



University of Glasgow | School of
Computing Science

Implementation of a Binarized Convolutional Neural Network in Fortran

Adrian Borg

School of Computing Science

Sir Alwyn Williams Building

University of Glasgow

G12 8RZ

A dissertation presented in part fulfilment of the requirements
of the Degree of Master of Science at the University of Glasgow

6th September 2019

The research work disclosed in this publication is partially funded by the
Endeavour Scholarship Scheme (Malta). Scholarships are part-financed
by the European Union - European Social Fund (ESF) -
Operational Programme II – Cohesion Policy 2014-2020

“Investing in human capital to create more opportunities and promote the well-being of society”.



European Union – European Structural and Investment Funds
Operational Programme II – Cohesion Policy 2014-2020
*“Investing in human capital to create more opportunities
and promote the well-being of society”*
Scholarships are part financed by the European Union -
European Social Funds (ESF)
Co-financing rate: 80% EU Funds;20% National Funds



Abstract

As technology has advanced and processors have become more powerful, the field of AI has flourished, with a significant interest in machine learning and neural networks. Recent research has been concerned with reducing memory accesses and arithmetic operations by binarizing the network parameters, creating a Binarized Neural Network (BNN). The goal of these research efforts is to increase performance efficiency of neural networks for FPGAs and mobile devices to be capable of processing them offline, on the device itself.


This project develops a framework in Fortran of efficient and convenient code blocks to construct neural networks. In order to test efficiency and accuracy, the CNV network (a BNN) has been implemented and tested using the cifar10 dataset, and was shown to achieve a good timed efficiency of 0.036s per inference but an unexpected accuracy result of 50.6%, speculatively attributed to using weights that were not fully trained. The result of the project gives a solid foundation from which to further develop other network blocks and construct other networks.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic form.

Name: Adrian Borg

Signature:

A handwritten signature in black ink, appearing to be 'AB' followed by a stylized flourish.

Acknowledgements

I would like to thank Dr. Wim Vanderbauwhede being my supervisor on this project and providing useful insight and guidance throughout.

I would also like to thank MEDE and the employees at the Endeavour Scholarship Scheme (Malta) office for help with the Endeavour Scholarship.

Finally I would like to thank my family who have supported me throughout my upbringing and education.

Contents

Chapter 1	Introduction.....	1
Chapter 2	Analysis	2
2.1	Background.....	2
2.1.1	Neural Networks.....	3
2.1.2	Network Features.....	4
2.1.2.1	Activation Functions	4
2.1.2.2	Fully Connected Layers.....	5
2.1.2.3	Batch Normalization Layers	6
2.1.2.4	Convolutional Neural Networks and Convolutional Layers	6
2.1.2.5	Max Pooling Layers.....	8
2.1.3	Forward Pass.....	8
2.1.4	Backward Pass.....	8
2.1.5	Binarization.....	9
2.2	Objectives.....	10
Chapter 3	Design and Implementation	11
3.1	Language Choice - Fortran	11
3.2	Network Choice - The CNV Network.....	12
3.3	Subroutines.....	13
3.3.1	Nesting of Subroutines	13
3.3.2	Subroutines for Each Type of Layer.....	14
3.3.3	Limitation - Popcount.....	14
3.4	Developing Automated Testing Subroutines	15
3.5	File System Layout.....	15
Chapter 4	Testing and Evaluation	16
4.1	Unit Testing.....	16
4.2	Testing with Cifar10 Images	16
4.2.1	Accuracy Test Results	17
4.2.2	Performance Test Results	17
4.3	Evaluation.....	18
4.3.1	Accuracy Discussion.....	18
4.3.2	Performance Discussion	18
Chapter 5	Conclusion.....	19
5.1	Future Work.....	19
Chapter 6	References	21
Appendix A	Convolutional Layer Code	1
Appendix B	Pseudocode for Major Subroutines.....	2

Chapter 1 Introduction

With the rapid development of technology in recent history, machine learning (ML) has also seen an increase in performance and application [1]. These range from recommender systems, to autonomous driving, speech transcription and medicine [2] [3]. Neural networks (NNs) are one type of ML which has been inspired by the human brain, using math and statistics in order to create a network which learns how to recognize patterns [4]. Similar to a brain, a NN is composed of several neurons connected by synapses, which define the relations between the neurons [5]. The rise in popularity of NNs has brought about the development of several different software frameworks, such as; Torch [6], Tensorflow [7], Caffe [8] and Theano [9], which help to create, customise and train the networks.

When attempting to classify an image or decipher a sound clip, there may be many variations for the input that lead to the same output (e.g. different coloured cats, furrrier cats, partially covered cats, different sized cats, etc.). Separating these variations from what is of importance in the data can prove to be difficult. Deep learning is a subset of ML which solves this problem by introducing several layers of representations in order to extract the simpler features in an image (e.g. edges, which the next layer can use to define corners, corners would then define shapes, and shapes can define features of the classification object, etc.) [4]. The benefit of deep learning is that these layers are not tediously designed by humans but use data to learn the important patterns and features through learning techniques.

Convolutional neural networks (CNN) are a type of NN which are designed for the processing of data in dimensions higher than two [2]. This is especially useful for processing of images as they generally have multiple channels for colours (e.g. red, green and blue). Modern CNNs require a large number (over 1 million) floating point parameters, which in turn leads to billions of operations to process an image [10]. These are therefore mostly trained and run on GPUs, which are fast but require a lot of power. Because of this, lower powered devices, such as mobile devices, lack the capability of running them [11]. Binarized neural networks (BNNs) have been created in order to solve this problem. These constrain the parameters and signals between layers to -1 or 1, thereby reducing the memory required and complexity of operations involved [12]. In the long term, this works towards allowing BNNs to be run on offline mobile devices or FPGAs, without needing any connection to a more powerful server.

This project aims to implement a BNN, based on the CNV network [10] in Fortran, which uses the cifar10 dataset [13]. This involved developing an efficient framework within Fortran to create layers and perform a forward pass to classify an image.

The remainder of this report is divided into the following sections: Chapter 2 describes some background on NNs, CNNs, BNNs as well as further discussion of the aims and objectives. Chapter 3 discusses the main design and implementation choices taken throughout the project. Chapter 4 shows the evaluation of the network produced, while Chapter 5 concludes the report.

Chapter 2 Analysis

This chapter will highlight relevant background knowledge in machine learning (ML), neural networks (NNs) and binarization, as well as define the aims and objectives for the work.

2.1 Background

Artificial intelligence (AI) was initially developed in order to perform functions that are easy for computers but more challenging for people. These problems were defined using a series of formal rules that the computer must follow. Nowadays AI is sought out to automate more mundane tasks such as identifying images or understanding speech. While these tasks are intuitive for a human, they are less straightforward for a computer to solve. ML is one type of AI which attempts to overcome this issue by learning and gaining knowledge through experience. Using this knowledge, the ML system can understand concepts within the application it was trained for and perform its task [4].

Conventional ML systems tended to be challenging to set up for reading and processing raw data, requiring experienced engineers and careful design in order to extract important features and use these to classify an input. Representation learning is another ML technique which allows the machine to automatically detect these features. Neural Networks (NNs) fall under the representation learning family of techniques [2].

As explained in Chapter 1, a significant obstacle within ML and representation learning is separating variations of an input from the important features that it is looking for (e.g. separating a person's accent from what they are saying). Separating the important features out in one step is a task as difficult as the original task [4]. Deep learning (DL) is a technique which solves this problem by introducing several layers of representation which gradually highlight important features going through each layer. Each layer of representation defines concepts which are a bit more abstract than the previous layer, and with enough layers complex tasks can be performed (e.g. classifying an image as a cat) [2]. So, to summarise, DL is a subset of representation learning, which is a subset of ML, which is under the superset of AI. The main advantage of using DL is that the rules which the system should follow are not explicitly defined by a human but are learnt through the training process, contrary to conventional/shallow architectures which would need specific definition of feature extractors [4] [14].

