POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

PoET-BiN: Power Efficient Tiny Binary Neurons

SIVAKUMAR CHIDAMBARAM
Département de génie électrique

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie électrique

Décembre 2019

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

PoET-BiN: Power Efficient Tiny Binary Neurons

présenté par **Sivakumar CHIDAMBARAM**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

DEDICATION

To my parents

This is just the beginning...

ACKNOWLEDGEMENTS

Three years ago, a knock on the door of Prof. Jean-Pierre David is something I will always remember. Since then he has been with me through thick and thin making me a better individual professionally and personally. Setting research directions, constructing a solution step by step, not losing courage in dire situations are some the things I learnt from him. He has been a great source of motivation and inspiration for me. There is no amount of words enough to express my gratitude.

I would extend my heartfelt gratitude to my co-Supervisor Prof. Pierre Langlois. Apart form research, his pragmatic approach to solve problems, fast context-switching and thinking in a clutter-free manner are some of the things I deeply admire and wish to emulate. Apart from my supervisors, Ahmed has always been my mainstay at Polytechnique to discuss any topic under the sun from research to philosophy. The weekly meetings with him was something I cherished and looked forward to.

I would like to thank Roberto and Federico for their support and guidance during the initial days of my stay in Montréal. The laboratory wouldn't have been the same without Alex and Hossein. It was great work alongside them.

I would like to thank my friends who were a second family in Montreal- Aryan, Ajin, Arna, Pratap, Komal and Irfan.

Last, but not the least, I would like to my parent for their sacrifices, their belief in me and also for being with me in every step and endeavour I take.

RÉSUMÉ

Le succès des réseaux de neurones dans la classification des images a inspiré diverses implémentations matérielles sur des systèmes embarqués telles que des FPGAs, des processeurs embarqués et des unités de traitement graphiques. Ces systèmes sont souvent limités en termes de puissance. Toutefois, les réseaux de neurones consomment énormément à travers les opérations de multiplication/accumulation et des accès mémoire pour la récupération des poids. La quantification et l'élagage ont été proposés pour résoudre ce problème. Bien que efficaces, ces techniques ne prennent pas en compte l'architecture sous-jacente du matériel utilisé. Dans ce travail, nous proposons une implémentation économe en énergie, basée sur une table de vérité, d'un neurone binaire sur des systèmes embarqués à ressources limitées. Une approche d'arbre de décision modifiée constitue le fondement de la mise en œuvre proposée dans le domaine binaire. Un accès de LUT consomme beaucoup moins d'énergie que l'opération équivalente de multiplication/accumulation qu'il remplace. De plus, l'algorithme modifié de l'arbre de décision élimine le besoin d'accéder à la mémoire. Nous avons utilisé les neurones binaires proposés pour mettre en œuvre la couche de classification de réseaux utilisés pour la résolution des jeux de données MNIST, SVHN et CIFAR-10, avec des résultats presque à la pointe de la technologie. La réduction de puissance pour la couche de classification atteint trois ordres de grandeur pour l'ensemble de données MNIST et cinq ordres de grandeur pour les ensembles de données SVHN et CIFAR-10.

ABSTRACT

The success of neural networks in image classification has inspired various hardware implementations on embedded platforms such as Field Programmable Gate Arrays, embedded processors and Graphical Processing Units. These embedded platforms are constrained in terms of power, which is mainly consumed by the Multiply Accumulate operations and the memory accesses for weight fetching. Quantization and pruning have been proposed to address this issue. Though effective, these techniques do not take into account the underlying architecture of the embedded hardware. In this work, we propose PoET-BiN, a Look-Up Table based power efficient implementation on resource constrained embedded devices. A modified Decision Tree approach forms the backbone of the proposed implementation in the binary domain. A LUT access consumes far less power than the equivalent Multiply Accumulate operation it replaces, and the modified Decision Tree algorithm eliminates the need for memory accesses. We applied the PoET-BiN architecture to implement the classification layers of networks trained on MNIST, SVHN and CIFAR-10 datasets, with near state-of-the art results. The energy reduction for the classifier portion reaches up to six orders of magnitude compared to a floating point implementations and up to three orders of magnitude when compared to recent binary quantized neural networks.

TABLE OF CONTENTS

DEDICATION iii

ACKNOWLEDGEMENTS iv

RÉSUMÉ v

ABSTRACT vi

TABLE OF CONTENTS vii

LIST OF TABLES ix

LIST OF FIGURES x

LIST OF SYMBOLS AND ACRONYMS xi

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 BACKGROUND 5
- 2.1 A neuron 5
- 2.2 Fully connected layer 6
- 2.3 Convolutional neural networks 7
- 2.4 Training neural networks 8
- 2.5 Inference of neural networks 9

CHAPTER 3 LITERATURE REVIEW 11
- 3.1 Algorithm level modifications 11
  - 3.1.1 Quantization 11
  - 3.1.2 Pruning 14
  - 3.1.3 Decision Trees 15
- 3.2 Hardware implementation of neural networks 19

CHAPTER 4 THE PoET-BiN ARCHITECTURE 24
- 4.1 Multi-level RINC architecture 24
  - 4.1.1 RINC-0 : Modified Decision Tree algorithm 24
  - 4.1.2 RINC-1: Boosting the MAT units 29

4.1.3 RINC-L: Hierarchical Adaboost algorithm 32
4.2 Binary to multiclass classification 35
4.2.1 Replacing the hidden layers 35
4.2.2 Sparsely connected output layer 37
4.3 Parser to generate VHDL code 37

CHAPTER 5 EXPERIMENTAL SETUP, RESULTS AND DISCUSSIONS 41
5.1 Experimental setup 41
5.1.1 MNIST 42
5.1.2 CIFAR-10 42
5.1.3 SVHN 43
5.2 Results and Discussions 43
5.2.1 Classification accuracy 43
5.2.2 Power 46
5.2.3 Latency and Area 49
5.2.4 Simulation results 51

CHAPTER 6 CONCEPTION OF PoET-BiN 54
6.1 Attempt 1: Accelerating the Inference Phase in Ternary Convolutional Neural Networks using Configurable Processors 54
6.2 Attempt 2: Decision Trees and Adaboost 54
6.3 Attempt 3: Conception of the PoET-BiN architecture 55

CHAPTER 7 CONCLUSION 61
7.1 Limitations 62
7.2 Future Research 62

REFERENCES 63

LIST OF TABLES

3.1 FPGA implementation comparisons 23
5.1 Network architecture 42
5.2 Overall classification accuracy on MNIST, CIFAR-10 and SVHN dataset 43
5.3 Comparison with other techniques 44
5.4 Comparison with other works in the literature 45
5.5 RINC power results 46
5.6 Single arithmetic operation power results 47
5.7 Total arithmetic operations 47
5.8 Energy consumption 49
5.9 Implementation results 50
5.10 Hardware implementation comparisons 53
6.1 Accuracy on MNIST 56
6.2 Comparisons of various MAC operation grouping technique in Adaboost 60

LIST OF FIGURES

2.1 Neuron structure in human brain *[Image source]* 5
2.2 Artificial neuron 6
2.3 Fully connected layer 7
2.4 Convolutional layer 8
2.5 LeNet CNN architecture (Chidambaram et al.,2018) 9
2.6 Gradient descent 10
3.1 XNORNet (Rastegari et al., 2016) [With Permission] 13
3.2 Gated-XNOR Weight Update (Deng et al., 2018) [With Permission] . 15
3.3 Pruned neural networks 15
3.4 Basic decision tree 16
3.5 POLYBiNN (Abdelsalam et al., 2018) [©, 2018 IEEE] 17
3.6 Deep Forest 17
3.7 Deep neural decision forest 18
3.8 FINN 20
3.9 Vector processor (Chidambaram et al., 2018) 21
4.1 LUT and its equivalent DT 27
4.2 RINC-1 architecture with p=6 30
4.3 RINC-2 architecture 32
4.4 Intermediate layer 36
5.1 Overall work flow 41
5.2 Parser generated VHDL code 51
5.3 MNIST simulation of generated VHDL code 52
5.4 SVHN simulation of generated VHDL code 52
5.5 CIFAR-10 simulation of generated VHDL code 52
6.1 Deep Forest 57
6.2 Multi-level Adaboost 58
6.3 Architecture for a level of the multi-level Adaboost method 59
6.4 Grouped Adaboost 60

LIST OF SYMBOLS AND ACRONYMS

ALU	Arithmetic Logic Unit
BDD	Binary Decision Diagrams
CIFAR	Canadian Institute for Advanced Research
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DDR	Double Data Rate
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
DT	Decision Tree
FE	Feature Extractor
FPGA	Field Programmable Gate Arrays
GPU	Graphical Processing Unit
GOPS	Giga Operations Per Second
LUT	Look Up Table
MAC	Multiply and Accumulate
MNIST	Modified National Institute of Standards and Technologies
NDF	Neural Decision Forests
ReLU	Rectified Linear Unit
RINC	Reduced Input Neural Circuit
SIMP	Single Instruction Multiple Pipeline
SOP	Sum of Products
SRAM	Static Random Access Memory
SVHN	Street View House Number
VHDL	Very High Speed Integrated Circuit Hardware Description Language

CHAPTER 1 INTRODUCTION

Neural networks form the backbone of current technologies such as face recognition [1], text comprehension [2] and speech emulation [3]. Neural networks are mathematical models inspired by the neurons in our brains. The mathematical operations of each neuron can be broken down into a series of multiply and accumulate operations. This was first proposed by Rosenblatt in 1958 [4] . Except for a brief revival in the 1980s, neural networks were then mostly been confined to research labs. It wasn't until the late 2000s that neural networks started gaining traction in the industry for commercial use. Nowadays, neural networks are at the forefront of domains such as natural language processing [5], forecasting [6], anomaly detection [7], etc. The recent successes of neural networks can be attributed to the algorithmic advances made possible in part due to the availability of powerful computational devices and of large datasets to train them. Especially, the advent of powerful Graphical Processing Units (GPUs) [8] and the availability of large quantities digital data [9] in the late 2000s fueled the current revolution of neural networks (Artificial Intelligence).

Originally, neural networks were inspired by the interconnection of neurons in our brain to solve various classification and regression problems. Nowadays the neural networks have become more advanced and are used for more complex tasks such as generation of videos, medical image analysis, data reduction etc. However, the base mathematical operation in the neural networks still are MAC operations. Typical neural networks have millions of MAC operations. This makes GPU a natural choice to implement these neural networks. The first widely used commercial GPU, RIVA 128 was introduced by Nvidia in 1997 [10]. It consisted of 3 million transistors. It was mainly used for video games and 3D graphics acceleration programmed with OpenGL. During the same period, the performance of the Central Processing Units (CPUs) were limited by the clock rate and the number of cores to execute parallel threads. Even today the CPUs can only accommodate thousands of threads in parallel even with hyper threading. On the other hand, present day NVIDIA GPUs have 512 CUDA processor cores organized as 16 streaming multiprocessors architecture specifically optimized for computations and can schedule billions of threads while running tens of thousands of threads in parallel [11]. These GPUs have a Single Instruction Multiple Pipelines (SIMP) architecture that requires a single instruction to processes multiple data in parallel, akin to a streaming architecture. Hence, a need arises for fast memory access to provide data to the thousands of computation elements. These GPUs have various levels of shared memory with some requiring just a single cycle for memory read operations. Due to these factors, in spite of having slower clocks than CPUs, GPUs provide up to 500 $\times$ the throughput for data intensive tasks such as neural networks [12]. A major drawback with these devices is that they have extremely high-power consumption making them unsuitable for embedded applications. Therefore, we look at other embedded devices such as Field Programmable Gate Arrays (FPGAs) for a mobile deep learning applications that consume less power than GPUs.

With the advancements in neural networks various new applications started to emerge. Some of these applications required neural networks to be implemented on embedded devices. The embedded devices have their restriction in terms of resources, power and latency. Hence, researchers started exploring techniques to make the neural networks smaller and lighter to fit in these embedded devices. These modifications should not be detrimental to the accuracy of the system and also conform to the restriction of the embedded environment. FPGAs have been the goto devices to test these new proposals. FPGAs are semiconductor devices that can be reprogrammed to perform a given task after manufacturing. They consist of millions of configurable gates interconnected via programmable interconnects. FPGAs offer a unique hardware software co-design approach to build a highly parallelized and targeted system for a particular functionality. Their flexibility in designing makes them suitable a wide range of applications from cryptography [13] to space system designs [14]. The versatile nature of FPGAs make them an ideal candidate for neural networks as well. A neural network consists of two phases, namely the training phase and the inference phase. The training phase requires repeated processing of the large dataset. GPUs are able to stream multiple data parallely thus resulting in lower access time to fetch data that makes them suitable for training of neural networks. On the other hand, the inference task is time critical and is required to be implemented on mobile low power hardware, thus more suited for FPGAs than GPUs. Over the last few years, the neural network hardware accelerators built on FPGAs have been gaining traction over other devices such as embedded GPUs and embedded processors. These implementations of neural networks on FPGAs incorporate certain modifications to the existing algorithms to reduce their complexity.

A Vanilla neural network can be designed to achieve the best accuracy without much considerations for power consumption and memory requirements. Traditionally, these neural networks use 32-bit floating point representations that require expensive MAC operations and memory read operations that are unsuitable for FPGA implementations. Early full precision implementations consumed high power and had high latency. This led to a realization that to efficiently implement neural networks on FPGAs, it is important to tweak the training of neural network algorithms to make it more hardware friendly. This necessitated a trade-off between accuracy and hardware parameters (power, latency and area). Different innovations were proposed to maintain the accuracy while making the neural networks more hardware friendly. Two methods namely quantization and pruning have shown promising results.

Quantization techniques have been the go-to method to make neural networks more suitable for target FPGA hardware. It provides a dual advantage of reduced memory usage and smaller arithmetic units (multiplier and adders). This technique has shown promising results that resulted in current generation GPUs to include 16-bit, 8-bit and 4-bit MAC hardware. Initially, the effectiveness quantization techniques were demonstrated on smaller datasets such as MNIST, CIFAR-10 and SVHN. Nowadays, these methods are used to address more difficult challenges Imagenet classification. For smaller datasets, quantization has been applied to the extreme extent of single bit binary values of weights and activations. Surprisingly, in some cases quantization may not be detrimental to the accuracy of the network as it helps to better generalize on the unseen data. Similarly, pruning methods remove certain weight parameters from the network resulting in lower memory reads and MAC operations while maintaining the accuracy. However, pruning results in irregular network structure which requires sparse matrix techniques to efficiently implement it on hardware.

These methods, though effective, do not take the underlying hardware into consideration. The quantization and pruning are versatile techniques that can be applied over a spectrum of devices ranging from generic CPUs, massively parallel GPUs to task specific FPGAs. However, tailoring the neural networks specifically for the underlying hardware in FPGAs could lead to better implementations. To achieve a tailor-made architecture for FPGA, it is necessary to closely examine the computational units in the FPGA. A FPGA mainly consists of interconnected Configurable Logic blocks (CLBs). The CLBs are made of LUTs which are the computational powerhouse inside a FPGA. The structure of FPGAs is intrinsically closer to the architecture of a neural network because they embed millions of small computing elements, such as LUTs, that can be interconnected to form a large network of computing elements. However, these LUTs have limited input and output, for example Xilinx FPGAs have 4 or 6 input bits and 1-2 output bits while each neuron in a neural network typically contains thousands of inputs and upto 32 output bits. This mismatch still remains with quantization or pruning thus leading to inefficient implementations on FPGA.

In this work, we build networks of tiny binary neurons and map them to FPGA LUTs, thereby providing an improvised way to implement equivalent MAC operations while achieving near state of the art accuracy. We name this architecture and its associated building algorithm *PoET-BiN*. It is very power efficient since most of the processing is done as Look-Up operations in the binary domain and does not require floating point multipliers, adders or external memory accesses. Our LUT-based architecture combines Decision Trees (DTs) and weighted sums of binary classifiers. A major motivation to use LUT-based DTs is because it eliminates memory reads and can be implemented using simple logic gates, thus considerably saving power. On the other hand, DTs alone are weak classifiers. Hence, we modify the inherently weak classifiers to solve complex non-linear classification tasks consuming a fraction of the power compared to other implementations. This work makes the following contributions to combine DTs and weighted sums of binary signals to best exploit embedded hardware resources in the context of neural network implementations :

- A modified DT training algorithm to better handle a fixed number of inputs LUTs.
- The Reduced Input Neural Circuit (RINC) : A LUT-based architecture founded on modified DTs and the hierarchical version of the well known Adaboost algorithm to efficiently implement a network of binary neurons.
- A sparsely connected output layer for multiclass classification.
- The PoET-BiN architecture consisting of multiple RINC modules and a sparsely connected output layer.
- Automatic VHDL code generation of the PoET-BiN architecture for FPGA implementation.

The thesis is organized as follows:

- Chapter 2 provides a background on the basic concepts on neural networks
- Chapter 3 presents a comprehensive survey of current hardware-oriented deep learning research and hardware implementations of deep learning algorithms
- Chapter 4 details the proposed PoET-BiN architecture
- Chapter 5 explores the experimental setup, results and discussions
- Chapter 6 presents some discussions on how the how the proposed architecture was conceived
- Chapter 7 concludes the work and provides further avenues for this work

CHAPTER 2 BACKGROUND

In this chapter we introduce the basic concepts of neural networks starting from biological neurons to training of artificial neural networks.

2.1 A neuron

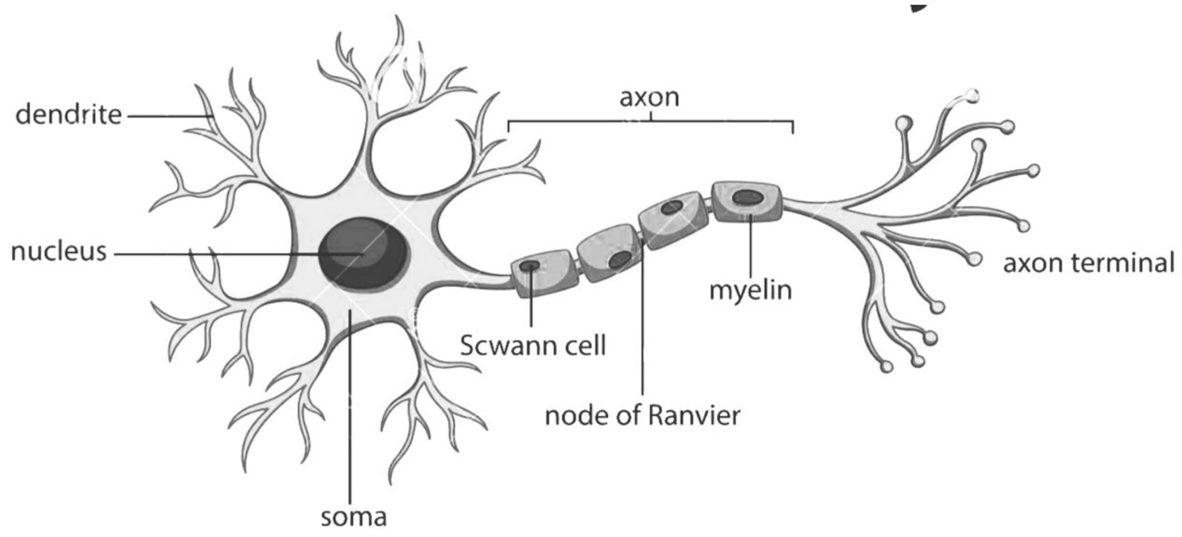


Figure 2.1: Neuron structure in human brain *[Image source]*

Fig. 2.1 illustrates the structure of a neuron in the human brain. These neurons have dendrites that collect information from the sensory organs and pass it to the axon. The axon is cable-like projection that extends up to a few millimeters in length. The axon have sodium dependent voltage channels that control the spiking of the axons. The axons fire only over a certain potential known as threshold potential. An artificial neuron shown in Fig. 2.2 follows a similar principle. The inputs (x_i) are similar to the dendrites. These inputs are multiplied with the weights(w_{nj}) and the result is added to obtain the output(net_j) of the transfer function. This output is passed through an activation function $(f(S))$ to get the final activation(o_j) of the neuron. The neuron has a sufficient output when the output of the transfer function is greater than a certain threshold (θ_j), otherwise it is insignificant. This process is similar to the process in the axon of the neurons in a human brain. Mathematically this can be represented as

$$o_j = f(\sum_{i=0}^{n-1} W_{ji}x_i + b_i) \tag{2.1}$$

$$o_j = f(W.x + b) \tag{2.2}$$

where b refers to the bias variable that is introduced to prevent non zero values of the transfer function.

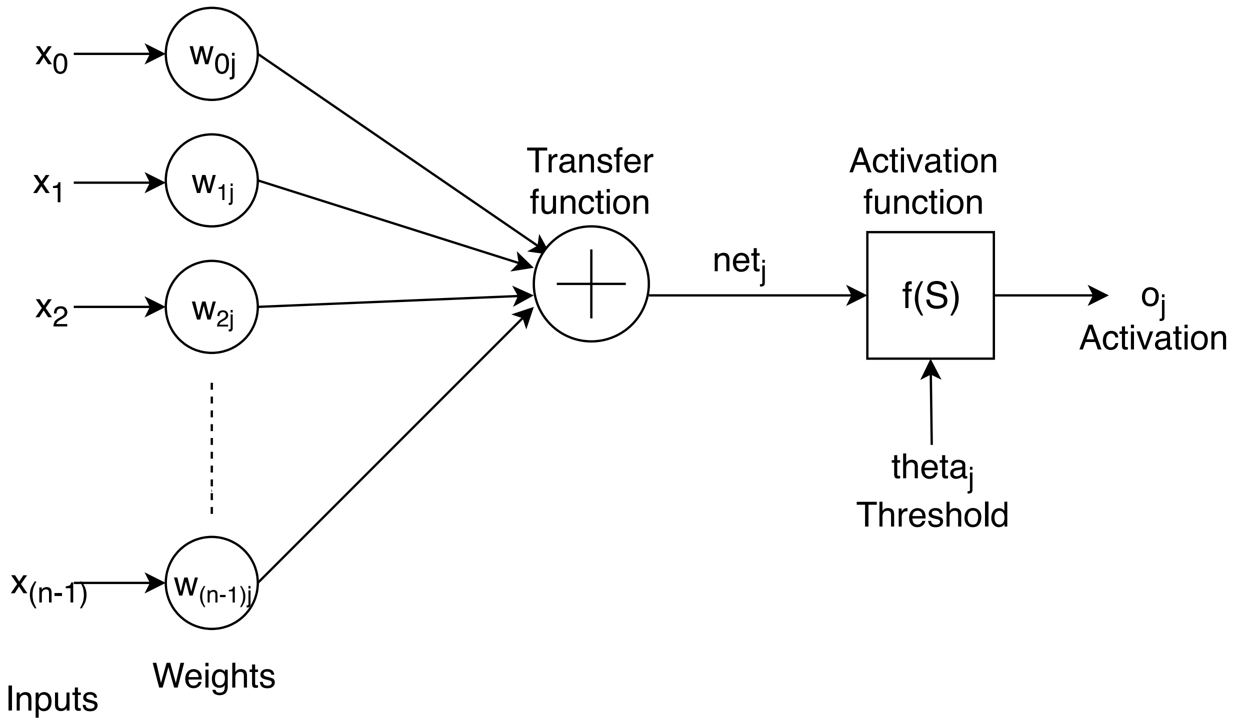


Figure 2.2: Artificial neuron

2.2 Fully connected layer

A number of these artificial neurons are stacked in parallel to form a layer of the neural network. The number of neurons in a layer can be upto 4096 neurons. A number of these artificial neuron layers are arranged one after the other sequentially where the inputs to the current layer are the outputs of the previous layer as seen in Fig. 2.3. A major advantage of the approach is that it allows non-linear representation of data. The non-linearity is introduced by the activation function. There are various activation functions, the most widely used being ReLU (Rectified Linear Unit) [15] and sigmoid [16] activations.

