UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM MICROELETRÔNICA

AUGUSTO ANDRÉ SOUZA BERNDT

Boolean Optimization of Neural Network Circuits Using Signal Probabilities and Approximate Computing Through Constant Propagation

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Microelectronics

Advisor: Prof. Dr. Paulo Francisco Butzen
Coadvisor: Prof. Dr. André Inácio Reis

Porto Alegre
March 2020

CIP — CATALOGING-IN-PUBLICATION

Berndt, Augusto André Souza

Boolean Optimization of Neural Network Circuits Using Signal Probabilities and Approximate Computing Through Constant Propagation / Augusto André Souza Berndt. – Porto Alegre: PGMICRO da UFRGS, 2020.

101 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica, Porto Alegre, BR–RS, 2020. Advisor: Paulo Francisco Butzen; Coadvisor: André Inácio Reis.

1. EDA. 2. Neural network. 3. AND-inverter graph. 4. Boolean optimization. 5. Constant propagation. 6. Approximate computing. I. Butzen, Paulo Francisco. II. Reis, André Inácio. III. Título.

“The only absolute truth is that there are no absolute truths.”

— Paul Karl Feyerabend

AGRADECIMENTOS

Gostaria de agradecer aos meus pais Nara Berndt e Ricardo Berndt que me incentivaram desde cedo a ser uma pessoa estudiosa, investindo seu esforço para minha formação acadêmica e como pessoa. Agradeço por terem desenvolvido meu intelecto de uma pessoa curiosa que busca entender a essência dos acontecimentos ao nosso redor e que envolvem nossa existência. Se não fosse por isto, nunca teria iniciado um projeto de mestrado acadêmico.

Agradeço também meus mentores André Reis e Paulo Butzen por toda sua paciência e tempo investidos comigo e para o desenvolvimento desse projeto. Graças a eles aprendi uma diversidade de assuntos e técnicas na prática durante o desenvolvimento do projeto. Agradeço todo seu conhecimento e orientações que foram transmitidos. Tive uma evolução como pessoa e pesquisador notável ao longo destes dois anos de mestrado, graças a eles.

Agradeço também a minha namorada Bruna Martins e sua família por terem me acolhido como um membro a mais em sua família enquanto residia em Porto Alegre realizando a execução deste mestrado.

Por fim agradeço aos membros da banca, os professores Cristina Meinhardt, José Azambuja e Raphael Brum pela sua participação, críticas e comentários complementando com a finalização deste trabalho.

ABSTRACT

The development of electronic devices has demonstrated amazing capabilities since the introduction of the transistor device. Humanity is more than ever, virtually connected. Information is at the grasp of most current human beings, thanks to the expansion and improvement of integrated circuits. The construction of an integrated circuit usually follows an iterative design flow, and research on this topic is crucial to keep this technology naturally undergoing progress. Although, the implementation of emerging technologies such as artificial intelligence and even the recess that Moore's Law faces are obstacles that researchers face and attempt to overcome by finding suitable solutions to such present problems composing in current technologies. Even though the concept and experimental implementation of artificially intelligent devices is a long craving wish among researchers, it is only in the present days that its actual usage in the real world is happening. The development of integrated circuits jointly with the introduction of *big data* and *internet of things* (IoT) allows for a plausible common ground for the employment of artificially intelligent devices in current commodities. Nonetheless, the research and study on the area also sustain an increase in attention by researchers to achieve better, faster, and less power consuming technologies. On the other hand, neural networks (NNs) specifically tend to be extensively power and area consuming. Companies work around it by using data centers and provide inference for their users. But even still, optimizations seeking more compact or even less power consuming neural networks should be pursued. Integrated circuit designers face a considerable challenge when attempting to implement hardware with a certain limited budget, like smartphones or IoT devices, that can process current neural networks. This work focuses on optimizing circuits representing neural networks in the form of AND-inverter graphs (AIGs). The optimization is done by analyzing the training set of the neural network to find constant bit values among the AIG nodes. The constant values are then propagated through the AIG, which results in removing unnecessary nodes. Furthermore, a trade-off between neural network accuracy and its reduction due to constant propagation is investigated by replacing with constants those nodes that are likely to be zero or one. The experimental results show a significant reduction in circuit size with negligible loss in accuracy. For example, for a neural network, we were able to reduce its size to 63.3% and its depth to 82.0% from their original values with no reduction in accuracy, or even 59.1% in size and 77.9% in depth from original values while loosing only 1% in accuracy.

Keywords: EDA. Neural network. AND-inverter graph. Boolean optimization. Constant propagation. Approximate computing.

Otimização Booliana de Circuitos de Redes Neurais Usando Probabilidade de Sinais e Computação Aproximada Através da Propagação de Constantes

RESUMO

O desenvolvimento de dispositivos eletrônicos tem demonstrado capacidades surpreendentes desde a introdução do dispositivo transistor. A humanidade está mais do que nunca virtualmente conectada, a informação está ao alcance da maioria dos seres humanos atuais graças a expansão e melhorias de circuitos integrados. A construção de um circuito integrado geralmente segue um projeto iterativo e pesquisas neste tópico são cruciais para mantê-la naturalmente progredindo. No entanto, a implementação de tecnologias emergentes como inteligência artificial e até o recesso que a Lei de Moore enfrenta são obstáculos que pesquisadores enfrentam e tentam superar através da procura soluções aplicáveis para estes problemas atuais. Por mais que o conceito e a implementação de dispositivos compostos de inteligência é um desejo almejado por pesquisadores a muito tempo, somente nos dia atuais que sua utilização está realmente acontecendo no mundo atual. O desenvolvimento de dispositivos eletrônicos juntamente com a introdução de *big data* e da *internet das coisas* (IoT) proporcionam um ambiente saudável para a concepção de dispositivos compostos de inteligencia artificial em *commodities* atuais. Não obstante o estudo e pesquisa na área recebe um aumento na atenção dada por pesquisadores para atingir tecnologias melhores, mais rápidas e que consomem meno energia. No entanto, redes neurais especificamente tendem a consumir muita energia e exigir muita área de implementação, companhias industriais improvisam soluções utilizando *data centers* para providenciar a capacidade de inferência para seus usuários. Mesmo assim, otimizações que buscam redes neurais mais compactas ou que utilizem menos energia devem ser almejadas. Projetistas de circuitos integrados enfrentam um gigantesco desafio enquanto tentam implementar um *hardware* com orçamento limitado, como *smartphones* ou dispositivos IoT que são capazes de processar redes neurais atuais. Este trabalho foca na otimização de circuitos que representam redes neurais na forma de grafos AND-inversores (AIG). A otimização é feita analisando o conjunto de treinamento da rede neural para encontrar *bits* constantes nos nodos do AIG. Os valores constantes são então propagados pelo AIG, no qual resulta em remover nodos desnecessários. Além disso, o comprometimento entre a precisão da redes neurais e a sua redução é investigado devido a propagação de constantes através da reposição de nodos que tendem a serem constantes zero ou um. Os resultados experimentais mostram uma significante redução em tamanho e profundidade do circuito com uma perda insignificante de precisão. Por exemplo nós conseguimos reduzir uma rede neural em tamanho para 63.3% e profundidade para 82.0% dos seus valores originais sem nenhuma perda de precisão, ou até 59.1% de tamanho e 77.9% de profundidade dos valores originais enquanto a rede perdeu somente 1% de precisão.

Palavras-chave: EDA, grafo AND-inversor, redes neurais, computação aproximada, otimização booleana.

LIST OF ABBREVIATIONS AND ACRONYMS

AIG	AND-Inverter Graph
AI	Artificial Intelligence
AN	All Nodes
ASIC	Application-Specific Integrated Circuit
BDD	Binary Decision Diagram
CEC	Combinational Equivalence Check
CIFAR	Canadian Institute For Advanced Research
CNN	Convolutional Neural Network
DAG	Directed Acyclic Graph
EDA	Electronic Design Automation
FPGA	Field Programmable Gate Array
FB-LD	Functional Based on Logic Depth
FB-NPD	Functional Based on Number of Nodes Per Level
GPU	Graphics Processing Unit
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
IoT	Internet of Things
LD	Logic Depth
LUT	Look-Up Table
MAC	Multiply Accumulate
MIG	Majority-Inverter Graph
MNIST	Modified National Institute of Standards and Technology
NN	Neural Network
OPI	Only Primary Inputs
PI	Primary Input

PO	Primary Output
ReLu	Rectified Linear Unit
RDF	Random Decision Forest
SOP	Sum-of-Products
TH	Threshold Probability Variable
VHDL	Very High Speed Integrated Circuit Hardware Description Language
XAIG	XOR-AND-Inverter Graph

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Artificial intelligence (AI) is the study behind the capability that an entity has to understand and solve problems in the real world. Its objective is to perceive its environment and react upon it, usually learning about the environment in an attempt to maximize its chances of achieving a goal (RUSSELL; NORVIG, 2016). *Neural networks* (NNs) present themselves as an AI entity, which are non-linear mathematical models inspired by biological neural networks (NIELSEN, 2015; HAYKIN, 1994). This technology is capable of receiving information, processing this information, and giving a response as output. Neural Networks are usually used to recognize patterns and also on classification problems. Classification by image recognition has shown itself a problem that NNs excel at solving, sometimes even surpassing human capabilities (RUSSAKOVSKY et al., 2015; GYSEL, 2016). Although this high expertise in solving problems comes with the cost of high demand for computational capacity and effectiveness (XU et al., 2018; MISRA; SAHA, 2010). The emerging of *internet of things* (IoT) devices and smartphones are leveraging the demand for AI inferences in everyday life.

A NN is an interconnected group of neurons, inspired by the structure of a living being's brain. Each neuron inside a NN may form one or multiple weighted synapses, and the neurons are grouped in layers. The first layer being the input layer, followed by an arbitrary number of hidden layers and an output layer that gives the NN's solution to the presented input. NN implementations go through a training phase that covers thousands of examples to learn from. Each iteration of the learning phase processes instances of the training set and refines the NN parameters with a backpropagation process based on the stimulation presented by the training set, responsible for finding intricate structures in the training set. The same set of data is processed multiple times as the network is continually refined. The type in which the learning occurs is defined by how the NN's *parameters* adjust during the learning stage. The idea is to search for a good possible solution by iteratively modifying the NN's parameters, usually a *gradient descent* approach is used to search for a local minimum about the NN's loss function (HAYKIN, 1994; NIELSEN, 2015; BISHOP et al., 1995).

1.1 Problem Definition

NNs are mostly implemented as software (WEI et al., 2018; BAI et al., 2018; ZHANG et al., 2018; HU; SHEN; SUN, 2018). Specifically, NNs presented in the *ImageNet Large Scale Visual Recognition Challenge* (ILSVRC) challenge have shown an amazing development of knowledge to infer on images. Although this inference capability is software-based and comes with a high processing requirement, in some of the challenges processing budgets are imposed, making a closer correlation with the requirements for the implementation and usage of NNs as hardware (RUSSAKOVSKY et al., 2015). That much improvement on the NN's capabilities also comes with the price of high processing and power consumption demand. Even though there are cloud-based servers that offer NN's inference online and companies make use of them to perform the highly demanded inference and training processes needed for their AI-based commodities (LECUN, 2019). So many requirements delay the implementation of NNs as hardware, for example, their implementation in mobile systems, since an excessive amount of components and energy power is required, which are not available. Nonetheless, the device's edge inference capabilities should reduce the processing and power requirements to execute and employ NNs on present mobile and IoT devices, which have low computational capabilities and require low power consumption. Research on this topic is required, so this may be possible.

Neuromorphic hardware already implemented vary from analog, digital and optical signal types (XU et al., 2018; CALVERT; MARINOV, 2000; HAMERLY et al., 2019) and platforms such as FPGAs, GPUs and ASIC (LECUN, 2019; NIELSEN, 2015; XU et al., 2018). All of them have a different set of flaws (like accuracy loss, large space, and power usage or low processing speed) and delays their commercial practice (MISRA; SAHA, 2010; XU et al., 2018). As shown by (XU et al., 2018), there is a gap between the required processing power by NNs implemented as software in academia and the development of hardware to process such NNs, which lags in processing capability compared to the NNs implemented in software. Also, EDA tools have a hard time to develop an efficient synthesis for NNs. Neuromorphic hardware requires a large silicon area due to a large number of logic gates and a high number of connections between layers, which introduces routing congestion (ARDAKANI et al., 2017). This is why NNs are still lagging on market nodes such as smartphones, IoT devices, and edge devices. NNs are already being used by researchers to execute logic synthesis (HAASWIJK et al., 2018). However, it is yet to be discovered how to efficiently reproduce logic synthesis steps to design an NN circuit (HE; SUN, 2015; QIU et al., 2016; ARDAKANI et al., 2017).

In the work presented herein, we propose four different variants to reduce the size and depth of NNs when represented in a boolean format, more precisely NNs represented with the *AND-inverter graph* (AIG) data structure. During the project execution of an integrated circuit, its project goes through iterative processes, and the AIG data structure is actively used in *logic synthesis* stages such as technology mapping and independent technology optimizations (CHANDRASEKHARAN et al., 2016; MISHCHENKO; CHATTERJEE; BRAYTON, 2006; WANG et al., 2017).To expose the unusual number of building blocks required by NNs, we can see that, for example, the NN reported by (COATES et al., 2013) had around 1 million neurons and 10 billion synaptic weights. The number of *multiply accumulate* (MAC) units required to implement an NN in hardware does not reflect the size in which researchers are used to working with to build electronic devices. For example, in the IWLS'19 contest, its largest AIG had almost 1.8 million AND nodes. We would require various amounts of AND nodes to represent each MAC for the NN design. The machine learning models represented in AIG used in our work range from 45 to 80 million AND nodes, at least 25 times larger than the largest AIG in the IWLS'19 contest (International Workshop on Logic and Synthesis, 2019). Without saying that the NNs used in this work has only 333 thousand parameters (the trainable components of a NN are usually known as *parameters* such as the weight and bias, while the non-trainable components are called *hyperparameters* such as the number of hidden layers in the NN), in comparison to the winner of ILSVRC ImageNet 2016 contest the ResNet which had 60 million parameters (HE et al., 2016). This exposes the challenge it is to have a boolean representation for NNs with the current knowledge on the field. The NNs used in the present work show themselves with enormous sizes when passed to the AIG format, being not even close to what the logic synthesis research community are used to work with.

1.2 Proposal and Contributions

To improve the design of hardware for NNs, we work on its boolean representation and adopt an approximate computing strategy. We insert small acceptable errors to decrease the project's cost. This is a common approach to the design of arithmetic, energy-efficient, or quality configurable circuits (HAN; ORSHANSKY, 2013;

WU; QIAN, 2016; YAO et al., 2017; VENKATARAMANI; ROY; RAGHUNATHAN, 2013; CHANDRASEKHARAN et al., 2016; WANG et al., 2017). In our work, we use the training set to extract the probabilities of all the nodes in the circuit. We might then consider signals with high probability values to be constants, which allows us to apply simplifications on the circuit. This process is done on already trained NN circuits represented as AIGs.

We propose an analysis to evaluate the behavior of the NN when its size is reduced based on constant and nearly constant signal propagation. The simplification under constants is a technique already known (MICHELI, 1994), although not every circumstance allows its use as the circuit's input set is not always known. Also, the problem's complexity grows exponentially with the circuit's input size. NNs are good candidates to make use of such technique because the training set exposes a considerable amount of its possible input vectors.

Altogether four different ways are presented to define which nodes will be established as constants. Furthermore, we propose two different metrics to compare the trade-off in accuracy loss versus size and depth among the methods proposed. Our contributions are listed as follows.

- We implement a probability calculation of signals in an NN represented as a circuit in its AIG format by simulation and directed by the NN's training dataset.
- We define nodes to be set as constants based on the nodes' probabilities in which surpass a certain *threshold*. The AIG may be simplified after the definitions of such constants.
- We propose four different manners to define a node's threshold probability, the idea behind them is to search for the unimportant nodes and set them as constants while leaving the important ones untouched.

  1. At first, a fresh approach where only the AIG's primary inputs may be turned as constants is proposed. Even a naive concept such as this one, can considerably simplify the NN's size and depth. This method was previously published in (BERNDT et al., 2019).
  2. A second approach is proposed where we may turn into constants any node of the AIG with a fixed threshold probability value. In this method, the threshold is fixed, meaning its the same for all nodes in the AIG.
  3. A third and fourth method is presented in which variable thresholds are used,

meaning that the nodes in the AIG have different thresholds. These thresholds are calculated based on the node's characteristics, such as its *logic depth* (LD) or the *amount of nodes* present in an equal logic depth.

We apply our simplification methods on two NNs and a *Random Decision Forest* (RDF) model, the RDF model is used mainly to show that our approach is not structurally constrained. The four methods proposed presented different behavior on each of the machine learning models. Some analyses are made at the last chapter of the present work to expose the worthwhile trade-off from accuracy loss versus size and depth reduction, by making a comparison among the methods concerning the utilized experimental models. We were able to achieve a reduction of 18% in depth and 27% in size with no reduction in accuracy for a NN model and 22.1% reduction in depth and 40.9% in size with the cost of only 1% reduction in accuracy for the same NN, for example.

1.3 Dissertation Outline

The rest of this dissertation is organized as follows. The next Chapter 2 presents some background and preliminary information on subjects that are essential for the proper understanding of the contributions of this work, such as details about a NN design, AIG simplification particularities and the conversion of a NN to a boolean representation. Chapter 3 discusses the experimental machine learning models used to evaluate our contributions and present our contributions based on optimization for NNs in an AIG representation. Afterward, Chapter 4 presents the results achieved with experiments realized on top of the machine learning models with the usage of the proposed methods. Analyses are made for the performance achieved with each different method and each type of machine learning model with the results presented by the experiments realized. We attempt to analyze what is the trade-off for size and depth reduction versus accuracy loss for each situation. In Chapter 5, a conclusion is presented enclosing the work presented herein.

2 BACKGROUND

This chapter describes the necessary background on topics that are fundamental to perceive and understand the work presented herein properly. We begin by explaining the data structure used for the reduction of NN circuits in their logical representation. Next, we briefly expose the idea behind the probability of elements in a circuit. Afterward, we explain how the reduction of logic circuits is made with the presence of constant variables. Afterward, it is discussed about the structure and learning process of a NN. Next, we present an overview of the bibliography concerning the implementation and optimizations of neuromorphic hardware. In the last section of this chapter, we present the issues and requirements for the implementation of a boolean representation of a NN.

2.1 Logic Circuits

The manufacture and design of integrated chips are based on iteratively compiling, optimizing, and verifying the hardware description language of a circuit to achieve its physical representation layout. These processes are usually summarized in a design flow, in which chip designers follow to coordinate the synthesis of an error-free integrated chip. A general design flow is divided into categories: high-level synthesis, logic synthesis, physical synthesis, and verification (WESTE; HARRIS, 2015; MATOS; CARRABINA; REIS, 2018; POSSANI et al., 2018).

When progressing along with the design flow, there are intermediate steps to follow, such as the *logical synthesis* step that is responsible for making technology-independent optimizations and technology mapping. Following logic synthesis, it is performed the *physical synthesis* step, which is responsible for placing and routing of logic gates. It is required that the circuit's specification is already mapped with respect to a library of cells. The library specifies the logic and electric characteristics for each type of logic gates. A good mapping algorithm is one able to find a combination of logic gates that can precisely represent the circuit's logic with the least amount of area, power dissipation, and delay as possible, these may vary depending on the synthesis objectives. The verification steps are the ones responsible for estimating such metrics and analyzing the quality of the synthesized circuit along each step of the design flow, including the logic synthesis steps. To find a good combination of logic gates available in a library is a non-deterministic problem since there may be different combinations of logic gates that can map into the same boolean logic (MATOS; CARRABINA; REIS, 2018; MISHCHENKO et al., 2015).

Another process executed during logic synthesis is the one called technology-independent optimization, or multi-level logic optimization as its also called, in which optimizations are applied before technology mapping attempting to reduce the size and depth of the structure representing the circuit while maintaining its logic function, regardless of technology mapping and physical characteristics (BRAYTON et al., 1987; MISHCHENKO; CHATTERJEE; BRAYTON, 2006; YU; CIESIELSKI; MISHCHENKO, 2018; POSSANI et al., 2018). The reduction in size and depth of the logic representation of a circuit is proven to be correlated with the circuit's area and delay, respectively (BRAYTON; HACHTEL; SANGIOVANNI-VINCENTELLI, 1990; MICHELI, 1994). More precisely, for the AIG data structure that is used in this work, the size is determined by the number of nodes in the AIG, and its depth is determined by the number of nodes in the longest path going from a primary input to a primary output. The technology-independent optimization steps usually takes a considerable amount of time to execute and are of great importance, since the logic representation synthesized during logic synthesis will have a direct impact on the *Quality of Result* in later steps of the design flow, like technology mapping (CHATTERJEE et al., 2006; LIU; ZHANG, 2017) and also in placing and routing steps (WANG; CHANG; CHENG, 2009).

2.1.1 Data Structures to Represent Logic Circuits

In this subsection, we briefly present some of the data structures used to represent logic circuits. A circuit may be structured as a boolean function, and there are some different data structures able to depict such functions.

2.1.1.1 Binary Decision Diagram

There are several types of data structures that represent logic circuits used during logic synthesis steps. For example, the binary decision diagram (BDD) is a rooted Directed Acyclic Graph (DAG); in other words, it is a directed graph without the presence of cycle paths within it. It represents a set of binary decisions converging in a final decision that may be true or false. These decisions would be represented by the graph's vertices and the edges the path to follow after a decision is made. In (BRYANT, 1986) it was introduced the ability to make canonical representations with BDDs. However, this data structure grows exponentially in size with relation to the number of variables, being an impractical approach a lot of the times (MICHELI, 1994; WANG; CHANG; CHENG, 2009).

2.1.1.2 Boolean Networks

A *boolean network* is also a DAG structure able to represent multi-level logic circuits, although its vertices are not decisional ones, they represent the circuit's logic itself. A boolean network is composed of three types of vertices (also called nodes): *primary inputs* (PIs), which have no incoming edges, *primary outputs* (POs) which have no outgoing edges and internal nodes which compose the circuit's internal structure. Each internal node may also be considered an intermediate or local function, which is associated with a circuit's segment (MATOS; CARRABINA; REIS, 2018; MICHELI, 1994). Next, we shall briefly expose some examples of boolean networks. The last is the AIG data structure in which we make use of and explain in deeper detail.

2.1.1.3 Sum-of-Products

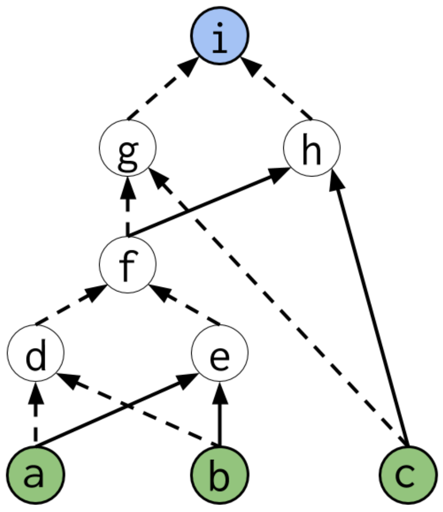
The *sum-of-products* (SOP) is a form of boolean formula, also known as *disjunctive normal form*, composed of disjunctions (sums) of conjunctions (products) of literals. In (BRAYTON et al., 1987), SOPs are used to make an area estimation of the circuit's representation, in (BARTLETT et al., 1987) SOPs are not only used to estimate area but also to apply logic manipulations. Even though any boolean formula may be represented in the SOP format, its representation is not canonical and heterogeneous, reducing its practical use in logic synthesis.

2.1.1.4 AND-Inverter Graph

On the other hand, a homogeneous representation of logic circuits is more attractive, since it makes for easier manipulation of internal nodes. Homogeneous boolean networks examples of data structures would be: AND-inverter graphs, *XOR-AND-inverter graphs* (XAIG) or even *majority-inverter graphs* (MIG). These data structures can reproduce any possible logic under their primitive operations (MATOS; CARRABINA; REIS, 2018; Soeken et al., 2016). The XAIG would be an extension to the AIG structure with the possible usage of XOR nodes, in the work (Háleček; Fišer; Schmidt, 2017) the XAIG structure shows some improvements on synthesis concerning area and delay with some benchmark circuits, although with inferior results for other circuits when compared to the AIG structure. In the work (Amarú; Gaillardon; De Micheli, 2016) the MIG structure was introduced, where it achieved some minor, but still relevant, reductions in delay, area and power with experimental results when compared to the conventional AIG structure. In the present work, we make use of the AIG data structure and it shall be further explained.

The usage of AND-inverter graphs for logic synthesis dates back to the '60s, with the works (HELLERMAN, 1963; DARRINGER et al., 1981), even though the community still acclaims the AIG data structure and its variants to its usage in technology mapping and multi-level logic optimizations. An AND-inverter graph is a directed acyclic graph composed of primary inputs, primary outputs, and two-input AND nodes. All internal nodes in an AIG are AND nodes with exactly two *fanin* inputs and an arbitrary number of *fanout* outputs. Any AND node may be established as a PO. Direct or negated edges connect the nodes. Continuous lines represent a direct edge, while dotted lines represent the negated ones (YU; CIESIELSKI; MISHCHENKO, 2018; MISHCHENKO; CHATTERJEE; BRAYTON, 2006).

Figure 2.1: XOR in AIG representation.



Source: The Author.

Figure 2.1 shows an AIG representation of a three-input XOR logic function, composed of six AND nodes in white, three PI nodes in green and one PO node in blue. Since an AIG is built with AND and NOTs (negated edges), they can represent any logic circuit. During the execution of the Design Flow, the logic synthesis stage consists of converting a high-level description of a design into an optimized gate-level representation. The AIG data structure is broadly used in such steps, mainly to apply technology-independent optimizations and technology mapping on circuits (YU; CIESIELSKI; MISHCHENKO, 2018; MISHCHENKO; CHATTERJEE; BRAYTON, 2006). The ABC academic logic synthesis CAD tool that offers a diversified range of operations able to optimize an AIG representation of a circuit or even execute a technology mapping if a library is provided (Berkeley Logic Synthesis and Verification Group, 2019).

