UNIVERSIDADE DO VALE DO RIO DOS SINOS - UNISINOS
UNIDADE ACADÊMICA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA
NÍVEL MESTRADO PROFISSIONAL

Pablo Eduardo Pereira de Araujo Cottens

Development Of An Artificial Neural Network Architecture Using Programmable Logic

São Leopoldo, RS

2016

Pablo Eduardo Pereira de Araujo Cottens

Development Of An Artificial Neural Network Architecture Using Programmable Logic

Trabalho de qualificação apresentado como requisito parcial para a obtenção do título de Mestre, pelo Programa de Pós-Graduação em Engenharia Elétrica da Universidade do Vale do Rio dos Sinos – UNISINOS.

Advisor Prof. Dr. Marcio Rosa da Silva

Co-advisor Prof. Ms. Rodrigo Marques de Figueiredo

São Leopoldo, RS

2016

Pablo Eduardo Pereira de Araujo Cottens

Development Of An Artificial Neural Network Architecture Using Programmable Logic

Trabalho de qualificação apresentado como requisito parcial para a obtenção do título de Mestre, pelo Programa de Pós-Graduação em Engenharia Elétrica da Universidade do Vale do Rio dos Sinos – UNISINOS.

São Leopoldo, RS
2016

Resumo

Normalmente Redes Neurais Artificiais (RNAs) necessitam estações de trabalho para o seu processamento, por causa da complexidade do sistema. Este tipo de arquitetura de processamento requer que instrumentos de campo estejam localizados na vizinhança da estação de trabalho, caso exista a necessidade de processamento em tempo real, ou que o dispositivo de campo possua como única tarefa a de coleta de dados para processamento futuro. Este projeto visa criar uma arquitetura em lógica programável para um neurônio genérico, no qual as RNAs podem fazer uso da natureza paralela de FPGAs para executar a aplicação de forma rápida. Este trabalho mostra que a utilização de lógica programável para a implementação de RNAs de baixa resolução de bits é viável e as redes neurais, devido à natureza paralelizável, se beneficiam pela implementação em hardware, podendo obter resultados de forma muito rápida.

Palavras-chaves : FPGA. SoC. Redes Neurais Artificiais. Neuronio em Hardware.

Abstract

Currently, modern Artificial Neural Networks (ANN), according to their complexity, require a workstation for processing all their input data. This type of processing architecture requires that the field device is located somewhere in the vicintity of a workstation, in case real-time processing is required, or that the field device at hand will have the sole task of collecting data for future processing, when field data is required. This project creates a generic neuron architecture in programmabl logic, where Artifical Neural Networks can use the parallel nature of FPGAs to execute applications in a fast manner, albeit not using the same resolution for its otputs. This work shows that the utilization of programmable logic for the implementation of low bit resolution ANNs is not only viable, but the neural network, due to its parallel nature, benefits greatly from the hardware implementation, giving fast and accurate results.

Key-words: FPGA. SoC. Artificial Neural Network. Hardware Neuron.

List of Figures

Figure 1 – Architecture of an FPGA 22
Figure 2 – Microprocessor Architecture 22
Figure 3 – ARM A8/A9 pipeline 24
Figure 4 – ARM Cortex-A9 Architecture 25
Figure 5 – Zynq7000 Architecture 25
Figure 6 – Generic Neuron Life cycle 28
Figure 7 – Neuron Life cycle 29
Figure 8 – Neuron 29
Figure 9 – Sigmoid activation function 30
Figure 10 – Continuous Tan-Sigmoid Function 31
Figure 11 – Layers of an ANN 33
Figure 12 – One layer ANN 33
Figure 13 – Recurring ANN 34
Figure 14 – Neuron Architecture 38
Figure 15 – Main building blocks 42
Figure 16 – Three neuron ANN - 01 input 45
Figure 17 – Three neuron ANN - 00 input 45
Figure 18 – Three neuron ANN - 11 input 46
Figure 19 – Four neuron ANN - 01 input 48
Figure 20 – Four neuron ANN - 00 input 48
Figure 21 – Four neuron ANN - 11 input 48
Figure 22 – Four neuron ANN - 10 input 49
Figure 23 – Eight neuron ANN - 01 input 51
Figure 24 – Eight neuron ANN - 00 input 51
Figure 25 – Eight neuron ANN - 11 input 52
Figure 26 – Eight neuron ANN - 10 input 52

List of Tables

Table 1 – Comparison table 37
Table 2 – Table of weights for three neurons. Input layer to hidden layer 45
Table 3 – Table of weights for three neurons. Hidden layer to Output layer 45
Table 4 – Results obtained from XOR function for three neurons 47
Table 5 – Table of weights for four neurons. Input layer to Hidden layer 47
Table 6 – Table of weights for four neurons. Hidden layer to Output layer 47
Table 7 – Results obtained from XOR function for four neurons 50
Table 8 – Table of weights for eight neurons. Input layer to Hidden layer 50
Table 9 – Table of weights for eight neurons. Hidden layer to Output layer 51
Table 10 – Results obtained from XOR function for eight neurons 53

List of abbreviations and acronyms

RNA	Rede Neural Artificial
SoC	System-on-Chip
FPGA	Field Programmable Gate Array
ARM	Advanced RISC Machine
RISC	Reduced Instruction Set Computer
ANN	Artificial Neural Network
AMBA	Advanced Microcontroller Bus Architecture
RTOS	Real-Time Operating System
HDL	Hardware Description Language
CPU	Central Processing Unit
RAM	Random Access Memory
DSP	Digital Signal Processing
PS	Programming System
ASIC	Application-Specific Integrated Circuit
MLP	Multi-Layer Perceptron

Contents

1 INTRODUCTION . . . 17
1.1 Objective . . . 18
1.2 Document outline . . . 19

2 FUNDAMENTALS . . . 20
2.1 Embedded Systems . . . 20
2.2 FPGAs . . . 21
2.3 Microprocessors . . . 21
2.4 ARM Architecture . . . 23
2.5 SoC . . . 24
2.6 Computational intelligence . . . 26
2.7 Artificial Neural Networks . . . 27
2.7.1 Life Cycle . . . 27
2.7.2 Artificial Neurons . . . 29
2.7.2.1 Activation Function . . . 29
2.7.2.2 Learning Methods . . . 31
2.7.3 ANN Architectures . . . 32
2.7.3.1 One-Layer ANNs . . . 33
2.7.3.2 Multilayer ANNs . . . 33
2.7.3.3 Recurrent networks . . . 34

3 STATE OF THE ART . . . 35

4 METHODOLOGY . . . 38
4.1 Materials . . . 38
4.2 Proposed System . . . 38
4.2.1 Programmable Logic . . . 39
4.2.1.1 Interface . . . 40
4.2.1.2 Control . . . 40
4.2.1.3 Data streaming . . . 40
4.3 Testing Methodology . . . 41
4.4 Implementation . . . 41
4.4.1 Process . . . 41
4.4.2 Hardware Neuron . . . 42

5 ANALYSIS OF RESULTS . . . 44

5.1 **Results** **44**
5.1.1 ANN topology with three hidden neurons 44
5.1.2 ANN topology with four hidden neurons 46
5.1.3 ANN topology with eight neurons 49
5.2 **Discussion** **52**
5.2.1 Discussion of results 52
5.2.2 Difficulties encountered 53
5.2.3 Limitation analysis 54
5.3 **Comparison between a hardware approach and a software approach** **54**

6 **CONCLUSION** **56**
6.1 **Future work** **57**

BIBLIOGRAPHY **59**

1 Introduction

Neural networks are a powerful tool widely used to solve computer vision problems, language processing, control, pattern recognition, among other type of applications, Haykin et al. (1999). It does so by emulating the working principle of biological neurons.

Despite being a powerful tool, its required computational power increases in a non-linear manner as the task's complexity increases. Thus making the processing of said complex applications to be almost exclusively performed by workstations, for real and non real-time applications, due to the limited computational capacity and lack of a proper power supply of embedded devices. Embedded devices that are capable of providing the necessary hardware requirements exist, but are not the norm in most applications.

Because of the limitation of most embedded devices, when a real-time field application requires the utilization of a powerful notebook/workstation, or a complex setup of workstations, and one or more simple embedded devices confided with the task of collecting physical data and transmitting it to the PC processing the ANN.

