MULTIPLICATION FREE NEURAL NETWORKS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

By
Maen M. A. Mallah
January 2018

Multiplication Free Neural Networks
By Maen M. A. Mallah
January 2018

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

A. Enis Çetin(Advisor)

Muhammet Mustafa Özdal

Ramazan Gökberk Cinbiş

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

MULTIPLICATION FREE NEURAL NETWORKS

Maen M. A. Mallah
M.S. in Electrical and Electronics Engineering
Advisor: A. Enis Çetin
January 2018

Artificial Neural Networks, commonly known as Neural Networks (NNs), have become popular in the last decade for their achievable accuracies due to their ability to generalize and respond to unexpected patterns. In general, NNs are computationally expensive. This thesis presents the implementation of a class of NN that do not require multiplication operations. We describe an implementation of a Multiplication Free Neural Network (MFNN), in which multiplication operations are replaced by additions and sign operations.

This thesis focuses on the FPGA and ASIC implementation of the MFNN using VHDL. A detailed description of the proposed hardware design of both NNs and MFNNs is analyzed. We compare 3 different hardware designs of the neuron (serial, parallel and hybrid), based on latency/hardware resources trade-off.

We show that one-hidden-layer MFNNs achieve the same accuracy as its counterpart NN using the same number of neurons. The hardware implementation shows that MFNNs are more energy efficient than the ordinary NNs, because multiplication is more computationally demanding compared to addition and sign operations. MFNNs save a significant amount of energy without degrading the accuracy. The fixed-point quantization is discussed along with the number of bits required for both NNs and MFNNs to achieve floating-point recognition performance.

Keywords: Neural Networks, Machine Learning, Classification, VHDL, Energy, Fixed-point, Floating-point.

ÖZET

ÇARPMA İŞLEMSİZ SİNİR AĞLARI

Maen M. A. Mallah
Elektrik ve Elektronik Mühendisliği, Yüksek Lisans
Tez Danışmanı: A. Enis Çetin
Ocak 2018

Sinir ağları olarak da bilinen yapay sinir ağları son yıllarda, özellikle yüksek doğruluk oranlarına erişebilmesi ve önceden tahmin edilemeyen örüntüleri genelleştirebilmesi sebebi ile son yıllarda tekrar popüler oldu. Genel olarak sinir ağlarının hesap yükleri fazladır. Bu tez, çarpma işlemi gerektirmeyen bir grup sinir ağı içermektedir. Bu tezde, Çarpma İşlemsiz Sinir Ağları adı altında, çarpma işlemlerinin işaret ve toplama işlemleri ile değiştirildiği sinir ağlarının uygulaması sunulmaktadır.

Bu tezde Çarpma İşlemsiz Sinir Ağlarının VHDL kullanarak FPGA ve ASIC uygulamarı üzerinde durulmaktadır. Sinir ağları ve Çarpma İşlemsiz Sinir Ağları için detaylı bir açıklama ve önerilen donanım planı analiz edilmiştir. Bir nörünun gecikme süresi-donanım kaynağı ödünleşimi açısından, üç farklı donanım için dizaynı (seri, paralel ve hibrid) performansı karşılaştırılmaktadır.

Bir katmanlı çarpma işlemsiz yapay sinir ağlarının performansının, bir katmanlı standart sinir ağlarıyla aynı oranlara erişebildiğini göstermekteyiz. Donanım uygulaması ile ise, çarpma işlemsiz yapay sinir ağlarının, toplama ve işaret işlemi çarpma işlemine göre çok daha az enerji harcadığı için, enerji açısından çok daha verimli olduklarını göstermekteyiz. Çarpam işlemsiz Sinir Ağları, doğruluk performansından fazla ödün vermeden yüksek oranda enerji tasarrufu sağlamaktadırlar. Kayan noktalı tanıma performansı için, sabit noktalı sayısallaştırma ile birlikte sinir ağları ve çarpma işlemsiz sinir ağları için gerekli bit sayısı ayrıca tartışılmıştır.

Anahtar sözcükler: Sinir Ağları, Makine Öğrenimi, Sınıflandırma, VHDL, Enerji, Sabit nokta, Kayan nokta.

Acknowledgement

I would like to express my deepest appreciation to my supervisor, Dr Prof. A. Enis Çetin, for his patient guidance, valuable insight, and constructive suggestions. I have been extremely lucky to have a supervisor who cares so much about my research, and works so close with me at every step throughout my M.Sc studies.

I am particularly grateful for the guidance given by Mr. Martin Leyh during my internship at Fraunhofer IIS institute in the past 6 months. His experience and insight were crucial for the quality of this work.

I would like to extend my thanks to Prof. F. Yarman-Vural and her students for their fruitful discussions.

I would like to thank TÜBİTAK for supporting me through BİDEB 2215 Scholarship.

Special thanks to Fatima Villa, Diaa Badawi and Hamed Salah, who have invested their time to assist me with this work.

I would like to thank my family and friends, you should know that your support and encouragement was worth more than I can express on paper.

Finally, to Mom and Dad, all the support you have provided me over the years was the greatest gift anyone has ever given me. This one is for you!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

1.1.1 Machine Learning

Machine learning is a computer science field that emerged from artificial intelligence field. As the name suggests, machine learning enables the computers (machines) to learn without being explicitly programmed [1]. This is done by building generic models that have a set of parameters into the computers. The computer determines the models' parameters using previous collected data. This process of determining the parameters is refereed to as learning.

Tom M. Mitchell provided a widely quoted, more formal definition of the algorithms studied in the machine learning field: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E." [2]

Machine learning is a wide field. In this work, we focus on the supervised classification task using neural networks models [3].

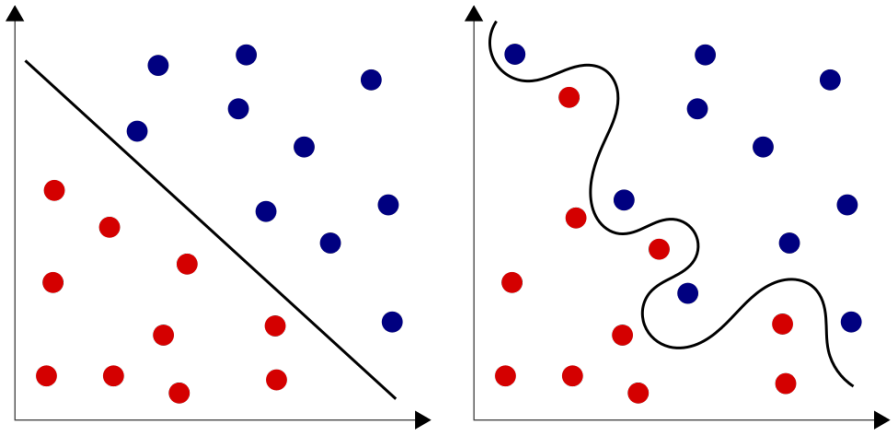
1.1.2 Classification

Classification is a supervised learning task that studies the problem of identifying to which set (class) does a new observation belong to. This work, studies single-class classification, i.e., every point is assigned to one and only one class. The classification model (classifier) uses previous examples of labeled data (training set) to classify new observations. Labeled data means that the classes of the observations are known. Supervised machine learning models have been widely used to solve classification problems in various fields, e.g. image classification, computer aided diagnosis and video tracking [4–10].

A classifier f maps an input observation $x \in R^N$ to an output class $y \in \{c_1, c_2, ..., c_M\}$ where N is the number of features and M is the number of classes.

$$f : R^N \rightarrow \{c_1, c_2, ..., c_M\} \tag{1.1}$$

Data can be categorized into: I. Linearly separable where the data can be separated by a hyperplane and II. Non linearly separable where the data cannot be separated by a hyperplane (Fig. 1.1)[1].



(a) Linearly separable (b) Non linearly separable

Figure 1.1: Data Separability

[1]Mekeor (`https://commons.wikimedia.org/wiki/File:Separability_NO.svg`), "Separability NO", Mekeor (`https://commons.wikimedia.org/wiki/File:Separability_YES.svg`), "Separability YES", `https://creativecommons.org/licenses/by-sa/3.0/legalcode`

1.1.3 Neural Network (NN)

The first attempt to build a neuron was performed in 1943 by McCulloch and Pitts [11]. Later, in 1958, Rosenblatt invented the perceptron [12]. The following years, neural networks faced some challenges that slowed their improvement. These challenges are the lack of powerful computers, the lack of training algorithms and the inability of the perceptron to separate non linearly separable data [13]. The neural networks exploded in the 1980s after discovering the multilayer perceptrons [14] and formulating the error back-propagation algorithm to train the weights [15].

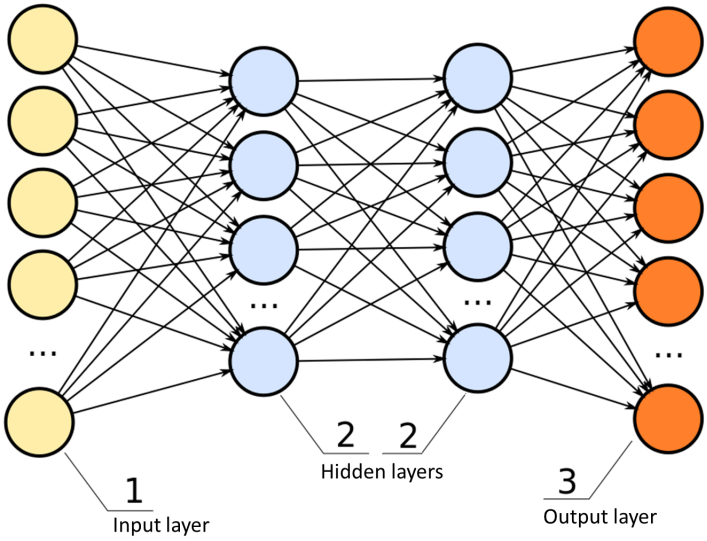


Figure 1.2: Multilayer Perceptron

Artificial Neural Networks, commonly known as Neural Networks (NN), have become popular in the last decade, following their huge success in the classification problems, especially after the advent of Convolutional Neural Networks (CNN) [16]. NN have found applications in business, commerce and industry from image classification to natural language processing, with accuracies as good as humans or even better. However, such systems contain up to millions of parameters that require training and storage to be later used for inference. Moreover, these many parameters need high computation power and storage space. Therefore,

convolutional neural networks are yet to find their way to mobile phones, ARM processors and embedded systems, where energy is also a big concern [17].

The NN have different architectures. One of them is Multilayer Perceptron (MLP) [18]. In MLP, the neurons (perceptrons) are organized in layers. Each neuron is connected to all neurons in the previous layer with different weights. The layers of the network are of three types: input, hidden and output. In MLP, there is one input, one output, and any number of hidden layers (Fig. 1.2).

1.1.4 Notation

Throughout this work, all vectors are column vectors and represented by boldface lowercase letters. Matrices are represented by boldface uppercase letters. o_j^l and b_j^l are the output and bias terms of the j^{th} neuron in layer l, respectively. w_{ij}^l is the connection weight between the i^{th} and j^{th} neurons in layers $l-1$ and l, respectively. N_l is the number of neurons in layer l. Layer $l=1$ and $l=L$ are the input and output layers, respectively, where L is the number of layers in the MLP. L is restricted such that $L \geq 2$ where $L=2$ is a network without any hidden layers, i.e., the network consists of only the input and output layers. $\boldsymbol{x}(n)$, $y(n)$ and $\boldsymbol{t(n)}$ are the n^{th} input and its corresponding predicted and true outputs of the network.

1.1.5 Multi-Layer Perception (MLP)

The conventional neuron (Fig. 1.3) in MLP carries out a weighted sum of the inputs followed by adding a bias term and finally passed through an activation function:

$$o_j^l = f\left(\sum_{i=1}^{N_{l-1}} w_{ij}^l o_i^{l-1} + b_j^l\right) \tag{1.2}$$

where o_j^l and b_j^l are the output and bias term of the j^{th} neuron in the l^{th} layer. w_{ij}^l is the weight connection between the i^{th} and j^{th} neurons in layers $l-1$ and l,

respectively. Finally, $f(.)$ is a non-linear activation function e.g. hyperbolic tangent (tanh), sigmoid or LeakyReLU. Table 1.1 lists some of the famous activation functions and their derivatives.

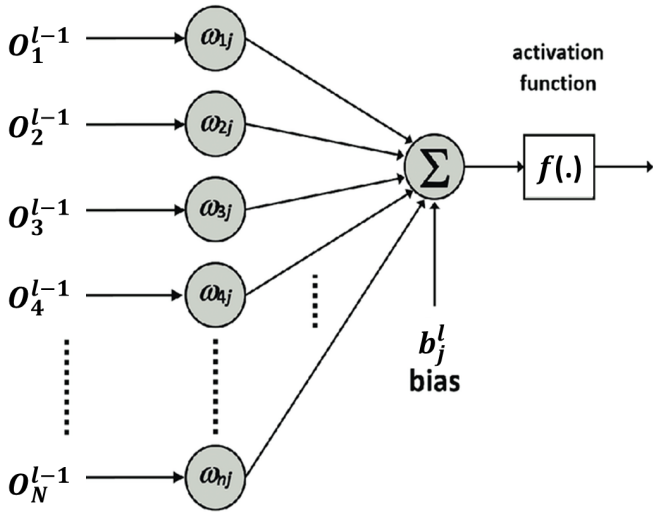


Figure 1.3: Perceptron

In matrix notation (1.2) is:

$$\boldsymbol{o}^l = \boldsymbol{f}\left(\mathbf{W}^{l^T}\boldsymbol{o}^{l-1} + \boldsymbol{b}^l\right) \tag{1.3}$$

where $\mathbf{W}^l$ is a matrix of w_{ij}^l.

In the feed-forward algorithm, the outputs of the network $\boldsymbol{o}^L$ are calculated by carrying out (1.3) for $l = 2, ...L$, where $\boldsymbol{o}^1 = \boldsymbol{x}$ is the input vector. Therefore, $N_1 = N$ (the number of features) and $N_L = M$ (the number of classes).

For a sample observation $\boldsymbol{x}(n)$, the final classification $y(n)$ is obtained by calculating the maximum of $\boldsymbol{o}^L(n)$, i.e.:

$$\text{predicted label} = \arg\max_j o_j^L(b) \tag{1.4}$$

From (1.3), we see that for each layer $l > 1$ ($l = 1$ is the input layer where there is no processing done) in the network there are: $N_{l-1} \times N_l$ multiplication operations, $(N_{l-1} + 1) \times N_l$ addition operations, and N_l non-linear activation function operations.

Activation Function	Formula	Derivative
Sigmoid	$sigm(x) = \frac{1}{1+e^{-x}}$	$sigm(x)(1 - sigm(x))$
Hyperbolic tangent	$tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$	$1 - tanh(x)^2$
Leaky ReLU[2]	$ReLU(x) = max(x, ax)$	$max(1, a)$

Table 1.1: List of some activation functions and their derivatives

1.1.6 Training

Neural networks as a statistical classifier involve two main tasks: I. Train the parameters (weights) of the network using previous data and then II. Inference on the new data using the parameters obtained in I. The training task is where time and effort are spent. However, it is done prior to the system deployment using powerful computers. On the other hand, the inference has to be done in real-time on the targeted device which requires good computational power. Otherwise, inference can be performed in the cloud on more powerful servers, but this requires Internet connectivity and good bandwidth to transfer to the data.

1.1.6.1 Stochastic Gradient Descent (SGD)

The training task is the task of using previous data to find the model parameters (weights and bias in the case of NN). The model can later use these parameters to determine the output of new observations.

The learning problem can be seen as an optimization problem formulated as $\min_{\mathbf{W},\boldsymbol{b}} J(\boldsymbol{o}^L(n), \boldsymbol{t}(n))$ where $\mathbf{W}$ is a collection of all weights $\{\mathbf{W}^2, \mathbf{W}^3, ...\mathbf{W}^L\}$, $\boldsymbol{b}$ is a collection of all bias terms $\{\boldsymbol{b}^2, \boldsymbol{b}^3, ...\boldsymbol{b}^L\}$, and $J(\boldsymbol{o}^L(n), \boldsymbol{t}(n))$ is a cost function that measures the distance between the predicted output $\boldsymbol{o}^L(n)$ and the true output $\boldsymbol{t}(n)$ of observation n. The model (NN) is optimized (trained) using stochastic gradient descent (SGD) algorithm [19].

[2]a is the leakage coefficient (scale) of the Leaky ReLU where $ale1$

SGD is an iterative algorithm that updates the weights as follows:

$$\mathbf{W}^l_{iter+1} = \mathbf{W}^l_{iter} - \eta \frac{\partial J(\boldsymbol{o}^L(n), \boldsymbol{t}(n))}{\partial \mathbf{W}^l_{iter}} \tag{1.5}$$

$$\boldsymbol{b}^l_{iter+1} = \boldsymbol{b}^l_{iter} - \eta \frac{\partial J(\boldsymbol{o}^L(n), \boldsymbol{t}(n))}{\partial \boldsymbol{b}^l_{iter}} \tag{1.6}$$

where η is the step size. $\mathbf{W}^l_{iter}$ are the weights at iteration $iter$. $\frac{\partial J(\boldsymbol{o}^L(n), \boldsymbol{t}(n))}{\partial \mathbf{W}^l_{iter}}$ and $\frac{\partial J(\boldsymbol{o}^L(n), \boldsymbol{t}(n))}{\partial \boldsymbol{b}^l_{iter}}$ are the partial derivative of the cost function w.r.t $\mathbf{W}^l_{iter}$ and $\boldsymbol{b}^l_{iter}$, respectively.

In other words, SGD updates the weights by trying to shift them to minimize the cost function in the next iteration. That is achieved through the derivatives. The derivatives are composed of the direction and the magnitude of the increase in the cost functions. Therefore, updating the weights in the opposite directions (multiplying by -1) leads to a descent through the cost function. Additionally, the magnitude is normalized by $\eta < 1$ so that the jumps are not drastic. Very small η can drive the optimization to a local minimum and the learning is very slow, while, a big η can lead to no convergence in the algorithm.

The cost function is non-convex, thus, the optimization could yield a local minimum rather than the global one depending on the initial starting point, $\mathbf{W}^l_0$, $\boldsymbol{b}^l_0$, which is usually randomly selected. This problem is addressed by training the models several times with different starting points.

One of the most widely used cost functions is the Mean Square Error (MSE) defined as:

$$J(\boldsymbol{o}^L(n), \boldsymbol{t}(n)) = \frac{1}{2} \sum_i^M (o^L_i(n) - t_i(n))^2 \tag{1.7}$$

where $\boldsymbol{t}(n)$ is the true label vector for observation n defined as:

$$t_i(n) = \begin{cases} 1 & \text{if } i = \text{label of observation } n \\ 0 & \text{otherwise} \end{cases} \tag{1.8}$$

Cross-Entropy is another example of a widely used cost function.

1.1.6.2 Back-Propagation

In order to update the weights using SGD, the partial derivative of the cost function w.r.t each weight is to be computed. This involves a lot of computations when performed separately. However, the back-propagation algorithm is used to propagate the the cost (error) derivate from layer ($l = L$) to layer ($l = 2$) [15]. This technique reduces the computations by reusing the values that have already been calculated.