The main advantages of the AIG structure would be: (1) its homogeneous characteristic, meaning that it is easier for logic manipulations. (2) its structure is easily compressed, the AIGER format may be used to do so, where a binary compression is applied to the AIG description, taking less memory to store the AIG (BIERE, 2007) such representation is used on the proposed work. And (3) the number of nodes and the logic depth in an AIG is correlated with the final circuit's area and delay respectively (BRAYTON; HACHTEL; SANGIOVANNI-VINCENTELLI, 1990; MICHELI, 1994). In this work, we make use of the AIG data structure to implement and experiment on our proposed methods to apply technology-independent optimizations oriented for NNs.

2.1.2 Nodes Probabilities

In this session, we describe a known topic, which is the probability of nodes in an AIG. Such an idea of defining probabilities for a circuit's components is commonly used on the logic gate level of abstraction to estimate static power dissipation, reliability, and aging metrics of a circuit (FRANCO et al., 2008; FLAQUER et al., 2010). A signal refers to the binary information that is being passed through a node or nodes in an AIG. The probability of a boolean node is the likelihood that this node will have certain boolean value when the circuit is operating, and signals are passing through it. A boolean variable or node has only two possible values, for example a boolean variable f = {true, false} or even {1,0}. If for example, we had the XOR AIG from Figure 2.1 on an operating circuit and we were able to count each time the node f was set to *true*, we could divide this counter by the total number of times an input vector of signals was applied to the circuit. We would have the signal probability for such AND node to be *true*, denoted by $P(f = 1)$. Equation 2.1 demonstrates the function to calculate the probability of node f to be *true* and Equation 2.2 demonstrates this nodes probability of to be *false*, denoted by

$P(f = 0)$.

$$P(f = 1) = \frac{Number\, of\, Operations\, f = true}{Total\, Operations} \tag{2.1}$$

$$P(f = 0) = \frac{Number\, of\, Operations\, f = false}{Total\, Operations} \tag{2.2}$$

On real-world verification of circuits, the designers almost in every situation can't apply the process just explained of simulating the circuit for known combinations of input vectors. Then, a naive way to calculate the probabilities in a circuit is to process all the possible combinations of input vectors for such a circuit. Although this approach is limited, since its complexity grows exponentially for the number of primary inputs, being viable only with tiny circuits. This is true not only for AIGs but also for any other way to represent and simulate a circuit.

As all the possibles outcomes of a signal are only two possible values, we may state equations 2.3 and 2.4 complement each other, meaning that if the probability of a node being *true* is known its probability of being *false* may also be calculated and vice versa.

$$P(f = 1) = 1 - P(f = 0) \tag{2.3}$$

$$P(f = 0) = 1 - P(f = 1) \tag{2.4}$$

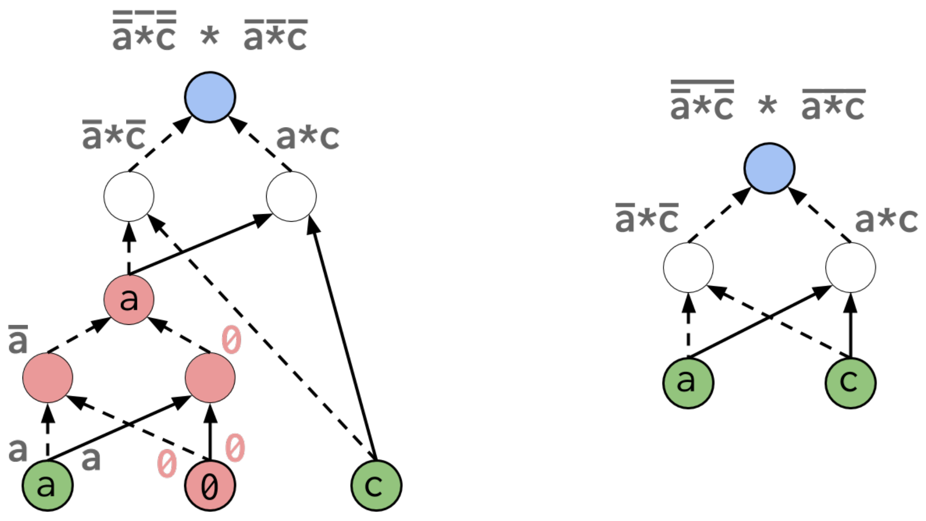
Such a task of calculating the probability of all nodes in a logic circuit or even in an AIG representation is not trivial. It is yet to be discovered an algorithm that can calculate the exact value of signal probabilities in a logic circuit, so far heuristics have been used to do so (FLAQUER et al., 2010; ANGLADA et al., 2016). On the other hand, with the fact that a training dataset supplements a NN trained with supervised learning, it may be used to calculate the probability of internal nodes in an AIG that implements a NN. When we calculate the probability of nodes, we would like to take into consideration all possible input vectors. On the other hand, if we use the NN's training set, we are not calculating such probability based on the whole set of possible combinations of inputs. However, a calculation based on the behavior denoted by the training set, which usually has a large number of items. In other words, the training set allows the nodes probabilities calculation based on the input pattern behavior of the problem to be classified or solved by the NN.

2.1.3 AIG Simplification Under Constants

In this section, we shall demonstrate how a circuit in an AIG format has some of its nodes removed when constant variables are present. A variable that is a constant may be a constant-1 or 0. It is a node present in the AIG in which its value never changes, regardless of the input vector that is being applied on the AIG. There is no possible combination of inputs that may flip a constant.

If we assume that the XOR's PI node b from Figure 2.1 is a constant-1, then we would be able to propagate this signal along the XOR in AIG format. Figure 2.2a shows the constant propagation process. Notice that PI b was set as 0, while the other PIs remain the same variables. Then the constant is propagated along the graph. This may be done with a *depth first search* starting from the AIG's POs. Any node that a constant reaches will also be a constant, except when a constant-1 reaches a node jointly with a non-constant. In this case, the constant ceases to exist. In Figure 2.2a, it can be seen that after propagation the constant on node b ended up removing three nodes and in node f (referencing Figure 2.1) the constant met the non-constant propagated as a and ceased to exist.

Figure 2.2: Constant propagation procedure and resulting AIG


$\overline{\bar{a}*\bar{c}} * \overline{a*c}$
$\bar{a}*\bar{c}$
$a*c$
a
$\bar{a}$
0
a
a
0
0
a
0
c
$\overline{\bar{a}*\bar{c}} * \overline{a*c}$
$\bar{a}*\bar{c}$
$a*c$
a
c

(a) Original AIG. (b) Reduced AIG.

Source: The Author.

Since all nodes in an AIG are two-input AND nodes, we may make some assumptions based on the logic AND behavior in the presence of a constant variable. Table 2.1 shows the truth table of an AND logic with the variable b as a constant-0 and as a constant-1. Notice that when the PI b is set to 0, this AND node fanout will always be 0, and when the PI b is set to 1, its fanout will always replicate the other non-constant fanin. In both cases, a node representing this AND logic with a constant as input may be removed. Applying this simplification to the XOR in Figure 2.2a supposing PI b is a constant-0, we get as a result from the AIG in Figure 2.2b, which has fewer nodes than the original AIG.

Table 2.1: AND logic under constants.

constant-0			constant-1		
a	b	a AND b	a	b	a AND b
0	0	0	0	1	0
1	0	0	1	1	1

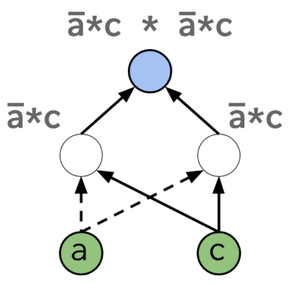
Source: The Author

2.1.4 Structural Hashing

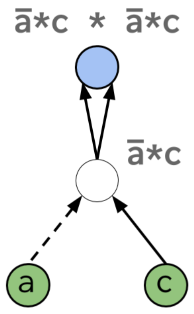
When the propagation of constants simplifies an AIG, several re-connections are made in the AIG. For example, the AIG in Figure 2.2 had a replacement of the fanin for the two white nodes that remained after simplification. The replacement was the PI a. But let us assume that the resulting AIG after simplification was the one from Figure 2.3a with some polarity connections modified, notice that in this case, both white AND nodes have exactly the same pair of fanins: $(!\,a * c)$. These two nodes are redundant. One node is sufficient to represent this intermediate logic inside the AIG. The *structural hashing* technique is used to resolve such a problem. It consists of using a *hash table* to store each node's fanin information. The *key* value for the hash is the combination of the node's fanin pair. For each re-connection done during simplification, a new key is generated for the reconnected node, and a lookup is realized in the hash table to check if the new key is already present in an existing hash instance. If it is, a new fanout connection on the existing node is added, while the redundant node generated is disregarded. If otherwise, the new key is not found, the key value of the node that was reconnected must be updated in the hash table. Figure 2.3b presents the case where one of the repeated white nodes was found in the hash table and added as a fanout on the located node.

Notice in the resulting AIG from Figure 2.3b that the output node ends up having

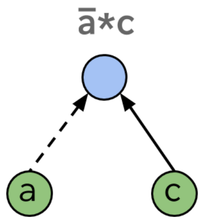
Figure 2.3: Structural hashing usage on example AIG.



(a) Original AIG.



(b) Reduced AIG - duplicated nodes in the AIG.



(c) Reduced AIG - node with repeated fain.

Source: The Author.

the same nodes in its two fanins, this follows the *idempotent* rule of boolean algebra, meaning that for an arbitrary boolean variable we may state $A * A = A$, or even for the example demonstrated in Figure 2.3b: $(!\,a * c) * (!\,a * c) = (!\,a * c)$. Then, following the simplification process in the AIG, re-connections are done on the PO's fanins and its key value on the hash table is to be updated, although the fanins both have the same value, this way we may remove the current PO and set the intermediate node with logic $(!\,a * c)$ as the new PO, resulting in the AIG from Figure 2.3c.

The process demonstrated in Figure 2.3 presents two different situations that the re-connections done during simplification may bring to the nodes in the simplified AIG. They are both unnecessary for the AIG's structural representation, increasing the logic depth and number of nodes and they should be always treated when they occur. The experiments executed in this work presented a relatively rare occurrence of such situations, occuring in no more than 1% of the nodes for the AIGs used in this work.

2.2 Neural Networks

The study of NNs for image recognition dates back to the late 80's (HAYKIN, 1994; BISHOP et al., 1995), although its practical usage on real world applications could only be happening now a days in the world, with the introduction of IoT devices for example. Aside from this influences on the research area but also the introduction of *Big Data*, thanks to people that work hard to assemble and label images in a large scale is one of the factors that allowed this technology of image recognition by NNs to grow as much. The proper compiling of image datasets allows for an outstanding development of supervised training of NNs in a substantially low amount of years (GYSEL, 2016; MIT 6.S191, 2019).

Competitions and contests are a competent approach to motivate researchers into finding better solutions for present unsolved or partially solved problems. The ImageNet challenge ILSVRC has demonstrated great improvements in the last 10 years among the solutions proposed by competitors (RUSSAKOVSKY et al., 2015). The introduction of *Convolutional Neural Networks* (CNN) was a remarkable achievement, since the first time a CNN won the competition with AlexNet (KRIZHEVSKY; SUTSKEVER; HINTON, 2012) the accuracy results presented by competitors improved significantly, to a point that every winner since 2012 made use of CNNs, even coming to surpass human capabilities in image recognition after further improvements with ResNet (HE et al., 2016). These NNs are composed of millions of parameters and demand for billions of operations bringing an extremely high demand to its hardware implementation (GYSEL, 2016; XU et al., 2018).

A great part of the development in NNs for image recognition comes with the increment of deeper NNs, for example the ResNet comes with the cost of being composed of 152 layers of neurons. In the NNs proposed in this work, the NNs have only 3 fully connected layers and they already exceed at least 25 times the number of nodes than the largest AIG presented in the IWLS contest of 2019 (International Workshop on Logic and Synthesis, 2019). With such high requirements, optimization processes that enable a more compact NN should enhance the capability of building neuromorphic hardware.

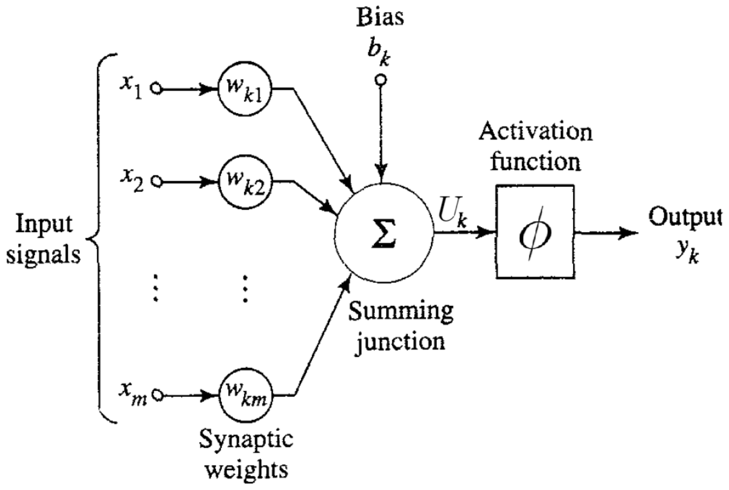
2.2.1 Neural Network Structure and Learning

In this section we shall briefly explain the structure and the learning process of NNs and how they are able to learn to classify input images. In its essence a NN is nothing more than a mathematical model composed of a network of multiple MAC operations. A NN may be represented by an interconnected number of neurons and we shall begin by explaining how the neuron structure work, followed by the non-linearity aspect of the NN and also how to learning process takes place.

2.2.1.1 The Neuron

The basic building block of a NN is the neuron, inspired by the structure of a living being's brain. Figure 2.4 demonstrates an illustration of a model of a neuron denoted by k, it is composed of an arbitrary number of input signals $x_1, x_2, ..., x_m$ and an output y_k, each input connection (synapse) is composed of a *weight* value $w_{k1}, w_{k2}, ..., w_{km}$ multiplied by its referenced input signal x. Equation2.5 demonstrates the summation of all input synapses for neuron k.

Figure 2.4: Model of a neuron.



Source: Adapted from (HAYKIN, 1994).

$$\begin{aligned} U_k &= \sum_{j=1}^{m} w_{kj} x_j \\ &= w_{k1}x_1 + w_{k2}x_2 + ... + w_{km}x_m \end{aligned} \tag{2.5}$$

The neuron from Figure 2.4 also consists of an *activation function* denoted by ϕ. There are different types of activation functions and their purpose is to convert the neuron's output to be non-linear. This limits the amplitude of its output to a finite value. The neuron's model is composed of a *bias* value, denoted by b_k, it is an additional parameter that is used to adjust each neuron's output along with the inputs *weighted sum*. Thus, the bias is a constant which helps the model in a way that it can fit best for a given type of data. That being said, Equation2.6 presents the function for the neuron's output value denoted by y_k, where the bias is simply a constant added to the sum and u_k is a summation. The output of the neuron is then given by these two variables applied to the activation function. The bias and the activation function of a neuron are used to improve the NN's learning capabilities and the NN's weights are iteratively modified to find weight values that output the proper labeling for the input images. This updating of the weights is the learning process properly said. An important measurement for a NN is its accuracy, which in supervised learning classification, is defined as the number of elements that the NN was able to correctly classify for the training set or the test set (HAYKIN, 1994; BISHOP et al., 1995; NIELSEN, 2015).

$$y_k = \phi(U_k + b_k) \tag{2.6}$$

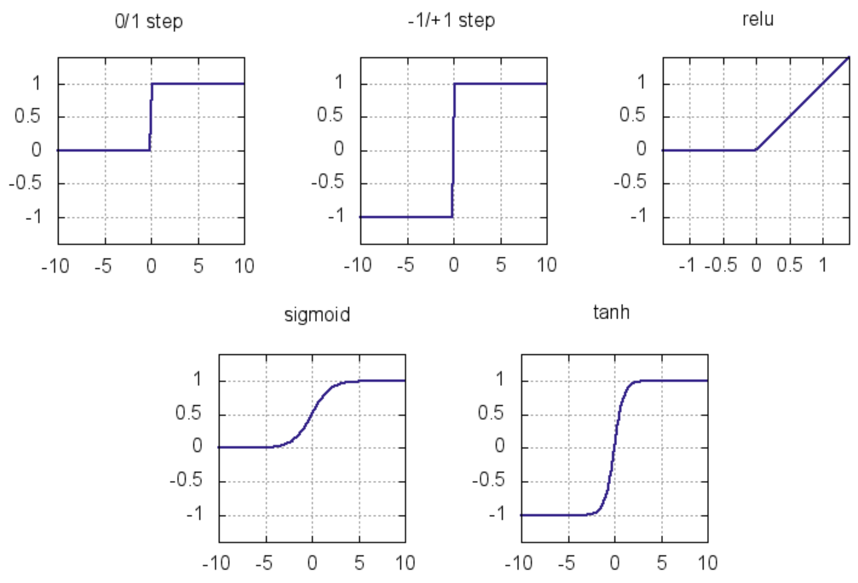
2.2.1.2 Activation Functions

Figure 2.5 presents five different activation functions and demonstrates how each of them modifies the neuron's output y_k with regard to a range of u_k values. The purpose of an activation function is to make the neuron's output to be non-linear and different activation functions present different distribution of values for certain types of data. In the *Rectified Linear Unit* (ReLu) for example, its non-linearity lies in the fact that for every negative U_k value it renders a y_k value of 0, while for values of U_k greater or equal to 0 the function's output is the same. Without its non-linear aspect, the ReLu function would be the *identity function* $y_k = u_k$. The addition of non-linearity to the neuron's function is done because real world problems are almost always non-linear. The neuron's activation function calculation is the last procedure done by a neuron. If the neuron is not an output neuron, its output will be connected to the input signal of another neuron (HAYKIN, 1994; BISHOP et al., 1995; NIELSEN, 2015).

The Sigmoid activation function used to be the most frequently used in NNs (GYSEL et al., 2018), but it has an issue during learning called the *vanishing gradient* with regards to the loss function, which is an equation used during training to measure the NN's learning rate. For this reason the ReLu was introduced by (NAIR; HINTON, 2010), which solves the vanishing gradient problem and it is present in most state-of-the-art NNs' activation functions (GYSEL et al., 2018).

Figures 2.6a and 2.6b presents an example of classifications done by two NNs which differ with their activation functions used in their neurons. These types of graphs are called *feature spaces*. Their data points represent the input dataset and the NN would be responsible for dividing the feature space. In the examples presented, we have a NN with two input neurons, one represented by the vertical axis and and the other by the horizontal axis. No matter the size of a NN, if a linear activation function is used, the NN will only be able to linearly divide the feature space, like in Figure 2.6a, on the other hand if a non-linear activation function is used one is able to modify the NN's weights and achieve a division such as in figure 2.6b. Real world problems actually

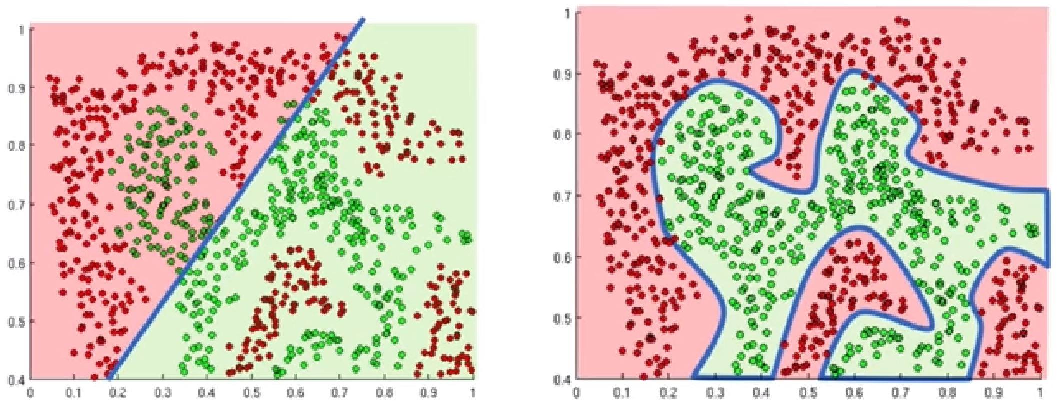
Figure 2.5: Set of activation functions for a neuron.



Source: Adapted from (BAGCHI, 2019).

have many more inputs required, a NN model might even have millions of inputs and the visualization of its feature space is unfeasible, although the example demonstrated should give a better understanding on how a NN classifies the input set presented (MIT 6.S191, 2019; HAYKIN, 1994; BISHOP et al., 1995; NIELSEN, 2015).

Figure 2.6: Feature space for neural networks with different activation functions.



(a) Linear activation function. (b) Non-linear activation function.

Source: (MIT 6.S191, 2019)

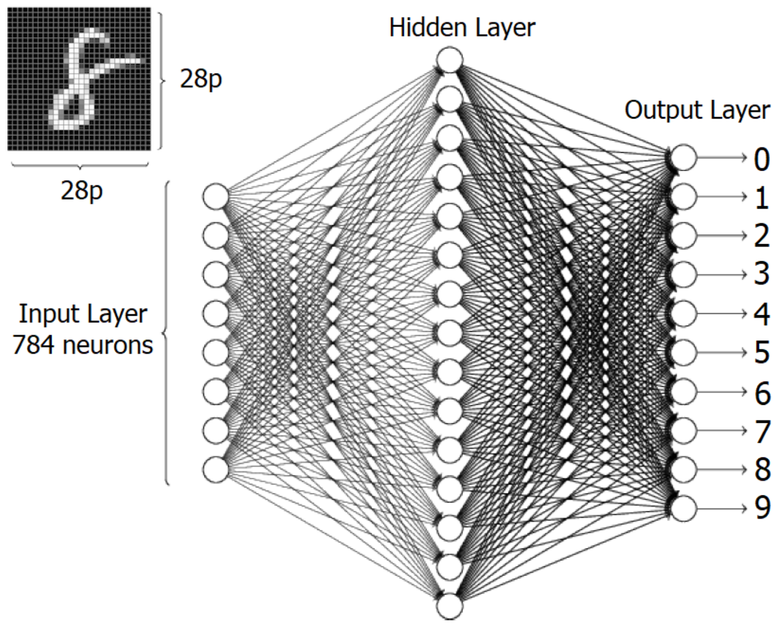
2.2.1.3 Fully Connected Layer

If we stack up an arbitrary number of neurons, we would have a *layer*, and if we put together an arbitrary number of layers, we would have a Neural Network. A layer may be an *input layer*, an *output layer* or a *hidden layer*. The purpose of a layer is to extract *features* from the input data, a feature being a certain portion of the image in which may be frequently found in images belonging to the same category. There are different types of hidden layers with different approaches to extract features from input data, sometimes by having a different type of activation function on its neurons or a different set of connections among neurons, such as a convolutional layer, soft-max, max-pooling, ReLu or fully connected layer. The latter being the most straight forward one (NIELSEN, 2015; MIT 6.S191, 2019). Even though the great advancements in NNs is due to the introduction of *convolutional neural networks*, for the sake of simplicity, in the present work we shall bring our focus to fully connected layers, since our focus is not to improve the capabilities of NNs but to reduce their complexity in their boolean representation.

Figure 2.7 presents a *feedforward* NN with a *fully connected* hidden layer. This NN is called feedforward because it doesn't have any cycles, *recurrent neural networks* are the ones that would have such configuration (NIELSEN, 2015). In our feedforward example, we have only one hidden layer, and each of its neurons is connected to all neurons in the input layer and all neurons from the output neuron. The NN presented in Figure 2.7 has the goal of identifying handwritten digits from the images in the *Modified National Institute of Standards and Technology* (MNIST) dataset (DENG, 2012). The NN is composed of 784 neurons in the input layer since the image area is composed of $28 * 28 = 784$ pixels; this way, we have one neuron for each pixel in the image. The pixel is a grey-scale ranging from 0 to 255, 15 neurons for the fully connected layer, and 10 neurons in the output layer, one for each labeling of the digits (NIELSEN, 2015; HAYKIN, 1994; BISHOP et al., 1995).

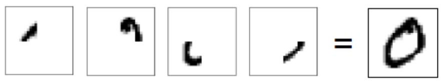
The purpose of a neuron in the hidden layer is to detect *features*, which are patterns with a frequent occurrence among the input images. For example, the shape of an object or letter. The idea is to modify the weights of the synapses during learning in such a way that the features may be identified. Figure 2.8 presents a set of features for the digit 0. We could say, for example, that the first four neurons in the hidden layer are each one responsible for identifying each of the four features. This is settled by applying high weight values for the synapses that are connected to the set of input neurons that represent the feature's drawing (NIELSEN, 2015; HAYKIN, 1994).

Figure 2.7: Fully connected single layer neural network for recognition of the MNIST dataset.



Source: Adapted from (NIELSEN, 2015).

Figure 2.8: Example of a set of features for the digit 0.



Source: Adapted from (NIELSEN, 2015).

2.2.1.4 Pooling Layer

While a fully connected or a convolutional layer is used to identify features in the input data, a *pooling* layer is responsible for summarizing the average presence of a feature and the most activated presence of a feature. In other words, the pooling layer is responsible for downsampling the present features in the input. It manipulates the features by reducing the data dimensions by summarizing the output of batches of neurons into a single neuron. Pooling layers are mostly seen on the final layers of a NN, but some times may be used in between layers to streamline the computation. To mention a few types of pooling layers that are commonly seen: *max-pooling*, where it retrieves the maximum value between a batch of neurons in the previous layer. *Average-pooling*, which retrieves the average values among a batch of neurons in the previous layer. And *soft-max* pooling layer in which assigns decimal probabilities to each class. In other words, we would have one neuron for each class that the NN is responsible for classifying, where each neuron has a value range of 0.0 up to 1.0. The summation among the neurons in the soft-max layer adds up to 1.0; this way, the soft-max outputs the probability that input has to be classified in each class, according to the NN.