This project intends to solve the existent problems found in this type of applications. To do so it is necessary to understand how the entire system must function. Here's a list of requirements for field setups:

- Obtain data from the physical media;
- Convert the data into comprehensible information;
- Package the information inside a protocol, which must be designed according to the number of embedded devices and time and/or location restrictions for the main equipment to receive and decode;
- Decode the information and verify its validity;
- Process the main application and command the embedded devices, if necessary;

It is possible to notice that this type of system contains many moving parts, increasing the probability of failure, and taking a notebook to the field might be dangerous, since the risk of breaking is high. This work tries to find a solution for this type of systems, turning the traveling process more efficient and to decrease the likelihood of broken or lost expensive equipment, as well as providing a simpler and smaller solution which can replace them. It is important to notice that systems using the solution provided by this work will also be able to function in remote locations that have no way of communicating relevant data. This is the case for applications in geology, for which this project is intended for.

Considering the need of computational power, compactability, reduced power consumption and independence from notebooks/workstations, it was possible to arrive at the conclusion that this project must include the use of an FPGA for computational power and a microprocessor for the independence. A suitable SoC was found, which is capable of incorporating all the required specifications.

The selected SoC is the Zynq7000, which consists of two hard wired ARM processors in its core, surrounded by programmable logic area, all encapsulated inside the same IC. This reduces the space that would be required by one microprocessor and one FPGA, as well as providing a protected environment for the data exchange between both applications and less power consumption than two separate components.

1.1 Objective

The objective of the dissertation project is to provide a hardware neuron architecture that can be implemented in programmable logic. This neuron will then be validated by comparing the result generated by an ANN implemented in hardware with the results from the same ANN implemented in software.

In order to achieve the main objective, a few requirements need to be met, such as:

- build a multiply accumulate (MAC) unit;
- create a datapath for the input data;
- exchange data between the microprocessor core and the programmable logic;
- optimize the datapath so that the outputs can be generated faster, which often means pipelining the implementation;
- generalize the building method;
- create networks automatically;

The first three items listed above are the minimum requirements for having a working architecture. Each neuron, as presented in chapter 2.7.2, consists of a simple MAC unit. The datapath refers to how the information flows from the input of the neuron to its output and finally the data exchange refers to the control mechanism given by the software side of the project, which is presented in figure 15. The optimization of the datapath is necessary to have a valid output as fast as possible. As stated above, the way to implement this is to pipeline the module, which means that the functionality of the entire module is converted into simpler steps, so that the operations may occur simultaneously.

The last two items are required to facilitate the use of this project as a product, as well as facilitating the creation of different networks for each test case. The generalization is the first step before automating the creation of an entire network. For this it is necessary to comprehend what is necessary for the creation of an artificial neural network (ANN) in hardware and how to automate the process of creating such network.

1.2 Document outline

The document contains the following chapters: Fundamentals, that contains a brief explanation of the concepts being used in this document; State of the art, which describes the recent works in this field that are relevant to the subject; Methodology, where the method of implementation is described as well as the materials being used and the test method to validate the architecture presented; Analysis of results, that shows the results obtained in a table followed by the discussion of such results and Conclusion that gives a conclusion by comparing the objectives and the obtained results, as well as a suggestion of future works.

2 Fundamentals

The necessary concepts needed to understand this work are presented in this chapter in a succinct way. The concepts defined are: embedded systems and its components, and Artificial Intelligence.

2.1 Embedded Systems

An embedded system consists of the union of hardware and software, usually designed to accomplish a specialized task complying with real-time demands.

A wide variety of products fall into this category, such as:

- home automation;
- safety equipment;
- industrial automation;
- automotive systems;
- medical equipment;
- naval systems;
- avionics systems;
- military equipment;
- cellular phones.

There are several ways to achieve the real-time constraints necessary for such systems. One of them is with the use of Real-Time Operating Systems (RTOSs), which provide the developer with the ability of designing parallel tasks and thus, using the microprocessor's/microcontroller's full capabilities. The other method for meeting real-time constraints is bare-metal programming, in which the programmable device doesn't possess an operating system, along with its concepts and software components. In turn, different tasks are programmed in the device's interruptions. The selection between one of these falls to the system's complexity. Simpler systems may be programmed bare-metal, but more complex systems, that require hard timing constraints and a diversity of tasks to accomplish, then an RTOS is recommended.

In some cases, hard time constraints cannot be achieved with the use of just microprocessors, needing the use of an FPGA to treat such information. These systems that require FPGAs are usually ones where the device must communicate with a high speed bus, or the amount of data that must be processed takes too long in something other than an FPGA. There is always the possibility of implementing the neuron in an ASIC format, however the high cost of ASICs are a disadvantage as well as the lack of flexibility compared to FPGAs.

2.2 FPGAs

FPGAs (Field-Programmable Gate Array) are programmable logic devices that are based on configurable logic blocks connected through programmable interconnections. This type of devices are programmed in Hardware Description Languages (HDLs).

The main characteristic of FPGAs are their ability to reconfigure their entire hardware and their ability to execute in parallel, unlike microcontrollers and microprocessors, which use a software-based parallelism.

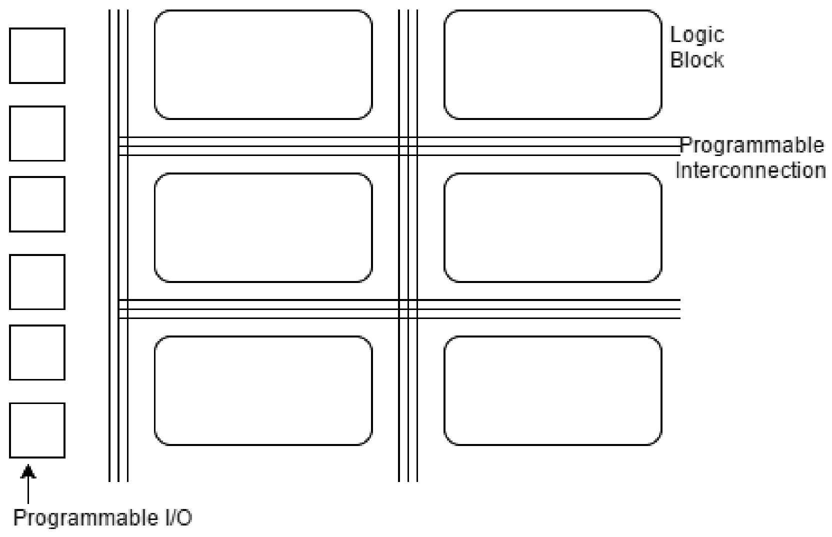
In order to generate a different hardware, FPGA vendors also provide a software in charge of interpreting the HDL typed, and knowing the IC's internal hardware, it executes and Artificial Intelligence process, which is responsible for generating the desired hardware for the desired application. It is important to note that these vendors do not possess open development platforms or a common file type or hardware architecture, each vendor is responsible for its own hardware synthetization and generation.

Figure 1 shows the basic structure for an FPGA. In it, there are three basic parts of an FPGA, the logic block, programmable $Input/Output(I/O)$ block and programmable interconnections . The logic block consists of the basic logic elements, such as combinational logic, flip-flops, etc, and in modern FPGAs it is possible to find block with dedicated functionalities, such as DSP blocks and dedicated memory blocks. The programmable interconnection connects the logic blocks and the programmable I/O, according to the description generated in order to obtain the described functionality. And finally the programmable I/O blocks connect the FPGA to the external design.

2.3 Microprocessors

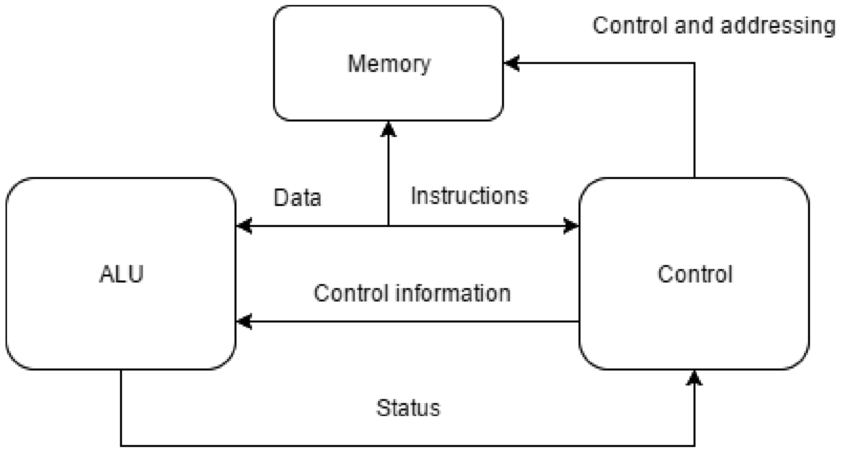
A microprocessor consists of a hardware architecture, which, basically, performs at least three simple tasks in its pipeline: find instruction in memory, fetch instruction to the CPU and executing the instruction there. The more complex architectures require larger pipelines in order to increase the processor's clock frequency and processing efficiency.

