DESIGN OF ENERGY ADAPTIVE NEURAL NETWORKS USING APPROXIMATE COMPUTING AND PARTIAL DYNAMIC RECONFIGURATION

By

Salma Hassan Sayed Abo Elmagd

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfilment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Electrical Engineering

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2020

DESIGN OF ENERGY ADAPTIVE NEURAL NETWORKS USING APPROXIMATE COMPUTING AND PARTIAL DYNAMIC RECONFIGURATION

By

Salma Hassan Sayed Abo Elmagd

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfilment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Electrical Engineering

Under the Supervision of

Dr.Hassan Mostafa	Dr. Ahmed Nader
Position	Position
Electronics and Electrical Communication Engineering	His Department
Department	Department
Faculty of Engineering, Cairo University	Faculty of Engineering, Cairo University

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2020

DESIGN OF ENERGY ADAPTIVE NEURAL NETWORKS USING APPROXIMATE COMPUTING AND PARTIAL DYNAMIC RECONFIGURATION

By

Salma Hassan Sayed Abo Elmagd

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfilment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Electrical Engineering

Approved by the Examining Committee:

Dr.Hassan Mostafa, Thesis Main Advisor

Prof. First E. Name, Member

Prof. Second S. Name, Internal Examiner

Prof. Third S. Name, External Examiner
(Some Faculty, Some University)

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2020

Engineer’s Name: Salma Hassan Sayed Abo Elmagd
Date of Birth: 16/11/1992
Nationality: Egyptian
E-mail: salmahassansayed@gmail.com
Phone: 0100078029
Address: Electronics and Electrical Communication Engineering
Department, Cairo University,
Giza 12613, Egypt

Insert photo here

Registration Date: dd/mm/yyyy
Awarding Date: dd/mm/yyyy
Degree: Master of Science
Department: Electronics and Electrical Communication Engineering

Supervisors:
Dr.Hassan Mostafa
Dr. Ahmed Nader

Examiners:
Dr.Hassan Mostafa (Thesis main advis
Prof. First E. Name (Member)
Prof. Second S. Name (Internal examiner
Prof. Third S. Name (External examine

Title of Thesis:

DESIGN OF ENERGY ADAPTIVE NEURAL NETWORKS
USING APPROXIMATE COMPUTING AND PARTIAL
DYNAMIC RECONFIGURATION

Key Words:
ANN; SNN; neuromorphic computing; PDR; approximate computing

Summary:
This thesis represents the idea of using partial dynamic reconfiguration in the design and implementation of adaptive artificial neural network. The thesis also represents the idea of using approximate multipliers in the design of Izhikevich neuron model

Acknowledgments

First and foremost, I would like to thank God for giving me the strength, ability and opportunity to complete this work.

I can't express enough thanks to my family for their continuous support and encouragement throughout my journey.

Special thanks to Dr.Hassan Mostafa, my supervisor, and Eng.Sameh Attia how supported me with many ideas to have this thesis in that shape.

Finally, Thanks to my friend Manar, how helped me in every single step in this thesis.

Dedication

To my mother, my father

Without you both, I wouldn't have been anywhere

Thank you for everything!

Table of Contents

Acknowledgments i

Dedication ii

Table of Contents iii

List of Tables vi

List of Figures vii

List of Publications xi

List of Symbols and Abbreviations xi

Abstract xii

1 Introduction 1

- 1.1 Neural Networks 1
- 1.2 Neural Network Types 3
  - 1.2.1 Artificial neural network 3
  - 1.2.2 Convolutional Neural Network 3
  - 1.2.3 Spiking Neural Network 3
- 1.3 Organization of the thesis 4

2 Literature Review 5

- 2.1 Introduction to artificial neural network and its hardware components . . 5
- 2.2 Approximate computing in the hardware implementation of artificial neural networks 9
- 2.3 Literature Review for Approximate Hardware Implementations of Artificial Neural Network 10
- 2.4 Literature Review of the Approximate Multipliers 17
  - 2.4.1 Broken Array Multiplier (BAM) 17
  - 2.4.2 Under-designed Multiplier (UM) 20
  - 2.4.3 Error Tolerant Multiplier (ETM) 22
  - 2.4.4 Truncated Multiplier 24

2.5 Introduction to Spiking Neural Networks and neuron models approximation 25
  2.5.1 Biologically plausible models 27
    2.5.1.1 Hodgkin-Huxley model 27
    2.5.1.2 Morris Lecar model 28
    2.5.1.3 Fitzhugh-Nagumo model 29
  2.5.2 Integrate and fire models 29
    2.5.2.1 Leaky integrate and fire 29
    2.5.2.2 Quadratic Integrate and fire model 30
    2.5.2.3 Exponential integrate and fire model 31
  2.5.3 Biologically Inspired models 31
    2.5.3.1 Hindmarsh-Rose model 31
    2.5.3.2 Mihalas-Niebur model 32
    2.5.3.3 The Quartic model 33
    2.5.3.4 Izhikevich neuron model 33

3 Design of Adaptive Artificial Neural Network using Approximate Computing and Partial Dynamic Reconfiguration and Experimental Results 35
3.1 Introduction to Artificial Neural Network Learning Process 35
3.2 Research Hypothesis 36
3.3 Approximate Computing Techniques 37
  3.3.1 Precision Scaling 37
  3.3.2 Approximate Activation Function 39
    3.3.2.1 Sigmoid Activation Function 39
    3.3.2.2 RELU Activation Function 40
  3.3.3 Computation Skipping Approximation 41
  3.3.4 Neuron Skipping Approximation 42
  3.3.5 Inaccurate Arithmetic 43
  3.3.6 Approximate Adders 43
3.4 Design Approach 43
  3.4.1 Data-sets 44
    3.4.1.1 MNIST data-set and its artificial neural network architecture 44
    3.4.1.2 SVHN data-set its artificial neural network architecture 45
  3.4.2 Partial Dynamic Reconfiguration (PDR) 46
3.5 Experimental Setup 48
  3.5.1 Software Setup 49
  3.5.2 Hardware Setup 49
    3.5.2.1 MNIST Block Diagram 50
    3.5.2.2 SVHN Block Diagram 50
    3.5.2.3 Neuron Top Block Diagram 52
3.6 MNIST Results 53
  3.6.1 MNIST Energy results 53

3.6.2 Effect of computation skipping 53
3.6.3 Sigmoid Vs RELU 55
3.7 SVHN Results 56
3.7.1 SVHN Energy results 56
3.8 Proposed Algorithm and Configurations Selection 57
3.8.1 comparison between conventional system and EANN system . . . 60
3.8.2 Partial Dynamic Reconfiguration Results 60
3.9 Conclusion 63

4 Design and Implementation of Izhikevich neuron model using Approximate Multiplier 65
4.1 Izhikevich neuron model 65
4.1.1 Izhikevich neuron patterns experimental results 66
4.1.1.1 Regular spiking 66
4.1.1.2 Intrinsically Bursting 67
4.1.1.3 Chattering 68
4.1.1.4 Fast Spiking 68
4.1.1.5 Low Threshold spiking 68
4.2 Piece Wise Linear Implementations of Izhikevich neuron model 70
4.3 Approximate Multiplier Based implementation of Izhikevich neuron model 73
4.3.1 Truncated Multiplier Implementation 73
4.3.2 Error Tolerant Multiplier Approximation 74
4.3.3 Broken Array Multiplier Approximation 76
4.4 Proposed approximate multiplier based Izhikevich model 80
4.5 Error Analysis 85
4.5.1 ERRt 85
4.5.2 NRMSD 88
4.5.3 MRE 89
4.5.4 PWL4 model Error Results 89
4.5.5 Approximate multiplier based model Error Results 89
4.6 Hardware Implementation 90
4.6.1 PWL4 hardware implementation results 92
4.6.2 Approximate multiplier based hardware implementation results . 92
4.7 Network Behavior of the proposed Approximate multiplier model 93
4.8 Comparison between the PWL4 neuron model and Approximate Multiplier based model 94
4.9 Conclusion 96

5 Future Work 98

References 99

Arabic Abstract

List of Tables

2.1 Multiplication operation decomposition example 14
2.2 Map for the inaccurate 2x2 multiplier for all possible input combinations 20

3.1 A-law approximation of sigmoid function 39
3.2 power and accuracy regions for MNIST data-set 58
3.3 power and accuracy regions for MNIST data-set 59
3.4 Area, Power and accuracy results for SVHN configurations 62
3.5 Area, Power and accuracy results for MNIST configurations 63

4.1 PWL4 Error results using WL=17 89
4.2 PWL4 Error results using WL=16 90
4.3 PWL4 Error results using WL=15 90
4.4 PWL4 Error results using WL=14 90
4.5 Truncated multiplier error results using WL=16 and FL=7 91
4.6 Truncated multiplier error results using WL=15, FL=6 91
4.7 Truncated multiplier error results using WL=14, FL=5 91
4.8 Hardware implementation results of PWL4 using WL=17 92
4.9 Hardware implementation results of PWL4 using WL=16 92
4.10 Hardware implementation results of PWL4 using WL=15 92
4.11 Hardware implementation results of PWL4 using WL=15 92
4.12 Hardware implementation results of truncated multiplier using WL=16, FL=7 93
4.13 Hardware implementation results of truncated multiplier using WL=15, FL=6 93
4.14 Hardware implementation results of truncated multiplier using WL=14 , FL=5 93

List of Figures

1.1 Simplified biological neuron [2] 2

2.1 Basic structure of Artificial Neural Network 6
2.2 Single artificial neuron structure 6
2.3 Sigmoid activation function 7
2.4 Tanh activation function 8
2.5 RELU activation function 8
2.6 Design steps of Approximate Neural Network [5] 12
2.7 Hardware architecture of the quality control neuromorphic processing engine [5] 13
2.8 Hardware architecture of processing element and memory [6] 13
2.9 Operation of Alphabet Set Multiplier [8] 14
2.10 Retraining algorithm for neural network uses Alphabet Set Multiplier [8] 15
2.11 Differences between synapses approximation and neuron approximation [10] 16
2.12 Hardware structure of the Broken Array Multiplier (BAM) [11] 18
2.13 (A) Carry Save Adder cell used in the broken array multiplier , (B) Vector merging adder cell in the broken array multiplier 18
2.14 The effect of increasing horizontal break level and vertical break level on the mean error of the broken array multiplier output [11] 19
2.15 Hardware implementation of 2x2 accurate multiplier unit 21
2.16 Hardware implementation of 2x2 under-designed multiplier unit 21
2.17 Building larger multipliers from smaller ones in the under-designed multiplier 22
2.18 Example of generating larger multipliers from smaller multiplier units in the under-designed multiplier method 23
2.19 Algorithm of the Error Tolerant Multiplier [13] 23
2.20 Truncated signed multiplier partial product matrix with n=8 and h=2 [14] 25
2.21 Main spiking patterns of the biological neuron [15] 26
2.22 Electrical circuit model of Hodgkin-Huxley biologically plausible neuron [16] 27
2.23 Leaky integrate and fire circuit model [18] 30

2.24 Trade off between the complexity of the neuron model vs its biological characteristics [3] 34

3.1 Limited-Energy application block diagram for using energy adaptive neural networks 37
3.2 Simplified block diagram illustrating the computation skipping in the hardware implementation 41
3.3 Sample data of MNIST data-set [25] 45
3.4 Sample data of SVHN data-set [26] 46
3.5 Illustration of Partial Reconfiguration 48
3.6 Block Diagram of the Network used for MNIST Data-set 51
3.7 Block Diagram of the Network used for SVHN Data-set 51
3.8 Block Diagram of the neuron unit used in both MNIST and SVHN networks 52
3.9 MNIST : Approximation results for MNIST data-set, sig: Sigmoid activation function, Cs: Computation skipping, AM: Approximate Multiplier, TA: Truncated Accumulation, Ns: Neuron skipping, RELU: RELU activation function 54
3.10 MNIST : Effect of computation skipping on the accuracy versus consumed energy 54
3.11 MNIST : Comparison between Sigmoid and RELU in terms of Energy and Accuracy Loss 55
3.12 MNIST : Configuration points used as a searching space to choose the suitable implementation at a given energy for MNIST data-set, sig: Sigmoid activation function, Cs: Computation skipping, AM: Approximate Multiplier, TA: Truncated Accumulation, Ns: Neuron skipping 56
3.13 Approximation results for SVHN data-set, No-app: No approximation, Ns: Neuron skipping, TA: Truncated Accumulation 57
3.14 MNIST: Energy levels that the proposed EANN system uses to adapt to the given energy budget 58
3.15 SVHN: Energy levels that the proposed EANN system uses to adapt to the given energy budget 59
3.16 Comparison between conventional systems and energy adaptive system when exposed to variable energy 61
3.17 SVHN : Floor-planning and static routing (re-configurable area) of the physical neurons in case of SVHN data-set 62
3.18 MNIST : Floor-planning and static routing (re-configurable area) of the physical neurons in case of MNIST data-set 62

4.1 Izhikevich model parameters and the resulting spiking patterns RS(Regular Spiking), IB (Intrinsically Bursting), CH(Chattering), FS(Fast Spiking), LTS(Low Threshold Spiking) [28] 66

4.2 Regular spiking pattern, and the increasing inter-spike frequency at increasing the input dc current 67
4.3 Intrinsically bursting spiking pattern 68
4.4 Regular spiking pattern, and the increasing inter spike frequency at increasing the input dc current 69
4.5 Fast spiking pattern 69
4.6 Low threshold spiking firing pattern 70
4.7 Second-Order Piece-wise Linear approximation 71
4.8 Third-Order Piece-wise Linear approximation 72
4.9 Fourth-Order Piece-wise Linear approximation 72
4.10 U-V plot for the truanted multiplier implementation for different fraction lengths, FL: Fraction Length 75
4.11 Maximum error vs the fraction length in the truncated multiplier implementation 76
4.12 Different spiking patterns reproduced using the truncated multiplier approximation with fraction length = 4, RS: Regular Spiking, IB: Intrinsically Bursting, CH: Chattering, FS: Fast Spiking, LTS: Low Threshold Spiking 77
4.13 Different spiking patterns reproduced using the truncated multiplier approximation with fraction length = 8, RS: Regular Spiking, IB: Intrinsically Bursting, CH: Chattering, FS: Fast Spiking, LTS: Low Threshold Spiking 78
4.14 U-V plot for the error tolerant multiplier implementation for different non multiplication parts, NMB: Non Multiplication Part 79
4.15 Maximum error vs Non multiplication part in the error tolerant multiplier implementation 80
4.16 Different spiking patterns reproduced using the error tolerant multiplier approximation with non multiplication part = 8, RS: Regular Spiking, IB: Intrinsically Bursting, CH: Chattering, FS: Fast Spiking, LTS: Low Threshold Spiking 81
4.17 Different spiking patterns reproduced using the error tolerant multiplier approximation with non multiplication part = 8, RS: Regular Spiking, IB: Intrinsically Bursting, CH: Chattering, FS: Fast Spiking, LTS: Low Threshold Spiking 82
4.18 U-V plot for the broken array multiplier implementation for different horizontal break level values from 12 to 7, HBL: Horizontal Break Level . . 83
4.19 U-V plot for the broken array multiplier implementation for different horizontal break level values from 6 to 1, HBL: Horizontal Break Level . . . 84
4.20 Maximum error vs Horizontal break level in the broken array multiplier implementation 85

4.21 Different spiking patterns reproduced using the broken array multiplier approximation with horizontal break level=9, RS: Regular Spiking, IB: Intrinsically Bursting, CH: Chattering, FS: Fast Spiking, LTS: Low Threshold Spiking 86
4.22 Different spiking patterns reproduced using broken array multiplier approximation with horizontal break level =4, RS: Regular Spiking, IB: Intrinsically Bursting, CH: Chattering, FS: Fast Spiking, LTS: Low Threshold Spiking 87
4.23 ERRt calculation as the time difference between the spikes of the proposed model and the original model 88
4.24 Raster diagram of a network of 1000 connected neurons modeled using the approximate multiplier based implementation compared to the original model output 94
4.25 Spikes count of randomly selected neurons from the network for both the original model and the proposed model 95
4.26 Figure of Merit (FoM)of both approximate multiplier based model and the PWL4 model Versus Word Length(WL) 96

List of Publications

Published:

[1] S. Hassan, S. Attia, K. N. Salama, and H. Mostafa, “Eann: Energy adaptive neural networks,” *Electronics*, vol. 9, no. 5, p. 746, May 2020. [Online]. Available: http://dx.doi.org/10.3390/electronics9050746.

[2] S. Hassan, K. N. Salama, and H. Mostafa, “An approximate multiplier based hardware implementation of the izhikevich model,” in *2018 IEEE 61st International Midwest Symposium on Circuits and Systems (MWSCAS)*, IEEE, 2018, pp. 492–495.

Abstract

Artificial intelligence is a new era that depends mainly on the machine learning. Neural networks are one of the well known algorithms used in the machine learning field. Neural networks have emerged in the last decades due to their computation efficiency and their ability to perform intensive computations in a fast and efficient way. Several types of neural networks are designed to solve many problems such as classification, pattern recognition and prediction. They also have a great role in the area of computer vision and autonomous cars. Several types of neural networks have evolved as time passes. All of them have the vision to mimic how the human brain identifies and processes the information, how the learning process of the information is held in the brain, and how it manipulates the information and many other human brain processes.

The simplest type of neural networks is the artificial neural networks(ANNs), that are used to solve problems such as pattern recognition and classifications. They are simple enough to be implemented on a hardware platform. The second type of networks that emerged to solve more complex problems such as image processing, speech processing, and video processing is the convolutional neural networks(CNNs), that have more computational capabilities than the ANNs.

The third generation of the neural networks is the spiking neural networks(SNNs). SNNs are the most realistic neural networks, as they are not only mathematical models that mimic the processing capabilities of the brain, but they are also biological models that contain the biological behaviors of the brain such as the spiking patterns and the timing relations between the inputs and the outputs.

The hardware implementation of the neural networks is a challenging task as the area and power consumption are the main metrics that govern the hardware design. Approximate computing is also a well known technique adopted in the hardware implementation of the neural network to balance the power, area, and accuracy. Using approximate computing in the neural networks hardware units sacrifices some accuracy to gain less area and power consumption. This is acceptable in the applications that use neural networks and do not require high accuracy due to the error resilient nature of the machine learning algorithm.

In this thesis, several hardware approximation techniques are used in the implementation of the artificial neural network such as precision scaling, computation skipping, neuron skipping, approximate activation functions, approximate multipliers, and truncated accumulation. These techniques are then implemented on the FPGA platform. The idea in the thesis is that the partial dynamic reconfiguration feature of the FPGA is used to make the hardware adaptable to the changes in the battery energy that feeds the system. The hardware adapts and reconfigures its units in response to the input energy at the expense of accuracy degradation.

The spiking neural networks are also investigated in the thesis. First, a survey of the neuron models is made from the biologically plausible models to the biologically inspired ones. Izhikevich neuron model is one of the well known models due to its ability to reproduce the spiking patterns seen in the biological experiments and due to its mathematical simplicity. The main drawback of this model is the square operation used in its equations. This thesis provides a new approximate multiplier based implementation to that model. Following that, the approximate multiplier based model is compared with the well known piece-wise linear implementations.

Chapter 1

Introduction

Neural Networks are brain-inspired systems that mimic human brain behavior and how it processes the information. They are used in a wide range of applications that need intensive and complex processing and that perform many operations in parallel. Some of the applications that use neural networks are classification, pattern recognition, image recognition, data analysis, and computer vision[1]. Neural networks have achieved superiority over Von-Neumann computing systems in such applications since they overcome the bottleneck between the instruction memory, data memory, and the CPU. Neural networks process the information in a parallel manner, thus reducing the processing time and make it faster than the conventional systems.

There are several types of neural networks, each type has a different structure and a different way of data processing. Neural network types are also different in their complexity level, the degree of mimicking the biological brain and the complexity of data processed by the network with good accuracy. Some of the neural network types are Artificial Neural Network(ANN), Convolutional Neural Network(CNN) ,and Spiking Neural Network(SNN).

1.1 Neural Networks

In the human brain, there are around 100 billion interconnected neurons. Each neuron is connected to about 10000 neighboring neurons and receives stimuli from them. The information flows from the weighted synapses to the axon. Then the axon of each neuron conveys the information to its neighboring neurons[1]. Figure 1.1 shows a simplified model of a biological neuron.

The biological neuron itself consists of dendrites, each dendrite is connected to another one through an axon, the junction between the dendrite and the axon is the synapse and then the synapses are connected to the cell body. The dendrites are where the cell body receives its inputs. The axons are where the cell body transmits its outputs to other neurons. The connection between one dendrite of a neuron to an axon of another neuron is the

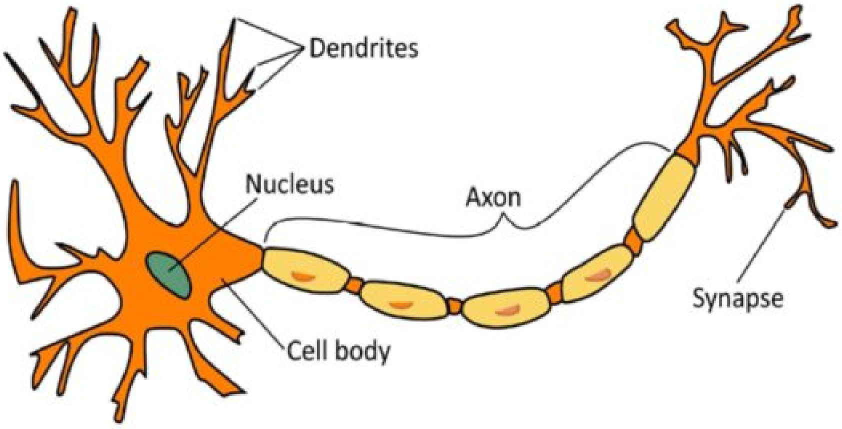


Figure 1.1: Simplified biological neuron [2]

synapse, each synapse has a certain weight that allows a different weight for each neuron input.

The information is transmitted between the neurons in the form of chemical or electrical signals. The neuron body uses these input signals and accumulates charges in the form of voltage, and when the voltage potential exceeds a certain threshold the neuron fires and generates a signal that is transmitted to another neuron as an input pulse through the axon. The operation of the neuron is to accumulate charges, compare the potential with a threshold, and then fire a spike to transmit the output.

Based on the explanation of the biological neuron nature, several neuron models are implemented to have the same effect of the biological neuron and to process the information in the same manner. Some neuron models are biologically plausible and others are biologically inspired. The neuron models are categorized into three categories as follows[3]:

1. Biologically plausible: The neuron models that implement the biological neuron features in the same way as seen in biological systems, thus the physical characteristics are taken into consideration in such models. The dendrites, axons models, and membrane dynamics are modeled in these models.
2. Biologically inspired: The neuron models that mimic the same biological neuron's behavior such as the firing patterns, however, they do not have the physical parameters of the biological neurons.
3. Integrate and fire: They are a simpler category of the biologically inspired neuron models.

1.2 Neural Network Types

1.2.1 Artificial neural network

The artificial neural network (ANN) consists of several connected artificial neurons that form a simplified system to mimic the biological neurons. The excitation is applied to the network inputs, then it is processed and passed through the activation function to produce the output. The neurons are connected in a network by weighted synapses. The synapses weights are learned in the learning phase to predict the output at high accuracy. The artificial neural networks are simple in their architecture compared to the convolutional neural networks or the spiking neural networks. They consist of the input layer, one or more hidden layer and an output layer. The applications that use artificial neural networks are pattern recognition applications and classification tasks.

1.2.2 Convolutional Neural Network

The second type of neural networks that emerged to solve more complex problems such as image processing, speech processing, and video processing is the convolutional neural networks(CNNs). They have more computational capabilities than ANNs. They can be seen as the evolution of artificial neural networks. The CNNs are mainly composed of some convolution layers used for feature extraction, and then fully connected layers for the classification task.

1.2.3 Spiking Neural Network

Spiking neural networks are the third generation in the field of machine learning and neural networks. Unlike artificial neural networks, spiking neural networks use biological and realistic models to mimic the human brain processing and thus perform the computations in a biological manner. A spiking neural network (SNN) applies the concept of spikes, which is the same as the human brain's way of processing and manipulating information. In the human brain, when a neuron cell is exposed to a certain stimulus, the membrane potential increases and when it exceeds a certain threshold it spikes. These spikes travel from one neuron to its neighboring neurons for further information processing. There are two types of neurons: excitatory neurons and inhibitory. The excitatory neurons result in a positive change to the membrane potential. The inhibitory neurons result in a negative change to the membrane potential. The spikes travel from one neuron to another conveying information necessary for processing. The information herein these models is contained in the presence or absence of a spike. The cell potential resets after a spike and it remains under reset till it is exposed to another stimulus, during the reset time it does not produce any spikes. The spiking neural networks operate with a discrete spikes that occur at a certain point in time not a continuous value of inputs. There are various neuron models developed over the years. Some of them are biologically plausible models that are developed from the lab experiments and observations. The others are only mathematical models used to produce the same biological spiking patterns.

1.3 Organization of the thesis

The next chapters of the thesis are organized as follows: Chapter 2 is the literature review of the thesis. It begins with an introduction to the artificial neural network, its architecture and the activation functions used in it. It also introduces the approximate computing techniques used in the literature. The second part of the review is the literature review of the common approximate multipliers techniques.The last part of the chapter is the introduction to spiking neural networks and the neuron models different approximations.

Chapter 3 is the design approach and methodology used to design an adaptive artificial neural network using approximate computing and partial dynamic reconfiguration techniques. First, it explains the learning process of the network and the learning algorithm. Then, it explains the design methodology and the PDR technique.

Chapter 4 is the Experimental setup and results of the adaptive artificial neural network implementation. It explains the hardware and software setup used to obtain the listed results. It also explains the block diagram of both the networks used for MNIST and SVHN data-sets. And at last the results obtained for both MNIST and SVHN data-sets.

Chapter5 is an explanation of the Izhikevich neuron model. It also introduces the main used hardware implementation of the Izhikevich neuron model, i.e., the piece wise linear implementation. Next, the results of utilizing the approximate multipliers in Izhikevich implementation are listed. The error metrics used for judging a model are explained. The hardware implementation approach of the approximate multiplier based model and the PWL model are explained. The last thing is the comparison between the PWL model and the approximate multiplier based model.

Chapter 6 is the last chapter and it contains the future work intended in the thesis.

Chapter 2

Literature Review

2.1 Introduction to artificial neural network and its hardware components

The artificial neural network (ANN) consists of many connected artificial neurons to form a simplified system mimicking the biological neurons. The excitation is applied to the network inputs, then it is processed and passed through the activation function to produce the output. The neurons are connected in a network by weighted synapses, the weights are learned in the learning phase to predict the output at high accuracy.

The simplest type of neural networks is ANN. It is a feed-forward neural network, where data propagates from the input layer to the output layer without any feedback. The main structure of ANN is illustrated in Figure2.1. In Figure2.1, the network is structured as an input layer, one or more hidden layers and an output layer. The layers are fully connected where neurons are connected by weighted synapses, thus each neuron output is defined by the layer inputs and the weights. Let X be the layer input, and yi is the i-th output from the fully connected layer, the yi is calculated as follows:

$$yi = \phi(w1x1 + w2x2 + ... + wnxn) \tag{2.1}$$

where ϕ is the activation function used at that layer.

In each layer, the basic processing unit is the neuron unit as shown in Figure 2.2. It consists of a multiplier, an accumulator and an activation function. The input data is multiplied by its corresponding weights, then they are added in the accumulator, after that the output passes through an activation function to add some non-linearity to the product to mimic the human brain behavior. The output of each layer is the input to the next layer until the output layer is reached.

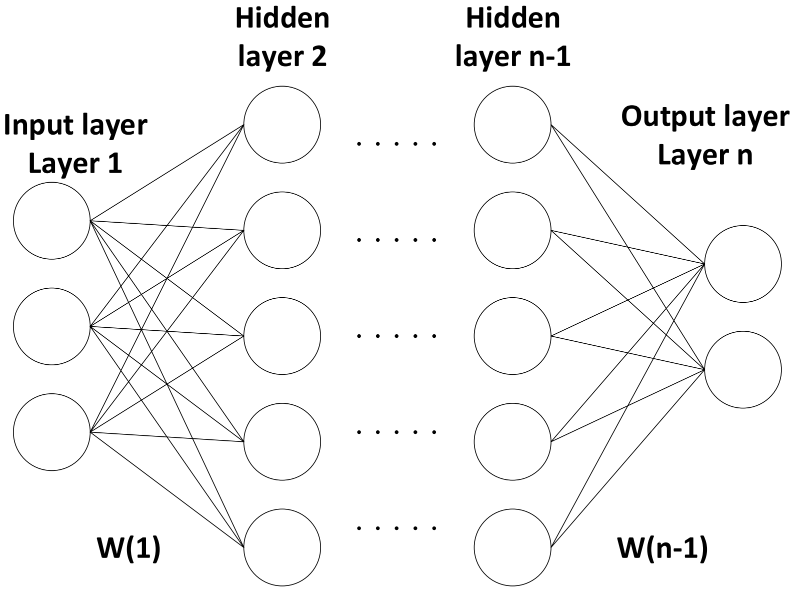


Figure 2.1: Basic structure of Artificial Neural Network

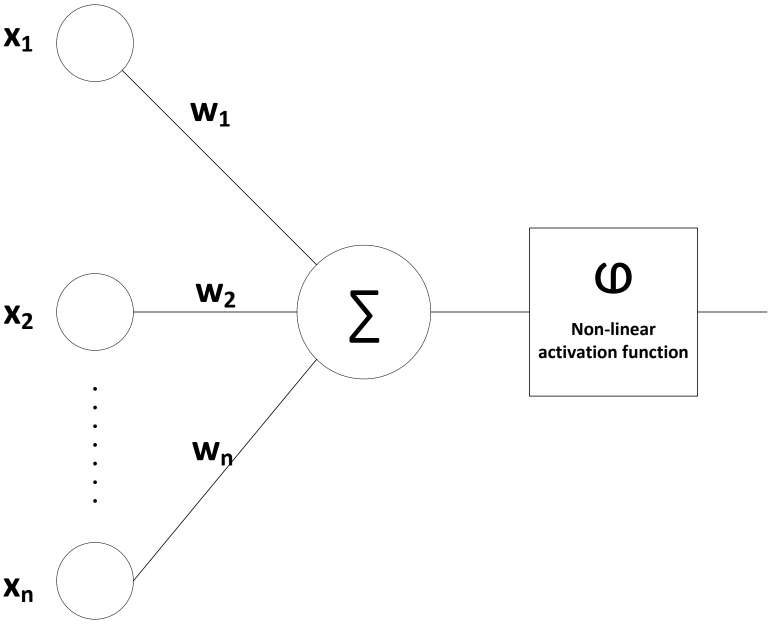


Figure 2.2: Single artificial neuron structure

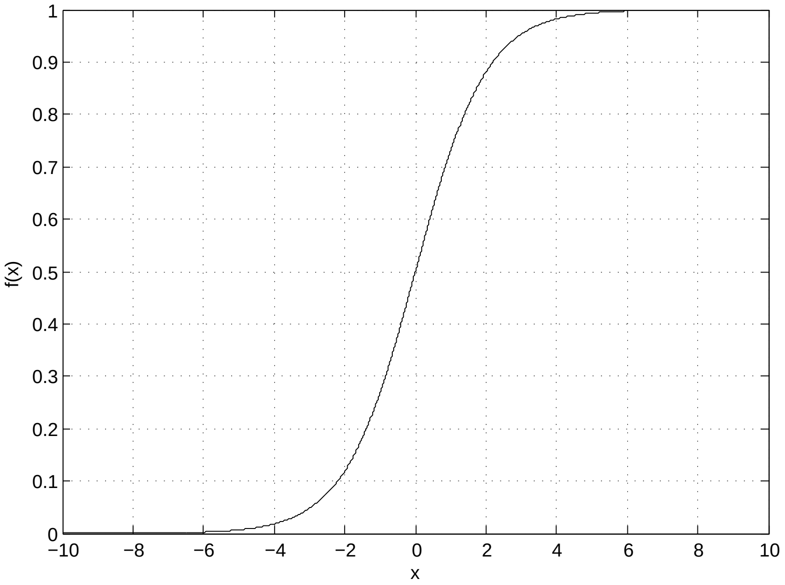


Figure 2.3: Sigmoid activation function

The rule of the activation function is to add non-linearity to the neural network to help to solve more complex problems. Activation functions attached to each neuron are used to determine whether that neuron will produce output due to the stimuli or not. They are also used to normalize the neurons' output in a defined range (-1, 1) or (0,1) depending on its type.

The most commonly used activation functions are Sigmoid and RELU :

1. Sigmoid:
   It normalizes the output between 0 and 1, deep negative inputs results in 0 output and deep positive input result in 1 output. Outputs are centered around 0.5 as shown in Figure2.3.
   Its formula is as follows:
   $$f(x) = \frac{1}{1+e^{-x}} \tag{2.2}$$

2. Tanh:
   It is similar to the sigmoid function, but it normalizes the output between 1 and -1, and it is centered around 0 as shown in Figure 2.4.

3. RELU:
   RELU has a different behavior other than the sigmoid function, it allows all values that are greater than 0 to propagate with the same value, however, the values that are

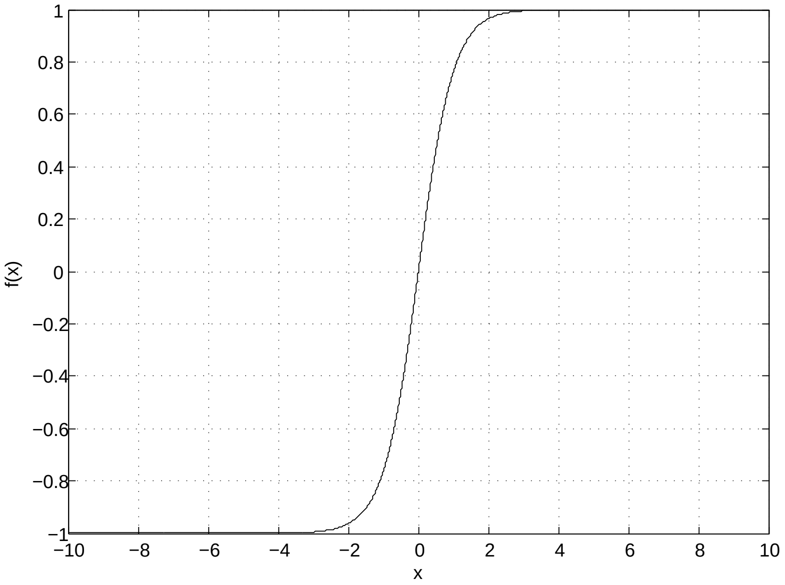


Figure 2.4: Tanh activation function

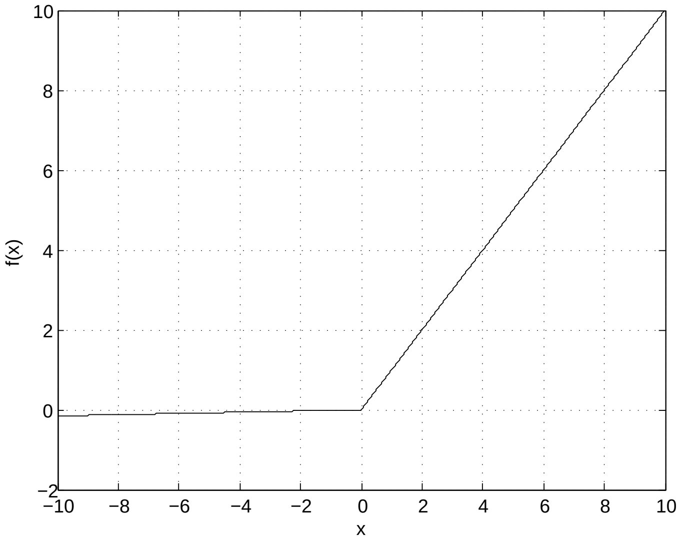


Figure 2.5: RELU activation function

less than 0 are multiplied by a very small value as shown in Figure 2.5. Following is the equation of RELU activation function.

$$f(x) = \begin{cases} x & x \geq 0 \\ 0.015625 * x & x < 0 \end{cases} \tag{2.3}$$

2.2 Approximate computing in the hardware implementation of artificial neural networks

Hardware implementation of neural networks is challenging since it trades the accuracy with the complexity of the hardware components. ANNs require intensive computations, they use a large number of processing elements (neurons). As the number of layers or the number of neurons in each layer increases, the power and the area of the hardware system is increased as well. Hardware implementation of such networks has a lot of challenges. ANNs are defined by many parameters: network architecture, learning approach (online-offline), activation function, hardware precision, number of layers, number of neurons in each layer, and weights' memory. Each one of these parameters contributes to the hardware implementation.

