**POLYTECHNIQUE MONTRÉAL**
affiliée à l'Université de Montréal

# Efficient FPGA-based Inference Architectures for Deep Learning Networks

**AHMED ABDELSALAM**
Département de génie informatique et génie logiciel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Génie informatique

Novembre 2019

**POLYTECHNIQUE MONTRÉAL**
affiliée à l'Université de Montréal

Cette thèse intitulée :

**Efficient FPGA-based Inference Architectures for Deep Learning Networks**

présentée par **Ahmed ABDELSALAM**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

## DEDICATION

*Dedicated to my parents, with love. I hope to meet again in the gardens of heaven.*

## ACKNOWLEDGEMENTS

All praise and thanks are due to Allah, Lord of the worlds. The one who blessed me with the opportunity to start my PhD, and surrounded me with all the support I needed to carry it through. I am grateful for many people who helped me through this long journey since the day I decided to join Polytechnique Montreal, 4 years ago.

I was blessed to have Prof. Pierre Langlois as my PhD Supervisor. Pierre was a continuous source of motivation and support throughout my PhD. The amount of energy and excitement he had about our work was always an example to follow. He always pushed me to get the best out of myself, we together started to explore the field of deep learning, and in few years, expanded our knowledge to be orders of magnitude bigger. Pierre as a person, he was always there whenever I needed him. His understanding, care and support made my life much easier during my PhD. I am proud to be one of Pierre's students, and I hope to bring the lessons I learned from him into practice.

I have been fortunate to work under Prof. Jean-Pierre David's supervision as well. Jean-Pierre has been deeply involved in the design and implementation of all the systems that we developed for my PhD research. Not only that, but thanks to Jean-Pierre I was able to learn many things in a short time. Working with Jean-Pierre helped me strengthen my understanding about FPGAs, neural networks and many other things inside work and outside.

I would like to express my sincere gratitude to my colleague and friend Ahmed Elshiekh. Ahmed is a walking encyclopedia, this work would not have been possible to come to the present shape without him. It was always invaluable and enjoyable at the same time to discuss ideas with Ahmed. No only this, but his suggestions and recommendations were always notable.

I am grateful for every conversation that I had with my colleagues: Rana, Hamza, Argy, Imad, Jeferson, Thibaut and Siva, the constructive feedback that I got on each project I worked on, and the friendly environment that helped me fit in very quickly and be a productive member of the group.

A big thank you to the Egyptian community at Polytechnique: Walid Dyab, Ahmed Sakr, Loay, Mohab and Adham. On the personal level, they never stopped supporting me all the way in all aspects during my PhD.

I am thankful for all my family and friends who have been a continuous support for me throughout my PhD. My mother, who passed away six months ago and literally was dreaming day and night about witnessing me defending my PhD. She was always worried about me and did everything she could to support me, with love. My father, who taught me patience, responsibility, hard work and dedication. My brother Mo and sisters Ayat and Alaa, who never stopped believing in me. My friends Ibrahim, Karim, Amr, Lotfy and Mo who are always there for me whenever I need them, advising me and cheering me up.

I have survived things that I have never thought I could get through. I am proud of myself and hope I made you all proud as well.

# RÉSUMÉ

L'apprentissage profond est devenu la technique de pointe pour de nombreuses applications de classification et de régression. Les modèles d'apprentissage profond, tels que les réseaux de neurones profonds (Deep Neural Network - DNN) et les réseaux de neurones convolutionnels (Convolutional Neural Network - CNN), déploient des dizaines de couches cachées avec des centaines de neurones pour obtenir une représentation significative des données d'entrée. La puissance des DNN et des CNN provient du fait qu'ils sont formés par apprentissage de caractéristiques extraites plutôt que par des algorithmes spécifiques à une tâche. Cependant, cela se fait aux dépens d'un coût de calcul élevé pour les processus d'apprentissage et d'inférence. Cela nécessite des accélérateurs avec de hautes performances et économes en énergie, en particulier pour les inférences lorsque le traitement en temps réel est important. Les FPGA offrent une plateforme attrayante pour accélérer l'inférence des DNN et des CNN en raison de leurs performances, dû à leur configurabilité et de leur efficacité énergétique.

Dans cette thèse, nous abordons trois problèmes principaux. Premièrement, nous examinons le problème de la mise en œuvre précise et efficace des DNN traditionnels entièrement connectés sur les FPGA. Bien que les réseaux de neurones binaires (Binary Neural Network - BNN) utilisent une représentation de données compacte sur un bit par rapport aux données à virgule fixe et à virgule flottante pour les DNN et les CNN traditionnels, ils peuvent encore nécessiter trop de ressources de calcul et de mémoire. Par conséquent, nous étudions le problème de l'implémentation des BNN sur FPGA en tant que deuxième problème. Enfin, nous nous concentrons sur l'introduction des FPGA en tant qu'accélérateurs matériels pour un plus grand nombre de développeurs de logiciels, en particulier ceux qui ne maîtrisent pas les connaissances en programmation sur FPGA.

Pour résoudre le premier problème, et dans la mesure où l'implémentation efficace de fonctions d'activation non linéaires est essentielle à la mise en œuvre de modèles d'apprentissage profond sur les FPGA, nous introduisons une implémentation de fonction d'activation non linéaire basée sur le filtre à interpolation de la transformée cosinus discrète (Discrete Cosine Transform Interpolation Filter - DCTIF). L'architecture d'interpolation proposée combine des opérations arithmétiques sur des échantillons stockés de la fonction de tangente hyperbolique et sur les données d'entrée. Cette solution offre une précision $3\times$ supérieure à celle des travaux précédents, tout en utilisant une quantité similaire des ressources de calculs et une petite quantité de mémoire. Différentes combinaisons de paramètres du filtre DCTIF peuvent être choisies pour compenser la précision et la complexité globale du circuit de la fonction tangente hyperbolique.

Pour tenter de résoudre le premier et le troisième problème, nous introduisons une architecture intermédiaire sans multiplication de réseau à une seul couche cachée (Single Hidden Layer Neural Network - SNN) avec une performance de niveau DNN entièrement connectée. Cette architecture intermédiaire d'inférence pour les FPGA peut être utilisée pour des applications qui sont résolues avec des DNN entièrement connectés. Cette architecture évite le temps nécessaire pour synthétiser, placer, router et régénérer un nouveau flux binaire de programmation FPGA lorsque l'application change. Les entrées et les activations de cette architecture sont quantifiées en valeurs de puissance de deux, ce qui permet d'utiliser des unités de décalage au lieu de multiplicateurs. Par définition, cette architecture est un SNN, nous remplissons la puce FPGA avec le maximum de neurone pouvant s'exécuter en parallèle dans la couche cachée. Nous évaluons l'architecture proposée sur des données de référence types, et démontrons un débit plus élevé en comparaison avec les travaux précédents tout en obtenant la même précision. En outre, cette architecture SNN met la puissance et la polyvalence des FPGA à la portée d'une communauté d'utilisateurs DNN plus large et améliore l'efficacité de leur conception.

Pour résoudre le second problème, nous proposons POLYBiNN, un engin d'inférence binaire qui sert d'alternative aux DNN binaires sur les FPGA. POLYBiNN est composé d'une pile d'arbres de décision (Decision Tree - DT), ces DT sont des classificateurs binaires, et utilise des portes AND-OR au lieu de multiplicateurs et d'accumulateurs. POLYBiNN est un engin d'inférence sans mémoire qui réduit considérablement les ressources matérielles utilisées. Pour implémenter des CNN binaires, nous proposons également POLYCiNN, une architecture composée d'une pile de forêts de décision (Decision Forest - DF), où chaque DF contient une pile de DT (POLYBiNN). Chaque DF classe l'une des sous-images entrelacée de l'image d'origine. Ensuite, toutes les classifications des DF sont fusionnées pour classer l'image d'entrée. Dans POLYCiNN, chaque DT est implémenté en utilisant une seule table de vérité à six entrées. Par conséquent, POLYCiNN peut être efficacement mappé sur du matériel programmable et densément parallèles. Aussi, nous proposons un outil pour la génération automatique d'une description matérielle de bas niveau pour POLYBiNN et POLYCiNN. Nous validons la performance de POLYBiNN et POLYCiNN sur des données de référence types de classification d'images de MNIST, CIFAR-10 et SVHN. POLYBiNN et POLYCiNN atteignent un débit élevé tout en consommant peu d'énergie et ne nécessitent aucun accès mémoire.

## ABSTRACT

Deep learning has evolved to become the state-of-the-art technique for numerous classification and regression applications. Deep learning models, such as Deep Neural Networks (DNNs) and Convolutional Neural Networks (CNNs), deploy dozens of hidden layers with hundreds of neurons to learn a meaningful representation of the input data. The power of DNNs and CNNs comes from the fact that they are trained through feature learning rather than task-specific algorithms. However, this comes at the expense of high computational cost for both training and inference processes. This necessitates high-performance and energy-efficient accelerators, especially for inference where real-time processing matters. FPGAs offer an appealing platform for accelerating the inference of DNNs and CNNs due to their performance, configurability and energy-efficiency.

In this thesis, we address three main problems. Firstly, we consider the problem of realizing a precise but efficient implementation of traditional fully connected DNNs in FPGAs. Although Binary Neural Networks (BNNs) use compact data representation (1-bit) compared to fixed-point data and floating-point representation in traditional DNNs and CNNs, they may still need too many computational and memory resources. Therefore, we study the problem of implementing BNNs in FPGAs as the second problem. Finally, we focus on introducing FPGAs as accelerators to a wider range of software developers, especially those who do not posses FPGA programming knowledge.

To address the first problem, and since efficient implementation of non-linear activation functions is essential to the implementation of deep learning models on FPGAs, we introduce a non-linear activation function implementation based on the Discrete Cosine Transform Interpolation Filter (DCTIF). The proposed interpolation architecture combines arithmetic operations on the stored samples of the hyperbolic tangent function and on input data. It achieves almost 3$\times$ better precision than previous works while using a similar amount of computational resources and a small amount of memory. Various combinations of DCTIF parameters can be chosen to trade off the accuracy and the overall circuit complexity of the tanh function.

In an attempt to address the first and third problems, we introduce a Single hidden layer Neural Network (SNN) multiplication-free overlay architecture with fully connected DNN-level performance. This FPGA inference overlay can be used for applications that are normally solved with fully connected DNNs. The overlay avoids the time needed to synthesize, place, route and regenerate a new bitstream when the application changes. The SNN overlay inputs and activations are quantized to power-of-two values, which allows utilizing shift units instead of multipliers. Since the overlay is a SNN, we fill the FPGA chip with the maximum possible number of neurons that can work in parallel in the hidden layer. We evaluate the proposed architecture on typical benchmark datasets and demonstrate higher throughput with respect to the state-of-the-art while achieving the same accuracy. In addition, the SNN overlay makes the power and versatility of FPGAs available to a wider DNN user community and to improve DNN design efficiency.

To solve the second problem, we propose POLYBiNN, a binary inference engine that serves as an alternative to binary DNNs in FPGAs. POLYBiNN is composed of a stack of Decision Trees (DTs), which are binary classifiers in nature, and it utilizes AND-OR gates instead of multipliers and accumulators. POLYBiNN is a memory-free inference engine that drastically cuts hardware costs. To implement binary CNNs, we also propose POLYCiNN, an architecture composed of a stack of Decision Forests (DFs), where each DF contains a stack of DTs (POLYBiNNs). Each DF classifies one of the overlapped sub-images of the original image. Then, all DF classifications are fused together to classify the input image. In POLYCiNN, each DT is implemented in a single 6-input Look-Up Table. Therefore, POLYCiNN can be efficiently mapped to simple and densely parallel hardware programmable fabrics. We also propose a tool for the automatic generation of a low-level hardware description of the trained POLYBiNN and POLYCiNN for a given application. We validate the performance of POLYBiNN and POLYCiNN on the benchmark image classification tasks of the MNIST, CIFAR-10 and SVHN datasets. POLYBiNN and POLYCiNN achieve high throughput while consuming low power, and they do not require any memory access.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# LIST OF SYMBOLS AND ACRONYMS

| | |
|---|---|
| AN | Artificial Neuron |
| ANN | Artificial Neural Network |
| ASIC | Application-Specific Integrated Circuit |
| AdaBoost | Adaptive Boosting |
| BNN | Binary Neural Networks |
| BWN | Binary Weights Neural Networks |
| CNN | Convolutional Neural Networks |
| CPU | Central Processing Unit |
| DCTIF | Discrete Cosine Transform Interpolation Filter |
| DF | Decision Forests |
| DI | Downsampled Image |
| DMA | Direct Memory Access |
| DNN | Deep Neural Network |
| DSP | Digital Signal Processing |
| DT | Decision Tree |
| EIE | Efficient Inference Engine |
| FANN | Fast Artificial Neural Network |
| FC | Fully Connected |
| FFT | Fast Fourier Transform |
| FINN | Fast, Scalable Binarized Neural Network Inference |
| FLOPS | Floating Point Operations Per Second |
| FPGA | Field Programmable Gate Array |
| FPS | Frames Per Second |
| GOPS | Giga Operations Per Second |
| GPU | Graphical Processing Unit |
| HDL | Hardware Description Language |
| HLS | High-Level Synthesis |
| LBP | Local Binary Pattern |
| LSTM | Long Short-Term Memory |
| LUT | Look-Up Table |
| MAC | Multiply and Accumulate |
| MLP | Multi-Layer Perceptron |
| NN | Neural Network |

| | |
|---|---|
| NoC | Network-on-Chip |
| PL | Programmable Logic |
| PR | Processing Region |
| PS | Processing System |
| PWL | Piecewise Non-Linear |
| RAM | Random Access Memory |
| RNN | Recurrent Neural Network |
| RSSI | Received Signal Strength Indicator |
| RTL | Register Transfer Logic |
| ReLU | Rectified Linear Unit |
| SNN | Single hidden layer Neural Network |
| SOP | Sum-of-Product |
| SVM | Support Vector Machine |
| TOPS | Tera Operations Per Second |
| TPU | Tensor Processing Unit |

# CHAPTER 1 INTRODUCTION

Machine learning is a field of artificial intelligence where computer algorithms are used to autonomously learn from data. Machine learning algorithms have been ubiquitous in several applications such as objects classification [1], pattern recognition [2] and regression problems [3]. Machine learning started early in 1950 when Alan Turing developed the Turing test to determine if a computer has real intelligence. The test is about fooling a human into believing that he/she had a natural text language conversation with another human while it is a machine. This machine is mainly designed to generate human-like responses. In 1952, Arthur Samuel wrote a computer learning program which was the game of checkers. A big step in the field of machine learning occurred when Frank Rosenblatt designed the first neural network for computers which simulates the thought processes of the human brain in 1957. Hence, the nearest neighbor algorithm was developed in 1967 that allows computers to recognize very basic patterns. In 1985, Terry Sejnowski developed NetTalk, which is a neural network that learns to pronounce words the same way a baby does. The power of machine learning started to appear when IBM's Deep Blue computer beat the world champion at chess in 1997. Lately, the term deep learning started to appear. Deep learning offers new algorithms that can be used to let computers see and distinguish objects and text in images and videos using deep models with a large number of parameters. All in all, the latest technological advancements in machine learning approaches paved the way for new exciting and complex applications [4].

## 1.1 Overview

Artificial Neural Network (ANN) is one of the machine learning algorithms that achieves high performance in a diversity of applications. Usually the end user iteratively modifies the ANN's architecture to better represent the provided examples and generalize to new data. ANNs consist of an input layer, a few hidden layers and an output layer. Each layer has a number of Artificial Neurons (ANs) that receive signals from inputs or other neurons and compute the weighted-sum of their inputs. Thereafter, an activation function is applied on the weighted-sum of each AN in order to introduce non-linearity into the network. Ten years ago, although there was no real restriction on the number of hidden layers in a Neural Network (NN), having more than two or three hidden layers was impractical in terms of computational complexity and memory access [5]. The main reason behind this is that the existing Central Processing Units (CPUs) at that time were spending weeks or months to train a deep model.

Recently, deep learning has introduced deep models such as Deep Neural Networks (DNNs) and Convolutional Neural Networks (CNNs) where large numbers of hidden layers are used [4]. The deep models with many hidden layers usually achieve better performances than NNs especially at complex applications [6]. There have been attempts to address why DNNs usually outperform NNs in terms of accuracy [6]. One reason is that DNNs usually expand traditional NNs to a large scalable network with larger capacity of neurons and hidden layers [6]. In other words, deep models have the ability to learn more complex functions than simple NNs [7]. In the case of CNNs, they are trained to extract representative features from their inputs through several non-linear convolutional layers. Consequently, DNNs and CNNs achieve better performance than NNs in terms of classification accuracy.

Hardware accelerators played a major role in the development process of machine learning algorithms. Traditionally, CPUs with limited cores are insufficient in executing deep learning models. Nowadays, hardware accelerators have massive computational resources and are capable of processing data faster. The success of deep machine learning models is massively linked to the development of parallel Graphical Processing Units (GPUs). GPUs brought a paradigm shift in reducing the computational time of the execution process of the training and inference processes of deep models. Although GPUs give developers the opportunity to use deeper models and train these models with more data, the complexity of some deep models exceeds the computational abilities of existing GPUs [8]. Moreover, GPUs consume high power which is a problematic for battery-powered devices. In addition, the high power consumption of GPUs limits the number of processors that can be used to improve GPUs' performance in terms of throughput. Therefore, more efficient specialized hardware accelerators for DNNs are highly desired [8].

Application Specific Integrated Circuits (ASICs) are dedicated hardware for a specific application where both area and performance are well optimized. Although ASICs achieve good performance in terms of computational time, they are dedicated for specific applications and have low level of configurability, and their time to market process is long. Field Programmable Gate Arrays (FPGAs) are hardware logic devices that have different amount of computational and memory resources that can be reconfigured to meet the requirements of several applications. FPGAs can be used to prototype ASICs with the intention of being replaced in the final production by the corresponding ASIC designs. However, FPGAs can achieve throughputs close to ASICs. In addition, the time-to-market of FPGA implementations is significantly shorter than for ASIC implementations. This is mainly because that, unlike ASICs, FPGAs are not fully custom chips. So, no layouts, masks and manufacturing steps are required for an FPGA implementation. It is all about compiling the Hardware Description Language (HDL) code of the implementation on the target FPGA device. The manual intervention of the complex processes like placement, routing and time analysis in FPGA implementations is less than in ASIC. That is why the time-to-market of FPGA implementations is shorter than ASIC implementations, and FPGAs are used in accelerating several complex applications.

## 1.2 Problem Statement

Deep learning hardware accelerators that score high on the 3Ps - Performance, Programmability and Power consumption - are highly desired. FPGAs achieve good performance in the 3Ps, but they have not been widely used for accelerating deep learning models compared to GPUs. The limited computation and memory resources of FPGAs might be the reason why they have not been used in accelerating deep learning applications. Moreover, the ease of the process of using GPUs over FPGAs in accelerating deep learning applications is another reason. One motivation of the present work is to thus improve the mapping and scheduling of deep learning models such as DNNs and CNNs on current FPGAs within the available resource and external memory bandwidth constraints. Another motivation is the dearth of software libraries, frameworks and template-based compilers that can help developers transform their high-level description of a deep learning architecture to a highly optimized FPGA-based accelerator with limited hardware design expertise. In this work, these motivations have led us to address three problems.

Firstly, we consider the problem of realizing a precise implementation of Fully Connected (FC) DNNs in FPGAs. This implementation entails a large number of additions and multiplications that would badly increase the overall resources required for implementing a single AN with its non-linear activation function, and a fully parallel DNN. In addition, the process of describing a DNN for FPGA implementation often involves HDL modeling, for which designer productivity tends to be lower. These concerns must be addressed before FPGAs can become as popular as GPUs for DNN implementation.

Secondly, we consider the problem of implementing FC Binary Neural Networks (BNNs) in FPGAs. Recent works on compressing deep models have opened the door to implement deep and complex models in FPGAs with limited computational and memory resources. Moreover, reducing the precision of the data representation of DNN parameters from double floating-point to fixed-point and binary representation proved efficient in terms of performance and compression. Although BNNs drastically reduce hardware resources consumption, complex

BNN models may need more computational and memory resources than those available in many current FPGAs. Therefore, optimization techniques that efficiently map BNNs to hardware are highly desired.

A third problem is how to realize efficient CNNs in FPGAs for solving classification problems. CNNs achieve state-of-the-art accuracy in many applications, however they have weaknesses that limit their use in embedded applications. A main downside of CNNs is their computational complexity. They typically demand many Multiply and Accumulate (MAC) and memory access operations in both training and inference. Another drawback of CNNs is that they require careful selection of multiple hyper-parameters such as the number of convolutional layers, the number of kernels, the kernel size and the learning rate. Consequently, other classifiers that suit the nature of FPGAs and achieve acceptable classification accuracy are worthy of exploration.

## 1.3 Research Contributions

The main objective of this work is to design and implement an efficient FPGA-based accelerators for DNNs and CNNs with the aim to achieve comparable performance as CPU and GPU-based accelerators and high throughput with high-level of design simplicity and flexibility. This section reviews the different contributions of this thesis. These contributions represent solutions to the problems detailed in section 1.2.

First, we propose a high precision approximation of the non-linear hyperbolic tangent activation function while using few computational resources. We also study how the accuracy of the hyperbolic tangent activation function approximation changes the performance of different DNNs. The proposed approximation is configurable and its parameters can be chosen to trade off the accuracy and the overall circuit complexity. Moreover, it can be used for other activation functions such as sigmoid, sinusoid, etc. The proposed approximation achieves almost 3$\times$ better precision than previous works in the literature while using a similar amount of computational resources and a small amount of memory. This work was published in the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays in 2017 [9] entitled “Accurate and Efficient Hyperbolic Tangent Activation Function on FPGA using the DCT Interpolation Filter”, and in IEEE Annual International Symposium on Field-Programmable Custom Computing Machines in 2017 in [10] entitled “A configurable FPGA Implementation of the Tanh Function using DCT Interpolation”.

Second, we propose a single hidden layer NN multiplication-free overlay architecture with DNN-level performance. The overlay is cheap in terms of computations since it avoids multiplications and floating-point operations. Moreover, it is user friendly especially for users with no FPGA experience. In a couple of minutes, the user can configure the overlay with the network model using a traditional C code. This work was published in the International Conference on ReConFigurable Computing and FPGAs (ReConFig) in 2018 [11] entitled "An Efficient FPGA-based Overlay Inference Architecture for Fully Connected DNNs".

Third, we propose POLYBiNN, an efficient FPGA-based inference engine for DNNs using decision trees, which are binary classifiers by nature. POLYBiNN is a memory-free inference engine that drastically cuts hardware costs. We also propose a tool for the automatic generation of a low-level hardware description of the trained POLYBiNN for a given application. This work was published in the IEEE Conference on Design and Architectures for Signal and Image Processing (DASIP) in 2018 [12] entitled "POLYBiNN: A Scalable and Efficient Combinatorial Inference Engine for Neural Networks on FPGA" and got best paper award, and in Journal of Signal Processing Systems in 2019 [13] entitled "POLYBiNN: Binary Inference Engine for Neural Networks using Decision Trees".

Fourth, we propose POLYCiNN, a classifier inspired by CNNs and decision forest classifiers. POLYCiNN migrates CNN concepts to decision forests as a promising approach for reducing both execution time and power consumption while achieving acceptable accuracy in CNN applications. POLYCiNN can be efficiently mapped to simple and densely parallel hardware designs since each decision tree is implemented in a single Look-Up Table. This work was accepted for publication in the IEEE Conference on Design and Architectures for Signal and Image Processing (DASIP) in 2019 [14] entitled "POLYCiNN: Multiclass Binary Inference Engine using Convolutional Decision Forests".

## 1.4 Thesis Organization

This thesis is organized as follows. Chapter 2 introduces the basic concepts of NNs and deep learning models. Moreover, it provides a review of recent literature about the different software and hardware acceleration approaches of DNNs and CNNs. We detail the contributions of this work in the four subsequent chapters. In chapter 3, we present a configurable FPGA implementation of the tanh activation function. In chapter 4, we describe an efficient FPGA-based overlay inference engine for DNNs. We introduce POLYBiNN, a binary inference engine for DNNs in chapter 5. Chapter 6 presents POLYCiNN, a multiclass binary inference engine for CNN applications. Chapter 7 gives a general discussion about the proposed implementations. Chapter 8 concludes this thesis and outlines future research directions.

# CHAPTER 2 BACKGROUND AND LITERATURE REVIEW

In this chapter, we provide background information and a literature review on DNNs and CNNs. We begin by giving the basic principles of NNs in terms of their architectures and working theorems. We describe the concept of deep learning and present its different models and applications. We also review the literature regarding the different implementation approaches of DNNs and CNNs. Finally, we discuss the pros and cons of the different hardware acceleration platforms for DNNs and CNNs.

## 2.1 Artificial Neural Networks

ANNs are computational models inspired by the principles of computations performed by the biological NNs of human brains [15]. Nowadays, several applications in different domains use ANNs, for example, signal processing, image analysis, medical diagnosis systems, and financial forecasting [15]. In these applications, the main role of ANNs is either classification or functional approximation (regression) [3]. In classification, the main objective is to provide a meaningful categorization or proper classification of input data. In functional approximation, ANNs try to find a functional model that smoothly approximates the actual mapping of the given input and output data.

### 2.1.1 Neural Network Components

ANNs consist of an input layer, a number of hidden layers and an output layer. Each layer is composed of a number of ANs that are considered the basic elements of NNs. These ANs receive signals from either inputs or other neurons and compute the weighted-sum of their inputs, as shown in Fig. 2.1. Theoretically, each AN serves as a gate and uses an activation function to determine whether to fire and produce an output from its input or not [3]. Usually, activation functions are applied in order to introduce non-linearity into the network.

An activation function is a transfer function that takes the weighted-sum of a neuron and transfers it to an output signal. Fig. 2.2 shows the most common activation functions that are used in different NN applications. Generally, NNs activation functions can be placed into two main categories: 1) sigmoidal and 2) non-sigmoidal activation functions [16]. The term sigmoidal function refers to any function that takes an S-shape curve. The sigmoid and hyperbolic tangent functions are bounded, differentiable and continuous sigmoidal functions.

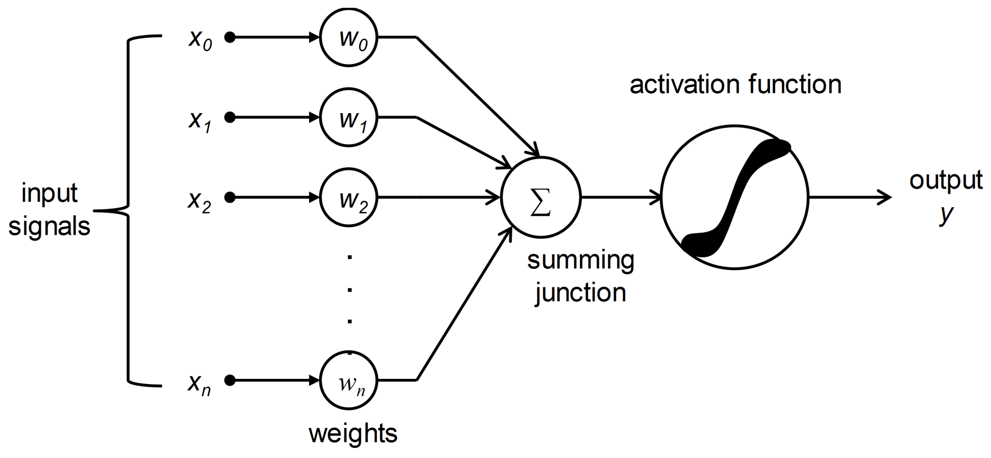


Figure 2.1 The basic components of an artificial neuron

Non-sigmoidal activation functions such as Rectified Linear Unit (ReLU) are not bounded and achieve better performance than sigmoidal functions in many applications [16]. Moreover, non-sigmoidal functions are computationally efficient as they require a simple comparison between two values. They also have a sparse representation in their output values.

An artificial neuron, or perceptron, can separate its input data into two classes in a linear relation [17]. In order to solve non-linear complex functions, several ANs are used in multiple layers where the outputs of the ANs of one layer are connected to the inputs of the next layer. A Multi-Layer Perceptron (MLP) is a feed-forward NN that consists of an input layer of ANs, followed by two or more layers of ANs with a final output layer. The term feed-forward indicates that the network feeds its inputs to the hidden layers towards the output layer in only one direction. The performance of a NN for a given application depends on the associated weights of its ANs. Those weights are determined through training the network on the given data iteratively. Once NNs are well trained and tested on a sufficient amount

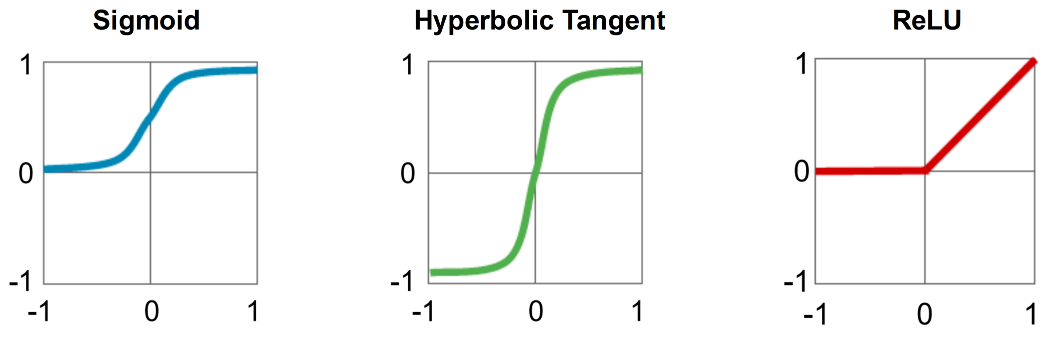


Figure 2.2 Non-linear activation functions (sigmoid, tanh and ReLU)

of data in order to generalize their classification or functional approximation surface, their models can be applied on new data.

### 2.1.2 Model Selection

The objective of model selection process in NNs is to find the network architecture with the best generalization properties, which minimize the error on the selected samples (training and validation samples) of the dataset. This process defines the number of layers of the network, number of ANs per layer, the interconnections, type of activation function, etc. These settings are called hyper-parameters that control the behaviour of the corresponding network. Conceptually, hyper-parameters can be considered orthogonal to the learning model itself in the sense that, each network architecture with a set of hyper-parameters is considered a hypothesis space. The training process of that network optimizes the hypothesis within that space. For example, we can think of model selection for a given application as the process of choosing the degree of the polynomial that best represents the given data of this application. Once the polynomial degree is determined, adjusting the parameters of the polynomial is considered as the optimization of the hypothesis.

All these hyper-parameters, alongside with learning rate, epochs, etc. must be chosen before training and are picked manually based on experience [17]. The major drawback of this strategy is the difficulty of reproducing the results as this optimization strategy takes time and depends on the experience of the developer [22]. There are some other automatic strategies such as grid search [23] and random search [22], however they do not perform as well as manual optimization especially in deep models [22].

### 2.1.3 Training Artificial Neural Networks

NNs can be classified into three main types according to how they learn: supervised, semi-supervised and unsupervised learning networks. In supervised learning, the aim is to discover a function that approximates the true function of a given training set that has sample input-output pairs [18] with a high degree of accuracy. In this case the output value of each input is available and the labeled examples of the training set serve to supervise network training. On the other hand, unsupervised NNs learn patterns of the input data even though no explicit feedback is supplied. These networks are often used in clustering applications [18]. Semi-supervised NNs have few labeled examples and a large collection of unlabeled examples. Semi-supervised NNs use both types of examples to learn the best classification or clustering model of the given data.

In supervised learning, training NNs is mainly about adjusting the network weight and bias values to minimize the loss function [18]. The loss function is defined as the amount of utility lost by approximating a model for the given data to its correct answer. In other words, the loss function measures how the predicted model fits the data. Therefore, adjusting the weight and bias values can be addressed by a hill-climbing algorithm that follows the gradient of the loss function to be optimized. Although there are complex techniques to initialize the weight and the bias values, random weight values and zero bias values are often used [20].

