UNIVERSIDADE D
COIMBRA

Virgílio Manuel Henriques Cruz

EXPLORING STOCHASTIC COMPUTING APPLIED TO ARTIFICIAL PERCEPTION

A CONVOLUTIONAL NEURAL NETWORK IMPLEMENTATION ON RECONFIGURABLE LOGIC

VOLUME 1

Dissertation submitted to the Department of Electrical and Computer Engineering of the Faculty of Science and Technology of the University of Coimbra in partial fulfillment of the requirements for the Degree of Master Science

July 2022

University of Coimbra
Faculty of Sciences and Technology

EXPLORING STOCHASTIC COMPUTING APPLIED TO ARTIFICIAL PERCEPTION
A Convolutional Neural Network implementation on Reconfigurable Logic

Virgílio Manuel Henriques Cruz

Dissertation submitted to the Department of Electrical and Computer Engineering of the Faculty of Science and Technology of the University of Coimbra in partial fulfillment of the requirements for the Degree of Master Science in Electrical and Computer Engineering. Supervised by Prof. Dr. Jorge Nuno de Almeida e Sousa Almada Lobo

July 2022

Acknowledgments

I begin by thanking my supervisor, Professor Jorge Lobo, for his availability throughout the dissertation, for his follow-up, and his work suggestions.

I also want to thank my classmates for all the support and help throughout my academic journey and all my friends who accompanied me and helped me personally and academically.

To my family who supported me in all adversities, especially to my mother who made this journey possible, with great effort and patience.

A special thanks to Inês, for her company and love. Thank you for believing in me, and for your patience and support in the most challenging times.

Abstract

Stochastic Computing is a computing method that performs probability-based operations, allowing simple circuits with low energy consumption and significant fault tolerance. Convolutional Neural Networks essentially require weighted sums, despite the huge computational weight associated with these algorithms. Due to the simplicity of operations calculated by these algorithms, we propose applying Stochastic Computing to Convolutional Neural Networks to implement more efficient edge computing classifiers.

This dissertation intends to contextualize this problem and review the literature in the corresponding areas to base the work carried out later. Brief reviews of possible implementation methodologies have been carried out, to give an understanding of how to use Stochastic Computing in CNN image classification.

We propose the Modified LeNet-5 based on LeNet-5, where the modifications serve to divide the network into blocks easily exchangeable for Stochastic Computing blocks. The modified LeNet-5 achieved a training performance of 97% on Tensorflow. After training, we implemented the Modified LeNet-5 on custom reconfigurable hardware for the classification of images from the MNIST dataset. At the end of our work, we have a CNN baseline full RTL implementation on which Stochastic Computing can be added. The pipeline is fully implemented and working but not fully debugged.

KEYWORDS: Stochastic Computing, Convolutional Neural Networks, Machine Learning, Reconfigurable Logic

Resumo

A Computação Estocástica é um método de computação que realiza operações baseadas em probabilidade, permitindo circuitos simples com baixo consumo de energia e significativa tolerância a falhas. As Redes Neurais Convolucionais requerem essencialmente somas ponderadas, apesar do enorme peso computacional associado a esses algoritmos. Devido à simplicidade dos cálculos realizados por estes algoritmos, propomos a aplicação de Computação Estocástica a Redes Neurais Convolucionais para implementar classificadores de tecnologia de ponta mais eficientes.

Esta dissertação pretende contextualizar este problema e rever a literatura nas áreas correspondentes para fundamentar o trabalho realizado posteriormente. Foram realizadas breves revisões de possíveis metodologias de implementação para dar uma compreensão de como a Computação Estocástica pode ser usada na classificação de imagens em CNNs.

Propomos a rede LeNet-5 Modificada baseada na rede LeNet-5 original, onde as modificações servem para dividir a rede em blocos facilmente substituíveis por blocos de Computação Estocástica. A rede LeNet-5 modificada alcançou um desempenho de treino de 97% no Tensorflow. Após o treino, implementámos a rede LeNet-5 Modificada em hardware reconfigurável personalizado para a classificação de imagens do conjunto de dados MNIST. No final do nosso trabalho, obtivemos uma implementação RTL modelo completa da CNN à qual pode ser adicionada a computação estocástica. O pipeline está totalmente implementado e funcional, no entanto carece de debug futuro.

PALAVRAS-CHAVE: Computação Estocástica, Redes Neuronais Convolucionais, Aprendizagem de Máquina, Lógica Reconfigurável

Acronyms

AI Artificial Intelligence.

AP Artificial Perception.

ASIC Aplication-Specific Integrated Circuit.

BI Bayesian Inference.

CCE Categorical Cross-Entropy.

CNN Convolutional Neural Network.

CPU Central Processing Unit.

DNN Deep Neural Network.

FPGA Field Programmable Gate Array.

GPU Graphics Processing Unit.

HDL Hardware Description Language.

ISR-UC Institute of Systems and Robotics - University of Coimbra.

LFSR Linear Feedback Shift Register.

LUT Look-up Table.

QPS Quartus Prime Software.

RAM Random Access Memory.

ReLU Rectified Linear Unit.

RMSE Root-Mean-Square Error.

RNG Random Number Generator.

RTL Register Transfer Level.

SC Stochastic Computing.

SDRAM Synchronous Dynamic Random-Access Memory.

SNG Stochastic Number Generator.

TPU Tensor Processing Unit.

VHDL Very High Speed Integrated Circuit Hardware Description Language.

WBG Weighted Binary Generator.

List of Figures

List of Tables

Contents

Chapter 1. Introduction

This chapter summarizes the motivations and objectives of this dissertation and provides a guide to contextualize the reader. A brief description of the related works on the basis for the development of our work is made. Moreover, our main contributions to the scientific community are analyzed.

1.1 Motivation

Moore's law, which says that the number of transistors on a chip doubles approximately every two years, is slowing down. Trying to increase the operating frequency of the circuits leads to an increase in the heat generation by the circuits, which has created the need to change the architectures through a greater number of processors per chip. While evermore transistors are still being built per chip, there are limits. For instance there is a cap on increasing the operating frequencies due to the excessive heat generated, and performance gains are now pursued with multi-core and non diverse processor architectures [1].

Modern computing hardware has limitations by stringent application requirements like extremely small size, low power consumption, and high reliability [2]. As result, the electronics industry explored other strategies, such as Stochastic Computing (SC), originally proposed in 1956 by John Von Neumann [3]. He considered designing computers based on probabilistic arithmetic with simple components instead of floating-point units. Initially, the idea was not very popular, as it would always be possible to increase performance by adding more transistors to the chips. The purpose of these circuits was to overcome unreliable hardware, a problem with the early developed logic circuits.

Complex algorithms such as Convolutional Neural Network (CNN) used to perform Artificial Perception (AP) can be implemented with probabilistic approaches in reconfigurable logic, using less power consumption and fewer hardware resources than deterministic methods [4].

This dissertation proposes the study of the implementation of artificial perception algorithms in CNNs computing with SC to understand if it is possible to achieve more efficient artificial perception systems, enabling more intelligent low power systems. It will follow previous work at the Institute of Systems and Robotics - University of Coimbra (ISR-UC) using SC. The base model of our work is the LeNet-5 neural network model, with several modifications to simplify its implementation on Field Programmable Gate Array (FPGA) environment.

1.2 Objectives

The main objective of this dissertation is to explore the use of Stochastic Computing in CNNs. A second objective is to compare the trade-offs of the proposed approach with deterministic computing methods and evaluate parameters such as energy consumption, resource utilization, and fault tolerance. That said, the goal is not to achieve a system with greater classification efficiency than existing models but a system capable of classifying images with relevant efficiency, lower use of hardware resources, and lower energy consumption.

1.3 Related Work

In this section, we describe the most relevant works to take care of in the development of this dissertation. These works present an overview of Stochastic Computing algorithms appliances to develop our work.

Stochastic Computing in Bayesian Inference:

This dissertation got its basis on Stochastic Computing, and some works carried out at ISR-UC are used as a reference [5]. These works are an excellent introduction to Stochastic Computing methods in reconfigurable logic, even mainly directed to Bayesian Inference (BI). These works are briefly described below. For more details, read [6, 7, 8]

- In [6] is proposed a Bayesian system to avoid collisions in an autonomous robot, fully implemented on an FPGA. The robot example uses up 60% of a low-end FPGA, with the random number generation requiring the most resources. This implementation can be interesting when the onboard computing power is limited.
- In [7] is proposed the development of a toolchain that, given a high-level description, builds a low-level description of a circuit computing probabilistic inference. The machine uses stochastic arithmetic to approximate the result of exact inference and behaves as a fault-tolerant circuit that can run at low voltage with low energy. The authors believe stochastic binary signals are better candidates for bio-inspired machines than analog signals to code for probabilities.
- In [8] is proposed a general hardware architecture dedicated to solving sensor fusion problems using Bayesian Inference. The proposed machine is an excellent candidate for solving tractable inference problems and could successfully be used in real applications.

Stochastic Computing to Implement Image Processing:

In order to better understand how stochastic computing algorithms work in the field of image processing, the following works are an excellent introduction to later arrive at Stochastic Computing algorithms in the implementation of Convolutional Neural Networks. For more details, read [9, 10].

- In [9] is presented an introduction and prove a stochastic absolute value function, a demonstration of a mathematical analysis of a stochastic Tanh function, and a presentation of a quantitative analysis of a one-parameter linear gain function and proposition of a new two-parameter version. The authors concluded that SC systems are highly tolerant of soft errors and proved that SC typically

consumes substantially fewer hardware resources than conventional computing while performing the same algorithm.

- In [10] is presented a proposal of three new stochastic architectures for three edge detection algorithms. The proposed architectures require less area and also consume less power.

Stochastic computing in Convolutional Neural Network:

As our work consists of implementing a Convolutional Neural Network using Stochastic Computing, the following works talk about this. Through them, it is possible to have an improved view of the operation of Stochastic Computing algorithms in the implementation of Convolutional Neural Networks. For more details, read [4, 11].

- In [4], an overview of Stochastic Computing in Convolutional Neural Networks is presented. SC might be a good idea, specifically when people are still actively researching and optimizing SC circuits.
- In [11], stochastic techniques that efficiently enable hardware training of implemented networks are presented. The authors present stochastic techniques to implement all the elements that constitute a feedforward neural network. The authors proved that stochastic neural networks are especially suitable for latency problems.

LeNet-5:

LeNet-5 is a Convolutional Neural Network (CNN) proposed by Yann LeCun et al. in 1998 for handwriting character recognition [12]. The authors present methods and architectures that offer generic solutions for real-life document recognition systems composed of multiple modules. The authors also built the MNIST dataset, used for train and test the LeNet-5 CNN.

In our work we will build upon the above related work, and use LeNet-5 as a testbed for efficient FPGA classifier implementations and exploring gains from introducing Stochastic Computing.

1.4 Main Contributions

The main contributions of this dissertation are:

- We provide an overview on Stochastic Computing solutions applicable to CNN computation.
- Version of the LeNet-5 CNN adapted for further SC implementation with approximately 97% accuracy on Tensorflow.
- Matlab script to convert and sort the parameters obtained on Tensorflow to memory format for FPGA, enabling an efficient RTL implementation.
- Implementation of the CNN on FPGA built modularly to enable future changes to have Stochastic Computing in some of the computation steps. The full pipeline is implemented and working but not fully debugged.

1.5 Dissertation Outline

This dissertation is divided in the following chapters:

- Chapter 1 – Introduction

  A chapter on the motivation and objectives behind our work, where we describe significant related works and the main contributions of our project.

- Chapter 2 – Background on Stochastic Computing and Convolutional Neural Networks

  This chapter describes the theoretical background of Stochastic Computing, Artificial Perception algorithms, Reconfigurable Logic, and Convolutional Neural Networks.

- Chapter 3 – Exploratory implementation of Stochastic Computing and Convolutional Neural Networks

  Description of the implementation process and results analysis.

- Chapter 4 – Experimental Results

  Analysis of results obtained on Tensorflow and Quartus Prime Software (QPS).

- Chapter 5 – Conclusions and Future Work

  Conclusions on the project and suggestions for future work.

- References

  Bibliographical references on the basis of the topics covered.

Chapter 2. Background on Stochastic Computing and Convolutional Neural Networks

In this chapter we provide a theoretical background on Stochastic Computing, Convolutional Neural Networks, and Stochastic Computing appliance in Convolutional Neural Networks. We also present an overview of the hardware and tools used.

2.1 Reconfigurable Logic

The low complexity of stochastic circuits and the flexibility to have low energy consumption associated with the reconfigurable logic make its implementation on Field Programmable Gate Array (FPGA) a good solution. Compared to FPGAs, Central Processing Unit (CPU) and Graphics Processing Unit (GPU) devices are quite complex. The FPGA devices are characterized by two-dimensional arrangements of logic elements, capable of performing Boolean operations and storing their results. The ability to reconfigure a chip reduces the time to market and the cost of production of a system.

What defines the reconfigurability of these devices is the Look-up Table (LUT) of their logical elements, formed by a small memory that associates the results with a combination of input variables. Custom dedicated circuits have greater efficiency than FPGA devices. However, the great flexibility of FPGA devices is a determining factor for our implementation due to the ease of making changes to the circuits. Development boards provide power and communication support for the main chip and peripherals that expand its capabilities [11, 13].

2.1.1 Field Programmable Gate Array (FPGA) and Project Environment

The chosen board for the implementation of the system is Terasic DE2-115 (Figure 2.1). The chip present in the board is the Altera Cyclone IV [14]. It has 114480 logic elements, 3888Kbits of embedded memory, 528 user input/output pins, 8MB of flash memory, and two external 64MB Synchronous Dynamic Random-Access Memory (SDRAM). The main reason for choosing this device was its availability, as, among the available FPGAs, it meets all the requirements for carrying out the project.

Logic elements (LE) are the smallest units of logic in the Cyclone IV device architecture. LEs are compact and provide advanced features with efficient logic usage. Each LE has the following features:

- A four-input look-up table (LUT), which can implement any function of four variables.
- A programmable register.
- A carry and register chain connection.
- The ability to drive local, row, column, register chain and direct link interconnects.
- Register packing and feedback support.

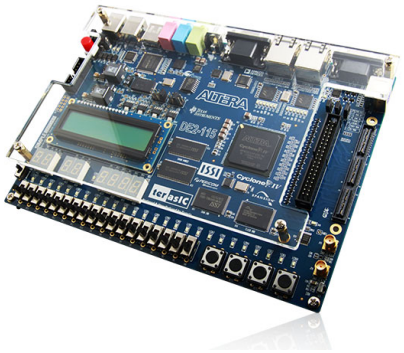

Figure 2.1: Altera DE2-115 [15].

An overview of the Cyclone IV Device Logic Elements is shown in Figure 2.2. For more details on the Cyclone IV device, read [14].

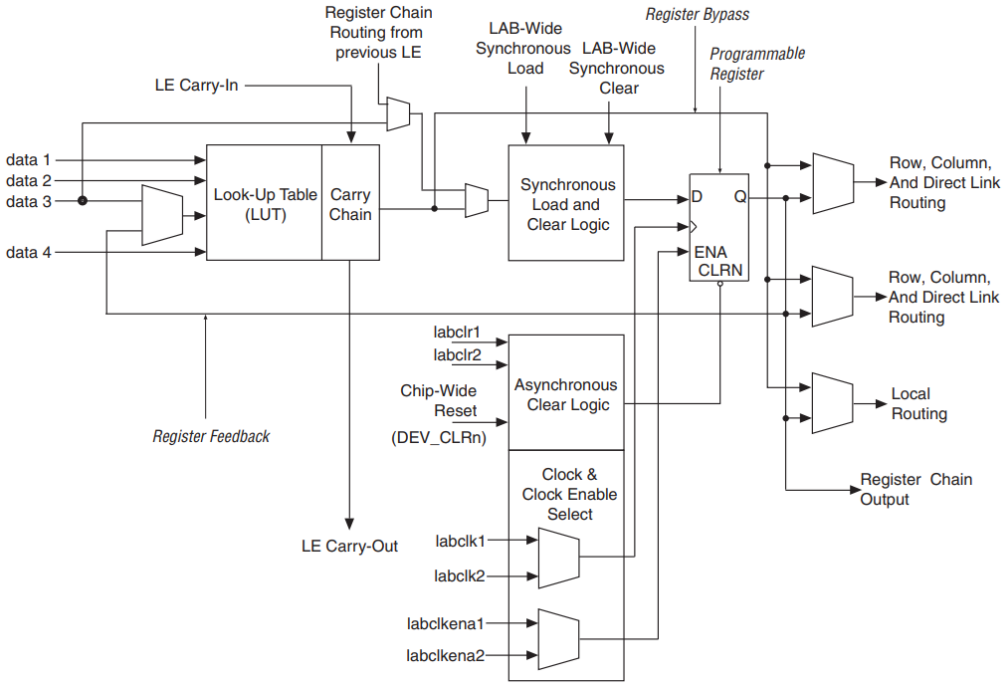

Figure 2.2: Cyclone IV Device Logic Elements [14].

The project's development environment is Quartus Prime Software (QPS) 18.1 Standard Edition [16], a programmable logic device design software produced by Intel FPGA. Quartus Prime Software allows the analysis and synthesis of Hardware Description Language (HDL) designs, compiles the designs, performs temporal analysis, examines Register Transfer Level (RTL) diagrams, simulates the design's reaction to different stimuli, and then configures the device with the programmer. Although it is possible to simulate the design directly in the QPS, ModelSim - Intel FPGA Starter Edition 10.5b [17] is used, allowing a more dynamic design simulation. This software is recommended for simulating all Intel FPGA designs.

2.2 Stochastic Computing

The first person to design computers based on probabilistic arithmetic was John Von Neumann in 1956 [3], but it was Brian R. Gaines who popularized the term of Stochastic Computing (SC) [18]. SC is an unconventional computing method with great potential due to its extreme simplicity of computing elements and high fault tolerance. Initially, it decodes a binary number in a bitstream that represents the magnitude of the value with the frequency of 1's. Then, the computations are in the stochastic domain with simple logic gates. In the end, the stochastic bitstream will be converted back to binary with a bit counter, counting the frequency of 1's. With SC, it is possible to perform complex numerical operations with simple circuits, such as a single AND gate to perform multiplication, defining the output as in Equation 2.1, and a single multiplexer unit to perform scaled addition, defining the output as in Equation 2.2. In both equations, 'P' refers to the probability of the stochastic streams, 'S1', 'S2', 'S3', and 'S4'. We can simplify Equation 2.2 to Equation 2.3 if 'P(S3)'=0.5.

$$S3 = P(S3) = P(S1)P(S2) = S1 \times S2. \tag{2.1}$$

$$S4 = P(S3)P(S1) + (1 - P(S3))P(S2). \tag{2.2}$$

$$S4 = \frac{P(S1) + P(S2)}{2} = \frac{S1 + S2}{2}. \tag{2.3}$$

Figures 2.3 (A) and (B) show examples of the AND gate multiplication and the MUX scaled adder operations, respectively. The multiplication is done with an AND gate in the unipolar format and an XNOR gate in the bipolar format. The addition can be done with a single MUX in both representation formats. The unipolar format maps the value between 0 and 1, while the bipolar format maps the value between -1 and 1. The bipolar representation can be mapped from unipolar via the function $y = 2x-1$, where y is the bipolar value and x is the unipolar value.