2.2.1.5 Learning

For a NN trained with supervised learning, it needs something to learn from. Image datasets are then used, and there are different datasets available online with a vast amount of types of images such as animals, vehicles, objects and so forth. For example the CIFAR (KRIZHEVSKY; HINTON et al., 2009), the MNIST (DENG, 2012) or even the ImageNet (YANG et al., 2019) datasets are commonly used in researches on the topic of NN for image recognition. This type of learning based on previously known data is called *supervised learning*. It may be used on other types of datasets, aside from images, such as sound recording, video and text speech (HAYKIN, 1994; BISHOP et al., 1995; NIELSEN, 2015).

To measure a NN's learning rate, a *loss function* is used. It is calculated after feeding the NN with all of the input elements and iteratively modifying it to find weight values that output the proper labeling for the inputs. This updating of the weights is the learning properly said (HAYKIN, 1994; BISHOP et al., 1995; NIELSEN, 2015).

A fully connected NN may be represented with a matrix-vector product, such as demonstrated by Equation 2.7, the variable **y** represents the output vector for each of the output neurons, **x** represents the input vector, **W** the weights matrix for all layers, **b** the bias vector for each layer.

$$\mathbf{y} = \mathbf{W} \cdot \mathbf{x} + \mathbf{b} \tag{2.7}$$

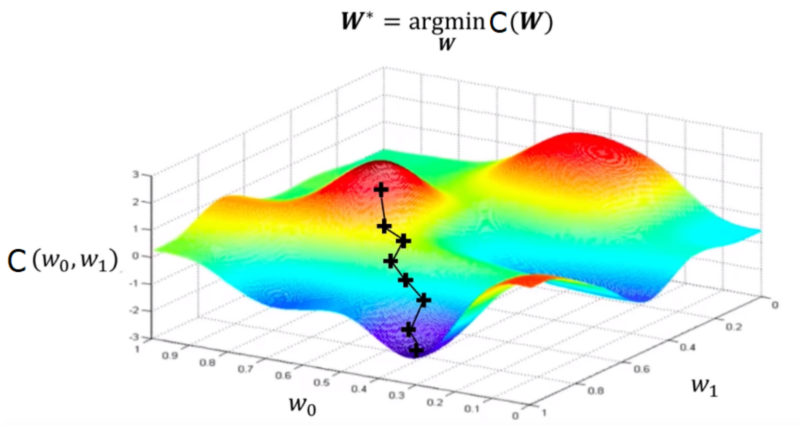
The output vector from the NN in Figure 2.7 would be a 10-dimensional vector **y**, remember that this NN classifies the MNIST dataset composed of handwritten digits ranging from 0 to 9. And for an input image represented as a vector **x** the NN's output is given by $\mathbf{y}(\mathbf{x})$. For example, if an input image is labeled as a 5, we would want the NN's output to $\mathbf{y}(\mathbf{x})$ = $\mathbf{t}(\mathbf{x})$ = (0,0,0,0,0,1,0,0,0,0), the variable **t** being the *target* for the input **x**. The idea is to modify the weights and bias of the NN so that its output $\mathbf{y}(\mathbf{x})$ approximates to $\mathbf{t}(\mathbf{x})$ for the training inputs **x**. Equation 2.8 presents the *cost function* denoted by C , also known as *loss function*, which maps the error for all the neurons in the NN into a real value. This equation is the quadratic cost function and it is also known as the *mean squared error* calculated for all of the NN's neurons (HAYKIN, 1994; BISHOP et al., 1995; NIELSEN, 2015).

$$C(\mathbf{W}, \mathbf{b}) = \frac{1}{2n} \sum_{\mathbf{x}} ||\mathbf{t}(\mathbf{x}) - \mathbf{y}(\mathbf{x})||^2 \tag{2.8}$$

In Equation 2.8 the target for an input **x** is denoted by $\mathbf{t}(\mathbf{x})$, the actual output value being $\mathbf{y}(\mathbf{x})$ and n the number of training inputs. Notice that the function $\mathbf{y}(\mathbf{x})$ is dependent on the NN's weights denoted by **W** and we wish to find a set of weight values that approximates every input **x** to its target $\mathbf{t}(\mathbf{x})$ and this is achieved by minimizing the cost function. The learning problem consists of searching for a set of weight values denoted by **W*** in the cost function space that has a small cost (HAYKIN, 1994; BISHOP et al., 1995; NIELSEN, 2015).

Figure 2.9 presents an hypothetical cost function space for a NN with only two inputs, the weight set for each input is defined by the w_0 and the w_1 axis, while the $C(w_0, w_1)$ shows the cost achieved for every weight value between 0 and 1. In real-world applications, some NN might have even millions of input neurons, and the graphical representation of such NNs is not suitable. Also, during training, it is impractical to compute all the possible combination of weights to find out which is the lowest cost function possible, heuristics are then used to find an acceptably small enough cost function to define the NN's weights (HAYKIN, 1994; BISHOP et al., 1995; NIELSEN, 2015).

Figure 2.9: Graphical representation of the cost function of a two input neural network.



Source: Adapted from (MIT 6.S191, 2019).

The goal of the learning process is to search for the minimal cost function by modifying the weights of the NN. The *gradient descent* technique may be used to do so, the gradient being the partial derivative of the loss function. The gradient is calculated iteratively, each time updating the NN's weight values, to look for the nearby minimum value in the loss function space. The process known as *backpropagation* is used to calculate the gradients of all layers in a NN, where the partial derivatives are calculated layer by layer. The iterative calculation of the gradient descent is illustrated in Figure 2.9. It is represented by the updating of weight values in the NN and is depicted as a few connected black data points cross following a downwards path in the cost function space. An improvement in the gradient descent technique is the *stochastic gradient descent*, which increases the reach of the search space and speeds up the learning process by making use of batches of training set instances (HAYKIN, 1994; BISHOP et al., 1995; NIELSEN, 2015).

During training of a NN *epochs* are used, where an epoch is the number of times a NN completes a full cycle reviewing the whole training set, in other words, when we say a NN was trained with 100 epochs, the NN went through the whole training set 100 times and each time modifying its weight values. If a NN is trained with too many epochs, it tends to get *overfitted*, where an overfitted NN is too attached to the training set. For example, if we are building a NN model able to recognize dogs present in an image and its training set happens to be composed of dogs only facing left, it might not be able to recognize dogs facing to the right, if too many epochs are used. The overfitting of a model is defined by the difference in accuracy it gives from the training set to the accuracy resulting on the test set. If it provides a high accuracy for the training set and a lower accuracy for the test set, it may be considered an overfitted NN. We would like for a NN to perform well on unseen data, then we may say that usually, we would not like for a NN to be overfitted.

2.2.1.6 Random Forest

A *Random Forest* or *Random Decision Forest*, as it is also known, is another technique of machine learning aside from NNs. Like with NNs, a RDF may be used to solve *classification* or *regression analysis* problems. An RDF is composed of multiple amounts of *decision trees*, where its random aspect brings the ability to overcome the overfitting lead by the usage of simple decision trees. The input is presented at the tree branches (also known as conjunctions of features), while the classification is given as output at the leaves (FRIEDMAN; HASTIE; TIBSHIRANI, 2001).

2.2.2 Neuromorphic Hardware

The desire for superior processing power and storage capacity brings researchers to explore different manufacturing implementations than the usual technologies. Accelerators are becoming acclaimed alternatives besides conventional designs, although this is only the case for specific types of workloads such as Machine Learning in general. Hardware accelerators for NNs are an emerging technology, and there is not an established standard design for it yet. So far, the technologies that have been proposed for NNs dedicated hardware are based on CPU, FPGA, GPU, and ASIC designs. However, as stated by (XU et al., 2018; NURVITADHI et al., 2017) GPU implementations are staggering on their processing power capabilities, ASIC designs seem to be more promising strategies since they are achieving better performance density overall. On the other hand, some works such as (NURVITADHI et al., 2016; NURVITADHI et al., 2017) show that there might be a reascend for FPGA designs for NNs, with future improvements on this technology, at the very least FPGAs are performing better than GPU designs.

The need for edge inference is growing larger. So far, companies adopt the usage of large scale servers in data centers for cloud-based inference. For example, as stated by (LECUN, 2019), Facebook executes $3 * 10^{14}$ predictions each day on their data-centers and the need for better performance density on devices grows larger (NURVITADHI et al., 2017).

Memristors are resistive devices able to both store and process information. They recently have been used to implement NNs accelerators and demonstrate good results on its processing capabilities, because it allows high processing elements density with good power efficiency. Memristors are built in a crossbars fashion to emulate plastic synapses in hardware devices. This approach is a good candidate to be explored because of its efficient matrix-vector multiplication, and NNs happen to require large amounts of such arithmetic (HU et al., 2016; JO et al., 2010; LI et al., 2018). A new approach to the usage of hardware accelerators for NNs is presented in the work (ZHOU et al., 2018) that exploits the sparsity characteristic of NNs. Their design achieves higher power efficiency and speedup when compared to the previous state-of-the-art NN accelerator published at (CHEN et al., 2014).

The works (AKOPYAN et al., 2015; DEBOLE et al., 2019) introduces a complete Design Flow for their neuromorphic hardware implementation. It is an event-driven, configurable, and scalable design, and it is said to be the "largest neurosynaptic computer ever built" with 256 million low-precision synapses and 1 million neurons on a spiking NN system. This design also has a crossbar fashion like memristors implementations, but its processing elements are made of their proposed neurosynaptic cores instead of memristors.

With the high processing power requirement that NNs bring, it is a hard challenge to manufacture a trained NN as an affordable chip. There have been a fair amount of proposed ways to compact NNs, mainly by handling its parameters like weights and synapses, during or after training, for example. But not as much has been done in the logic circuit level of abstraction. When we work at this stage we are attempting to reduce the parameters of a NN's logic representation like its number of nodes in an AIG or even its logic depth. In the subsections to come we present two approaches researchers take to compact NNs, the first and most common is focused on optimizing the NN by handling its core parameters like weights and synapses and another direction to reduced neuromorphic hardware requirements, which is most related to this work, where optimizations are applied to the boolean representation or logic circuits that represent NNs.

2.2.3 Neural Network Optimizations

Among the existing optimizations proposed by researchers, *approximate computing* is commonly one used to reduce the complexity of NNs (WANG et al., 2017; GYSEL, 2016; HAN; ORSHANSKY, 2013). For example, the *quantization* technique, which consists of reducing the number of bits to compose the digital representation of the NN's decimal values, is seen to be studied and used. Different approaches to do so are proposed. In (GYSEL et al., 2018), the need for multiply operations is reduced some times even completely removed, leaving adder-only arithmetic for the NN's implementation in hardware. This work uses an approach of *dynamic* fixed-point quantization (COURBARIAUX; BENGIO; DAVID, 2014), where different parts of the NN take different fixed-point lengths for the fractional part. Other works also make use of fixed-point arithmetic and attempt to quantize the bit-width of parameter representation (LIN; TALATHI; ANNAPUREDDY, 2016; HWANG; SUNG, 2014; SUNG; SHIN; HWANG, 2015; GUPTA et al., 2015). The works (COURBARIAUX; BENGIO; DAVID, 2015; HWANG; SUNG, 2014) can quantize the NNs to have binary weights and prove to achieve the slight loss in accuracy with such a simple representation for the NN's parameters. The quantization approach is an approximate computing strategy since it reduces the numerical representation of parameters.

Another common approach to reducing NN parameters is the pruning technique, in which attempts to cut-off pieces of the NN but preserving its accuracy. Pruning may be applied at network connections and removes unnecessary ones by learning which synapses are essential or even searching for redundant ones. This way, making a sparser version of the NN. The motivation behind the pruning procedure is on top of the assumption that NNs tend to be over parametrized, and similar accuracy may be obtained by making them smaller. In reference (LEE; AJANTHAN; TORR, 2018), pruning is applied before training in a single run, unlike other previous pruning approaches such as (HAN et al., 2015; HAN; MAO; DALLY, 2015; KARNIN, 1990; CARREIRA-PERPINAN; IDELBAYEV, 2018), in which require iteratively retraining of the NN. A commonly used pruning measurement is by limiting the magnitudes of the weights, where weights smaller than a certain threshold are removed (LEE; AJANTHAN; TORR, 2018; HAN et al., 2015; CARREIRA-PERPINAN; IDELBAYEV, 2018).

To cite some other techniques commonly seen and well accepted in the research area of compacting NNs: rank approximation (JADERBERG; VEDALDI; ZISSERMAN, 2014), structured sparsity (WEN et al., 2016), weight sharing (ULLRICH; MEEDS; WELLING, 2017; HAN et al., 2016), since it is common for a NN to have multiple similar weight values, the NN complexity is reduced by putting similar weight values in bins and retraining the NN with the binned values. Specifically, the work (HAN et al., 2016) discovered that 16 weight values are sufficient in many cases.

2.2.4 Boolean Optimizations for Neural Network Circuits

Approximate computing is a design paradigm that has proven to be an efficient resource saver in application level and consequently at the hardware level. A good amount of effort has been put in to reduce the NN's parameters. However, only a few works propose to optimize algorithms for logic synthesis that are driven by NN characteristics.

In reference (NAZEMI; PASANDI; PEDRAM, 2019), it is presented as a method to compact NNs implemented in FPGA. They propose a way to implement the logic of a neuron based on the input combinations by creating a truth table for the neuron, then limiting its exponential growth, where the NN's training set defines this limitation. With that, they can reduce the use of MAC operations. The digital implementation of the neurons in the NN is not done by realizing the dot product among weights and signals, but by synthesizing the logic representation of the neuron based on the training set. It is also implemented a shared logic when computing the neuron's combinational gate logic representation. Since neurons in the same layer may share the same inputs, their logic gate representation might also do. Results are shown based on NNs implemented with the use of MACs compared with their method, which uses fewer MACs, their method presents considerable savings in memory access due to fewer MACs required to implement the NN.

The usage of constant propagation with NNs implemented was already proposed by (WIRTHLIN; HUTCHINGS, 1997), although this work is directed to FPGAs. They proposed a circuit reconfiguration during run-time based on constants. It saves hardware resources, cycle times, and sometimes even power consumption, although it comes with the cost of reconfiguration time. The reconfiguration capability of an FPGA allows for a more flexible approach for the implementation of NNs. The work (WIRTHLIN; HUTCHINGS, 1997) analyzes the trade-off with the time cost of re-configuring the circuit during run-time versus the reduction in the area achieved. Our work and the reference (WIRTHLIN; HUTCHINGS, 1997) both make use of constant propagation to reduce the circuitry representation of a NN. However, the reference proposes a reduction in area usage and delay only for FPGAs and sets constants based on a *template matching* technique. In contrast, our work sets constants based on signal probability, without saying it is done on the boolean representation format of a NN and introduces ways to look for better candidates to become constants.

The reference (CHATTERJEE; MISHCHENKO, 2019) introduces a technique to identify if a machine learning model is overfitted or not. They do so by identifying nodes in the model's boolean representation that usually have the same value when exposed to the training set. If a node happens to have the same binary value for almost all of the training images, this node's signal value is perturbed to check if the NN is still able to generalize for the test set. If a NN is overfitted, its structure should have components that are present only for identifying the training set. Their idea is to find those components by perturbing nodes that have constant values when influenced by the training set. Their approach to *detect the overfitting* of a machine learning model is based on identifying constant values in the circuit representation of a NN, being a similar way to how we set constants to *reduce* the NN's boolean representation.

2.3 Boolean Representation of Neural Networks

To convert a NN into a logic circuit, there are some aspects to be considered, such as what would be the boolean representation used for the decimal values used on NN parameters. Such representation could be implemented with the enhancement of approximations or not, depending on the NN application. The boolean implementation of the MAC operation, which is essential for the proper functionality of a neuron and the boolean implementation of the activation functions.

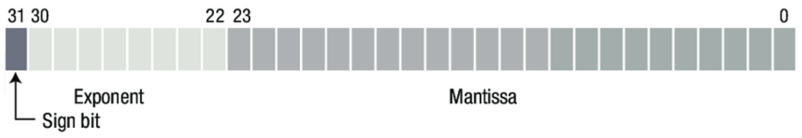
2.3.1 Boolean Representation of Decimal Values

Neural networks are composed of a set of parameter values such as its weight, biases, and activation signals. These parameters usually are represented by decimal values. Digital computers don't understand real numbers like humans do, then a digital representation for such values must be implemented if we wish to have a hardware implementation of a NN. There are two main approaches to represent decimal values in a manner that a computer understands, that being a *fixed-point* or a *floating point* representation. Traditionally, NNs make use of the floating-point representation. However, a fixed-point representation is less resource-hungry (GYSEL, 2016; HASHEMI et al., 2017), and it has been proven that NNs work fine with low bit-width fixed-point representations (GUPTA et al., 2015; COURBARIAUX; BENGIO; DAVID, 2014).

The *IEEE 754 Standard* is a document that presents an official standard format for floating-point arithmetic operations for decimal values. The IEEE 754 defines standard formats for floating-point arithmetic operations, and it is defined by a *single precision* format composed of 32 bits, a *double precision* format composed of 64 bits and among three other formats with greater bit-widths. Figure 2.10 presents the single-precision format for the IEEE 754 standard, where one bit is for representing the number's sign, being positive or negative, 8 bits for the exponent that is biased by 127 this allows for an exponent range of -127 to +127, and 23 bits for the mantissa. Numbers composed of only zeroes or ones in the exponent are unique situations. When the exponent has all zeroes, it is representing the number 0 or a normalized number, based on the mantissa bits. When the exponent has all bits as one, it is representing either a +/- infinity or not a number (NaN) (GYSEL, 2016; IEEE Standards Association, 2019).

The fixed point representation is also presented. This representation consists of

Figure 2.10: IEEE 754 single precision floating point decimal in binary.

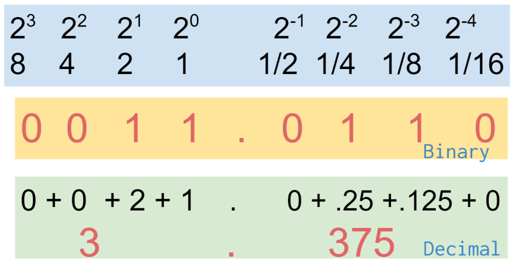


Source: Adapted from (GAO et al., 2017)

having a fixed amount of bits to represent the integer share of the number and another fixed bit length to represent the decimal part of the number. Figure 2.11 presents how the number 0011.0110_2 in binary is calculated to be 3.375_{10} in decimal format for a fixed point representation composed of 8 bits. This is a simple example with 4 bits for the integer part and 4 bit for the decimal part of the number. However, a fixed point representation may have more digits available to represent numbers, this way achieving a broader range of possible decimal values to be reproduced.

A fixed point format may be *dynamic*. This concept is that two different numbers with the same bit-width may have their fractional parts with different sizes, and arithmetic operations may still be done on top of them (GYSEL, 2016).

Figure 2.11: Example of a fixed point decimal number in binary (0011.0110=3.375).



Source: The Author

Among the possible implementations of a binary representation of decimal numbers, all of them have precision limitations when arithmetic operations are being executed. For example, if we were using the fixed-point representation from Figure 2.11 and the result of an arbitrary binary operation was equal to 0.4_{10}, this value would require more bit positions in the fractional section to be able to represent such value precisely. The binary representation of 0.4_{10} could be approximated to $0000.0110_2 = 0.375_{10}$ even though it is not the exact value. But even still, NNs usually perform well with such approximations due to imprecision with the representation of binary values.

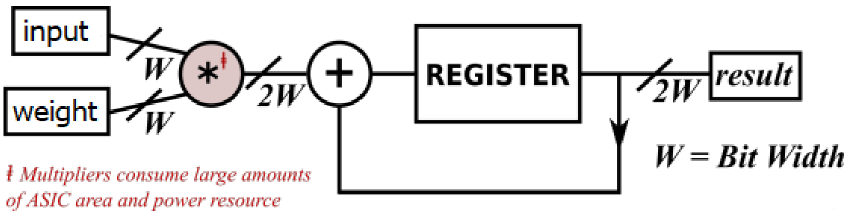
The more bits we use to represent decimal values, the broader the range of such representation. On the other hand, if we use a smaller bit-width representation on NNs, it usually results in savings in computational and memory requirements. The quantization process presented previously in Section 2.2.3 is the process that reduces the bit-width representation of decimal values for a NN while attempting to maintain its accuracy. Previous works that introduce quantization techniques usually look for the proper amount of bits to represent a NN, achieving reductions to bit-widths much smaller than the IEEE 754 standard, such as 12 bit, 8 bit or even 6 bit-widths (GYSEL, 2016; LIN; TALATHI; ANNAPUREDDY, 2016; SUNG; SHIN; HWANG, 2015; HAN; MAO; DALLY, 2015) or even a single bit representation of a NN's parameters with excellent preservation of accuracy (COURBARIAUX; BENGIO; DAVID, 2015; HWANG; SUNG, 2014).

2.3.2 Multiply Accumulate Operation

A MAC unit is usually used to build hardware accelerators for NNs and general-purpose DSPs (HASHEMI et al., 2017; GARLAND; GREGG, 2018; GYSEL, 2016; SZE et al., 2017). In Subsection 2.2.1.1, we presented the structure of a neuron and its mathematical representation. This representation being the *weighted sum* of a neuron's input signals. This is the exact process that a MAC unit is responsible for calculating. A MAC unit is responsible for *multiplying* two input values, then adding its solution to an *accumulator* variable, sometimes a register.

Figure 2.12 presents a general block diagram of a MAC unit for a NN. It is presented the abstraction of a neuron's single input synapse, with its weight being multiplied with its activation input signal, then accumulated to a register. However, a neuron would have multiple inputs. To calculate the output of a neuron, we would require to accumulate all weighted input signals to this register.

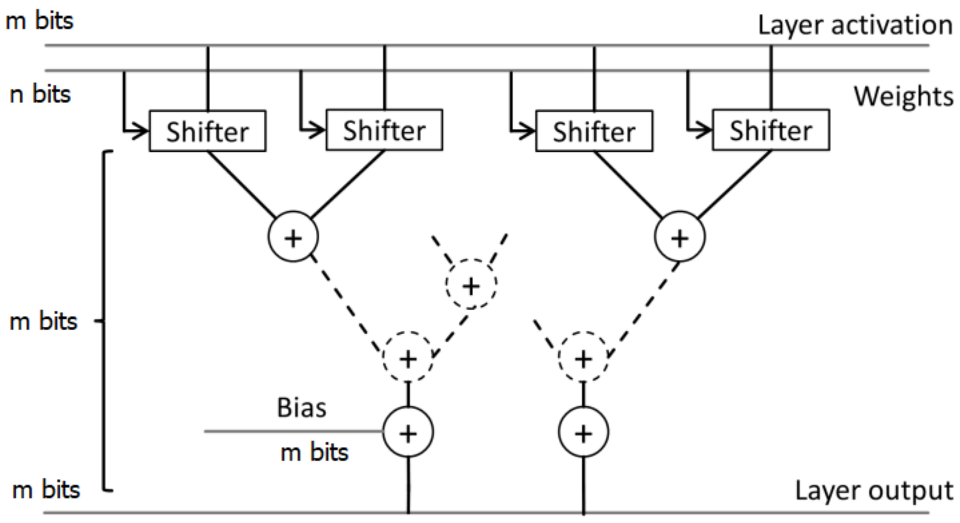
Figure 2.12: Block diagram of a MAC unit for NNs.


input
weight
W
W
*ǂ
2W
+
REGISTER
2W
result
W = Bit Width
ǂ Multipliers consume large amounts of ASIC area and power resource

Source: Adapted from (GARLAND; GREGG, 2018)

We could take an approach that would not require the usage of registers by implementing the accumulation of all weighted activation signals of neurons in a layer with the structure proposed in Figure 2.13. This way, we avoid the usage of sequential segments to implement a MAC operation for a NN by stacking the additions in a combinational format. The accumulation process requires that all of the weighted inputs of a neuron are added with one another, in Figure 2.13 we have the abstraction of two neurons with an arbitrary number of input signals, where the information goes from one layer to another.

Figure 2.13: Multiple MAC units implementing a NN layer with a tree shape.



Source: Adapted from (GYSEL, 2016)

When we make a digital implementation of a NN, we must define what is the type of binary representation of decimal numbers that will be used (e.g., fixed or floating-point) and also the bit-width representation of parameters. The references (QIU et al., 2016; GYSEL, 2016) proposes the usage of *mixed precision*, in which a fixed point is used, and different parts of the NN use different bit-widths. As can be seen in the example MAC structure in Figure 2.13, we have a bit-width of *m* bits for the activation signals and the operations in the accumulation tree, while *n* bits are used for the weights. This approach has the intention to reduce the bit length requirement for some of the NN's parameters.

We would wish to avoid the usage of multipliers for the multiplication step since multipliers require a large amount of silicon area to be implemented (GYSEL, 2016; GARLAND; GREGG, 2018; QIU et al., 2016). Notice that the multiplication steps in Figure 2.13 were replaced by *shifters* to multiply each of neurons inputs with their respective weights, this way bypassing the usage of multipliers.

If we use shifters to execute the multiplication process, then we could use a relatively small bit-width to store the weight values. (GYSEL, 2016) demonstrated that it is possible to use 4 bits to represent the weights while using 32 bits for the activation signals, for example. Although, the usage of shifters brings an approximation to the calculation since the shift may only realize multiplications where the multiplier operand is a power of 2 value. For example, if we wish to multiply $5x_1 + 11x_2$, none of them are the power of two numbers, then we would have to break it down to the summation $4x_1 + x_1 + 16x_2 - 4x_2 - x_2$, enabling the usage of shifters only to multiply the weights and activation signals. If we are working with a pre-trained NN, its weight values would be constant. Then they could be manipulated to fit the shifter only multiplications.

