly on the CPU. During this time it was noted that specific bitwise operations where being tested in a development version of TensorFlow which included needed functions like population count. To use these new features required compiling a development version of TensorFlow.

With these obstacles dealt with the only remaining task to complete was generating input data. Since

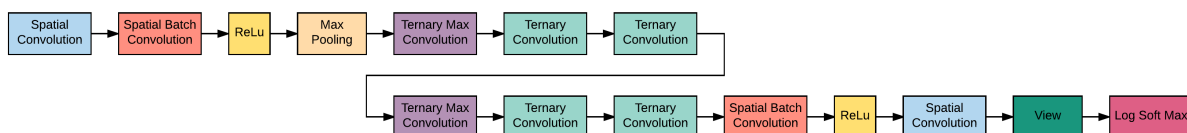


Figure 5: An example of the layout of a network in TensorFlow.

both Torch and TensorFlow binarized their data in different ways a new importer was needed. Not only that, there was no universal way to test different networks with different datasets and view the results easily on TensorFlow. To easily build new networks, train with any datasets, and view the results, a new module was created that handles all the creation and training functions that were needed to conduct extensive research, the new module was added to the repository pyNet.

## 4.2 Research and Results

With this new module it is possible to compare the run-times of several different models using the TensorBoard application. Using Torch the timings and accuracies of AlexNet[7] like models had already been measured. TensorFlow provided the opportunity to modify several other models that had already been converted to the framework, including VGG, Inception, and ResNet. It is now a simple matter of redefining the activation functions for specific layers to convert a network to a binary and ternary equivalent.

Overall by changing the activation functions the accuracy of the network did decrease by a similar amount seen in the Torch models. However, the speed at which the batched operations were processing was noticeably faster than the non-linear equivalent models. As well, the file sizes for the models decreased, this was mainly due not because of the changed activation function, but because the output of the new function was an integer instead of a floating point number, and since layers connected with these new functions would only receive integers there was no need to add weights to any of the connected layers..

Additionally this with this new framework it was possible to change the datatype of the weights of certain layers quite easily as well. This allowed the possibility of benchmarking the differences that can happen when the weights are binarized as well. This was done by using the population count function which can count the number of 1s in a binary tensor natively on the GPU. The resulting models that were trained using this new method had a consistently smaller file size that resulted from changing the weights from floating point type to integer type.

## 4.3 Improvements and Adaptations

The new models show a significant improvement in terms of speed and training times, because of the selected threshold for the ternary functions, this models had a lower accuracy than was expected, apart from trial and error there is no easy way to change the threshold. This can hopefully be iterated on in the future and improved.

While I achieved all the objectives that were outlined in the progress report and requirements analysis there is still room for improvement in the TensorFlow implementation created. For starters being while it is possible to add any image based dataset into the trainer, there is still no way to add other types of data, this could be added in a future revision.

While most of the training that was conducted for these models was run on dual Nvidia 970 chipsets, the models did work fairly well on smaller machines such as my laptop that contains a Nvidia 870m chipset on-board. Using the OpenCV library it was possible to create a script that could evaluate live video feed into the model, without any optimisation the fastest recorded framerate was 21fps, which is about the same as the file I/O that was recorded earlier in training. Finally in the last few weeks of the semester I was able to port a CPU version on TensorFlow onto a raspberry Pi and run a rudimentary network on the quad core processor. The network averaged a batch speed of approximately 10 images in 20 seconds, but this demonstration shows that reducing the overall size and computation power required by these models can be advantageous for many different machines.

# 5 Results and Analysis

## 5.1 Network Error Rates

In analysing the error rates of networks it is possible to understand how quickly a type of network can change it's hyperparameters to increase its training accuracy. The lower the error the better the prefor-

mace of a network. The results that will be interpreted come from an optimised implementation of alexnet running in Torch.