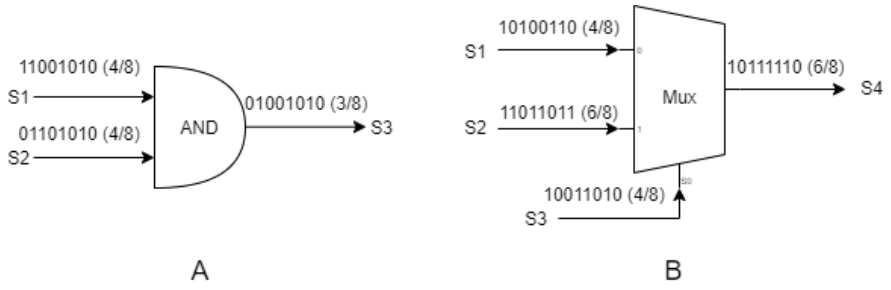

Figure 2.3: SC arithmetic operations. (A) Multiplication. (B) Scaled addition. Adapted from [2].

A Stochastic Number Generator (SNG) is required to perform stochastic arithmetic operations, which consists of a Random Number Generator (RNG), and a comparator working synchronously to generate a stochastic bitstream from a binary number. However, there is a challenge, as there may be a correlation between the bitstreams, creating accuracy problems. A SC output will only be accurate if both working streams are not correlated. The correlation is defined in Equation 2.4.

$$\sum_{i=1}^{n} S1(i)S2(i) = \frac{\sum_{i=1}^{n} S1(i) \times \sum_{i=1}^{n} S2(i)}{n}. \tag{2.4}$$

Where ‘S’ is the stochastic bitstream and ‘n’ is the bit length. The accuracy is dependent on the randomness and the length of the stochastic stream. There are some strategies to avoid the correlation between the bitstreams [4, 19]. One strategy is the use of a Toggle Flip-Flop and an XOR gate to perform addition (Figure 2.4). If the inputs are equal at each clock cycle, they propagate to the output. Otherwise, the output is the state of the TFF, and the TFF is toggled. If the bitstream length is sufficient to represent the result of the adder, it will always be accurate. Otherwise, the output will be rounded to the nearest representable number [20].

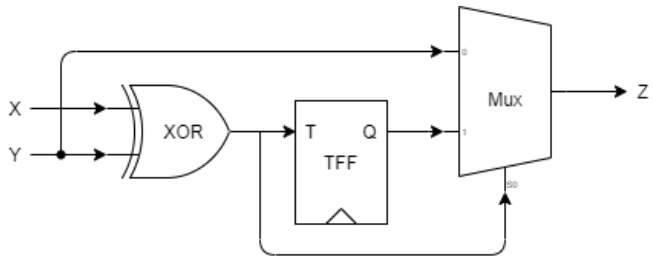

Figure 2.4: SC addition with Toggle Flip-Flop. Adapted from [20].

Figure 2.5 represents an example of a stochastic process, where a pseudorandom bitstream is generated in a Linear Feedback Shift Register (LFSR), compared with a binary input by a binary comparator or a weighted binary generator, generating a stochastic bitstream. The LFSR consists of XOR gates and a bit shift register (8-bit example in Figure 2.6), initialized with a specific value, generating a new bit in the bitstream at each clock cycle. Then, the stochastic logic circuits process the stochastic bitstream and finally convert back to the binary domain in a counter, which is usually a flip-flop counter.

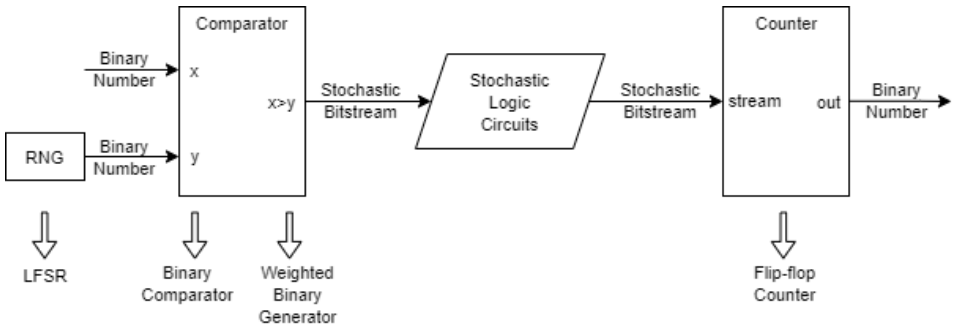

Figure 2.5: Example of an SC process. Adapted from [2].

2.2.1 Stochastic Bitstream Generator

Stochastic bitstreams are usually produced using an Random Number Generator (RNG) based on a Linear Feedback Shift Register (LFSR) to generate pseudo-random numbers and a Binary Comparator or Weighted Binary Generator (WBG) for comparing the generated pseudo-random number with the binary number to be converted and then generating the stochastic bitstream. Although LFSRs are typically used, there is a wide variety of RNGs [21].

- **LFSR**

  An LFSR consists of m flip-flops and goes through cycles of $n = 2^m - 1$ states cycles. The n-bit sequence produced is called pseudo-random because it is deterministic, despite the success in

some randomness tests. Shifted versions of sequences generated by LFSR have low correlation, as intended [2]. Figure 2.6 shows an example of an LFSR with a feedback polynomial equal to $X^8 + X^6 + X^5 + X^4$.

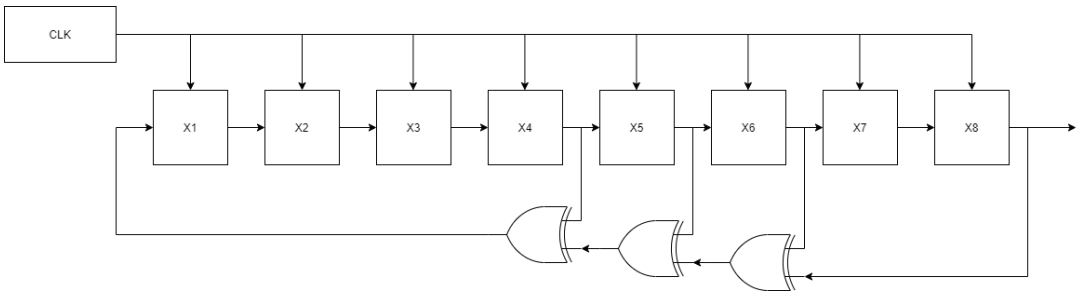

Figure 2.6: Example of 8-bit LFSR. Adapted from [22].

- **Binary comparator**

  A binary comparator receives two binary words with the same size, one of which corresponds to the input data and the other to the sequence generated in the LFSR. Figure 2.7 shows an example of a 4-bit binary comparator.

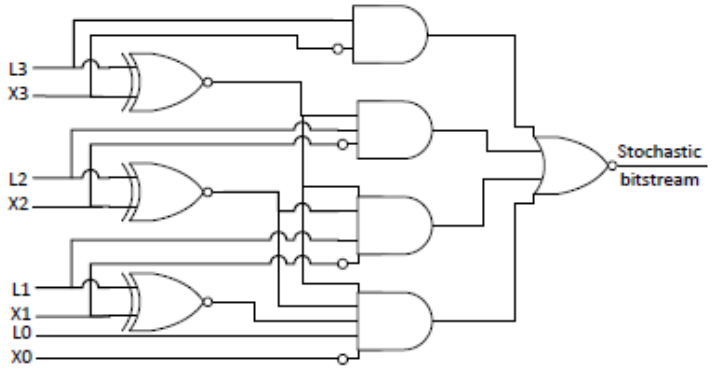

Figure 2.7: 4-bit binary comparator. Adapted from [4].

- **WBG**

  Proposed by Gupta [23] to reduce Stochastic Number Generator (SNG) cost, it has the same purpose as a binary comparator, although it works differently. Figure 2.8 shows that a WBG is composed of two levels. The first level consists of AND gates that receive random unbiased bitstreams from an LFSR and send the outputs, corresponding to Equation 2.5, to the next level. The second level consists of AND gates that receive the outputs from the previous level and the binary number to be converted, and their output will then enter an OR gate, from where the final output comes out. The final output probability is in Equation 2.6.

$$w_3 = r_3, w_2 = \overline{r_3}.r_2, w_1 = \overline{r_3}.\overline{r_2}.r_1, w_0 = \overline{r_3}.\overline{r_2}.\overline{r_1}.r_0. \tag{2.5}$$

$$P(x) = P(w_3)c_3 + P(w_2)c_2 + P(w_1)c_1 + P(w_0)c_0 = \frac{1}{2}c_3 + \frac{1}{2^2}c_2 + \frac{1}{2^3}c_1 + \frac{1}{2^4}c_0. \tag{2.6}$$

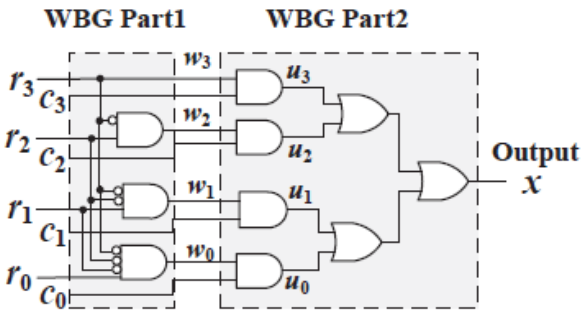

Figure 2.8: Weighted binary comparator. Adapted from [24].

In the case that several bitstreams can be related, it is possible to share the LFSR and the WBG part 1 between them, thus reducing the use of logic gates [23, 24].

2.2.2 Stochastic functions

This section describes the stochastic approach of CNN common activation functions as well as the max function to obtain the maximum value in a Max-pooling layer.

- **SC ReLU**

  ReLU function performs rectification and cuts off any negative value such that $f(x) = max(0, x)$. Li et al. [25] proposed a SC-based ReLU function block, as depicted in Fig. 2.9. In the stochastic domain, the ReLU amplitude will naturally be maxed out at 1, but this is not a problem because clipped ReLU has no appreciable accuracy deterioration [26]. A negative value must also be clipped to zero. The magnitude of negativity is determined by the number of 0's bits in the bipolar stochastic stream. The output will be required to be 1 when the accumulated value is less than the reference half value in a particular sampling time. Otherwise, the output will follow the pattern of emulated linear function from the FSM [4].

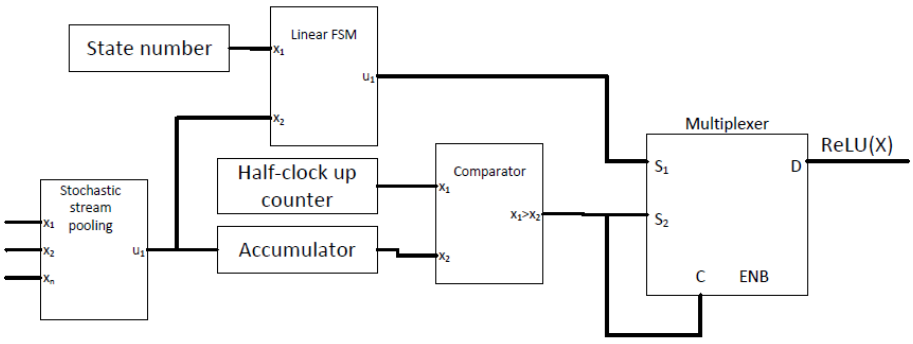

Figure 2.9: SC Relu. Adapted from [4].

- **Stanh**

  A stochastic approximation to the tanh function, Stanh, with both input and output signals coded as bipolar stochastic signals, may be implemented using a state machine, as in Figure 2.10. The output of the states has also been chosen in a particular way to obtain the tanh characteristic.

The approximate transfer function is $Stanh(N, x) \cong tanh(xN/2)$, when N is even. For small values of N, the approximation to a tanh function is relatively poor, but for large values of N, the approximation is much better [27].

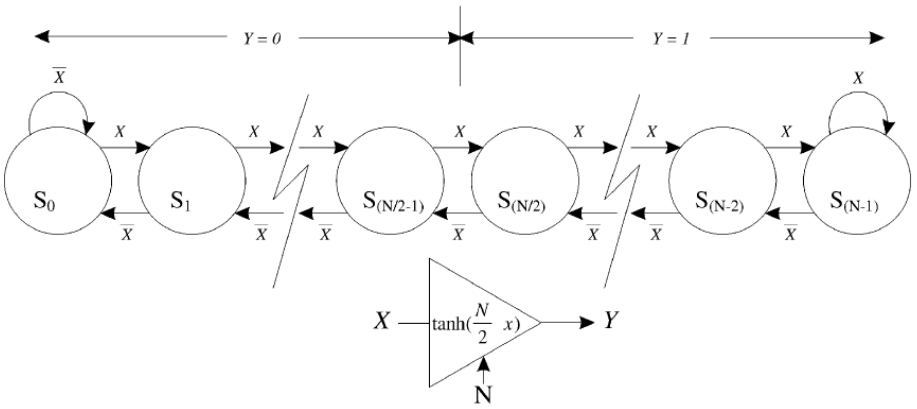

Figure 2.10: Stanh element: state transition. Adapted from [27].

Squaring stochastic stream can be conducted by delaying the input stream with D flip-flop before multiplying itself, as in Figure 2.11. In the case of a non-linear function, such as hyperbolic tangent (TanH), stochastic TanH (Stanh) uses a k-state finite state machine (FSM) [4].

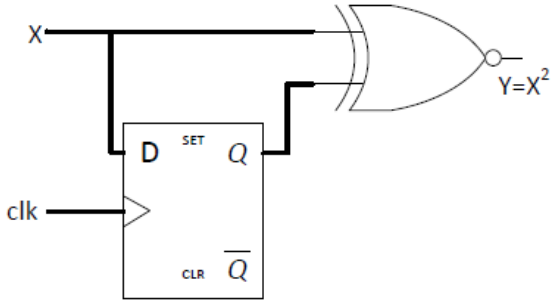

Figure 2.11: Stochastic squaring with D flip-flop. Adapted from [4].

- **Maxpool**

A stochastic maxpool block was proposed by Yu et al. in 2017 [28]. A novel stochastic MAX block could select whichever stream of higher value with only an XOR gate, FSM and MUX. With the XOR gate controlling the FSM state jumping, the probability of the opposite stream could be inferred from another bitstream by generating the condition of bit entanglement. Whenever the FSM sampled a 0 bit from the current bitstream, it implies a 1 bit on the opposite bitstream. If inequality between two bitstreams exists, the FSM state will be biased to the one with a higher magnitude, completing the MAX function with the MUX [4]. Cascading the MAX function block could realize the max-pooling function block, as shown in Figure 2.12.

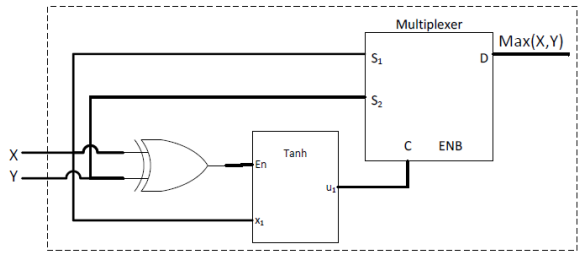

Figure 2.12: Stochastic MAX function. Adapted from [4].

2.3 Machine Learning

Machine Learning is an area of Artificial Intelligence (AI) responsible for algorithms dedicated to learning from data to solve complex problems, such as image classification. In this type of problem, the algorithm is fed with a set of specific data to detect patterns and subsequently identify those patterns in another different set of data. Some machine learning algorithms can perform image classification tasks better than humans.

Image classification is a Computer Vision problem based on pattern recognition through image processing algorithms, such as edge detection. A typical classification algorithm consists of a training phase and a classification phase. In the first phase, the algorithm learns the model's parameters from a set of labelled classified data, making the model able to map an image input to an output class. At the end of the training, the model can classify new inputs.

2.3.1 Convolutional Neural Networks

A Deep Neural Network (DNN) is a class of machine learning algorithms used to process complex information, such as images and videos. The nature of DNN consists of layers of addition and multiplication of numerical weights that compute the overall dimensionless probability values of an output class, which in turn allows the computer to decide based on the output value. Many DNN algorithm variations exist for a particular purpose, such as Convolutional Neural Network (CNN) for image processing and long short-term memory for neural-linguistic processing.

CNN can reduce multidimensional images into simple classes and is very popular in image classification and object recognition. A CNN can reduce large matrices into single values. The most distinctive component that discriminates CNN from other DNN algorithms is its convolution layer [4].

A Convolutional Neural Network consists of various layers, such as convolutional and fully-connected layers (performing the majority of the operations), pooling layers (used to prevent overfitting), and a classification layer (to classify the final results). A typical layer consists of 3D volumes of neurons, as shown in Figure 2.13. An overfitted model is a model that contains more parameters than can be justified by the data. One way to avoid overfitting is to stop the training as soon as performance on a validation set starts to get worse [29].

Convolutional Neural Networks take advantage of the fact that the input consists of images, and they constrain the architecture more sensibly. In particular, unlike a regular Neural Network, the layers of a CNN have neurons arranged in three dimensions (width, height, depth) [30].

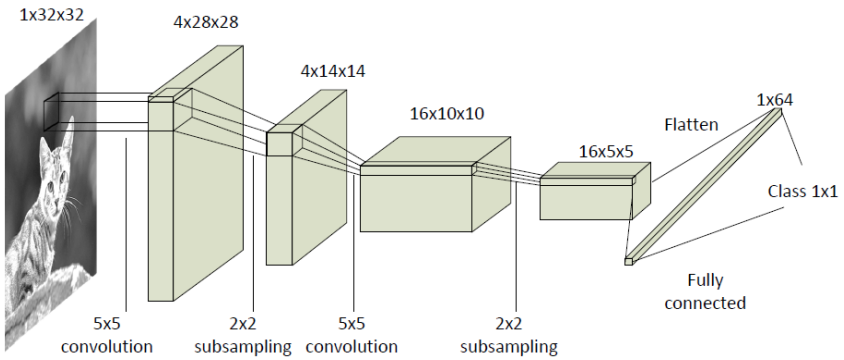

Figure 2.13: Architecture of classical LeNet-5 CNN. Adapted from [4].

The convolution process can be generalized as in Equation, 2.7

$$y_j^l = f(x_j^l) = f(\sum_{i=1}^{n}(x_i^{l-1} \times w_{ij}^{l-1}) + b_j^l), \tag{2.7}$$

where x_j^l is the convolved feature of the next layer, x_i^{l-1} is the feature from the previous layer, w_{ij}^{l-1} is the kernel weight matrix, and b_j^l is bias. 'l' is the layer number, 'i' denotes scan window number, 'n' is the total number of scan window, and 'j' is the depth of next feature map. Then, there is an activation function $f(x_j^l)$ (ReLU or Tanh). Finally, the final product y_j^l is aggregated so that the process can be repeated.

In a CNN, the convolution and activation layers are essential, and there may be other layers to reduce the variance of the output (normalization), to save memory (pooling), or to prevent overfitting (dropout). At the end of convolution, the matrix will be flattened into a single list of data. Then, those data will be fed to the neurons [4].