Generally, the error back-propagation algorithm is the most common technique for training ANNs [17]. It consists of two paths through the ANN layers: a feed-forward and a back-propagation paths. First, in the feed-forward path, the network takes the input values of a single training sample (stochastic) or a number of samples (mini-batch) or all training samples (batch). These inputs are multiplied by the initialized weight values and then the weighted-sum values are calculated and passed through the pre-defined activation function of each AN till the output layer. In the feed-forward path, the weights and bias values are fixed and are not updated. Once the outputs appear at the output layer, the loss function is computed by subtracting the output layer response from the expected output vector. In the back-propagation path, the loss function value is back propagated through the different layers of the network to update the weights and biases. The gradient descent algorithm [21] computes the gradient of the loss function with respect to each weight and bias value at the different layers. The gradient values along with the learning rate are used to update the weights and biases of the model. The feed-forward and back-propagation paths are repeated several times, called epochs, until the network is trained according to the training algorithm's constraint.

When training NNs, it is crucial to have separate training, validation and test sets [19]. The training set usually represents 70% of the overall data and it is used to train the network fitting the model to the given data. The validation set, which usually represents 15% of the data, is used to tune the ANN's hyper-parameters. It also prevents overfitting during training by performing early stopping. The key point of early stopping is to find the weight values and biases during training for which the validation set curve begins to deteriorate [19]. In other words, when the classification performance of the validation dataset meets the requirements, the training algorithms stops updating the weigh and bias value. After training the model, the test set is used to assess the generalization of the trained model on new data.

## 2.2 Deep Learning

Deep learning is a form of machine learning that uses multiple transformation steps in order to discover representative features or patterns in the given data [19]. The word deep refers to the fact that the outputs are derived from the inputs by passing through multiple layers of transformations. In deep learning, the models jointly learn the most representative complex features across the different layers of the network. Recently, impressive results were achieved in speech recognition and computer vision applications using different machine learning models such as DNNs, CNNs, and Recurrent Neural Networks (RNNs) [4, 19].

### 2.2.1 Deep Learning into Action

The use of deep learning models is not a novel concept especially that it has been proposed since developing NNs. However, deep models were out of reach as their computational complexity were exceeding the available computational platforms capabilities [5]. Three facts have facilitated the resurgence of deep learning models; a) the availability of large datasets, b) the development of faster and parallel computation units and c) the development of new sparsity, regularization and optimization machine learning techniques [4]. Nowadays, researchers and organizations can collect massive amount of data that can be used to train deep models with many parameters. In addition, techniques such as dropout and data augmentation [24] can generate more training samples from a small dataset. On the other hand, the availability of high-speed computational units such as GPUs and cloud computing are valued in deep learning since they can adequately handle large amounts of computational work quickly. Consequently, the revolution of deep learning has been enabled by the existing hardware accelerators.

### 2.2.2 Deep Neural Networks

NNs have been considered standard machine learning techniques for decades, however DNNs with higher capacity that have a larger number of hidden layers achieve better performance compared to NNs [6]. DNN is considered the basic model of deep learning that performs computations on deep FC layers. Usually, DNNs use the same activation functions used in NNs such as sigmoid and tanh. However, for DNNs, ReLU activation function is favorable in some applications since it outputs zero for all negative inputs. This is often desirable because there are fewer computations to perform and this results in concise DNN models that achieve more generalization towards the corresponding application and more immune to noise. Training DNNs is exactly the same process as training NNs with more number of hidden layers using the back-propagation algorithm.

### 2.2.3 Convolution Neural Networks

CNNs are a special kind of feed-forward networks that have achieved success for image analysis, pattern recognition and image based classification applications. CNNs process the given data in the form of arrays containing pixel values, for example a color RGB image. A typical CNN, as shown in Fig. 2.3, consists of a series of stages where each stage has three layers: a convolution layer, an activation layer and a pooling layer.

The convolution layer applies learnable filters on small spatial regions of the input image. The output of each filter is called a feature map. This process extracts the most important and representative features of the input image. Once an image has been filtered, the output feature maps are passed through a non-linear activation function in the second layer. Usually, the ReLU activation function achieves good performance when working with images [1, 4]. Finally, a pooling or a decimation layer is applied which subsamples the output feature maps from the activation layer. The main task of pooling layers is to reduce the computational complexity of the network and generalize the model more to achieve good performance of the testing dataset. The subsampling process usually uses the max operation that activates a feature if it is detected anywhere in the pooling zone. A few FC layers before the output layer link all the extracted features together and classify an input sample.

## 2.3 DNN and CNN Acceleration

GPUs, custom ASICs and FPGAs have been the main approaches for accelerating the training and the inference of a deep model [8, 25]. Not only hardware accelerators, but algorithmic and software approaches have been used as well to reduce DNN and CNN computations.

### 2.3.1 Algorithmic and Software Approaches

There are some algorithmic trends in accelerating deep models [25]: using more compact data types, taking advantage of sparsity, models compression and mathematical transforms.

Recently, it has been shown that deeper models can achieve higher accuracies than simple models [2]. However, deeper models require more computations and memory accesses to perform their tasks. For example, ResNet [26] has significantly increased the top-5 accuracy on the Imagenet [27] dataset to 96.4% compared to 84.7% using AlexNet [1]. However,

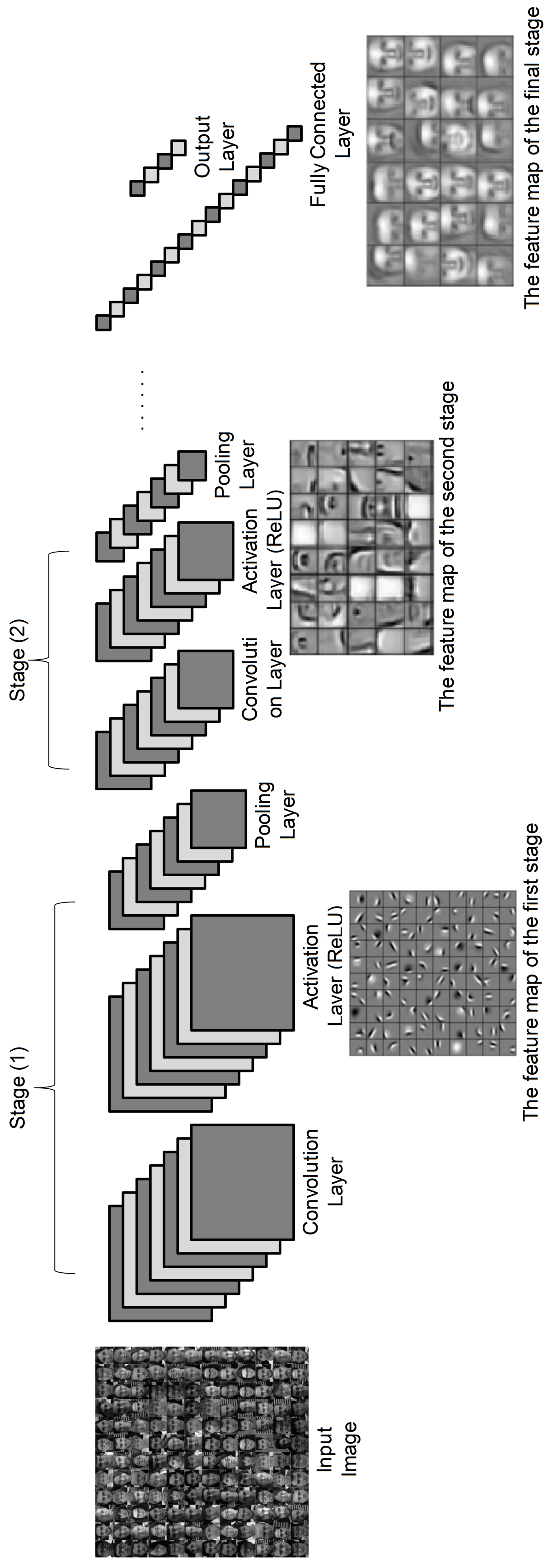


Figure 2.3 Convolution neural network architecture

ResNet uses 152 layers and requires 11 B Floating Point Operations Per Second (FLOPS) while AlexNet has only eight layers and requires 1.5B FLOPS. The computational efficiency of the network can be improved by using more compact data types. Usually, double floating-point data representation is used in the training process of DNNs. However, many researchers have shown that it is possible to train and perform the inference process using fixed-point data representation [6, 28–30]. Moreover, more compact data representations such as BNNs [30] have achieved performances comparable to fixed and floating-point data representations. BNNs can significantly reduce network parameter storage. In addition, the multiply and the accumulation processes are replaced by boolean operators which dramatically cut the required computations of the network.

Sparsity is another trend in DNNs that reduces their computational complexity in the training and the inference stages. Albreicio et al. [31] reported that more than 50% of the AN values of some popular DNNs are zeros. Moreover, the ReLU function, which is the most commonly used activation function [4], helps in building sparse activations since it outputs zero for negative inputs. Computations on such zero-valued activations are unnecessary. This results in transforming the traditional matrix multiplications to sparse matrix multiplications that require fewer operations than dense matrix [25]. Furthermore, Han et al. [28] proposed a pruning technique that zeros out the non-important neurons to generate a sparse matrix. This reduces the computational complexity of the network while maintaining comparable accuracy. Moreover, several compression techniques of DNNs have been proposed such as weight sharing [28] and hashing function [32].

Several software libraries have been developed for designing, simulating and evaluating DNNs and CNNs. Most of these libraries allow users to deploy their models to different hardware platforms i.e. CPUs and GPUs. Some popular examples of open source NN libraries are Fast Artificial Neural Network (FANN) developed in C [34], OpenNN developed in C++ [35] and OpenCV library for NNs developed in C/C++ [36]. Moreover, open source software platforms for machine learning, e.g. Tensorflow [37] and PyTorch [38] can realize DNN and CNN applications. Although GPUs are powerful hardware accelerators mainly designed for image and video processing application, they can also be used to accelerate any parallel single instruction multiple data application which exactly fits to NN applications. GPUs are working in the sense that the users have to translate their program into an explicit graphics language, e.g., OpenGL. However, NVIDIA introduced CUDA [39], a C/C++ language that enables more straightforward programming on the parallel architecture of a GPU. Therefore, we consider GPUs as a software approach as most of the discussed software libraries e.g. Tensorflow, PyTorch and Matlab deep learning library [40] are compatible with GPUs without any required modifications by the users.

DNNs are represented in software as control programs that operate on memory locations containing the inputs and the output data of each AN. These ANs are connected through pointers that are flexible. The execution time of DNN applications with software approaches depends on the processor performance of the host computing platform. Moreover, the instruction set of these processors are not specific for DNN applications. Therefore, the major drawback of DNN software-based implementations is the slow execution process [33]. Moreover, the execution time is affected by the memory access latency in order to read/write the required data/result, respectively. Therefore, hardware accelerators are highly desired for DNN and CNN applications [8].

### 2.3.2 GPU Acceleration

GPUs are often preferred for computations that exhibit regular parallelism and demand high throughput. In addition, GPUs are high-level language programmable devices and their development process is accessible to most designers. Consequently, GPUs are popular in machine learning applications as they match the nature of these models [8] [17] [19]. Recently, GPUs offer increased FLOPS, by incorporating more floating-point units, on-chip RAMs, and higher memory bandwidth [25]. However, GPUs still face challenges in deep learning models such as handling irregular parallel operations. In addition, GPUs support only a fixed set of native data types and they do not have the ability to support other custom different data types. GPUs still cannot meet the performance requirements of several real-time machine learning applications [17] [41] [42]. It might take weeks or even months in order to train some DNN applications. In addition, GPUs usually consume a significant amount power which is problematic especially for mobile and embedded devices.

### 2.3.3 ASIC-based Acceleration

ASICs can optimize both area and performance for a given DNN application in a better way than FPGAs since they are specific for a given application. The main concerns when using NN ASIC accelerators are the time to market, the cost and the low-level of flexibility of the implementation. Han and colleagues [43] introduced a compression technique for DNN weights and also proposed skipping the computational activities of zero weights. They implemented their proposed Efficient Inference Engine (EIE) for DNNs on ASIC 45nm technology. On nine fully connected layers benchmark, EIE outperforms CPU, GPU and mobile GPU by factors of 189×, 13× and 307×, respectively. It also consumes 24,000×, 3,400× and 2,700× less energy than CPU, GPU and mobile GPU, respectively. Wang et al. [44] proposed a low power CNN on a chip by quantizing the weight and bias values. The proposed dynamic quantization method diminishes the required memory size for storing the weight and bias values and reduces the total power consumption of the implementation.

Eyeriss by Chen et al. [45,46], which is an energy efficient reconfigurable CNN chip, uses 16-bit fixed point instead of floating data representation. The authors mapped the 2D convolution operations to 1D convolution across multiple processing engines. The Eyeriss chip provides 2.5× better energy efficiency over other implementations. The authors extended their work and proposed Eyeriss v2, which is a low-cost and scalable Network-on-Chip (NoC) design that connects the high-bandwidth global buffer to the array of processing elements in a two-level hierarchy [46]. This enables high-bandwidth data transfer and high throughput. Following Eyeriss chip, Adri et al. [47] proposed the YodaNN that uses binary weights instead of 16-bit fixed point data representation.

DianNao [48] is an ASIC DNN accelerator designed with a main focus of minimizing off-chip communication. The accelerator achieves up to 452 Giga Operations Per Second (GOPS) while consuming 485 mW and running at 980 MHz. The authors extended DianNao accelerator to a multi-mode supercomputer for both DNN inference and training [49, 50]. The chip consumes 16 W while running at 600 MHz, and it allows the use of 16-bit precision for inference and 32-bit precision for training with minimal effect on accuracy. Cnvlutin [31] is an ASIC CNN accelerator that is designed to skip ineffectual operations in which one of the operands is zero. The authors reported 4.5% area overhead and 1.37 higher performance compared to DaDianNao [50] without any loss in accuracy.

To alleviate the limitations of fixed-bitwidth ASIC accelerators, Sharma et al. [51] proposed BitFusion, a dynamic bithwidth DNN accelerator with 16-bit fixed-point arithmetic computational units that can be combined to create higher precision arithmetic units. Google's Tensor Processing Unit (TPU) [52] is another ASIC inference accelerator that contains a 2D systolic matrix multiply unit that uses 8-bit fixed-point precision. It runs at 700 MHz and achieves a peak performance of 92 Tera Operations Per Second (TOPS). The TPU work has been extended to support both training and inference with a peak performance of 11.5 peta FLOPS for a single TPU [53].

### 2.3.4 FPGA-based Acceleration

Although FPGAs are used in accelerating many applications, they still have some disadvantages that might limit their roles in deep learning applications. One issue with FPGAs is that in terms of power consumption they consume more power than ASICs. In addition, there is no control over the power optimization in FPGAs which is not the case in ASICs. Another issue is that the design size is limited by the available resources of the target FPGA device. Moreover, the synthesis, place and route processes of the design on FPGAs take time and these processes should be repeated after each new design.

There have been several attempts to implement both the feed-forward and back-propagation algorithms on FPGAs. Zamarano and his colleagues [54] implemented on FPGAs the back-propagation and feed-forward algorithms that are used in the training, the validation and the testing processes of NNs. They introduced a different AN representation by combining the ANs of the input and the first hidden layers in order to reduce the computational resources usage. Yu et al. [55] presented an FPGA-based accelerator for NNs, but it does not support variable network size and topology. Since memory capacities of FPGAs are limited, Park and his colleagues [56] proposed a NN accelerator on FPGA that uses 3-bit weights and a fixed point weight optimization in the training phase. This approach allows storing the NN weight values on the on-chip memory. This reduces the external memory access latency and speeds-up the overall implementation. Wang et al. [57] proposed a scalable deep learning accelerator unit on FPGA. It employs three different pipelined processing units: tiled matrix multiplication, part sum accumulation and activation function acceleration units. The proposed inference implementation achieves 36.1× speedup compared to current CPU implementations with reasonable hardware cost and lower power utilization on the MNIST dataset.

Zhang et al. [58] presented an analytical methodology for design space exploration on FPGA to find the design parameters that result in highest throughput within the given resource and memory bandwidth constraints for each convolutional layer separately. This work was extended to a multi-FPGA cluster instead of a single FPGA [59] that uses 16-bit fixed point precision. Li et al. [60] proposed a high performance FPGA-based accelerator for the inference of large-scale CNNs. They implemented the different layers to work concurrently in a pipelined structure to increase the throughput. In the fully connected layer, a batch based computing method is adopted to reduce the required memory bandwidth. The proposed implementation was tested by accelerating AlexNet, it can achieve a performance of 565.94 GOPS at 156 MHz. Targeting more flexible FPGA architectures, Ma et al. [61] presented a scalable RTL compilation of CNNs onto FPGA. They developed a compiler that analyzes a CNN structure and generates a set of scalable computing primitives that can accelerate the inference of the given CNN model. The authors tested their idea on accelerating AlexNet. On Altera Startix-V FPGA, the implementation achieves 114.5 GOPS. Recently, mathematical optimizations such as Winograd Transform and Fast Fourier Transform (FFT) have been used to decrease the number of arithmetic operations required when implementing DNNs and CNNs in FPGAs [62, 63].

Binary DNNs and CNNs have proposed using extremely compact 1-bit data types for both the weights and biases values [64]. This novel idea massively simplifies the computations of the weight-sums of ANs by replacing the matrix multiplications by XNOR and bit-counting operations. Umuroglu et al. [65] implemented a framework for fast scalable binarized NNs. They implemented the fully connected, pooling and convolution layers of the MNIST CNN on a ZC706 embedded FPGA platform consuming less than 25 W total system power. They reported 95.8% accuracy while processing 12.3M images per second of the MNIST CNN. Moreover, Fraser et al. [66] demonstrated numerous experiments on binary NNs and a Framework for Fast, Scalable Binarized Neural Network Inference (FINN) in order to show the scalability, flexibility and performance of FINN on large and complex deep models. On the other hand, Alemdar et al. [67] implemented a fully connected ternary weight NN that trains the weights with only three values (-1, 0, 1). They reported that the proposed implementation processes 255K frames per second on the MNIST dataset.

Venieris et al. [68] introduced fpgaConvNet, a framework that takes a CNN model described in the C programming language and generates a Vivado High-Level Synthesis (HLS) hardware design. The framework employs a set of transformations that explore the performance-resource design space. Wei et al. [69] proposed another framework that generates a high-performance systolic array CNN implementation in FPGAs from a C program. Noronha et al. [70] proposed LeFlow, a tool that maps numerical computation models such as DNNs and CNNs written in Tensorflow to synthesizable hardware in FPGA.

FPGAs have advanced significantly in recent years by incorporating large on-chip RAMs, large amount of Look-Up Tables (LUTs) and Digital Signal Processing (DSP) slices for complex arithmetic operations. In addition, the off-chip memory bandwidth is also increasing with the integration of recent memory technologies. Moreover, the development process on FPGAs has become much easier using recent tools such Vivado HLS. Such tools allow C/C++ algorithms to be compiled and synthesized which made the development process easier to software developers. Hence, FPGAs have the opportunity to do increasingly well on the next-generation DNNs and CNNs as they become more irregular and use custom data types.

### 2.3.5 DNN and CNN Acceleration Summary

Figure 2.4 summarizes the four approaches of accelerating DNNs and CNNs detailed in section 2.3. The approaches can be classified into four major directions. The first approach employs different computational transforms to vectorize the implementations and reduce the number of arithmetic operations occurring during inference. The second approach is the compression

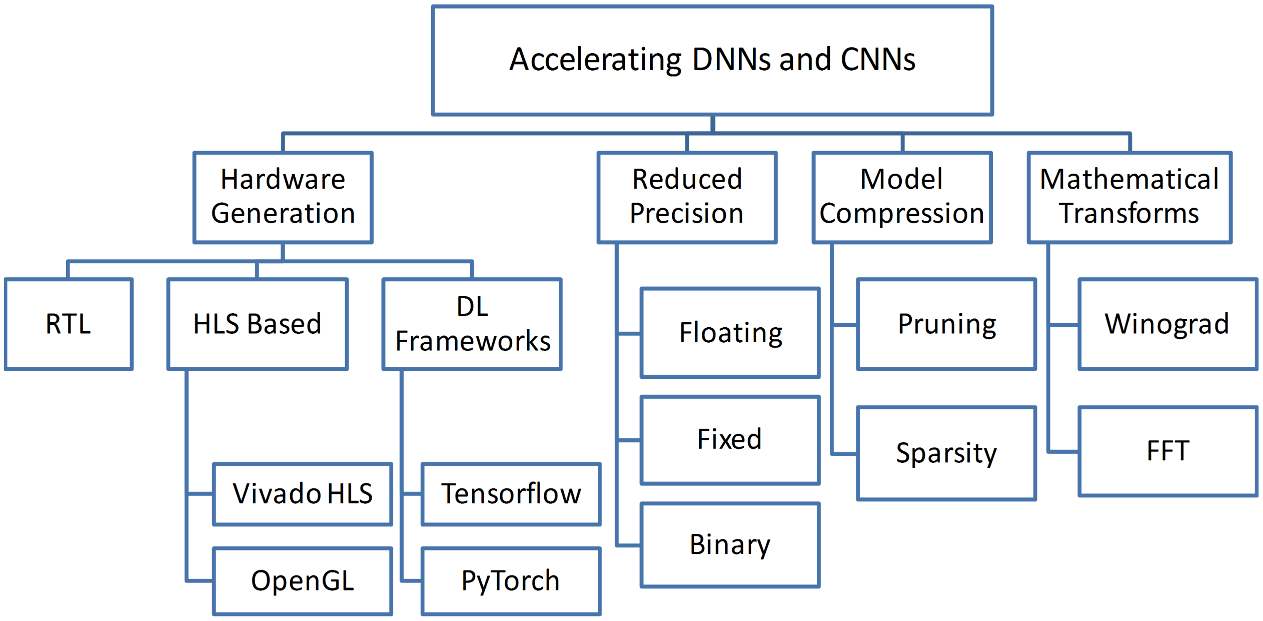


Figure 2.4 Different approaches of accelerating DNNs and CNNs on FPGAs

of the model and removing the unnecessary ANs and connections. The third approach is the use of reduced precision weights and activations without critically affecting DNN and CNN performances. The fourth approach is related to how to describe any DNN or CNN model on FPGA.

## 2.4 Summary and Research Objectives

The DNN and CNN acceleration approaches can be used separately or together to achieve the required performance for a given application on FPGAs. The two main approaches that are used to accelerate DNNs and CNNs on FPGAs are the reduced precision and the hardware generation. The reduced precision approach of DNNs and CNNs suits the nature of FPGAs especially when it comes to using binary precision. The hardware generation approach solves the difficulty of expressing DNN and CNN models in HDL. Recent FPGA-based CNN and DNN accelerators can outperform GPUs in terms of performance while consuming less power but they still pose special challenges. One issue is that FPGAs have limited or costly computational capabilities, which hinders the realization of large DNNs and CNNs. Another challenge is that the synthesis, place and route processes can take an unacceptable amount of time. In addition, the process of describing DNNs or CNNs on FPGAs often involves modeling in a HDL, for which designer productivity tends to be lower.

The existing high level synthesis tools typically generate a non-optimized hardware solutions for DNNs and CNNs. On the other hand, expressing DNNs and CNNs with Register Transfer Logic (RTL) take a long development round. Therefore, in this thesis, we focus on the reduced precision and hardware generation approaches.

The main goal of this thesis is to propose a designer friendly and efficient FPGA-based inference architectures suitable for DNN and CNN applications with varied computational and memory requirements. In order to reach our goals, the following specific objectives are identified:

- Propose adequate optimized hardware architectures of DNNs and CNNs in FPGAs. The proposed architectures should satisfy precise performance while consuming few computations and memory.
- Develop tools able to generate optimized low-level HDL description automatically for the proposed architectures.
- Simulate, implement, test and evaluate the proposed architectures to assess their performance, and compare them to the existing works.

# CHAPTER 3 ARTICLE 1: A CONFIGURABLE FPGA IMPLEMENTATION OF THE TANH FUNCTION USING DCT INTERPOLATION

Authors: Ahmed M. Abdelsalam, J.M. Pierre Langlois and F. Cheriet

Published in: IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) 2017 [10]

***Abstract*–Efficient implementation of non-linear activation functions is essential to the implementation of deep learning models on FPGAs. We introduce such an implementation based on the Discrete Cosine Transform Interpolation Filter (DCTIF). The proposed interpolation architecture combines simple arithmetic operations on the stored samples of the hyperbolic tangent function and on input data. It achieves almost 3$\times$ better precision than previous works while using a similar amount computational resources and a small amount of memory. Various combinations of DCTIF parameters can be chosen to trade off the accuracy and the overall circuit complexity of the tanh function. In one case, the proposed architecture approximates the hyperbolic tangent activation function with 0.004 maximum error while requiring only 1.45 kbits BRAM memory and 21 LUTs of a Virtex-7 FPGA.**

## 3.1 Introduction

Deep Neural Networks (DNN) have been widely adopted in several applications such as object classification, pattern recognition and regression problems [4]. Although DNNs achieve high performance in many applications, this comes at the expense of a large number of arithmetic and memory access operations [71]. Therefore, DNN accelerators are highly desired [8]. FPGA-based DNN accelerators are favorable since FPGA platforms support high performance, configurability, low power consumption and quick development process [8].

DNNs consist of a number of hidden layers that work in parallel, and each hidden layer has a number of Artificial Neurons (AN) [4]. Each neuron receives signals from other neurons and computes a weighted-sum of these inputs. Then, an activation function of the AN is applied on this weighted-sum. One of the main purposes of the activation function is to introduce non-linearity into the network. The hyperbolic tangent is one of the most popular non-linear activation functions in DNNs [4]. Realizing a precise implementation of the tanh function in hardware entails a large number of additions and multiplications [72]. This implementation would greatly increase the overall resources required for implementing a single AN and a fully parallel DNN. Therefore, approximations with different precisions and resources are generally employed [8].

There are several approaches for the hardware implementation of the tanh function based on Piecewise Linear, Piecewise Non-Linear (PWL), Lookup Table (LUT) and hybrid methods. All of these approaches exploit the fact that the tanh function is negatively symmetric about the Y-axis. Therefore, the function can be evaluated for negative inputs by negating the output values of the same corresponding positive values and vice versa. Armato et al. [73] proposed to use PWL which divides the tanh function into segments and employs a linear approximation for each segment. Zhang and his colleagues [74] used a non-linear approximation for each segment. Although both methods achieve precise approximations for the tanh function, this comes at the expense of the throughput of the implementation. LUT-based approximations divide the input range into sub-ranges where the output of each sub-range is stored in a LUT. Leboeuf et al. [75] proposed using a classical LUT and a Range Addressable LUT to approximate the function. LUT-based implementations are fast but they require more resources than PWL to achieve the same accuracy. Therefore, most of the existing LUT-based methods limit the approximation accuracy to the range [0.02, 0.04].

Several authors have observed that the tanh function can be divided into three regions: Pass Region, Processing Region (PR) and Saturation Region as shown in Fig. 3.1. The tanh function behaves almost like the identity function in the Pass Region, and its value is close to 1 in the Saturation Region. Some hybrid methods that combine LUTs and computations were used to approximate the non-linear PR. Namin and his colleagues [76] proposed to apply a PWL algorithm for the PR. Meher et al. [77] proposed to divide the input range of the PR into sub-ranges, and they implemented a decoder that takes the input value and selects which value should appear on the output port. Finally, Zamanloony et al. [72] introduced a mathematical analysis that defines the boundaries of the Pass, Processing and Saturation Regions of the tanh function based on the desired maximum error of the approximation.

In this paper, we propose a high-accuracy approximation using Discrete Cosine Transform Interpolation Filter (DCTIF) [78]. This paper is based on a previously presented abstract [9]. The proposed approximation achieves higher accuracy than the existing approximations, and it needs fewer resources than other designs when a high precision approximation is required. The rest of the paper is organized as follows: the operation principle of the proposed DCTIF approximation is described in Section 3.2. In Section 3.3, an implementation of the proposed approximation is detailed. Section 3.4 is dedicated to the experimental results and a comparison with other approximations. Finally, Section 3.5 concludes the paper.

## 3.2 DCT Interpolation Filter Design

The DCT-based Interpolation Filter (DCTIF) interpolates data points from a number of samples of a function [78]. First introduced for interpolating fractional pixels from integer pixels in the motion compensation process of the latest video coding standard H.265 [78]. DCTIF interpolates values with a desired accuracy by controlling the number of samples involved in the interpolation process and the number of interpolated points between two samples. We propose to use DCTIF to approximate the tanh function.

The DCT transformation used to generate DCTIF coefficients is defined by (3.1), where $L_{max}$ and $L_{min}$ define the range of the given sample points used in the interpolation process, *Size* is defined as ($L_{max}$ - $L_{min}$ + *1*) and the center position of a given size is *Center* = ($L_{max}$ + $L_{min}$)/2. By substituting by (3.1) in the inverse DCT formula defined in (3.2), we get the DCTIF co-efficients generation formula for position $i+r\alpha$ as in (3.3).

As shown in Fig. 3.1, let's assume that $\{p_{2M}\}$ denotes a set of *2M* given sample points (no. of DCTIF filter's tabs) used to interpolate $p_{i+r\alpha}$ at fractional position $i+r\alpha$ between two adjacent samples at positions $i$ and $i+1$ of the function *x(n)*. The parameter $\alpha$ is a positive fractional number that is equal to ($2^{-j}$) where $j$ is the number of interpolated points between two sample points. The parameter $r$ is a positive integer that represents the position of the interpolated point between two sample points where it is $\in$ [1, $2^{j}$-1]. A fractional position value $p_{i+r\alpha}$ is interpolated using an even number of samples when $r\alpha$ is equal to 1/2 , which means that the interpolated point is exactly between two adjacent samples. Otherwise, $p_{i+r\alpha}$ is interpolated using an odd number of samples since the interpolated point is closer to one of the samples than the other. Therefore, (3.3) is modified to generate the DCTIF co-efficients for even and odd numbers of tabs as in (3.4) and (3.5), respectively.

The DCTIF co-efficients can be smoothed using a smoothing window of size $W$ [78]. For hardware implementation, the smoothed co-efficients are scaled by a factor of ($2^{s}$) and rounded to integers, where $s$ is a positive integer value. In addition, the scaled co-efficients should be normalized which means that their summation is equal to $2^{s}$. Consequently, (3.6) defines the final DCTIF co-efficients.

