UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

JUAN SUZANO DA FONSECA

Investigating The Use Of Approximate Computing On a Case-Study Neural Network implemented Into FPGA by using HLS

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Engineering

Advisor: Prof. Dr. Fernanda Kastensmidt

Porto Alegre
May 2022

“If I have seen farther than others,
it is because I stood on the shoulders of giants.”
— Sir Isaac Newton

ACKNOWLEDGMENTS

First, I would like to thank my mom, dad, stepmom, and my whole family for all the love, support, and dedication you gave me. My academic journey so far has been replete with big decisions and you have always supported me in my choices. Having you as a reference was essential for my personal and professional development. I would also like to thank my friends from Brazil and France for their support and company through this adventure.

I would like to thank my friend Prof. Dr. Fernanda Kastensmidt for all the work we developed together during my two years as a scientific initiation scholarship student, for everything you taught me, and for having advised me during the development of this work. I would also like to thank UFRGS and especially the Computer Science Institute and its professors for the excellent teaching and opportunities that were provided to me.

ABSTRACT

Neural networks have been used for all types of applications, ranging from stock market predictions to image recognition. They can be trained and synthesized into FPGAs using engines or entirely in parallel. However, implementing fully parallelized neural networks can be challenging due to the number of parameters and multiply-accumulate operations required. Optimization is very important to achieve area, power, and performance requirements. Approximate computation is a paradigm that aims at the tradeoff between the accuracy and cost of a computing operation. Several applications can be considered error-resilient. This means that they do not need 100% accurate operations to work correctly. In these cases, it is possible to make approximations in the operations performed, reducing the cost involved, and keeping the accuracy within acceptable limits. It can help optimize neural networks in terms of FPGA resources consumption. This work will investigate the benefits that the fixed-point data quantization technique can bring to the development of neural networks in FPGA.

Keywords: Convolutional neural network. approximate computing. FPGA.

Investigando o Uso de Computação Aproximada em Uma Rede Neural Convolucional Implementada em FPGA Utilizando HLS

RESUMO

Redes neurais tem sido utilizadas em diferentes aplicações, de previsões comportamentais do mercado de ações a reconhecimento de imagem. Elas podem ser treinadas, sintetizadas e implantadas em FPGA usando circuitos especializados ou de forma totalmente paralela. Implementar redes neurais totalmente paralelizadas pode ser desafiador devido ao número de parametros e operações de multiplicar-acumular exigidos. A otimização é muito importante para alcançar os requisitos de área, potência e desempenho. Computação aproximada é um paradigma que visa a troca entre a precisão e o custo de uma operação computacional. Várias aplicações podem ser consideradas resistentes a erros. Isto significa que elas não precisam de operações 100% precisas para funcionar corretamente. Nesses casos, é possível fazer aproximações nas operações realizadas, reduzindo o custo envolvido e mantendo a precisão dentro de limites aceitáveis. Isto pode ajudar a otimizar as redes neurais em termos de consumo de recursos da FPGA. Este trabalho investigará os benefícios trazidos pela técnica de quantização em ponto fixo para o desenvolvimento de redes neurais em FPGA.

Palavras-chave: Rede neural convolucional, computação aproximada, FPGA.

LIST OF ABBREVIATIONS AND ACRONYMS

ANN	Artificial Neural Networks
BNN	Binary Neural Network
BRAM	Block RAM
c.c.	Clock Cycles
CNN	Convolutional Neural Network
DSP	Digital Signal Processing Units
DRUM	Dynamic Range Unbiased Multiplier
FF	Flip-Flop
HDL	Hardware Description Language
HLS	High-level synthesis
LUT	Look-Up-Table
MVTU	Matrix-Vector-Threshold Unit
MNIST	Modified National Institute of Standards and Technology Database
MLP	Multilayer Perceptron
MACC	multiply-Accumulate
Vitis	Vits HLS Software

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Artificial neural networks (ANN) have been proved useful for many everyday applications. Image recognition (SIMONYAN; ZISSERMAN, 2015), natural language processing (GREFENSTETTE et al., 2014), time series forecasting (TSANTEKIDIS et al., 2017), and computer games (MADDISON et al., 2015) are some of the many applications that take advantage of ANN capabilities. However, ANN inference may be extremely resources-hungry. For example, the VGG-19 is a state-of-the-art convolutional neural network (CNN) used for classifying images (SIMONYAN; ZISSERMAN, 2015). It has over 100M parameters and performs over 40M multiply-accumulate operations (MACC) to classify one image. As ANN grew in complexity, it became hard for CPU systems to meet the high performance and low power requirements in mobile applications (GUO et al., 2019).

Field-Programmable Gate Array (FPGA) and other platforms are being used for accelerating ANN inference. For FPGAs, the main challenges to be surpassed are the FPGA's size and lack of raw performance and fast access memory. A state-of-the-art ANN model may not fit in the limited size of an FPGA. The performance of the FPGA may be insufficient for executing the MAC operations for real-time applications. The parameters may not fit in the on-chip memory, meaning more accesses to the off-chip memory, bottlenecking the performance. This scenario only adds difficulty to the already difficult task of implementing ANN at the Register-Transfer Level (RTL). Therefore, it is necessary to explore solutions that facilitate the implementation of ANN on FPGA.

ANN algorithms are error-resilient, meaning that internal erroneous computations have a negligible effect on the application's result quality (WANG et al., 2019). Therefore, approximate computing emerges as a paradigm that can solve the challenges of deploying ANN on FPGAs. The use of approximate MAC units and data types may reduce the size, complexity, data storage requirement, and power consumption of the ANN, with a small tradeoff in accuracy. Therefore, approximate computing may allow the utilization of FPGAs as a platform for ANN inference.

High-Level Synthesis (HLS) is a process that facilitates the development of register transfer level (RTL) designs by automatically converting a C algorithm into an RTL architecture described in a Hardware Description Language (HDL). It allows design exploration based on different architectural parallelism. By using an HLS tool, it is possible to transform an ANN described in a programming language into an RTL model that can be synthesized into FPGA or ASIC. The HLS allows for fast analysis of the impact of modifications on the algorithm and on the HLS configuration in the area and performance of the generated hardware.

In this undergraduate thesis, we will study different architecture implementations of a case-study ANN trained to classify handwritten letters into two groups, ’X’ and ’O’. We will analyze the impact on the area and performance of different HLS directives. In addition, we will explore approximate computing applied to the ANN algorithm by exploring the differences between 32-bits floating-point and arbitrary precision fixed-point data representation.

The rest of this work is organized as follows: Chapter 2 presents the required background to comprehend our work. Chapter 3 briefly reviews existing work in the literature for approximate computing techniques for hardware-implemented ANN. Chapter 4 presents the CNN used, the methodology for approximating the CNN, the tool used to generate the RTL models, the methodology for the RTL models generation, and the methodology to evaluate the results. Chapter 5 presents and discusses the results achieved, and chapter 6 presents the conclusion and perspectives for future research created by this study.

2 BACKGROUND

This chapter presents an overview of the concepts and technologies that were studied and used on the development of this work.

2.1 Artificial Neural Network

The first artificial neural network models date back to the forties. Warren Sturgis was the first to introduce the concept of neurons in computer science. Warren formalized the concept of “formal neurons”. It was an abstract element that would produce a single output as a function of its inputs. They attempted to demonstrate that a finite network of formal neurons could be Turing-complete.

Since then, various neuron models and neuron networks were proposed. In this work, we are interested in two classes of neural networks: The Multilayer Perceptron and the Convolutional Neural Network.

2.1.1 Multilayer Perceptron

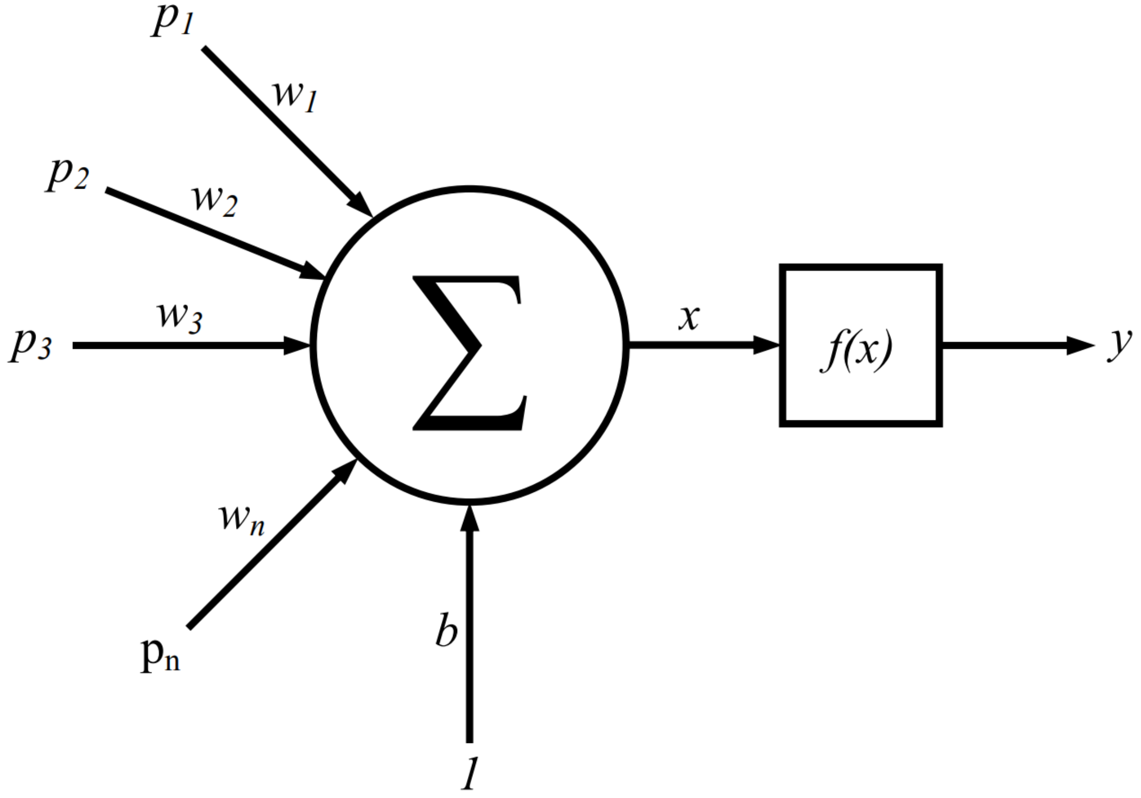
Multilayer perceptron (MLP) is a class of artificial neural network (ANN). It is a machine learning method inspired by concepts of the human brain. The MLP is composed by nodes that are interconnected, analogous to the interconnection between neurons in the human brain. MLPs are characterized in three main ways: Node character, network topology, and learning rules (ZOU; HAN; SO, 2009).

Node character determines how data is computed by the node. It includes the connections of the node (inputs and outputs), the weight associated with each input of the node, and its activation function (ZOU; HAN; SO, 2009). Figure 2.1 shows the graphical representation of a basic single node model. P_i represents the input data, W_i represents the weight associated with the input data, x_i represents the sum of each input multiplied by its associated weight, y_i is the output of the activation function, and b is the bias.

The node character is also usually represented by equation 2.1, following the same nomenclature than figure 2.1.

$$y = f(\sum_{i=0}^{n} w_i p_i - b) \tag{2.1}$$

Figure 2.1: Visual representation of the structure of a artificial neural network node.



Source: (MUTHURAMALINGAM; HIMAVATHI; SRINIVASAN, 2007)

Generally, Linear (2.2), Log-sigmoid (2.3) and Tan-sigmoid (2.4) are used as activation functions in state-of-the-art ANNs (MUTHURAMALINGAM; HIMAVATHI; SRINIVASAN, 2007).

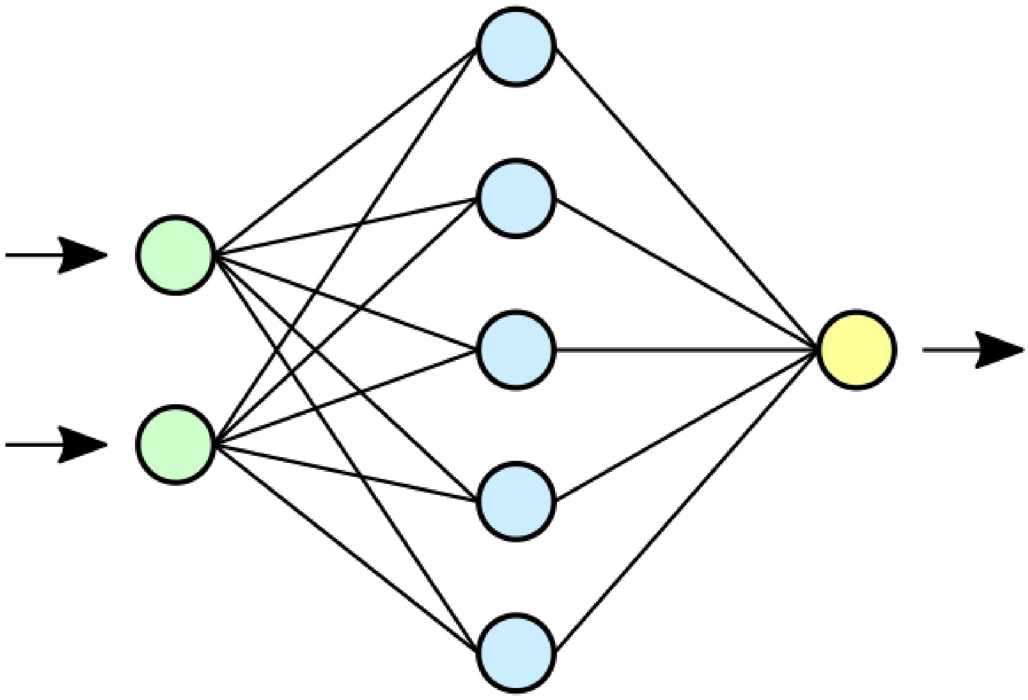
$$f(x) = x \tag{2.2}$$

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.3}$$

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.4}$$

The general MLP architecture consists of multiple arrays of nodes, called layers. Layers are classified as input layers, output layers, and hidden layers. The MLP topology is defined by the number of hidden layers, the number of nodes at each layer, and the connections between each layer (ZOU; HAN; SO, 2009). Figure 2.2 shows a diagram of a simple MLP with two nodes at the input layer, one hidden layer with five nodes, and one node at the output layer.