A bit-wise *overflow* event may occur during multiplications or additions; an overflow happens when the binary representation of the NN's parameters achieved its maximum possible value. As in a carry-out event that surpasses the possible representation of the number, resulting from an addition or multiplication in an unsigned binary representation. For example, adding the two binary values: $1101_2 + 0011_2$, since we are using only 4 bits here, the carry-out would require an extra bit position to give the exact result. An overflow may also occur differently for a signed representation, for both situations a *saturation* is used, where the value is rounded to the maximum or minimum value that the binary representation of decimals can reproduce. For example, the adding of $1101_2 + 0011_2$ would saturate to 1111_2, even though the precise answer should be 11111_2 with 5 bits.

2.3.2.1 Research on Multiply Accumulate for Neural Networks

Fully connected and convolutional layers are the most resource-demanding part in a NN (HASHEMI et al., 2017; GYSEL, 2016; SZE et al., 2017; GARLAND; GREGG, 2018), they are based on the same arithmetic, namely a series of multiply and accumulate operations. In other words, the most costly part of a NN is implemented with the usage of MAC operations. There is a vast amount of works that attempt to optimize MAC units architectures, or optimizations approaches for the MAC operation, optimizations directed for NNs or not, since MACs may also be used in other applications or even general-purpose devices.

In work (GARLAND; GREGG, 2018) they make use of the weight sharing technique for NNs, which usually stores only a few weight values and maps the weights with an index and bins with the weight values. They exploit this weight sharing idea to extend it to their MAC unit implementation. The reference (ZHU et al., 2020) introduces a dataflow for hardware accelerators designed for CNNs in an FPGA platform, their approach can skip MAC operations by avoiding zero-valued weights, and this way bypassing unnecessary computations. The work presented by (MEI et al., 2019) introduces a parallel approach where batches of MAC operations coincide, also their process is applied during run-time.

The works (COVELL et al., 2019; CHATTERJEE, 2018) introduces the ability to implement NNs without the usage of MAC operations and without the need for floating or fixed point representations by making use of only *look-up tables* (LUT) to implement a NN. The usage of LUTs for machine learning may be considered a counter-intuitive approach since a single LUT maps an input to an output. Although, the reference (CHATTERJEE, 2018) presents a study about the *memorization* and *generalization* capabilities of a machine learning model and demonstrates that an interconnected network of LUTs is able to generalize on data.

3 METHODOLOGY

In the present chapter, we discuss the methodology behind the contributions this work brings. It starts by explaining the implementation of the machine learning models used in our experiments, and some characteristics are exposed concerning their signal probabilities in their AIG representation. Afterward, the definition of the threshold probability variable is presented, which is crucial for the proper understanding and usage of the proposed methods. Afterward, we explain in detail how each method works and different approach. Each one has to determine nodes to be set as constants.

3.1 Experimental Models

In this chapter, we discuss some aspects and characteristics of the machine learning models used for the experimental results of our proposed contributions. We make use of two neural networks and one random forest model. The NNs both are the same model with the exception that one of them is overfitted, going through 100 epochs during training, and the other is not overfitted going through 2 epochs, namely NN-2 and NN-100.

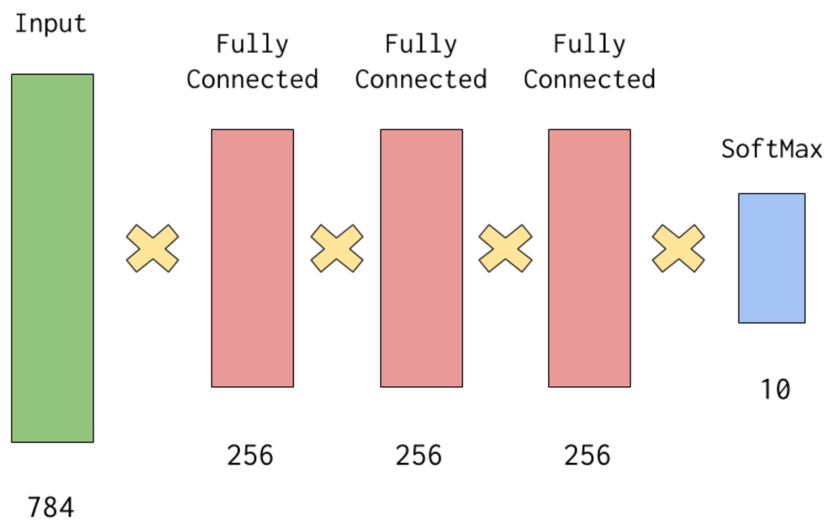
Since we are working on the boolean abstraction of a model, we may use other types of machine learning sources, such as a RDF, for example. This way we demonstrate that our proposed methods are able to handle other types of machine learning models. In other words, the methods proposed are not structurally constrained, being able to handle a supervised machine learning model structured on a digital format.

The machine learning models used in this work were converted to a boolean representation after training, more specifically to the AIG format. Some of the details concerning the conversion of NNs to a logic circuit were discussed in Section 2.3, and the usage and aspects of the AIG data structure was discussed in Section 2.1. The weights and activation signals for the models were represented using a signed 8 bit and 16 bit, respectively, with fixed-point numbers, in which 6 bits were reserved for the fractional part. After training, the values of the weights are cut down to [-2.0,2.0) before their conversion to a fixed point. Each MAC unit multiplies the weight with 8 bits and the activation signal with 16 bits and accumulates with 24 bits with saturation. The multiplications are done with shifters only structure, which requires breaking down the constant weight values into powers of two multipliers.

The NN models specifically were built with three fully connected hidden layers, where the hidden layer neurons are built with the ReLu activation function (XU et al., 2015), with its digital implementation composed of a comparator and a multiplexer. The input layer is defined by the bit representation of the image pixels in the MNIST dataset. The images are composed of pixels, in which represent a grayscale ranging from 0 (black) to 255 (white) and requiring 8 bits to be implemented in a boolean format. The output layer in each of the three models is a soft-max layer with 10 signed 16-bit values, with one value for each digit ranging from 0 to 9, and the largest value represents the picked class by the NN. This process was performed by the authors of (CHATTERJEE; MISHCHENKO, 2019), and their AIGs were provided to us for a further investigation on the subject.

Figure 3.1 presents an illustration of the composition of the layers for the NN trained to recognize the handwritten digits in the MNIST dataset as described in the previous paragraph, on top of each layer drawing it is denoted the layer type and on its bottom the number of neurons composing such layer.

Figure 3.1: Types of layers and number of neurons per layer for the experimental NN models.



Source: The Author.

In the next subsection, we discuss some characteristics of the NNs for their boolean representation in AIG. We present the signal probability values for the AIGs nodes and their relation with the threshold probability used on our proposed contributions to set constants.

3.1.1 AIG Models Signal Probabilities

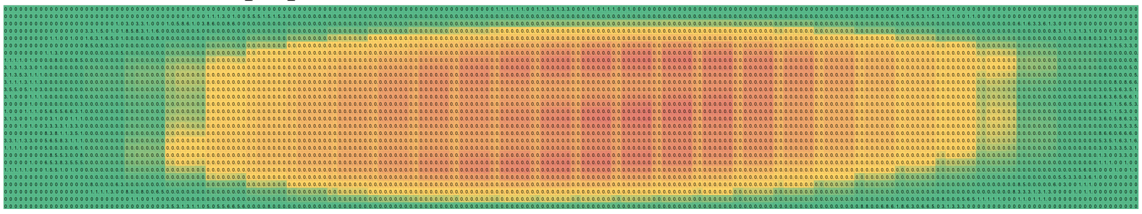
Our proposed methods are made to reduce the size and depth of NNs represented in AIG. This reduction is based on the definition of some AND nodes present in the AIG to be set as constants. In which the definition of an AND node in an AIG to be a constant is given by the fact that such node will have, for every possible combination of signals in its fanin, the same signal fanout. A node may also be *nearly a constant*, meaning that in almost every situation, this node has the same fanout value and how close a node is to be a constant is defined by its signal probability and a threshold probability.

To calculate the probability value of signals in a circuit in whichever logical representation is not a trivial task, because the input set of such problem grows exponentially. It is not yet known how to calculate the exact value of signal probabilities of nodes in a circuit, so far heuristics have been used to do so (FLAQUER et al., 2010; ANGLADA et al., 2016). On the other hand, the NNs' training set gives us the possibility of calculating its internal nodes signal probabilities in its representation as a logic circuit. We may do so by simulating the circuit with every entry in the training set. This way we avoid the effort of discovering every possible combination to calculate the circuit's signals probabilities and calculate the probabilities of the nodes based on the training set, which won't give the exact value, but a good approximation.

In Figure 3.2, it is presented the probability of each bit to be true (1) for the PI nodes in the AIG that reads the images from the MNIST dataset. Such images are $28 * 28$ pixels in the area, and each pixel is represented in an 8-bit format for digital representation. This way, the total area in Figure 3.2 is given by $28 * 28 * 8 = 6272$ bits. The probability is given by the chance that the node is a *true* signal, where the values range from 0$\%$ to 56.3$\%$ with green color for smaller, yellow for intermediate and red for larger probability values. It clearly shows how the PIs that represent the image's edges are not often set to 1; there are 697 PIs that were never set to 1 among all the 60,000 training images. This set of PIs may be defined as constants as they have 100$\%$ probability of being 0 $P(0) = 100\%$. Similarly, we may say that it has a 0% probability of being 1 $P(1) = 0\%$.

The graph presented in Figure 3.3 falls under different circumstances than the graph presenting the information on the PI nodes, it presents the probability for the internal AND nodes of the two different NNs and the RDF model in their AIG representation. The probability values calculated are based on the MNIST dataset. They were calculated

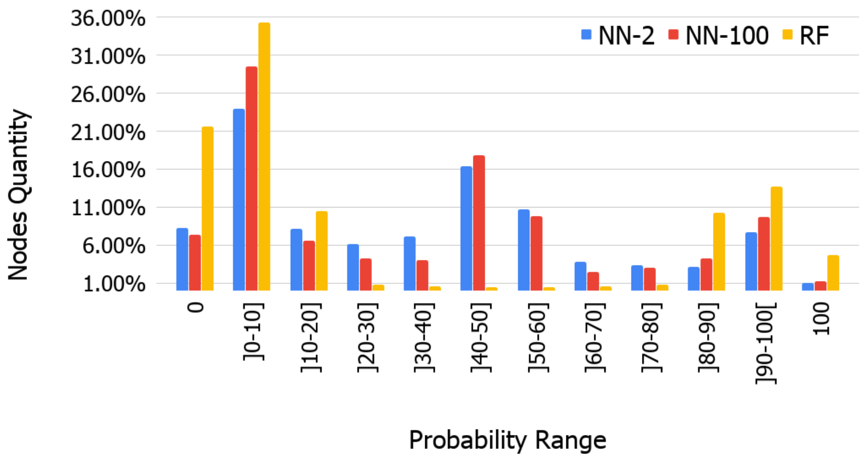
Figure 3.2: A probability color scale calculated based on the MNIST dataset of the PI nodes of the AIGs proposed.



Source: The Author.

by propagating all of the training set images, summing the number of times a node occurs to be set to 1 and dividing by the total number of images in the training set. The values in the horizontal axis represent the probability range in which a node belongs to and the vertical axis the percentage from the total number of nodes that are in such a probability range. For example, about 1% of NN-2 and NN-100 internal AND have 100% chance of being a value 1 based on the images in the MNIST training set. Also, the nodes that fall within the probability range of 0% were never set to 1 during simulation among all of the images in the MNIST training set. They represent at least more than 6% of NN-2 and NN-100 nodes. Figure 3.3 shows that a considerable portion of each NN is near constants. At the same time, the RDF implementation has a considerably higher amount of nodes that were always the same values during simulation.

Figure 3.3: A probability color scale calculated based on the MNIST dataset of the internal AND nodes for the experimental AIGs.



Source: The Author.

Aside from the fact that there is a considerable amount of nodes with the same value for all images, there is also a considerable amount of nodes that are near constants, taking into account the ones in the probability ranges 0% to 10% and 90% to 100%, which are the nodes that have a boolean value 0 and 1 respectively in at least 90% of the images.

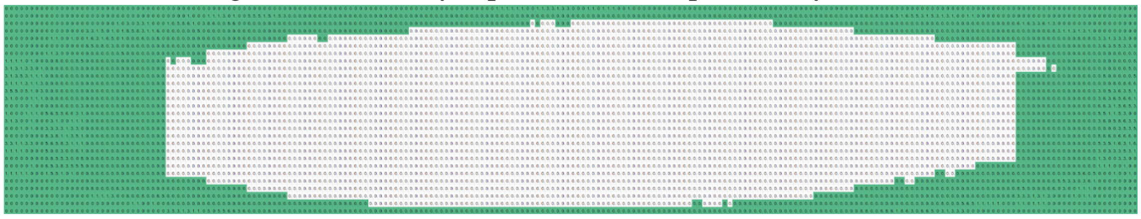
On the contrary, the nodes that are closer to the probability ranges in the middle are the ones that vary the most and should not be considered to be constants.

Even though a node has most of the time, the same value, it doesn’t mean that this node won’t possibly result in an error for an image when it is set to be a constant. We should choose carefully which nodes to turn into constants.

3.1.1.1 Threshold Probability Variable

We present a *threshold probability variable* (TH) in which is based on the signal probability of a node. If a node’s probability of being 1 is equal or greater than the TH, such node is considered to be a constant-1. Similarly happens to be with a constant-0. If a node has a probability of being 0 equal or greater than the TH, it becomes a constant-0. Figure 3.4 presents the set of input nodes that are within a TH probability of 99%. In other words, the PIs that are colored in green are the ones that have at least 99% chance of being 0, although the PIs probabilities of being 1 never reach such TH. As expected, we can see that a large part of the PIs has almost always the same input among all images from the MNIST training set. Similarly, the TH would also be used to define which of the internal AND from Figure 3.2 would be set as constants, for example, for a TH of 100% the nodes in the first and last columns would be set as constants.

Figure 3.4: Primary inputs within TH probability of 99%.



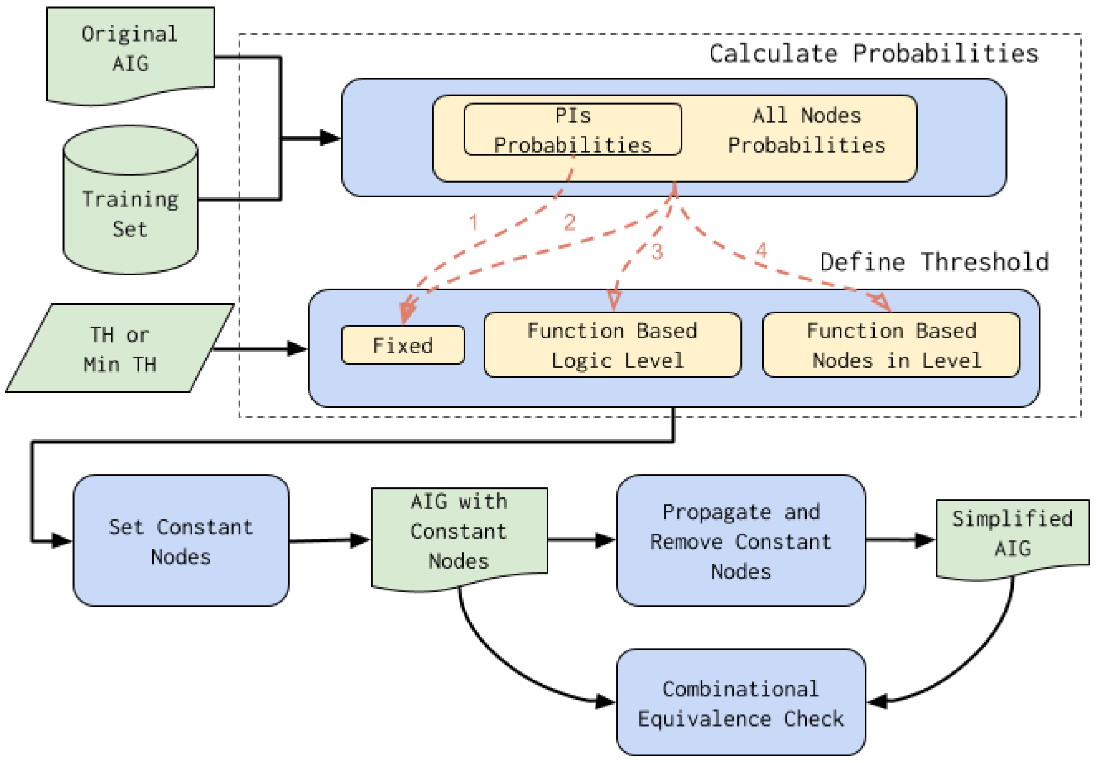
Source: The Author.

In the following sections, we present our proposed methods to reduce NNs in their AIG representation. All of them make use of the TH value. We are making use of approximate computing by inserting small errors to make the circuit’s logic representation smaller, so we must be mindful of doing so. Since, if we are too greedy by using a TH that is too low, we would be setting too many nodes as constants and thus might bring too many errors to the circuit’s accuracy. In the following sections, we attempt to present what would be a good trade-off between the loss in accuracy versus reduction in size and LD based on different approaches to reduce the AIG.

3.2 Methods

Figure 3.5 presents a flowchart of the processes executed by our implementation and presents the four proposed methods to simplify the AIG. The simplification is applied based on the probability a node has to have same independent of the input values. The probability calculation is defined base on the NN's training set. The simplification is done with the AIG representation of the circuit and by reducing its number of nodes and logic depth, it will directly reduce the project's final area and delay time. Four dashed arrows represent the four proposed methods to reduce the AIG, each is a combination of the possible sequence of events they go through. This work's contributions are represented inside the dashed rectangle. The two first methods consider a fixed TH, that is, all possible *target* nodes will respond to the same TH to be possibly set as constants. Whereas, our third and fourth methods take into account a variable TH. That is, different nodes have different TH values; in such cases, the threshold is called *Minimum Threshold* since the nodes' THs will all vary from the minimum value up to 100%.

Figure 3.5: Flowchart summary for the proposed methods and the performed procedures.



Source: The Author.

Our proposed methods take the same three inputs: the original NN represented in AIG format, already trained, the training image dataset, and the TH or Minimum TH to set the constants. With the input data set, the AIG's nodes probabilities are calculated, and with the TH, we can define which nodes will be set to be constants. Concerning the nodes probabilities calculation, only the first method differs, since it only is applied to the primary inputs. The nodes probabilities calculation is a straight forward process, requiring to simulate the AIG under the complete training set. It is worth mentioning that there isn't a tool or program that provides a probability calculation for a circuit based on the training set of a NN. After the nodes probability calculation, the method's target nodes are analyzed to define their TH values. Each method has its manner of calculating the nodes' TH values, and that is what renders a better or worst reduction.

With the TH values defined, an AIG file is generated where the nodes that pass the TH are set as constants. Then we use this AIG file in the ABC tool, which applies the process of propagating the constants with the support of structural hashing. These procedures are applied automatically by ABC when reading a file. Then only two commands from ABC were used for the propagation and simplification: (1) a read file (*&read original_file.aig*) and (2) a write file command (*&write simplified_file.aig*). The constants are always propagated every time a constant is applied on a node. This happens for every method, and the propagation process is always the same regardless of the method being used. After the simplification is done, it is provided a new NN represented in AIG, which is smaller than the original one. To be sure the AIG is logically intact even after simplification, an *Combinational Equivalence Check* (CEC) is performed comparing the file with constants present with the file after propagation and simplification, also with the ABC tool. The simplification process was not only performed with the usage of ABC but also a separate implementation was designed by the authors. It allows a further investigation about the location of the removed nodes. Even though it doesn't support the structural hashing process, the difference in size for the ABC simplification and the one implemented by the authors is tiny, corresponding to about 0.5% of the AIG's original size on average.

Notice the presence of two *Functional Based* methods denoted by the dashed lines 3 and 4, both of them establish different THs for different nodes. We propose two different ways to establish the TH values for all the nodes present in the AIG: One based on the node's LD, where nodes closer to the output will have higher THs, while nodes closer to the PIs will have lower THs. This way, nodes closer to the PIs will be more acceptable, and nodes closer to the POs will be more restricted when the method defines nodes as constants. We also propose a manner to define TH values based on the number of nodes in a level, where LDs that have only a few nodes will have higher TH values and will be less acceptable to set them as constants.

In each of the following subsections, the four methods proposed in our work are explained. The XOR in AIG format from Figure 2.2a previously used to expose the constant propagation procedure will also be used to demonstrate how each method sets the nodes as constants with the usage of the TH variable.

3.2.1 Only Primary Inputs

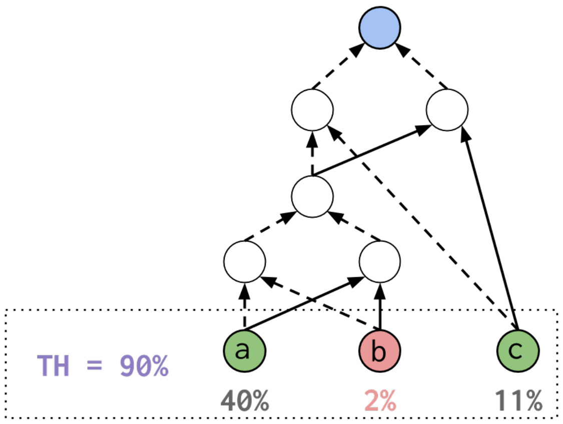
The *only primary inputs* method (OPI) is an initial approach to reduce the AIG's size and depth. In it, we are considering only the AIG's PIs to be analyzed. They are the only available *targets* to be possibly turned into constants. In other words, we are taking into account only the nodes present in Figure 3.2 and excluding the AIG's internal AND nodes to be possibly set to be constants. In this approach, all PIs have the same TH value; in other words, they will respond to a fixed identical TH.

Figure 3.6 presents the definition of constants for the OPI method supposing a TH of 90% presented in purple right beside the AIG. Hypothetical probability values for this AIG's PIs are also presented; this is the case for this example. However, when the procedure is used on a real NN, the probability values would be calculated from the NN's training set. Only the PI b probability surpasses the TH value. This is indicated by a red color in the node's b probability value and drawing. In the example presented, we are considering the probability of nodes being 1, and the TH is 90%, but we could also set the nodes to be a constant-0. In this case, we should also check if the probability of a node to be 0 surpass the TH value. For node b we would have $P(b = 0) = 1 - 0.02 = 0.98$ and 98% is greater than the TH, then this node should be turned into a constant. With this example, only the PI b is set as a constant by this method, no other nodes.

Remember that after the constants are set, they are always propagated along the AIG, then if we define only a set of PIs as constants, internal AND nodes will also be removed after the signal propagation. For the example presented in Figure 3.6, we would have the same constant propagation and simplification as the one presented in Section 2.1.3, which showed how the constant propagation and simplification procedure works with a constant-0 set in the PI b.

Most likely, nodes closer to the AIG's output won't be removed by this method, as the constant's propagation won't probably reach them. In this case, the constants cease

Figure 3.6: Probability and threshold values with OPI method.



Source: The Author.

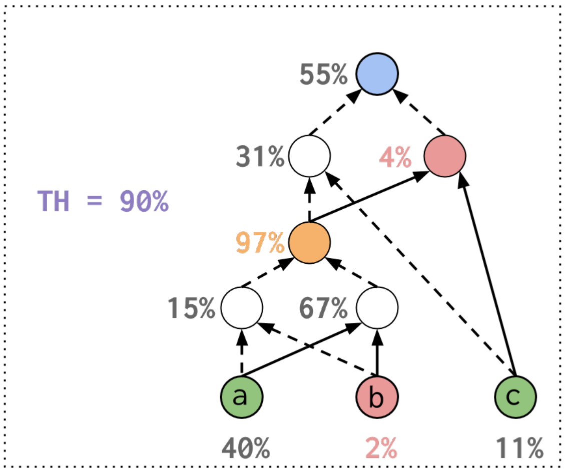
to exist before reaching longer depths of the NN represented as AIG. This method contemplates only the NN's input information to reduce it. The idea behind this approach would be to focus the AIG's reduction on the input behavior. In the MNIST dataset, for example, corner pixels are rarely used. Such a method lets us evaluate the NN's inference capabilities when such corner pixels are disregarded. This initial OPI method was previously published in (BERNDT et al., 2019).

3.2.2 All Nodes

Concerning our second method entitled *all nodes* (AN), it also uses a fixed TH, although it may turn internal AND nodes into constants before signal propagation. In other words, all nodes in the AIG are targets for this method, and they all have the same TH value. While the previous method considers only the NN's input behavior, this second method considers the whole AIG's behavior when stimulated by the NN's training set to reduce it.

Figure 3.7 presents the same XOR in the AIG format used previously to illustrate the procedure for the previous method. In the present method, all nodes may be targeted to become constants. Hypothetical probability values for the internal nodes were added this time. However, they also would be calculated from the NN's training set when the method is operating normally on a NN. The TH value also incorporates all the nodes present in the AIG. This way, not only the input b is set as a constant but also two other internal AND nodes since they all surpass the TH value. A constant-0 is illustrated in red,

Figure 3.7: Probability and threshold values with AN method.



Source: The Author.

while a constant-1 is illustrated in orange. Remember that the signal propagation would happen before removing the AIG nodes, meaning that not only the nodes that pass the TH would be removed, but also the ones that the constat propagation reached.

Such a method is more ambitious than the previous one when selecting nodes to remove in the AIG. Because, for example, if we set a TH of 99.9% using this method, for every node that has at least 99.9% signal probability, they might result in an error for the signal of this set of nodes. Since for calculating the probability of such nodes, we use the training set, and for 60 images, such nodes had one value, and for 59.940 images, it had another value. For 60 images in the training set, such nodes won't be propagating the signal it should for the integral functionality of the NN. In other words, such nodes bring an error signal being propagated along the AIG. Although this error might be masked since the signal may have to pass through other nodes, and these nodes might fix the boolean value that the signal should have. The error will only have a real negative impact on the NN if this signal reaches a PO. It usually would take various amounts of error signals to reach the POs for the NN to miss the inference on the input image. Since the present method targets all nodes of the AIG simultaneously, lowering the TH usually have a significant impact on the NN's accuracy, as it will be seen in the results chapter.