There is a variety of options when implementing an ANN. It can be all digital, all analog, or a mix of both technologies, it can be implemented using memristors. If it is all digital, it may be implemented on an FPGA or ASIC. What governs the choice of a certain technology is the application that uses the ANN, maybe the application needs low power, high speed, small area, high accuracy or a combination of these specifications

In [4], it presents an overview of the examples of various hardware implementations across a wide range of ANN models. The fact that ANNs are used in error resilience applications has permitted some approximation in the hardware implementation of them. A lot of research has been carried out to achieve energy efficient hardware implementation for the ANNs and that results of many hardware approximation techniques such as computation skipping, neurons skipping, inaccurate arithmetic (approximate multipliers and approximate adders), truncated accumulation, precision scaling, and approximate activation functions.

The use of hardware approximation techniques help to reduce the area and power which are the major concern in any hardware implementation. The approximation may result in accuracy degradation, but again most of ANN applications are error-tolerant, thus 100% accuracy is not a must. The hardware approximation target is to reduce the area and power while having the highest possible accuracy and that is the trade-off.

Following is a list of the most common hardware approximation techniques:

1. Precision scaling: it is the most used approximation technique. By nature, the data exists in the software layer in a floating-point representation, however, to use it at the hardware level it should be scaled down to a fixed point representation. If the precision of the fixed point numbers is reduced, the word length of all numbers is reduced as well. Accordingly, the sizes of multipliers, adders, and activation functions are reduced, the area, power, and accuracy are reduced as well.

2. Inaccurate arithmetic: The neural networks have a huge number of processing elements (neurons) that work in parallel, thus a large number of multipliers and accumulators are required. The multipliers are power-hungry hardware elements, as a result, these units should be optimized as much as possible to save area and power. An approximated version of multipliers and accumulators are then used in the ANNs, they do not perform all the intermediate computations needed to produce a result so they are called inaccurate arithmetic.

3. Neuron skipping: As the ANN may have more than one hidden layer and each layer has a large number of neurons, so it is a good approach to select the neurons that have the least contribution to the output of the network and skip them from the computation or approximate their hardware units. This approach reduces the energy used by the network.

4. Computation skipping: In the applications that use the ANNs in image processing, the images may have many zeros as the row data. The zero-valued inputs do not contribute or change the output of the accumulator. Thus, when the hardware detects a zero input, it skips the computation for that input. This approach reduces the activity factor on the hardware units and reduces the overall dynamic power consumption.

5. Approximate activation function: There are two commonly used activation functions RELU and Sigmoid. Sigmoid has exponential in its formula, and this can not be implemented exactly because it consumes area and power. Therefore, an approximation to sigmoid function is used and the most common approximation is Piece-Wise Linear (PWL) that produces a response close to the sigmoid's exact response, thus saves area and power. RELU has a problem that it passes any value greater than zero as it is unlike sigmoid that limits its output between 0 and 1. At the hardware level and as a fixed point number representation is used, RELU should have a limit for its output, thus a truncated version of its output is used.

2.3 Literature Review for Approximate Hardware Implementations of Artificial Neural Network

A literature review is carried out to find the techniques that are used in the hardware design of artificial neural networks and to find how the approximate computing is adopted in the hardware implementation. In the following paragraphs a study of the approximate implementations of the ANNs is presented.

Venkataramani et al. [5] in their work propose a new approach to design energy-efficient hardware implementations of large-scale neural networks (NNs) using approximate computing. They propose to approximate a given neural network selectively by calculating how each neuron affects the overall accuracy of the network, and then approximate the neurons that have the least impact on the accuracy. After approximating the selective neurons they retrain the network to improve the efficiency after approximation. They also propose a quality-configurable neuromorphic processing engine (qcNPE) that executes the computations with dynamically configurable accuracy.

The proposed AxNN (Approximate Neural Network) depends on three main design steps:

1. Resilience characterization, in which the least contributors to the output quality are identified using the back-propagation algorithm. This is a challenging step since it is needed to identify the neurons that can be approximated and will not affect the output quality and the neurons that are highly sensitive and can not be approximated. This is achieved using the back-propagation algorithm, since it defines the amount of the error at the output of each neuron, so it can measure the contribution of each one.
2. Neurons approximations, in which the error-resilient neurons are approximated and the sensitive neurons are not approximated. The neuron approximation is done by using inaccurate hardware components such as precision scaling and piece-wise linear approximation of the activation function.
3. Incremental retraining, in which the approximated network is retrained to minimize the quality loss by minimizing the errors. The retraining process in an iterative loop between approximation and retraining to get the best approximation with the least error as shown in Figure 2.6.

A quality configurable Neuromorphic Processing Engine (qcNPE) is introduced as a platform to execute AxNN. It contains hardware elements that are dynamically configurable to execute different neuron approximations. It has two processing elements: a 2D array of neural compute units (NCUs) and a 1D array of activation function units (AFUs). The NCU is mainly used to calculate the weighted sum of the neuron. It takes its inputs from a FIFO (First In First Out) memory element, the inputs are streamed to the neurons along the rows and the weights along the columns. The NCU is designed with a control register to control the precision of its units. In AFU, the activation function is performed on the weighted sum that is output from the NCU. The hardware architecture of the qcNPE is shown in Figure 2.7.

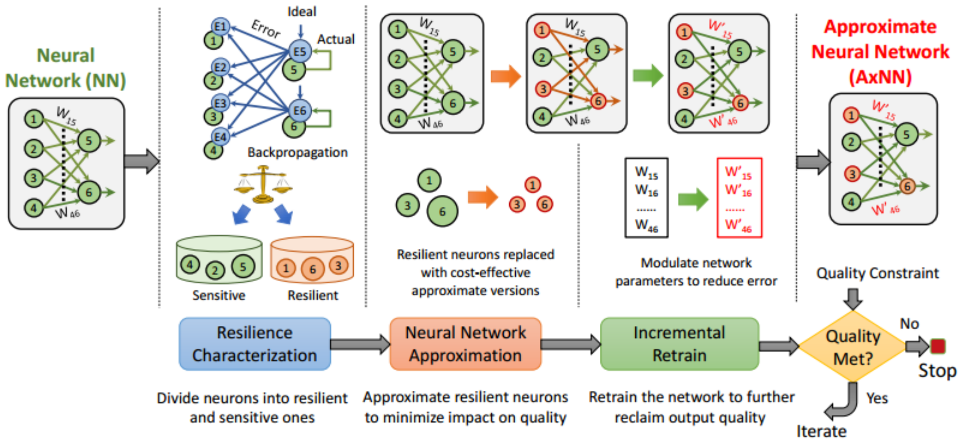

Neural Network (NN)
Ideal
Error
Actual
Backpropagation
Sensitive
Resilient
Resilient neurons replaced with cost-effective approximate versions
Modulate network parameters to reduce error
Approximate Neural Network (AxNN)
Quality Constraint
Resilience Characterization
Neural Network Approximation
Incremental Retrain
Quality Met?
No
Stop
Yes
Iterate
Divide neurons into resilient and sensitive ones
Approximate resilient neurons to minimize impact on quality
Retrain the network to further reclaim output quality

Figure 2.6: Design steps of Approximate Neural Network [5]

Zhang et al. [6] proposed ApproxANN that considers approximation for both computation and memory accesses. This is done by the assessment of how each neuron is critical to the output quality and energy consumption. They also proposed a theoretical neuron criticality analysis that can be used with any network topology.

To judge the criticality of a neuron, it is observed if a small jitter in the neuron's computation produces a large difference in output quality, then this neuron is critical and its approximation should be done consciously. If a small jitter in the neuron's computation produces a small difference in output quality, then the neuron is considered error-tolerant. To determine each neuron's criticality, intuitively, a random error is injected on each neuron input and its influence on the final output is recorded. The target is to minimize the error between the target output and the expected output. After identifying the error-resilient neurons, the neurons are ranked in the network for further computations.

Each neuron computes the product of the inputs by weights, then passes the result to the activation function. In their implementation, each processing element is composed of an arithmetic unit and a local memory for weights. The local memory is used to read the corresponding weights from off-chip memory.

The implemented approximate design choices are memory access skipping, precision scaling, and approximate arithmetic blocks, especially, approximate multiplier. Memory access skipping means that a specific reading from the off-chip memory to the weights matrix is skipped. The assessment of the skipped weights' reading is done in the analysis step. Precision scaling means that the word length is reduced by truncating some of the least significant bits, reducing the computation quality and saves energy as well.

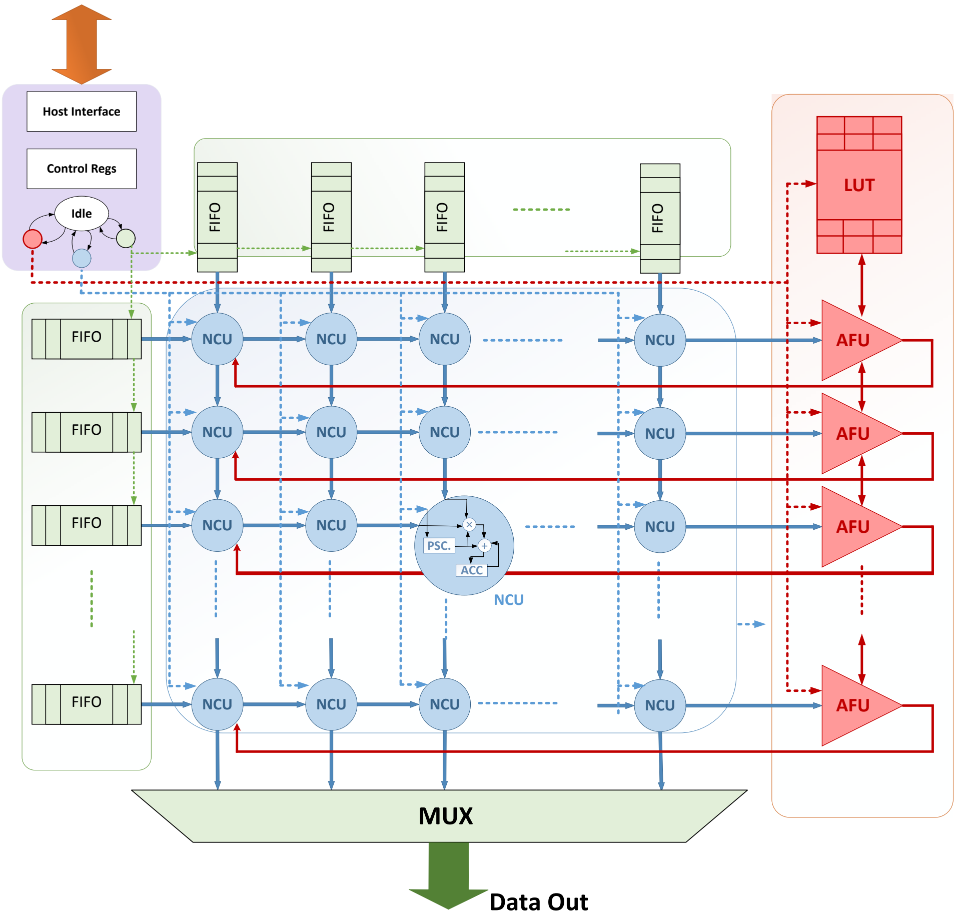


Figure 2.7: Hardware architecture of the quality control neuromorphic processing engine [5]

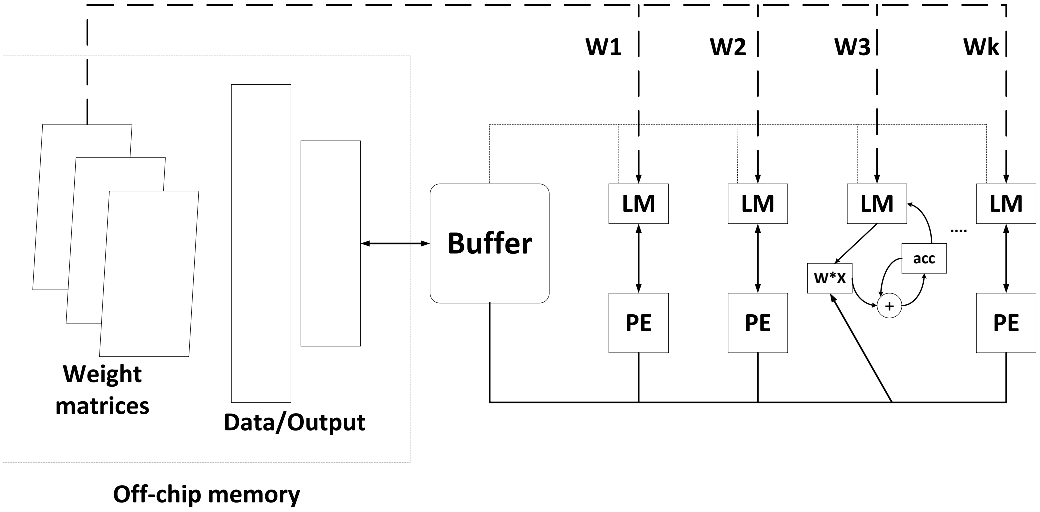


Figure 2.8: Hardware architecture of processing element and memory [6]

Table 2.1: Multiplication operation decomposition example

Weights	Decomposition of product
W_1 = 011010012(10510)	$W_1 \times$ I = 2^5.(0011).I +2^0.(1001).I
W_2 = 010000102(6610)	$W_2 \times$ I = 2^6.(0001).I + 2^1 .(0001).I

In Kung et al. [7], the digital feed-forward neural network is approximated by using precision scaling and/or approximate multipliers. First, the approach decides a set of approximate synapses that have the least impact on the output quality using a greedy algorithm. The output sensitivity is identified during the training phase by knowing how much error is produced from small perturbation at each synaptic weights. The selected synapses are approximated by using precision scaling and/or approximate multipliers. They achieved a power saving of about 53% by comparing the accurate processing element that uses an accurate multiplier and the other approximated one.

Sarwar et al. [8] proposed Alphabet Set Multiplier (ASM), it replaces the conventional accurate multiplier in the neurons by a multiplier-less one and thus reduces the consumed energy. In a multiplication operation, the product is generated from lower-order multiples of the multiplier input 'I'. The decomposition is based on the multiplicand 'W' (weight). Sample decomposition of two multiplication operations W1×I and W2×I are shown in 2.1. If I, 3I, 5I, 7I, 9I, 11I, 13I, and 15I are available, the entire multiplication is reduced to a few shifts and add operations. A pre-computer bank is required to generate the alphabets. Then the multiplication is performed by generating the alphabets, selecting the appropriate alphabets, shifting them, then adding the shifted alphabets as shown in Figure 2.9. The number of alphabets is less than required to get an accurate result, thus it uses less energy. The architecture also reduces the area by sharing the alphabets between the multiplication units.

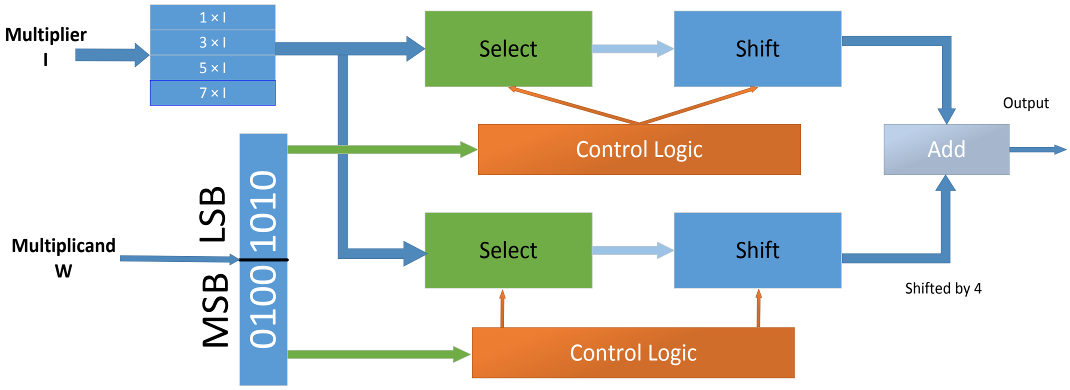


Figure 2.9: Operation of Alphabet Set Multiplier [8]

The efficiency of the ASM depends on the number of alphabets used to calculate the multiplications needed. If the bit sequences used for the decomposition of the multiplication operation contain 4bits, then an alphabet set of 8 alphabets {1,3,5,7,9,11,13,15} is enough to calculate any product accurately. For more improvement, the alphabet set is reduced and this results in multiplication approximation. To overcome this problem, a constrained training is performed so that it does not generate multiplications that need these unsupported alphabets. The network is retrained with these constraints for better accuracy. The design methodology and the retraining algorithm are shown in Figure 2.10.

Mrazek et al. [9] proposed a methodology for the design of a power-efficient Neural Network that has a uniform structure (all nodes are identical in all layers). The network at first is accurate, then there is an algorithm responsible for the approximation by identifying the accepted error. The parts of the network to be approximated are specified by error metrics such as the average error magnitude or maximum arithmetic error.

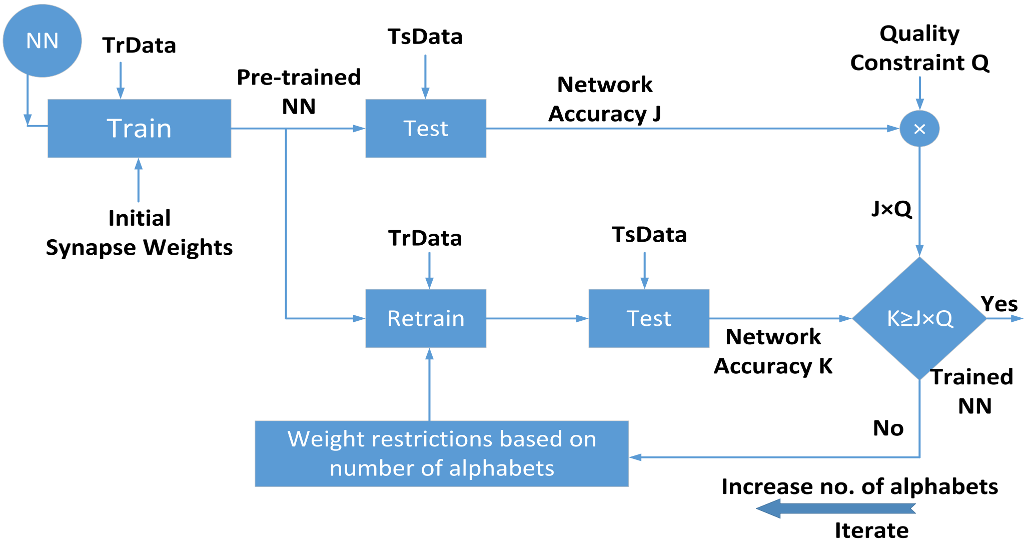


Figure 2.10: Retraining algorithm for neural network uses Alphabet Set Multiplier [8]

Kim et al. [10] proposed a network that is designed for on-chip training, but others considered off-chip training. This is done to study the different conditions such as bit precision during the training, the number of iterations of training, the number of layers of Multi-Layer Perceptron (MLP) on the effectiveness of the approximation, and the amount of power savings. It also presents a method for finding a near-optimum approximation for synapses to minimize the power consumption of the network while keeping the accuracy at a reasonable value. They used bit-precision and inexact multipliers for approximation and this is based on the selected synaptic weights during the learning phase.

The main idea is to select the synapses that have the least impact on the output results, not the neurons. Error sensitivities of weights are calculated during the training phase

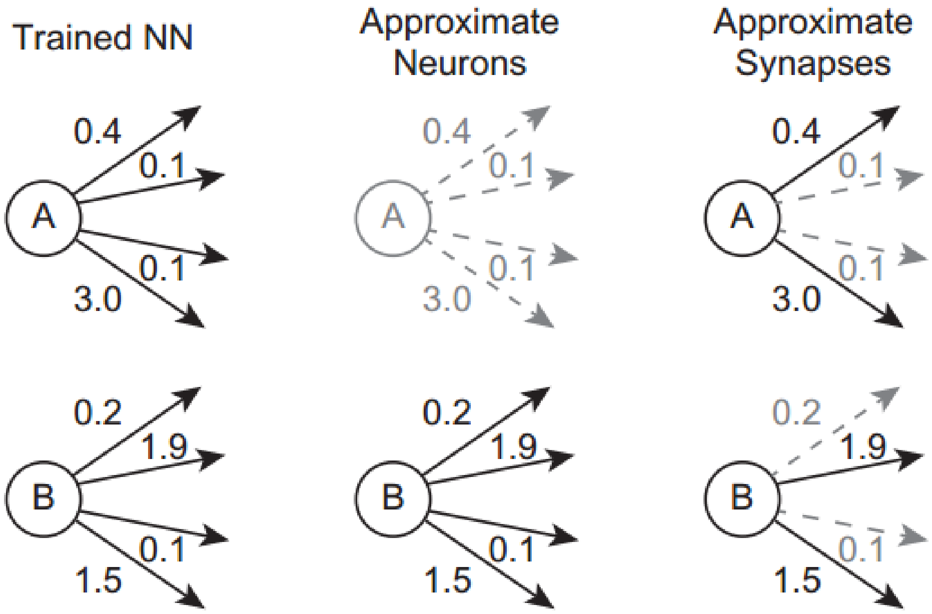


Figure 2.11: Differences between synapses approximation and neuron approximation [10]

using the back-propagation algorithm, following that the synapses that are less sensitive to errors are selected for approximation. As shown in Figure 2.11, if the whole neuron is selected for approximation it results in A approximation since it has an average error of 0.9 while it has a synapse of 3.0 error sensitivity. B is not approximated since it has an average error sensitivity of 0.93. In the case of approximating the synapses, the synapses with low error sensitivity are the only approximated ones, and the synapses with high error sensitivity are ignored.

Another approximation used is to choose the proper bit precision that minimizes the power consumption while maintaining the target accuracy. The algorithm sorts all synaptic weights of the neural network regardless of the layer. Therefore, each layer has different percentages of approximation and this is enhanced by considering different sensitivity for different layers.

In summary, a lot of research work has been proposed for the hardware implementation of artificial neural networks. The most common approximation methods are precision scaling and approximate arithmetic especially approximate multipliers since they are the greatest source of power consumption. All the work has been directed on how to find the best combination of these approximation methods. The main target of all mentioned works is to find the optimal approximation that would result in the highest possible accuracy with the least energy and power consumption.

2.4 Literature Review of the Approximate Multipliers

Multipliers are one of the main hardware units used in artificial neural network. They are the most power hungry hardware element. A study of the approximate multipliers used in the literature is held.

2.4.1 Broken Array Multiplier (BAM)

In [11], it discusses an approximate multiplier called broken array multiplier. As known, the multiplication operation in the traditional array multiplier is performed by calculating the partial product terms, then adding them to produce the final result. Figure 2.12 shows the basic structure of the 6x6 array multiplier which is used later to omit some cells to form the BAM. This array consists of 6x6 similar cells of carry-save adder in which a certain 2 bits are and-ed, the diagonal sum that comes from its above neighboring cell, and the input carry are all added together in that cell. The hardware structure of the CSA cell is an AND gate and Full Adder (FA) cell as shown in Figure2.13. Finally a vector merging adder is used to add the last 2 vectors and produces the output result.

It is obvious that the array multiplier is symmetric and all the cells that form it are symmetric too. The ability to reduce the CSA cells reduces the area of the multiplier and also reduces the size of the vector adder that is used in the last stage of the output generation. The power consumption is also reduced by the reduction of CSA cells, but the final result is imprecise. The Broken Array Multiplier is based on the reduction of the CSA cells, it breaks the array multiplier and omits some CSA cells using two break levels, Horizontal Break Level (HBL), and Vertical Break Level (VBL).

The number of omitted CSA cells depends on both the HBL and VBL. If the HBL = 0 and VBL = 0, then the BAM is the same as the array multiplier. HBL= 0 means that no CSA cells are omitted horizontally. VBL=0 means that no CSA cells are omitted vertically. As the HBL increases, this means that all the CSA above that level are omitted. Similarly, as the VBL increase, this means that all the CSA cells to the left of that level are omitted as shown in Figure 2.12. The output values of the horizontally and vertically omitted cells are assumed to be zero. Thus, these cells are replaced by zero value and so no need to consider them in the following calculations. As the HBL and VBL increase, the number of CSA omitted cells increases too, causing more area reduction and less optimum output result.

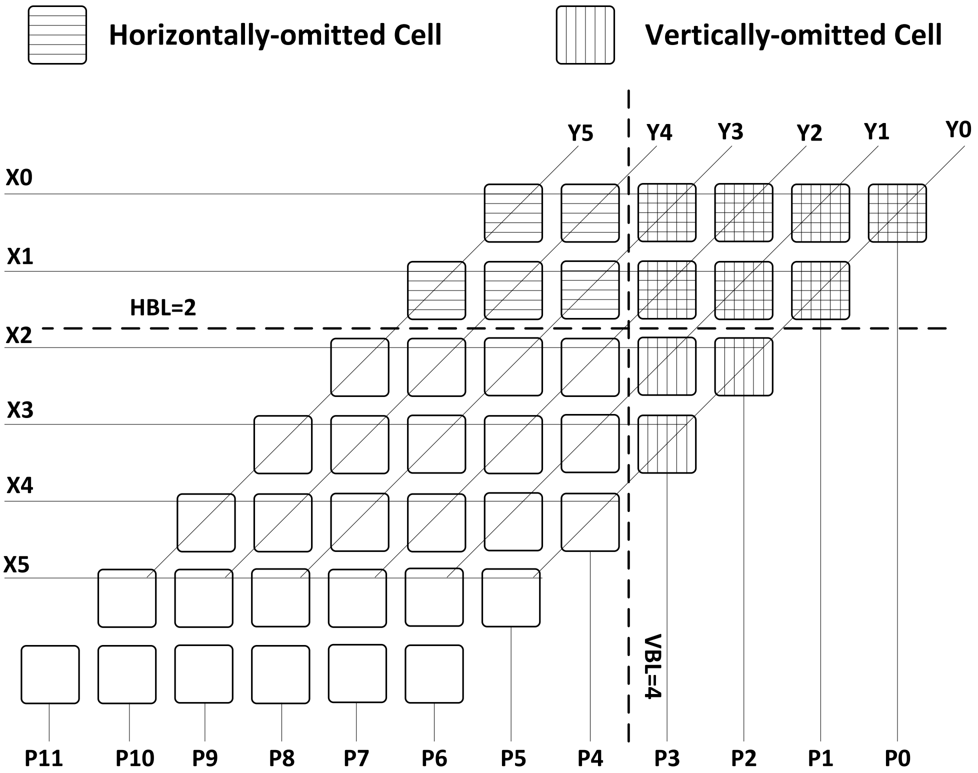


Figure 2.12: Hardware structure of the Broken Array Multiplier (BAM) [11]

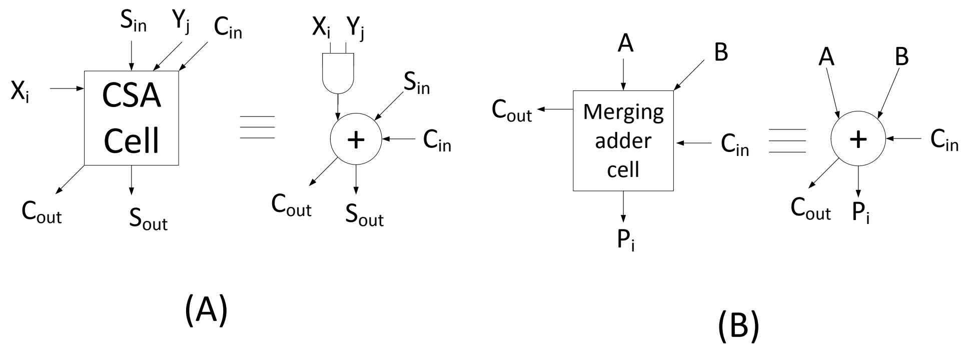


Figure 2.13: (A) Carry Save Adder cell used in the broken array multiplier , (B) Vector merging adder cell in the broken array multiplier

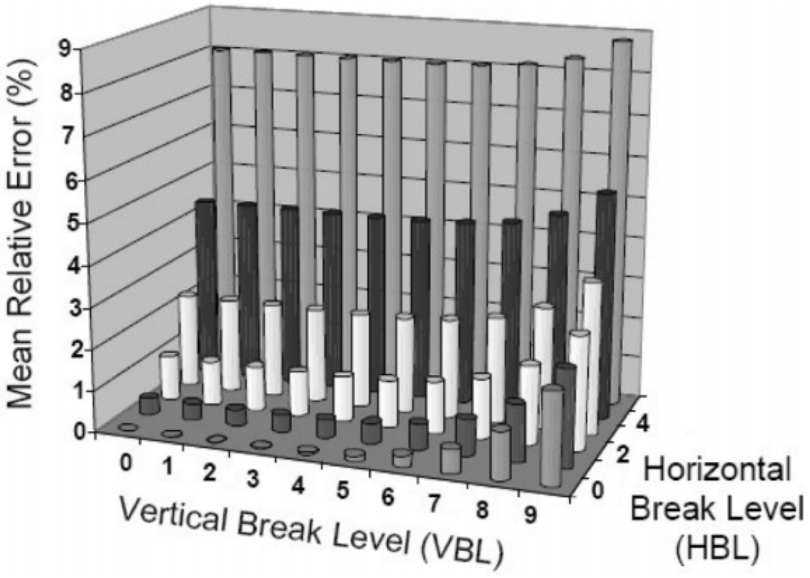


Figure 2.14: The effect of increasing horizontal break level and vertical break level on the mean error of the broken array multiplier output [11]

The differences in the results obtained using BAM and the accurate multiplier is discussed in [11]. The maximum and minimum differences between the BAM and the precise multiplier are calculated by the equations 2.4 and 2.5, respectively.

$$MAX_{BAM} = (2^{WL} - 1) * (\sum_{i=0}^{HBL-1} 2^i) + 2^{HBL} * (\sum_{i=0}^{VBL-HBL-1} (2^{VBL-HBL} - 2^i)) \tag{2.4}$$

$$MIN_{BAM} = 0 \tag{2.5}$$

The equations indicate that if the HBL = VBL = 0, maximum and minimum differences between BAM and the precise multiplier are also 0 and the BAM produces 100% accurate results with 0 error. Another metric to estimate the error caused by BAM is the mean error. It is calculated in terms of HBL, VBL, and WL of the multiplier and multiplicand. Equation 2.6calculate the mean error (ME) of BAM. As noticed, the ME is 0 if the HBL=VBL=0and the results equal the precise multiplier results. As HBL and VBL increase, the ME of the BAM increases as well with different rates. For a constant HBL, increasing the VBL does not affect the mean error of BAM greatly. However, with a constant VBL, increasing the HBL by one value increases the mean error of BAM significantly. This is because the HBL truncates values that affect the most significant bits of the result, but the VBL truncates values that affect the least significant bits of the result. Accordingly, the HBL has to increase consciously in order not to degrade the output quality in a significant way. Figure 2.14 shows the effect of increasing the HBL and VBL of the BAM mean error in (%).

Table 2.2: Map for the inaccurate 2x2 multiplier for all possible input combinations

B1B0

A1A0	00	01	10	11
00	000	000	000	000
01	000	001	010	011
10	000	010	100	110
11	000	011	110	111

$$ME_{BAM} = (\frac{2^{WL}-1}{4}) * (\sum_{i=0}^{HBL-1} 2^{i}) + 2^{VBL-2} * (\sum_{i=0}^{VBL-HBL-1} (1-2^{-(i+1)})) \tag{2.6}$$

2.4.2 Under-designed Multiplier (UM)

All techniques targeting to approximate the multiplier introduce an error in the result calculation and this error is used to optimize the hardware implementation of the multiplier. Most of the introduced errors manipulate the logical design of the multiplier to reduce the needed hardware elements. The under-designed multiplier introduced in [12] uses this concept to build an approximate multiplier. It designs a 2x2 under-designed multiplier block, then uses it as a building block for any larger size approximate multiplier.

To design the 2x2 inaccurate under-designed building block, the logic of the 2x2 multiplication is manipulated. As shown in table 2.2, the multiplication result of any 2x2 values has to output a result in 3 bits only. As known the multiplication result of 2x2 numbers has a maximum value presented in 4 bits, i.e., if $(11)_b$* is multiplied by $(11)_b$, the accurate result is $(1001)_b$ that is represented in 4 bits. In the under-designed multiplier, it has to be represented in 3 bits only. This causes $(1001)_b$ to be approximated to $(111)_b$as shown in table 2.2. Table 2.2 indicates that the only value approximated in the 2x2 multiplier unit is the result of maximum multiplication of $(11)_b$* $(11)_b$ and all other multiplication results are accurate. Consequently, the error occurs with a probability of $\frac{1}{16}$, one error output out of 16 output results. Such approximation causes the area of the 2x2 multiplier to be about half the area of the accurate version as shown in figures 2.15 and 2.16.

The next step in the implementation of an under-designed multiplier is to build larger multiplier using the 2x2 building blocks discussed earlier. Figure 2.17 shows the main idea of rearranging the 2x2 inaccurate multiplier blocks to form the larger 4x4 multiplier. A_L, X_L are the least significant 2 bits of the inputs, and A_H, X_H are the most significant 2 bits of the inputs. The procedure of multiplication is to multiply every 2 bits of the inputs using the 2x2 under-designed unit, shift them, and then add the shifted blocks to form the output. This concept can be extended to build any multiplier size out of the 2x2 multiplier unit. Another example of building a 16x16 multiplier is shown in Figure 2.18. First, the

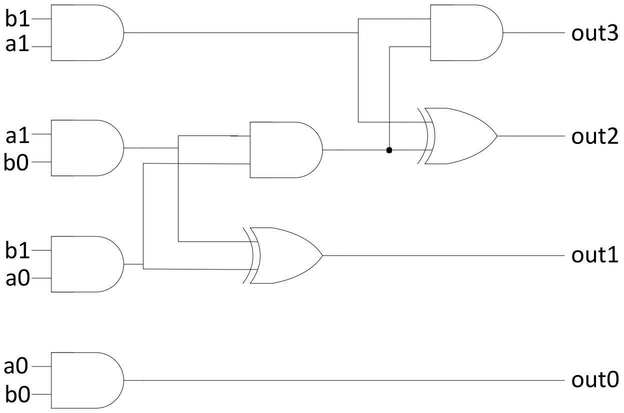


Figure 2.15: Hardware implementation of 2x2 accurate multiplier unit

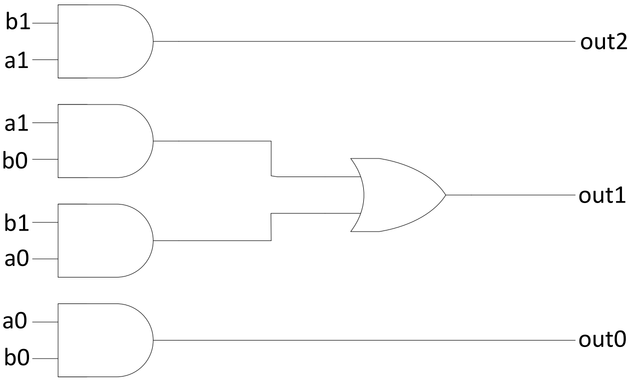


Figure 2.16: Hardware implementation of 2x2 under-designed multiplier unit

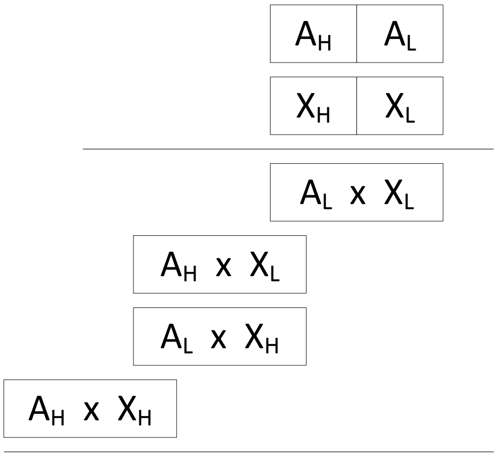


Figure 2.17: Building larger multipliers from smaller ones in the under-designed multiplier

blocks of 4x4 multiplier are built, then they are shifted and added to form the larger 8x8 multiplier. Finally, using the 8x8 block, shift and add them to build the output of 16x16 multiplier.