Figure 2.2: Visual representation of a Multilayer Perceptron.



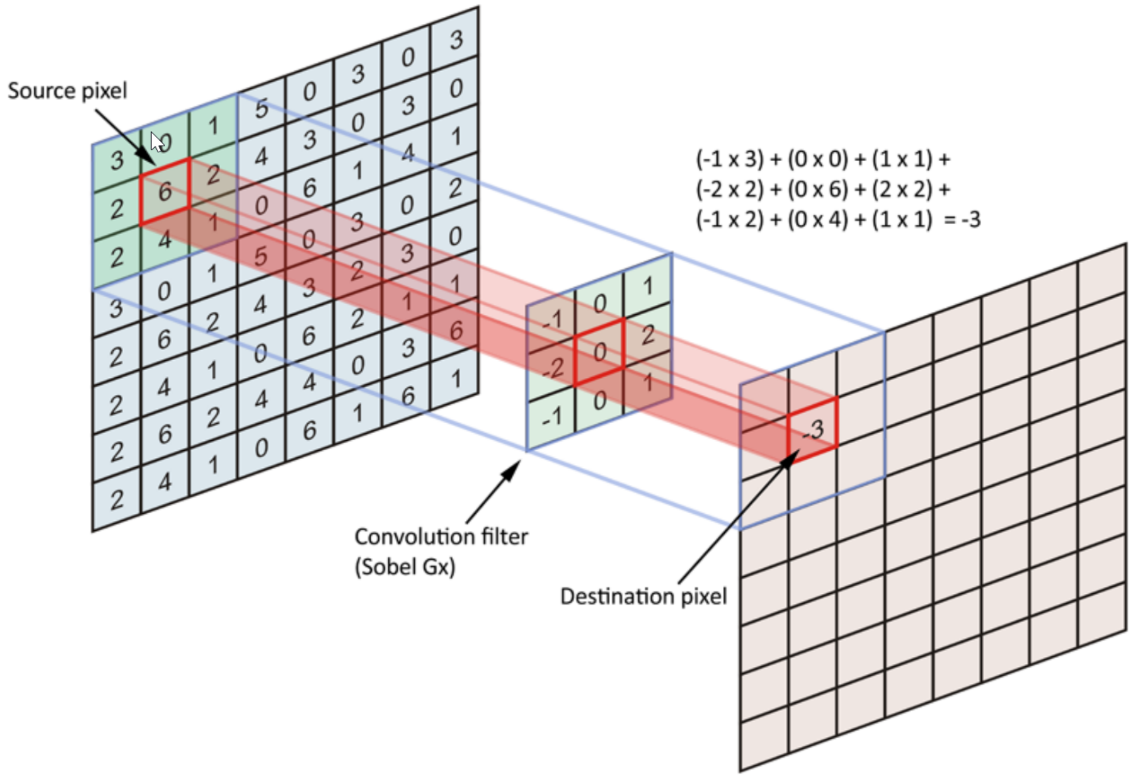
Source: (RéSEAU. . . , 2021)

In the learning – or training – process, the ANN weights are adjusted to accomplish the given task. Learning is divided into two categories: Supervised and unsupervised learning. In supervised learning, the ANN is given a dataset with inputs and the desired output for each input. The weighs are then adjusted to minimize the error between the network output and the desired output provided by the dataset (ZOU; HAN; SO, 2009). Different from the supervised learning method, in unsupervised learning the dataset does not include a desired output for the inputs. It is up to unsupervised learning algorithms to found patterns in the training dataset (HINTON; SEJNOWSKI, 1999).

2.1.2 Convolutional Neural Network

Convolutional neural network (CNN) is a class of artificial ANN commonly applied as a tool for computer vision. The CNN differs from the others ANNs by having features extraction layers before the MLP layers. These layers are formed by convolutional layers and pooling layers. The convolutional layer applies a filter, composed of an array of learnable weights, over the input data. The filter array is positioned over the input data array and the overlapping elements are multiplied and accumulated to compute an element of the result array. Then, the filter slide to the next position and the operation is performed again to obtain a new element of the result array. The convolution ends when all elements of the input data are reached by the filter. Figure 2.3 illustrates one step of the convolution operation for an 8x8 input data and a 3x3 filter.

Figure 2.3: The operations in a convolutional layer.

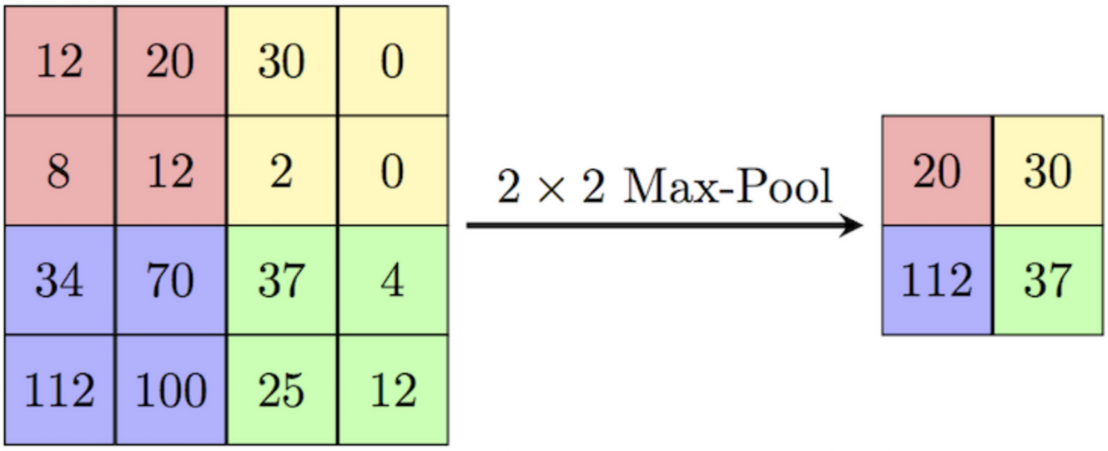


Source: (WHY..., 2021)

One convolutional layer may have multiple filters, each one configured to detect a different feature of the input data. If one convolutional layer applies 64 filters to the input data and the following layer applies 32 filters to its input data, the third layer would receive 64 * 32 = 2048 features. Each feature would be composed of a data array with the size related to the size of the input data and applied filters. This makes the number of stored parameters and computations performed by the CNN grow fast. The pooling layer is responsible for reducing the number of parameters and computations of the CNN.

The pooling layer combines a data cluster of its input array into a single element of its output array. The most common pooling operations are maximum and average pooling. The pooling operation is similar to the convolution operation, in the sense that sliding filters passes over the input data performing an operation. But instead of performing a multiply-accumulate operation, it can identify the average value of the elements overlapped by the filter, for example. Figure 2.4 exemplifies a 2x2 maximum pooling operation over a 4x4 input data array.

Figure 2.4: The operations in a pooling layer.



Source: (LíBANO, 2018)

After all the feature extraction performed by the convolution and pooling layers, the CNN may resolve into the MLP topology, working as described in section 2.1.1.

2.2 Approximate Computing

Approximate computing is defined as a paradigm that aims at the tradeoff between the quality and cost of a computing operation. Recently, computing systems have become increasingly embedded and mobile. At the same time, demanding computational tasks became popular. Applications that require an enormous amount of time and power to execute, as media processing and image recognition, are run by devices that rely on limited power availability. At the same time, many of these applications are error resilient. Error resilience is when the acceptable result of an application is within a margin of tolerance. The wideness of the margin can vary depending on the application. It allows for optimizations on algorithms and hardware while maintaining the quality of the result within the margin of acceptance. This error resiliency is due to several factors: A golden result does not exist or is difficult to define and obtain, the limited perceptual capability of humans, the users are willing to accept less-than-perfect results (VENKATESAN et al., 2011). In these cases, it is possible to make approximations in the operations performed, reducing the cost involved, while keeping the quality of the result within acceptable limits. The figure 2.5 exemplifies how an image may have its quality degraded at different levels while keeping its meaning.

The use of approximate computing can bring many benefits. It can reduce the execution time of heavy loads applications. It could allow heavy applications to reach real-time performance that otherwise would not be reachable. It can also improve the fault tolerance of a system. In this context, error is related with the occurrence of a fault, and not with the expected quality of the application output. Approximate computing can affect the fault tolerance of a system in two ways: First, it can intrinsically improve the reliability of the system. Secondly, it can reduce the cost of fault tolerance techniques (RODRIGUES; KASTENSMIDT; BOSIO, 2020).

Figure 2.5: Abstraction of the relation between cost and quality degradation of an image processing application.

Source: Alberto Bosio

Approximate computing can also reduce the costs of hardware accelerators. On hardware, it is implemented through the use of approximate hardware components. Approximate components do not exactly implement the intended mathematical or logical operation. However, it implements faster and more power-efficient mathematical or Boolean functions while keeping the quality of the erroneous result within an acceptable margin. This is defined as functional approximation (VASICEK; SEKANINA, 2015). Using this technique, (KULKARNI; GUPTA; ERCEGOVAC, 2011) were able to implement a two-bit multiplier using only 5 logic gates. It is a reduction of approximately 60% when comparing whit the conventional solution that requires 8 logic gates. At the same time, the output delay was reduced by 33%. The two-bit approximate multiplier was able to produce the correct result for 15 out of the 16 possible inputs. This multiplier has been used in larger approximate multipliers for image processing applications.

Another way to apply approximate computing on digital circuits is by performing over-scaling. In this case, circuits that are designed to work perfectly under normal conditions may have their power consumption reduced by voltage over-scaling. Meaning powering the circuit with lower supply voltage in which the circuit is known to occasionally produce erroneous outputs (VASICEK; SEKANINA, 2015).

2.3 FPGAs, GPUs, and CPUs

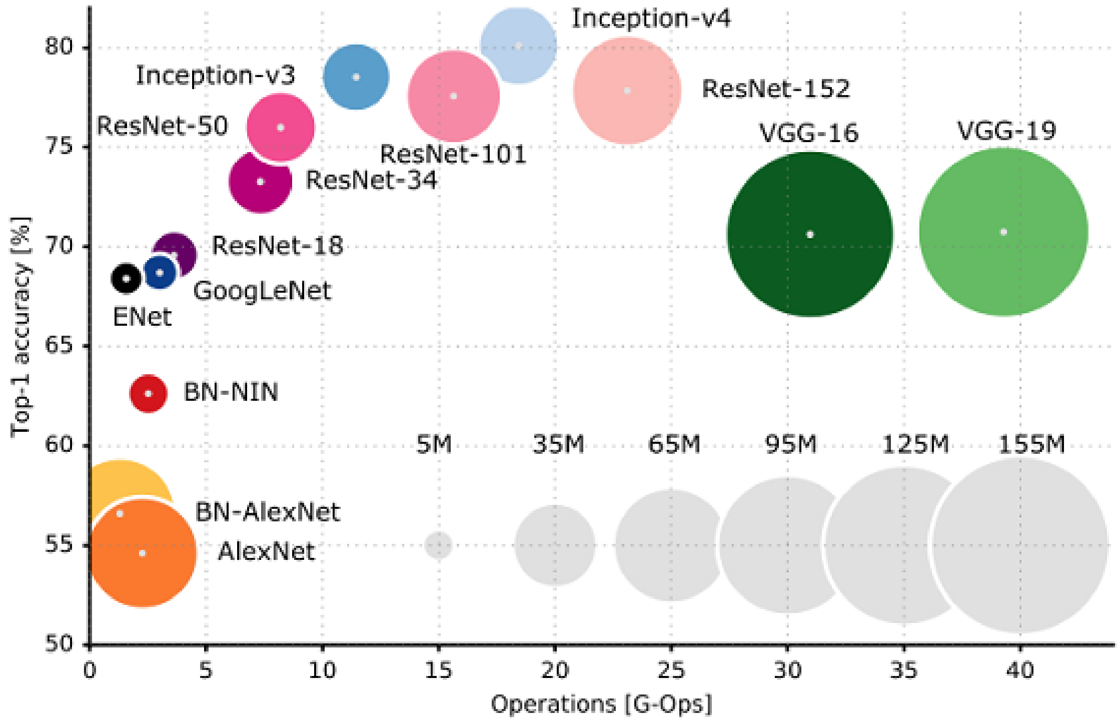
The number of operations and parameters involved in a state-of-the-art ANN execution has become extremely high. Figure 2.6 shows the number of operations related to the top 1% accuracy for some state-of-the-art ANNs. CPU platforms quickly became incapable of offering enough computation capacity for real-time ANN inference (GUO et al., 2019). GPUs became the most suitable platform for machine learning applications. Counting with many 32-bit floating-point multiplications units, GPUs provide superior performance than other platforms for machine learning workloads (NURVITADHI et al., 2017). However, recent research has shown that FPGAs have become a good candidate to achieve energy-efficient ANN inference (GUO et al., 2019).