Table 3.1 shows the generated DCTIF co-efficient values using different numbers of DCTIF tabs, $r\alpha$ values and scaling factors by substituting in (3.6). The co-efficient values exhibit similarity among some $r\alpha$ positions. For example, the $i$+1/4 and $i$+3/4 positions have

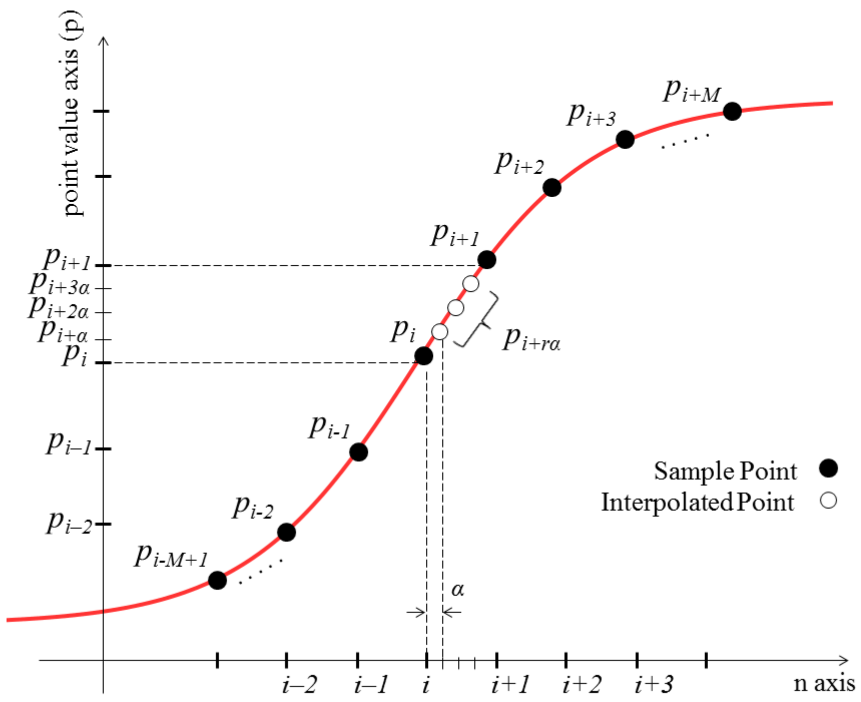


Figure 3.1 DCTIF approximation for tanh function

the same set of co-efficient values. Moreover, at the $i$+1/2 position, the set of co-efficients is symmetric about the center element. These properties can be exploited to reduce the implementation cost.

$$X(k) = \sqrt{\frac{2}{Size}} \sum_{n=L_{min}}^{L_{max}} \left( x(n) \,.\, \cos\left( \frac{2n - \ (2 \ \times \ Center) \ + \ Size}{2 \ \times \ Size} \right) \pi k \right) \tag{3.1}$$

$$x(n) = \sqrt{\frac{2}{Size}} \sum_{k=L_{min}}^{L_{max}} \left( X(k) \,.\, \cos\left( \frac{2n - \ (2 \ \times \ Center) \ + \ Size}{2 \ \times \ Size} \right) \pi k \right) \tag{3.2}$$

$$\begin{aligned} x(i + r\alpha) = \frac{2}{Size} \sum_{k=L_{min}}^{L_{max}} \Bigg( \cos\left( \frac{2n - \ (2 \ \times \ Center) \ + \ Size}{Size} \right) \pi k \ . \\ \cos\left( \frac{2(i + r\alpha) - \ (2 \ \times \ Center) \ + \ Size}{Size} \right) \pi k \end{aligned} \tag{3.3}$$

$$filter_{even}(i + r\alpha) = \frac{1}{M} \sum_{k=0}^{2M-1} \left( \cos\left( \frac{2n - 1 + 2M}{4M} \right) \pi k \ . \ \cos\left( \frac{2r\alpha - 1 + 2M}{4M} \right) \pi k \right) \tag{3.4}$$

$$filter_{odd}(i + r\alpha) = \frac{2}{2M+1} \sum_{k=0}^{2M} \left( \cos\left(\frac{2n+1+2M}{2(2M+1)}\right) \pi k \ . \ \cos\left(\frac{2r\alpha+1+2M}{2(2M+1)}\right) \pi k \right) \tag{3.5}$$

$$Filter_{even/odd}(i + r\alpha) = filter_{even/odd}(i + r\alpha). \ \cos\left(\frac{n - r\alpha}{W-1}\right) \pi \ . \ 2^s \tag{3.6}$$

## 3.3 Proposed DCTIF Architecture

The proposed DCTIF approximation divides the input range of the tanh function into Pass, Processing and Saturation Regions. The boundaries of these regions are computed based on the targeted maximum error of the approximation [72]. The output is equal to the input when the input is in the Pass Region. The proposed DCTIF approximation is utilized for the inputs in the Processing Region. In the Saturation Region, all the bits of the output port are set to one which represents the maximum value of the output signal.

The block diagram of the proposed architecture is shown in Fig. 3.2. It is composed of a 4-input multiplexer that selects the appropriate output based on the input range decoder that determines the proper region of its input value. The decoder has four possible outputs that represent a) Pass Region, b) Saturation Region, c) Processing Region and the output is stored as a sample and finally d) Processing Region and the output of the given input needs to be interpolated. The truncation process, shown in Fig. 3.2, is implemented in order to pass the $\mathrm{N_{out}}$ fraction bits of the input. The implementation cost of the DCTIF approximation, shown in Fig. 3.2, depends on the number of tabs and the values of $s$ and $\alpha$. Figure 3.3 shows the DCTIF implementation using four tabs, $s = 4$ and $\alpha = 1/4$ where the co-efficient values are shown in Table 3.1.

Table 3.1 DCTIF co-efficient values for tanh function approximation

| No. of Tabs | Position ($\alpha$+$ri$) | Filter Co-efficients for $s = 4$ | Filter Co-efficients for $s = 5$ |
|---|---|---|---|
| 4 | $i$+1/4 | {-2, 15, 3, 0} | {-3, 29, 6, 0} |
| | $i$+1/2 | {-2, 10, 10, -2} | {-3, 19, 19, -3} |
| | $i$+3/4 | {0, 3, 15, -2} | {0, 6, 29, -3} |
| 6 | $i$+1/4 | {1, -2, 14, 4, -1, 0} | {1, -5, 29, 9, -2, 0} |
| | $i$+1/2 | {1, -3, 10, 10, -3, 1} | {1, -5, 20, 20, -5, 1} |
| | $i$+3/4 | {0, -1, 4, 14, -2, 1} | {0, -2, 9, 29, -5, 1} |

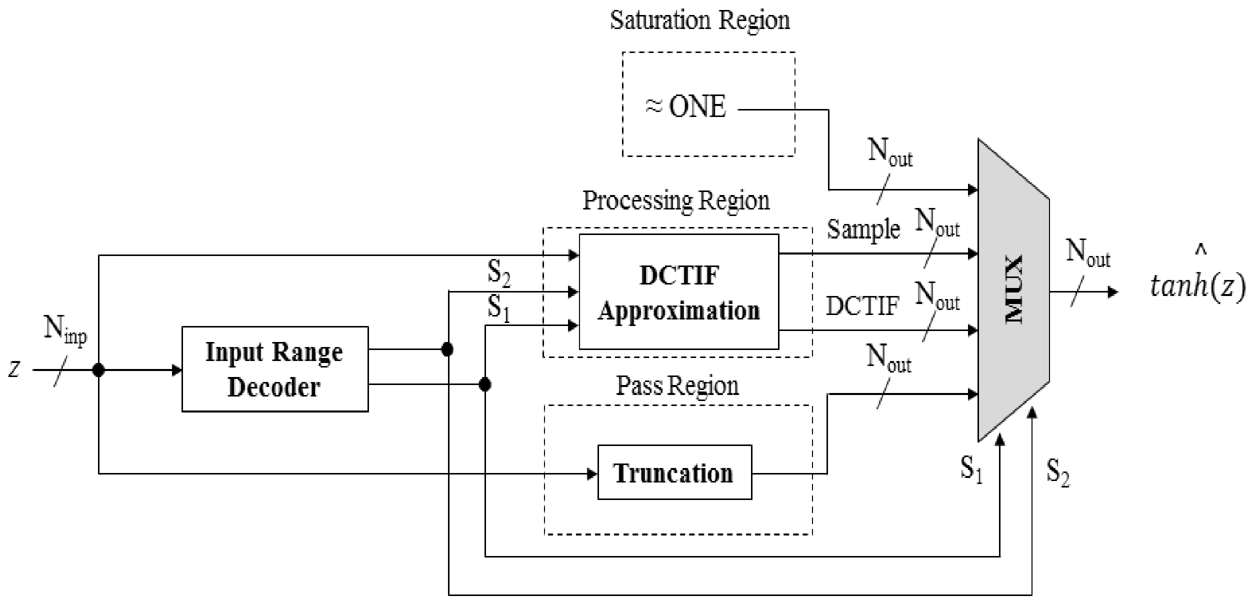


Figure 3.2 Block diagram of the proposed tanh approximation

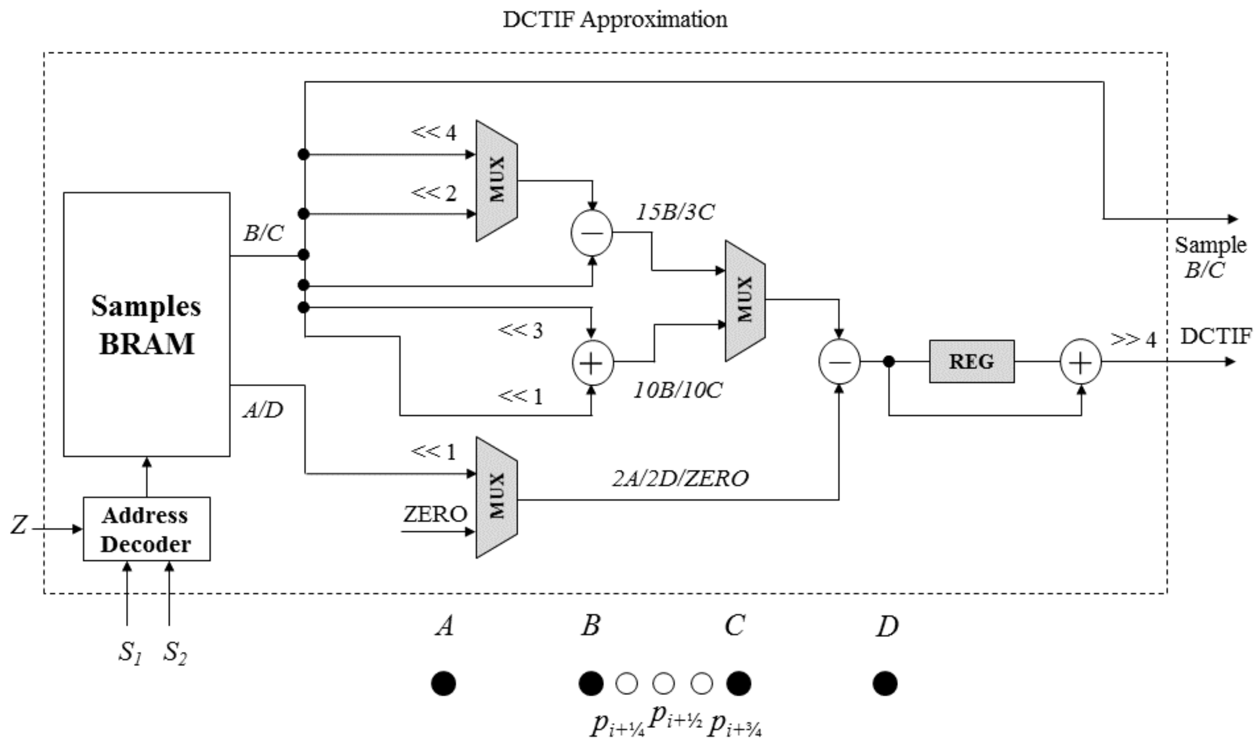


Figure 3.3 The proposed DCTIF approximation architecture using 4 tabs, $\alpha = 1/4$, $s = 4$

The address decoder of the DCTIF approximation, shown in Fig. 3.3, takes the input value $Z$ and the select lines $S1$ and $S2$ of the input range decoder. It generates the addresses of the required samples (A, B, C, D) stored in the BRAM for the interpolation process. The samples A, B, C and D correspond to samples $p_{i-1}$, $p_i$, $p_{i+1}$ and $p_{i+2}$, respectively, in Fig. 3.1. Since the $p_{i+1/4}$ and $p_{i+3/4}$ interpolation equations are symmetric, the same hardware can be used to interpolate them. Therefore, we only implement the interpolation equations of $p_{i+1/4}$ and $p_{i+1/2}$. In order to reduce the area required for the proposed implementation, we divide the computation of $p_{i+1/4}$ and $p_{i+1/2}$ equations into four pairs (-2A + 15B), (3C + 0D), (-2A + 10B) and (10C - 2D). A set of three multiplexers, two subtractors and one adder, shown in Fig. 3.3, is used to calculate the output value of any of these pairs. Each pair of these simple equations is computed in one clock cycle and the full equation takes two clock cycles to be calculated using an accumulator. A single cycle computation would also be possible, at the expense of more resources. Finally, the outputs of the DCTIF interpolation block are the interpolated value and the stored sample B when the input has its tanh output as a stored sample.

## 3.4 Experimental Results

The proposed DCTIF approximation was described in Verilog HDL and synthesized for a Virtex-7 XC7VX550T FPGA using Xilinx ISE 14.6. Table 3.2 compares the implemented DCTIF approximation to previous works in terms of maximum error, computational resources and throughput. The maximum error is the maximum absolute difference between the exact tanh function and the approximated tanh function for all input values. The DCTIF method can approximate the tanh function accurately with 0.00001 maximum error using 64-bit floating point data representation.

Table 3.2 shows the results of the proposed DCTIF approximation while using fixed-point data representation of both input and output values. In all cases, we use a 3 bits for the integer part of input values. Moreover, the same number of bits for both input and output values are used. That is why $N_{in}$ is always equal to $N_{out}$ + 3. Table 3.2 shows that the proposed DCTIF approximation using 8-bit outputs achieves 0.004 maximum error while using only 21 LUTs and 1.45 kbits of memory which represents 0.0034% of the available BRAMs on the FPGA device. The DCTIF approximation can achieve higher accuracy when using more precise data representation of input and output values. However, this comes at the expense of both computational and memory resources. All existing works have been implemented as ASICs using TSMC 180 nm$^2$ technology. The most accurate approximation achieves 0.0178 maximum error using 1,791 gates [75]. Zamanloony and colleagues [72] achieved 0.0196

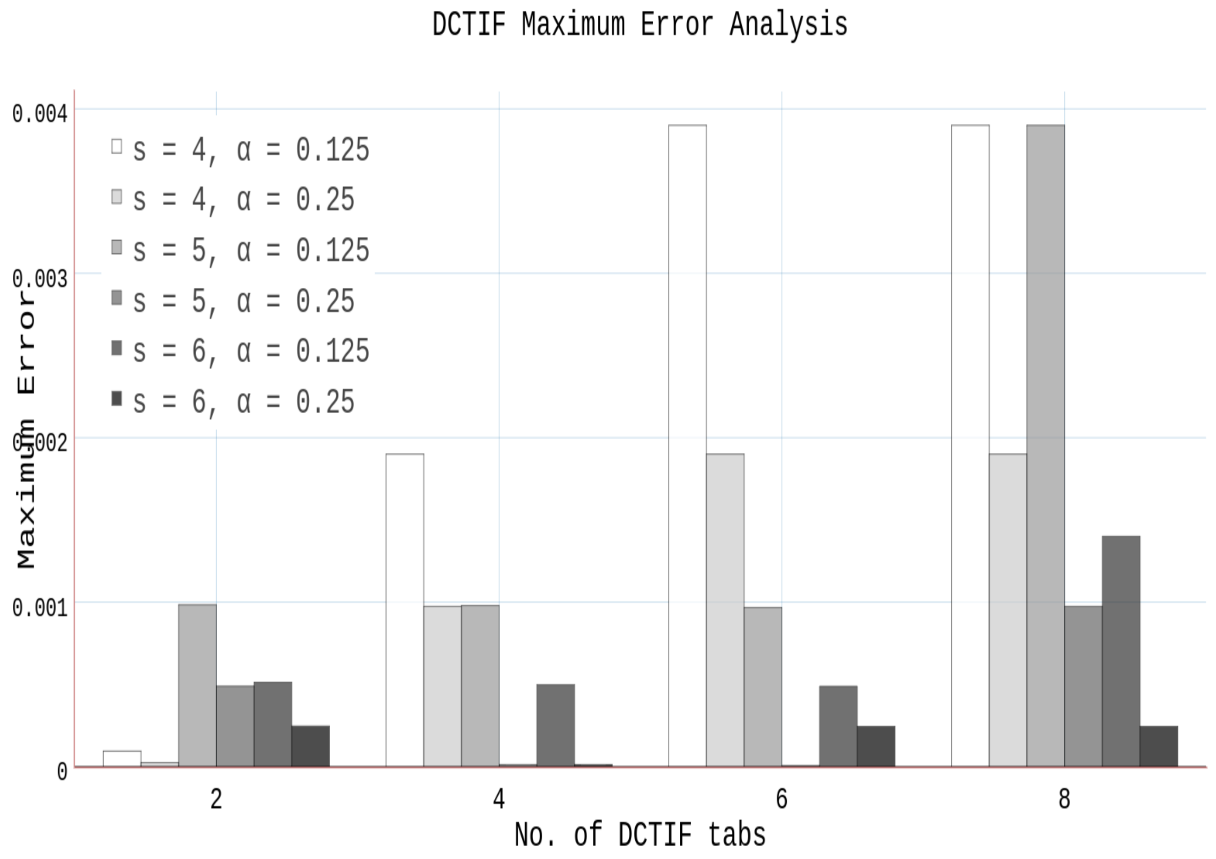


Figure 3.4 DCTIF tanh approximation accuracy vs no. of tabs, $\alpha$ value and the scaling parameter $s$ using double floating-point data representation

maximum error using only 129 gates. In addition, their implementation can be reconfigured in order to achieve higher accuracy at the expense of computational resources. In order to have a fair comparison, we re-implemented the design in [72] achieving 0.0118 maximum error for a Xilinx FPGA Virtex-7. We chose to re-implement the work in [72] as it requires the least amount of computational resources of all the existing implementations. Table 3.2 shows that our proposed DCTIF approximation outperforms the work in [72] by almost 3× in terms of accuracy, using a similar amount of computational resources and a small amount of memory. Therefore, our proposed DCTIF approximation thus makes a balanced usage of available FPGA resources, while outperforming existing works.

The DCTIF tanh approximation error analysis is presented in Fig. 3.4. It can be seen that the DCTIF approximation error increases for small $\alpha$ values. Although a large $\alpha$ value means that fewer points need to be interpolated, this comes at the expense of memory resources since more samples must be stored. A large value of $s$ increases the accuracy of the approximation, but increases complexity as well because the interpolation coefficients take larger values, potentially expressed with more signed digits as shown in Table 3.1. Moreover, using more DCTIF tabs comes at the expense of the computational resources.

The proposed DCTIF tanh approximation is based on interpolating the missing points in the Processing Region. High accuracy approximation can be achieved using the DCTIF approach by widening the boundaries of the Processing Region with respect to the two other regions. This directly increases the required amount of memory to store the sample values used in the interpolation process. In addition, this comes at the expense of the computational resources of the implementation as shown in Table 3.2. The proposed DCTIF approximation achieves 0.0001 maximum error, requiring 1250.5 kbits of memory and 129 LUTs. This implementation computes a value every 7.6 ns in two cycles of 3.8 ns each.

## 3.5 Conclusion

The accuracy of the activation function is a bottleneck of the performance DNNs implementations on FPGA. We proposed a high-accuracy approximation technique that is based on Discrete Cosine Transform Interpolation Filter. The proposed DCTIF approach outperforms the existing works in terms of accuracy for similar amounts of computational resources. Moreover, it achieves better approximation accuracy at the expense of computational and memory resources.

Table 3.2 Different tanh function implementations

| ASIC Results on 180 nm [2] TSMC Technology | | | | |
|---|---|---|---|---|
| Architecture | Max. Error | Area ($nm^2$) | Gate Count | Delay (ns) |
| ICCIT [75] | 0.01800 | 17864.2 | 1791 | 2.45 |
| ICCIT [75] | 0.01780 | 11871.5 | 1190 | 2.12 |
| ISCAS [76] | 0.01890 | 5130.8 | 515 | 2.80 |
| VLSI-SOC [77] | 0.02050 | 1603.3 | 161 | 2.82 |
| TVLSI [72] | 0.01960 | 1280.3 | 129 | 2.12 |
| FPGA Results on Xilinx Virtex-7 | | | | |
| Architecture | Max. Error | Slice LUTs | Memory (kbits) | Delay (ns) |
| TVLSI [72] | 0.01180 | 20 | — | 1.245 |
| DCTIF[1] $N_{in} = 11$, $N_{out} = 8$ | 0.00400 | 21 | 1.45 | 1.640 |
| DCTIF[1] $N_{in} = 12$, $N_{out} = 9$ | 0.00200 | 27 | 2.39 | 1.662 |
| DCTIF[1] $N_{in} = 13$, $N_{out} = 10$ | 0.00100 | 36 | 9.14 | 1.784 |
| DCTIF[1] $N_{in} = 14$, $N_{out} = 11$ | 0.00050 | 37 | 22.17 | 1.993 |
| DCTIF[2] $N_{in} = 19$, $N_{out} = 16$ | 0.00010 | 129 | 1250.5 | 7.632 |

[1]2-tabs, $s = 4$, and $\alpha$ = *1/4* [2]4-tabs, $s = 6$, and $\alpha$ = *1/4*

# CHAPTER 4 ARTICLE 2: AN EFFICIENT FPGA-BASED OVERLAY INFERENCE ARCHITECTURE FOR FULLY CONNECTED DNNS

Authors: Ahmed M. Abdelsalam, Felix Boulet, Gabriel Demers, J. M. Pierre Langlois and Farida Cheriet

Published in: IEEE International Conference on ReConFigurable Computing and FPGAs (ReConFig) 2018 [11]

***Abstract*–Deep Neural Networks (DNNs) have gained significant popularity in several classification and regression applications. The massive computation and memory requirements of DNNs pose special challenges for FPGA implementation. Moreover, programming FPGAs requires hardware-specific knowledge that many machine-learning researchers do not possess. To make the power and versatility of FPGAs available to a wider DNN user community and to improve DNN design efficiency, we introduce a Single hidden layer Neural Network (SNN) multiplication-free overlay architecture with fully connected DNN-level performance. This FPGA inference overlay can be used for applications that are normally solved with fully connected DNNs. The overlay avoids the time needed to synthesize, place, route and regenerate a new bitstream when the application changes. The SNN overlay inputs and activations are quantized to power-of-two values, which allows utilizing shift units instead of multipliers. Since the overlay is a SNN, we fill the FPGA chip with the maximum possible number of neurons that can work in parallel in the hidden layer. On a ZYNQ-7000 ZC706 FPGA, it is thus possible to implement 2450 neurons in the hidden layer and 30 neurons in the output layer. We evaluate the proposed architecture on typical benchmark datasets and demonstrate higher throughput with respect to the state-of-the-art while achieving the same accuracy.**

## 4.1 Introduction

Neural Networks (NNs) have shown promising performance in many applications including computer vision, speech recognition and regression problems [4]. NNs consist of an input layer, a few hidden layers and an output layer. Deep models with a large number of layers are now commonly used in the form of Deep Neural Networks (DNNs) and Convolutional Neural Networks (CNNs) [4]. These deep models often perform better than shallow NNs in terms of accuracy [6], and efforts have been made to understand why. One reason is that DNNs usually expand traditional NNs to a large scalable network with a larger number of neurons and hidden layers [6]. In other words, deep models have the ability to learn more complex functions than NNs.

Although DNNs achieve high performance in many applications, this comes at the expense of a large number of arithmetic operations and memory accesses [71]. GPUs can accelerate the training and the inference processes of large and complex deep learning models. GPUs allow developers to use deeper models and train these models with more data. User appetite for ever deeper models creates a need for other hardware accelerators with more specialized computational abilities than GPUs [8]. Moreover, GPUs consume a relatively high amount of power, which is problematic for battery-based devices and embedded systems. The high power consumption of GPUs also limits the number of cores that can be integrated to increase throughput [25]. Therefore, hardware accelerators that show good 3Ps for DNNs - Performance, Programmability and Power consumption - are highly desired [8, 79].

FPGAs can outperform GPUs in terms of performance while consuming less power [80] but they pose special challenges [81]. One issue is that the synthesis, place and route processes can take an unacceptable amount of time [82]. In addition, the process of describing a DNN for FPGA implementation often involves modeling in a Hardware Description Language (HDL), for which designer productivity tends to be lower. FPGAs also have limited or costly floating-point computational capabilities, which hinders the realization of large DNNs on FPGAs. These concerns must be addressed before FPGAs can become as popular as GPUs for DNN implementation.

The main focus of this work is to propose a designer-friendly and efficient FPGA framework suitable for DNN applications with varied computational and memory requirements. The specific contributions of this paper are as follows:

- an efficient Single hidden layer Neural Network (SNN) FPGA-based overlay inference architecture for fully connected DNNs;
- the quantification of the inputs of the overlay to power-of-two values using stochastic rounding;
- a quantized version of the tanh activation function and its hardware implementation;
- a study of the effect of quantizing the inputs and the activations on the performance of DNNs and SNNs; and,

- an FPGA implementation of the proposed overlay architecture that demonstrates remarkable improvements over existing FPGA-based accelerators.

The rest of the paper is organized as follows. Different DNNs inference engines are reviewed in Section 4.2. The proposed quantized SNN operation and analysis are described in Section 4.3. In Section 4.4, the proposed FPGA-based overlay inference architecture is detailed. Section 4.5 is dedicated to the experimental results and comparison with other architectures. Section 4.6 concludes the paper.

## 4.2 Related Work

Realizing a precise implementation of a DNN in hardware entails a large number of additions, multiplications and memory accesses [25]. Nevertheless, the hardware implementation of a DNN on an FPGA is always constrained by the available computational and memory resources [83]. A Multiply and Accumulate (MAC) operation is the basic element of each Artificial Neuron (AN) in a NN. Therefore, compression techniques of MAC operations with different accuracies and resource utilizations are employed in order to fit a DNN within the available resources of an FPGA. These compression approaches can be categorized based on weight quantization, activation quantization and DNN model compression. Weight and activation quantization implies reducing their precision, while DNN model compression reduces the required number of MAC operations by removing some ineffective weights and activations. These techniques have been combined in several DNN hardware accelerators on FPGAs [25, 79, 81].

Weight quantization involves mapping the trained network weights to a smaller set of weights quantization levels. Razlighi and colleagues [84] proposed LookNN, a neural network with no multiplications. The authors quantize the weights of each AN to a small number of clusters. They replace the multipliers by Look-Up Tables (LUTs) where each LUT stores all the possible input combinations of an AN and their corresponding outputs. Kwan et al. [85], Tann et al. [86] and Gudovskiy et al. [87] proposed different approaches to map floating-point weights to integer power-of-two values with no change in the network architecture. Although that method dramatically reduces the required computational resources, it badly affects the accuracy of the network. Most recently, Courbariaux et al. [30] proposed Binary Neural Networks (BNNs) where the weights are represented with a single bit {-1, 1}. They also proposed to quantize the activations to binary values {-1, 1} in order to replace MAC operations by XNOR gates and activation functions by counters.

Several authors noticed that BNNs with XNOR gates and counters match the nature of FPGAs. Umuroglu et al. [65] proposed FINN, a framework for fast and scalable BNN inference. The authors implemented a full BNN inference engine with fully connected, convolution and pooling layers. On the Xilinx ZYNQ-ZC706 FPGA platform, the proposed engine processes 12.3 million MNIST [88] image classifications per second with 95.8% accuracy compared to 98.2% baseline accuracy. Rastegari et al. [89] proposed XNOR-Net, they extended BNNs by keeping the first and last layers with floating-point precision. In addition, the authors multiply the outputs by a scaling factor to recover the dynamic range of the weights. This approach helps in reducing the accuracy loss of BNN. Alemdar et al. [67] proposed another way to maintain the same accuracy as floating-point DNNs with ternary NNs. The authors report 3.1× better energy efficiency with respect to the floating-point network while maintaining the same accuracy.

A different trend for DNN implementation on FPGAs involves compression techniques [25]. Deep Compression [28] compresses DNNs without loss of accuracy through a combination of pruning, weight sharing and Huffman encoding. Pruning sets non-important weights to zero, which results in a sparse weights matrix that reduces the number of required LUTs. Weight sharing involves the Huffman coding of non-zero weights and their representation with few bits.

## 4.3 Quantized SNN Operation and Performance Analysis

In this work we focus on SNN implementation. SNNs have fewer hyper-parameters than DNNs: only the number of ANs in the hidden layer must be specified. SNNs should thus lead to simpler implementations and more flexibility. However, classification accuracy should not be reduced. The universal approximation theorem [90] states that SNNs can approximate any decision boundary given a sufficient number of sigmoid ANs. Recently, Ba et al. [6] empirically demonstrated that shallow feed-forward NNs could learn complex functions and achieve almost the same accuracy as deep models. In this section, we present an SNN overlay inference architecture. We show how it operates without multipliers in the hidden and output layers by quantizing the inputs and the activations to power-of-two values. We show how it can be trained to achieve DNN-equivalent performance and demonstrate this with results.

### 4.3.1 Inputs Quantization

We quantize the inputs to power-of-two integer values in order to replace the multipliers in the hidden layer by shift units. This significantly increases the number of ANs that can be

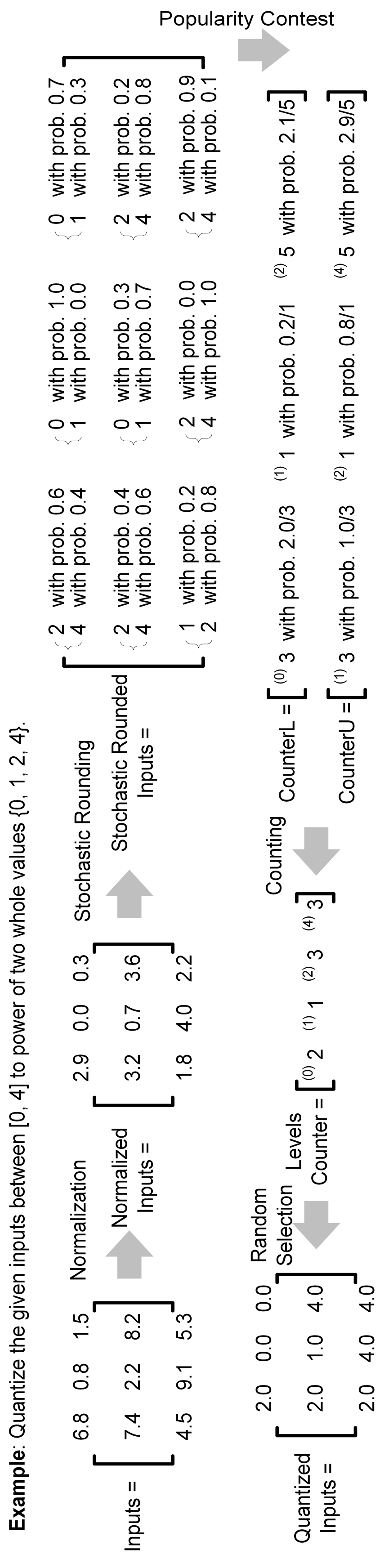

Example: Quantize the given inputs between [0, 4] to power of two whole values {0, 1, 2, 4}.
Inputs =
6.8 0.8 1.5
7.4 2.2 8.2
4.5 9.1 5.3
Normalization
Normalized Inputs =
2.9 0.0 0.3
3.2 0.7 3.6
1.8 4.0 2.2
Stochastic Rounding
Stochastic Rounded Inputs =
2 with prob. 0.6, 4 with prob. 0.4
0 with prob. 1.0, 1 with prob. 0.0
0 with prob. 0.7, 1 with prob. 0.3
2 with prob. 0.4, 4 with prob. 0.6
0 with prob. 0.3, 1 with prob. 0.7
2 with prob. 0.2, 4 with prob. 0.8
1 with prob. 0.2, 2 with prob. 0.8
2 with prob. 0.0, 4 with prob. 1.0
2 with prob. 0.9, 4 with prob. 0.1
Popularity Contest
CounterL = (0) 3 with prob. 2.0/3, (1) 1 with prob. 0.2/1, (2) 5 with prob. 2.1/5
CounterU = (1) 3 with prob. 1.0/3, (2) 1 with prob. 0.8/1, (4) 5 with prob. 2.9/5
Counting
Levels Counter = (0) 2, (1) 1, (2) 3, (4) 3
Random Selection
Quantized Inputs =
2.0 0.0 0.0
2.0 1.0 4.0
2.0 4.0 4.0

Figure 4.1 Quantizing the inputs to power-of-two values using stochastic rounding

implemented. We quantize the inputs instead of the weights because low precision weights affect DNN and SNN accuracy more than low precision inputs [91]. Input quantization is done offline as a pre-processing step. This reduces the required bandwidth between the input memory and the inference architecture. The two most popular quantization algorithms in machine learning are deterministic and stochastic rounding [30]. Deterministic rounding quantizes a value to the nearest quantization level. It involves trivial computation but it introduces high quantization error. Stochastic rounding quantizes a value to the nearest quantization level with a probability dependent on the proximity to the quantization level. Consequently, the expected stochastic rounding error is zero. Therefore, we use stochastic rounding as it leads to better results than deterministic rounding.

Assume floating-point DNN inputs in an interval [a, b]. First, we normalize the inputs to an interval [L, U], where L is zero and U takes one of the following values $\{1, 2, 4, 64, \ldots , 2^{\mathrm{s}}\}$. The parameter *s* is defined as $2^{2\mathrm{n}\text{ - }2}$ and *n* is the number of bits to represent the quantized inputs. The interval [L, U] is split into (2n - 1) sub-intervals where the boundaries of each sub-interval are power-of-two values. Then, we stochastically quantize each data point *x* to the endpoints of its interval:

$$Q^{L,U}_{2^{\mathrm{n}}}(x) = \begin{cases} 2^{\lfloor \log_2 \mathrm{x} \rfloor} & \text{with prob. } 1 - \frac{2^{\lceil \log_2 \mathrm{x} \rceil} - x}{2^{\lceil \log_2 \mathrm{x} \rceil} - 2^{\lfloor \log_2 \mathrm{x} \rfloor}} \\ 2^{\lceil \log_2 \mathrm{x} \rceil} & \text{otherwise.} \end{cases} \tag{4.1}$$

Although stochastic rounding ensures the expected value of a quantized input is equal to the normalized input itself, it is expensive in terms of data representation as both the quantized value and its probability should be considered in computations. So, we apply a popularity contest (dithering) algorithm on the quantized inputs with their probabilities in order to represent each quantized input with a single power-of-two. The dithering algorithm counts the number of quantized inputs at each lower and upper level and their corresponding probabilities. Then, the number of points at each power-of-two level is computed based on the number of points at each lower and upper level and their corresponding probabilities. Finally, the number of points at each power-of-two level is applied on the normalized inputs randomly. Fig. 4.1 shows an example of rounding the inputs stochastically to power-of-two values.