Fig. 2.3 represents a fully connected network. It is called fully connected because a neuron in the current layer is connected to all the outputs of the neurons in the previous layer.

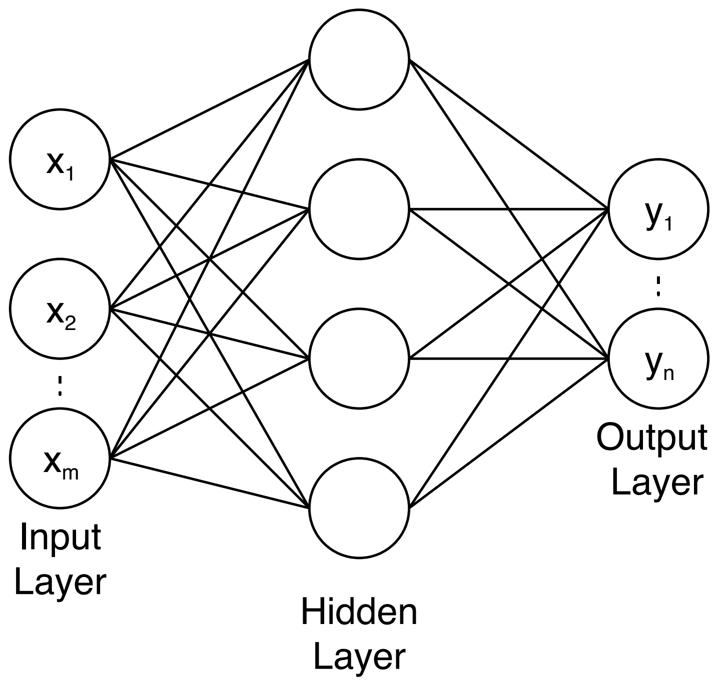


Figure 2.3: Fully connected layer

This results in millions of MAC operation in a network with 2 fully connected layers of 4096 neurons and a last layer of 100 neurons. The final fully connected layer in a network is called the output layer. It is different from the other layers as the number of neurons in the last layer is equal to the number of classes in the dataset. Each class is mapped to a neuron in the output layer and the activation of each neuron represents the probability of the input belonging to that class. Hence, it is used a classifier.

2.3 Convolutional neural networks

Sometimes, this non-linear classification by fully connected layers is not enough to adequately classify the images. It is necessary to extract meaningful information from the image first. This is called feature engineering and it has been done traditionally using filters [17], SIFT [18] and other techniques. However, these processes become tedious for large datasets. Lecun et al. [19] introduced the Convolutional Neural Network (CNN) that automatically extracts features from the image without any hand crafted features. It uses a 2d-Convolution layer to extract features from the input image. The input image is either in RGB or gray-scale format. The convolution layer contains 4 dimensional weights of size, $output\_channel \times$

$input\_channel \times kernel\_size \times kernel\_size$. A subimage containing $input\_channel \times kernel\_size \times kernel\_size$ is multiplied with each weight and added to form one feature of the convolutional layer as shown in Fig. 2.4. This helps capture spatial information such as edges or other distinct features. The new feature map spans over $out\_channels$. The $output\_size$ can be obtained from the input size as follows:

$$output\_size = \frac{input\_size + 2 \times padding - (kernel\_size - 1)}{stride} \tag{2.3}$$

There are numerous variations of the convolution layer such as tiled convolution [20], convolution with padding, large stride convolution etc.

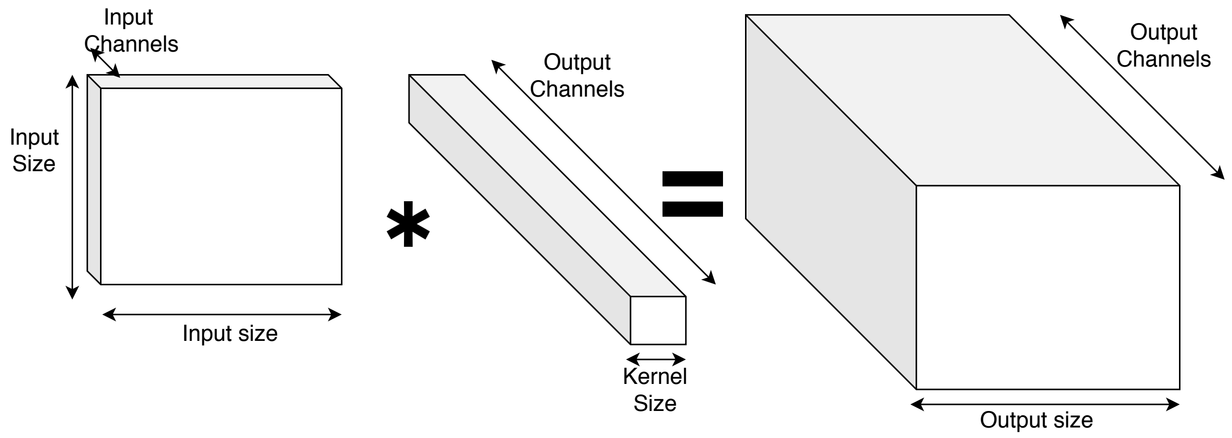


Figure 2.4: Convolutional layer

2.4 Training neural networks

A typical neural network consists of a series of convolutional layers stacked one after the other followed by the fully connected layer. The convolution layer acts as feature extractor while the fully connected layers act as classifiers. Fig. 2.5 shows a typical neural network. There are various standard architectures such as AlexNet [21], VGGNet [22], ResNet [23], etc. The max pooling layer is a non-parameterized layer used to reduce the size of the feature map for better generalization and reducing computations. Each instance of the convolution and fully connected layers contains a set of weights and bias. These weights and a bias must be chosen appropriately so that the whole network works in tandem to achieve higher classification accuracy. Larger networks such as VGGNet and ResNet have millions of parameters. It is impossible to exhaustively search entire combinations of inputs. Hence, we need to iteratively train the network to achieve higher classification accuracy.

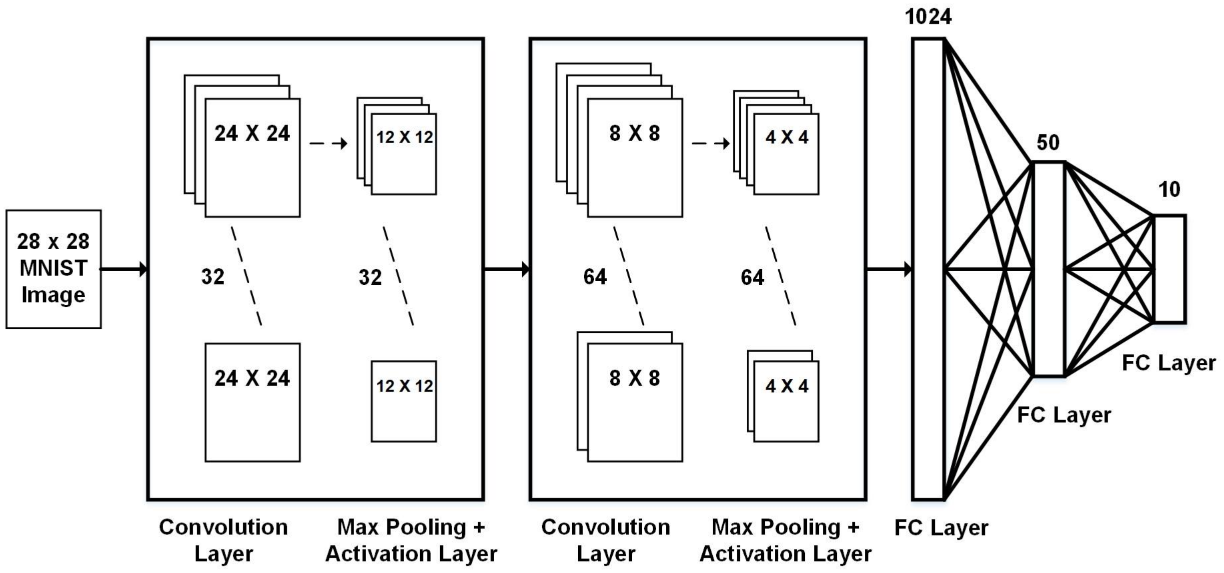


Figure 2.5: LeNet CNN architecture (Chidambaram et al.,2018)

Initially the weights of the neural networks are randomly initialized. An image is propagated through the network from the first convolution layer to the last fully connected layer. This step is called forward propagation. The last fully connected layer is connected to a loss function. The loss function calculates how far the predicted system output is from the original output in the dataset. The loss is directly proportional to the difference between the predicted and correct output. If the system accurately predicts the output, then the loss function is zero. Then, the next step is to calculate the gradient of the loss with respect to each parameter. This is called back propagation as we calculate gradients from the last layer and progressively move towards the first layer. Each parameter is subtracted by the gradient value (weighted by a learning rate). This can be seen as making small baby steps towards the local minima of the loss function. This is known as weight update. This process is repeated multiple times for each image in the dataset. Fig. 2.6 shows the loss function (J) that depends on weights (w). J_{min} is the local minima of the function. The arrows indicate the small steps taken by the network to achieve the lowest loss.

2.5 Inference of neural networks

Once the algorithm is trained, it performs well on the samples in the dataset. However, we need the network to perform on unseen data as well. Hence a given dataset is divided into 3 parts. First part is the training set that is used to train the parameters of the neural network. Second part is the validation set used to fine tune the hyper-parameters such as

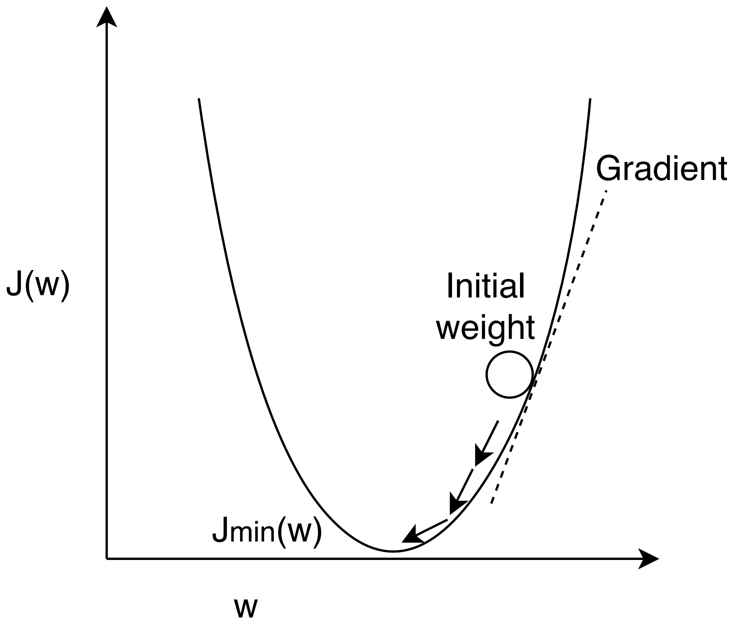


Figure 2.6: Gradient descent

the learning rate. Third part is the testset that is not used during training. It is only used to test the generalization capability of the network i.e the ability of the network to perform any unseen data. Generally 80% of the dataset is used as training set, 10% as validation set and 10% as testset. During the inference phase of neural networks only forward propagation is performed. It does not affect the weights. It is the inference phase that is implemented on embedded systems such as robots, microprocessors etc.

CHAPTER 3 LITERATURE REVIEW

In the earlier chapter basic concepts of neural networks was introduced. Traditionally, floating-point weights and activation are used in a neural networks [19]. It was necessary so as to allow sufficient representational capacity of the intermediate layers. Moreover, the hardware for neural networks were meant to only accelerate full-precision MAC operations. Hence, there was no necessity for smaller networks. In the meantime with better algorithms, more data and powerful computation devices, these neural networks have shown tremendous results [24]. Neural networks, especially for computer vision applications, started performing better than traditional computer vision application on image classification tasks [25]. With further innovation, neural networks started bettering humans [26]. This led to wide spread use of neural networks in industry [27]. Tasks ranging from chat boxes [28] to medical diagnosis [29] deployed neural networks extensively. Some of these applications were time critical and required limited power consumption. This necessitated modifications in the neural network algorithms to make them more hardware aware. In this chapter we mainly discuss two levels of modifications/innovations, firstly innovations at the algorithmic level and secondly innovations at the hardware level to better implement neural networks in space and power constrained hardware.

3.1 Algorithm level modifications

The inference phase of the Vanilla neural network is computation intensive. In order to implement these algorithms on power and memory constrained hardware, it is necessary to reduce the computation effort. In this section, we discuss three techniques, namely quantization, pruning and decision tree approach to efficiently implement these neural networks for classification of images.

3.1.1 Quantization

Quantization techniques have been the most popular technique for reducing the computations in the neural networks. It was first introduced by Courbariaux et al. [30] in 2016. The authors used a simple yet powerful technique of quantization of weights to reduce the computation complexity. Fig. 1 explains in detail the BinnaryConnect algorithm used to train these binary weighted neural networks. First the weights are randomly initialized as 32 bit floating point numbers. During forward propagation, the weights are binarized to

Algorithm 1 BinaryConnect Training Algorithm (Adapted from [30]

```
Input: minibatch of (inputs,targets), C is the cost function, L is the number of layers, $w_b$ are the binarized weights, $w_t$ are the full precision true weights, $b$ is the bias, $a$ is the activation, clip and binarize are clipping and binarizing functions respectively
1. Forward propagation
$w_b \leftarrow \text{binarize}(w_{t-1})$
For k=1 to L, compute $a_k$ knowing $a_{k-1}$,$w_b$ and $b_{t-1}$
2 . Backward Propagation
Initialize output layer's activations gradient $\frac{dC}{da_L}$
For k=L to 2, compute $\frac{dC}{da_{k-1}}$ knowing $\frac{dC}{da_k}$ and $w_b$
2. Parameter Update
Compute $\frac{dC}{dw_b}$ and $\frac{dC}{db_{t-1}}$ knowing $\frac{dC}{da_k}$ and $a_{k-1}$
$w_t \leftarrow \text{clip}(w_{t-1} - \alpha\frac{dC}{dw_b})$
$b_t \leftarrow b_{t-1} - \alpha\frac{dC}{db_{t-1}}$
```

either -1 or 1. Stochastic and deterministic binarization were experimented with and were found to produce similar results. These binary weights eliminate the need for floating-point multipliers. During backward propagation as well the binarized weights are used to calculate the gradients. However, these gradients are small, hence requiring floating point precision. These gradients are further multiplied by the learning rate (less than 1). The resulting value needs to subtracted from the current parameter values. This step is known as weight update. To better capture the resulting changes, the weight update step is performed on the original full precision weights that are stored in the memory. Though this does not decrease the complexity during training, the inference (forward propagation) complexity is greatly reduced. The reduction in inference time and complexity is due to reduced memory requirements to store the binary weights and as well as the elimination of floating-point multiplication. The reduction in memory by 32× allows the weights to be stored in on-chip memory thereby reducing the long delays associated with fetching weights from external DDR3 memory.

BinaryConnect still require floating point adders and subtractors for the forward propagation. These operations are power and time intensive. Hence, Courbariaux et al. [31] then introduced BinaryNet, a neural network with binarized weights and activations. In this work, along with binarizing the weights, they also binarized the activation during the forward and backward pass. Hence the floating point additions and subtractions are substituted by XNOR multiplications and popcount [32] additions. With this architecture they were able to achieve 7× faster inference as compared to Vanilla neural networks.

Along with reducing the computation effort, the BinaryConnect and BinaryNet architecture were able to maintain near state-of-the-art accuracy on the MNIST [33], CIFAR-10 [34], and

SVHN datasets. In fact the validation accuracy of BinaryConnect is better than Vanilla neural network (neural networks with full precision weights and activations) in the case of CIFAR-10 and SVHN and almost similar in the case of MNIST. This is due to the regularization effect [35]. In BinaryConnect, using binary weights injects some noise in the system preventing over-fitting. This results in better generalization on unseen data. Hence, BinaryConnect performs better than the Vanilla neural networks. On the other hand, in BinaryNet there is not enough precision to represent the entirely the features extracted. Hence, the BinaryNet accuracy is slightly worse than Vanilla neural networks. Nevertheless, it is negligible reduction and the computation and memory reduction achieved by binarization is massive.

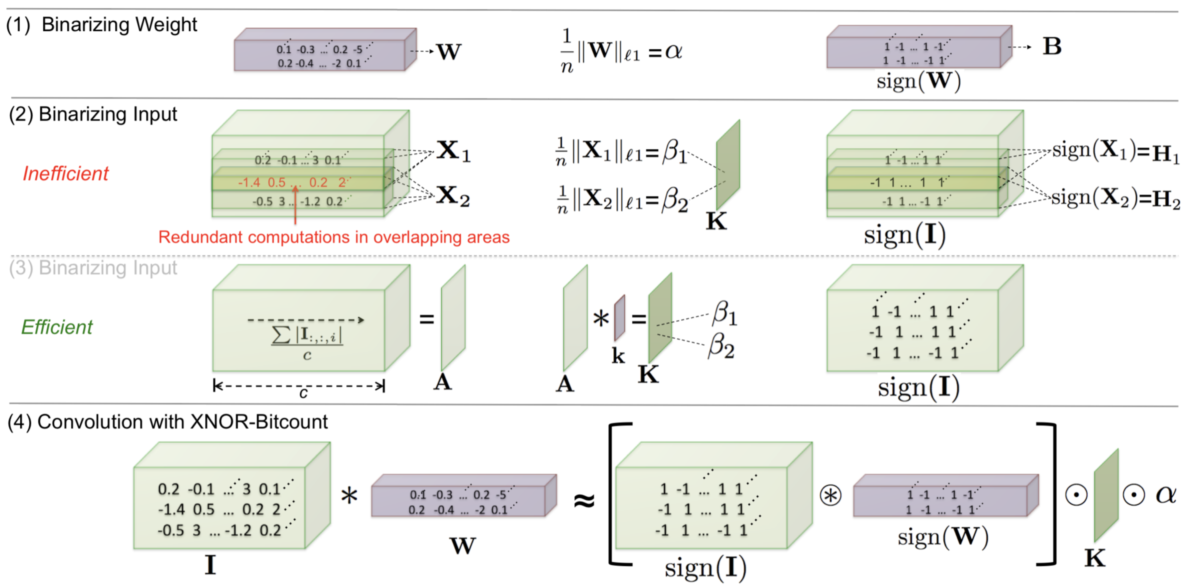


Figure 3.1: XNORNet (Rastegari et al., 2016) [With Permission]

The BinaryNet and BinnaryConnect focus on reducing the computations and memory requirements for the inference part of the network. There are other works in the literature that use various other quantization methods to reduce the computation. One such work, XNOR-Net developed by Rastegari et al. [36], proposes a unique method to quantize the weights and activations while achieving better accuracies than BinaryNet or BinaryConnect. In this work, in addition to binarizing the weights and activations during forward propagation, they introduce a scaling constant layer (K) to scale the outputs after convolution operation to scale the the output values. These scaling constants are learned during the training phase. It can be represented mathematically as

$$X * W \approx (sign(X) * sign(W)) \odot K\alpha \tag{3.1}$$

where X is the input to a convolution layer, W refers to the weights of the layer and α refers to the learning rate. Using this technique the authors achieved 80% top-5 accuracy on the ImageNet dataset [9] while BinaryNet achieves 69% with AlexNet architecture.

These were some of the first papers to introduce Quantized neural networks for a better, faster and cheaper inference phase. These works still need full precision weights to be stored during the training phase. Since then, there have been many works to reduce the computation during the training phase. One such work, Gated-XNOR [37] inspired by XNOR-Net uses a similar approach for the forward and backward propagation albeit a ternary precision -1,0,1 network instead of binary precision. The major change is in the weight update step. In XNOR-Net the weights are stored and updated with full precision. However, in the Gated-XNOR architecture the gradient information is used to stochastically update the weights to either one of the ternary states. Hence weights and activations are represented with ternary precision throughout the training and inference phase. The authors achieved similar accuracies to that of XNOR-Net for the MNIST and CIFAR-10 datasets. Fig 3.2 represents the state changes for various possible combination in the Gated-XNOR work. v refers to the gradient, τ is a stochastic transformation function applied on the gradients.

These are some of the influential works in quantizing neural networks. Other works in quantization employ techniques such as quantization of gradient [38] and quantization of loss function [39] to name a few.

3.1.2 Pruning

Pruning is another technique often used to reduce computations in neural networks. It involves removing select neurons from the network while achieving similar accuracies to that of Vanilla networks. In some extreme cases, the network is reduced by 49× without affecting the accuracy [40]. Karnin [41] was one of the first to show that it is possible to achieve similar accuracies by removing some neurons from a neural network. He modified the cost function to account for the removal of neurons. Pruning neurons eliminates MAC operations and also eliminates the need to store the corresponding weights of the neuron. Fig. 3.3 illustrates the connections in a pruned neural network. A similar work by Wan et al. [42], Dropconnect proposed to make the weight parameters zero instead of the entire neuron. They achieved almost 90% accuracy on CIFAR-10 and 98% accuracy on SVHN.

As this work does not use pruning we do not go into the details of these implementations.