First, let us defined v_j as:

$$v_j^l = \sum_{i=1}^{N_{l-1}} w_{ij}^l o_i^{l-1} + b_j^l \tag{1.9}$$

Or in matrix notation:

$$\boldsymbol{v}^l = \mathbf{W}^{l^T} \boldsymbol{o}^{l-1} + \boldsymbol{b}^l \tag{1.10}$$

Then (1.2) and (1.3) become:

$$o_j^l = f(v_j^l) \tag{1.11}$$

$$\boldsymbol{o}^l = f(\boldsymbol{v}^l) \tag{1.12}$$

Using chain rule, $\frac{\partial J(\boldsymbol{o}^L, \boldsymbol{t}(n)}{\partial \mathbf{W}_{iter}^l}$ (n was dropped from $\boldsymbol{o}^L(n)$ to simplify the term) is expressed as follows:

$$\frac{\partial J(\boldsymbol{o}^L, \boldsymbol{t}(n))}{\partial \mathbf{W}_{iter}^l} = \frac{\partial J(\boldsymbol{o}^L, \boldsymbol{t}(n))}{\partial \boldsymbol{o}^L} \circ \frac{\partial \boldsymbol{o}^L}{\partial \boldsymbol{v}^L} \frac{\partial \boldsymbol{v}^L}{\partial \mathbf{W}_{iter}^l} \tag{1.13}$$

The back-propagation sensitivity term is defined as:

$$\boldsymbol{\delta}^l = \begin{cases} \frac{\partial J(\boldsymbol{o}^L, \boldsymbol{t}(n))}{\partial \boldsymbol{o}^L} \circ \frac{\partial \boldsymbol{o}^L}{\partial \boldsymbol{v}^L} & \text{if } l = L \\ \mathbf{W}^l \boldsymbol{\delta}^{l+1} \circ \frac{\partial \boldsymbol{o}^l}{\partial \boldsymbol{v}^l} & \text{otherwise} \end{cases} \tag{1.14}$$

where the derivatives of the MSE cost function (1.7) and $\frac{\partial \boldsymbol{o}^l}{\partial \boldsymbol{v}^l}$ are:

$$\frac{\partial J(\boldsymbol{o}^L, \boldsymbol{t}(n))}{\partial \boldsymbol{o}^L} = \boldsymbol{o}^L(n) - \boldsymbol{t}(n) \tag{1.15}$$

$$\frac{\partial \boldsymbol{o}^l}{\partial \boldsymbol{v}^l} = \frac{\partial f(\boldsymbol{v}^l)}{\partial \boldsymbol{v}^l} = f'(\boldsymbol{v}^l) \tag{1.16}$$

Substituting (1.15) and (1.16) into (1.14) gives:

$$\boldsymbol{\delta}^l = \begin{cases} (\boldsymbol{o}^L(n) - \boldsymbol{t}(n)) \circ f'(\boldsymbol{v}^L) & \text{if } l = L \\ \mathbf{W}^l \boldsymbol{\delta}^{l+1} \circ f'(\boldsymbol{v}^l) & \text{otherwise} \end{cases} \tag{1.17}$$

Finally, using $\boldsymbol{\delta}^l$ from (1.17) in (1.13) yields:

$$\frac{\partial J(\boldsymbol{o}^L, \boldsymbol{t}(n))}{\partial \mathbf{W}^l_{iter}} = \boldsymbol{\delta}^l \boldsymbol{o^{l-1}}^T \tag{1.18}$$

where $\frac{\partial J(\boldsymbol{o}^L, \boldsymbol{t}(n))}{\partial \boldsymbol{b}^l_i}$ can be similarly derived as:

$$\frac{\partial J(\boldsymbol{o}^L, \boldsymbol{t}(n))}{\partial \boldsymbol{b}^l_i} = \boldsymbol{\delta}^l \tag{1.19}$$

Substituting (1.18) and (1.19) in (1.5) and (1.6), the weights and bias updates become:

$$\mathbf{W}^l_{iter+1} = \mathbf{W}^l_{iter} - \eta \boldsymbol{\delta}^l \boldsymbol{o^{l-1}}^T \tag{1.20}$$

$$\boldsymbol{b}^l_{iter+1} = \boldsymbol{b}^l_{iter} - \eta \boldsymbol{\delta}^l \tag{1.21}$$

Please note that the $\mathbf{W}^l$ term in (1.17) comes from:

$$\frac{\partial v^l_j}{\partial o^{l-1}_k} = w^l_{kj} \tag{1.22}$$

While $\boldsymbol{o^{l-1}}$ term in (1.18) comes from:

$$\frac{\partial v^l_j}{\partial w^l_{kj}} = o^{l-1}_k \tag{1.23}$$

1.1.6.3 Momentum

There are different variations of the SGD algorithm. One of the methods used to speed up the training is the momentum method [19]. This is a simple extension to SGD that has been successfully implemented for decades [20]. The intuitive idea behind the momentum method is trying to accumulate the derivatives. This accumulation accelerates the training for dimensions in which the gradient is consistently pointing to the same direction. On the other hand, the training is

slower for dimensions where the gradient sign keeps changing. This is done by keeping track of past parameter updates with an exponential decay:

$$\Delta W^l(iter+1) = \mu \Delta W^l(iter) + \eta \frac{\partial J(\boldsymbol{o}^L(n), \boldsymbol{t}(n))}{\partial \mathbf{W}^l_{iter}} \tag{1.24}$$

where $\mu < 1$ is a constant controlling the decay of the previous parameter updates, and $\Delta W^l(0) = \mathbf{0}$

The final SGD with momentum weight update rule is:

$$\mathbf{W}^l_{iter+1} = \mathbf{W}^l_{iter} - \Delta W^l(iter+1) \tag{1.25}$$

The final SGD with momentum bias update rule can be similarly derived.

1.1.7 MNIST Dataset

The dataset used to train the networks in this thesis is the Modified National Institute of Standards and Technology database (MNIST dataset) [21]. MNIST is a large dataset of handwritten digits images which is commonly used to train and test the neural networks or other machine learning classifiers [22–24].

The dataset contains 60,000 training images and 10,000 testing images. The digits are of gray-scale 28×28 images (Fig. 1.4).

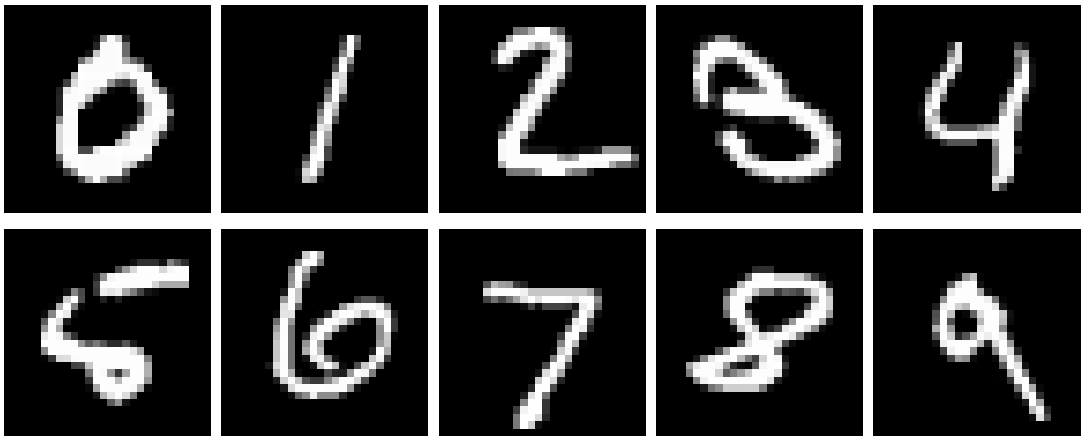


Figure 1.4: Sample images from MNIST dataset

1.2 Related Work

In this work, we study a new neural network architecture. This architecture is based on replacing multiplication with multiplication-free (mf) operator. The mf-operator was first introduced in [25] and used in several applications in image processing [26–28]. The mf-based neural network was first proposed in [29]. Nevertheless, the classification rate of the mf-based neural network was 10% less than the ordinary neural network. Later on, state-of-the-art accuracy was achieved in H&E dataset [30] and in the MNIST dataset [31] using a higher number of neurons in the mf-based neural network than the conventional neural network.

Neural networks have become popular in the last decade, for their achievable accuracies, and because of their ability to generalize and respond to unexpected patterns [32]. This is due to two main reasons. First, the advancements in computing and storage power made it possible to train huge models with millions of parameters. Second, the large amount of stored data today provide enough observations to teach the neural network [33]. However, today's low-power systems such as mobile phones, ARM processors and embedded systems do not have the computational power and battery (energy source) to operate these big models. Therefore, different approaches and solutions are proposed in order to solve the computational power and energy problems of NN [34–37].

In [34], the authors propose using Alphabet Set Multiplier (ASM) where the multiplication is performed using look up tables (alphabet set) followed by shift and add operations. In this method, the efficiency highly depends on the size of the alphabet set. Thus, for efficiency purposes they decreased the size of the alphabet set and approximated the values to the nearest existing multiplication. However, this approach requires a special hardware changes to save the energy.

Han et al. propose a 3-step approach to save energy and storage by discarding insignificant features [35]. First, they train the network. Then, they remove the redundant weights and neurons stochastically to obtain a sparser network.

Finally, they retrain the network to compensate the loss in accuracy caused by the removal of redundant weights and neurons. The method was tested on ImageNet and VGG-16 causing the reduction of the parameter size of between 9X and 13X without any accuracy loss.

Two other methods to save energy and memory space in CNN are proposed in [36]. The first technique, Binary-Weight-Network(BWN), approximates the weights to binary values (1 or -1). Therefore, the inner product is computed using only addition and subtraction operations. The second approach is called XNOR-Networks, where in addition to the weights, the input is also binarized. Thus, the inner product is computed using XNOR and bit counting operations. This method offers X58 faster computation on CPU, although it costs 12% reduction in the accuracy. The BWN reduces the feed-forward multiplication operation to $\boldsymbol{o}^l = f(\boldsymbol{\alpha}^l \circ sgn(\mathbf{W}^{l^T})\boldsymbol{o}^{l-1})$ where $\alpha_j^l = \frac{1}{n}||\boldsymbol{w}_j^l||_1$. Whereas, we propose a different approach to calculate the feed-forward, such that, $\boldsymbol{o}^l = f(\boldsymbol{\alpha}^l \circ (sgn(\mathbf{W}^{l^T})\boldsymbol{o}^{l-1} + \mathbf{W}^{l^T} sgn(\boldsymbol{o}^{l-1}) + \boldsymbol{b}^l))$ with no restrictions on the values of $\boldsymbol{\alpha}^l$. However, for one-hidden-layer NN, we found $\boldsymbol{\alpha}^l = \alpha$ to be sufficient.

Tong et al. attempted the problem of saving power by limiting the mantissa bit length of the floating-point arithmetics [37]. They show that significant power saving can be achieved, without sacrificing any accuracy, by reducing the mantissa bit length. However, this work can be extended by eliminating floating-point completely and replacing it with fixed-point.

Hardware implementation of NN using VHDL is detailed in [38, Ch 10]. First, efficient hardware implementations of the fixed-point non-linear activation functions are proposed. Then, the network architecture is analyzed with the focus on the neuron component. The neuron component is a basic multiply and accumulate unit. Two different architectures are compared: I. using one multiply-accumulate unit for the entire network and II. using one multiply unit and accumulate unit per neuron. The first approach uses minimal area on the board, but takes significantly more cycles due to the full serial implementation; while the implementation is half parallel in the latter. However, the book does not discuss the power consumption or more efficient ways to implement the network.

Different hardware designs of neural networks are proposed and implemented [39, 40]. Cao et al describe the implementation of CNNs into spike-based neuromorphic hardware using Spiking Neural Networks (SNNs). SNNs show 2 orders of magnitude savings in power in simulation. However, they impose many restrictions that limit their performance for harder classification tasks. Moreover, unlike our approach, they need special hardware to be implemented.

Orimo et al. describe the implementation of feed-forward sequential memory network into FPGA [40]. The paper proposed an FPGA architecture to implement neural networks. They discuss the design required resources and logic area but not the power consumption since they are not trying to optimize it.

1.3 Goals and Results

In this work, a novel neural network architecture is devised, in which the neurons implement modified addition operations instead of multiplications as in conventional neurons [29, 31].

Ordinary neurons perform an inner product operation before the nonlinearity. We developed a multiplication free vector product-like operation based on additions and sign operations. We use this new vector product in artificial neurons instead of the regular inner product. Regular inner product induces the ℓ_2 norm, while the new vector product induces the ℓ_1 norm [31].

In this work, we prove that the new architecture has the same classification accuracy achieved by the state-of-the-art NNs for one-hidden-layer networks. We also prove that the new architecture is more energy-efficient compared to the conventional one. To prove the energy efficiency, we built a hardware design of both architectures on FPGA and ASIC using VHDL. The objective of our hardware implementation is to perform inference on the new observations, while the training takes place in MATLAB.

Finally, we show a comparison of the architectures based on fixed-point and floating-point arithmetics and study the effect of quantization and limited precision on accuracy and power consumption.

1.4 Outline

After Chapter 1, this thesis is organized as follows:

In Chapter 2, we introduce the new multiplication free (mf) operator. The properties and challenges of the mf-operator are detailed by comparison with other suggested operators and the already established multiplication. Then, mf-based neural network is introduced with a discussion of the necessary changes in both inference and training of the network.

We continue in Chapter 3 with the hardware design of both ordinary and mf-based neural networks. Three types of hardware neurons are compared for processing time and required hardware resources. Moreover, we discuss the differences between floating-point and fixed-point arithmetics along with the variables quantization and activation functions approximation. We also point out gained advantages of implementing the fixed-point hardware model over the floating-point one. Finally, the simulation and synthesis results of the hardware designs are compared to the MATLAB results as proof of concept.

In Chapter 4 we present and discuss our results. First, we present the accuracy results of both ordinary and mf-based neural networks on the MNIST dataset for different setups. Second, we compare the area and power measurements of the hardware design of both networks. Finally, we present other miscellaneous results, such as fixed-point vs floating-point achieved recognition rates, the distribution and sparsity of the weights, and the effect of pruning the connection.

Finally, we conclude in Chapter 5 with the most important findings of this thesis.

Chapter 2

Neural Networks without Multiplication

In general, neural networks are computationally expensive, where most of the power is consumed by the multiplication operations (as will be shown later). A new operator is introduced to replace multiplication. This work investigates the application of this new multiplication free (mf) operator to neural networks and how the power and accuracy are affected.

This chapter discusses conventional Neural Networks (NN), the new mf operator, its properties, and how it compares to the conventional multiplication. This is followed by a discussion on how to apply the mf operator to the NN to generate Multiplication Free Neural Networks (MFNN) and how to train them. Finally, the chapter also discusses in details different potential operates to replace multiplication.

2.1 Multiplication Free (mf) Operator

The objective of this work is to make NN less computationally expensive by replacing multiplication operations with more efficient operations. However, this improvement should not come at the expense of the network accuracy. Due to the aforementioned reasons, designing the new proposed operator should take into account the computational complexity as well as maintaining some of the multiplication properties. These multiplication properties are:

- sign preservation: for $c = a \times b$, $sgn(c) = sgn(a \times b) = sgn(a) \times sgn(b)$
- contribution from both operand values: in $c = a \times b$ the value of c is composed of the values of both operands. This is unlike min function, for example, where only the value of one operand determines the result's value.
- absorbing element: $a \times 0 = 0 \times a = 0$

For two numbers, a and b, the new proposed binary operator symbolized as $\oplus$ is defined as:

$$a \oplus b = sgn(ab)(|a| + |b|) \tag{2.1}$$

The operator is called multiplication free (mf) operator since it only consists of sign and addition operations. This makes it energy efficient as shown later in Chapter 4. The mf operator , just like multiplication, preserves the sign and has contribution from both operand values. Moreover, the absorbing element is achieved by using the following sign (signum function) definition:

$$sgn(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases} \tag{2.2}$$

Then, using this sign definition there exists an absorbing element that is 0, such that: $a \oplus 0 = 0 \oplus a = 0$

Comparison between different operators and how well they approximate multiplication is presented in Table 2.1, where min, smin (signed min) and binary-weights [36] operations are defined as follows:

$$min(a,b) = \begin{cases} a & \text{if } a \leq b \\ b & \text{if } a > b \end{cases} \tag{2.3}$$

$$smin(a,b) = sgn(ab)min(|a|,|b|) \tag{2.4}$$

$$\text{binary-weights}(a,b) = sign(a)b \tag{2.5}$$

Operations / Properties	$\times$	$\oplus$	min	smin	binary-weights
sign preservation	✓	✓	✗	✓	✓
contribution from both operands values	✓	✓	✗	✗	✗
absorbing element	✓	✓	✗	✓	✗

Table 2.1: Comparison between different operators and multiplication

The comparison in Table 2.1 shows the advantages of replacing multiplication with the mf operator. This work studies the mf operator in detail in the context of neural networks for both achievable accuracy and power consumption. The mf operator is visualized in Fig. 2.1 against multiplication.

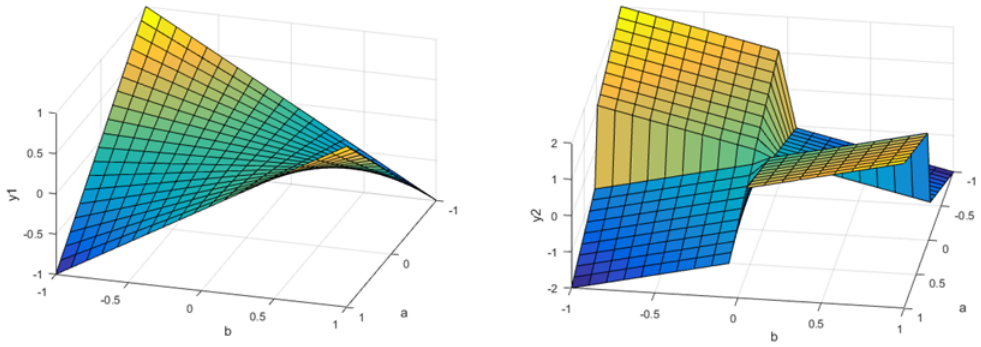


Figure 2.1: Comparison between $a \times b$ (right) and $a \oplus b$ (left)

Using both $sgn(ab) = sgn(a)sgn(b)$ and $|a|\, sgn(a) = a$ facts, the mf operator can be rearranged to:

$$a \oplus b = sgn(a)b + asgn(b) \tag{2.6}$$

This form of writing the mf operator is advantageous to implement it in hardware and software codes. On the one hand, this form can be used as matrix-vector operation as in (2.7). This is beneficial in the software training and inference codes since they are based on matrix-vector multiplication, which could be easily replaced with matrix-vector mf operation. On the other hand, this form can be expressed using only XOR and addition operations in the hardware, as illustrated in Section 3.3.

$$\mathbf{A} \oplus \boldsymbol{b} = sgn(\mathbf{A})\boldsymbol{b} + \mathbf{A}sgn(\boldsymbol{b}) \tag{2.7}$$

where $\mathbf{A}$ is a matrix of size $N \times M$ and $\boldsymbol{b}$ is vector of length M.

2.2 Multiplication Free Neural Netowrk (MFNN)

After demonstrating the proposed mf operator, its properties and how it can approximate the multiplication, we applied it to neural networks. By replacing multiplication in the feed-forward calculations in (1.2) it yields:

$$o_j^l = f\left(\sum_{i=1}^{N_{l-1}} w_{ij}^l \oplus o_i^{l-1} + b_j^l\right) \tag{2.8}$$

Moreover, the compact matrix-vector notation in (1.3) becomes:

$$\boldsymbol{o}^l = f\left(\mathbf{W}^{l^T} \oplus \boldsymbol{o}^{l-1} + \boldsymbol{b}^l\right) \tag{2.9}$$

The network represented by (2.8) and (2.9) does not contain any multiplications. Thus, it is called Multiplication Free Neural Network (MFNN).

2.2.1 SGD with Back-Propagation in MFNN

For training the MFNN, we used Stochastic Gradient Descent (SGD) with back-propagation. The algorithm is explained in detail in section 1.1.6. Moreover, the modification to the NN feed-forward with the new mf operator has to be incorporated in the training of the new MFNN.

First, we isolate the sum term in 2.8 to v_j^l as follows:

$$v_j^l = \sum_{i=1}^{N_{l-1}} w_{ij}^l \oplus o_i^{l-1} + b_j^l \tag{2.10}$$

Then, using the mf-operator-based definition of v_j^l the derivatives in (1.22) and (1.23) become:

$$\frac{dv_j^l}{do_k^{l-1}} = sgn(w_{kj}^l) + 2\delta(o_k^{l-1})w_{kj}^l \tag{2.11}$$

$$\frac{dv_j^l}{dw_{kj}^l} = 2\delta(w_{kj}^l)o_k^{l-1} + sgn(o_k^{l-1}) \tag{2.12}$$

where $\delta(.)$ is the Dirac delta function [41]. $\delta(x) = 0$ almost everywhere except for $x = 0$. In practice, exact values of zero are unlikely to occur. Therefore, the Dirac delta term can be approximated as $\delta x \approx 0$ and dropped out. The updated derivatives of (2.11) and (2.12) then become:

$$\frac{dv_j^l}{do_k^{l-1}} = sgn(w_{kj}^l) \tag{2.13}$$

$$\frac{dv_j^l}{dw_{kj}^l} = sgn(o_k^{l-1}) \tag{2.14}$$

With these changes, the back-propagation sensitivity term in (1.17) of MFNNs is defined as:

$$\boldsymbol{\delta}^l = \begin{cases} (\boldsymbol{o}^L - \boldsymbol{t}(n)) \circ f'(\boldsymbol{v}^L) & \text{if } l = L \\ sgn(\mathbf{W}^l)\boldsymbol{\delta}^{l+1} \circ f'(\boldsymbol{v}^l) & \text{otherwise} \end{cases} \tag{2.15}$$

Additionally, the weights and bias updates in (1.20) and (1.21) become:

$$\mathbf{W}_{iter+1}^l = \mathbf{W}_{iter}^l - \eta\boldsymbol{\delta}^l sgn(\boldsymbol{o}^{\boldsymbol{l-1}})^T \tag{2.16}$$

$$\boldsymbol{b}_{iter+1}^l = \boldsymbol{b}_{iter}^l - \eta\boldsymbol{\delta}^l \tag{2.17}$$

Please note that no other changes to the SDG are required. The derivatives of the cost and activation functions stays the same.

2.2.2 Normalization

Fig. 2.1 shows that the mf operator is discontinuous around the axes. The discontinuity is larger for bigger a or b. This makes the operator sensitive during training, since a small modification of the operands (i.e., the weights during training) can have a significant impact on the result. This is case when the change changes the sign of the weight. On the other hand, this is not the case for normal multiplication, as it is continuous over the whole range.

Moreover, using the mf operator yields larger values than multiplication for operand values less than 1. That is, for $|a| \, and \, |b| \leq 1$, $|ab| \leq |a|$ and $|ab| \leq |b|$ while $|a \oplus b| \geq |a|$ and $|a \oplus b| \geq |b|$. These individual larger values lead to a far larger overall neuron sum i.e., v_j. Ideally, it is preferred to have a v_j value in the desired region and avoid the saturation region (see Fig. 2.2). Saturation region is the region where the derivative of the activation function is zero. The zero derivative values lead to no weight update during the training of the network. This is due to the $f'(\boldsymbol{v}^l)$ term in the derivatives of the update rule in (1.17) and (2.15).

To solve these issues, a layer normalization term α is introduced to normalize down the sum values (v_j) before being passed through the activation functions.