As discussed earlier, the probability of error in a 2x2 multiplier block is $\frac{1}{16}$, and it is constant for that multiplier size, and it is the same case for larger size multipliers. Each size has a fixed probability of error and a fixed error magnitude as illustrated in [12]. Since the multiplier is built from 2x2 blocks, whenever the block is located in a position that is sensitive to errors it can be replaced with an accurate block to enhance the quality of the result. The most significant bits are an example of the most sensitive locations that cause high errors, or in other words, the error in these bits reflects on the output accuracy significantly.

2.4.3 Error Tolerant Multiplier (ETM)

Error Tolerant Multiplier is another way of multiplier approximation that reduces the operations needed to perform the accurate multiplication operation. ETM is introduced in [13]. In the ETM, the input operands are divided into two parts: the non-multiplication part and the multiplication part. The multiplication part is the part of the input that is used to calculate the output as the accurate multiplier and it includes the most significant bits of the inputs. The non-multiplication part is the part of the input that its contribution to the output is approximated and it includes the least significant bits of the inputs. The length of the multiplication part and the non-multiplication part does not have to be the same.

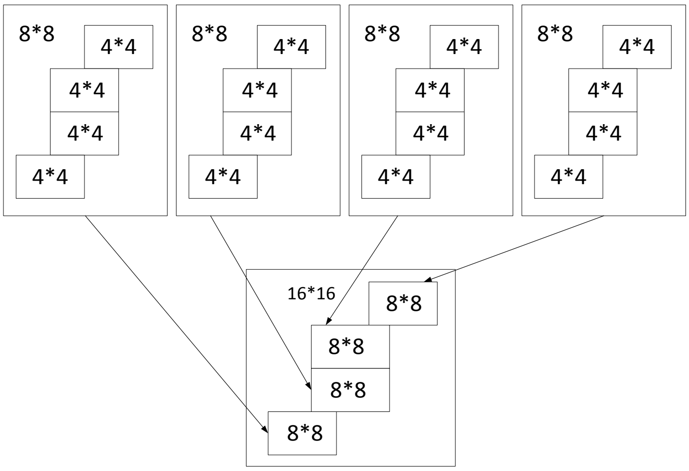


Figure 2.18: Example of generating larger multipliers from smaller multiplier units in the under-designed multiplier method

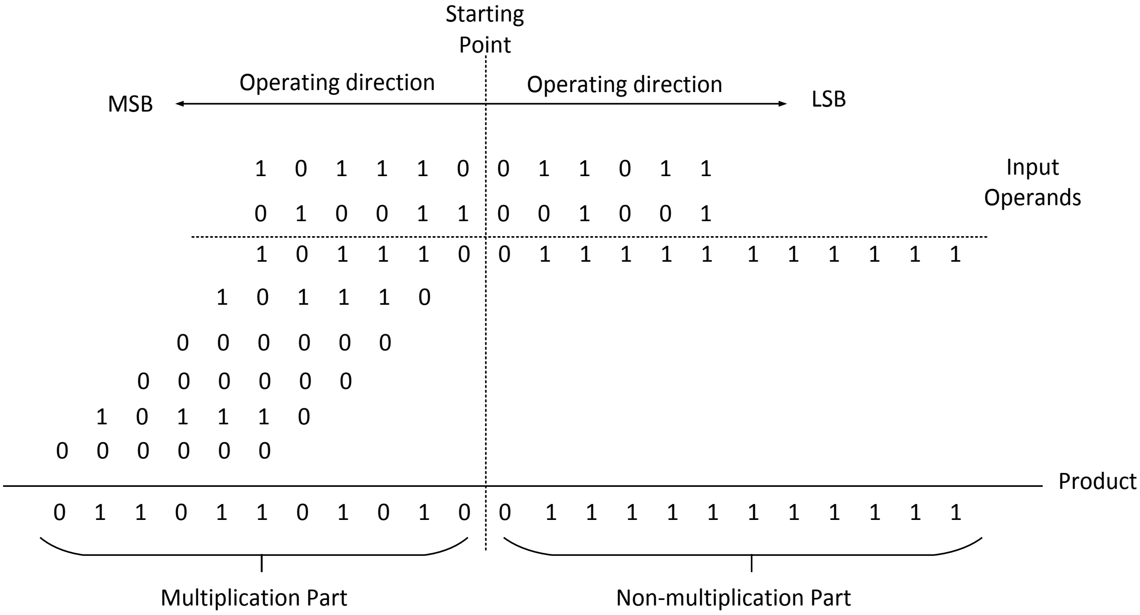


Figure 2.19: Algorithm of the Error Tolerant Multiplier [13]

To illustrate the multiplication procedure of the ETM, refer to Figure2.19 as an example. In this example, both the multiplication and non-multiplication parts are of the same size, i.e., the multiplication part of the input is 6 bits and the non-multiplication part is 6 bits as well. First, begin at the splitting point, in the non-multiplication part, move from the left to right while searching for a logic “1” in one of the inputs or both and when the first one is found, all the output bits starting from that position are set to one. If both input bits are 0, then the corresponding output is 0 as well. In this case, no need to calculate the partial products for the non-multiplication part, thus, the hardware needed is reduced. For the multiplication part, its share in the output is calculated normally as in the traditional multiplier by calculating the partial products, shifting and adding them. The multiplication part is chosen to be the most significant bits of the inputs so that it does not reduce the output accuracy as it has a higher weight than the non-multiplication part.

The quality of the output results of the ETM is assessed using two defined metrics namely: Overall-Error(OE), and Accuracy(ACC). The overall-error(OE) is calculated as in 2.7.

$$OE = |R_c - R_e| \tag{2.7}$$

where $R_c$is the correct result and $R_e$is the result produced from the ETM multiplier. The Accuracy (ACC) is calculated as follows in 2.8.

$$ACC = (1 - OE/R_c) * 100\% \tag{2.8}$$

From the previous equations, the accuracy of the Error tolerant multiplier depends greatly on the input pattern and the chosen size of the non-multiplication part. The trade-off here is between choosing the appropriate size for the multiplication part and the non-multiplication part and the overall input size. As the non-multiplication part increases, the area and power consumption are reduced, but the accuracy is degraded as well.

2.4.4 Truncated Multiplier

The truncated multiplier is another approximation approach to enhance the hardware implementation of the multiplier at the cost of some accuracy reduction. The conventional multipliers calculate nxn multiplication operation and the exact output size is 2n bits. In some circuits, the exact result is not a must and the size of the output result needn’t be the full size, i.e., the output result is rounded to n bits instead of the 2n bits. This reduces the size of all the hardware units that follow the multiplier in the data-path. The truncated multiplier is the approach that rounds the output result to a smaller number of bits than the number of bits needed to hold the full accurate result as discussed in [14]. That truncation takes several forms and can be used to truncate any number of bits. As shown in Figure 2.20, the example assumes that the inputs x and y are fractional numbers. The partial product of the inputs is calculated as in the normal multiplier by anding the bits of the two inputs. Then the partial product can be seen as two sets, the least significant part (LSP) and the most significant part (MSP). The LSP includes the least significant columns of

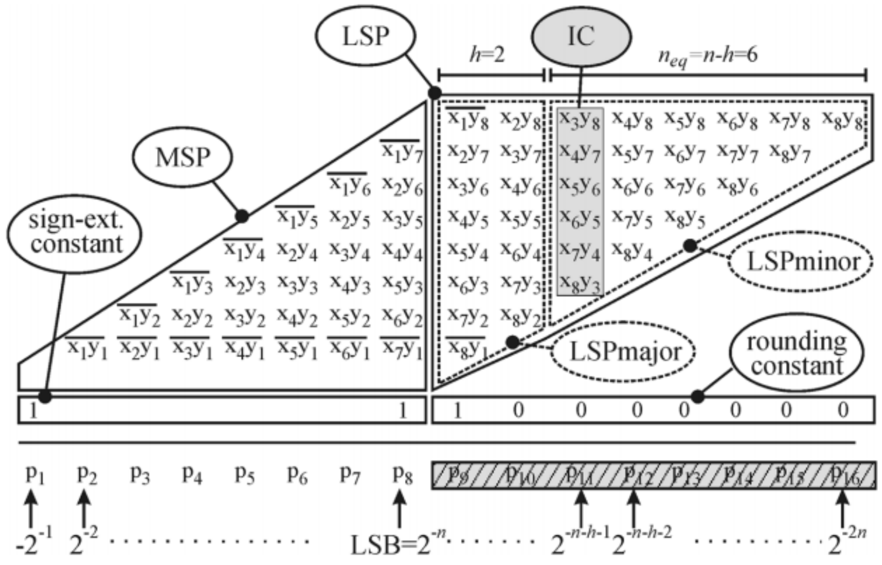

LSP
IC
MSP
sign-ext. constant
h=2
n_{eq}=n-h=6
LSPminor
LSPmajor
rounding constant
1 1 1 0 0 0 0 0 0 0
p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8 p_9 p_{10} p_{11} p_{12} p_{13} p_{14} p_{15} p_{16}
-2^{-1} 2^{-2} LSB=2^{-n} 2^{-n-h-1} 2^{-n-h-2} 2^{-2n}

Figure 2.20: Truncated signed multiplier partial product matrix with n=8 and h=2 [14]

the partial products that are the result of the LSBs of the inputs. The MSP that includes the most significant columns of the partial products that are the result of the MSBs of the inputs. The LSP itself is divided into two parts, the LSPminor and the LSPmajor and these columns are defined by a design parameter h that ranges from 0 to n. Since the least significant bits of the result will be truncated from the result, no need to calculate the partial products that produce them. To enhance the output quality of the truncated multiplier, a correction term is added to the result to compensate for the truncated partial products, thus, increasing the multiplier accuracy.

The input correction (IC) term in the left-most column of the LSPminor part of the partial products is introduced for the compensation of the truncated partial product terms. The whole partial products of the LSPminor are truncated and compensated for by $f(IC)$. $f(IC)$ is a function that uses the IC column to calculate a compensation term for the LSPminor. Generally, $f(IC)$ is much simpler than calculating the sum of partial product.

2.5 Introduction to Spiking Neural Networks and neuron models approximation

The Spiking neural network (SNN) is the next generation of artificial neural networks. Spiking neural networks use biological and realistic models to mimic the human brain processing and thus perform the computations in a biological manner. There are some of the spiking and bursting patterns that are observed in the biological neurons in response to a certain stimulus. Not all neurons fire the same patterns, but each neuron fires a certain set of patterns [15]. Some of the well known patterns are described as follows:

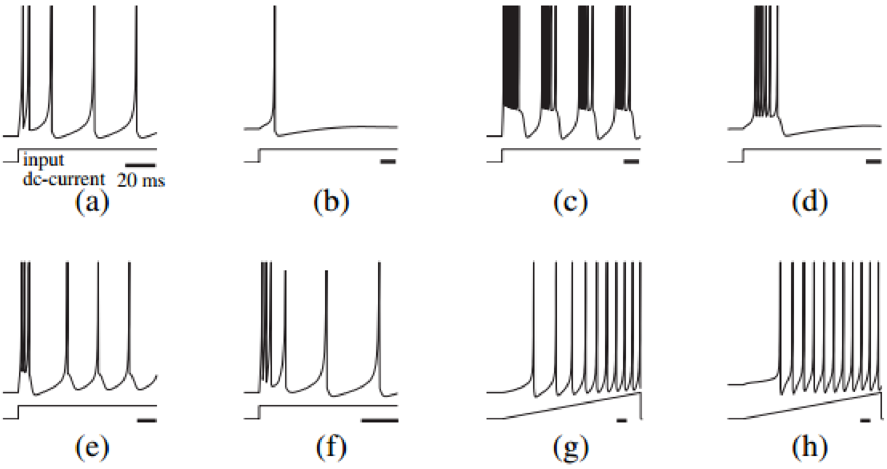


Figure 2.21: Main spiking patterns of the biological neuron [15]

1. Tonic spikes (Regular spikes), Figure 2.21.a: They are a train of spikes that fire in response to continuous input current, and the frequency of the spikes increases with the input stimulus.
2. Phasic spiking, Figure 2.21.b: It is the behavior of generating an impulse at the beginning of the stimulus, then resting at low potential until the end of the stimulus.
3. Tonic bursting, Figure 2.21.c and phasic bursting, Figure 2.21.d: They are similar to tonic and phasic spiking in the repetitive behavior, the main difference is that they spike in bursts as a response to the stimulus. The tonic bursting generates a set of bursts with a frequency proportional to the input stimulus and the phasic bursting fire one set of bursts at the start of the stimulus. There is also a mixed mode where the spikes are bursts at first, then they switch to a single spike each time, Figure 2.21.e.
4. Spike frequency adaption: the neurons can reduce the spiking frequency or increase it in response to the input stimulus as shown in Figure 2.21.f,g,h.

As discussed previously, the neuron model is implemented in different ways that define how much it is close to the biological neurons. The Biologically plausible models are the models that implement the biological neurons with their physical parameters and behaviors. The biologically inspired neurons are the ones that mimic the biological behavior without the need to model the physical characteristics of the neurons. In the following subsections, the most common neuron models are discussed.

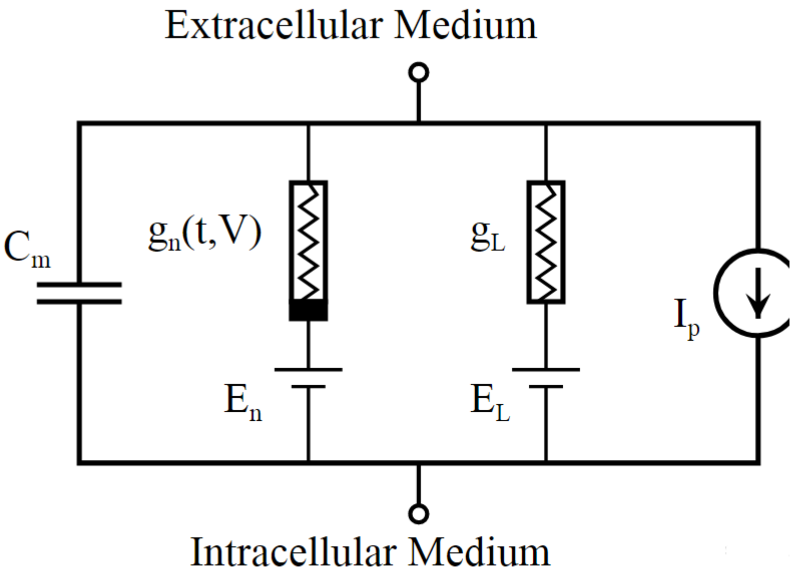


Figure 2.22: Electrical circuit model of Hodgkin-Huxley biologically plausible neuron [16]

2.5.1 Biologically plausible models

2.5.1.1 Hodgkin-Huxley model

Hodgkin-Huxley model is the most complex neuron model and the closest one to the biological neurons. It considers each physical characteristic in the neuron cell as an electrical element in the model equation. It emulates the sodium and potassium ion currents and their effects on the spiking patterns generated by the neuron. The model could obtain a very realistic biological neuron behavior that can regenerate all the spiking and bursting patterns and also the frequency adaption phenomenon of the biological neurons [16].

The electrical model of the Hodgkin-Huxley neuron is shown in figure 2.22. The capacitance $C_m$represents the lipid bilayer. Each voltage-gated ion channel is represented as time and voltage-dependent conductance g_n. Leak channel is represented as constant conductance g_l. $E_n$represents the electrochemical gradients that cause the ions flow. And $I_p$represents the ion pumps. Finally, the membrane voltage is V_m.

Through a series of experiments, Hodgkin-Huxley developed a mathematical model of four differential equations to model the neuron cell biological behavior. The Hodgkin-Huxley equations are described in the following equations:

$$I = C_m \frac{dV_m}{dt} + \bar{g_k} n^4 (V_m - V_k) + \bar{g_{Na}} m^3 h (V_m - V_{Na}) + \bar{g_l}(V_m - V_l) \tag{2.9}$$

$$\frac{dn}{dt} = \alpha_n(V_m)(1-n) - \beta_n(V_m) n \tag{2.10}$$

$$\frac{dm}{dt} = \alpha_m(V_m)(1-m) - \beta_m(V_m)m \tag{2.11}$$

$$\frac{dh}{dt} = \alpha_h(V_m)(1-h) - \beta_h(V_m)h \tag{2.12}$$

Where g_k is the potassium conductance, g_{Na} is the sodium conductance per unit area, V_k is the potassium reverse potential and V_{Na} is the sodium reverse potential. n, m, h are quantities between 0 and 1 and relate to the potassium channel activation, sodium channel activation, and sodium channel inactivation. α_i,$\beta_i$are rate constants for the i-th ion channel. As shown in the previous equations, Hodgkin-Huxley is the most complex neuron model that includes all the physical characteristics of the biological neuron. However, it is the most accurate biologically plausible neuron model.

2.5.1.2 Morris Lecar model

It is a reduced model from the Hodgkin-Huxley that is discussed in the previous section. It reduces the Hodgkin-Huxley model to two-dimensional nonlinear equations, however, its parameters still have a physical meaning. The activation of calcium ions is assumed to be very fast, that it can be modeled as instantaneous and the model is then reduced to two-dimensional equations. It has the same modeling concept as Hodgkin-Huxley and it represents the physical parameters of the neuron cells as electrical elements. Following are the equation of the Morris Lecar model [17]:

$$C\frac{dV}{dt} = -I_{ion}(V, w) + I_{app} \tag{2.13}$$

$$\frac{dw}{dt} = \phi[w_\infty(V) - w]/\tau_w(V) \tag{2.14}$$

$$I_{ion} = \bar{g_{Ca}} m_\infty(V)(V - \bar{V_{Ca}}) + \bar{g_K}(V - \bar{V_K}) + \bar{g_L}(V - \bar{V_L}) \tag{2.15}$$

$$m_\infty(V) = 0.5(1 + tanh((V - V_1)/V_2)) \tag{2.16}$$

$$w_\infty(V) = 0.5(1 + tanh((V - V_3)/V_4)) \tag{2.17}$$

$$\tau_w(V) = 1/cosh((V - V_3)/(2V_4)) \tag{2.18}$$

w is the fraction of open potassium channels, it provides the slow voltage dependent feedback required for excitability, $\bar{g_{Ca}} and \bar{g_K}$are the maximum conductances of calcium and potassium ions, respectively, and $\bar{V_{Ca}} and \bar{V_K}$ are the reversal potentials for calcium and potassium channels, respectively. $m_\infty and w_\infty$ are the activation functions, $\tau_w$is the voltage dependent activation time constant. $\phi$is a temperature factor.

As seen in the previous two models, the main concern in the biologically plausible models is to maintain the same dynamics, physical characteristics, and exact behaviors of the neuron cells, which complicates the model equations and makes the hardware implementations of such models difficult. The biologically inspired models evolved to overcome the complexity issues of the biologically plausible models. They mimic the neuron processing capabilities without modeling the physical parameters exactly.

2.5.1.3 Fitzhugh-Nagumo model

This is a very simplified version of Hodgkin-Huxley and it still has some physical meaning in its parameters, but it does not consider all physical parameters. It simplifies the modeling of the sodium and potassium ions. The sodium ion current is fast and is strongly dependent on the membrane voltage, as a result, it is modeled as a time-independent nonlinear conductance. The potassium ion current is slow and it does not depend on the membrane potential, thus it is modeled by a linear resistance in series with an inductor L and a voltage source $V_0$to represent the resting potential of the membrane. The mathematical representation of the Fitzhugh-Nagumo model is shown in the following equations:

$$C_m\frac{dV_m}{dt} = I - i_K - f_{Na}(V_m) \tag{2.19}$$

$$L\frac{di_K}{dt} = V_m + V_o - Ri_K \tag{2.20}$$

It is noted from the previous discussion the main trade-off is between the simple equations in terms of the parameters and variables and the regeneration of the exact dynamical behavior of the biological neurons. The simpler the model, the simpler the hardware implementation.

2.5.2 Integrate and fire models

Integrate and fire models are a simpler set of neuron models that ranges in their complexity from the simplest model (leaky integrate and fire) to more complex models that approach the Izhikevich model. These neuron models are less biologically realistic, however, they produce a reasonable set of spiking behaviors suitable for simple spiking neural networks. The following sections describe the most common integrate and fire neuron models with more details with an implementation example of each model.

2.5.2.1 Leaky integrate and fire

Leaky integrate and fire model is a very simple neuron model that describes the neuron dynamics. Although it is computationally simple, it is not capable of producing all spiking dynamics. The model is described as follows: when an input current $I(t)$ is injected to the neuron cell, it charges the cell membrane that can be modeled as a capacitor C. Some charges leak through the cell membrane and this is modeled as a leak resistance R. Thus,

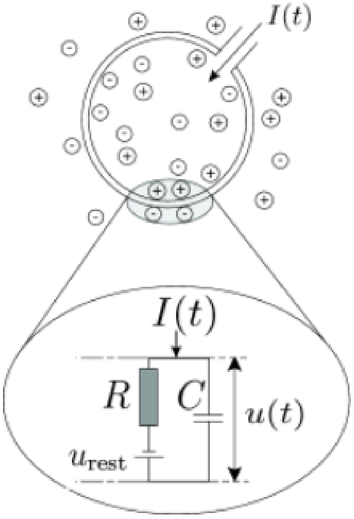


Figure 2.23: Leaky integrate and fire circuit model [18]

the circuit model that represents the leaky integrate and fire model is a capacitor C in parallel to a resistance R and an input current I(t) as shown in Figure 2.23[18].

The membrane voltage at the normal conditions is at its resting value u_{rest}, then the voltage across the membrane is calculated as follows:

$$I(t) = I_R + I_C \tag{2.21}$$

$$I(t) = \frac{u(t) - u_{rest}}{R} + C\frac{du}{dt} \tag{2.22}$$

$$\tau_m \frac{du}{dt} = -[u(t) - u_{rest}] + RI(t) \tag{2.23}$$

Where $\tau_m = RC$ is the time constant of the leaky integrator. The leaky integrate and fire model is expanded further to a more complex model which is the non-linear leaky integrate and fire. An example of the non-linear leaky integrate and fire model is the quadratic integrate and fire model that is discussed in the following subsection.

2.5.2.2 Quadratic Integrate and fire model

Quadratic integrate and fire is a more complex version of the leaky integrate and fire neuron model. It regenerates some of the most important spiking patterns. It has many analog and digital hardware implementations in literature [19]. The equations of the model are as follows:

$$\tau \frac{du}{dt} = a_0(u - u_{rest})(u - u_c) + RI \tag{2.24}$$

where $a_0 > 0 and u_c > u_{rest}$, the parameter $u_c$is the critical voltage for spike initiation by a short current stimulus. The quadratic model is simpler than the exponential integrate and fire model that is investigated in the following subsection in terms of the hardware implementation, however, it appears that the experimental data of the spikes profiles is much better in the case of the exponential model than the quadratic model [18].

2.5.2.3 Exponential integrate and fire model

Exponential integrate and fire model is an enhancement of the quadratic integrate and fire model that added a term to fire with a low stimulus current. In this model, the same basic terms of the integrate and fire model exists. The added term in that equation is the exponential term as in the following equation:

$$\tau \frac{du}{dt} = -(u - u_{rest}) + \triangle_T \, exp(\frac{u - v_{th}}{\triangle_T}) + RI \tag{2.25}$$

$\triangle_T$ is called the sharpness parameter and the parameter $v_{th}$is the threshold voltage. Again, when the voltage reaches the threshold, the neuron fires a spike and the membrane potential resets. The exponential integrate and fire model results in more exact spikes firing times relative to that obtained experimentally. The main drawback of that model is the exponential term that is hardware costly.

2.5.3 Biologically Inspired models

The biologically inspired models are a wide set of neuron models that are interested in mimicking the processing capabilities of the neuron cells such as their firing and excitation patterns rather than emulating the physical parameters. They are very much simpler than biologically plausible models as their equations are simpler and have less set of parameters that have no physical meaning. Thus, they are more hardware friendly, as a result, several implementations of such models exist in both analog and digital domains. In the following subsections, some of the common biologically inspired neuron models are discussed, their model equations and some of their hardware implementations are investigated as well.

2.5.3.1 Hindmarsh-Rose model

Hindmarsh-Rose model is one of the common biologically inspired neuron models. It is one of the models that are interested in regenerating the spiking patterns and spiking timing of the biological neurons on the cost of lower biological accuracy. It models the temporal behaviors and the dynamics of the neuron cells. The Hindmarsh-Rose model is described in the form of three coupled differential equations that describe the behavior of the neuron potential as in the following equations:

$$\begin{cases} \frac{dx}{dt} = y - f(x) - z + I_{app} \\ \frac{dy}{dt} = g(x) - y \\ \frac{dz}{dt} = r(h(x) - z) \end{cases} \tag{2.26}$$

where,

$$\begin{cases} f(x0 = x^3 - 3x^2 \\ g(x) = 1 - 5x^2 \\ h(x) = 4(x + \frac{8}{5}) \end{cases} \tag{2.27}$$

x is the membrane potential, y is the recovery current and z is the adaption current. $I_{app}$is the applied current to the neuron, r controls the spiking frequency and also affects the number of spikes per burst in case of bursting. As shown in [20], the Hindmarsh-Rose models two dynamic behaviors: the spiking and the bursting. In the spiking mode, the bursting variable is set to zero, the spiking frequency is then dependent on the stimulus current. As the stimulus increases, the spiking frequency increases. Due to the simplicity of the model equations and its ability to regenerate the spiking patterns at a reasonable accuracy, many works implemented it in hardware. Its equations can be rewritten in a discrete form and then implemented as a piece-wise linear implementation as done in [20]. It can be implemented in an analog form by CMOS implementation as in [21].

2.5.3.2 Mihalas-Niebur model

It is a generalized form of the leaky integrate and fire model and it has some parameters that have a biological meaning, the following equations are the Mihalas-Niebur model equations.

$$I_j^{`}(t) = -k_j I_j(t), \quad j = 1, ...,N \tag{2.28}$$

$$V_m^{`}(t) = \frac{1}{C}(I_{ext} + \sum_j I_j(t) - G(V_m(t) - E_L)) \tag{2.29}$$

$$\Theta^{`}(t) = a(V_m(t) - E_L) - b(\Theta(t) - \Theta) \tag{2.30}$$

where I_j are the internal currents, $I_{ext}$is the external input current to the neuron, V_m is the membrane potential, C is the membrane capacitance, Θ is the adaptive threshold. The adaptive threshold is updated continuously not only at a spike.

A spike is generated when $V_m \geq \Theta$ and following the spike, the variables are updated again to their reset conditions[15]. This model is also implemented in hardware due to its simplicity. As an example of implementation, [15] implements Mihalas-Niebur by switched capacitor circuits and proves that it is able to generate the main spiking and bursting patterns and the neurons dynamic behaviors as well.

2.5.3.3 The Quartic model

The Quartic model is very common in the biologically inspired neuron models. It is relatively simple from the mathematical point of view while having the ability to reproduce the dynamical neuron behaviors. The dynamics of that model are defined by two coupled differential equations as listed in the following equations:

$$\begin{cases} \dot{v} = v^4 + 2av - w + I \\ \dot{w} = a(bv - w) \end{cases} \tag{2.31}$$

$$If \;\; v(t^-) > \alpha \;\; then \; \begin{cases} v(t) = v_r \\ w(t) = w(t^-) + d \end{cases} \tag{2.32}$$

I is the input stimulus, a, b are the parameters that control the dynamical behavior of the neuron model. $\alpha$is the threshold after which the neuron fires a spike. v_r is the reset value of the membrane potential after a spike. So, $v_r and d$ are the parameters that control the reset behavior of the neuron. The quartic model proves its ability to reproduce the dynamical activities of the neurons the same way as the Izhikevich model and exponential integrate and fire model.

From a hardware implementation point of view, the quartic model is not hardware friendly because of the quadratic term in the differential equations, however, it has many hardware implementations as in [22].

2.5.3.4 Izhikevich neuron model

The Izhikevich neuron model is one of the well-known neuron models that gained interest in the area of spiking neural networks due to its simplicity and the ability to reproduce much dynamical behavior in a timely manner. It is a simplified version of the Hodgkin-Huxley that regenerates spiking and bursting behaviors of the known types of cortical neurons. It is a mathematical model consisting of two-dimensional differential equations as follows:

$$\dot{v} = 0.04v^2 + 5v + 140 - u + I \tag{2.33}$$

$$\dot{u} = a(bv - u) \tag{2.34}$$

$$if\; v \geq 30mV,\; then \begin{cases} v \;\leftarrow c \\ u \;\leftarrow u + d \end{cases} \tag{2.35}$$

It is obvious from the equations that the Izhikevich neuron model is simple enough to be implemented in the hardware, however, it is a very good model in regenerating the neuron dynamics. In this work, the Izhikevich model is discussed in deep details and

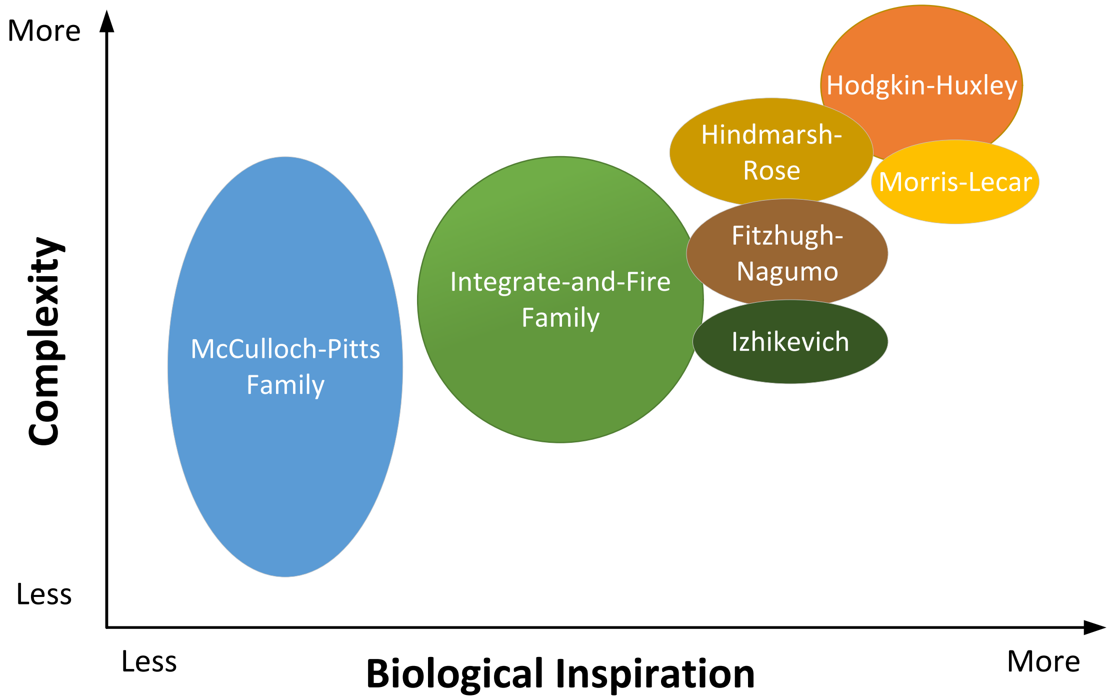


Figure 2.24: Trade off between the complexity of the neuron model vs its biological characteristics [3]

it is also implemented using approximate computing. The critical point in the hardware implementation of that model is the square operation in the potential equation since the multiplication is the power and area consumer in hardware designs. Thus, the approximate multipliers are adopted in the implementation and their effect on the spiking patterns and model accuracy is studied in the next chapters.

As illustrated in the previous list of neuron models, there is a wide variety of models that reproduce the biological dynamics. Some of the models are realistic and take the physical parameters and experimental observations into considerations. Others are interested in mimicking the neuronal behavior without the need to have the real parameters, they are just mathematical models. The trade-off is always between the model complexity and the biological reality. The most complex model is the most real biological model. Figure 2.24 shows the different models, trading off between complexity and biological inspiration.

Chapter 3

Design of Adaptive Artificial Neural Network using Approximate Computing and Partial Dynamic Reconfiguration and Experimental Results

3.1 Introduction to Artificial Neural Network Learning Process

Unlike the common ways that depend on the relation between inputs and outputs to detect the corresponding output to a certain stimulus or input, the artificial neural networks do not have a formula between inputs and outputs. ANNs learn the relation between inputs and outputs in the training phase, adjust the weights between the connections of the neurons and then the trained networks are used to generate the outputs directly from the inputs. This is called offline training, where the network is trained on the software layer, then the pretrained network is used in the application by using the saved weights. Online training is another training approach, in which the weights are adapted in the real-time. Object tracking is an example of applications that require online training.

To train the network, learning data is needed with its correct labels to enhance the weights. The network starts with random weights, then the learning data is used to produce output from the network. The produced output is compared with the expected output(labels). The error between the expected output and the network output is used to tune the weights accordingly in a manner such that the error is decreased. The weights' tuning is performed using a learning algorithm. The testing subset of the data-set should not be used for learning to avoid over-fitting.

The most used learning algorithm for the artificial neural network is the back-propagation and it is based on the gradient descent algorithm[23]. The weights are initially random data and have no meaning. The error between the target and the network output is calculated. Then, the network weights are adjusted as follows:

$$W_{new} = W_{old} - \delta\alpha \tag{3.1}$$

where α, δ are learning rate and error term, respectively. The learning rate starts with a large value, then it is tuned with a decay factor as the error decreases. This is called supervised learning. After the learning step of the network, the adjusted weights are used in the network's hardware implementation.

3.2 Research Hypothesis

The designers of limited energy Internet of Things (IoT) applications, such as wireless sensor nodes (WSNs), have several design trade-offs when selecting the most suitable ANN for their application. Given that approximated ANNs designs are reported in the literature with various energy-accuracy flavors, the limited energy applications designers usually select the ANN that exhibits moderate energy consumption and acceptable accuracy. In this work, the following research question is investigated ”Is it possible to reconfigure the application with different ANN flavors adaptively based on the available energy budget? and if yes, how much energy is saved?”. In other words, this work attempts to provide an energy-driven adaptive IoT application platform that is reconfigured with a specific ANN design according to the available battery energy at the expense of accuracy degradation. The ANN is used as a case study to answer this research hypothesis question with the application of the Partial Dynamic Reconfiguration feature of the FPGA, however, any design that has energy trade-offs can be used following the same work flow and methodology. Figure 3.1 displays a block diagram of the target limited energy application that consists of static modules and dynamic modules. The static modules are fixed and performing other functions of the application such as decision making circuits. The dynamic modules are reconfigured as follows. The battery energy sensor determines the battery energy level and accordingly, the PDR controller reconfigures the dynamic modules of the application with the corresponding ANN. For example, assuming that ANN1 has the highest energy consumption with the highest accuracy and ANN5 has the lowest energy consumption with the lowest accuracy. The reconfiguration methodology should be as follows. (1) when the battery energy level is from 1% to 20%, ANN5 should be selected, (2) when the battery energy level is from 21% to 40%, ANN4 should be selected, (3) when the battery energy level is from 41% to 60%, ANN3 should be selected, (4) when the battery energy level is from 61% to 80%, ANN2 should be selected, and (5) when the battery energy level is from 81% to 100%, ANN1 should be selected. The number of reconfigurations depends on the available ANN flavors and correspondingly, the energy levels values. In addition, the energy levels might be non-uniform in some applications depending on the ANN design energy trade-offs.