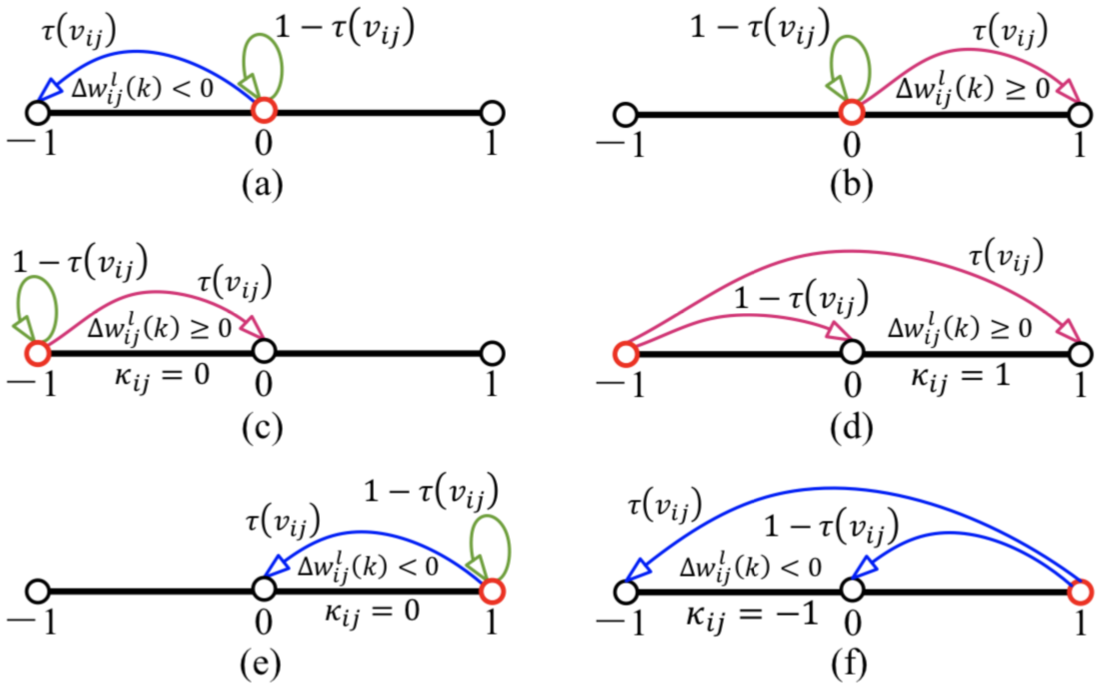


Figure 3.2: Gated-XNOR Weight Update (Deng et al., 2018) [With Permission]

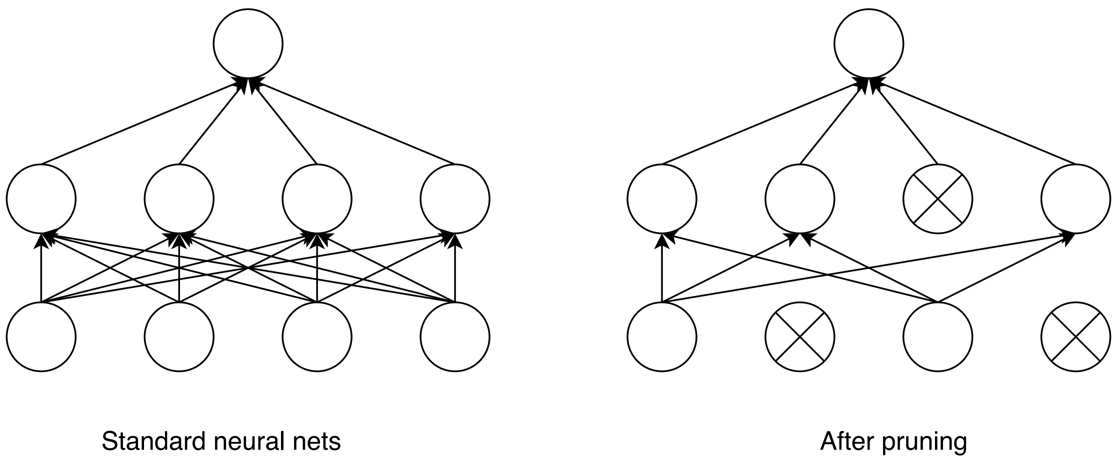


Figure 3.3: Pruned neural networks

3.1.3 Decision Trees

Decision Trees (DT) are a classical machine learning method used for data classification. With the advent of neural networks they have often been overlooked. Decision trees are flowchart like structures made of nodes, each implementing one if condition. The DTs try

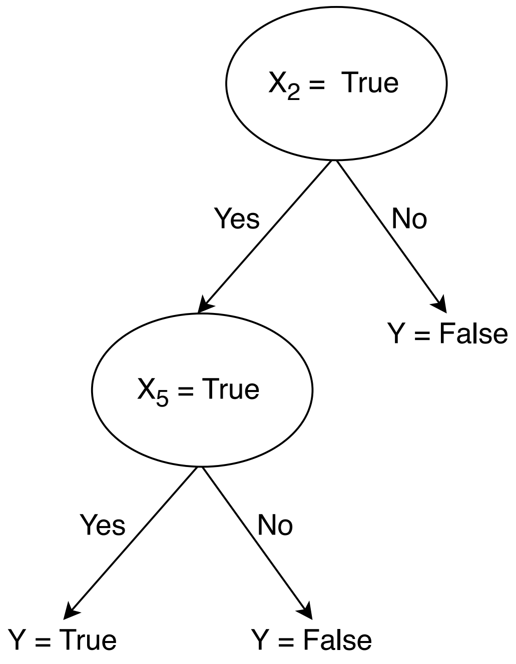


Figure 3.4: Basic decision tree

to reduce the entropy of the system to a maximum extent possible by choosing the best features at each node. They follow a greedy approach in reducing the entropy. The branch nodes represent an if condition and the leaf nodes contain the class label. DTs can be used for binary as well as multiclass classification. Fig. 3.4 illustrates a basic DT for binary classification. Depending on the feature X_2, the right or left path is chosen at each node. Similarly, at the second node the path taken is determined by input X_5. The leaf nodes contains the class label ($True$ or $False$). Each sample arrives at any one of the leaf nodes based on the input values of X_2 and X_5. If the DT can contain more nodes then other input features (X_1, X_3,etc.) are considered.

Abdelsalam et al. [43] used this approach of DT for the classification of the MNIST and CIFAR-10 datasets. They implemented the DT as a Sum of Products (SOP) shown in Fig. 3.5 for binary classification. Multiple decision trees are grouped together using the Adaboost algorithm. The authors employed a one-vs-all method to use these binary classifiers for one-vs-all classification. The one-vs-all classification involves building a binary classifier for each class of the dataset. In each binary classifier each class is considered true and all other classes are considered false. Once a binary classifier has been trained for a single class, the target labels are reassigned in the data making the the next class true and and other class false. Once all the class have been trained it is possible to have a multiple true prediction by the classifiers. These ties are solved using a confidence circuit. These decision trees, the

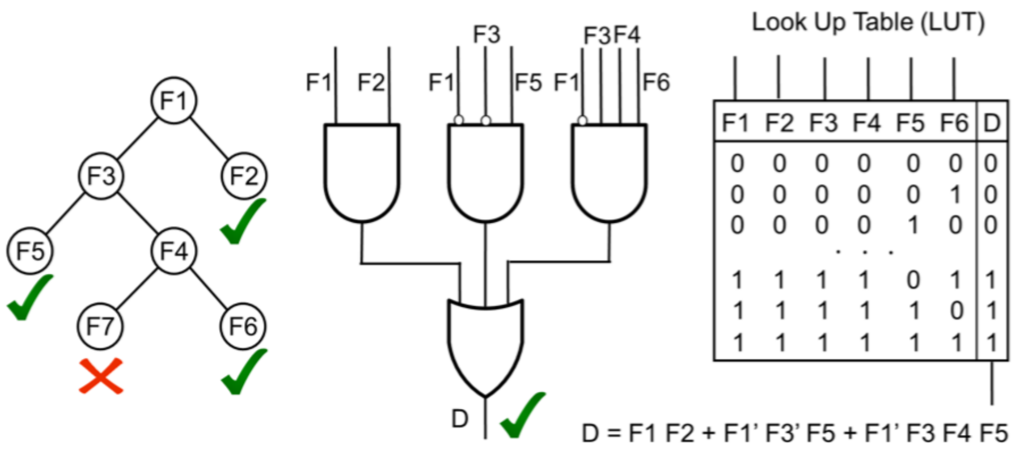


Figure 3.5: POLYBiNN (Abdelsalam et al., 2018) [©, 2018 IEEE]

Adaboost algorithm and the confidence circuit are implemented on the FPGA together known as POLYBiNN. They achieved competitive accuracy in the case of MNIST and CIFAR-10 while using a fraction of the resources as compared binary neural network implementations as they eliminate all MAC operations and memory fetches.

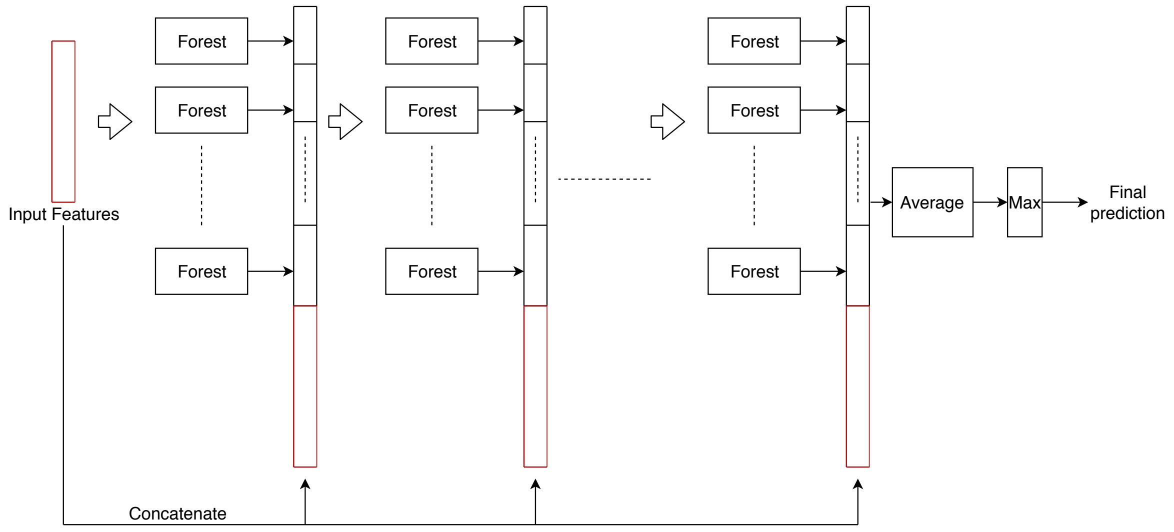


Figure 3.6: Deep Forest

The decision trees are weak classifiers as compared to Multi Layer Perceptrons (MLP). To augment the capacity of decision trees, Zhou et al. in their work, cascaded decision layers one after another [44]. This increased the classification capacity of the decision trees. Fig.3.6 shows the Deep Forest architecture where the input feature along with the features generated from the previous layer are fed as inputs to the current layer. The authors use Random forests [45] instead of Adaboost to group the decision trees. In this work, the authors try to eliminate the convolution layers with series of these random forest layers. However, the authors could only achieve 68% accuracy on the CIFAR-10 dataset. This suggests that Convolution layers are indispensable to achieve higher accuracies. Moreover, a hardware implementation of the deep forest architecture would be power and resource intensive owing to the use of multi-class decision trees [46]. The authors conclude that DTs alone cannot be used for classification of complex datasets such as CIFAR-10. A mixed approach using neural networks and decision trees is needed to classify complex problems in a resource and power efficient manner.

Neural Decision Forests [47] propose such a mixed approach with differentiable decision trees to integrate neural networks and decision trees. The authors achieve almost 94% accuracy on the the Imagenet dataset. However, the authors only replace the final layer with decision tree as shown in Fig. 3.7. The rest of the network remains the same. The figure shows the neurons in the final fully connected (FC) layer neuron implemented as DTs. The decision nodes (d_x) replace the corresponding last fully connected layer nodes (f_x). The inputs to the network come from the CNN architecture and gets routed to one of the leaf nodes depending on the decisions at each of the decision node. The leaf nodes (l_x) of the DTs provide the probability for each class (named A,B and C for illustration purpose). As most of the network architecture remains the same to that of an original CNN network it does not allow for huge gains in terms of power and resources. Nevertheless, it opens the door for more power efficient networks using the representational capacity of neural networks and power efficient decision trees.

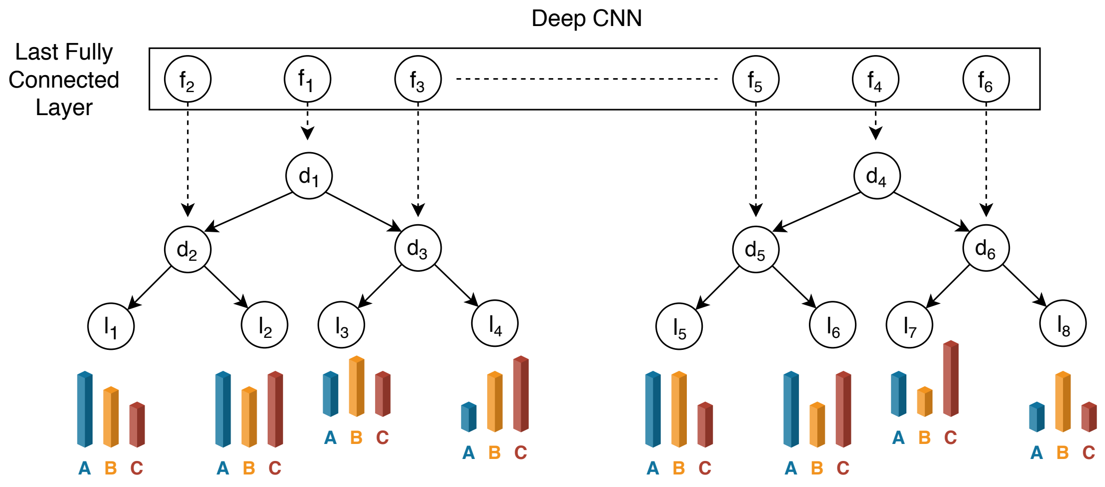


Figure 3.7: Deep neural decision forest

3.2 Hardware implementation of neural networks

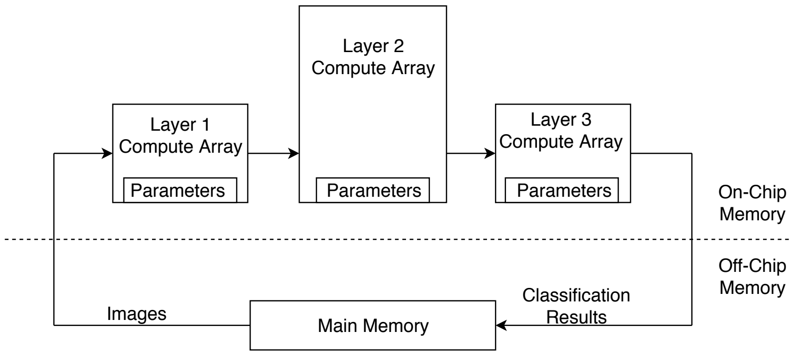
In the previous subsections various innovations to better implement neural networks in hardware were discussed. These innovations proposed changes at the algorithmic level. In this section, we introduce and compare actual implementations of neural networks in hardware. Hardware implementation of neural networks come in four different types [48] :

- A single inference engine processing each layer sequentially.
- A streaming architecture with a single processing engine per layer and all the layers work in parallel.
- A vector processor with application specific instructions to accelerate convolutions and matrix multiplications
- A neurosynaptic processor implementing spiking neural networks

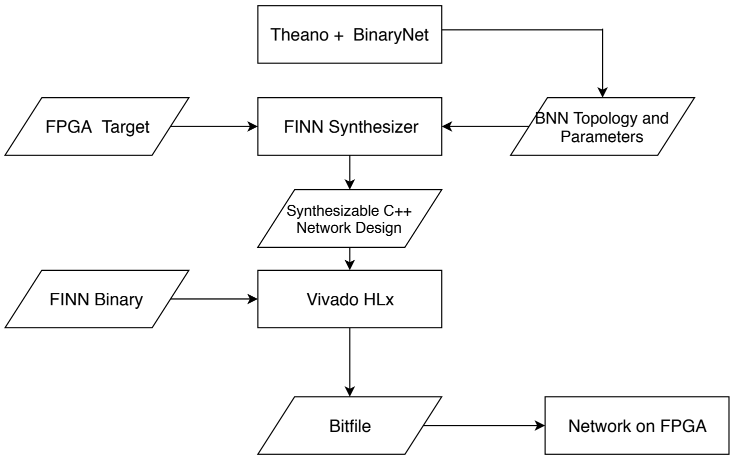
One of the first implementations of binary neural networks was FINN (A Framework for Fast, Scalable Binarized Neural Network Inference) by Umuroglu et al. [48]. The FINN architecture is a streaming architecture with a single processing engine per layer as shown in Fig. 3.8a. The processing engines of each layer are pipelined. The authors main focus was to easily build custom low power inference engines on FPGAs. As shown in Fig. 3.8b, the major part in a FINN system is a synthesizer that takes high level network parameters from the Theano code and generates the high level C++ code to be given to the Vivado HLx [49]. The Vivado HLx generates the Hardware Description Language (HDL) for hardware implementation. The generated hardware is targeted for low power implementations. As seen in Fig. 3.8a, the FINN is a streaming architecture that uses a processing engine for each layer. These layers operate in parallel increasing the utilization of shared peripherals such as memory and drastically reducing the latency.

Horowitz et al. [50] provide the power consumption of each operation in a neural network. It can be observed that the DRAM memory fetching instruction consumes 10$\times$ more power than a multiplication operation. Hence, it becomes vital to reduce the memory requirements. Therefore, by employing a binary quantization scheme, the weights were stored in the on-chips SRAM rather on than external DRAMs. This significantly reduced the power consumption and increased the throughput. The authors could achieve 15000 frames per second (FPS) at 250 KHz while consuming just 0.2 W of power.

Our previous work [51] is an example of a vector processor. In this work we introduced a vector processor to accelerate a CNN for the MNIST dataset on Application specific configurable



(a) FINN architecture



(b) FINN synthesis flow

Figure 3.8: FINN

processor. We built a custom module to accelerate the MAC operation in the convolution and fully connected layer by a factor of 5× and 3.6× respectively. We also built specific custom instructions for max pooling and activation functions. We achieved an overall acceleration of 4.8× as compared to a sequential implementation of the same neural network on the configurable processor. Fig. 3.9 illustrates the processor architecture. The user custom block generated are used as alternative to the ALU to accelerate a specific computation. Other embedded modules such as DSP are added to further accelerate the computations. Apart from the computations, other memory management techniques are used to accelerate the data fetches. First, the weights fetched from the external memory are stored in the extra register file to prevent repeated costly and time consuming memory fetch operation. Second, we reuse overlapping weights in the convolutional layer using a special circuitry in the dot product custom instruction. Our work was published at in the IEEE DASIP conference proceedings [51].

Apart from the conventional artificial neural networks another class of neural networks called

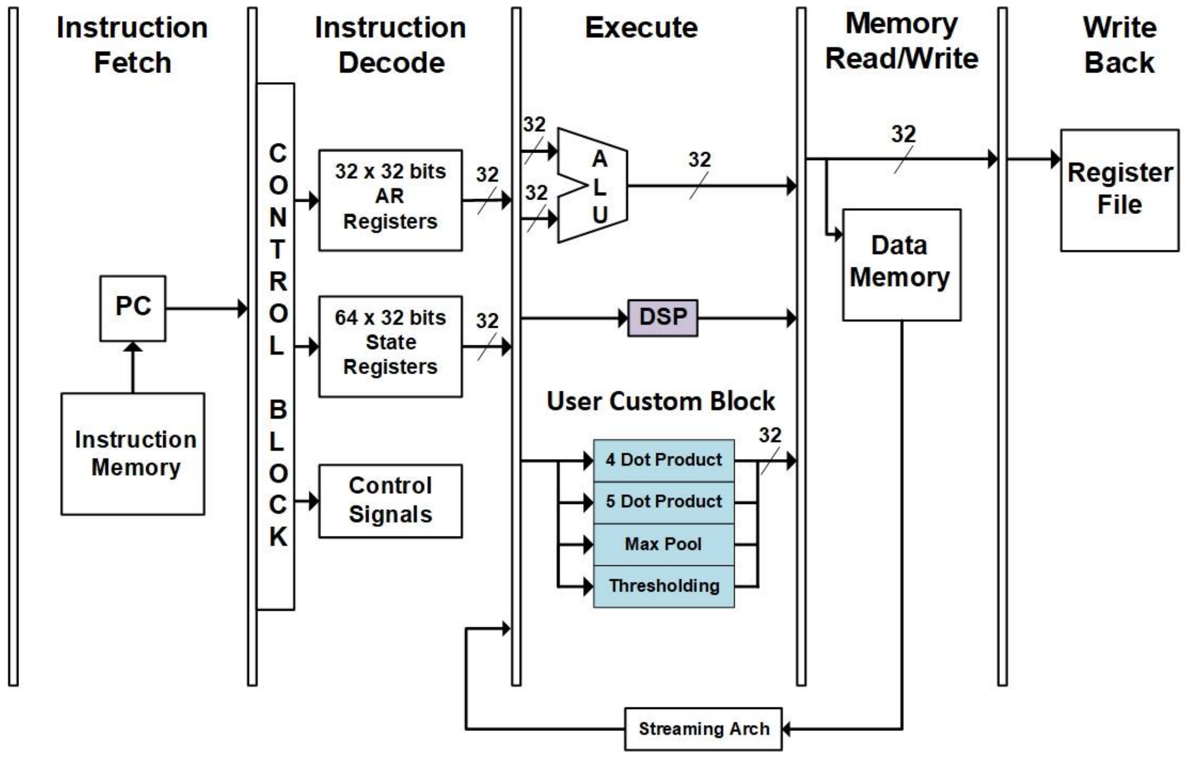


Figure 3.9: Vector processor (Chidambaram et al., 2018)

Spiking neural networks have shown promising results. Spiking neural networks were introduced by Xin et al. in [52]. Spiking neural networks are mathematical structures more resembling the neurons in the human brain. They use differential equations to model the neurons. Apart from the spatial configuration of the neurons, they take into consideration the time between firing of connected neurons. TrueNorth by Akopyan et al. [53], is a cornerstone of spiking neural networks achieving state-of-the-art accuracies on the MNSIT dataset while consuming a fraction of the power as compared to full precision neural networks. They decode the (Echo ElectroGram)EEG of the brain to design the neurosynaptic processor encoding the information in the spikes of the neurons.