Each layer can be described as it follows:

- **Convolutional Layer** - Performs convolution, taking the input image and decomposing it into different feature maps. A kernel of weights is multiplied by a set of inputs, and the weighted inputs are added together. A bias, whose value is usually 1, is added to the summed weighted inputs to ensure that neurons fire. An activation function is applied to the accumulated sum to limit the output to a reasonable range. Results from the activation function goes through to the corresponding neurons in the next layer. Equation 2.8 describes the computation of a feature map's output size.

$$output_{size} = \frac{input_{width} - filter_{size} + 2 \times padding}{stride} + 1. \tag{2.8}$$

- **Activation Function** - Used to ensure nonlinearity in the network and eliminate unnecessary information. The most commonly used activation function is Rectified Linear Unit (ReLU), Equation 2.9, which returns 0 if it receives any negative input, but for any positive value x, it returns that value. Hyperbolic Tangent, Equation 2.10, is another example of an activation function that maps the negative inputs as strongly negative and the zero inputs near zero. In both equations x refers

to the pixel value.

$$ReLU(x) = max(0, x). \tag{2.9}$$

$$tanh(x) = \frac{sinh(x)}{cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}. \tag{2.10}$$

- **Pooling Layer** - Reduces the dimension of the features at each level of the network to save memory. Standard pooling methods are maximum and average pooling. In maximum pooling, a set of neurons are sub-sampled based on the size of a pooling filter. In contrast, the maximum neuron value in that filter passes to the corresponding neuron in the next layer, and the remaining neurons are dropped out, as shown in Equation 2.11 (with $Filter_{size} = 2 \times 2$). In average pooling, the forwarded value to the corresponding neuron in the next layer is the average of all neurons in a filter, as shown in Equation 2.12.

$$Passed_{neuron} \rightarrow max(4, 3.5, 1, 3) = 4. \tag{2.11}$$

$$Passed_{neuron} \rightarrow avg(4, 2, 1, 3) = 2.5. \tag{2.12}$$

- **Fully-Connected Layer** - Comprises the highest number of parameters because every neuron connects to all neurons in the previous layer, and parameters translate on the connections between those neurons. Inputs in this layer are multiplied with the corresponding weights and added to biases respectively, then nonlinearity is applied, as shown in Equation 2.13. The output passes to a classifier that converts the output neurons value to a probability between (0, 1) for the classification layer. The Final layer compares labels of the top probabilities from the classifier with actual labels of the available classes, usually using the Softmax function that provides probabilities for each possible output class, giving the model's accuracy [4, 30].

$$Out^i_{neuron} = \sum_{j=1}^{k_{input}} Input^i \times weight^{ij} + Bias^i. \tag{2.13}$$

2.3.2 LeNet-5

LeNet-5 is a CNN proposed by Yann LeCun in 1998 [12] for handwriting character recognition. The architecture of the LeNet-5 (Figure 2.14) consists of three convolutional layers, two sub-sampling layers, and two fully connected layers, making a total of seven layers.

In the learning phase, the neural network receives a set of N images with an associated label for each one. The categorical cross-entropy loss function, described later, computes the difference between the predicted label and the actual label. The input consists of 32x32 sized images, which correspond to the images from the MNIST [31] dataset with a padding of 2. The pixels are normalized from 0 to 255 in values between -0.1 and 1.175. The main reason is to ensure that the batch of images has a mean equal to 0 and a standard deviation equal to 1, which reduces the length of training time. The authors labels the convolutional layers as Cx, subsampling layers as Sx, and fully connected layers are labeled Fx, where x is the sequential position of the layer in the neural network.

The first convolutional layer, C1, outputs six feature maps and has 5x5 kernels. The dimensions of

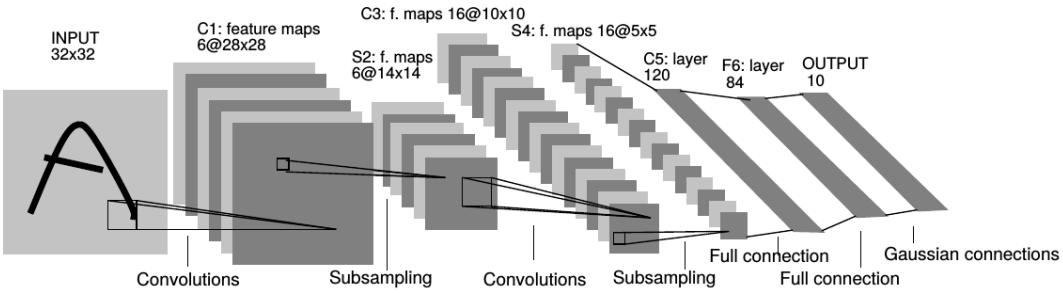

Figure 2.14: Architecture of LeNet-5, a Convolutional Neural Network for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical. Adapted from [12].

the six feature maps produced by the first convolution layer are 28x28.

Subsampling layer S2 halves the size of the feature maps it receives from the previous layer, reducing their resolution. It also produces six feature maps, each corresponding to the feature maps received as input from the last layer. The inputs inside the 2x2 pooling window are added, multiplied by a trainable coefficient, and added to a trainable bias. The result goes through a sigmoidal function. The pooling window moves through the pixels by a stride of 2, never overlapping the previous positions.

Layer C3 receives as input six feature maps of the last layer and outputs sixteen feature maps, in which the combinations are according to Figure 2.15. The author designed local connections to avoid increased computation, memory complexity, and symmetry between the learned features.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

Figure 2.15: Combinations of feature maps to obtain the C3 output. Each column indicates which feature map in S2 are combined by the units in a particular feature map of C3. Adapted from [12].

Subsampling layer S4 has sixteen feature maps of size 5x5 and has the same behavior as S2. Layer C5 is a convolutional layer with 120 feature maps. Inputs are convoluted with 5x5 kernels, which result in 1x1 feature maps.

The fully connected layer, F6, has 84 units and is fully connected to the C5. The last dense layer has ten units that correspond to the number of classes within the MNIST dataset. The activation function for the output layer is a Softmax activation function. The summarized structure of LeNet-5 can be seen on Table 2.1.

2.3.3 Dataset

The criterion of simplicity is the basis of our dataset choice. Since this work aims to have a convolutional neural network to classify images, a simple dataset was chosen, such as MNIST [31].

The MNIST database of handwritten digits, Figure 2.16, has a training set of 60,000 images and a

Layer Name	Feature Maps	Kernel	Feature Maps Size	Trainable Parameters	Connections	Activation Function
Input	-	-	32x32	-	-	-
C1	6	5x5	28x28	156	122304	Hyperbolic tangent
S2	6	2x2	14x14	12	5880	Sigmoid
C3	16	5x5	10x10	1516	151600	Hyperbolic tangent
S4	16	2x2	5x5	32	2000	Sigmoid
C5	120	5x5	1x1	48120	-	Hyperbolic tangent
F6	-	-	84	10164	-	Hyperbolic tangent
Output	-	-	10	-	-	Softmax

Table 2.1: LeNet-5 Structure. Adapted from [12].

test set of 10,000 images. It is a subset of a more extensive set available from NIST. The digits have been size-normalized and centered in a fixed-size image. NIST's original black and white images were sizes normalized to fit in a 20x20 pixel box while preserving their aspect ratio. The resulting images contain grey levels due to the anti-aliasing technique used by the normalization algorithm. The images are centered in a 28x28 image by computing the center of mass of the pixels and translating the image to position this point at the center of the 28x28 field.

Figure 2.16: Sample of the MNIST dataset, with ten different classes representing the numbers from 0 to 9 [31].

2.3.4 Tensorflow - CNN training

The CNN can be trained on Tensorflow [32] servers to obtain the weights and biases referring to each layer and, later, directly implementing the network with its predetermined values.

TensorFlow is a free and open-source software library for machine learning and artificial intelligence that runs on Google Colaboratory [33] that allows to write and execute Python in the web browser. Despite its usage in various tasks, it focuses on training and inferring deep neural networks. TensorFlow is a machine learning system that operates on a large scale and in heterogeneous environments. It uses data flow graphs to represent computation, shared state, and operations that change that state. TensorFlow allows developers to experiment with new optimizations and training algorithms. TensorFlow supports various applications, focusing on training and inference in deep neural networks [32, 34].

The hardware available in Tensorflow are Tensor Processing Unit (TPU)s [35], Intel(R) Xeon(R) CPU [36] operating at a frequency of 2.20GHz and Tesla K80 GPU [37] operating at a frequency of 562 MHz. A TPU is an AI accelerator Aplication-Specific Integrated Circuit (ASIC) developed by Google specifically for neural network machine learning, while GPUs have the ability to break complex problems into thousands or millions of separate tasks and work them out all at once. We will compare the training performance between TPU and GPU implementation.

The training process can be divided into the following steps:

- **Download of MNIST dataset and python libraries**

  The first thing to do is to download the dataset and the necessary Python libraries directly from Tensorflow.

- **Data pre-processing**

  After the dataset download, we must normalize the values of the pixels. Each pixel in the image is an integer belonging to the range [0,255] and must be normalized between [0,1] for the model to work correctly.

- **Model definition**

  In this step, we define the layers that are going to be a part of the model and then we define all the necessary parameters of each layer. Then, we need to define the loss function and the optimizer.

- **Loss Function**

  The loss function has the purpose of measuring the error between the predicted value in a CNN and its real value and is used to optimize it through backpropagation that aims to minimize the loss function by adjusting the trainable parameters of the model [38].

  For multiclass classification problems, as in this problem, Categorical Cross-Entropy (CCE) Loss, Sparse CCE Loss, and Kullback Leibler Divergence Loss can be used. The CCE and the Sparse CCE work the same way. However, the Sparse CCE is much faster. Kullback Leibler divergence is primarily used for more complex functions than simple multiclass classification.

  Both, CCE and Sparse CCE cross-entropy have the same loss function, defined in 2.14, where i is the number of rows, j is the number of categories, and y_{ij} is one hot-encoded target vector. The only difference is the way true labels are defined. For Sparse CCE, one needs only provide a single integer unit rather than an n-dimensional vector. Using Sparse CCE can save computation time with lower memory requirements. However, it can only be used when each input belongs to a single class only.

$$Loss = -\sum_{j=1}^{t} y_{ij} * log(\hat{y_{ij}}). \tag{2.14}$$

- **Optimizer**

  Optimization in machine learning is finding the best weights and biases to return the lowest loss. We use the Adam optimizer, a replacement optimization algorithm for stochastic gradient descent for training deep learning models based on adaptive estimates of lower-order moments [39].

- **Batch size**

  The neural network model training must use small batch sizes, offering good training stability and lower generalization error. It is easier to fit one small batch in memory. The batch size should be a power of two to take full advantage of the processor. Several studies suggest that a batch size equal to 32 or smaller achieves the best results [40, 41].

- **CNN training**

  In this step, we define the number of epochs to train the neural network. After this step, it is possible to obtain the trainable parameters and the loss and accuracy values of the training phase.

2.3.5 Fixed-Point Representation

In neural networks, calculations are traditionally performed with floating-point, either on GPU or CPU. Depending on the hardware resources, floating-point representation can slow down computations due to the need to process many fractional bits at some point in the algorithm. An algorithm with greater precision should use double-precision floating-point data or single-precision if low resource utilization is essential. However, a large amount of processing power is required to perform floating-point calculations. Fixed-point representation intends to decrease energy consumption by using a fixed amount of fractional bits. It can also increase the speed of the operations. However, it is subject to a loss of precision of the number representation. Described in more detail in [42].

Fixed-point can be represented by Q[QI].[QF], where QI is the number of bits of the integer part and QF is the number of bits of the fractional part, the sum of QI and QF is called word length, WL. For example, a Q3.5 number would be an 8-bit value with three integer bits and five fractional bits. A sign bit will be included in QI for signed integer variables. Using fixed-point representation limits the ratio of the maximum absolute value representable and the minimum positive absolute value representable, known as dynamic range [43]. The dynamic range can be calculated using Equation 2.15 for a signed fixed-point rational representation Q[QI].[QF], and using Equation 2.16 for an unsigned fixed-point rational representation Q[QI].[QF]. For WL of any significant length, the "-1" is negligible, and therefore signed and unsigned representations of the same word length have roughly the same dynamic range.

$$2 \times (2^{QI}/2^{-QF}) = 2^{QI+QF+1} = 2^{WL}. \tag{2.15}$$

$$(2^{QI} - 2^{-QF})/2^{-QF} = 2^{QI-QF} - 1 = 2^{WL} - 1. \tag{2.16}$$

In our case, we intend to obtain the values of the trainable parameters of the CNN in Tensorflow to store them in the FPGA memories. We can use these values and the pixels values of the images, between 0 and 1, to calculate QI using Equation 2.17, where α is the floating-point variable to represent in fixed-point. QI will always be greater than zero since the logarithmic function (log_2) can never be negative. As for QF, it is possible to choose the number of bits of the fractional part through the difference between the pretended word length and QI.

$$QI = ceiling(log_2(max(abs[\alpha_{max}, \alpha_{min}]) + 1)) + 1. \tag{2.17}$$

To analyze the addition and multiplication operations, consider two signed fixed-point 8-bit numbers, A and B, where A is represented in Q1.7 format and B in Q2.6, Figure 2.17.

It is necessary to give special attention to the alignment of the fractional part and the possibility of an overflow to perform addition. This is accomplished by right shifting and sign extending A, which is the number that has a shorter integer part (Figure 2.17), aligning the fractional part, then the numbers are added, and the carry bit (c) is verified to see if there is an overflow.

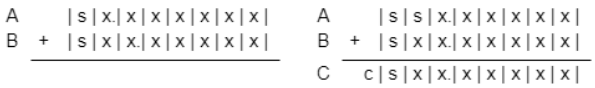

Figure 2.17: Signed fixed-point addition.

When performing fixed-point multiplication, the number of integers and fractional bits in the product is the sum of the corresponding multiplier and Q points of the corresponding multiplicand, as described in Equations 2.18 and 2.19.

$$QI_{product} = QI_{multiplicand} + QI_{multiplier}, \tag{2.18}$$

$$QF_{product} = QF_{multiplicand} + QF_{multiplier}. \tag{2.19}$$

When multiplying a Q1.7 and a Q2.6 number, the result is a 16-bit Q3.13 number. The 16-bit Q3.13 number can be reduced to an 8-bit representation. The result will have to maintain the 3 bits of the integer part, though it is possible to remove 8 bits of the fractional part in Q3.5 format. This removal leads to a significant loss of precision. The resulting 8-bit Q3.5 number within the 16-bit result is in Figure 2.18.

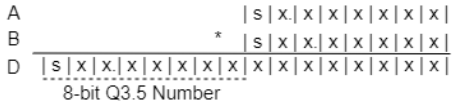

Figure 2.18: Signed fixed-point multiplication.

Chapter 3. Exploratory implementation of Convolutional Neural Networks

This chapter describes the implementation of our CNN algorithm divided into two phases. The first phase consists of training the CNN. This provides the parameters to implement the classifier, and also provides baseline results for later comparison. The second phase involves implementing the CNN on reconfigurable logic and subsequent image classification. TensorFlow 2 [32], Matlab R2021a [44], Quartus Prime Software (QPS) 18.1 [16] and ModelSim 10.5b [17] tools were used.

3.1 Implementation Workflow

The implementation is divided into the following aspects:

1. Tensorflow – Python:
    - CNN training;
    - Obtaining trainable parameters;
    - Comparison between our Modified LeNet-5 and the original LeNet-5.
2. Matlab:
    - Representation of images and trainable parameters with 8-bit fixed-point resolution, where 3-bit corresponds to the integer part and 5-bit to the fractional part;
    - Writing image values and weights into memory initialization files (MIF).
3. Quartus Prime Software – VHDL:
    - Definition of required blocks per layer, control signals and process timing;
    - RTL synthesis and test on hardware.

Figure 3.1 shows the data workflow of our implementation.

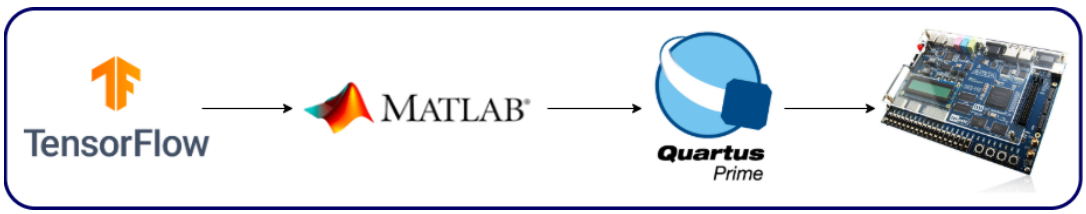

Figure 3.1: Workflow of our implementation.

3.1.1 Tensorflow

On Tensorflow, we implemented the Original LeNet-5 and our modified version of LeNet-5, adapted for SC. There are different hardware hypotheses for training CNNs, and we trained the networks on TPU and GPU. After the implementation, we studied the temporal, loss, and accuracy differences between the neural network train using a TPU or a GPU, described in 4.1.2. However, for implementing the classifier on FPGA, we only need to obtain the trainable parameters. The method we use to obtain them is not a key point, although there are timing differences between the two types of hardware. In this section, we describe the CNNs training.

3.1.1.1 Original LeNet-5 and Modified LeNet-5

We started by implementing LeNet-5, following 2.14, with two convolutional layers, two average-pooling layers, a flattening layer, and three dense layers. The number of trainable parameters is computed differently in each layer. In a convolutional layer, the number of parameters is calculated using Equation 3.1, and in a dense layer is calculated using Equation 3.2. In a max-pooling or a flattening layer, there are no parameters.

$$N_{Parameters_{conv2D}} = Filter_{size}(5 \times 5) \times Input_{depth} \times Filter_{quantity} + Bias. \tag{3.1}$$

$$N_{Parameters_{Dense}} = Input_{Dimension} \times Output_{Dimension} + Bias. \tag{3.2}$$

Figure 3.2 shows the summary of the layers and parameters of the LeNet-5 model. We avoided padding the input images in the original LeNet-5 to have the same input and output sizes as our modified model. In layer C3 (2.3.2), we do not limit the local connections since the author only did this to save computational effort due to the hardware limitations at that time. With current technology, it makes no sense to limit connections, as the resulting computational difference would not be relevant.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 6)         156

 average_pooling2d (AverageP  (None, 14, 14, 6)        0
 ooling2D)

 conv2d_1 (Conv2D)           (None, 10, 10, 16)        2416

 average_pooling2d_1 (Averag  (None, 5, 5, 16)         0
 ePooling2D)

 flatten (Flatten)           (None, 400)               0

 dense (Dense)               (None, 120)               48120

 dense_1 (Dense)             (None, 84)                10164

 dense_2 (Dense)             (None, 10)                850

=================================================================
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
_________________________________________________________________
```

Figure 3.2: Summary of the LeNet-5 model.

We implemented our Modified LeNet-5 with two convolutional layers, two max-pooling layers, a flattening layer, and a dense layer. Figure 3.3 shows the number of parameters calculated and the different layers in the network.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 24, 24, 4)         104

 max_pooling2d (MaxPooling2D  (None, 12, 12, 4)        0
 )

 conv2d_1 (Conv2D)           (None, 8, 8, 4)           404

 max_pooling2d_1 (MaxPooling  (None, 4, 4, 4)          0
 2D)

 flatten (Flatten)           (None, 64)                0

 dense (Dense)               (None, 10)                650

=================================================================
Total params: 1,158
Trainable params: 1,158
Non-trainable params: 0
_________________________________________________________________
```

Figure 3.3: Summary of the Modified LeNet-5.

The Modified LeNet-5, Figure 3.4, starts with one input image (28, 28), which is convoluted with four filters (5, 5) generating four feature maps (24, 24). These four feature maps go through a ReLU activation function, which outputs the pixel value if positive and 0 if negative. It then goes through a max-pooling layer that reduces the values of the feature maps to a quarter (12, 12), thus, reducing memory usage.