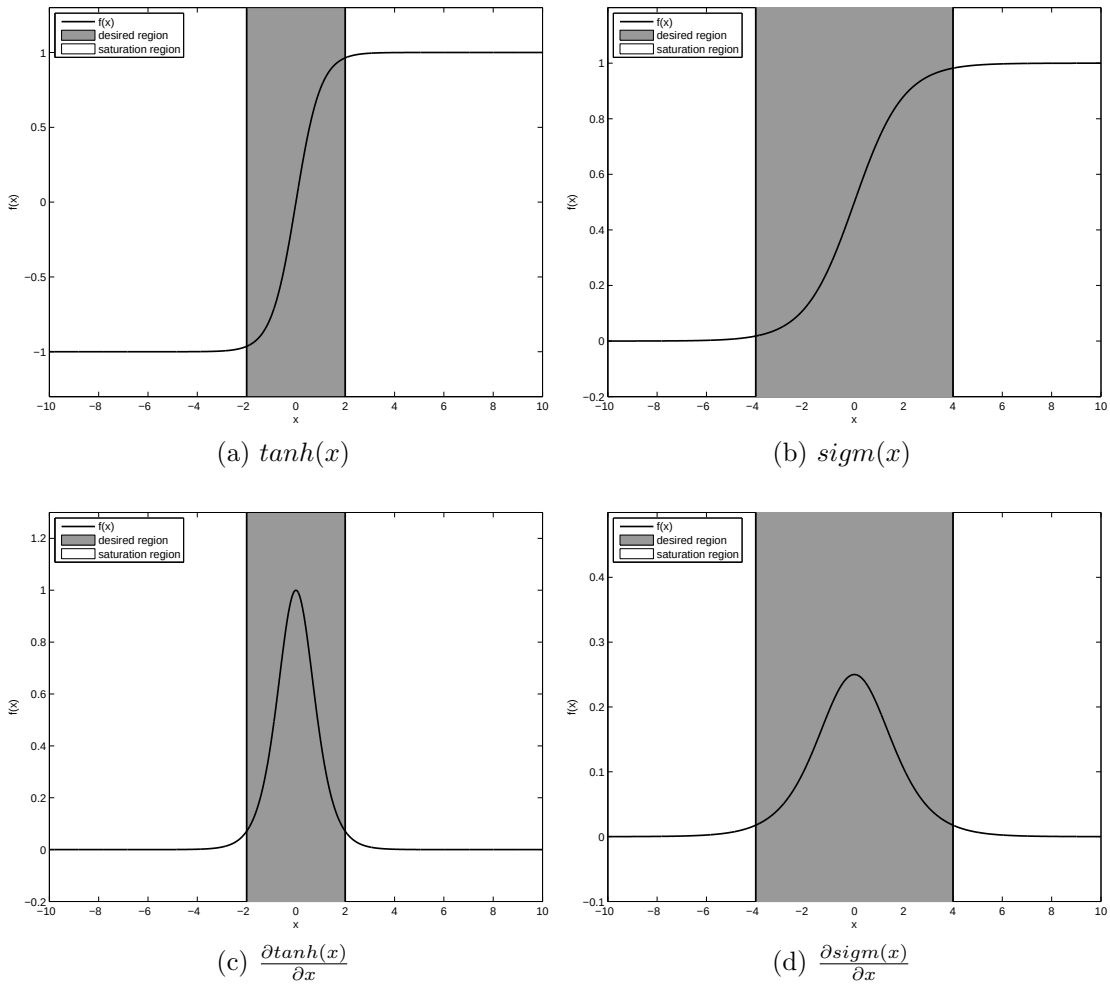
Adding the layer normalization term α to (2.8) and yields:

$$o_j^l = f\left(\frac{1}{\alpha}\left(\sum_{i=0}^{N_{l-1}-1} w_{ij}^l \oplus o_i^{l-1} + b_j^l\right)\right) \tag{2.18}$$

And in matrix-vector notation in (2.9) becomes:

$$\boldsymbol{o}^l = f\left(\frac{1}{\alpha}\left(\mathbf{W}^{l^T} \oplus \boldsymbol{o}^{l-1} + \boldsymbol{b}^l\right)\right) \tag{2.19}$$

In addition to the α term, the input values are scaled down with an input normalization factor β. Finally, the weights are initialized to small values to insure convergence.



(a) $tanh(x)$ (b) $sigm(x)$

(c) $\frac{\partial tanh(x)}{\partial x}$ (d) $\frac{\partial sigm(x)}{\partial x}$

Figure 2.2: Activation functions with their derivatives

α and β values are restricted to powers of 2. Thus, the division is performed using shift operations only. Hence, the network is still a multiplication free, even with the added α and β normalization terms.

The values of α and β are chosen experimentally using validation dataset. The normalization can extended for harder task. for example, the scalar α for the whole network could be extended to a scalar for every neuron α_j^l. Furthermore, these parameters could be made trainable using the back-propagation algorithm.

2.2.3 SGD and Back-Propagation in MFNNs with Normalization

Isolating the sum term v_i^l as in (2.10) yields:

$$v_j^l = \frac{1}{\alpha}\left(\sum_{i=0}^{N_{l-1}-1} w_{ij}^l \oplus o_i^{l-1} + b_j^l \right) \tag{2.20}$$

The added α normalization term is to be reflected to the final derivatives in (2.13) and (2.14) as follows:

$$\frac{dv_j^l}{do_k^{l-1}} = \frac{1}{\alpha}\left(sgn(w_{kj}^l) \right) \tag{2.21}$$

$$\frac{dv_j^l}{dw_{kj}^l} = \frac{1}{\alpha}\left(sgn(o_k^{l-1}) \right) \tag{2.22}$$

With these changes, the back-propagation sensitivity term in (2.23) of MFNNs is defined as:

$$\boldsymbol{\delta}^l = \begin{cases} (\boldsymbol{o}^L - \boldsymbol{t}^n) \circ f'(\boldsymbol{v}^L) & \text{if } l = L \\ \frac{1}{\alpha} sgn(\mathbf{W}^l)\boldsymbol{\delta}^{l+1} \circ f'(\boldsymbol{v}^l) & \text{otherwise} \end{cases} \tag{2.23}$$

Additionally, the weights and bias updates in (1.20) and (1.21) become:

$$\mathbf{W}_{iter+1}^l = \mathbf{W}_{iter}^l - \eta\boldsymbol{\delta}^l \frac{1}{\alpha} sgn(\boldsymbol{o^{l-1}})^T \tag{2.24}$$

$$\boldsymbol{b}_{iter+1}^l = \boldsymbol{b}_{iter}^l - \eta\boldsymbol{\delta}^l \tag{2.25}$$

Chapter 3

Hardware design

After training, testing, and verifying that the new proposed MFNN achieves the same accuracy as NN, both network architectures are to be examined for power consumption and computational complexity. For this reason, both architectures were implemented into hardware using VHSIC Hardware Description Language (VHDL). The VHDL codes are synthesized for Field-Programmable Gate Arrays (FPGA) and Application-Specific Integrated Circuit (ASIC) technologies.

The FPGA is used to test the hardware network designs, i.e., to make sure the hardware inference works as expected and produces the same results as the software. In addition, using the FPGA ensures that the model is feasible in terms of hardware resources and timing constraints. Virtex-7 XC7VX485T-2FFG1761C FPGA board is used for the hardware testing. The board specifications and components are analyzed in Section 3.1. On the other hand, the ASIC technologies synthesis is mainly used for power and area measurements. The power measurements on ASIC are more reliable and accurate because of the synthesis only generates the specific logic required. This avoids the unwanted power overhead from the unused FPGA resources/trails.

The hardware design is responsible for the inference only, i.e., no training of the networks is done using the hardware design. The networks are trained beforehand on more powerful computers using software languages, e.g., MATLAB or Python. After the training is complete, the trained network parameters (weights and biases) are loaded into the FPGA RAMs to be used in the real-time inference. Loading the parameters takes place both at power up or later during run time.

In this section, we discuss the overall hardware design and some of the challenges and trade-offs between hardware resources, processing time (latency) and achievable frequency. In addition, several designs of both conventional and multiplication free neurons are detailed. We also explain how to implement the nonlinear activation functions in hardware and fixed-point arithmetics. Finally, we describe the power and area measurements process of the hardware design.

3.1 VC707 Evaluation Board

The VC707 evaluation board was used to test the hardware design. The board contains a vertex-7 (XC7VX485T) as an FPGA along with other peripheral components to facilitate the FPGA [42]. Some of these components are: clock generators, USB JTAG, LCD, LEDs, push buttons, switches and I^2C bus. Fig. 3.1 shows the schematic of the VC707 evaluation board with all the peripherals, while Fig. 3.2 shows the actual VC707 evaluation board with the components highlighted.

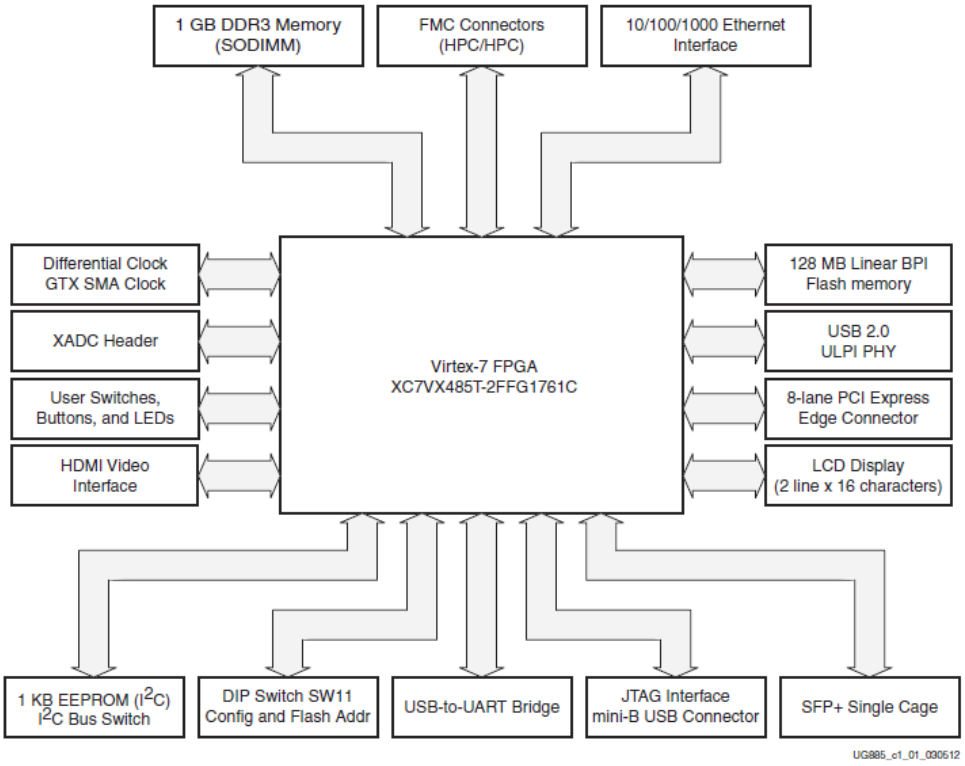


Figure 3.1: VC707 Evaluation Board schematic [42]

The Virtex-7 (XC7VX485T) FPGA has the following specifications [43]:

- Logic Cells: 485,760
- Slices[1]: 75,900
- DSP Slices[2]: 2,800
- Block RAM Blocks[3]: 2,060 (18 Kb) or 1,030 (36 Kb)
- Block RAM Max Size: 37,080 (Kb)
- Max User I/O: 700 (Distributed in 14 banks)

[1]Each 7 series FPGA slice contains four LUTs and eight flip-flops; only some slices can use their LUTs as distributed RAM.

[2]Each DSP slice contains a pre-adder, a 25 x 18 multiplier, an adder, and an accumulator.

[3]Block RAMs are fundamentally 36 Kb in size; each block can also be used as two independent 18 Kb blocks.

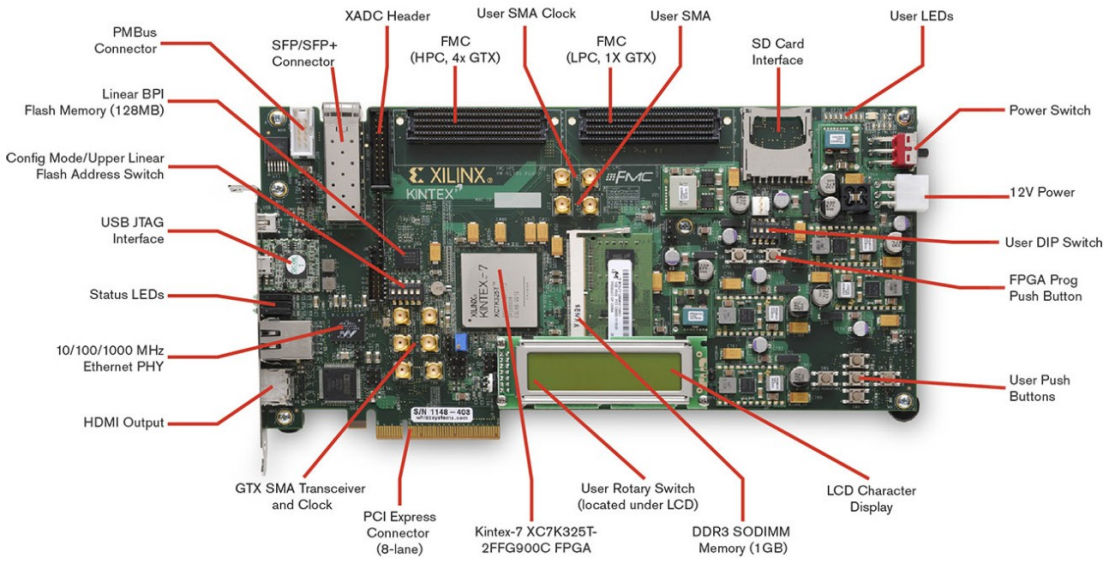


Figure 3.2: VC707 Evaluation Board [42]

3.2 Overall Hardware Design

In this section, we discuss the overall hardware design (Fig. 3.3) with all the components. Moreover, we shine some light on some of the challenges faced to realize the hardware model.

The hardware model is implemented using VHDL and organized in components as follows. The basic backbone components, i.e. the neurons (detailed in section 3.4), are instantiated and organized inside layers. Each layer instantiates N_l neurons, where N_l is the number of neurons in that l layer. The layers are fully connected to each other. Any l_{th} layer (except input and output layers) is inter-connected to two layers, $l-1$ and $l+1$. The input layer ($l=1$) is only connected to the $l=2$ layer; while the output layer is only connected to $L-1$ layer.

All the layers and their inter-connections are contained in the CTRL block. The CTRL block also contains the finite state machine (FSM) that is responsible for controlling the entire network.

Two types of storage units are used in the model: RAM and ROM. The ROM is used to store the input data (MNIST images in this case). The RAMs are used

to store the biases and the weights of the connections between the layers, where RAM l stores the weights of the connections between layers l and $l-1$.

In the real time system, the input data are to be fed serially through another data acquisition model. The current setup is build to test the hardware neural networks; therefore, the images are stored inside a ROM.

The weights and biases are also stored in VHDL package files to be loaded into the RAMs on start-up. In addition to start-up, the weights and biases can be loaded into the FPGA RAMs anytime using the I^2C bus. This feature allows updating the weights or biases whenever needed. For example, the weights could be updated after obtaining more training data and retraining the network.

The FSM is responsible for controlling the network's processing. It handles the data flow from the storage (ROM and RAM) into the layers and between the adjacent layers. The outputs of layer l are inputs to layer $l+1$. Thus, the processing of layer l should be finished completely before the FSM can start processing layer $l+1$.

The FSM controlling a one-hidden-layer network comprises of the following states that instantiate each other in the order they are mentioned:

- Initialization: in this state, the weights are loaded at power up or later during operation.
- IDLE: waits for a trigger from Next IMG signal.
- Read IMG and 2^{nd} layer processing: reads the IMG pixels from ROM and processes them in the 2^{nd} layer neurons.
- 3^{rd} layer processing: processes the 3^{rd} layer neurons where the inputs to the 3^{rd} layer are the 2^{nd} layer outputs.
- Classify IMG: determines the final classification of the network in the MAX block component.

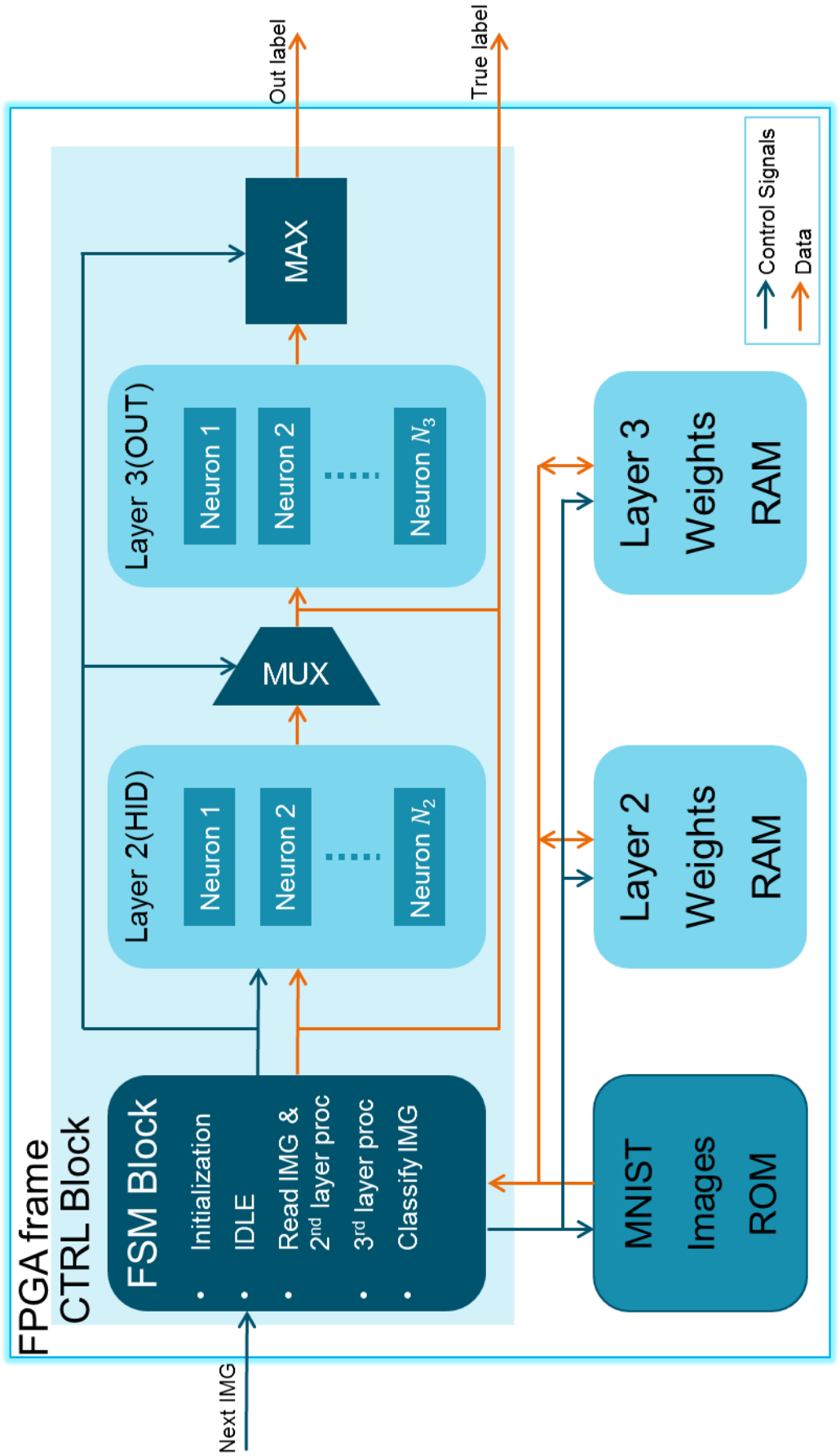


Figure 3.3: Hardware design diagram

The FSM reads the input data from the ROM sequentially (one pixel every clock cycle). For pixel p_i being processed, the FSM reads all the weights connected to it (i.e all w_{ij}, for $1 \leq j \leq N_2$ where N_2 is the No. of neurons in the first layer). After fetching the values, the FSM passes the data (i.e., p_i and w_{ij}) to the neurons in the next layer.

The neurons process the data in each layer sequentially (discussed in Section 3.4). The processing is done exactly as the input layer. However, instead of reading p_i for the ROM, the outputs of the previous layer are passed sequentially through a MUX controlled by the FSM as well.

The classification task is to be carried out after finishing the processing of the last layer. The outputs of the last layer are passed to the MAX block. MAX block, also controlled by the FSM, compares the output of the network and produces the final classification of the network by reporting the label of the maximum output.

All of the FPGA related components are organized in the top level component FPGA_frame. Some of these are: RAMs, ROMs, Switches, Clock Generators, I^2C interface... etc. This enables the configuration of the network into different hardware platforms. It is done by changing the FPGA_frame to NEW_DEVICE_frame incorporating all the components there to the new hardware platform.

Fig. 3.3 illustrates a detailed diagram of the hardware design with all the components highlighted.

3.3 Hardware Implementation of The mf Operator

Two implementations of the mf operator were tested in the hardware trying to minimize the power consumption, the two implementation add_1 and add_2 are as follows:

```
function add_1 (A, B)
    return(sgn(A)B + sgn(B)A);
    //sgn(x) retuns a two bit vector to represent -1,0,1
end function add_1;

function add_2 (A, B)
   if (A = 0 or B = 0) then
         return 0;
   else
         return (A(H) XOR B) + (B(H) XOR A) + A(H) + B(H);
         // A(H) and B(H) are the most significant bits of
         // A and B vectors respectively.
   end if;
end function add_2;
```

The second implementation (add_2) proved to be more energy efficient since it used only bit operations (XOR) and addition to implement the two's complement addition with the special case of 0 handled. Whereas, add_1 uses a sign function that generates a 2 bit vector which requires a additional hardware logic to implement the result. Therefore, add_2 function was used in the hardware design to generate the power results.