There have been numerous implementations of neural networks on a variety of embedded devices such as FPGAs, microprocessors, configurable processors [54] and embedded GPUs [55]. Table 3.1 adapted from [56] gives a brief comparison between hardware implementations of neural networks on FPGAs. Various recent implementation of Quantized neural networks for MNIST, CIFAR-10 and SVHN datasets were studied. They implementation are compared in terms of throughput, power and resources (LUTs, BRAMs and DSP) used. The implementations are arranged in terms of their level of quantization. Binary networks such as [57], [58], [58], [59] employ extreme form of quantization. Ternary networks [60] quantize the weights and activations to 2-bits. Other quantized networks range from 8 bits to 32 bits fixed point implementation. As expected, the binary networks have the highest throughputs. [58] could achieve a throughput of 849 GOPs and 32 bits fixed point implementations have some of the lowest throughputs with Zhang et al. [61] achieving only 9 GOPs. Although, Moss et al. [58] could achieve very high throughput, it comes at the expense of resources. They consume 9 times more resources as compared to other binary implementations [57]. This trend is observed across the table with various implementation sacrificing resources for better througputs [61], [60], etc. The power reported is the static + dynamic power. The static power depends on the resources and the area utilized by the design, while the dynamic power depends on the throughput. Hence the total power depends on both the throughput and the resources utilized. It should be also noted that the throughput, power and area heavily depends on the FPGA board and its technology.

This concludes the literature review chapter. More focus has been given to the quantization techniques and decision tree over pruning and the hardware implementations as the majority of the work in this thesis is built on these techniques.

Table 3.1: FPGA implementation comparisons

Network	FPGA	Dataset	Quant	Power(W)	Throughput	LUTs	BRAM	DSP
Nakahara et al., 2017, [57]	Zynq	CIFAR-10	1-bit	2.3	143	14509	32	1
Moss et al, 2017, [58]	Arria	CIFAR-10	1-bit	48	849	115000	-	-
Umuroglu et al., 2017 [48]	PYNQ	CIFAR-10	1-bit	2.5	166	42823	270	32
Zhao et al., 2017, [59]	Zynq	CIFAR-10	1-bit	4.7	143	46900	94	3
Jiao et al., 2017, [60]	Zynq	MNIST	2-bit	2.26	181	44000	105	89
Guo et al., 2017, [62]	Zynq	SVHN	8-bit	4.5	24	29867	85	190
Ma et al., 2017, [63]	Arria	CIFAR-10	8-bit	21.2	31	16100	1900	1518
Qiu et al., 2016, [64]	Zynq	CIFAR-10	16-bit	9.63	14	-	-	-
Zhang et al., 2018, [65]	Virtex-7	CIFAR-10	16-bit	26	13	300000	1248	2833
Shen et al., 2018, [66]	Virtex	SVHN	16-bit	26	30	170000	1232	1376
Zhang et al., 2017, [67]	Arria	SVHN	16-bit	37.4	48	-	1250	1320
Podili et al., 2017, [68]	Stratix V	CIFAR-10	32-bit	8.04	29	196370	256	1100
Zhang et al., 2015, [69]	Virtex-7	SVHN	32-bit	18.6	3	186251	1024	2240
Zhang et al., 2017, [61]	Stratix V	CIFAR-10	32-bit	13.2	9	200522	4096	224

CHAPTER 4 THE PoET-BiN ARCHITECTURE

In this chapter we detail the principal contribution, multi-level RINC architecture for binary feature representation (4.1) and also the sparsely connected output layer for multiclass classification (4.2), who together constitute PoET-BiN.

4.1 Multi-level RINC architecture

The Reduced Input Neural Circuit (RINC) is a network of tiny binary neurons with limited inputs. In a traditional neural network, each neuron may be connected to a large number of other neurons, for instance up to 4096 neurons in the VGG architecture [70]. This impedes efficient implementation on hardware as it leads to numerous and interdependent logical circuits that adversely affect the power consumption, speed and area of the architecture. In our architecture, the total number of inputs to each neuron is limited, usually less than 8. This poses a major challenge to choose the best inputs among the ones available and classifying the data based on only these selected inputs.

We follow an approach inspired by DTs to implement a tiny binary neuron (subsection 4.1.1). The size of the DTs is limited by the number of LUT inputs. DTs are inherently weak classifiers and boosting techniques are used to group weak classifiers thus forming stronger classifiers. The Adaboost algorithm is one of the most widely used boosting algorithms. In sub-section 4.1.2, we detail our LUT-based implementation of the Adaboost algorithm. Still, the LUT-based algorithm is limited by the number of weak classifiers that can be grouped together. To further enhance our classifiers, we introduce a hierarchical Adaboost algorithm, where the number of DTs increases exponentially with every level. At each level, all the operations are designed to exactly fit in a single LUT, thereby optimizing power and area efficiency. The hierarchical algorithm is detailed in subsection 4.1.3.

4.1.1 RINC-0 : Modified Decision Tree algorithm

A binary neuron only has two possible outputs, so all possible input combinations can be classified into two groups. Hence, each binary neuron is a binary classifier that can be implemented as an Input vs Output table for all the possible input combinations in LUTs (Fig. 4.1). Hardcoding such tables in a LUT or a memory is feasible when the number of inputs is limited. With each added input the complexity increases by a factor of 2. In most cases, we need to model a binary neuron with more inputs than can be accommodated in a single

LUT. Therefore, hardcoding the input-output relation is not a viable option. This demands an algorithm to choose the best inputs from the input set so as to fit the implementation in a single LUT. We use a greedy approach inspired by DTs to choose the best inputs from the available set of inputs. There are many other traditional machine learning classifier such as Support Vector Machines [71] and Naive Bayes classifier [72]. However, DTs provide a distinctive edge as they can be easily and efficiently implemented in LUTs [43].

The original DT algorithm [73] is limited either by the depth or by the number of nodes, which often leads to under utilization of LUTs, because the LUTs used to implement the DTs are neither constrained by the depth or nodes, rather by the number of distinct inputs. Hence, we propose a modified DT algorithm that attempts to optimize DTs for a given number of inputs. Off-the-shelf DT classifiers are built one node at a time. Each node is associated with a feature that divides the input feature space and minimizes the entropy of the DT. Contrarily, we train DTs layer-wise. Hence, all nodes in the same level of the DT have the same features. This divides the input feature space into 2^P (where P is the number of inputs to a LUT) sub-spaces and increases the capacity of the DT for a given number of input features. Moreover, DTs can be evaluated in $O(1)$ time, when implemented as a LUT. The resulting LUT-based DT is named as Reduced Input Neural Circuit (RINC-0), where-"0" signifies the level, which is further explained in the following sections. With this modified DT approach we have an increased capacity RINC-0 architecture that can fit exactly into one LUT and is limited by the number of input features.

Algorithm 2 details the training algorithm for the RINC-0 module. The training dataset contains n examples of F dimensions each. Our goal is to choose the best P features from the available F features that best classify the data, or, in other words, reduce the entropy. We train a level-based DT approach where we choose the best feature ($best\_feature$) from all features that have not been used before, that reduces the entropy of the entire level to the largest extent. The best feature is appended to the Used feature array($Used\_features$). The label array ($Label\_array$) contains the class label for each leaf node. A leaf node is assigned a class that has the highest number of training examples that end up in that leaf node. These class labels form the output column of the LUT and the best features form the input indices to the LUT.

Fig. 4.1 illustrates a DT and its equivalent LUT. The red and green arrows at each node represent the path taken when the feature at the node is 0 or 1, respectively. This implementation of a RINC-0 module is versatile and not limited to LUTs alone. The approach can also be implemented in memory blocks as well, as we only have to store a table with precomputed output values for each combination of input values. Since the memory size is computed as

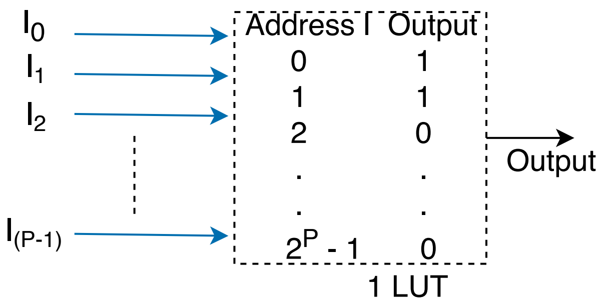
Algorithm 2 RINC0: Level wise DT training algorithm

```
Input: data $X$, size $n \times F$
Initialize $Used\_features$ = []
Initialize $Label\_array$ = []
for $i \rightarrow 1$ to $p$ do
    for $feat \rightarrow 1$ to $F$ do
        if feat not in Used_features then
            level_entropy → 0
            for $node \rightarrow 1$ to $2^{p-1}$ do
                Calculate entropy of the current node
                level_entropy += node_entropy
            if level_entropy ≤ min_entropy then
                min_entropy → level_entropy
                best_feature → feat
    Append Used_features array with best_feat
for $cur\_node \rightarrow 1$ to $2^P$ do
    $S_0 \rightarrow$ Sum of $class_0$ training examples at cur_node
    $S_1 \rightarrow$ Sum of $class_1$ training examples at cur_node
    if $S_0 \leq S_1$ then
        Append Label_array with 1
    else
        Append Label_array with 0
Return Label_array, Used_features
```

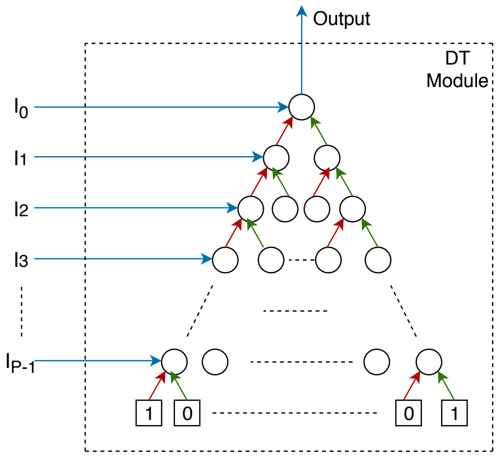
the base-2 exponential of the number of inputs, a 30-input LUT already requires one gigabit of data.

Technically, implementing a N-input LUT is efficient only for small values of N (typically under 12 inside an FPGA). In any case, it is completely unrealistic to implement a LUT for a binary circuit that has more than 40 inputs, which is still far less than the number of inputs in a typical neuron. In order to increase the number of inputs taken into account, we can build several RINC-0 DTs and combine them at higher level by applying a boosting algorithm as described in the next sub-section.

Code extract 4.1.1 illustrates the python code for RINC-0 modules. The *construct_lut* function trains a P-input LUTs. The *best_feature* function is called by *construct_lut* function and chooses the feature that has not been used before and reduces the entropy of the level by the largest extent. This feature is then made unavailable for further levels in the same RINC-0 module. The *assign_class* function assigns the class labels to the leaf nodes. Finally, the chosen features and the leaf node class labels are returned. The *predict* function performs the inference on the constructed p-input LUTs. It takes the chosen feature



(a) LUT-based binary neuron implementation



(b) RINC-0 architecture with DT

Figure 4.1: LUT and its equivalent DT

indices, class labels and the test dataset as inputs. Depending on the selected feature values of each input image, corresponding class label is chosen. This value is compared with the true output in the dataset. It evaluates all the images in the dataset to produce the accuracy of the RINC-0 module.

Listing 4.1: RINC-0 Code Extract

```
#RINC-0 Code Extract###
    def construct_lut(X,y,max_features,weights):
        feature_list = np.arange(np.size(X,1))
        lists = np.arange(np.size(X,0))
        lists_new = np.zeros((1,len(lists)))
        lists_new[0] = lists
        feature_array = np.zeros(max_features)

        for i in range(max_features):
                feature_list_new = \\
                    feature_list[feature_list >=0]
                (best_feat2,lists_new) = \\
                    best_feature(X,y,feature_list_new,i,weights)
                feature_array[i] = best_feat2
                feature_list[best_feat2] = -1

        label_class = assign_class(X,y,lists_new,weights)
        return (feature_array, lists_new, label_class)


    def predict(X_train,y_train,feature_array,label_class,lvl):
        m=lvl
        indices_my = np.arange(2**lvl)
        sel_ind = indices_my[label_class == 1]
        sparse_mat = \\
            np.flip((((sel_ind[:,None] & \\
                (1 << np.arange(m))))>  0),1)
        feature_array = feature_array.astype(int)
        y_predicted = np.zeros(np.shape(X_train)[0])
        for i in range(np.shape(X_train)[0]):
                X_t = (X_train[i,feature_array] == sparse_mat)
                y_predicted[i] = \\
                    np.any(np.sum(X_t,axis = 1) == lvl)
        accuracy = np.sum(y_predicted == y_train) \\
            /np.shape(X_train)[0]
        return accuracy
```

4.1.2 RINC-1: Boosting the MAT units

Even with the modified DT training approach, the RINC-0 modules have low capacity and cannot predict the output of the large binary neuron with sufficient accuracy. Hence, we increase the capacity of the weak DTs by grouping them together using a boosting algorithm. One of the most common boosting algorithms is Adaboost [74], where each weak classifier (a LUT in this case) is trained sequentially and focuses on the misclassified examples of the previous classifier. The Adaboost algorithm is explained in the following sub-section.

Adaboost algorithm

Consider a dataset (x_i, y_i) where i is the i^{th} example from N samples. A weak classifier(k_j) such as a DT is boosted to become a strong classifier using the Adaboost algorithm. The Adaboost is an iterative algorithm where we boost m weak classifiers to work together to form a strong classifier. Initially all the samples are given equal weight$(w_i$ of $\frac{1}{N}$. Hence the sum of weights of all samples is 1. We create the first weak classifier k_1. There are some samples that are correctly classified while others are wrongly classified. Let W_e be the sum of weights of all the wrongly classified samples. It can be represented as

$$W_e = \sum_{i \epsilon (y_i \neq k_j(m))} w_i$$

Using this W_e, we calculate the weight of the weak classifier (α_j) as follows

$$\alpha_j = 0.5 \times ln(\frac{1 - W_e}{W_e}) \tag{4.1}$$

Using this weight(α_j), we reassign the weights(w_i) of each sample. For the correct samples, we reduce the weight by

$$w_i^{(m+1)} = w_i^{(m)} \times \sqrt{\frac{1 - W_e}{W_e}} \tag{4.2}$$

while the samples that have have been incorrectly classified, their weights(w_i) are increased by

$$w_i^{(m+1)} = w_i^{(m)} \times \sqrt{\frac{W_e}{1 - W_e}} \tag{4.3}$$

Then the weights(w_i) are normalized to make the sum equal to 1 again. This weight adjustment allows the next weak classifier to focus more on the samples that have been incorrectly classified by the current classifier and focus less on correctly classified samples. In this way

Adaboost creates a series of weak classifiers that works together to form a strong classifier. The final strong classifier is represented as a weighted sum of the weak classifiers:

$$C_m(x) = \alpha_0 k_0 + \alpha_1 k_1 + \alpha_2 k_2 + \alpha_3 k_3 + ... + \alpha_{m-1} k_{m-1} \tag{4.4}$$

LUT based implementation of the Adaboost algorithm

The hardware for the Adaboost algorithm is detailed in the Multiply-Accumulate and Threshold (MAT) unit shown in Fig. 4.2. Each RINC-0 module is assigned a weight (W_x), x indicates the corresponding RINC-0 module and ranges from 0 to $P-1$. The output of each classifier is multiplied with its respective weight and added. Finally, this weighted sum is thresholded and the binary output is obtained. The architecture is detailed in the MAT unit shown in Fig. 4.2. Each MAT unit theoretically requires P multiplications and $P-1$ additions. Nevertheless, since each MAT module consists of P input bits and one output bit, it can also be implemented in a LUT where we pre-compute the 1-bit output for all possible 2^P inputs combinations. This MAT operation can now be performed as a single look-up operation. Thanks to the addition of the Adaboost layer, the number of inputs to the overall architecture has increased from P to P^2. We denote this module as RINC-1, where-"1" is the number of Adaboost levels. However, the number of inputs to the LUT-based implementation of the MAT module is still limited. Hence, it is not possible to group more than P weak classifiers. To overcome this issue, we propose a hierarchical Adaboost algorithm in the following sub-section.

The Python code extract in 4.1.2 is used to build a RINC-1 classifier. It uses the *construct_lut* and *predict_out_lut* function to train and perform inference on the RINC-0 modules respectively. Each RINC-0 modules is created iteratively and assigned a corresponding weight (*weights*) according to the Adaboost algorithm.

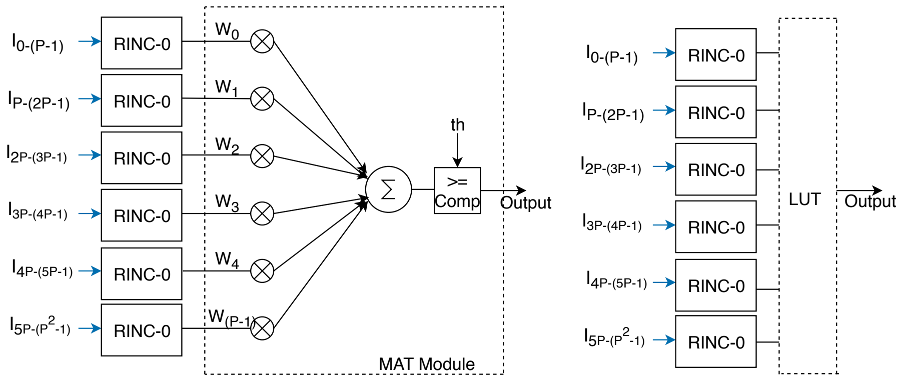


Figure 4.2: RINC-1 architecture with p=6

Listing 4.2: RINC-1 Code Extract

```
    def adaboost_train(X_train,y_train,X_test,y_test, \\
        n_trees,lvl,init_weights):

        weights = np.copy(init_weights)
        label_array = np.zeros((n_trees,2**lvl))
        feats_array = np.zeros((n_trees,lvl))
        new_weights = np.copy(init_weights)
        alpha = np.zeros(n_trees)
        p_outputs_store = np.zeros((n_trees,shape(X_train)[0]))
        p_test_store = np.zeros((n_trees,shape(X_test)[0]))

        for i in range(n_trees):
                (feature_array, lists_new, l_class) = \\
                    lvl_wise_copy2.construct_lut \\
                    (X_train,y_train,lvl,weights)
                label_array[i,:] = l_class
                feats_array[i,:] = feature_array
                predicted_outputs = predict_out_lut \\
                  (X_train,y_train,feature_array,l_class,lvl)
                p_outputs_store[i] = predicted_outputs
                predicted_test_out = predict_out_lut \\
                  (X_test,y_test,feature_array,l_class,lvl)
                p_test_store[i] = predicted_test_out
                cur_error = \\
                sum(weights[where(predicted_outputs!=y_train)])
                new_weights[where(predicted_outputs!=y_train)]
                =0.5*weights[where(predicted_outputs!=y_train)]
                / cur_error
                new_weights[where(predicted_outputs==y_train)]
                =0.5*weights[where(predicted_outputs==y_train)]
                / (1 - cur_error)
                weights = new_weights

        return p_test_store,p_outputs_store, \\
            alpha,label_array,feats_array
```

4.1.3 RINC-L: Hierarchical Adaboost algorithm

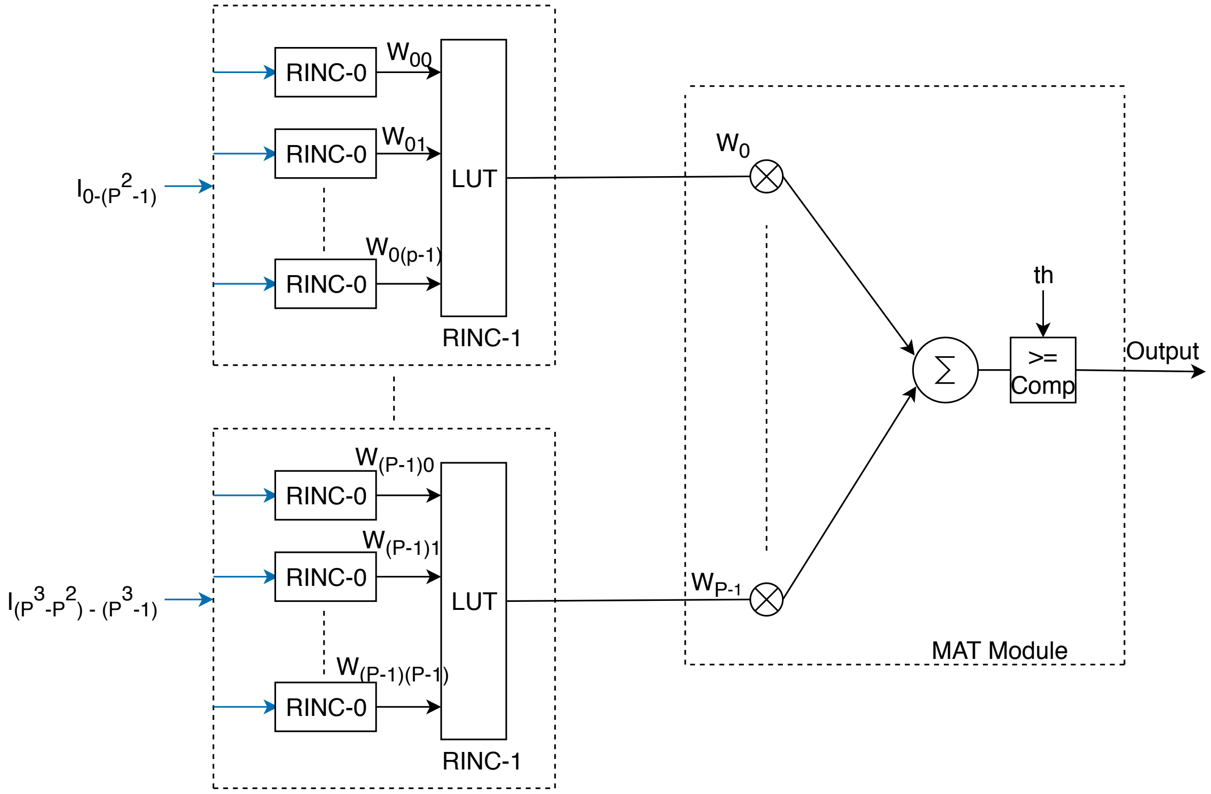


Figure 4.3: RINC-2 architecture

In FPGAs with 6-input LUTs, even with P^2 inputs, the RINC-1 module can accommodate only 36 inputs which is highly insufficient. Hence, we propose a hierarchical Adaboost algorithm to increase the capacity. Firstly, we build a RINC-1 module with P DTs. We also call it a subgroup in this architecture. In each subgroup, the weights are represented as w_{xy} where y indicates the index of a RINC-0 module and x indicates the sub-group number as shown in Fig. 4.3. This subgroup is considered a weak classifier. Using the Adaboost algorithm, we construct up to P subgroups. Each subgroup is assigned a weight (W_x), where x indicates the sub-group number. Hence, this creates two levels of Adaboost, one within the subgroup and second across subgroups. The binary outputs of the subgroups are multiplied, added and thresholded in a MAT module. Again this module can be implemented as a LUT. We can observe from Fig. 4.3 that adding another level of Adaboost increases the number of RINC-0 modules exponentially, thus accommodating P^{L+1} inputs. In a hierarchical Adaboost algorithm with L levels and p inputs per LUT, there are p^L RINC-0 modules and

$\sum_{l=0}^{L-1} p^l$ Look-Up based MAT modules. Thus,

$$LUTs\ required = p^L + \sum_{l=0}^{L-1} p^l = \sum_{l=0}^{L} p^l = \frac{P^{L+1} - 1}{P - 1} \tag{4.5}$$

Algorithm 3 details the hierarchical Adaboost algorithm. We create groups of P DTs together. Each DT is associated with a weight, that is multiplied with the corresponding output of the DT and thresholded. Now we consider this P group of DTs as a weak classifier and assign a new weight to each group. We further build such groups of DTs and assign a weight to each of them according to the Adaboost algorithm. Again, a MAT module is required to group these sub groups and it is implemented as a LUT. This can be viewed as 2 levels of Adaboost. Considering this RINC-2 as weak classifier, we can build sub-groups of RINC-2 classifier and group them together using a MAT module to build a RINC-3 architecture. Similarly, further levels of hierarchical Adaboost can be added to build a RINC-L architecture.