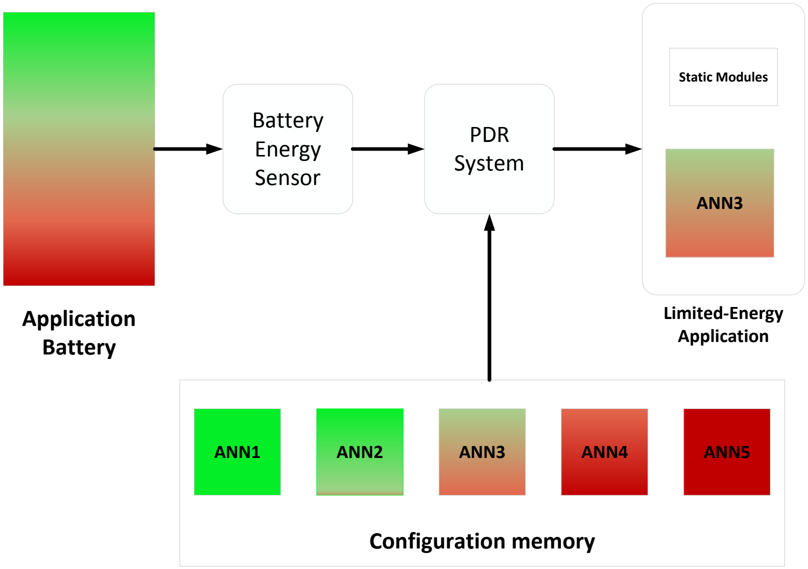


Figure 3.1: Limited-Energy application block diagram for using energy adaptive neural networks

3.3 Approximate Computing Techniques

As introduced in the previous chapter, there are so many used approximation techniques in literature that trade off the energy-efficient designs with the accuracy of the network. Each type of approximation reduces the power consumption and also degrades the accuracy as well. Not all the approximations reduce the power and accuracy by the same amount and each approximation combination has its degree of optimization.

As a first insight into the design approach, the approximations used in the thesis are listed with deep details, and then the combinations of approximations with each other to form a network is introduced. These different combinations are then used to form energy adaptive artificial neural network using the PDR(Partial Dynamic Reconfiguration) feature of the FPGAs. The approximations are precision scaling, approximate multiplier, approximate activation functions, truncated accumulation, neuron skipping and computation skipping.

3.3.1 Precision Scaling

Precision scaling is a widely used approximate computing technique [6, 7, 10]. For power and energy savings, the fixed point is used in the implementation of neural networks instead of the computationally expensive floating-point implementations. Floating-point arithmetic is much more expensive and complex than fixed-point ones, but they are more accurate and produce results similar to the software results.

Precision scaling is carried out by forcing the least significant bits(LSBs) of fixed-point operands to zero (Software precision scaling), or by implementing the design with reduced word length (hardware precision scaling). Both techniques reduce energy consumption by decreasing the switching activity. The word length is the number of bits of each piece of data in the design. If the word length is 10, then all data used in the design has to be represented in 10 bits (fraction part and integer part). Also, all hardware units have to support inputs of that word length. For example, the inputs to the neural network are the input data and weights, so they have to be represented as an array of vectors each vector is 10 bits. The inputs and weights are multiplied, then the multiplier will be 10bits x 10bits with an output of 20bits data and so on. As a result, every single hardware unit is affected by the choice of the word length. Another hardware element that is greatly affected by the word length is the memory required to store both the initial inputs and weights and the intermediate results between the network layers.

The choice of the word length is done on the software layer since the training of the network is done using Python (Offline training). It is required to tune the word length to get the least word length that produces the same accuracy result as the floating-point calculation. Lots of iterations are performed to decide which word length produces the floating-point accuracy and then a fixed-point representation of the data is used at the hardware layer. The more the word length is reduced, the higher the saving in the area and power at the expense of accuracy degradation.

At a certain word length, the full dynamic range of the bits should be used to achieve the highest accuracy for the chosen configuration. This requires a smart selection of the integer and fraction portions of the fixed point word length. For instance, when using Sigmoid or Tanh activation functions, the inputs of the neuron do not exceed ±1. Consequently, the integer length of the operands of the neuron unit is selected to be two bits, including the sign bit, which saves a room for representing the fraction bits. However, the values of the weights might exceed the range of the input values. In the training phase, the weights generated by the learning algorithm may be in any number. To overcome this problem and to keep the input size consistent for both inputs and weights, the values of the weights are forced to be in the same range of the input value (i.e., the weights are forced to have 1 as the integer part and any value as the fractional part). Again, this is performed on the software layer, after achieving the required accuracy, all weights of the network for all layers are observed and if their integer portion exceeds 1 they are scaled down to by a factor that makes them back again to 1 as their integer portion. The network is then tested after scaling the weights and if the accuracy is degraded a continue-training phase is performed to increase the accuracy again. If the retraining causes an increase in the integer portion, the scaling is done again. The loop of weights scaling, testing network and continue-training the network breaks when the required accuracy is achieved with all network weights have an integer portion of 1.

Table 3.1: A-law approximation of sigmoid function

x	-8	-4	-2	-1	1	2	4	8
y	0	0.0625	0.12	0.25	0.75	0.87	0.937	1

3.3.2 Approximate Activation Function

Activation functions are very important hardware processing elements in artificial neural networks. There are types of activation functions, this thesis uses two types of activation functions RELU and Sigmoid.

3.3.2.1 Sigmoid Activation Function

It is a commonly used activation function that has the formula 3.2

$$y = \frac{1}{1+e^{-x}} \tag{3.2}$$

and its derivative is shown in equation 3.3

$$\frac{dy}{dx} = y(1-y) \tag{3.3}$$

It is obvious from the sigmoid equation that the exact hardware implementation of Sigmoid is costly and requires intensive computation to implement the exponential term. There are many approaches to approximate the sigmoid computation as listed in the following section.

1. Lookup table implementation: in this approach, each value of x is used to calculate its corresponding y value and the results are stored in memory. This would be inefficient if high precision is used as it causes the use a large memory attached to each processing element.

2. A-law approximation: This method depends on modifying the resulting curve to linear segments, the curve is presented by 7 segments as shown in table 3.1

3. Alippi and Storti-Gajani Approximation: It depends on selecting a set of breakpoints of the first derivative and setting the function as a sum of the power of two numbers. For the reason that the sigmoid function has a symmetry point at coordinates (0, 0.5), only half pairs of x-y is calculated. Its expression is shown in 3.4:

$$Alippi(x) = \begin{cases} 1 - \frac{0.5+FRAC(-x)/4}{2^{|INT(x)|}} & x > 0 \\ \frac{0.5+FRAC(x)/4}{2^{|\mathrm{INT(x)}|}} & x \leq 0 \end{cases} \tag{3.4}$$

4. Piece-wise second-order approximation: It can be implemented as a second-order approximation, this approximation using the square operation which uses multiplier and thus not the most efficient implementation as shown in equation 3.5

$$f(x) = \begin{cases} 0.5(\frac{x}{4} - 1)^2 & -4 > x < 0 \\ 1 - 0.5(\frac{x}{4} + 1)^2 & 4 > x \geq 0 \end{cases} \tag{3.5}$$

5. Piece-wise linear approximation: It is a kind of approximation that perform the sigmoid function by modifying its curve to linear segments, each segment equation is calculated using shift and add operations, thus it is very hardware efficient implementation and consumes low area and power. The formula expressed in 3.6 describes the equation of each segment :

$$f(x) = \begin{cases} 1 & x \geq 5 \\ \frac{x}{32} + 0.84375 & 2.375 \leq x < 5 or - 5 \leq x < -2.375 \\ \frac{x}{8} + 0.625 & 1 \leq x < 2.375 or - 2.375 \leq x < -1 \\ \frac{x}{4} + 0.5 & 0 \leq x < 1 or - 1 \leq x < 0 \\ 0 & x < -5 \end{cases} \tag{3.6}$$

The last approximation PWL (piece-wise linear approximation) is used to approximate the sigmoid function since it is efficient and uses fewer hardware resources. As indicated in its equation, all division factors are powers of 2, and this is synthesized to shift operation and addition to the constant term, thus making the implementation easy and hardware friendly.

A comparative analysis between different approximations, their exact hardware utilization, power consumption, and maximum operating frequency can be found in [24].

3.3.2.2 RELU Activation Function

Linear activation functions such as RELU activation function is widely used in the implementation of artificial neural networks due to its small hardware and its efficiency in power reduction as well. Its hardware implementation is easy compared to the Sigmoid implementation, as shown in equation 3.7, it mainly contains two linear functions in the ranges when $x > 0$ and $x < 0$, thud it is seen from a hardware point of view as comparing with zero and assigning the output to a value accordingly. The main implementation issue in RELU function is that it passes the input to the output directly for the values of $x > 0$. If the full range of x is allowed to be transferred to the output y, this requires larger arithmetic units in the following layers that have the activation function outputs as their inputs. This is because the integer length of the fixed point representation is increased due to MAC operations. Instead of using bigger hardware in the following layers and to keep the design consistent, truncated versions of the RELU outputs are used to save power at the cost of reduced accuracy. The saturation means that if the input reached a certain threshold or

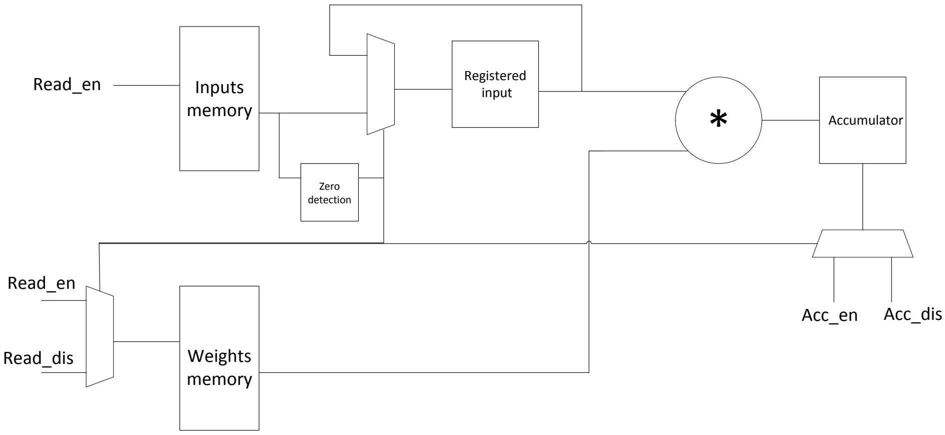


Figure 3.2: Simplified block diagram illustrating the computation skipping in the hardware implementation

a value above that threshold, the output retains the maximum value and can not increase above it. This helps to limit the linear increase in the output value as the input increases. The chosen input threshold is 8 so that the integer length does not exceed four bits.

$$f(x) = \begin{cases} x & x \geq 0 \\ 0.015625 * x & x < 0 \end{cases} \tag{3.7}$$

3.3.3 Computation Skipping Approximation

Computation skipping technique mainly depends on the shape of the input, in other words, it depends on the values that present the input and its efficiency or usefulness is directly related to the zero-valued inputs.

In many applications or data-sets, the number of zero, or near zero, valued inputs is high. Moreover, using precision scaling significantly increases this number. Also, using Rectified Linear Unit (RELU) layers, which force the negative valued layer outputs to zero, participates in increasing the number of zeroes in the system. These zero-valued inputs do not affect the operation of the network by any means, as they do not contribute to the output by any value. For example, if the input is zero it is multiplied by the corresponding weight and results in zero output, thus the first hardware operation that can be bypassed is the multiplication operation as its result is known ahead. Then this zero is added to the other inputs in the accumulator. The addition to zero does not change the output, so this operation can be skipped as well.

On the hardware side, a zero detection unit is added. It compares the input value to zero and indicates if the upcoming data to the neuron unit is zero, then the unnecessary computations are skipped. As shown in figure 3.2, the input data is read, then the zero detection unit checks if it is zero or not. Its output controls the inputs of the multiplier, so in case of zero, the registered input is used. The weights memory is not enabled for reading at this cycle, as a result, the second input of the multiplier is not changed. The switching activity on the multiplier inputs is decreased, and this saves the dynamic power dissipation. The accumulator is also disabled for that input, and this is another source of decreasing the switching activity. This procedure is very effective as it reduces the switching activity and dynamic power dissipation without any loss in the accuracy of the network result.

3.3.4 Neuron Skipping Approximation

To further decrease energy, some neurons in the hidden layers are skipped and all operations associated with them are not performed. For example, in a system with 10 physical neuron units running on MNIST data-set with a single hidden layer of 100 neurons, if ten of the 100 hidden neurons are skipped, 7860 (i.e., the number of inputs ×10) Multiply-and-Accumulate (MAC) operations and memory accessing are not performed reducing the time and energy needed for inference.

In contrast to computation skipping which avoids unnecessary computations and has no effect on accuracy, neurons skipping does have an impact on accuracy, but saves more energy. To reduce the impact of neuron skipping on the accuracy, the skipped neurons should be selected carefully. The selected neurons should be the most resilient ones so that skipping them has the smallest effect on the inference.

A neurons resilience ranking method has been proposed in [5]. This method depends on back-propagating the error at the output layer for each instance of the training set and calculates the average error contribution for each neuron to identify the least contributing neurons to the output error. After training the network, the neuron resilience ranking is done. The associated weights of the most resilient neurons are placed at the top of the memory of the weights so that if 10 neurons are selected to be skipped to reduce the consumed energy, the controller skips the weights of the first ten neurons in the memory and starts reading normally after them. Skipping some neurons in a neural network layer affects also the following layers. The input to the following layer from a skipped neuron is set to $g(zero)$ where $g(x)$ is the activation function of the previous layer, which equals to zero when using Tanh or RELU functions, allowing more energy reduction if computation skipping is used, and $g(x)$ equals to 0.5 when Sigmoid is used.

3.3.5 Inaccurate Arithmetic

Neural networks involve thousands of arithmetic operations such as multiplications and accumulations. Using inaccurate arithmetic operations introduces some errors which are tolerated due to neural network error resiliency. However, using these inaccurate arithmetic operations saves a big chunk of energy. A multiplier is one of the power-hungry units in digital neural networks. Many approximate multipliers are proposed in the literature to design a less accurate multipliers that save much power and area. Based on the discussion held in the previous chapter, this thesis uses the truncated multiplier due to its hardware efficiency and due to its good accuracy.

3.3.6 Approximate Adders

Approximate adders are used also to reduce the power consumption, however, they have a smaller impact on the energy savings than approximate multipliers. In this work, instead of using an approximate adder, a truncated accumulation is used rather than accumulating the whole output of the multiplier. Inspired by the truncated multiplier that is used in the thesis, a truncated accumulation is used. Instead of accumulating the whole output of the multiplier, it is truncated to the specified word-length of the design. The least significant bits of the multiplication result are truncated. Since most numbers in the design are fractional, the truncation would not degrade the output quality in a significant way. As a result, the size of the used accumulator is reduced, and the area and power dissipation of the accumulator is reduced with a very small impact on the accuracy.

3.4 Design Approach

In this work, the artificial neural network implementation utilizes all the previously mentioned hardware approximations. Different combinations of these approximations are also tested to get the best set of approximations suitable for a certain power and area budget at a certain acceptable accuracy. The main idea here is to have a large set of hardware elements, each with a different value of area, power, energy, and accuracy. And according to the available budget of power which varies across the time, the network can adaptively tune its hardware elements to fit within that budget at the cost of accuracy reduction. As time passes, the available energy reduces and the hardware adapts to that available energy with less accuracy, and then after the energy increases again the hardware adapts again to the new budget with higher accuracy and so on. With the help of this idea, the system is more flexible to the possible energy changes that always happen to any battery-based device such as smartphones. This would not be an easy task without the Partial Dynamic Reconfiguration (PDR) feature available on the FPGA platform. As an example for the low power applications that benefits from the PDR technique is the low-cost battery-powered wireless image sensor of precision agriculture. This application is used to perform pest control monitoring and is based on the use of insect traps conveniently spread over the specified control area. Depending on the targeted insect, each trap is properly installed with pheromones or other chemical substances that attract the insect that is intended to be captured. The wireless image sensor function is to classify the number of insects around the trap. When the battery energy is low, the neural network that performs the classification task reconfigures itself to a lower classification accuracy. As a result, the battery life lasts for more time. The approach here is to prove the concept that neural network generally can be power adaptive. Two data-sets are used with two different artificial neural network architecture for each data-set. The two data-sets are the well known MNIST data-set [25] and SVHN data-set[26]. In the following sections, the two data-sets are illustrated in more details and the hardware architecture of the neural network used for each data-set. Then the Partial Dynamic Reconfiguration (PDR) approach is introduced.

3.4.1 Data-sets

The data-sets is a very large database that contains a big amount of data collected and made available for use. Any neural network needs to be trained to set and adjust its weights at a suitable value to get the required accuracy. This training may be online in the real time operation of the network, or it may be offline by adjusting the weights, saving them in memory, and using them in the network real time processing. To perform the task of learning the network a large set of data is needed so the network learns from different sources to predict the output from the input. The learning data-set contains inputs and the labeled outputs, these labeled outputs are used as a reference. The output of the network is compared to the correct labeled output and the error is used to adjust and tune the network weights. The second set of data in any data-set is the testing set. This set is also a large set of data different from the training set and is used to test the network and calculate the accuracy. The testing data-set should not be used in the learning phase. The neural network architecture affects greatly the accuracy results obtained for each data-set, thus the network architecture that works with the MNIST data-set will not work with another data-set or application. Consequently, the network architecture is application based and its parameters(number of hidden layers and the number of neurons in each hidden layer) are tuned to get the required accuracy.

3.4.1.1 MNIST data-set and its artificial neural network architecture

MNIST (Modified National Institute of Standards and Technology) data-set is the basic data-set used in the area of the neural network, pattern recognition, and machine learning. It is set of binary images of handwritten digits, real-world data that consists of 60000 samples for the learning phase and 10000 sample for the testing phase. Each sample is a black and white image whose size is 28 x 28 pixels. The pixels values vary from 0 to 255, 0 means white and 255 means black. The attached labels take values from 0 to 9 as they are the digits. A sample data of MNIST data-set is shown in Figure 3.3.

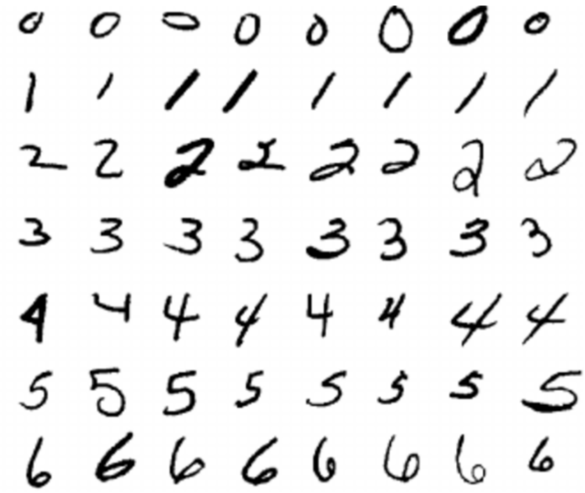


Figure 3.3: Sample data of MNIST data-set [25]

The artificial neural network that is used with the MNIST data-set is simple as the data to be processed is simple by nature. It is chosen to be a network of (785 - 100 -10). 785 input neurons (one for each pixel + 1 that is used as a bias to the network). 10 output neurons to have a neuron for each output digit. One hidden layer of 100 neurons, the number of hidden neurons is chosen to be multiple of 10 to reuse a set of physical neurons as will be discussed later.

3.4.1.2 SVHN data-set its artificial neural network architecture

SVHN (Street View House Number) data-set is a more complex data-set than the SVHN. It is real-world images that are taken from real house numbers in Google street view images so it is a harder and more complex data-set. A sample data of the SVHN data-set is shown in 3.4. Unlike the MNIST data-set, the SVHN data-set comes in RGB color format so it needs some sort of preprocessing before it is used. Another difference is that its images are a larger size, Each image is 32 x 32 pixels. It has 73257 digits in the training set and 26032 digits in the testing set. It also has 10 labels, but they are from 1 to 10. Digit 1 has label 1, digit 9 has label 9, but digit 0 has label 10.

The artificial neural network that is used with the SVHN data-set is bigger in terms of hidden layers and the number of neurons in each layer as the data to be processed is more complex and requires more processing to achieve reasonable accuracy. It is chosen to be a network of (1025- 300-300 -10). 1025 input neurons (one for each pixel + 1 that is used as a bias to the network). 10 output neurons to have a neuron for each output digit. Two hidden layers of 300 neurons each. The choice of the architecture, in this case, is done in an iterative, trial and error process to get the best architecture that results in a reasonable

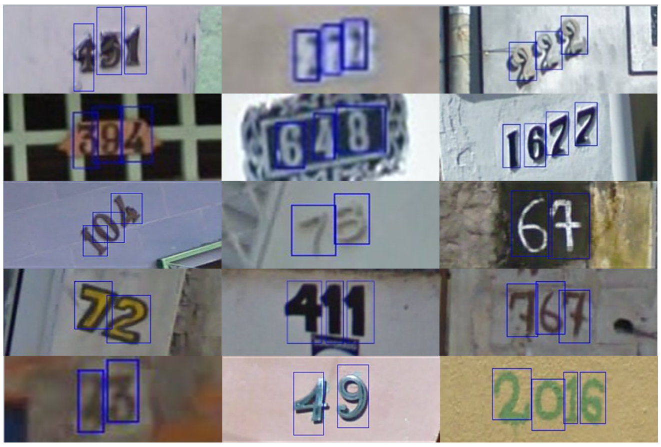


Figure 3.4: Sample data of SVHN data-set [26]

accuracy. Many architectures are trained and tested, but this is the one that achieved the highest accuracy among them.

3.4.2 Partial Dynamic Reconfiguration (PDR)

Partial Dynamic Reconfiguration is a very powerful technique in hardware implementations using the FPGAs (Field Programmable Gate Arrays). FPGAs are hardware platforms used to implement a specific design by configuring its resources, it allows programmability and flexibility for most design options. The advantage of dynamic reconfiguration is that it allows another degree of freedom to reconfigure part or all the FPGA hardware in the run time according to the application needs.

The FPGA internal structure differs from vendor to vendor and from version to another. But in general, any FPGA consists of two main parts: the hardware elements that it supports and the configuration memory that holds the bit-stream (binary) files used to program that hardware. The hardware layer in the FPGA contains the basic infrastructure needed to implement any design by changing the connections and contents of it. These hardware resources are mainly Lookup Tables (LUTs), Flip-Flops, memory elements and it may contain DSPs as well. The size and the specifications of the hardware elements differ with the type of the used FPGA. There is also a routing network that is used to route between the different hardware elements to create the needed logical connections. Another element is the clock tree that is used to supply the logic with the clocks needed for proper operation. The configuration memory in the FPGA is used to hold the information needed to configure the hardware elements, the routing details between the different resources, the values stored in the LUTs that implement a specific function, the reset values of the flip-flops used in the design, and all other details needed for proper operation. Thus, to change the functionality done by the FPGA, one has to change the configuration file, and when loaded on the FPGA, new hardware with a new function is operating.

The flexibility of programming the FPGA through altering the loaded bit-stream file is the main force that made the partial dynamic reconfiguration possible. First, the term partial refers to the ability to change part of the configuration file that affects a certain hardware logic to change or modify its functionality, and thus the other hardware elements that are not affected by this part of the configuration file are not changed. Second, the term dynamic means that the reconfiguration process is performed in the run time. The reconfiguration is done through an interface between the hardware and the configuration memory called Internal Configuration Access Port(ICAP) in Xilinx devices.

There are many advantages to using partial dynamic reconfiguration. It made it possible to have more than one hardware implementation of a function with the ability to choose among them according to the application needs. Even better, you have the ability to store many implementations for different functions and time multiplex between them as long as they have the same interface. The reconfiguration time for the partial reconfiguration is less than that of the full configuration as the time is directly proportional to the bit-stream file size. Consequently, the PDR is the best choice in an adaptive hardware design that has to respond according to an always changing environment [27].

In this work, and as discussed in the previous sections, all previous approximation techniques on the ANN are adopted to design an energy adaptive neural network (EANN) that re-configures the neural network hardware units with the required approximations to meet the available energy budget of the application. This energy adaptation comes at the expense of lower accuracy. As a result, the hardware is reconfigured in the run-time to have less optimum units and reduce the required energy for operation.

In the non-adaptive systems, if the available energy is not enough for the hardware unit to operate, the unit stops functioning and maybe it completely turns OFF. However, in the proposed EANN system, if the energy is not enough the hardware is reconfigured and adapted to work with this lower available energy at the expense of some loss in the accuracy which is acceptable in error-tolerant applications in which 100% accuracy is not a must all the time. In the PDR systems, the hardware modules are dynamically changed within an active design in response to the available application energy requirements as shown in Figure 3.5. The reconfiguration task is conducted in the run-time with no need to switch off the system for reconfiguration. The PDR technique is highly recommended when the application has multiple configurations to choose among and to reuse the same hardware physical resources for all configurations instead of implementing all of them.

The advantages of using the PDR technique is the flexibility of redefining the modules according to the current requirements. It also saves much area since there is no need to implement all different configurations of the design, and build a controller to switch between these different configurations, but instead, they are reconfigured and replaced.

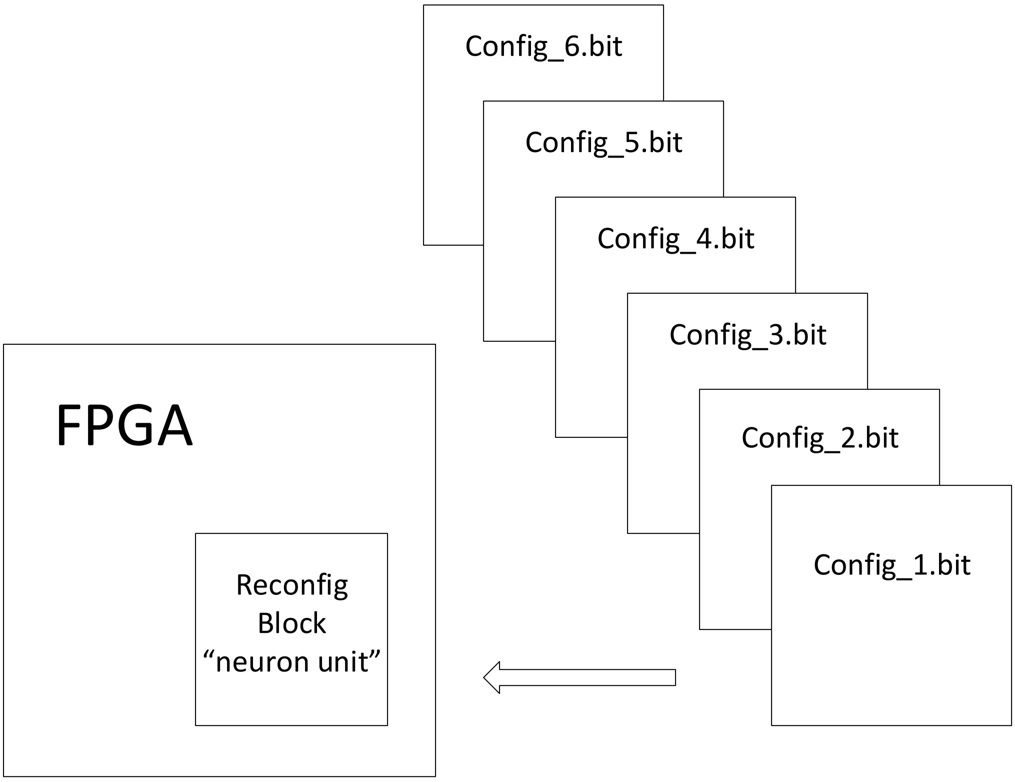


Figure 3.5: Illustration of Partial Reconfiguration

3.5 Experimental Setup

In this work, and as discussed in the previous chapters, approximate computing techniques used in the artificial neural network are adopted to design an energy adaptive neural network (EANN). The main idea of the EANN is that the hardware reconfigures its units to work with a less accurate and more energy-efficient version. The neural network is approximated with the required approximations to meet the available energy budget of the application. The hardware keeps track of the available energy over time, and when the energy level changes, it adapts its unit to fit in the available energy level. As a result, the hardware remains functioning for a longer time. The energy adaptation comes at the expense of lower accuracy. As the available energy decreases, the more hardware units are approximated, and the less accurate results are obtained. Another important point is that the hardware is reconfigured during the run-time to have a less optimum units, so the required energy for operation is reduced. The dynamic reconfiguration is achieved by using the FPGA platform. In the non-adaptive systems, if the available energy is not enough for the hardware unit to operate, the unit stops functioning and maybe it completely turns OFF, and this is the main strength in the EANN.

The experimental setup used to test the EANN is divided into two parts: the first is software-based and the second is hardware-based. The software part is used to perform the training task to the networks, and to get the weights and the accuracy results. The hardware part is used to implement the designs, and to get the area and power results using the weights obtained from the training step. The accuracy is also verified on the hardware after its implementation and compared to the software accuracy results as a part of the hardware verification.

3.5.1 Software Setup

The software is used as an abstract model of the artificial neural network, to train the networks, and to generate the weights needed in the hardware implementation. Python is used to model the artificial neural networks. The first step in the experiments was to choose suitable network architecture. The networks have many parameters to be tuned. Each configuration of the parameters is suitable for a certain application, and results in a different accuracy result. The MNIST data-set is easier than the SVHN, so the network needed for the MNIST simulations was a simple network that has the structure (785*100*10). First, the weights start with random values. The network training begins with a high learning rate, as the accuracy approaches the needed value, the learning rate is decreased. The weights are scaled-down with a ratio that makes all the weights lie between 1 and -1 for more optimized hardware. To overcome the scaling weights effects, a continue training option is used to continue the network training to enhance the accuracy. Finally, the weights are dumped to files to be loaded to the memory into the hardware implementations.

The SVHN data-set is more complex than the MNIST data-set as it is generated from real pictures taken from the street house numbers. The raw data of this data-set needs preprocessing to convert the RGB images to black and white scale to be the same format of the MNIST data-set. Another difference between the two data-sets is that the network used for the SVHN data-set is more complex. The network is chosen to be (1025*300*300*10). It has two hidden layers not only one layer, and each layer has more neurons, i.e., 300 instead of 100. The same process applies to the SVHN data-set, it is trained, the weights are tuned and scaled, and then dumped in the files for use in the hardware implementation.

3.5.2 Hardware Setup

The second important part in the implementation is the hardware implementation of the two neural networks used for MNIST and SVHN data-sets. Again, each network has a different hardware architecture that is discussed in details in the following block diagrams.

To implement such networks on the hardware, one can implement all the neurons physically. The second option is to implement certain number of neurons physically and reuse them in the calculations of other neurons. In this work, the second approach is adopted as this work targets limited energy applications. Ten physical neurons are implemented on the hardware and they are reused to obtain the results of all neurons. Thus, the first ten neuron operations are carried out and their results are saved in a hidden memory, then the second ten neuron operations are carried out, and so on until all the first layer neuron operations are carried out. The output of the first hidden layer is saved in the hidden memory and is used as the input to the output layer. Then, the same ten physical neurons are used to carry out the computation of the output layer. In the following subsections, the exact block diagram of each neural network is introduced.

3.5.2.1 MNIST Block Diagram

In this section, the block diagram of the network used in MNIST data-set is illustrated. The system top level consists of two memory blocks, one to store the inputs, and the other to store the intermediate results of the first hidden layer to use it as the input to the output layer. The system has ten physical neurons. One memory is attached to each one of the physical neurons to hold the weights associated to this neuron. A multiplexer is also attached to each neuron input to select the whether the input is taken from the input memory, or from the hidden memory. The hidden memory is used to store the output of the first layer. As a result, there is a multiplexer on the input of the hidden memory to select which neuron output to be saved, and to respect the arrangement of the neuron outputs in the memory. The output prediction unit is used in the output layer to predict the result of a certain input. A main control finite state machine is used to control the data movement between the blocks. It produces the control signals to all blocks such as memories read and write enables, registers resets, multiplexer selections, counters increment and decrement signals, and so on. The block diagram of MNIST network is shown in figure 3.6.

3.5.2.2 SVHN Block Diagram

In this section, the block diagram of the network used in SVHN data-set is illustrated. The system top level consists of three memory blocks, one to store the inputs similar to the one used in the MNIST block diagram, and two memories to store the intermediate results of the first and second hidden layers. The first hidden memory is used to store the outputs of the first hidden layer, and it is used as an input to the second hidden layer. The second hidden memory is used to store the outputs of the second hidden layer, and it is used as an input to the output layer. The system has ten physical neurons. One memory is attached to each neuron to hold the weights associated to this neuron. A multiplexer is also attached to each neuron input to select whether the input is taken from the input memory, the first hidden memory, or the second hidden memory. A multiplexer exists on the input of each memory to select which neuron output to be stored in which hidden memory, and to keep them in order. The output prediction unit is used in the output layer to predict the result of a certain input. A main control finite state machine exists to control the data movement between the blocks. It produces the control signals to all blocks such as memories read and write enables, registers resets, multiplexer selections, counters increment and decrement signals, and so on. The block diagram of MNIST network is shown in figure 3.7.

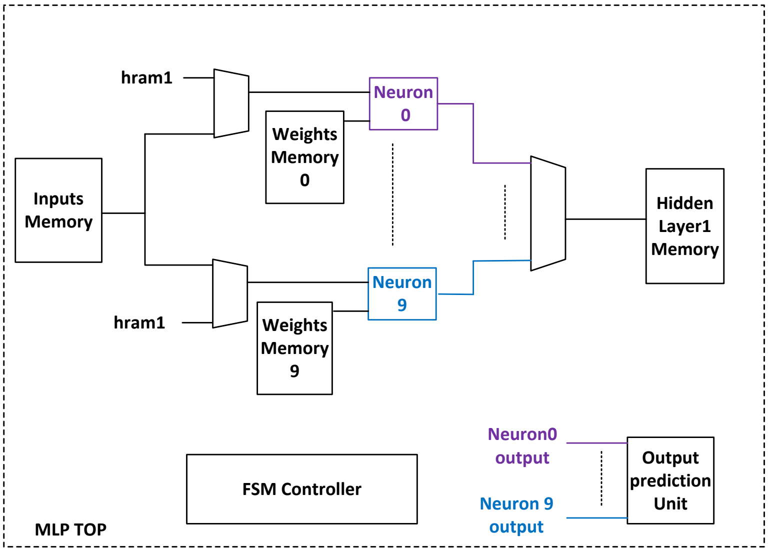


Figure 3.6: Block Diagram of the Network used for MNIST Data-set

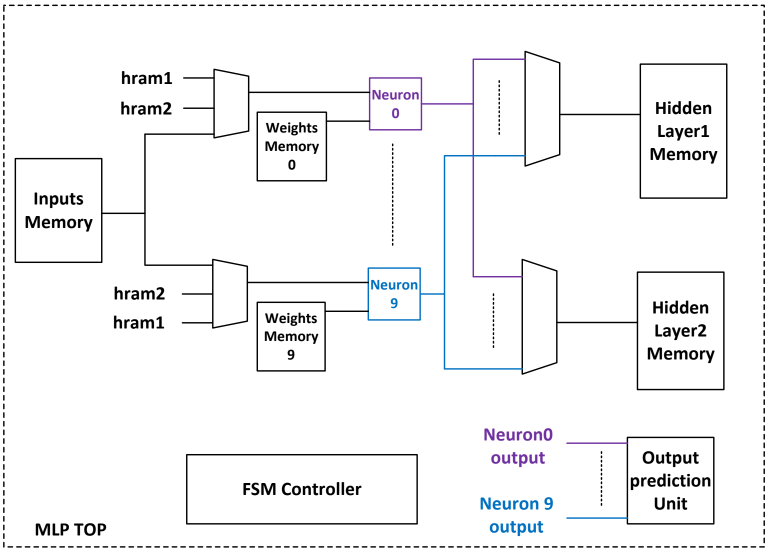


Figure 3.7: Block Diagram of the Network used for SVHN Data-set

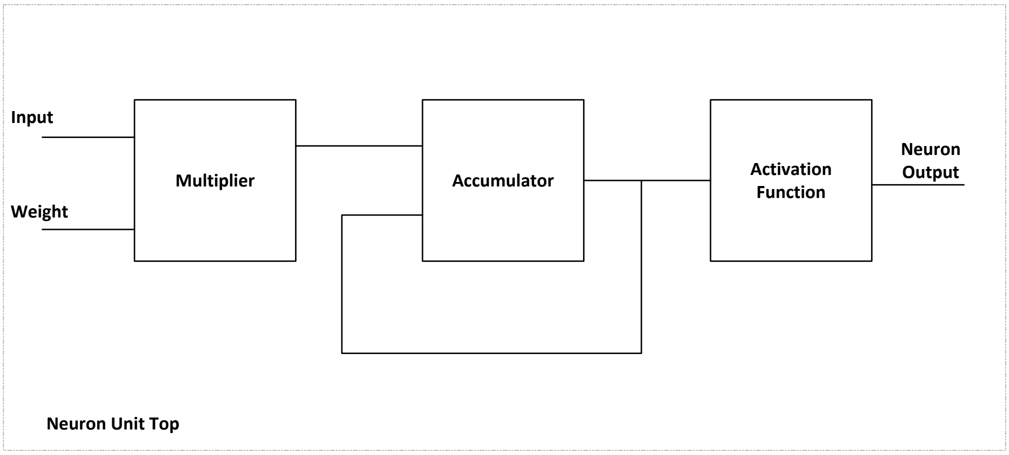


Figure 3.8: Block Diagram of the neuron unit used in both MNIST and SVHN networks

3.5.2.3 Neuron Top Block Diagram

Each physical neuron consists of three main hardware blocks. The multiplier which is used to multiply the inputs by the weights. The accumulator which is used to accumulate the output of the multiplier to the previous outputs. Finally, the activation function block that is used to generate the scaled output according to its type (sigmoid or RELU). The block diagram of the neuron unit is shown in figure 3.8.