The basic component that determines the efficiency of the whole design is the neuron. Therefore, it is discussed in details in the next section.

3.4 Neuron Hardware Design

The neuron component is the backbone of the whole neural network design. It does the heavy work and determines the efficiency of the system. The hardware implementation of the neurons carries out the calculations in (1.2) and (2.18).

Two hardware designs have been implemented to carry out the neuron's calculations: I. Parallel neuron and II. Serial neuron. Both of these designs have trade-offs between computation time and used resources. On one hand, the parallel implementation uses more resources but processes the data much faster (1 clock per layer). On the other hand, the serial implementation requires more time (1 clock per input) but it uses less resources.

We will also introduce a third implementation III. Hybrid neuron of both previous implementations. This has not been implemented yet. However, we present it here to show the benefits that can be achieved by implementing this neuron in the future.

Fig. 3.4, Fig. 3.5 and Fig. 3.6 (parallel, serial and hybrid neurons diagrams, respectively) illustrate the hardware diagram of the j_{th} neuron in layer $l+1$. There are N_l neurons in the previous layer (l). This means there are N_l inputs and their respective N_l weight connections between the layer l and one neuron in layer $l+1$. These numbers will be used to calculate the hardware complicity, i.e., the hardware components used to construct the network in this design.

3.4.1 Parallel Neuron Hardware Design

This hardware implementation carries out all the multiplications or the multiplication free operations at the same time. Fig. 3.4 illustrates the hardware diagram of one parallel neuron, where OP is a binary operator defined as conventional multiplication in NNs or mf operator in MFNNs. Adder is a full Adder component that adds 2 operands (not more). *alpha* is the normalization factor

in MFNNs. $alpha = 1$ in the case of NNs. Finally, $f(.)$ is a non-linear activation function.

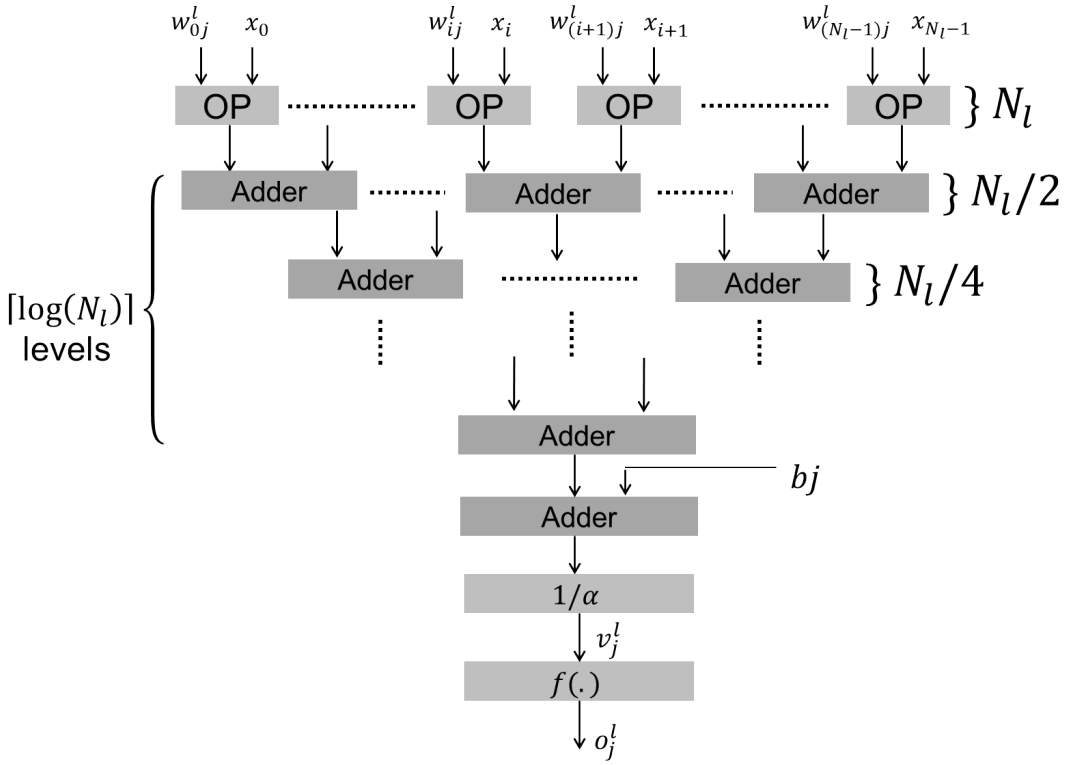


Figure 3.4: Parallel neuron diagram of the j_{th} neuron in layer $l+1$

For each neuron in layer N_{l+1}, there are N_l OP operations carried out simultaneously. There are $log_2(N_l)$ levels of Adders, where each level k contains $N_l/2^k$ Adders (and one Adder for the bias term). In total, these make $1 + \sum_{k=1}^{log_2(N_l)} N_l/2^k = N_l + 1$ Adders. Additionally, there is one division by α, however, as α is restricted to powers of 2, the division can be reduced to only shift operations. Finally, each neuron carries out one $f(.)$ (Hardware implementation of nonlinear activation functions is discussed in 3.5.2).

In the parallel implementation, all these operations take place at the same time (in one clock cycle). Thus, dedicated hardware resources are to be allocated accordingly. In total, every neuron in layer N_{l+1} requires total hardware resources ($H^{l+1}_{Neuron}(j)$) to be built:

$$H^{l+1}_{Neuron}(j) = N_l\, H_{OP} + N_l\, H_{Adder} + H_{Shift} + H_F \tag{3.1}$$

where H_{OP}, H_{Adder}, H_{Shift} and H_F stand for the hardware resources required to implement OP, Adder, $1/\alpha$ and $f(.)$ operations, respectively.

Moreover, there are N_{l+1} neurons in layer $l+1$. Therefore, the amount of hardware resources required to build one layer ($H_{Layer}(l+1)$) is:

$$\begin{aligned} H_{Layer}(l+1) &= \sum_{j=0}^{N_{l+1}-1} H_{Neuron}^{j+1}(j) \\ &= N_{l+1}(N_l\, H_{OP} + N_l\, H_{Adder} + H_{Shift} + H_F) \end{aligned} \tag{3.2}$$

Note that $H_{Neuron}^{l+1}(j)$ is constant $\forall j \in layer(l+1)$.

Finally, the amount of hardware resources required to build an entire parallel-neuron-based network with M layers ($H_{Net}^{P}(M)$) is:

$$H_{Net}^{P}(M) = \sum_{l=2}^{L} H_{Layer}(l)) \tag{3.3}$$

From the previous analysis, we see that the parallel neuron implementation requires a lot of hardware resources. However, when this implementation was tested it proved to be impractical. The required hardware resources are too big for one FPGA to handle. Even a relatively small one-hidden-layer network with 100 neurons did not fit in the FPGA.

This parallel implementation requires one cycle to compute one layers outputs. Therefore, an entire network with L layers can be computed in $L-1$ cycles. Moreover, compared to the serial and hybrid implementations, the parallel implementation is less complex since there is no need for a finite state machine or controlling signals. That is because there is no Scheduling needed. In the contrary, the serial and hybrid designs requires a FSM to control the network for scheduling the data.

To summarize, the parallel neuron implementation is fast to compute and easies to implement(less complex). However, it requires a lot of hardware resources. These hardware resources requirements could not be met using the Virtex-7 (XC7VX485T) FPGA. For this reason, the parallel-neuron-based network is impractical to be synthesized.

3.4.2 Serial Neuron Hardware Design

After testing the design of the parallel neuron and finding out it is impractical, we designed a serial hardware neuron. Fig. 3.5 illustrates the hardware diagram of one serial neuron, where OP is a binary operator defined as conventional multiplication in NNs or mf operator in MFNNs. Adder is a full Adder component that adds 2 operands (not more). *alpha* is the normalization factor in MFNNs. $alpha = 1$ in the case of NNs. Finally, $f(.)$ is the non-linear activation function.

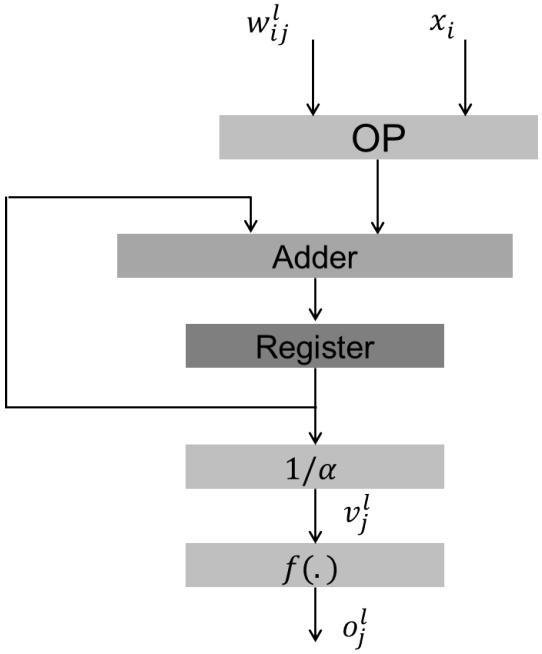


Figure 3.5: Serial neuron diagram of the j_{th} neuron in layer $l + 1$

The previous parallel design (Fig. 3.4) carries out all the operations (OP) for all input data at the same time(one clock cycle). In contrast, the serial neuron design carries out one OP operation per clock cycle. Therefore, the whole neuron processing takes place over sequential clock cycles. The result of OP operation is accumulated in the register by adding it up to the previous partial sum. The final sum is scaled with α and then passed through the nonlinear activation function $f(.)$. This implements (1.2) and (2.18).

The Register is initialized to b_j so that it is accumulated in the sum.

The serial neuron in layer $l+1$ needs $N_l + 1$ clock cycles to finish the computation. N_l cycles are needed to carry out N_l OP on the input data from the previous layer. The additional one clock is needed to process the sum with α and $f(.)$. Unlike the parallel design, since the computations take place sequentially in different clock cycles, there is only one OP and Adder hardware resources needed. This model requires an additional Register to store the partial sum. The serial neuron requires the same hardware resources as the parallel neuron to process α and $f(.)$.

In total, every neuron in layer N_{l+1} requires total hardware resources ($H^{l+1}_{Neuron}(j)$) to be built:

$$H^{l+1}_{Neuron}(j) = H_{OP} + H_{Adder} + H_{Register} + H_{Shift} + H_F \tag{3.4}$$

where H_{OP}, H_{Adder},$H_{Register}$, H_{Shift} and H_F stand for the hardware resources required to implement OP, Adder, Register, $1/\alpha$ and $f(.)$ operations, respectively.

Moreover, there are N_{l+1} neurons in layer $l+1$. Therefore, the amount of hardware resources required to build one layer ($H_{Layer}(l+1)$) is:

$$\begin{aligned} H_{Layer}(l+1) &= \sum_{j=0}^{N_{l+1}-1} H^{l+1}_{Neuron}(j) \\ &= N_{l+1}(H_{OP} + H_{Adder} + H_{Register} + H_{Shift} + H_F) \end{aligned} \tag{3.5}$$

Note that $H^{l+1}_{Neuron}(j)$ is constant $\forall j \in layer(l+1)$.

Finally, the amount of hardware resources required to build an entire serial-neuron-based network with L layers ($H^S_{Net}(L)$) is:

$$H^S_{Net}(L) = \sum_{l=2}^{L} H_{Layer}(l)) \tag{3.6}$$

The hardware resources required by the serial-neuron-based network ($H^S_{Net}(L)$) compared to the parallel-neuron-based one ($H^P_{Net}(L)$) are as follows:

$$\begin{aligned} \frac{H^P_{Net}(L)}{H^S_{Net}(L)} &= \frac{\sum_{l=2}^{L} N_{l+1}(N_l\, H_{OP} + N_l\, H_{Adder} + H_{Shift} + H_F)}{\sum_{l=2}^{L} N_{l+1}(H_{OP} + H_{Adder} + H_{Register} + H_{Shift} + H_F)} \\ &\approx \frac{\sum_{l=2}^{L} N_l N_{l+1}}{\sum_{l=2}^{L} N_{l+1}} \end{aligned} \tag{3.7}$$

The Approximation is valid for $N_l >> 1$ where:

$$N_l\, H_{OP} + N_l\, H_{Adder} >> H_{Register} + H_{Shift} + H_F \tag{3.8}$$

For one layer the hardware usage factor becomes N_l.

From the previous analysis, we see that the serial neuron implementation requires less hardware resources than the parallel one. Therefore, the serial implementation is practical in the sense there are enough hardware resources in the FPGA for the serial-neuron-based network. However, this comes at the expense of the execution time. The serial-neuron-based layer $(l + 1)$ takes $N_l + 1$ cycles to be processed in comparison to one cycle for the parallel-neuron-based layer.

3.4.3 Hybrid Neuron Hardware Design

The two previous hardware neuron designs show two extremes. On one hand, the parallel design carries out all the computation parallelly in one clock cycle. On the other hand, the parallel design carries out one OP operation per cycles, thus, one layer takes many clock cycles to finish processing.

The parallel design is more efficient in terms of latency as it computes and processes the data faster than the serial one. However, the design proved to be impractical since there are not enough hardware resources for all the parallel processing units. On the contrary, the serial design is tested to be practical although it requires more processing time. However, as discussed in the synthesis results later (3.6), the serial design does not utilize the FPGA 100%. Therefore, a hybrid hardware neuron between serial and parallel neurons is designed to utilize the hardware resources.

Please note that the hybrid neuron is designed for future work to enhance the network processing time by utilizing more hardware resources. This design is not implemented or tested yet.

Fig. 3.6 illustrates the hardware diagram of one hybrid neuron, where OP is

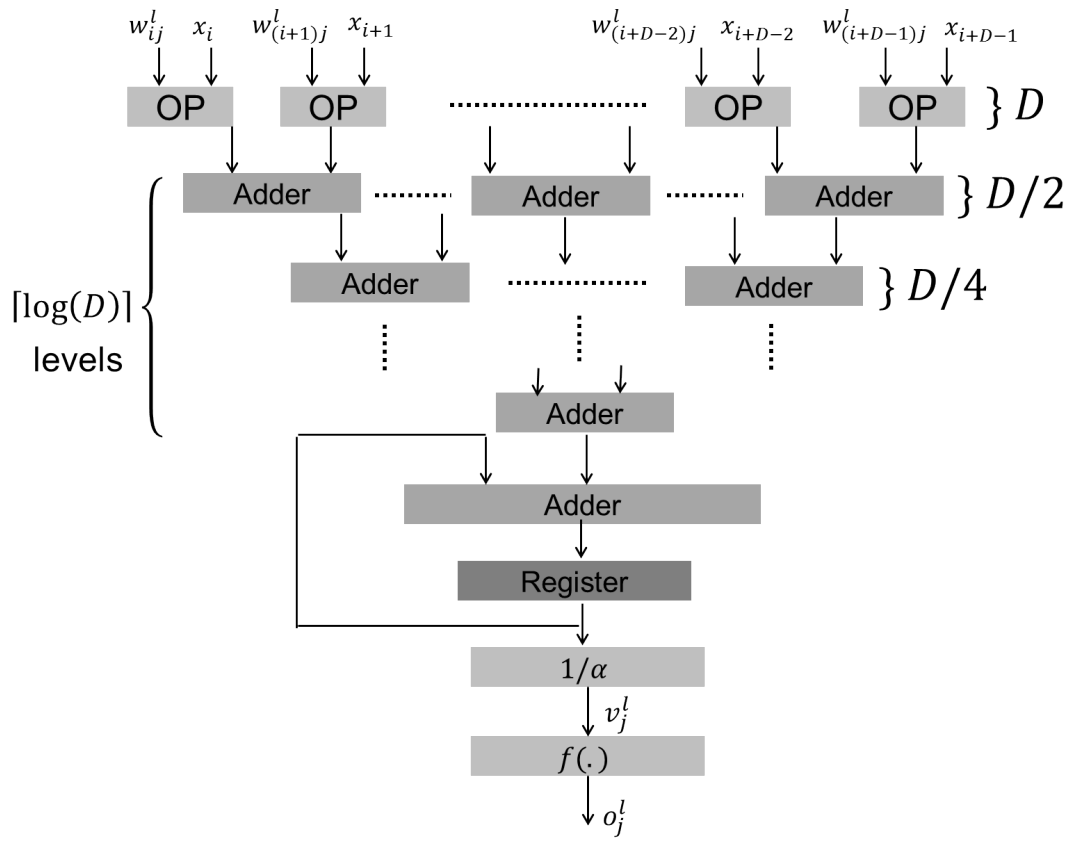


Figure 3.6: Hybrid neuron diagram of the j_{th} neuron in layer $l+1$ with a parallel degree of D OP operations per cycle

a binary operator defined as conventional multiplication in NNs or mf operator in MFNNs. Adder is a full Adder component that adds 2 operands (not more). *alpha* is the normalization factor in MFNNs. $alpha = 1$ in the case of NNs. Finally, $f(.)$ is the non-linear activation function.

Similar to the serial neuron, the hybrid neuron carries out the computations sequentially over multiple clock cycles. However, instead of only one OP operation per clock cycle in the serial neuron, the hybrid neuron carries out D OP operations per clock cycle. The results of the D OP operations are added up using $log_2(D)$ levels of Adders, where, every level k contains $D/2^k$ Adders. In total, these make $\sum_{k=1}^{log_2(D)} D/2^k = D$ Adders. This result is accumulated in the register by adding it up to the previous partial sum. The final sum is scaled with α and then passed through the nonlinear activation function $f(.)$. This implements (1.2) and (2.18).

The Register is initialized to b_j so that it is accumulated in the sum.

In the hybrid implementation, D operations and $(D+1)$ additions take place at the same time (in one clock cycle). Thus, dedicated hardware resources are to be allocated accordingly. The hybrid neuron in layer $l+1$ needs $\lceil N_l/D \rceil + 1$ clock cycles to finish the computation. $\lceil N_l/D \rceil$ cycles are needed to carry out N_l OP (D OP operations/cycle $\times$ $\lceil N_l/D \rceil$ cycles) on the input data from the previous layer. The additional one clock is needed to process the sum with α and $f(.)$.

In total, every neuron in layer N_{l+1} requires total hardware resources ($H^{l+1}_{Neuron}(j)$) to be built:

$$H^{l+1}_{Neuron}(j) = D\, H_{OP} + (D+1)\, H_{Adder} + H_{Register} + H_{Shift} + H_F \tag{3.9}$$

where H_{OP}, H_{Adder},$H_{Register}$, H_{Shift} and H_F stand for the hardware resources required to implement OP, Adder, Register, $1/\alpha$ and $f(.)$ operations, respectively. D is the parallel degree of the hybrid design.

Moreover, there are N_{l+1} neurons in layer $l+1$. Therefore, the amount of hardware resources required to build one layer ($H_{Layer}(l+1)$) is:

$$\begin{aligned} H_{Layer}(l+1) &= \sum_{j=0}^{N_{l+1}-1} H^{l+1}_{Neuron}(j) \\ &= N_{l+1}(D\, H_{OP} + (D+1)\, H_{Adder} + H_{Register} + H_{Shift} + H_F) \end{aligned} \tag{3.10}$$

Note that $H^{l+1}_{Neuron}(j)$ is constant $\forall j \in layer(l+1)$. Finally, the amount of hardware resources required to build an entire hybrid-neuron-based network with L layers ($H^{Hyb}_{Net}(L)$) is:

$$H^{Hyb}_{Net}(L) = \sum_{j=2}^{L} H_{Layer}(l)) \tag{3.11}$$

The hardware resources required by the serial-neuron-based network ($H^{S}_{Net}(L)$) compared to the hybrid-neuron-based one ($H^{Hyb}_{Net}(L)$) are as follows:

$$\begin{aligned} \frac{H^{Hyb}_{Net}(L)}{H^{S}_{Net}(L)} &= \frac{\sum_{j=2}^{L} N_{l+1}(D\, H_{OP} + (D+1)\, H_{Adder} + H_{Shift} + H_F)}{\sum_{j=2}^{L} N_{l+1}(H_{OP} + H_{Adder} + H_{Register} + H_{Shift} + H_F)} \\ &\approx D \end{aligned} \tag{3.12}$$

The Approximation is valid for $D >> 1$ where:

$$D\ H_{OP} + (D+1)\ H_{Adder} >> H_{Register} + H_{Shift} + H_F \tag{3.13}$$

From the previous analysis, we see that the hybrid neuron implementation requires less hardware resources than the parallel one. Therefore, the hybrid implementation is practical in the sense there are enough hardware resources in the FPGA for the hybrid-neuron-based network. Moreover, the hybrid neuron implementation requires less processing time per layer since it utilizes the hardware.

Table 3.1 compares different neural networks w.r.t their hardware neuron designs (serial, parallel and hybrid hardware neuron designs).