There are several forms of machine learning, the most common of which are; supervised, unsupervised and reinforcement learning (these are not exclusive of representational learning i.e. a representational learning system could also be a supervised or unsupervised learning system). The main focus of this project is classification with supervised learning, however it is worth mentioning the other two. In supervised learning, the system is trained using labelled data with known expected results. This data is used to optimise an objective function which will allow the system to predict an output for unseen inputs [15]. This technique's most common applications are classification and regression. Unsupervised learning involves taking some unstructured input data and finding

some structure within the data, as such in cluster analysis [14]. Reinforcement learning involves having the computer perform actions which would give maximum reward, suitable for example to teach a computer how to play Go [16].

2.1.1 Neural Networks

NNs are composed of an input layer, one or many hidden layers and an output layer. Each layer consists of neurons which are then connected to neurons in other layers through connections called synapses. Synapses have a weight which describe the strength of the connection between the two nodes. There are a few types of NNs, namely feed forward NNs, recurrent NNs and recursive NNs [5]. The most widely used are the feed forward NNs as they currently have the most powerful and well researched learning algorithms [17].

This report will be concerned with feed forward NNs, where the outputs of one layer are used as the inputs for the following layer (outputs from a layer are never fed backwards to previous layers). An example of a simple feed forward NN is shown in Figure 1. In practise, each layer in a NN can have an arbitrary number of neurons, as well as an arbitrary number of hidden layers (but at least one). In the context of DL, as described in 2.1, the first hidden layer would identify simple abstract concepts (e.g. vertical edges), and each layer after that would combine several of the previous layers' concepts to create more abstract ones. Therefore, having more neurons and layers may allow the network to perform more complex tasks, however this would also require more computations and make the network require more resources to operate at the same speed.

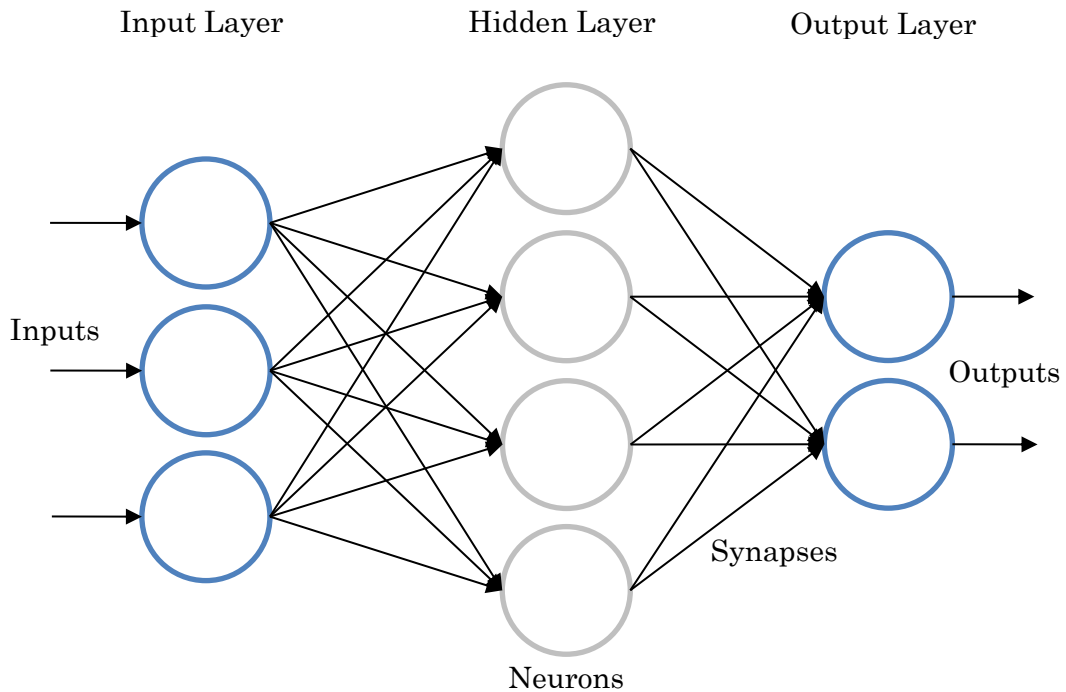


Figure 1. Illustrated example of a simple feed forward neural network architecture with one hidden layer. The input, hidden and output layer have three, four and two neurons respectively. Each neuron in one layer is connected to all the neurons in the next.

Hidden layers are usually a composite of simpler functions which perform operations on the input to a neuron in series. The relevant functions and their uses are described below in 2.1.2. Additionally, NNs are generally run in two distinct modes, the first is a training mode in which the network recursively performs a forward pass through each layer followed by a backward pass to iteratively optimise the parameters of each layer. The second mode is the inference mode where the NN deduces a result after one forward pass in order to obtain an output, using the optimised parameters calculated in the training mode.

2.1.2 Network Features

Nodes within a neural network are where the calculations are performed. Several different types exist and choosing the right configuration will depend on the application at hand. Below is a description of the main components a network node and where they fit in.

2.1.2.1 Activation Functions

Activation functions are used to manipulate the outputs of layers within a neural network. So, in general these are used after the main node calculation is performed. Two issues that activation functions aim to help with when training and using a NN are the vanishing gradient and exploding gradient problems. In training of multi-layered architectures, consecutive multiplication of small derivative terms leads the values to tend towards 0, therefore ignoring the term. This is the vanishing gradient problem. Similarly, consecutive multiplications of large derivative terms lead values to increase exponentially, which can lead to an unstable network. Both issues could result in the network not being able to continue learning from the training data. Activation functions aim to solve this problem by maintaining the outputs within specific limits [18].

Several activation functions exist, these include the Sigmoid function, the hyperbolic tangent (tanh) function, the softmax function and the rectified linear unit (ReLU) function which each also have some variations. These functions are all also non-linear which allow for more complex computations (since linear activation functions do not compound well in multi-layered architectures since their derivatives are constants [19]).

The sigmoid and tanh functions are similar, with the tanh being preferred over the sigmoid due to its better performance during training. Both these functions produce values bound between 2 values; $[0, 1]$ for sigmoid and $[-1, 1]$ for the tanh, their mathematical expressions are shown in (1) and (2) for sigmoid and tanh respectively, and can also be visualised in Figure 2. Both functions however suffer from the vanishing gradient problem [18].

The ReLU function is an improvement over the sigmoid and tanh functions. The ReLU function gives better performance due to the function not needing any divisions or exponentials as can be seen in its expression in (3) [18].

Finally, the softmax function is generally used on the output layer of a network in order to give the probabilities of the predictions for the network, this is because it produces outputs in between 0 and 1, with all outputs summing up to 1. The expression for this function can be seen in (4) [18].