The four feature maps (12, 12) go through a second convolution layer with four filters (5, 5), generating four feature maps for each sub-image, in a total of sixteen feature maps (8, 8). The sixteen feature maps go through the ReLU activation function again and through another layer of max pooling, resulting

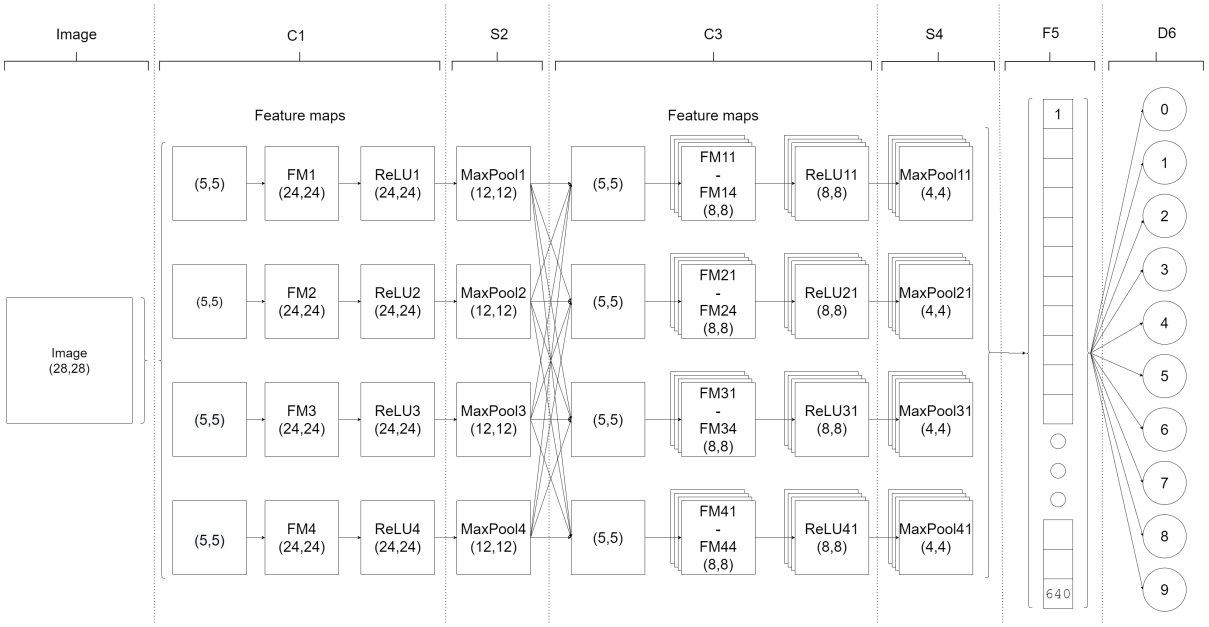

Figure 3.4: Modified LeNet-5 layout.

in sixteen feature maps (4, 4), corresponding to 256 pixels for one image. Then the values are converted into a one-dimensional array, which will be the input of a dense layer, before classifying the image.

In the dense layer, the result for each output neuron, d_{n_i}, i being the neuron number, is calculated using the Equation 3.3, where p_j corresponds to the layer's input pixel, j is the pixel number, wij is the corresponding weight to each pixel and b_i is the bias corresponding to each neuron, with i varying from 0 to 9 and j varying from 1 to 64.

$$d_{n_i} = (\sum_{j=1}^{64} p_j \times w_{ij}) + b_i. \tag{3.3}$$

At this point, we have everything we need to move on to training the two LeNet-5 models on TensorFlow.

3.1.2 Training

As described in 2.3.4, the CNN training is performed on Tensorflow to obtain the weights and bias for each layer to be later implemented on FPGA. We use the 60000 training images to obtain possible weights and bias values.

In Table 3.1 is possible to see the inputs and outputs of each layer. The neural network training process starts with an input, which is an image (28x28) pixels that are convolved with four filters (5x5), generating four (24x24) feature maps. These images go through the ReLU activation function. It then goes through a max-pooling layer to reduce feature maps to (12x12) pixels. After the pooling layer, the output goes through a new convolution layer with four filters (3x3) that generate new four feature maps for each feature map of the previous layer, going again through the ReLU activation function and a max-pooling layer, having as output 18 feature maps of (5x5) pixels.

	Input	Output
C1	Image (28x28)	4 Feature maps (24x24)
S2	4 Feature maps (24x24)	4 Feature maps (12x12)
C3	4 Feature maps (12x12)	16 Feature maps (5x5)
S4	16 Feature maps (5x5)	16 Feature maps (4x4)
F5	16 Feature maps (4x4)	Vector (1x640)
D6	Vector (1x640)	10 classes

Table 3.1: Inputs and Outputs by layers.

In the next layer, the pixel values are flattened into a single vector, which will be the input of the dense layer and then multiplied, one by one, by the weights obtained and added to the bias to reach a final value. This value corresponds to the input of the dense layer, which will go through a softmax function to classify the image into one of the ten possible classes. We summarize the Tensorflow algorithm in Algorithm 1.

Algorithm 1: Neural network algorithm in Tensorflow.

Import:

- Libraries, Dataset, Google Drive

Normalize the pixels with values between [0,255] into [0,1]:

- $train\_dataset = train\_dataset.map(normalize)$
- $test\_dataset = test\_dataset.map(normalize)$

Model creation:

- $model = tf.keras.Sequential(layers)$
- $model.compile(optimizer, loss)$

Training:

- $model.fit(train\_dataset, epochs, steps\_per\_epoch)$

Save weights:

- $model.save\_weights(path)$

Evaluate model:

- $model.evaluate(test\_dataset)$

At this point, we have obtained the weights and biases we will use to implement the Modified LeNet-5 on FPGA.

3.2 Weight pre-processing with Matlab

After obtaining the weights in Tensorflow, we use Matlab to sort them properly to use on FPGA. Each layer has a different format from the others, and it is essential to order their weights individually. There are three layers (C1, C3, and D6) where we obtain the weights. The index corresponding orders the weights to each filter in the convolution layers. The first four weights correspond to the first value of each filter and so on. We intended to sort the weights by filters, starting with the first value of the first filter, then with the second value of the same filter, and so on.

In layer C1, weights are in the format {5, 5, 1, 4}, with a total of 5*5*1*4=100 weights, where {5, 5} corresponds to kernel size, {1} to the number of channels, and {4} to the number of filters. It is necessary to save the values starting with the first of the first filter and skip to the fourth value after that, repeating until reaching the last value of that filter. In the following filters, the process is repeated, considering the first value of each filter. In Figure 3.5 on the left, it is possible to see how we ordered the weights in Tensorflow and our pretended order on the right. We refer to the kernels as $k_{i,j}$, where i is the kernel number, and j is the kernel element.

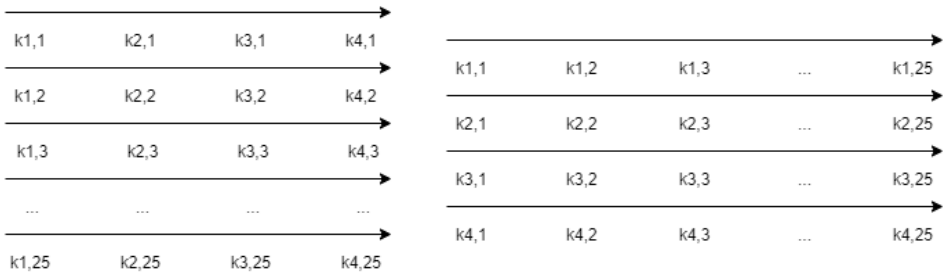

Figure 3.5: Order obtained in Tensorflow for C1 on the left. Pretended order on the right.

In C3, the weights are in the format {5, 5, 4, 4}, with a total of 5*5*4*4=400 weights. The difference between this layer and the previous one is that there are four channels instead of one, meaning there are four filters per channel. The process is similar to the last layer, differing in the number of skipped values to obtain the desired order, with sixteen being jumped instead of four (Figure 3.6).

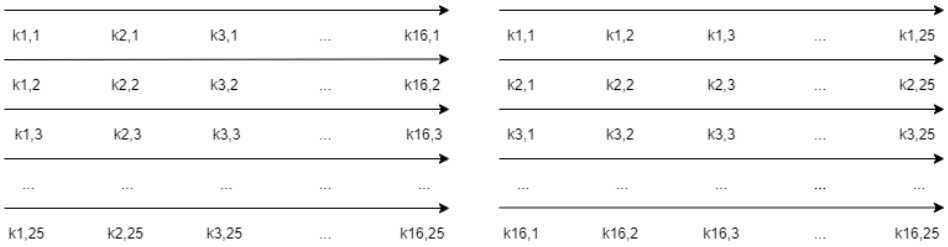

Figure 3.6: Order obtained in Tensorflow for C3 on the left. Pretended order on the right.

Regarding layer D6, its weights are in the format {64, 10}, being 640 weights (64*10). In this layer, sorting the weights is unnecessary since they are already sorted by the weights relative to each neuron. Each neuron uses 64 weights to get the final value (Figure 3.7).

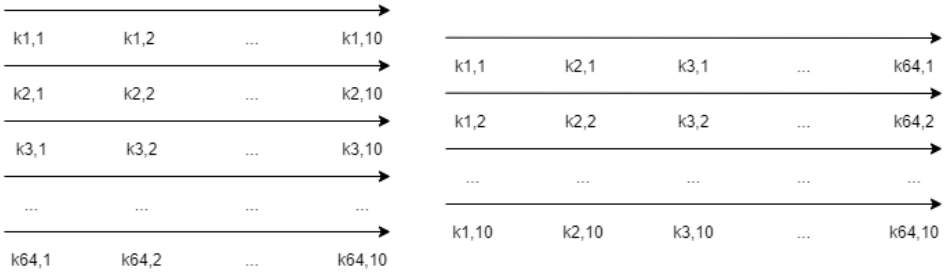

Figure 3.7: Order obtained in Tensorflow for D6 on the left. Pretended order on the right.

After the weight sorting, we converted from decimal to signed fixed-point 8 bits using the 'deq2q' function, which converts decimal (base 10) numbers to fixed-point Qa.b format. Where "a" is the number of integer bits (not including the sign bit), and "b" is the number of fractional bits. The output format is either binary or hexadecimal. We call the function using "dec2q(x, a, b, format)". Where x is the decimal input (which can either be a scalar or a vector), "a" is the number of bits to the left of the binary point not including the sign bit, and "b" is the number of bits to the right of the decimal point. The format is either 'bin' or 'hex' [45].

After converting to fixed-point, we write the weights in three MIFs (Memory Initialization File), each corresponding to each layer's weights. We summarize the Matlab script in Algorithm 2.

3.2.1 Fixed-point format

We decided to use an 8bit representation in the CNN inputs. However, due to a large number of additions and multiplications, it is necessary to use more bit representations as we advance through the CNN, which led us to use a dynamic format to avoid losing information. Considering the worst-case scenario, it is necessary to know the absolute maximum value of the input $|in_{max}, in_{min}|$, the kernel size, the weights $|w_{max}, w_{min}|$, and the biases $|b_{max}, b_{min}|$ to calculate the theoretical maximum value of each convolutional layer $layer_{max}$, using Equation 3.4.

The maximum value from the dense layer can be calculated using Equation 3.5, similar to Equation 3.4, where the $kernel_{size}$ is replaced by the number of neurons, $nr_{neurons}$. The values obtained by using Equations 3.4 and 3.5 are shown in Table 3.2.

$$Layer_{max_{conv}} = (|in_{max}, in_{min}| * |w_{max}, w_{min}|) * kernel_{size} + |b_{max}, b_{min}|. \tag{3.4}$$

$$Layer_{max_{dense}} = (|in_{max}, in_{min}| * |w_{max}, w_{min}|) * nr_{neurons} + |b_{max}, b_{min}|. \tag{3.5}$$

The worst-case scenario uses the highest possible absolute value of the input, the weights, and the biases values. In convolutional layers, the multiplication between the pixel and the weight value is added 25 times to the bias value. In C1, the maximum input value is equal to 1, and in C2, it is similar to the absolute maximum of the first layer. Regarding the dense layer, the highest input value is equal to the absolute maximum of the second convolutional layer, then multiplied by the highest value of weights of this layer 256 times and added to the bias value. By analyzing Table 3.2, we can save power by using an adaptive fixed-point representation, using a fewer bit representation at earlier layers. Layer 1 requires a

Algorithm 2: Weight sorting and fixed-point representation

Input : $c1\_w(25,4)$, $c3\_w(100,4)$, $d6\_w(64,10)$
Output: $weightsc1.mif$, $weightsc3.mif$, $weightsd6.mif$
Separate and sort the weights by filter for C1:

- $f_{n,C1} = [c1\_w(1,n), c1\_w(2,n), c1\_w(3,n), ..., c1\_w(25,n)]', n\epsilon[1,4]$
- $c1\_w\_ordered = [f_{1,C1}, f_{2,C1}, f_{3,C1}, f_{4,C1}]$

Separate and sort the weights by filter for C3:

- $f_{1n,C3} = [c3\_w(1,n), c3\_w(5,n), c3\_w(9,n), ..., c3\_w(97,n)]', n\epsilon[1,4]$
- $f_{2n,C3} = [c3\_w(2,n), c3\_w(6,n), c3\_w(10,n), ..., c3\_w(98,n)]', n\epsilon[1,4]$
- $f_{3n,C3} = [c3\_w(3,n), c3\_w(7,n), c3\_w(11,n), ..., c3\_w(997,n)]', n\epsilon[1,4]$
- $f_{4n,C3} = [c3\_w(4,n), c3\_w(8,n), c3\_w(12,n), ..., c3\_w(100,n)]', n\epsilon[1,4]$
- $c3\_w\_ordered = [f_{11,C3}, f_{12,C3}, f_{13,C3}, f_{14,C3}, f_{21,C3}, f_{22,C3}, f_{23,C3}, f_{24,C3}, f_{31,C3}, f_{32,C3}, f_{33,C3}, f_{34,C3}, f_{41,C3}, f_{42,C3}, f_{43,C3}, f_{44,C3}]$

Sort the weights by neuron for D6:

- $d6\_w\_ordered = d6\_w(:)$

Conversion to fixed-point representation:

- $c1\_fp = dec2q(c1\_w\_ordered(:), 2, 5, 'bin')$
- $c3\_fp = dec2q(c3\_w\_ordered(:), 2, 5, 'bin')$
- $d6\_fp = dec2q(d6\_w\_ordered(:), 2, 5, 'bin')$

Write MIF file for C1:
$fid = fopen('weightsc1.mif', 'w+')$
$count = 0$
if fid **then**
  $fprintf(width = size(c1\_fp, 2), depth = size(c1\_fp, 1), addressradix = Hex, dataradix = Bin, 'CONTENTBEGIN')$
  for $i = 1 : size(c1\_fp, 1)$ **do**
    $fprint(count)$
    for $j = 1 : size(c1\_fp, 2)$ **do**
      $fwrite(fid, c1\_fp(i,j)$
    end
    $count++$
    $fprint(';')$
  end
  $fprintf('END;')$
  $fclose$
end
Repeat write MIF file for C3 and D6.

fixed-point output format of Q8.5, layer 3 requires a Q11.5 format, and layer requires a Q19.5 format.

From this point, we now have all the CNN parameters in ROM Memory Initialization Files (MIF) ready for the Register Transfer Level (RTL) circuit implementation.

Layer	Parameter	Value	Fixed-Point Format
Layer 1 Convolutional	Max	21.31444	Q6.5
	Min	-42.5570	Q8.5
Layer 3 Convolutional	Max	413.9171	Q10.5
	Min	-663.9777	Q11.5
Layer 6 Dense	Max	91001.5141	Q18.5
	Min	-160532.5760	Q19.5

Table 3.2: Maximum and minimum per layer and the corresponding fixed-point format, considering the worst-case scenario.

3.3 Register Transfer Level (RTL) Project

This section describes the Quartus Prime Software (QPS) implementation of each layer individually. The description corresponds only to the processing of an image. All layers have a checkpoint before moving to the next layer by writing the results to RAM. The addresses are in hexadecimal format. To implement our SC CNN, we started by implementing the network fully sequential, without stochastic elements, with the purpose of inserting them in future work. The objective is to gradually go from a network without any stochastic computing components to a network with all its modules in stochastic computing. This section describes the implementation of the CNN without SC modules.

3.3.1 Layer C1 - Convolutional Layer

We designed a controller in VHDL to generate memory addresses to access both the image and weight memories. The image memory is where the pixel values are stored, 784 pixels for each image (from address 0x000 to 0x30F), and the weight memory (from address 0x00 to 0x63) is where the weights are stored. Since there is only four bias in this layer, they are written manually on the controller.

When generating the addresses, the controller generates five addresses from 0x000 to 0x004, then jumps twenty-four, generating five consecutive addresses. This procedure repeats until there are twenty-five kernel elements. After that, it repeats the same process starting an address after the first previous address until it reaches the final element of the line (0x01B). It then jumps to the first address of the second row of pixels (0x01C) and repeats until it has gone through the entire image.

Generating weight addresses is simpler than image addresses, as twenty-five addresses are generated sequentially per kernel and repeated until the end of the kernel. As soon as a kernel changes, the controller must jump to the address where the new kernel starts. Kernels start at addresses 0x00, 0x19, 0x32 and 0x4B. Algorithm 3 describes the implementation of the address generation for C1.

After choosing the correct addresses, the values will enter a block with two MACs, multiplying the pixels by their respective weight. The use of two MACs serves to alternate each convolution, as it makes it simpler to return the value of each convolution in an orderly way. Two controllers do this alternation where one selects which MAC to use and the other selects which should be the output. The schematic of the MAC-based convolution is in Figure 3.8. The block receives another input, 'k_elm', beside the values to multiply. This input is an index indicating the multiplication index, where the range of the index goes from 1 to 25. We use a demultiplexer to select the MAC and a multiplexer to select the

Algorithm 3: Address generation for C1

$kernel\_count = 0,\ line\_count = 0, conv\_total = 0, addr\_img = 0,\ addr\_w = 0$
if $kernel\_count < 24$ **then**
  $kernel\_count++$
  $addr\_w++$
  if $line\_count < 4$ **then**
    $line\_count++$
    $addr\_img++$
  else if $line\_count = 4$ **then**
    $line\_count = 0$
    $addr\_img += 24$
  end
else if $kernel\_count = 24$ **then**
  $kernel\_count = 0$
  $addr\_w = start\ address\ of\ the\ filter\ to\ be\ used$
  if $conv\_total < 576$ **then**
    $conv\_total++$
    $addr\_img = start\ address\ of\ the\ new\ kernel$
  else if $conv\_total = 576$ **then**
    $conv\_total = 0$
    $addr\_img = start\ address\ of\ the\ next\ image$
  end
end

output to use. After these operations, the bias value referring to the kernel used is added, and then it goes through the activation function ReLU.

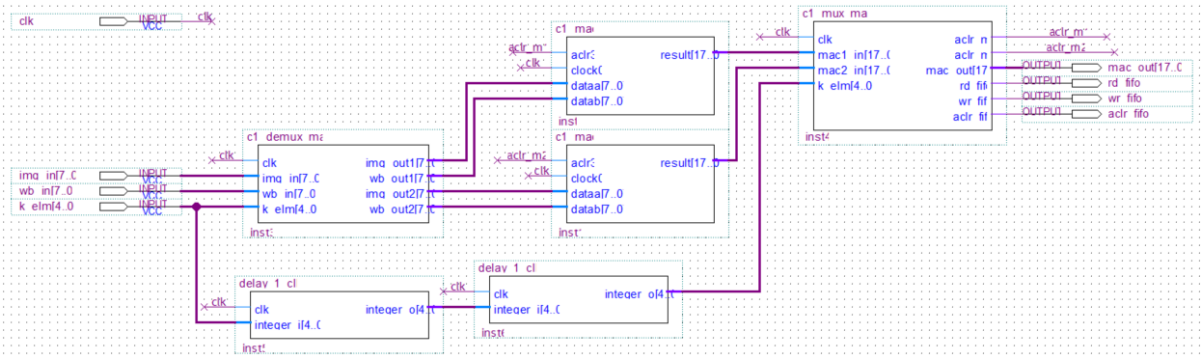

Figure 3.8: MAC-based convolution.