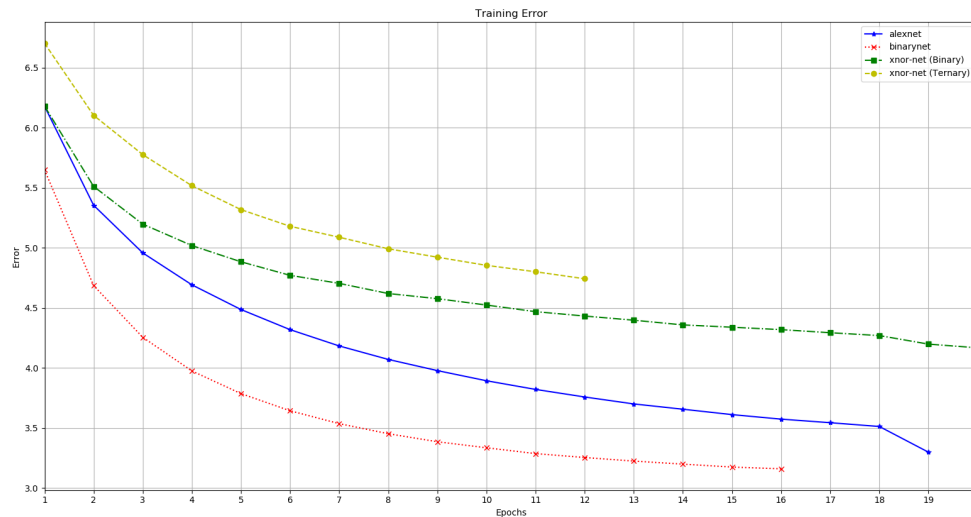


Figure 6: Comparison of training errors for optimised and regular networks.

The training data seen in Figure 6 shows how the error in all the networks decreases over time. This shows that the implementation of the networks is successfully working and that eventually the networks will converge individually at a minimum error by following the backpropagation optimisation algorithm.

The data in Figure 7 shows that the training algorithm is not overfitting the data as the error in testing is also decreasing at an equivalent rate. It can be seen in both figures that the binary and ternary functions are decreasing at the same rate. This is because both models work on the same underlying principle and that the algorithms are implemented in a similar way.

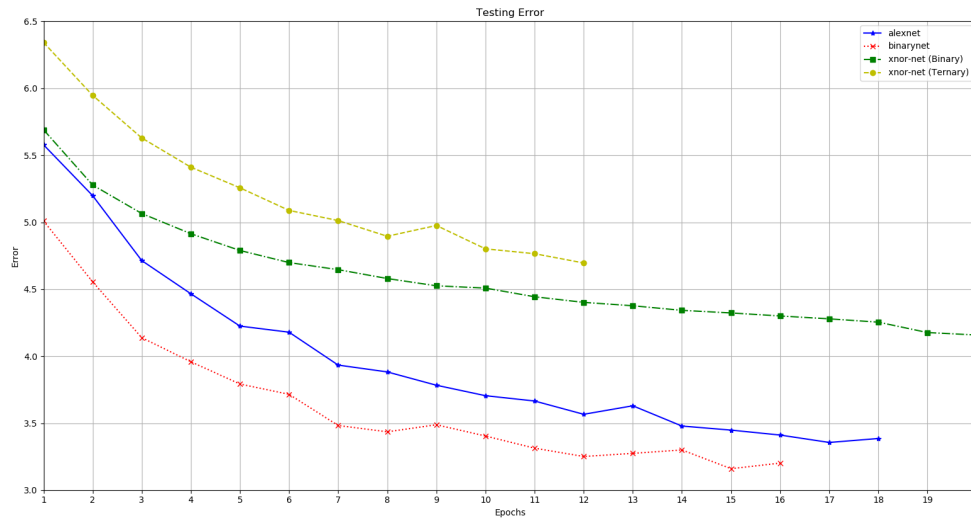


Figure 7: Comparison of testing errors for optimised and regular networks.

## 5.2 Network Training and Testing Accuracies

### 5.2.1 Top-1 Accuracies

The Top-1 accuracies provide an insight as to how often the model correctly selects the right classification in the first try. The progress of binary, and ternary networks can be viewed in both Figures 8 and 9. One interesting observation that can be made in all the training figures is that ternary net is not as accurate as binary net, this is somewhat counter-intuitive because the whole idea of adding an extra step in the activation function is to add more precision, how could adding more precision be making the network less accurate?