### 4.3.2 Activations Quantization

In the proposed SNN overlay, obtaining one AN output involves the computation of the *tanh* function. Approaches for the hardware implementation of the *tanh* function include the Piecewise Linear (PWL), Piecewise Non-Linear, Look-Up Table (LUT) and other hybrid

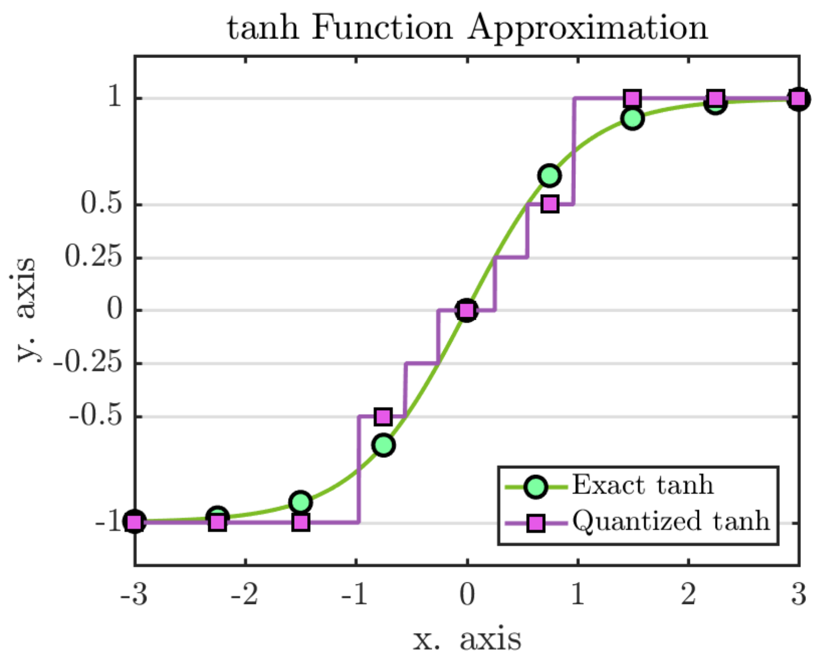


Figure 4.2 Exact versus quantized tanh activation function

methods [10]. We quantize the activations of the *tanh* function to power of two values in order to replace the multipliers in the output layer by shift units. Figure 4.2 shows exact versus approximated *tanh* function to power-of-two values. We quantize the *tanh* activation function to the seven power-of-two values {-1, -1/2, -1/4, 0, 1/4, 1/2, 1}. The *tanh* function is negatively symmetric around the Y-axis. Therefore, it can be evaluated for negative inputs by negating the output values of the same corresponding positive values and vice versa. The proposed quantized *tanh* function for positive inputs is given by:

$$Qtanh(x) = \begin{cases} 1 & \text{if } tanh(x) \geq 3/4 \\ 1/2 & \text{if } tanh(x) \geq 1/2 \\ 1/4 & \text{if } tanh(x) \geq 1/4 \\ 0 & \text{if } tanh(x) \geq 0 \end{cases} \tag{4.2}$$

### 4.3.3 Teacher-to-Student Training Algorithm

DNNs generally perform better than SNNs in complex applications such as speech recognition and image classification [4]. However, it has been shown that SNNs can achieve the same performance as DNNs with the same number of parameters [6]. Accordingly, we use the two-stage teacher-to-student training algorithm [6] to train an SNN with quantized inputs and activations. First, we train a teacher DNN on the original training floating-point data in order to be as accurate as possible. Then, the student SNN, with the same number of ANs as the teacher DNN, is trained to mimic the teacher's behavior. It works by passing the training data through the trained teacher DNN in order to collect the produced outputs by the teacher DNN. The synthetic outputs of the teacher DNN are used as labeled data to train the student network using the same training dataset. In our work, the teacher network operates with floating-point inputs and activations and the student SNN network's activations and inputs are quantized to power-of-two values as described before.

### 4.3.4 Experimental Assessment of the Multiplier-less SNN

We evaluated the performance of the SNN trained using the teacher-to-student algorithm with quantized inputs and activations with respect to state-of-the-art floating-point DNNs. All experiments were performed on four datasets (MNIST [88], Speech Recognition [92], Indoor Localization [93] and Human Activity Recognition [94]), each of which has a different data dimensionality. Table 4.1 shows the baseline NN topologies of the four datasets and their baseline accuracies. For each application, Table 4.1 summarizes the inference classification accuracies of the corresponding baseline DNN, SNN and SNN trained using teacher-to-student approach. Table 4.1 shows that SNNs with power-of-two values quantized inputs and quantized activations trained using the teacher-to-student approach can achieve the same accuracy as the baseline teacher DNNs.

In all experiments, we quantize the inputs using stochastic rounding to the set {0, 1, 2, 4, 8, 16, 32, 64}. Our experiments show that three quantization bits are sufficient for the tested applications. For the activations, we quantize the *tanh* outputs to seven power-of-two values {-1, -1/2, -1/4, 0, 1/4, 1/2, 1}. Adding more power-of-two values would improve the accuracy in the linear region of the *tanh* function, which is not needed.

## 4.4 Proposed DNN-Equivalent Inference Architecture

As discussed in Section 4.3, a SNN with quantized power-of-two inputs and activations can achieve accuracies comparable to those of baseline floating-point DNNs when trained with the teacher-to-student approach. Therefore, we propose a DNN-equivalent SNN overlay architecture that is synthesized, placed and routed only once. The AN coefficients are stored in memories that can be modified without a new synthesis and implementation cycle. The architecture size depends on the available resources in the chosen FPGA, and should include as many ANs as possible, thus allowing the implementation of the largest possible DNN-

Table 4.1 Classification performance of DNN and SNN - FP: trained using floating-point inputs and activations. Q: trained using quantized inputs and activations. QT2S: quantized inputs and activations trained using teacher-to-student approach.

| **Dataset** | **Inputs** | **Outputs** | **DNN Topology** | **SNN Topology** | **Accuracy (%)** | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | **DNN FP** | **SNN FP** | **SNN Q** | **SNN QT2S** |
| **MNIST [88]** | 784 | 10 | 784×512×512×10 | 784×1024×10 | 98.4 | 95.2 | 73.6 | 98.4 |
| **ISOLET [92]** | 617 | 26 | 617×512×512×26 | 617×1024×26 | 95.8 | 93.6 | 82.9 | 95.7 |
| **Indoor Localization [93]** | 520 | 13 | 520×512×512×13 | 520×1024×13 | 94.9 | 94.1 | 71.5 | 94.8 |
| **Human Activity Recognition [94]** | 561 | 6 | 561×512×512×6 | 561×1024×6 | 96.6 | 97.5 | 95.8 | 96.6 |

equivalent architectures. The proposed overlay architecture consists of two different AN types: hidden and output. The selection of the number of each AN type can be established from the target application class. The smallest possible number of output ANs should be selected to allow the largest possible DNN-equivalent architecture, which is determined by the number of hidden ANs.

Figure 4.3 shows the dedicated hardware implementation of hidden layer ANs. Each AN stores its corresponding weights in a BRAM. A separate BRAM stores the current quantized input shared among all hidden ANs. Input quantization is done offline as a pre-processing step using stochastic rounding, as discussed in section 4.3.1. At each clock cycle, all the hidden ANs are fed with the same input, while each hidden AN reads the corresponding weight from its corresponding weights BRAM. For each hidden AN, an accumulator computes the sum of all its weighted inputs. The accumulator needs as many clock cycles as the number of inputs produce its output. As shown in Fig. 4.3, the quantized activation function is implemented as a priority encoder which takes the output of the accumulator and produces the corresponding activation. The activations of the hidden ANs are represented with 3 bits corresponding to the quantized *tanh* values {-1, -1/2, -1/4, 0, 1/4, 1/2, 1}.

Figure 4.4 shows the implementation of an output AN, which is similar to the implementation of hidden ANs. Each output AN stores its weights in a BRAM. All output ANs share another BRAM to store the output activations of the hidden layer. In the same vein as hidden ANs architecture design, we use shift right units instead of multipliers for the output ANs since all the activations of the hidden layer are represented in power-of-two values. Moreover, we use an accumulator that computes the sum of all weighted activations of each output AN. The accumulator processes one input per clock cycle. The output ANs are fully connected to the hidden ANs.

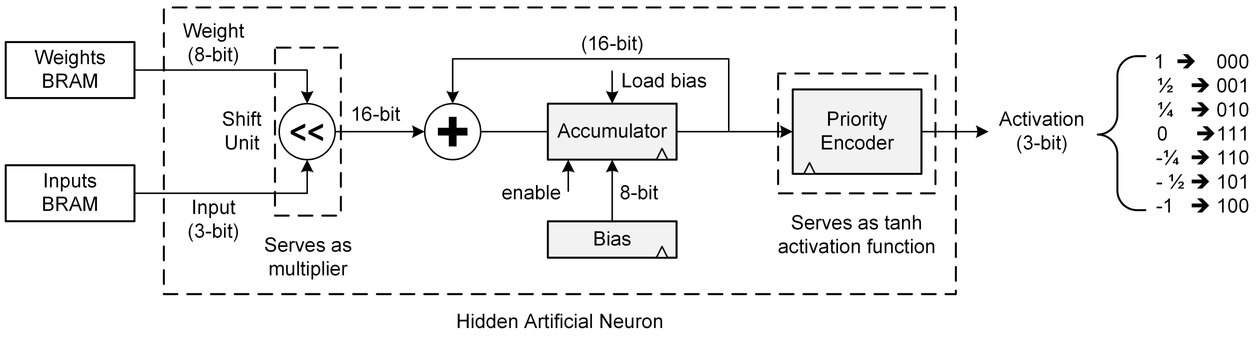


Figure 4.3 Proposed hidden artificial neuron architecture

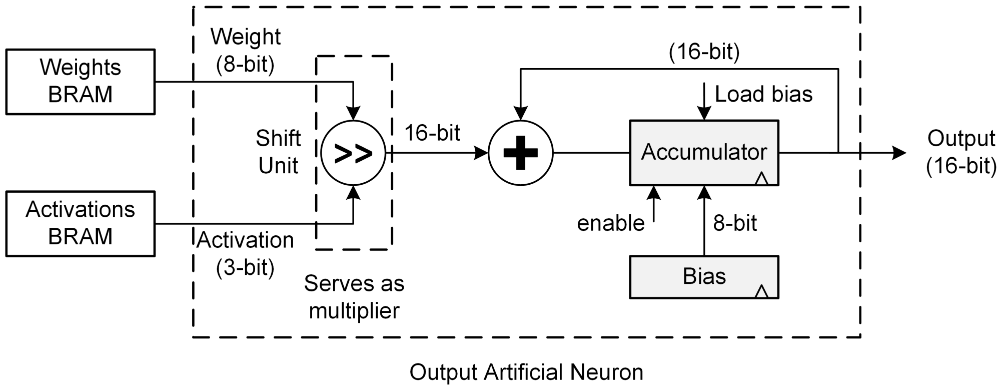


Figure 4.4 Proposed output artificial neuron architecture

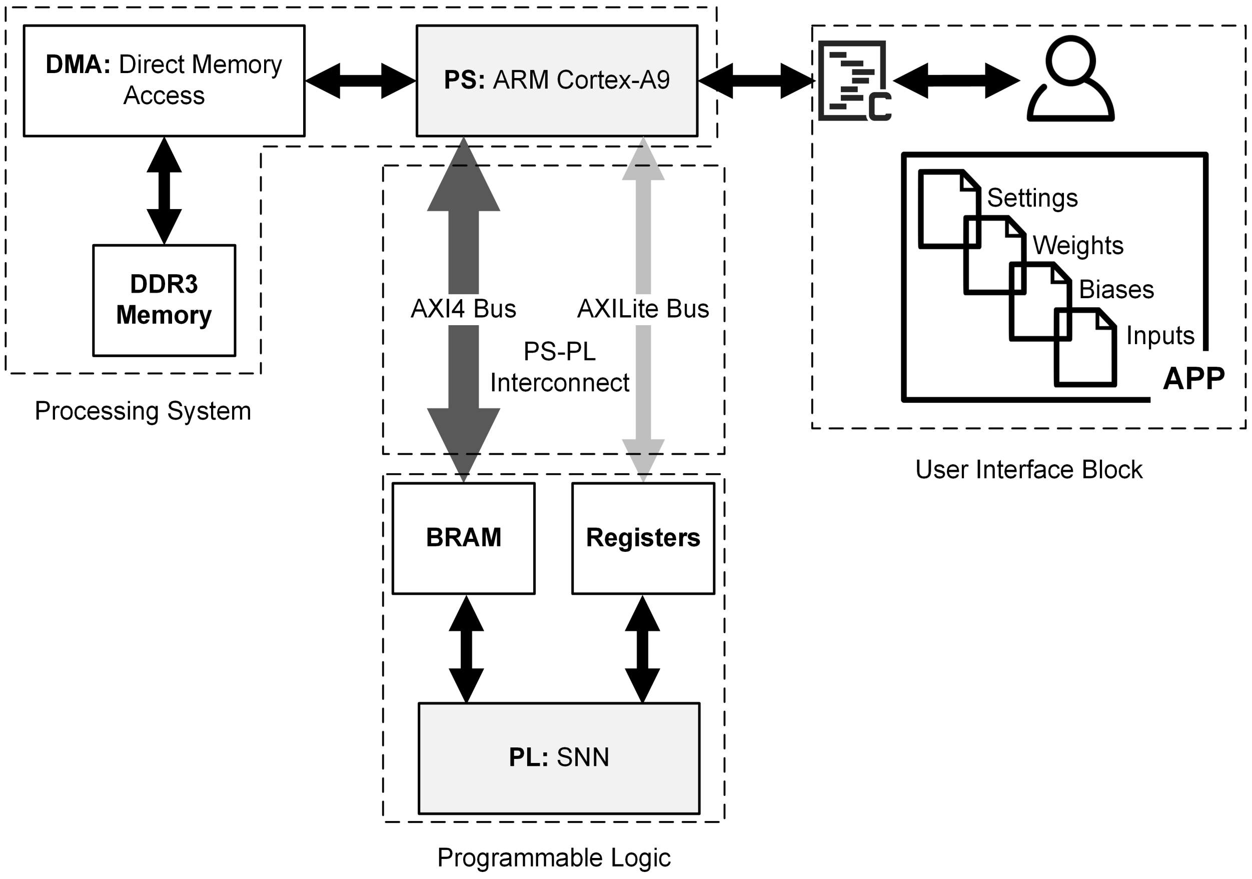


Figure 4.5 The testing platform of the proposed SNN overlay

## 4.5 Experimental Results and Discussion

The proposed SNN architecture was described in Verilog HDL and synthesized for a ZYNQ-7000 ZC706 development kit using Xilinx Vivado 15.2. We tested the SNN overlay architecture using the proposed hidden and output neurons implementations. Figure 4.5 illustrates the organization of the proposed SNN overlay architecture. The implementation includes four different blocks: a) User Interface Block, b) Processing System (PS), c) Programmable Logic (PL) and d) PS-PL Interconnect, detailed in the following subsections.

### 4.5.1 User Interface Block

The user interface block manages the data exchange between the user and the FPGA board. First, the user should feed four different types of data to the FPGA board through a C program that runs on the PS. The first type of data is the student SNN settings that include the number of inputs, number of outputs and number of ANs in the hidden layer. The second and the third type of data are the weights and the biases of the student SNN. In the current system, the stochastic quantization of the inputs happens as a pre-calculation step on the dataset and is not performed on the FPGA board. Computed results are transferred to the PS.

### 4.5.2 Processing System

The PS was implemented in a Zynq-7000 ZC706 that integrates a dual-core ARM Cortex-A9 based PS and a 7-series Xilinx PL. The ARM processor is clocked at 667 MHz and runs a user installed Linux operating system. The main objective of the PS is to ease the process of data exchange between the user and the PL through the PS-PL interconnect.

### 4.5.3 PS-PL Interconnect

The PS and PL communicate with the AXI4 stream [95] and AXILite [95] interfaces. We use the AXI4 stream interface for high-speed streaming data including weights, biases and quantized inputs from the PS to the PL. The AXI4 stream interface uses a Direct Memory Access (DMA) and a set of buffers to transfer the data between the PS and the PL through the DDR3 memory. The AXILite interface provides low-throughput and low-latency communication between the PS and the PL. Therefore, we use the AXILite interface to transfer the network settings and the control signals from the PS to the PL, and vice versa.

### 4.5.4 Programmable Logic

For the purposes of testing, we used the ZYNQ-7000 SOC ZC706 development board. We use the available PL in the FPGA chip to implement the SNN overlay architecture with the maximum possible number of ANs in the hidden layer. Since the maximum number of outputs of the studied applications is 26 as detailed in Table 4.1, we implemented 30 ANs in the output layer. Based on the available resources of the XC7Z045 FPGA device, it could be packed with 2450 ANs in the hidden layer using the proposed SNN. The 2450 ANs in the hidden layer and the 30 ANs in the output layer utilize almost all the device's LUTs.

The implemented quantized SNN on the ZYNQ-7000 ZC706 FPGA board, with 2450 ANs in the hidden layer with 1000 inputs each and 30 ANs in the output layer, works as an overlay for the inference part of different DNN applications. The numbers of ANs in the hidden layer and output layer are limited by the available amount of LUTs on the chip. The benefit of this overlay architecture is that the user specifies the network architecture of their application (no. of inputs, number of ANs in the hidden layer and number of outputs) using a C program that runs on the ARM processor of the processing system side. As long as the numbers do not exceed the maximum value of each parameter, the C program reads the network settings alongside the weights and the biases of the given applications and passes them to the overlay on the PL side. The PL side activates the corresponding number of inputs, ANs in the hidden layer and ANs in the output layer of the implemented quantized SNN. This flow does not require any optimization on the PL side for each application. There is no need to synthesize place, route and regenerate a bitstream for each application, which is advantageous since those processes entail time and effort, and are tricky for many machine-learning researchers. For different applications, our proposed overlay saves up to 40 man-hours that are needed for the hand-coded hardware design. In addition, it saves up to 18 hours that are needed for the synthesize, placement, routing and generating a new bitstream processes.

The proposed overlay can be used to solve many applications including the four studied applications in Table 4.1. Table 4.2 lists the post place-and-route results in terms of FPGA resources usage, power consumption, classification accuracy and throughput of the proposed SNN architecture for MNIST. In this case and to have a fair comparisons, we implement an SNN overlay with 1024 ANs in the hidden layer and 10 ANs in the output layer. With a 300 MHz clock frequency, the proposed SNN processes up to 210 k MNIST images/s with 98.4% accuracy at 3.4 W. The most accurate FPGA implementation of a DNN for MNIST achieves 98.92% and processes 70 k images/s [56]. However, it requires 900 DSPs, 213,593 LUTs and 750 BRAMs. The binary DNN FPGA implementation [65] achieves the highest throughput and avoids using DSPs. It processes 12,361 k images/s while utilizing 91,131 LUTs and 4.5

Table 4.2 Comparison of several NN FPGA implementations on ZC706 for MNIST

| | **LUTs** | **FFs** | **BRAM** | **DSPs** | **Freq. (MHz)** | **Power (W)** | **Acc. (%)** | **Through. (kfps)** |
|---|---|---|---|---|---|---|---|---|
| FPGA'17 [65] | 91,131 | NR* | 4.5 | 0 | 200 | 7.30 | 95.83 | 12,361 |
| ICASSP'16 [56] | 213,593 | 136,677 | 750.5 | 900 | 172 | 4.98 | 98.92 | 70 |
| IJCNN'17 [67] | 82,134 | 131,414 | NR* | 0 | 200 | 3.80 | 97.89 | 195 |
| **Proposed** | 78,679 | 16,864 | 174 | 0 | 300 | 3.40 | 98.40 | 210 |

* Not reported

BRAMs. However, its accuracy shrinks to 95.83%. Table 4.2 shows that the proposed SNN achieves comparable accuracy to recent works [56, 65, 67] while utilizing fewer LUTs, FFs and BRAMs, and no DSPs. The proposed overlay can also be used for other applications as long as the required number of ANs in the hidden and output layers does not exceed the implemented number of ANs in the overlay. Moreover, if the required number of ANs in the hidden and output layers exceeds the number of implemented ANs in the overlay, the overlay could be scaled using different time-multiplexing approaches to reuse the same implemented ANs in both the hidden and output layers.

## 4.6 Conclusion

FPGA-based accelerators are a promising solution for DNN implementation. However, the traditional FPGA accelerator design flow requires time and experience that hinder realizing FPGA-based DNN accelerators. In order to overcome that problem, this paper proposes an FPGA multiplier-less SNN overlay architecture with DNN-level performance. We used a teacher-to-student approach to train the SNN in order to achieve the same DNN performance. The multiplier-less SNN overlay is an energy and area efficient architecture since it avoids multiplications and floating-point operations. The SNN overlay is configurable for many applications. In a couple of minutes, the user can configure the overlay with the network model using a traditional C code. The overlay avoids the unacceptable latency and tedious synthesis, place and route steps for regenerating a new bitstream of a given application. Our test cases show that the multiplier-less SNN overlay achieves the same accuracy as a floating-point DNN with fewer computational resources.

## 4.7 Acknowledgments

The authors would like to thank Safa Berrima, Ahmed Elsheikh, Imad Benacer, Rana Farah, Jeferson Santiago da Silva and Jean-Pierre David for their insightful comments.

# CHAPTER 5 ARTICLE 3: POLYBINN: BINARY INFERENCE ENGINE FOR NEURAL NETWORKS USING DECISION TREES

Authors: Ahmed M. Abdelsalam, Ahmed Elsheikh, Sivakumar Chidambaram, Jean-Pierre David and J.M. Pierre Langlois

Published in: Journal of Signal Processing Systems (JSPS) 2019 [13]

***Abstract*–Convolutional Neural Networks (CNNs) and Deep Neural Networks (DNNs) have gained significant popularity in several classification and regression applications. The massive computation and memory requirements of DNN and CNN architectures pose particular challenges for their FPGA implementation. Moreover, programming FPGAs requires hardware-specific knowledge that many machine-learning researchers do not possess. To make the power and versatility of FPGAs available to a wider deep learning user community and to improve DNN design efficiency, we introduce POLYBiNN, an efficient FPGA-based inference engine for DNNs and CNNs. POLYBiNN is composed of a stack of decision trees, which are binary classifiers in nature, and it utilizes AND-OR gates instead of multipliers and accumulators. POLYBiNN is a memory-free inference engine that drastically cuts hardware costs. We also propose a tool for the automatic generation of a low-level hardware description of the trained POLYBiNN for a given application. We evaluate POLYBiNN and the tool for several datasets that are normally solved using fully connected layers. On the MNIST dataset, when implemented in a ZYNQ-7000 ZC706 FPGA, the system achieves a throughput of up to 100 million image classifications per second with 90 ns latency and 97.26% accuracy. Moreover, POLYBiNN consumes 8× less power than the best previously published implementations, and it does not require any memory access. We also show how POLYBiNN can be used instead of the fully connected layers of a CNN and apply this approach to the CIFAR-10 dataset.**

## 5.1 Introduction

Learning models with a large number of hidden layers have been recently introduced, such as Deep Neural Networks (DNNs) and Convolutional Neural Networks (CNNs) [4]. These models have shown promising performance in many applications including computer vision, speech recognition and regression problems [4]. Although these models achieve high performance, they typically demand a large number of arithmetic and memory access operations in both training and inference [71]. Consequently, existing DNN and CNN applications are typically run on clusters of computers with Graphical Processing Units (GPUs). However, the high power consumption of GPUs and the limited throughput of mainstream processors limit their roles in accelerating DNN and CNN applications that have high throughput and low power consumption requirements [25]. Therefore, hardware accelerators that score high on the 3Ps - Performance, Programmability and Power consumption - are highly desired [8, 79].

Field Programmable Gate Arrays (FPGAs) are hardware logic devices that can be programmed in order to meet the computational and memory requirements of different applications. Low-precision DNN and CNN models [30, 89, 96], such as Binarized Neural Networks (BNNs), reduce the computational complexity and memory requirements significantly [30]. They appear to be well suited for FPGA implementation since most of their computations are bitwise logic operations. Moreover, their memory requirements are highly reduced compared to traditional floating-point DNNs and CNNs. It would thus seem reasonable to expect that FPGAs would outperform GPUs for BNN implementation, while consuming less power [25]. While it is true that ASICs are more energy efficient and can achieve higher performance than FPGAs, FPGAs still have their advantages and market as well. FPGAs are configurable and suit different applications where time to market and design flow matter. Moreover, they are preferred over ASICs for prototyping and validating a design concept.

Although BNNs drastically cut hardware resources consumption down, complex BNN models may need more computational and memory resources than those available in many current FPGAs [25]. Therefore, optimization techniques that efficiently map BNNs to hardware are highly desired. Many machine learning based algorithms have been proposed for classification, including Naive Bayesian [97], K-means [98] and Support Vector Machine (SVM) [99]. Among these algorithms, the Decision Tree (DT) [100] based algorithm has demonstrated the simplest implementation in most experiments. By contrast to NNs, it is easy to implement simple DTs with few splits as Look-Up Tables (LUTs) in FPGAs [101, 102]. Decision trees, however, do not usually generalize as well as DNNs. By contrast to the hidden Artificial Neuron (AN) in a Neural Network (NN), a typical node at the lower levels of a DT is only used to classify a small fraction of the training data. Therefore, DTs tend to overfit the training data unless the training dataset is large enough compared to the depth of the tree.

This paper is an extended version of our previous work [12] in which we presented POLYBiNN, a scalable combinatorial classifier that can be efficiently implemented in FPGAs for FC applications. POLYBiNN is composed of a stack of decision trees, which are binary classifiers by nature, and it utilizes AND-OR gates instead of multipliers and accumulators. In this work, we show how to train POLYBiNN in order to perform millions of classifications per second while achieving the same accuracy as recent BNNs, making it ideal for supporting real-time embedded applications. We show POLYBiNN's capabilities on five datasets including MNIST and CIFAR-10. The achieved classification throughput surpass the best previously published results by over 67× for MNIST while achieving 97.26% classification accuracy.

In our previous work [12] we introduced:

- POLYBiNN, an efficient classifier based on logic functions without any arithmetic operations.
- A tool that generates automatically a low-level hardware description of the trained POLYBiNN.
- A method to integrate POLYBiNN into existing DNNs and CNNs by replacing their fully connected layers.
- FPGA implementation of POLYBiNN using the proposed tool leading to remarkable improvements over existing FPGA-based BNN accelerators.

The new contributions of this paper are as follows:

- We propose a heuristic method to train POLYBiNN based on Boosting, which creates complex classifiers by combining many simple DTs.
- We propose an optimized voting circuit and its implementation to solve multinomial classification problems using binary DT classifiers.
- We propose a range of POLYBiNN prototypes that demonstrate the potential of using POLYBiNN for different applications.
- We explore different techniques for optimizing POLYBiNN in terms of classification accuracy and resource utilization.

The rest of the paper is organized as follows. Section 5.2 provides background on BNNs and DTs and their hardware implementations. A detailed view of POLYBiNN, its training and inference are described in Section 5.3. The POLYBiNN hardware implementation is detailed in Section 5.4. Section 5.5 is dedicated to the experimental results, comparison with other architectures and discussion. Section 5.6 concludes the paper.

## 5.2 Background

### 5.2.1 Binary Neural Networks

Realizing a precise implementation of DNNs and CNNs normally entails a large number of additions, multiplications and memory accesses [25]. One approach to simplify the computational and memory complexity of DNNs and CNNs is to use a compact data representation for inputs and parameters [103]. An extreme case is to use single-bit values; arithmetic operations are then replaced by logic operations. Single-bit values can be used in three places: binary inputs, binary weights and binary activations.

Courbariaux et al. [104] proposed Binary Weights Neural Networks (BWNs) where the weights are constrained to the set {-1, 1} and represented with a single bit, and multiplications are replaced by additions. Courbariaux et al. [30] extended their work and proposed a BNN where both weights and activations are binary values {-1, 1}. They proposed to quantize the activations to replace the multiply-accumulate operations with logic operations, i.e. XNORs and counters. The authors reported competitive accuracies for the MNIST, SVHN and CIFAR-10 datasets. Fig. 5.1a) shows a real-valued AN which computes the weighted sum of its inputs then applies a non-linear activation function to this sum. Fig. 5.1b) shows a binary AN that uses XNOR gates instead of multipliers, a counter instead of a multi-operand adder, and a step activation function (a comparator) instead of a non-linear activation function. Following that, a binary AN can be implemented using a single LUTs with a sufficient number of inputs and a single output. In other words, using boolean expression, a combinatorial circuit with AND and OR gates as shown in Fig. 5.1c) can serve as a binary AN.

### 5.2.2 Binary Neural Networks in Hardware

Many prior works have proposed mappings BNNs on FPGAs and Application-Specific Integrated Circuits (ASICs). Umuroglu et al. [65] proposed FINN, a framework for fast and scalable BNN inference. The authors implemented a full BNN inference engine with Fully Connected (FC), convolution and pooling layers. On the Xilinx ZYNQ-ZC706 FPGA, the proposed engine performs 12.3 million MNIST image classifications per second with 95.8% accuracy. Nakahara et al. [105] proposed to eliminate all but the last Fully Connected (FC) layers of a binary CNN, replacing the eliminated FC layers by average pooling operations. On a Xilinx ZYNQ Z020 FPGA, the authors saved 66% of the LUTs required for FINN [65] while achieving the same accuracy and throughput. Zhao et al. [106] leveraged HLS design tools for BNNs with ternary weights {-1, 0, 1} on FPGAs. However, for a Xilinx ZYNQ Z020 FPGA,

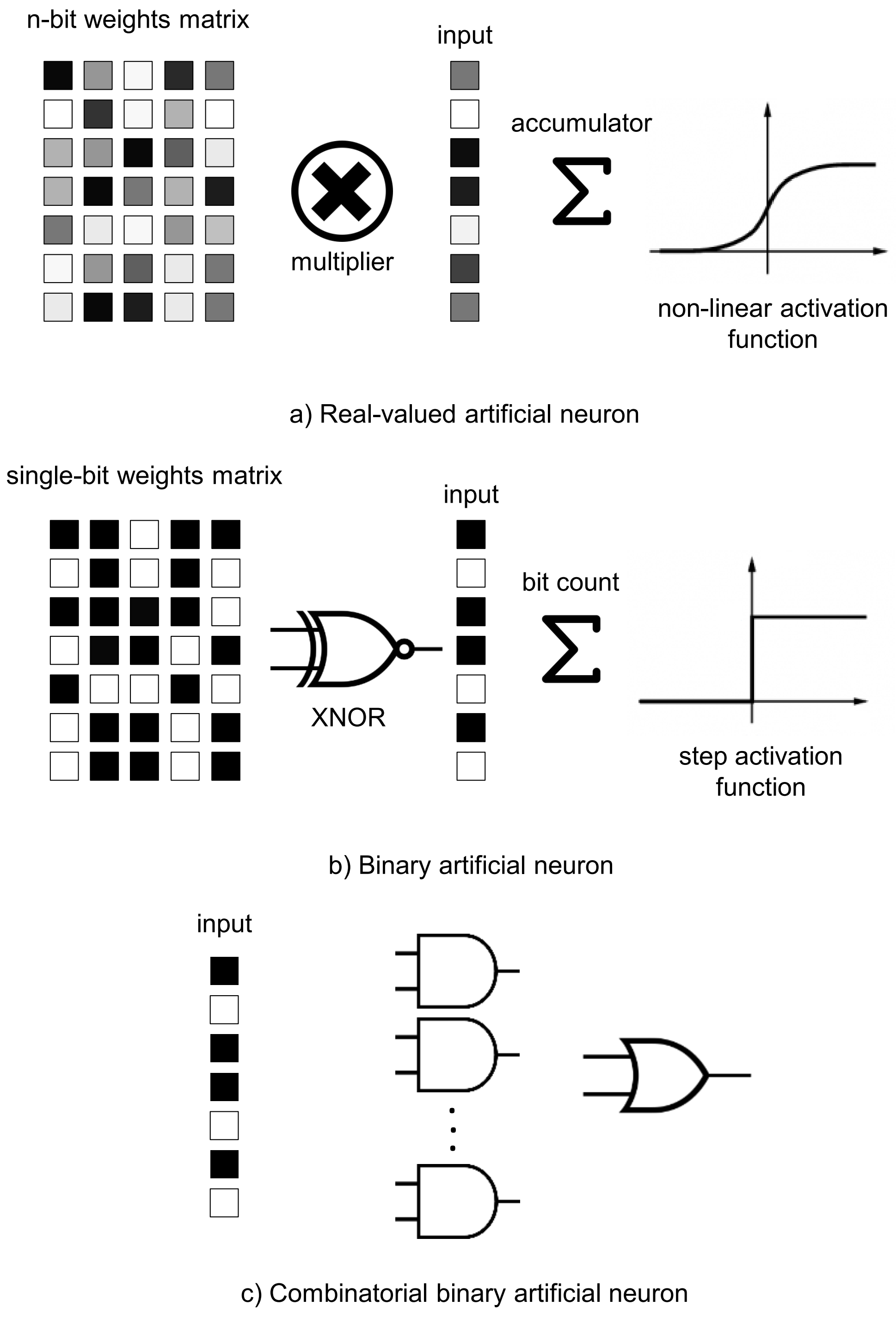

n-bit weights matrix
input
accumulator
multiplier
non-linear activation function
a) Real-valued artificial neuron
single-bit weights matrix
input
bit count
XNOR
step activation function
b) Binary artificial neuron
input
c) Combinatorial binary artificial neuron

Figure 5.1 Real-valued artificial neuron vs binary artificial neuron

the ternary weights take too much space and must be stored in a separate DDR memory, resulting in degraded throughput and power efficiency. Alemdar et al. [67] proposed another way to maintain the same accuracy as floating-point DNNs with ternary weights NNs. They train the ternary weights NNs using a teacher-to-student approach, transferring the knowledge of trained floating-point NNs to ternary NNs. The authors report 3.1× higher energy efficiency with respect to floating-point NNs while maintaining the same accuracy.