The output of the ReLU block is written in a FIFO memory to ensure the sequential writing of the outputs in RAM, which will be the input of the next layer. This layer outputs four feature maps with 2304 pixels (576 each). Figure 3.9 shows the data flow in C1, where 'C1_address_generator' and 'RAM_C1_controller' sends addresses to other blocks. All the other blocks send data to the following block.

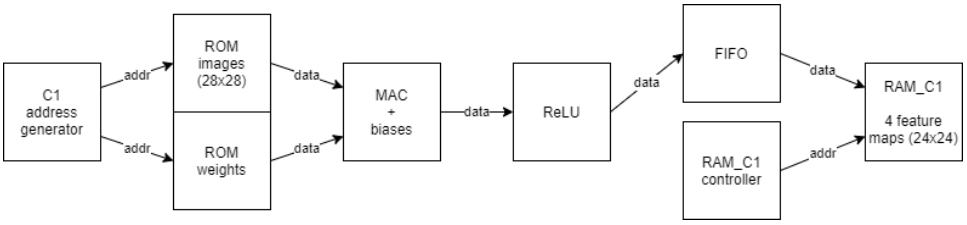

Figure 3.9: Data flow in layer C1.

3.3.2 Layer S2 - Max Pool Layer

Layer 2 starts with an address generator, which generates the first two addresses of two consecutive lines (0x00, 0x01, 0x18, 0x19). It then generates the following two addresses for each line, repeating until the end of the lines. When it reaches the end of a feature map, it restarts at the first addresses of the following two lines (0x30, 0x31, 0x48, 0x49). The process is repeated until it reaches the end of the four feature maps. The corresponding pixels enter a Maxpool block with another input to know where to store the pixel values until there are four, then compute the maximum between them and write the result in a FIFO memory and later in a RAM, that will be the input of the next layer. The layer output is four feature maps with 576 pixels (144 each). Figure 3.10 shows the data flow in S2, where 'S2_address_generator' and 'RAM_S2_controller' sends addresses to other blocks. All the other blocks send data to the following block.

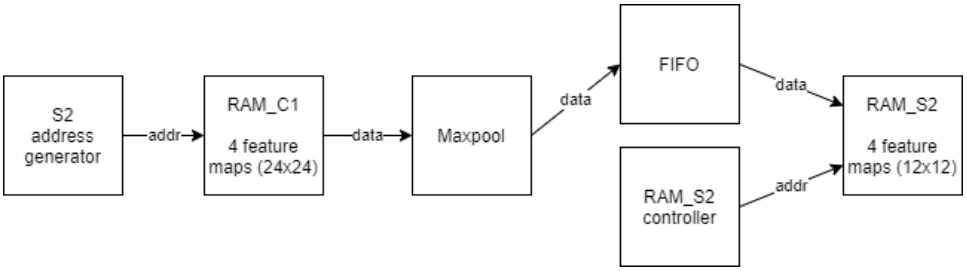

Figure 3.10: Data flow in layer S2.

3.3.3 Layer C3 - Convolutional Layer

In this layer, we generate five addresses in a row starting from 0x000, then skips eight and generate five from that, repeating until there are 25. Then a convolution with the first filter occurs and then returns to the first address and repeats with the following filters. When the convolution finishes with the four filters of the first channel, it switches to the second channel and repeats until the fourth filter of the fourth channel is complete. The first channel starts at 0x000, the second at 0x090, the third at 0x120, and the fourth at 0x1B0. Algorithm 4 describes the implementation.

Figure 3.11 shows the data flow in C3, where 'C3_address_generator' and 'RAM_C3_controller' sends addresses to other blocks. All the other blocks send data to the following block.

Algorithm 4: Address generation for C3

$kernel\_count = 0$, $line\_count = 0$, $conv\_total = 0$, $addr\_img = 0$, $addr\_w = 0$
if $kernel\_count < 24$ **then**
  $kernel\_count++$, $addr\_w++$
  if $line\_count < 4$ **then**
    $line\_count++$, $addr\_img++$
  else if $line\_count = 4$ **then**
    $line\_count = 0$, $addr\_img += 8$
  end
else if $kernel\_count = 24$ **then**
  $kernel\_count = 0$
  $addr\_w = start\ address\ of\ the\ filter\ to\ be\ used$
  if $conv\_total < 64$ **then**
    $conv\_total++$
    $addr\_img = start\ address\ of\ the\ new\ kernel$
  else if $conv\_total = 64$ **then**
    $conv\_total = 0$
    $addr\_img = start\ address\ of\ the\ next\ image$
  end
end

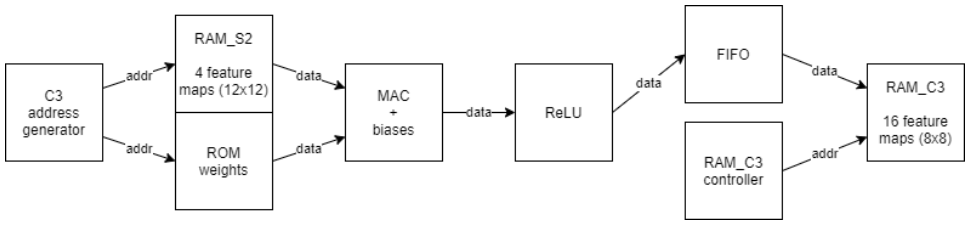

Figure 3.11: Data flow in layer C3.

3.3.4 Layer S4 and F5 - Max Pool and Flattening Layers

This layer has the same behavior as layer 2, differing in the number of inputs and outputs. It takes the 16 feature maps of the previous layer, with 1024 pixels in total (64 each), and returns 16 feature maps with 256 pixels (16 each). The address generator generates the first two addresses of two consecutive lines (0x00, 0x01, 0x08, 0x09). It then generates the following two addresses for each line, repeating until the end of the lines. When it reaches the end of a feature map, it restarts at the first addresses of the posterior two lines (0x10, 0x11, 0x18, 0x19). The pixel values go through a MaxPool block, a FIFO memory, and RAM.

The output of this layer is already on the output format of the Flattening Layer, making it unnecessary to apply changes to layer five. Figure 3.12 shows the data flow in S4 and F5, where 'S4_address_generator' and 'RAM_S4_F5_controller' sends addresses to other blocks. All the other blocks send data to the following block.

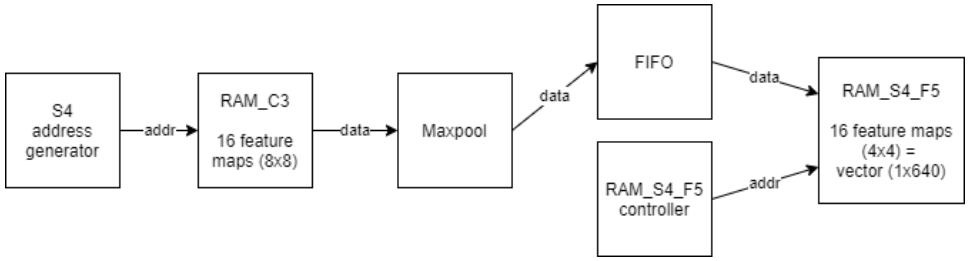

Figure 3.12: Data flow in layer S4.

3.3.5 Layer D6 - Dense Layer

There are four channels of feature maps in this layer, each with 64 values. There are also 640 weights, 64 corresponding to each of the ten neurons. We implemented two counters (counter64 from 0 to 63 and counter256 from 0 to 255) to control the address generation of the memory containing the weights and the output from the previous layer, respectively. The operations are performed per neuron, meaning that all the four channels' values multiply the weights corresponding to each neuron. The feature map addresses are traversed from 0 to 255 and restart when counter256 reaches 255. The weight addresses are traversed from 0 to 63 and start again when counter64 reaches 63. If it reaches 63 and counter256 to 255, it should start at the address of the next neuron, repeating the process until reaching the last neuron. Neurons 0 to 9 correspond to addresses 0x000, 0x040, 0x080, 0x0C0, 0x100, 0x140, 0x180, 0x1C0, 0x200 and 0x240, respectively. Each bias corresponds to each neuron. Algorithm 5 describes the implementation.

Algorithm 5: Address generation for D6

$count64 = 0$, $count256 = 0$, $addr\_img = 0$, $addr\_w = 0$
if $count64 < 63$ **then**
  $count64++$, $addr\_w++$
else if $count64 = 63$ **then**
  $count64 = 0$
  $addr\_w = start\ address\ of\ the\ neuron\ to\ be\ used$
end
if $count256 < 255$ **then**
  $count256++$, $addr\_img++$
else if $count256 = 255$ **then**
  $count64 = 0$, $addr\_w = 0$
end

The data flow in D6 is in Figure 3.13, where 'D6_address_generator' and 'RAM_D6_controller' sends addresses to other blocks. All the other blocks send data to the following block.

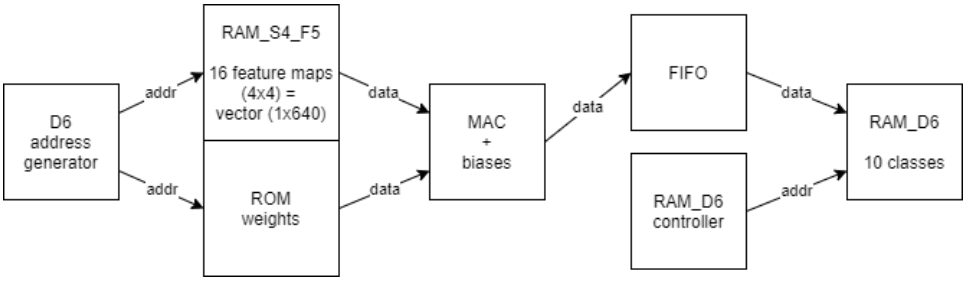

Figure 3.13: Data flow in layer D6.

3.4 Stochastic Computing approach circuit

To familiarize ourselves with the numerical conversion processes for the domain of Stochastic Computing, we studied the approximation of a stochastic number to its fractional value. To verify this, we designed a small circuit consisting of conversion from binary to stochastic, one AND gate performing a multiplication between two numbers, and conversion from stochastic to binary. The binary to stochastic conversion consists of two LFSR and two comparators. The conversion from stochastic to binary consists of two up counters, one for zeros and one for ones.

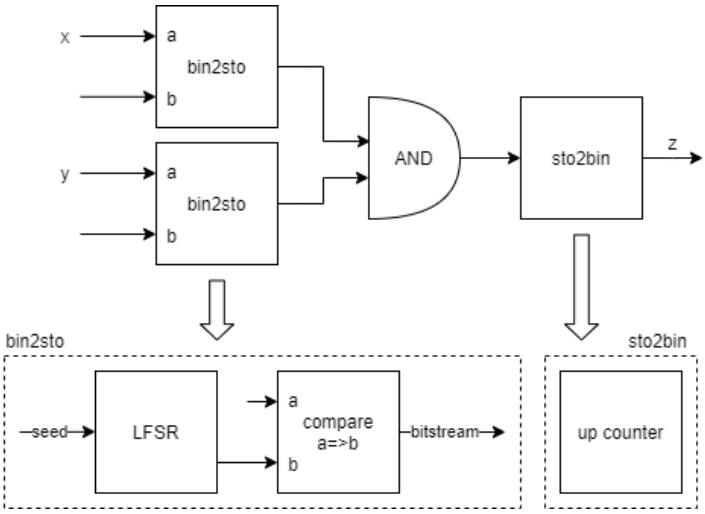

Figure 3.14: Stochastic computing circuit to multiply two numbers.

The circuit is shown in Figure 3.14. To avoid correlation between the seeds, the LFSRs are fed with different initial seeds, ’00111011’ and ’10001011’ (59 and 139 in decimal, respectively). The LFSR outputs connect to a comparator, which compares its value with two numbers, x and y, respectively. The comparators generate stochastic streams of bits, which will serve as input to the AND gate, multiplying them and generating a single stream. This stream is converted back to binary through counters.

Chapter 4. Experimental Results

This chapter presents our results relative to the previously described implementation. We present a performance comparison between TPU and GPU implementations on Tensorflow, as well as a comparison between LeNet-5 and Modified LeNet-5. Then the RTL implementation results of the Modified LeNet-5 and the Stochastic Computing circuit. Despite having a full implementation of our Modified LeNet-5, we verified that the network is not working as intended due to memory access errors.

4.1 Training on Tensorflow

Training the CNNs on Tensorflow allowed us to obtain comparative results between the training on TPU and GPU concerning Modified LeNet-5 and LeNet-5. We also analyze the classification phase results of the two CNNs on Tensorflow.

4.1.1 TPU vs. GPU

In an initial phase, we decided to verify the relevance of the number of training epochs of the network. We evaluated this by analyzing the training's loss value and accuracy. We analyze five cases for each implementation, with the number of epochs equal to 1, 5, 10, 20, and 50. In Figure 4.1, we can verify that, both in GPU and TPU implementations, we can obtain a relevant efficiency with few epochs, in which the most significant variation occurs between one and five epochs, varying 2.11% and 3.45% in GPU and TPU implementation, respectively. This variation occurs because the networks did not train enough epochs to adjust their parameters efficiently. The loss value varies most between one and five epochs in both implementations. In the case of TPU, the loss value differs 0.1215 between one and five epochs and 0.0326 between five and fifty epochs. In the case of GPU, it varies by 0.0806 between one and five epochs and 0.0279 between five and fifty epochs. For more details, see Table A.1 in Appendix A.

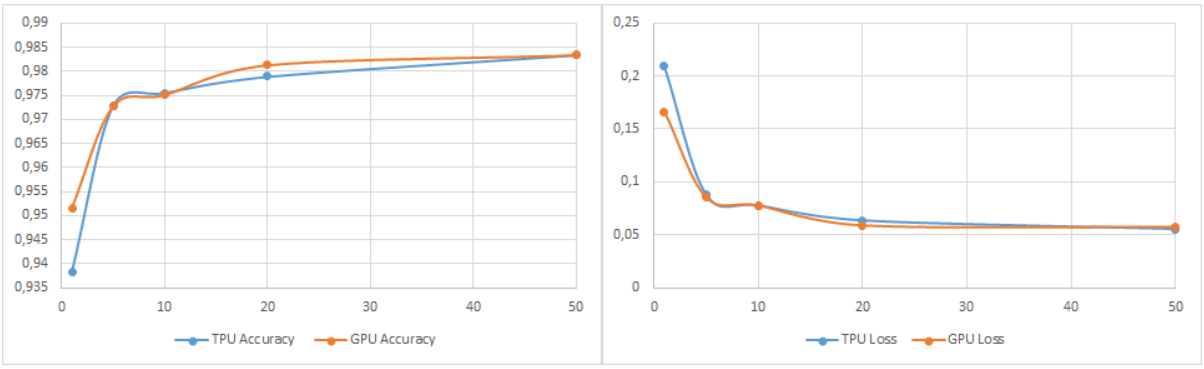

Figure 4.1: Accuracy and loss variance in function of the number of epochs.

With our study, we can verify that by training the CNN with a low number of epochs, the network will still have very high accuracy, both on TPU and GPU, mainly due to the great simplicity of the

dataset, which makes the algorithm capable of classifying images very quickly. We can also verify that the loss value tends to close to zero with few epochs as they tend to be the best possible value, validating the obtained trainable parameters.

When executing the code on Tensorflow and not locally, we can not control the hardware used, which may not be the same in each code execution. In order to understand if there are significant differences between the execution times, the loss values, and the accuracy obtained in different executions, we decided to test two scenarios. In the first one, we manually changed the number of epochs each time we ran the code, starting with one epoch and incrementing one unit each time, up to 20 epochs. We ran the code only once in the second scenario and checked the values corresponding to each epoch from 1 to 20. We tested both scenarios on TPU and GPU. Figure 4.2 shows that the three parameters tend to significantly close values. The trend is more linear in the second scenario since we trained the CNN twenty epochs once.

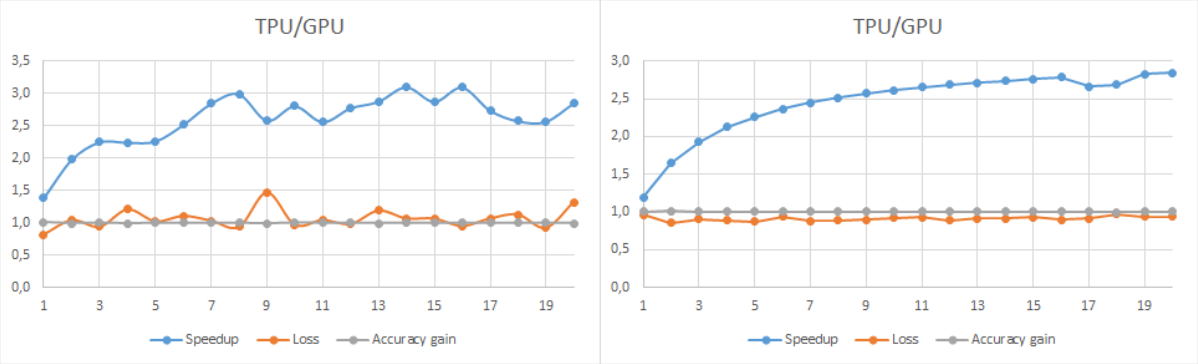

Figure 4.2: Comparison of speedup, loss, and accuracy between TPU and GPU implementation of the neural network. First scenario on the left and second scenario on the right. For more details, see Tables A.2 and A.3 in Appendix A.

We found that the processing of networks in GPU is faster than in TPU, despite the TPUs being ASICs explicitly designed for neural networks. One factor that explains this result is the optimization of this hardware for complex neural network models with a large number of parameters, which is precisely the opposite of the Modified LeNet-5 model.

4.1.2 Modified LeNet-5 vs. LeNet-5

We tested each network on GPU and TPU. The tests were done for five epochs to avoid overfitting. Firstly, we can observe in Table 4.1 the values of loss and accuracy of the training of the two models, as well as the time they take to be executed. As for the loss value, we consider that the difference between our network and the LeNet-5 is not problematic since values closer to zero would cause overfitting, indicating that the Modified LeNet-5 would take more epochs than the LeNet-5 until we have overfitting problems. We obtained a training accuracy equal to 96.79% on GPU and 96.68% on TPU, which are excellent values and are close to the values obtained in LeNet-5, 98.91% on GPU and 98.88% on TPU. Our network is 1.2175x faster than LeNet-5 on GPU and 1.6985x on TPU, which was expected due to the low number of trainable parameters in the proposed model.

We can observe in Table 4.2 the values of loss and accuracy of the classification phase of the two

CNN	GPU Loss	GPU Accuracy	GPU Time	TPU Loss	TPU Accuracy	TPU Time
LeNet-5	0,0351	0,9891	50,15	0,0353	0,9888	232,19
Proposed CNN	0,1053	0,9679	41,19	0,1102	0,9668	136,70

Table 4.1: Performance comparison between LeNet-5 and the Proposed CNN in the training stage.

models, as well as the time they take to be executed. In the classification phase, we achieved similar loss values. The accuracy obtained was very close to those obtained in the training phase, indicating no overfitting in any model. We can also highlight that the classification times are significantly faster than the training times. We consider our model valid for classifying handwritten digits belonging to the MNIST dataset with the obtained results.

CNN	GPU Loss	GPU Accuracy	GPU Time	TPU Loss	TPU Accuracy	TPU Time
LeNet-5	0,0374	0,9873	1,06	0,0561	0,9826	2,19
Proposed CNN	0,1017	0,9676	1,05	0,0918	0,9719	1,11

Table 4.2: Performance comparison between LeNet-5 and the Proposed CNN in the classification stage.

As mentioned in 4.1.1, TPUs are ASICs explicitly designed for neural networks optimized for complex neural network models with a large number of parameters. As the Modified LeNet-5 is a CNN with only 1158 parameters, TPU presented an expected behavior, being slower than GPU.