3.2.3 Functional Based

These methods are called Functional Based because they define the AIG's nodes TH based on a different function in each method. The idea is to select a set of nodes to become constants that are less likely to bring an error to the NN's inference capability. On the previously presented methods, both make use of the signal probability calculated based on the NN's training set. This allows us to use approximate computing to optimize the size of the NN in AIG format. In Functional Based methods, we are still using the nodes' signal probabilities, although we are choosing better candidate nodes to become constants. Notice that the higher a node's TH is, the less likely it will be that such node will turn into a constant since it must have a signal probability at least as high as the TH defined by the method. That being said, Functional Based methods attempt to set higher THs on nodes that are more susceptible to bring an error to the circuit and set lower TH values for nodes that are less likely to propagate an error to the AIG's output if it would be turned into a constant.

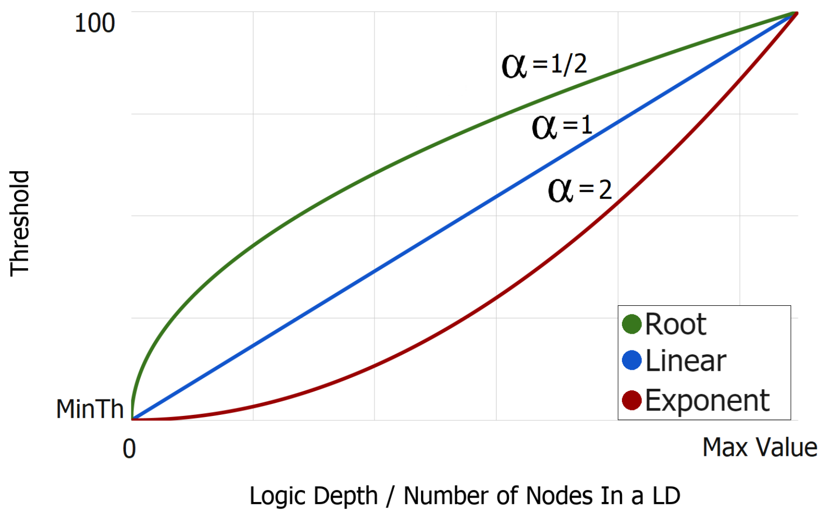
We present two different methods, which are Functional Based. One which sets THs for the nodes based on their LD and another, which defines THs for nodes based on the Number of Nodes in an LD. An important aspect to consider when defining the nodes' TH values for the Functional Based methods is to determine what would be the probability value distribution among nodes, in other words, the increase or decrease of probability values for different nodes in the AIG. With that in mind we present the graph in Figure 3.8, it shows three different distributions to define the TH values (vertical axis) with respect to the *logic depth* or the *number of nodes in a level* (horizontal axis). Both Functional Based methods use the same three distributions, although their horizontal axis differs. For now, take notice that the α variable is used to control the curvature of the function. With that, we shall make use of three different α values, as presented in Figure 3.8.

3.2.3.1 Logic Depth

We might not only define the TH based on the node's signal probability but also by its LD; likewise, this method is called *functional based on logic depth* (FB-LD). A node's LD is defined by the longest path from this node to a PI, where the distance is counted by the number of nodes in the path. In other words, the LD of a node tells how far a node is from a PI. Nodes with low LDs will have a lower TH value and will be more acceptable when they are being turned into constants. If it happens to generate an error, it will have a greater chance to be masked, because there are several nodes for the signal to go through until it reaches the AIG's output.

On the other hand, nodes that have greater LDs will be less likely to be turned to

Figure 3.8: Graph showing the three different distributions used on the functional based methods.



Source: The Author.

constants, since they will have greater TH values. Nodes closer to the AIG's output have a higher chance of bringing an error to a primary output signal if they are turned into a constant. It has a lower chance of being masked, as there are fewer nodes for the signal to go through until it reaches a primary output.

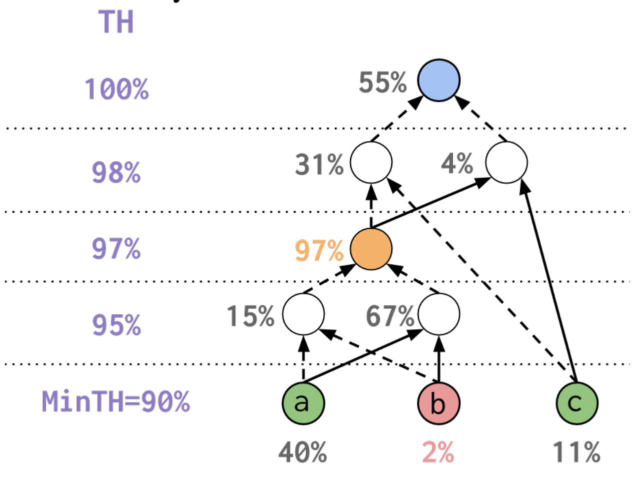
In this method, nodes that share the same LD will have the same TH value, and the TH values are different among the LDs within the circuit. The TH values grow in order with the LD, meaning that lower LDs imply a lower TH, and higher LDs imply a higher TH for the nodes in this LD. Equation 3.1 presents the function to define the TH distribution among the AIG's LD. α modifies distribution's curve, *MinTh* is an input parameter, and it is the minimum threshold an LD may be given, in other words, no other node would have a TH lower than the *MinTh* mark, *Level* is the LD to be given a TH value from the function, *GraphLevel* is the longest path in the graph going from a PI to a PO. It is calculated by executing a depth-first search in the AIG. With Equation 3.1, we can assure that the longest depth will always have a TH of 100%, and the first LD TH will always be the *MinTh* value, and each of the three distributions will set different TH values among the nodes. By modifying the α value in Equation 3.1, we can set different THs with the same method and look for a better solution by comparing the method's behavior when under different TH distributions.

$$TH(Level) = \left[(1 - MinTh) * \left(\frac{Level}{GraphLevel}\right)^{\alpha}\right] + MinTh; \tag{3.1}$$

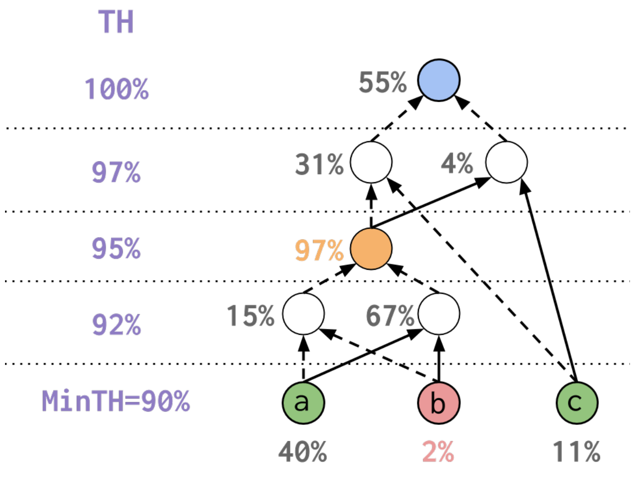
Figure 3.9 presents three illustrations with the previously used AIG, one figure is presented for each TH distribution. The figures demonstrate how the TH values are defined for each LD concerning each distribution in this method and which node ends up being set as a constant, orange for a constant-1 and red for a constant-0. The probability values are kept the same from the ones used in the previous method, notice that different nodes are set to constants, not only comparing with the previous method in Figure 3.6, but also among the different distributions used for this method. The TH values following the LDs in the AIGs have a different growth in each distribution when comparing Figures 3.9a, 3.9b and 3.9c. The Exponential distribution ends up being more acceptable when setting nodes as constants since it always gives lower TH values for the AIG's LDs when comparing to the other distributions. On the contrary, the Root distribution is more conservative when setting nodes as constants, as it always gives higher TH values for the AIG's LDs.

When analyzing the images presented in Figure 3.9, one might think that the difference among the distributions is not considerable, but remember that we are using these methods on AIGs with dozens of millions of nodes. They end up getting a considerable amount of different sets of nodes as constants. This can be seen in the results section.

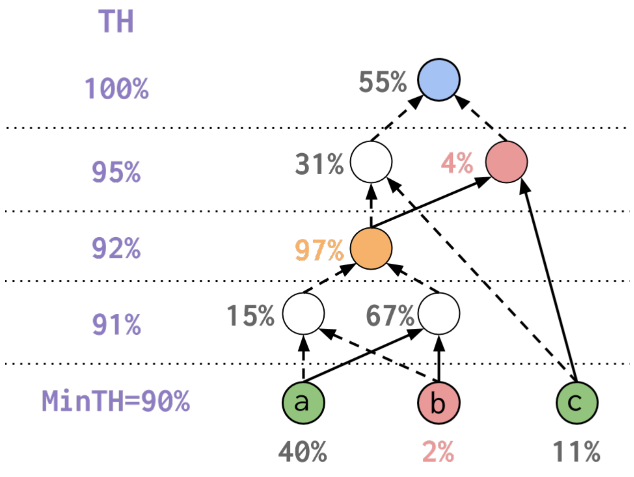
Figure 3.9: Probability and threshold values with FB-LD Method.



(a) Root.



(b) Linear.



(c) Exponential.

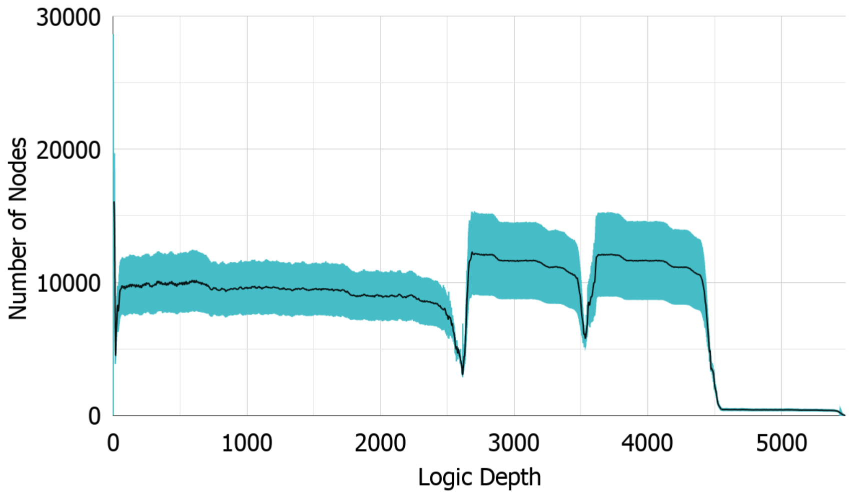
Source: The Author.

3.2.3.2 Number of Nodes

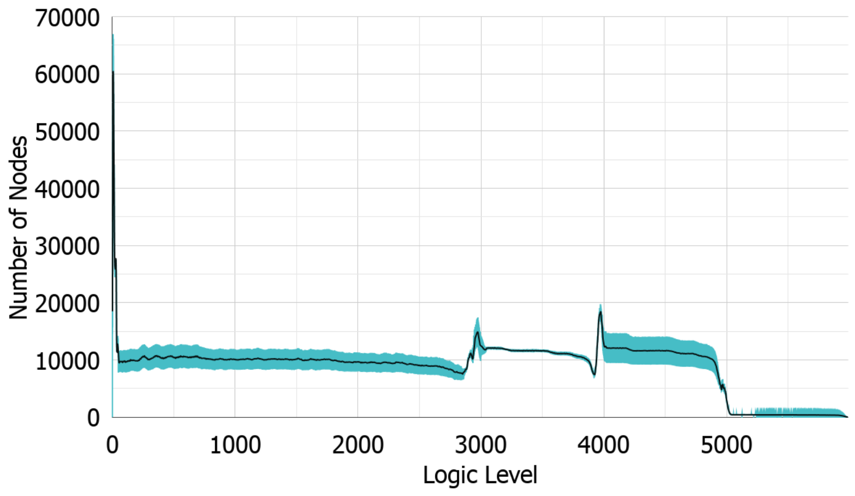
In this *functional based on the number of nodes per depth* (FB-NPD) method to set the nodes' TH, we count the number of nodes within the same LD. In other words, we count the set of nodes that are within the same LD domain. The intention behind this method is to preserve the connection between the NN's layers when it is represented in a digital AIG format. Figure 3.10 shows the number of nodes in each AIG level for the NNs and the RDF model, for Figures 3.10a and 3.10b we have a blue line which has a dense drawing in some areas. This unusual way of drawing the graph line happens because neighboring LDs have distant quantities of nodes. The line jumps up and down on really close points. With that, we also present a black line centering the actual number of nodes in each depth. This is the *trendline*, calculated based on a *moving average* window on the actual values in blue. The trendline will be used on future graphs to be shown, and its usage is demanding since when we insert multiple reduction lines in the same graph, they overlap too much with one another, but it doesn't occur with the trendline drawing.

The NN-2 and NN-100 are composed of three NN fully connected hidden layers and Figures 3.10a and 3.10b exposes them by the three valleys in the number of nodes in three regions of the AIG's LDs. Ideally, it is desired to make it less likely for these nodes close to the valleys to be turned into constants, as the information that is passing in between the layers is crucial and there is a higher risk of inserting a circuit error if turning those nodes as constants. In other words, the AND nodes that bridge the connections in between the NN's layers are the ones present in the those valleys. The presence of such valleys happens because when the NN is being compiled to a logic circuit, each neuron will be turned into multiple amounts of interconnected AND nodes, increasing the graph's overall height and depth. In contrast, it doesn't happen as much on the synapses, which are the connections among layers and are composed of constant weight values. In other words, each layers' neurons will expand to a larger amount of interconnected AND nodes, while the synapses in between the layers will not grow as large. The nodes present in such valleys are LDs with only a few nodes and should be carefully turned to constants since they will tend to fan-out to a high amount of nodes, and if an error is present in such nodes, the error would be propagated to its fan-outs.

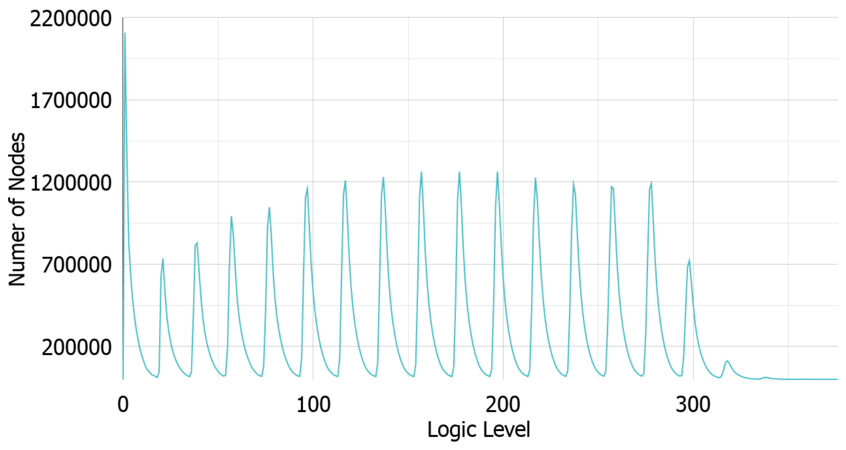
Figure 3.10: Number of Nodes in each LD for the Proposed NNs and RDF.



(a) NN-2.



(b) NN-100.



(c) RDF.

Source: The Author

That being said, for LDs containing a high amount of nodes, this method sets their TH to smaller values while for LDs with a lower amount of nodes, their TH is set to larger values, attempting to preserve unmodified the LDs with a lower amount of nodes. The LD in an AIG that contains the least number of nodes will always receive a TH value of 100%. Since deeper LDs usually have a low number of nodes, the idea behind the previous method which was based in the LD itself is partially maintained, since they will automatically be given higher TH values. This can be seen on the graphs in Figure 3.10, in the three models the longest LDs are always composed of only a few nodes. This portion of the AIGs closer to outputs represents the soft-max region of its representation as a NN and the soft-max is responsible for classifying the features. This is a commonly used classification technique in NNs, and we may safely assume that this method also uses the concept from the previous method.

This method's function to calculate the LDs' THs is presented in Equation 3.2. The variables α and *MinTh* are the same as in the previously presented method, being the α responsible for the function's curvature and *MinTh* the minimal TH. *LargestLevel* is the LD that contains the most number of nodes, and *NodesInLevel* is the number of nodes in the Level that is to define a TH value. The same three distributions of THs as the previous method are used for this one. They are shown in Figure 3.8. We are keeping the nodes in the same LD with also the same TH, although they don't follow a growth with the LD's value like in the previous method, TH values end up being disordered concerning the LDs. In the previous method, the TH values calculated with the three distributions would always follow the same shape as in Figure 3.8. In the present method, this is not true, since the number of nodes in an LD has no relation with its ordering. However, it is related to its NN structure. Remember that we are attempting to keep the nodes that represent the borders of each layer less modified.

The three different distributions to explore and diversify the TH values calculated for the LDs follows the same principle as in the previous method. The Root distribution is the one that is more conservative, establishing higher THs overall, and the Exponential distribution is a looser one, establishing lower THs and consequently setting more nodes as constants overall.

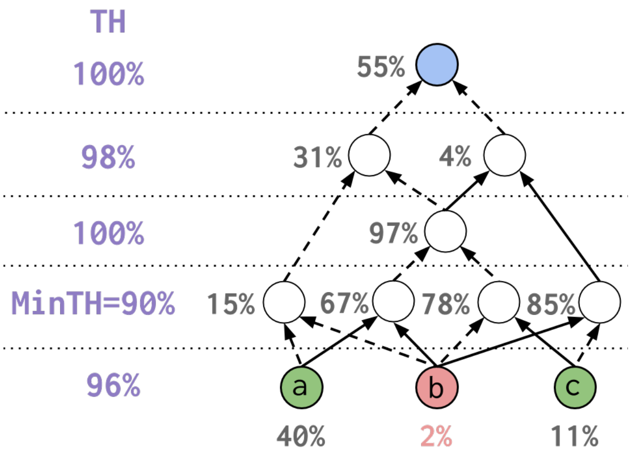
$$\begin{aligned} TH(Level) = \Bigg[(1 - MinTh) * \Bigg[-\left(\frac{NodesInLevel}{LargestLevel - 1}\right)^{\alpha} + \\ + \left(\frac{LargestLevel}{LargestLevel - 1}\right)\Bigg]\Bigg] + MinTh; \end{aligned} \tag{3.2}$$

The figures presented in 3.11 demonstrates the different TH values given to each

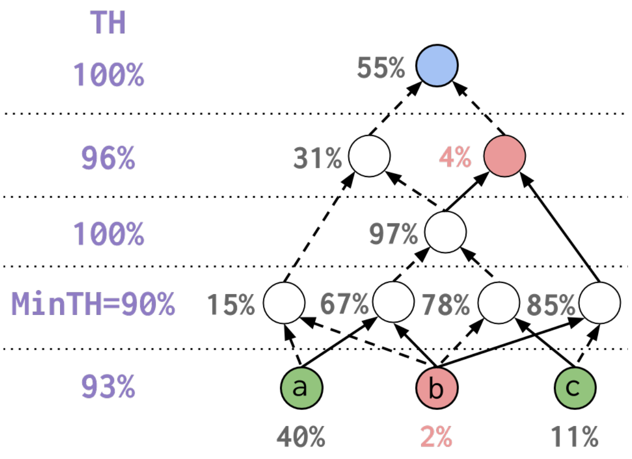
LD with each distribution and the constants that were set with this method in an XOR example. The AIG used as illustration was modified slightly with the addition of two extra nodes, so it looks more like the structure from a real NN in AIG. The hypothetical probabilities are kept unchanged, aside from the two extra nodes added with new probability values. In the previous method, the minimal TH would always be in the first LD; in other words, it was always the PIs that received the minimal TH. In the present method, any LD might end being the one with the lowest TH possible and different LDs may have the same TH, if they end up having the same number of nodes. In the example from Figure 3.11, there is two LDs with the same TH of 100%. The LD which will receive the largest TH value could also be anywhere in the AIG, although it usually is a PO like in the previous method or at least present in the soft-max section of the AIG since those positions usually have a low number of nodes, as shown in Figure 3.10.

The AND node with 97% signal probability is the only one present in its LD than it receives the maximum TH of 100%. In other methods that targeted this node, it was always set as a constant, and its signal probability was kept intact. In the present method, which is based on the number of nodes in an LD, this node was not set as a constant. Even though this AIG was modified to be more similar to a NN, remember that a NN layer would never map to a single LD in its AIG representation. If for example, we would have a NN with one input layer, two hidden layers, and one output layer, transforming this NN in its AIG representation we would have a denser portion of the AIG with several LDs to represent each hidden layer of neurons. Although we would have some LDs with only a few nodes, as can be seen in Figure 3.10 denoted by the valleys in the graphs. The valleys in the real NNs are represented by the node with 97% signal probability in Figure 3.11 and we would wish to keep it unchanged, since it represents the connections among the NN's layers and the example presented shows that is what happens by not turning the node into a constant, even with a high signal probability.

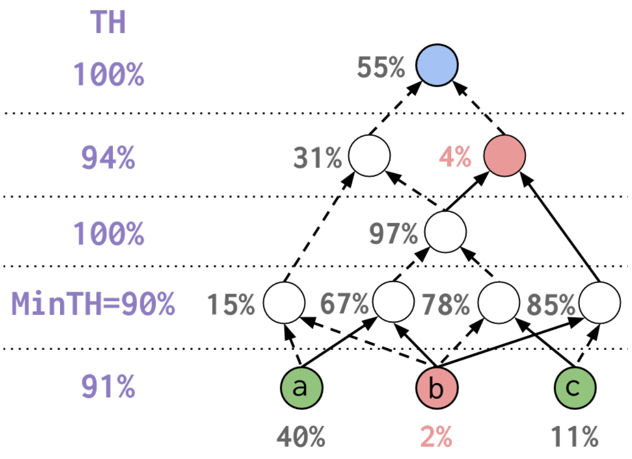
Figure 3.11: Probability and threshold values with the FB-NPD method.



(a) Root.



(b) Linear.



(c) Exponential.

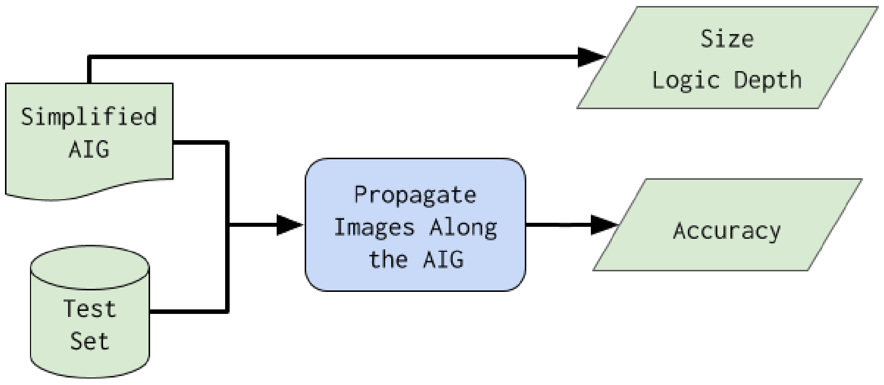
Source: The author.

In the sections to come we present some explanations of how our experiments were realized. Following an explanation of the metrics used to evaluate each of the proposed methods. The results obtained with experiments done with two NNs and one RDF are then presented in the next chapter.

3.3 Description of Experiments

We assign two NNs and one RDF designs, in their AIG representation, under simplification with our four proposed methods, which were explained in Section 3.2. For each design, four methods were used to reduce their complexity, and for the last three methods, the same set of threshold values were used for the three AIGs, while for the first method, it is required a different set of TH values for the simplification to take place. After a simplification is applied to an AIG (as is shown in Figure 3.5), we need to check what is the new accuracy, the number of ANDs, and the LD for the modified circuit, where the last two measurements are retrieved right away with the AIG file. Figure 3.12 presents a flowchart on the process of extracting such information. For the new accuracy value extraction, we take the image test set jointly with the simplified AIG file and propagate all the images by simulation. This process is repeated for every different TH value used in every method. For example, if we want to check 10 different TH values in the four methods, the process presented in Figure 3.12 will be repeated 40 times, after each respective simplification process. Simulations are done with the MNIST test set composed of 10 thousand images, and the AIG composed of about 50 million nodes. Each image on the dataset requires a simulation under this large AIG. To mitigate this large processing requirement the simulation process was implemented by making use of the binary representation of the data structure's attribute used for the AIG's signals. We made use of the fact that a *long long int* in C++ has 64 bits and enabled the processing of 64 images at each of the program's iterations. By processing one image at a time, our implementation was taking around 7 and 25 days to process the test and training set respectively, while it took 2 and 13 hours respectively after the implementation using all the bits for the signal attribute.

Figure 3.12: AIG's data extraction after simplification.



Source: The Author.

3.4 Used Metrics

Considering the proposed methods to set constants to reduce the AIGs, it is not a trivial issue to define which is the best one. Even the definition of what *best* is, is something unclear. Some applications would require the NN's accuracy to be as high as possible. At the same time, other applications could be more flexible and accept a reduction in its accuracy if a reduction in its complexity would also be present. Even the definition of what value is acceptable to trade between accuracy versus circuit size and LD is uncertain. With that in mind, we present two metrics in an attempt to find a good trade-off in complexity reduction of the proposed designs with the cost of some , or even none, accuracy loss.

3.4.1 Accuracy Loss - Interpolation

To attempt and explore which of the methods gives a better result, we shall make interpolations to estimate a spot in common that all the methods would have. This spot in common is for the same accuracy value reached by the methods while iteratively reducing the AIGs complexity. As the methods set different THs among the nodes, we can't expect a common data point in accuracy. To estimate that we take, for each accuracy curve, the two accuracy data points which are neighbors to an accuracy value that is 1% less than the model's original accuracy. With this two points we can define a line equation among them and find the depth and size estimated when the accuracy would reach exactly 1% less than its original value. It is as if we would trace a vertical line in this accuracy data point and find the LD and ANDs values for that accuracy value. With that, we may have a fair comparison among the methods proposed with an acceptable accuracy loss of 1%.