All the approximations are hardware implemented (i.e., approximate multiplier, truncated accumulation, approximate activation function, computation skipping, neuron skipping and precision scaling). The precision scaling is common with other approximations. This means the word length changes gradually while using another approximation. This method creates a very wide set of hardware combinations, and each combination has different accuracy, power, area, and energy. Not necessarily that each combination has a unique result. Different approximations may produce the same results. The energy is obtained using the following formula:

$$Energy = \frac{n * p}{f} \tag{3.8}$$

where n is the number of cycles, p is the power in mw and f is the frequency of operation.

To explain the energy formula, the network of MNIST dataset is taken as an example. It consists of 785 input neurons, 100 hidden neurons, and 10 output neurons. It is a fully connected network, all the layer outputs are used as inputs to the following layer. The output of each neuron is calculated as the weighted sum of the neuron inputs passed by the activation function. To get the output for a certain input from such a network, it is needed to perform 78500 multiply-accumulate (MAC) operations in the first layer and 1000 operations in the output layer. Since the hardware uses only 10 physical neurons, then the number of cycles (n) to finish the whole computation is the total number of needed operations divided by the physical neurons count and equals 79500/10 = 7950 cycles.

Another item is calculated for each configuration is the accuracy loss. The accuracy loss is the percentage decrease of the accuracy measured from the highest accuracy value. Each data set has a different point that produces the highest accuracy result and this is discussed in the following sections.

3.6 MNIST Results

In this section, the results obtained from the MNIST data-set are discussed. For all the results of MNIST data-set, the accuracy loss is calculated with reference to the highest accuracy. The highest consumed energy is obtained when the ANN operates using word length of 12-bits (because experimental results show that the 12-bits word length results in the same accuracy as the 32-bits and the floating point in case of the MNIST dataset) and utilizing computation skipping technique, and this implementation (i.e., 12-bits and computation skipping) also gives the highest accuracy. The configuration that results in the highest accuracy result also consumes the highest energy.

3.6.1 MNIST Energy results

In figure 3.9, the different configurations are introduced. Each point on the graph shows a certain accuracy loss and a certain normalized energy value. Each color on the graph indicates a certain configuration. For example, the red color indicates using RELU activation function, computation skipping, and truncated accumulation techniques. Each color has more than one point, each point indicates a certain word length. As the word length decreases, the accuracy loss increases, however, the normalized energy decreases. At each energy level, many points fulfill this energy, but the best point to choose is the one with the lowest accuracy loss. This graph is used to pick the best point in each energy level to be used later for reconfiguration.

3.6.2 Effect of computation skipping

The first important optimization in this network is the computation skipping. The MNIST data-set by nature is black and white images represented as pixel, and each pixel has intensity value. This results in a large number of pixels with zero value. The computation skipping is the best optimization to use in this case, as it skips the unneeded computation while maintaining the same accuracy.

Figure 3.10 shows two curves, one using the computation skipping technique, and the other is the normal results without using the computation skipping technique. The result

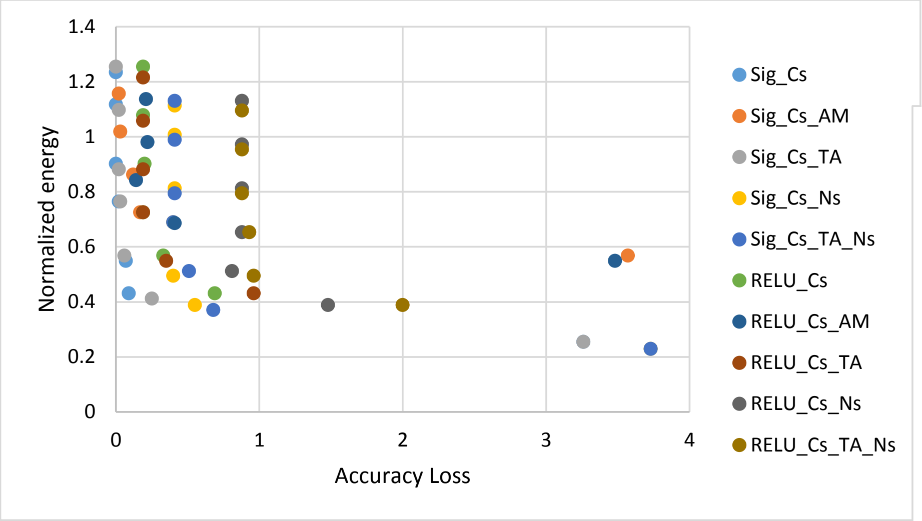


Figure 3.9: MNIST : Approximation results for MNIST data-set, sig: Sigmoid activation function, Cs: Computation skipping, AM: Approximate Multiplier, TA: Truncated Accumulation, Ns: Neuron skipping, RELU: RELU activation function

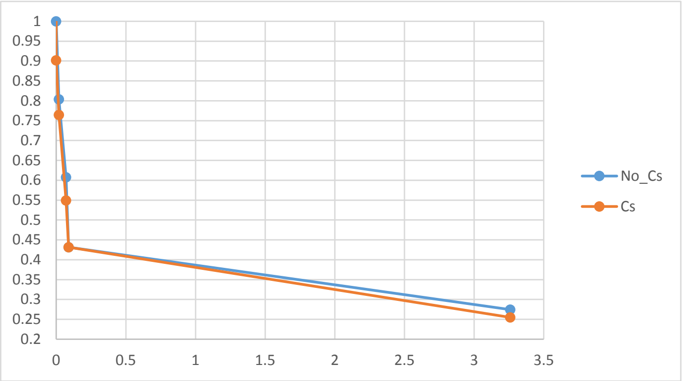


Figure 3.10: MNIST : Effect of computation skipping on the accuracy versus consumed energy

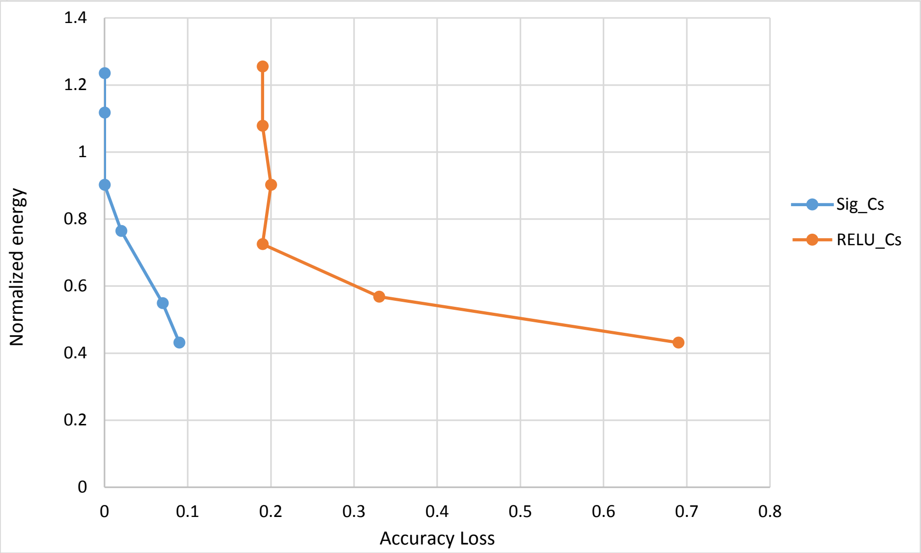


Figure 3.11: MNIST : Comparison between Sigmoid and RELU in terms of Energy and Accuracy Loss

indicates that the hardware utilizing the computation skipping has the same accuracy loss as the hardware without computation skipping, but it uses less energy. Therefore, the computation skipping technique is used as the default hardware configuration.

3.6.3 Sigmoid Vs RELU

The second observation from the results of the MNIST data set is the major difference between RELU and Sigmoid activation functions. In terms of accuracy, using the RELU activation function produces lower accuracy than the Sigmoid activation function. Each point on the graph shown in 3.11 indicates that any configuration using RELU activation function for different word lengths can be found using Sigmoid activation function on the same energy level with a lower accuracy loss. This means that sigmoid approximation is better than RELU approximation. The reason that RELU produces lower accuracy results is that saturation is used in the implementation of RELU to limit the integer part of the output of RELU. This saturation is used to save power at the cost of reduced accuracy. According to this observation, all RELU points are excluded from the search space and the configuration selection step includes Sigmoid points only.

After removing the RELU activation function results from the selection points, the remaining configuration that can be used for the selection of the best combination between area, power and accuracy are shown in figure 3.12.

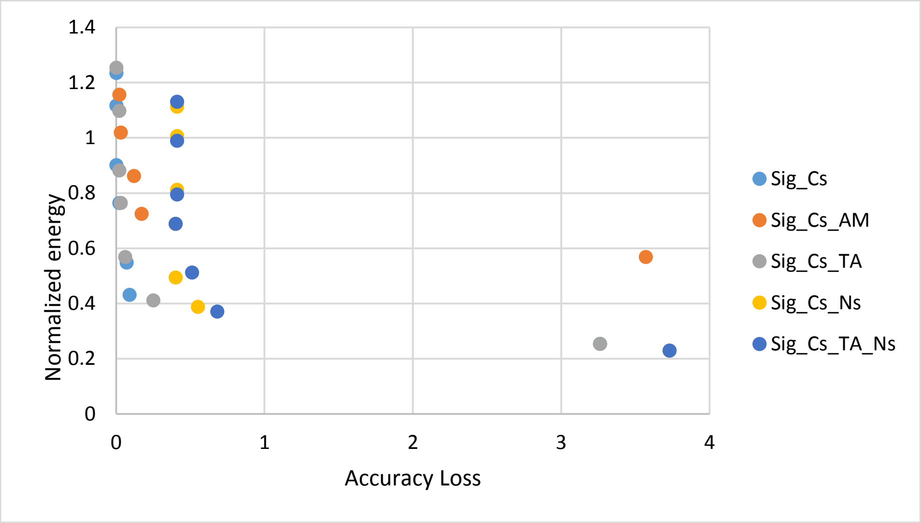


Figure 3.12: MNIST : Configuration points used as a searching space to choose the suitable implementation at a given energy for MNIST data-set, sig: Sigmoid activation function, Cs: Computation skipping, AM: Approximate Multiplier, TA: Truncated Accumulation, Ns: Neuron skipping

3.7 SVHN Results

In this section, the results obtained using MNIST data-set SVHN are discussed. For all the results of the SVHN data-set, the accuracy loss is calculated with reference to the highest accuracy. The highest consumed energy is obtained when the ANN operates using word length of 8-bits (because experimental results show that the 8-bits word length results in the same accuracy as the 32-bits and the floating point in case of the SVHN dataset) and without utilizing any other approximation technique, and this configuration also gives the highest accuracy. The configuration that results in the highest accuracy result also consumes the highest energy.

3.7.1 SVHN Energy results

In figure 3.13, the different configurations are introduced. Each point on the graph shows a certain accuracy loss and a certain normalized energy. Each color on the graph indicates a certain approximation. For example, the red color indicates not using any approximation technique while getting these results. Each color has more than one point, each point indicates a certain word length. As the word length decreases, the accuracy loss increases, but the normalized energy decreases. At each energy level, many points fulfill this energy, but the best point to choose is the one with the lowest accuracy loss. This graph is used to pick the best point in each energy level to be used later for reconfiguration.

It is worth to note that the computation skipping technique is not used in the SVHN network implementation. This is because that the input images in the SVHN data set

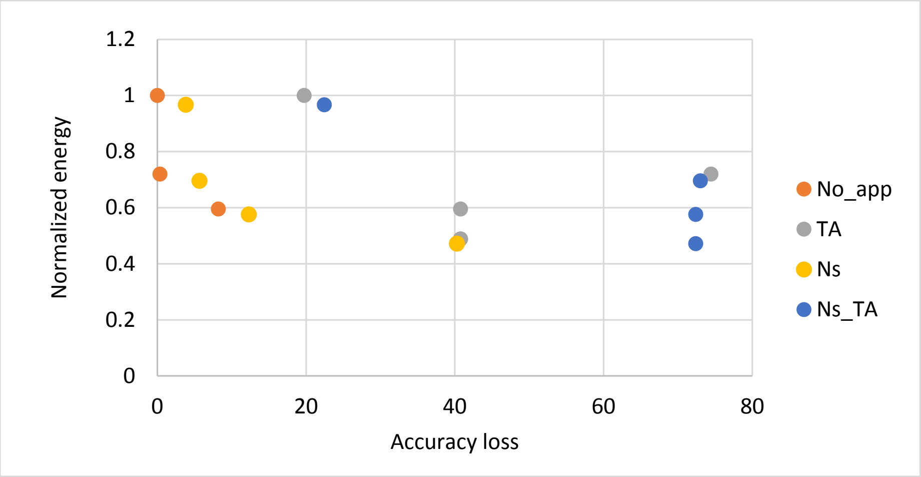


Figure 3.13: Approximation results for SVHN data-set, No-app: No approximation, Ns: Neuron skipping, TA: Truncated Accumulation

come in an RGB format, they are not black and white images. As a result, not many zero-valued inputs exist in that data-set. If the computation skipping is added for this data-set, this would cause a little increase in the area due to the added hardware for checking the value of the input. With this increase in hardware, the dynamic power does not decrease due to the absence of the zero inputs. Therefore, the computation skipping technique is not used at all in the SVHN data set implementation.

3.8 Proposed Algorithm and Configurations Selection

To introduce the proposed energy adaptive neural network (EANN) algorithm, suppose the use of a battery-powered application. In these applications, if the voltage level that feeds the hardware circuits falls below a certain level, it causes the system to power OFF. If the profile of the voltage drop with time is known, then the amount of the available energy for the system to operate for a certain amount of time is also known. The EANN system can make use of this information to adapt its hardware units, and to operate with lower energy at a lower accuracy mode, thus the battery lasts for more time.

As explained in the earlier sections in this chapter, both MNIST and SVHN networks have more than one hardware configuration. Many approximations with different word length are implemented, thus produces many configurations to the system. Each neuron configuration has a different area, power, and accuracy. These different configurations are used to adapt the hardware units to the given energy budget.

The hardware reconfiguration is achieved by using the FPGA platform and the partial dynamic reconfiguration feature. The logic in FPGA is divided into two parts: the static part which models the memory used for storing weights and the top-level circuits, and the dynamic (re-configurable) part which is the neuron unit. During reconfiguration, the

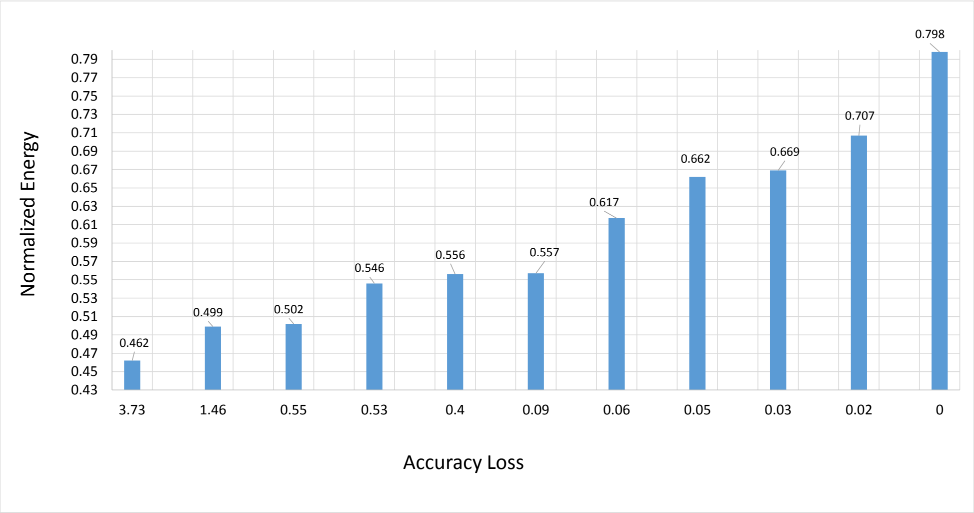


Figure 3.14: MNIST: Energy levels that the proposed EANN system uses to adapt to the given energy budget

Table 3.2: power and accuracy regions for MNIST data-set

Normalized Energy	Accuracy Loss%	Configurations
0.23	3.73	Cs_TA_Ns_4
0.255	3.26	CS_TA_4
0.371	0.68	Cs_TA_Ns_6
0.389	0.55	Cs_Ns_6
0.412	0.25	Cs_TA_6
0.431	0.09	Cs_6
0.549	0.07	Cs_8
0.569	0.06	Cs_TA_8
0.765	0.02	Cs_10
0.9	0	Cs_12

CS: Computation Skipping, NS: Neuron Skipping, TA: Truncated Accumulation

static part remains functioning and the dynamic part changes without turning the system OFF. This allows the EANN system to select among the different configurations to adapt itself to meet the energy constraint without completely turning OFF the system.

For the MNIST data-set, the energy is divided into ten levels in this test case as shown in figure 3.14. In the EANN system, there is a sensing circuit that could determine the available energy, then the comparators select the nearest level to that energy level. After choosing the best-fit energy level, the suitable configuration that fits with this energy budget is selected from 3.2. For example, if the sensing circuit outputs energy greater than or equal to 0.9 (i.e., 90% of the total energy of the unapproximated neural network), then the proposed EANN system operates with the highest performance and zero accuracy loss.

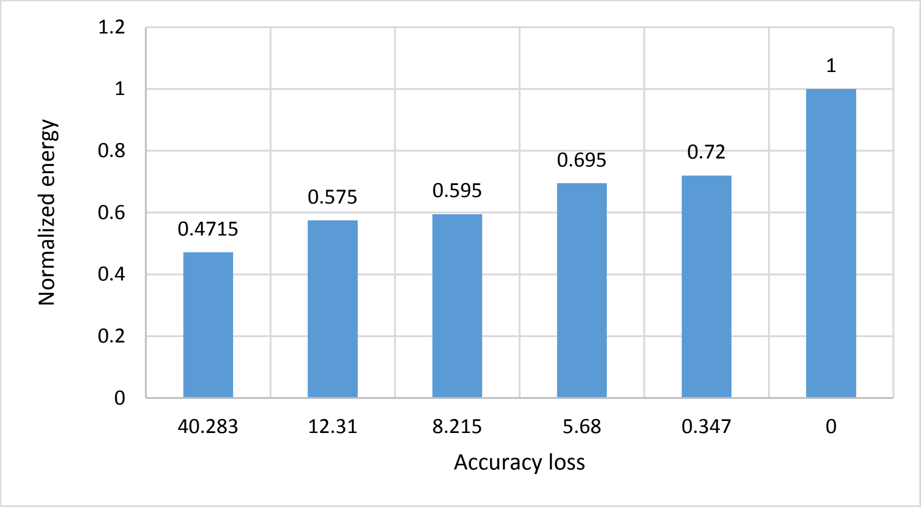


Figure 3.15: SVHN: Energy levels that the proposed EANN system uses to adapt to the given energy budget

Table 3.3: power and accuracy regions for MNIST data-set

Normalized Energy	Accuracy Loss%	Configurations
0.4715	40.283	Ns_4
0.575	12.31	Ns_5
0.595	8.215	No-App_5
0.695	5.68	Ns_6
0.72	0.347	No-App_6
1	0	No-App_8

No-App: No approximation, NS: Neuron Skipping

From Table 3.2, the configuration that is selected in this case is the neuron unit that supports computation skipping using a word length of 12. If the energy drops to 0.765 (i.e., 76.5% of the total energy of the unapproximated neural network), then the second energy level with accuracy loss 0.02% is selected and the FPGA is reconfigured. The configuration that is used in this case is the neuron unit that supports computation skipping and truncated accumulation with word length 10.

For the SVHN data-set, the energy is divided into six energy levels as shown in figure 3.15. Here in this case, if the sensing circuit outputs full energy, then the proposed EANN system operates with the highest performance and zero accuracy loss. From Table 3.3, the configuration that is selected in this case is the neuron unit with no approximation using a word length of 8. If the energy drops to 0.72 (i.e., 72% of the total energy of the unapproximated neural network), then the second energy level with accuracy loss 0.347% is selected and the FPGA is reconfigured. The configuration that is used in this case is the neuron unit with no approximation and word length 6.

It should be noted that none of the selected configuration contains approximate multiplier, this is because that any combination contains approximate multiplier has higher accuracy loss compared to other configurations with the same energy. Therefore, there is always a replacement to the approximate multiplier approximation with any other approximations.

The cost of the re-configurability is the reconfiguration time, which is the time consumed by the dynamic part to change from one configuration to another one. Another overhead of re-configurability is that the area occupied on the FPGA is the area of the largest re-configurable module of the selected re-configurable modules. However, the strength of the proposed energy adaptive technique is that the EANN system does not shut down when it fails to achieve the highest accuracy. Instead, it keeps operating with the lower available energy budget at the expense of lower accuracy.

3.8.1 comparison between conventional system and EANN system

To compare the proposed EANN system and the conventional ANN systems, suppose that system A (a conventional ANN system) needs normalized energy of 0.8 (i.e., 80% of the total energy of the unapproximated neural network) to work with the best performance and no accuracy loss. In system A, if the energy falls below 0.8, the system turns off. In the proposed EANN system B, if it is given 0.8 energy, it works with the best performance and no accuracy loss. However, in system B (the proposed EANN system) if the energy falls below 0.8 and becomes 0.6 (i.e., 60% of the total energy of the unapproximated neural network), it re-configures its units and works with the available energy budget with an accuracy loss of 0.09%.

If there is always available energy of 0.8, systems A and B are of the same performance. When system B has energy less than 0.462 (i.e., 46.2% of the total energy of the unapproximated neural network), it also shuts down. The main drawback of system B is that it occupies a total area equivalent to the conventional ANN system (i.e., system A) even if the available energy is between 0.4 and 0.8. However, from an energy perspective, system B adapts itself to the available energy budget trading off the accuracy for energy adaptively. System B is the best choice if the available energy is always changing. The overhead of the reconfiguration is the reconfiguration time between the different modules. Figure3.16 shows the difference between the proposed EANN system and the conventional system that targets maximum accuracy over time from the functionality perspective.

3.8.2 Partial Dynamic Reconfiguration Results

Figure 3.17 shows the design floor-planning on “ZYNQ Ultra-Scale Plus” MPSoC in case of SVHN data-set, and figure 3.18 shows the design floor-panning in case of MNIST

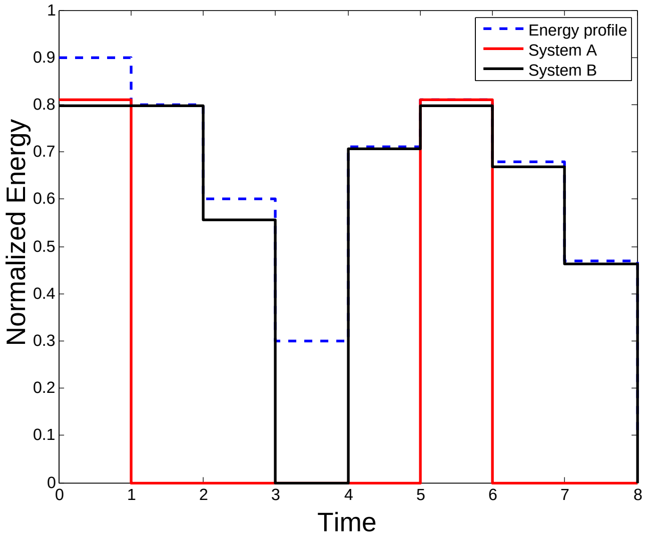


Figure 3.16: Comparison between conventional systems and energy adaptive system when exposed to variable energy

data-set. The re-configurable partitions of this design are the 10 physical neurons, they are floor-planned to accommodate the maximum hardware needed among all configurations. The most important benefit of partial dynamic reconfiguration is that there is no need to switch the system off to perform the reconfiguration task, instead, it takes place in the run time. The re-configurability of the FPGA allows utilizing multiple implementations of the same module that does the same functionality, but with different accuracy and energy consumption. This work applies the idea of PDR on the artificial neural networks, where the re-configurable module is the neuron unit that has different implementations with different approximations. It is tested on two different artificial neural network structures one for SVHN data-set and the other for MNIST data-set.

Partial dynamic configuration saves much area. It is not needed to implement all the possible configurations of the system and build a controller to perform the switching task between them. As shown in Table 3.4 for SVHN data-set, if all possible configurations of the neuron unit are implemented on the board, this consumes much area and power consumption.

The needed area to implement all configuration is about (11974 CLBs and 466 BRAMs) and consumes a total power of about 339 mw. The area needed to implement the EANN system is the area of the largest reconfiguration module (highest accuracy), and the area of the static routing connections which is about (4297 CLBs and 161.5 BRAMs). This indicates that the EANN system achieves about 2.8X area and power reduction.

In the case of MNIST data-set as shown in Table 3.5, if all configurations are implemented on the same board, the area needed is about (27987 CLBs and 140.5 BRAMs) and consumes a total power of about 198 mw. The area needed to implement the EANN system is the area of the largest reconfiguration module (highest accuracy) and the area of

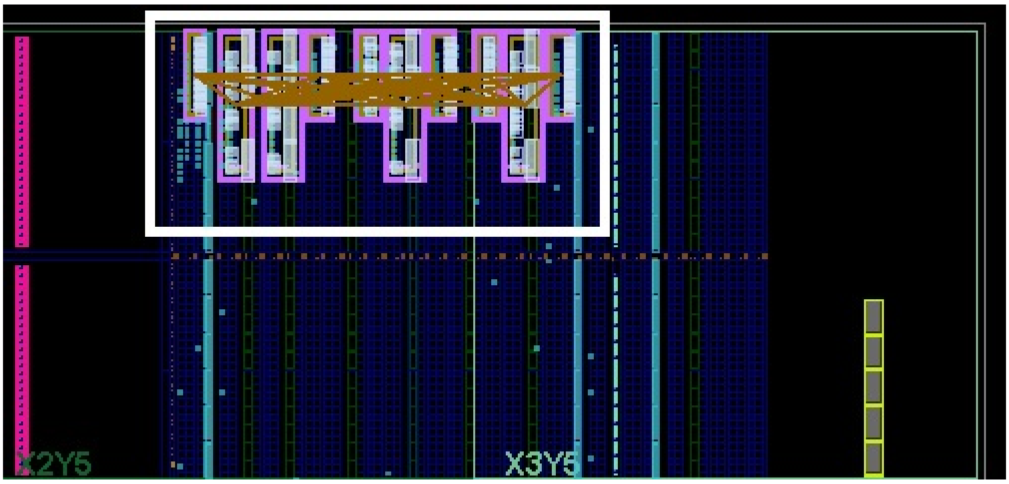


Figure 3.17: SVHN : Floor-planning and static routing (re-configurable area) of the physical neurons in case of SVHN data-set

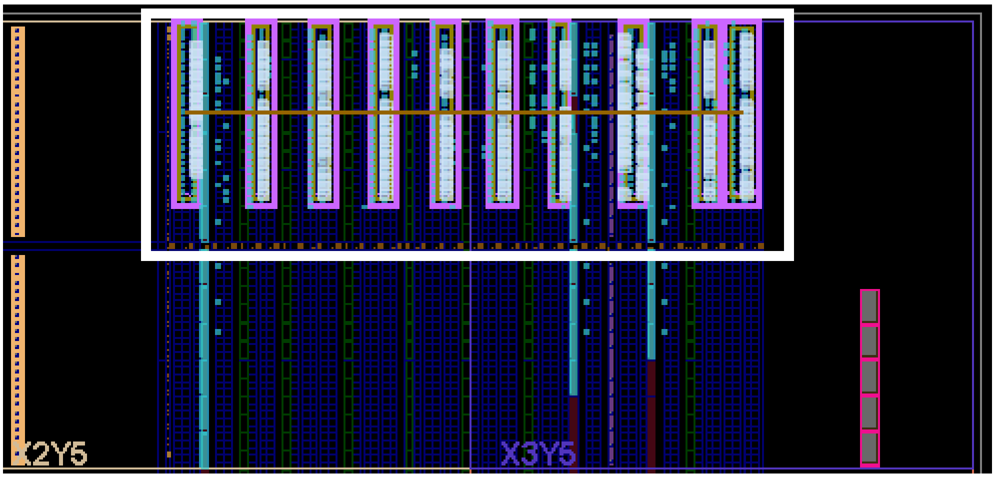


Figure 3.18: MNIST : Floor-planning and static routing (re-configurable area) of the physical neurons in case of MNIST data-set

Table 3.4: Area, Power and accuracy results for SVHN configurations

Configuration	Area		Power(mw)	Accuracy(%)
	CLB	BRAM		
WL_8	4297	161.5	121	80.58
WL_6	3015	121.5	87	80.3
WL_5	2489	101.5	72	73.96
WL_4	2173	81.5	59	47.73

Table 3.5: Area, Power and accuracy results for MNIST configurations

Configuration	Area		Power(mw)	Accuracy(%)
	CLB	BRAM		
Cs_12	7369	31	46	97.92
Cs_10	5715	26	39	97.9
Cs_TA_8	3736	21	28	97.86
Cs_8	3980	21	29	97.85
Cs_6	2728	15.5	22	97.83
Cs-_TA_6	2606	15.5	21	97.67
Cs_TA_4	1853	10.5	13	94.66

the static routing connections which is about (7369 CLBs and 31 BRAMs). This indicates that the EANN system achieves about 3.8X area and around 4X power reduction.

The overhead of using PDR is the reconfiguration time needed to load one configuration of the neuron unit and the need for external memory to store the partial bit files of all neuron unit configurations. The reconfiguration time = (Total Partial Bit-Stream File Size / Throughput). The commonly used access port for PDR is the Internal Configuration Access Port (ICAP). The throughput of the ICAP is 400 MB/sec. The reconfiguration time depends on the area of the partition and the speed of the PDR controller. Using the"ZYNQ Ultra-Scale Plus" MPSoC, the maximum reconfiguration time to reconfigure the ten physical neurons is 3.7625 ms for MNIST data-set, and 2.7825ms for SVHN data-set. The reconfiguration time is very short compared the rate of energy changes. This assumption is emphasized by noting that the energy level changes are mainly due to the harvested energy variations or the battery energy level fluctuations, which are usually in the order of seconds.

3.9 Conclusion

This work represents a new method to make the neural network energy adaptive using the partial dynamic reconfiguration feature in the FPGA platform. The proposed EANN system uses a variety of network approximation techniques such as precision scaling, approximate multipliers, truncated accumulation, approximate activation function, computation skipping, and neuron skipping. The idea is tested using two data-sets, SVHN and MNIST. Each one is tested on a different neural network with different network architecture. A combination of different configurations is then utilized to achieve a wide range of energy levels to adapt to the available energy budget in the proposed EANN system at the expense of degraded accuracy. Using the partial dynamic reconfiguration technique, the proposed EANN system selects one of the configurations at a time to adapt to the energy budget at the expense of lower accuracy. With the proposed approach, the EANN system is always functioning with variable accuracies according to the available energy level rather than the ON-OFF conventional system behavior.

Chapter 4

Design and Implementation of Izhikevich neuron model using Approximate Multiplier

4.1 Izhikevich neuron model

The Izhikevich neuron model is seen as a very effective model. It combines some of the biologically plausible properties as Hodgkin–Huxley model, while it is computationally efficient as integrate and fire models. The model depends on four parameters a,b,c,d. Tuning these parameters reproduce different spiking patterns. The model is a two differential equations :

$$\frac{dv}{dt} = 0.04v^2 + 5v + 140 - u + I \tag{4.1}$$

$$\frac{du}{dt} = a(bv - u) \tag{4.2}$$

With the reset after spike equations :

$$if\ v \geq 30mV,\ then \begin{cases} v & \leftarrow c \\ u & \leftarrow u + d \end{cases} \tag{4.3}$$

Where a, b, c, d are dimensionless parameters, and t is the time variable. v is the membrane potential, and u is the membrane recovery variable. After the spike reaches its threshold 30 mv, the membrane voltage and recovery variables reset to their default values. I represents the input stimulus current.

The resting potential in the model is about -70 mv to -60 mv. The model does not have a fixed value for the threshold as real neurons. It can range from -55mv to -40mv. Each parameter of a, b, c, and v has a certain meaning:

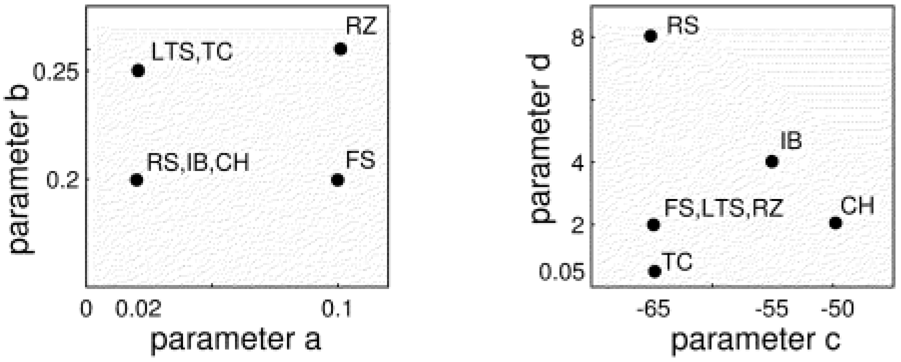


Figure 4.1: Izhikevich model parameters and the resulting spiking patterns RS(Regular Spiking), IB (Intrinsically Bursting), CH(Chattering), FS(Fast Spiking), LTS(Low Threshold Spiking) [28]

1. a is the time scale of the recovery variable u. Smaller values of a lead to slow recovery. Its typical value is 0.02.
2. b is the sensitivity of the recovery variable to the dynamics of the membrane voltage. Larger values of b mean that u and v are directly coupled and results in oscillations and low-threshold dynamics. Its typical value is 0.2.
3. c is the reset value of the membrane voltage after a spike. Its typical value is -65mv.
4. d is the reset value of the recovery variable after a spike. Its typical value of 2.