4.2 CNN - RTL implementation

Figure 4.3 shows the number of clock cycles each layer takes in our implementation. It clearly shows that convolutional layers take longer to be completed, since they are the most computationally intensive layers. As expected, the first layers take longer to execute than layers of the same type than the following ones since the number of calculations decreases as the layer advances. Maxpool layers are the layers that run the fastest. One suggestion to improve network performance is that convolutional layers are parallelized whenever possible.

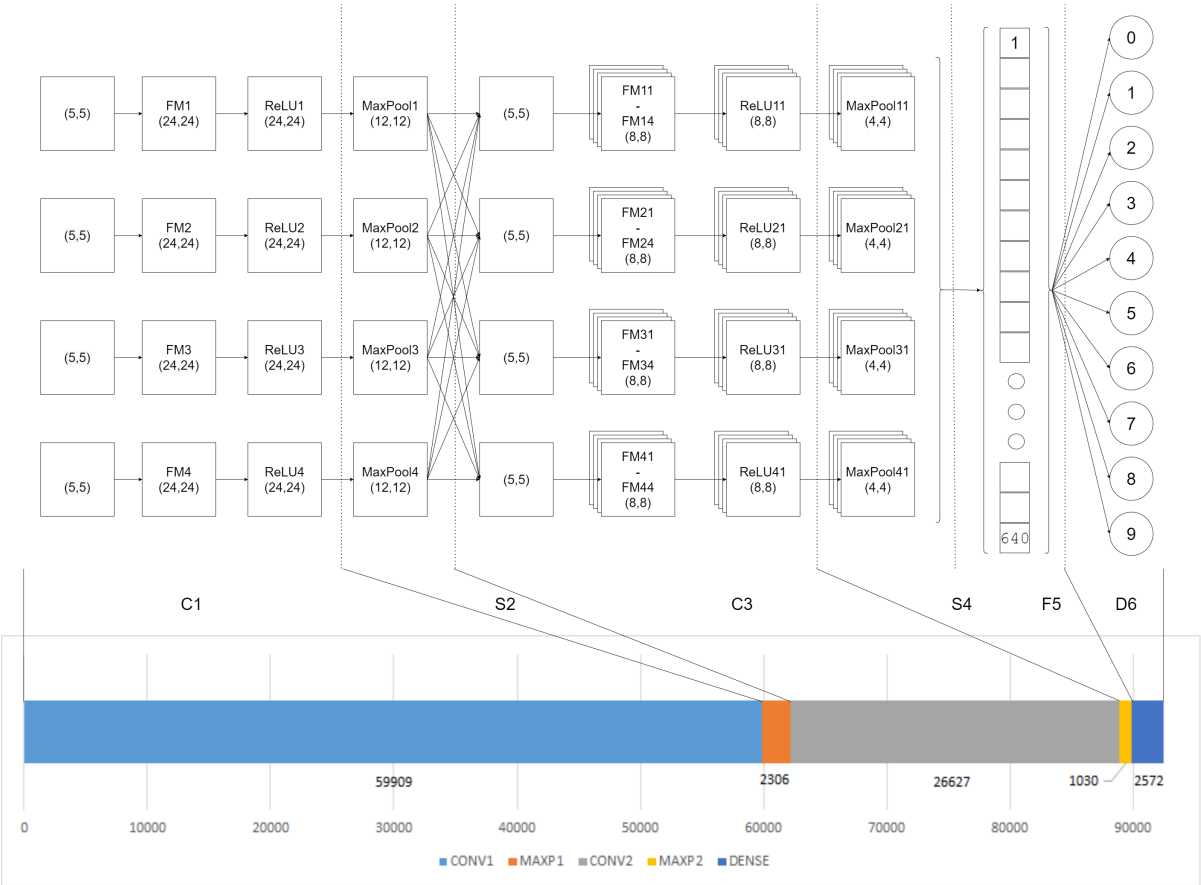

Figure 4.3: Number of clock cycles per layer.

Although we could classify an image, the classification is not accurate. In Figure 4.4, the CNN classifies an image of the number 7 as 0, despite 'class_o' is 1. On the right side of the figure, we verify the same error as in 4.10, where the info is shifted one address up and an information loss in the second and third addresses.

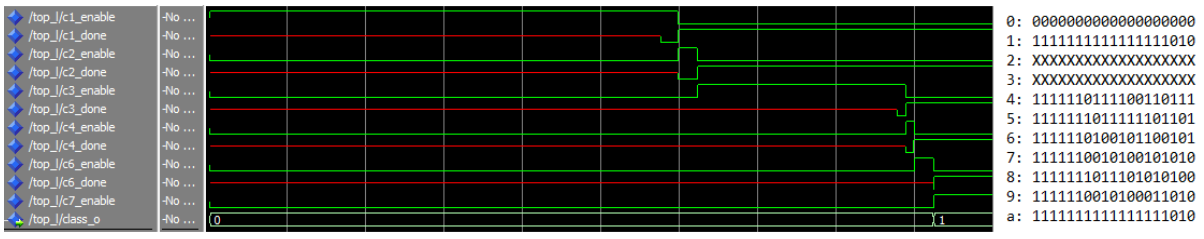

Figure 4.4: Error in classification.

All layers of the network were implemented sequentially, and in Figure 4.5 it is possible to see the RTL overview of our implementation, where each rectangle indicates a layer. Every layer only starts when the previous layer has finished the computations. For more details see Appendix B. To control the sequential execution of the layers, we designed a controller with the state machine in Figure 4.6. The controller is designed for classifying one image. However, it can be easily adapted for more images.

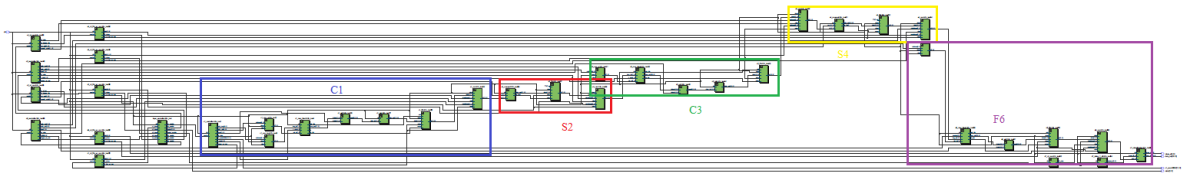

Figure 4.5: RTL implementation overview of Modified LeNet-5.

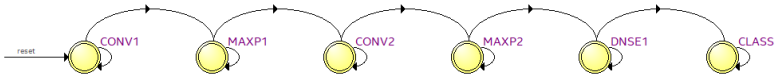

Figure 4.6: State machine for layer execution control.

We verified that all address generators are generating addresses correctly and that the ReLU (Figure 4.7) and Maxpool (Figure 4.8) functions also work correctly. An example of MAC operations is in Figure 4.9. However, we have detected that the output of C1 is not written properly in RAM.

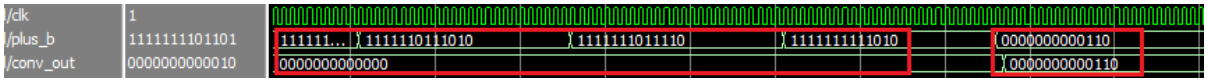

Figure 4.7: Example of ReLU operations. The input of the ReLU block is '$plus\_b$' and the output is '$conv\_out$'.

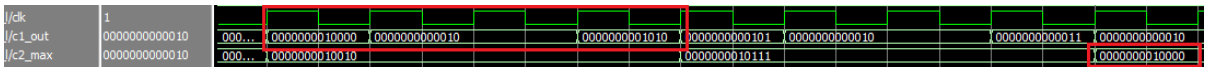

Figure 4.8: Example of a Maxpool operation. The input of the Maxpool block is '$c1\_out$' and the output is '$c2\_max$'.

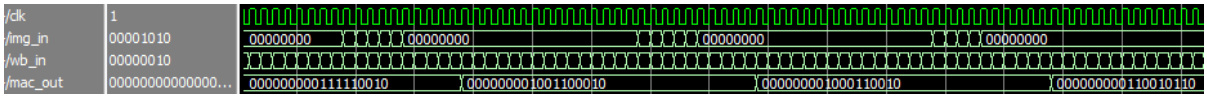

Figure 4.9: Example of a MAC based convolution. The inputs of the MAC block are 'img_in' and 'wb_in' and the output is 'mac_out'.

In Figure 4.10, we observed that we lose information on address 0x002. After debugging, we concluded that this error occurred due to bad memory access techniques. Better addressing techniques when writing to memories must be studied to solve this problem. To find if the writing error is at the moment we write in RAM, we can replace the data written by the corresponding address. So, if the written address is different from the actual address, we are experiencing a delay while writing. If we get the address properly written, we must verify if the FIFO before the RAM is receiving only the intended data, otherwise we must force a FIFO clear at the clock cycle before the first data write.

```
0: 0000000000000
1: 0000000000010
2: XXXXXXXXXXXX
3: 0000000000010
4: 0000000000010
5: 0000000000010
6: 0000000000010
```

Figure 4.10: Information loss on the output of layer C1.

4.3 Stochastic approach circuit

In our approach to Stochastic Computing, we were able to verify some concepts and limitations of this type of computation. We obtained results concerning the randomness of an LFSR and verified the limitations in generating numbers. We also evaluate the Root-Mean-Square Error (RMSE) as a function of the bitstream size, while generating the bitstream, and the RMSE between the predicted and the observed values after the multiplication circuit of Figure 3.14.

4.3.1 Pseudo-random sequence

First, we studied the behavior of LFSR to validate its implementation. In Figure 4.11, we can observe the behavior of the 8bit LFSR. The 'random_a' signal is generated through the 'seed_a', equal to 58, while the 'random_b' signal is generated through the 'seed_b', equal to 139. It is also possible to verify that a new pseudo-random number is generated every clock cycle. Since we are using a LFSR with 8bit resolution, it is only possible to generate 255 different numbers before the random sequence repeats itself.

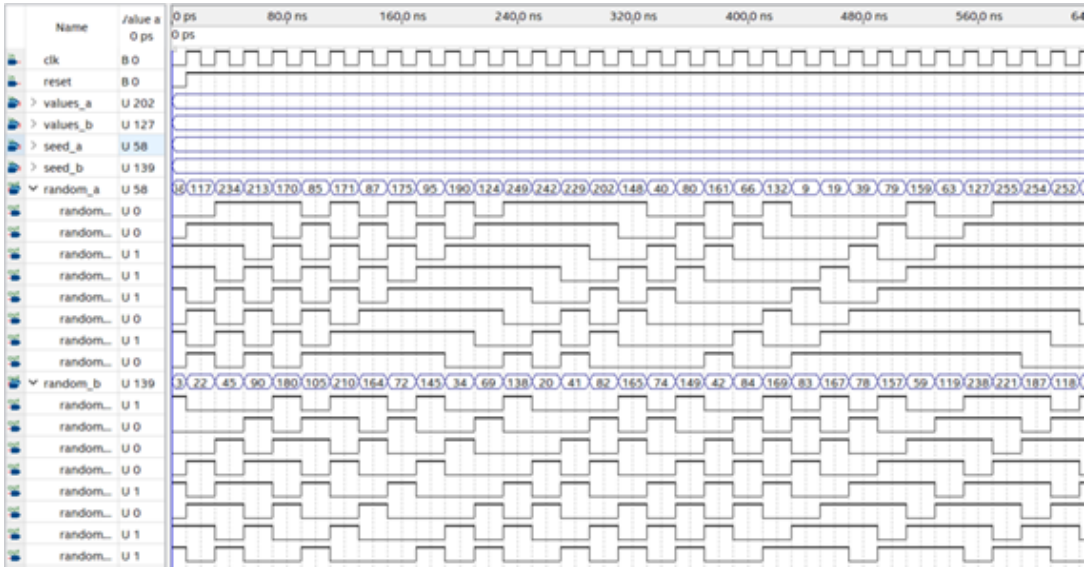

Figure 4.11: Snippet of the pseudo-random numbers generated by the LFSRs from Figure 3.14.

With our RNG working properly, we advanced to the binary to stochastic conversion, where we studied the RMSE between the predicted and the expected values.

Predicted value (Stochastic)	Predicted value (Decimal)	Predicted value (Binary)
0.0157	4/255	00000100
0.2000	51/255	00110011
0.4980	127/255	01111111
0.8000	204/255	11001100
0.9843	251/255	11111011

Table 4.3: Test cases for binary to stochastic conversion.

4.3.2 Root-Mean-Square Error in number generation

The tests were done using the numbers from Table 4.3, where we evaluated the RMSE on the binary to stochastic conversion as the bitstream size increases.

The Root-Mean-Square Error (RMSE), Equation 4.1, is frequently used to measure the differences between the predicted and the observed values. We use the RMSE to find the error between the predicted and the observed value of P. $P_{predicted}$ refers to the predicted value of the number, $P_{observed}$ to the observed value of the number and N to the total elements in the test case.

$$RMSE = \left[\sum_{i=1}^{N} \left(P_{predicted} - P_{observed}\right)^2 / N\right]^{1/2}. \tag{4.1}$$

Figure 4.12 presents the variation of the RMSE in function of the bitstream size. For more details, see Table C.1 in Appendix C.

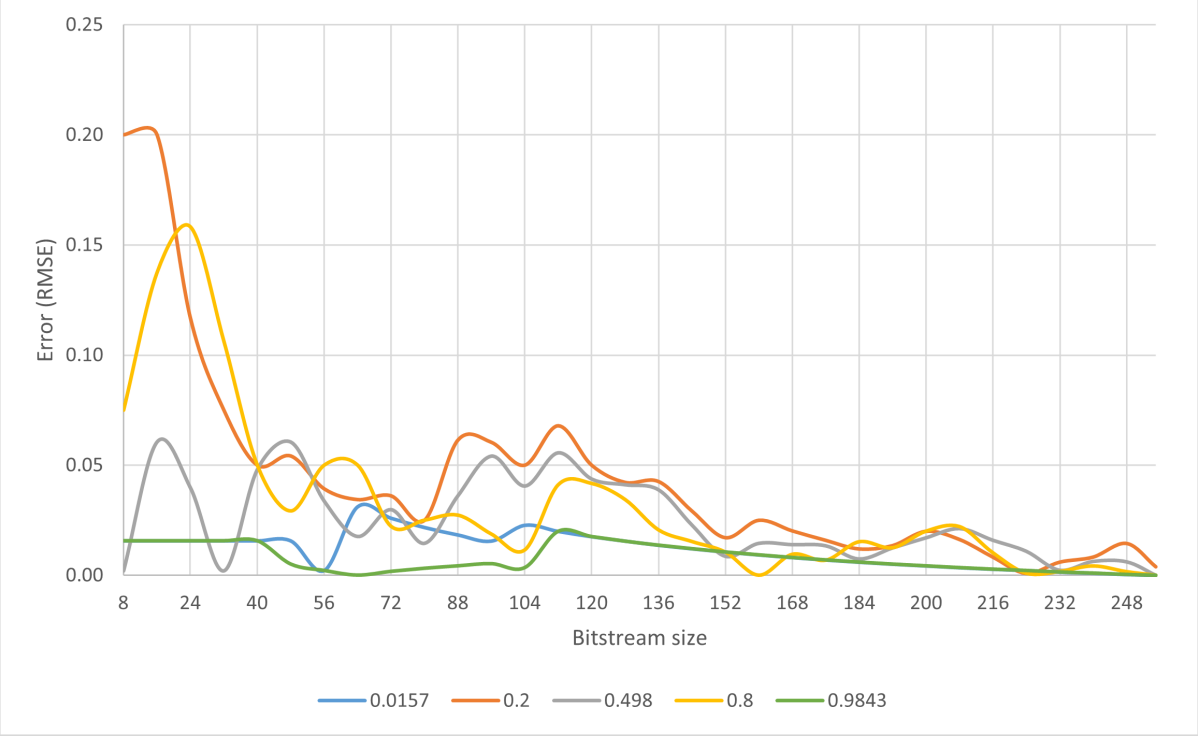

Figure 4.12: RMSE in function of bitstream length.

From the figure, we can see that when the numbers are very close to 0 and 1, the RMSE is quite low compared to the other cases. Since when the value is very close to 0 it is quite likely that the LFSR value is greater, almost always leaving a bit equal to 0 from the comparator and when the value is very close to 1 it is quite likely that the LFSR value is smaller, almost always leaving a bit equal to 1 from the comparator. In the case of the intermediate value, the comparator output alternates between 0 and 1 equally. In the remaining cases, with small bitstreams, the RMSE is larger compared to the previous cases, requiring longer bitstreams to approach its predicted value.

4.3.3 Root-Mean-Square Error variation in multiplication

To evaluate the RMSE after multiplication, we used the test cases from Table 4.4. The table indicates the expected values (P_expected) of the multiplication between the elements of the first row and the first column represented by a bitstream with 255 bits. Table 4.5 represents the observed values (P_observed) after our implementation.

x	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.1	0.0100	0.0200	0.0300	0.0400	0.0500	0.0600	0.0700	0.0800	0.0900
0.2	0.0200	0.0400	0.0600	0.0800	0.1000	0.1200	0.1400	0.1600	0.1800
0.3	0.0300	0.0600	0.0900	0.1200	0.1500	0.1800	0.2100	0.2400	0.2700
0.4	0.0400	0.0800	0.1200	0.1600	0.2000	0.2400	0.2800	0.3200	0.3600
0.5	0.0500	0.1000	0.1500	0.2000	0.2500	0.3000	0.3500	0.4000	0.4500
0.6	0.0600	0.1200	0.1800	0.2400	0.3000	0.3600	0.4200	0.4800	0.5400
0.7	0.0700	0.1400	0.2100	0.2800	0.3500	0.4200	0.4900	0.5600	0.6300
0.8	0.0800	0.1600	0.2400	0.3200	0.4000	0.4800	0.5600	0.6400	0.7200
0.9	0.0900	0.1800	0.2700	0.3600	0.4500	0.5400	0.6300	0.7200	0.8100

Table 4.4: Test cases with the expected values (P_expected) for the multiplication circuit.

x	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.1	0.0078	0.0196	0.0314	0.0392	0.0471	0.0627	0.0627	0.0784	0.0902
0.2	0.0118	0.0353	0.0588	0.0745	0.0980	0.1176	0.1373	0.1608	0.1765
0.3	0.0235	0.0510	0.0863	0.1098	0.1451	0.1725	0.2078	0.2392	0.2627
0.4	0.0471	0.0784	0.1255	0.1647	0.2000	0.2471	0.2863	0.3255	0.3647
0.5	0.0510	0.0980	0.1490	0.2000	0.2471	0.3020	0.3490	0.4000	0.4510
0.6	0.0549	0.1137	0.1804	0.2431	0.3020	0.3647	0.4275	0.4824	0.5412
0.7	0.0745	0.1412	0.2157	0.2941	0.3529	0.4314	0.5020	0.5686	0.6392
0.8	0.0824	0.1569	0.2392	0.3255	0.3961	0.4824	0.5608	0.6392	0.7255
0.9	0.0863	0.1804	0.2706	0.3569	0.4471	0.5373	0.6275	0.7137	0.8039

Table 4.5: Test cases with the observed values (P_observed) from the multiplication circuit.

Using the values from the table in Equation 4.1, we obtained the variation of the RMSE, which we represented in Figure 4.13. From the analysis of the figure, we verified that the RMSE is higher when we

multiply opposite values, indicating that there is a greater deviation of the RMSE between P_expected and P_observed. This deviation indicates that multiplication is more accurate when multiplying values close to 0 by the same values and when multiplying values close to 1 by the same values.