The Python code for RINC-2 architecture is illustrated in 4.1.3. The *rinc_2* function takes the trainset and testset as inputs. The *adaboost_train* function called in the *rinc_2* function creates a RINC-1 module. We build $\frac{total_estimators}{inp_DT}$ such RINC-1 modules, where *total_estimators* refers to the total number of decision trees required per neuron, and *inp_DT* refers to the value P. The corresponding weights for each of the RINC-1 modules are calculated in the *new_init_weights*. The RINC-2 MAT module weights are calculated in the *alpha_next_level* variable. The function returns the predicted outputs of the RINC-2 module for the trainset and testset. This function is called once each for a neuron. Moreover, given sufficient computation and memory resources, multiple instances of this function can be run in parallel.

Algorithm 3 RINC-L: Hierarchical Adaboost training algorithm

```
Input: data $X$, size $n \times F$
for $l \rightarrow 1$ to $L$ do
    Construct $P$ RINC-$(l-1)$ classifiers
    Multiply their outputs with corresponding weights
    Sum and threshold the result
    Encode the MAT operation as LUT
    Consider these P RINC-$(l-1)$ classifiers as a single weak classifier
    Assign the new weights to each training example
Output: Thresholded result of the final MAT operation
```

Listing 4.3: RINC-2 Code Extract

```
def rinc_2(X_train,X_test,y_train,y_test):

    for i in range(int(total_estimators/inp_DT)):
        BNNout_temp,BNNout_train_temp,alpha,label_array, \\
        feats_array = adaboost_v1.adaboost_train(X_train,\\
        y_train,X_test,y_test,inp_DT,inp_DT,init_weights)
        BNNout = np.transpose(BNNout_temp)
        BNNout_train = np.transpose(BNNout_train_temp)
        cur_weights = np.zeros((hyp_inp,1))
        cur_weights[:,0] = alpha
        sum_cur = np.sum(cur_weights)
        th = sum_cur * 0.5
        F_next_lvl_test[:,i] = \\
        np.matmul(BNNout,cur_weights)[:,0] > th
        F_next_lvl_train[:,i] = \\
        np.matmul(BNNout_train,cur_weights)[:,0] > th
        cur_error = np.sum(init_weights[y_train != \\
        F_next_lvl_train[:,i]])
        new_init_weights[where(F_next_lvl_train[:,i] \\
        != y_train)] = 0.5  * init_weights[where \\
        (F_next_lvl_train[:,i] != y_train)] / cur_error
        new_init_weights[where(F_next_lvl_train[:,i] \\
        == y_train)] = 0.5  * init_weights[where \\
        (F_next_lvl_train[:,i] == y_train)] /(1-cur_error)
        init_weights = new_init_weights

    all_weights = np.zeros((total_estimators/inp_DT),1)
    all_weights[:,0] = alpha_next_level
    sum_aw = np.sum(all_weights)
    th = sum_aw / 2
    BNNout_f = matmul(F_next_lvl_test,all_weights)
    BNNout_f_train = matmul(F_next_lvl_train,all_weights)
    BNNout_f_bin = BNNout_f >  th
    BNNout_f_train_bin = BNNout_f_train > th
    BNNacc = np.sum(y_test == BNNout_f_bin[:,0]) \\
    / shape(BNNout_f_bin)[0] *100.0;
    return BNNout_f_train_bin[:,0],BNNout_f_bin[:,0]
```

4.2 Binary to multiclass classification

The RINC-L modules can be used to implement any tiny binary neuron in the network. However, RINC-L being a binary classifier, it cannot be used directly for multiclass classification. Traditionally, multiclass classification using DTs have been solved using two approaches, Multiclass DTs [46] and One-vs-all classifications [75]. Modifying the RINC architecture as Multiclass DT makes it expensive to implement on hardware. In Multiclass DTs the leaf nodes refer to either one of the n_c classes, where n_c is the number of classes. This requires each DT to be represented over $\log_2 n_c$ bits. Therefore, the RINC-0 and MAT modules cannot be confined to a single LUT. This effect would cascade over the entire architecture and would make it inefficient. Also, we do not consider one-vs-all classifications using Binary DTs as there is a large drop in accuracy between each individual binary classifier and the overall multi-class classifier designed by comparing the confidences [43]. Moreover, the one-vs-all classification also requires a confidence comparison circuit that consumes more resources. Hence, it is imperative to look for other alternatives to use our RINC classifiers for multiclass classification.

On the other hand, fully connected layers have been successful in multiclass classification than DTs. Hence, we formulate a combined approach with our RINC architecture and fully connected layer to overcome the multiclass challenge. We preserve the output fully connected layer, while replacing the hidden layers in the classifier with our RINC-L architecture. We then adapt the output fully connected layer to work in tandem with the RINC-L architecture for multiclass classification. In a way, we break the task of the classifier into two parts, first the binary classification of the hidden layer representations (sub-section 4.2.1) and then the multiclass classification of the output layer (sub-section 4.2.2).

4.2.1 Replacing the hidden layers

We use a back-to-front approach where we start replacing the binary neurons in the network with our RINC-L architecture from the last layers and progressively move towards the initial layers. Given sufficient capacity of the RINC-L modules, it is possible to replace multiple layers of the neural network with a single RINC-L module. In order to optimize the number of RINC-L modules required, we add a fully connected layer with binary sigmoid activation [76] after the last hidden layer. We call this fully connected layer, intermediate layer. It consists of $n_c \times P$ neurons, where n_c is the number of classes in the multiclass classification. Typically, the value of P ranges from 6 to 8 and the value of n_c is 10 for the MNIST, CIFAR-10 and SVHN datasets. Hence, the intermediate layer has 60-80 neurons for these datasets. Thus,

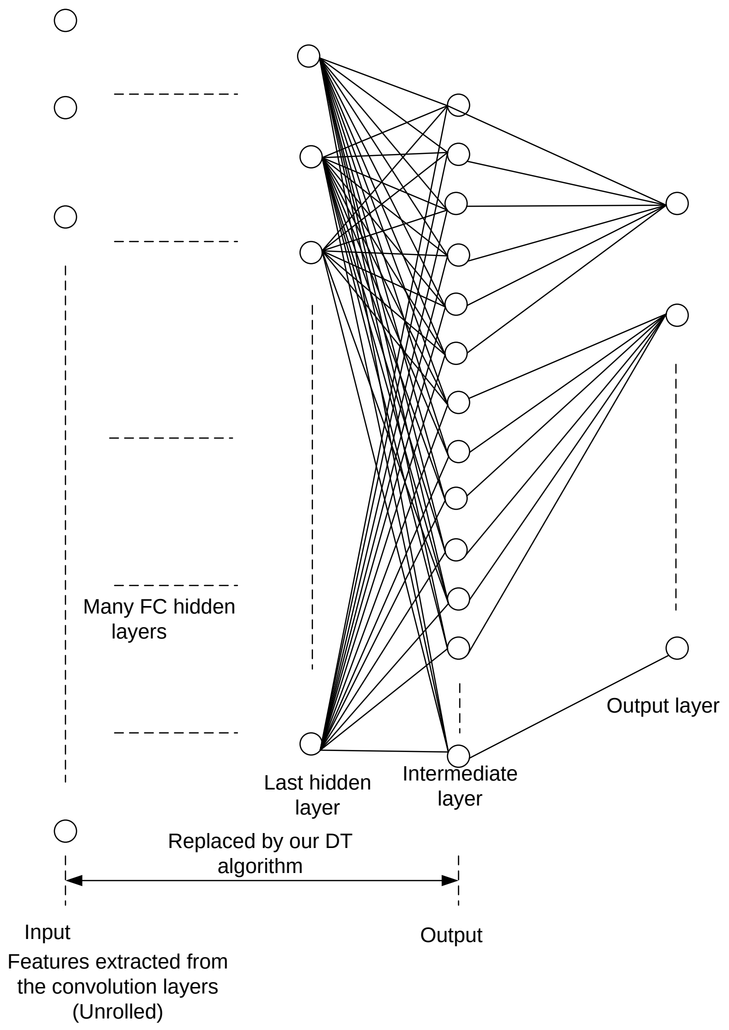

Many FC hidden layers
Last hidden layer
Intermediate layer
Output layer
Replaced by our DT algorithm
Input
Features extracted from the convolution layers (Unrolled)
Output

Figure 4.4: Intermediate layer

the intermediate layer with binary sigmoid activation can be viewed as a set of binary neurons. This enables us to train a RINC-L module to emulate a binary neuron representation in the intermediate layer. Similarly, binary sigmoid activation can be introduced in the earlier layers and can be replaced with RINC-L modules. Since there are fewer neurons in the intermediate layer than in the hidden layers, it takes fewer resources to train a RINC-L classifier for each of the neuron in the intermediate layer. On the other hand, this restricts the representation space. Hence, the hyper parameter P must be chosen carefully to balance the trade-off between accuracy and resources.

4.2.2 Sparsely connected output layer

The outputs of the RINC-L modules (emulating the intermediate layer representation) are connected to the output layer. However, this output layer needs to be optimized for LUT based implementation. Firstly, we modify the output layer to be sparsely connected to the intermediate layer. Each neuron in the output layer is connected to only P neurons of the intermediate layer as shown in Fig. 4.4. Hence, each neuron depends on P inputs and therefore can be implemented as a single Look-Up operation. Also, the output layer can be seen as a small set of $P-$input fully connected layers stacked in parallel. Since the output layer is trained as a fully connected network, it inherits all the properties of neural networks to classify multiclass data effectively. The output layer is separately retrained with RINC-L outputs to adapt the weights of the output layer. The retraining of the sparsely connected output layer for multiclass classification adapts the weights to the RINC-L binary hidden layer representations.

The output layer activation is not binary. Nevertheless, since it is sparsely connected to the previous layer, it can still be efficiently implemented with LUTs. Each output neuron is quantized to q-bits precision, thus requiring q LUTs per output neuron. Therefore, the output layer can be implemented using $q \times n_c$ LUTs. This is negligible as compared to the resources required by the RINC-L modules. Thus, with the help of few more LUTs we achieve higher accuracy on multiclass classification than using costly multiclass DTs or One-Vs-All classifiers. This final architecture with multiple RINC-L modules and q-bit quantized output layer implemented as LUTs make the announced PoET-BiN.

4.3 Parser to generate VHDL code

Once the architecture of the network is decided and the accuracy obtained, we need to implement it on a embedded system. FPGA with its programmable logic provides an ideal platform to implement our architecture. However, our architecture can contain tens of thousands of LUTs. It is impractical to manually code each LUT. Hence we designed parsers that take in the features, class labels and weights from the Python files to generate the VHDL code. Code 4.3 illustrates the parser for RINC-0 modules. First, we generate the intermediate signals then we generate the actual LUT. Similarly Code 4.3 is a parser for the MAT modules. Code 4.3 illustrates a parser for the output layer. While code 4.3 is a parser to generate the VHDL testbench to compare the outputs generated by the VHDL module and Python.

Listing 4.4: Parser for RINC-0 modules

```
### Signal RINC L1###
for i in range(np.shape(RINC_0_feats)[0]):
        for j in range(np.shape(RINC_0_feats)[1]):
                for k in range(np.shape(RINC_0_feats)[2]):
                        f.write('signal C_'+str(i)+'_S_'+str(j)
                        +'_L_'+str(k)+'_out : std_logic := \\
                        \'0\'; \n')
f.write('\n')
### Signal RINC L2###
for i in range(np.shape(RINC_0_feats)[0]):
        for j in range(np.shape(RINC_0_feats)[1]):
                f.write('signal C_'+str(i)+'_S_'+str(j)+ \\
                '_out : std_logic := \'0\'; \n')
f.write('\n')
### First level of RINC ###
for i in range(np.shape(RINC_0_feats)[0]):
        for j in range(np.shape(RINC_0_feats)[1]):
                for k in range(np.shape(RINC_0_feats)[2]):
                        f.write('C_'+str(i)+'_S_'+str(j) \\
                        +'_L_'+str(k)+'_inst : LUT8 generic map
                            (INIT => "')
                        for l in range(shape(RINC_0_labels)[3]):
                                f.write(\\
                                str(RINC_0_labels[i,j,k,shape \\
                                (RINC_0_labels)[3] - l - 1]))
                        f.write('") port map( O =>' + \\
                        'C_'+str(i)+'_S_'+str(j)+'_L_'+str(k) \\
                        +'_out')
                        for l in range(shape(RINC_0_feats)[3]):
                                f.write(', I' + str(l) + ' => \\
                                inp_feat(' + \\
                                str(RINC_0_feats[i,j,k,shape \\
                                (RINC_0_feats)[3]-l-1]) + ')')
                        f.write('); \n')
f.write('\n');
```

Listing 4.5: Output layer parser

```
for i in range(10):
        for j in range(8):
                f.write('C_'+str(i)+'_B_'+str(j)+'_inst : \\
                LUT8 generic map(INIT => "')
                ###LUT 8###
                for k in range(2**np.shape(MAT_weights)[1]):
                        f.write(str(MAT_labels_bin[i,\\
                        shape(MAT_labels_bin)[1] - k - 1,j]))
                        ###In reverse order###
                f.write('") port map( O =>' + 'C_'+str(i)\\
                +'_B_'+str(j)+'_out')
                for k in range(np.shape(MAT_weights)[1]):
                        f.write(', I' + str(k) + ' => \\
                        C_'+str(i*8 + k)+'_out')
                        ###Inputs Linearly###
                f.write('); \n')
        f.write('\n');

f.write('\n');
```

Listing 4.6: Testbench parser

```
for i in range(100):
        f.write('inp_feat <= \"' );
        for j in range(np.shape(inp_data)[1]):
                f.write(str(inp_data[i, \\
                shape(inp_data)[1] - j - 1]))
        f.write('\";')
        f.write('cor_in <= \"' )
        for j in range(np.shape(out_data)[1]):
                f.write(str(out_data[i,j]))
        f.write('\" ; wait for 10 ns; \n');
```

Listing 4.7: MAT module parser

```
###Printing the signals###
for i in range(80):
        f.write('C_' + str(i) + '_out <= inp_feat(\\
        ' + str(i) + '); \n')
f.write('\n');
for i in range(80):
        f.write('signal C_' + str(i) + \\
        '_out : std_logic := \'0\'; \n ')
f.write('\n');
for i in range(np.shape(MAT_labels)[0]):
    f.write('C_'+str(i)+'_inst : LUT8 generic map(INIT =>"')
    for l in range(np.shape(MAT_labels)[2]):
        f.write(str(MAT_labels[i,-1,\\
        shape(MAT_labels)[2] - l - 1]))
    f.write('") port map( O =>' + 'C_'+str(i)+'_out')
    for l in range(np.shape(RINC_0_feats)[3]):
        if l < (np.shape(MAT_labels)[1] - 1):
                f.write(', I' + str(l) + ' => \\
                C_'+str(i)+'_S_'+str(l)+'_out')
        else:
                f.write(', I' + str(l) + \\
                ' => \' 0 \' ')

f.write('); \n')
```

This chapter introduces our LUT based approach to build neural networks. Traditionally neural networks can be expressed as multiplication, addition and other non-linear activation operations. We replace these operations as LUT access, which consumes less power, area and is faster then multiplication or addition operation. We propose a method to train these LUTs using the modified Adaboost algorithm and also propose a hierarchical Adaboost algorithm to group these LUTs together making a strong binary classifier. We also proposed a mixed DT and neural network approach using LUTs for multiclass classification. In the next chapter we will look at using these techniques to build CNN networks to solve image classification tasks on the MNIST, CIFAR-10 and SVHN datasets.

CHAPTER 5 EXPERIMENTAL SETUP, RESULTS AND DISCUSSIONS

5.1 Experimental setup

In this section we detail how we train the proposed architecture and test its performance with various datasets.

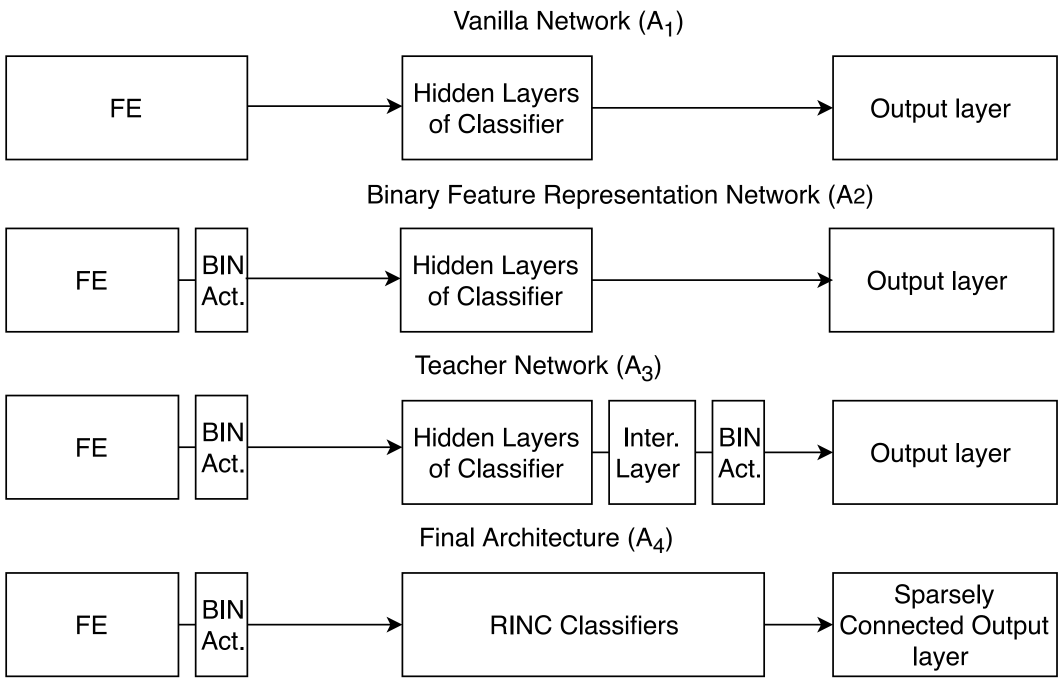


Figure 5.1: Overall work flow

We developed the workflow shown in Fig. 5.1 to train the RINC modules starting from a Vanilla CNN network. Firstly, we use a pretrained full precision CNN as our base architecture (Vanilla network). The base architecture for each dataset is mentioned in Table 5.1. FE refers to the feature extractor which consists of convolutional, maxpooling and activation layers. In the vanilla network, the features are represented with full precision. However, RINC modules can only be trained on binary features. Hence, we replace the ReLU with binary sigmoid activation after the last convolutional layer to obtain the binary features. This is represented by the Bin act. module in the Binary Feature Representation Network in Fig. 5.1. Further, an intermediate layer and a binary sigmoid activation are added after the last hidden layer. This forms the teacher network. Then, we replace all the hidden layers and the intermediate layer in the classifier using our RINC architecture which is the student architecture in our work. Finally, the output layer is retrained with the RINC outputs. The output layer activations are quantized to q bits for efficient hardware implementation. In our test it was observed that when $q = 4$, the loss in accuracy was quite significant as compared to the original floating point implementation. On the other hand, with $q = 8$ the loss in accuracy was minimal. In the case when $q = 16$ the accuracies were similar to that of 8-bit quantization but it requires twice the amount of LUTs as compared to 8-bit quantization. Hence we use 8-bit quantized output layer.

The architecture hyperparameters are listed in Table 5.1 and explained in detail in the following sub-sections for each dataset. We use techniques such as batch normalization [77], exponentially decreasing learning rate, squared hinge loss [78] and ADAM optimizer [79] in all the vanilla networks. Also, we do not retrain with the validation set. We do not use any image augmentation techniques except for padding in CIFAR-10.

Table 5.1: Network architecture

Architecture (Arch.)	Symbol	Dataset
$LeNET_{FE} - (512FC) - (10FC)$	M1	MNIST
$VGG11_{FE} - (4096FC) - (4096FC) - (10FC)$	C1	CIFAR-10
$VGG11_{FE} - (2048FC) - (2048FC) - (10FC)$	S1	SVHN

5.1.1 MNIST

As seen in Table 5.1, we use the LeNet architecture for the MNIST dataset. Using two convolutional layers of 5×5 convolutions and two pooling layers of size 2×2, we transform the feature space to 512 binary features. The classifier portion consists of only one hidden fully connected layer of 512 neurons with ReLU activation and an output layer of 10 neurons. We use 8-input LUTs ($P = 8$). Hence, the intermediate layer contains $10 \times 8 = 80$ neurons. We train a 2-level RINC (RINC-2) module with 32 DTs for each neuron in the intermediate layer. Therefore each RINC-2 module selects a maximum of 256 ($= 32 \times 8$) features from the available 512 binary features. These predicted outputs of the RINC-2 modules are used to retrain the final $8-$bit quantized output layer.

5.1.2 CIFAR-10

The CIFAR-10 training procedure is similar to the one used for MNIST, but for a bigger network, i.e. a VGG-11 architecture with 8 convolution layers and 3 fully connected layers. The full precision implementation proposed by Kuang [80] is used as reference. The convolutional layers transform the input to a binary feature space of 512 features. There are 2 hidden fully connected layers with 4096 neurons each. To augment the capacity of RINC modules, we use 8−input LUTs ($P = 8$) and 40 DTs for each of the neuron in the intermediate layer. The intermediate layer consists of $8 \times 10 = 80$ binary neurons. The predicted outputs of the RINC-2 modules are used to retrain the output layer which is quantized to 8 bits.

5.1.3 SVHN

SVHN is implemented with an architecture similar to the one used for CIFAR-10, except that LUTs have 6 inputs ($P = 6$), leading to 36 DTs per neuron with 2 hierarchical levels (RINC-2) of Adaboost. We also use the extra dataset from the SVHN dataset for training.

5.2 Results and Discussions

We analyse the accuracy, power consumption and latency of PoET-BiN for the MNIST, CIFAR-10 and SVHN datasets.