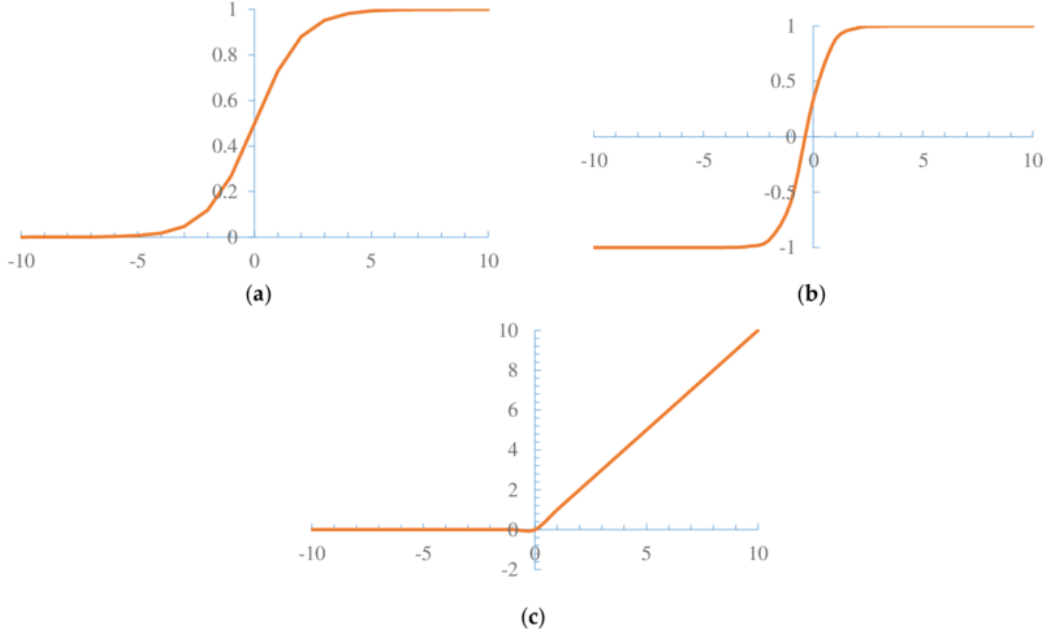


Figure 2. Activation functions of the a) Sigmoid function, b) Tanh function and c) ReLU function [20].

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

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

$$f(x) = \max(0, x) \quad (3)$$

$$f(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \quad (4)$$

2.1.2.2 Fully Connected Layers

When mentioning layers within a neural network, fully connected layers are the default type which are being referred to. With fully connected layers, each node is connected to all nodes of a previous layer by a synapse. The strength of that connection is quantified by a weight. Outputs of each neuron is calculated from the weighted sum of the neurons connected to it [1]. This can be expressed mathematically as in (5)

$$y_{ij} = \sigma \left(\sum_n w_{n(j-1)} x_{n(j-1)} + \beta_{ij} \right) = \sigma(w_{(j-1)} \cdot x_{(j-1)} + \beta_{ij}) \quad (5)$$

Where y_{ij} is the output of neuron i in layer j , σ is the activation function, $w_{n(j-1)}$ is the weight of the synapse from neuron n in layer $j - 1$, $x_{n(j-1)}$ is the value of neuron n in layer $j - 1$ and β_{ij} is the bias of neuron i in layer j . Converting this to a layer operation, it can be represented as in (6).

$$y_j = \sigma(W_{(j-1)} \cdot X_{(j-1)} + \beta_j) \quad (6)$$

Where y_j is the output of layer j , $W_{(j-1)}$ is the weight values of synapses from layer $j - 1$, $X_{(j-1)}$ are the values of the neurons in layer $j - 1$ and β_j are the bias values [1]. The weights and biases are trained parameters and change throughout the training process.

2.1.2.3 Batch Normalization Layers

When training a network, the inputs to each layer will have differing distributions since the parameters of the layers are constantly changing. This network will therefore need to adapt to these new distributions, slowing down the training. By normalizing layer inputs, batch normalization aims to minimise this change in distribution, and speeding up training in the process. Batch norm layers also has a regularisation effect on the network [21]; it protects the model to some degree from overfitting, where the network performs well using the training data but generalises poorly and underperforms when testing with unseen data [22]. A batch norm layer can be expressed mathematically as in (7) [23] [24].

$$y = \frac{x - \mu}{i} \gamma + \beta \quad (7)$$

Where y is the output, x is the input, μ is the mean, i is the square root of the variance ($\sqrt{\sigma^2}$), and γ and β are two trainable parameters.

2.1.2.4 Convolutional Neural Networks and Convolutional Layers

Convolutional neural networks (CNNs) introduce a new layer to substitute some of the fully connected layers, a convolutional layer. The idea behind the creation of these networks is to further improve distortion and shift invariance (correctly identifying features/concepts regardless of position in the input and noise) [25].

An important, and distinctive feature of convolutional layers over fully connected layers is the filter. This has a defined size (e.g. 3x3) and is analogous to the weights of a fully connected layer, as such all values within the filter are trainable. Each element of a layer takes all elements of the previous layer within the receptive field as inputs. The output matrices of performing this operation over the whole input matrix is called a feature map.

A CNN could have multiple filters in the same layer, each filter determining different features. Each filter creates its own filter map, so if the input to a convolutional layer is two dimensional then the output could have three dimensions. These filters can be used to detect features within the receptive field at early layers (e.g. edges) and subsequent layers will use feature maps to detect features in the same region (the receptive field) and find more complex features. For example, if the first layer has a filter detecting edges, then the second layer could have a filter which detects edges in the same locality, identifying a corner. A filter moves along the input with a certain offset for each output element. This offset is called the stride and in combination with the filter size, defines the size of the output. A layer with input having dimensions $n \times m$, and p number of filters of size $f_n \times f_m$ and stride s can have output dimensions as calculated in (8).

$$\text{dimensions} = \frac{n - f_n + 1}{s} \times \frac{m - f_m + 1}{s} \times p \quad (8)$$

It is important to note also that an input to a CNN can also have more dimensions, however the filters would have the same number of dimensions. In a 2D convolution ('striding' over the input in two dimensions) the filter would have the same size in the extra dimensions (a 2D convolutional layer taking input of size $l \times m \times n \times o$, striding over the l and m dimensions will have a filter of size

$f_l \times f_m \times n \times o$, and in this case, a single filter would produce one 2D feature map).

The operation of ‘striding’ the filter over the input, elementally multiplying the receptive field with the filter, and summing them to produce a single element in the output is called a convolution, this can be visualised in Figure 3. Additionally, a trainable bias term is also added to each element as with the fully connected layers.

Using convolutional layers over fully connected layers comes with a few advantages. Fully connected layers require quite a bit more weight parameters, since each element requires a weight for every element in the previous layer, while in a CNN each element only requires the weights in the filters which are used over the whole input. Additionally, since the CNN works with filters that are translated over the input, the network would be more resilient to shifts and invariance within the input. Finally, the CNN also considers the topology of the input to the layer, since the filters work on localised areas and so is more suited for application where this matters (like image processing), unlike with fully connected layers [26].

As features are detected, the approximate location of the feature becomes more important than the precise location. Therefore, convolutional layers are often followed by subsampling layers (e.g. max pool layers discussed in 2.1.2.5) to reduce the resolution [25]. An example of a simple CNN architecture can be seen in Figure 4.

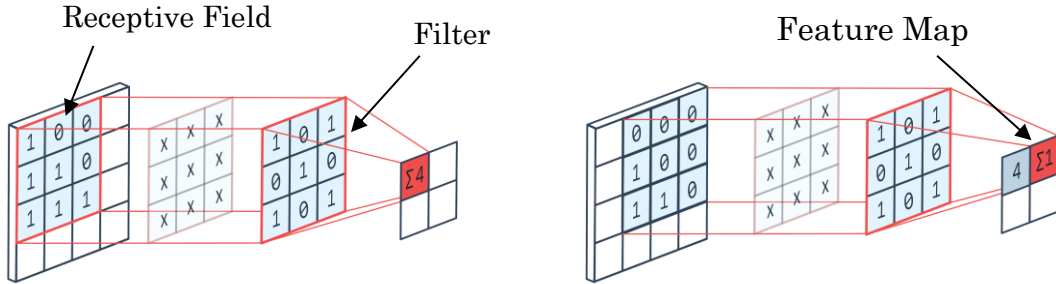


Figure 3. The first two steps in a convolution operation with a filter of size 3×3 and stride 1 [27].