	Relative required hardware $\frac{H_{Net}(L)}{H^S_{Net}(L)}$	**Processing time**	**Pratical**	**Simplicity**
Serial neuron	1	$\sum_{j=1}^{L-1} N_l + L$	✓	Complex
Parallel neuron	$\frac{\sum_{l=2}^{L} N_l N_{l+1}}{\sum_{l=2}^{L} N_{l+1}}$	L	✗	Simple
Hybrid neuron	D	$\sum_{j=1}^{L-1} \lceil N_l/D \rceil + 1$	✓	Complex

Table 3.1: Comparison between neural networks with different hardware neuron designs

Please note that the processing time in Table 3.1 is only of the layers. The total processing time of the network should account for the MAX calculations. The total processing time then becomes:

$$\text{total processing time} = \text{processing time} + N_L \tag{3.14}$$

where N_L is the No. of neurons in the last layer, which is also the No. of classes M. This is because the MAX block carries out N_L comparisons to classify the image. Each comparison takes place in one clock cycle.

To sum up, the comparison in Table 3.1 shows the advantages of the parallel-neuron-based network in terms of processing time and the simplicity of the hardware design. However, this design proved impractical. Therefore, the serial-neuron-based network is selected instead.

For future work, a new hybrid-neuron-based network is analyzed as a compromise between the serial and hardware neuron design. The hybrid design exploits the hardware resources of the FPGA, that are not utilized by the serial design.

3.5 Floating-Point vs. Fixed-Point

The main goal of this work is to reduce the power consumed by the neural network. Using fixed-point variables and arithmetic over floating-point saves a significant amount of power [37, 44]. Using fixed-point instead of floating-point yields less logic resources usage which inherently leads to lower power consumption [45].

Using fixed-point instead of floating-point does not suppose large accuracy losses, e.g. image classification was found to only require INT8 or less fixed-point precision to keep satisfactory recognition rates [46, 47]. We show later in Section 4.3.1 that fixed-point quantization for both NN and MFNN achieves the floating-point recognition performance.

The used fixed-point word is defined as IL.FL, where IL (integer length) and FL (fractional length) are the number of bits used for the integer part and the fractional part of the word, respectively. The total word length (WL) is then calculated as follows: WL = IL + FL + 1 (the additional one is the sign bit). Moreover, the IL is allowed to have negative values. In such a case, the IL most significant bits of the fractional part are not used.

3.5.1 Quantization

The hardware design is implemented using fixed-point variables and arithmetics due to the aforementioned reasons. However, the trained networks, NN and MFNN, are implemented on MATLAB using floating-point variables and arithmetics. Therefore, quantization of these trained networks is needed.

All the inputs (images) are quantized using 8 bits. The images are then stored in fixed-point format of variable length. The input and neuron outputs are quantized using the same fixed-point length. The weights are quantized separately from the inputs and neuron outputs. This is to increase the hardware design flexibility.

3.5.2 Non-Linear Activation Functions

The quantization of the non-linear activation functions is not as simple as the other variables and operations. Some of the most famous activation functions such as sigmoid and hyperbolic tangent include exponential terms, i.e., e^x. This term makes the functions harder to implement. The activation functions can still be implemented but it would be costly in terms of hardware resources and processing time [48].

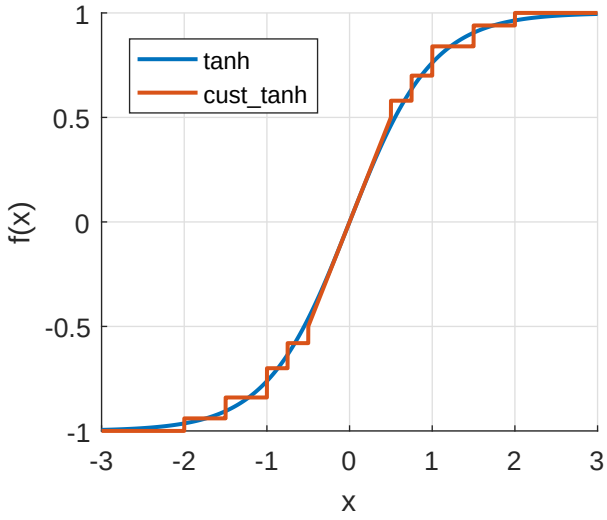


Figure 3.7: Approximation of tanh to a piecewise function

Fig. 3.7 presents and alternative solution that was used in this work. In this solution, the hyperbolic tangent is approximated to a 1^{st} order piecewise function. This approximation does not cause any degradation in the networks' recognition rates.

3.6 Simulation and Synthesis

The overall serial-neuron-based NN and MFNN are implemented in VHDL. Both networks are trained on MNIST dataset. 10 sample images (one from each class) are loaded to the FPGA ROM. The networks weights are loaded into the FPGA RAMs.

Both NN and MFNN designs were first simulated using ModelSim to verify the correctness of the hardware design. The simulation results of the values of the neurons in the output layer along with the true and classified classes are shown in Fig. 3.8 (NN) and Fig. 3.9 (MFNN).

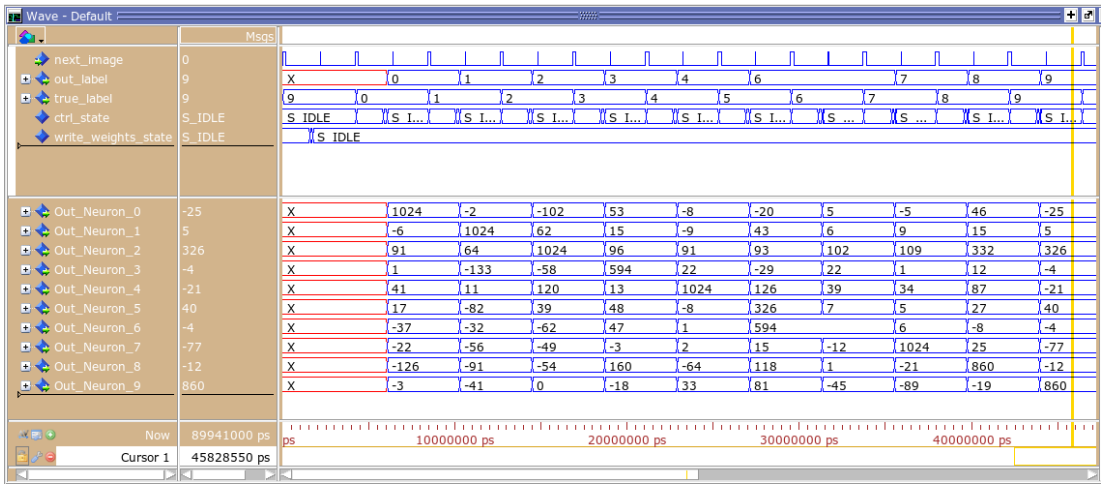


Figure 3.8: Wave from simulation for one-hidden-layer NN

The above mentioned figures show the simulation result of one-hidden-layer NN and MFNN, respectively. The networks parameters of both networks are listed in details in Table 3.2. Fixed-point registers and arithmetic are used with word length of 16 (5.10 with an extra sign bit). In VHDL the fixed-point variable is stored as STD_LOGIC_VECTOR of length WL (WL = IL + FL + 1) bits, where the location of the point (the split between the fractional and integer parts) is to be tracked manually. ModelSim is not aware of the location of the fixed-point. Therefore, the output values are scaled accordingly, i.e., $1 = 2^{10} = 1024$. Moreover, we also observe the model latency due to the long processing time of the serial neuron ($785(l = 1) + 101(l = 2) + 10$(MAX calculation) $= 896$ cycles).

	NN	MFNN
No. of neurons	100	100
α	1	4
β	1	16
Activation function	tanh	LeakyReLU scale = 1/16

Table 3.2: NN and MFNN model parameters

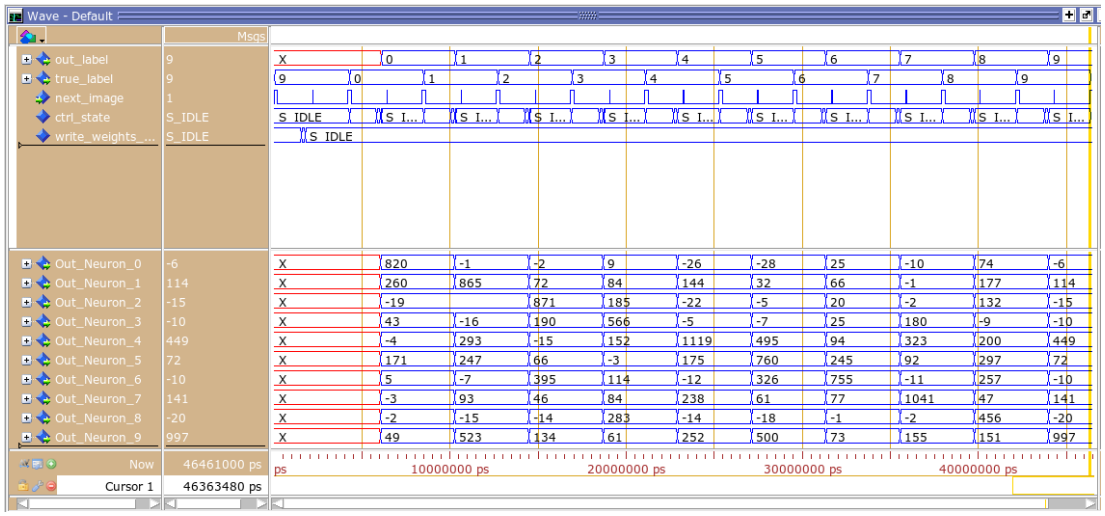


Figure 3.9: Wave from simulation for one-hidden-layer MFNN

The same NN and MFNN were tested in MATLAB (fixed-point inference) to verify the correctness of the model and that all the computations were carried out as expected. The results of the inference along with the true and classified labels are shown in Table. 3.4 (NN) and Table. 3.5 (MFNN). Both ModelSim simulation and MATLAB fixed-point inference produce the same results.

		NN	MFNN
LUTs	slices	15523	10808
	util	20.4%	14.2%
DSP units		110	0
Memory	Blocks	59	59
	util	5.73%	5.73%

Table 3.3: Hardware utilization of one-hidden-layer NN and MFNN

After verifying the hardware networks in ModelSim with MATLAB results,

they were synthesized to the FPGA using Vivado 2017.1. The hardware resources occupied on the FPGA for both NN and MFNN are listed in Table 3.3. It is worth noting that, in the MFNN synthesis, there are no DSP units used. This is because DSP units are used to preform the multiplications in NN. One DSP is used per neuron (100 hidden neurons + 10 output neurons). This is one reason why we used ASIC to measure power (avoid mapping to DSP units) [49].

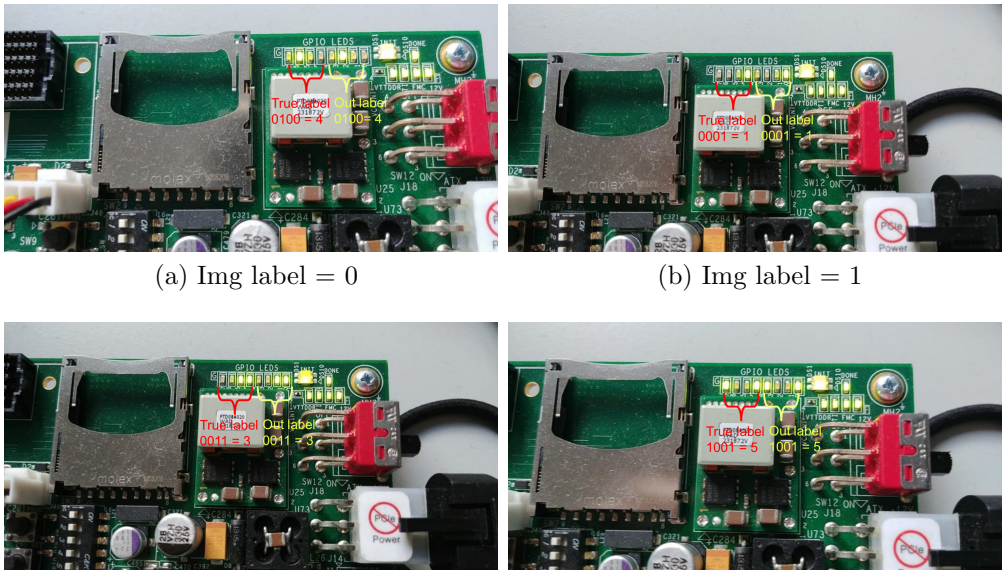
	0	1	2	3	4	5	6	7	8	9
o_0	1024	-2	-102	53	-8	-20	5	-5	46	-25
o_1	-6	1024	62	15	-9	43	6	9	15	5
o_2	91	64	1024	96	91	93	102	109	332	326
o_3	1	-133	-58	594	22	-29	22	1	12	-4
o_4	41	11	120	13	1024	126	39	34	87	-21
o_5	17	-82	39	48	-8	326	7	5	27	40
o_6	-37	-32	-62	47	1	594	594	6	-8	-4
o_7	-22	-56	-49	-3	2	15	-12	1024	25	-77
o_8	-126	-91	-54	160	-64	118	1	-21	860	-12
o_9	-3	-41	0	-18	33	81	-45	-89	-19	860
out_label	0	1	2	3	4	6	6	7	8	9
true_label	0	1	2	3	4	5	6	7	8	9

Table 3.4: MATLAB results for for one-hidden-layer NN

	0	1	2	3	4	5	6	7	8	9
o_0	820	-1	-2	9	-26	-28	25	-10	74	-6
o_1	260	865	72	84	144	32	66	-1	177	114
o_2	-19	-19	871	185	-22	-5	20	-2	132	-15
o_3	43	-16	190	566	-5	-7	25	180	-9	-10
o_4	-4	293	-15	152	1119	495	94	323	200	449
o_5	171	247	66	-3	175	760	245	92	297	72
o_6	5	-7	395	114	-12	326	755	-11	257	-10
o_7	-3	93	46	84	238	61	77	1041	47	141
o_8	-2	-15	-14	283	-14	-18	-1	-2	456	-20
o_9	49	523	134	61	252	500	73	155	151	997
out_label	0	1	2	3	4	5	6	7	8	9
true_label	0	1	2	3	4	5	6	7	8	9

Table 3.5: MATLAB results for for one-hidden-layer MFNN

Fig. 3.10 shows the actual working FPGA. The true and output labels are displayed on the LEDs to verify that the hardware network functions as expected when deployed on the FPGA.



(a) Img label = 0 (b) Img label = 1

(c) Img label = 3 (d) Img label = 5

Figure 3.10: FPGA board operational with output and true labels

3.7 Power and Area Measurements

Section 3.6 listed the FPGA utilization of both one-hidden-layer NN and MFNN with 100 neurons. MFNN does not use any DSP units, and instead, replaces them with logic as LUTs. Moreover, measuring the power on the synthesized design on the FPGA includes all rails power consumption and all the leakage power. Thus, the power and area measurement on the FPGA is not very accurate and also not detailed since no specific info can be extracted.

For these reasons, the power and area measurements were carried out using Synopsys tool. Both NNs and MFNNs with different parameters were synthesized to ASIC (Application-specific integrated circuit) 55 nm technology. The synthesized designs were later used to produce the detailed area and power reports.

Chapter 4

Results and Discussion

The main objective of this work is to provide a new more energy efficient NN architecture. Therefore, we first have to show that the new proposed MFNN is able to classify the data correctly. Then, we demonstrate the power and area comparisons of both networks to show the achievable potential savings when using MFNNs instead of NNs.

Finally we present some results about the weights' distribution, sparsity and comparison between floating-point and fixed-point arithmetics in both NNs and MFNNs.

4.1 Accuracy

First, we present and discuss the classification accuracy results on one-hidden-layer networks for various combinations of parameters. Then, we expand the discussion to two-hidden-layer networks.

4.1.1 One-Hidden-Layer Networks

4.1.1.1 Conventional NN

One-hidden-layer NNs with different No. of neurons were trained on the MNIST dataset. Fig. 4.1 presents the classification error as measured on the test dataset. The models were trained 10 times for 300 epochs using μ(Momentum factor) = 0.1, η(update rate) = 0.1 and batch size of 200 instances, where one epoch comprises 60.000 instances, i.e., the entire training dataset. Min (best case), mean, standard deviation (vertical bars on the mean curve) and max (worst case) statistics are presented.

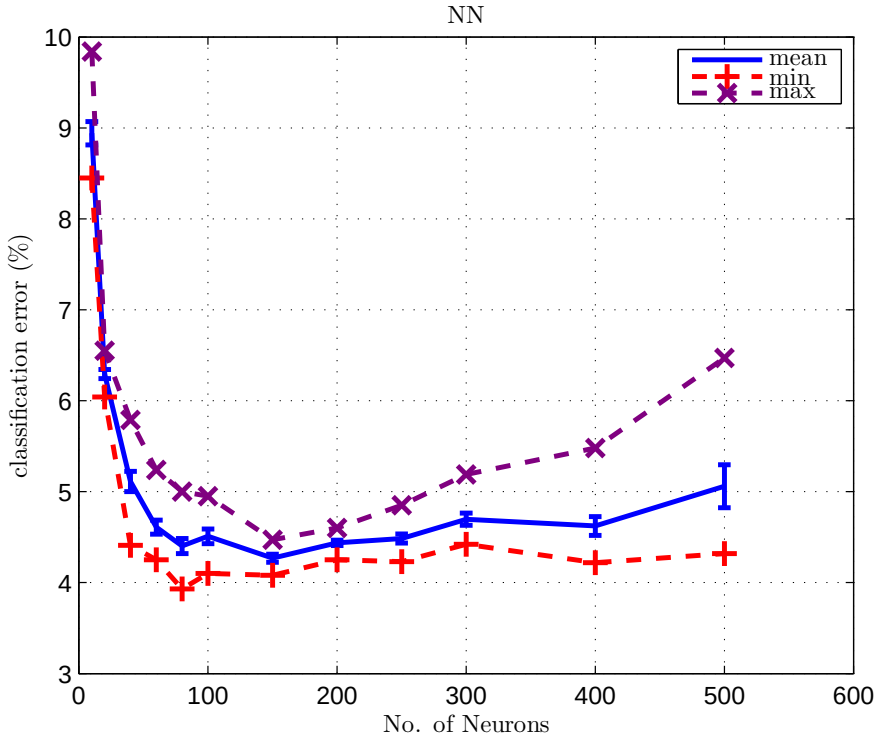


Figure 4.1: Classification error (%) in one-hidden-layer NN

Fig. 4.1 shows the effect of increasing the number of neurons in NNs. The optimal case is around 100-150 neurons. Beyond that, the NN achieves less accuracy which is probably due to the bigger model and higher degree of freedom. These larger models require more training time or/and more input training instances.

4.1.1.2 MFNN without Normalization

One-hidden-layer MFNNs with different No. of neurons were trained on the MNIST dataset. Fig. 4.2 presents the classification error as measured on the test dataset. The models were trained for 300 epochs using $\mu = 0.1$, $\eta = 0.1$, the layer normalization $\alpha = 1$, the input normalization $\beta = 1$ and batch size of 200 instances. The networks were trained 10 times each. Min (best case), mean, standard deviation (vertical bars on the mean curve) and max (worst case) statistics are presented.

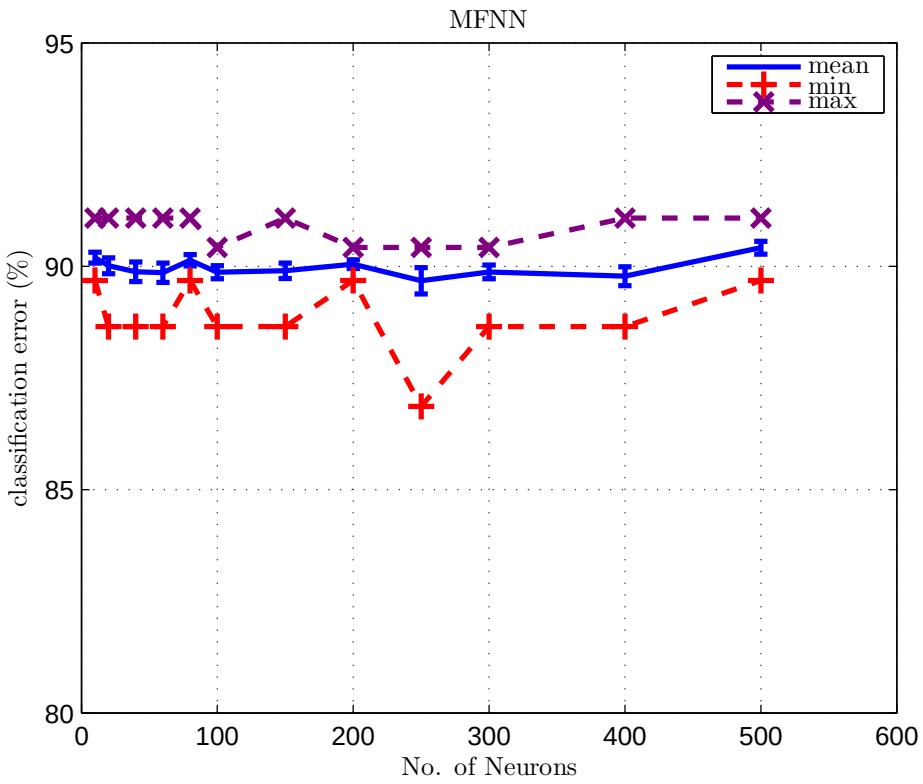


Figure 4.2: Classification error (%) in one-hidden-layer MFNN without normalization

Fig. 4.2 shows that a MFNN without normalization does not converge. The classification error is about **90**% that is the classification accuracy is **10**%. Since we have 10 equal probable classes, the accuracy of random guess is $1/10 = \mathbf{10}\%\%$. Therefore, we can say that the MFNN without normalization was not trained.