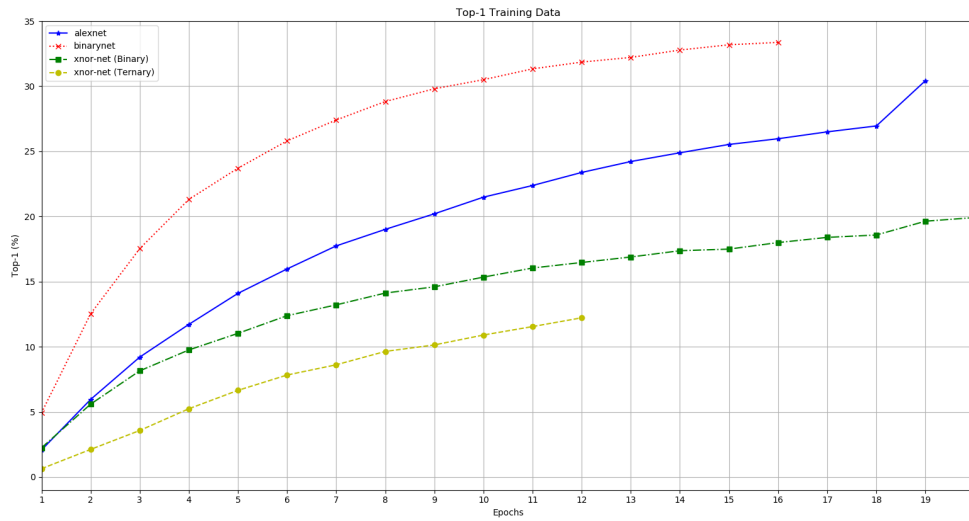


Figure 8: Comparison of Top-1 testing accuracies for optimised and regular networks.

The answer can be found when changing the threshold value for the step size in ternary net. In initial testing the activation threshold was set to 1, this means that any number in the range of  $-1 < x < 1$  was set to zero. By looking at the comparison of activation functions in Figure 2 it clearly shows that in that range is where the most changing between levels happens, or putting it another way - where the gradient of the functions has a highest magnitude (This is incredibly important for backpropagation). Because the step in binary net happens when  $x = 0$  there is such a small gap that none of the output values can be zero, as expected.

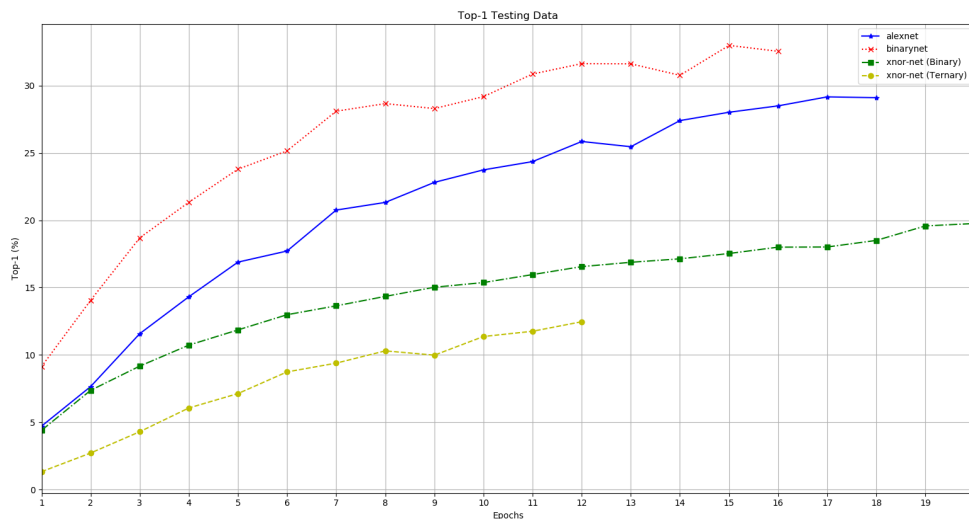


Figure 9: Comparison of Top-1 training accuracies for optimised and regular networks.

What's happening here is that too many values are being zeroed in the convolution layers, this means

that too much data is being lost by the quantisation effects. A simple way to fix this is to traverse the range of reasonable threshold numbers for each dataset and optimise each network per dataset. Essentially choose a threshold value smaller than 1.