5.2.1 Classification accuracy

We report four sets of accuracies for each dataset in Table 5.2. Firstly, we report the accuracy of the Vanilla network (A_1), followed by the accuracy with binary sigmoid activation after the last convolutional layer to obtain the binary features (A_2). Then we report the accuracy after binarizing the intermediate layer (A_3). This forms the teacher network. Finally, we replace the classifier portion of the teacher network with the RINC classifiers and quantize final layer whose accuracy is reported as (A_4). This helps isolate and study the effect of each modification. We report the best accuracy achieved over different sets of hyper-parameters such as number of DTs and LUT size for RINC modules. Fig. 5.1 illustrates the progressive modifications with relevant accuracies.

Table 5.2: Overall classification accuracy on MNIST, CIFAR-10 and SVHN dataset

Arch.	Dataset	A_1(%)	A_2(%)	A_3(%)	A_4(%)
M1	MNIST	99.20	99.06	98.93	98.15
C1	CIFAR-10	91.02	89.88	89.10	92.64
S1	SVHN	97.36	96.98	96.22	95.13

In Table 5.2 , we observe a drop in accuracy of approximately 0.3% for MNIST, 1.9% for the CIFAR-10 dataset and 1.1% for the SVHN datasets between the Vanilla and teacher network ($A_1 - A_3$). This is expected as we restrict the feature space by using binary representations.

This teacher network is used to train our RINC classifiers and quantized sparsely connected output layer. This results in a further dip in accuracy of 0.8% for MNIST and 1% for SVHN. An interesting observation in the case of CIFAR-10 is that by replacing the fully connected layers with PoET-BiN, the accuracy improves by 1.5% for CIFAR-10. This anomaly could be due to better generalization as a result of the noise injected into the system due to the inaccuracies in intermediate layer prediction by the RINC modules. Similar observations were seen in Dropconnect [42].

Table 5.3: Comparison with other techniques

Implementation	Accuracy (%)		
	MNIST	CIFAR-10	SVHN
BinaryNet [31]	98.97	89.76	95.06
POLYBiNN [43]	97.52	91.58	94.97
NDF [47]	99.42	90.46	95.20
Our work	98.15	92.64	95.13

Now that we have obtained the final accuracy of the PoET-BiN architecture, it is necessary to have a fair comparison with other architectures present in the literature. We choose three starkly different architectures, namely BinaryNet [31], POLYBiNN [43] and Neural Decision Forest (NDF) [47]. BinaryNet is a quantized fully connected layer approach, while POLYBiNN is a complete Decision Tree approach and NDF is a hybrid mixture of both with differentiable DTs. To ensure fairness we use the same feature extractor across all architectures, and change the classifier portion of the architecture. We used our Python implementation for BinaryNet and POLYBiNN. While, Jing's Pytorch implementation of NDF [81] was adapted for the comparative analysis. From Table 5.3, we can see that our architecture performs the best in the case of CIFAR-10 and second best in case of SVHN. Though the NDF architecture performs better than PoET-BiN on MNIST and SVHN, it is not optimized for hardware implementations.

For MNIST, the significant reduction in accuracy can be overcome by increasing the number of RINC classifiers. In the MNIST architecture, rather than training the RINC classifiers to predict the intermediate layer outputs, we can train a RINC classifier for each of the neuron in the only hidden layer in M1 architecture. This results in 512 RINC-2 modules. Retraining the fully connected output layer with the 512 RINC classifier outputs results in an accuracy of 98.62% that is more closer to that of NDF. However, this implementation consumes more resources. Therefore, we do not consider this accuracy. However, it proves the versatility of the RINC architecture in implementing binary neurons. We do not implement similar architectures for SVHN and CIFAR-10, as they have significantly more neurons in the last hidden layer (2048 for SVHN and 4096 for CIFAR-10), requiring long training times.

Another important observation to be noted in Table 5.3 is that the PoET-BiN architecture performs better than off-the-shelf DTs used in POLYBiNN across all datasets, in spite of them having significantly more nodes in each DT. This can be attributed to our hierarchical training algorithm and unique binary to multiclass classification technique.

Table 5.4: Comparison with other works in the literature

Architecture	Accuracy (%)		
	MNIST	CIFAR-10	SVHN
Our work	98.15	92.64	95.13
Ternary Weighted Networks [82]	99.35	92.56	-
Gated XNOR-Net [37]	99.32	92.50	97.37
XNOR-Net [36]	-	91.12	-
BinaryConnect [30]	98.71	90.10	97.70
BinaryNet [31]	98.60	89.88	97.47
TNN [83]	98.33	87.89	97.27
TrueNorth [53]	92.70	83.41	96.66
POLYBiNN [43]	97.18	81.1	-
FINN [48]	95.8	80.1	94.9
Deep Forest [44]	99.26	65.67	-

In Table 5.4 we compare the accuracy of our architecture with other works in the literature. It shows that our network has the best accuracy for the CIFAR-10 dataset among the compared works. The base architecture adapted from Kuang [80] achieves a high accuracy on the CIFAR-10 dataset. This effect trickles down till the final architecture. In the case of the MNIST dataset, we can see that our network achieves a comparable accuracy with respect to other quantized implementations. On the other hand for the SVHN dataset, our network under performs. This could be because our base floating point architecture itself was not state-of-the-art. We didn't use state-of-the-art architectures for the SVHN dataset as those networks entailed complex mathematical operations to help boost the accuracy. Using such networks would have lead to inefficient hardware implementations. It must be noted each of these networks compared have different architectures. For example, the BinaryConnect and BinaryNet architectures use AlexNet for CIFAR-10 classification, while XNOR-Net uses a Resnet architecture and we use a VGGNet architecture. On the other hand, FINN uses a custom architecture, TrueNorth uses Spiking neural networks and Deep Forest does not use any convolutional layers.

Table 5.5: RINC power results

Data set	MNIST	CIFAR-10	SVHN
Dynamic(W)	0.468	0.300	0.374
Static(W)	0.045	0.041	0.043
Total(W)	0.513	0.341	0.417

5.2.2 Power

One of the most important metrics apart from accuracy for hardware implementations of neural networks is power. We compare the power consumed by the PoET-BiN architecture to the classifier portion of the Vanilla neural network and quantized neural networks. We implement our PoET-BiN architecture based classifier on Spartan-6 45-nm FPGA from Xilinx to calculate the power consumption. Typically, the PoET-BiN architecture used for the three datasets consists of thousands of LUTs. It is quite cumbersome to write a HDL script for such a large implementation. Hence, we developed a Python script to generate the VHDL code automatically from the trained LUTs. The retrained final layer is also implemented on the FPGA automatically by our script. To report the power of this architecture, it has to fit in the FPGA. The Spartan-6 FPGA has 276 input/output ports but our architecture has 512 features. Hence the inputs are provided through a shift register with a single input. This enables us to fit the architecture into the target FPGA. However, this also adds some logic and signal power which is 4 mW in the case of MNIST and CIFAR-10 and 6 mW in the case of SVHN. The power consumed by the PoET-BiN architecture (subtracting the power consumed in the shift registers) are reported in Table 5.5. The outputs generated by the FPGA and those generated by PyTorch are verified in the testbench, that is automatically produced by another Python script.

We compare the power consumed to that of the Vanilla neural network and quantized neural networks. Most of the hardware implementations of neural network in the literature provide the power consumed for the entire network including the convolutional layers. It is difficult to accurately estimate the power consumed in the fully connected layers from this data. Hence we use a bottom-up approach to estimate the power of the classifier portion of these networks. Mathematical operations (multiplication and addition) and memory fetching operations consume most of the power in the fully connected layers. Power consumption of memory fetching operations depends on the evaluation platform, memory type and other factors. Hence, it is quite difficult to estimate the power required for memory fetching operations accurately without an actual implementation. On the other hand, we can estimate the power required for the mathematical operations. First, we implement a single multipli-

Table 5.6: Single arithmetic operation power results

Operation (at 62.5 Mhz)	Dynamic (W) clock	logic	signal	IO	Static (W)	Total (W)
Multiplication (16 bits)	0.001	0.001	0.000	0.020	0.036	0.058
Addition (16 bits)	0.001	0.000	0.001	0.024	0.036	0.062
Multiplication (32 bits)	0.002	0.001	0.001	0.035	0.037	0.076
Addition (32 bits)	0.001	0.000	0.002	0.048	0.037	0.088
Multiplication (FP)	0.005	0.006	0.005	0.046	0.037	0.098
Addition (FP)	0.004	0.003	0.005	0.034	0.037	0.083

Table 5.7: Total arithmetic operations

Operation	MNIST	CIFAR-10	SVHN
Addition	267,264	18,915,328	5,263,360
Multiplication	267,264	18,915,328	5,263,360

cation and an addition on the same FPGA. Table 5.6 provides the power consumption for a single multiplication and an addition operation. The multiplication is implemented with a Digital Signal Processor (DSP) block in the FPGA, which consumes less power compared to a LUT-based implementation. The addition operation is implemented with LUTs and dedicated carry chains. We use IP cores provided by Xilinx to implement the multiplication and addition. Now, that we know the power consumed by each operation, we calculate the total number of multiplications and additions in each of the fully connected layers of the Vanilla architecture Table 5.7. From these data we estimate the power consumed in the classifier portion of the Vanilla and quantized neural networks except binary quantized networks.

In the case of binary quantized neural network, each multiplication or addition operation consumes an insignificant amount of power. Hence, we estimate the power consumption by implementing a binary neuron. Each binary neuron consists of multiple binary multiplications (XNOR operation) followed by a tree structured adder [84] and a comparator. We then multiply this value with the number of neurons in the classifier portion of the respective network for each dataset to estimate the total power consumption of classifier portion of each network. In the case of MNIST, each binary neuron consumes 34 mW of logic + signal power. However, this includes the power consumed by two shift registers that are used to feed in the values for the inputs and weights. Each shift register consumes $4mW$ of power that needs to be subtracted from the total power. Hence, the power consumed by the binary neuron portion alone is $34 - 4 - 4 = 26$ mW. There are 522 binary neurons in the classifier portion of the M1 architecture. Therefore the total dynamic power consumed by all the binary neurons is $26 * 522 = 13.572\ W$. This value is multiplied by the time period of the clock (16 ns) to obtain the energy shown in Table 5.8. Similarly, we estimate the energy in the classifier portion of binary quantized networks for the CIFAR-10 and SVHN datasets.

Such a method to estimate the power has the advantage of considering the same target device for all the estimations as illustrated in Table 5.6. Other works use the metrics proposed by Horowitz in [50] but we have not been able to find a fair estimation of a small LUT. Moreover, the power analyzer gives a detailed report on the power distribution in the FPGA. The total power can be coarsely divided into static and dynamic power. The static power, as the name suggests, is constant for a given FPGA device. The dynamic power can be further sub-divided into clock, logic, signal and IO power. The clock and IO power are also constant for a given FPGA device at a given frequency of operation. Hence, the actual energy involved in the computation of a combinational function is only concerned by the *logic* and *signal* columns of Table 5.6. Therefore, we only use these values to estimate the power of a given architecture.

Along with power, energy is also an important metric to be taken into consideration. The energy is calculated in Table 5.8 from the power values mentioned in Table 5.5 and Table 5.6. To calculate the energy value we use the time period of the clock. Our PoET-BiN classifier requires single cycle to implement the inference. For the SVHN dataset, we use a RINC-2 classifier with $P = 6$ that is easily implementable on Xilinx LUTs as they support 6 input LUTs. Hence, we use a 100 MHz clock for the RINC-2 classifier implementation for SVHN. On the other hand, MNIST and CIFAR-10 require RINC-2 classifiers with $P = 8$. As each 8-input LUT requires four $6-$input LUTs, the critical path increases. Therefore, we use a slower clock with a frequency of 62.5 MHz. We can increase the frequency of the implementation by pipelining the architecture, but this will lead to more power consumption due to the extra registers. Hence, we stick to single cycle implementations. Using these information, the energy results are calculated and detailed in Table 5.8. We observe that the PoET-BiN architecture as compared to a full precision vanilla network consumes 1×10^4 times less energy in the case of MNIST, almost 1×10^6 for CIFAR-10 and 4×10^5 in the case of SVHN. Even in the case of 16-bits quantized network, the PoET-BiN architecture consumes almost 1×10^3 less energy in MNIST, 1×10^5 in CIFAR-10 and 2.5×10^4 in the case of SVHN. Comparing the PoET-BiN to 1-bit quantization (binary), we observe our architecture consumes $25\times$ less energy in the case of the MNIST dataset, 7×10^3 less energy for the CIFAR-10 dataset and 2×10^3 less in the case the SVHN dataset.

Actually, these values are the worst case scenario of power reduction since we do not consider the power required for memory fetching operations in vanilla and quantized neural networks. These operations are $10\times$ more power intensive than multiplication operations [50]. On the

Table 5.8: Energy consumption

Technique	Energy (J)		
	MNIST	CIFAR-10	SVHN
Vanilla	8.0×10^{-5}	5.7×10^{-3}	1.6×10^{-3}
1-bit Quant	2.1×10^{-7}	3.9×10^{-5}	9.2×10^{-6}
16-bit Quant	8.5×10^{-6}	6.0×10^{-4}	1.0×10^{-4}
32-bit Quant	1.7×10^{-5}	1.2×10^{-3}	3.6×10^{-4}
PoET-BiN	8.2×10^{-9}	5.4×10^{-9}	4.1×10^{-9}

other hand, the PoET-BiN architecture does not need any memory access operations. Hence, in reality our architecture is even more power efficient than what we report here.

5.2.3 Latency and Area

Apart from accuracy and power, embedded application are often time critical and demand low latency. Even with a single cycle implementation, the PoET-BiN architecture has a low latency. From Table 5.9, we can observe that the latency is 5.85 ns in the case of SVHN, while for CIFAR-10 and MNIST it is 9.48 ns and 9.11 ns respectively. This translates to a throughput of up to 166 M images per second in the case of SVHN and 100 M images per second in the case of MNIST and CIFAR-10.

Along with accuracy, power and latency, the area required to implement the inference architecture is an important metric. The PoET-BiN architecture stands out from other inference techniques such as quantization or pruning in this regard as our implementation is focused on making each sub operation fit a single LUT. This yields a highly optimized architecture in terms of area as seen in Table 5.9. Especially in the case of SVHN, the PoET-BiN architecture with $P = 6$ and 2 levels (RINC-2) requires 2660 LUTs. This can be verified with manual calculations as follows. Firstly, each RINC-0 module requires a single LUT. Then, a RINC-1 module with $P = 6$, consists of 6 RINC-0 modules and a MAT module, hence requiring $6 + 1 = 7$ LUTs. A RINC-2 module consists of 6 RINC-1 module and a MAT module, thus requiring $7 * 6 + 1 = 43$ LUTs. Sixty such RINC-2 modules are required to emulate the intermediate layer, therefore consuming $43 * 60 = 2580$ LUTs. The final output layer consists of 10 neurons whose values are quantized to 8 bits. Hence each neuron in the output layer requires 8 LUTs. Therefore, $2580 + 80 = 2660$ LUTs are required to implement the classifier architecture for SVHN. This is the exact count given by the Xilinx synthesizer as well. As there are no overlaps between inputs in each DT, the Xilinx synthesizer cannot further simplify the design. This supports the idea that our training algorithm produces a

Table 5.9: Implementation results

Data set	MNIST	CIFAR-10	SVHN
Latency(ns)	9.11	9.48	5.85
LUTs	11899	9650	2660

highly efficient implementation.

In the case of MNIST and CIFAR-10 we use 32 and 40 DTs per RINC-2 module, respectively, with $P = 8$. Since Xilinx LUTs have a maximum of 6 inputs, each $8-$input LUT requires four $6-$input Xilinx LUTs. Sometimes, this conversion results in redundancy and the synthesizer removes a few LUTs that do not affect the result. Further analysis reveals that most of the LUTs removed by the synthesizer are MAT modules. This is because some DTs have a very low weight assigned by the Adaboost algorithm, which finally does not affect the result of the MAT operation. This suggests that it is possible to further improve our training algorithm. Such opportunity is predominately visible in the case of CIFAR-10 where approximately 36% of the LUTs are removed by the synthesizer producing a smaller architecture in spite of having more DTs per RINC-2 module as compared to the PoET-BiN implementation of MNIST.

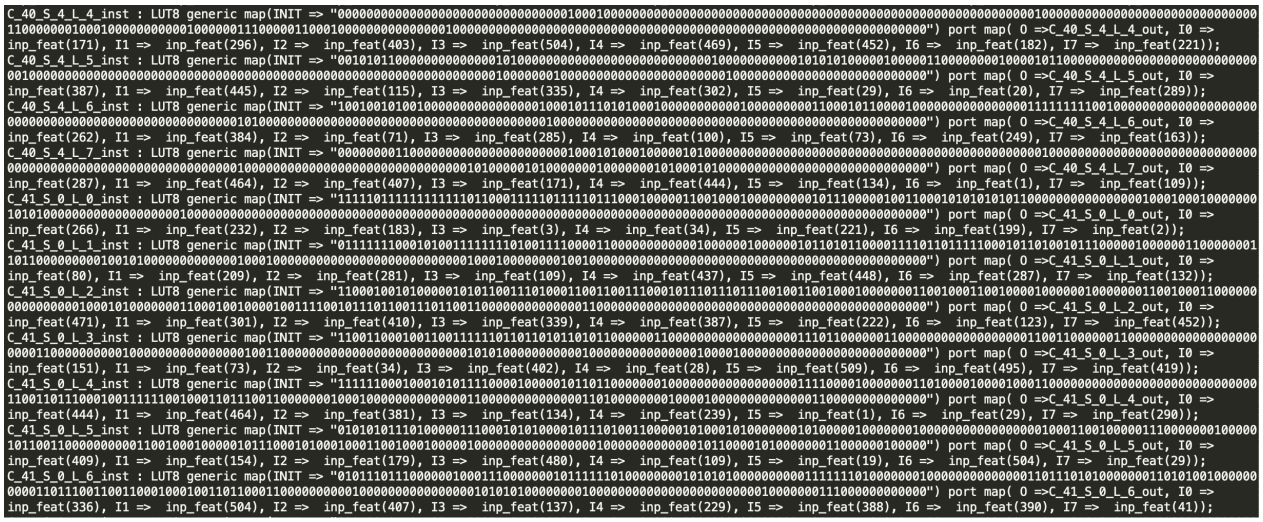
In Table 5.10 we compare our architecture to the other implementations in the literature on the basis of power, throughput and resources consumed. The implementations marked with * indicate implementation results of only the classifier portion of the network. We can observe that our architecture consumes a fraction of the power compared to the other architectures. It is because we map each operation in our architecture to the LUTs in the networks, and eliminate all MAC and memory read operations. A LUT access consumes a fraction of energy compared to MAC and memory operations. Even the binary neural networks that use XNOR operation for multiplication and Popcount operation for additions consume considerably more power and resources than our POLYBiNN architecture. The XNOR multiplication and Popcount additions in binary neural networks cannot be efficiently mapped to the LUTs, thus consuming more power and resources(Table 5.8).

Also, in Table 5.10, we compare the throughput across various architectures. In neural network implementations, they measure the MAC and memory operations per second while in our case, we measure the LUT access operations in our network. We calculate the throughput (GOPS) from Table 5.9. In our architecture, the number of operands is the number of LUTs and the time taken is the latency rounded to the nearest higher integer. Dividing both gives us the GOPS for our architecture. We achieve a much higher throughput than other implementations.

We have to keep in mind that the values for our implementation take into account only the classifier part of the architecture. A direct comparison with the other classifier only implementation of Abdelsalam et al. [43] shows that we achieve an higher accuracy with fewer resources. Other implementations report the resource and power consumed for the entire network. It is difficult to estimate the power and resource utilized of only the classifier portion these networks from the information provided in their papers. However, we believe that even with full implementation of the PoET-BiN architecture for the convolutional layers and the classifier will yield significantly less power and resources compared to other implementations.