The choice of these parameters produces different spiking pattern, figure 4.1 shows the selection of the different parameters and the resulting pattern for each set of parameters [28].

4.1.1 Izhikevich neuron patterns experimental results

The excitatory neurons are classified according to the spiking patterns that they produce. Their classifications are: Regular spiking (RS), Intrinsically bursting (IB), and Chattering (CH). Inhibitory neurons are classified to Fast spiking (FS) and Low Threshold Spiking (LTS). First of all, the Izhikevich neuron model is simulated on MATLAB to make sure that it regenerates most of the spiking patterns. In the following subsection the results obtained from the simulations are presented.

4.1.1.1 Regular spiking

Regular spiking is the most common spiking pattern. The neuron fires spikes when it is exposed to a DC current. There is a period between each spike called inter-spike period

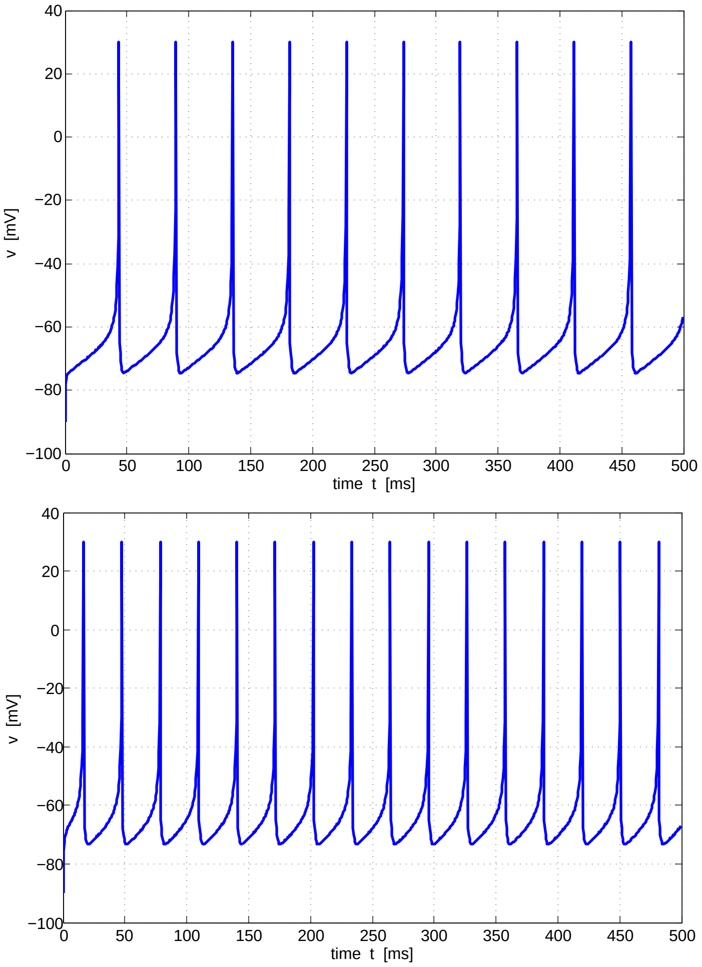


Figure 4.2: Regular spiking pattern, and the increasing inter-spike frequency at increasing the input dc current

and the frequency of the spikes increases by increasing the input current. The model parameters of this type of spiking are: a=0.02, b=0.2, c=-65mv, and d=8. Figure 4.2 shows the regular spiking and the effect of increasing the DC current on the inter-spike period/frequency.

4.1.1.2 Intrinsically Bursting

Intrinsically bursting neurons are the second type of the excitatory neurons, their firing pattern is somehow the same as the regular spiking, however, they fire a burst of spikes at the start followed by regular spikes with a certain inter-spike period. The model parameters to reproduce this pattern are: a = 0.02, b=0.2, c=-55mv, and d=4. Figure 4.3 shows the intrinsically bursting spike pattern.

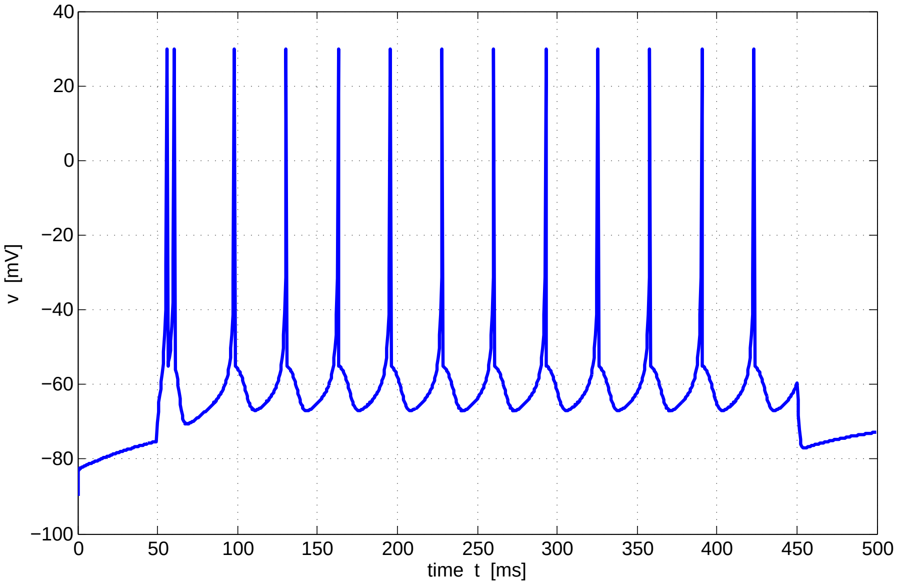


Figure 4.3: Intrinsically bursting spiking pattern

4.1.1.3 Chattering

Chattering neurons are the last set of excitatory neurons. Their firing pattern is different from the last two types. They fire bursts in a repetitive manner with a certain inter-bursts period. As the injected DC current increases, the number of spikes per burst increases. To reproduce this pattern, the model parameters are: a=0.02, b=0.2, c=-50mv, and d=2. Figure 4.4 shows the chattering pattern and the effect of increasing the input stimulus on increasing the number of bursts per spike.

4.1.1.4 Fast Spiking

Fast spiking neurons are the first type of inhibitory neurons. The neurons fire periodic spikes with very high frequency relative to the frequency of the regular spiking neurons. The model parameters of this pattern are: a =0.1 that provides the needed fast recovery and b = 0.2, c =-65mv, and d=2. Figure 4.5 shows the fast spiking pattern.

4.1.1.5 Low Threshold spiking

The last type of inhibitory neurons is the low threshold spiking pattern. In this pattern, the neurons fire a high frequency spikes with a frequency adaption, i.e., slowing down the frequency after some time. They also fire at a low threshold which is obvious from the value of b = 0.25 in the model parameters. The rest of the model parameters are: a=0.02, c=-65mv, and d=2. Figure 4.6 shows the low threshold spiking pattern.

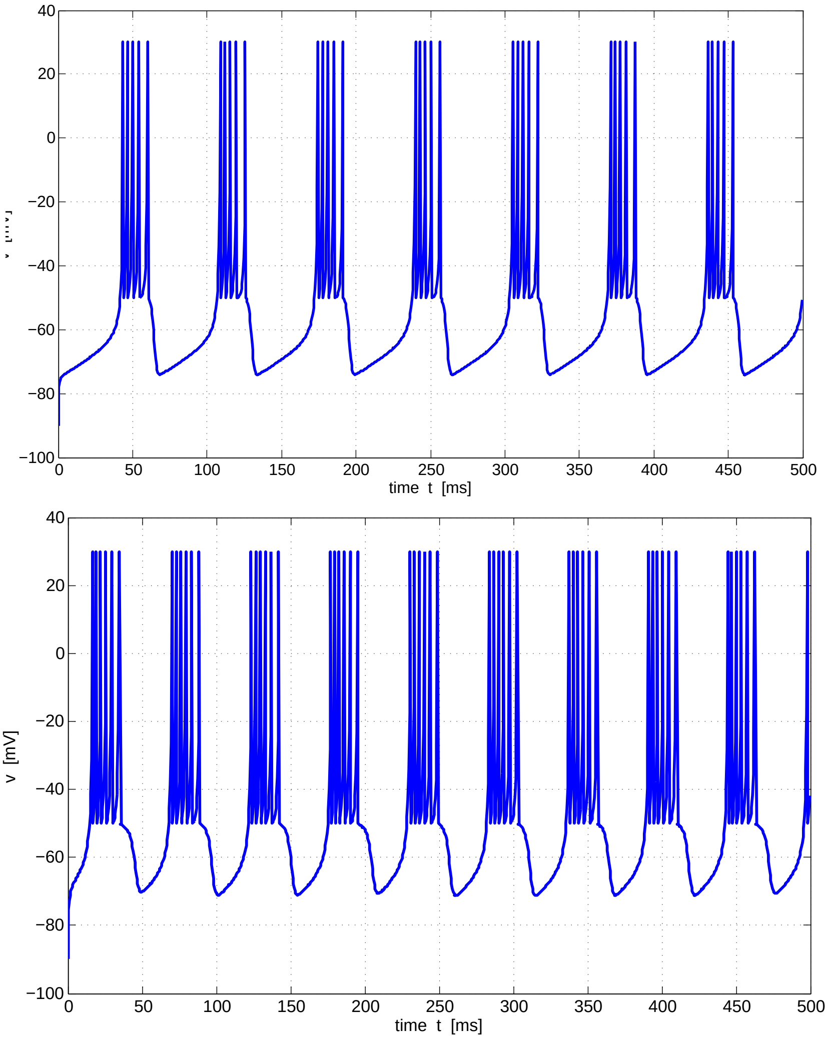


Figure 4.4: Regular spiking pattern, and the increasing inter spike frequency at increasing the input dc current

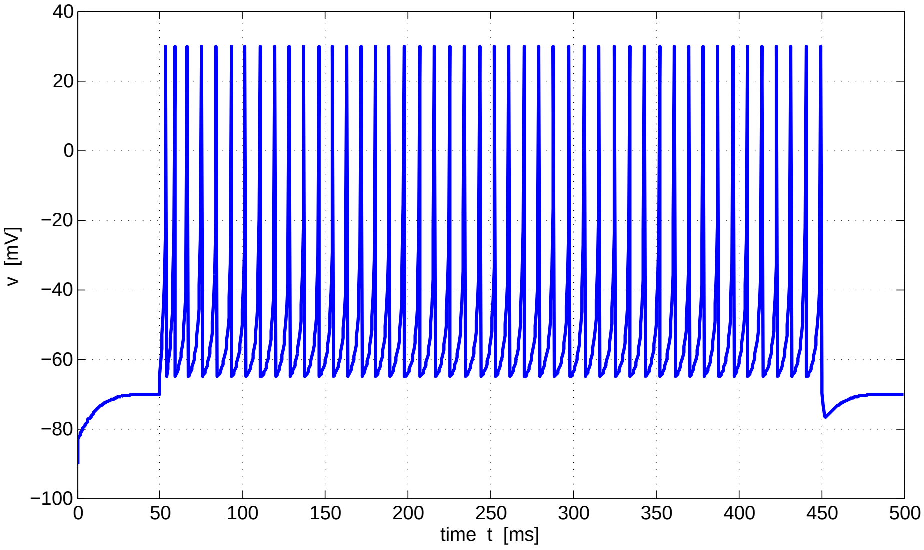


Figure 4.5: Fast spiking pattern

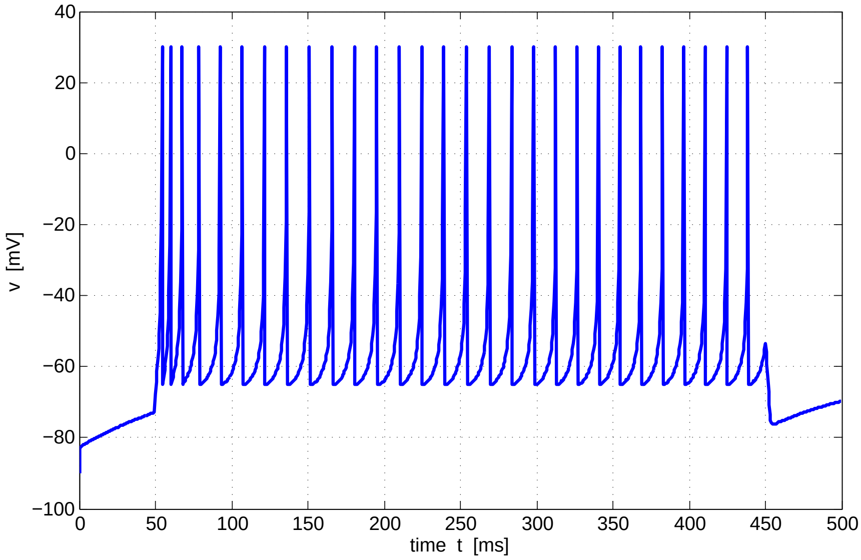


Figure 4.6: Low threshold spiking firing pattern

4.2 Piece Wise Linear Implementations of Izhikevich neuron model

There are several hardware implementations of the Izhikevich neuron model. As noted from the equation of Izhikevich, most area and power consumption results from the square term in the equation. The implementations addressing this neuron model are interested in approximating the equations to be more hardware friendly.

The common hardware approximation of Izhikevich neuron model is the PWL (Piece Wise Linear) implementation as presented in [29]. The PWL implementation of the model is a multiplier-less implementation. The implementation approximates the square function in the model and replaces it with a comparison or absolute value operations that are far less expensive than the exact square function. This approximation simplifies the hardware needed for the implementation since the hardware of a multiplier is much greater than the hardware of shift and add operations. As a result, the implementation of a network containing thousands of neurons is doable on the FPGA board.

The paper [29] discusses three types of Piece Wise Linear implementations:

1. Second-Order Piece-wise Linear Model: 2PWL model approximates the square operation of the original model with two crossed linear lines as shown in figure 4.7, the approximation is formulated as follows:

$$\begin{cases} \frac{dv}{dt} = K_1|v+62.5| - K_2 - u + I \\ \frac{du}{dt} = a(bv-u) \end{cases} \tag{4.4}$$

   In this model, there are two parameters K_1, K_2 that can be tuned to get a near exact behaviors, so they provide two degrees of freedom.

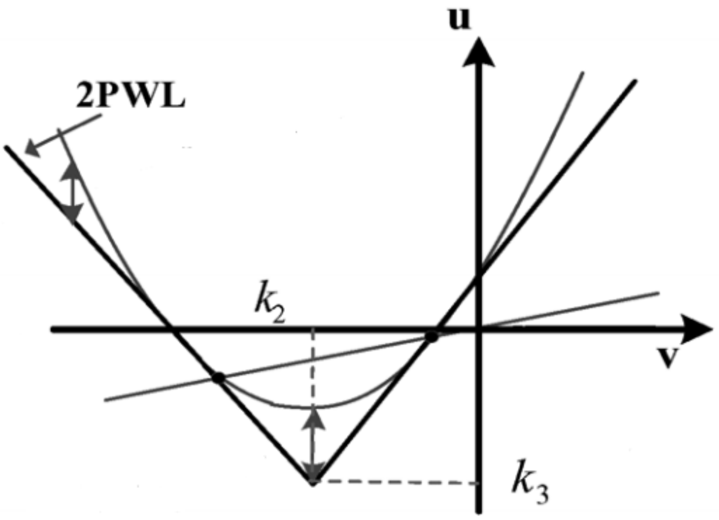


Figure 4.7: Second-Order Piece-wise Linear approximation

2. Third-Order Piece-wise Linear model: 3PWL model approximates the square operation of the original model with three crossed linear lines as shown in figure 4.8. The 3PWL model approximation is formulated as follows:

$$\begin{cases} \frac{dv}{dt} = K_1(|v+62.5+K_2|+|v+62.5-K_2|) - K_3K_2K_1 - u + I \\ \frac{du}{dt} = a(bv-u) \end{cases} \tag{4.5}$$

   This approximation has three tunable parameters K_1, K_2, *and* K_3. These parameters provide three degrees of freedom to the model to be able to achieve a near behavior to the original model. The 3PWL model is more expensive than the 2PWL model, but it is closer to the original model.

3. Fourth-Order Piece-wise Linear Model: The last model proposed in this paper and the closest one to the original model. It approximates the original model by four intersecting linear lines as shown in figure 4.9. It is formulated as follows:

$$\begin{cases} \frac{dv}{dt} = K_2(|v+62.5+K_3|+|v+62.5-K_3|) - K_1|v+62.5| - 4K_3K_2 - u + I \\ \frac{du}{dt} = a(bv-u) \end{cases} \tag{4.6}$$

   Similar to the 3PWL model, the 4PWL model has three degrees of freedom provided by the three parameters K_3, K_2, K_1. It is the most complex model in hardware implementation, yet it is the closest one to the original behavior.

In all PWL model approximations the coefficients are chosen so that they simplify the hardware implementation and achieve a closer behavior to the original model. These coefficients are chosen to be implemented as a shift and add operations.

The proposed hardware implementation in this thesis is based on the concept of approximate multipliers. The approximate multipliers are discussed in deep details in a

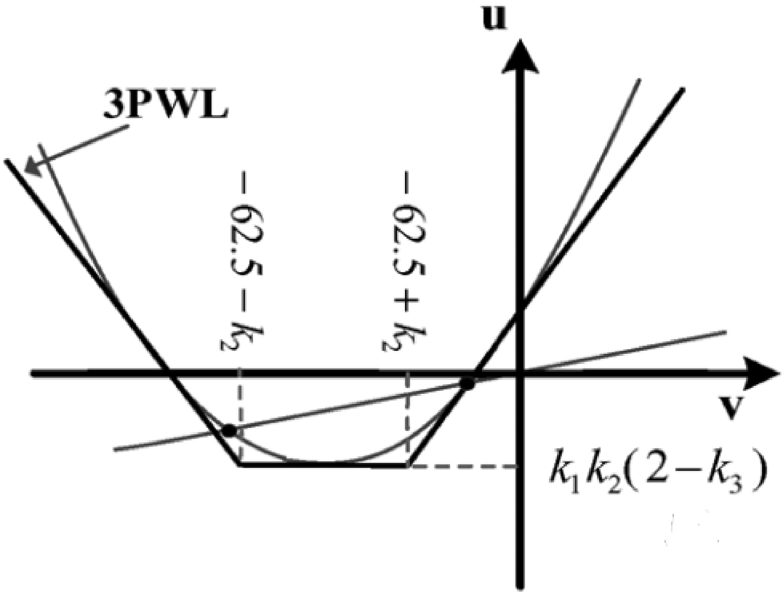


Figure 4.8: Third-Order Piece-wise Linear approximation

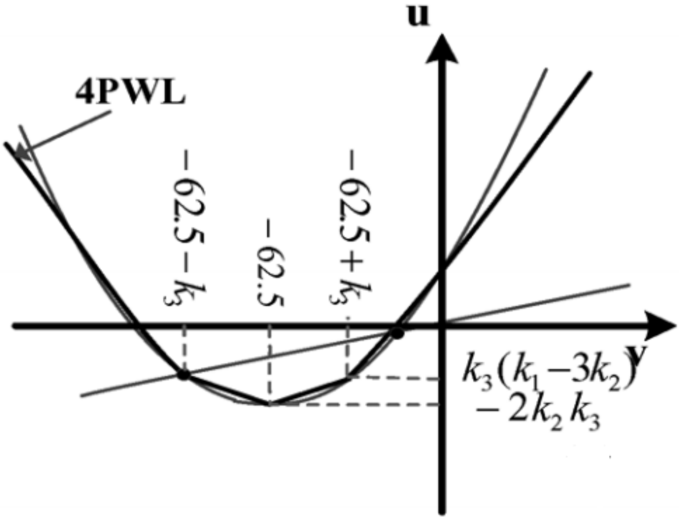


Figure 4.9: Fourth-Order Piece-wise Linear approximation

previous chapter. The approximate multipliers are hardware friendly that consume less area and power than the exact multipliers. The approximate models have to produce the same firing patterns as the original model and they also have to produce the same behavior when integrated into a network of many spiking neurons.

Both approximate based Izhikevich neuron model and PWL neuron models are simulated and compared with each other to discover the key strengths and weaknesses in each of them. Error metrics are evaluated for both models and a figure of merit is introduced to judge the effectiveness of both models. The network behavior is also simulated and the spiking patterns for them are reproduced and compared to the original model.

In the following sections, the detailed implementation of the approximate multiplier based Izhikevich neuron model is discussed and its simulation results are introduced as well. Then a comparison between the approximate multiplier based Izhikevich model and the PWL implementations is held.

4.3 Approximate Multiplier Based implementation of Izhikevich neuron model

Multiplication is the most hardware costly operation in the Izhikevich neuron model. Unlike the exact multipliers, the approximate multipliers consume less area and power and introduce some errors to the original model. In this work, the use of approximate multipliers in the hardware implementation of the Izhikevich neuron model is studied. There are several approximate multipliers in the literature and they can be used for implementing the square term in the Izhikevich neuron model. What gives an advantage to an approximate multiplier over another is its ability to reproduce the same spiking patterns of the original model with minimum errors. To choose a certain approximate multiplier to implement the Izhikevich model, simulations are held to discover the errors introduced by each approximate multiplier.

U and v variables in Izhikevich equations are plotted against each other in the original model and the approximation models to see how far these models away from the original one. U and v equations are plotted in the equilibrium state so $\frac{du}{dt} = \frac{dv}{dt} = 0$. As a result, their equations map to the following equations :

$$\begin{cases} u_1 = 0.04v^2 + 5v + 140 \\ u_2 = bv \end{cases} \tag{4.7}$$

4.3.1 Truncated Multiplier Implementation

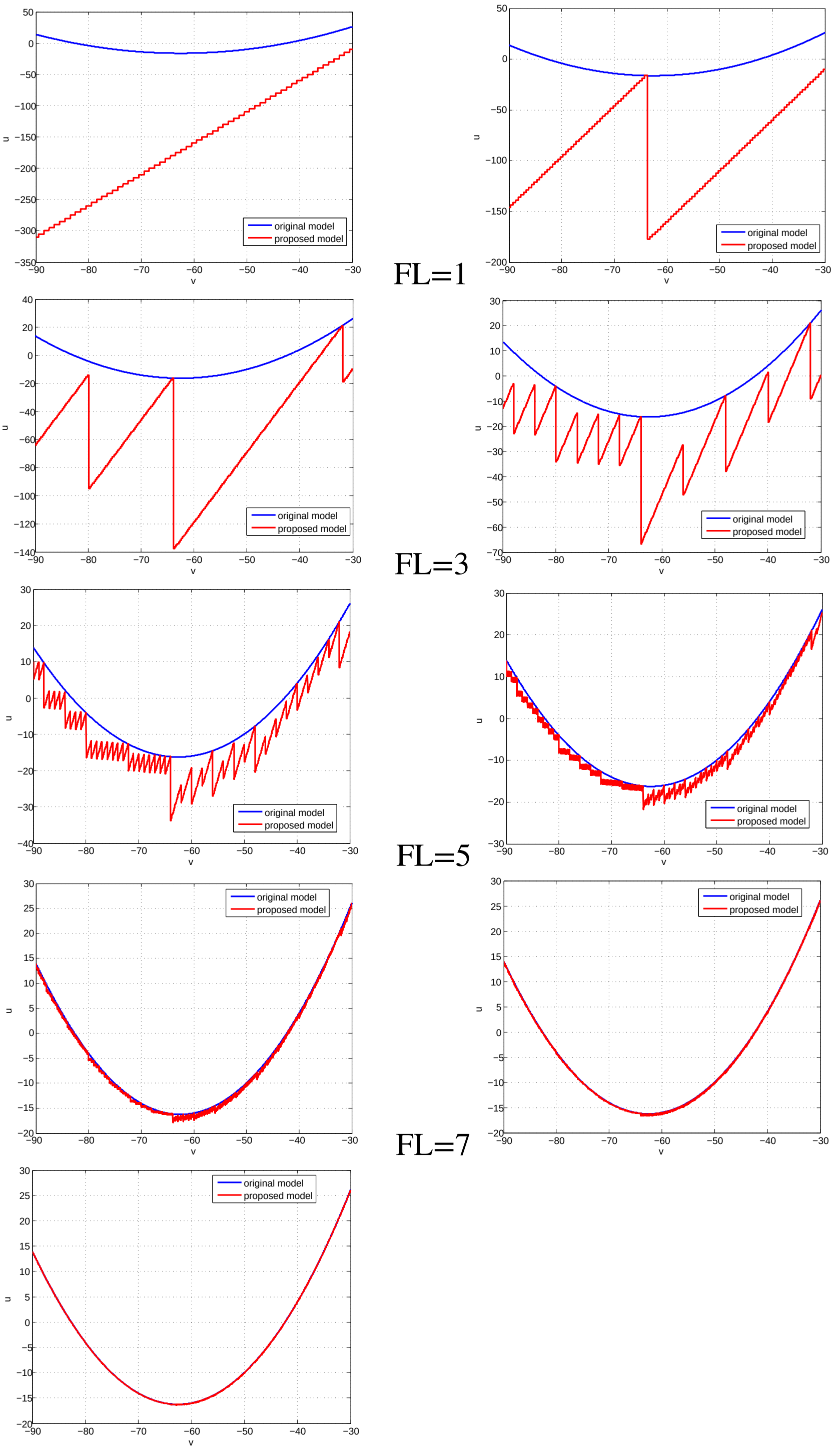
The integer length and fraction length of the word used in that implementation are chosen so that they produce the least errors while having the least possible value in order not to complicate the hardware implementation. The integer part has to be 8 to be able to hold the maximum value of u and v (-90). The trade-off here is to choose the fraction length. As the fraction length decreases, the total word-length decreases and the hardware units are simpler. To choose the least possible fraction length, Matlab simulation is used to calculate the error between the approximate multiplier implementation with different fraction lengths and compare it with the original model in the u-v plot. Figure 4.10 shows how the fraction length affects the behavior of the u-v plot. As the fraction length increases, the approximate multiplier based model becomes very close to the original behavior in the u-v plot. The u-v plot in the equilibrium state gives an indication of what will happen in the dynamic behavior. Figure 4.11 also presents the maximum error introduced by implementing the model using truncated multiplier versus different fraction lengths. It also shows that a fraction length of 8 is a good choice for the fraction length since it produces zero error in the u-v plot.

It is still needed to show the dynamic behavior of the neuron model and how it is affected by the fraction length variations. Again, the model is simulated in its dynamic state and the different responses of the model for different firing patterns are shown in figures 4.12, 4.13. In figure 4.12, the model is simulated with fraction length 4. It is very clear that this case is far away from the original model. The approximate model does not produce any of the firing patterns except for the Low Threshold Spiking mode. Even in the case of low threshold spiking, the model is not accurate in timing and does not follow the original model. In figure 4.13, the firing patterns are improved in all cases, and now the approximate multiplier based model follows the original model except for some slight errors in the timing between them. In the next subsections, the same analysis is done for the remaining types of approximate multipliers.

4.3.2 Error Tolerant Multiplier Approximation

In this model, there is a tunable parameter in the multiplier which is the non-multiplication part(NMB). The word length and fraction length in this model are fixed and chosen to be WL=16, and FL=8 to be consistent with the truncated multiplier model. The task now is to see the effect of varying the non-multiplication part on the equilibrium state, i.e., u-v plot and on the dynamic behavior, i.e., the spiking patterns. It is worth to note that increasing the non-multiplication part simplifies the hardware implementation, however, it reduces the output quality.

Figure 4.14 shows how the u-v plot is affected by varying the non-multiplication part. The best result occurs when the non-multiplication part is 0 which means exact multiplication. As the non-multiplication part increases the results get worse. A near-optimum result is obtained by choosing the non-multiplication part to be 4, but it still has some errors. Figure 4.15 demonstrates the same observation and shows that non-multiplication



FL=0

FL=1

FL=2

FL=3

FL=4

FL=5

FL=6

FL=7

FL=8

Figure 4.10: U-V plot for the truanted multiplier implementation for different fraction lengths, FL: Fraction Length

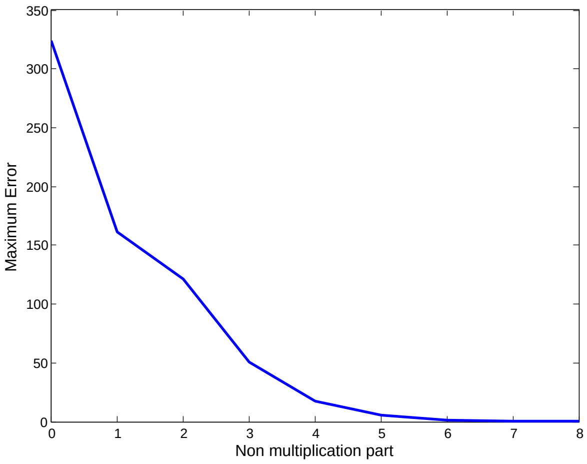


Figure 4.11: Maximum error vs the fraction length in the truncated multiplier implementation

parts of 4 and 5 results in a maximum error of less than 1. Thus, it is chosen to perform the dynamic simulations using non-multiplication part of 4 and compare it with the non-multiplication part of 8.

The dynamic simulations presented in figures 4.16 and 4.17 show that the approximate model reproduces the spiking pattern in both cases, i.e., non-multiplication part =8 and non-multiplication part =4. Referring back to the results of the truncated multiplier, the model was not able to regenerate the spiking patterns when the fraction length is 4, but when fraction length is 8 it could regenerate the patterns with good accuracy. The error-tolerant multiplier regenerates the spiking patterns when the non-multiplication part = 8 but with a very bad accuracy. And when the non-multiplication part = 4 the patterns' accuracy is good in most cases. There is still a slight shift between the original model and the approximated model at higher values of time as in the case of Intrinsically Bursting, Fast spiking, and Low Threshold Spiking in figure 4.17.

4.3.3 Broken Array Multiplier Approximation

The last approximate multiplier is the Broken Array multiplier. In this approximation, fixed word length and fraction length are used. They are chosen the same as the previous ones WL=16 and FL=8. This multiplier has two degrees of freedom to tune for better results, the vertical break level(VBL) and the horizontal break level(HBL). As known from the illustration of the broken array multiplier, the VBL does not degrade the output quality very much. The VBL can increase very much before it has a significant effect on the results. This is not the case with HBL, every single increase in the HBL affects the output quality. In the simulations, the results have fewer errors in case of increasing the VBL and it gets much worse as the HBL increases. As the WL is chosen to be 16, then the multiplication result needs 32bits to be accurate. The FL is 8, so the first 16 bits are

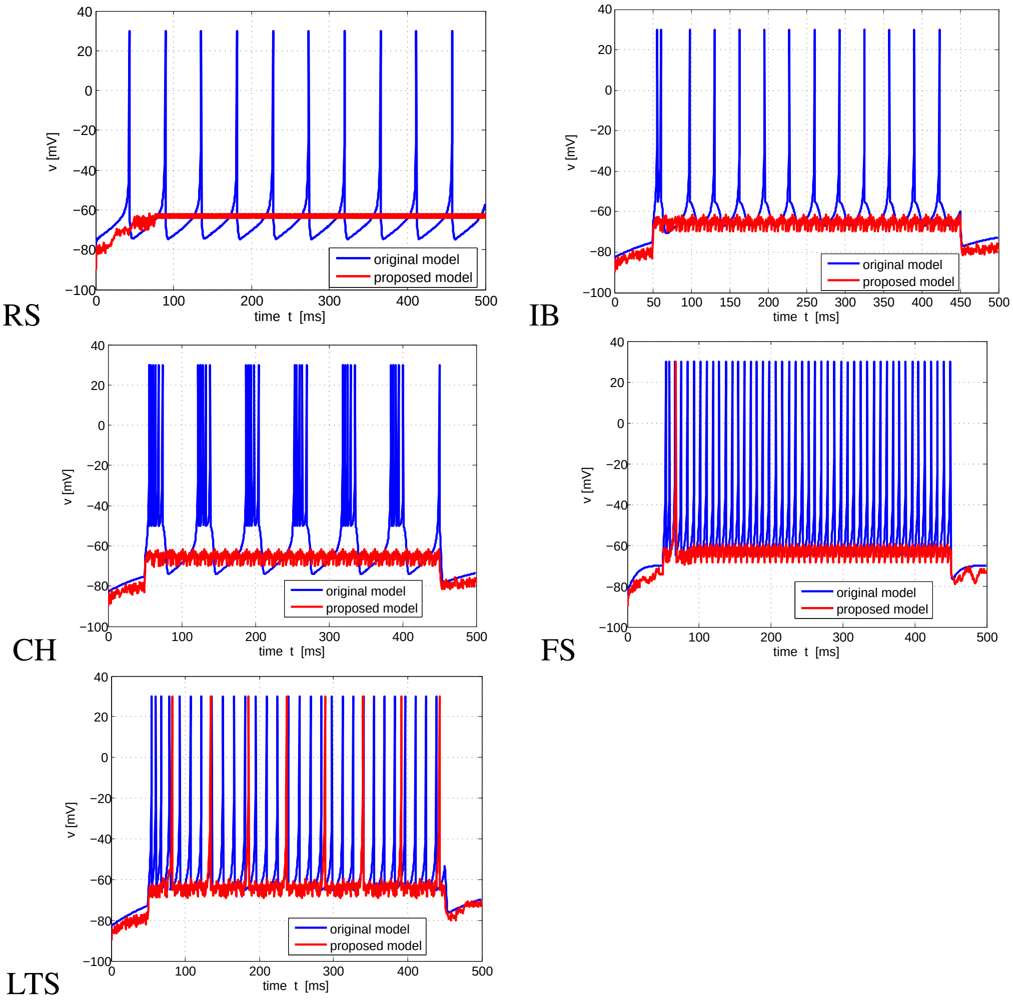


Figure 4.12: Different spiking patterns reproduced using the truncated multiplier approximation with fraction length = 4, RS: Regular Spiking, IB: Intrinsically Bursting, CH: Chattering, FS: Fast Spiking, LTS: Low Threshold Spiking