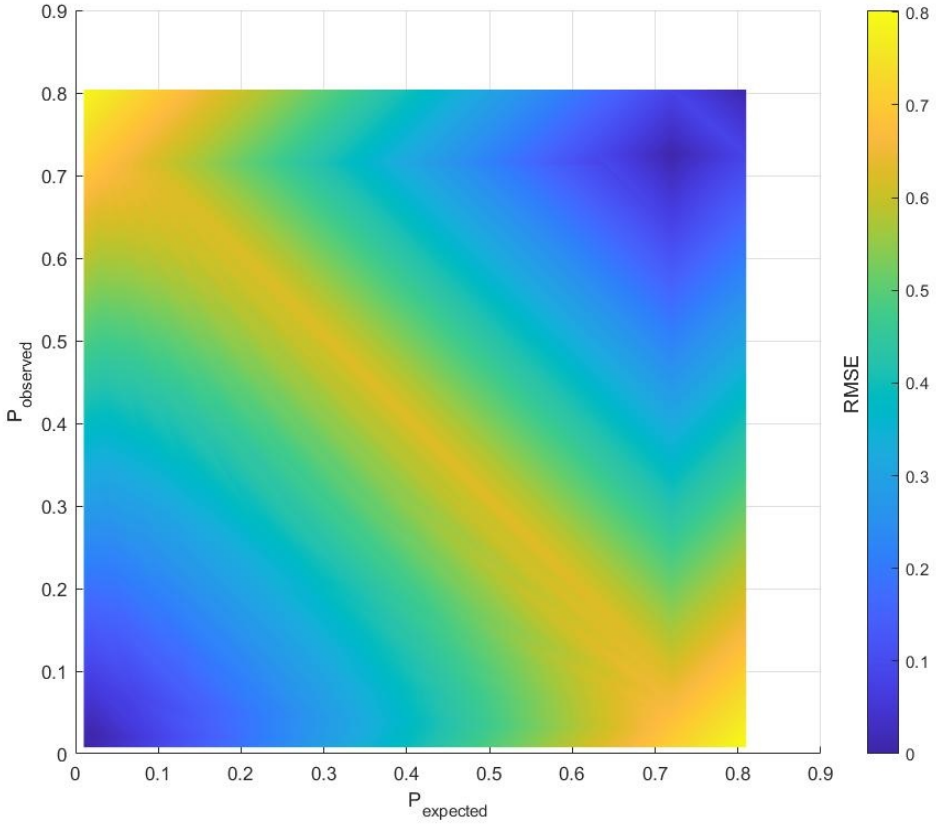

Figure 4.13: RMSE variation between the expected and observed values after multiplication.

Despite this variation, it is possible to obtain significantly accurate values, as the RMSE has a reduced magnitude. From this analysis we were able to validate the stochastic multiplication circuit, being possible to add it to the Modified LeNet-5 when it is ready to receive Stochastic Computing features.

4.4 Using SC in the CNN computation pipeline

Having shown some of the SC elements in the previous section and with the stochastic functions presented in Section 2.2.2, we move on to several proposals to introduce them in Modified LeNet-5. The options vary on where the data is in binary or stochastic domain in the computation pipeline.

Convolutional Layers

Figure 4.14 shows four possible SC changes in the convolutional layer data flows represented in Figures 3.9 and 3.11. All mathematical operations are done in the stochastic domain in the following four options:

- The top option of Figure 4.14 presents a data conversion to the stochastic domain, and after the

operations, the data is reconverted to the binary domain.

- In the middle option, the data is maintained in the stochastic domain at the output of the layer, in case the data can be processed in the same domain by the next layer.
- In the third option, data arrives at the current layer in the stochastic domain and is always kept in this domain. This option does not apply to the first layer of the network, as the data enters the network in the binary domain.
- Another option is to convert the stochastic input data to the binary domain after the convolutions.

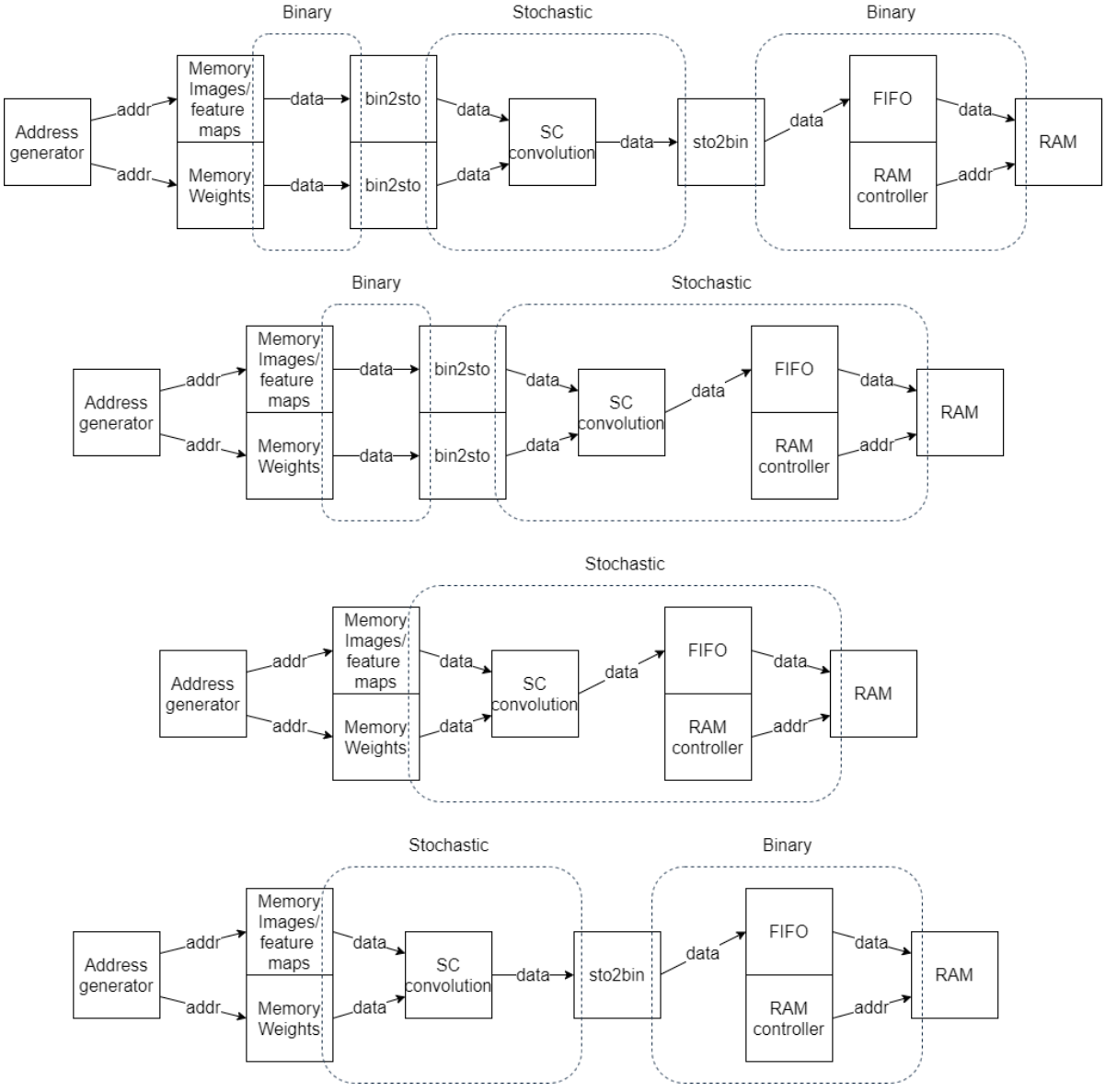

Figure 4.14: Convolutional layer options with SC.

Maxpool Layers

Figure 4.15 shows four possible SC changes in the data flows represented in Figures 3.10 and 3.12. All mathematical operations are done in the stochastic domain in the following four options:

- The top option of Figure 4.15 presents a data conversion to the stochastic domain, and after the

maxpool block, the data is reconverted to the binary domain.

- In the second option, the data is maintained in the stochastic domain at the output of the layer, in case the data can be processed in the same domain by the next layer.
- In the third option, data arrives at the current layer in the stochastic domain and is always kept in this domain. This option does not apply to the first layer of the network, as the data enters the network in the binary domain.
- Another option is to convert the stochastic input data to the binary domain after the convolutions.

Figure 4.15: Maxpool layer options with SC.

Dense Layer

Figure 4.16 shows four possible SC changes in the dense layer data flow represented in Figure 3.13. All mathematical operations are done in the stochastic domain in the following four options:

- The top option of Figure 4.16 presents a data conversion to the stochastic domain, and after the operations, the data is reconverted to the binary domain.
- Another option, when the input data is in the stochastic domain, is to reconvert the data to the binary domain after the operations in the stochastic domain.

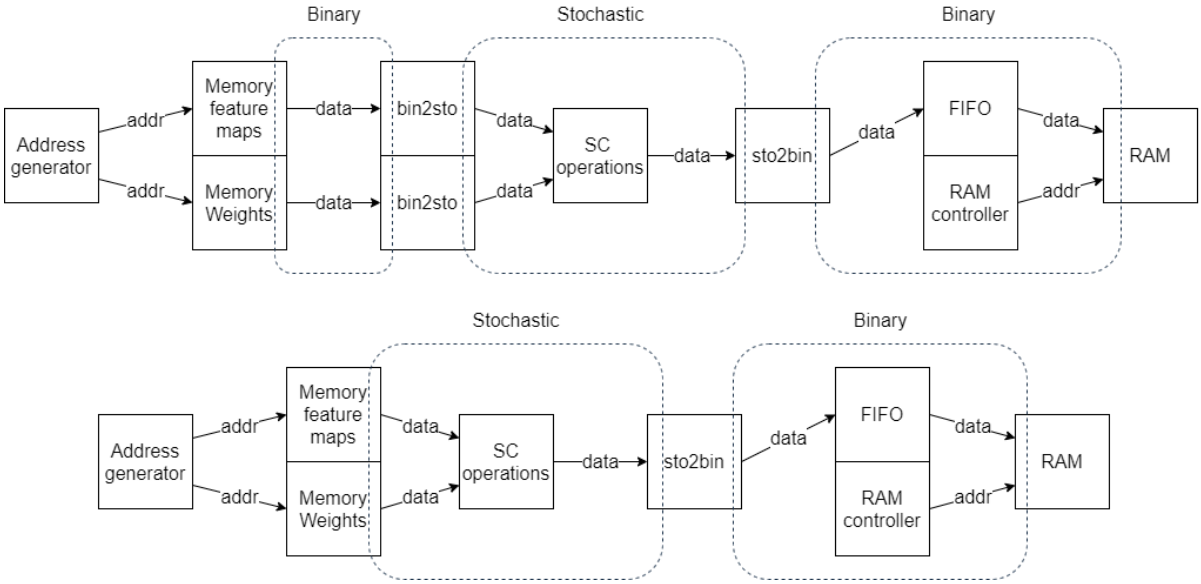

Figure 4.16: Dense layer options with SC.

Tests on timings and precision need to be performed to find the best option for each layer, combining the different options. Therefore is possible to adapt the Modified LeNet-5 to be fully implemented on the stochastic domain, having only data in the binary domain in the CNN input and output.

Chapter 5. Conclusions and Future Work

5.1 Conclusions

We were able to adapt the CNN LeNet-5 to a simpler version in terms of the number of complex layers and trainable parameters, with a negligible reduction in accuracy, to be implemented in an FPGA with Stochastic Computing features. Our Modified LeNet-5 performed classification of MNIST dataset images with approximately 97% accuracy on Tensorflow. During the implementation of CNNs on Tensorflow, it was possible to verify differences depending on the choice of hardware for the training, as the train in GPU is 2.8442x faster than TPU.

After training, we changed the format of the trainable parameters to enable their implementation on FPGA through a script developed in Matlab that converts the parameters into a Q3.5 fixed-point representation, sorts them, and writes them in Memory Initialization Files (MIF).

We built a skeleton of the CNN on FPGA, which despite not being fully functional, is capable of image classification, even though the classification is incorrect. However, we could verify the well-functioning of the MAC-based convolution, and the ReLU and MaxPool functions.

Finally, we tested a stochastic circuit with an RNG, Binary to Stochastic and Stochastic to binary conversions, and multiplication in the stochastic domain, where we verified the limitation of the 8bit LFSR and the problem in generating accurate representations in the stochastic domain using smaller bitstream sizes.

5.2 Future work

In order to pursue the ultimate goal of a CNN implemented on FPGA with stochastic features, memory writing techniques should be studied, since it was not possible to have the CNN working correctly due to writing wrong values in the memories which propagate through the following layers.

After successfully writing the memories, the CNN should be fully functional and ready to receive stochastic computing features and then compare the trade-offs of the proposed approach with deterministic computing methods and evaluate parameters such as energy consumption, resource utilization, and fault tolerance.

References

[1] M. Waldrop. “The chips are down for Moore’s law”. In: *Nature News* 530 (Feb. 2016), p. 144. DOI: 10.1038/530144a.

[2] Armin Alaghi and John P. Hayes. “Survey of Stochastic Computing”. In: *ACM Trans. Embed. Comput. Syst.* 12.2s (May 2013). ISSN: 1539-9087. DOI: 10.1145/2465787.2465794.

[3] J. Neumann. “Probabilistic Logics and the Synthesis of Reliable Organisms From Unreliable Components”. In: Dec. 1956, pp. 43–98. ISBN: 9781400882618. DOI: 10.1515/9781400882618-003.

[4] Abdul Halim Z. Lee YY. “Stochastic computing in convolutional neural network implementation: a review”. In: (Nov. 2020). URL: https://doi.org/10.7717/peerj-cs.309.

[5] *Bottom-up Approaches to Machines dedicated to Bayesian Inference*. URL: https://cordis.europa.eu/project/id/618024.

[6] H. Fernandes, M. A. Aslam, J. Lobo, J. F. Ferreira, and J. Dias. “Bayesian inference implemented on FPGA with stochastic bitstreams for an autonomous robot”. In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. 2016, pp. 1–4. DOI: 10.1109/FPL.2016.7577312.

[7] M. Faix, E. Mazer, R. Laurent, M. Othman Abdallah, R. Le Hy, and J. Lobo. “Cognitive computation: A Bayesian machine case study”. In: *2015 IEEE 14th International Conference on Cognitive Informatics Cognitive Computing (ICCI*CC)*. 2015, pp. 67–75. DOI: 10.1109/ICCI-CC.2015.7259367.

[8] A. Coninx, P. Bessière, E. Mazer, J. Droulez, R. Laurent, M. A. Aslam, and J. Lobo. “Bayesian sensor fusion with fast and low power stochastic circuits”. In: *2016 IEEE International Conference on Rebooting Computing (ICRC)*. 2016, pp. 1–8. DOI: 10.1109/ICRC.2016.7738672.

[9] P. Li and D. J. Lilja. “Using stochastic computing to implement digital image processing algorithms”. In: *2011 IEEE 29th International Conference on Computer Design (ICCD)*. 2011, pp. 154–161. DOI: 10.1109/ICCD.2011.6081391.

[10] M. Ranjbar, M. E. Salehi, and M. H. Najafi. “Using stochastic architectures for edge detection algorithms”. In: *2015 23rd Iranian Conference on Electrical Engineering*. 2015, pp. 723–728. DOI: 10.1109/IranianCEE.2015.7146308.

[11] Lucas Neves Carvalho. “Projeto e treinamento de redes neurais em hardware FPGA usando computação estocástica”. In: (2016).

[12] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[13] Zeyad Assi Obaid Nasri Sulaiman, MH Marhaban, and MN Hamidon. “Design and implementation of FPGA-based systems-a review”. In: *Australian Journal of Basic and Applied Sciences* 3.4 (2009), pp. 3575–3596.

[14] Altera Corporation. “Cyclone IV Device Handbook, Volume 1”. In: 2016.

[15] *Altera DE2-115 Development and Education Board*. URL: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=502.

[16] *Intel Quartus Prime Software*. URL: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html.

[17] Intel FPGA. *ModelSim - Intel FPGA Edition Software*. URL: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/model-sim.html.

[18] Brian Gaines. “Stochastic Computing Systems”. In: *Adv Inf Syst Sci* 2 (Jan. 1969). DOI: 10.1007/978-1-4899-5841-9_2.

[19] A. Alaghi, W. Qian, and J. P. Hayes. “The Promise and Challenge of Stochastic Computing”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.8 (2018), pp. 1515–1531. DOI: 10.1109/TCAD.2017.2778107.

[20] Vincent T Lee, Armin Alaghi, John P Hayes, Visvesh Sathe, and Luis Ceze. “Energy-efficient hybrid stochastic-binary neural networks for near-sensor computing”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE. 2017, pp. 13–18.

[21] José Alves. *Survey of Random Number Generation in FPGA or Other/External Devices - BAMBI Tech Report*. 2015. URL: https://home.deec.uc.pt/~jlobo/publications/survey_of_rng_jalves_jlobo_BAMBI.pdf.

[22] Hounghun Joe and Youngmin Kim. “Novel Stochastic Computing for Energy-Efficient Image Processors”. In: *Electronics* 8.6 (2019). ISSN: 2079-9292. DOI: 10.3390/electronics8060720. URL: https://www.mdpi.com/2079-9292/8/6/720.

[23] Prabhat Kumar Gupta and Ramdas Kumaresan. “Binary multiplication with PN sequences”. In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 36.4 (1988), pp. 603–606. DOI: 10.1109/29.1564.

[24] Meng Yang, Bingzhe Li, David J Lilja, Bo Yuan, and Weikang Qian. “Towards theoretical cost limit of stochastic number generators for stochastic computing”. In: *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2018, pp. 154–159. DOI: 10.1109/ISVLSI.2018.00037.

[25] Zhe Li, Ji Li, Ao Ren, Ruizhe Cai, Caiwen Ding, Xuehai Qian, Jeffrey Draper, Bo Yuan, Jian Tang, Qinru Qiu, et al. “HEIF: Highly efficient stochastic computing-based inference framework for deep neural networks”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.8 (2018), pp. 1543–1556.

[26] Li Fei-Fei, Jia Deng, and Kai Li. “ImageNet: Constructing a large-scale image database”. In: *Journal of vision* 9.8 (2009), pp. 1037–1037.

[27] Bradley D Brown and Howard C Card. “Stochastic neural computation. I. Computational elements”. In: *IEEE Transactions on computers* 50.9 (2001), pp. 891–905.

[28] Joonsang Yu, Kyounghoon Kim, Jongeun Lee, and Kiyoung Choi. “Accurate and efficient stochastic computing hardware for convolutional neural networks”. In: *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE. 2017, pp. 105–112.

[29] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. “Dropout: a simple way to prevent neural networks from overfitting”. In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[30] M. K. Hamdan and D. T. Rover. “VHDL generator for a high performance convolutional neural network FPGA-based accelerator”. In: *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2017, pp. 1–6. DOI: 10.1109/RECONFIG.2017.8279827.

[31] *MNIST dataset*. URL: http://yann.lecun.com/exdb/mnist/.

[32] *Tensorflow*. URL: https://www.tensorflow.org/.

[33] Google. *Welcome to Colab!* URL: https://colab.research.google.com/?utm_source=scs-index.

[34] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. “TensorFlow: A System for Large-Scale Machine Learning”. In: *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 2016, pp. 265–283.

[35] Google. *Cloud Tensor Processing Units (Cloud TPUs)*. URL: https://cloud.google.com/tpu/docs/tpus.

[36] Intel. *Intel® Xeon® Processor*. URL: https://www.intel.com/content/www/us/en/products/sku/27267/intel-xeon-processor-2-20-ghz-512k-cache-400-mhz-fsb/specifications.html.