Figure 1 – Architecture of an FPGA



The CPU contains decoding blocks that read the instruction, decode which type of instruction it is supposed to execute and then execute it. Whenever illegal, or undefined instructions are decoded, the CPU is responsible for signaling the system. This decoder is essential because of the existence of several architectures with different kinds of permitted operations and different operating codes (refered as "opcodes"). An example is given in figure 2. In this figure it is possible to see three main blocks, the memory, control and Arithmetic Logic Unit (ALU). The memory unit contains the code stored for the microprocessor (instructions in the figure) to run and also contains the data used by the program, where they are distributed to the control and ALU respectively via the control and addressing signal originated from the control unit. In the control unit is where the decoding of the instructions is performed, which then sends control signals to the ALU so that it can perform the desired operation and finally the ALU signals the control unit regarding the executed operation and outputs the signal to the system.

Figure 2 – Microprocessor Architecture



Basic micorprocessor architecture

As aforementioned, several architectures exist, each with their own specific characteristics, opcodes and behavior. A few examples are: PowerPC, x86, ARM, among others. The ones cited are the most commonly used for high performance systems, being PowerPC the one used for older embedded systems and ARMs for newer embedded systems. The next chapter describes the ARM architecture briefly, as it is the processor's architecture contained in the SoC.

2.4 ARM Architecture

The vast majority of the embedded devices market consist of ARM microprocessors and microcontrollers, especially the ones that require more elegant and simple solutions with better power consumption and performance. Other architectures used are 8051 (old and gradually fading away), PIC (very robust microcontrollers with limited processing power) and PowerPC (architecture with high power consumption).

Some of the best known families of ARM microprocessor and microcontrollers are:

- ARM7

  First popular ARM microcontroller. Despite being quite old it is still very popular due to its low power consumption and decent processing power, which make them ideal for simple embedded devices.

- ARM9

  First general applications ARM microprocessor. Its pipeline was modified and incrementd and also complex hardware features, such as cache memory, were added to this family, permitting the use of complex operating systems (Linux distributions).

- ARM11

  Second family of ARM's general applications microprocessor. It's architecture is similar to the ARM9's except that it contains one more stage in its pipeline, allowing for higher clock frequencies.

- Cortex-M

  New generation of ARM microcontroller's. They were inserted in the market with the purpose of supersede old ARM7 microcontrollers. They consist of an architecture similar to ARM7, allowing for easy ports to another hardware platform, which is highly recommended since this family has a much better performance in comparison to old ARM7s, and they are also smaller in footprint and require less power.

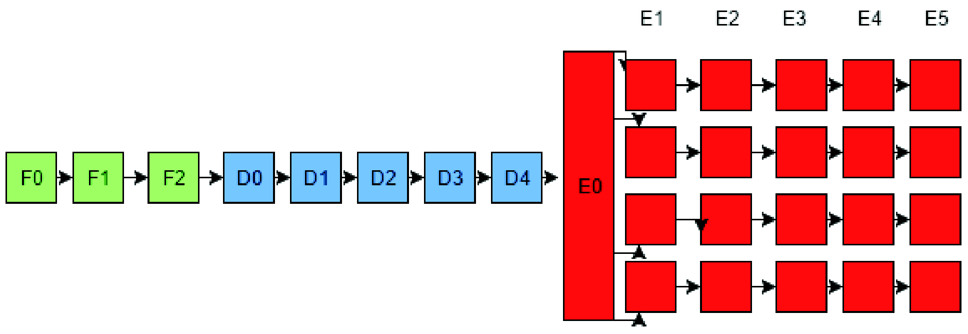
- Cortex-R

New generation of real-time microcontrollers. It possesses an architecture similar to higher performance ARM devices, but, as it demands real-time constraints, cache memories were exchanged for high speed RAMs located near its core.

- Cortex-A

  New generation of general applications ARM microprocessor. They possess a 13 stage pipeline, as seen in figure 3, and possess very high performance. They are very used in the manufacturing of cellphones. In the figure it is possible to see all the stages of the pipeline, the stages F0, F1 and F2 are the instruction fetch stages of the pipeline (stage F0 is the starting point, this stage is where the virtual memory is accessed and is not counted as the first stage). The following stages ranging from D0 to D4 are the instruction decoding stages. And finally stages ranging from E0 to E5 are the execution stages. As it is possible to see there is one common block, called the Architectural Register File that consist of the architecture's registers and then the execution follows one of four lines, depending on which instruction is being executed, represented as four lines of connected red blocks. One of the lines is dedicated to instructions being executed in the microprocessor's Vector Floating-Point Unit (VFP) or the NEON co-processor, the other one is called an ALU multiply pipeline, the third one is the ALU pipeline and finally there is the *load/store* pipeline, which is where the microprocessor executes the load and store instructions.

Figure 3 – ARM A8/A9 pipeline

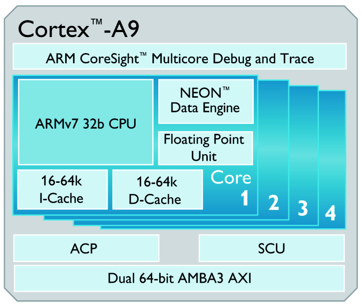


The selected SoC consists of two hard wired ARM Cortex-A9 processor.

2.5 SoC

The previous sections containing the brief explanation of FPGAs, microprocessors and more specifically ARM microprocessors are there in order to give a brief preview of the features of the selected SoC, which consists of two hard wired ARM microprocessors surrounded by programmable logic. Its architecture can be seen in 5. In the figure it is possible to see two main distinctive blocks, the Programming system(PS) block that

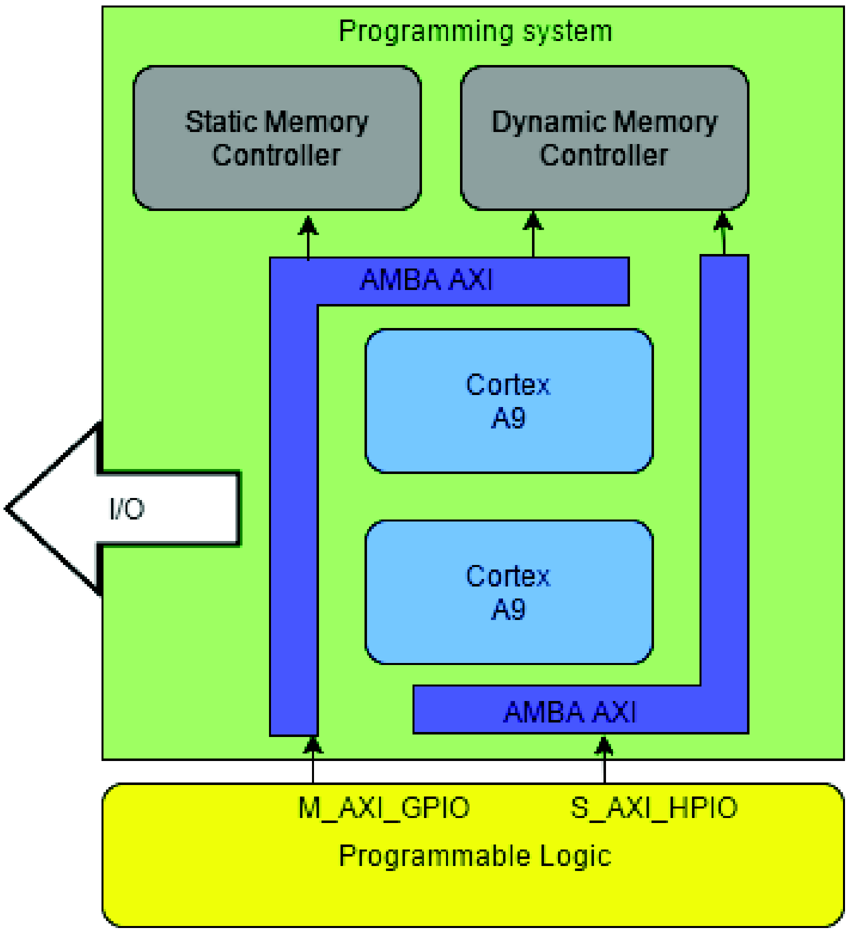
Figure 4 – ARM Cortex-A9 Architecture



Source: ARM

contains two hard wired ARM microprocessors and the Programmable Logic(PL) block area, which allows for reconfigurable hardware. The feature that excels this SoC from others, was Xilinx's ability to integrate the ARM processor's interface bus (AMBA AXI in the figure) into the programmable logic, allowing for high performance data exchange and processing. It is possible to see that the AMBA AXI is divided into two different blocks in the figure. One of them is used to create AXI master modules in the PL side of the SoC, which allows it access to the system's peripherals and the other one is used to create AXI slave modules, which allows for high performance communication.