Approximate computing emerged as a paradigm that can reduce the dependency on heavy floating-point computations. ANN algorithms are evolving to use compact data types (ZERVAKIS et al., 2021), which are difficult for GPUs to handle (NURVITADHI et al., 2017). At the same time, FPGAs allow the implementation of arbitrary logic using LUTs. FPGAs' flexibility allows the implementation of custom parallel designs that take maximum advantage of approximate data types. Although GPUs are still a good platform for ANN inference and CPU multicore implementations are possible. This work will focus on FPGAs as platforms for ANN accelerators.

2.4 Approximate Hardware

Multiply-accumulate (MACC) circuits are considered the fundamental building blocks of ANNs deployed in FPGA. It has been reported that MAC operations consume approximately 99% of the energy involved in Deep Neural Networks' computations (JAIN et al., 2018). Therefore, approximating the MAC unit is the most efficient way of approx-

Figure 2.6: Top-1 accuracy versus amount of operations required for a single execution of state-of-the-art ANNs .



Source: (CANZIANI; PASZKE; CULURCIELLO, 2017)

imating the ANN. Within the MAC unit, the multiplier is more complex, requires more resources than the adder. Hence, approximating the multiplier may be the more efficient way of increasing the overall efficiency of the system (ZERVAKIS et al., 2021).

2.4.1 Goals Of The Approximate Hardware

One of the key characteristics of an approximate system is its error-configurability. As stated in (VENKATARAMANI et al., 2015), different applications have different levels of error resilience and acceptable output quality degradation. Therefore, the importance of an approximate system having a certain level of error-configurability. Error-configurability can be applied in design-time or run-time.

In the design-time approximation, the approximation level is fixed before synthesis or fabrication. This allows the system to have a maximum level of area, latency, and energy optimization for a given accuracy constraint. However, accuracy requirements are not constant for keeping the same output quality. Therefore, systems using design-time approximation are designed under worst-case conditions, limiting the potential benefits of approximation (ZERVAKIS; AMROUCH; HENKEL, 2020).

Run-time approximation may be a better alternative for applications where output accuracy is highly dependent on the input. (TASOULAS et al., 2020) proposes a method to identify the accuracy requirement at run-time and thus decide the approximation level of the multiplier.

Another important characteristic for approximate designs is to have a low error bias. Approximate results can be greater (positive error) or smaller (negative error) than the golden value. To avoid error accumulation an approximate system should produce both positive and negative errors at, ideally, the same rate. In this manner, errors cancel each other instead of accumulating.

2.5 Data Quantization

Quantization allows the compression of neural network models into approximate models by the implementation of low-precision operations. Usually, weighs and activations of ANNs are represented using floating-point data. However, recent works have demonstrated that representing data with less precision (fewer bits) reduces bandwidth and storage requirements, and the hardware cost for each operation (GUO et al., 2019).

2.5.1 Fixed-Point Representation

In fixed-point quantization weights and activations are mapped to their nearest fixed-point value. This sacrifices the wider range of representable value provided by the floating-point representation. However, gains are noticeable in many other metrics. The range of the weights and activations greatly differs across layers. Therefore, it is important to assign carefully the integer and fractional bits length (QIU et al., 2016). (MA et al., 2017) proposed a 16 bits fixed-point FPGA implementation of the VGG CNN. Their implementation improved performance (GOPS and latency) by a factor of 3.2, comparing with other state-of-the-art implementations on FPGA, while keeping accuracy loss negligible.

2.5.2 Logarithmic Quantisation

Logarithmic multipliers were first introduced in 1962 by (MITCHELL, 1962). This approach takes advantage of the binary logarithms (log) properties to approximate data and simplifying the multiply operation. Usually, multiplications were performed using a series of shifts and additions. However, logarithms have been used to simplify this process, as the logarithm function reduces multiplications and divisions into additions and subtractions.

An important part of this technique is getting a good approximation of the log function curve. Having an exact table of logarithms would be impractical. On the other hand, the calculation of logarithms is also very power and time-consuming. Therefore, many ways of approximating the log function curve were proposed. (MITCHELL, 1962) proposed a simple technique where the log of a binary number is found by shifting the number and observing the position of the most significant 'one'. (ANSARI; COCKBURN; HAN, 2019) improved this method by creating an algorithm that rounds the number to its nearest power of two. With this method, they were able to reach a 6x smaller average error than the Mitchell method.

2.5.3 Binarisation

Binary neural networks (BNNs) are a class of neural networks quantized into just two values, represented by one bit. Typically, these values are -1 and 1. (COURBARIAUX et al., 2016) demonstrated that binarizing both weights and activations can reduce memory size and access, and simplify the MAC operation.

BNNs allow the replacement of the resources-hungry MAC operation by simpler operations. The multiply operation is replaced by an XNOR operation. It requires smaller, faster, and less power-consuming circuitry than the fixed or floating-point multiplication. Furthermore, the accumulation operations are replaced by a population count operation (WANG et al., 2019).

2.6 High-level synthesis

High-level synthesis (HLS) is a process that transforms an abstract behavioral specification of a design into an RTL structure that reproduces the behavior of the design (MCFARLAND; PARKER; CAMPOSANO, 1990). he cost (time and effort) of developing complex designs using hardware description languages (HDL) can increase rapidly. That is the case with systems like NNs and CNNs. HLS allows raising the level of abstraction during the development of a specification to facilitate the deployment of such systems in hardware. Traditional HLS tools usually support programming languages like C, C++, SystemC, and MATLAB. The code is transformed into an RTL design described in an HDL through the HLS process. Traditionally, HLS tools generate outputs in VHDL and Verilog. The architecture generated by the tool is algorithmically optimized for the execution of the specification, following user-specified time, power, and resource constraints (GAJSKI; RAMACHANDRAN, 1994).

During the HLS process the HLS tool performs the following tasks (COUSSY et al., 2009):

- *1.* compiles the specification
- *2.* allocates hardware resources from the RTL components library
- *3.* schedules the operations to clock cycles
- *4.* binds the operations to functional units
- *5.* binds variables to storage elements
- *6.* binds transfers to buses, and
- *7.* generates the RTL architecture

HLS can be used in both FPGA and application-specific integrated circuit (ASIC) development. The generated RTL model may vary depending on the target device and its constraints. Furthermore, it is important to notice that it is possible that the approximation mechanism used in this work is more efficient on one of the platforms. The results obtained using one of the platforms should not be extrapolated to the other.

3 RELATED WORK

In this chapter, we briefly review some works that applied the approximate computing paradigm to the implementation of ANNs in hardware platforms. Four of the seven related works presented are very similar to the work carried out on this undergraduate thesis. To facilitate the comparison between our work and the state-of-the-art, table 3.1 presents a summary of works that implemented approximate ANN on FPGA.

Table 3.1: Summary of most similar related works.

Reference	**ANN**	**FPGA**	**Technique**	**Activements**
(ZHOU; JIANG, 2015)	AlexNet	Virtex7	11-bits fixed-point architecture	16x faster than double precision CPU implementation with zero accuracy loss.
(LO; LAU; SHAM, 2018)	LeNet-5	Zynq UltraScale+	4-bits fixed-point architecture with 8-bit addition	Throughput of 5783 images/s at 50MHz with only 0.57% accuracy loss compared with floating-point implementation.
(CHO; KIM, 2020)	LeNet-5	Zynq UltraScale+	20-bit fixed-point architecture	90% latency reduction with similar area and only 0.01% accuracy loss compared with floating-point implementation.
(FENG et al., 2016)	LeNet-5	Zynq-7000	24-bit fixed-point architecture and approximate TanH function	Higher frequency, overall area reduction, and 93% energy saving with zero accuracy loss compared with floating-point implementation.

Source: The authors

3.1 An FPGA-based Accelerator Implementation for Deep Convolutional Neural Networks

In (ZHOU; JIANG, 2015) the authors used HLS to implement an approximate FPGA accelerator for the AlexNet neural network. The accelerator was implemented on a Virtex7 FPGA. The CNN was trained for digit recognition using the MNIST dataset. In this work, the CNN was approximated using 11-bits fixed-point arithmetic. The accelerator was compared to the original CPU implementation. The CNN was run through Matlab software using 64-bit floating-point data representation.

Table 3.2: Resource Utiization In Vivado Synthesis Report.

Resource	DSP	BRAM	Memory LUT	LUT	FF	IO
Used	83	0	5196	80175	46149	329
Available	2800	2060	130800	303600	607200	700
Utilization (%)	2.96	0	3.97	26.41	7.6	47

Source: (ZHOU; JIANG, 2015)

The accelerator developed using fixed-point occupied less than half the area of the FPGA and achieved a clock frequency of 150MHz. Table 3.2 shows the FPGA resource allocation obtained. The accuracy of the approximate accelerator was identical to that of the original CNN, both with a 96.8% accuracy rate. However, the approximate accelerator achieved 16 times better performance than the CPU implementation.

3.2 Fixed-Point Implementation of Convolutional Neural Networks for Image Classification

In (LO; LAU; SHAM, 2018) the authors implemented a CNN accelerator for handwritten digits recognition on a Zynq UltraScale+ FPGA using HLS. The CNN was developed in C++ using 4-bit fixed-point arithmetic with 8-bit additions.

The authors tested different levels of approximation. As Table 3.3 shows, the 4-bit fixed-point version showed similar accuracy as random guessing one of the ten possible digits. However, as stated by the authors, such poor accuracy was due to truncation followed by low-bits addition. The problem was mitigated by keeping the level of truncation but increasing the size of the adders from 4 to 8 bits. The result is an approximate model with only 0.57% accuracy loss compared to the 32-bit float version.

Table 3.3: Accuracy of the fixed-point CNN accelerator using different bit widths.

Bit width	Accuracy
Float 32	99.55%
Fixed 16	99.55%
Fixed 8	99.27%
Fixed 4	10.38%
Fixed 4 (with 8-bit addition)	98.98%

SoSource: (LO; LAU; SHAM, 2018)

Table 3.4 shows the result of the FPGA implementation. The authors have managed to achieve a good level of parallelism without exceeding the FPGA's resource limit.

91.27% of the available DSP was allocated to perform multiply-accumulate operations. This allowed the accelerator to achieve a throughput of 5783 images/s while running at only 50MHz.

Table 3.4: FPGA Resource Utiization In Vivado Synthesis Report.

Resource	Utilization	Available	Utilization (%)
LUT	30984	274080	11.30
LUTRAM	760	144000	0.53
FF	63555	548160	11.59
BRAM	466	912	51.10
DSP	2300	2520	91.27

Source: (LO; LAU; SHAM, 2018)

3.3 Implementation of Data-optimized FPGA-based Accelerator for Convolutional Neural Network

In (CHO; KIM, 2020), the authors proposed an approximate version of the LeNet-5 CNN implemented in FPGA for handwritten digit recognition. The CNN was developed in C++ and the RTL model was generated using HLS. The design was implemented on a Zynq UltraScale+ FPGA. The approximate accelerator was compared with a 32-bit float implementation.

One constraint imposed by the authors for the development of the accelerator was to achieve a design with zero accuracy loss. For this, they decided to use a 20-bit fixed-point architecture. The results showed that with this architecture they achieve 98.63% accuracy, only 0.01% less than the 32-bit float version.

Table 3.5: FPGA resources consumption comparison between the 32-bits float design (Conventional design) and the 20-bits approximate design (Proposed work).

Resource	Conventional Design		Proposed Design	
	Used	Utilization	Used	Utilization
BRAM	163	8%	95	5%
DSP	79	3%	143	5%
FF	19195	3%	33585	6%
LUT	21260	7%	32589	11%

Source: (CHO; KIM, 2020)

Table 3.5 shows the FPGA resource consumption for the approximate accelerator (Proposed design) and the 32-bit float-based version (Conventional design). The proposed design showed a small increase in the consumption of DSP, FF, and LUT. BRAM was the only element where there was a reduction in the allocation. However, the proposed design presented a 90% reduction in latency, as shown in Table 3.6.

Table 3.6: Comparison of the estimated performance between the 32-bits float design (Conventional design) and the 20-bits approximate design (Proposed work).