### 5.2.2 Top-5 Accuracies

The Top-5 accuracies show how often a model correctly chooses the classification in the first 5 tries. Again in Figure 10 and 11 the accuracy of the networks can be observed and the underfitting caused by the thresholding is also visible.

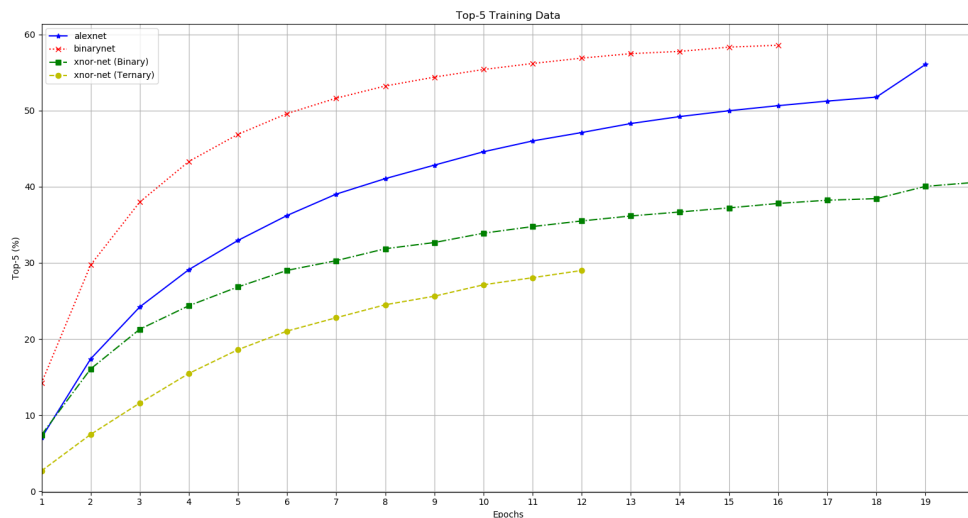


Figure 10: Comparison of Top-5 testing accuracies for optimised and regular networks.

Another property of the modified networks is the difference in the rate of change of accuracy. In Figures 6, 7, 8, 9, 10, and 11 it can be seen that Binary and Ternary Net are slower to change and end up lagging behind the original network implementations. This was another anomaly that was not well understood when it was first recorded, with more research it is now better understood why this happened. Both types of networks calculate error by backpropagation using gradient decent. Gradient decent works by changing the weights of a layer by finding out how much error each weight adds to the total. To achieve this the algorithm needs a function that can calculate the derivative of the activation function, an example of this can be seen in Figure 3. Because in Binary and Ternary net the activation function is piecewise there is no continuous derivative for these functions. To resolve this the derivative of the step functions was left unchanged for the non-linear counterparts. While this allowed the algorithm to work the inaccuracies between the step functions and their pseudo derivatives meant that small errors grew more rapidly in later iterations, hence why the algorithms plateau at a lower level.

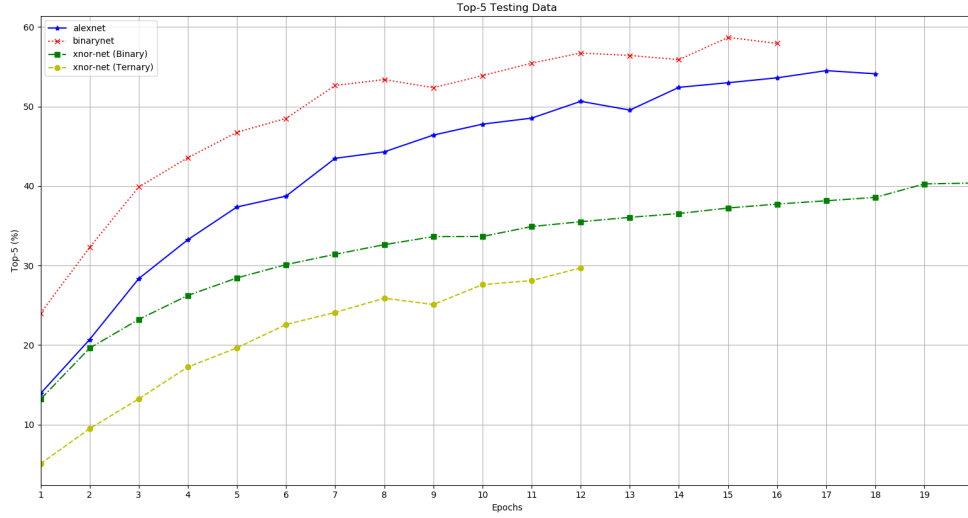


Figure 11: Comparison of Top-5 training accuracies for optimised and regular networks.