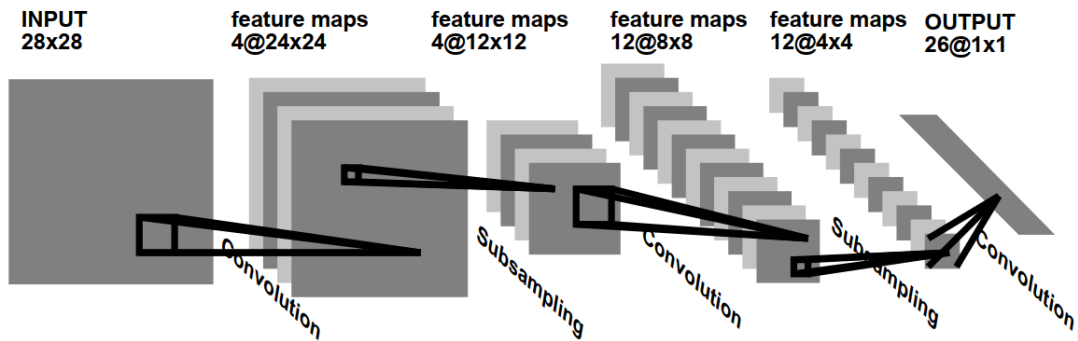


Figure 4. An example of a CNN architecture, with subsampling layers between convolutional ones. This CNN has filters of size 5×5 and convolves with a stride of 1. The number of filters in each convolutional layer is 4, 12 and 26 for the first, second and third respectively [25]

2.1.2.5 Max Pooling Layers

Pooling (or subsampling) layers are generally used to lower the resolution of the layers in a neural network and make the output less sensitive to variations and shifts in the input [26] [28]. These are commonly used in CNNs and are beneficial because at times the rough location of a feature in a feature map is enough to help identify more complex features. The way max pooling layers work is by taking a certain filter size (e.g. 2x2), and scanning over the input matrix (without overlapping any elements), then forming the output matrix using the maximum values in the filter window. A 2x2 max pooling layer would therefore quarter the number of values, lowering the resources required for memory and computations. An example of a 2x2 max pooling layer can be seen in Figure 5. Other pooling layers exist, such as a minimum and average pooling which use different criteria for deciding the output value.

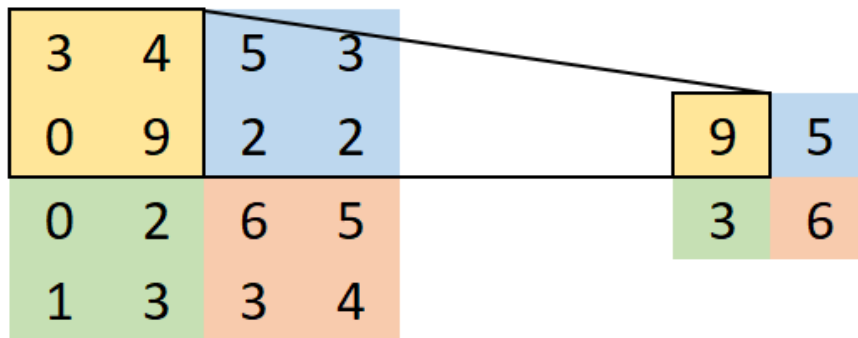


Figure 5. An example of a 2x2 max pool function done on a 4x4 matrix input to produce the 2x2 matrix output

2.1.3 Forward Pass

The forward pass of a NN is the state in which it used for its application. It is the process of the NN predicting (in a regression task) or identifying (in a classification task) based on the input. This project is concerned with classification of images. When training, a forward pass is followed by a backward pass in order to update and optimise the trainable parameters.

2.1.4 Backward Pass

While the training mode (and backward pass) is a critical part of designing a NN, this project is more concerned with the implementation of a NN. Therefore, this section will only give a brief overview of the backward pass.

The training of a NN begins by initialising the trainable parameters and perform a forward pass in order to get an output and check how it performs. At this point, a resulting metric must be used to compare the performance of the network. This is called the cost function. The training process is a method which minimises the error within the network, so minimises the cost function [4].

This is followed by the backward pass (or backpropagation) where the aim is to modify the trainable parameters in order to minimise the cost function. This is done by using the derivative of the cost function, then using the chain rule to propagate back through the network and find the derivatives at each layer [2].

These derivatives are then used, along with a user defined learning rate to update the trainable parameters. The learning rate is what defines how large of an update the method will make on a parameter. A low learning rate will lead to long training times, while setting the learning rate to a high value may lead to unstable parameters during learning and therefore not lead to convergence [4].

This process (apart from parameter initialisation) is then repeated iteratively till convergence of the parameters is met. Training time depends on the number of parameters and layers in the network, the learning rate, the initialised values of the parameters, the specific optimisation method used, and the training set.

2.1.5 Binarization

Binarization of neural network refers to limiting values of the weights and activations within a network to -1 and 1 [10] [11] [29]. These are beneficial when processing efficiency is concerned as more efficient operations can be used over regular multiplication and accumulation, leading to speeds being seven times quicker [12].

Generally, values within non binarized networks are fixed point values. To convert these two different binarization functions can be used, a deterministic and a stochastic method [29]. The deterministic method is shown in (9), it is also easier to implement as opposed to the stochastic method, it does not require the generation of random bits [12].

$$x_b = \text{sign}(x) = \begin{cases} +1 & x \geq 0 \\ -1 & x < 0 \end{cases} \quad (9)$$

Due to the binarization, binarized neural networks (BNNs) can use more efficient operations. These replace any multiplication with XNOR operations, accumulation with popcount, batchnorm-activation with thresholding, and max-pooling with OR operations. Other than the thresholding operation, these operations use binary operations to speed up the process and are especially efficient when implemented in hardware [12].

XNOR – When using binary values where 0 represents -1, multiplication operations can be replaced with XNOR operations, where different inputs lead to an output of 0, and similar inputs lead to an output of 1 [30].

Popcount – Since the values of the outputs of a layer in a BNN are represented by either 0 or 1, and the number of outputs is set, it is possible to derive the summation result by solely knowing the number of 0s or 1s. This can be shown by considering; if the number of outputs is N , number of 1s is N_1 , number of -1s is N_0 , and total is Y , then $N = N_1 + N_0$ and $Y = N_1 - N_0$. Knowing the values of N , and N_1 through a popcount, then the equations can be solved simultaneously to give $Y = 2N_1 - N$ [10].

OR max pooling – Since values at the input for a max pooling layer will either be 0 or 1, if one of the values in the pooling window is 1, then the result of the pooling operation will be a 1. The same result is obtained if performing an OR operation between each input [10].

Thresholding – The thresholding operation can be used in place of batch normalisation layers in a BNN. In a CNN, batch normalization is performed just before the activation function is applied, therefore in a BNN, the activation function essentially just uses the sign of the result of batch normalisation. The point at which the activation function changes value can be found by equating equation (7) to 0, as in (10);

$$x_T = \mu - \frac{\beta}{\gamma \cdot i} \quad (10)$$

Where x_T is the threshold value at which the sign of the batch normalisation result changes. Since the values within the network are represented as 0s and 1s, as opposed to -1s and 1s, then the actual range of the representations is half of the real values, and all operations only deal with positive representation values. To account for this, the threshold value is taken as an averaged sum of x_T and the number of inputs (S), i.e. $x_T^+ = (x_T + S)/2$. Additionally, in order to ensure that all threshold activations represent an output of 1, rather than an activation from 1 to 0, the weight values for that neuron are switched if $\gamma \cdot i < 0$ [10].

2.2 Objectives

The main objective of this project is to create a Fortran implementation of a binarized convolutional neural network.

Performance of the implementation of the network is of importance as it is intended to be used in the future with compilers developed within the faculty with the aim of being implemented on a FPGA. Due to this, the code produced must be compatible with these compilers, which can compile Fortran 77 code. For this reason, the implementation must be compatible with the Fortran 77 standard.