Figure 5 – Zynq7000 Architecture



2.6 Computational intelligence

It's a field in Computer Science with the end of solving problems of increasing complexity. It does so by the modeling of biological(evolution, neural systems, etc) systems and natural intelligence, resulting in intelligent systems and intelligent algorithms, Engelbrecht (2007). Most techniques utilized in computational paradigms fall into one of two paradigms: the traditional symbolic paradigm and the connectionist paradigm, (SUN, 2000).

The symbolic paradigm focuses on the high-level symbolic representation of problems and was the main paradigm until the 1980's. The main technique used in symbolic systems are expert systems. These technique is similar to an if-else structure where its main objective is to mimic the decision making ability of humans.

The connectivist paradigm is an approach that models mental phenomena as interconnected networks. Its most prominent technique is Artificial Neural Networks. There are several techniques to achieve human-like intelligence and their use depend highly on which type of application is intended. Among the several Computational Intelligence techniques it is possible to highlight:

- Decision theory

  Belongs to the set of techniques of the symbolic paradigm. Hansson (1994) states that almost everything that a human being does involves decisions. Therefore, to theorize about decisions is almost the same as to theorize about human activities. However decision theory is a more specific technique. It needs choices and possible end results in order to reach a decision, which means that all decisions made using this technique are a result of logic and not random decisions. These choices and possible end results are known as the variables and uncertainties of a system, and it is necessary to define them and find the relation between them in order to obtain a proper model of the system and reach a logical conclusion.

- Tabu search

  Metaheuristic method used for mathematical optimization. It uses an iterative method to get to and optimized solution. It has been used in classical and practical problems in applications ranging from scheduling to telecommunications and from character recognition to neural networks, Glover (1990). Given a problem to be optimized, tabus search proceeds interactively from one solution to another until a chosen termination criterion is reached, Glover (1995).

- Genetic Algorithms

  This method tries to mimic the natural selection process. There is an initial population with different random characteristics (different data points) and an environment (in some cases applications), with which the members of the population interact (the data points are used to execute the application). Just like the evolutionary process, the best results are selected for each generation (iteration), removing the least adequate and generating new data points, blending the characteristics or the surviving members and adding a random mutation probability for the system to improve, or not, creating different characteristics. This process goes on until the desired result, or the best possible result is attained.

- Artificial Neural Networks (ANNs)

  This technique tries to imitate the way in which neurons and synapses operate in the central nervous system and is mainly used for pattern recognition problems. They possess the ability to learn by an iterative process in which it is given an input and its internal components are modified until the ANN is capable of recognizing the new data.

  The main components of an ANN are the neurons and synapses. Neurons consist of activation functions, which denote the neuron's behavior and the synapses consist of weights to which inputs are multiplied and inserted in the following neuron.

  An important component of ANNs is the learning method. Each different type of ANN, requires a different learning method, with which the new results are obtained.

  Failing to use an adequate learning method, adequate ANN architecture or neural connections are very likely to guide the problem in a wrong direction, and an adequate result could never be achieved sometimes.

  Section 2.7 defines ANNs in a more concise manner since it is the selected method for implementation in this project.

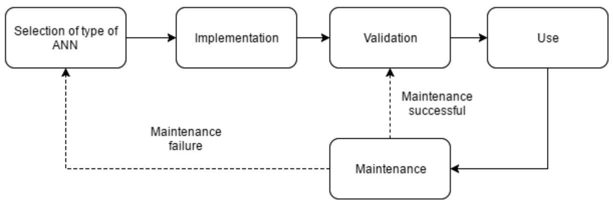
2.7 Artificial Neural Networks

An Artificial Neural Network is a computational model that mimics the biological functions of neural networks (HAYKIN et al., 1999). This model is highly parallelizable and its structure changes via learning algorithms, allowing it to be able to understand the problem at hand in a more profound manner.

2.7.1 Life Cycle

The life cycle of any ANN can be seen in figure 6. In this figure it is possible to see the generic life cycle of any ANN. It consists of five main moments: the selection, the implementation, the validation, the use and the maintenance. The selection moment is when the problem at hand is formalized and a type of ANN is chosen to be implemented to solve the problem. The next step, implementation, consists of developing the desired topology for the problem at hand and the creation of all the necessary data for future use. Then comes the validation, where the ANN is deemed reliable to solve the problem at hand or the process needs to restart. The use moment consists of the utilization of the developed ANN and finally there is the maintenance moment. This moment must occur periodically, and in it, the problem at hand is re-evaluated, and the results obtained are analyzed. At that moment, if the maintenance is considered successful, it will go to the validation moment indicating that the ANN developed is capable of solving the problem, and if the maintenance is considered a failure the process must be reinitialized or re-implemented, depending on the results of the maintenance. A reinitialization of the process indicates that the type of ANN is deemed unreliable to solve the problem or not the best available option, and a re-implementation indicates that the ANN must have its topology redesigned or it simply must be retrained with more data sets.

Figure 6 – Generic Neuron Life cycle



The life cycle of a Multi-Layer Perceptron(MLP) can be seen in figure 7. The life cycle of an MLP is being displayed due to the fact that this is the type of ANN that will be implemented in this project. Each cycle state consists of the following activities.

- Training

  This process receives two inputs, the ANN's output and the desired outputs. It then proceeds to modify the weights through an iterative process until a certain threshold is reached.

- Validation

  The process is deemed as validated when some test inputs are given to the ANN and it obtains the expected result. This means that this specific neural network is ready for production.

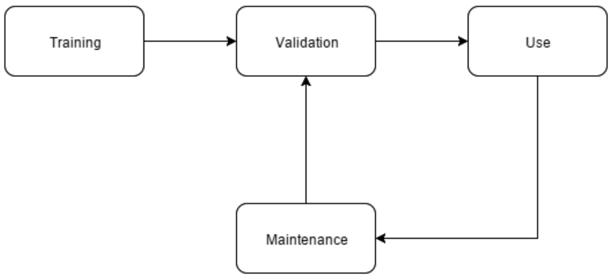
- Use

  At this moment the ANN is executed indefinitely until the need for retraining arises or it is deemed as not fit for the problem at hand.

- Maintenance

  When the production phase ends for any of the aforementioned situations, then the ANN reaches this state, where a decision is to be made regarding the proceeding activities for the ANN. At this point, it either decides that the ANN is not capable of solving the problem at hand and is then redesigned, or it forces the ANN to go into a new training process.

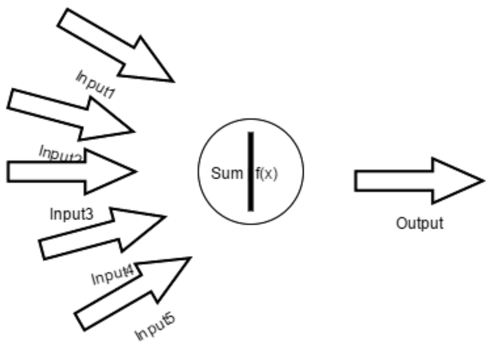
Figure 7 – Neuron Life cycle



2.7.2 Artificial Neurons

Artificial Neurons are mathematical models that try to replicate the function of biological neurons. A neuron receives any given number of inputs, which are multiplied by given weights for each input, then performs a sum of all the received values and finally inputs the resulting value into the activation function (described in section 2.7.2.1) in order to obtain the neuron's activation value. See figure 8 for better understanding.