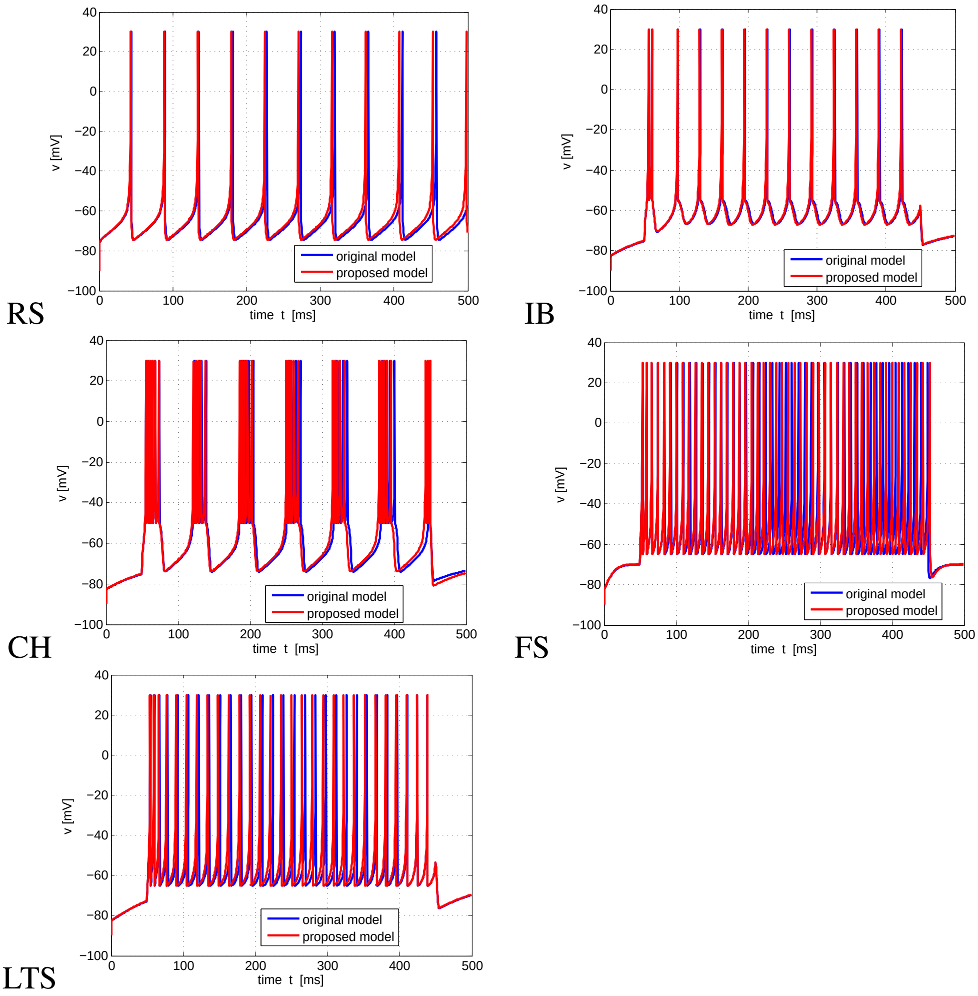
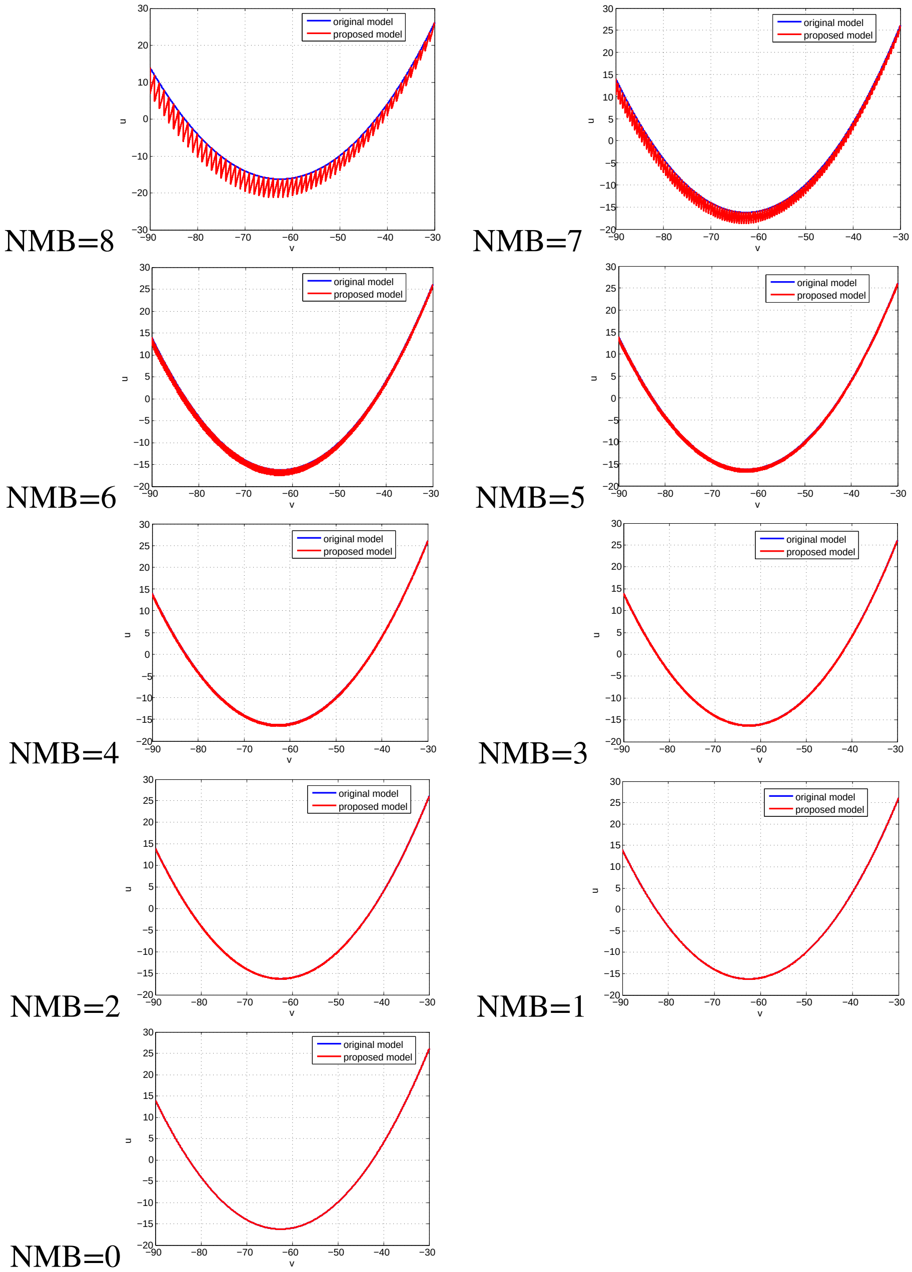


Figure 4.13: Different spiking patterns reproduced using the truncated multiplier approximation with fraction length = 8, RS: Regular Spiking, IB: Intrinsically Bursting, CH: Chattering, FS: Fast Spiking, LTS: Low Threshold Spiking


NMB=8 NMB=7

NMB=6 NMB=5

NMB=4 NMB=3

NMB=2 NMB=1

NMB=0

Figure 4.14: U-V plot for the error tolerant multiplier implementation for different non multiplication parts, NMB: Non Multiplication Part

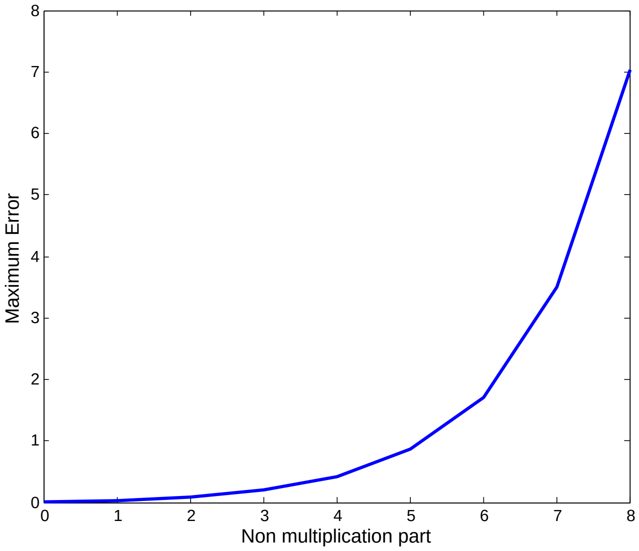


Figure 4.15: Maximum error vs Non multiplication part in the error tolerant multiplier implementation

the fraction part. The VBL can be chosen to be 16 so that it affects only the fraction part. To choose the HBL, it is needed to study its effect very well. To do that, the HBL is tuned and simulation results are analyzed to get the best compromise between the HBL and VBL. The simulations are held on the equilibrium state as well as the dynamic behavior. In figures 4.18 and 4.19, the different results of the U-V plot are shown while varying the HBL, it is noticed that as the HBL approaches 0 the results are better. As the HBL increases, the U-V plot experiences more errors. Figure 4.20 shows that the lower values of the HBL have fewer error values. The error is less than 5 in the case of HBL of 4. As the HBL increases, the error increases as well. The dynamic simulations are performed using HBL=9 and HBL= 4, then their results are compared.

The dynamic simulations in figures 4.21 and 4.22 show that the approximate model reproduces the spiking pattern in both cases, i.e., horizontal break level = 9 and horizontal break level = 4. The broken array multiplier regenerates the spiking patterns when the horizontal break level = 9, but with a very bad accuracy. All the patterns have a noticeable error in the timing of the spikes. When the horizontal break level = 4 the patterns' accuracy is good in most cases with a slight shift between the original model and the approximate model at higher values of time. This error is seen in the case of Intrinsically Bursting, Fast spiking, and Low Threshold Spiking in figure 4.22.

4.4 Proposed approximate multiplier based Izhikevich model

From the previously discussed results, the truncated multiplier is the best of them in terms of the accuracy results and the hardware simplicity. Thus, it is chosen to implement

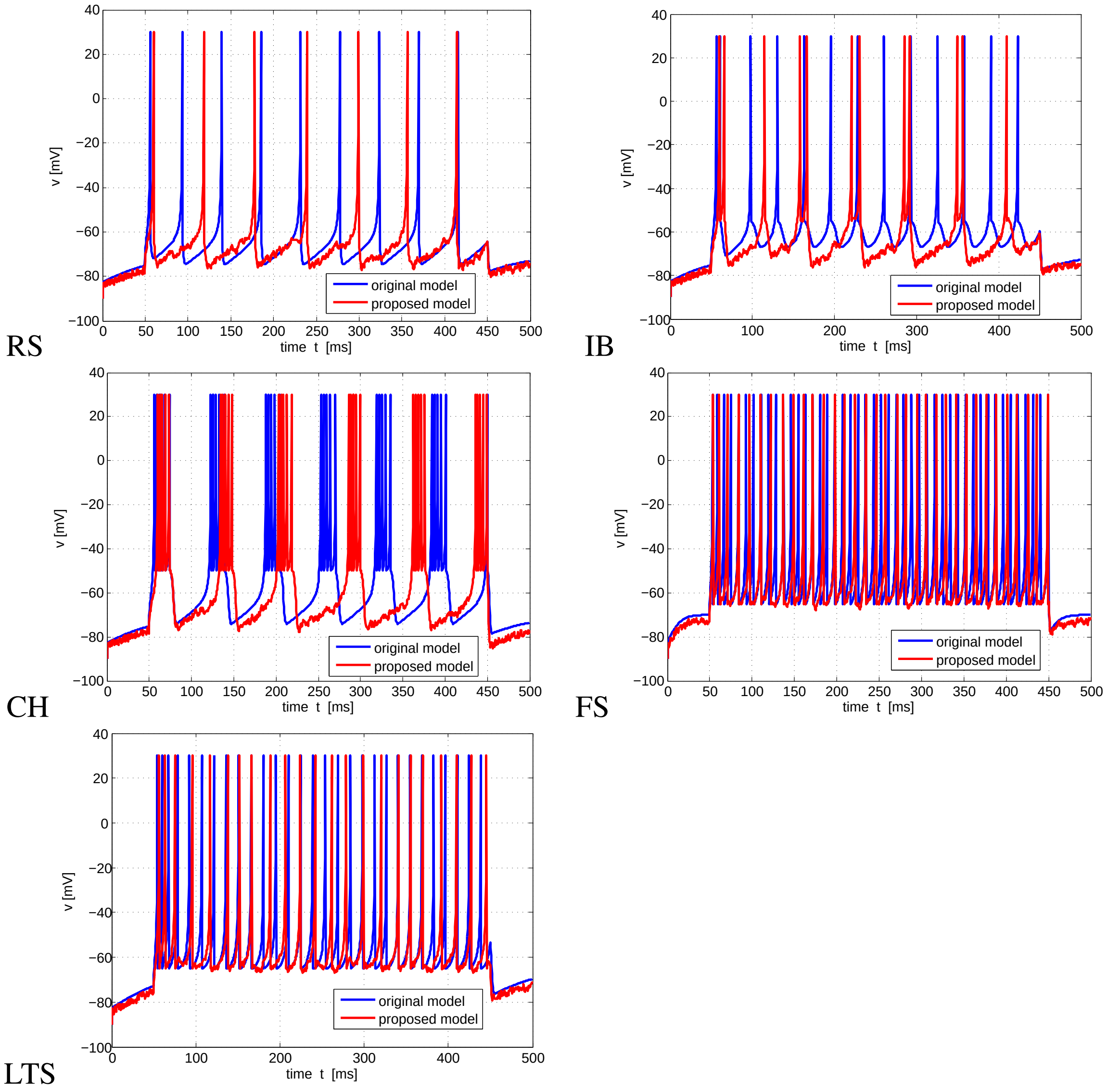


Figure 4.16: Different spiking patterns reproduced using the error tolerant multiplier approximation with non multiplication part = 8, RS: Regular Spiking, IB: Intrinsically Bursting, CH: Chattering, FS: Fast Spiking, LTS: Low Threshold Spiking

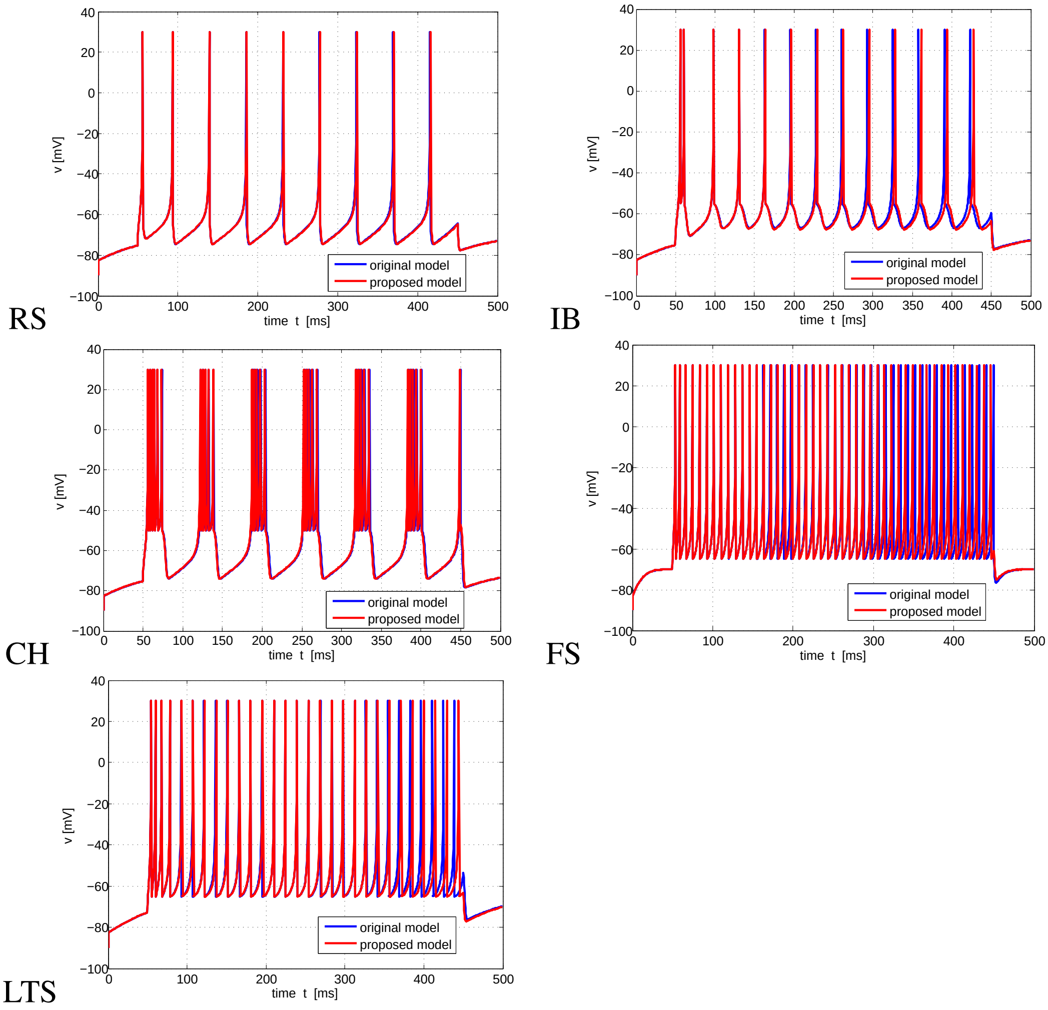
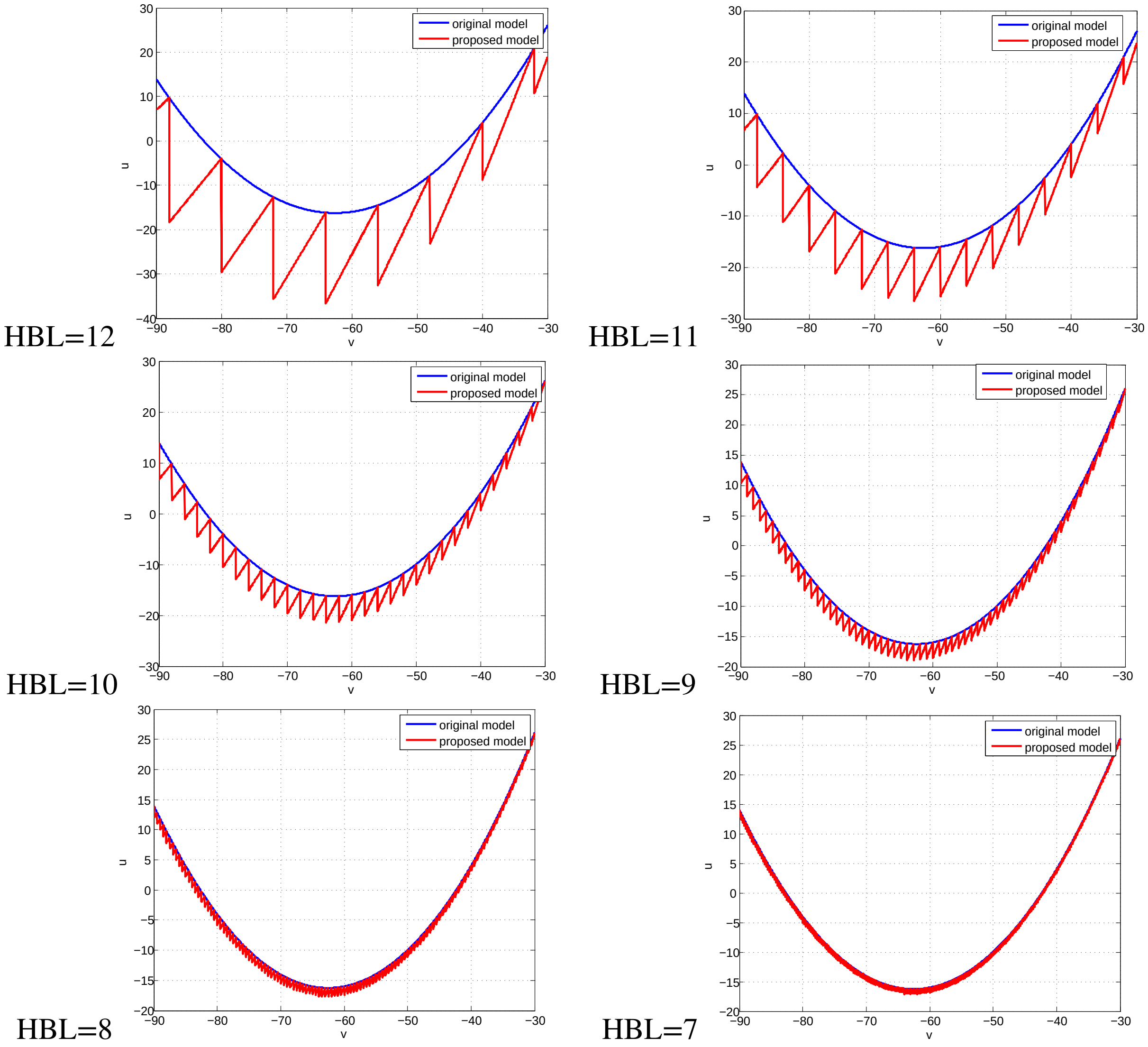


Figure 4.17: Different spiking patterns reproduced using the error tolerant multiplier approximation with non multiplication part = 8, RS: Regular Spiking, IB: Intrinsically Bursting, CH: Chattering, FS: Fast Spiking, LTS: Low Threshold Spiking


HBL=12

HBL=11

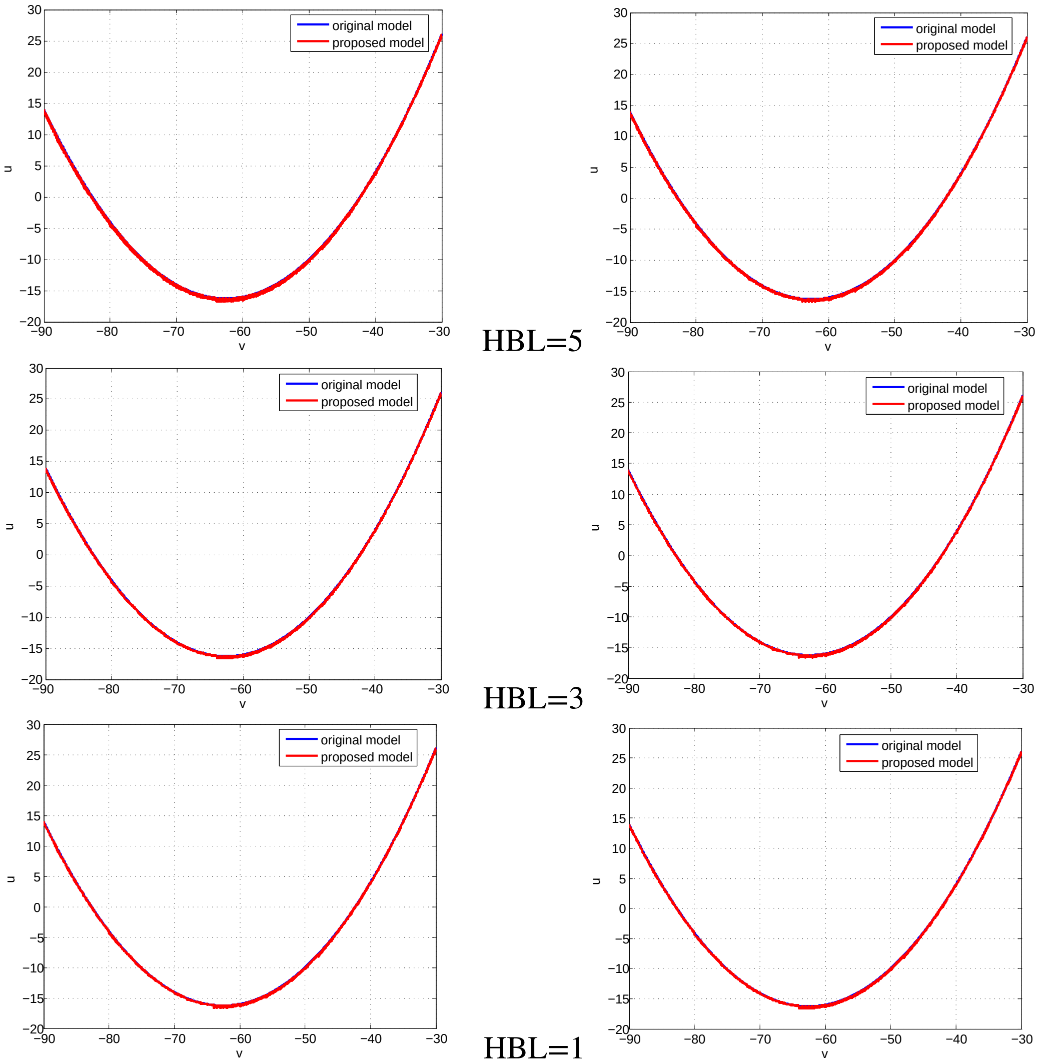
HBL=10

HBL=9

HBL=8

HBL=7

Figure 4.18: U-V plot for the broken array multiplier implementation for different horizontal break level values from 12 to 7, HBL: Horizontal Break Level



HBL=6 HBL=5

HBL=4 HBL=3

HBL=2 HBL=1

Figure 4.19: U-V plot for the broken array multiplier implementation for different horizontal break level values from 6 to 1, HBL: Horizontal Break Level

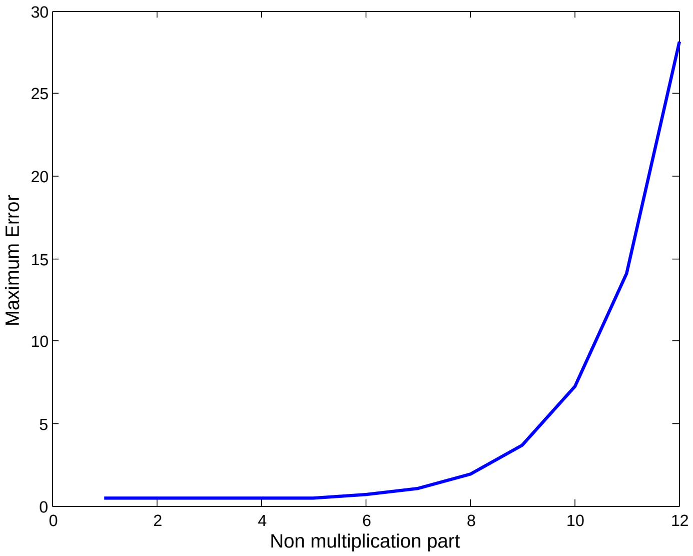


Figure 4.20: Maximum error vs Horizontal break level in the broken array multiplier implementation

the approximate multiplier based Izhikevich neuron model using the truncated multiplier. The results of the approximate model using the truncated multiplier are compared to the piece-wise linear implementation results. The comparison is based on the error analysis in the next section, the network behavior, and the hardware cost.

4.5 Error Analysis

To investigate the accuracy of any model implementing the Izhikevich neuron model, many error metrics should be calculated for the approximate model. The calculated errors are: ERRt, Normalized Root Mean Square Deviation (NRMSD), and Mean Relative Error (MRE). These errors indicate how much the model is close to the original model. Most of the errors depend on the timing of the reproduced spiking, since the timing is the most important factor in the spiking neural networks. The proposed model that implements the Izhikevich neuron model using the truncated multiplier is simulated using different word lengths and fraction lengths and all the errors are calculated for them. The PWL4 model is the best implementation of all PWL models, so it is used for comparison with our model. All errors are calculated for it to provide a fair comparison between the PWL model and the approximate multiplier based model. In the following subsections, the different errors are illustrated and then the results for both the approximate multiplier based model and PWL model are listed.

4.5.1 ERRt

ERRt measures the amount of time shift between the original model and the proposed model. It is calculated by once synchronizing the spikes of the two models and then

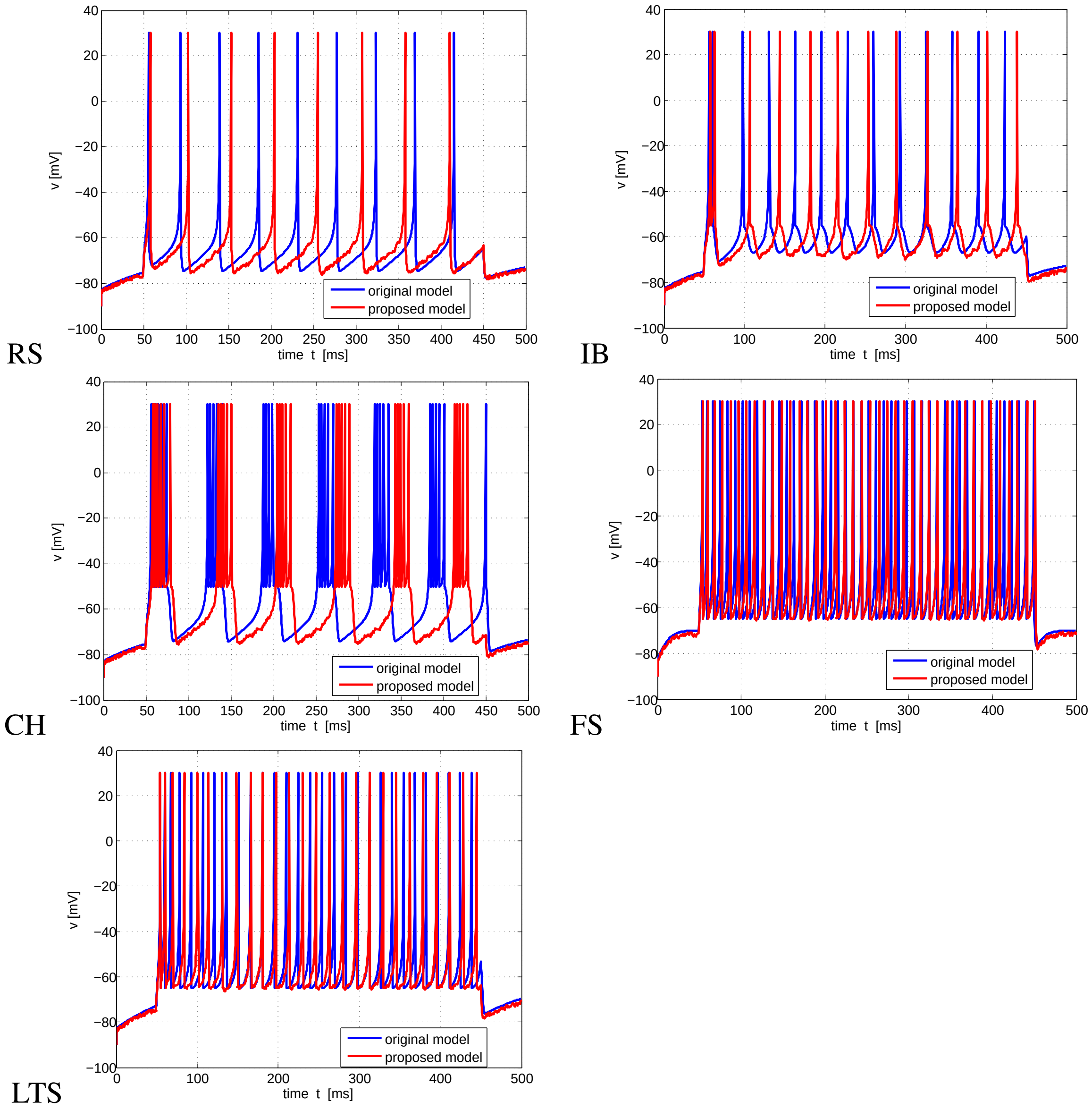


Figure 4.21: Different spiking patterns reproduced using the broken array multiplier approximation with horizontal break level=9, RS: Regular Spiking, IB: Intrinsically Bursting, CH: Chattering, FS: Fast Spiking, LTS: Low Threshold Spiking

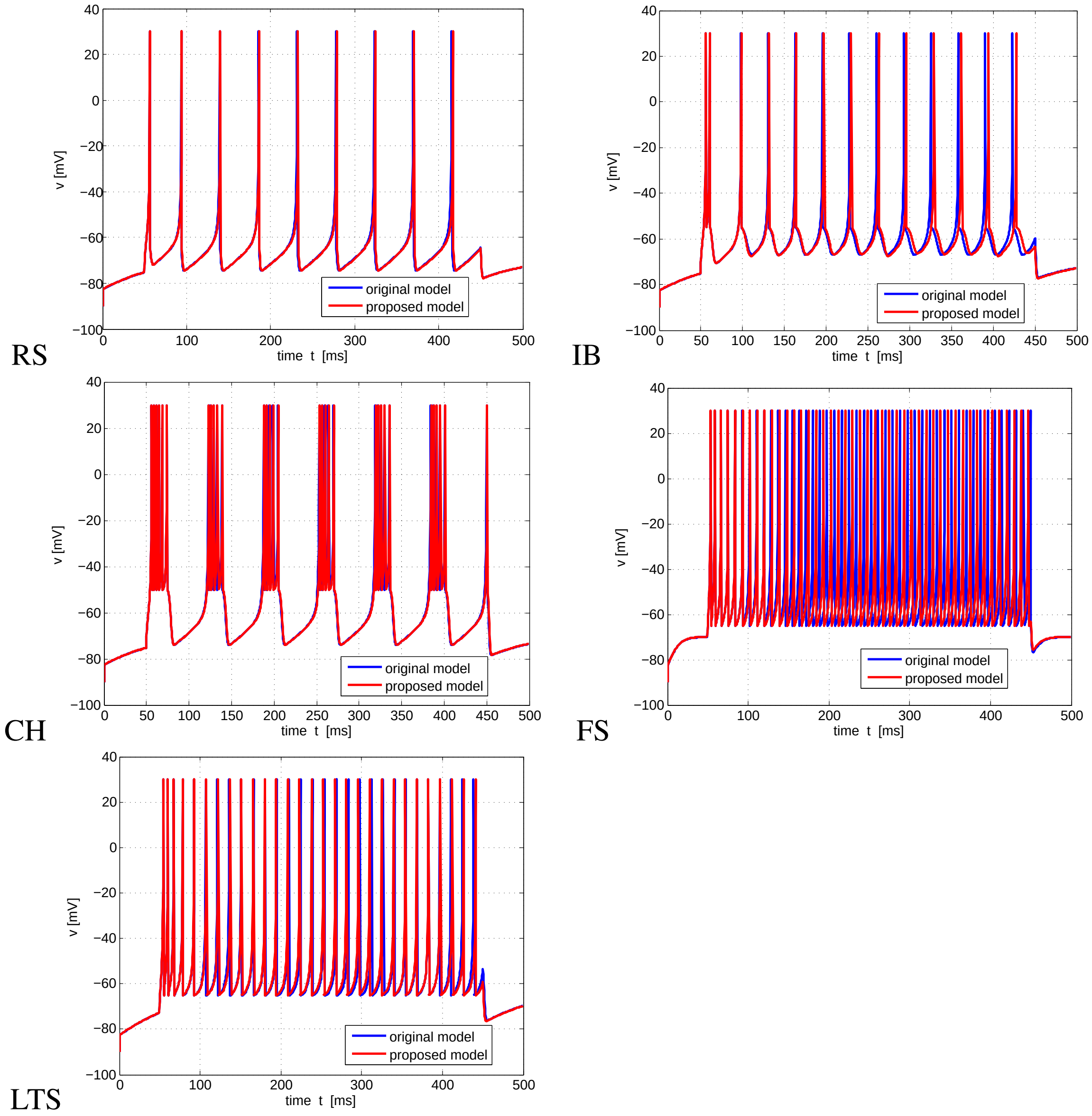


Figure 4.22: Different spiking patterns reproduced using broken array multiplier approximation with horizontal break level =4, RS: Regular Spiking, IB: Intrinsically Bursting, CH: Chattering, FS: Fast Spiking, LTS: Low Threshold Spiking

measures the difference in time between the next two spikes as shown in 4.23. The error is formulated as

$$ERRt = |\frac{\triangle t_p - \triangle t_o}{\triangle t_o}| * 100 \tag{4.8}$$

$$\triangle t = t_{spike2} - t_{spike1} \tag{4.9}$$

where $\triangle t_p$ and $\triangle t_o$are the time difference between two spikes in the proposed model and original model, respectively.

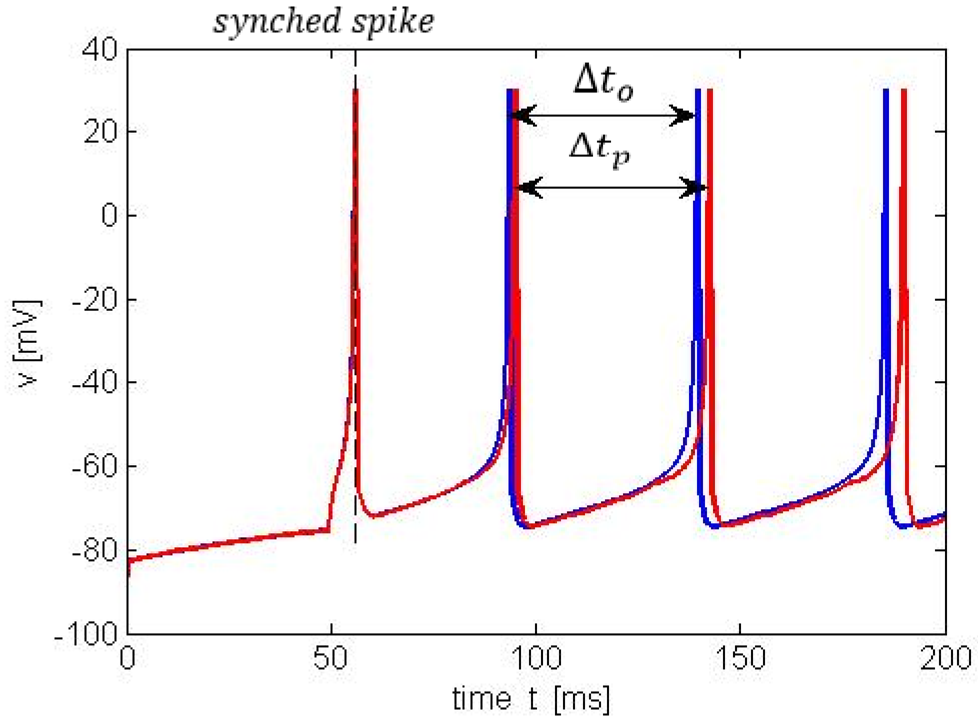


Figure 4.23: ERRt calculation as the time difference between the spikes of the proposed model and the original model

4.5.2 NRMSD

NRMSD is used to measure the difference in voltage values between the original and proposed model and is given by:

$$RMSD = \sqrt{\frac{\sum_{i=1}^{n}(vp(n) - vo(n))^2}{n}} \tag{4.10}$$

NRMSD is calculated as

$$NRMSD = \frac{RMSD}{v_{max} - v_{min}} \tag{4.11}$$

where v_p is the value of the voltage of the proposed model and v_o is the value of the voltage of the original model, v_{max} and v_{min} are the maximum and the minimum voltage values from the original model in the range of calculating RMSD. For better calculation of RMSD, the two models are synchronized, then a window is taken before and after the spike at half the distance between two spikes then the error is calculated from 4.10 [30].