The network to be implemented is the CNV network [10], taking input images in the cifar10 format (32x32 pixel images with 24 bits/pixel). It would be tested against the cifar10 test batch, evaluating the computational and classification performance of the implementation produced. The code shall be able to read in pre trained parameters and use these parameters during the inference of images.

Chapter 3 Design and Implementation

This chapter will discuss the major design decisions and their rationale, as well as give details on the implementation of the project.

3.1 Language Choice - Fortran

Since the future purpose of producing the network was to implement it on a FPGA using the compilers produced by the faculty, the code needed to be compatible with these compilers. Due to this, the first design choice was to implement the network in Fortran, specifically to adhere as much as possible with the Fortran 77 standard.

Fortran 77, while being quick for numerical calculations [31], has a few characteristics which make it a bit different, and sometimes more challenging to code compared to modern languages. Firstly, it is not object oriented, and so all required data must be passed between subroutines or functions as arguments. Additionally, all variables must be initialised before any processing is performed, which causes some difficulties when passing arrays as arguments, and initialising them since their shape is not stored in an array object. Leading to the need of passing all dimension shapes as arguments also as shown in Figure 6.

Another important feature which differs Fortran from other popular languages, is that it operates with column major format for multi-dimensional arrays. This means that when storing arrays, values from the first axes are stored in memory after each other, followed by the second and so on (e.g. $x[0, 0]$ is followed by $x[1, 0]$). Other languages follow the same style as the C languages, where values from the last axes follow each other in memory (e.g. $x[0, 0]$ is followed by $x[0, 1]$). Other than for notation, this must be considered when writing loops, as it is faster to access array values in the order in which they are stored [31]. This means that for more efficient loops, loop indices should be incremented with the left most index being updated the most frequently, and with the frequency dropping with each index to the right. This means that for an array $x[n, m]$, with size n for the columns and m for the rows, loop over m in the outer loop, and n in the inner loop for the best performance, as shown in Figure 6.

```
subroutine Example(res, ch, W, H)
integer :: ch, W, H
integer :: res(ch, H, W)
integer :: i, j, k

do i = 1, W
  do j = 1, H
    do k = 1, ch
      if (...) then
        res(k, j, i) = 0
      end if
    end do
  end do
end do
end subroutine Example
```

Figure 6. An example of part of a typical subroutine

Another feature of Fortran 77 is that it is a fixed format language, where the first five columns of a line are blank or have a number as a label, while the sixth column is set to any value other than zero to indicate a continuation from the previous line. Additionally, only the first 72 columns of a line are read. Due to the inflexibility of this format and in order to make the coding less tedious, it was decided to write the program in a newer version, Fortran 90, which does not require this fixed format. However, adherence to other Fortran 77 features was still kept, and new intrinsic functions introduced with the new version were not to be used in order to ensure compliance with Fortran 77.

Fortran 77, unlike other modern programming languages does not have dynamic arrays. This means that an array cannot be initialised without explicitly stating the size/shape. This could be solved in two ways; have arrays in subroutines be initialised with constant values that are much larger than expected (to ensure the array is always big enough to fit all values) or pass the size and shape of each array as extra arguments through subroutines. The first method would be the simplest, however would cause issues when looping over the array (since many unused elements lie between the end of the useful elements in one column and the start of the next). The second method would be more efficient but would require more arguments to be passed to the subroutine, causing subroutine calling to be more complicated and less user friendly. In this case however, since the performance of the network is of importance, the second method was chosen as the default way to tackle the static array problem.

3.2 Network Choice - The CNV Network

Since the objective of the project was to implement a network, not specifically to design and train one, a few criteria were used in its selection. Firstly, ensuring that the network is compatible with binary weights and activations. Additionally, it must be complex enough to ensure that the code developed is robust, but not too complex such that it is too time consuming to construct. Most importantly however was the availability of data in the form of network parameters and testing images.

Keeping these in mind, the CNV network was chosen, described in [10]. It is a CNN with 8 hidden layers, the architecture is detailed in Table 1, with the first, second and third pair of convolutional layers having 64, 128 and 256 filters respectively. Thresholding is applied at every layer (except the output), before any subsampling is performed. The configuration chosen was one which classifies images based on the cifar10 dataset (available at [13]), which categorises 32x32 pixel, 24bit/pixel images into 10 different categories.

The network parameter data (weight and threshold values) was sourced and modified from the BNN-PYNQ GitHub project [32], which is the code repository for [10]. The data taken from the project was stored in a npz file, which stores compressed numpy arrays. In order to make the data format more convenient and usable with Fortran, the files were converted to binary files by modifying some Python scripts available in the BNN-PYNQ repository [32]. The weights and threshold for each layer were then output into their own file in order to make the importing of data more convenient, as the original python code formatted the data for importing into the PYNQ FPGA board scripts.

Table 1. The architecture of the CNV network

Layer	Type	Input Dimensions	Output Dimensions
Input	Input	-	$32 \times 32 \times 3$
1	3x3 Convolution	$32 \times 32 \times 3$	$30 \times 30 \times 64$
2	3x3 Convolution followed by a 2x2 max pool	$30 \times 30 \times 64$	$14 \times 14 \times 64$
3	3x3 Convolution	$14 \times 14 \times 64$	$12 \times 12 \times 128$
4	3x3 Convolution followed by a 2x2 max pool	$12 \times 12 \times 128$	$5 \times 5 \times 128$
5	3x3 Convolution	$5 \times 5 \times 128$	$3 \times 3 \times 256$
6	3x3 Convolution	$3 \times 3 \times 256$	$1 \times 1 \times 256$
7	Fully connected	$1 \times 1 \times 256$	$1 \times 1 \times 512$
8	Fully connected	$1 \times 1 \times 512$	$1 \times 1 \times 512$
Output	Fully connected	$1 \times 1 \times 512$	$1 \times 1 \times 10$

Two layers within the network are different than the others, these being the first hidden layer, and the output layer. The first layer takes inputs as the pixel values of the images. This layer therefore does not have binary inputs and performs a regular convolution, using multiplication rather than XNOR operations. The result however has thresholding applied, which results in binary outputs. The output layer does not apply thresholding, this results in an array with 10 values (one for each category), with the prediction of the network being the category with the highest value.

3.3 Subroutines

In order to provide some order, make the code more maintainable and improve readability the code was divided into several subroutines each taking responsibility of smaller tasks, therefore breaking down the problem. This allows the code to also be more easily tested (unit testing) and allows for sections of the code to be easily replaced. The following sections will describe the design and implementation choices relating to subroutines. Pseudocode for the major subroutines can be found in Appendix B

3.3.1 Nesting of Subroutines

In general, the design philosophy followed while coding the project was to create subroutines which were as general and abstracted as possible, in order to make them more generic and useful for future iterations of networks. This however led to subroutines which require many arguments, causing them to be less easy to use. To overcome this, other more specific subroutines were created which call the generic ones with specific arguments. An example of this can be seen in Figure 7, which shows a condensed version of the code used for a fully connected layer (without comments). Also known as a dense layer, this is essentially just a matrix multiplication of input values with the weights. In this case, the mmulbin subroutine is similar to a regular matrix multiplication, however uses a XNOR operation. The densebin subroutine therefore just formats the input and outputs of mmulbin to the more convenient forms, and calls mmulbin with the extra arguments. In this case it would be easy to modify the code to be used with non-

binarized systems by replacing the mmulbin subroutine with one which performs a regular matrix multiplication.

```
subroutine densebin(res, a, nOut, nIn, weights)
  integer, intent(in) :: nIn, nOut, a(nIn), weights(nIn, nOut)
  integer, intent(out) :: res(nOut)
  integer tempIn(1, nIn), tempOut(nOut, 1)

  tempIn(1, :) = a(:)
  call mmulbin(tempOut, tempIn, weights, 1, nIn, nIn, nOut)
  res = tempOut(:, 1)

end subroutine densebin
```