A key optimization of BNN models in hardware is to improve resource utilization of FC layers [107]. Pruning [28] is a known method to do so where some less important connections are trimmed. It reduces the computational complexity of the model and improves resource utilization. Even using such techniques, several FC layers are necessary and require large amounts of computational and memory resources that influence the throughput and power consumption of the implementation. Neural networks were first introduced as continuous values multiclass classifiers, and then BNNs were introduced as a binarized version of NNs. However, such a conversion from continuous to binary values causes information loss. In addition, BNNs are not fully optimized in terms of resource utilization since NNs are not binary classifiers by nature.

### 5.2.3 Decision Trees

Decision Trees are one of the simplest but effective binary classification algorithms to have been widely used is several disciplines [108]. The concept of DTs is to divide the training dataset into sub-divisions to ease the classification process of the dataset samples. The main components of DT models are nodes and branches. There are three different types of nodes in DTs. The first type of DT node is the root node, which represents the best predictor of the training dataset that further gets divided into two or more sets. The second type of DT node is the decision or split node since it splits a subset into two or more subsets. Leaf nodes are the third type of DT node, where the subsets are not split further. The leaf nodes represent the final result of a combination of decision and root nodes.

Binary DT models include a single binary target variable and a number of binary input variables. Input variables are used to split the root node and decision nodes down to the leaf nodes. The input variable that best discriminates the target samples is chosen as the root node, and then splits continue at the subsequent decision nodes into two or more subsets. The splitting procedure carries on until a pre-determined DT architecture or a stopping criteria are met. The more complex a DT is, the less reliable it is to classify new samples [109]. Although using a single DT to classify a complex multiclass problem is unlikely to achieve

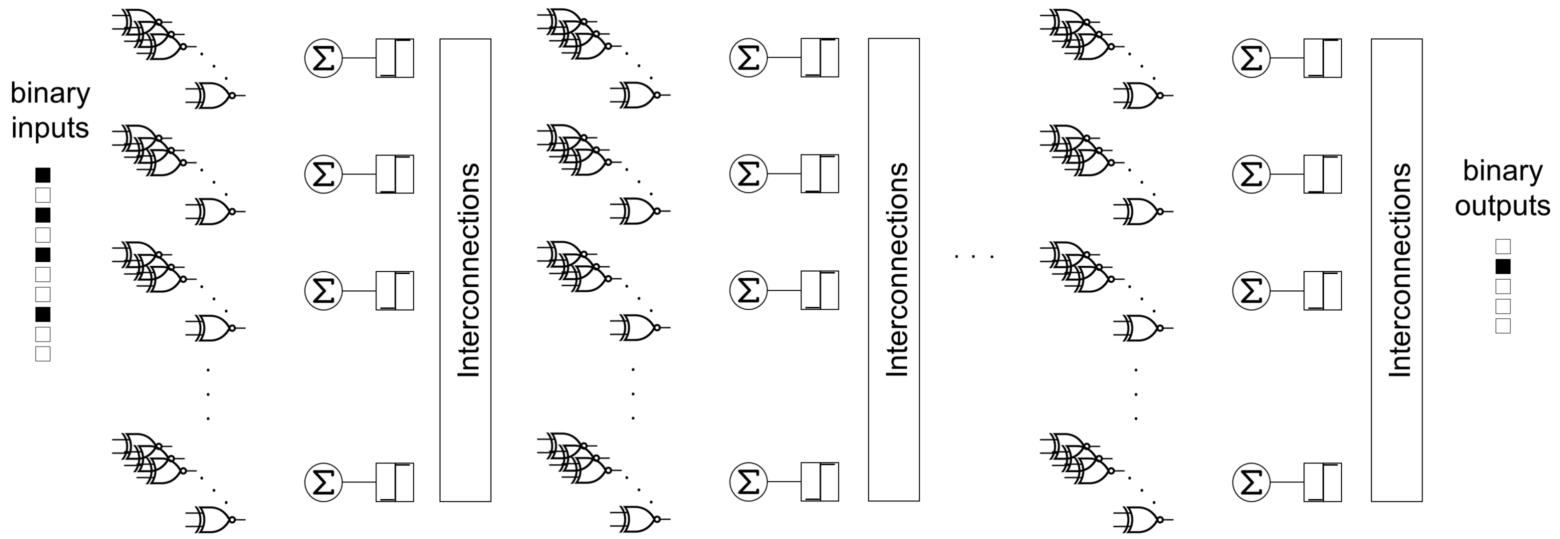


Figure 5.2 Traditional BNN architecture with FC layers

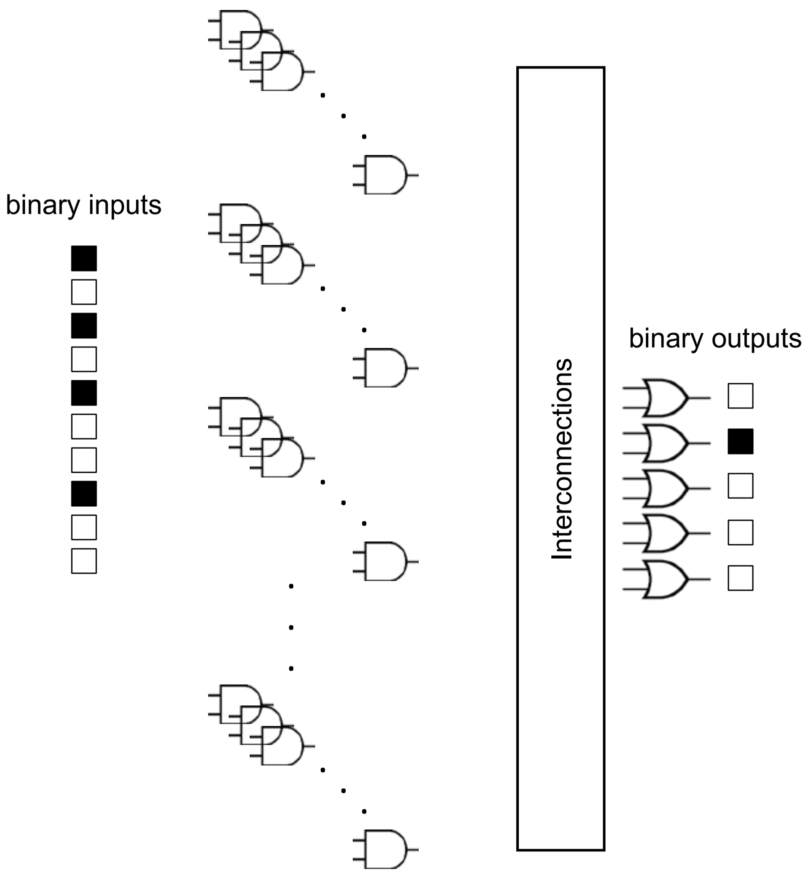


Figure 5.3 BNN architecture using a single FC layer with AND-OR gates

acceptable classification accuracy, DTs are well suited to FPGAs since they can easily fit their LUTs.

In this paper, we focus on FC applications. The motivation of this paper is based on two observations. The first observation stems from the fact that BNNs are classifiers that compute the corresponding class of their binary inputs. They can have several FC layers composed of XNOR gates followed by counters and step activation functions as shown in Fig. 5.2. They can thus be viewed as large binary Sum-of-Products (SOPs) functions [101] as shown in Fig. 5.3, and a binary AN can be implemented as shown in Fig. 5.1c). Those SOPs can then be mapped to an FPGA's LUTs [101] [102]. However, the number of LUT inputs is small compared to the complexity of the BNNs' logic functions [25]. The second observation is that binary classifiers, such as DTs, can solve multiclass problems by dividing them into several simple binary classification problems and solving the sub-problems independently [110]. However, binary classifiers do not usually solve multiclass problems as efficiently as NNs and BNNs do [109]. Boosting [109] can be used to build a series of binary classifiers, with the later classifiers rectifying the mistakes of the earlier classifiers, allowing the binary classifiers to solve complex multiclass problems. This motivates us to propose a method to train a single binary classification layer based on DTs to achieve high performance with a significantly reduced amount of resources, and to propose a corresponding architecture.

## 5.3 The POLYBiNN Architecture

In this section, we detail the POLYBiNN architecture introduced in [12] and show how we train it using simple binary classifiers. We then show how it can be efficiently implemented in an FPGA.

### 5.3.1 Architecture Overview

Fig. 5.4 shows an overview of the POLYBiNN architecture. It is composed of a $M \times N$ array of DTs as shown in Fig 5.5, where $M$ is the number of classes and $N$ is the number of trees per class, followed by a voting circuit. The array of DTs are implemented as SOPs. Each AND gate represents the path from the root node to an active leaf node (leaf node that has an output of 1), and the DT output is high when one leaf is active. An example is given in Fig. 5.6 for a case with seven features and three active leaves. The AND gate inputs can be the natural or complemented version of features. While each class has $N$ DTs, each DT can be made up of a different number of SOPs, depending on its classification complexity.

A pre-processing step is necessary to binarize the inputs to POLYBiNN. In the case of applications that can be solved using FC layers, each input is directly binarized using an arbitrary threshold. For application that require CNNs, e.g. CIFAR-10, a number of convolutional layers extract features from the input images, and they are then binarized using an arbitrary threshold.

Decision trees are deceptively simple, but implementing large DTs with dozens of splits (decision nodes) in an efficient way is a complex problem [111]. As the number of splits increases, the number of included features for each branch increases, which can lead to overfitting of the training dataset. A single large DT can lead to poor classification on data with complex relationships since it begins with the same root for all splits [112]. Moreover, DTs with many splits may result in SOPs with a large number of inputs. The maximum number of splits must thus be selected carefully according to the implementation technology, e.g. 6-input LUTs in some FPGAs. Therefore, using multiple but smaller DTs is preferred for complex problems since they can have different roots and result in reasonable-size SOPs.

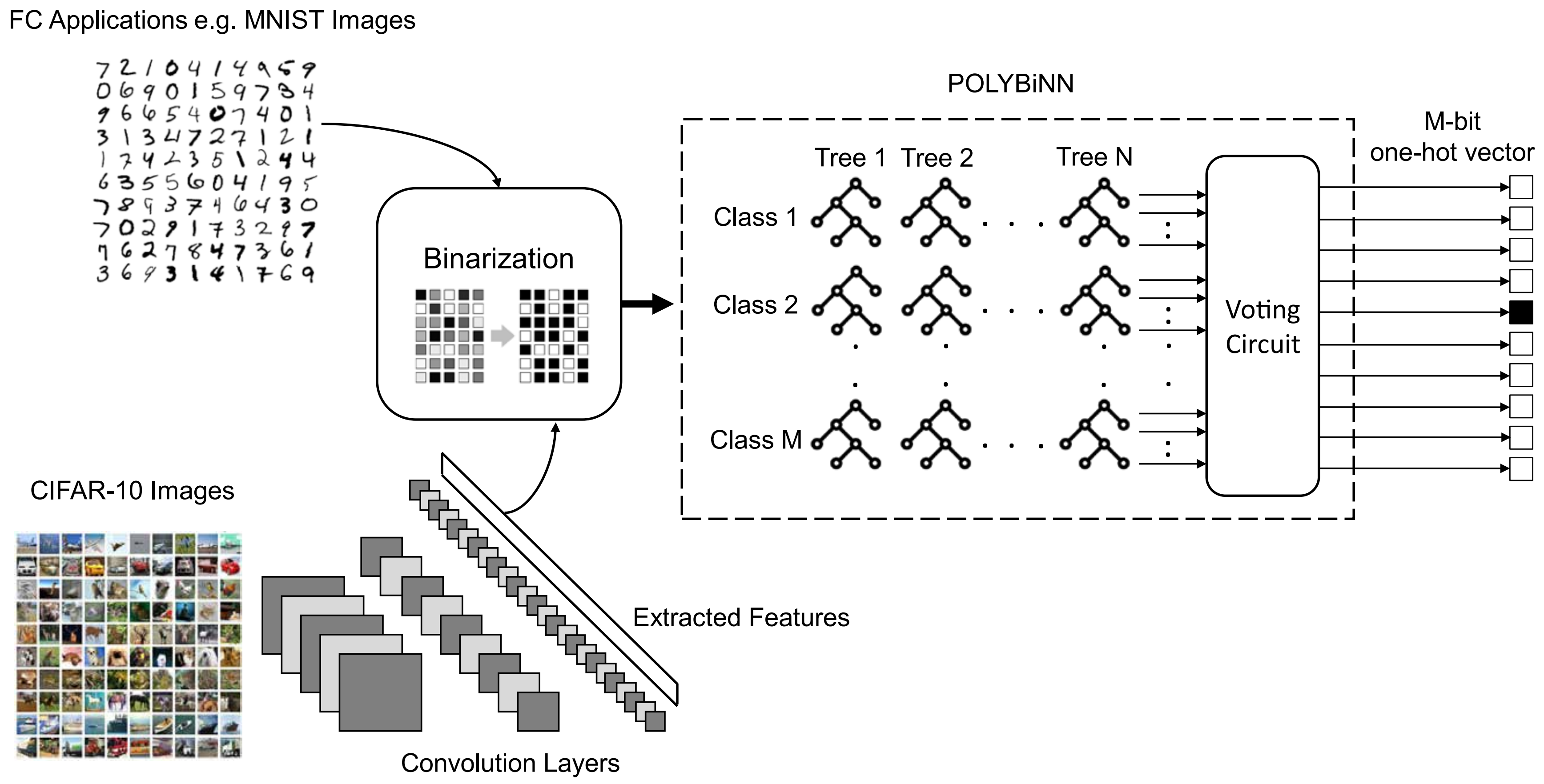


Figure 5.4 Overview of the POLYBiNN architecture and its use to classify FC applications and CIFAR-10 datasets

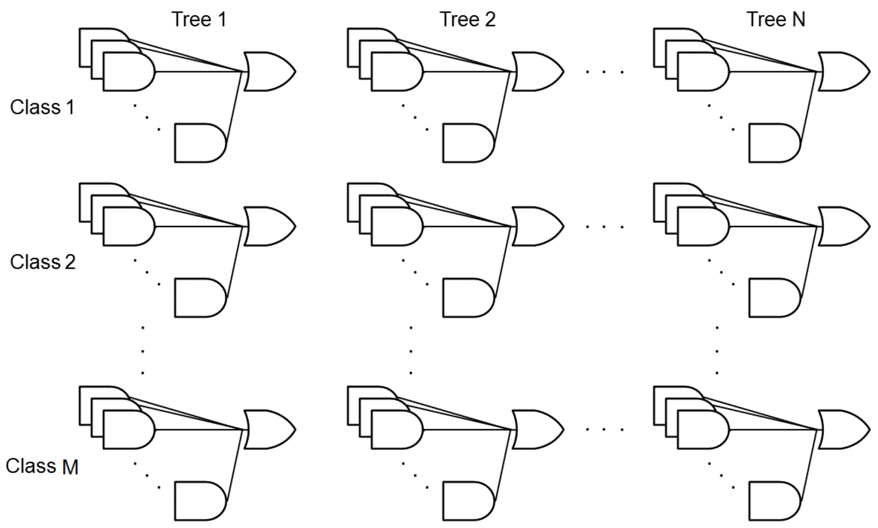
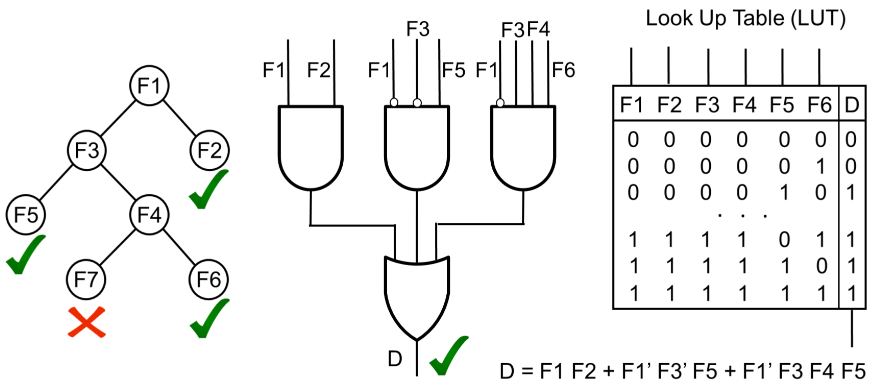


Figure 5.5 POLYBiNN DT implementation in AND and OR gates



| F1 | F2 | F3 | F4 | F5 | F6 | D |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| . | . | . | | | | |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Figure 5.6 Decision tree implementation as a SOP and a LUT

### 5.3.2 POLYBiNN Training Algorithm

We train POLYBiNN using AdaBoost [109], an ensemble-learning algorithm that creates complex classifiers by combining many weak classifiers. Each class is trained separately using $N$ binary classifiers. We choose DTs as weak binary classifiers because they can be directly represented in Boolean logic without requiring any mapping or simplification like other machine learning techniques [112].

For each class, the DTs are trained iteratively by splitting the observation space using a top-down greedy search algorithm. The feature chosen to make the split is the one that best discriminates between the training classes. Splitting starts with the first feature in the root node of the tree, then it is repeated on each partition of the divided data, as shown in Fig. 5.6. The first node is known as the root node of the tree, which is the first feature used for splitting. Then the splitting is repeated on each partition of the divided data, as shown in Fig. 5.4. Searching for the best discriminating feature among all possible features is impractical and time consuming for large datasets. Therefore, the search for the best discriminating feature takes place only among a set of possible candidate features where the class labels of observations change from one class to another. This is due to the fact that there is no need to search for splits between observation members of the same class. The selection for the best discriminating feature is then done according to a quality measure of each of the candidate features and the best is selected. The measure of discrimination used for choosing the features is the information gain, which is the decrease in entropy when a split is made. For a partition of the data $D_s$, its entropy $H(D_s)$ for binary classification can be formulated as shown in Eq. (5.1):

$$H(D_s) = p_+ \log_2 \frac{1}{p_+} + p_- \log_2 \frac{1}{p_-} \tag{5.1}$$

where $p+$ is the portion of positive examples in the considered split in the partition $D_s$, and $p^{\smile}$ is the portion of negative examples. The number of splits either stops after the performance on a validation set starts to deteriorate, or by choosing the maximum number of splits to suit the target hardware platform. We stop splitting the decision trees after a pre-defined number of splits. It is a hyper-parameter of our training algorithm which affects the accuracy and computational resources as detailed section 5.5. Another hyper-parameter is the number of decision trees for each binary class, which is also studied in section 5.5.

Although decision trees are relatively simple to understand and implement compared to NNs, large decision trees with dozens of decision nodes and splits are not accurate and difficult to implement [109]. As the number of splits increases, the included features for each branch increase, which leads to conditions that are more specific of the training dataset rather than a good generalization of the actual problem. This might cause overfitting and poor accuracy of the classifier. Moreover, implementing such decision trees with deeper branches requires LUTs with a large number of inputs which may not perfectly fit the LUTs available on the existing FPGAs. In addition, a single large DT still achieves poor resolution on data with complex relationships since it starts with the same root for all decision nodes [112]. A multiple DTs approach is preferred for complex applications since the trees start with different roots and are easy to implement. Therefore, we use the Adaptive Boosting (AdaBoost) algorithm to combine the decision trees classifiers together to improve the performance of the overall classifier.

AdaBoost iteratively trains the DTs with an adaptive sample-weighting scheme [113]. The weights of samples are computed as a function of the error made by the previous DT and emphasize wrongly classified samples. Given a binary classification problem with a training set $\{(x_1, y_1), (x_2, y_2), ...(x_m, y_m)\}$, where $m$ is the number of samples, $x$ is the observation vector and $y$ is the class label. Each class label takes a value from the set {–1, +1}, where –1 indicates that the observation does not belong to the target class and +1 indicates that the observation belongs to the target class. Let $D_n(x_i)$ be the weight of sample $x_i$ at DT $n$, and $\alpha_n$ be the $n^{\text{th}}$ decision tree's confidence vote. The first decision tree is trained with equal sample weights where all the samples have the same weight value $1/m$. For all the subsequent decision trees, the sample-weighting scheme depends on the confidence of the previously trained decision tree ($\alpha_n$). The confidence of the $n^{th}$ decision tree is calculated as shown in Eq. (5.2):

$$\alpha_n = \lambda \times \frac{1}{2} \ln \left(\frac{1-\epsilon_n}{\epsilon_n}\right) \tag{5.2}$$

where $\lambda$ is the learning rate of training DTs, which decreases the impact of a training iteration on the whole model, so it slows down the change of the weights of the samples. In other words, the learning rate helps to compensate the fact that the selected features to split might not be the best for future splits. $\epsilon_n$ is the ratio of the misclassified samples by the trained decision tree and is given by Eq. (5.3):

$$\epsilon_n = \frac{\sum_{i=1}^{m} D_n(x_i) I_{h_n(x_i) \neq y_i}}{\sum_{i=1}^{m} D_n(x_i)} \tag{5.3}$$

where $h_n(x)$ is the decision of the $n^{\text{th}}$ DT, and $I_{h_{n(x_i)} \neq y_i}$ is an indicator function which outputs 1 when the condition is satisfied and 0 otherwise.

The training sample weights for the next decision tree are then updated as follows:

$$\begin{aligned} D_{n+1}(x_i) &= \frac{D_n(x_i)\exp(-\alpha_n y_i h_n(x_i))}{Z_n} \\ &= \frac{D_n(x_i)}{Z_n} \times \begin{cases} e^{-\alpha_n} & \text{when } \mathrm{h}_n(x_i) = y_i \\ e^{\alpha_n} & otherwise \end{cases} \end{aligned} \tag{5.4}$$

where $Z_n$ is a normalization factor chosen so that $D_{n+1}$ will sum to 1. When the prediction of decision tree $n$ is correct ($h_n(x_i) = y_n$) for sample $x_i$, the weight of the same sample in the following decision tree $n+1$ is decreased, and vice versa, when the prediction of decision tree $n$ is incorrect $h_n(x_i) = y_n$ for sample $x_i$, the weight of the same sample $D_{n+1}(x_i)$ in the following decision tree $n+1$ is increased.

The computed sample weights change the effective portions of the positive and negative samples ($p_+$ and $p_-$) which are used in Eq. (5.1). This affects the entropy values while training the following decision tree. In turn, this affects the decision of choosing the best discriminating splits within a feature among the candidate splits. Once the $N$ decision trees are properly trained, the final POLYBiNN classification decision $K(x)$ is a confidence vote between the trained binary decision trees as follows:

$$K(x) = sign(\sum_{n=1}^{N} \alpha_n h_n(x)) \tag{5.5}$$

where the sign functions gives 1 when its input is greater than zero, and -1 otherwise, which indicate the class of the given sample.

The procedure of the POLYBiNN training algorithm is summarized in Algorithm 1. We use a one-vs.-all training strategy that involves training each class separately with the samples of that class as positive samples and all other samples as negatives. Following that strategy of training each class separately, we might end up with ties among classes. Therefore, we apply a voting algorithm based on the confidence of each strong binary classifier in order to predict the output class for a given input. When two or more strong binary classifiers are active for the same input, the voting circuit selects the class with the highest confidence. When all strong binary classifiers are inactive for the same input, the voting circuit chooses the class with the lowest confidence. Otherwise, the voting circuit selects the one active classifier.

Once all $N$ trees of class $M$ have been trained, their outputs must be combined to come to a decision for that class. The output $d_{nm}$ of tree $n$ in class $m$ can be 0 or 1. AdaBoost also

**Algorithm 1:** POLYBiNN Training Algorithm

```
1  Initialize Sample Weights: $D_1(x_i) = \frac{1}{m}$
2  $n \leftarrow 1$;
3  while $n < N$ do
4      Train a DT that using a sample weight D$_n(x_i)$
5          Computes DT confidence $\alpha_n$
6          Computes the mis-classification ratio $\epsilon_n$
7      Update sample weights D$_{n+1}(x_i)$
8      $n \leftarrow n + 1$;
9  end
10 Output: A confidence vote decision of POLYBiNN $K(x)$ using $\alpha_n' s$
```

assigns a confidence $c_{nm}$ to the output of each tree, a floating point value in the range [0, 1[. For each class $m$, the decision $D_m$ and confidence $C_m$ are computed as follows:

$$D_m = \begin{cases} 1 & \text{when} \sum_{n=1}^{N} d_{nm}.\ c_{nm} > \sum_{n=1}^{N} c_{nm}/2 \\ 0 & \text{otherwise.} \end{cases} \tag{5.6}$$

$$C_m = \frac{\sum_{n=1}^{N} d_{nm}.\ c_{nm}}{\sum_{n=1}^{N} c_{nm}} \tag{5.7}$$

The final step is to decide to which class each input belongs. Each of the $M$ classifiers produces a decision $D_m$ and a confidence $C_m$. For each input, if only one classifier is active, its class is selected. If more than one classifier are active, we select the class with the highest confidence. If all classifiers are inactive for the same input, the class for which the confidence is lowest is selected.

## 5.4 POLYBiNN High-Level Architecture on FPGAs

The POLYBiNN architecture lends itself well to an FPGA implementation. As shown in Fig. 5.4, POLYBiNN consists of an array of DTs followed by a voting circuit, whose details are shown in Fig. 5.7. Each DT corresponds to a SOP and can be implemented in a single LUT if the number of inputs is not larger than six, a constraint that can be set during training. Equations (5.6) and (5.7) are used to compute the final decision for each input, but their output can be obtained without any arithmetic computation. The decisions $d_{nm}$ are binary values and the confidences $c_{nm}$ are constants once training is completed. In Fig. 5.7, the overall decisions $D_m$ and confidences $C_m$ are computed by a separate block for each class, without any arithmetic operations. Then, the confidences are quantized into 2-bit values, resulting in three output bits for each class: one for the decision and two for the confidence. A set of pipelined comparators is used in the Argmax block to select one of the output classes, according to the rules given in the previous subsection. The complexity of the voting circuit depends on the number of DTs per class.

### 5.4.1 Simplified Voting Circuit

In several classification applications, the training dataset is not balanced: the output classes have a different number of samples. We exploit this fact to simplify the computational complexity of the voting circuit. We propose a simplified version of the voting circuit that does not require to compute, quantize and compare the confidences of each DT per class.

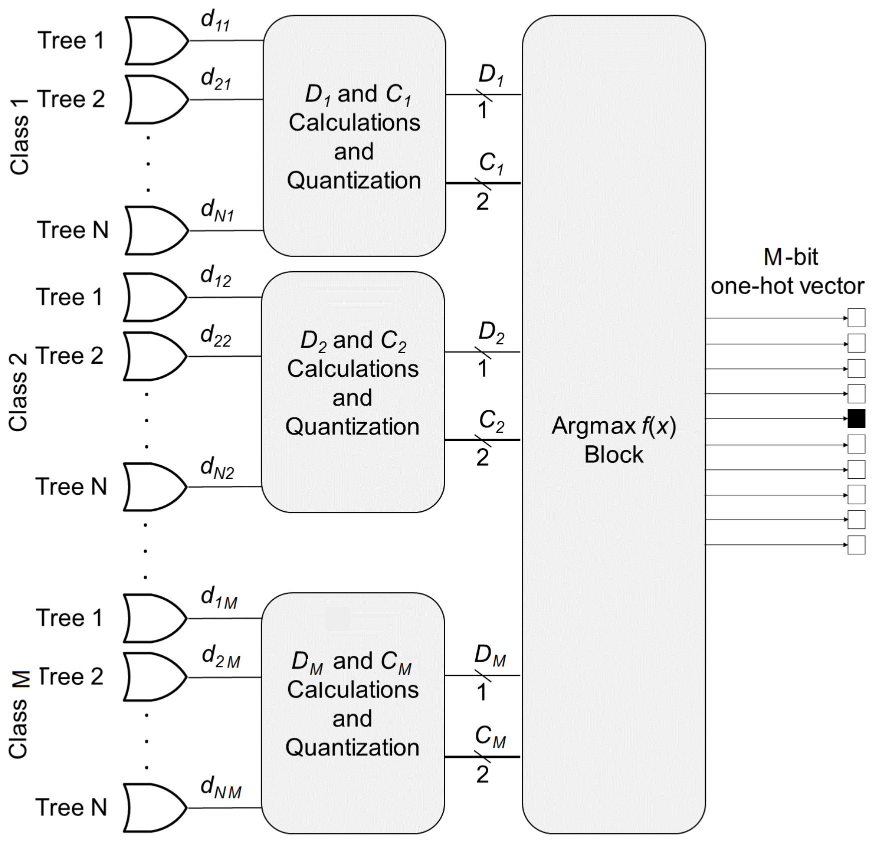


Figure 5.7 Voting circuit implementation of POLYBiNN

The idea is that if more than one classifier is active for the same input sample, the class that has the greater number of samples is selected. This idea is effective in terms of achieving the same classification accuracy and reducing the computational complexity of the voting circuit. Therefore, the optimized voting circuit is implemented as a priority encoder. It assigns priorities to the output classes based on the number of samples of each. The effect of that simplification in terms of classification accuracy and LUT usage is discussed in Section 5.5. It should be noted that if the dataset is balanced with an equal number of samples for each class, the original voting circuit should be used.