Another solution to this problem could be to change how the derivatives are calculated, in an extension to this I tested several different derivatives for the activation functions, most alternatives, such as changing the derivative to be piecewise did not have the anticipated results, however by condensing the derivative so that the rate of change increased closer to the steps did show some improvement.

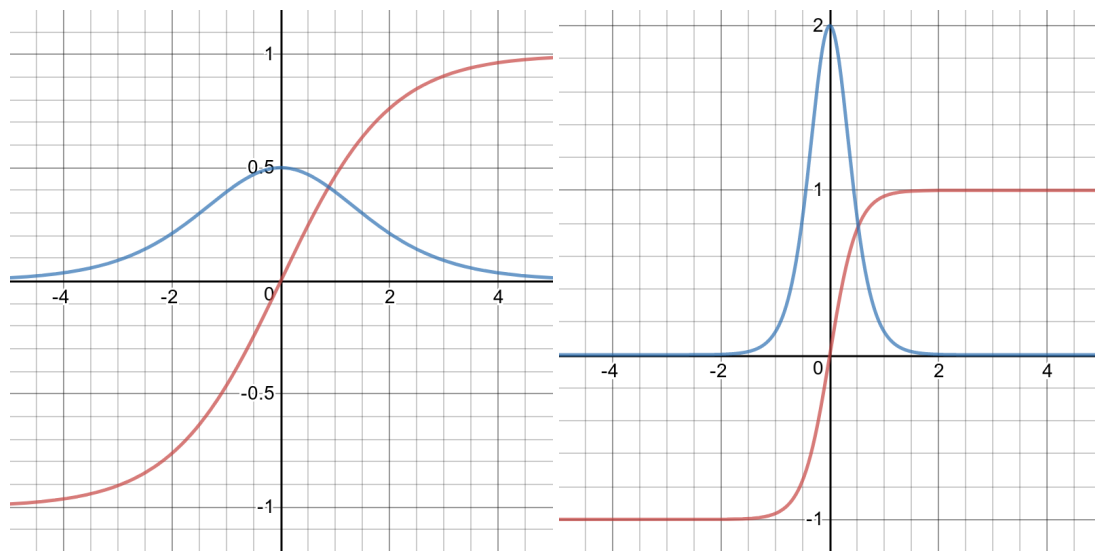
$$f(x) = \frac{2e^x}{e^x + 1} - 1 \quad (3a)$$

$$f'(x) = \frac{2e^x}{(e^x + 1)^2} - 1 \quad (3b)$$

$$f(x) = \frac{e^{4x}}{e^{4x} + 1} - 1 \quad (3c)$$

$$f'(x) = \frac{8e^{4x}}{(e^{4x} + 1)^2} - 1 \quad (3d)$$

The equations above as well as the plots in Figures 12 show how this would be achieved when taking into account a binary activation function. For ternary the activation function would need to still either be piecewise or composed of more complicated numerical components.



(a) A standard sigmoid(red) and it's derivative(blue). (b) A squished sigmoid(red) and its derivative(blue).

Figure 12: A comparison of activation functions and their derivatives.

### 5.3 Batch and Filesize Comparisons

Another component that was investigated in this project was how the file size and average batch time changed when the activation functions and weights were modified. In initial testing in Torch and TensorFlow the batch size was seen to reflect how fast a batch could be calculated. The larger the batch the longer it would take both frameworks to process and calculate the values. In the example in Figure 13 a batch size of 20 was used due to memory constraints.