Figure 8 – Neuron



2.7.2.1 Activation Function

The activation function consists of a function that receives an input, and then calculates an output which indicates whether this neuron is currently active or not for the given input data set.

Some of the existent activation functions are:

– Step function

It’s the simplest of them all. Whenever the function’s input exceeds a threshold value, the function’s output is ’1’, and when it doesn’t, its output is ’0’.

$$f(x) = \begin{cases} 1, & \text{if x >= 0} \\ 0, & \text{if x < 0} \end{cases} \tag{2.1}$$

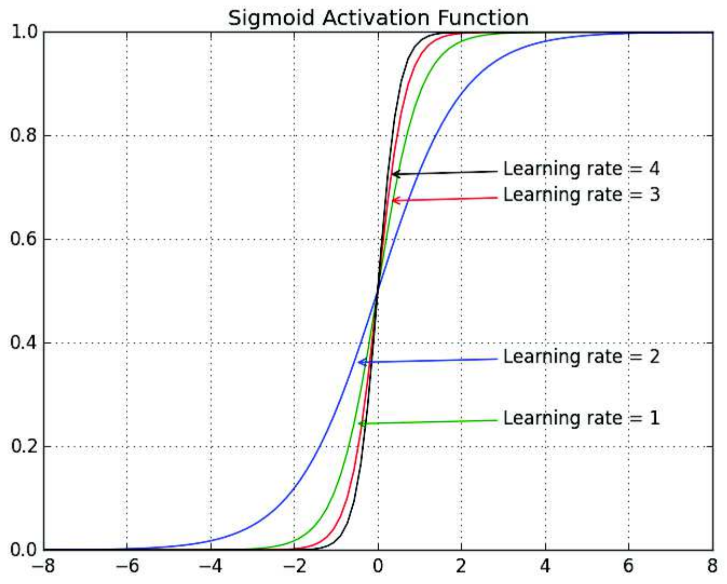
– Sigmoid function

This function is given by the following equation:

$$f(x) = \frac{1}{1 + e^{\beta \cdot x}} \tag{2.2}$$

Where β is the slope parameter. An example is given in figure 9. In the figure it is possible to see four different sigmoid functions, each with a different learning rate.

Figure 9 – Sigmoid activation function



The sigmoid function resembles a lot the activation of biological neurons and it is fairly used not only for its success(meaning that there is a high chance that the implemented architecture in solving the problem at hand), but also because its easiness to calculate its derivative, which is useful for some learning methods.

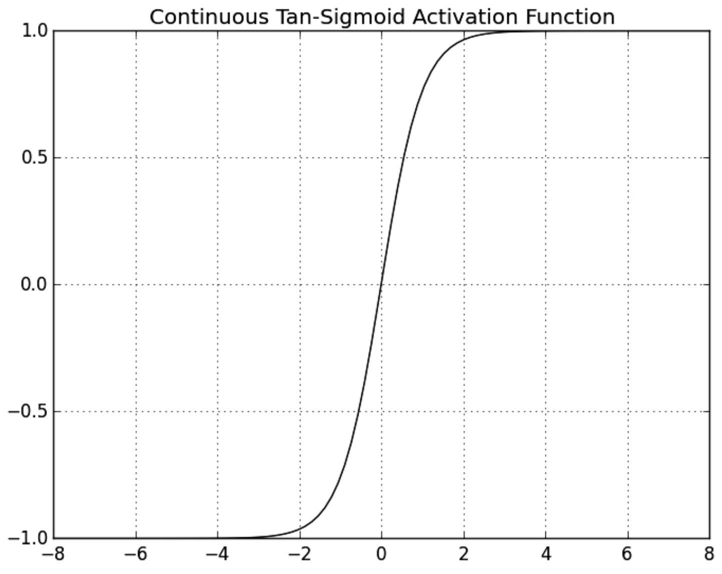
– Continuous Tan-Sigmoid Function

It is also highly used for its success rate. Its equation is:

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

Its waveform can be seen in figure 10

Figure 10 – Continuous Tan-Sigmoid Function



2.7.2.2 Learning Methods

Each input data set is given an initial random weight (preferably between zero and one, because high initial weights tend to diverge). After the input flows through the entire ANN, a result is obtained, but sometimes said result is not the one expected. Then, a learning algorithm is used to modify the initial weights, until an expected output is reached, within an acceptable uncertainty range.

There are two ways for an ANN to learn. It can do it with a supervised method or an unsupervised method. The supervised method consists of a set of inputs with its corresponding outputs and the unsupervised methods consist of forming natural groups or cluster of patterns, Sumathi (2010). Some of the learning methods are:

– Error back-propagation

  Compares the system's output with a desired value, and then uses that error to guide the training process. This value is then used to adjust the weights. The error is given by:

$$error = expected\_value - output\_value \tag{2.4}$$

– Memory based

  The inputs and outputs are stored in memory. Thus learning method requires two parts, Sumathi (2010): criterion for defining the local neighborhood of the test vector and learning rule applied to the training in the local neighborhood.

  Those outputs are then used as past when the process is executed again, obtaining a new set of weights. The gradient descent is an example of such method.

- Hebbian

  Developed by Donald Hebb in the 1940's. It consists of a set of 2 rules:

  * The weight between two neurons increases if the neurons at the ends of the synapse are activated simultaneously;
  * The weight between two neurons decreases if the neurons at the synapse's ends are not activated simultaneously;

  Hebb's Law is expressed as:

$$\delta W_(ij) = \mu \dot{a}_i \dot{a}_j \tag{2.5}$$

  The equation calculates the needed change (delta) in weights from the connection from neuron i to neuron j and μ represents the learning rate, Sumathi (2010).

- Competitive

  This type of law was inspired in biological systems. The units contained in the system compete with one for the opportunity to learn and update its weights. This process goes on until one of the units of the system reaches the largest output and is considered the winner, allowing the winner to inhibit competitors and excite neighbors, Sumathi (2010). This type of learning leads to specialized neurons.

- Boltzmann

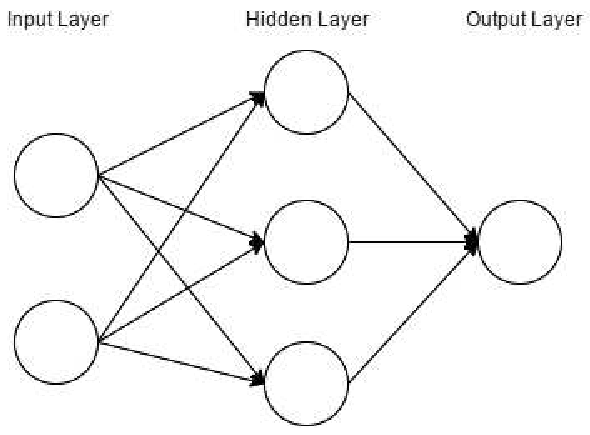
  Statistical method. It is similar to the error-correction method and it is used in supervised learning. In this algorithm, the state of each individual neuron, in addition to the system output, are taken into account. In this respect, the Boltzmann learning rule is significantly slower than the error-correction learning rule. Neural networks that use Boltzmann learning are called Boltzmann machines (ANN. . . ,).

2.7.3 ANN Architectures

There exist three basic structures for ANNs, and each one of them is highly linked to the selected learning method chosen Haykin et al. (1999). One example of ANN architecture is given in figure 11. In it a Multi-Layer Perceptron is shown. This architecture was chosen due to the fact that it contains all the layers that the other architectures use. A better explanation is given in 2.7.3.2.

The input layer receives the input dataset and routes them, along with their weights to the hidden layer. The hidden layer then proceeds to execute the neuron's behaviour with all its components previously detailed, and its generated output is given to a new hidden layer or the output layer, which then indicates which output neurons are active.

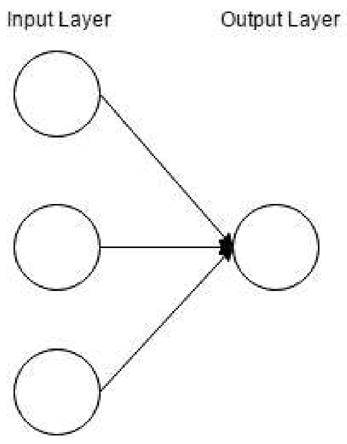
Figure 11 – Layers of an ANN



2.7.3.1 One-Layer ANNs

The simplest of all the architectures available. The input neurons are connected to the output neurons, and the behavior of the ANN is given by the output neurons. One example of this architecture can be seen in figure 12.