3.4.2 Figure of Merit

In an attempt to give a better understanding of which method presented better results, we also propose a *Figure of Merit*(FoM) metric. An FoM is a performance metric sometimes used in a broad range of different research areas, usually attempting to estimate a balance or a trade-off for resulting values from experiments. Some examples of FoM usages would be: the thermoelectric FoM used to measure the maximum efficiency of a thermoelectric material (HICKS; DRESSELHAUS, 1993), different FoMs are used to compare designs such as analog-to-digital converters (WALDEN, 1999), voltage-controlled oscillators (KINGET, 1999), or to measure power dissipation on oscillators (GAO et al., 2009), just to cite a few FoM usages.

To measure and explore a good trade-off between AIG reductions in depth, size, and accuracy loss, we compare the FoM values calculated among the methods proposed in the sections to come. To calculate a method's M Figure of Merit $FoM(M)$, Equation 3.3 is presented, where $TestError$ is the percentage of misses by the NN after simplification, in other words the complement value for the accuracy: $TestError = 1 - TestAccuracy$, $NumberOfAnds$ the percentage of remaining AND nodes and $LogicDepth$ the percentage decrease in the AIG's depth. Notice that the variable $TestError$ is squared. This is because we want it to weigh evenly on the equation in comparison to the other two variables. In other words, the accuracy loss weights the same for the FoM calculation as both the number of ANDs and the LD together.

$$FoM(M) = TestError^2 * NumberOfAnds * LogicDepth \tag{3.3}$$

With the proposed FoM we would be looking for the lowest possible values, since we would want to keep not only the error low but also the size and depth, then smaller FoM values means a better trade-off. We compute the FoM values with all THs used on each method. The interpolation at 1% lost accuracy, and the FoM calculation shall both be used to compare and analyze which of the methods presented a better simplification result. Notice that each of the Functional Based methods has three different TH distributions, and we also have the OPI and the AN methods, resulting in eight different manners to set the TH of nodes in the AIG to be simplified.

4 RESULTS

In this chapter, we present the results obtained with experiments done with the granted NNs and RDF in their logical AIG representation. For each method, it will be presented three graphs, one for each circuit. The measurements presented are the accuracy degradation for the test set only, the total number of AND nodes present in the AIG and its LD after the removal of such nodes. The horizontal axis in the graphs represents the TH or minimal TH values used for the experiments. For each graph, the AIG's original measurements are shown in a data point on the extreme right, since the original AIG size and LD is the starting point of 100%, a drop in such values can be seen on the data point right next to the one indicating the original measurements. In other words, the original values appear as an extrapolation on the horizontal axis, going past the value of 1.0.

4.1 Only Primary Inputs

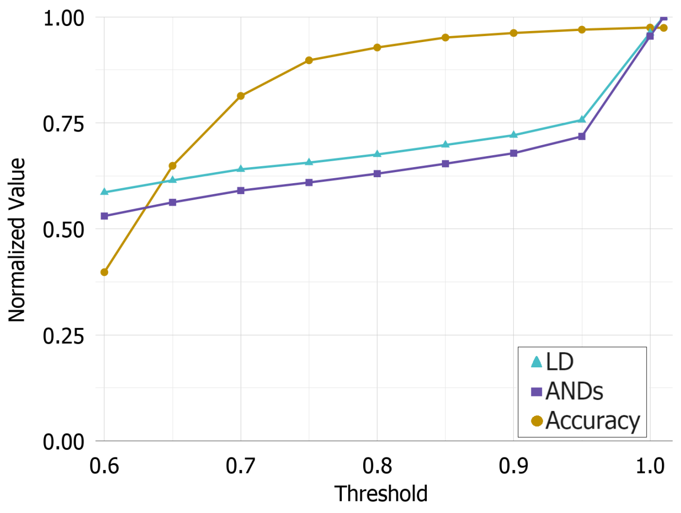
The method based on only the primary inputs is the simplest of them all; it targets only the AIG's PIs to possibly become constants before propagation. Figures 4.1a, 4.1b and 4.1c presents the results for the NN-2, NN-100 and RDF respectively with the OPI method. In dark yellow, there is the NN's accuracy. In light blue, the AIG's LD, in purple, the number of ANDs, and the data points represent the results for each experiment performed with different TH values. Lower TH values are needed in this method for the accuracy to have an abrupt drop, and we can see that for low TH values, starting at around 75% down to 60%, the accuracy degrades a lot and the reduction in the number of ANDs and LD doesn't pay off with such low accuracy. Since it targets only PI nodes, this method modifies a number of nodes much smaller than other methods for equal TH values.

By comparing the overfitted NN-100 and the NN-2, which is not overfitted, we can clearly see that this method is not influenced by the NNs' overfitting since the results are almost the same. Finally, it can be seen a sound reduction in the number of ANDs and LD without any evident accuracy loss at least on TH values from around 80% up to 100%. The RDF model was barely reduced by the OPI method, Figure 4.1c demonstrates that since the LD and AND nodes curves have a small decline with higher TH values, but don't seem to reduce when using lower TH values. This indicates that removable nodes would be located further away from PI nodes. The next methods to be presented can reach those further away nodes, as it will be seen more significant reductions with the

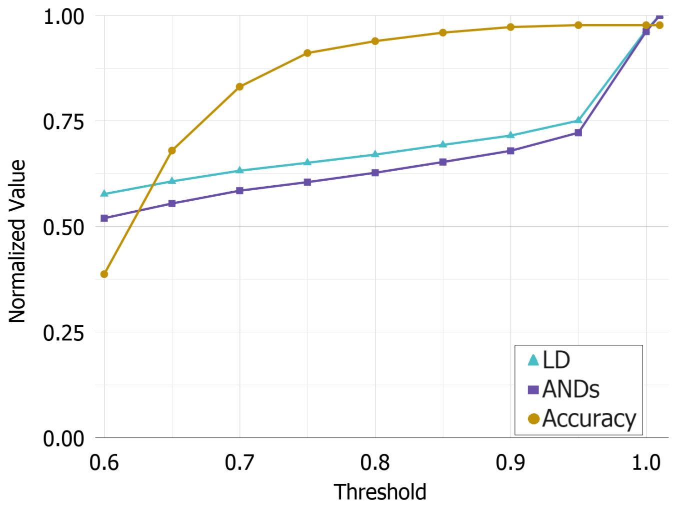
RDF model.

Figures 4.2a, 4.2b and 4.2c presents the positions in which nodes are begin removed from the AIGs. The curves presented for NN-2 and NN-100 had to be drawn as *trendlines* with a moving average calculation, the actual brute values are unable to be analyzed, because neighboring LDs have distant values and the drawing of the lines results in overlapping with one another. The graphs' horizontal axes refer to the AIG's LDs and the vertical axes the number of nodes present in each LD. The topmost curve in each graph refers to the original model in AIG format, synthesized after training, and the following downward lines are for the TH values used to simplify the AIGs. In each graph's legend, it is shown the corresponding TH and the accuracy achieved after simplification for such TH. The overlapping occurring on the lines is because the THs used to result in LDs with the same number of nodes as the original values, meaning that such LDs suffered no reduction. We can clearly see that for all the three simplified models, no matter the TH used, the OPI method was only able to remove nodes nearby the AIG's PIs. Notice that the two NNs had nodes removed only at its first fully connected layer, meaning that no propagated constant was able to reach after the NN's first layer. We may also notice a similar but still different behavior for the overfitted and non-overfitted NNs. While for the RDF model, the lines shown are not trendlines, since there was no need to. The RDF had only a subtle change by the OPI method, meaning that the constants propagation reach is limited for the RDF model, only nodes in its very first depths were possibly removed by this method.

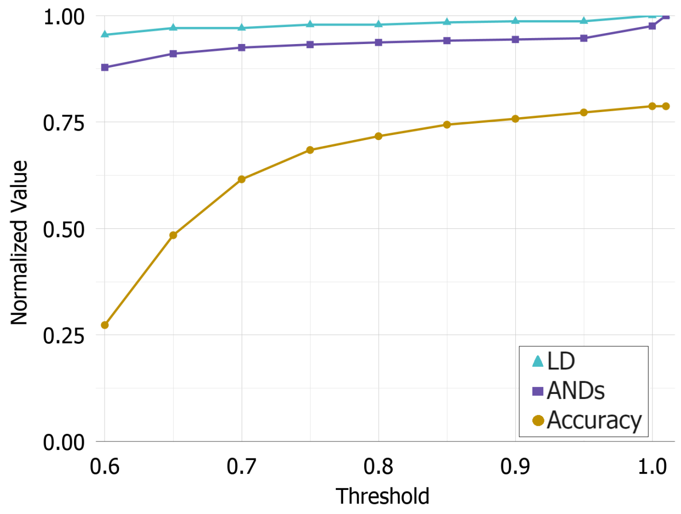
Figure 4.1: Accuracy, logic depth and number of AND nodes for the OPI method with fixed THs.



(a) NN-2.



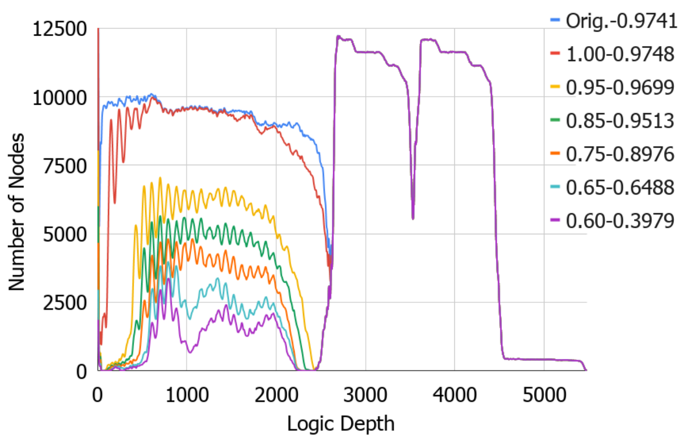
(b) NN-100.



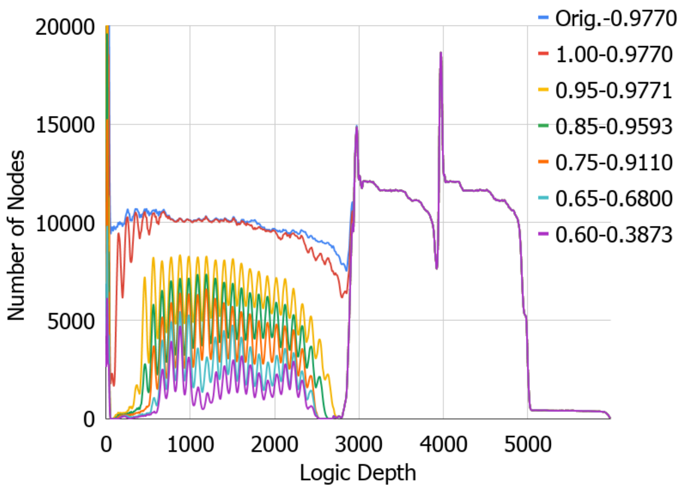
(c) RDF.

Source: The Author.

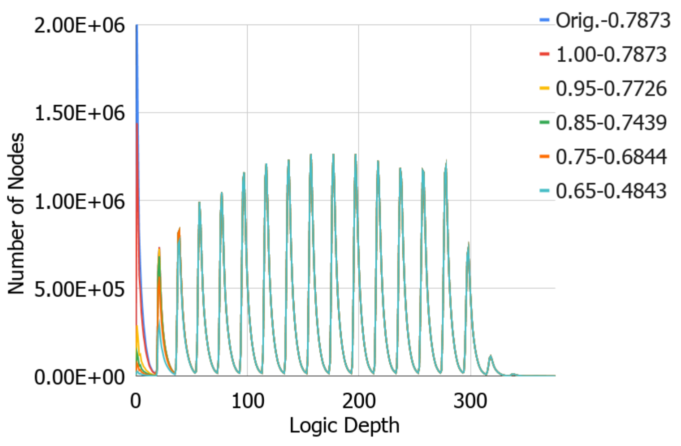
Figure 4.2: Number of AND nodes in each LD with different TH values for the OPI method.



(a) NN-2.



(b) NN-100.



(c) RDF.

Source: The Author.

4.1.1 100% Threshold

Table 4.1 presents the AIGs that were reduced in LD and number of ANDs from their original values with a TH of 100% to select PI nodes to be turned into constants. When we use a TH of 100%, we are selecting the PI nodes that were never set to 1 with the training set simulation. When we take into consideration a TH 100%, it is the safest approach to reduce the AIG. Obviously, these instances had their training accuracy intact, while we can even see an insignificant increase in their test accuracy, as shown in Table 4.1. In other words, with a TH of 100%, this method was able to make the AIGs slightly less complex with absolutely no adverse cost in accuracy.

Table 4.1: Simplification measurements with 100% TH for the OPI method.

		NN-2	NN-100	RDF
Original Acc.(%)		97.41	97.7	78.72
100% TH Acc.(%)		97.48	97.7	78.73
Only PIs	LD (%)	96.2	96.5	100.0
	ANDs (%)	95.5	96.2	97.5

Source: The Author.

4.1.2 1% Accuracy Loss

Interpolations were applied to estimate the LD and number of ANDs when test accuracy dropped by 1% in each AIG. The interpolation calculation was explained in Section 3.4.1. Then, for an accuracy loss of 1%, we present Table 4.2, it shows the values achieved of size and LD when the three models reach exactly 1% accuracy in the test set. Clearly, the RDF design was significantly less altered by this method. This is because the RDF unnecessary nodes are closer to its PO nodes, and the constant propagation might not reach them, this behavior can be clearly seen by comparing the graphs in Figure 4.1. In Figure 4.1c specifically, we can see that even removing only a few nodes with a TH as low as 80% down to 60%, the accuracy was already able to go degrade considerably. This shows that unnecessary nodes in the RDF model should be further away from lower LDs. On the other hand, the NNs had a great reduction in their size and depth, with a loss in test accuracy of only 1%. Notice that for the three AIGs, we are setting constants only on some of their PIs, and we are still able to remove about a million of internal nodes in the NNs by propagating the constants.

Table 4.2: Simplification measurements at 1% accuracy loss for the OPI method.

		NN-2	NN-100	RDF
Original Acc.(%)		97.41	97.7	78.72
1% Loss Acc.(%)		96.41	96.7	77.72
Only PIs	LD (%)	73.0	70.6	99.1
	ANDs (%)	68.9	66.8	95.6

Source: The Author.

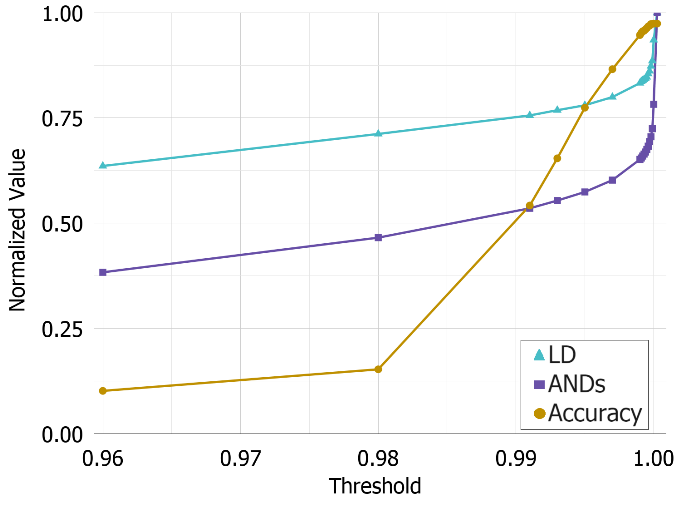
4.2 All Nodes

With this method not only PI nodes may be targeted to become constants but also any internal AND nodes. Also, the TH value is the same for *all nodes* each time the AIG is simplified. The results for this method are presented in Figures 4.3a, 4.3b and 4.3c, the circuits follow the same order as the method presented previously.

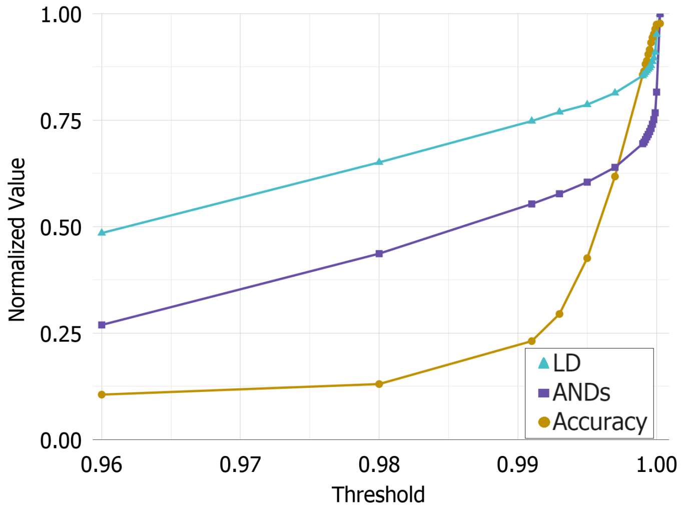
It can be noticed that this method is influenced by the NN's overfitting, unlike the previously presented method, the difference can be seen due to faster degradation of the overfitted model when comparing the accuracy drop in Figure 4.3a with the accuracy drop in Figure 4.3b. This happens because as the NN is overfitted, it is more sensitive to modifications since it is too closely fit the training set. An overfitted NN is expected to be missing input samples if it is even slightly modified because it ends up memorizing the training set instead of learning its behavior.

Overall this method can remove a noticeable portion of the AIG by analyzing the node's probabilities from the training set. Although, the TH values have to be precise to not degrade the accuracy too much. Notice that the accuracy stays high only for the TH values above 99% for the NNs. Interestingly the RDF model had close to half of its AND nodes removed already with a TH of 100%, and it removes even more with close TH values, this means that the RDF model had a lot of nodes with high probability values and the simplification method chose them to be removed already with high TH values. Also, the previous method was barely able to remove nodes from the RDF model, meaning that removable nodes are located further away from initial logic depths, since the present method may target them as constants.

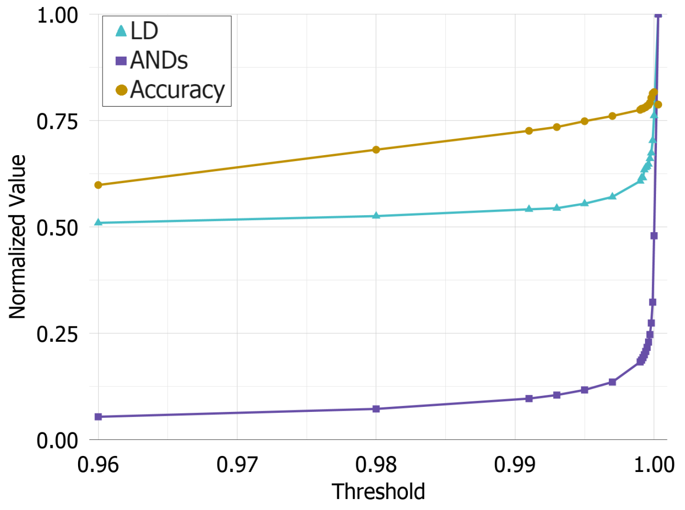
Figure 4.3: Accuracy, logic depth and number of AND nodes for the AN method with fixed THs.



(a) NN-2.



(b) NN-100.



(c) RDF.

Source: The Author.

4.2.1 100% Threshold

As was stated for the previous method, this one may also turn into constants all the nodes that had the same value for all images in the training set. This set of nodes is represented by the simplification under the TH of 100%, such simplification leaves the training set accuracy unaffected, and it rendered the reduction values presented in Table 4.3, percentages are given based on original values. This method has a considerably larger reduction without any cost in accuracy due to the TH value of 100% when compared to the previous method since it is considering all the nodes instead of only the PIs. As for the modification in test accuracy achieved with the TH that assures the safest removal of nodes, there was a noticeable improvement for the RDF model accuracy. It is hard to give an exact explanation as to how this happens, although it may probably have to do with the model's structure definition, before its conversion to a boolean representation.

Table 4.3: Simplification measurements with 100% threshold probability value for the AN method.

		NN-2	NN-100	RDF
Original Acc.(%)		97.41	97.7	78.72
100% TH Acc.(%)		97.41	97.45	81.66
All Nodes	LD (%)	93.5	95.1	76.1
	ANDs (%)	78.2	81.6	47.9

Source: The Author.

4.2.2 1% Accuracy Loss

The interpolation to estimate the reduction in LD and number of ANDs when the accuracy degrades by 1% for the AN method is presented at Table 4.4. For comparison, it also presents the results for the interpolation with the previous method. By comparing the AN and OPI methods on the overfitted model NN-100, we may notice that the previous method removed more nodes and depths until reaching 1% less accuracy. Also, comparing NN-2 and NN-100 on the AN method, there is a considerable difference in reductions. The overfitted model is sensitive to modifications, when targeting the whole AIG to be modified. For the RDF model, the AN method was able to reduce a lot of its size with the cost of losing only 1% accuracy.

Comparing the 1% loss in accuracy mark from the previous method, NN-2 and NN-100 presented better results overall, with a higher reduction for the LD values. Even by targeting only PI nodes, the OPI method was able to remove considerably more nodes and depth before reaching the 1% mark for the overfitted model. At the same time, for the NN-2, it had an even reduction in ANDs but considerably more reduction in LD. This shows that better choosing which nodes to remove is crucial to achieving better results. One might guess that the present method would always be the best when compared to the previous one since it targets more nodes, but the results presented in the 1% mark proves that this is not true and targeting fewer nodes with more precision is better than targeting more nodes without as much mindfulness.

Table 4.4: Simplification measurements at 1% accuracy loss for the OPI and AN methods.

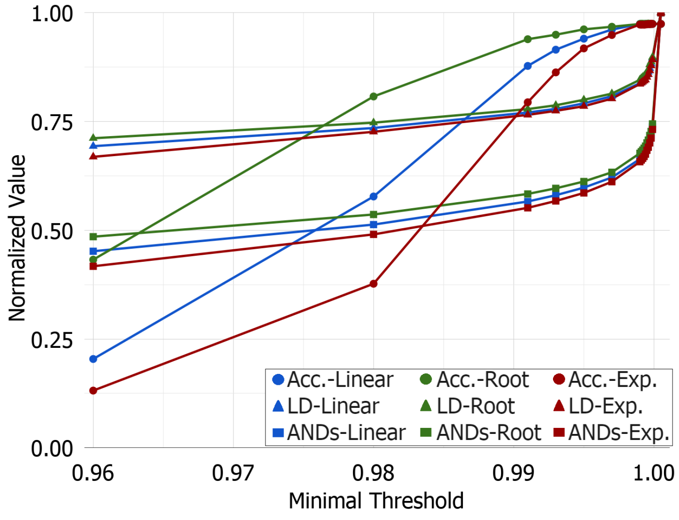
		NN-2	NN-100	RDF
Original Acc.(%)		97.41	97.7	78.72
1% Loss Acc.(%)		96.41	96.7	77.72
Only PIs	LD (%)	**73.0**	**70.6**	99.1
	ANDs (%)	68.9	**66.8**	95.6
All Nodes	LD (%)	85.6	96.5	**61.5**
	ANDs (%)	**68.5**	86.8	19.2

Source: The Author.

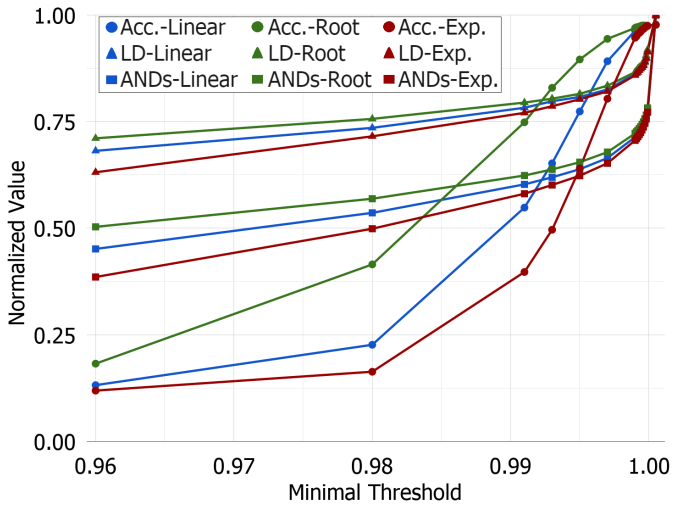
4.3 Functional Based - Logic Depth

The results for the FB-LD method are presented in Figures 4.4a, 4.4b and 4.4c. The THs given to nodes by the FB-LD method are variable and based on the node's LD. For the results with this method, the TH values in the horizontal axis are a minimal value meaning that for such simplification, the AIG nodes received THs ranging from the minimal value up to 100%. Longer depths mean higher THs, and shorter depths mean lower THs. You will see three curves for each AIG; these curves are related to the distributions presented in Figure 3.8 with each color related to each different distribution. They are used to set different TH values for this method and analyze the different behavior for each. At first glance, one would say that the green curves are always the best ones since they preserve the accuracy, although that might not be true every time since the other distributions have a quicker drop in accuracy, they remove more nodes than the green one. A trade-off in accuracy loss and reduction must be evaluated.

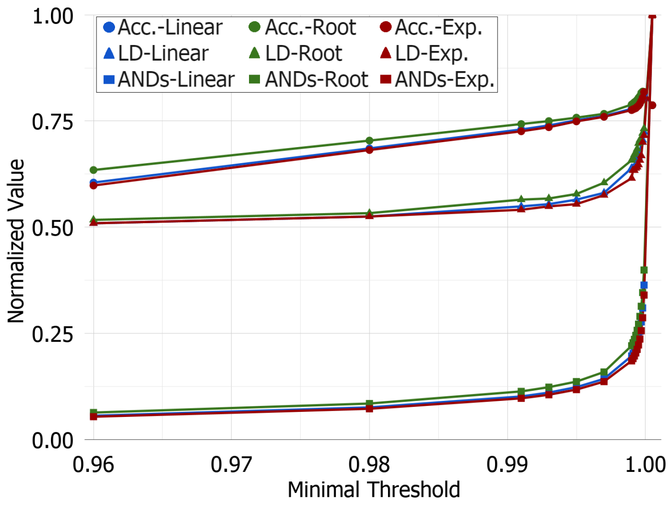
Figure 4.4: Accuracy, logic depth and number of AND nodes for the FB-LD method with minimal THs.