Implementation	Clock Cycles	Latency (nS)
Conventional Design	3,895,572	32,800,716
Proposed Design	410,758	3,581,810

Source: (CHO; KIM, 2020)

3.4 Energy-Efficient and High-Throughput FPGA-based Accelerator for Convolutional Neural Networks

In (FENG et al., 2016), the authors proposed an energy-efficient and high-speed approximate FPGA-based accelerator for CNN. The RTL model was generated from the LeNet-5 C++ model using HLS and implemented on a Zynq-7000 FPGA. For comparison, they implemented the accelerator using floating-point and fixed-point processing. The LeNet-5 architecture uses the hyperbolic tangent (TanH) as the activation function. However, the authors verified that TanH computation was very expensive on FPGA. One of the contributions of this work was the approximation of TanH using Lambert's continued fraction. The authors were able to reduce the DSP consumption from 14 to 3 DSP blocks per TanH function.

Table 3.7: Comparison of performance and resource Costs between 32-bits floating-point and 24-bits fixed-point implementations.

Design	Freq	BRAM	DSP	FF	LUT
Floating-point	100MHz	83%	80%	25%	77%
Fixed-point	166MHz	66%	43%	26%	73%

Source: (FENG et al., 2016)

For the fixed-point version, the authors found that a 24-bits fixed-point architecture was enough to match the accuracy of the 32-bits floating point implementation. The approximate implementation improved performance and reduce BRAM, DSP, and LUT consumption, as shown in table 3.7. Authors also compared their fixed-point implementation with CPU and GPU implementations, table 3.8 shows that the approximate accelerator showed up to 93.7% energy saving while improving execution time and maintaining accuracy.

Table 3.8: Comparison of execution time, power consumption, and error rate between the proposed accelerators with GPU and CPU implementations.

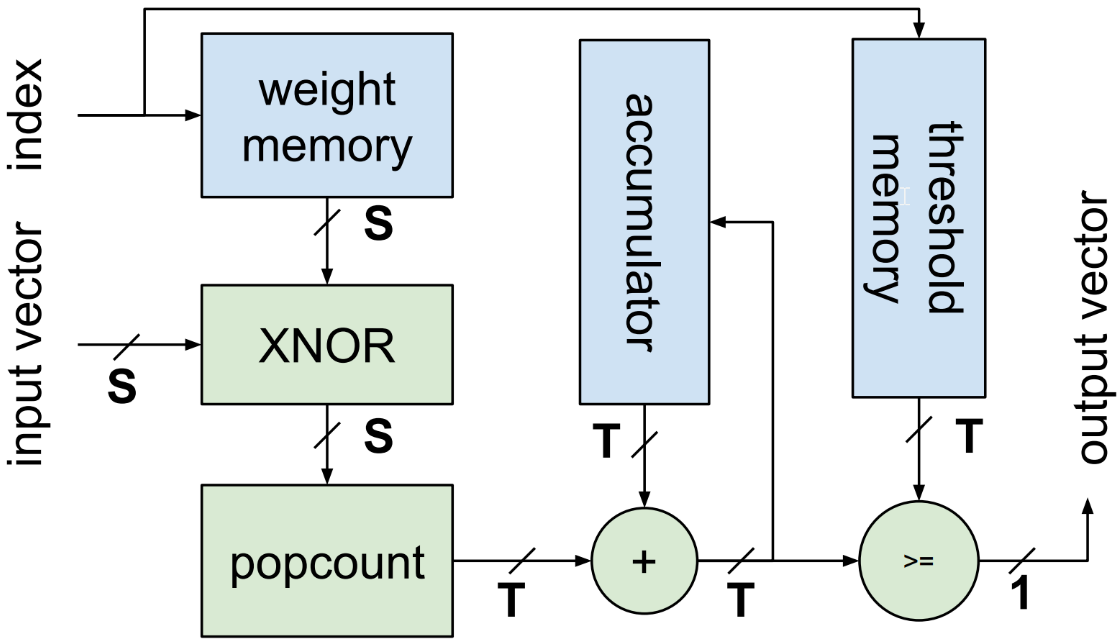
Device	Design or Software	Time	Power	Error Rate
Zynq-7000	Floating-point	71.66ms	2.47W	0.99%
ARM Cortex A9 666 MHz	Software	51.10ms	1W	0.99%
Intel Core i5 3.20Ghz	Tiny-CNN v0.1.0	2.06ms	86W	0.99%
AMD Radeon HD7450	DeepCL 8.3.1	0.714ms	18W	1.01%
NVidea GTX 840M	CUDA	0.646ms	33W	0.99%
Zynq-7000	Floating-point optimized	0.599ms	2.77W	0.99%
NVidea GTX 840M	Caffe re3	0.240ms	33W	1.09%
Zynq-7000	Fixed-point	0.151ms	3.32W	0.99%

Source: (FENG et al., 2016)

3.5 FINN

FINN is a framework for building binarized ANNs (CNNs and MLPs) accelerators on FPGA. FINN's strategy is to build custom architectures for each topology, instead of mapping operations to a fixed architecture. FINN also takes advantage of the reduced sizes of binarized parameters, that are kept stored in on-chip memory. This reduces off-chip memory accesses, minimizing latency. The Authors reported great performance on a ZC706 embedded FPGA, with implementations reaching up to 12.3 million classifications per second on the MNIST dataset with more than 95% accuracy and 0.31 uS of latency. (UMUROGLU et al., 2017).

Figure 3.1: The Matrix-Vector-Threshold Unit.



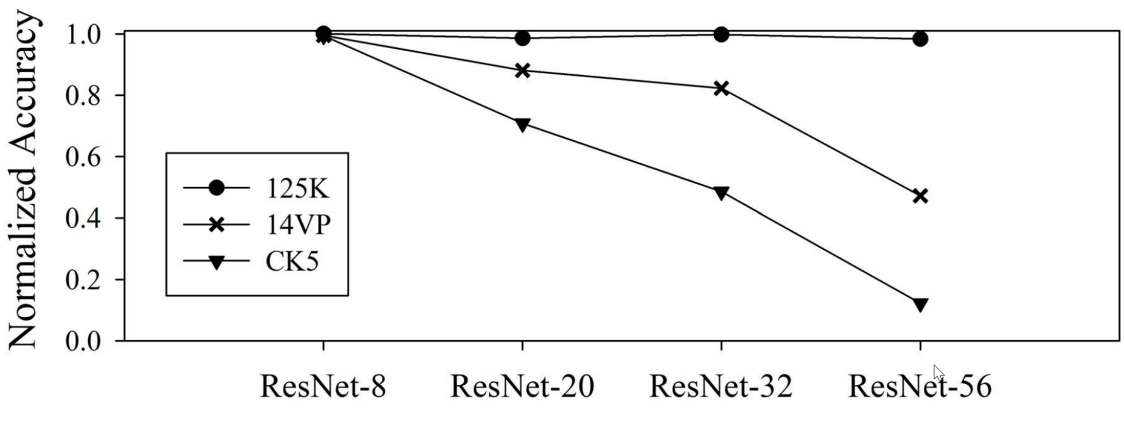
Source: (UMUROGLU et al., 2017)

FINN builds an architecture optimized for FPGAs using custom designs. The Matrix-Vector-Threshold Unit (MVTU) was introduced to form the core of the accelerator. It relies on the binary properties to implement the MAC operation through the XNOR and population count operations. The max-pooling operation is also optimized and implemented with the Boolean OR-operator (UMUROGLU et al., 2017). Figure 3.1 shows the diagram of an MVTU processing unity.

3.6 Weight-Oriented Approximation for Energy-Efficient Neural Network Inference Accelerators

In (TASOULAS et al., 2020), authors proposed a framework for mapping ANN weights to the accuracy levels of the approximate multipliers of the accelerator. This work was motivated by the realization that, if static approximate multipliers are used across the entire ANN, the overall accuracy drops when the ANN becomes deeper. Figure 3.2 shows the normalized accuracy drop for three different approximate multipliers when used in deeper CNNs.

Figure 3.2: Comparison of the accuracy of three approximate multipliers as the number of layers increases .



Source: (TASOULAS et al., 2020)(MRAZEK et al., 2017)

First, the authors developed a reconfigurable approximate multiplier optimized for the convolution operation. The multiplier is designed to perform in three modes: Exact operations mode or two different approximated operations modes. The authors also proposed a methodology for deciding which approximate mode would be used depending on the weight value for each layer (TASOULAS et al., 2020). This approach was able to save 17.8% of power consumption on average while keeping the accuracy loss at 0.5%.

3.7 High Speed, Approximate Arithmetic Based Convolutional Neural Network Accelerator

(ELBTITY et al., 2020) presents a CNN accelerator using the Dynamic Range Unbiased Multiplier (DRUM) (HASHEMI; BAHAR; REDA, 2015). DRUM is an approximated multiplier with a low error bias that saves up to 58% in power consumption. (ELBTITY et al., 2020) also uses an approximate adder to perform the accumulate operation. In this work, a CNN trained with the MNIST dataset was implemented using the accelerator. The approximate accelerator reduced the area of the CNN by 15% at the cost of approximately 1% of accuracy.

4 MATERIALS AND METHODS

In this work, our goal is to investigate how the use of approximate computing can reduce the cost of implementing convolutional neural networks (CNN) on FPGA. We will use as a case study a simple CNN trained to identify the letters 'X' and 'O'. This CNN will be called tic-tac-toe CNN. The development of the tic-tac-toe CNN is not part of this work. We will compare the implementation (logic synthesis, and place and route) of different approximate versions of the CNN with its original non-approximate version.

To achieve the goal of this work, the main tasks performed were:

- *1*. Creation of approximate versions of the tic-tac-toe CNN.
- *2*. High-level synthesis of the default CNN model.
- *3*. High-level synthesis of the approximate CNN models.

4.1 Dataset

The CNN used in this work aims to classify handwritten letters into two groups: 'X' and 'O'. Handwriting recognition is a well-known application for CNNs. Therefore, there are several databases of labeled images of human written letters and numbers. We choose to use the Modified National Institute of Standards and Technology database (MNIST). MNIST contains over 800,000 images with hand-checked classifications. However, our database is much smaller as we only have to account for the letters X and O. Figure 4.1 shows an X and an O from the MNIST database.

Because we are using a simple CNN model that only classifies images into two groups, a small dataset is enough to test our model. Furthermore, the training of the tic-tac-toe CNN was performed previously and is not part of this work. Therefore, we assembled a single testing dataset with 500 images. To form the dataset we needed to convert the 128x128 pixels images to 32x32 pixels, which is the size of the input matrix of our CNN model. The images on MNIST are in grayscale. Thus, each pixel of the image is represented by one element of the pixels matrix and each element can contain a value from 0 (black) to 255 (white). Finally, a .cpp files was created containing the 1024 pixels of each image.

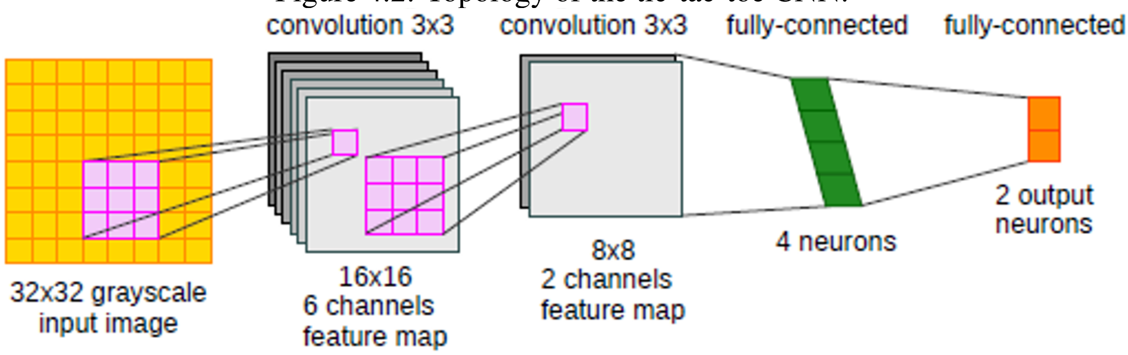
Figure 4.1: Example of the letters X and O extracted from the MNIST dataset.

Source: The authors

4.2 Tic-Tac-Toe CNN

In this work, we are interested in studying the possible benefits of approximate computing in FPGA-implemented neural networks. As a case study, we have chosen a simple CNN whose application is to identify the letters X and O for the tic-tac-toe game. The CNN receives as input 32x32 grayscale images and classifies them as containing one of the two letters. The CNN topology presented in figure 4.2 consists of two convolutional layers and two MLP layers, totaling four layers.

Figure 4.2: Topology of the tic-tac-toe CNN.



Source: Fabio Benevenuti

The first convolutional layer produces a 6-channels 16x16 feature map. Next, the second and last convolutional layer reduces it to a 2-channels 8x8 feature map. Both layers used kernels of size 3x3, a stride value of 2, and no padding. The convolutional layers are followed by a 4-neurons MLP layer. Lastly, we have a 2-neurons MLP output layer. The activation function used between layers is the ReLu function.