Figure 12 – One layer ANN



2.7.3.2 Multilayer ANNs

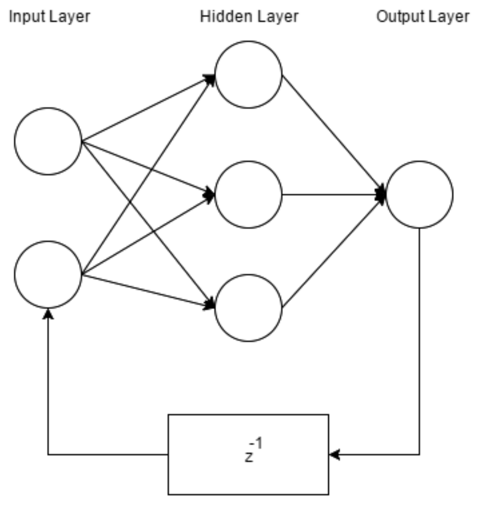
Like the One-Layer ANNs, it has input layer and output layer with a given number of hidden layers, where the ANN's behavior is given. An example can be seen in figure 11. In this type of ANN architecture, the data flows from the input layer, through all the hidden layers and finally reach the output layer. The neurons in the input layer are responsible for passing the input data to the hidden layer, and during the flow of data, the inputs are multiplied by their respective weights and all the results are summed in each connected neuron in the hidden layer. Following this, each neuron is responsible for applying an activation function to the summed data and passes it to the next layer. The next layer, may be another hidden layer or an output layer. If the data flows from one hidden layer to another, the previous step is performed repeatedly until it reaches the output layer. Finally, when the output layer is reached, all the outputs from the hidden layer are passed to the output layer with a weighted connection, and the neuron in the output layer is responsible for summing all the aforementioned multiplications and applying the result to another activation function, which will give the network's final output. This entire process is called the forward propagation, which is the flow of data from its input to its output, and the backpropagation is the usual learning method used for this type of architecture.

Due to its high neural connectivity, it is ideal for networks with a high number of input data sets and it's also important to note that not all nodes must be interconnected like figure 11, depending of the application, some nodes are better left with no connection.

2.7.3.3 Recurrent networks

The input to the system receives a feedback of its output, which causes a dynamic response to the application. These networks are better suited for systems where random inputs are an occurrence. Figure 13 depicts a recurrent network. Recurrent networks possess a feedback connection, where the previous state of the hidden layer is stored, represented by z^-1 in the figure, that allows it to learn temporal characteristics of the data set, **??**.

Figure 13 – Recurring ANN



3 State of the art

In this chapter, some projects that had the objective of developing an ANN in hardware are presented in chronological order. There are a few different architectures described.

The project developed by Gadea et al. (2000), a sistolic multi-layer arrange of perceptrons was developed, containing 3 layers in a Viertx XCV400 FPGA with a learning algorithm designed to be executed in hardware. In order to make good use of the parallelism property of the FPGA, a backpropagation algorithm with multiple layers of pipeline. The parallelism of an FPGA is what lead to the decision of using the backpropagation algorithm, since it allows the ANN to do the forward propagation and the backpropagation simultaneously. In this article they chose to save the weights of the synapses in RAM memories contained inside the FPGA and then the results of a pipelined and non pipelined were used for comparison. This lead to a promising result with good performance, despite not showing tests with more complex ANNs. This ANN, unlike the developed, provides an entire hardware environment for the training and execution of the ANN. This reduces the implementation's flexibility, since it is necessary to reconfigure the entire hardware each time a new ANN needs to be implemented. the ANN developed in this this thesis seeks to improve on the flexibility, building the system in a way that only the forward propagation of the ANN must be reconfigured.

The project developed by Venkatesan e Balamurugan (2001) had as an objective the creation of a real-time failure detection problem for distance protection. It presented a short training time implemented in FPGA. The objective of the implementation of the FPGA ANN was to obtain high processing speeds. In order to test the implementation, first a software ANN was created and then a hardware ANN was created and their outputs were compared. It was possible to generate and output in one third of the time that took the software to process the problem. And similar to previously mentioned articles, it lacks flexibility and a high performance communication exchange.

In the article by Coric, Latinovic e Pavasovic (2001) a neuron was generated with the purpose of implementing it in a backpropagation ANN. Six types of neurons were implemented, however, only one of them was described, the 8 bit neuron with a Booth's sequential multiplier, which was chosen due to its speed. In this, the inputs are stored in a register, two registers are used to store the partial products and the input's weight. As any complex system, there is also a control unit in charge of the data flow. This control unit has two three-bit counters, one of them used for the Booth multiplier and the other is used to codify the even coefficient numbers of the inputs. The used FPGA in this project is very outdated, which was a limiter for what could be done. The project being presented makes use of a modern SoC, which not only allows for more flexibility, it also provides better hardware resources.

On the other hand, the project developed by Fascicola Tom et al. (2004) had as an objective, the development of an ANN application, however, it was created by using Xilinx's System Generator tool in order to simplify the development process. The architecture chosen by the developers and the Simulink neuron model (both figure in Fascicola Tom et al. (2004)) are similar to the proposed model in the project being presented. Despite being similar to the one being presented, it differs on the component selection, which allows for much higher communication rates, less on-board space, and it uses a simple microcontroller communication with the FPGA, which doesn't allow for the use of an operating system, such as linux, to help develop more advanced applications, such as a user interface, a database with all the known ANNs with their training data and other types of applications that would turn it into a highly flexible product.

Tsai e Fu (2009) opted to use an architecture that makes use of multiple data stream of toroidal series to process backpropagation RNA in hardware. To implemented the ANN in hardware it is necessary a well planned structure in order to maintain its control. For such, it was developed an architecture containing: a control unit, activation function unit, forward propagation and the backpropagation unit. The control unit is responsible for generating signals at the proper moment to guide the data flow through the ANN. The activation function unit contains the activation function per se, in order to generate an output in accordance to a Look-up table that was created with interpolation. Finally the forward propagation unit is responsible for generating an output dependent of the inputs and the backpropagation unit is used to process the system's learning process. This hardware implementation was capable of finishing the processing of the ANN in half the time that the same problem took the software implementation. This type of project, as stated, needs a well designed control unit, that could be simplified with the use of software, which is one of the objectives of the project being presented.

Pinjare e M (2012) in his article had the objective of developing a trainable ANN in FPGA, differently from Fascicola Tom et al. (2004), where the training process occurred in a PC in an algorithm developed in Matlab. In this project the backpropagation algorithm was used to train logic gates as well as image compression. To validate the training algorithm the results were compared to the results obtained from a Matlab program. For the development of the training for the logic gates a hyperbolic activation function was used due to the fact that it was the one that had the best adaptation to this type of problem and in order to verify the result, a simple ANN was developed which generated an erroneous output previous to its training and it was possible to verify that the output obtained after training was correct. For the image compression problem, a similar process was developed. From this article it is possible to notice that putting the training algorithm in the FPGA has a great improvement in the speed of the training, as well as the processing of the ANN. Like the article developed by GADEA et al., it lacks the flexibility that software provides to these type of systems.

As it is possible to see, all the presented articles, despite presenting a similar idea(the development of a hardware ANN) were either limited to the time's resources or limited to the idea of an ANN developed entirely in hardware, that limits the system's flexibility, which is the presented project's main difference, along with the high speed communication between microprocessor and FPGA, which is allowed by the use of the selected SoC. It was possible to notice the lack of more modern articles in the subject, which indicates that it was thought that hardware ANNs were possible to implement, so other type of approach are being given to this type of projects, such as focusing solely on the study of the data resolution needed to implement working hardware ANNs or focusing on the implementation of activation functions and the creation of mathematical approximations that can be implemented in hardware in the fastest way possible and using the least amount of resources. The project presented intends to focus on the implementation of the hardware ANN because of the selected device for implementation, since this is a newly available resource that brings a different approach to the implementation. In table 10 it is possible to see the comparison between all the projects.