### 5.4.2 Automated HDL generation of POLYBiNN

The POLYBiNN architecture is regular and lends itself well to a parameterized description. Therefore, one of the contributions of this work is a tool that automatically generates a synthesizable HDL description of POLYBiNN given a set of parameters. The steps followed by the tool are given in Fig. 5.8. In the first step, the tool generates the HDL for the necessary headers, definitions for input and output ports and signal definitions. The HDL

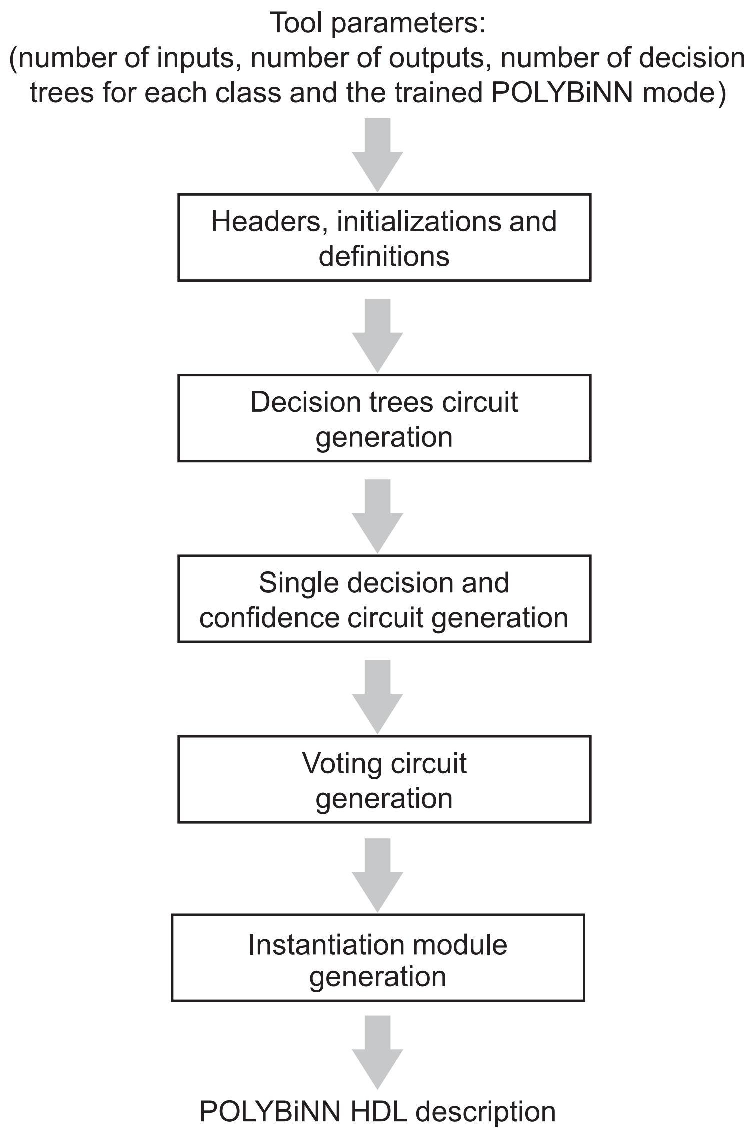

Tool parameters:
(number of inputs, number of outputs, number of decision trees for each class and the trained POLYBiNN mode)
Headers, initializations and definitions
Decision trees circuit generation
Single decision and confidence circuit generation
Voting circuit generation
Instantiation module generation
POLYBiNN HDL description

Figure 5.8 POLYBiNN HDL generation steps

descriptions of the SOP circuit module for each class are then generated according to the parameters and the trained POLYBiNN model. Then, the outputs of those modules are connected to the inputs of the voting circuit module that is generated in the voting circuit generation step. The tool can be configured to implement the original or the optimized voting circuit according to users preferences. The last step is the generation of the instantiation module that incorporates the different generated modules. File generation time is negligible compared to the training time.

## 5.5 Experimental Results

This section presents the classification accuracy and hardware implementation cost of POLYBiNN in FPGAs.

### 5.5.1 POLYBiNN Classification Performance

We trained POLYBiNN in MATLAB using DTs and AdaBoost, uisng a 0.7 learning rate. The input data was normalized to [0, 1] and binarized using a fixed threshold of 0.5. We implemented POLYBiNN to accelerate BNN inference on the following datasets:

**MNIST** [88]: MNIST is a well-studied machine learning dataset of handwritten digits images. The objective is to classify the input image into one of the ten digits $\{0 \ldots 9\}$. The MNIST dataset consists of 60,000 gray-scale images each of $28 \times 28$ pixels.

**Speech Recognition** [92]: This speech recognition dataset (ISOLET) contains vocal signals of the 26 letters of the English alphabet $\{A \ldots Z\}$. The goal is to classify each vocal signal into one of the 26 English letters. ISOLET has a dataset of 7,797 examples each with 617 features collected from 150 speakers.

**Indoor Localization** [93]: The objective is to determine the location (building 0, 1, or 2 and floor 0, 1, 2, 3, or 4 (only for building 2)) of each input signal. A set of 520 Received Signal Strength Indicators (RSSI) for each input signal of the 21,048 examples is given.

**Human Activity Recognition** [94]: The goal is to recognize the human activity {WALKING, WALKING_UPSTAIRS, WALKING_DOWNSTAIRS, SITTING, STANDING and LAYING} based on motion sensor signals. The human activity recognition dataset has 10,299 examples each with 561 features.

**CIFAR-10** [114]: CIFAR-10 is a well-known machine learning and computer vision dataset that has 60,000 $32 \times 32$ color images in 10 categories.

We considered two different POLYBiNN topologies to classify the datasets.

- FC is a trained POLYBiNN for classification of the first four datasets that can be solved using only FC layers.

- CNV is made up of a CNN in which a trained POLYBiNN replaces the FC layers. A succession of ($3 \times 3$ convolution, $3 \times 3$ convolution, $2 \times 2$ maxpool) layers repeated three times with 64-128-256 channels is used to extract the features of a CIFAR-10 image. POLYBiNN then accepts the binarized extracted features computed by the convolution layers and outputs a 10-bit one-hot vector indicating the corresponding class of a given input. The first convolution layer accepts 32×32 images with 24 bits/pixel.

Table 5.1 compares the accuracy of POLYBiNN for the studied FC application and CIFAR-10 with prior works. In all cases we obtain near state-of-the-art results in terms of accuracy. POLYBiNN achieves 97.26% accuracy with the FC architecture on MNIST. For the other FC applications, Table 5.1 shows that POLYBiNN can achieve acceptable accuracy. On the ISOLET, Indoor Localization and Human Activity Recognition datasets, we compare POLYBiNN results to the floating-point baseline results. Although POLYBiNN uses binary weights compared to the floating-point weights used to achieve the baseline results, the accuracy diminishes by just 5% in the worst case. On the CIFAR-10 dataset, POLYBiNN achieves 81.3% accuracy with the CNV architecture. Fig. 5.9 shows the accuracy for MNIST and CIFAR-10 as a function of the number of DTs and the number of splits of each DT. Fig. 5.9 shows that the accuracy of POLYBiNN for these datatsets is more sensitive to the number of DTs than to the number of splits when the number of DTs is small.

### 5.5.2 POLYBiNN Implementation Results

POLYBiNN models were synthesized for a ZYNQ-7000 ZC706 development kit using Xilinx Vivado 15.2. The development kit contains a ZYNQ Z7045 SoC with dual ARM Cortex-A9 and FPGA fabric with 218,600 LUTs. For all tests the data was stored in BRAM and these resources are not included in the POLYBiNN cost. The resource utilization and power consumption are reported in Xilinx Vivado Design Suit after implementation. Table 5.1 presents performance and cost results.

For MNIST, Table 5.1 shows that POLYBiNN with 20 DTs and 100 splits achieves 95.97% accuracy using 9,943 LUTs. Constraining the clock frequency to 100 MHz, the implementation can process 100 M frames per second, which is 8.3× the throughput of FINN [65] that uses binary weights and activations, while consuming 0.286 W, which is 25× less than FINN [65] for virtually the same accuracy. The power consumption of POLYBiNN is significantly lower than the power consumption of the previous works because POLYBiNN is implemented using combinational circuits and requires few registers and no counters, adders or multipliers.

POLYBiNN can still achieve better accuracy using more DTs and splits. As shown in Table 5.1, POLYBiNN achieves 97.26% accuracy using 109,653 LUTs. Although POLYBiNN then utilizes 20% more computational resources than FINN [65], its throughput is up to 67× greater while consuming 8× less power, for slightly less accuracy. POLYBiNN achieves a latency of 90 ns compared to 2.5 $\mu$s needed by FINN [65]. Moreover, POLYBiNN does not

Table 5.1 Performance comparison with existing FPGA-based DNN and CNN accelerator designs

| **MNIST (FC)** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Architecture** | **Platform** | **Through. (FPS)** | **LUTs** | **Lat. (ns)** | **Freq. (MHz)** | **BRAM 36 Kb** | **Power (W)** | **Acc. (%)** |
| **FINN [65] 3 Layers × 256 AN** | ZC706 | 12,361 k | 91,131 | 310 | 200 | 4.5 | 7.3 | 95.83 |
| **FINN [65] 3 Layers × 1024 AN** | ZC706 | 1,561 k | 82,989 | 2,440 | 200 | 396 | 8.8 | 98.40 |
| **Proposed$^1$** | ZC706 | 100,000 k | 9,943 | 70 | 100 | 0 | 0.286 | 95.97 |
| **Proposed$^2$** | ZC706 | 100,000 k | 109,653 | 90 | 100 | 0 | 1.106 | 97.26 |
| **ISOLET (FC)** | | | | | | | | |
| **[11] 1 Layer × 1024 AN** | ZC706 | 242 | 88,236 | NR | 300 | 287 | 4.2 | 95.70 |
| **Proposed$^1$** | ZC706 | 100,000 k | 28,553 | 70 | 100 | 0 | 0.602 | 92.36 |
| **Speech Recognition (FC)** | | | | | | | | |
| **[11] 1 Layer × 1024 AN** | ZC706 | 266 | 82,856 | NR | 300 | 192 | 3.9 | 94.80 |
| **Proposed$^1$** | ZC706 | 100,000 k | 13,302 | 70 | 100 | 0 | 0.392 | 90.59 |
| **Indoor Localization (FC)** | | | | | | | | |
| **[11] 1 Layer × 1024 AN** | ZC706 | 273 | 77,671 | NR | 300 | 148 | 2.8 | 96.60 |
| **Proposed$^1$** | ZC706 | 100,000 k | 7,587 | 70 | 100 | 0 | 0.205 | 91.87 |
| **CIFAR-10 (CNV)** | | | | | | | | |
| **Architecture** | **Platform** | **Through. (FPS)** | **LUTs** | **Lat. (ns)** | **Freq. (MHz)** | **BRAM 36 Kb** | **Power (W)** | **Acc. (%)** |
| **Proposed$^3$** | ZC706 | 100,000 k | 28,735 | 40 | 100 | 0 | 0.591 | 81.3 |

Proposed$^1$: FC, 20 Trees and 100 Splits.
Proposed$^2$: FC, 50 Trees and 300 Splits.
Proposed$^3$: CNV, 10 Trees and 400 Splits.

require any memory access since it does not have any weights nor activations, which is not the case in FINN [65] that requires 396 36-kb BRAMs to store its weights and activations.

For other studied FC applications such as ISOLET, Speech Recognition and Indoor Localization, POLYBiNN achieves acceptable accuracy compared to the baseline accuracy in [11]. Although the work in [11] uses power-of-two inputs and activations, which allows using shift units instead of multipliers, POLYBiNN utilizes 70%-90% fewer LUTs compared to the LUTs required to implement [11] for the same application. In terms of throughput, POLYBiNN achieves 366× greater throughput than [11] while consuming 9× less power. Moreover, POLYBiNN does not require any memory access.

Comparing the implementation costs in the case of CIFAR-10 cannot be made with prior works since POLYBiNN only replaces the fully connected layers of a CNN. Still, our implementation achieves 81.3% accuracy while using 28,735 LUTs using the set of convolution layers in [65] and [105] (3×3 CONV, 3×3 CONV and 2×2 MAXPOOL repeated three times with 64-128-256 channels). We expect that the accuracy could be improved with more complex convolution layers. The fully connected layers implemented with POLYBiNN consume only 0.591 W. POLYBiNN is expected to achieve higher classification accuracy on the CIFAR-10 dataset with a more complex set of convolutional layers.

The reported results in Table 5.1 correspond to the original voting circuit implementation. We also tested the optimized voting circuit on the MNIST dataset since it has an unbalanced number of training samples of its output classes. The classification accuracy of the testing dataset reduces by 0.05% (only 5 more samples were wrongly classified) of the full testing dataset as compared to the classification accuracy. In terms of resource utilization, the optimized voting circuit uses only 15% of the required LUTs to implement the original voting circuit. It reduces the number of required LUTs to implement POLYBiNN by 8% in the case of MNIST with 20 DTs and 100 splits.

### 5.5.3 Teacher-to-Student POLYBiNN

In this section, we discuss an attempt to improve the classification accuracy of POLYBiNN architecture with fewer resources in FPGAs.

We tested the two-stage teacher-to-student training algorithm [6] to train POLYBiNN with the predicted outputs of a trained BNN to simplify the task for POLYBiNN, make POLYBiNN immune to overfitting and achieve equivalent performance as BNNs. First, we trained a BNN on the original training data. The trained BNN served as a teacher for POLYBiNN. Then, POLYBiNN, was trained to mimic the teacher's behavior. This works by training the

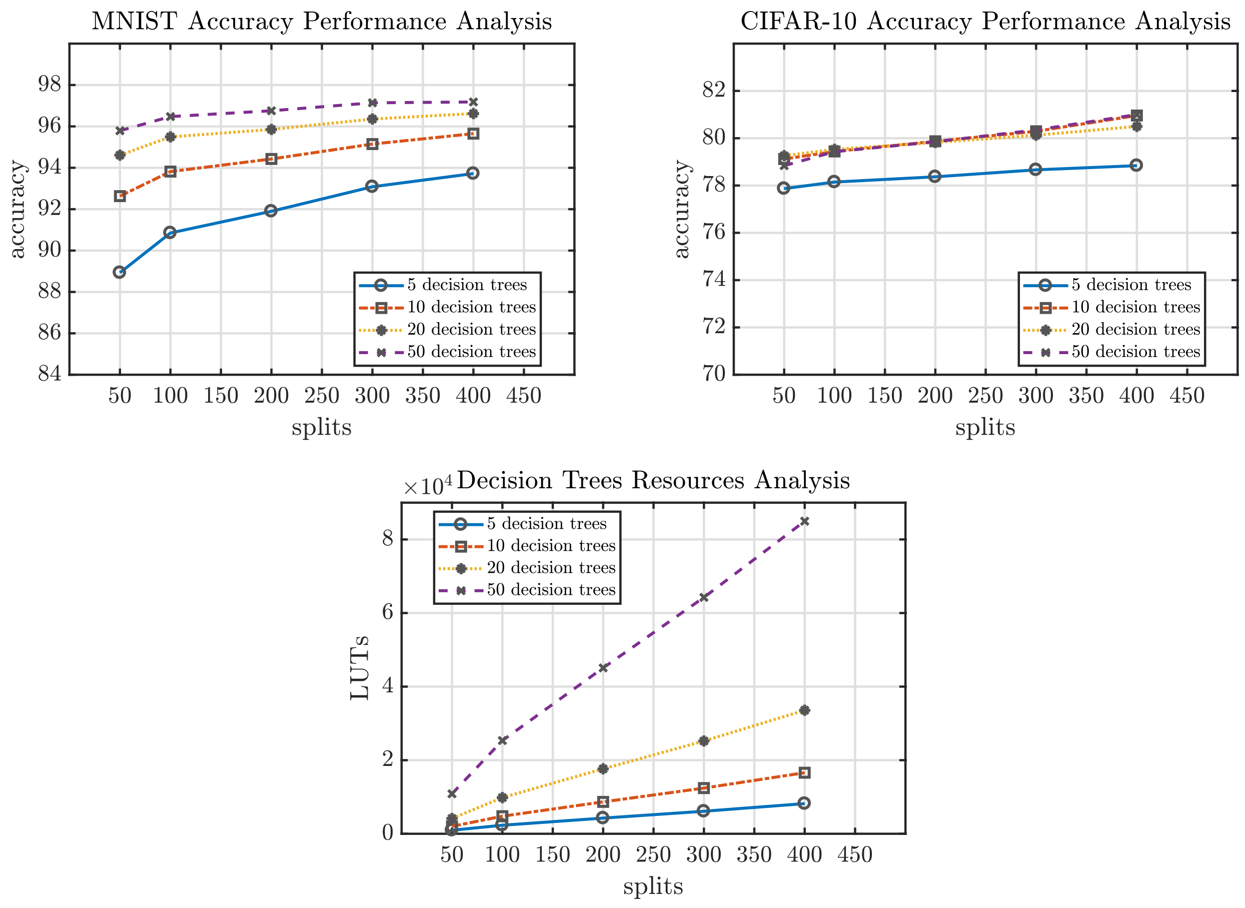


Figure 5.9 POLYBiNN performance analysis for MNIST and CIFAR-10 in terms of accuracy vs number of decision trees and number of splits for each class

teacher BNN with the original data to generate the corresponding predicted outputs. The predicted outputs of the teacher BNN were used as labeled data to train POLYBiNN using the same training dataset. In other words, instead of training POLYBiNN using the original labels, we trained POLYBiNN using the expected outputs of the trained BNN. Using this training approach on MNIST, POLYBiNN achieves the same classification accuracy with fewer resources. On MNIST, POLYBiNN with just 10 DTs per class and 50 splits, when trained using the teacher-to-student approach, achieves the same classification accuracy of POLYBiNN with 20 DT and 100 splits when trained using the traditional training approach.

## 5.6 Conclusion

This paper presented POLYBiNN, a high performance and scalable combinatorial inference engine for binary NNs. POLYBiNN consists of a stack of binary decision trees that are integrated into existing NNs to achieve the same performance as binary DNNs and CNNs. POLYBiNN is an energy and area efficient architecture since it avoids multiplications and floating point operations. In order to avoid the unacceptable latency and tedious steps for generating a low-level hardware description of a given application, we propose a tool that automates that process. Our test cases show that the POLYBiNN architecture on FPGA achieves the same accuracy as binary DNNs with fewer computational resources and up to 67$\times$ higher throughput on the MNIST dataset. Future works will focus on the full integration of POLYBiNN into CNN-based applications and their full implementations.

## 5.7 Acknowledgments

The authors would like to thank Safa Berrima, Imad Benacer, Jeferson Santiago da Silva, Thibaut Stimpfling and Thomas Luinaud for their insightful comments.

# CHAPTER 6 ARTICLE 4: POLYCINN: MULTICLASS BINARY INFERENCE ENGINE USING CONVOLUTIONAL DECISION FORESTS

Authors: Ahmed M. Abdelsalam, Ahmed Elsheikh, Jean-Pierre David and J.M. Pierre Langlois

Published in: IEEE International Conference on Design and Architectures for Signal and Image Processing (DASIP) in 2019 [14]

***Abstract*–Convolutional Neural Networks (CNNs) have achieved significant success in image classification. One of the main reasons that CNNs achieve state-of-the-art accuracy is using many multi-scale learnable windowed feature detectors called kernels. Fetching of kernel feature weights from memory and performing the associated multiply and accumulate computations consume massive amount of energy. This hinders the widespread usage of CNNs, especially in embedded devices. In comparison with CNNs, decision forests are computationally efficient since they are composed of decision trees, which are binary classifiers by nature and can be implemented using AND-OR gates instead of costly multiply and accumulate units. In this paper, we investigate the migration of CNNs to decision forests as one of the promising approaches for reducing both execution time and power consumption while achieving acceptable accuracy. We introduce POLYCiNN, an architecture composed of a stack of decision forests. Each decision forest classifies one of the overlapped sub-images of the original image. Then, all decision forest classifications are fused together to classify the input image. In POLYCiNN, each decision tree is implemented in a single 6-input Look-Up Table and requires no memory access. Therefore, POLYCiNN can be efficiently mapped to simple and densely parallel hardware designs. We validate the performance of POLYCiNN on the benchmark image classification tasks of the MNIST, CIFAR-10 and SVHN datasets.**

## 6.1 Introduction

Convolutional Neural Networks (CNNs) have been overwhelmingly dominant in many computer vision problems, especially image classification [4]. The recent success of CNNs is mainly due to the tremendous development of many deep architectures such as AlexNet [1], GoogleNet [2] and ResNet [4]. These deep CNN architectures are trained to extract representative features from their inputs through several non-linear convolutional layers. Typically, in each convolutional layer many pre-trained windowed feature detectors called kernels are applied on their inputs. One or more fully connected layers connect the top-level extracted features and make a classification detection.

Although CNNs achieve state-of-the-art accuracy in many tasks, they have deficiencies that limit their use in embedded applications [4]. A main downside of CNNs is their computational complexity. They typically demand many Multiply and Accumulate (MAC) and memory access operations in both training and inference [71]. Another drawback of CNNs is that they require careful selection of multiple hyper-parameters such as the number of convolutional layers, the number of kernels, the kernel size and the learning rate [4]. This results in a large design space exploration that makes the training process of CNNs time consuming because of several interfering parameters with many configurational combinations. Current CNN applications are typically trained and run on clusters of computers with Graphical Processing Units (GPUs). However, the limited throughput of mainstream processors and the high power consumption of GPUs limit their applicability in embedded and edge computing CNN applications [25].

Recently, there has been increased interest in other classifiers that should 1) suit the nature of hardware accelerators by fully utilizing their specific computing resources to maximize parallelism when executing a large number of operations [12, 115]; 2) achieve acceptable classification accuracy [29, 116, 117]; 3) be amenable to finding a robust model for a given task; and 4) be simple to train [118]. Decision Forests (DFs) were introduced as efficient models for classification problems [119]. They operate by constructing a stack of Decision Trees (DTs) and then voting on the most popular output class. Since DTs are binary in nature and can be implemented using AND-OR gates, DFs can be efficiently mapped to simple and densely parallel hardware architectures [101, 102]. Moreover, DFs can be trained quickly and are considered handy classifiers since they do not have many hyper-parameters [97]. However, by contrast to CNNs, DFs do not achieve state-of-the-art accuracy on several applications [115]. CNNs outperform DFs in terms of accuracy because they deploy several convolutional layers with many kernels to extract representative features from raw data. On the other hand, DFs divide the feature space into subspaces based on simple comparison operations on the input data.

The motivation of this paper is based on three observations. The first observation stems from the fact that CNNs achieve state-of-the-art accuracy by sliding many kernels over images. This motivates us to propose convolutional DFs, where DFs are applied over a sliding window (sub-images) on the original image. The second observation is that most Field-Programmable

Gate Arrays (FPGAs) fit any function with 6-bit inputs in a single Look-Up Table (LUT). Therefore, we limit the number of nodes of the DTs utilized in our convolutional DFs to six. Each DT can thus be optimally implemented in one LUT. This idea could be generalized to wider LUTs. The third observation is that DFs, by contrast to CNNs, are not good feature extractors. We thus integrate a low complex feature extraction layer before the proposed convolutional DFs. This allows to achieve high performance with a significantly reduced amount of resources, and to propose a corresponding architecture.

Training DFs to learn both representative features of the input data and the final classifiers in a joint manner is a difficult problem [116]. This paper thus introduces POLYCiNN, an architecture composed of a stack of DFs. POLYCiNN follows the sliding kernels idea of CNNs and divides each input image into several overlapped sub-images. Then, POLYCiNN trains a different DF to classify each sub-image. A decision fusion algorithm is applied to combine all DF classifications. In order to achieve near state-of-the-art accuracy, we use a Local Binary Pattern (LBP) layer that extracts representative features of the inputs efficiently and with simple computations. We demonstrate POLYCiNN's capabilities on the MNIST, CIFAR-10 and SVHN datasets.

The specific contributions of this paper are as follows:

- We introduce POLYCiNN, an efficient classifier based on 6-input LUT DTs.
- We integrate a simple LBP feature extraction layer to POLYCiNN and show that we can obtain near state-of-the-art accuracy with a reduced input set of binary parameters.
- We explore different meta-parameters to optimize POLYCiNN in terms of classification accuracy and resource utilization.
- We validate POLYCiNN on the MNIST, CIFAR-10 and SVHN datasets and demonstrate the potential of using POLYCiNN for different applications.

The rest of the paper is organized as follows. Section 6.2 provides a review of the related works. A detailed view of POLYCiNN, including its training, inference and its hardware implementation, are described in Section 6.3. Section 6.4 is dedicated to the experimental results, comparison with other architectures and discussion. Section 6.5 concludes the paper.

## 6.2 Related works

Various approaches have been proposed to simplify the computational and memory requirements of CNNs. One of these approaches is to use single-bit data representation for inputs and parameters [103]. Courbariaux et al. [30] proposed binary kernels and activations of convolutional layers. Since kernels and activations are binary, multiply and accumulate operations of convolutional layers are then replaced by XNORs and counters. Several works [117] [120] exploited this paradigm and proposed its implementation on different hardware accelerators such as FPGAs and Applications-Specific Integrated Circuits (ASICs). Although binary CNNs achieved competitive accuracies for the MNIST, SVHN and CIFAR-10 datasets, they are still expensive in terms of computations and memory access since they required many layers and many kernels in each layer.

Another approach to simplify the computational and memory complexity of CNNs is to implement them as DTs. Frosst et al. [121] proposed a method to distil knowledge from trained neural networks to DTs. This method allows DTs to generalize better than DTs learned directly from the training data. Zhang et al. [122] roughly represented the rationale of each CNN prediction using a semantic DT structure. Both methods achieve acceptable accuracy on the MNIST dataset but they lack performance on complex applications such as CIFAR-10 and SVHN. Abdelsalam et al. [12] [13] proposed POLYBiNN, a stack of DTs that replaces fully connected layers of CNNs. Although they achieve near state-of-the-art accuracy on CIFAR-10, they require several convolutional layers to extract features from raw data.

The success of CNNs over DFs owes to layer-by-layer processing and in-model feature extraction [4]. Therefore, many works explored the possibility of building deep layered DFs to extract representative features that achieve near state-of-the-art accuracy. Zhou et al. [116] proposed deep DFs where the output vector of each DF is fed as the input to the next layer of DFs. Miller et al. [123] proposed forward thinking, a general framework for training deep DFs layer by layer. The authors demonstrated a proof of concept of their ideas on the MNIST dataset. However, the idea does not scale up with complex applications such as CIFAR-10 and SVHN. In addition, implementing multi-layer DFs faces the same complexity as implementing CNNs, especially when DFs have many DTs with many nodes.

## 6.3 The POLYCiNN Architecture

In this section, we detail the POLYCiNN architecture, show how we extract representative features using LBP, demonstrate how we train POLYCiNN using simple sliding DFs, and show how it can be implemented in an FPGA.

### 6.3.1 Architecture Overview

Fig. 6.1 shows an overview of the POLYCiNN architecture. POLYCiNN starts by encoding input images using a LBP descriptor, a simple yet powerful descriptor for image classification applications [124]. The features vectors alongside with the downsampled version of the input image are fed to a stack of POLYBiNNs [12]. Each POLYBiNN is a DF that is composed of a $M \times N$ array of DTs, where $M$ is the number of classes and $N$ is the number of trees per class, followed by a voting circuit. The voting circuit outputs of different POLYBiNNs are combined together using a decision fusion circuit to make the final classification.

Sliding windows and image pyramids play an integral role in image classification since they allow classifiers to localize different objects at various scales [**?**]. We exploit that concept and divide the input images into $w$ overlapped windows. Moreover, we downsample the original images and divide them into the same number of windows. We train a stack of POLYBiNNs, where each POLYBiNN classifies one image window using the extracted LBP features vector of the original window and the corresponding window of the Downsampled Image (DI). Fig. 6.1 shows an example for the CIFAR-10 dataset, where $w = 9$, with 16×16 windows and a stride of eight pixels, and the downsampled image is 8×8 with nine windows of size 6×6 with a stride of one pixel.

### 6.3.2 Local Binary Pattern Feature Extraction

The main goal of this layer is to obtain the most relevant information from inputs and represent that information in a lower dimensionality space. We choose LBP descriptors [124] because they measure the spatial structure of local image texture efficiently and with parallel simple computations that fit the nature of most hardware accelerators such as FPGAs. The LBP descriptor is formed by comparing the intensity of the center pixel to its neighboring pixels within a patch. Neighbor pixels with higher intensity than the center pixel are assigned a value of 1, and 0 otherwise. LBP patch sizes are normally 3×3, 5×5, etc., however, we restrict the intensity comparison of the center pixel to its four adjacent neighbor pixels (top, right, bottom and left) which reduces memory access cost. This approach is more suitable for hardware implementation since the comparisons are computed row-wise and column-wise, as discussed in Section 6.3.4.

Each pixel is now represented with a 4-bit string computed by comparing the pixel intensity to its four corresponding neighbors intensities. The final feature vector of each window is the histogram of the feature values within the corresponding window. The histogram provides better discrimination of the inputs and diminishes the dimensionality space of the inputs to

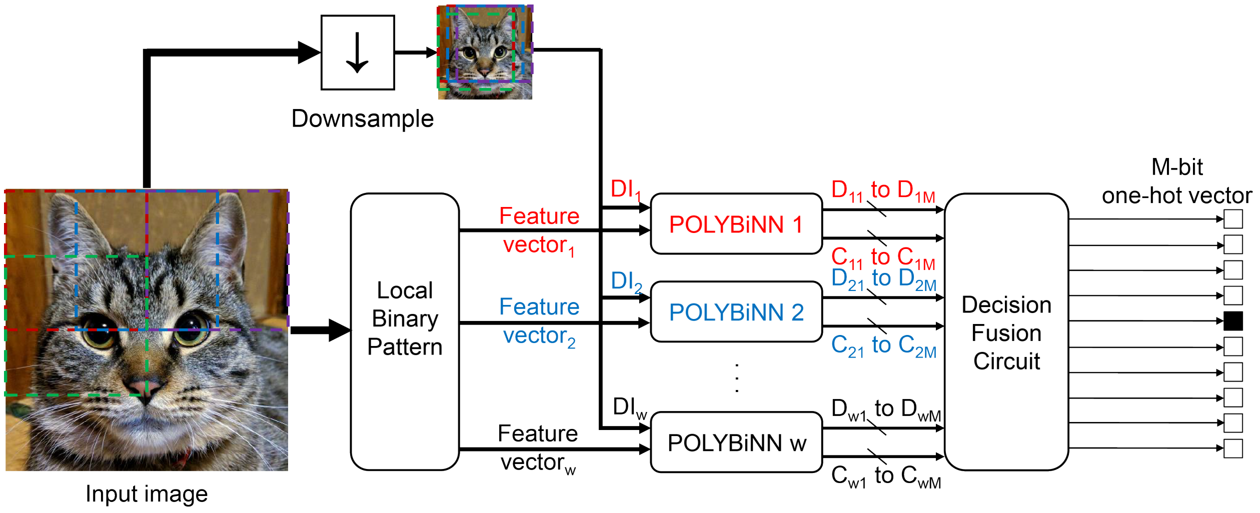


Figure 6.1 Overview of the POLYCiNN architecture with $w$ windows and $M$ classes

16 (all possible values of a 4-bit string). Fig. 6.2 shows an example of computing the LBP features vector of a local image window.

## 6.3.3 POLYCiNN Training Algorithm

We train each POLYBiNN classifier on its corresponding LBP feature vector and downsampled image window. POLYBiNNs are trained using AdaBoost, an ensemble learning algorithm that creates complex classifiers by combining many weak DTs [13]. We limit the number of nodes of each DT to six in order to implement each DT as a single 6-input LUT. Once all $N$ DTs within the same POLYBiNN have been trained, their outputs are combined to come to $M$ decisions ($D_1$ to $D_M$) with $M$ confidences ($C_1$ to $C_M$). The output ($D_m$) is

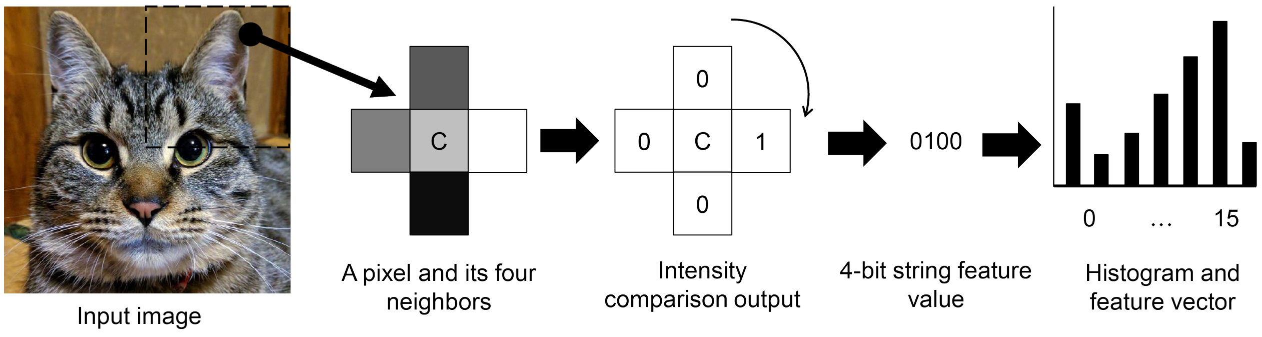

Input image
C
A pixel and its four neighbors
0
0
C
1
0
Intensity comparison output
0100
4-bit string feature value
0
...
15
Histogram and feature vector

Figure 6.2 Local binary pattern encoding process

the binary decision of class ($m$) in the corresponding POLYBiNN and can be 0 or 1. The output ($C_m$) is a 2-bit confidence value of the corresponding binary decision ($D_m$).