To analyze the effects of approximate computation on the FPGA-implemented tic-tac-toe CNN, we first need to define a default version. For this work, we decided on a 32-bits float version of the tic-tac-toe CNN. Meaning that every data, variable, or parameter in the CNN will be represented using the 32-bits float datatype. Using our test dataset described in section 4.1, the default version of the tic-tac-toe CNN has an accuracy of 98.2%.

The CNN used in this work is coded in C++. Each layer of the CNN is represented by an individual function. Thus, we can modify the data type of each function and its inputs individually. For example, listing 4.1 presents the function macc2_3x3. This function is part of the second convolutional layer. It performs the multiply-accumulate operation within the layer. It receives as inputs FEATURES and WEIGHTS. FEATURES and WEIGHTS are arrays of custom types fm_conv1_t and weight_conv2_t, respectively. The macc2_3x3 function itself is of custom type macc_conv2_t.

We can define fm_conv1_t, weight_conv2_t, and macc_conv2_t with data types less precise than the default 32-bits float. Therefore, creating a less precise, approximate version of macc2_3x3().

```
macc_conv2_t macc2_3x3(fm_conv1_t features[CONV2_WH_KERNEL*
    CONV2_WH_KERNEL], weight_conv2_t weights[CONV2_WH_KERNEL*
    CONV2_WH_KERNEL]) {
  macc_conv2_t accumulator = 0;
  int i;
  for (i = 0; i < CONV2_WH_KERNEL*CONV2_WH_KERNEL; i++) {
    #pragma HLS UNROLL

    accumulator += features[i] * weights[i];
  }
  return accumulator;
}
```

Listing 4.1 – C++ source code of the macc2_3x3 function showing the parametrization of its elements.

4.2.1 HLS Arbitrary Precision Types Library

As shown in section 4.2, the CNN model originally uses the 32-bits float representation to declare arrays, variables, and functions. In this work, we will approximate the tic-tac-toe CNN by changing the data type used to represent its elements. To accomplish this task, we utilize the HLS Arbitrary Precision Types library provided by Xilinx. The format for creating arbitrary fixed-point datatypes is ap_[u]fixed<W,I,Q,O,N> where:

- *W* defines the total size of the datatype in bits.
- *I* defines the size of the integer part of the fixed-point representation. It can also be interpreted as the number of places above the decimal point.
- *Q* defines the rounding behavior.
- *O* defines the overflow behavior.
- *N* defines the number of saturation bits in overflow wrap modes.

For this work, the following options for 'Q' and 'O' were used:

- *Q* = AP_RND: Round the value to the nearest representable value for the specific ap_[u]fixed type
- *O* = AP_SAT: Saturate the value to the maximum value in case of overflow or to the negative maximum value in case of negative overflow.

4.2.2 Aproximate Tic-Tac-Toe CNN

As introduced previously, we can create approximate versions of the tic-tac-toe CNN by modifying the data type used to represent the elements of the CNN. The technique used was Fixed-Point Data Quantization, whose benefits were discussed in section 2.5.1. At this stage of the project, we are interested in finding approximate versions of the CNN that achieve similar performance in terms of accuracy as the default CNN. To do this, we decided to approximate the CNN at four different points:

- CNN input.
- CNN weights and bias.
- CNN feature maps.
- MACC units output.

It is important to note that for this work, we decided that all layers would receive the same approximation level. For example, by changing the output size of the MACC units, we are affecting all MACC units of the CNN. It would be possible to configure each layer of the CNN with different approximation levels, but this would make the analysis

Table 4.1: The first round of experiments in the search for the best approximate versions. Input, weights/bias, MACC Units, and Feature Maps values are in bits.

Experiments	**Input**	**Weights/Bias**	**MACC Units**			**Feature Maps**			**Result (%)**
			total	**int**	**frac**	**total**	**int**	**frac**	
0	8	6	24	12	12	12	9	3	98.2
1		5	24	12	12	12	9	3	97.2
2		4	24	12	12	12	9	3	98
3		3	24	12	12	12	9	3	76
4		6	16	8	8	12	9	3	95.4
5		5	16	8	8	12	9	3	94.8
6		4	16	8	8	12	9	3	94.6
7		3	16	8	8	12	9	3	75.8
8		6	16	8	8	8	6	2	95.8
9		5	16	8	8	8	6	2	95.6
10		4	16	8	8	8	6	2	94
11		3	16	8	8	8	6	2	78.2
12		6	8	4	4	8	6	2	80.4
13		5	8	4	4	8	6	2	74.2
14		4	8	4	4	8	6	2	81
15		3	8	4	4	8	6	2	65.4

Source: the authors

too complex for the purposes of this work.

Due to the time required to perform each experiment, an exhaustive testing campaign to find the best configuration was not feasible. Therefore, we performed the first round of experiments with arbitrarily chosen settings. The idea was to check how these first versions would behave. Table 4.1 shows the settings and results for this first experiment.

We decided to fix the pixel size of the CNN input matrix to 8 bits. In a grayscale image, the pixel value ranges from 0 to 255. Therefore 8 bits is enough to represent the pixels without loss of quality. The CNN weights and bias ranged from 6 to 3 bits with all bits dedicated to representing the fractional part of the values. The MACC units are implemented using 24, 16, or 8 bits. For this round of experiments, 50% of the bits are assigned to the integer part and 50% are assigned to the fractional part. Finally, the feature maps were configured with 12 or 8 bits. Since the feature maps are, essentially, registers storing the outputs from the MACC units, we opted for splitting 75% of the bits for the integer part and 25% for the fractional part. This accommodates a wider range of values, rather than having a higher resolution in the fractional part.

The first information we can extract from this result is that there is a significant loss of precision when using weights and bias with 3 bits compared to 4 bits. You can

Table 4.2: The second and final round of experiments in the search for the best approximate versions. Input, weights/bias, MACC Units, and Feature Maps values are in bits.

Experiments	Input	weights/bias	MACC Units			Feature Maps			Result (%)
			total	int	frac	total	int	frac	
0	8	6	24	9	15	12	9	3	98.2
1		5	24	9	15	12	9	3	97.2
2		4	24	9	15	12	9	3	98
3		3	24	9	15	12	9	3	76
4		6	16	9	7	12	9	3	98.2
5		5	16	9	7	12	9	3	97.2
6		4	16	9	7	12	9	3	98
7		3	16	9	7	12	9	3	76
8		6	16	9	7	8	6	2	97
9		5	16	9	7	8	6	2	97.4
10		4	16	9	7	8	6	2	95
11		3	16	9	7	8	6	2	78.8
12		6	12	9	3	8	6	2	96.2
13		5	12	9	3	8	6	2	97.2
14		4	12	9	3	8	6	2	94.8
15		3	12	9	3	8	6	2	78.4

Source: the authors

notice this behavior by comparing tests 2 and 3, 10 and 11, and 14 and 15. Another behavior that we can notice when comparing tests 4 and 8 is that there is no significant loss in accuracy when we decrease the size of the feature map. There was even an increase in accuracy. It is important to note that this increase is small and certainly within a margin of error. Comparing tests 0 and 4, we can notice a loss of accuracy when decreasing the size of the MACC unit. However, investigating this particular case, we found that this loss of accuracy was due to using only 8 bits for the integer part.

Then, we repeated experiment 4 but changed the number of bits dedicated to the integer part. In this new test, we increased the bits of the integer part from 8 to 9 and decreased the bits of the fractional part from 8 to 7. This new experiment indicated that for this trained CNN, 9 bits is the minimum size possible for the integer part in the MACC units. With this new constraint, we performed a second round of experiments. Table 4.2 shows the new results obtained. In this round, we followed the same logic as in the previous round for the size of the input data, weights and bias, and feature map. However, we fixed the integer part of the MACC units to 9 bits.

With these results, we decided to use the configurations from experiments 6 and 13 in the continuation of this work. For convenience, the configuration of experiments 6 and 13 will be referred to as Approx_0 and Approx_1, respectively. Table 4.3 presents the two versions of the tic-tac-toe CNN that we aim to use in the following steps of this

Table 4.3: Description of the two approximate configurations of the tic-tac-toe CNN that will be used in the continuation of this work.

Experiments	Input	weights/bias	MACC Units			Feature Maps			Result (%)
			total	int	frac	total	int	frac	
Approx_0	8	4	16	9	7	12	9	3	98
Approx_1	8	5	12	9	3	8	6	2	97.2

Source: The authors

work.

The two versions chosen have a low overhead in accuracy. Approx_0 showed a loss of only 0.2% while Approx_1 showed a loss of 1.2%. Approx_0 was chosen because it presents a large reduction in the width of the CNN elements while maintaining virtually the same accuracy. The second version, on the other hand, was chosen to represent a different tradeoff between approximation level and accuracy. The expectation is that Approx_1 will generate FPGA implementations that consume fewer FPGA resources than Approx_0.

4.3 High-Level Synthesis Tool - Vitis HLS

In this work, we use the Vits HLS software (Vitis) to perform High-Level Synthesis of models implemented in C++. Vitis replaced Vivado HLS in the Vivado Design Suite provided by Xilinx. The automated HLS process should create an optimized hardware to realize the target application. However, Vits gives the user the possibility to influence the HLS process through the use of pragmas or directives. In this work we use two pragmas: HLS unroll and HLS array_reshape.

The unroll pragma transforms loops by creating multiples copies of the loop body in the RTL design. This allows different loop iterations to happen in parallel. It is possible to unroll a loop partially or fully. In this work we only use the fully loop unroll. Listing 4.2 shows how the HLS unroll pragma can be applied to the code to fully unroll a loop. In this example, Vits must create N adders in parallel to perform this operation.

```
loop_1: for(int i = 0; i < N; i++) {
  #pragma HLS unroll
  a[i] = b[i] + c[i];
}
```

Listing 4.2 – Using the HLS unroll pragma to completely unroll the loop_1 loop. Source: Xilinx

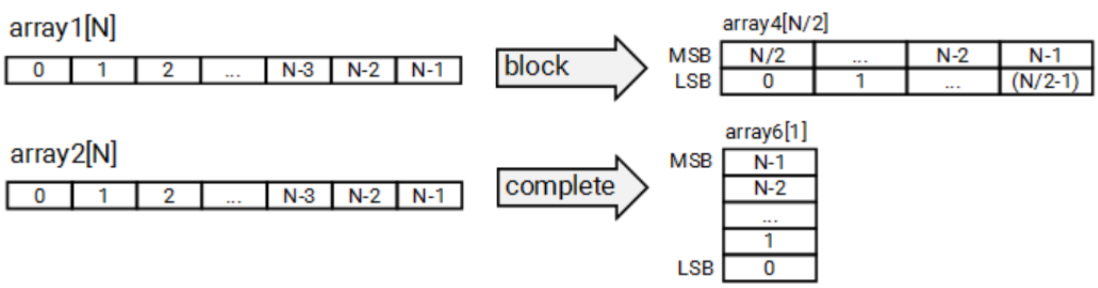
The HLS array_reshape pragma transforms an array into a new array with fewer elements but with greater bit-width. This allows parallel access to the data of the array, allowing Vitis to find new ways to parallelize a task. In this work, we use two of the three versions of array_reshape supported by Vitis: block and complete array_reshape. Listing 4.3 exemplifies how we can use this pragma to modify arrays array1[N] and array2[N].

```
void foo (...) {
int array1[N];
int array2[N];
#pragma HLS ARRAY_RESHAPE variable=array1 block factor=2 dim=1
#pragma HLS ARRAY_RESHAPE variable=array2 complete dim=1
...
}
```

Listing 4.3 – Using the HLS array_reshape pragma to reshape the arrays array1 and array2. Source: Xilinx

To perform the reshape, it is necessary to specify the variable to be affected, the type of reshape (block or complete), the reshape factor, and the dimension of the array that will be affected by the pragma. The reshape factor defines how many times the size of the original array will be divided. In other words, it defines how many times the throughput of the register bank will increase. In the complete array reshape, the factor is always N. That is, all array elements become accessible in a single read from the register bank. Figure 4.3 illustrates the transformation applied to the arrays array1[N] and array2[N].

Figure 4.3: Visual representation of the complete and block array_reshape pragma applied to the array1 and array2 arrays respectively.