(a) NN-2.



(b) NN-100.



(c) RDF.

Source: The Author.

4.3.1 1% Accuracy Loss

The overfitted NN had a quicker degradation on the accuracy, like in the previous method, this is true for each of the different distributions. The overall behavior is similar to the previous method with fixed THs, although, for this method, we get more reduction in the AIG's size and depth with better accuracy preservation. To show that this is true, Table 4.5 is presented. From Table 4.5, we can see the reduction for each of the AIGs for each distribution with the FB-LD method when its accuracy reached 1% less than the original value. The results from the previous method are also shown for comparison. Notice that the accuracy for both methods is the same, but all the three variations of the FB-LD method presented a better reduction concerning the removal of AND nodes from all models. Although, this is not true only for the overfitted model where the OPI method still presents a better performance for depth and size reduction among all methods presented so far. The OPI method actually showed a better performance when removing depths from both NNs. Concerning the RDF model, a small improvement in size but a deterioration in depths removed may be seen when comparing the FB-LD with AN. The root distribution is the one with the most reduction achieved when reaching 1% less accuracy.

Table 4.5: Simplification measurements at 1% accuracy loss for the OPI, AN and FB-LD methods.

		NN-2			NN-100			RDF		
Original Acc.(%)		97.41			97.7			78.72		
1% Loss Acc.(%)		96.41			96.7			77.72		
Only PIs	LD (%)	**73.0**			**70.6**			99.1		
	ANDs (%)	68.9			66.8			95.6		
All Nodes	LD (%)	85.6			96.5			**61.5**		
	ANDs (%)	68.5			86.8			19.2		
Distribution		Root	Lin.	Exp.	Root	Lin.	Exp.	Root	Lin.	Exp.
Logic Depth	LD (%)	80.6	81.5	82.4	86.3	87.2	88.1	62.9	63.4	63.0
	ANDs (%)	**62.1**	63.1	64.1	**71.8**	72.4	73.7	**18.8**	19.2	18.9

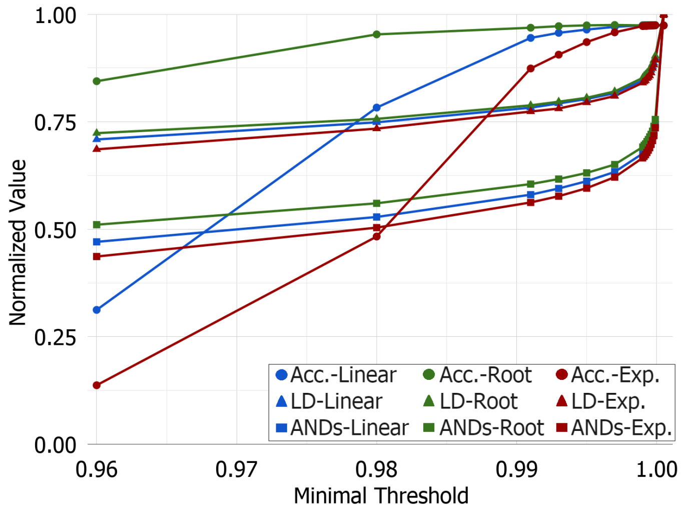
Source: The Author.

4.4 Functional Based - Number of Nodes Per Depth

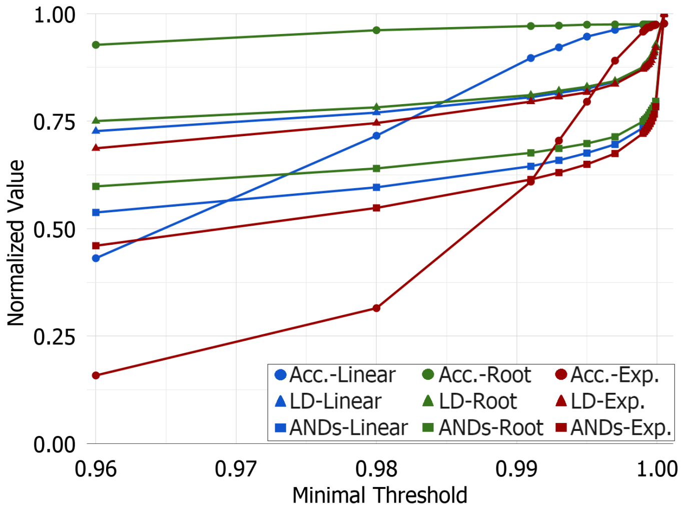
This method also sets variable THs on nodes, ranging from a minimum value up to 100% among them. Although, this method sets the TH based on the *number of nodes* concentrated on an LD. All the nodes on the same depth will all receive the same TH value. Depths containing more nodes will receive higher THs, and depths containing fewer nodes will receive lower THs. The depth with the most number of nodes will always receive a TH of 100%, and the depth with the least concentration of nodes will receive the minimal TH value, the intermediate depths among those two will receive THs based on the distribution used, as of Figure 3.8.

Figures 4.5a, 4.5b and 4.5c presents the accuracy loss, the LD and ANDs reduction for simplifications with this method. At first glance, one would say that for such a method, the overfitted accuracy didn't behave like the previous methods presented, at least for the green curve, but don't be mistaken, notice that the size and depth also have a slower reduction. With the metrics and measurements to be presented next, it is shown that the present method also behaves like the previous ones for the overfitted model, meaning that NN-100 accuracy still is more sensitive to modifications in its structure.

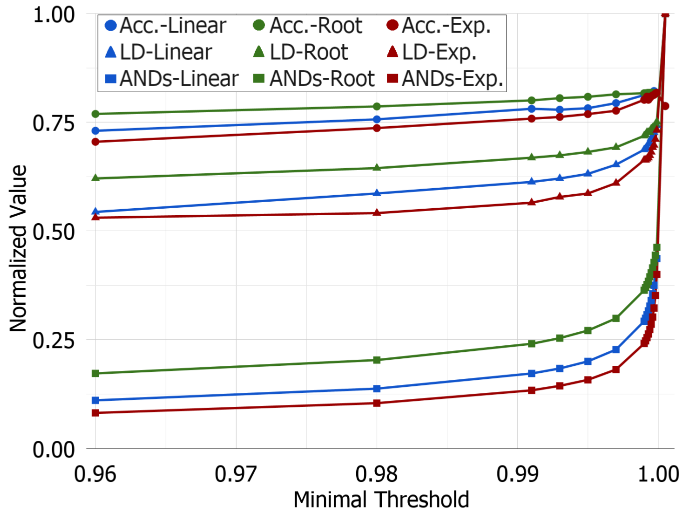
Figure 4.5: Accuracy, logic depth and number of AND nodes for the FB-NPD method with minimal THs.



(a) NN-2.



(b) NN-100.



(c) RDF.

Source: The Author.

4.4.1 1% Accuracy Loss

Table 4.6 presents the interpolation estimation for the present method, including the previous ones for comparison. All methods present the same accuracy for each model, which is a reduction of exactly 1% less from its original accuracy. With Table 4.6, we show a comparison for all of the proposed methods based on a 1% accuracy loss marker. The overfitted model (NN-100) actually responded better with the first and most simple method of all, the OPI method, with a reduction in size and depth considerably lower than the other models, losing only to the FB-NPD method by a small amount in size reduction. The OPI method also had a better reduction than all the other methods for the LD reduction with the non-overfitted model (NN-2). Aside from the first method, all the other ones modify a lot of nodes from the AIG's structure, since it may target all nodes.

In contrast, the first method is more subtle when making modifications; for this reason, we would expect that the most simple of all methods would actually perform better with an overfitted model. On the other hand, an overfitted NN usually is not what one would be looking for when designing a NN because they tend to not be able to generalize on unseen input data. Still, if an overfitted NN is available, the first method is the one to go for, when attempting to make the NN less complex to be implemented as a chip.

Table 4.6: Simplification measurements at 1% accuracy loss for all methods.

		NN-2			NN-100			RDF		
Original Acc.(%)		97.41			97.7			78.72		
1% Loss Acc.(%)		96.41			96.7			77.72		
Only PIs	LD(%)	**73.0**			**70.6**			99.1		
	AND(%)	68.9			66.8			95.6		
All Nodes	LD(%)	85.6			96.5			61.5		
	AND(%)	68.5			86.8			19.2		
Distribution		Root	Lin.	Exp.	Root	Lin.	Exp.	Root	Lin.	Exp.
Logic Depth	LD(%)	80.6	81.5	82.4	86.3	87.2	88.1	62.9	63.4	63.0
	AND(%)	62.1	63.1	64.1	71.8	72.4	73.7	18.8	19.2	18.9
Nodes P. Depth	LD(%)	77.9	80.2	82.3	79.8	85.4	87.9	63.2	**60.9**	61.1
	AND(%)	**59.1**	61.1	64.0	**66.0**	71.1	73.3	18.7	**16.7**	18.3

Source: The Author.

Aside from the overfitted NN, the other two models (NN-2 and RDF) had better results with the method based on the number of nodes in each LD (FB-NPD). Even with the same accuracy value, this method was able to reduce the size of the non-overfitted NN and the RDF model reasonably more than the other proposed methods. By comparing the present method with the OPI method, we went from a reduction in the size of 68.9% to 59.1% from the original size for the NN-2 just by choosing better candidate nodes to become constants based on the NN structure. We had a reduction in the size of 40% for both NNs and a reduction of 83.3% for the RDF model by conceding only 1% of accuracy.

In the current Subsection, we presented an analysis of the behavior of the proposed methods regarding a common ground among them. This common ground being the 100% TH usage, which concerns the actual constants determined by the probability calculation, and the 1% accuracy loss marker, being an estimation calculated on interpolations for the resulting experiments. We also presented each method's behavior individually and for the TH values used for the experiments. With that, we were able to determine that the definition of the best method is dependent on what the reduction objective is, for example, the OPI method performing better for LD reduction and the FB-NPD method performing better for a reduction in the number of AND nodes. In summary, we gradually presented the methods individually and used metrics that they have in common for comparison. On the other hand, in the next subsection, we shall demonstrate a general view of the methods to compare and investigate their behaviors with the models.

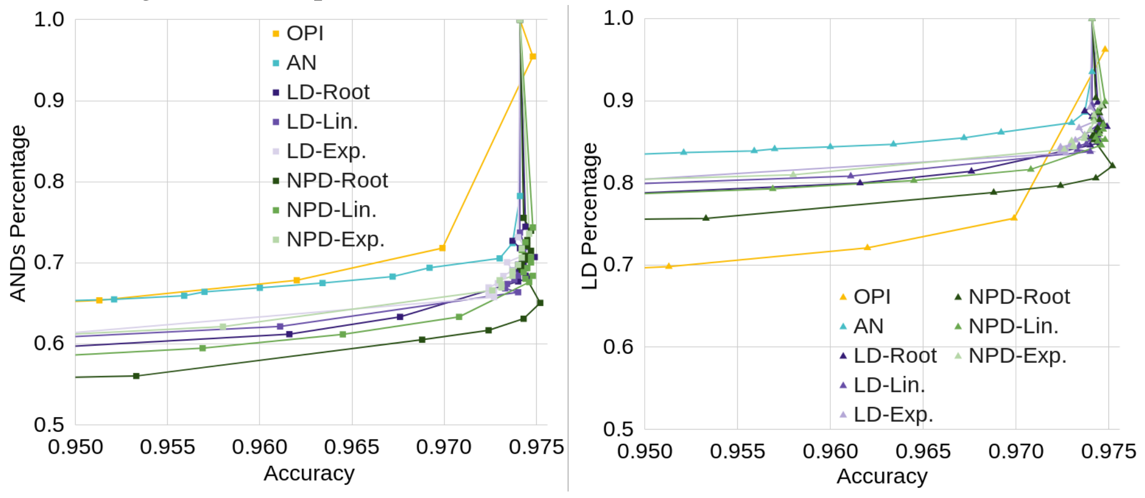
4.5 General Comparison of the Proposed Methods

Instead of presenting each method one by one as the previous subsection, we may also present the experimental data for all methods in a single graph. Figures 4.6, 4.7 and 4.8 presents the test accuracy values (horizontal axis) with respect to LDs and number of ANDs (vertical axis) achieved after simplifications, values are shown as percentage values from the original AIG prior to simplification. These graphs reinforce the assertion of the best method for each objective, as in the metrics used in the previous subsection, but also shows the overall behavior of the methods based on different TH values used on experiments. For these graphs, the TH values are not shown. Notice that we have the accuracy values in the horizontal axes, and the data points don't align with one another, but still, the overall behavior of each method can be seen with the lines drawn, exposing the simplification behavior of the proposed methods. We have the OPI method in yellow, the AN method in light blue, the FB-LD method with its three distributions in color tones of purple, and the FB-NPD in color tones of green, the Root distribution, which is usually the best one is put as a darker tone.

We would be interested in the lines that were able to achieve lower values of LD and number of ANDs, in other words, the best results are the ones that go as low as possible while remaining to the right. The test accuracy values achieved go no further than 95% since we would be looking for a simplification that doesn't degrade the accuracy too much. The original accuracy values for the models may be seen on top of each graph, closer to the right side. The test accuracy for the original models is 97.41% for NN-2, 97.7% for NN-100, and 78.72% for RDF. In some situations, the accuracy was improved after simplification, and this happens by chance simply. The increase in the test accuracy is clearly more evident with the RDF (Figure 4.8).

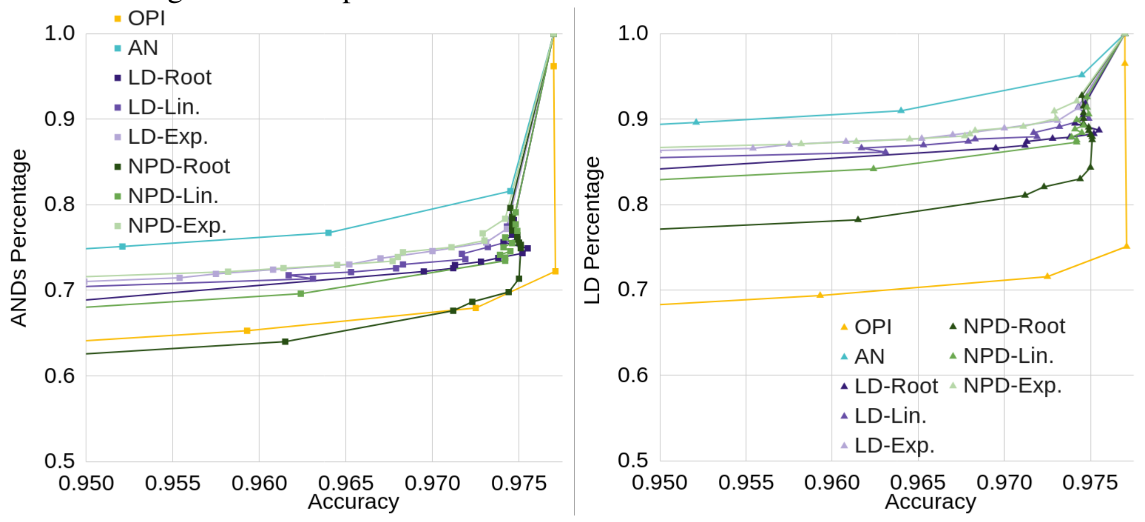
If we analyze the graphs NN-2 and NN-100, we may notice that the darker tones for the functional-based methods tend to be lower than their counterpart distributions in lighter color tones. This means that the root distribution is, most of the time, the best one to go for. It is the most conservative one when setting TH values since it will always give higher TH values for the nodes than its counterpart distributions the linear and exponential distributions.

Figure 4.6: Simplification measurements with all methods for the NN-2.



Source: The Author.

Figure 4.7: Simplification metrics with all methods for the NN-100.

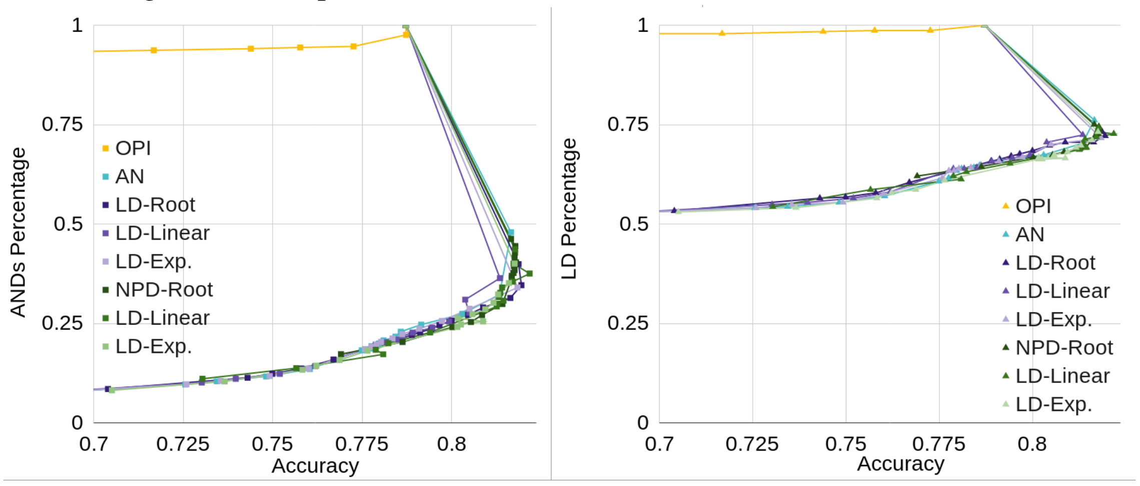


Source: The Author.

Concerning the reduction in LD for the AIGs, remember that the LD of an AIG is defined by the longest path going from any PI to any PO. Then, if during simplification, we remove a node that is a member of such path, automatically, the AIG's LD will be reduced by one. One situation that frequently happens during the usage of the OPI method is the removal of all nodes in a depth, this can be seen on Figures 4.2a and 4.2b where the lines reach the bottom of the graph, meaning that no nodes were remaining for such depth. With the removal of a whole logic depth, we would not only reduce the size of the longest path but all the paths present in the AIG. As for the reduction in AND nodes in the circuit, it is a straight forward process, where a node is removed, and its size is reduced.

The OPI method is, most of the time, the best one to remove LDs from the AIGs, at least for NN-2 and NN-100. On the other hand, it is the worst for removing nodes from the non-overfitted model NN-2. We can also see a consistent superiority with the OPI method when reducing NN-100 for both metrics. The worst method in bot NN models was by far the AN one, that targets all nodes of the AIG at the same time, with a fixed TH. We can also see a general improvement with both functional based methods concerning the AN method. For both NNs we have the following order going top to bottom on the graphs: first, the light blue lines (AN), followed by the purple lines (FB-LD) and at last the green lines (FB-NPD) for both LD and ANDs reduction, with the OPI method varying in between them. In general, the FB-NPD with the root distribution demonstrated a better performance than the other methods, being closer to the bottom than the others, most of the time, resulting in a more significant reduction in LD and AND nodes with similar accuracy values than other methods.

Figure 4.8: Simplification measurements with all methods for the RDF.



Source: The Author.

Concerning the RDF model, we have the graph presented in Figure 4.8, the accuracy shown for the reduction experiments go as far as 70%, with an original accuracy of 78.72%. The RDF model presented an at least interesting behavior when under simplification with the proposed methods. The OPI method wasn't able to present considerable reductions in LD and AND nodes. Although, the other methods were able to present a significant improvement in the model's accuracy, with an improvement of about 10% from its original accuracy. This improvement in accuracy reflects imperfections on the model's synthesizing process, meaning that there are a lot of present AND nodes that are disturbing the model's inference capability. Without saying that, we were able to reduce the AIG's size by less than 10% of its original size and still preserve its accuracy with values close to the original one, an enormous reduction. This reflects that an RDF model is suitable for simplifications under approximate computing.

When comparing the methods under the RDF model, the difference presented among them isn't as expressive as in the NN models. Although, in the previous subsection we saw that we have subtle differences among them, and the FB-NPD method performed better than others when reducing both metrics, at least when we settle an accuracy loss of 1%.

4.5.1 Trade-off Among Methods

So far, we considered our analyzes on the behavior of the proposed methods distinctly for the LD and the number of ANDs the AIG presented with relation to accuracy values achieved after the process of simplification. Though, for the current subsection, we shall consider analyzes for the FoM metric explained in Subsection 3.4.2. The FoM metric gives us the ability to find the data point from experiments realized that presents the best trade-off for accuracy loss in contrast to the reductions achieved in size and depth. In other words, we will be analyzing the reductions achieved in the number of ANDs and LD jointly, in the sense of a mixed objective, for example, as if we wished to reduce both area and delay of a circuit at the same time.

The FoM values are calculated with Equation 3.3 previously presented. Such an equation gives a doubled weight for the accuracy; this way, the accuracy metric weights the same as both the LD and ANDs removed. We are also looking for the lowest value as possible, since we wish to reduce the error, altogether with the LD and number of ANDs in the AIG. Tables 4.7, 4.8 and 4.9 presents the lowest values achieved for the FoMs calculated for NN-2, NN-100 and the RDF respectively. The best method, as indicated by the FoM values, is given by the lowest value calculated for each method. In the presented tables, we are showing only the lowest FoM value for each method, while we omit the other values since we are interested in finding which method is the best one in each situation model. For each FoM value, we also present the logic depth, the number of ANDs, the accuracy values achieved after simplification, and also the TH value used to achieve that values. Notice that the best trade-offs occur with relatively high probability THs, usually around 99%.

Table 4.7: Simplification measurements for the lowest FoM achieved in each method with NN-2.

	OPI	AN	LD Root	LD Lin.	LD Exp.	NPD Root	NPD Lin.	NPD Exp.
LD(%)	75.69	88.57	84.89	83.81	87.83	82.06	84.60	85.36
AND(%)	71.81	72.44	66.85	66.37	71.87	63.33	67.61	69.34
ACC(%)	96.99	97.37	97.45	97.40	97.46	97.52	97.46	97.37
Lowest FoM (10^{-4})	5.00	4.44	3.76	3.76	4.03	**3.28**	3.69	4.04
TH(%)	95.00	99.99	99.91	99.90	99.98	99.70	99.90	99.94

Source: The Author.

The method that responded the best for NN-2 was the method FB-NPD with the root distribution, as it is presented by Table 4.7. As in the previous subsection, this method presented itself as the best one to reduce the non-overfitted NN represented in AIG. We were able to maintain the test accuracy intact (with a minor increase) while reducing its size to 63.33% and depth to 82.06%, definitively an excellent reduction result. Even though the OPI method is the best one to reduce an AIG's LD, as was demonstrated in the previous subsection, the OPI method presented the worst trade-off when considering the three metrics: accuracy, size, and depth altogether, with the highest FoM value.

Table 4.8: Simplification measurements for the lowest FoM achieved in each method with NN-100.

	OPI	AN	LD Root	LD Lin.	LD Exp.	NPD Root	NPD Lin.	NPD Exp.
LD	75.09	95.14	88.70	90.07	91.32	84.34	87.29	92.10
ANDs	72.23	81.59	74.25	75.99	77.45	69.60	73.45	79.13
ACC	97.71	97.45	97.55	97.49	97.43	97.50	97.42	97.42
Lowest FoM (10^{-4})	**2.84**	5.05	3.99	4.31	4.66	3.76	4.27	4.80
TH	95.00	100	99.96	99.98	99.99	99.70	99.90	99.99

Source: The Author.

Considering the overfitted NN-100, its lowest FoM data points information is presented in Table 4.8. The OPI method is by far the best one to reduce the overfitted model, with an FoM value considerably lower than all other methods, keeping its accuracy intact, while reducing the AIG's LD to 75.09% and number of ANDs to 72.23%, a pretty decent reduction, considering no accuracy was lost. This is expected since the overfitted model is the most sensible one to modifications, and the OPI method applies modifications only at the NN's first layer. The FB-NPD with the root distribution also has a noticeably low FoM value, with an at least acceptable result. Obviously, AN presented itself as the worst method when considering all metrics at the same time, since it modifies the AIG without much mindfulness.

Table 4.9: Simplification measurements for the lowest FoM achieved in each method with RDF.

	OPI	AN	LD Root	LD Lin.	LD Exp.	NPD Root	NPD Lin.	NPD Exp.
LD	100	52.52	53.32	52.52	52.52	62.07	54.38	53.05
ANDs	97.56	7.21	7.58	7.58	7.58	11.07	11.07	11.07
ACC	78.73	68.12	70.40	68.56	68.18	76.91	73.04	70.51
Lowest FoM (10^{-4})	441.4	38.47	39.63	39.30	38.57	57.09	43.74	**37.72**
TH	100	98.00	98.00	98.00	98.00	96.00	96.00	96.00

Source: The Author.

In Table 4.9 it is presented the lowest FoM values achieved for each method with the RDF model. This model presented higher FoM values than the NNs overall because its accuracy is lower even for its original value of 78.72%. The OPI method had a hard time reducing the RDF model and presented a considerably high value for the trade-off calculated with the FoM metric, barely reducing the size of the AIG and unable to reduce its depth. But even in the worst method, it was able to remove some nodes and not degrade the model's accuracy. The method that presented the best trade-off in reductions versus accuracy loss for the RDF model was the FB-NPD method with the exponential distribution, which was able to maintain only 11% of the AIG nodes, a reduction in LD cut in half and still hold on to an accuracy of 70.5%. This restates the susceptibility that an RDF model has to be reduced with the usage of approximate computing.