5.2.4 Simulation results

Figure 5.2 shows a part of the VHDL code automatically generated by the parser from high level network information. We can observe the repeating structure of the code.



```
C_40_S_4_L_4_inst : LUT8 generic map(INIT => "0000000000000000000000000000000010001000000000000000000000000000000000000000000000000000000000000010000000000000000000000000000000
1100000001000100000000000010000001110000011000100000000000000001000000000000000000000000000000000000000000000000000000000000000") port map( O =>C_40_S_4_L_4_out, I0 =>
inp_feat(171), I1 =>  inp_feat(296), I2 =>  inp_feat(403), I3 =>  inp_feat(504), I4 =>  inp_feat(469), I5 =>  inp_feat(452), I6 =>  inp_feat(182), I7 =>  inp_feat(221));
C_40_S_4_L_5_inst : LUT8 generic map(INIT => "0010101100000000000000010100000000000000000000000000000100000000000101010100000100000110000000010000101100000000000000000000000000
0010000000000000000000000000000000000000000000000000000000000000010000000100000000000000000000000100000000000000000000000000000") port map( O =>C_40_S_4_L_5_out, I0 =>
inp_feat(387), I1 =>  inp_feat(445), I2 =>  inp_feat(115), I3 =>  inp_feat(335), I4 =>  inp_feat(302), I5 =>  inp_feat(29), I6 =>  inp_feat(20), I7 =>  inp_feat(289));
C_40_S_4_L_6_inst : LUT8 generic map(INIT => "1001001010010000000000000000000100010111010100010000000000001000000000110001011000010000000000000000111111110010000000000000000000
0000000000000000000000000000000101000000000000000000000000000000000000000000001000000000000000000000000000000000000000000000000") port map( O =>C_40_S_4_L_6_out, I0 =>
inp_feat(262), I1 =>  inp_feat(384), I2 =>  inp_feat(71), I3 =>  inp_feat(285), I4 =>  inp_feat(100), I5 =>  inp_feat(73), I6 =>  inp_feat(249), I7 =>  inp_feat(163));
C_40_S_4_L_7_inst : LUT8 generic map(INIT => "0000000011000000000000000000000001000101000100000101000000000000000000000000000000000000000000000010000000000000000000000000000000
0000000000000000000000000000000010000000000000000000000000000000101000001010000000100000001010001010000000000000000000000000000") port map( O =>C_40_S_4_L_7_out, I0 =>
inp_feat(287), I1 =>  inp_feat(464), I2 =>  inp_feat(407), I3 =>  inp_feat(171), I4 =>  inp_feat(444), I5 =>  inp_feat(134), I6 =>  inp_feat(1), I7 =>  inp_feat(109));
C_41_S_0_L_0_inst : LUT8 generic map(INIT => "1111101111111111111011000111110111110111000100000110010001000000000101110000010011000101010101011000000000000000010001000100000000
1010100000000000000000000010000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000") port map( O =>C_41_S_0_L_0_out, I0 =>
inp_feat(266), I1 =>  inp_feat(232), I2 =>  inp_feat(183), I3 =>  inp_feat(3), I4 =>  inp_feat(34), I5 =>  inp_feat(221), I6 =>  inp_feat(199), I7 =>  inp_feat(2));
C_41_S_0_L_1_inst : LUT8 generic map(INIT => "0111111100010100111111110100111100001100000000000001000000100000010110101100001111011011111000101101001011100000100000011000000001
1011000000000100101000000000000001000100000000000000000000000000001000100000000100100000000000000000000000000000000000000000000") port map( O =>C_41_S_0_L_1_out, I0 =>
inp_feat(80), I1 =>  inp_feat(209), I2 =>  inp_feat(281), I3 =>  inp_feat(109), I4 =>  inp_feat(437), I5 =>  inp_feat(448), I6 =>  inp_feat(287), I7 =>  inp_feat(132));
C_41_S_0_L_2_inst : LUT8 generic map(INIT => "1100010010100000101011001110100011001100111000101110111011100100110010001000000011001000110010000100000010000000110010001100000000
0000000001000101000000011000100100001001111001011101100111011001100000000000000011000000000000000000000000000000000000000000000") port map( O =>C_41_S_0_L_2_out, I0 =>
inp_feat(471), I1 =>  inp_feat(301), I2 =>  inp_feat(410), I3 =>  inp_feat(339), I4 =>  inp_feat(387), I5 =>  inp_feat(222), I6 =>  inp_feat(123), I7 =>  inp_feat(452));
C_41_S_0_L_3_inst : LUT8 generic map(INIT => "1100110001001100111111011011010110101100000011000000000000000000111011000000110000000000000000001100110000001100000000000000000000
0000110000000000100000000000000010011000000000000000000000000000101010000000000001000000000000000100010000000000000000000000000") port map( O =>C_41_S_0_L_3_out, I0 =>
inp_feat(151), I1 =>  inp_feat(73), I2 =>  inp_feat(34), I3 =>  inp_feat(402), I4 =>  inp_feat(28), I5 =>  inp_feat(509), I6 =>  inp_feat(495), I7 =>  inp_feat(419));
C_41_S_0_L_4_inst : LUT8 generic map(INIT => "1111110001000101011110000100000101101100000000100000000000000000001111000010000000110100001000010001100000000000000000000000000000
1100110111000100111111001000110111001100000000100010000000000000011000000000000000110100000000010000100000000000000001100000000000000") port map( O =>C_41_S_0_L_4_out, I0 =>
inp_feat(444), I1 =>  inp_feat(464), I2 =>  inp_feat(381), I3 =>  inp_feat(134), I4 =>  inp_feat(239), I5 =>  inp_feat(1), I6 =>  inp_feat(29), I7 =>  inp_feat(290));
C_41_S_0_L_5_inst : LUT8 generic map(INIT => "0101010111010000011100010101000010111010011000001010001010000000101000001000000010000000000000000100011001000001110000000001000000
1011001100000000001100100010000010111000101000100011001000100000100000000000000000010000000000000101100001010000000110000000100000") port map( O =>C_41_S_0_L_5_out, I0 =>
inp_feat(409), I1 =>  inp_feat(154), I2 =>  inp_feat(179), I3 =>  inp_feat(480), I4 =>  inp_feat(109), I5 =>  inp_feat(19), I6 =>  inp_feat(504), I7 =>  inp_feat(29));
C_41_S_0_L_6_inst : LUT8 generic map(INIT => "0101110111000000100011100000001011111101000000001010101000000000001111111010000000100000000000000110111010100000001101010010000000
0000110111001100110001000100110110001100000000000100000000000000001010101000000000100000000000000000000000001000000011100000000000000") port map( O =>C_41_S_0_L_6_out, I0 =>
inp_feat(336), I1 =>  inp_feat(504), I2 =>  inp_feat(407), I3 =>  inp_feat(137), I4 =>  inp_feat(229), I5 =>  inp_feat(388), I6 =>  inp_feat(390), I7 =>  inp_feat(41));
```

Figure 5.2: Parser generated VHDL code

Figures 5.3, 5.4, 5.5 show the predicted outputs of the VHDL code generated (*pred_out*) and provides a comparison with the the GPU generated output(*cor_in*). Each output contains contains 80 bit that represents the 8 bit output of each neuron in the final output layer of each network. As there are 10 neurons in the output layer, the output contains 80 bits. The *and_output* signal compares the two outputs and generates 1 when all the bits matches and 0 otherwise.

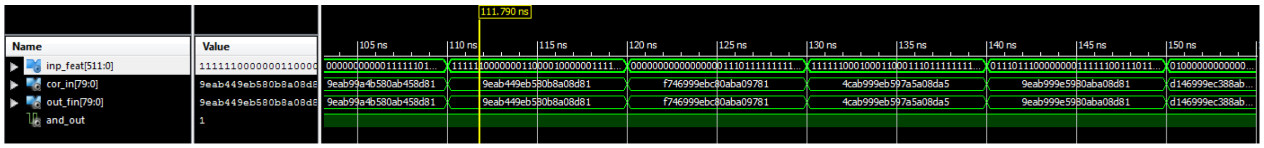


Figure 5.3: MNIST simulation of generated VHDL code

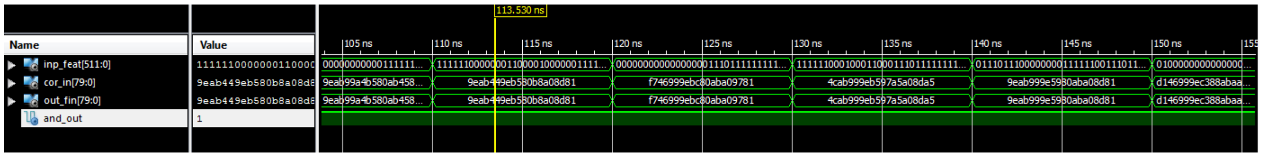


Figure 5.4: SVHN simulation of generated VHDL code

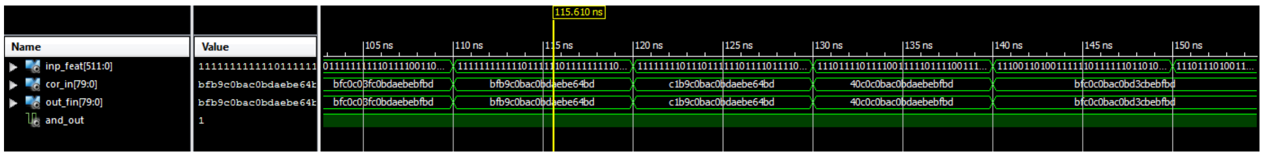


Figure 5.5: CIFAR-10 simulation of generated VHDL code

Table 5.10: Hardware implementation comparisons

Network	FPGA	Dataset	Met- -hod	Power (W)	Throughput (GOPS)	LUTs	BRAM (kB)	DSP
Nakahara et al., 2017, [57]	Zynq	CIFAR-10	1-bit	2.3	143	14509	32	1
Moss et al., 2017, [58]	Arria	CIFAR-10	1-bit	48	849	115000	-	-
Umuroglu et al., 2017 [48]	PYNQ	CIFAR-10	1-bit	2.5	166	42823	270	32
Zhao et al., 2017, [59]	Zynq	CIFAR-10	1-bit	4.7	143	46900	94	3
Ma et al., 2017, [63]	Arria	CIFAR-10	8-bit	21.2	31	16100	1900	1518
Qiu et al., 2016, [64]	Zynq	CIFAR-10	16-bit	9.63	14	-	-	-
Zhang et al., 2018, [65]	Virtex-7	CIFAR-10	16-bit	26	13	300000	1248	2833
Podili et al., 2017, [68]	Stratix V	CIFAR-10	32-bit	8.04	29	196370	256	1100
Zhang et al., 2017, [61]	Stratix V	CIFAR-10	32-bit	13.2	9	200522	4096	224
Abdelsalam et. al. 2018, [43]*	ZYNQ	CIFAR-10	DT	0.591	-	28,735	0	0
Our Work*,	Spartan-6	CIFAR-10	DT	0.341	965	9650	0	0
Umuroglu et al., [48]	ZYNQ	MNIST	1-bit	8.8	2465	82989	396	-
Jiao et al., 2017, [60]	ZYNQ	MNIST	2-bit	2.26	181	44000	105	89
Abdelsalam et al. 2018, [43]*	ZYNQ	MNIST	DT	1.106	-	109653	0	0
Our Work*,	Spartan-6	MNIST	DT	0.513	1189	11899	0	0
Guo et al., 2017, [62]	Zynq	SVHN	8-bit	4.5	24	29867	85	190
Shen et al., 2018, [66]	Virtex	SVHN	16-bit	26	30	170000	1232	1376
Zhang et al., 2017, [67]	Arria	SVHN	16-bit	37.4	48	-	1250	1320
Zhang et al., 2015, [69]	Virtex-7	SVHN	32-bit	18.6	3	186251	1024	2240
Our Work*,	Spartan-6	SVHN	DT	0.417	476	2660	0	0

CHAPTER 6 CONCEPTION OF PoET-BiN

The conception of PoET-BiN was not a straight forward process. Rather we had to experiment with various ideas and techniques. In this chapter we details some of the other ideas that we experimented with.

6.1 Attempt 1: Accelerating the Inference Phase in Ternary Convolutional Neural Networks using Configurable Processors

First I started my thesis research with an implementation of Ternary Neural Networks [37] on a configurable processor. This was an exploratory work to analyse the viability of implementing neural networks on configurable processors. We implemented a LeNet network for classification of the MNIST dataset. The work was published in the IEEE DASIP conference. We could achieve almost $5\times$ increase in throughput with just 30% increase in resources. However, the overall throughput was still less compared to other FPGA implementations due to the hardware limitations of configurable processors.

6.2 Attempt 2: Decision Trees and Adaboost

We then came across Decision Trees and the Adaboost algorithm to implement neural networks. This led us to work on overcoming the challenges faced by [43]'s POLYBiNN architecture. Some of the problems we tried to address were :

- The sum of products were unoptimized, i.e the sum of product expressions were not in the canonical form.
- They used a one-vs-all classification technique that was both expensive and led to large degradation in the accuracy when compared to an individual classifier.
- They only implemented the classifier portion of the neural network. The convolution layers were not implemented on the FPGA.

In this thesis, first we tried to address the issue of reducing the sum of product expressions. We reduced the binary expressions generated by [43] to the lowest possible canonical form. We used Binary Decision Diagrams (BDD) to reduce the binary expression. Once, we checked the equality of our reduced expression with the original expression, we implemented our binary expression on the FPGA and compared it to the original implementation. Unfortunately, both the implementations consumed similar amounts of LUTs. This was because the Xilinx synthesizer reduced the expression to the lowest possible canonical form before implementing on FPGA. Hence, the work done by us was already done by the synthesizer. Therefore, we did not achieve lower resource utilization. Nevertheless,the experience gained in writing parsers came in handy to generate scripts of the PoET-BiN architecture.

Then we tried to address the second weakness of one-vs-all classification technique. We implemented multi-class Decision Trees using the *scikit-learn* library. These multiclass decision trees performed slightly better in terms of accuracy than one-vs-all decision trees. However, after the estimated hardware costs for multi-class decision trees was much higher than the one-vs-all classifier. Moreover, we required large DTs that necessitated the use of external memory, which significantly increases the delay. These increases in resources and delay would not justify the slight increase in accuracy. Hence, this idea was dropped.

With repeated unsuccessful attempts in building DTs with off-the-shelf classifiers for large scale classification, we had to abandon the above ideas. We came to the realization that to efficiently build DTs for large scale image classification, we had to tackle the problem from a bottom to top approach. We tried a completely new approach that lead to the conception of the PoET-BiN architecture explained in the previous chapters. The conception of the PoET-BiN architecture was itself filled with many twists and turns detailed in the next section.

6.3 Attempt 3: Conception of the PoET-BiN architecture

The investigation of hardware resources used by multiclass decision trees sparked the idea of building binary decision trees to fit the smallest computing element in a FPGA. The smallest computing element in a FPGA is a LUT. These are the salient features of a LUT in FPGA:

- Can implement any function as a Look-Up operation.
- These Look-Up operations consume a fraction of energy as compared to onchip memory or external memory operations.
- The LUTs can be directly programmed using built-in functions.

The decision trees need to be adapted to these LUTs. There are many off-the-shelf libraries to build decision trees. These off-the-shelf libraries build decision trees based on the hyper parameters such as the number of nodes, the number of levels and number of samples in each leaf node. However, there is no way to build a DT that is limited by the number of distinct inputs. Using this idea, we built decision trees from scratch that are limited by the number of inputs as described in section 4.1.1. Using this algorithm, we could fit a decision tree in a single LUT. However, these decision trees are weak classifiers and insufficient.

Hence, to augment the capacity of the decision tree, we experimented with various boosting and bagging techniques. First, we tried a popular bagging technique called random forest. In the random forest algorithm, a subset of the features is chosen at random. These selected features are used by the DT algorithm for classification. For the next DT another random subset of features are chosen. Finally a voting circuit averages the results of all the DTs. This technique results in better performance than a single DT. However, since the features are chosen randomly each time ,the accuracy varies for each try. Sometimes, the variations in the case of the CIFAR-10 dataset, were quite significant. Hence, we had to drop this method. Then we tried the common boosting technique of Adaboost that is explained in section 4.1.2. The results are shown in Table 6.1. We can see that the number of DTs is quite large. With 100 DTs per class we were able to achieve 94% accuracy on the MNIST dataset without any convolution layers. Since each DT has only 6 inputs, it can fit in a single LUT. It is easy to estimate the number of LUTs required by such an architecture. As each DT can fit in one LUT, 100 DTs require 100 LUTs. Since we use one-vs-all classification, there are n_c classes and each class requires 100 LUTs. In MNIST, there are 10 classes, therefore we require 1000 LUTs in total. The total LUTs are

$$n_c \times 100 = 10 \times 100 = 1000LUTs \tag{6.1}$$

Table 6.1: Accuracy on MNIST

No. of DTs	No. of Inputs	Accuracy (%)
10	6	86.18
20	6	89.74
50	6	92.46
100	6	94.06

With this architecture, we were able to achieve reasonable accuracy on all datasets. However, it required a large number of decision trees per class. This resulted in expensive MAC operations that formed a bottleneck in the architecture. Moreover, we still used one-vs-all classification which was a major hindrance in improving the classification accuracy. Inspired by [44], we tried to limit the number of DTs in the Adaboost algorithm. Instead, we added multiple layers of DTs one after the other. The intermediate outputs generated by the DTs in each layer were used as inputs to the subsequent layers similar to the architecture shown in Fig 6.1. We used our single LUT instead of standard DTs. We observed that the DTs in the subsequent layers preferred the features generated from the previous layer over the input feature vector. This suggested that the information produced by each layer was indeed useful. However, even with four layers, the increase in accuracy was not significant. We could achieve a maximum of 95% accuracy on the MNIST dataset.

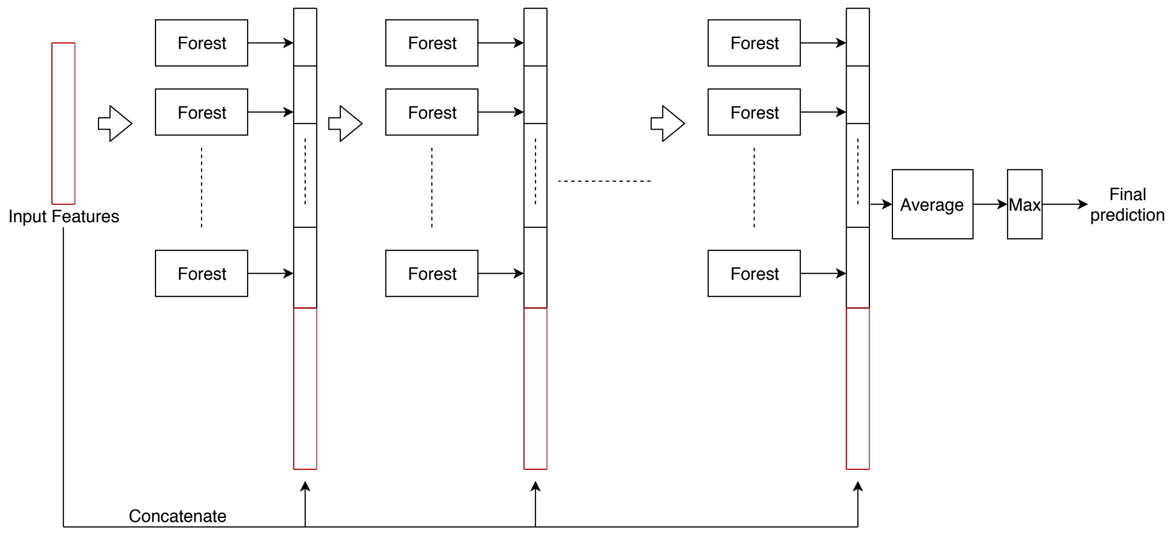


Figure 6.1: Deep Forest

Since this technique did not provide promising results, we experimented with another variant of the multi-level decision trees as illustrated in Fig. 6.2. In this method, we used the Adaboost algorithm with varying threshold. The first level had a very high threshold of 0.8. We implemented a one-vs-all classification across all classes. Each sample in the dataset were classified into three categories namely correctly classified, wrongly classified and confused samples. The correctly classified samples were those in which only one binary classifier out of the 10 ($= n_c$) binary classifiers in the one-vs-all classification produced a positive result and that predicted class matched the true output class in the dataset. The wrongly classified samples were those in which only one binary classifier out of the 10 ($= n_c$) binary classifiers in the one-vs-all classification produced a positive result and that predicted class did not match the true output class in the dataset. The confused sample were those in which more than one binary classifier produced a positive result or none of the binary classifier produced a positive result. These confused samples were provided to the next level of multiclass classification with an Adaboost of lower threshold. In this way, we built multiple levels until we ran out of training samples. This architecture is close to how our brain perceives an image. First, our brain recognizes clearer (easier to perceive) images. Then, we further analyse the image to recognize more occluded (difficult to perceive) objects.

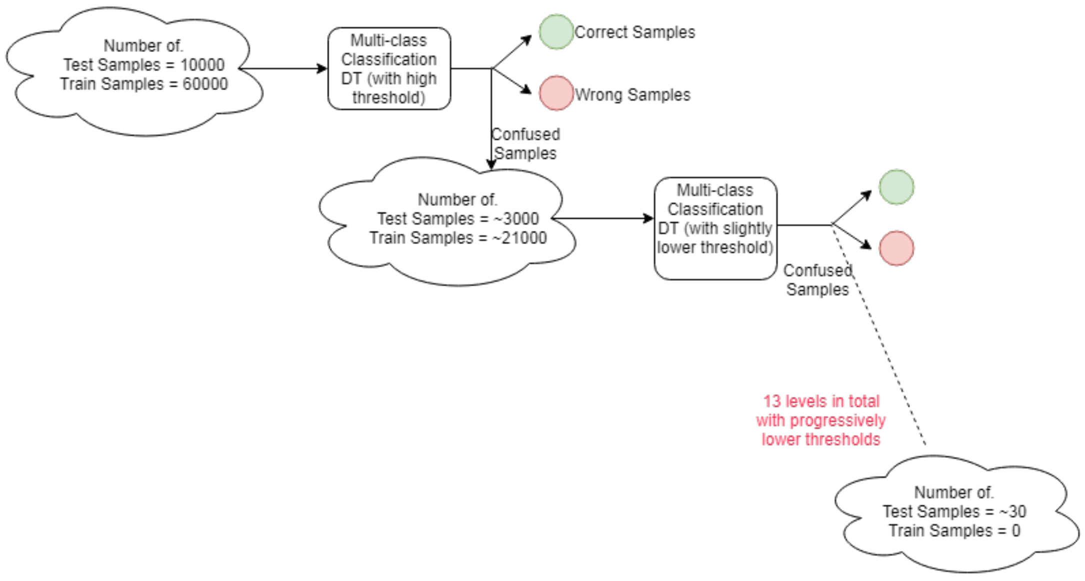


Figure 6.2: Multi-level Adaboost

Fig 6.3 provides an equivalent architecture for one level of the multi-level Adaboost method. Each DT with 6 inputs can be accommodated in a single LUT. The output of the DTs are multiplied with the corresponding weight and all the results are added. Then this result is thresholded to produce an one bit output. This structure is repeated over all the 10 classes. The output of the 10 classifiers are compared. If only one classifier outputs a 1 and other classifiers output a zero, then the predicted class is matched with the true output in the dataset. If there are multiple ones or all the outputs are zero, then the sample is sent to the next level. The intermediate results of all the DTs are used as features in the next level (represented in red in the figure).

This classification method yielded an accuracy of 95.6% on the MNIST dataset. This accuracy is less than state-of-the-art neural networks. Moreover, the MAC operations would consume more LUTs. Hence, after extensive analysis, we had to drop this idea of multi-level decision trees.