Source: Xillinx

4.4 Evaluation

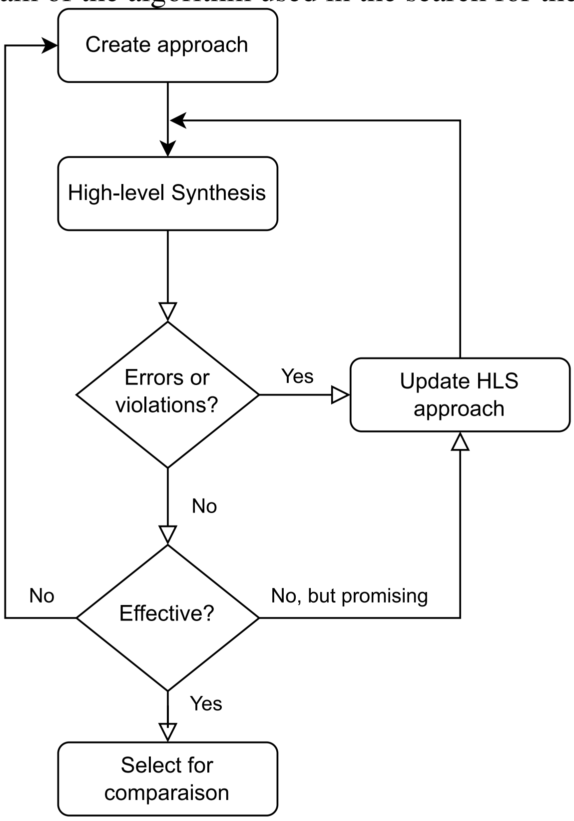
In the following sections, we will be comparing the FPGA implementations of different versions of the CNN presented in section 4.2. Therefore, it is necessary to define what metrics will be used to evaluate the effects of approximation on the synthesized circuits.

- *Latency (in clock cycles)*: The number of clock cycles (c.c.) required to run the application. As discussed in section 2.2, one of the possible benefits of approximate computing is the simplification of the mathematical operations involved in the CNN inference process. The performance gain obtained by this simplification can be noticed in the decrease of the clock period or the decrease of the c.c latency. The latter is more pertinent to this work since the HLS tool will always try to respect the target period provided in the synthesis settings. Resulting in a similar final clock frequency for all designs.
- *FPGA resources consumption*: FPGAs have a limited amount of resources for logic circuit implementation. A large design may not fit on the FPGA available for a given project or even not fit on any FPGA available in the market. One of the possible benefits of approximate computing is the reduction of FPGA resource consumption through the simplification of logic expressions, mathematical operations, and reduced data storage. In this work, we will consider as "FPGA resources" the following elements:
  - Block RAM (BRAM)
  - Digital Signal Processing Units (DSP)
  - Flip-Flop (FF)
  - Look-up-Table (LUT)
- *Energy consumption*: As discussed in chapter 1, the inference process of a CNN may require millions of mathematical operations. The energy consumption of the required hardware can be prohibitive for many applications. Approximate hardware may be smaller and require less time to perform a given task, possibly reducing energy consumption compared to non-approximate versions.

4.5 Methodology

In this work, we are interested in using Vitis and the loop unroll and array reshaping pragmas to produce FPGA implementations of the tic-tac-toe CNN. Since we can apply the pragmas to different regions of the CNN and with different intensities (factor configuration), there are many possible approaches we can use. This flexibility creates a situation where testing all possibilities is virtually impossible. The process of high-level synthesis requires high computational power. The Vitis HLS takes from 40 to 120 minutes to execute on the system we used for this task. One solution to this problem would be to carefully choose the approaches, selecting only the best ones for synthesis. However, this task is difficult since the Vitis optimization algorithm is too complex. This makes Viti's behavior difficult to predict.

Figure 4.4: Diagram of the algorithm used in the search for the HLS approaches.



Source: The authors

In this context, we decided to follow the algorithm illustrated in Figure 4.4. First, we arbitrarily create an approach for the use of the pragmas. After running the high-level synthesis we review the reports of errors and violations from the tool to identify if there are any problems with the approach. In the case of an error or violation, we modify the approach to address the reported issue. Usually, problems encountered in this step were related to memory violation due to the lack of loop unrolling in some convolution-related loops. Next, we evaluate the synthesis results (FPGA resources consumption and latency) to assess the approach's effectiveness. Depending on the result we would either select the design for future comparison, modify the configuration used and re-launch synthesis, or discard the configuration used and create a new approach.

5 RESULTS AND COMPARISONS

This chapter will present the comparison between the designs produced with Vitis HLS. First, we will describe the development and present the 32-bits float versions of the CNN tic-tac-toe that will serve as a baseline for the comparison. Next, we will present the approximate designs generated from the approximate CNN tic-tac-toe models. The approximate designs will be produced using the same or similar HLS approach as the non-approximate versions. The goal is to isolate as many variables as possible, allowing the analysis of the difference between using the approximate and non-approximate models during high-level synthesis and implementation.

First, we will present the three HLS approaches used as a baseline in the approximate versions evaluations. We defined three approaches that produce three different levels of parallelism in the RTL model: Multicycle, Balanced, and Parallel. Multicycle aims to represent a use case where the main constraint is the consumption of FPGA resources. Balanced represents a use case where more FPGA resource consumption was allowed in exchange for improved latency. Lastly, the Parallel approach aims to have the best latency possible with no restrictions on FPGA resource consumption. The results are presented in tables, where each column presents an HLS approach and its results. The first part of the table presents the target period and the pragma configuration used. In this work, we use the HLS UNROLL and HLS ARRAY_RESHAPE pragmas. As explained in section 4.3, HLS UNROLL is used to parallelize the interactions of loops and HLS ARRAY_RESHAPE is used to parallelize register banks. The reshape pragma is used on the input bus of the model or the internal registers. The second part of the table presents the reports provided by Vitis during the HLS or implementation.

5.1 32-bits Float Models Designs

In section 4.2, we have defined which CNN models would be implemented, including two approximate models and one 32-bits baseline model. In this section, we are interested in finding good implementations for the 32-bit float model. These implementations will serve as a baseline for when we will be implementing the approximate versions of the CNN. The definition of "good implementation" may vary depending on the application. Therefore, we want to find good implementations with different tradeoffs between performance (execution time) and FPGA resource consumption. The methodology applied in this task was described on section 4.5.

We decided to start by aiming for a multicycle version with low FPGA resource consumption. Next, we defined a more balanced version allowing more FPGA resource consumption but improving the execution time. Lastly, we produced a fully parallelized implementation, where the goal is to achieve the shortest execution time possible.

As presented in section 4.3, we only use the unroll and array_reshape pragmas. Both pragmas aim to allow the tool to increase the parallelism of the generated RTL model. Therefore, we started by performing a synthesis without using any of the pragmas. This approach did not work as Vitis returned several memory dependency violations. After investigation, we found that the violations were happening because of the lack of loop unrolling on multiple For loops. When the HLS Unroll pragma is not used, Vitis tries to generate a pipeline architecture to execute the loop. However, it seems that this architecture could not handle the inherent memory dependency of the convolution operation and the Relu function.

Gradually the HLS unroll pragma was added in every loop of the code to try to resolve the violations. However, it turned out that the pragma had to be added in almost every loop related to convolutional layers. So, to avoid having to deal with the complexity of dealing with pipeline, we decided to use the unroll pragma in all loops for all versions.

Finally, we advance to the study of the influence of array_reshape on the generated RTL model. It is important to note that we have different areas where we can apply the array reshape pragma. We can apply the pragma to the internal register banks, but also to the CNN input bus. We decided that we would analyze these two possibilities separately.

Each layer of the CNN has its register bank. Since the tic-tac-toe CNN has four layers, we have four register banks where we can apply the pragma separately. However, the register banks of the MLP layers are much smaller than the register banks of the convolutional layers. The register banks of the convolutional layers 1 and 2 have 1536 and 128 elements respectively. In contrast, the register banks of MLP layers 1 and 2 have only 4 and 2 elements respectively. Therefore, we defined that in the MLP layers, we would always use the array_reshape type "complete". For the convolutional layers, we applied the array_reshape pragma using different factor values.

Table 5.1 displays the result of the first round of high-level synthesis performed. For this round, the only configuration that varies between each synthesis is the configuration of the array reshape on the internal registers. Each column of the table shows the set of configurations used in Vitis and the report about the generated RTL model.

Table 5.1: Results of the high-level synthesis using different approaches for the use of the array_reshape pragma on the internal registers. Percentage values are relative to the Artix-7 AC701 FPGA.(App is the abbreviation of "approach")

	App_1	**App_2**	**App_3**	**App_4**	**App_5**	**App_5**
Target Period (nS)	10					
Loop Unroll	YES					
Input Reshape (factor)	NO					
Registers Reshape (factor)	NO	conv1: 96 conv2: 8 FCs: complete	conv1: 192 conv2: 16 FCs: complete	conv1: 384 conv2: 32 FCs: complete	conv1: 768 conv2: 64 FCs: complete	complete
Estimated Period (nS)	9.017	8.833	9.017	8.772	8.772	8.772
Latency (c.c)	37,537	35,789	33,119	16,775	16,776	16,771
External Interface	1 BRAM addr: 10b data: 32b					
#BRAM	12	287	58	0	0	0
#DSP	42	68	240	2,448	2,448	2,448
#FF	102,112	143,077	134,299	51,548,144	97,532,568	10,467,028
#LUT	62,353	62,725	123,745	48,966,784	64,877,620	49,355,040
BRAM (%)	2%	39%	8%	0%	0%	0%
DSP (%)	6%	9%	32%	**331%**	**331%**	**331%**
FF (%)	38%	53%	50%	**19,149%**	**36,231%**	**3,888%**
LUT (%)	46%	47%	92%	**36,379%**	**48,200%**	**36,668%**

Source: The authors

As discussed in section 4.4 (evaluation) we are interested in evaluating the designs in three aspects: Latency, FPGA resources consumption, and energy consumption. Vitis does not estimate the energy consumption during the high-level synthesis. Therefore, this evaluation will be done later in this work.

App_1 is the only one where the array reshape was not applied, and this reflects in the latency of the design, the highest among all versions. However, this design consumed the least FPGA resources. App_2 and App_3 showed a modest gain in latency at the cost of more FPGA resources compared with App_1.

App_4, App_5, and App_6 showed a significant decrease in latency. However, this decrease came with a notable increase in FPGA resource consumption. The three designs would not fit on the Artix-7 AC701 FPGA that we targeted during the high-level synthesis.

Table 5.2: Results of the high-level synthesis using different approaches for the use of the array_reshape pragma on the input of the design. Percentage values are relative to the Artix-7 AC701 FPGA.(App is the abbreviation of "approach")

	App_4e	**App_4d**	**App_4c**	**App_4b**	**App_4**
Target Period (nS)	10				
Loop Unroll	YES				
Registers Reshape (factor)	conv1: 384 conv2: 32 FCs: complete				
Input Reshape (factor)	complete	factor 768	factor 384	factor 8	NO
Estimated Period (nS)	7.52	8.772	8.772	8.772	8.772
Latency (c.c)	4,052	16,793	16,775	16,784	16,775
External Interface	1 bus 32,767b	2 BRAM addr: 4b data: 4,096b	2 BRAM addr: 4b data: 4,096b	1 BRAM addr: 07b data: 256b	1 BRAM addr: 10b data: 32b
#BRAM	0	0	0	0	0
#DSP	3,718	2,448	2,457	2,448	2,448
#FF	53,545,156	51,565,240	51,554,480	51,548,844	51,548,144
#LUT	83,668,976	48,981,820	48,982,884	48,970,400	48,966,784
BRAM (%)	0%	0%	0%	0%	0%
DSP (%)	**502%**	**331%**	**332%**	**331%**	**331%**
FF (%)	**19,890%**	**19,155%**	**19,151%**	**19,149%**	**19,149%**
LUT (%)	**62,161%**	**36,391%**	**36,391%**	**36,382%**	**36,379%**

Source: The authors

Next, we investigate the influence of applying the array reshape pragma to the interface (input) of the RTL model. The use of array_reshape to the interface increases the width of the external interface of the circuit. It enables the circuit to receive more data per clock cycle, potentially increasing the internal parallelism of the generated design. Table 5.2 presents this round of high-level synthesis. Here, the only configuration that changes is the interface array reshape. Despite our expectations, in most experiments, increasing the array reshape at the interface did not significantly decrease latency. Nor does it appear to have significantly altered the generated RTL model, since all but App_4e synthesis showed similar values of FPGA resource consumption. However, we can see that by using the complete array_reshape in the App_4e test, Vitis was able to parallelize the design enormously, decreasing the latency to just 4052 clock cycles. However, such parallelism came with a tremendous overhead in FPGA resources consumption.

Table 5.3: Description and result of the high-level synthesis for the three chosen approaches. Percentage values are relative to the Artix-7 AC701 FPGA.

	Multicycle	Balanced	Parallel
Model config	float 32 bits		
Model Accuracy	98.2%		
Target Period (nS)	10		
Loop Unroll	YES		
Input Reshape (factor)	NO	NO	complete
Registers Reshape (factor)	NO	conv1: 192 conv2: 16 FCs: complete	complete
Estimated Period (nS)	9.017	9.017	7.52
Latency (c.c)	37,537	33,119	4,048
External Interface	1 BRAM addr: 10b data: 32b	1 BRAM addr: 10b data: 32b	1 bus 32,767b
#BRAM	12	58	0
#DSP	42	240	3718
#FF	102,112	134,299	12,463,786
#LUT	62,353	123,745	84,057,304
BRAM (%)	2%	8%	0%
DSP (%)	6%	32%	**502%**
FF (%)	38%	50%	**4,630%**
LUT (%)	46%	92%	**62,450%**

Source: The authors

From the results obtained in the various high-level syntheses performed, we selected three approaches to serve as the baseline for the continuation of this work. Table 5.3 lists the four selected approaches. The names of each approach got relabeled for convenience.