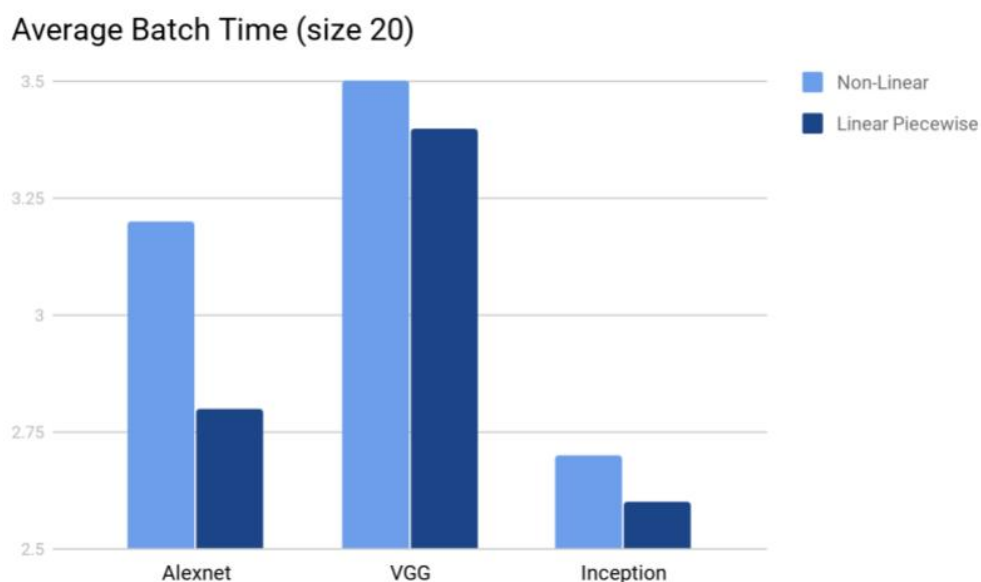


Figure 13: Comparison of batch times for optimised and regular networks.

What was found is that with modified activation functions with input weights being of type `uint8` the speed of the batch processing dropped by varying amounts based on the different networks used. Alexnet showed the most impressive speed up as the model was able to process batches nearly 1 whole second faster.

When looking at the file sizes of the networks there was considerable improvement here as well. The most significant improvement occurred with VGG as approximately 100MB of storage space was saved, this was a saving of approximately 20%.

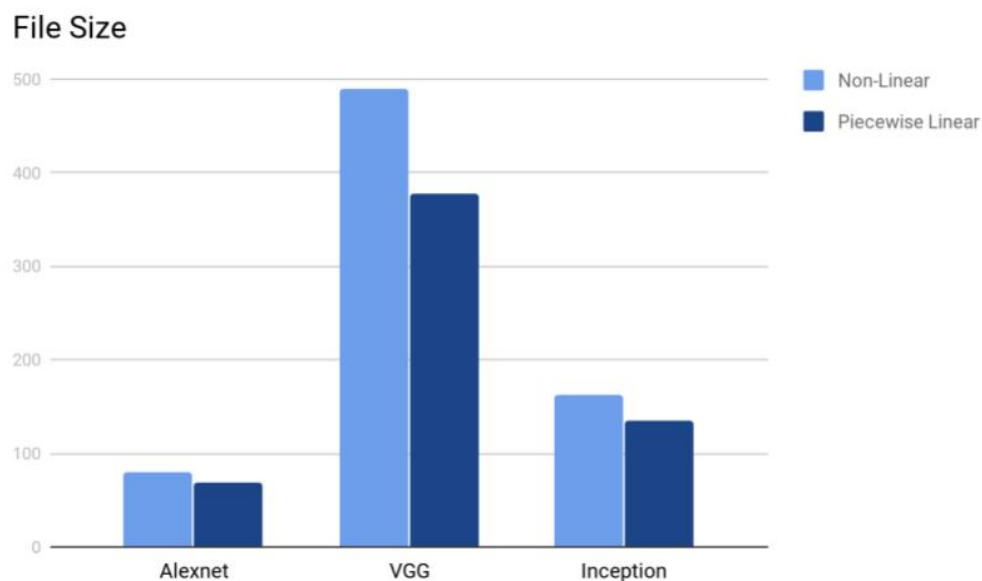


Figure 14: Comparison of file sizes for optimised and regular networks.