4.5.3 MRE

The mean relative error(MRE) calculates the difference in the spike time between the original model and the proposed model. For fair calculation of that error, a window of 1000 ms in time is taken and the error is averaged over it. The MRE is formulated as:

$$MRE\% = \frac{|\frac{\triangle t_1}{to_1}| + |\frac{\triangle t_2}{to_2}| + ...}{N} * 100 \tag{4.12}$$

where $\triangle t_i$ is the time difference between the i_{th}spike time in the original model and the proposed model, and $to_i$is the i_{th}spike time of the original model [29].

4.5.4 PWL4 model Error Results

The following tables 4.1, 4.2, 4.3, and 4.4 list all the error values ERRt, NRMSD, MRE, for the PWL4 implementation for different word length. The errors are calculated in most common spiking patterns, regular spiking, tonic spiking, fast-spiking, low threshold spiking. Then the average of each type of error is calculated and the average of all errors is calculated as well.

Table 4.1: PWL4 Error results using WL=17

PWL4, WL=17	ERRt	NRMSD	MRE
Regular Spiking	1.09%	0.90%	0.52%
Tonic Spiking	3.07%	19.62%	3.17%
Fast Spiking	5.88%	0.83%	1.19%
Low Threshold Spiking	3.57%	3.24%	0.37%
Average of each error	3.40%	6.15%	1.31%
Average of all errors	3.62%		

4.5.5 Approximate multiplier based model Error Results

The following tables 4.5, 4.6, and4.7 list the error values ERRt, NRMSD, MRE, for the approximate multiplier based implementation for different word length. The errors are calculated in most common spiking patterns, regular spiking, tonic spiking, fast-spiking, low threshold spiking. Then the average of each type of error is calculated and the average of all errors is calculated as well.

Table 4.2: PWL4 Error results using WL=16

PWL4, WL=16	ERRt	NRMSD	MRE
Regular Spiking	2.17%	1.13%	1.04%
Tonic Spiking	3.07%	19.63%	3.17%
Fast Spiking	5.88%	0.89%	0.60%
Low Threshold Spiking	3.33%	1.45%	0.38%
Average of each error	3.61%	5.78%	1.30%
Average of all errors	3.56%		

Table 4.3: PWL4 Error results using WL=15

PWL4, WL=15	ERRt	NRMSD	MRE
Regular Spiking	1.09%	0.70%	0.52%
Tonic Spiking	2.63%	19.62%	2.90%
Fast Spiking	5.56%	0.65%	1.35%
Low Threshold Spiking	3.57%	3.37%	0.91%
Average of each error	3.21%	6.09%	1.42%
Average of all errors	3.57%		

Table 4.4: PWL4 Error results using WL=14

PWL4, WL=14	ERRt	NRMSD	MRE
Regular Spiking	1.09%	0.73%	0.52%
Tonic Spiking	3.07%	19.63%	3.17%
Fast Spiking	5.88%	0.54%	1.16%
Low Threshold Spiking	3.57%	2.53%	1.62%
Average of each error	3.40%	5.86%	1.62%
Average of all errors	3.63%		

4.6 Hardware Implementation

In this the hardware implementation for both the PWL4 model and the approximate multiplier based model are discussed in detail and the area, power, and frequency results are listed. First, the differential equations are discretized using the Euler method and all constant multiplications are chosen to be powers of 2 so that they are performed as shift and add operations. The equations used are as follows:

$$v[n+1] = v[n] + dt(x.v^2[n] + 4v[n] + 109.375 - u[n] + I[n]) \tag{4.13}$$

$$u[n+1] = u[n] + dt(a(bv[n] - u[n])) \tag{4.14}$$

The constant x = 0:0400390625 and is evaluated as $\frac{1}{32} + \frac{1}{128} + \frac{1}{1024}$, 5:v [n] is evaluated as v[n] + 4.v [n], and dt is 0.5. To compare between the model and PWL4 model, both

Table 4.5: Truncated multiplier error results using WL=16 and FL=7

PWL4, WL=16, FL=7	ERRt	NRMSD	MRE
Regular Spiking	0%	0.14%	0%
Tonic Spiking	0%	0.43%	0%
Fast Spiking	5.56%	0.48%	0.52%
Low Threshold Spiking	3.33%	2.58%	0.34%
Average of each error	2.223%	0.91%	0.2150%
Average of all errors	1.115%		

Table 4.6: Truncated multiplier error results using WL=15, FL=6

PWL4, WL=15, FL=6	ERRt	NRMSD	MRE
Regular Spiking	3.26%	1.02%	2.22%
Tonic Spiking	4.12%	0.48%	1.33%
Fast Spiking	5.56%	1.54%	0.50%
Low Threshold Spiking	3.45%	2.82%	0.80%
Average of each error	4.098%	1.47%	1.2125%
Average of all errors	2.258%		

Table 4.7: Truncated multiplier error results using WL=14, FL=5

PWL4, WL=14, FL=5	ERRt	NRMSD	MRE
Regular Spiking	6.52%	1.18%	5.18%
Tonic Spiking	7.14%	14.73%	2.92%
Fast Spiking	11.76%	2.23%	3.77%
Low Threshold Spiking	13.33%	2.49%	2.96%
Average of each error	9.688%	5.16%	3.7075%
Average of all errors	6.184%		

models are implemented on the RTL level and synthesized on Zynq XC7Z020-1CLG484C FPGA. The area and power results are estimated using Vivado tool.

4.6.1 PWL4 hardware implementation results

Tables 4.8, 4.9, 4.10, and 4.11 shows the results of area, power, and frequency of PWL4 model.

Table 4.8: Hardware implementation results of PWL4 using WL=17

Dynamic power(W)	0.01
Total power(W)	0.13
Area(LuTs)	381
Frequency(MHz)	25
Word-Length	17

Table 4.9: Hardware implementation results of PWL4 using WL=16

Dynamic power(W)	0.009
Total power(W)	0.129
Area(LuTs)	347
Frequency(MHz)	25M
Word-Length	16

Table 4.10: Hardware implementation results of PWL4 using WL=15

Dynamic power(W)	0.008
Total power(W)	0.128
Area(LuTs)	324
Frequency(MHz)	25M
Word-Length	15

Table 4.11: Hardware implementation results of PWL4 using WL=15

Dynamic power(W)	0.008
Total power(W)	0.128
Area(LuTs)	286
Frequency(MHz)	25M
Word-Length	14

4.6.2 Approximate multiplier based hardware implementation results

Tables 4.12, 4.13, and 4.14 show the results of area, power and frequency of PWL4 model.

Table 4.12: **Hardware implementation results of truncated multiplier using WL=16, FL=7**

Dynamic power(W)	0.011
Total power(W)	0.131
Area(LuTs)	597
Frequency(MHz)	25M
Word Length	16

Table 4.13: **Hardware implementation results of truncated multiplier using WL=15, FL=6**

Dynamic power(W)	0.01
Total power(W)	0.13
Area(LuTs)	560
Frequency(MHz)	25M
Word Length	15

Table 4.14: **Hardware implementation results of truncated multiplier using WL=14 , FL=5**

Dynamic power(W)	0.009
Total power(W)	0.129
Area(LuTs)	457
Frequency(MHz)	25M
Word Length	14

4.7 Network Behavior of the proposed Approximate multiplier model

Another important metric to judge the proposed model is its ability to behave well and generate spikes in a network of connected neurons. To simulate the network behavior of the proposed model against the original model, a network of randomly connected 1000 neurons is simulated as in [28]. Motivated by the anatomy of a mammalian cortex, the ratio of excitatory to inhibitory neurons is chosen to be 4 to 1. The results of the firings are shown in the so-called raster diagram. The raster diagrams shown in 4.24 show that the network behavior of the proposed model is very close to the original model. Both models shown in the figure have the same behavior and both have the neurons fire with a rate of 5 Hz in the network. Another measure to test the accuracy in the network is the count of spikes that each neuron fires in a defined time interval. Some neurons are selected randomly from the previous raster diagram and their spikes count is evaluated and compared to the original model. 4.25 shows that a randomly selected neuron in the original model and the proposed model fire almost the same spikes count.

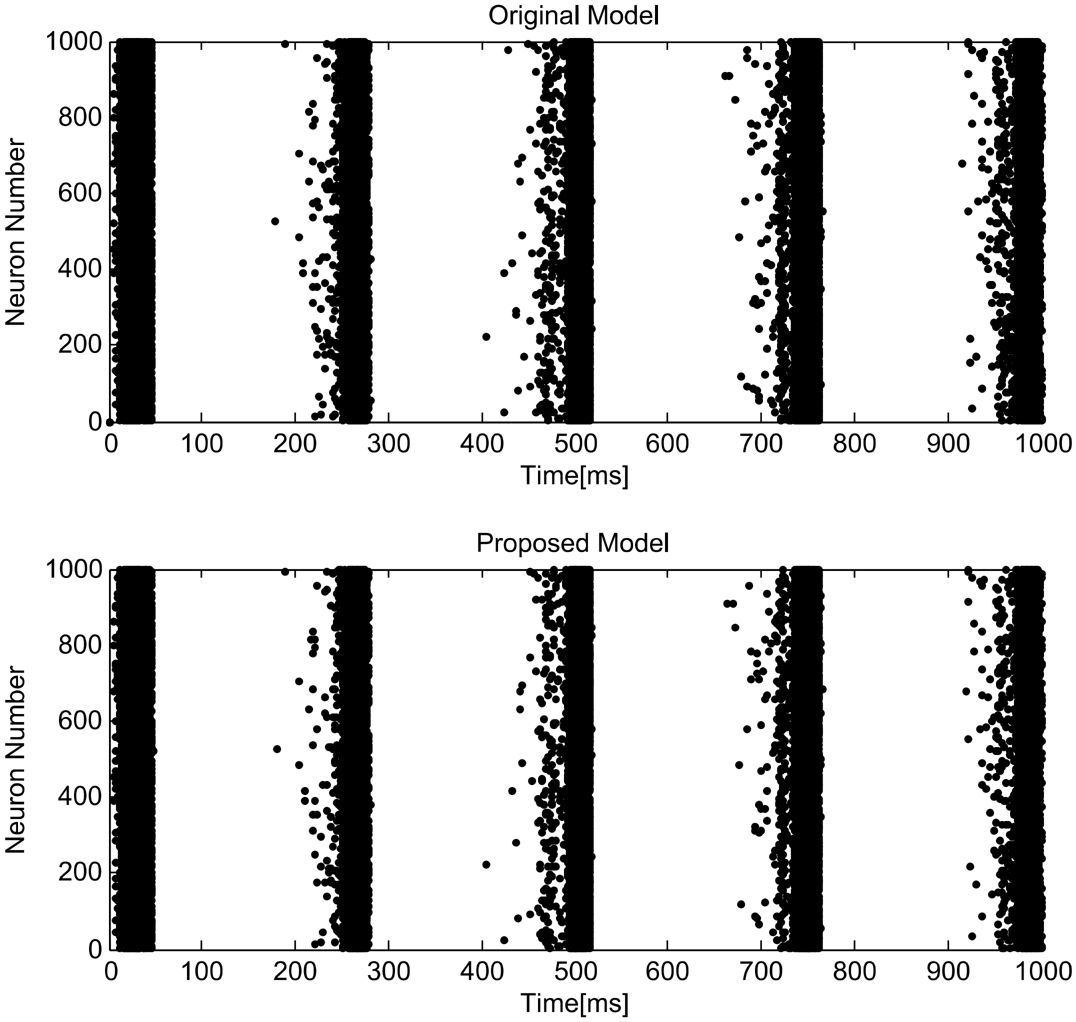


Figure 4.24: Raster diagram of a network of 1000 connected neurons modeled using the approximate multiplier based implementation compared to the original model output

From this discussion, the proposed implementation of the neuron using the approximate multiplier has a very close behavior to the original model.

4.8 Comparison between the PWL4 neuron model and Approximate Multiplier based model

After the above discussion, it is proved that the approximate multiplier based neuron model is a good implementation of the original model. It can reproduce the spiking pattern the same as the original model. It can generate the spikes with a near accurate time frame as shown in the errors section. Moreover, its behavior in the network in terms of the spikes frequency and the spike count of the neurons is close to the original model. The PWL4 implementation discussed in [29] proves that the PWL4 neuron model can also

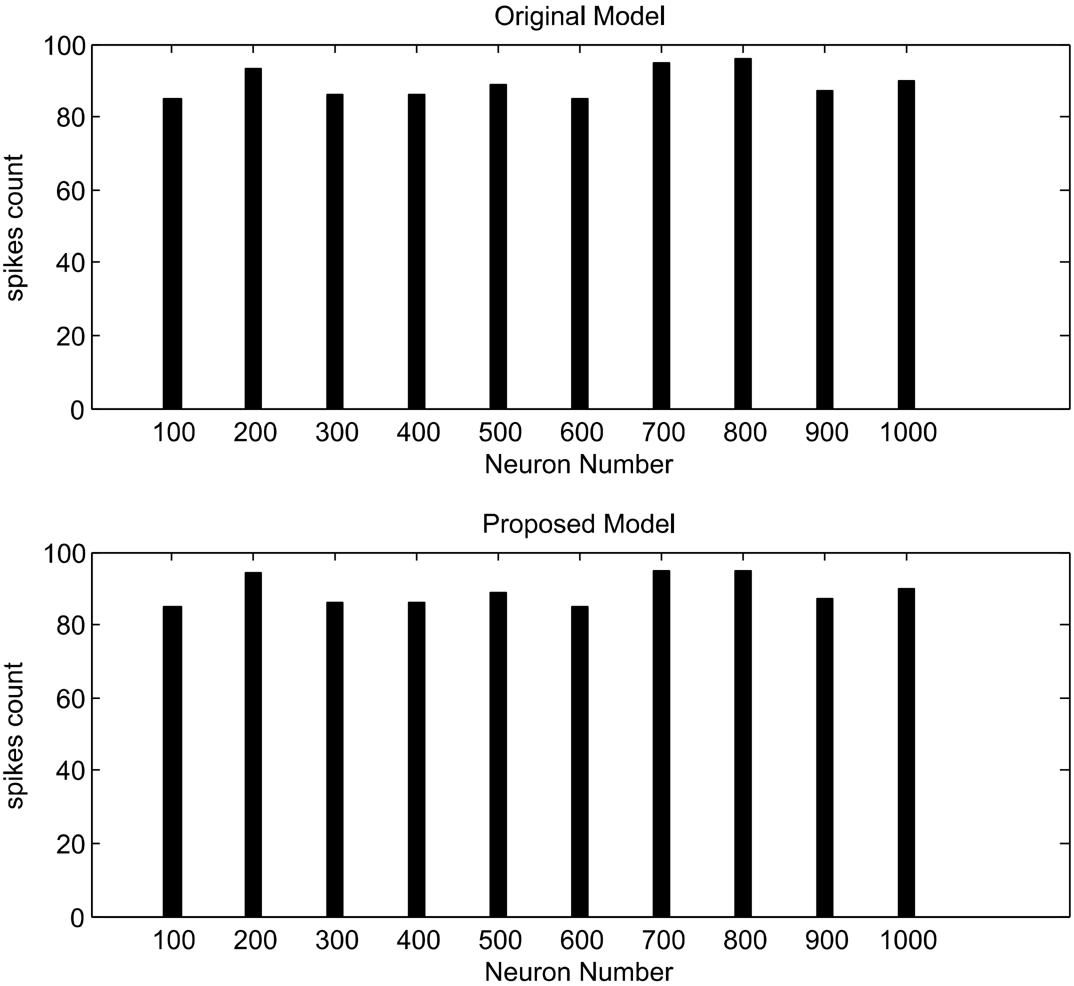


Figure 4.25: Spikes count of randomly selected neurons from the network for both the original model and the proposed model

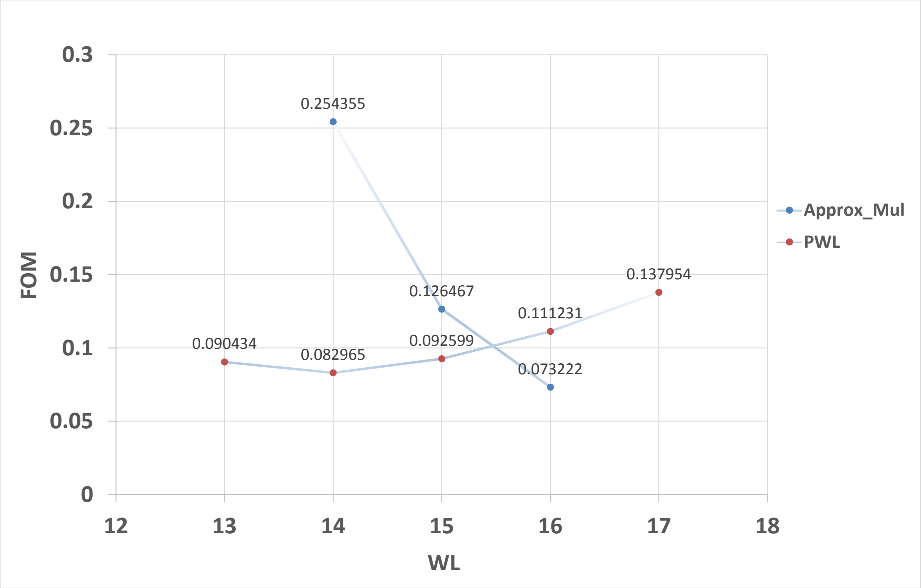


Figure 4.26: Figure of Merit (FoM)of both approximate multiplier based model and the PWL4 model Versus Word Length(WL)

be used for implementing the Izhikevich model. Both PWL4 and approximate multiplier based implementations are hardware friendly. To hold a fair comparison between both implementations, one should take into consideration all design aspects. The important measures in that comparison are The accuracy of the model, and this can be measured by the errors produced due to the approximations, the area ,and power consumption of the model. To group all these aspects in one measure, a Figure of Merit(FoM) is introduced. This is calculated to conclude the effect of accuracy in terms of errors, area, and power. The FoM is defined as follows:

$$FoM = Erros_{avg} * Area * Power \tag{4.15}$$

Where the $Errors_{avg}$ is the average of all errors calculated for the spiking patterns, Area is the number of used LUTs, and Power is the dynamic power consumed by the design in mw. It is better to have a small FoM value to make the best trade-off of accuracy, area, and power. FoM is evaluated for both PWL4 model and the approximate multiplier based model for different word length (WL) used in hardware implementation. Figure 4.26 shows that as the WL increases, The FoM increases as well and that the WL of 14 gives the best FoM in the case of the PWL4 model. A WL of 16 gives the best FoM in the case of an approximate multiplier based model.

4.9 Conclusion

The Izhikevich neuron model is a well-known model in the implementation of Spiking Neural Networks for its simplicity and its ability to reproduce the same spiking patterns of the biological neurons. The main issue with that model is the need for a square operation which is not a hardware friendly operation and consumes much area and power. This work proposes the use of the Approximate Multiplier in the implementation of the model. Three approximate multipliers are used to generate the model and the simulation results prove that the truncated multiplier is the best as its accuracy is the best of the other two approximate multipliers. Simulation results show that the approximate multiplier model is a good alternative to the well known PWL models in terms of accuracy, power, and area. The calculated Figure of Merit proves that the approximate multiplier based implementation is better than the PWL4 model using Word Length of 16, this implementation has better accuracy while maintaining low area and power.

Chapter 5

Future Work

The previous chapters discuss the hardware implementation of the artificial neural network and the utilization of approximate computing in the implementation. Moreover, the energy adaptive neural network(EANN)is introduced. This EANN has the ability to adapt its hardware components to the available energy budget on the fly. The dynamic reconfiguration adds advantage to the system that it can continue functioning for more time before it completely turns off.

The next work shall be directed to implement the idea of partial dynamic reconfiguration to the spiking neural networks. The reconfiguration shall be between the different neuron models. There is a variety of the neuron models, some of them are biologically plausible, but they are very complex. Some of them are biologically inspired, and they are much simpler. This variety shall provide many models for reconfiguration. The models shall be different in the accuracy and the ability to reproduce the spiking patterns.

References

[1] S. A. Kalogirou, “Applications of artificial neural-networks for energy systems,” *Applied energy*, vol. 67, no. 1-2, pp. 17–35, 2000.

[2] *Biological neuron*, https://www.ee.co.za/article/application-of-machine-learning-algorithms-in-boiler-plant-root-cause-analysis.html, Accessed : 2020-01-05.

[3] C. D. Schuman, T. E. Potok, R. M. Patton, J. D. Birdwell, M. E. Dean, G. S. Rose, and J. S. Plank, “A survey of neuromorphic computing and neural networks in hardware,” *arXiv preprint arXiv:1705.06963*, vol. abs/1705.06963, 2017.

[4] J. Misra and I. Saha, “Artificial neural networks in hardware: A survey of two decades of progress,” *Neurocomputing*, vol. 74, no. 1-3, pp. 239–255, 2010.

[5] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan, “AxNN: Energy-Efficient Neuromorphic Systems using Approximate Computing,” *International symposium on Low Power Electronics and Design*, pp. 27–32, 2014.

[6] Q. Zhang, T. Wang, Y. Tian, F. Yuan, and Q. Xu, “ApproxANN: An Approximate Computing Framework for Artificial Neural Network,” *Conference on Design, Automation and Test in Europe*, pp. 701–706, 2015.

[7] J. Kung, D. Kim, and S. Mukhopadhyay, “A power-aware digital feedforward neural network platform with backpropagation driven approximate synapses,” *International Symposium on Low Power Electronics and Design*, pp. 85–90, 2015.

[8] S. S. Sarwar, S. Venkataramani, A. Raghunathan, and K. Roy, “Multiplier-less Artificial Neurons Exploiting Error Resiliency for Energy-Efficient Neural Computing,” *Conference on Design, Automation and Test in Europe*, pp. 145–150, 2016.

[9] V. Mrazek, S. S. Sarwar, L. Sekanina, Z. Vasicek, and K. Roy, “Design of power-efficient approximate multipliers for approximate artificial neural networks,” *International Conference on Computer-Aided Design*, pp. 1–7, 2016.

[10] D. Kim, J. Kung, and S. Mukhopadhyay, “A Power-Aware Digital Multilayer Perceptron Accelerator with On-Chip Training based on Approximate Computing,” *IEEE Transactions on Emerging Topics in Computing*, vol. 5, no. 2, pp. 164–178, 2017.

[11] H. R. Mahdiani, A. Ahmadi, S. M. Fakhraie, and C. Lucas, "Bio-inspired imprecise computational blocks for efficient vlsi implementation of soft-computing applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 57, no. 4, pp. 850–862, 2009.

[12] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," *Internatioal Conference on VLSI Design*, pp. 346–351, 2011.

[13] K. Y. Kyaw, W. L. Goh, and K. S. Yeo, "Low-power high-speed multiplier for error-tolerant application," *International Conference of Electron Devices and Solid-State Circuits (EDSSC)*, pp. 1–4, 2010.

[14] N. Petra, D. De Caro, V. Garofalo, E. Napoli, and A. G. Strollo, "Truncated binary multipliers with variable correction and minimum mean square error," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 57, no. 6, pp. 1312–1325, 2009.

[15] F. Folowosele, T. J. Hamilton, and R. Etienne-Cummings, "Silicon modeling of the mihalaş–niebur neuron," *IEEE transactions on neural networks*, vol. 22, no. 12, pp. 1915–1927, 2011.

[16] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *The Journal of physiology*, vol. 117, no. 4, pp. 500–544, 1952.

[17] A. Borisyuk, "Morris–lecar model," in *Encyclopedia of Computational Neuroscience*, D. Jaeger and R. Jung, Eds. New York, NY: Springer New York, 2015, pp. 1758–1764.

[18] W. Gerstner, W. M. Kistler, R. Naud, and L. Paninski, *Neuronal dynamics: From single neurons to networks and models of cognition*. Cambridge University Press, 2014.

[19] E. J. Basham and D. W. Parent, "An analog circuit implementation of a quadratic integrate and fire neuron," *Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pp. 741–744, 2009.

[20] M. Hayati, M. Nouri, D. Abbott, and S. Haghiri, "Digital multiplierless realization of two-coupled biological hindmarsh–rose neuron model," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 63, no. 5, pp. 463–467, 2015.

[21] J. Lu, J. Yang, Y.-B. Kim, J. Ayers, and K. K. Kim, "Implementation of excitatory cmos neuron oscillator for robot motion control unit," *journal of semiconductor technology and science*, vol. 14, no. 4, pp. 383–390, 2014.

[22] F. Grassia, T. Levi, T. Kohno, and S. Saïghi, "Silicon neuron: Digital hardware implementation of the quartic model," *Artificial Life and Robotics*, vol. 19, no. 3, pp. 215–219, 2014.

[23] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning internal representations by error propagation,” California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[24] A. Tisan, S. Oniga, D. Mic, and A. Buchman, “Digital implementation of the sigmoid function for fpga circuits,” *Acta Technica Napocensis Electronics and Telecommunications*, vol. 50, no. 2, p. 6, 2009.

[25] *Mnist database*, http://yann.lecun.com/exdb/mnist, Accessed : 2017-06-12.

[26] *Svhn database*, http://ufldl.stanford.edu/housenumbers, Accessed : 2017-06-12.

[27] K. Vipin and S. A. Fahmy, “Fpga dynamic and partial reconfiguration: A survey of architectures, methods, and applications,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 72, 2018.

[28] E. M. Izhikevich, “Simple model of spiking neurons,” *IEEE Transactions on neural networks*, vol. 14, no. 6, pp. 1569–1572, 2003.

[29] H. Soleimani, A. Ahmadi, and M. Bavandpour, “Biologically inspired spiking neurons: Piecewise linear models and digital implementation,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 59, no. 12, pp. 2991–3004, 2012.

[30] M. Heidarpour, A. Ahmadi, and R. Rashidzadeh, “A cordic based digital hardware for adaptive exponential integrate and fire neuron,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 63, no. 11, pp. 1986–1996, 2016.

ملخص الرسالة

يعد العصر الحديث هو عصر الذكاء الاصطناعي, والذي يعتمد بشكل كبير علي الاله و قدراتها علي التعلم للقيام بالوظائف المعقده بدقة بالغه و قدرات حسابية معقدة.و تعد الشبكات العصبية الاصطناعية من اشهر التقنيات المستخدمة في مجال الذكاء الاصطناعي. ظهرت الشبكات العصبية الاصطناعية بقوة في العقود الاخيرة و اثبتت قدرتها علي القيام بالعمليات الحسابية المعقدة بكفائه عالية , كما انها تتعامل مع العمليات الحسابية بدقة و سرعة وهما من أهم مقاييس الدوائر الفعالة.

تختلف نوعيات الشبكات العصبية الاصطناعية باختلاف المشكلة المطلوب حلها , ومن المشاكل التي تساعد الشبكات العصبية الاصطناعية علي حلها : التصنيف و التعرف علي الانماط و التنبؤ. كما تلعب الشبكات العصبية الاصطناعية دور مهم في مجال السيارات ذاتية القيادة . و تعتمد عملية تصميم الشبكات العصبية الاصطناعية علي محاكاة العقل البشري في طريقة قيامة بالعمليات المعرفية و التعرف علي الاشياء وتحليل المعلومات و كيفية استنتاج المخرجات منه.

الشبكات العصبية الاصطناعية هي ابسط انواع الشبكات العصبية و تستخدم في حل المشكلات البسيطة نسبيا مثل التعرف علي الانماط و التصنية ,كما يسهل تصميم دوائر الكترونية لهذا النوع من الشبكات نتيجة بساطته. و هناك نوع اخر من الشبكات العصبية الاصطناعية وهو الشبكات العصبية التلافيفية. هذا النوع من الشبكات يتمكن من حل نوعيات اكثر صعوبة من المشكلات مثل معالجة الصور و الصوت و الفيديوهات. هذا النوع من المشكلات يتطلب عمليات حسابية اعقد و اكثر و هو ما تتمكن الشبكات العصبية التلافيفية من القيام به.

النوع الثالث والاحدث من الشبكات العصبية هو الشبكات العصبية المتصاعدة. يتميز هذا النوع من الشبكات بأنه ليس مجرد نموذجا حسابيا لمحاكاة العقل البشري ولكنه يعتمد بشكل كبير علي محاكاة كل العمليات البيولوجية التي تتم بداخل العقل البشري و بالتالي فهو نموذج بيولوجي للعقل البشري فهو يحاكي انماط التصاعد التي تتم داخل العقل البشري عند معالجة المعلومات كما يحاكي توقيت هذا التصاعد والذي يؤثر بشكل كبير علي نتائج العمليات.

يحتاج تصميم الدوائر الاكترونية لتنفيذ ايا من الشبكات العصبية الي الحفاظ علي ان تكون مساحة الدائرة اصغر ما يمكن بالاضافة الي عدم احتياجها الي قدر كبير من الطاقة و هذان العاملان من أهم العوامل الاساسية في الحكم علي كفاءة الدائرة الالكترونية. تستخدم تقنية تقريب الحسابات في تصميم الدوائر الالكترونية لعمل توازن بين مساحة الدائرة و استهلاكها للطاقة.

تعتمد تقنية تقريب الحسابات في الاساس علي التنازل عن جزء من دقة الحسابات قي مقابل تقليل مساحة الدوائر و الطاقة المستهلكة لتشغيلها. ونتيجة استخدام هذا النوع من الدوائر في تطبيقات لا تحتاج لدقة عالية فقد اصبحت تقنية تقريب الحسابات من أهم التقنيات المستخدمة في تصميم الشبكات العصبية.

تستخدم هذه الرسالة العديد من تقنيات تقريب الحسابات في تصميم الشبكات العصبية. هذه التقنيات تتضمن: تقليل دقة الارقام و تخطي الحسابات و تخطي الخلايا العصبية وتقريب دوال التفعيل و تقريب عملية الضرب و اقتطاع جزء من الارقام قبل جمعها. يتم تصميم هذه التقنيات باستخدام الشرائح الالكترونية القابلة للبرمجة و التي تدعم تقنية اعادة التكوين الجزئي الديناميكي. باستخدام هذه الخاصية تتمكن الدائرة الالكترونية من التكيف مع تغيير الطاقه التي تستخدم في تشغيلها. فعندما تقل الطاقة المستمدة تتغير تكوين الدوائر المكونة للنظام بشكل يمكنها من العمل بطاقة اقل ولكن يأتي هذا علي حساب تقليل الدقة.

تتضمن هذه الرسالة ايضا الشبكات العصبية المتصاعدة . في هذا النوع من الشبكات يتم تكوين الشبكات من وحدات اصغر تيسمي الخلايا العصبية, كما يوجد العديد من النماذج لهذه الخلايا العصبية. بعض من هذه النماذج يماثل تماما تكوين الخلايا العصبية البيولوجية و البعض الاخر يحاكيها فقط لخلق نموذج اقل تعقيدا من الخلايا البيولوجية ولكن مع الاحتفاظ بخصائصها. تتناول هذه الرسالة تصميم نموذج من الخلايا العصبيه يسمي " Izhikevich neuron model " و ذلك باستخدام تقنيه من تقنيات تقريب الحسابات الا وهي تقريب عملية الضرب و مع استخدام هذه التقنية يصبح تصميم الخلايا العصبية كدائرة الكترونية اقل تعقيدا و اقل استهلاكا للمساحة و الطاقة.

Insert photo here

مهندس :		
تاريخ الميلاد:		
البريد الالكتروني:	salmahasssansayed@gmail.com	
تلبفون:	01000789029	
العنوان:		
تاريخ التسجيل:		
تاريخ المنح:		
الدرجة:	الماجستير	
القسم:	الالكترونيات الكهربية	
المشرفون:	د/حسن مصطفي	
	د/أحمد نادر	
الممتحنون:	دكنور اخر	(الممتحن الخارجي)
	دكتور اخر	(الممتحن الخارجي)
	المشرف الرئيسي	(المشرف الرئيسي)
	دكتور اخر	(عضو)

عنوان الرسالة:

تصميم الشبكات العصبية الاصطناعية
باستخدام تقنية تقريب الحسابات و تقنية
اعادة التكوين الجزئي الديناميكي

الكلمات الدالة:

اعادة التكوين الجزئي الديناميكي, تقريب الحسابات, الشبكات العصبية الاصطناعية, الحسابات العصبية, الشبكات العصبية المتصاعدة

ملخص الرسالة:

تتضمن هذه الرسالة فكرة تصميم الشبكات العصبية الاصطناعية باستخدام تقنية اعادة التكوين الجزئي الديناميكي لاجزاء الدوائر الالكترونية و هذا لجعلها تتكيف مع التغيرات في الطاقة التي تقوم بتشغيل هذه الدوائر بصورة مستمرة, كما تحتوي هذه الرسالة علي تصميم لنمذج من نماذج الخلايا العصبيه الذي يسمي " Izhikevich neuron model " باستخدام تقنية تقريب عملية الضرب .

تصميم الشبكات العصبية الاصطناعية باستخدام تقنية تقريب الحسابات و تقنية اعادة التكوين الجزئي الديناميكي

اعداد

سلمي حسن سيد أبو المجد

رسالة مقدمة الي

كلية الهندسة – جامعة القاهرة

كجزءء من متطلبات الحصول علي درجة

الماجستير

في

الالكترونيات الكهربية

يعتمد من لجنة الممتحنين:

____

المشرف الرئيسى – المشرف الرئيسى

____

دكتور اخر

____

دكتور اخر – الممتحن الداخلي

____

دكتور اخر – الممتحن الخارجي
كلية الهندسة – كلية اخري

كلية الهندسة – جامعة القاهرة

الجيزه – جمهورية مصر العربية

مارس 2020

تصميم الشبكات العصبية الاصطناعية باستخدام تقنية تقريب الحسابات و تقنية اعادة التكوين الجزئي الديناميكي

اعداد

سلمي حسن سيد أبو المجد

رسالة مقدمة الي

كلية الهندسة – جامعة القاهرة

كجزءء من متطلبات الحصول علي درجة

الماجستير

في

الالكترونيات الكهربية

تحت اشراف

د/حسن مصطفي
الدرجه العلمية
الالكترونيات والاتصالات الكهربية
كلية الهندسة – جامعة القاهرة

د/أحمد نادر
الدرجه العلمية
الالكترونيات والاتصالات الكهربية
كلية الهندسة – جامعة القاهرة

كلية الهندسة - جامعة القاهرة

الجيزة – جمهورية مصر العربية

مارس 2020

بِحَمْدِهِ تَعَالَى

تصميم الشبكات العصبية الاصطناعية باستخدام تقنية تقريب الحسابات و تقنية اعادة التكوين الجزئي الديناميكي

اعداد

سلمي حسن سيد أبو المجد

رسالة مقدمة الي

كلية الهندسة – جامعة القاهرة

كجزءء من متطلبات الحصول علي درجة

الماجستير

في

الالكترونيات الكهربية

كلية الهندسة - جامعة القاهرة

الجيزة – جمهورية مصر العربية

مارس 2020