When the training process of all POLYBiNNs is completed, we merge their outputs using a decision fusion approach to obtain a decision for a given input. For each class $m$, the final confidence $CF_m$ is computed by summing all the corresponding $C_m$ together. We select the class with the highest confidence as the final classification decision. Fig. 6.3 shows the overall process.

### 6.3.4 Implementing POLYCiNN in Hardware

As shown in Fig. 6.1, POLYCiNN consists of a LBP feature extraction layer, a stack of POLYBiNNs where each POLYBiNN is composed of an array of DTs followed by a voting circuit, and finally a decision fusion circuit that merges POLYBiNN classifications. We propose an efficient hardware implementation of the LBP layer, as shown in Fig. 6.4. The architecture is composed of two arrays of comparators: row comparators and column comparators. The row array of comparators compares between the intensity of each given pixel and the intensity of its adjacent bottom neighbor pixel in the consecutive row. The column array of comparators compares between the intensity of each given pixel and the intensity of its adjacent right neighbor pixel in the consecutive column. The output of each comparator is assigned a value of 1 or 0, as discussed in Section 6.3.2.

In the proposed LBP layer, we compare the given pixel intensity to its four adjacent neighbor pixels (top, right, bottom and left). However, in the proposed array of comparators we only compute the south and east comparisons. Therefore, the natural and complemented versions of comparator outputs are used to form the 4-bit string feature value of each color channel of a given pixel. Once the 4-bit string of each channel of all pixels is formed, a set of 16 comparators and accumulators construct the histogram of the computed feature values and compute the feature vector of each window for a given image. Fig. 6.4 shows the proposed implementation of the LBP layer in hardware.

LBP feature vectors of different image windows are computed then fed alongside with their corresponding downsampled images the stack of POLYBiNNs, as shown in Fig. 6.1. Each DT of different POLYBiNNs corresponds to a Sum of Product (SOP) that is implemented in a single LUT since the number of inputs is six, a constraint set during the training process of POLYCiNN. Since DTs are binary classifiers by nature, their inputs should be binary. Therefore, the extracted feature vectors and the downsampled images, which serve as DTs inputs, are binarized with thresholds that are learned during the training process of

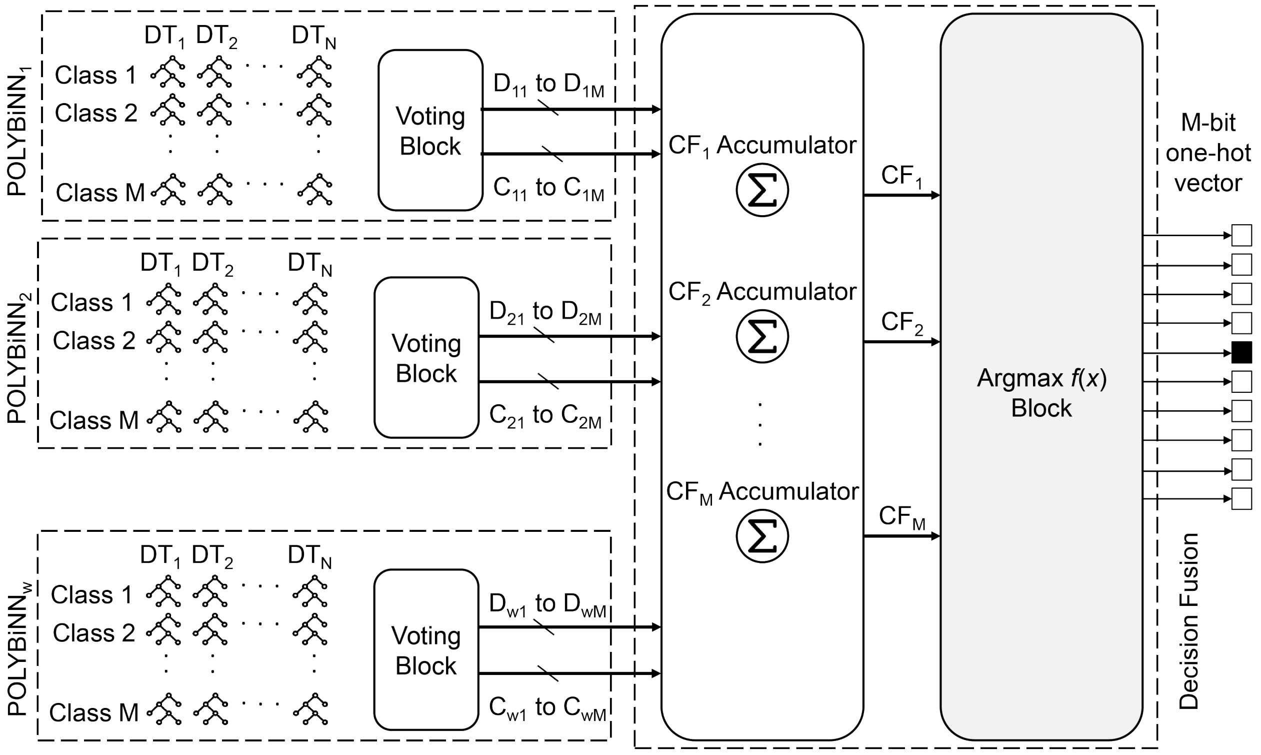


Figure 6.3 Decision forests and decision fusion implementation of POLYCiNN

POLYCiNN. A set of comparators binarize the extracted feature vectors and downsampled images according to the learned thresholds.

The decisions $D_{wm}$ and confidences $C_{wm}$ of each POLYBiNN of the corresponding image window are computed without any arithmetic operations as detailed in [12]. The $w$ confidences

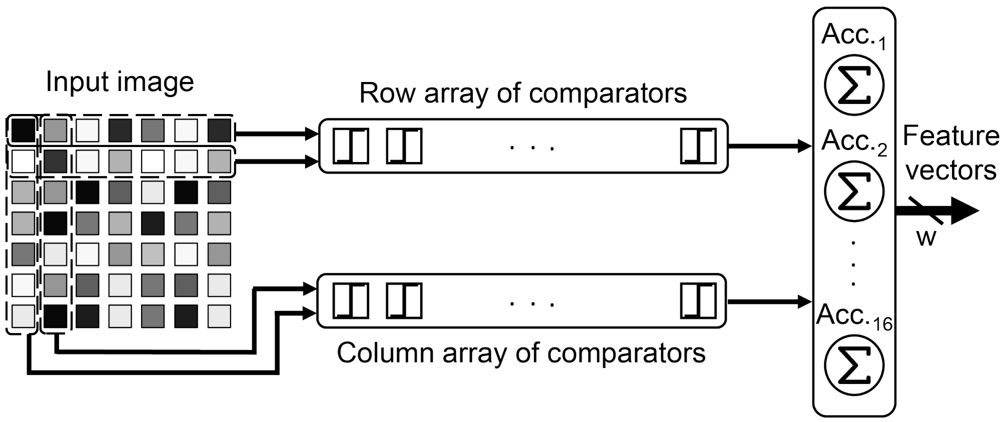


Figure 6.4 Local binary pattern hardware implementation in POLYCiNN

of each class $m$ are summed using a set of $M$ accumulators in the decision fusion circuit, as shown in Fig. 6.3. A set of pipelined comparators is used in the Argmax block to select the class with the highest confidence. Since the POLYCiNN architecture is well suited to parametrized descriptions, we expanded and adapted the tool proposed in [12] to generate a synthesizable HDL description of POLYCiNN given a set of parameters.

## 6.4 Experimental Results and Discussions

This section presents and analyzes the classification accuracy of POLYCiNN with different parameters, compares it to the literature and discusses its hardware cost.

### 6.4.1 POLYCiNN Classification Performance

We tested POLYCiNN on the MNIST (28×28 handwritten digits from 0 to 9), CIFAR-10 (32×32 color images in 10 categories) and SVHN (32×32 color image of street view house numbers in 10 categories) datasets. We used the binary version of MNIST version where each pixel is represented in one bit. In case of CIFAR-10 and SVHN, we only used the 4-MSB of each pixel value in the three color channels. We considered three different sets of experiments for classifying the datasets. The first set classified the three datasets when training POLYCiNN using raw data. In the second set of experiments, we trained POLYCiNN using the extracted LBP feature vectors. The third set trained POLYCiNN using the extracted LBP feature vectors alongside with the downsampled image.

For reasons of comparison, we reproduced the results of training POLYBiNN [12] on the raw data of the three datasets. We also studied the effect of changing the number of DTs for each POLYBiNN on the accuracy. We trained POLYBiNN for classifying the three datasets with 100, 500 and 1000 DTs per class. In the case of POLYCiNN, we trained it with 100, 500 and 1000 DTs per class for each window. For CIFAR-10 and SVHN, we divided each image into nine windows of size 16×16 and a stride of eight pixels. For MNIST, we divided each image into nine windows of size 22×22 and a stride of three pixels.

Fig. 6.5 shows the accuracy for MNIST, CIFAR-10 and SVHN as a function of the number of DTs in different experiments. In the three datasets, POLYCiNN outperforms POLYBiNN in terms of classification accuracy when both are trained using raw data. This is because POLYCiNN has the advantage of using sliding DFs. The accuracy curves for CIFAR-10 and SVHN indicate that using LBP features is a powerful approach that can achieve the same performance as using raw data. However, Fig. 6.5 shows that training POLYCiNN using LBP features and downsampled images achieves higher classification accuracy for the three datasets. This is because each POLYBiNN is trained using the LBP feature vector of its corresponding window alongside with its neighbor area from the downsampled image. The accuracy of POLYCiNN is sensitive to the number of DTs up to a limit where the classification accuracy saturates, as shown in Fig. 6.5. Table 6.1 compares the accuracy of POLYCiNN for the MNIST, CIFAR-10 and SVHN datasets with prior works. Although POLYCiNN is inferior to state-of-the-art CNNs [1,2,28] in terms or accuracy, we obtain better results when compared to other DF approaches [12,115,116].

POLYCiNN with LBP feature extraction layer and downsampled image suits the CIFAR-10 and SVHN datasets more than the MNIST dataset, as shown in Fig. 6.5. This is because the variability in the CIFAR-10 and SVHN datasets in terms of image translations, scales, rotations, color spaces and geometrical deformations are much more than those in the MNIST dataset. When a space of high variability is divided into sub-spaces using DFs, the variations become less evident and the overall performance of classifier fusion on all sub-spaces gains are notable. On the other hand, when there are few variations, the gains of space division are less notable since there is not much reduction in variability.

### 6.4.2 Hardware Implementation

The hardware implementation of POLYCiNN is composed of three main parts 1) extracting LBP features, 2) decision forests of decision trees and 3) decision fusion circuit. As discussed in Section 6.3.4, the simple computations needed to compute LBP feature vectors should not hinder the implementation process since these features are computed using arrays or comparators. The delay caused in this part by awaiting neighbor pixels to be loaded is negligible. This is because the arrays of comparators access the memory symmetrically (row by row and column by column). Moreover, this approach of implementing LBP can be parallelized by using many arrays of comparators, which increases the throughput at the cost of computational and memory access resources.

Table 6.1 Accuracy comparison with existing decision tree approaches

| | **Accuracy (%)** | | |
|---|---|---|---|
| | **MNIST** | **CIFAR-10** | **SVHN** |
| **[116]** | 99.10 | - | - |
| **[121]** | 96.76 | - | - |
| **[115]** | 99.26 | 63.37 | - |
| **[12]** | 97.45 | 55.12 | 71.68 |
| **POLYCiNN*** | 98.23 | 63.43 | 76.35 |

* nine windows per image and 1000 DTs per class of each window.

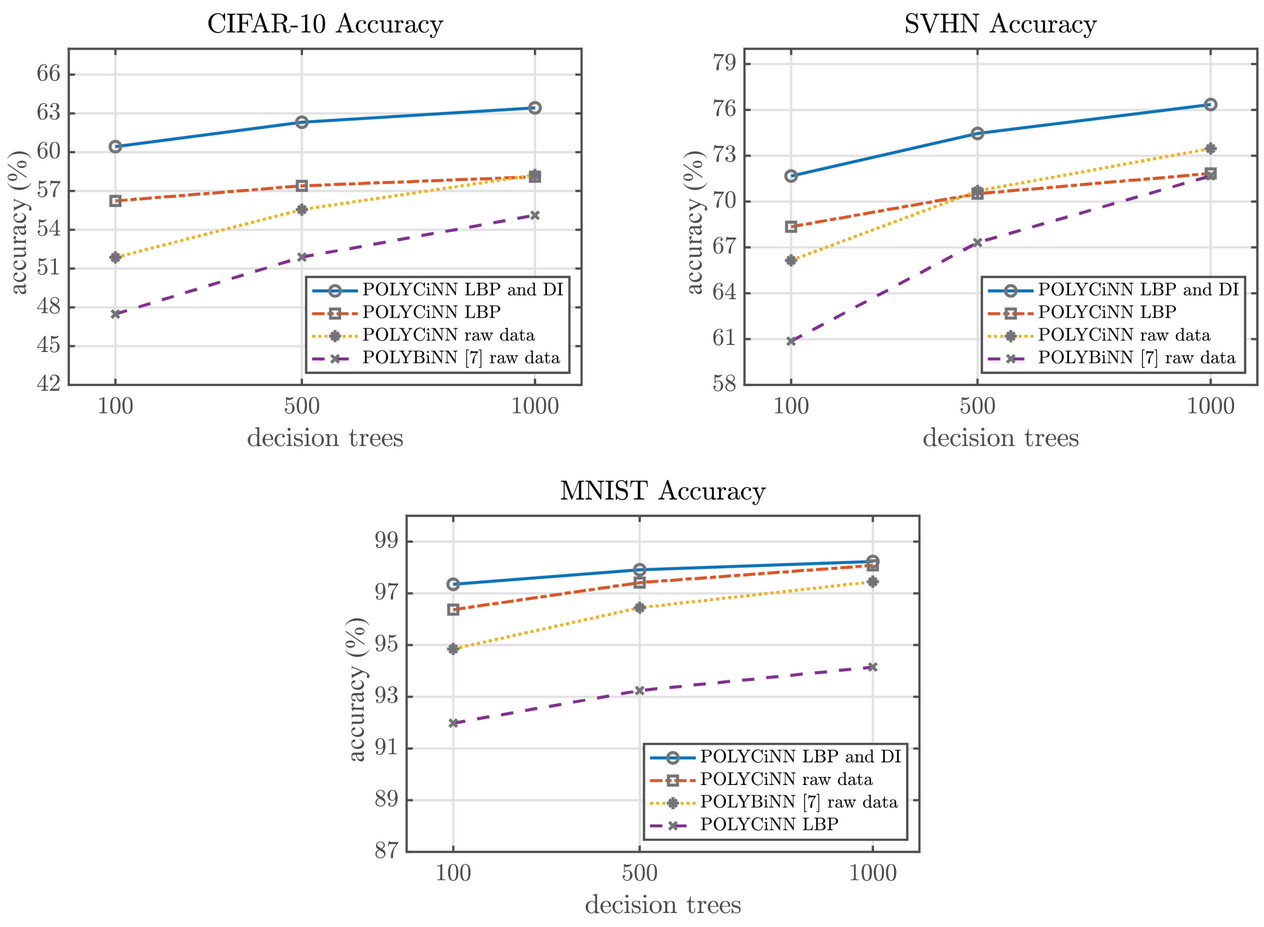


Figure 6.5 POLYCiNN accuracy for the CIFAR-10, SVHN and MNIST datasets

Concerning the second part, all DTs are have decision nodes as maximum. Consequently, each DT is implemented in a single LUT. It should be noted that DTs with more nodes can be utilized to increase the classification accuracy. However, this requires more LUTs to implement these DTs. The third part of the decision fusion circuits uses few accumulators and a set of pipelined comparators that should not restrict the implementation capabilities. Future works will focus on experimenting different implementations of POLYCiNN in FPGAs.

## 6.5 Conclusion

This paper presented POLYCiNN, a classifier inspired by CNNs and Decision Forest (DF) classifiers. POLYCiNN comprises a stack of DFs, where each DF classifies one of the overlapped image windows. POLYCiNN deploys an efficient LBP feature extraction layer that improves its classification accuracy. We demonstrated that POLYCiNN achieves the same accuracy as prior DF approaches on the MNIST, CIFAR-10 and SVHN datasets. From a hardware perspective, POLYCiNN can be implemented using efficient computational and memory resources. Moreover, it can be configured to suit various hardware accelerators and embedded devices.

## 6.6 Acknowledgments

The authors would like to thank Imad Benacer and Siva Chidambaram for their insightful comments.

# CHAPTER 7 GENERAL DISCUSSION

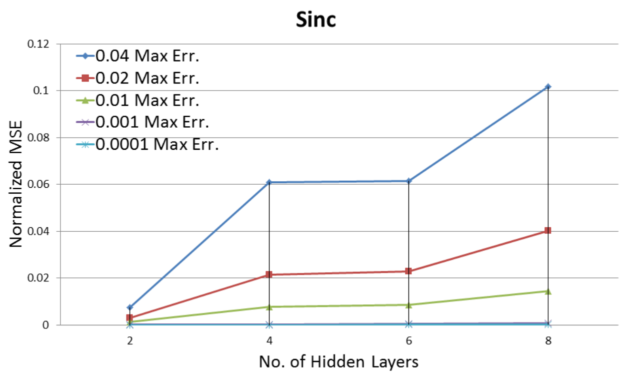
In the research presented in this thesis, we proposed efficient FPGA architectures to achieve higher performance for deep learning applications in general and more specifically for DNN and CNN inference. The contributions of this work can be summarized into four main components.

We proposed an efficient approximation and its implementation of the non-linear hyperbolic tangent activation function of NNs in FPGA. We started with the implementation of activation functions since it is a basic element of ANs, is crucial in terms of NN performance and entails many computations as well. The hyperbolic tangent activation function tends to work better than ReLU on deeper models across a number of applications, especially in RNNs and Long Short-Term Memories (LSTMs) [125]. The tanh function is used as the gating function in LSTMs. Since its output is bounded as a value between 0 and 1, it can either let no flow or complete flow of information throughout the gates. Our proposed FPGA implementation of the tanh function achieves 3× better precision than previous works while using a similar amount of computational resources and a small amount of memory. The proposed approximation is highly scalable according to the available computational and memory resources and the required precision of the function. In addition, it is highly configurable since it can approximate other non-linear functions.

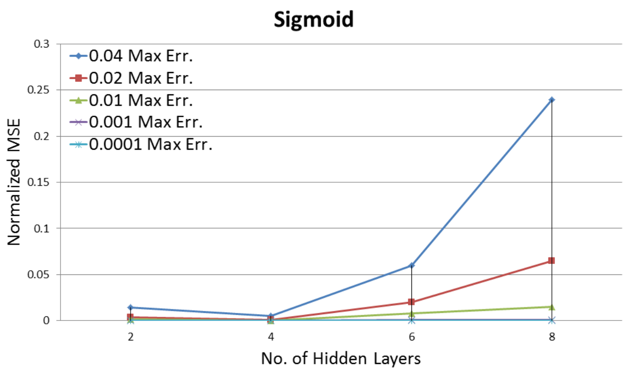
In order to assess the impact on the accuracy of out *tanh* approximation on DNN performance, we trained and tested several DNN architectures. We conducted this experiment on two classification problems, MNIST [88] and CANCER [126], and Sinc and Sigmoid functions as regression problems [127]. Table 7.1 shows the testing performance of four different datasets with several DNN architectures while employing several approximations in the testing process. All the architectures in Table 7.1 were trained using the exact hyperbolic tangent activation function without any approximation. The Sinc and Sigmoid functions were sampled in the range [-3,3] with 600 samples each and used as regression problems [127]. Training and testing instances were selected randomly by 420 and 180 samples, respectively, for both functions. Sinc and Sigmoid functions results in Table 7.1 show that the normalized Mean Squared Error (MSE) value ($\text{MSE}_{\text{approx}}$ - $\text{MSE}_{\text{exact}}$) is increased when using less accurate approximations for the same DNN architecture. In addition, the normalized MSE is getting larger when the DNN architecture becomes more complex with more hidden layers as shown in Figure 7.1. MNIST results in Table 7.1 show that the testing accuracy of the classification process is highly affected by the precision of the approximation. Although the testing per-

Table 7.1 Testing errors of Sinc, Sigmoid, MNIST and Cancer datasets using different hyperbolic tangent approximations

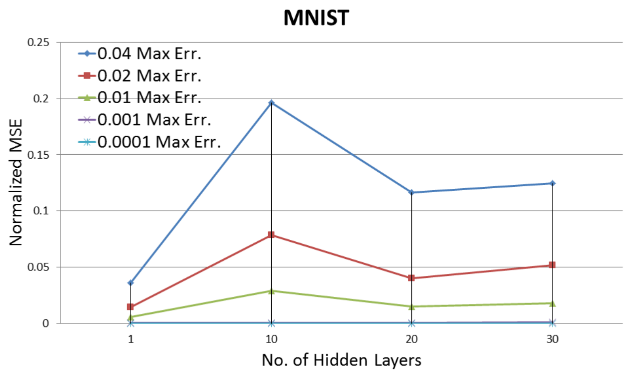
| | DNN Architecture | Tanh Max. Error | Correlation | Avg. MSE | Norm. MSE | | DNN Architecture | Tanh Max. Error | Testing Acc. (%) | Avg. MSE | Norm. MSE |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Sinc | 4 Hidden Layers x 5 ANs | 0.04 | 0.77237 | 0.1432 | 0.0608 | MNIST | 1 Hidden Layer x 30 ANs | 0.04 | 57.9 | 0.1960 | 0.0362 |
| | | 0.02 | 0.89024 | 0.1046 | 0.0216 | | | 0.02 | 72.8 | 0.1745 | 0.0147 |
| | | 0.01 | 0.92115 | 0.0902 | 0.0079 | | | 0.01 | 77.5 | 0.1657 | 0.0059 |
| | | 0.001 | 0.93335 | 0.0823 | 0.0002 | | | 0.001 | 77.8 | 0.1602 | 0.0004 |
| | | 0.0001 | 0.93373 | 0.0818 | 0.0000 | | | 0.0001 | 77.7 | 0.1599 | 0.0001 |
| | | 0 | 0.93376 | 0.0817 | 0 | | | 0 | 77.7 | 0.1598 | 0 |
| | 6 Hidden Layers x 5 ANs | 0.04 | 0.77797 | 0.1428 | 0.0615 | | 10 Hidden Layers x 30 ANs | 0.04 | 47.5 | 0.3045 | 0.1965 |
| | | 0.02 | 0.89177 | 0.1036 | 0.0229 | | | 0.02 | 73.4 | 0.1868 | 0.0788 |
| | | 0.01 | 0.91246 | 0.0899 | 0.0085 | | | 0.01 | 81.1 | 0.1369 | 0.0289 |
| | | 0.001 | 0.92553 | 0.0822 | 0.0006 | | | 0.001 | 82.6 | 0.1086 | 0.0006 |
| | | 0.0001 | 0.92601 | 0.0820 | 0.0001 | | | 0.0001 | 82.5 | 0.1080 | 0.0000 |
| | | 0 | 0.92605 | 0.0820 | 0 | | | 0 | 82.5 | 0.1080 | 0 |
| | 8 Hidden Layers x 5 ANs | 0.04 | 0.74849 | 0.1747 | 0.1018 | | 20 Hidden Layers x 30 ANs | 0.04 | 17.2 | 0.2881 | 0.1165 |
| | | 0.02 | 0.88751 | 0.1131 | 0.0402 | | | 0.02 | 22.3 | 0.2119 | 0.0403 |
| | | 0.01 | 0.92661 | 0.0874 | 0.0145 | | | 0.01 | 33.0 | 0.1867 | 0.0151 |
| | | 0.001 | 0.94100 | 0.0736 | 0.0007 | | | 0.001 | 38.1 | 0.1723 | 0.0007 |
| | | 0.0001 | 0.94137 | 0.0730 | 0.0001 | | | 0.0001 | 38.2 | 0.1716 | 0.0000 |
| | | 0 | 0.94140 | 0.0729 | 0 | | | 0 | 38.3 | 0.1716 | 0 |
| Sigmoid | 4 Hidden Layers x 5 ANs | 0.04 | 0.94904 | 0.2167 | 0.0050 | Cancer | 1 Hidden Layer x 3 ANs | 0.04 | 95.8 | 0.1166 | 0.0050 |
| | | 0.02 | 0.95131 | 0.2128 | 0.0011 | | | 0.02 | 95.8 | 0.1140 | 0.0024 |
| | | 0.01 | 0.95229 | 0.2122 | 0.0005 | | | 0.01 | 95.8 | 0.1128 | 0.0012 |
| | | 0.001 | 0.95308 | 0.2117 | 0.0000 | | | 0.001 | 95.8 | 0.1117 | 0.0001 |
| | | 0.0001 | 0.95315 | 0.2117 | 0.0000 | | | 0.0001 | 95.8 | 0.1116 | 0.0000 |
| | | 0 | 0.95316 | 0.2117 | 0 | | | 0 | 95.8 | 0.1116 | 0 |
| | 6 Hidden Layers x 5 ANs | 0.04 | 0.94403 | 0.2682 | 0.0601 | | 10 Hidden Layers x 3 ANs | 0.04 | 87.5 | 0.2663 | 0.0534 |
| | | 0.02 | 0.94615 | 0.2282 | 0.0201 | | | 0.02 | 87.5 | 0.2387 | 0.0258 |
| | | 0.01 | 0.95005 | 0.2160 | 0.0079 | | | 0.01 | 87.5 | 0.2255 | 0.0126 |
| | | 0.001 | 0.95268 | 0.2088 | 0.0007 | | | 0.001 | 87.5 | 0.2141 | 0.0012 |
| | | 0.0001 | 0.95304 | 0.2082 | 0.0001 | | | 0.0001 | 87.5 | 0.2130 | 0.0001 |
| | | 0 | 0.95305 | 0.2081 | 0 | | | 0 | 87.5 | 0.2129 | 0 |
| | 8 Hidden Layers x 5 ANs | 0.04 | 0.94042 | 0.4569 | 0.2394 | | 20 Hidden Layers x 3 ANs | 0.04 | 62.5 | 0.4314 | 0.2270 |
| | | 0.02 | 0.94836 | 0.2821 | 0.0646 | | | 0.02 | 62.5 | 0.3828 | 0.1784 |
| | | 0.01 | 0.95220 | 0.2326 | 0.0151 | | | 0.01 | 91.7 | 0.2139 | 0.0095 |
| | | 0.001 | 0.95435 | 0.2182 | 0.0007 | | | 0.001 | 91.7 | 0.2050 | 0.0006 |
| | | 0.0001 | 0.95447 | 0.2176 | 0.0001 | | | 0.0001 | 91.7 | 0.2045 | 0.0001 |
| | | 0 | 0.95448 | 0.2175 | 0 | | | 0 | 91.7 | 0.2044 | 0 |



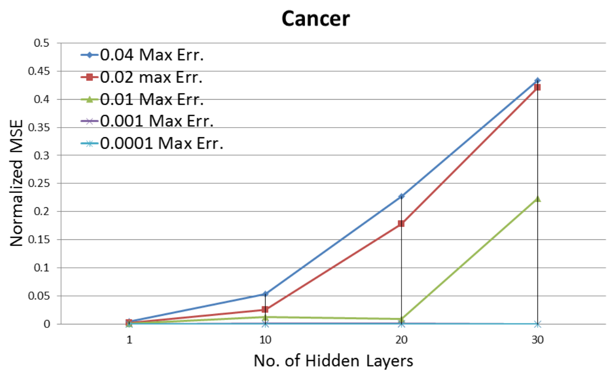
a) *Sinc* normalized MSE analysis



b) *Sigmoid* normalized MSE analysis



c) MNIST normalized MSE analysis



d) Cancer normalized MSE analysis

Figure 7.1 Performance analysis of testing different DNNs architectures employing hyperbolic tangent activation function with different accuracies

formance of the Cancer dataset does not change with different approximations for the same DNN architecture, the normalized MSE is still increasing when using DNN architectures with a large number of hidden layers as shown in Figure 7.1.

Table 7.2 shows the training accuracy of the four datasets employing the hyperbolic tangent activation function with five approximations and the exact hyperbolic tangent function in the training process of the network. The training accuracies of classification and regression problems decrease even when using precise hyperbolic tangent approximations with a maximum error of $10^{-4}$. We noticed that when the networks are trained using less accurate approximations, the training process stops early before applying the full number of epochs. Therefore, the training accuracies are badly affected compared to the training accuracies using the exact hyperbolic tangent activation function. Moreover, that would degrade the overall testing results of both classification and regression problems.

Generally, we showed that the performance of some widely used DNN architectures change using five hyperbolic tangent approximations with different accuracies. In some cases, a hyperbolic tangent function approximation with $10^{-5}$ is required in order to achieve the same performance of the exact function. Although implementing an approximation with high accuracy improves DNN performance, this requires more computational and memory resources and reduces the throughput. The proposed DCTIF approach achieves such an accurate approximation while using few computational and memory resources.

A complete AN and DNN implementation requires the extension of our *tanh* implementation work to include a weighted-sum processing unit. To improve DNN design efficiency on FPGA, we proposed a SNN multiplication-free overlay inference architecture. An FPGA overlay is a virtual configurable architecture implemented over the physical FPGA fabric. It enables FPGA programmability at a higher level of abstraction [128]. The overlay eases the way for software developers to use FPGA DNN accelerators since they can configure the overlay with the network model using a traditional C code in a couple of minutes. Moreover, it saves the HDL development, synthesis, place and route and bitstream generation time when moving from one application to another. The proposed overlay architecture uses power-of-two 3-bit inputs and activations in order to replace the multipliers in the hidden and output layers by shift units. We used a quantized power-of-two *tanh* function instead of the proposed DCTIF *tanh* function for two reasons. The first reason is that the quantized power-of-two *tanh* uses fewer computational resources and allows replacing ANs multipliers by shift units. The second reason is that we are training our SNN overlay using the teacher-to-student algorithm which compensates for the quantization of both inputs and activations. We can implement 2450 ANs in the hidden layer and 30 ANs in the output layer on ZYNQ-7000 ZC706 FPGA. The proposed overlay is a configurable SNN and it achieves DNN-level performance for many FC applications such as MNIST [88] and ISOLET [92].

It is important to recognize that while the proposed SNN overlay achieves DNN performance for FC applications, it still requires many computations and memory accesses since it uses 3-bit power-of-two inputs and activations and 8-bit fixed-point weights. Motivated by BNNs and the fact that they use binary weights and activations, we proposed a third contribution, POLYBiNN. Although BNNs use binary weights and activations as binarized versions of traditional floating point and fixed-point NNs, they are not binary by nature and such conversion from continuous to binary causes information loss. Therefore, we proposed POLYBiNN, an efficient FPGA-based inference engine for DNNs, which consists of a stack of DTs that are binary by nature. By contrast to NNs, DTs with few splits are easy to implement as LUTs in FPGAs. The proposed engine achieves a throughput of 100 million image classifications per second with 97.26% accuracy on MNIST [88]. Moreover, it does not require any memory

Table 7.2 Training errors of Sinc, Sigmoid, MNIST and Cancer using different hyperbolic tangent approximations

| DNN Architecture | Tanh Max. Error | Correlation | DNN Architecture | Tanh Max. Error | Training Acc. (%) |
|---|---|---|---|---|---|
| Sinc 8 Hidden Layers × 5 ANs, 10,000 epochs | 0.04 | 0.43279 | MNIST 1 Hidden Layer × 15 ANs, 10,000 epochs | 0.04 | 10.7 |
| | 0.02 | 0.78250 | | 0.02 | 16.4 |
| | 0.01 | 0.78976 | | 0.01 | 23.1 |
| | 0.001 | 0.84850 | | 0.001 | 31.1 |
| | 0.0001 | 0.87712 | | 0.0001 | 68.0 |
| | 0 | 0.90287 | | 0 | 68.1 |
| Sigmoid 8 Hidden Layers × 5 ANs, 10,000 epochs | 0.04 | 0.77945 | Cancer 1 Hidden Layer × 15 ANs, 10,000 epochs | 0.04 | 86.1 |
| | 0.02 | 0.80033 | | 0.02 | 86.9 |
| | 0.01 | 0.80068 | | 0.01 | 86.9 |
| | 0.001 | 0.84581 | | 0.001 | 86.9 |
| | 0.0001 | 0.85014 | | 0.0001 | 94.1 |
| | 0 | 0.86097 | | 0 | 94.1 |

access. Following the overlay idea that eased the way for software developers to use FPGA accelerators for DNNs, we proposed a tool that automatically generates a low-level hardware description of the trained POLYBiNN for a given application.