As we can see that the MAC operation has been a bottleneck throughout all the architectures, we set out to remove the MAC operation. Similar to the approach of just having P inputs to a DT to fit it in a single LUT, we planned to have MAC units with P inputs only. Hence, we first created large (P^2) DTs using the Adaboost algorithm. Then we grouped P DTs together and assigned a MAC unit to them. The output of the MAC units were thresholded to a binary value. P such MAC units were created from P^2 DTs. Each MAC unit was assigned a weight equivalent to the sum of the weights of its DTs. Now we had P binary

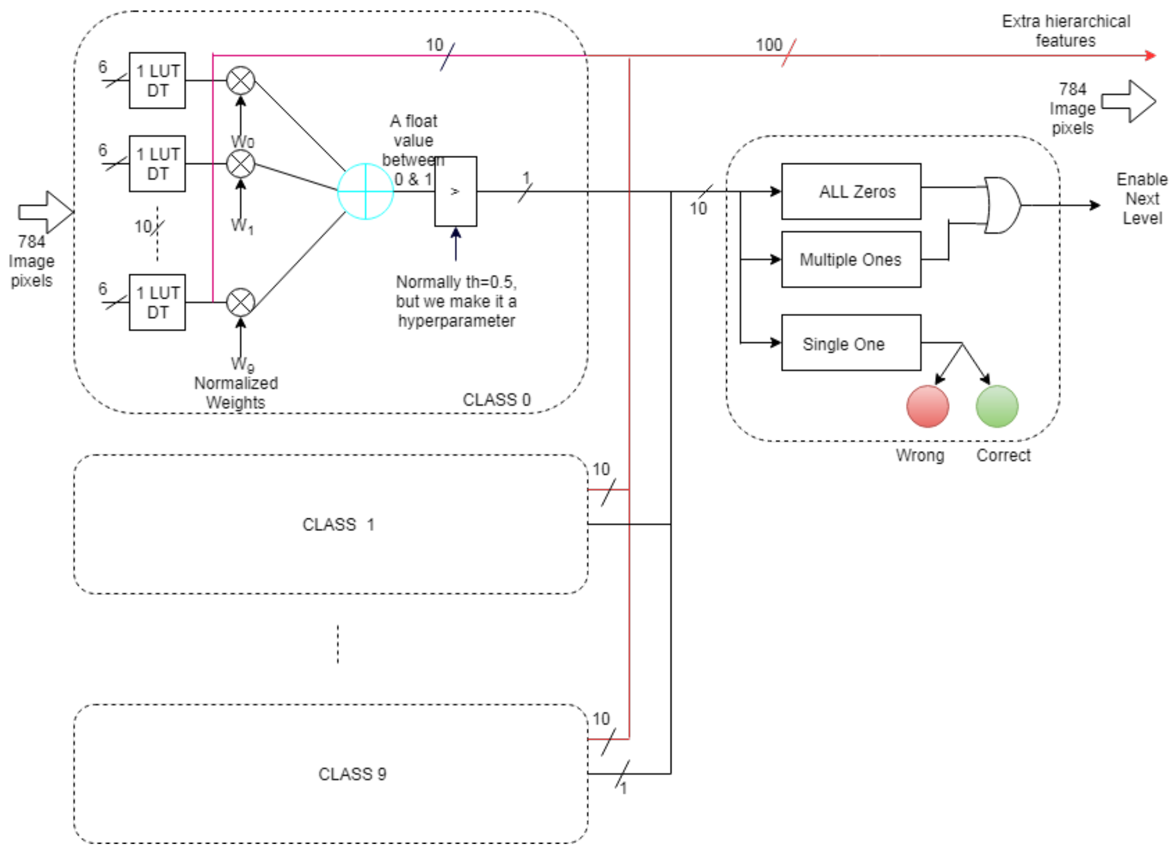


Figure 6.3: Architecture for a level of the multi-level Adaboost method

outputs of the MAC units with an associated weight for each. We used another MAC unit to multiply and add these P outputs. This value was thresholded to produce the final output. So, in a way, we approximated the MAC operation with P^2 values to a hierarchical MAC structure taking P inputs at a time. Fig. 6.4 illustrates the approximation of MAC operation between 9 inputs(I_i) and its corresponding weights (w_i). Originally the output of the MAC function should be

$$Output = threshold(\sum_{i=0}^{8} I_i \times w_i) \tag{6.2}$$

As explained earlier, we group the MAC units with $P = 3$ inputs and threshold each MAC. The weights W_0, W_1, W_2 are assigned as follows :

$$W_0 = w_0 + w_1 + w_2 \tag{6.3}$$

$$W_1 = w_3 + w_4 + w_5 \tag{6.4}$$

$$W_2 = w_6 + w_7 + w_8 \tag{6.5}$$

With this approximation, the Adaboost is easily implementable as LUTs. We call this approximation technique for the MAC operation in the Adaboost algorithm, Grouped Adaboost. However,it is detrimental to the accuracy of the Adaboost algorithm. Hence, to counter this effect, we came up with the hierarchical Adaboost algorithm to reduce the loss in accuracy as seen in Table 6.2 (on the MNIST dataset).

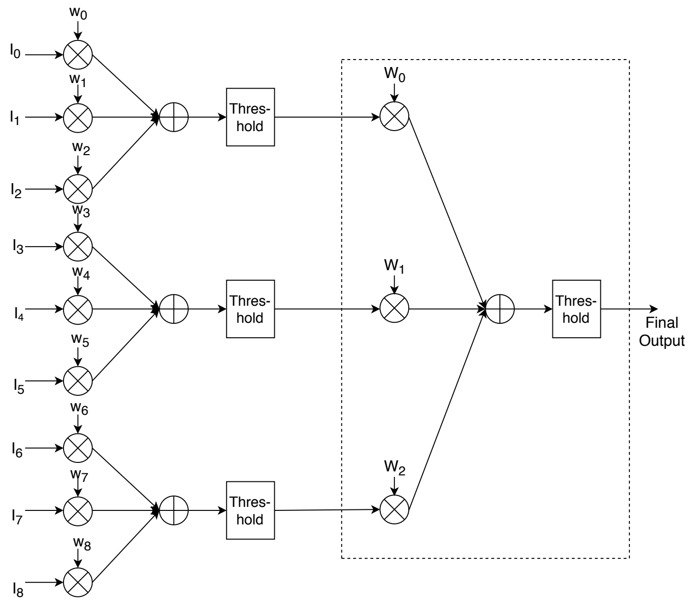


Figure 6.4: Grouped Adaboost

Table 6.2: Comparisons of various MAC operation grouping technique in Adaboost

No. of DTs	Accuracy (%)
Original Adaboost	98.26
Grouped Adaboost	97.03
Hierarchical Adaboost	98.15

With the hierarchical Adaboost algorithm we could achieve good accuracy on binary classification tasks. Still, the confidence circuit used to break ties between the various classifiers in a one-vs-all classification was a hindrance to achieve better accuracy for better classification. Inspired by the work of Kontschieder et al. in [47], we formulated a combined approach using our Hierarchical DTs and fully connected neural network to achieve near state-of-the-art accuracies on the MNIST,CIFAR-10 and SVHN datasets. The hierarchical Adaboost algorithm with feature limited DTs was used to represent the binary features in the fully connected network while a quantized sparsely connected network was used for the output layer. This LUT based architecture is named PoET-BiN.

CHAPTER 7 CONCLUSION

This work introduced PoET-BiN, an architecture and its associated building algorithm, that is optimized to fit the LUTs or memory blocks in embedded systems such as FPGAs. The PoET-BiN architecture combines the representational capacity of neural networks and low-cost implementation of binary neural networks to build a powerful classifier.

First, we trained vanilla neural networks with full precision weights and activations. Then we binarized select layers in the network to form a teacher network. These layers were then replaced with a modified DT based architecture. These DTs were optimized for a given set of unique inputs rather than depth or nodes as seen in off-the-shelf DT classifiers. We modified the node based entropy DT training algorithm to a level based entropy DT training algorithm. This helped optimize the DTs for a given number of unique inputs. However, these decision trees were weak classifiers. Hence, to boost the weak classifier, we used a modified version of the popular Adaboost algorithm. We call this modified version hierarchical Adaboost. In the hierarchical Adaboost algorithm in each level there the decision trees are limited and when there is a need for more decision trees that can be accommodated in that level then a new level is created. The number of decision trees in each level grows exponentially with an increase in the levels. The decision trees grouped together using a hierarchical Adaboost provide a strong binary classifier. It is used to replace a network of neurons in the hidden layers of the teacher networks. The final output layer cannot be binarized. Hence, we quantize the output layer till the drop in accuracy is in the acceptable range. This final layer is also implemented using LUTs. Thus we try to replace all the MAC computation in the network with our LUT based architecture to the maximum extent possible.

In this work, we replaced the classifier portion of various networks with our architecture to achieve accuracies similar to the ones obtained with full precision implementations for the MNIST, CIFAR-10 and SVHN datasets. This classifier portion of the neural network replaced with our LUT based implementation is refereed as PoET-BiN. It is call PoET-BiN as the LUTs in our architecture can be though of as a neuron in a conventional neural network. However, each LUT has limited number of inputs (often less than 10) as compared to neurons in a neural network that have upto 4096 inputs as seen in VGG networks. Moreover, the LUTs in our networks have a binary output. Hence our architecture contains "Tiny binary neurons". More importantly, as the computations in our architecture is trained to efficiently fit in an embedded system such as FPGA. Therefore, it is highly power efficient as compared to vanilla neural networks.

With the PoET-BiN architecture, we achieved similar accuracies to that of a vanilla network for the MNIST and SVHN dataset. In the case of CIFAR-10, we achieved better accuracies than the vanilla networking due to the regularization effect. More importantly, we reduced the energy consumption for the classifier portion by up to six orders of magnitude compared to a floating point implementations and by up to three orders of magnitude when compared to recent binary quantized neural networks. This is due to the fact that all the arithmetic operations are replaced by small LUTs on binary signals that are optimized for the underlying hardware in FPGAs. This work shows that for better efficiency, we need to understand the underlying architecture of the embedded system and train our network with computations that can be efficiently implemented such a system. Therefore, it is necessary to have better training strategies of neural networks for cost, power and area optimized inference engine.

7.1 Limitations

One of the major limitation of this work is that the RINC-L modules can be used to replicate a network of binary neurons only. Hence, we need to introduce binary activation in our network. This limits the representation capacity of our network leading to reduced accuracy on large dataset. Therefore, to compensate this reduction in accuracy we need to have some layers with more than 1-bit precision, leading to more power consumption in these layers. Another limitation is that we only implement the classifier part of the network. To effectively compare our work with that of other state-of-the-art implementations, we need to have a complete end-to-end implementation of our architecture including the convolution layers. Hence we only compare our work to that of the classifier portion of vanilla and quantized neural networks.

7.2 Future Research

There are various possible avenues for future research. First, we could extend the PoET-BiN architecture to the convolutional layers. This would provide an end-to-end model for vision classification tasks using CNNs. Second we could expand the network to classify larger datasets such as ImageNet. Further, this technique need not be restricted to just CNN. It can be used to implement Recurrent Neural Networks (RNNs) for language applications in a power efficient manner. It is possible to better the current results by using differentiable DTs. The differentiable DTs enable the training the of DTs and convolutional layers together to obtain better feature representation. This may result in higher accuracies as the entire network is trained together instead of training layerwise.

REFERENCES

[1] S. Yang, P. Luo, C.-C. Loy, and X. Tang, "From facial parts responses to face detection: A deep learning approach," in *International Conference on Computer Vision*, 2015, pp. 3676–3684.

[2] R. Kadlec, M. Schmid, O. Bajgar, and J. Kleindienst, "Text understanding with the attention sum reader network," in *Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2016, pp. 908–918. [Online]. Available: http://aclweb.org/anthology/P16-1086

[3] S. Pascual, A. Bonafonte, and J. Serrà, "SEGAN: Speech enhancement generative adversarial network," *arXiv preprint :1703.09452*, 2017.

[4] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review*, vol. 65, no. 6, pp. 386–391, 1958.

[5] T. Young, D. Hazarika, S. Poria, and E. Cambria, "Recent trends in deep learning based natural language processing [review article]," *IEEE Computational Intelligence Magazine*, vol. 13, no. 3, pp. 55–75, 2018.

[6] X. Qiu, L. Zhang, Y. Ren, P. N. Suganthan, and G. Amaratunga, "Ensemble deep learning for regression and time series forecasting," in *IEEE Symposium on Computational Intelligence in Ensemble Learning*, 2014, pp. 1–6.

[7] D. Kwon, H. Kim, J. Kim, S. C. Suh, I. Kim, and K. J. Kim, "A survey of deep learning-based network anomaly detection," *Cluster Computing*, vol. 22, no. 1, pp. 949–961, 2019. [Online]. Available: https://doi.org/10.1007/s10586-017-1117-8

[8] D. Luebke and G. Humphreys, "How GPUs work," *Computer*, vol. 40, no. 2, pp. 96–100, 2007.

[9] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *IEEE conference on computer vision and pattern recognition*, 2009, pp. 248–255.

[10] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE Micro*, vol. 30, no. 2, pp. 56–69, 2010. [Online]. Available: http://dx.doi.org/10.1109/MM.2010.41

[11] A. R. Brodtkorb, T. R. Hagen, and M. L. SæTra, “Graphics processing unit programming strategies and trends in GPU computing,” *J. Parallel Distrib. Comput.*, vol. 73, no. 1, pp. 4–13, 2013. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2012.04.003

[12] R. Al-Rfou *et al.*, “Theano: A Python framework for fast computation of mathematical expressions,” *arXiv preprint arXiv:1605.02688*, 2016.

[13] W. N. Chelton and M. Benaissa, “Fast elliptic curve cryptography on FPGA,” *IEEE transactions on very large scale integration systems*, vol. 16, no. 2, pp. 198–205, 2008.

[14] T. Vladimirova, X. Wu, and C. P. Bridges, “Development of a satellite sensor network for future space missions,” in *2008 IEEE Aerospace Conference*, 2008, pp. 1–10.

[15] A. F. Agarap, “Deep learning using rectified linear units,” *arXiv preprint arXiv:1803.08375*, 2018.

[16] A. C. Marreiros, J. Daunizeau, S. J. Kiebel, and K. J. Friston, “Population dynamics: variance and the sigmoid activation function,” *Neuroimage*, vol. 42, no. 1, pp. 147–157, 2008.

[17] G. Welch, G. Bishop *et al.*, “An introduction to the Kalman filter,” *UNC-Chapel Hill*, 1995.

[18] P. C. Ng and S. Henikoff, “SIFT: Predicting amino acid changes that affect protein function,” *Nucleic acids research*, vol. 31, no. 13, pp. 3812–3814, 2003.

[19] Y. LeCun, Y. Bengio *et al.*, “Convolutional networks for images, speech, and time series,” *The handbook of brain theory and neural networks*, vol. 3361, no. 10, pp. 1995–1999, 1995.

[20] J. Ngiam, Z. Chen, D. Chia, P. W. Koh, Q. V. Le, and A. Y. Ng, “Tiled convolutional neural networks,” in *Advances in neural information processing systems*, 2010, pp. 1279–1287.

[21] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[22] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in *International Conference on Learning Representations*, 2015. [Online]. Available: http://arxiv.org/abs/1409.1556

[23] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Computer Vision and Pattern Recognition*, 2016.

[24] F. Sultana, A. Sufian, and P. Dutta, “Advancements in image classification using convolutional neural network,” in *Fourth International Conference on Research in Computational Intelligence and Communication Networks*, 2018, pp. 122–129.

[25] S. A. Mingoti and J. O. Lima, “Comparing som neural network with fuzzy c-means, k-means and traditional hierarchical clustering algorithms,” *European journal of operational research*, vol. 174, no. 3, pp. 1742–1759, 2006.

[26] A. Buetti-Dinh *et al.*, “Deep neural networks outperform human expert’s capacity in characterizing bioleaching bacterial biofilm composition,” *Biotechnology Reports*, vol. 22, pp. 321–335, 2019.

[27] A. Blanco, R. Pino-Mejías, J. Lara, and S. Rayo, “Credit scoring models for the microfinance industry using neural networks: Evidence from peru,” *Expert Systems with applications*, vol. 40, no. 1, pp. 356–364, 2013.

[28] D. Rosen, V. Miagkikh, and D. Suthers, “Social and semantic network analysis of chat logs,” in *1st International Conference on Learning Analytics and Knowledge*, 2011, pp. 134–139.

[29] F. Amato, A. López-Rodríguez, E. Peña-Méndez, P. Vaňhara, A. Hampl, and J. Havel, “Artificial neural networks in medical diagnosis,” *Journal of Applied Biomechanics*, vol. 11, pp. 47–58, 2013.

[30] M. Courbariaux, Y. Bengio, and J.-P. David, “Binaryconnect: Training deep neural networks with binary weights during propagations,” in *Advances in neural information processing systems*, 2015, pp. 3123–3131.

[31] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, “Binarized neural networks: Training deep neural networks with weights and activations constrained to + 1 or -1,” *arXiv preprint: 1602.02830*, 2016.

[32] M. E. Becker, “Dynamic popcount/shift circuit,” 2004, uS Patent 6,754,685.

[33] L. Deng, “The MNIST database of handwritten digit images for machine learning research [best of the web],” *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[34] A. Krizhevsky, V. Nair, and G. Hinton, "The CIFAR-10 dataset," *online: http://www. cs. toronto. edu/kriz/cifar. html*, vol. 55, 2014.

[35] F. Girosi, M. Jones, and T. Poggio, "Regularization theory and neural networks architectures," *Neural computation*, vol. 7, no. 2, pp. 219–269, 1995.

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

[37] L. Deng, P. Jiao, J. Pei, Z. Wu, and G. Li, "GXNOR-Net: Training deep neural networks with ternary weights and activations without full-precision memory under a unified discretization framework," *Neural Networks*, pp. 49–58, 2018.

[38] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs," in *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[39] Y. Choi, M. El-Khamy, and J. Lee, "Towards the limit of network quantization," *arXiv preprint arXiv:1612.01543*, 2016.

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

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

[42] L. Wan, M. Zeiler, S. Zhang, Y. Le Cun, and R. Fergus, "Regularization of neural networks using Dropconnect," in *International Conference on Machine Learning*, 2013, pp. 1058–1066.

[43] A. M. Abdelsalam, A. Elsheikh, J.-P. David, and J. M. P. Langlois, "POLYBiNN: A scalable and efficient combinatorial inference engine for neural networks on FPGA," in *Design and Architectures for Signal and Image Processing*, 2018, pp. 19–24.

[44] Z.-H. Zhou and J. Feng, "Deep forest: Towards an alternative to deep neural networks," *arXiv preprint:1702.08835*, 2017.

[45] A. Liaw, M. Wiener *et al.*, "Classification and regression by Random forest," *R Language publications*, vol. 2, no. 3, pp. 18–22, 2002.

[46] G. Holmes, B. Pfahringer, R. Kirkby, E. Frank, and M. Hall, "Multiclass alternating decision trees," in *European Conference on Machine Learning.* Springer, 2002, pp. 161–172.

[47] P. Kontschieder, M. Fiterau, A. Criminisi, and S. Rota Bulo, "Deep neural decision forests," in *International Conference on Computer Vision*, 2015, pp. 1467–1475.

[48] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A framework for fast, scalable binarized neural network inference," in *International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 65–74.

[49] T. Feist, "Vivado design suite," *White Paper*, vol. 5, p. 30, 2012.

[50] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *International Solid-State Circuits Conference Digest of Technical Papers*, 2014, pp. 10–14.

[51] S. Chidambaram, A. Riviello, J. M. Pierre Langlois, and J. David, "Accelerating the inference phase in ternary convolutional neural networks using configurable processors," in *Conference on Design and Architectures for Signal and Image Processing*, 2018, pp. 94–99.

[52] Jianguo Xin and M. J. Embrechts, "Supervised learning with spiking neural networks," in *International Joint Conference on Neural Networks. Proceedings*, vol. 3, 2001, pp. 1772–1777 vol.3.

[53] F. Akopyan *et al.*, "Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1537–1557, 2015.

[54] J. Cong, J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-specific instruction generation for configurable processor architectures," in *12th international symposium on Field programmable gate arrays*, 2004, pp. 183–189.

[55] A. Maghazeh, U. D. Bordoloi, P. Eles, and Z. Peng, "General purpose computing on low-power embedded GPUs: Has it come of age?" in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2013, pp. 1–10.

[56] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "A survey of FPGA-based neural network inference accelerators," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 12, no. 1, p. 2, 2019.

[57] H. Nakahara, T. Fujii, and S. Sato, “A fully connected layer elimination for a binarizec convolutional neural network on an FPGA,” in *27th International Conference on Field Programmable Logic and Applications*, 2017, pp. 1–4.

[58] D. J. Moss, E. Nurvitadhi, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, and P. H. Leong, “High performance binary neural networks on the Xeon+ FPGA platform,” in *27th International Conference on Field Programmable Logic and Applications*, 2017, pp. 1–4.

[59] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, “Accelerating binarized convolutional neural networks with software-programmable FPGAs,” in *International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 15–24. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021741

[60] L. Jiao, C. Luo, W. Cao, X. Zhou, and L. Wang, “Accelerating low bit-width convolutional neural networks with embedded FPGA,” in *27th International Conference on Field Programmable Logic and Applications*, 2017, pp. 1–4.

[61] C. Zhang and V. Prasanna, “Frequency domain acceleration of convolutional neural networks on CPU-FPGA shared memory system,” in *International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 35–44.

[62] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, “Angel-eye: A complete design flow for mapping cnn onto embedded FPGA,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 35–47, 2017.

[63] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, “Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks,” in *International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 45–54.

[64] J. Qiu *et al.*, “Going deeper with embedded FPGA platform for convolutional neural network,” in *International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 26–35.

[65] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, “Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[66] J. Shen, Y. Huang, Z. Wang, Y. Qiao, M. Wen, and C. Zhang, “Towards a uniform template-based architecture for accelerating 2D and 3D CNNs on FPGA,” in *International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 97–106.

[67] J. Zhang and J. Li, “Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network,” in *International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 25–34.

[68] A. Podili, C. Zhang, and V. Prasanna, “Fast and efficient implementation of convolutional neural networks on FPGA,” in *IEEE 28th International Conference on Application-specific Systems, Architectures and Processors*, 2017, pp. 11–18.

[69] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, “Optimizing FPGA-based accelerator design for deep convolutional neural networks,” in *International Symposium on Field-Programmable Gate Arrays*, 2015, pp. 161–170.

[70] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv preprint arXiv:1409.1556*, 2014.

[71] C. J. Burges, “A tutorial on support vector machines for pattern recognition,” *Data Mining and Knowledge Discovery*, vol. 2, no. 2, pp. 121–167, 1998.

[72] P. Langley, W. Iba, and, and K. Thompson, “An analysis of bayesian classifiers,” in *Tenth National Conference on Artificial Intelligence*, 1992, pp. 223–228.

[73] J. R. Quinlan, “Induction of decision trees,” *Machine Learning*, vol. 1, no. 1, pp. 81–106, 1986.

[74] G. Rätsch, T. Onoda, and K.-R. Müller, “Soft margins for Adaboost,” *Machine learning*, vol. 42, no. 3, pp. 287–320, 2001.

[75] R. Rifkin and A. Klautau, “In defense of one-vs-all classification,” *Journal of machine learning research*, vol. 5, no. Jan, pp. 101–141, 2004.

[76] H. K. Kwan, “Simple sigmoid-like activation function suitable for digital hardware implementation,” *Electronic Letters*, vol. 28, no. 15, pp. 1379–1380, 1992.

[77] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” in *International Conference on Machine Learning - Volume 37*, ser. ICML’15, 2015, pp. 448–456. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045118.3045167

[78] L. Rosasco, E. D. Vito, A. Caponnetto, M. Piana, and A. Verri, “Are loss functions all the same?” *Neural Computation*, vol. 16, no. 5, pp. 1063–1076, 2004.

[79] D. P. Kingma and J. Ba, “ADAM: A method for stochastic optimization,” in *International Conference on Learning Representations*, 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[80] L. Kuang, “Train CIFAR10 with PyTorch,” *https://github.com/kuangliu/pytorch-cifar*, 2018.

[81] X. Jing, “An implementation of the deep neural decision forests in pytorch,” *https://github.com/jingxil/Neural-Decision-Forests*, 2018.

[82] F. Li, B. Zhang, and B. Liu, “Ternary weight networks,” *arXiv preprint arXiv:1605.04711*, 2016.

[83] H. Alemdar, V. Leroy, A. Prost-Boucle, and F. Pétrot, “Ternary neural networks for resource-efficient ai applications,” in *International Joint Conference on Neural Networks*, May 2017, pp. 2547–2554.

[84] D. H. K. Hoe, C. Martinez, and S. J. Vundavalli, “Design and characterization of parallel prefix adders using FPGAs,” in *43rd Southeastern Symposium on System Theory*, 2011, pp. 168–172.