The approach Multicycle was selected because it is the least resource-consuming version. Balanced represents a slightly different approach compared with Multicycle. It utilizes more FPGA resources, without extrapolating the limits of the FPGA, for a 12% better latency performance. Although the board resource consumption values are prohibitive, we decided to select the Parallel approach as well. It has a very high level of parallelism which is reflected in the latency result. It is important to note this design would not fit in even the largest FPGAs available on the market. However, we are interested in verifying if Vitis would produce better results using the same high parallelism approach when using the approximate versions of the CNN.

5.2 Approximate Models Designs

This section presents the approximate designs obtained using the methodology described in section 4.5 alongside the comparison with the baseline 32-bits float versions. When possible, we will use the implementation results (logic synthesis, and place and route) for the discussion. However, in cases where the designs are too large to fit on the FPGA used during the High-level synthesis (HLS), we will have to use the values estimated by Vitis post-HLS.

To have a well-defined point of reference, we defined that the comparison between the designs should be per approach. For example, table 5.3 presented three different approaches for the high-level synthesis. In Multicycle, there is no effort to parallelize the design. Therefore, this version must only be compared to other approximate versions that follow a similar approach.

5.2.1 Aproximate Tic-Tac-Toe CNN - Update

In section 4.2.2 we had presented two approximate versions of the CNN tic-tac-toe that would be used in the high-level synthesis step. However, the initial results of the synthesis led us to replace the approximate version labeled Approx_1 with a new version. This decision was made for two reasons. The first reason is that Approx_1 presented similar but always worse initial results than Approx_0. Because Approx_1 has smaller MACC units and Feature Maps than Approx_0, our expectations were that Approx_1 could result in smaller and more efficient RTL models. However, the internal Vitis algorithm was able to generate better results using the Approx_0 version.

The second reason was the lack of use of DSPs in the RTL models generated from both models. We noticed that in both cases, Vitis was using the FPGA LUTs to perform the mathematical operations. While in the 32-bits float version the DSPs were being used. The reason is that the approximate versions utilize so few bits that Vitis would not allocate DSPs to perform the operations. Vitis' requirement for allocating DSPs for mathematical

Table 5.4: Updated description of the two approximate configurations of the tic-tac-toe CNN that will be used in the continuation of this work.

Experiments	Input	weights/bias	MACC Units			Feature Map			Results (%)
			total	int	frac	total	int	frac	
Approx_0	8	4	16	9	7	12	9	3	98
Approx_dsp	10	10	12	9	3	12	9	3	98

Source: The authors

operation is that the operands must be at least 10-bits large. On account of this behavior, we decided to create a new version of tic-tac-toe with a slightly different approximate configuration. Table 5.4 shows the two versions that will be used in the following of this work. Approx_0 is the same as presented in section 4.2.2. The goal of Approx_dsp is to force the use of DSPs for mathematical operations.

5.2.2 Multicycle Approach

Starting with the Multicycle approach, table 5.5 shows the result of the high-level synthesis of the two approximate versions alongside the baseline 32-bits float version. It can be seen that there is a significant reduction in the latency of the approximate versions compared to the 32-bit float version. This gain is mainly due to the simplification of the multiply-accumulate operations. In this metric, the Approx_0 version showed a 25% improvement while Approx_dsp showed a 20% improvement compared to the 32-bit float version.

We can also notice an unexpected increase in the consumption of FPGA resources. However, the implementation results showed that this increase happened only in the estimation provided by Vitis after the high-level synthesis. Table 5.6 shows the results from the implementation (logic synthesis, and place and route).

Starting with the consumption of FPGA resources, we can now see that both approximate versions are less resource-hungry than the non-approximate version. The sole exception is the DSP consumption by the Approx_dsp version which slightly exceeded the consumption of Multicycle. Since this increase came along with a decrease in latency, we can assume that Vitis found a better way to parallelize the multiply-accumulate operations when synthesizing Approx_dsp.

In the Approx_0 version, it was expected that no DSP would be used. However, even with all the arithmetic logic being implemented in LUT, there was still a significant decrease in LUT consumption compared to the Multicycle version. In the case of the

Table 5.5: Comparison of the high-level synthesis results between the baseline 32-bits float version and two approximate versions using the Multicycle approach. Percentage values are relative to the Artix-7 AC701 FPGA.

	Multicycle	**Approx_0**	**Approx_dsp**
Model config	32b float	input: 8b parameters: 4b macc: 16b fm: 12b	input: 10b parameters: 10b macc: 12b fm: 12b
Model Accuracy	98.20%	97.90%	98%
Target Period (nS)	10		
Loop Unroll	YES		
Input Reshape (factor)	NO		
Registers Reshape (factor)	NO		
Estimated Period (nS)	9.017	9.738	8.787
Latency (c.c)	37,537	28,183	30,199
External Interface	1 BRAM addr: 10b data: 32b	1 BRAM addr: 10b data: 8b	1 BRAM addr: 10b data: 16b
#BRAM	12	6	7
#DSP	42	0	77
#FF	102,112	24,624	16,507
#LUT	62,353	141,816	107,919
BRAM (%)	2%	1%	1%
DSP (%)	6%	0%	10%
FF (%)	38%	9%	6%
LUT (%)	46%	**105%**	80%

Source: the authors

Approx_dsp version, we can see that the use of DSPs to perform the multiply-accumulate operations allowed for an almost 50% reduction in LUT usage compared to the 32-bits float version. Both Approx versions also showed a significant decrease in memory elements allocation (BRAM and FF).

Looking at the energy consumption, we can see that the Approx versions consume significantly less dynamic power while static power was nearly the same for all versions. It caused Approx_0 and Approx_dsp to consume 30% and 20% less power than the base version respectively. This power reduction coupled with the decrease in execution time resulted in an even greater reduction in the energy consumed by the approximated designs in comparison to the non-approximated version. In this regard, best approach was Approx_0 which had a 47% reduction in total energy consumed while Approx_dsp had a 34% reduction. For the calculus of energy, we fixed a 10ns clock period for all three

Table 5.6: Comparison of the implementation results between the baseline 32-bits float version and two approximate versions using the Multicycle approach. Percentage values are relative to the Artix-7 AC701 FPGA.

	Multicycle	**Approx_0**	**Approx_dsp**
Model config	32b float	input: 8b parameters: 4b macc: 16b fm: 12b	input: 10b parameters: 10b macc: 12b fm: 12b
Model Accuracy	98.20%	97.90%	98%
Target Period (nS)	10		
Loop Unroll	YES		
Input Reshape	NO		
Registers Reshape	NO		
Post-route Period (nS)	10.032	10.587	9.588
Latency (c.c)	37,537	28,183	30,199
Power (W)	0.42	0.295	0.34
Dynamic (W)	0.297	0.173	0.218
Static (W)	0.123	0.122	0.123
Energy (μ J)	15.765	8.313	10.267
External Interface	1 BRAM addr: 10b data: 32b	1 BRAM addr: 10b data: 8b	1 BRAM addr: 10b data: 16b
#BRAM	12	6	7
#DSP	42	0	77
#FF	79,622	23,528	15,340
#LUT	57,506	45,315	31,334
BRAM (%)	2%	1%	1%
DSP (%)	6%	0%	10%
FF (%)	30%	9%	6%
LUT (%)	43%	34%	23%

Source: the authors

versions.

5.2.3 Fully-Parallel Approach

The following comparison illustrates the effect of the approximate computing technique studied in a fully parallelized design. In this approach, we apply the complete array reshape to the input and all internal registers of the design. This enables the design to receive all input data in only one clock cycle and makes all data in all register banks accessible at once in a single access. This parallelization of the memory units allows Vitis to apply high parallelism to mathematical operations. Table 5.7 shows the result of the high-level synthesis of the approximate versions compared to the base 32-bit float version. It is important to note that all three designs are too large to be implemented on any FPGA known to the authors. Therefore, the implementation step will not be possible, and comparisons will only be made with the values obtained from the high-level synthesis.

Table 5.7: Comparison of the high-level synthesis results between the baseline 32-bits float version and two approximate versions using the Parallel approach. Percentage values are relative to the Artix-7 AC701 FPGA.

	Parallel	**Approx_0**	**Approx_dsp**
Model config	32b float	input: 8b parameters: 4b macc: 16b fm: 12b	input: 10b parameters: 10b macc: 12b fm: 12b
Model Accuracy	98.20%	97.9%	98%
Target Period (nS)	10		
Loop Unroll	YES		
Input Reshape (factor)	complete		
Registers Reshape (factor)	complete		
Estimated Period (nS)	7.52	7.3	7.3
Latency (c.c)	4,048	2,290	415
External Interface	1 bus 32,767b	1 bus 8,192b	1 bus 10,240b
#BRAM	0	0	96
#DSP	3,718	0	20,828
#FF	12,463,786	4,876,359	2,526,512
#LUT	84,057,304	21,376,700	46,102,336
BRAM (%)	0%	0%	13%
DSP (%)	**502%**	0%	**2,814%**
FF (%)	**4,630%**	**1,811%**	**938%**
LUT (%)	**62,450%**	**15,882%**	**34,251%**

Source: the authors

Following the trend observed so far, it is noticeable that the approximated versions show considerable latency improvements compared to the base version. Because of the approximations applied to the CNN, Vitis produced a design 45% faster when using the Approx_0 model, and 90% faster when using the Approx_dsp model. This gain is likely because Vitis can generate a more parallelized architecture when synthesizing the approximate models.

We can notice the difference in the generated architecture when comparing the two approximate versions. Approx_dsp achieved the best latency by using the largest amount of DSPs to perform the mathematical operations. In addition, the cost of the logic circuit needed to control this amount of DSPs also seems to be high. Since Approx_0 allocated less than half as many LUTs as Approx_dsp, despite not using DSPs to perform the math operations.

Comparing Approx_0 with the baseline version, we can say that Approx_0 is a better implementation in all metrics that we can evaluate with the high-level synthesis result. Approx_0 showed a reduction in the consumption of all FPGA elements while also reducing latency. In the case of Approx_dsp, we have to consider the increased DSP consumption compared to the 32-bit float version. Even though this increase provided an exceptional gain in latency, it is necessary to consider if this overhead would not impede the implementation of this approach.

5.2.4 Balanced Approach

Following the Balanced approach, we used Vitis to generate the RTL model of Approx_0 and Approx_dsp. In this approach, we tested different array reshape configurations on the internal registers in pursuit of versions that would provide an upgrade to the baseline 32-bits float version. Table 5.8 shows the result of the high-level synthesis step. With this approach, we allowed ourselves to change the array reshape factor of the Approx versions to get the best possible RTL model. Nevertheless, we can notice that despite the significant decrease in latency, Approx_0 resulted in a much worse RTL model in terms of FPGA resources consumption compared to the base version. However, the Approx_dsp version showed similar latency improvements as Approx_0 while also reducing the FPGA resources consumption compared to the 32-bits float version.

Unfortunately, due to the fact that Approx_0 consumes more resources than are available on the target FPGA, it was not possible to continue analyzing this design post-implementation. This shows that the approximate computing technique used may not be suitable for all scenarios. Therefore, table 5.9 presents the implementation results for the Balanced and Approx_dsp designs. We can notice that the approximate version outperforms the 32-bit float version in latency, power consumption, and FPGA resource consumption.

Approx_dsp showed the best result in terms of latency, where the use of the approximate model provided a 37% gain in this parameter. A similar improvement can be

Table 5.8: Comparison of the high-level synthesis results between the baseline 32-bits float version and two approximate versions using the Balanced approach. Percentage values are relative to the Artix-7 AC701 FPGA.

	Balanced	**Approx_0**	**Approx_dsp**
Model config	32b float	input: 8b parameters: 4b macc: 16b fm: 12b	input: 10b parameters: 10b macc: 12b fm: 12b
Model Accuracy	98.20%	97.90%	98%
Target Period (nS)	10		
Loop Unroll	YES		
Input Reshape (factor)	NO		
Registers Reshape (factor)	conv1: 192 conv2: 16 FCs: complete	conv1: 192 conv2: 16 FCs: complete	conv1: 48 conv2: 16 FCs: complete
Estimated Period (nS)	9.017	9.688	8.787
Latency (c.c)	33,119	22,435	21,046
External Interface	1 BRAM addr: 10b data: 32b	1 BRAM addr: 10b data: 8b	1 BRAM addr: 10b data: 16b
#BRAM	58	325	55
#DSP	240	0	131
#FF	134,299	688,964	101,227
#LUT	123,745	3,584,999	102,475
BRAM (%)	8%	45%	8%
DSP (%)	32%	0%	18%
FF (%)	50%	**256%**	38%
LUT (%)	92%	**2,663%**	76%

Source: the authors

noticed in energy consumption, where Approx_dsp consumed 39% less energy than the 32-bits float version. Since the power consumption of both designs is similar, we can conclude that the main cause of this reduction is the decrease in latency, which for a fixed clock period means a reduction in the total operating time of the circuit.

From the point of view of FPGA resource consumption, it can be seen that the approximate version is less resource-hungry than the base 32-bits float version. Apart from the consumption of FFs which has slightly increased, the consumption of all FPGA elements has significantly decreased.