POLYBiNN achieves a good tradeoff in terms of accuracy and computational complexity for FC applications. Moreover, we showed that POLYBiNN can be used instead of the FC layers of a CNN. However, POLYBiNN lacks accuracy when used as an alternative to a full CNN for complex applications such as CIFAR-10 and SVHN. This is mainly because DTs are not adequate as CNNs in extracting representative features of the inputs. Decision trees just divide the feature space into subspaces based on simple comparison operations on the input data. Consequently, we proposed POLYCiNN, a stack of POLYBiNNs, where each POLYBiNN classifies one of the overlapped image windows of the input image. In order to improve POLYCiNN's accuracy, we deployed an efficient LBP feature extraction layer. We demonstrated that POLYCiNN achieves the same accuracy as prior DF approaches on the MNIST, CIFAR-10 and SVHN datasets. Moreover, it can be implemented using efficient computational and memory resources compared to CNNs.

Table 7.3 shows the performance of the proposed SNN overlay and POLYBiNN on FPGA in comparison to several recent FPGA DNN and CNN accelerators on MNIST dataset. Our proposed system in [13], implemented in ZYNQ ZC706 FPGA, outperforms all the existing FPGA DNN and CNN accelerators in terms throughput, BRAM and DSP requirements and latency. Comparing to BNNs on FPGAs [65], our proposed architecture [13] reaches

Table 7.3 Comparison of FPGA-based DNN and CNN accelerators for MNIST dataset

| **Architecture** | **Frequency (MHz)** | **Latency (ns)** | **LUTs** | **DSPs** | **Memory (BRAM)** | **Power (W)** | **Accuracy (%)** | **Throughput (FPS)** |
|---|---|---|---|---|---|---|---|---|
| **FPGA'17 [65]** | 200 | 310 | 91,131 | 0 | 4.5 | 7.3 | 95.83 | 12,361 |
| **FPGA'17 [65]** | 200 | 2,440 | 82,989 | 0 | 396 | 8.8 | 98.40 | 1,561 |
| **ICASSP'16 [56]** | 172 | NR | 213,593 | 900 | 750.5 | 4.98 | 98.92 | 70 |
| **IJCNN'17 [67]** | 200 | NR | 82,134 | 0 | NR | 3.80 | 97.89 | 195 |
| **FCCM'15 [129]** | 237 | NR | 265,465 | 3,599 | NR | NR | 99.64 | 131 |
| **TCAD'17 [57]** | 200 | NR | 36,384 | 167 | 35 | 0.300 | 99.01 | NR |
| **FCCM'16 [130]** | 200 | NR | 8,523 | NR | 41 | 0.700 | 99.35 | 0.016 |
| **SqueezNet [131]** | 100 | 392,000 | 20,148 | 192 | 134.5 | 2.27 | NR | 0.00262 |
| **MobileNet [132]** | 150 | NR | 139,000 | 1,452 | 1,792 | NR | NR | 0.003 |
| **Proposed 2 [11]** | 300 | 10 | 78,679 | 0 | 174 | 3.4 | 98.40 | 210 |
| **Proposed 3 [13]** | 100 | 90 | 109,653 | 0 | 0 | 1.106 | 97.18 | 100,000 |

the same accuracy while achieving 8$\times$ the throughput. Moreover, our proposed architecture outperforms small CNN architectures for mobile devices such as MobileNet [132] and SqueezNet [131] in terms of throughput, DSP requirement and power consumption. Comparing to other classifiers such as SVMs in [129], our systems utilizes 55% fewer LUTs with no DSP requirements.

# CHAPTER 8 CONCLUSION

In this thesis, we aimed to improve the mapping and scheduling of deep learning models such as DNNs and CNNs on FPGAs. Moreover, we pursued to introduce libraries and template-based compilers that can help developers transform their high-level description of deep learning models to a highly optimized FPGA accelerator with limited design expertise. This chapter summarizes the contributions of this work and discusses its limitations and future works.

## 8.1 Summary of Works

We developed a high-accuracy approximation technique to implement non-linear activation functions on FPGAs efficiently. The proposed approximation is based on DCTIF. It outperforms the existing works in terms of accuracy for similar amounts of computational resources. We used the proposed implementation as a building block to implement a multiplier-less SNN overlay on FPGA with DNN-level performance thanks to teacher-to-student training approach. The proposed overlay is energy and area efficient architecture since it avoids multiplications and floating-point operations. The overlay is configurable for many applications and convenient for users with limited FPGA design expertise. The user can configure the overlay with the network model using a traditional C code. Moreover, the overlay avoids the latency and tedious synthesis, place and route steps for regenerating a new bitstream for a given applications. We showed that the proposed multiplier-less overlay achieves the same accuracy as floating-point DNN with fewer computational resources.

We also developed POLYBiNN, a high performance combinatorial inference engine for BNNs. POLYBiNN consists of a stack of binary DTs and it achieves the same performance as binary DNNs. POLYBiNN avoids multiplications and floating-point operations since all DTs are implemented as LUTs. Our test cases show that the POLYBiNN architecture on FPGA achieves the same accuracy as binary FC DNNs with fewer computational resources and up to 67× higher throughput on the MNIST dataset. To cover binary CNN applications, we extended POLYBiNN and proposed POLYCiNN, a DF classifier inspired by CNNs. POLYCiNN comprises a stack of POLYBiNNs, where each POLYBiNN classifies one of the overlapped image windows. POLYCiNN deploys an efficient LBP feature extraction layer that improves its classification accuracy. We demonstrated that POLYCiNN achieves the same accuracy as prior DF approaches on the MNIST, CIFAR-10 and SVHN datasets. For both POLY-

BiNN and POLYCiNN, we proposed a tool that automates the process of low-level hardware description of a given application.

## 8.2 Future Works

Even though the work described in this thesis presented multiple contributions in the field of accelerating deep learning models on FPGAs, several improvements and extensions could be proposed to the solutions presented.

One of the possible extensions is to implement POLYCiNN on FPGAs and show how it can be efficiently mapped to simple and densely parallel hardware designs. The POLYCiNN architecture should be divided into three different parts: 1) the LBP feature extraction layer, 2) POLYBiNN and 3) the final decision fusion circuit. The level of parallelism and the data flow between the three parts should be studied to achieve the expected results in terms of computational and memory requirements and compare them to the related works.

An important improvement that should be studied is how to use DTs to extract features from data. In POLYCiNN, we used a feature extraction layer to extract representative features and drive them to our DT-based classifier. If we could extract meaningful features using DTs, the implementation would be much improved in terms of computational resources since it would be implemented using DTs only.

Another extension is to apply the idea of using DFs and boosting in RNNs. Recurrent neural networks have shown outstanding performance in processing sequence data. However, they are both complex and memory intensive due to their recursive nature. These limitations make RNNs difficult to embed in mobile devices requiring real-time processing with limited hardware resources. Therefore, we believe that there is potential to use the idea of DFs as an alternative to traditional RNNs.

# REFERENCES

[1] A. Krizhevsky, I. Sutskever and G. Hinton, “Imagenet Classification with Deep Convolutional Neural Networks.” Advances in Neural Information Processing Systems, pp. 1097-1105, 2012.

[2] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke and A. Rabinovich, “Going Deeper with Convolutions.” IEEE Conference on Computer Vision and Pattern Recognition, pp. 1-9, 2015.

[3] S. Haykin, Neural Networks: A Comprehensive Foundation, 2nd Edition, Englewood Cliffs, NJ: Prentice-Hall, 1999.

[4] Y. LeCun, Y. Bengio and G. Hinton, “Deep Learning.” Nature, vol. 512, no. 7553, pp. 436-444, May 2015.

[5] D. Nguyen and B. Widrow, “Improving the Learning Speed of 2-Layer Neural Networks by Choosing Initial Values of the Adaptive Weights.” IEEE International Joint Conference on Neural Nets., pp. 21-26, Jun. 1990.

[6] J. Ba and R. Caruana, “Do Deep Nets Really Need to be Deep?” Advances in Neural Information Processing Systems, pp. 2654-2662, 2014.

[7] S. Ji, W. Xu, M. Yang and K. Yu, “3D Convolutional Neural Networks for Human Action Recognition.” IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 35, no. 1, pp. 221-231, Jan. 2013.

[8] J. Misra and I. Saha, “Artificial Neural Networks in Hardware: A Survey of Two Decades of Progress.” Neurocomputing, vol. 74, no. 1, pp. 239-255, Dec. 2010.

[9] A. M. Abdelsalam, J. P. Langlois and F. Cheriet, “Accurate and Efficient Hyperbolic Tangent Activation Function on FPGA using the DCT Interpolation Filter.” ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 287-288. Feb. 2017.

[10] A. M. Abdelsalam, J. M. P. Langlois and F. Cheriet, "A Configurable FPGA Implementation of the Tanh Function Using DCT Interpolation." IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 168-171, Apr. 2017.

[11] A. M. Abdelsalam, F. Boulet, G. Demers, J. M. P. Langlois and F. Cheriet, "An Efficient FPGA-based Overlay Inference Architecture for Fully Connected DNNs." IEEE International Conference on Reconfigurable Computing, pp. 1-6, Dec. 2018.

[12] A. M. Abdelsalam, A. Elsheikh A, J. P. David and J. M. P. Langlois, "POLYBiNN: A Scalable and Efficient Combinatorial Inference Engine for Neural Networks on FPGA." IEEE Conference on Design and Architectures for Signal and Image Processing, pp. 19-24, Oct. 2018.

[13] A. M. Abdelsalam, A. Elsheikh, S. Chidambaram, J. P. David and J. M. P. Langlois, "POLYBiNN: Binary Inference Engine for Neural Networks using Decision Trees." Journal of Signal Processing Systems, pp. 1-13, May 2019.

[14] A. M. Abdelsalam, A. Elsheikh A, J. P. David and J. M. P. Langlois, "POLYCiNN: Multiclass Binary Inference Engine using Convolutional Decision Forests." IEEE Conference on Design and Architectures for Signal and Image Processing, Oct. 2019.

[15] A. R. Omondi, and J. C. Rajapakse, FPGA Implementations of Neural Networks. Vol. 365. Dordrecht, The Netherlands: Springer, 2006.

[16] S. S. Liew, M. Khalil-Hani, R. Bakhteri, "Bounded Activation Functions for Enhanced Training Stability of Deep Neural Networks on Visual Pattern Recognition Problems." Neurocomputing, vol. 5, no. 216, pp. 718-734, Dec. 2016.

[17] M. H. Hassoun, Fundamentals of Artificial Neural Networks. MIT press, 1995.

[18] S. Russell S and P. Norvig P, Artificial Intelligence. A modern approach. Prentice-Hall, Egnlewood Cliffs, 1995.

[19] I. H. Witten, E. Frank, M. A. Hal and C. J. Pal, Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann, Oct. 2016.

[20] J. Y. Yam and T. W. Chow, "Feedforward Networks Training Speed Enhancement by Optimal Initialization of the Synaptic Coefficients." IEEE Transactions on Neural Networks, vol. 12, no. 2, pp. 430-434, Mar. 2001.

[21] S. Ruder, "An Overview of Gradient Descent Optimization Algorithms." arXiv preprint arXiv:1609.04747, Sep. 2016.

[22] J. Bergstra and Y. Bengio, "Random Search for Hyper-parameter Optimization." Journal of Machine Learning Research, vol. 13, pp. 281-305, Feb. 2012.

[23] H. Larochelle, “An Empirical Evaluation of Deep Architectures on Problems With Many Factors of Variation.” ACM International Conference on Machine learning, 2007.

[24] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting.” Journal of Machine Learning Research, vol. 15, no. 1, pp. 1929-1958, Jan. 2014.

[25] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra and G. Boudoukh, “Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?.” ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 5-14, Feb. 2017.

[26] K. He, X. Zhang, S. Ren and J. Sun, “Deep Residual Learning for Image Recognition.” IEEE Conference on Computer Vision and Pattern Recognition, pp. 770-778, 2016.

[27] J. Deng, W. Dong, R. Socher, L. J. Li, K. Li, L. Fei-Fei, “Imagenet: A Large-Scale Hierarchical Image Database.” IEEE Conference on Computer Vision and Pattern Recognition, pp. 248-255, Jun. 2009.

[28] 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. Oct. 2015.

[29] P. Gysel, M. Motamedi and S. Ghiasi, “Hardware-oriented Approximation of Convolutional Neural Networks.” arXiv preprint: 1604.03168. Apr. 2016.

[30] 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. Feb. 2016.

[31] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger and A. Moshovos, “Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing.” ACM/IEEE International Symposium Computer Architecture (ISCA), pp. 1-13, Jun. 2016.

[32] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger and Y. Chen, “Compressing Neural Networks with the Hashing Trick.” In International Conference on Machine Learning (ICML), pp. 2285-2294, Jul. 2015.

[33] D. Shapiro, J. Parri, M. Desmarais, V. Groza, and M. Bolic, “ASIPs for Artificial Neural Networks.” IEEE International Symposium Applied Computational Intelligence and Informatics (SACI), pp. 529-533, May 2011.

[34] S. Nissen, “Implementation of a Fast Artificial Neural Network Library (FANN).” Report, Department of Computer Science University of Copenhagen (DIKU), Oct. 2003.

[35] R. Lopez, OpenNN: An Open Source Neural Networks C++ Library [Online]. Available: http://www.opennn.net/download/index.html

[36] G. Bradski, “The OpenCV Library.” Doctor Dobbs Journal, vol. 25, no. 11, pp. 120-126, Nov. 2000.

[37] M. Abadi et al., “TensorFlow: Large-Scale Machine Learning Heterogenous Systems.” Software available from https://www.tensorflow.org/ , 2015.

[38] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga and A. Lerer, “Automatic Differentiation in Pytorch.” Annual Conference on Neural Information Processing Systems, 2017.

[39] Nvidia, C. U. D. A. “Programming guide.” 2010.

[40] MathWorks, 2011, Neural Network Toolbox - MATLAB. [Online]. Available: http://www.mathworks.com/products/neuralnet/

[41] T. Chen, I. Goodfellow and J. Shlens, “Net2net: Accelerating Learning via Knowledge Transfer.” arXiv preprint: 1511.05641. Nov. 2015.

[42] M. Courbariaux, Y. Bengio and J. P. David, “Training Deep Neural Networks with Low Precision Multiplications.” arXiv preprint: 1412.7024. Dec. 2014.

[43] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz and W. J. Dally, “EIE: Efficient Inference Engine on Compressed Deep Neural Network.” IEEE International Symposium on Computer Architecture, pp. 243-254, Jun. 2016.

[44] Y. Wang, L. Xia, T. Tang, B. Li, S. Yao, M. Cheng and H. Yang, “Low Power Convolutional Neural Networks on a Chip.” IEEE International Symposium Circuits and Systems (ISCAS), pp. 129-132, May 2016.

[45] Y. H. Chen, T. Krishna, J. S. Emer and V. Sze, “Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks.” IEEE Journal of Solid-State Circuits, Nov. 2016.

[46] Y. H. Chen, J. S. Emer and V. Sze, “Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks.” ACM/IEEE International Symposium on Computer Architecture (ISCA), vol. 44, no. 3, pp. 367-379, Jun. 2016.

[47] R. Andri, L. Cavigelli, D. Rossi and L. Benini, “YodaNN: An Ultra-Low Power Convolutional Neural Network Accelerator Based on Binary Weights.” IEEE Computer Society Annual Symposium on VLSI (IS-VLSI), pp. 236-241, Jul. 2016.

[48] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen and O. Temam, “Diannao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning.” ACM Sigplan Notices, vol. 49, no. 4, pp. 269-284, Feb. 2014.

[49] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun and O. Temam O, “Dadiannao: A Machine-Learning Supercomputer.” IEEE/ACM International Symposium on Microarchitecture, pp. 609-622, Dec. 2014.

[50] T. Luo, S. Liu, L. Li, Y. Wang, S. Zhang, T. Chen, Z. Xu, O. Temam O and Y. Chenm, “Dadiannao: A Neural Network Supercomputer.” IEEE Transactions on Computers, vol. 60, no. 1, pp. 73-88, May 2016.

[51] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra and H. Esmaeilzadeh, “Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Networks.” IEEE Annual International Symposium on Computer Architecture (ISCA), pp. 764-775, Jun. 2018.

[52] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers and R. Boyle, “In-Datacenter Performance Analysis of a Tensor Processing Unit.” IEEE International Symposium on Computer Architecture (ISCA), pp. 1-12, Jun. 2017

[53] J. Dean and U. Holzle, Build and Train Machine Learning Models on our New Google Cloud TPUs. May 2017. https://blog.google/products/google-cloud/google-cloud-offer-tpus-machine-learning/.

[54] F. O. Zamarano, J. M. Jerez, D. U. Munoz, R. Baena and L. Franco, “Efficient Implementation of the Backpropagation Algorithm in FPGAs and Microcontrollers.” IEEE Transactions on Neural Networks and Learning Systems, vol. 27, no. 9, pp. 1840-1850, Sep. 2016.

[55] Q. Yu, C. Wang, X. Ma, X. Li and X. Zhou, “A Deep Learning Prediction Process Accelerator Based FPGA.” IEEE/ACM International Symposium Cluster, Cloud and Grid Computing, pp. 1159-1162, May 2015.

[56] J. Park and W. Sung, “FPGA Based Implementation of Deep Neural Networks using on-chip Memory Only.” IEEE International Conference Acoustics, Speech and Signal Processing, pp. 1011-1015, Mar. 2016.

[57] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie and X. Zhou, “DLAU: A Scalable Deep Learning Accelerator Unit on FPGA.” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Jul. 2016.

[58] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao and J. Cong. “Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks." ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 161-170, Feb. 2015.

[59] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo and J. Cong, “Energy-Efficient CNN Implementation on a Deeply Pipelined FPGA Cluster.” ACM International Symposium on Low Power Electronics and Design, pp. 326-331, Aug. 2016.

[60] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou and L. Wang, “A High Performance FPGA-Based Accelerator for Large-Scale Convolutional Neural Networks.” IEEE International Conference Field Programmable Logic and Applications (FPL), pp. 1-9, Aug. 2016.

[61] Y. Ma, N. Suda, Y. Cao, J. S. Seo and S. Vrudhula, “Scalable and Modularized RTL Compilation of Convolutional Neural Networks onto FPGA.” IEEE International Conference Field Programmable Logic and Applications (FPL), pp. 1-8, Aug. 2016.

[62] L. Lu, Y. Liang, Q. Xiao and S. Yan, “Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs.” IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 101-108, Apr. 2017.

[63] C. Zhang and V. Prasanna, “Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System.” ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 35-44, Feb. 2017.

[64] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, Y. Bengio, “Binarized Neural Networks.” Advances in Neural Information Processing Systems, pp. 4107-4115, 2016.

[65] 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.” arXiv preprint:1612.07119. Dec. 2016.

[66] N. J. Fraser, Y. Umuroglu, G. Gambardella, M. Blott, P. Leong, M. Jahre and K. Vissers. “Scaling Binarized Neural Networks on Reconfigurable Logic.” ACM Workshop on Parallel

Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms, pp. 25-30, Jan. 2017.

[67] H. Alemdar, V. Leroy, A. Prost-Boucle F. and Petrot, “Ternary Neural Networks for Resource-Efficient AI Applications.” IEEE International Joint Conference Neural Networks, pp. 2547-2554, May 2017.

[68] S. I. Venieris and C. S. Bouganis CS, “fpgaConvNet: A framework for Mapping Convolutional Neural Networks on FPGAs.” IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 40-47, May 2016.

[69] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang and J. Cong, “Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs.” ACM Design Automation Conference, pp. 29-35, Jun. 2017.

[70] D.H. Noronha, B. Salehpour and S. J. Wilton, “LeFlow: Enabling Flexible FPGA High-Level Synthesis of tensorflow deep neural networks.” International Workshop on FPGAs for Software Programmers, pp. 1-8, Aug. 2018.

[71] D. Hunter, H. Yu, M. S. Pukish, J. Kolbusz and B. M. Wilamowski, “Selection of proper neural network sizes and architectures-a comparative study.” IEEE Transactions on Industrial Informatics, pp. 228-240, 2012.

[72] B. Zamanlooy and M. Mirhassani, “Efficient VLSI Implementation of Neural Networks with Hyperbolic Tangent Activation Function.” IEEE Transactions on Very Large Scale Integration Systems, pp. 39-48, 2014.

[73] A. Armato, L. Fanucci, E. P. Scilingo and D. De Rossi, “Low-Error Digital Hardware Implementation of Artificial Neuron Activation Functions and Their Derivative.” Microprocessors and Microsystems, pp. 557-567, 2011.

[74] M. Zhang, S. Vassiliadis and J. G. Delgado-Frias, “Sigmoid Generators for Neural Computing using Piecewise Approximations.” IEEE Transactions on Computers, pp. 1045-1049, 1996.

[75] K. Leboeuf, A. H. Namin, R. Muscedere, H. Wu and M. Ahmadi, “High Speed VLSI Implementation of the Hyperbolic Tangent Sigmoid Function.” IEEE International Conference on Convergence and Hybrid Information Technology, pp. 1070-1073, 2008.

[76] A. H. Namin, K. Leboeuf, R. Muscedere, H. Wu and M. Ahmadi, “Efficient Hardware Implementation of the Hyperbolic Tangent Sigmoid Function.” IEEE International Symposium on Circuits and Systems, pp. 2117-2120, 2009.

[77] P. K. Meher, “An Optimized Lookup-Table for the Evaluation of Sigmoid Function for Artificial Neural Networks.” IEEE International Conference VLSI and System-on-Chip, pp. 91-95, 2010.

[78] K. Ugur, A. Alshin, E. Alshina, F. Bossen, W. J. Han and J. H. Park, “Motion Compensated Prediction and Interpolation Filter Design in H. 265/HEVC.” IEEE Journal of Selected Topics in Signal Processing, pp. 946-956, 2013.

[79] V. Sze, Y. H. Chen, T. J. Yang and J. Emer, “Efficient Processing of Deep Neural Networks: A Tutorial and Survey.” Proceedings of the IEEE, Mar. 2017.

[80] G. Lacey, G. W. Taylor and S. Areibi, “Deep Learning on FPGAs: Past, Present, and Future.” arXiv preprint:1602.04283, Feb. 2016.

[81] V. Sze, Y. H. Chen, J. Emer, A. Suleiman and Z. Zhang, “Hardware for Machine Learning: Challenges and Opportunities.” arXiv preprint:1612.07625, Dec. 2016.

[82] A. Brant and G. G. Lemieux, “ZUMA: An Open FPGA Overlay Architecture." IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 93-96, Apr. 2012.

[83] S. Gupta, A. Agrawal, K. Gopalakrishnan and P. Narayanan, “Deep Learning with Limited Numerical Precision.” International Conference on Machine Learning, Aug. 2015

[84] M. S. Razlighi, M. Imani, F. Koushanfar and T. Rosing, “LookNN: Neural Network with No Multiplication.” IEEE Design, Automation & Test in Europe Conference & Exhibition, pp. 1779-1784, Mar. 2017.

[85] H. K. Kwan and C. Z. Tang, “Multiplierless Multilayer Feedforward Neural Networks.” IEEE Midwest Symposium on Circuits and Systems, pp. 1085-1088, Aug. 1993.

[86] H. Tann, S. Hashemi, R. Bahar and S. Reda, “Hardware-Software Codesign of Accurate, Multiplier-free Deep Neural Networks.” IEEE/ACM Annual Design Automation Conference, pp. 1-6, Jun. 2017.

[87] D. A. Gudovskiy and L. Rigazio, “ShiftCNN: Generalized Low-Precision Architecture for Inference of Convolutional Neural Networks.” arXiv preprint:1706.02393, Jun. 2017.

[88] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition." in Proceedings of the IEEE, pp. 2278-2324, 1998.

[89] M. Rastegari, V. Ordonez, J. Redmon and A. Farhadi, "XNOR-NET: Imagenet Classification using Binary Convolutional Neural Networks." Springer European Conference on Computer Vision, pp. 525-542, Oct. 2016.

[90] K. Hornik, "Approximation Capabilities of Multilayer Feedforward Networks." Neural Networks. vol. 4, no. 2, pp. 251-257, Jan. 1991.

[91] D. D. Lin and S. S. Talathi, "Overcoming Challenges in Fixed Point Training of Deep Convolutional Networks." arXiv preprint arXiv:1607.02241. Jul. 2016.

[92] UCI Machine Learning Repository. http://archive.ics.uci.edu/ml/datasets/ISOLET.

[93] UCI Machine Learning Repository.
https://archive.ics.uci.edu/ml/datasets/UJIIndoorLoc.

[94] UCI Machine Learning Repository.
https://archive.ics.uci.edu/ml/datasets/Daily+and+Sports+Activities.

[95] Xilinx, AXI Reference Guide.

[96] L. Deng, P. Jiao, J. Pei, Z. Wu and G. Li, "Gated XNOR Networks: Deep Neural Networks with Ternary Weights and Activations under a Unified Discretization Framework." arXiv preprint arXiv:1705.09283, Feb. 2017.

[97] T. Hastie, R. Tibshirani and J. Friedman, The Elements of Statistical Learning. Second Edition. NY: Springer, 2008.

[98] S. Lloyd, "Least Squares Quantization in PCM." IEEE Transactions on Information Theory, vol. 28, pp. 129–137, Mar. 1982.

[99] N. Cristianini and J. Shawe-Taylor, "An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods." Cambridge University Press, Mar. 2000.

[100] L. Breiman, J. Friedman, R. Olshen, and C. Stone, "Classification and Regression Trees." Boca Raton, FL: CRC Press, 1984.

[101] S. B. Akers, "Binary Decision Diagrams." IEEE Transactions on Computers, Jun. 1978.

[102] P. T. Tang, "Table-Lookup Algorithms for Elementary Functions and Their Error Analysis." IEEE Symposium on Computer Arithmetic, Jun. 1991.

[103] Y. Cheng, D. Wang, P. Zhou and T. Zhang. “A Survey of Model Compression and Acceleration for Deep Neural Networks.” arXiv preprint:1710.09282, Oct. 2017.

[104] M. Courbariaux, Y. Bengio and J. P. David, “Binaryconnect: Training Deep Neural Networks with Binary Weights During Propagations.” Advances in Neural Information Processing Systems, pp. 3123-3131, Dec. 2015.

[105] H. Nakahara, T. Fujii and S. Sato, “A Fully Connected Layer Elimination for a Binarized Convolutional Neural Network on an FPGA.” IEEE International Conference on Field Programmable Logic and Applications (FPL), pp. 1-4, Sep. 2017.

[106] 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.” ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 15-24, Feb. 2017.

[107] S. Liang, S. Yin, L. Liu, W. Luk and S. Wei, “FP-BNN: Binarized Neural Network on FPGA.” Neurocomputing. vol. 275, pp. 1072-1086, Jan. 2018.

[108] L. Rokach and O. Z. Maimon, Data Mining with Decision Trees: Theory and Applications. World scientific, 2008.

[109] C. Robert, Machine learning, A Probabilistic Perspective. 2014.

[110] A. C. Lorena, A. C. De Carvalho and J. M. Gama, “A Review on the Combination of Binary Classifiers in Multiclass Problems.” Artificial Intelligence Review, vol. 30, no. 1-4, pp. 1-19, Dec. 2008.

[111] J. R. Struharik, “Implementing Decision Trees in Hardware.” IEEE International Symposium on Intelligent Systems and Informatics, pp. 41-46, Sep. 2011.

[112] J. Furnkranz, D. Gamberger and N. Lavrac, “Foundations of Rule Learning.” Springer Science & Business Media, Nov. 2012.

[113] R. O. Duda, P. E Hart and D. G. Stork, Pattern classification. John Wiley & Sons, Nov. 2012.

[114] A. Krizhevsky and G. Hinton, “Learning Multiple Layers of Features from Tiny Images.” Technical Report, 2009.

[115] Z. H. Zhou and J. Feng, “Deep forest: Towards an Alternative to Deep Neural Networks.” arXiv preprint:1702.08835. Feb. 2017.

[116] P. Kontschieder, M. Fiterau, A. Criminisi and S. B. Rota, “Deep Neural Decision Forests.” IEEE International Conference on Computer Vision, pp. 1467-1475, 2015.

[117] K. Abdelouahab, M. Pelcat, J. Serot and F. Berry, “Accelerating CNN Inference on FPGAs: A Survey.” arXiv preprint:1806.01683, May 2018.

[118] C. Hettinger, T. Christensen, B. Ehlert, J. Humpherys, T. Jarvis and S. Wade, “Forward Thinking: Building and Training Neural Networks One Layer at a Time.” arXiv preprint:1706.02480, Jun. 2017.

[119] L. Breiman, “Random Forests.” Machine Learning, Oct. 2001.

[120] E. Wang, J. J. Davis, R. Zhao, H. C. Ng, X. Niu, W. Luk, P. Y. Cheung and G. A. Constantinides, “Deep Neural Network Approximation for Custom Hardware: Where We’ve Been, Where We’re Going.” arXiv preprint:1901.06955. Jan. 2019.

[121] N. Frosst and G. Hinton, “Distilling a Neural Network into a Soft Decision Tree.” arXiv preprint:1711.09784, Nov. 2017.

[122] Q. Zhang, Y. Yang, Y. N. Wu and S. C. Zhu, “Interpreting CNNs via Decision Trees.” arXiv preprint:1802.00121, Feb. 2018.

[123] K. Miller, C. Hettinger, J. Humpherys, T. Jarvis and D. Kartchner, “Forward Thinking: Building Deep Random Forests.” arXiv preprint:1705.07366, May 2017.

[124] T. Ojala, M. Pietikainen and T. Maenpaa, “Multiresolution Gray-Scale and Rotation Invariant Texture Classification with Local Binary Patterns.” IEEE Transactions on Pattern Analysis  Machine Intelligence, pp. 971-987, Jul. 2002.

[125] P. Ramachandran, B. Zoph and Q. V. Le, “Searching for Activation Functions.“ arXiv preprint arXiv:1710.05941, Oct. 2017.

[126] M. Lichman, “UCI machine learning repository.“ 2013.

[127] Y. J. Qu, and H. Bao-Gang, “Generalized Constraint Neural Network Regression Model Subject to Linear Priors.“ IEEE Transactions on Neural Networks, vol. 22, no. 12, pp. 2447-2459, 2011.

[128] H. K. So and C. Liu, “FPGA Overlays.“ In FPGAs for Software Programmers, Springer Cham., (pp. 285-305), 2016.

[129] Y. Zhou, W. Wang and X. Huang, "FPGA Design for PCANet Deep Learning Network." IEEE International Symposium on Field-Programmable Custom Computing Machines, pp. 232-232, May 2015.

[130] A. Page and T. Mohsenin, "FPGA-based Reduction Techniques for Efficient Deep Neural Network Deployment." IEEE International Symposium on Field-Programmable Custom Computing Machines, pp. 200-200, May 2016.

[131] P. G. Mousouliotis, K. L. Panayiotou, E. G. Tsardoulias, L. P. Petrou and A. L. Symeonidis, "Expanding a Robot's Life: Low Power Object Recognition via FPGA-based DCNN Ddeployment." IEEE International Conference on Modern Circuits and Systems Technologies, pp. 1-4, May 2018.

[132] J. Su, J. Faraone, J. Liu, Y. Zhao, D. B. Thomas, P. H. Leong and P. Y. Cheung, "Redundancy-Reduced MobileNet Acceleration on Reconfigurable Logic for ImageNet Classification." Springer International Symposium on Applied Reconfigurable Computing, pp. 16-28, May 2018.