Figure 14 shows the savings in file sizes. The reason for such significant savings come from how the numbers are stored in the model. Originally all the numbers that has to be stored were 32 or 64 bit floating point numbers, when these weights were converted to 8 their sizes reduced by a factor of 4 and 8 respectively.

## 5.4 Analysis

The results of these tests show that quantising any type of network can lead to significant performance improvements - if done correctly. Initially a lot of the testing for this report was done on a trial and error basis. Building off these results it should be possible to revisit some of these configurations in the future and improve and iterate on the decisions made in this report.

## 6 Conclusion

In this project, the behaviour of optimised and non-optimised neural networks was evaluated. This was done by developing and testing several networks in both the Torch and TensorFlow frameworks, with their results combined and analysed to determine how changing the activation functions of convolution layers varies the speed, accuracy, and training time of these networks. The results show that by converting the activation functions from non-linear to linear piecewise functions improves the speed at which a network can be trained and evaluated at because each individual batch takes less time to be computed. The analysis also showed that this improvement shrunk filesizes by as much as 20% meaning that the models could be more easily transferred and shared to smaller devices. Furthermore the decrease in accuracy and increase in error is possible to minimise by changing some parameters that were used in making this report, such as how the gradient of functions is calculated, and what threshold value should be used for the Ternary step activation function.

The module created for this project called "pyNet" will be available on GitHub and should provide a starting point for further investigations into the area of network quantisation. The pyNet can module can be used to train any neural network that supports the output format specified in the library, it also makes it possible to create a training set from any hierarchical file system and run these datasets on any supported network. This will allow future researches to quickly iterate on past work and improve performance.

Further areas on improvements for this project lie in how some of the calculations for optimisations are made. If more bitwise operations were supported in Torch and TensorFlow on CUDA devices then the performance of networks could again be increased. In the analysis section it is mentioned that if the quantisation of networks is done correctly then the accuracy of the system should remain state-of-the-art, however, in the testing done this was not fully achieved and the accuracy of the networks remained 5%-10% below what other researchers have reported, this could be addressed with more attention in a later endeavour.

With the mobile and IoT sectors of the market continuing to expand and with companies like Intel[12] and Apple focusing on releasing more powerful chipsets aimed at machine learning applications, the future of optimisation relating to neural networks will be an important topic that will require up to date tools and theories to satisfy demand.

## References

- [1] Katherine Bourzac. <https://spectrum.ieee.org/computing/embedded-systems/bringing-big-neural-networks-to-selfdriving-cars-smartphones-and-drones>, 2017.
- [2] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016.
- [3] W. Erwin Diewert. cost functions. In Steven N. Durlauf and Lawrence E. Blume, editors, *The New Palgrave Dictionary of Economics*. Palgrave Macmillan, Basingstoke, 2008.
- [4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [5] Aurélien Grun. *Hands-On Machine Learning with Scikit-Learn and Tensorflow*. O'Reilly Media Inc., 2017.
- [6] Diederich Hinrichsen. *Mathematical systems theory I : modelling, state space analysis, stability and robustness*. Springer, Berlin New York, 2005.
- [7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.
- [8] Elizabeth Landau. <https://www.jpl.nasa.gov/news/news.php?feature=4457>, 2017.
- [9] Yann Lecun, L Bottou, Genevieve Orr, and Klaus-Robert Müller. Efficient backprop. *Research Gate*, 1524, 07 1998.
- [10] Fengfu Li and Bin Liu. Ternary weight networks. *CoRR*, abs/1605.04711, 2016.
- [11] John Mannes. Google's tensorflow lite brings machine learning to android devices. <https://techcrunch.com/2017/05/17/googles-tensorflow-lite-brings-machine-learning-to-android-devices/>.
- [12] Dr. Michael Mayberry. <https://newsroom.intel.com/editorials/intels-new-self-learning-chip-promises-accelerate-artificial-intelligence/>, 2017.
- [13] David A. Medler. A brief history of connectionism. *Neural Computing Surveys*, 1, 1998.
- [14] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *CoRR*, abs/1603.05279, 2016.
- [15] A. J. F. van Rooij. *Neural network training using genetic algorithms*. World Scientific, 1998.
- [16] Claudio Turchetti. *Stochastic models of neural networks*. IOS Press,, 2004.