Table 1 – Comparison table

Author	**Hardware**	**Software**	**Trainable**	**Data exchange medium**
Gadera	Y	N	Y	FPGA
Fasciola	Y	Y	N	Communication Interface
Pinjare	Y	N	Y	FPGA
Tsai	Y	N	Y	FPGA
Venkatsean	Y	N	Y	FPGA
Coric	Y	N	Y	FPGA
Pablo	Y	Y	N	FPGA - AXI

4 Methodology

In order to achieve the objectives, a development method is needed, as well as the proper materials for it.

This chapter is divided into three mains parts: material's description, the development methodology and the procedure to reach the goals set by the development method.

4.1 Materials

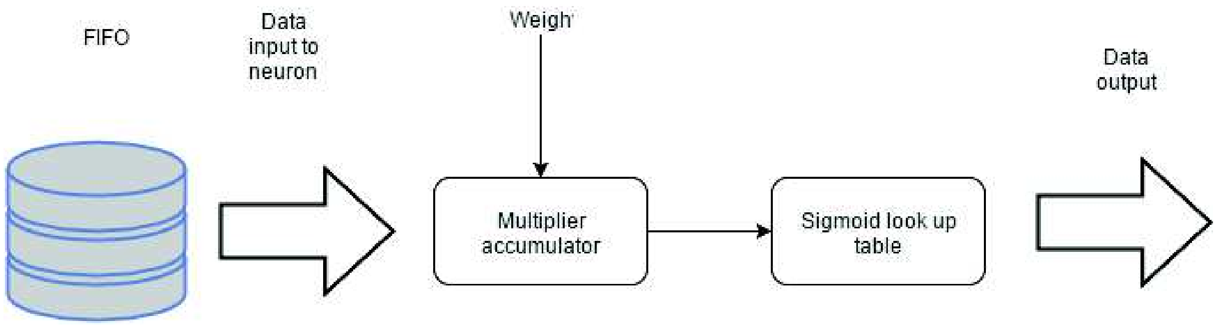
This section describes the physical and virtual materials to be used in the development of this project.

The Zedboard Development board was selected, due to its hardware description, which is suitable for this project and the Development IDE (Vivado) is given by Xilinx, along with the purchase of said development board.

4.2 Proposed System

According to the objectives set in the introduction, the proposed system must be modular to avoid component dependence (since the selected SoC could be replaced with a microprocessor-FPGA setup), as any development process requires, effective and fast enough to be utilizable without the intervention of a workstation. In this project the definition of utilizable is given by the quadratic error which must be inferior to 0.01, and a reduced number of cycles for execution. For such a MLP was chose to be implemented, due to the fact that it possesses a high success rate of execution. The proposed system can be seen in 14.

Figure 14 – Neuron Architecture



To reach the modularity objective three main development blocks were devised: Programmable Logic, Interface, Control and Data streaming. These block are interchangeable with new block as the need of a different hardware arises.

4.2.1 Programmable Logic

The ANN model chosen to be implemented in this project is the Multi-Layer Perceptron (MLP) due to the fact that it has the highest success rate among the existing ANN models according to Haykin et al. (1999), as well as the sigmoid activation function.

Due to the parallelizable nature of ANNs, it is placed in this block. The programmable logic block will be implemented by the programmable logic block in the SoC, but it can be replaced by any FPGA that has enough hardware features for the application.

The neuron consists of five different hardware functionalities.

- Data Input Feed

  This block is responsible for receiving the input data for the reset of the neuron to execute the ANN. For this it needs an interface to exchange data with the Interface block, a small storing block and a control method to not overflow the neuron and loose data or starve the process.

- Multiplier (Synapse)

  Multiplies the input with its corresponding weight, which are all readily sent by the microprocessor. These multiplications are to be implemented using the available hardware multipliers in the SoC.

- Sum

  All results of the previous multiplications must be summed by receiving a signal from the previous layer indicating that data is available for processing. In practice this functionality and the previous one will be implemented by one MAC (Multiplier Accumulator, generated by Vivado using the hardware's available resources).

- Activation Function

  Hardware multipliers are a precious resource in the selected hardware due to its scarcity. This specific SoC contains 220 18x25 multipliers, allowing for a maximum of 220 neurons, given the desired precision or the desired type of data (fixed point or floating point). In case of higher precision or floating point variables the amount of hardware multipliers for just one multiplication function can vary from 1 to 5, thus reducing the number of neurons that can be implemented. For this reason, the activation function will be generated by a script and converted into a table for the hardware to choose its output value given the input from the previous module. The script will generate 7 different sigmoid tables ranging from 5 bit resolution to 11 bit resolution.

- Output

  Here a signal will be given, stating whether the neuron was activated or not.

A control method, given by the microprocessor, must be implemented to avoid unwanted results.

4.2.1.1 Interface

This is the unit by which the input data is sent to the ANN. The input data consists of the neuron's weights, the data sets, and the control info from and to the PS.

4.2.1.2 Control

The third building block is a control module for the artificial neural network. This control is given by software, via memory mapped registers. These memory mapped registers are implemented using ARM's AXI interface (provided by Xilinx), in which 4 registers were implemented:

- **Control register**

  Is a register with the number of data sets needed, one bit for starting the execution of the neural network and one bit indicating the end of the neural network calculations.

- **Dataset Address**

  Indicates where the data sets are stored in memory

- **Weight Address**

  Indicates the address from where the neurons get their weights

- **Result address**

  Indicates the address where the results will be stored after execution

All this control is done by software, which is one of the greatest advantages of using the selected SoC.

4.2.1.3 Data streaming

This building block is responsible for transporting the data sets to the input FIFO to the neurons in the hidden layer. This block must signal the neuron while the data set is being transfered to the next block. After the transfer is done, it signals the next block so that it can clear its contents for the next set of data sets.

For this module to know whether the next block is done with the calculations, it must receive a signal indicating that it's empty and ready for more calculations.

4.3 Testing Methodology

Three different networks are implemented with a different number of neurons. All networks possess the same number of input neurons and the same number of output neurons and one hidden layer where the number of hidden neurons are varied, and all the neurons in the networks are implemented without bias. The reason for varying the number of hidden neurons in the system is to have different number of arithmetic operations in the system to test how it responds to them.

These networks are implemented in Python using the pybrain framework, and its resulting weights are used for the hardware implementation in order to validate its output.

The validation of all the hardware neural networks' output is given by a comparison with the expected result given by the expected result. In order to achieve a better comparison, the quadratic error generated by all outputs will be compared to the maximum allowed error of 0.01. For better comparison of results, several sigmoid quantizations will be tested, from five bits to eleven bits of resolution for the sigmoid.

The given test methodology is expected to give a better understanding of the system's behavior with respect to the sigmoid's resolution.

4.4 Implementation

This section describes the implementation process of the hardware implemented neuron and the neuron per se.

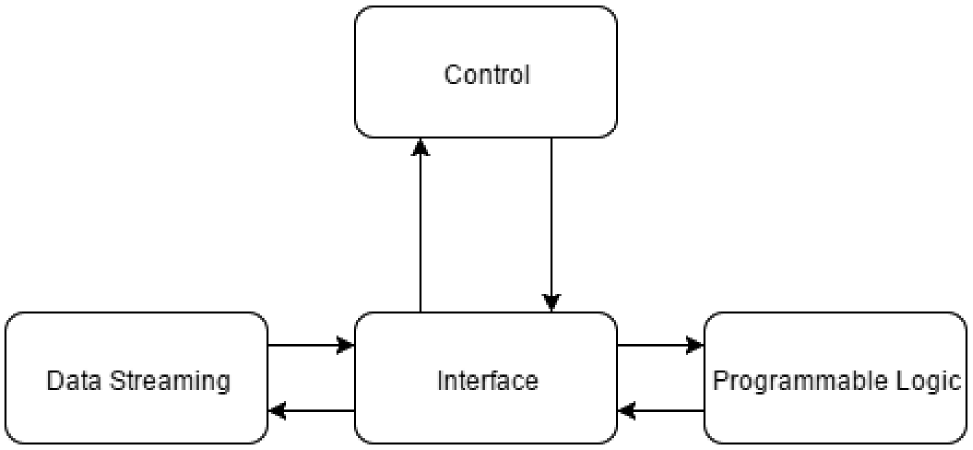
4.4.1 Process

The first initial step before starting the implementation is the idealization of the neuron scheme and the creation of a validation test.