5 CONCLUSION

This dissertation addressed a range of issues concerning the implementation of hardware to process neural networks, also known as neuromorphic hardware. The difficulty for implementing neuromorphic hardware is a known issue by the community, mainly due to scalability, power consumption, and processing capacity required by neural networks to train and infer on input data. Nonetheless, neural networks have shown outstanding inference performance, for example, classifying input data, realizing predictions, and decision making. These capabilities brought by such artificially intelligent electronic sources should be exploited to its fullest, always attempting to achieve its fullest potential. Proposing solutions to make it easier for integrated circuits designers to implement intelligent devices is crucial for the healthier development of such technology. With the reductions proposed by us, the circuit's final area will be reduced due to less AND nodes present on its boolean representation and reduce its delay due to shorter logic depth. In the previous chapter, a discussion was made attempting to compare the results obtained and demonstrate which method should be used, depending on the desired objective, or even which method presented a better trade-off.

5.1 Contributions Summary

The contributions for the work presented herein are focused on the boolean representation of neural networks, aimed at the logic synthesis stage of the design flow of integrated circuits. More specifically, we proposed four different approaches to determine nodes to be removed in an AND-inverter graph with the usage of signal probability and approximate computing paradigm employing constant signal propagation.

Our contributions include the probability calculation of signal probabilities of nodes in an AIG representation for a neural network. The probability calculation is based on the NN's training set, which gives a sizeable representation of the circuit's input behavior. Such a probability calculation process is not provided by current EDA tools, which make use of conventional heuristics for signal probability definition.

By comparing the proposed approaches, we may conclude that it is critical to perform a meticulous selection of nodes to be removed from the NN represented as an AIG. We have shown that superior accuracy preservation and even further reductions may be achieved with such a line of thought. Another critical remark to be mentioned is the fact that the simplest of all methods can remove more logic depths than other methods, although not reducing the AIG as much in size.

The Functional Based on Nodes per Depth method was consistently pointed out as the best option to reduce the nodes in the AIGs with a controlled loss in accuracy. They are proving that a proper selection of nodes based on the structure of the model may achieve further improvements to the reductions performed. In other words, we may achieve vast reductions in the size and depth of the AIG representation of a machine learning model based on approximate computing without any considerable loss in accuracy if appropriately done since the results shown by the AN method, for example, are considerably less attractable.

5.2 Future Work

Researching improvements in the implementation of neuromorphic hardware shouldn't stop. Neural networks are consistently improving, and the hardware to supplement them should attempt to keep track of such development. That being said, we present some recommended directions to research on the follow up of the present work:

- For the FB-LD method, instead of using the nodes' LD to set the TH, use the shortest distance from a node to any PO. This way is avoiding to set low THs on nodes close to a PO.
- Implement another method to set THs based on the NN layers. This method would have to find which AIG nodes belong to which layer and set THs based on that, maybe iteratively simplify the AIG layer by layer. The OPI method, for example, occasionally ends up only removing nodes from the first NN layer.
- Apply NNs optimizations, such as pruning and quantization techniques, to evaluate how susceptible the AIG would be to reductions after going through severer optimization during learning stages.
- Further optimizations and even new ways to implement neuromorphic hardware should be pursued and are always welcome.

REFERENCES

AKOPYAN, F. et al. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 34, n. 10, p. 1537–1557, 2015.

Amarú, L.; Gaillardon, P.; De Micheli, G. Majority-inverter graph: A new paradigm for logic optimization. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 35, n. 5, p. 806–819, May 2016. ISSN 1937-4151.

ANGLADA, M. et al. MASkIt: Soft error rate estimation for combinational circuits. In: **2016 IEEE 34th International Conference on Computer Design (ICCD)**. [S.l.: s.n.], 2016. p. 614–621.

ARDAKANI, A. et al. VLSI implementation of deep neural network using integral stochastic computing. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, v. 25, n. 10, p. 2688–2699, Oct 2017.

BAGCHI, S. **Artificial Neural Networks & their ACTIVATION FUNCTIONS**. 2019. Available from Internet: <https://medium.com/@shamitb/activation-function-is-any-non-linear-function-applied-to-the-weighted-sum-of-the\-inputs-of-a-a4326956cebf>. Accessed in: 1 Jan. 2020.

BAI, Y. et al. Finding tiny faces in the wild with generative adversarial network. In: **2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition**. [S.l.]: IEEE, 2018. p. 21–30.

BARTLETT, K. et al. Bold: A multiple-level logic optimization system. In: **International Conference on Computer Aided Design**. [S.l.: s.n.], 1987.

Berkeley Logic Synthesis and Verification Group. **ABC: A System for Sequential Synthesis and Verification**. 2019. Available from Internet: <http://www.eecs.berkeley.edu/~alanmi/abc/.html>. Accessed in: 1 Jan. 2020.

BERNDT, A. A. S. **This project's source code**. 2020. Available from Internet: <https://github.com/gudeh/AIG_ultimate>. Accessed in: 1 Jan. 2020.

BERNDT, A. A. S. et al. Reduction of neural network circuits by constant and nearly constant signal propagation. In: **Proceedings of the 32nd Symposium on Integrated Circuits and Systems Design**. New York, NY, USA: Association for Computing Machinery, 2019. (SBCCI '19). ISBN 9781450368445. Available from Internet: <https://doi.org/10.1145/3338852.3339874>. Accessed in: 1 Jan. 2020.

BIERE, A. The AIGER and-inverter graph (AIG) format version 20071012. **FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr**, v. 69, p. 4040, 2007.

BISHOP, C. M. et al. **Neural networks for pattern recognition**. [S.l.]: Oxford university press, 1995.

BRAYTON, R. K.; HACHTEL, G. D.; SANGIOVANNI-VINCENTELLI, A. L. Multilevel logic synthesis. **Proceedings of the IEEE**, v. 78, n. 2, p. 264–300, Feb 1990. ISSN 1558-2256.

BRAYTON, R. K. et al. Mis: A multiple-level logic optimization system. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 6, n. 6, p. 1062–1081, November 1987. ISSN 1937-4151.

BRYANT, R. E. Graph-based algorithms for boolean function manipulation. **Computers, IEEE Transactions on**, IEEE, v. 100, n. 8, p. 677–691, 1986.

CALVERT, B. D.; MARINOV, C. A. Another k-winners-take-all analog neural network. **IEEE Transactions on Neural Networks**, v. 11, n. 4, p. 829–838, July 2000. ISSN 1941-0093.

CARREIRA-PERPINAN, M. A.; IDELBAYEV, Y. "learning-compression" algorithms for neural net pruning. In: **2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition**. [S.l.: s.n.], 2018. p. 8532–8541. ISSN 1063-6919.

CHANDRASEKHARAN, A. et al. Approximation-aware rewriting of aigs for error tolerant applications. In: **2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. [S.l.]: IEEE, 2016. p. 1–8.

CHATTERJEE, S. Learning and memorization. In: **International Conference on Machine Learning**. [S.l.: s.n.], 2018. p. 755–763.

CHATTERJEE, S.; MISHCHENKO, A. Circuit-based intrinsic methods to detect overfitting. **arXiv preprint arXiv:1907.01991**, 2019.

CHATTERJEE, S. et al. Reducing structural bias in technology mapping. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 25, n. 12, p. 2894–2903, Dec 2006. ISSN 1937-4151.

CHEN, T. et al. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In: ACM. **ACM Sigplan Notices**. [S.l.], 2014. v. 49, n. 4, p. 269–284.

COATES, A. et al. Deep learning with cots hpc systems. In: **International conference on machine learning**. [S.l.: s.n.], 2013. p. 1337–1345.

COURBARIAUX, M.; BENGIO, Y.; DAVID, J.-P. Training deep neural networks with low precision multiplications. **arXiv preprint arXiv:1412.7024**, 2014.

COURBARIAUX, M.; BENGIO, Y.; DAVID, J.-P. **BinaryConnect: Training Deep Neural Networks with binary weights during propagations**. 2015.

COVELL, M. et al. Table-based neural units: Fully quantizing networks for multiply-free inference. **arXiv preprint arXiv:1906.04798**, 2019.

DARRINGER, J. A. et al. Logic synthesis through local transformations. **IBM Journal of Research and Development**, v. 25, n. 4, p. 272–280, July 1981. ISSN 0018-8646.

DEBOLE, M. V. et al. Truenorth: Accelerating from zero to 64 million neurons in 10 years. **Computer**, v. 52, n. 5, p. 20–29, May 2019. ISSN 1558-0814.

DENG, L. The mnist database of handwritten digit images for machine learning research. **IEEE Signal Processing Magazine**, v. 29, n. 6, p. 141–142, Nov 2012. ISSN 1558-0792.

EPFL. **The EPFL Logic Synthesis Libraries**. 2020. Available from Internet: <https://libalice.readthedocs.io/en/latest/?badge=latest>. Accessed in: 1 Jan. 2020.

FLAQUER, J. et al. Fast reliability analysis of combinatorial logic circuits using conditional probabilities. **Microelectronics Reliability**, v. 50, p. 1215–1218, 09 2010.

FRANCO, D. T. et al. Signal probability for reliability evaluation of logic circuits. **Microelectronics Reliability**, Elsevier, v. 48, n. 8-9, p. 1586–1591, 2008.

FRIEDMAN, J.; HASTIE, T.; TIBSHIRANI, R. **The elements of statistical learning**. [S.l.]: Springer series in statistics New York, 2001.

GAO, M. et al. Approximate computing for low power and security in the internet of things. **Computer**, v. 50, p. 27–34, 01 2017.

GAO, X. et al. Jitter analysis and a benchmarking figure-of-merit for phase-locked loops. **IEEE Transactions on Circuits and Systems II: Express Briefs**, v. 56, n. 2, p. 117–121, Feb 2009. ISSN 1558-3791.

GARLAND, J.; GREGG, D. Low complexity multiply-accumulate units for convolutional neural networks with weight-sharing. **ACM Transactions on Architecture and Code Optimization (TACO)**, ACM New York, NY, USA, v. 15, n. 3, p. 1–24, 2018.

GUPTA, S. et al. Deep learning with limited numerical precision. In: **International Conference on Machine Learning**. [S.l.: s.n.], 2015. p. 1737–1746.

GYSEL, P. **Ristretto: Hardware-Oriented Approximation of Convolutional Neural Networks**. 2016.

GYSEL, P. et al. Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks. **IEEE Transactions on Neural Networks and Learning Systems**, v. 29, n. 11, p. 5784–5789, Nov 2018. ISSN 2162-2388.

HAASWIJK, W. et al. Deep learning for logic optimization algorithms. In: IEEE. **2018 IEEE International Symposium on Circuits and Systems (ISCAS)**. [S.l.]: IEEE, 2018. p. 1–4.

HAMERLY, R. et al. Large-scale optical neural networks based on photoelectric multiplication. **Phys. Rev. X**, American Physical Society, v. 9, p. 021032, May 2019. Available from Internet: <https://link.aps.org/doi/10.1103/PhysRevX.9.021032>. Accessed in: 1 Jan. 2020.

HAN, J.; ORSHANSKY, M. Approximate computing: An emerging paradigm for energy-efficient design. In: **2013 18th IEEE European Test Symposium (ETS)**. [S.l.]: IEEE, 2013. p. 1–6.

HAN, S. et al. Eie: efficient inference engine on compressed deep neural network. **ACM SIGARCH Computer Architecture News**, ACM New York, NY, USA, v. 44, n. 3, p. 243–254, 2016.

HAN, S.; MAO, H.; DALLY, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. **arXiv preprint arXiv:1510.00149**, 2015.

HAN, S. et al. Learning both weights and connections for efficient neural network. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2015. p. 1135–1143.

HASHEMI, S. et al. Understanding the impact of precision quantization on the accuracy and energy of neural networks. In: IEEE. **Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017**. [S.l.], 2017. p. 1474–1479.

HAYKIN, S. **Neural networks: a comprehensive foundation**. [S.l.]: Prentice Hall PTR, 1994.

HE, K.; SUN, J. Convolutional neural networks at constrained time cost. In: **The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)**. [S.l.: s.n.], 2015.

HE, K. et al. Deep residual learning for image recognition. In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. [S.l.: s.n.], 2016. p. 770–778.

HELLERMAN, L. A catalog of three-variable or-invert and and-invert logical circuits. **IEEE Transactions on Electronic Computers**, EC-12, n. 3, p. 198–223, June 1963. ISSN 0367-7508.

HICKS, L.; DRESSELHAUS, M. S. Thermoelectric figure of merit of a one-dimensional conductor. **Physical review B**, APS, v. 47, n. 24, p. 16631, 1993.

HU, J.; SHEN, L.; SUN, G. Squeeze-and-excitation networks. In: **2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition**. [S.l.]: IEEE, 2018. p. 7132–7141.

HU, M. et al. Dot-product engine for neuromorphic computing: Programming 1T1M crossbar to accelerate matrix-vector multiplication. In: **2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)**. [S.l.: s.n.], 2016. p. 1–6. ISSN null.

HWANG, K.; SUNG, W. Fixed-point feedforward deep neural network design using weights +1, 0, and -1. In: **2014 IEEE Workshop on Signal Processing Systems (SiPS)**. [S.l.: s.n.], 2014. p. 1–6. ISSN 2162-3570.

Háleček, I.; Fišer, P.; Schmidt, J. Are xors in logic synthesis really necessary? In: **2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)**. [S.l.: s.n.], 2017. p. 134–139. ISSN 2473-2117.

IEEE Standards Association. **IEEE Standards Website**. 2019. Available from Internet: <https://standards.ieee.org/>. Accessed in: 1 Jan. 2020.

International Workshop on Logic and Synthesis. **IWLS 2019 Programming Contest: Legal AIGs**. 2019. Available from Internet: <http://www.iwls.org/iwls2019/>. Accessed in: 1 Jan. 2020.

JADERBERG, M.; VEDALDI, A.; ZISSERMAN, A. Speeding up convolutional neural networks with low rank expansions. **arXiv preprint arXiv:1405.3866**, 2014.

JO, S. H. et al. Nanoscale memristor device as synapse in neuromorphic systems. **Nano letters**, ACS Publications, v. 10, n. 4, p. 1297–1301, 2010.

KARNIN, E. D. A simple procedure for pruning back-propagation trained neural networks. **IEEE transactions on neural networks**, IEEE, v. 1, n. 2, p. 239–242, 1990.

KINGET, P. Integrated ghz voltage controlled oscillators. In: **Analog circuit design**. [S.l.]: Springer, 1999. p. 353–381.

KRIZHEVSKY, A.; HINTON, G. et al. **Learning multiple layers of features from tiny images**. [S.l.], 2009.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. Imagenet classification with deep convolutional neural networks. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2012. p. 1097–1105.

LECUN, Y. 1.1 deep learning hardware: Past, present, and future. In: IEEE. **2019 IEEE International Solid-State Circuits Conference-(ISSCC)**. [S.l.], 2019. p. 12–19.

LEE, N.; AJANTHAN, T.; TORR, P. H. Snip: Single-shot network pruning based on connection sensitivity. **arXiv preprint arXiv:1810.02340**, 2018.

LI, C. et al. Efficient and self-adaptive in-situ learning in multilayer memristor neural networks. **Nature communications**, Nature Publishing Group, v. 9, n. 1, p. 2385, 2018.

LIN, D.; TALATHI, S.; ANNAPUREDDY, S. Fixed point quantization of deep convolutional networks. In: **International Conference on Machine Learning**. [S.l.: s.n.], 2016. p. 2849–2858.

LIU, G.; ZHANG, Z. A parallelized iterative improvement approach to area optimization for lut-based technology mapping. In: **Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays**. New York, NY, USA: Association for Computing Machinery, 2017. (FPGA '17), p. 147–156. ISBN 9781450343541. Available from Internet: <https://doi.org/10.1145/3020078.3021735>. Accessed in: 1 Jan. 2020.

MATOS, J. M.; CARRABINA, J.; REIS, A. Efficiently mapping VLSI circuits with simple cells. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 38, n. 4, p. 692–704, 2018.

MEI, L. et al. Sub-word parallel precision-scalable mac engines for efficient embedded dnn inference. In: IEEE. **2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)**. [S.l.], 2019. p. 6–10.

MICHELI, G. D. **Synthesis and optimization of digital circuits**. [S.l.]: McGraw-Hill Higher Education, 1994.

MISHCHENKO, A. et al. Technology mapping into general programmable cells. In: ACM. **Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays**. [S.l.], 2015. p. 70–73.

MISHCHENKO, A.; CHATTERJEE, S.; BRAYTON, R. DAG-aware AIG rewriting: a fresh look at combinational logic synthesis. In: **2006 43rd ACM/IEEE Design Automation Conference**. [S.l.: s.n.], 2006. p. 532–535.

MISRA, J.; SAHA, I. Artificial neural networks in hardware: A survey of two decades of progress. **Neurocomput.**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 74, n. 1-3, p. 239–255, dec. 2010. ISSN 0925-2312. Available from Internet: <http://dx.doi.org/10.1016/j.neucom.2010.03.021>. Accessed in: 1 Jan. 2020.

MIT 6.S191. **Introduction to Deep Learning**. 2019. Available from Internet: <IntroToDeepLearning.com>. Accessed in: 1 Jan. 2020.

NAIR, V.; HINTON, G. E. Rectified linear units improve restricted boltzmann machines. In: **Proceedings of the 27th international conference on machine learning (ICML-10)**. [S.l.: s.n.], 2010. p. 807–814.

NAZEMI, M.; PASANDI, G.; PEDRAM, M. Energy-efficient, low-latency realization of neural networks through boolean logic minimization. In: **Proceedings of the 24th Asia and South Pacific Design Automation Conference**. New York, NY, USA: ACM, 2019. (ASPDAC '19), p. 274–279. ISBN 978-1-4503-6007-4. Available from Internet: <http://doi.acm.org/10.1145/3287624.3287722>. Accessed in: 1 Jan. 2020.

NIELSEN, M. A. **Neural networks and deep learning**. [S.l.]: Determination press San Francisco, CA, USA:, 2015.

NURVITADHI, E. et al. Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC. In: **2016 26th International Conference on Field Programmable Logic and Applications (FPL)**. [S.l.: s.n.], 2016. p. 1–4. ISSN 1946-1488.

NURVITADHI, E. et al. Can FPGAs beat GPUs in accelerating next-generation deep neural networks? In: ACM. **Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays**. [S.l.], 2017. p. 5–14.

POSSANI, V. et al. Unlocking fine-grain parallelism for aig rewriting. In: **2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. [S.l.: s.n.], 2018. p. 1–8. ISSN 1933-7760.

QIU, J. et al. Going deeper with embedded FPGA platform for convolutional neural network. In: **Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays**. New York, NY, USA: ACM, 2016. (FPGA '16), p. 26–35. ISBN 978-1-4503-3856-1. Available from Internet: <http://doi.acm.org/10.1145/2847263.2847265>. Accessed in: 1 Jan. 2020.

RUSSAKOVSKY, O. et al. Imagenet large scale visual recognition challenge. **International journal of computer vision**, Springer, v. 115, n. 3, p. 211–252, 2015.

RUSSELL, S. J.; NORVIG, P. **Artificial intelligence: a modern approach**. [S.l.]: Malaysia; Pearson Education Limited,, 2016.

Soeken, M. et al. Optimizing majority-inverter graphs with functional hashing. In: **2016 Design, Automation Test in Europe Conference Exhibition (DATE)**. [S.l.: s.n.], 2016. p. 1030–1035. ISSN 1558-1101.

SUNG, W.; SHIN, S.; HWANG, K. **Resiliency of Deep Neural Networks under Quantization**. 2015.

SZE, V. et al. Hardware for machine learning: Challenges and opportunities. In: IEEE. **2017 IEEE Custom Integrated Circuits Conference (CICC)**. [S.l.], 2017. p. 1–8.

ULLRICH, K.; MEEDS, E.; WELLING, M. Soft weight-sharing for neural network compression. **arXiv preprint arXiv:1702.04008**, 2017.

VENKATARAMANI, S.; ROY, K.; RAGHUNATHAN, A. Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits. In: **2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.]: IEEE, 2013. p. 1367–1372.

WALDEN, R. H. Analog-to-digital converter survey and analysis. **IEEE Journal on Selected Areas in Communications**, v. 17, n. 4, p. 539–550, April 1999. ISSN 1558-0008.

WANG, L.-T.; CHANG, Y.-W.; CHENG, K.-T. T. **Electronic design automation: synthesis, verification, and test**. [S.l.]: Morgan Kaufmann, 2009.

WANG, Q. et al. Energy efficient parallel neuromorphic architectures with approximate arithmetic on FPGA. **Neurocomputing**, v. 221, p. 146 – 158, 2017. ISSN 0925-2312. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0925231216311213>. Accessed in: 1 Jan. 2020.

WEI, L. et al. Person transfer gan to bridge domain gap for person re-identification. In: **2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition**. [S.l.]: IEEE, 2018. p. 79–88.

WEN, W. et al. Learning structured sparsity in deep neural networks. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2016. p. 2074–2082.

WESTE, N. H.; HARRIS, D. **CMOS VLSI design: a circuits and systems perspective**. [S.l.]: Pearson Education India, 2015.

WIRTHLIN, M. J.; HUTCHINGS, B. L. Improving functional density through run-time constant propagation. In: **Proceedings of the 1997 ACM fifth international symposium on Field-programmable gate arrays**. [S.l.: s.n.], 1997. p. 86–92.

WU, Y.; QIAN, W. An efficient method for multi-level approximate logic synthesis under error rate constraint. In: **Proceedings of the 53rd Annual Design Automation Conference**. New York, NY, USA: ACM, 2016. (DAC '16), p. 128:1–128:6. ISBN 978-1-4503-4236-0. Available from Internet: <http://doi.acm.org/10.1145/2897937.2897982>. Accessed in: 1 Jan. 2020.

XU, B. et al. **Empirical Evaluation of Rectified Activations in Convolutional Network**. 2015.

XU, X. et al. Scaling for edge inference of deep neural networks. **Nature Electronics**, Nature Publishing Group, v. 1, n. 4, p. 216–222, 2018.

YANG, K. et al. Towards fairer datasets: Filtering and balancing the distribution of the people subtree in the imagenet hierarchy. **arXiv preprint arXiv:1912.07726**, 2019.

YAO, Y. et al. Approximate disjoint bi-decomposition and its application to approximate logic synthesis. In: **2017 IEEE International Conference on Computer Design (ICCD)**. [S.l.]: IEEE, 2017. p. 517–524.

YU, C.; CIESIELSKI, M.; MISHCHENKO, A. Fast algebraic rewriting based on and-inverter graphs. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 37, n. 9, p. 1907–1911, Sep. 2018. ISSN 0278-0070.

ZHANG, X. et al. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In: **The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)**. [S.l.]: IEEE, 2018. p. 6848–6856.

ZHOU, X. et al. Addressing sparsity in deep neural networks. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, 2018.

ZHU, C. et al. An efficient hardware accelerator for structured sparse convolutional neural networks on fpgas. **arXiv preprint arXiv:2001.01955**, 2020.

APPENDIX A — PROGRAMMING TOOLS AND TECHNIQUES

In this Appendix, the programming tools and techniques used to implement the software behind this project are presented. At first, the programming tools used are shown:

- The software that implements the procedures presented in this work was developed with the C++ programming language, with the usage of features from the 2011 version update (also known as C++11). The *objected oriented* paradigm was used, where *classes* were implemented following the AIGER format (BIERE, 2007). The AIG is implemented as a graph, where its object attributes are multiple lists composed of *node* objects, there is one list for each class that inherits from the node class, which are: *PIs*, *POs* and *ANDs*. Pointers are used to make connections from node to node. These pointers are the representation of the graph's edges. The node class has a pair of pointer attributes. Each one is a pointer to its two fanins. The node's fanouts are stored as a list of fanout pointers, the node's fanouts information may be disregarded in some situations, saving processing time and space if not stored. Even though latches are present in the AIGER format, they were omitted in the implementation, since we worked with combinational circuits only.
- The project's software development was written with the assistance of the NetBeans software, which is an integrated development environment. Its a helpful programming tool for developers, it has integration not only with the C++ language but also with *cmake* and *git*, that were used during the project software development.
- The *git* system was used, which is a free and open-source distributed version control system. Specifically, *GitHub* was used, which is a web-based version control that offers a free database to remotely store and administrate projects' source codes using the git system. The usage of such version control software was crucial for this work. Since the AIGs used are extremely large, they require a powerful computer to process them. For this reason, the code was developed in the personal computers available by the author, which are not computationally powerful enough. While the software execution was realized remotely in the computer server offered by the university's laboratory. The server was the only one among the available computers that can handle the large AIGs used on this project. The software source code is publicly available at (BERNDT, 2020).
- *Cmake* is a cross-platform tool for managing the build process of C++ coding by

using a compiler-independent method, and it was used during this project's development. It is a useful tool when developing large projects that integrate different sets of tools, sometimes even tools from different programming languages. It can relieve some work from developers when making use of distinct tools.

- The *Alice* tool was integrated into this project and is one of a handful sum of *application programming interface* directed to EDA tool development, provided by the *EPFL logic synthesis libraries*. Such libraries have diversified libraries for EDA developers, such as *exact synthesis library*, *quantum circuit synthesis library*, and among others. For the present project, we made use of Alice, which is a command shell library, it is a handful tool to give a better organization for the code and even to make a standardization of the program functionalities, allowing for a cleaner and easier implementation of the shell calling of the implemented program functions. (EPFL, 2020)

We also present a few programming techniques learned and used along with the software project development:

- To store the polarity of each AND node in the AIG, a programming trick was used. This trick consists of saving each node's polarity information in the least significant bit of each node's input pointer. Each node has two inputs fanins, which may be direct or negated, where a pointer is used to indicate them, and the first bit of such pointer is used to store the polarity information. This is done based on the assumption that every memory address is an even number, meaning that the last bit will always be zero when storing the pointer address. This is almost always true, but may not be accurate in some computer architectures and should be used with mindfulness.
- A parallelism paradigm was used to apply the dataset images through the AIGs. In the data structure implemented a *long long int* was used to store the signal information in each node, this data type has a 64-bit length. This allows for the processing of 64 images at the same time for each of the program's iterations, meaning that each bit stores the signal passing through the node for each of the 64 images. With that, the implemented program went from months to hours of processing time to execute all images along one of the AIGs, where the first version used a *boolean* type for the signal information and processed one image at a time.