Figure 7. Condensed version of the code for a fully connected layer

3.3.2 Subroutines for Each Type of Layer

In order to make the code more reusable and to help generate a framework which could be used in the future, similar to established frameworks such as Theano [9], subroutines were created for each of the layer options. This involved creating subroutines for convolutional layers, dense layers, batch norm layers, max pool layers, threshold layers and activation functions (pseudocode in Appendix B). Some layers also required different configurations for the binary versions, an example of this is shown in Figure 7. This makes it easy to test and reuse the code to create new networks. The construction of the CNV network is shown in Figure 8. The key subroutine (named conv2dbinT) used to perform convolutional layer calculations followed by thresholding is shown in Figure 11 in **Error! Reference source not found.**

```
!Conv layers
call CNVconvT(...)
call conv2dbinT(...)
call maxpool2x23d(...)
call conv2dbinT(...)
call conv2dbinT(...)
call maxpool2x23d(...)
call conv2dbinT(...)
call conv2dbinT(...)

!Dense layers
call densebin(...)
call thresholdLayer(...)
call densebin(...)
call thresholdLayer(...)
call densebin(...)
```

Figure 8. Construction of the CNV network. Arguments of each subroutine have been condensed.

3.3.3 Limitation - Popcount

Fortran77 does not have an intrinsic popcount function and creating a popcount subroutine would be just as computationally intensive as summing up the values being counted (since they are all 1s or 0s). For this reason, and in order to make the code clearer, summation was used instead.

3.4 Developing Automated Testing Subroutines

In order to ensure the subroutines created functioned correctly, and produced the right results, unit testing was used. This gives ability to confirm whether any changes made to a subroutine have a negative effect on the output of that subroutine. In order to do this, a framework of Fortran testing functions was created. These were coded to have similar format to other popular testing frameworks in other languages such as JUnit for Java and unittest in Python. Since this project dealt with matrix and numerical operations a set of subroutines to check whether two arrays are equal were of most importance.

As with other testing frameworks, inputs to the testing subroutines are the expected result, the actual result and a message to identify the test. Additionally, due to issues using Fortran mentioned in 3.1, the shapes of input arrays and the number of dimensions of the arrays were needed as arguments also. Creating subroutines to accept arrays with different dimensions was a challenge, since processing (e.g. using conditional statements) is not possible before instantiating all variables. A system was coded where a subroutine would call the appropriate value checking subroutine depending on how many dimensions the input array has. The implementation of this internal subroutine can be seen in Figure 9.

```
subroutine equalmat(res, a1, a2, sh1, sh2, ndim)
  integer, intent(in) :: ndim, sh1(ndim), sh2(ndim), a1(sh1(1), &
    sh1(2), sh1(3), sh1(4)), a2(sh2(1), sh2(2), sh2(3), sh1(4))
  logical res
  if (ndim == 1) then
    call equalmat1(res, a1, a2, sh1(1), sh2(1))
  else if (ndim == 2) then
    call equalmat2(res, a1, a2, sh1, sh2)
  else if (ndim == 3) then
    call equalmat3(res, a1, a2, sh1, sh2)
  else if (ndim == 4) then
    call equalmat4(res, a1, a2, sh1, sh2)
  end if
end subroutine
```

Figure 9. Subroutine calling the appropriate checking method for matrices of different dimensions

3.5 File System Layout

In order to make the code system easier to navigate, the subroutines have been split up according to certain categories which were mostly based on their function. This segregation also makes the reuse of code easier as only the parts which are required need to be imported over, not the whole code base. The file groups methods in the following categories; methods concerned with manipulating matrices, methods concerned with the layers of networks, methods concerned with testing, methods concerned with the construction of the network and methods concerned with the evaluation testing of the project.

There are also several subroutines which were developed which could form part of a greater framework, however due to the context of this project (focussing on binarized networks) were not used in the final product. In the interest of keeping the generic framework created, these have also been included in a separate, miscellaneous subfolder.

Chapter 4 Testing and Evaluation

This section discusses the methods for unit testing the code and testing the overall success of the implementation. This will be followed by an evaluation and discussion of the results.

4.1 Unit Testing

As mentioned in 3.4, in order to ensure correctness and robustness of the methods developed, a set of testing methods were developed. These methods were used in order to write unit tests for the code.

For most of the simpler subroutines, writing out test cases to test all possible combinations of inputs was easy (for example in Figure 10). More complex subroutines, such as those for the convolutional layers, consisted of their own computational logic as well as that of the simpler subroutines (as these were called from within them). Unit testing for these complex subroutines was done by picking suitable sample inputs and ensuring the output matches the expected. For example, a test case for the binarized convolutional layer without thresholding took an input, filter and expected output as shown in (11).

$$input = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad filter = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \quad exp.ouptut = \begin{bmatrix} 2 & 3 \\ 1 & 1 \end{bmatrix} \quad (11)$$

```
subroutine testxnor()  
  integer res  
  call xnor(res, 1, 0)  
  call testequalintegermat([res], [0], [1], [1], 1, 'xnor1')  
  call xnor(res, 0, 1)  
  call testequalintegermat([res], [0], [1], [1], 1, 'xnor2')  
  call xnor(res, 1, 1)  
  call testequalintegermat([res], [1], [1], [1], 1, 'xnor3')  
  call xnor(res, 0, 0)  
  call testequalintegermat([res], [1], [1], [1], 1, 'xnor4')  
end subroutine testxnor
```

Figure 10. Unit testing subroutine testing all input combinations for the xnor subroutine.

4.2 Testing with Cifar10 Images

Two different aspects of the code were tested; the accuracy and the timed performance. Test scripts were developed in order to do this, using the cifar10 test batch and the pretrained weight and threshold files, images and parameters were loaded. Images were then passed through the network, checking whether the inference results matched with the label of the images. A matching label meant that the image was correctly identified.

Additionally, a few of the computational subroutines were timed in order to identify any bottlenecks within the system and find any possible opportunities for optimisations. The code was improved over several iterations of optimisations.

4.2.1 Accuracy Test Results

The cifar10 test batch contained 10000 images, 1000 in each category. These being; airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. The resulting accuracy of the test was 50.78%. Details on the results are given in Table 2. The distribution of choices was also checked, and results shown in Table 3.

Table 2. Results of the accuracy test

Index No.	Category	Correct Classifications	Accuracy
0	Airplane	747	74.7%
1	Automobile	532	53.2%
2	Bird	306	30.6%
3	Cat	595	59.5%
4	Deer	319	31.9%
5	Dog	443	44.3%
6	Frog	368	36.8%
7	Horse	510	51.0%
8	Ship	825	82.5%
9	Truck	433	43.3%
	Total Correct	5078	50.78%

Table 3. Distribution of classifications, with images by label in each row and each column showing the classification result of the network. The diagonal indicates correct classifications.

Index	0	1	2	3	4	5	6	7	8	9
0	747	0	18	28	9	10	4	4	177	3
1	131	532	3	38	15	17	4	9	211	40
2	311	1	306	158	33	75	19	16	80	1
3	164	2	32	595	12	95	14	12	69	5
4	219	1	53	214	319	86	17	19	70	2
5	131	0	24	299	19	443	11	24	46	3
6	119	1	33	272	35	77	368	10	83	2
7	98	0	28	139	57	101	4	510	59	4
8	127	3	4	28	1	7	2	0	825	3
9	123	31	5	64	14	8	2	9	311	433

4.2.2 Performance Test Results

Several iterations of the code were tested for performance, each being an optimised version of the previous. The initial test resulted in an average inference time of about 0.75s. The second iteration improved upon that to an average inference time of 0.12s. The final and most efficient version resulted in an average inference time of 0.036s, about 5% of the original time.