4.1.1.3 MFNN with Normalization

One-hidden-layer MFNNs with different No. of neurons were trained on the MNIST dataset. Fig. 4.2 presents the classification error as measured on the test dataset. The models were trained for 300 epochs using $\mu = 0.1$, $\eta = 0.1$, the layer normalization $\alpha = 4$, the input normalization $\beta = 16$ and batch size of 200 instances. The networks were trained 10 times each. Min (best case), mean, standard deviation (vertical bars on the mean curve) and max (worst case) statistics are presented.

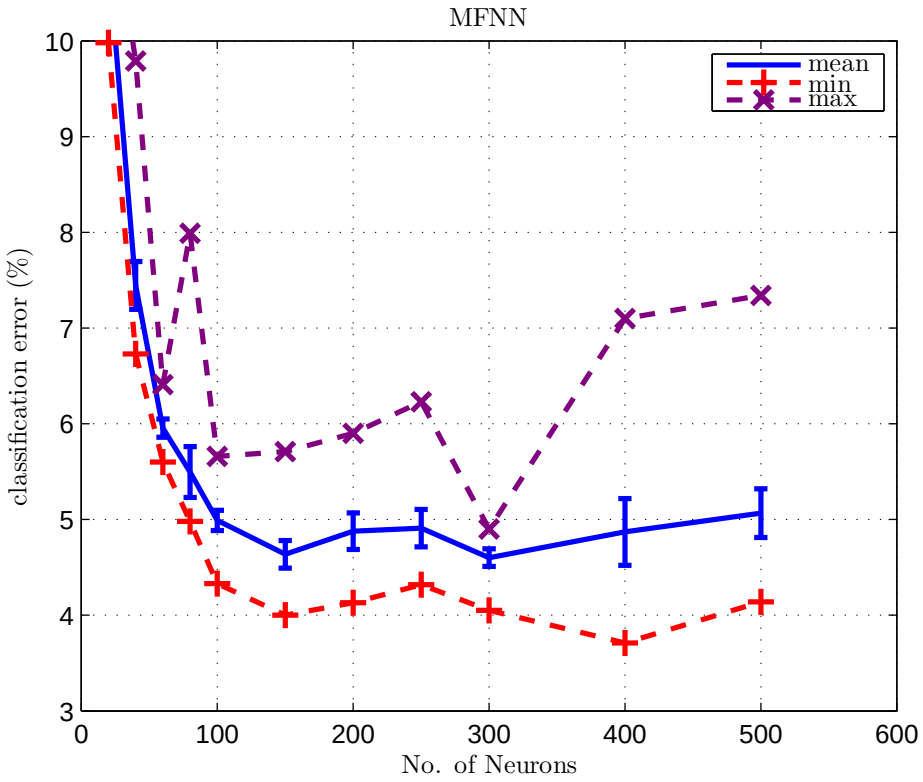


Figure 4.3: Classification error (%) in one-hidden-layer MFNN with normalization

Fig. 4.3 shows the effect of increasing the number of neurons in MFNNs. Similar to NN, the MFNN achieves optimal results for 100-150 neurons. However, contrary to NN, the accuracy of MFNN does not decrease with increasing the models' size, which stabilizes around the optimal error.

From the previous figures, we see that one-hidden-layer MFNN with normalization achieves the same accuracies as its counterpart NN using the same number of neurons. The mean accuracy achieved by MFNN is slightly less than the NN one. However, comparing the best case, both achieve the same accuracies using the same number of neurons. The convergence of the SGD optimization depends on the initial starting point and shuffling of the training dataset. It is a common practice in such optimization problems to train the model several times and then using and reporting the best model with regard to classification accuracy. With this argument, we can compare the best case results between NNs and MFNNs. The best case is the minimum error achieved by the different runs of the networks.

From Fig. 4.1 and Fig. 4.3 we see that MFNN is slightly less stable than NN in terms of convergence. In other words, training MFNN for several times yields varying test accuracies that have higher standard deviation compared to NN. The values of this standard deviation are shown in the graphs as the vertical bars over the mean curves. The length of the bars is the standard deviation value.

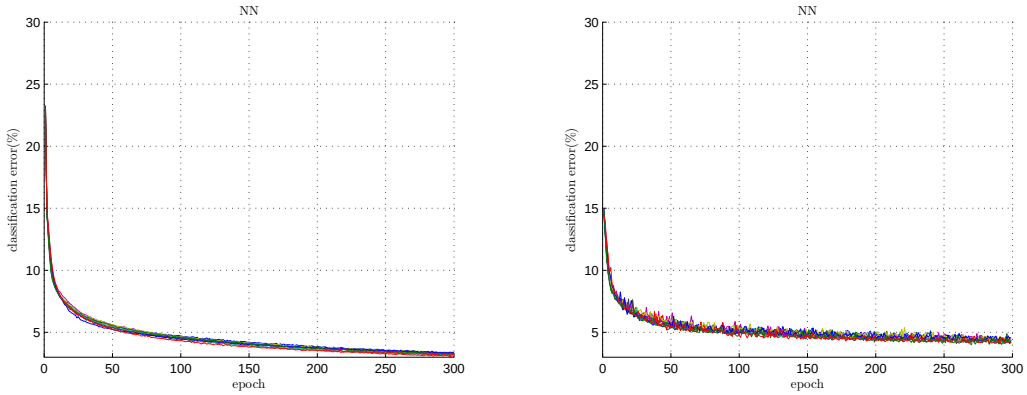
Table 4.1 summarizes the classification errors achieved on the test datasets for one-hidden-layer NNs and MFNNs for different No. of neurons using different values for α and β.

No. of neurons			10	100	150	200	300	400	500
NN			8.45	4.1	4.08	4.25	4.42	4.22	4.32
MFNN	α	β							
	1	1	90	89	89	87	89	89	90
	4	16	12.2	4.13	4.0	4.13	4.05	3.7	4.14
	8	8	18.5	5.4	4.3	4.0	3.95	3.8	4.0

Table 4.1: Classification error (%) in one-hidden-layer NN and MFNN achieved on MNIST dataset

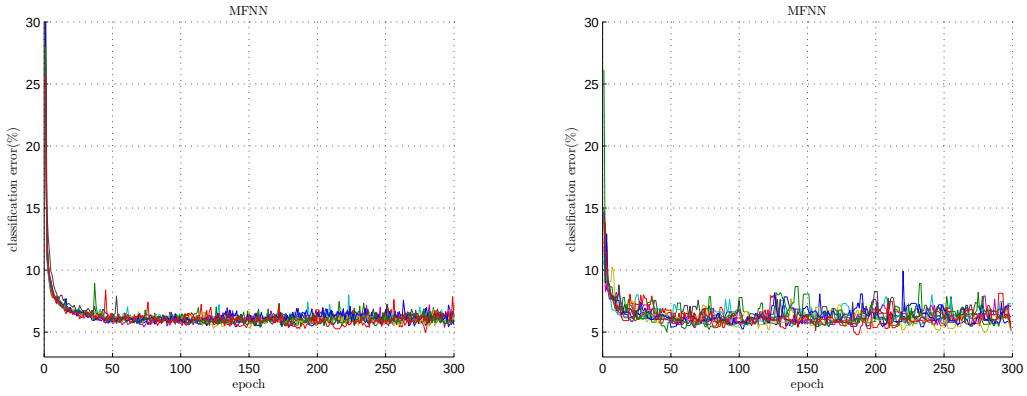
MFNNs achieve classification error rates same as NNs for the same number of neurons. For some cases, e.g., 400 neurons, $\alpha = 4$ and $\beta = 16$, the MFNN even got significantly less error rate (**3.7**% vs. **4.22**%). However, these are special cases, where most of the cases have almost the same error rates. Therefore, we can say that the MFNN and NN have the same performance for one-hidden-layer networks.

For better understanding of the convergence of test and training classification errors, we plotted them for 10 runs of one-hidden-layer NN and MFNN using 100 neurons. The models were trained for 300 epochs using $\mu = 0.1$, $\eta = 0.1$, the layer normalization $\alpha = 4$, the input normalization $\beta = 16$ and batch size of 200 instances. The networks were trained 10 times each.



(a) Training error (b) Test error

Figure 4.4: Classification error prorogation during training of NN



(a) Training error (b) Test error

Figure 4.5: Classification error prorogation during training of MFNN

We see how that MFNN is less stable during the training process. Although the error is decreasing, there are still small sudden jumps. This is probably due to the discontinuity problem addressed before in Chapter 2.

4.2 Area and Power

The hardware design was synthesized into ASIC 55 nm technology to obtain the power and area results. This is to avoid mapping hardware logic to DSPs and to have a better understanding of the hardware components areas and power measures [49].

4.2.1 Area

To better understand the power analysis and how the energy is consumed by the network, we did a detailed analysis of the synthesized design area. The area and power are linearly proportional [50]. The total area (A_T) of the MFNN and NN was measured. Moreover, we measured the neurons' area (A_N), i.e., the area only occupied by the neurons computations. The total and neurons' areas for MFNN and NN for different world lengths are presented in Fig. 4.6. $A_T(NN)$ and $A_T(MFNN)$ refer to the total area of NN and MFNN, respectively. Similarly, $A_N(NN)$ and $A_N(MFNN)$ refer to the area occupied by the neurons in NN and MFNN, respectively.

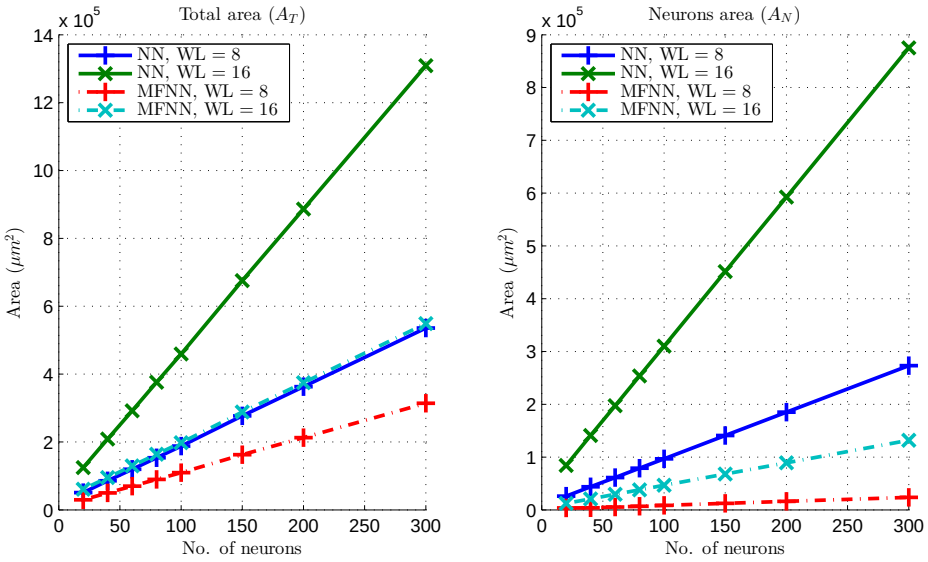


Figure 4.6: Area measurements of NN and MFNN for different word lengths

Fig. 4.6 shows significant amount of savings in the area used by the neurons in MFNN over NN, when compared for the same word lengths. Moreover, the whole MFNN occupies around **58**% and **40**% of the area occupied by NN for WL=8 and 16, respectively. The A_T includes the areas of neurons, and some other components such as the controlling logic, registers... etc. These other components have the same area in NN and MFNN since we only change the computation operation inside the neuron. Therefore, the savings in the total area are not as much as they are in the neurons.

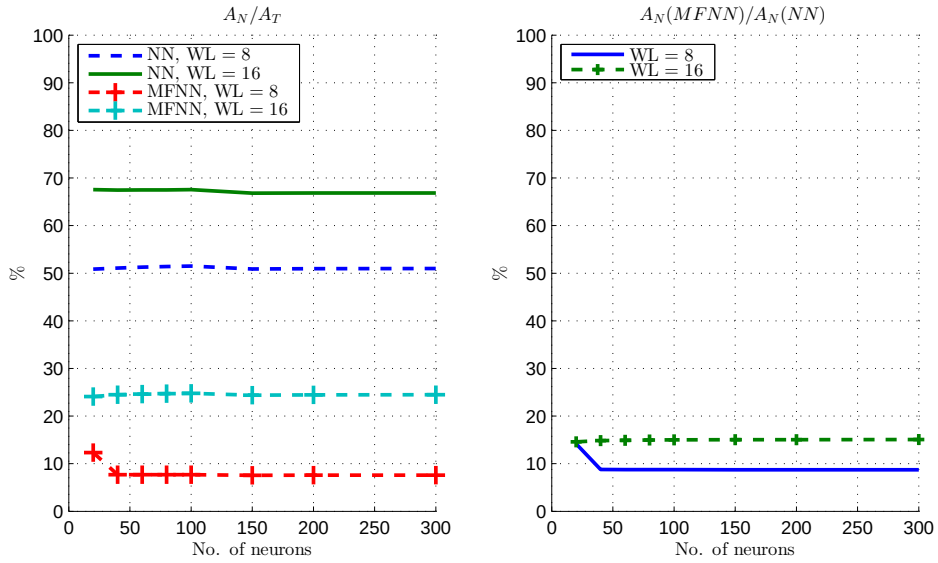


Figure 4.7: Relative area of MFNN and NN

Fig. 4.7 presents the relative area of the neurons w.r.t the total area ($A_N(NN)/A_T(NN)$, $A_N(MFNN)/A_T(MFNN)$) and the MFNN neurons' area w.r.t the NN neurons' area ($A_N(MFNN)/A_N(NN)$). It is obvious that MFNN neurons occupy much less area than NN ones. $A_N(MFNN)/A_N(NN) \approx$ **8**% and **15**% for WL=8 and 16, respectively.

We also see that, in NNs, the neurons occupy **70**% and **60**% for WL=8 and 16, respectively. This is a significant amount, since the neurons, more specifically multiplications, are responsible for most of the area and equivalently the power consumption. . These number are reduced to **8**% and **24**% when using MFNNs.

4.2.2 Power

In the previous section we have shown that the multiplication operations are occupying most of the area needed for the network. More area means more consumed energy by that component. Thus, multiplication operations are shown to consume significant amounts of power in NNs. This area is saved in MFNNs by replacing the multiplication with the mf operator. The saving in the area is to be translated in savings in the consumed power (Fig. 4.8).

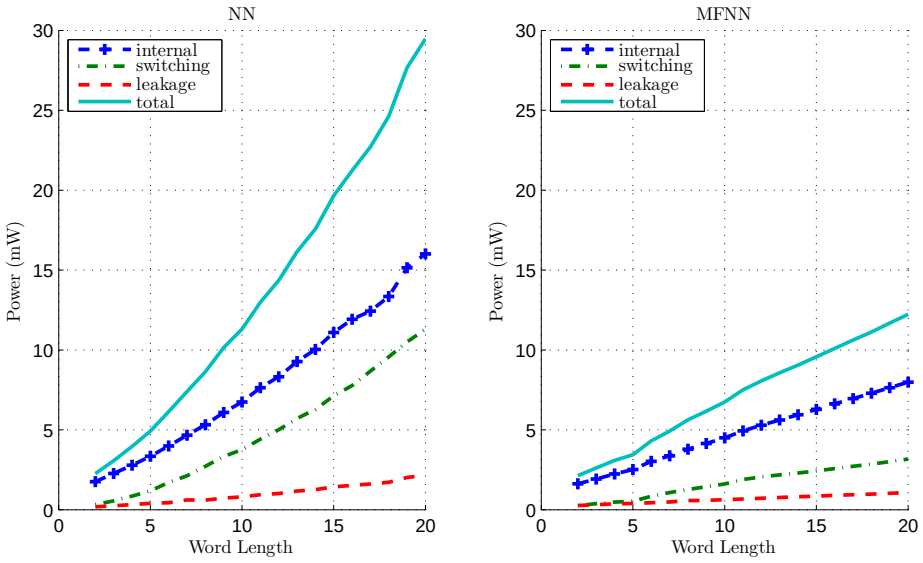


Figure 4.8: Power measurements of NN and MFNN for different word lengths

Fig. 4.8 shows the internal, leakage, switching and total powers consumed by one-hidden-layer NN and MFNN with 100 hidden neurons. All the powers are significantly improved using MFNNs. The improvement is better for longer word lengths. For examples, MFNN saves about **40**% of the total power consumed in NN for a word length of 10 bits, while it saves about **60**% for a word length of 20 bits. This is because for larger word length the neurons occupy more area in the network design. This finding is consistent with the area results. Thus, the power only consumed by the neurons can be inferred from the total power. The MFNN neurons save about **75**% and **80**% of the power consumed by the NN neurons for world lengths of 10 and 20 bits, respectively.

4.3 Other Results

4.3.1 Fixed-Point vs. Floating-Point Accuracy

In order to implement the NN and MFNN into hardware, the floating-point operations are replaced with fixed-point ones so all variables are quantized accordingly. Moreover, the power results in Section 4.2 show that the word length is a determining factor of the power consumption of the network. Thus, a detailed comparison is performed to show the accuracy achieved on both NN and MFNN using different word lengths.

The fixed-point word is illustrated as IL.FL, where IL (integer length) and FL (fractional length) are the number of bits used for the integer part and the fractional part of the word, respectively. The total word length (WL) is then calculated as follows: WL = IL + FL + 1 (the additional one is the sign bit). Moreover, the IL is allowed to have negative values. In such a case, the IL most significant bits of the fractional part are not used.

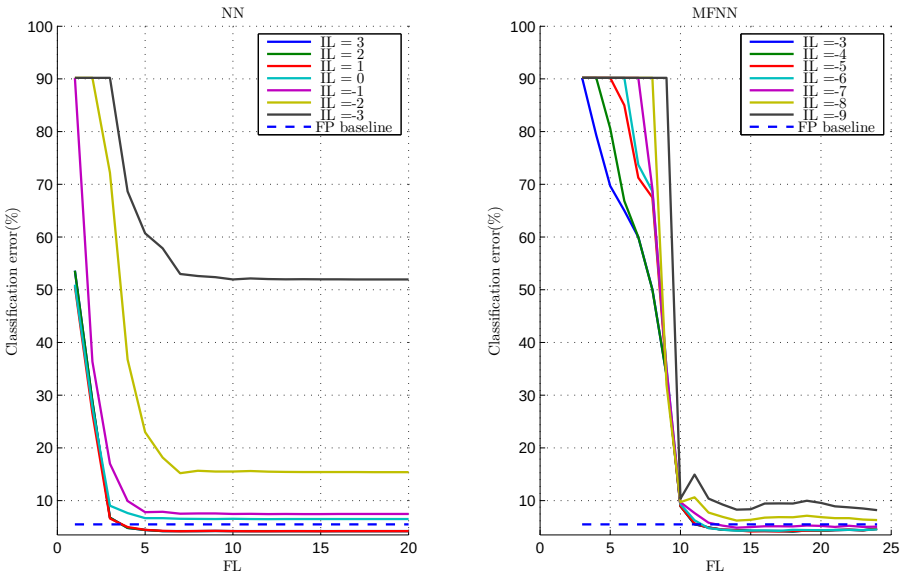


Figure 4.9: Classification error (%) for fixed-point one-hidden-layer NN and MFNN

One-hidden-layer NN and MFNN with 100 neurons are analyzed on the MNIST dataset for fixed-point operations (Fig. 4.9). The results show that the fixed-point-based NN requires a min WL of 8 bits (FL=6, IL=1) to reach the accuracies achieved by the floating-point-based NN. Similarly, the fixed-point-based MFNN requires also a min WL of 8 bits (FL=13, IL=-6). Therefore, both architectures require the same word length for this particular application. MFNN operates on lower significant bits than NN, which is expected due to the α and β normalization terms.

4.3.2 Weight Distribution

For a better understanding of MFNNs and NNs behavior, the weights of both networks are presented. The histograms of weights are computed to show the distribution of the weights. The histograms of the NN and the MFNN are presented in Fig. 4.10 and 4.11, respectively.