Table 5.9: Comparison of the implementation results between the baseline 32-bits float version and two approximate versions using the Balanced approach. Percentage values are relative to the Artix-7 AC701 FPGA.

	Balanced	**Approx_0**	**Approx_dsp**
Model config	32b float	input: 8b parameters: 4b macc: 16b fm: 12b	input: 10b parameters: 10b macc: 12b fm: 12b
Model Accuracy	98.20%	97.90%	98%
Target Period (nS)	10		
Loop Unroll	YES		
Input Reshape	NO		
Registers Reshape	conv1: 192 conv2: 16 FCs: complete	conv1: 192 conv2: 16 FCs: complete	conv1: 48 conv2: 16 FCs: complete
Post-route Period (nS)	9.017	—	8.787
Latency (c.c)	33,119	22,435	21,046
Power (W)	0.802	—	0.773
Dynamic (W)	0.668	—	0.648
Static (W)	0.134	—	0.125
Energy (μ J)	26.561	—	16.268
External Interface	1 BRAM addr: 10b data: 32b	—	1 BRAM addr: 10b data: 16b
#BRAM	58	—	56
#DSP	240	—	131
#FF	82,233	—	89,517
#LUT	85,166	—	63,047
BRAM (%)	8%	—	8%
DSP (%)	32%	—	18%
FF (%)	31%	—	33%
LUT (%)	63%	—	47%

Source: the authors

5.3 Comparison With State-Of-The-Art

We selected one Multicycle implementation and one Balanced implementation for a brief comparison with state-of-the-art works. The balanced version chosen was Approx_0 and the Balanced version chosen was Approx_dsp.

(LO; LAU; SHAM, 2018) presented an accelerator that achieved a throughput of 5783 images/s. We did not present this metric in our work, however, for a clock period of 10ns, Multicycle and Balanced would have a throughput of 3548 image/s and 4750/s respectively. It is important to notice that our use case is simpler. Nevertheless, our results are in the same order of magnitude even though we did not optimize for that specific metric.

(CHO; KIM, 2020) had a much greater latency reduction than our work. However, it was optimized for this specific metric, while in our work we tried to achieve an overall improvement. As shown in Table 3.5, the authors allowed the approximate accelerator to have a higher FPGA resources consumption than the 32-bit floating-point version in exchange for performance.

(FENG et al., 2016) presented a significantly better execution time improvement than our work with 66% improvement in frequency compared with the floating-point implementation. However, although their proposed approximate accelerator is up to 90% less energy-hungry than CPU and GPU implementations, it is more energy-hungry than the floating-point implementations.

Table 5.10: Summary of most similar related works.

Reference	**ANN**	**FPGA**	**Technique**	**Activements**
(ZHOU; JIANG, 2015)	AlexNet	Virtex7	11-bit fixed-point architecture	16x faster than double precision CPU implementation with zero accuracy loss.
(LO; LAU; SHAM, 2018)	LeNet-5	Zynq UltraScale+	4-bit fixed-point architecture with 8-bit addition	Throughput of 5783 images/s at 50MHz with only 0.57% accuracy loss compared with floating-point implementation.
(CHO; KIM, 2020)	LeNet-5	Zynq UltraScale+	20-bit fixed-point architecture	90% latency reduction with similar area and only 0.01% accuracy loss compared with floating-point implementation.
(FENG et al., 2016)	LeNet-5	Zynq-7000	24-bit fixed-point architecture and approximate TanH function	66% higher frequency and overall area reduction with zero accuracy loss compared with floating-point implementation. Up to 93% energy saving compared with CPU and GPU implementations.
Multicycle	tic-tac-toe	Artix-7	Custom fixed-point architecture: 8b input 4b weights and bias 16b adders 12b feature maps	Overall area reduction, 47% energy saving, and 25% latency reduction with only 0.3% accuracy loss compared with 32-bit floating-point implementation.
Balanced	tic-tac-toe	Artix-7	Custom fixed-point architecture: 10b input 10b weights and bias 12b adders 12b feature maps	Overall area reduction, 39% energy saving, and 37% latency reduction with only 0.2% accuracy loss compared with 32-bit floating-point implementation.

Source: The authors

6 CONCLUSION

In this work, we tested whether the approximate computing paradigm could improve the implementation of CNNs on FPGAs. The evaluation was done on three aspects: Application execution latency, total energy consumption during application execution, and FPGA resources consumption (BRAM, LUT, FF, and DSP). Starting from a non-approximate CNN trained to identify the letters X and O, we created two approximate versions using the fixed-point data quantization technique with tiny accuracy overhead.

We then defined three different FPGA implementations for the non-approximate CNN. Each FPGA implementation was developed using a different approach to the trade-off between performance (latency) and cost (energy and FPGA resources consumption). Next, using the same approaches, we implemented the two approximate versions. We managed to produce at least one approximate design that constituted an improvement over the non-approximate version for each approach used. With this work, we showed that it is possible to perform simple modifications in existing CNN models to improve their deployment on FPGAs. We also showed that the results obtained from this work are on par with state-of-the-art works in the literature.

In future work, we plan to test different data quantization techniques and sophisticate the method used for applying it to the CNN model. In this work, we applied the same approximation level to every CNN layer. We believe that by using a granular approach we can achieve better accuracy for the CNN model and better FPGA implementations using Vitis. Lastly, we want to explore different ways to influence the HLS using the pragmas available on Vitis, as we only used two of the twenty-two HLS pragmas available.

REFERENCES

ANSARI, M. S.; COCKBURN, B. F.; HAN, J. A hardware-efficient logarithmic multiplier with improved accuracy. In: **2019 Design, Automation Test in Europe Conference Exhibition (DATE)**. [S.l.: s.n.], 2019. p. 928–931.

CANZIANI, A.; PASZKE, A.; CULURCIELLO, E. **An Analysis of Deep Neural Network Models for Practical Applications**. 2017.

CHO, M.; KIM, Y. Implementation of data-optimized fpga-based accelerator for convolutional neural network. In: **2020 International Conference on Electronics, Information, and Communication (ICEIC)**. [S.l.: s.n.], 2020. p. 1–2.

COURBARIAUX, M. et al. **Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1**. 2016.

COUSSY, P. et al. An introduction to high-level synthesis. **IEEE Design & Test of Computers**, IEEE, v. 26, n. 4, p. 8–17, 2009.

ELBTITY, M. E. et al. High speed, approximate arithmetic based convolutional neural network accelerator. In: **2020 International SoC Design Conference (ISOCC)**. [S.l.: s.n.], 2020. p. 71–72.

FENG, G. et al. Energy-efficient and high-throughput fpga-based accelerator for convolutional neural networks. In: **2016 13th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)**. [S.l.: s.n.], 2016. p. 624–626.

GAJSKI, D. D.; RAMACHANDRAN, L. Introduction to high-level synthesis. **IEEE Design & Test of Computers**, IEEE, v. 11, n. 4, p. 44–54, 1994.

GREFENSTETTE, E. et al. **A Deep Architecture for Semantic Parsing**. 2014.

GUO, K. et al. [dl] a survey of fpga-based neural network inference accelerators. **ACM Transactions on Reconfigurable Technology and Systems**, v. 12, p. 1–26, 03 2019.

HASHEMI, S.; BAHAR, R. I.; REDA, S. Drum: A dynamic range unbiased multiplier for approximate applications. In: **2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. [S.l.: s.n.], 2015. p. 418–425.

HINTON, G.; SEJNOWSKI, T. J. **Unsupervised learning: foundations of neural computation**. [S.l.]: MIT press, 1999.

JAIN, S. et al. Compensated-dnn: Energy efficient low-precision deep neural networks by compensating quantization errors. In: **2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)**. [S.l.: s.n.], 2018. p. 1–6.

KULKARNI, P.; GUPTA, P.; ERCEGOVAC, M. Trading accuracy for power in a multiplier architecture. **J. Low Power Electronics**, v. 7, p. 490–501, 12 2011.

LO, C. Y.; LAU, F. C. M.; SHAM, C.-W. Fixed-point implementation of convolutional neural networks for image classification. In: **2018 International Conference on Advanced Technologies for Communications (ATC)**. [S.l.: s.n.], 2018. p. 105–109.

LíBANO, F. Reliability analysis of neural networks in fpgas. 2018.

MA, Y. et al. Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks. 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. 45–54. ISBN 9781450343541. Available from Internet: <https://doi.org/10.1145/3020078.3021736>.

MADDISON, C. J. et al. **Move Evaluation in Go Using Deep Convolutional Neural Networks**. 2015.

MCFARLAND, M. C.; PARKER, A. C.; CAMPOSANO, R. The high-level synthesis of digital systems. **Proceedings of the IEEE**, IEEE, v. 78, n. 2, p. 301–318, 1990.

MITCHELL, J. N. Computer multiplication and division using binary logarithms. **IRE Transactions on Electronic Computers**, EC-11, n. 4, p. 512–517, 1962.

MRAZEK, V. et al. Evoapprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods. In: **Design, Automation Test in Europe Conference Exhibition (DATE), 2017**. [S.l.: s.n.], 2017. p. 258–261.

MUTHURAMALINGAM, A.; HIMAVATHI, S.; SRINIVASAN, E. Neural network implementation using fpga: Issues and application. **Int. J. Inform. Technol.**, v. 4, 11 2007.

NURVITADHI, E. et al. Can fpgas beat gpus in accelerating next-generation deep neural networks? 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. 5–14. ISBN 9781450343541. Available from Internet: <https://doi.org/10.1145/3020078.3021740>.

QIU, J. et al. Going deeper with embedded fpga platform for convolutional neural network. In: . New York, NY, USA: Association for Computing Machinery, 2016. (FPGA '16), p. 26–35. ISBN 9781450338561. Available from Internet: <https://doi.org/10.1145/2847263.2847265>.

RODRIGUES, G.; KASTENSMIDT, F.; BOSIO, A. Survey on approximate computing and its intrinsic fault tolerance. **Electronics**, v. 9, p. 557, 03 2020.

RéSEAU de Neurones Artificiels. Available from Internet: https://fr.wikipedia.org/wiki/R%C3%A9seau_de_neurones_artificiels. 2021.

SIMONYAN, K.; ZISSERMAN, A. **Very Deep Convolutional Networks for Large-Scale Image Recognition**. 2015.

TASOULAS, Z.-G. et al. Weight-oriented approximation for energy-efficient neural network inference accelerators. **IEEE Transactions on Circuits and Systems I: Regular Papers**, v. 67, n. 12, p. 4670–4683, 2020.

TSANTEKIDIS, A. et al. Forecasting stock prices from the limit order book using convolutional neural networks. In: **2017 IEEE 19th Conference on Business Informatics (CBI)**. [S.l.: s.n.], 2017. v. 01, p. 7–12.

UMUROGLU, Y. et al. Finn: A framework for fast, scalable binarized neural network inference. 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. 65–74. ISBN 9781450343541. Available from Internet: <https://doi.org/10.1145/3020078.3021744>.

VASICEK, Z.; SEKANINA, L. Evolutionary approach to approximate digital circuits design. **IEEE Transactions on Evolutionary Computation**, v. 19, n. 3, p. 432–444, 2015.

VENKATARAMANI, S. et al. Approximate computing and the quest for computing efficiency. In: **Proceedings of the 52nd Annual Design Automation Conference**. New York, NY, USA: Association for Computing Machinery, 2015. (DAC '15). ISBN 9781450335201. Available from Internet: <https://doi.org/10.1145/2744769.2751163>.

VENKATESAN, R. et al. Macaco: Modeling and analysis of circuits for approximate computing. In: **2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. [S.l.: s.n.], 2011. p. 667–673.

WANG, E. et al. Deep neural network approximation for custom hardware: Where we've been, where we're going. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 52, n. 2, may 2019. ISSN 0360-0300. Available from Internet: <https://doi.org/10.1145/3309551>.

WHY convolutions always use odd-numbers as filter_size. Available from Internet: https://datascience.stackexchange.com/questions/23183/why-convolutions-always-use-odd-numbers-as-filter-size. 2021.

ZERVAKIS, G.; AMROUCH, H.; HENKEL, J. Design automation of approximate circuits with runtime reconfigurable accuracy. **IEEE Access**, v. 8, p. 53522–53538, 2020.

ZERVAKIS, G. et al. Approximate computing for ml: State-of-the-art, challenges and visions. In: **Proceedings of the 26th Asia and South Pacific Design Automation Conference**. New York, NY, USA: Association for Computing Machinery, 2021. (ASPDAC '21), p. 189–196. ISBN 9781450379991. Available from Internet: <https://doi.org/10.1145/3394885.3431632>.

ZHOU, Y.; JIANG, J. An fpga-based accelerator implementation for deep convolutional neural networks. In: **2015 4th International Conference on Computer Science and Network Technology (ICCSNT)**. [S.l.: s.n.], 2015. v. 01, p. 829–832.

ZOU, J.; HAN, Y.; SO, S.-S. Overview of artificial neural networks. **Methods in molecular biology (Clifton, N.J.)**, v. 458, p. 14–22, 01 2009.