In all cases, the time required for loading the images and network parameters into memory were about 4.4s and 0.2s respectively.

4.3 Evaluation

The following sections will discuss the results presented and give some explanation into mitigation strategies and possible discrepancies.

4.3.1 Accuracy Discussion

The accuracy results obtained, although significantly lower than the 80.6% presented in [10], still suggest that the network is functioning correctly. If the network was not functioning the results would be random, closer to an accuracy of about 10%. This was tested by altering parts of the code. One of such tests was transposing the image arrays when loading. This led to the expected accuracy of about 10%, seemingly a random selection. Apart from this, the parameter array reading order was inverted, leading to a similar accuracy of about 10%. Due to the results of these tests, the most probable cause of the discrepancy is the network parameters not being fully trained. It is possible that the parameter files obtained from the BNN-PYNQ repository [32] are not the fully trained weights. Given more time, retraining the parameters to ensure fully trained ones are used would be the next step.

Analysing the results in Table 2, it can be seen that some categories are more easily classified than others, with images of ships, airplanes and cats being the most accurately classified and birds, deer and frogs being the least accurately classified. Looking at Table 3, some interesting results can be noted, firstly that bird images are often misclassified as airplane images possibly due to similar features appearing in both (e.g. wings, clouds, tails, etc.), similarly with truck images being classified as ships and dog images being classified as cats. Other than this, it is also noticeable that airplanes and cats are the most common misclassifications of images. This could be a symptom of the parameters not being fully trained.

4.3.2 Performance Discussion

Overall there were 2 sets of improvements which significantly improved inference time over the iterations. The first iteration improved the time from 0.75s to 0.12s. This was achieved by in-lining some of the code in the convolutional layer function (shown in Appendix A). The xnor and summing operations that were initially subroutines were in-lined manually which while reducing the maintainability but improved performance significantly.

The second set of optimisations improved the performance from 0.12s to about 0.036s. This was done by optimising the looping over arrays to follow the column major format. This involved refactoring some of the layout of the arrays, in order to make it more convenient to loop over left most indices in the inner loops and right most indices in outer loops.

The efficiency was checked by roughly calculating convolutional layers having *input channels * output resolution * filter size * num of filters* operations and dense layers having *number of input nodes * number of output nodes* operations. Summing the results for each layer and dividing by 2.7×10^9 (the speed of the processor used at runtime in Hz) results in a time of about 0.022s for one inference. This is reasonably close to the obtained value; therefore, the current implementation is quite efficient.

Chapter 5 Conclusion

As the technology behind computing keeps advancing, the power of portable devices will increase too. Combining this with research working towards more efficient neural network architectures aids in bringing the idea of the offline neural network into common use. The aim of this project was to implement a binarized neural network which could be used in the future to help reach that goal.

The project meets the objectives set in 2.2, providing a functioning framework with which to build and implement binarized neural networks in Fortran. Due to the generalised approach followed code developed is not only tailored specifically for this network however can be used to construct other networks, binarized or not. The accuracy of the network implemented, while not being as expected, has been shown to be correctly classifying a significant amount of the images tested. The performance optimisations performed to the computational algorithms also ensure that the networks will process efficiently and in a timely manner. A set of testing methods were developed and used to create unit tests for the implementation ensuring its robustness.

While the code does produce a functioning network, the accuracy of the network is lower than anticipated. With the problem most likely stemming from using parameters which were not fully trained. Had there been more time, the training of parameters could have been done to ensure properly trained parameters. The choice to use the parameters was due to availability, as the binarized parameters were provided in an accessible format. Training of new parameters however would require GPU processing time starting from 48hrs [10] for fully trained parameters, so this must be taken account as extra resources may be required for timely training.

In hindsight, the use of Fortran 77 did cause some challenges which could have been avoided with the use of a more recent version of Fortran. This would allow the use of dynamic (allocatable) arrays, shortening subroutine signatures and making the development process simpler. Additionally, newer versions would allow the use of new intrinsic functions, and especially array manipulation functions which could make the processing even more efficient.

5.1 Future Work

This project provides a good first step on which to improve. Several ways have been identified which allow further functionality or improve the system. Firstly, a set of fully trained parameters can be trained and used within the implemented network in order to obtain better results in the accuracy tests. Additionally, the framework can be used to implement other networks in order to provide further case studies on the performance of the code produced. The first candidates could be the SVHN and MNIST networks discussed in [10], as these use similar methods and networks.

One limitation of the code produced is that it relies on other frameworks to train the network. The development of a system that would perform the training of

these networks within Fortran would allow easier integration and compatibility. Similarly, the code relied on other sources to format image into a useful format. Developing some subroutines which would take different image files and automatically process them into the required format would allow for a greater range of testing, outside the cifar10 dataset.

Once the compilers within the faculty have been prepared, it would also be interesting to use them to implement the network created on a FPGA board. This along with improving the code to process in parallel would allow the network to see a significant jump in timed performance as the processing load could be split up amongst more threads.

As a final remark, while the subroutines developed provide a few useful building blocks from where to build a network, the library could be expanded with the inclusion of different features like new types of layers or different activation functions. This would improve functionality and allow a greater range of configuration of any network produced.

Chapter 6 References

- [1] Curcic, Milan. A parallel Fortran framework for neural networks and deep learning. *CoRR*, abs/1902.06714 (2019).
- [2] LeCun, Yann, Bengio, Yoshua, and Hinton, Geoffrey. Deep Learning. *Nature*, 521 (2015), 436-444.
- [3] Vojt, Bc. Ján. *Deep neural networks and their implementation*. Charles University in Prague, Prague, 2016.
- [4] Goodfellow, Ian, Bengio, Yoshua, and Courville, Aaron. *Deep Learning*. MIT Press, 2016. Available at: <http://www.deeplearningbook.org>. Accessed: 5th Sept 2019.
- [5] Lopamudra, Baruah. *Performance Comparison of Binarized Neural Network with Convolutional Neural Network*. Michigan Technological University, Michigan, 2017. Available at: <https://digitalcommons.mtu.edu/etdr/487>. Accessed: 5th Sept 2019.
- [6] Colloert, Ronan, Kavukcuoglu, Koray, and Farabet, Clément. Torch7: A Matlab-like Environment for Machine Learning. In *NIPS* (2011).
- [7] Abadi, Martin et al. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016).
- [8] Jia, Yangqing et al. Caffe: Convolutional Architecture for Fast Feature Embedding. *CoRR*, abs/1408.5093 (2014).
- [9] Al-Rfou, Rami et al. Theano: A Python framework for fast computation of mathematical. *CoRR*, abs/1605.02688 (2016).
- [10] Umuroglu, Yaman et al. FINN: A Framework for Fast, Scalable Binarized Neural Network. *CoRR*, abs/1612.07119 (2016).
- [11] Hubara, Itay et al. Binarized Neural Networks. (2016), *NIPS*.
- [12] Courbariaux, Matthieu et al. BinaryNet: Training Deep Neural Networks with Weights and Activations. *CoRR*, abs/1602.02830 (2016).
- [13] Krizhevsky, Alex. *The CIFAR-10 dataset*. Available at: <https://www.cs.toronto.edu/~kriz/cifar.html>. Accessed: 5th Sept 2019.
- [14] Alom, Md Zahangir et al. The History Began from AlexNet: A Comprehensive Survey on Deep Learning. *CoRR*, abs/1803.01164 (2018).
- [15] Mohri, Mehryar, Rostamizadeh, Afshin, and Talwalkar, Ameet. *Foundations of Machine Learning, 2nd Edition*. MIT Press, 2018.