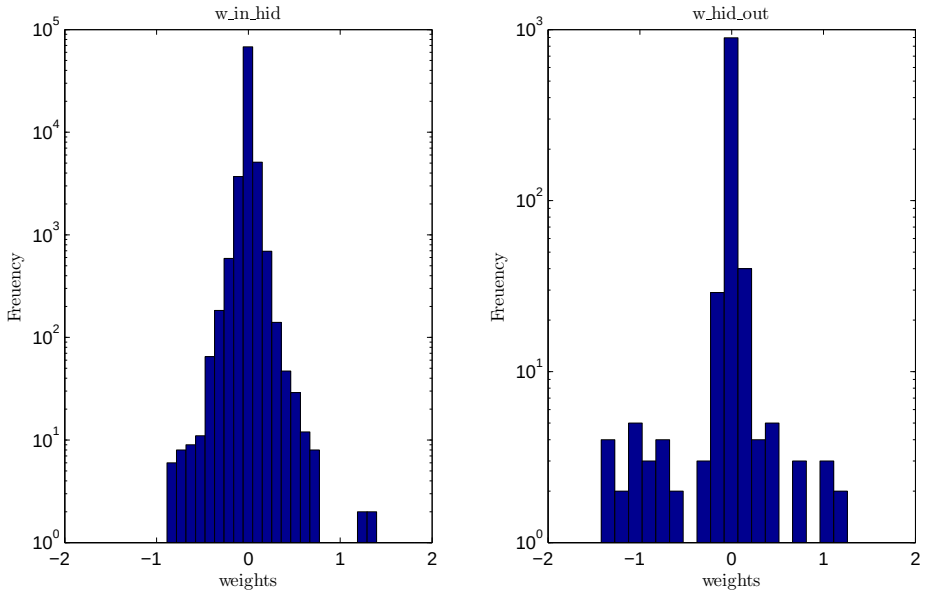


Figure 4.10: Weight distribution in one-hidden-layer NN

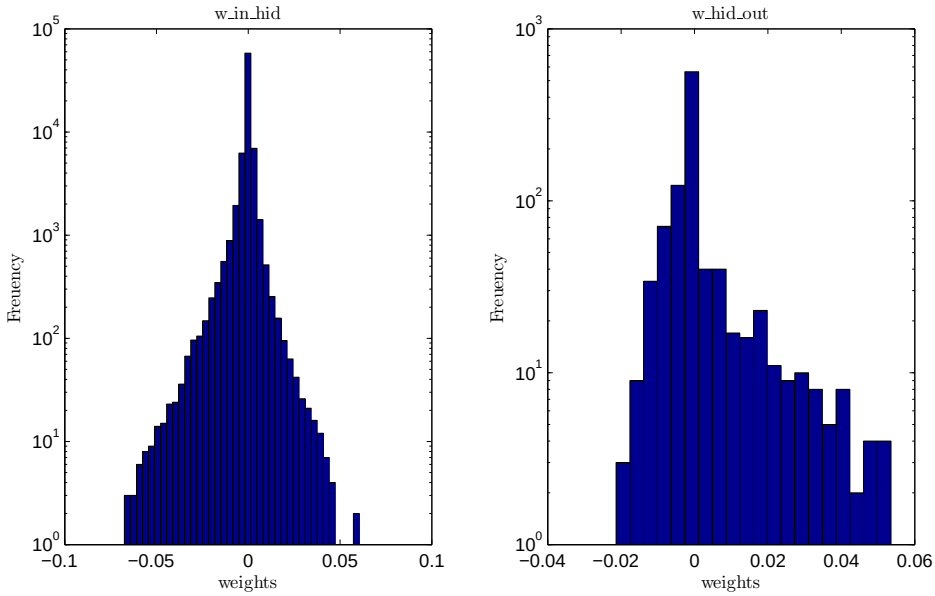


Figure 4.11: Weight distribution in one-hidden-layer MFNN

From the above mentioned figures, we see that the distribution of weights in both networks takes a natural bell-shaped distribution with a $mean \approx 0$. The weights in MFNN have much smaller values than in NN.

A very important measure is a sparsity one, which plays a key factor in determining how to store the weights and how many connections could be pruned without affecting the performance. For a vector $\boldsymbol{x}$ with a length of k, the sparsity measure used is defined as:

$$S(\boldsymbol{x}) = \frac{k - ||\boldsymbol{x}||_1}{k - ||\boldsymbol{x}||_\infty} \tag{4.1}$$

where $||\boldsymbol{x}||_1$ and $||\boldsymbol{x}||_\infty$ are the ℓ_1 and ℓ_∞ norms of $\boldsymbol{x}$. Here, $S(x) = 0$, whenever the vector is dense and $S(x) = 1$, whenever the vector is sparse.

Since the sparsity is defined for vectors, it is computed over the weight vectors $\boldsymbol{w}_j^l = [w_{1j}^l, w_{2j}^l ... w_{N_l j}^l]$, which comprises all the weights connected to neuron j. This measure shows that the weights in both MFNNs and NNs have high sparsity. However, MFNNs are slightly more sparse with, $mean \approx \mathbf{0.995}$, than NNs, with $mean \approx \mathbf{0.96}$. Fig. 4.12 and Fig. 4.13 show the sparsity of weight vectors in one-hidden-layer NN and MFNN, respectively.

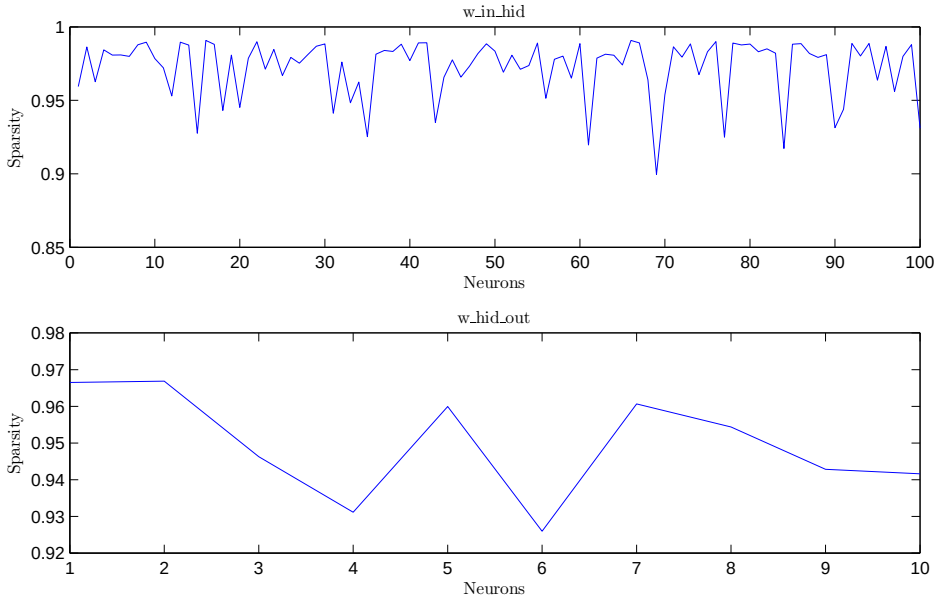


Figure 4.12: Weight sparsity in one-hidden-layer NN

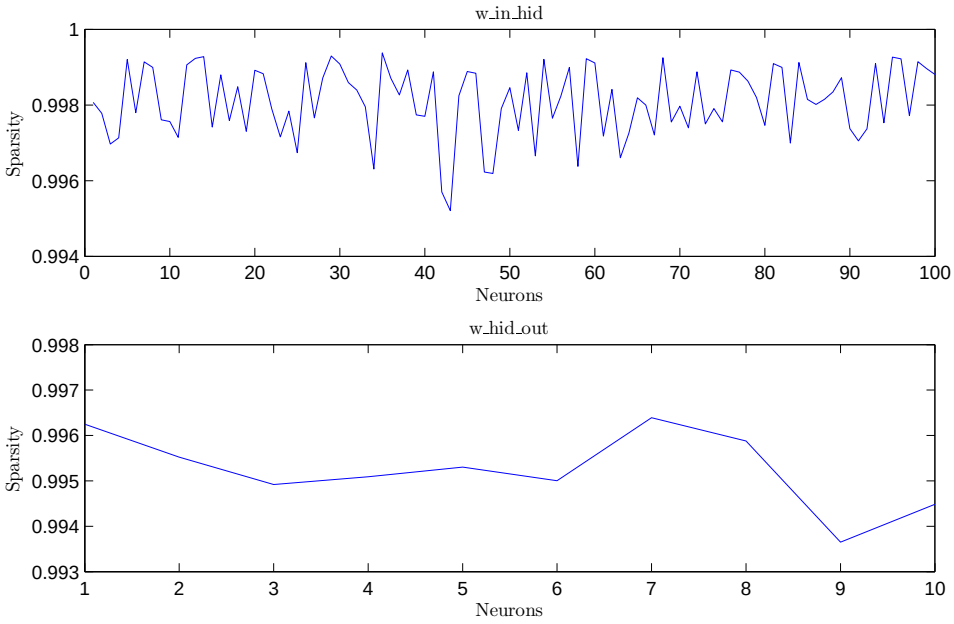


Figure 4.13: Weight sparsity in one-hidden-layer MFNN

4.3.3 Pruning

A very important method of saving energy is pruning the weights of the network [51, 52]. Pruning in NNs is the method of getting rid of the connections with insignificant weights. We observed in the previous section that the NNs and MFNNs weights are sparse. This means that small percentage of the weights store the important information about the network. We investigate the effect of pruning NNs and MFNNs on the classification error and the amount of saved storage. We implement a simple pruning technique:

$$w_{ij}^l = \begin{cases} 0 & \text{if } \left|w_{ij}^l\right| \leq T||\boldsymbol{w}_j^l||_p \\ w_{ij}^l & \text{otherwise} \end{cases} \tag{4.2}$$

where $T \leq 1$ is the pruning threshold and $||\boldsymbol{w}_j^l||_p$ is the p-norm of $\boldsymbol{w}_j^l$ [53]. For our experiments, $p = \infty$ is selected. In this case, $||\boldsymbol{w}_j^l||_\infty = \max_i \left|w_{ij}^l\right|$.

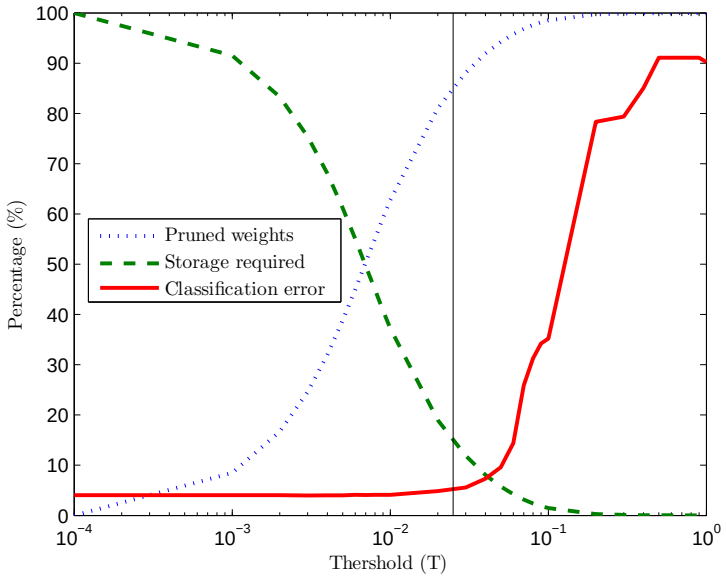


Figure 4.14: Pruning results for one-hidden-layer NN

Fig. 4.14 and Fig. 4.15 show the pruning results for one-hidden-layer NN and MFNN using different pruning thresholds T. We display the percentage of weights pruned, the percentage of the required storage for the remaining weights versus the classification error after pruning the network. The required storage is

calculated as follows:

$$\text{storage} = \frac{1 - N_p}{N} \quad 100\% \tag{4.3}$$

where N_p and N are the amount of pruned and total weights, respectively.

We see that, in NNs, we can prune approximately **85**% of the weights without losing any accuracy (black reference line). This leads to approximately **85**% savings in the storage (**15**% required storage). These numbers are less for MFNN. In order not to degrade the accuracy rates of MFNN, we can only prune approximately **30**% of the weights (black reference line). This leads to approximately **30**% savings in the storage (**70**% required storage). This is due to the aforementioned discountability problem of MFNN. In MFNN, a lot of the information is stored in the signs of the weights rather than the values. Thus, when mapping to 0 the sign is completely lost which affects the performance severely.

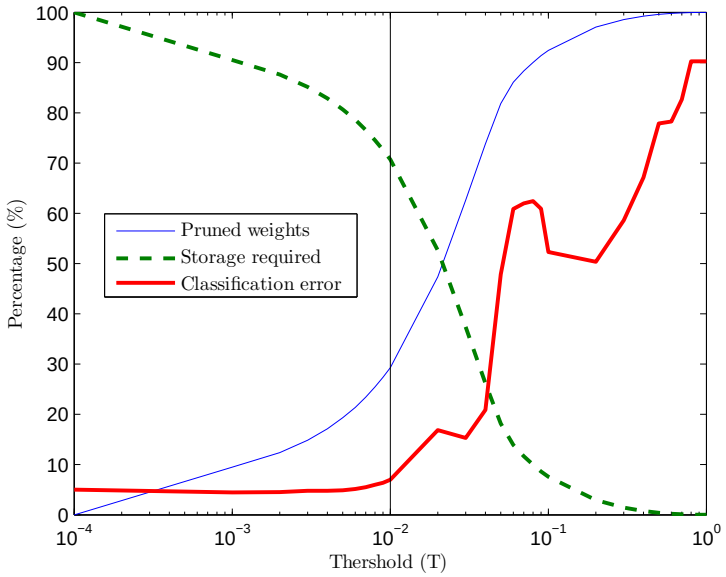


Figure 4.15: Pruning results for one-hidden-layer MFNN

To solve the issue of pruning MFNN weights discussed above. We changed the pruning rule to:

$$w_{ij}^l = \begin{cases} 0^+ & \text{if } \left|w_{ij}^l\right| \leq T||\boldsymbol{w}_j^l||_p \text{ and } w_{ij}^l > 0 \\ 0^- & \text{if } \left|w_{ij}^l\right| \leq T||\boldsymbol{w}_j^l||_p \text{ and } w_{ij}^l < 0 \\ w_{ij}^l & \text{otherwise} \end{cases} \tag{4.4}$$

where $sgn(0^+) = 1$, $|0^+| = 0$, $sgn(0^-) = -1$ and $|0^-| = 0$. The mf operator reduces to $a \oplus b = sgn(a)b$, when $a \in 0^+, 0^-$. In this case, one bit is needed to store the sign of the pruned weight, thus, the required storage becomes:

$$\text{storage} = \frac{W_L(N - N_p) + N_p}{W_L\ N} \quad 100\% \tag{4.5}$$

where W_L is the word length, N_p and N are the amount of pruned and total weights, respectively. We proved earlier that, in fixed-point-based networks, $W_L = 8$ is sufficient to achieve the floating-point-based network accuracies. Therefore, we used this value to calculate the required storage.

The results of performing this pruning on the same one-hidden-layer MFNN as before are presented in Fig. 4.16. We can prune approximately **95**% of the weights without losing any accuracy (black reference line). This pruning leads to approximately **83**% savings in the storage. When $T = 1$, the MFNN reduces to the Binary-weight Network [36]. We lose a total of **2**% for dropping all the weight values and keeping the signs. Hence, most of the information is stored in the sign term of the mf-operator, however, there is small information stored in the value term that can be significant to very sensitive applications.

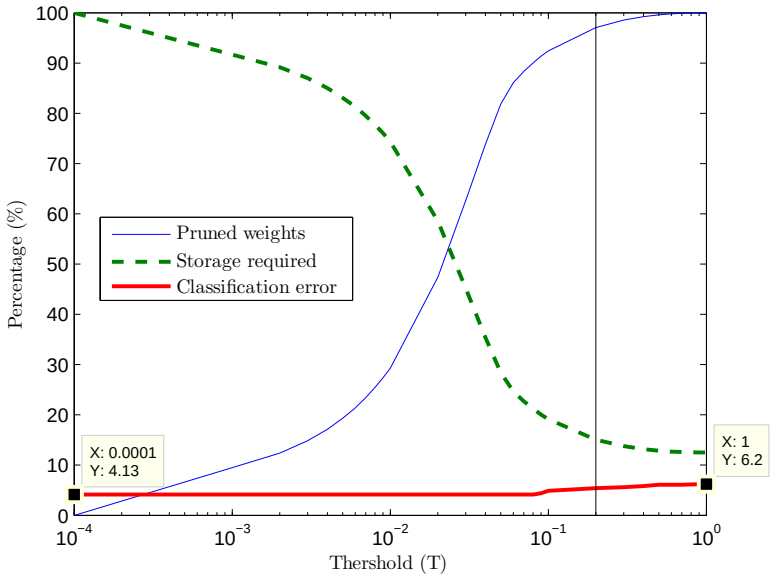


Figure 4.16: Enhanced pruning results for one-hidden-layer MFNN

Chapter 5

Conclusion

This thesis propose a new multiplication free neural network (MFNN), in which, the multiplication operations of the conventional NN are replaced with modified additions (multiplication free operator). We prove that most of the power is consumed by the multiplication operations carried out by the neurons. The consumption varies depending on the word length used. On average, multiplication operations account for roughly **60**% of the power consumption of the entire NN.

We also show that MFNNs achieve the same recognition rates as NNs, in one-hidden-layer networks, for the same number of neurons. Moreover, we show that using fixed-point over floating-point arithmetics does not yield any loss in the accuracy for both networks. Both architectures require the same fixed-point world length of **8** bits to achieve the floating-point-based architectures performance.

Furthermore, we show that, using pruning, both NNs and MFNNs can save up to **85**% of the required storage in one-hidden-layer networks. Hence, saving the energy required to fetch and process those pruned weights. However, this requires a different data structure to carry out the vector operations efficiently with sparse representation and storage. Furthermore, we show that in one-hidden-layer MFNNs, we are capable of saving up to **87.5**% of the required storage by saving only the signs of the weights while getting rid of all the values.

Finally, we demonstrate that MFNNs save a significant amount of energy compared to the conventional NNs without any loss in accuracy. The savings range from **40**% to **60**% of the total power depending on the word length of the fixed-point number format.

Deep and "convolutional" versions of MFNNs can be also realized in FPGA. In this class of NNs, the convolution operation is replaced by a sliding vector product based on the new operator defined in Chapter 2. In general, a significant amount of energy savings can be achieved by the multiplication free convolutional neural networks because of the reduced number of multiplication operations per layer.

Bibliography

[1] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of research and development*, vol. 44, no. 1.2, pp. 206–226, 2000.

[2] T. M. Mitchell, "Machine learning. 1997," *Burr Ridge, IL: McGraw Hill*, vol. 45, no. 37, pp. 870–877, 1997.

[3] T. Hastie, R. Tibshirani, and J. Friedman, "Overview of supervised learning," in *The elements of statistical learning*, pp. 9–41, Springer, 2009.

[4] F. Sebastiani, "Machine learning in automated text categorization," *ACM computing surveys (CSUR)*, vol. 34, no. 1, pp. 1–47, 2002.

[5] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, "Handwritten digit recognition with a back-propagation network," in *Advances in neural information processing systems*, pp. 396–404, 1990.

[6] S. B. Kotsiantis, I. Zaharakis, and P. Pintelas, "Supervised machine learning: A review of classification techniques," 2007.

[7] K. Polat and S. Güneş, "Breast cancer diagnosis using least square support vector machine," *Digital Signal Processing*, vol. 17, no. 4, pp. 694–701, 2007.

[8] L. Jack and A. Nandi, "Fault detection using support vector machines and artificial neural networks, augmented by genetic algorithms," *Mechanical systems and signal processing*, vol. 16, no. 2-3, pp. 373–390, 2002.

[9] O. Chapelle, P. Haffner, and V. N. Vapnik, "Support vector machines for histogram-based image classification," *IEEE transactions on Neural Networks*, vol. 10, no. 5, pp. 1055–1064, 1999.

[10] D. Ciregan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," in *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pp. 3642–3649, IEEE, 2012.

[11] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[12] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[13] M. Minsky and S. Papert, "Perceptrons.," 1969.

[14] S. Grossberg, "Contour enhancement, short term memory, and constancies in reverberating neural networks," pp. 332–378, 1982.

[15] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning representations by back-propagating errors*. na, 1986.

[16] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[17] S. Sarkar, V. M. Patel, and R. Chellappa, "Deep feature-based face detection on mobile devices," in *Identity, Security and Behavior Analysis (ISBA), 2016 IEEE International Conference on*, pp. 1–8, IEEE, 2016.

[18] M. W. Gardner and S. Dorling, "Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences," *Atmospheric environment*, vol. 32, no. 14, pp. 2627–2636, 1998.

[19] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural networks*, vol. 12, no. 1, pp. 145–151, 1999.

[20] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," tech. rep., California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[21] Y. LeCun, "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.

[22] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[23] E. Kussul and T. Baidyk, "Improved method of handwritten digit recognition tested on mnist database," *Image and Vision Computing*, vol. 22, no. 12, pp. 971–981, 2004.

[24] R. Salakhutdinov and H. Larochelle, "Efficient learning of deep boltzmann machines," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 693–700, 2010.

[25] H. Tuna, I. Onaran, and A. E. Cetin, "Image description using a multiplier-less operator," *IEEE Signal Processing Letters*, vol. 16, no. 9, pp. 751–753, 2009.

[26] A. Suhre, F. Keskin, T. Ersahin, R. Cetin-Atalay, R. Ansari, and A. E. Cetin, "A multiplication-free framework for signal processing and applications in biomedical image analysis," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 1123–1127, IEEE, 2013.

[27] C. E. Akbaş, O. Günay, K. Taşdemir, and A. E. Çetin, "Energy efficient cosine similarity measures according to a convex cost function," *Signal, Image and Video Processing*, vol. 11, no. 2, pp. 349–356, 2017.

[28] H. S. Demir and A. E. Cetin, "Co-difference based object tracking algorithm for infrared videos," in *Image Processing (ICIP), 2016 IEEE International Conference on*, pp. 434–438, IEEE, 2016.

[29] C. E. Akbaş, A. Bozkurt, A. E. Çetin, R. Çetin-Atalay, and A. Üner, "Multiplication-free neural networks," in *Signal Processing and Communications Applications Conference (SIU), 2015 23th*, pp. 2416–2418, IEEE, 2015.

[30] D. Badawi, E. Akhan, M. Mallah, A. Üner, R. Çetin-Atalay, and A. E. Çetin, "Multiplication free neural network for cancer stem cell detection in h-and-e stained liver images," in *SPIE Commercial+ Scientific Sensing and Imaging*, pp. 102110C–102110C, International Society for Optics and Photonics, 2017.

[31] A. Afrasiyabi, B. Nasir, O. Yildiz, F. T. Y. Vural, and A. E. Cetin, "An energy efficient additive neural network," pp. 1–4, 2017.

[32] I. Aleksander and H. Morton, *An introduction to neural computing*, vol. 3. Chapman & Hall London, 1990.