[37] NVIDIA. *NVIDIA TESLA K80*. URL: https://www.nvidia.com/en-gb/data-center/tesla-k80/.

[38] Prasun Biswas. *Importance of Loss functions in Deep Learning and Python Implementation*. URL: https://towardsdatascience.com/importance-of-loss-functions-in-deep-learning-and-python-implementation-4307bfa92810.

[39] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[40] Yoshua Bengio. “Practical recommendations for gradient-based training of deep architectures”. In: *Neural networks: Tricks of the trade*. Springer, 2012, pp. 437–478.

[41] Dominic Masters and Carlo Luschi. “Revisiting small batch training for deep neural networks”. In: *arXiv preprint arXiv:1804.07612* (2018).

[42] Erick L Oberstar. “Fixed-point representation & fractional math”. In: *Oberstar Consulting* 9 (2007).

[43] Randy Yates. “Fixed-point arithmetic: An introduction”. In: *Digital Signal Labs* 81.83 (2009), p. 198.

[44] *Matlab*. URL: https://www.mathworks.com/products/matlab.html.

[45] Joseph. *Decimal to Fixed-point Q Format Converter*. 2021. URL: https://www.mathworks.com/matlabcentral/fileexchange/61669-decimal-to-fixed-point-q-format-converter.

Appendix A. Tensorflow

This section contains support material for Figures 4.1 and 4.2 in Chapter 4.1. It covers data obtained on Tensorflow.

Table A.1 presents the values corresponding to the comparison of speedup, loss, and accuracy between TPU and GPU implementation of our convolutional neural network, with only five scenarios. The plot of Figure 4.1 was built using this values.

Table A.1: Comparison of speedup, loss, and accuracy between TPU and GPU implementation of our convolutional neural network.

	TPU			GPU			TPU/GPU		
Number of epochs	Time (s)	Loss	Accuracy	Time (s)	Loss	Accuracy	Speedup	Loss	Accuracy
1	41,15	0,2094	0,9383	33,06	0,166	0,9516	1,24	1,2614	0,9860
5	140,69	0,0879	0,9728	61,23	0,0854	0,9727	2,30	1,0293	1,0001
10	255,29	0,0778	0,9755	96,41	0,0778	0,9751	2,65	1,0000	1,0004
20	506,65	0,0635	0,9789	158,8	0,0591	0,9813	3,19	1,0745	0,9976
50	1297,84	0,0553	0,9834	374,36	0,0575	0,9834	3,47	0,9617	1,0000

Table A.2 presents the values corresponding to the comparison of speedup, loss, and accuracy between TPU and GPU implementation of our convolutional neural network, for 20 epochs. The first scenario present in Figure 4.2 was built using this values.

Table A.2: Comparison of speedup, loss, and accuracy between TPU and GPU implementation of our convolutional neural network. The plot on the left side of Figure 4.2 was built using this table.

	TPU			GPU			TPU/GPU		
Number of epochs	Time	Loss	Accuracy	Time	Loss	Accuracy	Speedup	Loss	Accuracy
1	43,16	0,1843	0,9426	31,05	0,2245	0,9296	1,3900	0,8209	1,0140
2	75,33	0,1545	0,9514	38,09	0,1484	0,9566	1,9777	1,0411	0,9946
3	101,47	0,1132	0,9649	45,12	0,1204	0,9640	2,2489	0,9402	1,0009
4	121,56	0,1182	0,9610	54,16	0,0979	0,9707	2,2445	1,2074	0,9900
5	140,71	0,0853	0,9742	62,21	0,0834	0,9744	2,2619	1,0228	0,9998
6	176,88	0,0899	0,9730	70,25	0,0813	0,9736	2,5179	1,1058	0,9994
7	197,01	0,0772	0,9746	69,28	0,0747	0,9778	2,8437	1,0335	0,9967
8	225,14	0,0699	0,9773	75,32	0,0745	0,9783	2,9891	0,9383	0,9990
9	246,25	0,0963	0,9678	95,37	0,0656	0,9800	2,5820	1,4680	0,9876
10	256,34	0,0759	0,9773	91,40	0,0783	0,9724	2,8046	0,9693	1,0050
11	285,47	0,0737	0,9768	111,45	0,0706	0,9775	2,5614	1,0439	0,9993
12	328,68	0,0701	0,9772	118,49	0,0714	0,9769	2,7739	0,9818	1,0003
13	340,78	0,0720	0,9768	118,52	0,0606	0,9813	2,8753	1,1881	0,9954
14	385,99	0,0720	0,9769	124,57	0,0674	0,9795	3,0986	1,0682	0,9973
15	411,13	0,0630	0,9802	143,06	0,0591	0,9811	2,8738	1,0660	0,9991
16	444,28	0,0641	0,9788	143,64	0,0673	0,9775	3,0930	0,9525	1,0013
17	451,33	0,0600	0,9797	147,68	0,0562	0,9808	3,0561	1,0676	0,9989
18	468,39	0,0693	0,9771	164,77	0,0614	0,9805	2,8427	1,1287	0,9965
19	491,52	0,0598	0,9791	181,83	0,0647	0,9790	2,7032	0,9243	1,0001
20	509,66	0,0788	0,9766	178,82	0,0597	0,9811	2,8501	1,3199	0,9954

Table A.3 presents the values corresponding to the comparison of speedup, loss, and accuracy between TPU and GPU implementation of our convolutional neural network, for 20 epochs. The second scenario present in Figure 4.2 was built using this values.

Table A.3: Comparison of speedup, loss, and accuracy between TPU and GPU implementation of our convolutional neural network. The plot on the right side of Figure 4.2 was built using this table.

	TPU			GPU			TPU/GPU		
Number of epochs	Time	Loss	Accuracy	Time	Loss	Accuracy	Speedup	Loss	Accuracy
1	31,13	0,3928	0,8750	26,06	0,4083	0,8701	1,1946	0,9620	1,0056
2	56,26	0,1442	0,9566	34,10	0,1689	0,9483	1,6499	0,8538	1,0088
3	81,39	0,1200	0,9636	42,14	0,1325	0,9593	1,9314	0,9057	1,0045
4	106,52	0,1056	0,9675	50,18	0,1186	0,9639	2,1228	0,8904	1,0037
5	131,65	0,0978	0,9692	58,22	0,1120	0,9660	2,2613	0,8732	1,0033
6	156,78	0,0927	0,9714	66,26	0,0992	0,9693	2,3661	0,9345	1,0022
7	181,91	0,0853	0,9740	74,30	0,0971	0,9711	2,4483	0,8785	1,0030
8	207,04	0,0815	0,9757	82,34	0,0917	0,9724	2,5145	0,8888	1,0034
9	232,17	0,0770	0,9765	90,38	0,0856	0,9735	2,5688	0,8995	1,0031
10	257,30	0,0750	0,9765	98,42	0,0817	0,9755	2,6143	0,9180	1,0010
11	282,43	0,0720	0,9778	106,46	0,0775	0,9766	2,6529	0,9290	1,0012
12	307,56	0,0684	0,9791	114,50	0,0771	0,9771	2,6861	0,8872	1,0020
13	332,69	0,0669	0,9790	122,54	0,0733	0,9777	2,7150	0,9127	1,0013
14	357,82	0,0659	0,9803	130,58	0,0722	0,9780	2,7402	0,9127	1,0024
15	382,95	0,0644	0,9806	138,62	0,0695	0,9791	2,7626	0,9266	1,0015
16	408,08	0,0603	0,9816	146,66	0,0674	0,9797	2,7825	0,8947	1,0019
17	433,21	0,0628	0,9803	154,70	0,0685	0,9792	2,8003	0,9168	1,0011
18	458,34	0,0596	0,9816	162,74	0,0611	0,9814	2,8164	0,9755	1,0002
19	483,47	0,0613	0,9811	170,78	0,0653	0,9801	2,8310	0,9387	1,0010
20	508,60	0,0577	0,9819	178,82	0,0617	0,9815	2,8442	0,9352	1,0004

Appendix B. Quartus Prime Software - CNN

In this section, we present the values used to build the plot of Figure 4.3. We also present Figure 4.5 divided in four larger images for a better comprehension.

The number of clock cycles per layers was obtained by simulating our Modified LeNet-5 on ModelSim. This value is obtained by dividing the time that each layer takes to complete by the clock period we used, which was 100ps.

Table B.1: Number of clock cycles per layer. The simulation used a 10GHz clock.

Layer	C1	S2	C3	S4	D6
Time (ps)	5990900	230600	2662700	103000	257200
Clock cycles	59909	2306	26627	1030	2572

The following four figures correspond to Figure 4.5, which is the RTL implementation overview of our Modified LeNet-5.

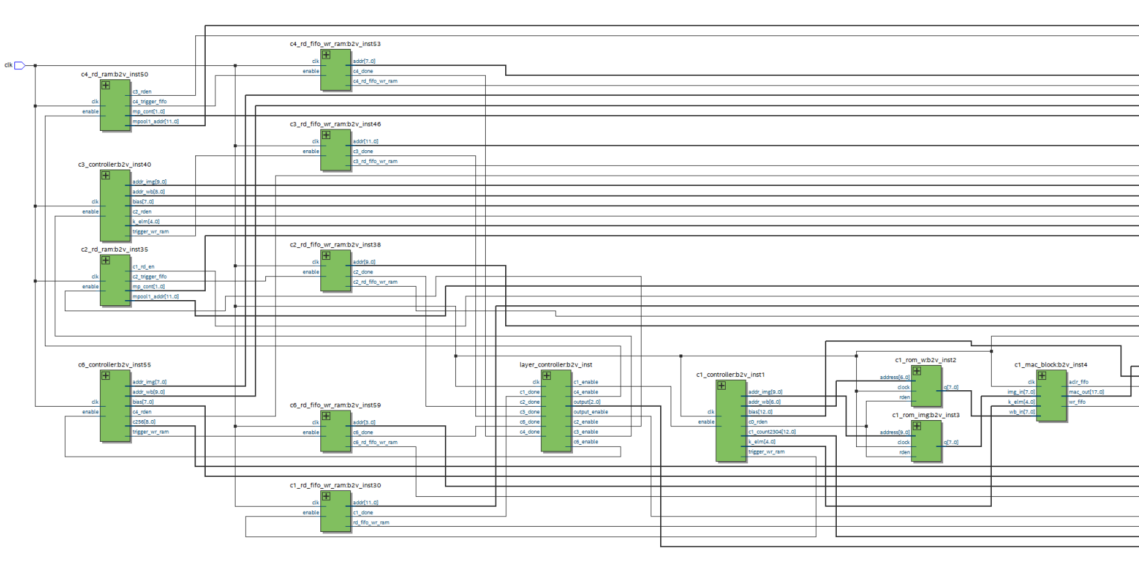

Figure B.1: RTL overview of Modified LeNet-5 - Part 1.

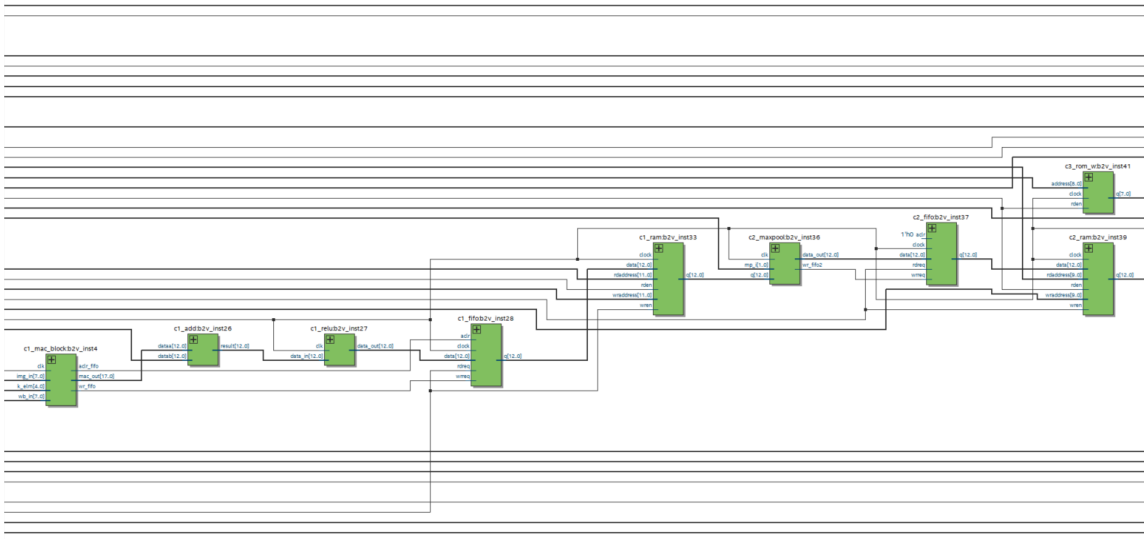

Figure B.2: RTL overview of Modified LeNet-5 - Part 2.

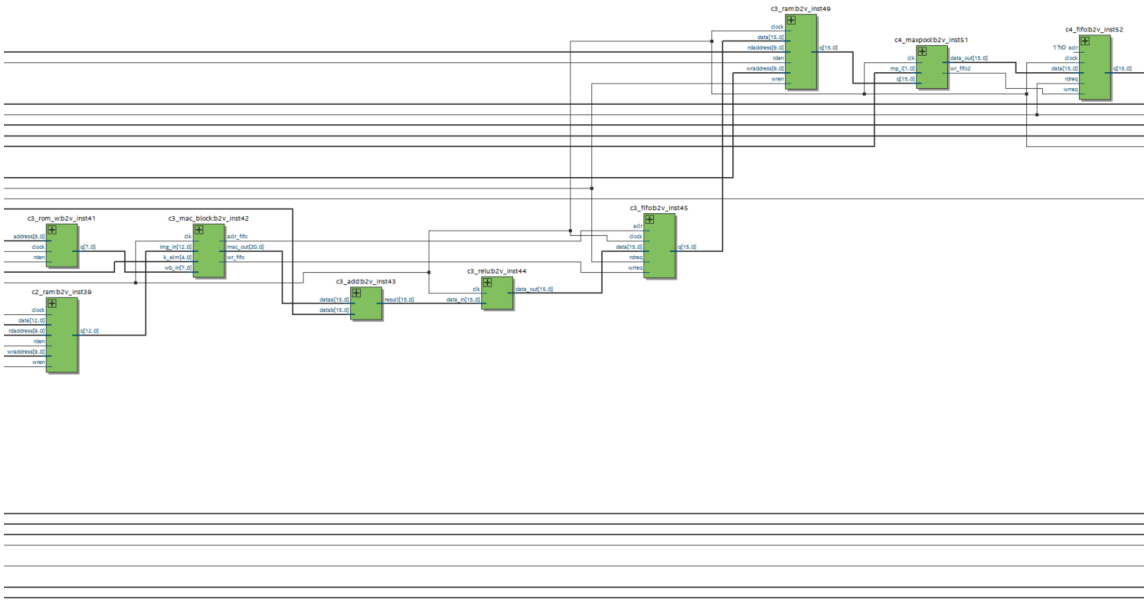

Figure B.3: RTL overview of Modified LeNet-5 - Part 3.

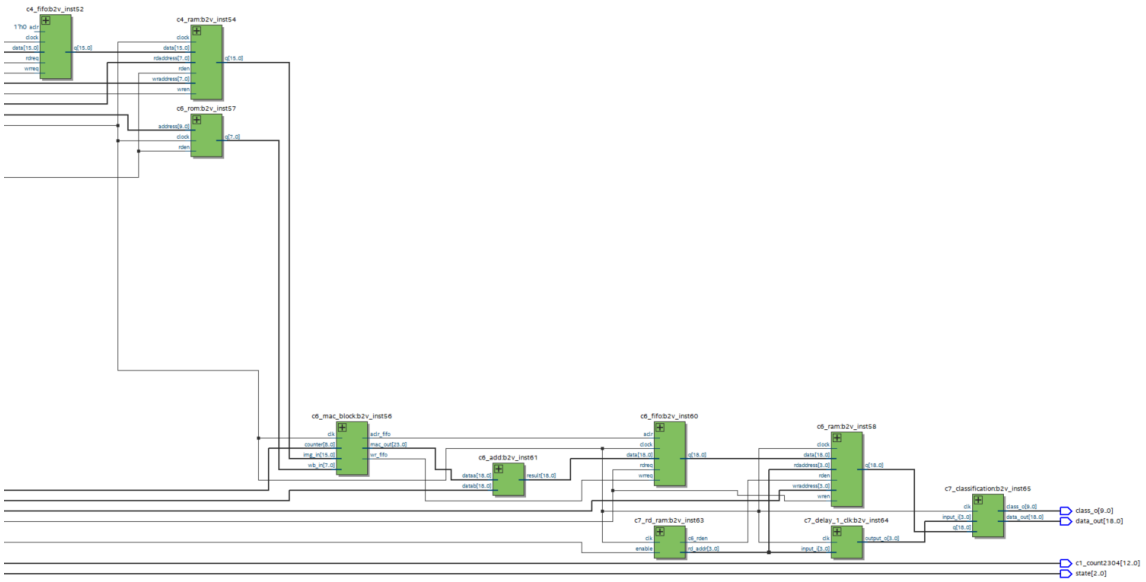

Figure B.4: RTL overview of Modified LeNet-5 - Part 4.

Appendix C. Stochastic

This section presents the values obtained while testing the stochastic computing multiplication circuit. We obtained this values by simulation of the circuit in Quartus Prime Software.

Bitstream length	**0.0157**	**0.2000**	**0.4980**	**0.8000**	**0.9843**
8	0.0157	0.2000	0.0020	0.0750	0.0157
16	0.0157	0.2000	0.0605	0.1375	0.0157
24	0.0157	0.1167	0.0397	0.1583	0.0157
32	0.0157	0.0750	0.0020	0.1063	0.0157
40	0.0157	0.0500	0.0480	0.0500	0.0157
48	0.0157	0.0542	0.0605	0.0292	0.0051
56	0.0022	0.0393	0.0337	0.0500	0.0022
64	0.0312	0.0344	0.0176	0.0500	0.0001
72	0.0260	0.0361	0.0298	0.0222	0.0018
80	0.0218	0.0250	0.0145	0.0250	0.0032
88	0.0184	0.0614	0.0361	0.0273	0.0043
96	0.0156	0.0604	0.0541	0.0187	0.0053
104	0.0228	0.0500	0.0405	0.0115	0.0035
112	0.0200	0.0679	0.0556	0.0411	0.0200
120	0.0176	0.0500	0.0437	0.0417	0.0176
128	0.0156	0.0422	0.0411	0.0344	0.0155
136	0.0137	0.0426	0.0388	0.0206	0.0137
144	0.0121	0.0292	0.0228	0.0153	0.0121
152	0.0106	0.0171	0.0086	0.0105	0.0106
160	0.0093	0.0250	0.0145	0.0000	0.0093
168	0.0081	0.0202	0.0139	0.0095	0.0081
176	0.0070	0.0159	0.0134	0.0068	0.0070
184	0.0060	0.0120	0.0074	0.0152	0.0060
192	0.0051	0.0135	0.0124	0.0125	0.0051
200	0.0043	0.0200	0.0170	0.0200	0.0043
208	0.0035	0.0163	0.0212	0.0221	0.0035
216	0.0028	0.0083	0.0159	0.0102	0.0028
224	0.0022	0.0009	0.0109	0.0009	0.0022
232	0.0015	0.0060	0.0023	0.0017	0.0015
240	0.0010	0.0083	0.0063	0.0042	0.0010
248	0.0004	0.0145	0.0061	0.0016	0.0004
255	0.0000	0.0039	0.0000	0.0000	0.0000

Table C.1: RMSE in function of bitstream size. The column headers correspond to the predicted value of each test case.