- [16] Silver, David et al. Mastering the game of Go without human knowledge. *Nature*, 550 (2017), 354-359.
- [17] Nielson, Micheal A. *Neural Networks and Deep Learning*. Determination Press, 2015. Available at: <http://neuralnetworksanddeeplearning.com>. Accessed: 5th Sept 2019.
- [18] Nwankpa, Chigozie et al. Activation Functions: Comparison of trends in Practice and Research. *CoRR*, abs/1811.03378 (2018).
- [19] Ng, Andrew. *Why do you need non-linear activation functions?* Coursera, Available at: <https://www.coursera.org/learn/neural-networks-deep-learning>. Accessed: 5th Sept 2019.
- [20] Jing, Yang and Guanci, Yang. *Modified from - Modified Convolutional Neural Network Based on Dropout and the Stochastic Gradient Descent Optimizer*. licensed under CC 4.0, Available at: https://www.researchgate.net/figure/Nonlinear-function-a-Sigmoid-function-b-Tanh-function-c-ReLU-function-d-Leaky_fig3_323617663. 2018. Accessed: 5th Sept 2019.
- [21] Ioffe, Sergey and Szegedy, Christian. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR*, abs/1502.03167 (2015).
- [22] Peixeiro, Marco. *How to Improve a Neural Network With Regularization*. Towards Data Science. Available at: <https://towardsdatascience.com/how-to-improve-a-neural-network-with-regularization-8a18ecda9fe3>. 2019. Accessed: 5th Sept 2019.
- [23] LASAGNE. *Lasagne Normalisation Layer Documentation*. Available at: <https://lasagne.readthedocs.io>. Accessed: 5th Sept 2019.
- [24] Ba, Jimmy Lei et al. Layer Normalisation (2016).
- [25] LeCun, Yann and Bengio, Yoshua. Convolutional networks for images, speeck and time series. In *The handbook of brain theory and neural networks*. MIT Press, Cambridge, MA, USA, 1998.
- [26] LeCun, Yann et al. Gradieant-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86, 11 (Nov 1998), 2278-2324.
- [27] PELTARION. *2D Convolution block*. Available at: <https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/blocks/2d-convolution-block>. Accessed: 5th Sept 2019.
- [28] Scherer, Dominik et al. Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition. (Thessaloniki, Greece 2010), ICANN.

- [29] Liang, Shuang et al. *FP-BNN: Binarised neural network on FPGA*. Available at: <https://doi.org/10.1016/j.neucom.2017.09.046>. 2017. Accessed: 5th Sept 2019.
- [30] Rastegari, Mohammed et al. *XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks*.
- [31] Price, Clive G. *Professional Programmer's Guide to Fortran77*. University of Leicester, UK. Available at: <https://www.star.le.ac.uk/~cgp/prof77.html>. 2015. Accessed: 5th Sept 2019.
- [32] Gambardella, Giulio et al. *BNN-PYNQ*. GitHub repository, <https://github.com/Xilinx/BNN-PYNQ>. 2019. Accessed: 5th Sept 2019.

Appendix A Convolutional Layer Code

```

subroutine conv2dbinT(res, a, chOut, chIn, W, H, fil, fn, fm, stride, thres)
!performs a 2d convolution, filter is binary, performs thresholding. Expects
!input array to be binary values
!res - output matrix
!a - input matrix (BINARY INPUT)
!chOut - number of output channels
!chIn - number of input channels
!W - width of each channel (num of cols)
!H - height of each channel (num of rows)
!fil - matrix of filters (BINARY)
!fn - number of
!stride - stride
!thres - thresholds
integer, intent(in) :: chOut, chIn, W, H, a(chIn, H, W), fn, fm, fil(chIn, &
fn, fm, chOut), stride
integer, intent(in) :: thres(chOut)
integer, intent(out) :: res(chOut, (H-fn)/stride+1, (W-fm)/stride+1)
integer i, j, l, tempsum, filmult(chIn, fn, fm)
integer i2, j2, k2

res = res * 0 !initialise result array

do l = 1, chOut !cycle through each channel
  do j = 1, H-fn+1, stride !slide the filter over the input
    do i = 1, W-fm+1, stride

      !perform an elementwise xnor operation between the filter and
      !receptive field
      do i2 = 1, fm
        do j2 = 1, fn
          do k2 = 1, chIn
            if (fil(k2, j2, i2, 1) == a(k2, j+j2-1, i+i2-1)) then

              filmult(k2, j2, i2) = 1
            else
              filmult(k2, j2, i2) = 0
            end if
          end do
        end do
      end do

      !sum all values from the result of the xnor operation above
      tempsum = 0

      do i2 = 1, fm
        do j2 = 1, fn
          do k2 = 1, chIn
            tempsum = tempsum + filmult(k2, j2, i2)
          end do
        end do
      end do

      !check the result against the threshold value, set as 0 if below or
      !1 otherwise
      if (tempsum < thres(l)) then
        res(l, (j-1)/stride+1, (i-1)/stride+1) = 0
      else
        res(l, (j-1)/stride+1, (i-1)/stride+1) = 1
      end if
    end do
  end do
end do

end subroutine conv2dbinT

```

Figure 11. Subroutine for a binarized convolutional layer with thresholding applied.

Appendix B Pseudocode for Major Subroutines

B.1 Binary Dense Layer:

Inputs	Variable	Description
	res	Array of output node values
	inp	Array of input node values
	nOut	Number of outputs
	nIn	Number of inputs
	weights	Array of weights with size

- 1) Reformat inp to have each value in a separate row
- 2) Perform binary matrix multiplication on inp with weights
 - a. For each row in inp do:
 - i. For each column in weights:
 1. Total = 0
 2. For each digit in the row:
 - a. Perform xnor operation between inp(row, digit) and weights(digit, column)
 - b. Total = total + xnor result
 3. res(row, column) = total

B.2 Binary Convolutional Layer with Thresh:

Inputs	Variable	Description
	res	Array of output node values
	inp	Array of input node values
	chOut	Number of output channels
	chIn	Number of input channels
	W	Width of each channel
	H	Height of each channel
	fil	Binary matrix of filters
	fn	Number of rows in filter
	fm	Number of columns in filter
	stride	Stride of the convolution
	thres	Threshold values for each output channel

- 1) For l = 1 to chOut:
 - a. For j = 1 to H-fn+1 incrementing by stride:
 - i. For i = 1 to W-fm+1 incrementing by stride:
 1. Perform elementwise xnor between fil[:, :, :, l] and inp[:, j:j+fn-1, i:i+fm-1].
 2. Sum all values in result of xnor operation in tempsum
 3. oj = (j-1)/stride + 1
 4. oi = (i-1)/stride + 1
 5. If tempsum < thres[l]:
 - a. res[l, oj, oi] = 0
 6. Else
 - a. Res[l, oj, oi] = 1

B.3 2d Max Pool Layer:

Inputs	Variable	Description
	res	Array of output node values
	inp	Array of input node values
	ps	Size of the pooling window
	n	Number of rows in inputs
	m	Number of columns in inputs

```
1) For row = 1 to n, incrementing by ps
  a. For col = 1 to m, incrementing by ps
    i. Identify largest value in pool window [row:row+ps-1,
      col:col+ps-1]
      1. Set max = a large negative number
      2. For each row2 in window
        a. For each column2 in window
          i. If value > max
            ii. loc = [row2, column2]
      ii. or = row/ps +1
      iii. oc = row/ps +1
      iv. res(or, oc) = input(row+loc[1]-1, col+loc[2] -1)
```

B.4 Threshold Layer:

Inputs	Variable	Description
	res	Array of output node values
	inp	Array of input node values
	ch	Number of channels
	W	Width of each channel
	H	Height of each channel
	thres	Array of threshold values for each input

```
1) For k = 1 to ch
  a. For j = 1 to H
    i. For i = 1 to W
      1. If inp[k, j, i] > thres[k] then
        a. res[k,j,i] = 1
      2. Else
        a. res[k,j,i] = 0
```