[33] M. Hilbert and P. López, "The world's technological capacity to store, communicate, and compute information," *science*, vol. 332, no. 6025, pp. 60–65, 2011.

[34] S. S. Sarwar, S. Venkataramani, A. Raghunathan, and K. Roy, "Multiplier-less artificial neurons exploiting error resiliency for energy-efficient neural computing," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pp. 145–150, IEEE, 2016.

[35] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in Neural Information Processing Systems*, pp. 1135–1143, 2015.

[36] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*, pp. 525–542, Springer, 2016.

[37] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar, "Reducing power by optimizing the necessary precision/range of floating-point arithmetic," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 3, pp. 273–286, 2000.

[38] A. R. Omondi and J. C. Rajapakse, *FPGA implementations of neural networks*, vol. 365. Springer, 2006.

[39] Y. Cao, Y. Chen, and D. Khosla, "Spiking deep convolutional neural networks for energy-efficient object recognition," *International Journal of Computer Vision*, vol. 113, no. 1, pp. 54–66, 2015.

[40] K. Orimo, K. Ando, K. Ueyoshi, M. Ikebe, T. Asai, and M. Motomura, "Fpga architecture for feed-forward sequential memory network targeting long-term time-series forecasting," in *ReConFigurable Computing and FPGAs (ReConFig), 2016 International Conference on*, pp. 1–6, IEEE, 2016.

[41] A. Oppenheim, A. Willsky, and S. Nawab, *Signals and Systems*. Prentice-Hall signal processing series, Prentice Hall, 1997.

[42] Xilinx, *VC707 Evaluation Board for the Virtex-7 FPGA User Guide*, August 2016. Available at `https://www.xilinx.com/support/documentation/boards_and_kits/vc707/ug885_VC707_Eval_Bd.pdf`, v1.7.1, last accessed on 05-Dec-2017.

[43] Xilinx, *7 Series FPGAs Data Sheet: Overview*, August 2017. Available at `https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf`, v2.5, last accessed on 05-Dec-2017.

[44] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pp. 1737–1746, 2015.

[45] A. Finnerty and H. Ratigner, "White paper: Reduce power and cost by converting from floating point to fixed point," Tech. Rep. WP491, Xilinx, March 2017. Available at `https://www.xilinx.com/support/documentation/white_papers/wp491-floating-to-fixed-point.pdf`, v1.0, last accessed on 11-Dec-2017.

[46] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[47] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented approximation of convolutional neural networks," *arXiv preprint arXiv:1604.03168*, 2016.

[48] A. Boudabous, F. Ghozzi, M. Kharrat, and N. Masmoudi, "Implementation of hyperbolic functions using cordic algorithm," in *Microelectronics, 2004. ICM 2004 Proceedings. The 16th International Conference on*, pp. 738–741, IEEE, 2004.

[49] I. Kuon and J. Rose, "Measuring the gap between fpgas and asics," *IEEE Transactions on computer-aided design of integrated circuits and systems*, vol. 26, no. 2, pp. 203–215, 2007.

[50] L. Deng, K. Sobti, and C. Chakrabarti, "Accurate models for estimating area and power of fpga implementations," in *Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on*, pp. 1417–1420, IEEE, 2008.

[51] A. Lazarevic and Z. Obradovic, "Effective pruning of neural network classifier ensembles," in *Neural Networks, 2001. Proceedings. IJCNN'01. International Joint Conference on*, vol. 2, pp. 796–801, IEEE, 2001.

[52] E. D. Karnin, "A simple procedure for pruning back-propagation trained neural networks," *IEEE Transactions on Neural Networks*, vol. 1, no. 2, pp. 239–242, 1990.

[53] E. Kreyszig, *Introductory functional analysis with applications*, vol. 1. wiley New York, 1989.

[54] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals, and Systems (MCSS)*, vol. 2, no. 4, pp. 303–314, 1989.

[55] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural networks*, vol. 4, no. 2, pp. 251–257, 1991.

[56] S.-C. Huang and Y.-F. Huang, "Bounds on the number of hidden neurons in multilayer perceptrons," *IEEE transactions on neural networks*, vol. 2, no. 1, pp. 47–55, 1991.

Appendix A

Comparison of Operators According to The Universal Approximation Theorem

The universal approximation theorem states that a one-hidden-layer neural network can approximate any arbitrary function defined on a unit hypercube [54,55].

Theorem A.0.1 (Universal Approximation Theorem)**.** *Let $\varphi(.)$ be a nonlinear, bounded, sigmoidal function. Let I_m donate the m-dimentional hybercube $[0,1]^m$. Then given any $\epsilon > 0$ and any continuous function, $f(\boldsymbol{x})$, defined on $\boldsymbol{x} \in I_m$, there exists an integer N, real constants $v_i, b_i \in \mathbb{R}$ and a real vectors $\boldsymbol{w}_i \in \mathbb{R}^m$ such that we may define $F(\boldsymbol{x})$ as*

$$F(\boldsymbol{x}) = \sum_{i=1}^{N} v_i \varphi(\boldsymbol{w}_i^T \boldsymbol{x} + b_i) \tag{A.1}$$

such that $F(\boldsymbol{x})$ is an approximation of $f(\boldsymbol{x})$, that is:

$$|F(\boldsymbol{x}) - f(\boldsymbol{x})| < \epsilon \tag{A.2}$$

In other words, $F(\boldsymbol{x})$ is dense on $L_1(I_m)$. The theorem still holds for any compact subset of $\mathbb{R}^m$ replacing I_m.

A.1 The Universal Approximation Theorem for Multiplication Free Neural Networks

Let the multiplication free neurons be defined as:

$$\boldsymbol{o}^l = \boldsymbol{a}^l \circ \mathbf{W}^l \oplus \boldsymbol{o}^{l-1} + \boldsymbol{b}^l \tag{A.3}$$

where $\boldsymbol{a}^l$ is a normalization vector and $\circ$ is the element-wise product.

Proposition 1. *The Multiplication Free Neural Network (MFNN) with identity activation functions, $f(\boldsymbol{x}) = \boldsymbol{x}$, is capable of realizing functions that are dense in $L_1(I_m)$.*

To prove the previous proposition, the following two lemmas are proved first:

Lemma A.1.1. *There exist a MFNN with identity activation functions that can compute $g_1(\boldsymbol{x}; \boldsymbol{w}', b') = \boldsymbol{w}'^T \boldsymbol{x} + b'$.*

Proof. Composing a MFNN that computes $g_1(\boldsymbol{x}; \boldsymbol{w}', b') = \boldsymbol{w}'^T \boldsymbol{x} + b'$ is enough to prove this lemma. First we define 0^+ term, such that $sgn(0^+) \triangleq 1$ and $|0^+| \triangleq 0$. Remember that, $sgn(0) \triangleq 0$. Now, we can construct a two-layer MFNN that computes $g_1(\boldsymbol{x}; \boldsymbol{w}', b') = \boldsymbol{w}'^T \boldsymbol{x} + b'$ for any given $\boldsymbol{w}' \in \mathbb{R}^m$ and $b' \in \mathbb{R}$ with the following parameters:

- Hidden layer 1,

$$\mathbf{W}^1 = \begin{bmatrix} 0^+ & 0 & 0 & \dots & 0 \\ 0 & 0^+ & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0^+ \end{bmatrix} \quad \boldsymbol{b}^1 = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad \boldsymbol{a}^1 = \begin{bmatrix} w'_1 \\ w'_2 \\ \vdots \\ w'_m \end{bmatrix}$$

- Hidden layer 2,

$$\mathbf{W}^2 = \begin{bmatrix} 0^+ \\ 0^+ \\ 0^+ \\ 0^+ \end{bmatrix} \quad \boldsymbol{b}^2 = \begin{bmatrix} b' \end{bmatrix} \quad \boldsymbol{a}^2 = \begin{bmatrix} 1 \end{bmatrix}$$

Then, the hidden layers outputs $\boldsymbol{o}^1$ and $\boldsymbol{o}^2$ can be represented as follows:

$$\boldsymbol{o}^1 = \boldsymbol{a}^1 \circ \mathbf{W}^1 \oplus \boldsymbol{x} + \boldsymbol{b}^1 = \begin{bmatrix} w'_1 x_1 \\ w'_2 x_2 \\ \vdots \\ w'_m x_m \end{bmatrix}$$

$$\boldsymbol{o}^2 = \boldsymbol{a}^2 \circ \mathbf{W}^2 \oplus \boldsymbol{o}^1 + \boldsymbol{b}^2 = \boldsymbol{w'}^T \boldsymbol{x} + b' = g_1(\boldsymbol{x}; \boldsymbol{w'}, b')$$

□

Lemma A.1.2. *There exist a MFNN defined with the identity activation function that can compute $g_2(x) = sgn(x)$.*

Proof. Similar to the proof of A.1.1, composing a MFNN that computes $g_2(x) = sgn(x)'$ is enough to prove this lemma. We can construct a two-layer MFNN that computes $g_2(x) = sgn(x)$ with the following parameters:

- Hidden layer 1,

  $\mathbf{W}^1 = \begin{bmatrix} 2 & 1 \end{bmatrix}$ $\boldsymbol{b}^1 = \begin{bmatrix} 0 & 0 \end{bmatrix}^T$ $\boldsymbol{a}^1 = \begin{bmatrix} 1 & 1 \end{bmatrix}^T$

- Hidden layer 2,

  $\mathbf{W}^2 = \begin{bmatrix} 2 & 1 \end{bmatrix}^T$ $\boldsymbol{b}^2 = \begin{bmatrix} 0 \end{bmatrix}$ $\boldsymbol{a}^2 = \begin{bmatrix} 1 \end{bmatrix}$

The output of the network can be simplified using the fact that, $\forall x \in \mathbb{R}$ and $\forall b \in \mathbb{R}^+$,

$$sgn(x + bsgn(x)) = sgn(x)$$

Then, the hidden layers outputs $\boldsymbol{o}^1$ and $\boldsymbol{o}^2$ can be represented as follows:

$$\boldsymbol{o}^1 = \boldsymbol{a}^1 \circ \mathbf{W}^1 \oplus \boldsymbol{x} + \boldsymbol{b}^1 = \begin{bmatrix} x_1 + 2sgn(x_1) \\ x_2 + sgn(x_2) \end{bmatrix}$$

$$\boldsymbol{o}^2 = \boldsymbol{a}^2 \circ \mathbf{W}^2 \oplus \boldsymbol{o}^1 + \boldsymbol{b}^2 = sgn(x) = g_2(x)$$

□

Proof of Proposition 1. Using Lemma A.1.1 and Lemma A.1.2 we can construct $y_i(\boldsymbol{x}; \boldsymbol{w}'_i, b'_i) = g_2(g_1(\boldsymbol{x}; \boldsymbol{w}'_i, b'_i)) = sgn(\boldsymbol{w}_i'^T \boldsymbol{x} + b'_i)$ for $i = 1,\ 2\ ...,\ N$.
Then we use Lemma A.1.1 again to perform the weighted sum over $\boldsymbol{y} = \{y_i\}_{i=1}^N$ as follows:

$$\begin{aligned} F(\boldsymbol{x}; \{v_i\}_{i=1}^N, \{\boldsymbol{w}'_i\}_{i=1}^N, \{b_i\}_{i=1}^N) &= g_1(\boldsymbol{y}(\boldsymbol{x}; \{\boldsymbol{w}'_i\}_{i=1}^N, \{b_i\}_{i=1}^N); \boldsymbol{v'}, \boldsymbol{0}) \\ &= \boldsymbol{v}^T \boldsymbol{y} + \boldsymbol{0} \\ &= \sum_{i=1}^N v_i \boldsymbol{y}_i \\ &= \sum_{i=1}^N v_i sgn(\boldsymbol{w}_i'^T \boldsymbol{x} + b_i) \end{aligned} \tag{A.4}$$

We see that the computed functions $F(\boldsymbol{x})$ realized by MFNNs satisfy the Universal Approximation Theorem A.0.1, since sgn is a sigmoidal function. Therefore, the functions computed using MFNNs are dense in $L_1(I_m)$. Moreover, two hidden layers are needed to implement $g_2(x) = sgn(x)$, which can be avoided by using a sigmoidal activation function on the output of $g_1(\boldsymbol{x})$. □

A.2 One-Hidden-Layer Upper Bound

Theorem A.0.1 states that conventional neural network with one-hidden-layer is capable of realizing functions that are dense in $L_1(I_m)$. However, the number of hidden neurons needed N is not restricted. Huang et al. proved an upper bound for the number of hidden neurons needed for a finite input subset $S \in E^m$ [56]. The upper bound found k is the input subset size.

For this to hold, there should be at least one separable element in S. That is, there exist an $\boldsymbol{x}_1 \in S$ such that:

$$sgn(\boldsymbol{w}_1^T \boldsymbol{x} + b_1) = \begin{cases} 1 & \text{if } \boldsymbol{x} = \boldsymbol{x}_1 \\ 0 & otherwise \end{cases} \tag{A.5}$$

then, we construct $S_1 = S - \{\boldsymbol{x}_1\}$, then, we can similarly separate $\boldsymbol{x}_2$ from S_1 and so on. Huang et al. proved that there are $\boldsymbol{w}_1$ and b_1 so that A.5 holds. This analysis is based on the neural network which uses multiplication [56].

We proved earlier in Section A.1 that MFNNs with identity activation functions satisfies the Universal Approximation Theorem. However, we needed more than one-hidden-layer network. Here we discuss if the upper limit set by Huang et al. is still valid for the one-hidden-layer MFNNs or the one-hidden-layer Binary-Weight Networks (BWNs) [36].

A.2.1 Multiplication Free Neural Network

Without any loss in generality, let $x \in \mathbb{R}^2$ for simplicity,. Consider the set of points $\boldsymbol{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \boldsymbol{x}_4, \boldsymbol{x}_5, \boldsymbol{x}_6, \boldsymbol{x}_7, \boldsymbol{x}_8\}$ where the coordinates of the points are defined in Table A.1 where $e_1 > e_2 > 0$.

Point	$\boldsymbol{x}_1$	$\boldsymbol{x}_2$	$\boldsymbol{x}_3$	$\boldsymbol{x}_4$	$\boldsymbol{x}_5$	$\boldsymbol{x}_6$	$\boldsymbol{x}_7$	$\boldsymbol{x}_8$
Coordinates	e_1	e_2	$-e_1$	$-e_2$	$-e_1$	$-e_2$	e_1	e_2
	e_2	e_1	e_2	e_1	$-e_2$	$-e_1$	$-e_2$	$-e_1$

Table A.1: Set $\boldsymbol{X}$ points coordinates

Let, $v_1(\boldsymbol{x})$ be defined as $v_1(\boldsymbol{x}) = \boldsymbol{w}_1^T \oplus \boldsymbol{x} + b_1$. To separate $\boldsymbol{x}_1$ from the points in the set there should exist $\boldsymbol{w}_1$ and b_1 such that:

$$\begin{aligned} h_1(\boldsymbol{x}) &= sgn(v_1(\boldsymbol{x})) \\ &= sgn(sgn(\boldsymbol{w}_1^T)\boldsymbol{x} + \boldsymbol{w}_1^T sgn(\boldsymbol{x}_1) + b_1) = \begin{cases} 1 & \text{if } \boldsymbol{x} = \boldsymbol{x}_1 \\ 0 & otherwise \end{cases} \end{aligned} \tag{A.6}$$

Lemma A.2.1. *To separate $\boldsymbol{x}_1$ from $\boldsymbol{x}_2$, $sgn(w_{11}) = -sgn(w_{21})$ condition should be met.*

Proof. If $sgn(w_{11}) = sgn(w_{21}) = 1$, then $v_1(\boldsymbol{x_1}) = e_1 + w_{11} + e_2 + w_{21} + b_1$ and $v_1(\boldsymbol{x_2}) = e_2 + w_{21} + e_1 + w_{11} + b_1 = v_1(\boldsymbol{x_1})$. Therefore, $h_1(\boldsymbol{x}_1) = h_1(\boldsymbol{x}_2)$ is always true. This still holds when $sgn(w_{11}) = sgn(w_{21}) = -1$. □

Lemma A.2.2. *To separate $\boldsymbol{x}_1$ from $\boldsymbol{x}_3$ and $\boldsymbol{x}_7$ $sgn(w_{11}) = sgn(w_{21})$ condition should be met.*

Proof. Let $sgn(w_{11}) = 1$ and $sgn(w_{21}) = -1$, then
$v_1(\boldsymbol{x_1}) = e_1 + w_{11} - e_2 + w_{21} + b_1$, $v_1(\boldsymbol{x_3}) = -e_1 - w_{11} - e_2 + w_{21} + b_1$,
$v_1(\boldsymbol{x_7}) = e_1 + w_{11} + e_2 - w_{21} + b_1$.
From these we get $v_1(\boldsymbol{x_3}) \leq v_1(\boldsymbol{x_1}) \leq v_1(\boldsymbol{x_7})$ since $e_1 > e_2 > 0$.

Therefore, we cannot separate $\boldsymbol{x}_1$ from $\boldsymbol{x}_3$ and $\boldsymbol{x}_7$, since no matter what value b_1 we choose, we cannot achieve the criteria of $h_1(\boldsymbol{x}_1) = 1$ while $h_1(\boldsymbol{x}_3) = 0$ and $h_1(\boldsymbol{x}_7) = 0$. This still holds when $sgn(w_{11}) = -1$ and $sgn(w_{21}) = 1$. □

From Lemma A.2.1 and Lemma A.2.2 we get a contradiction of the sign values of w_{11} and w_{21}. Therefore, $\boldsymbol{x}_1$ cannot be separated from the set $\boldsymbol{X}$. Similarly we can prove that any $\boldsymbol{x}_i \in \boldsymbol{X}$ cannot be separated from the set $\boldsymbol{X}$ using a MFNN.

Using a one-hidden-layer MFNN, we can separate $\boldsymbol{x}_1$ from $\boldsymbol{x}_i$ for $i = 2, 3...6$, but not from $\boldsymbol{x}_7$ and $\boldsymbol{x}_8$.

A.2.2 Binary-Weight Network

Similarly to the discussion of Section A.2.1, let $\boldsymbol{X}$ be the set of points defined in Table A.1. Let $v_1(\boldsymbol{x})$ be defined as $v_1(\boldsymbol{x}) = sgn(\boldsymbol{w}_1^T)\boldsymbol{x} + b_1$. To separate $\boldsymbol{x}_1$ from the points in the set there should exist $\boldsymbol{w}_1$ and b_1 such that:

$$\begin{aligned} h_1(\boldsymbol{x}) &= sgn(v_1(\boldsymbol{x})) \\ &= sgn(sgn(\boldsymbol{w}_1^T)\boldsymbol{x} + b_1) = \begin{cases} 1 & \text{if } \boldsymbol{x} = \boldsymbol{x}_1 \\ 0 & otherwise \end{cases} \end{aligned} \tag{A.7}$$

Lemma A.2.3. *To separate $\boldsymbol{x}_1$ from $\boldsymbol{x}_2$, $sgn(w_{11}) = -sgn(w_{21})$ condition should be met.*

Proof. If $sgn(w_{11}) = sgn(w_{21}) = 1$, then $v_1(\boldsymbol{x_1}) = e_1 + e_2 + b_1$
and $v_1(\boldsymbol{x_2}) = e_2 + e_1 + b_1 = v_1(\boldsymbol{x_1})$. Therefore, $h_1(\boldsymbol{x}_1) = h_1(\boldsymbol{x}_2)$ is always true. This still holds when $sgn(w_{11}) = sgn(w_{21}) = -1$. □

Lemma A.2.4. *To separate $\boldsymbol{x}_1$ from $\boldsymbol{x}_6$, $sgn(w_{11}) = sgn(w_{21})$ condition should be met.*

Proof. If $sgn(w_{11}) = 1$ and $sgn(w_{21}) = -1$, then $v_1(\boldsymbol{x_1}) = e_1 - e_2 + b_1$
and $v_1(\boldsymbol{x_6}) = -e_2 + e_1 + b_1 = v_1(\boldsymbol{x_1})$. Therefore, $h_1(\boldsymbol{x}_1) = h_1(\boldsymbol{x}_6)$ is always true. This still holds when $sgn(w_{11}) = -1$ and $sgn(w_{21}) = 1$. □

From Lemma A.2.3 and Lemma A.2.4 we get a contradiction of the sign values of w_{11} and w_{21}. Therefore, $\boldsymbol{x}_1$ cannot be separated from the set $\boldsymbol{X}$. Similarly we can prove that any $\boldsymbol{x}_i \in \boldsymbol{X}$ cannot be separated from the set $\boldsymbol{X}$ using a BWN.

Using a one-hidden-layer BWN, we can separate $\boldsymbol{x}_1$ from $\boldsymbol{x}_2, \boldsymbol{x}_3, \boldsymbol{x}_4$, but not from $\boldsymbol{x}_i$ and for $\boldsymbol{x}_5, \boldsymbol{x}_6, \boldsymbol{x}_7, \boldsymbol{x}_8$.

A.3 Summary

We proved earlier that MFNNs satisfy the Universal Approximation Theorem. Similarly, this could be proven for BWNs. However, both networks require more than one-hidden-layer. Also we showed that using MFNN is slightly better than BWN in terms of separating the data, although both fail to separate the data perfectly using only one-hidden-layer network. Binary-Weight neurons can separate only half of the selected set $\boldsymbol{X}$, while the multiplication free neuron can separate three-quarters of the same selected set $\boldsymbol{X}$. This is due to the extra term in the mf-operator. which stores the absolute value of the weight, while the Binary-Weight operator only uses the sign information.