Prior to the initialization of the building block diagram creation, it is important to notice that a neuron consists of a set of inputs that are then multiplied by a weight, added and then passed through a sigmoid function, as depicted in figure 14. Knowing the working process of a neuron it is possible to transform those functionalities into resources or modules that are contained within the SoC or must be implemented in order to gain its functionalities.

With this it is possible to conclude that the neuron must be implemented with four main blocks, which can be seen in figure 15.

Figure 15 – Main building blocks



4.4.2 Hardware Neuron

The neuron consists of four basic parts:

- The input FIFO;
- The gain multiplication phase;
- The sigmoid function;
- The output;

The neuron's input consists of a buffer where the input data enters the system. This submodule consists of a a data input system, a FIFO (First-in First-out) queue, queue control and data output system. The data input system collects the data being put into the neuron and stores it in the FIFO. The queue control consists of a circular queue, where the last item stored and the next item to be read are the control points. Finally the output system consists of a system similar to the data input system, with the difference that this module gets data from the FIFO and forwards it to the next module.

After the data input system comes the gain multiplication phase. This module consists of a MAC (multiplier accumulator). This module consists of a hardware multiply and sum module using the SoC's DSP48E1 DSP slices. The DSP slices consist of hardware implemented math modules that are present in the SoC. This module is responsible for receiving the input data and then multiplies it with the neuron's gain and storing all multiplications in a sum and sending it to the next module.

The next module consists of the sigmoid function. The sigmoid function consists of a look up table. In order to obtain this table, a script was implemented to quantize the sigmoid function, which was then used to fill the look up table. The result is then sent to the next module.

The next module consists of a scheme where it receives output data from the look up table and then presents it to the next stage in the Artificial Neural Network.

For simplicity, the neuron was implemented with a fixed point architecture where the most significant bit is the sign and the remnant 7 bits represent the the decimal values of the calculation in the case of the sigmoid output. For the input datasets and the weights 3 to 4 bits were used for the integer part of the number and 5 bits were used for the fractional part of the number.

In order to test the architecture of the neuron, it was decided that an Artificial Neural Network will be created with the purpose of emulating a xor gate. Since it is a nonlinear function, if it works properly, the neuron can be validated. For test purposes the neuron's weights will be calculated in a script and finally the results will be compared.

5 Analysis of results

This chapter contains the results obtained from the implemented ANNs. Here the outputs from a script will be compared to the results obtained with the Hardware Artificial Neural Network.

5.1 Results

In this section the results obtained from the Hardware Artificial Neural Network are compared to the results obtained from the one implemented in software. The results can be seen in the following figures and tables, each neural network topology is given given a subsection for better understanding of the resulting statistics.

5.1.1 ANN topology with three hidden neurons

This subsection contains the results for the neural network topology with three neurons in the hidden layer. The following figures and tables in this subsection show the expected results obtained with the script, the obtained results from the hardware neural network, the sigmoid resolution with its different outputs.

The bit resolution for the weights and data set inputs consists of four bits for the integer part and three bits for the fractional part. It was implemented this way, after several attempted implementations with four bits representing the fractional part of the weights, where the end result didn't have the expected result.

The weights from the input layer to the hidden layer are given in table 2, the weights from the hidden layer to the output layer are given in table 3. After the weights it is possible to see figures 16, 17 and 18, where the most external signals of the neuron module are shown in the screen caption so that the module's behavior with the variation of hidden neurons can be seen. The signals given are: result, result10, result9, result8, result7, result6 and result5, where the signal result represents the final result of the neuron with a resolution of 11 bits, and the other result signals refer to the result of each resolution configuration, for example: result10 refers to the result of the neuron with 10 bits of resolution. The signal $data_in$ refers to the moment the data starts being inserted into the system, where a high level on the signal indicates that data is being introduced into the system and a low level indicates that the input of data to the network has finalized. Finally it is possible to see the last seven signals called: neuralDone, neuralDone10, neuralDone9, neuralDone8, neuralDone7, neuralDone6, neuralDone5, where its rising edge indicates that the data processing for the neurons with 11, 10, 9, 8, 7, 6 and 5 bits of resolution respectively.

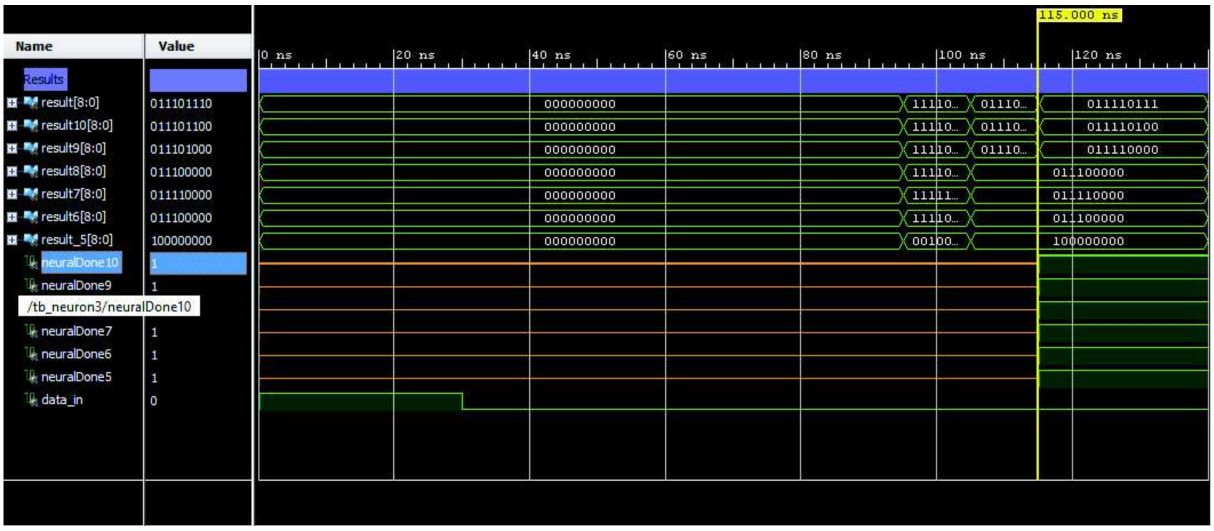
Table 2 – Table of weights for three neurons. Input layer to hidden layer

Weight 1	**Weight 2**
1.07155713	-0.19574127
3.62457177	3.44503749
1.64899717	-3.62053524

Table 3 – Table of weights for three neurons. Hidden layer to Output layer

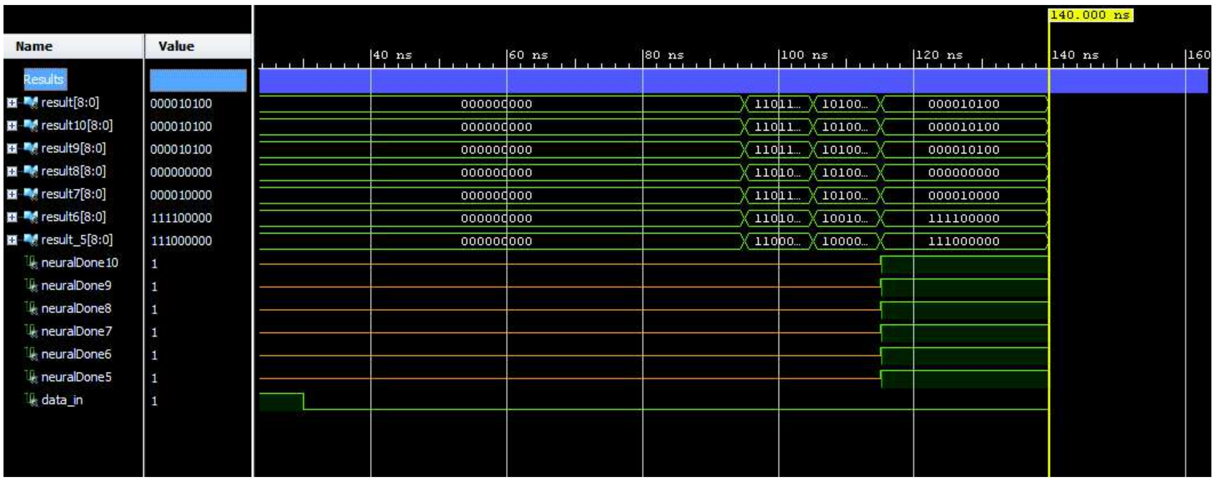
Weight
-4.64059552
3.11537058
1.64513519

Figure 16 – Three neuron ANN - 01 input



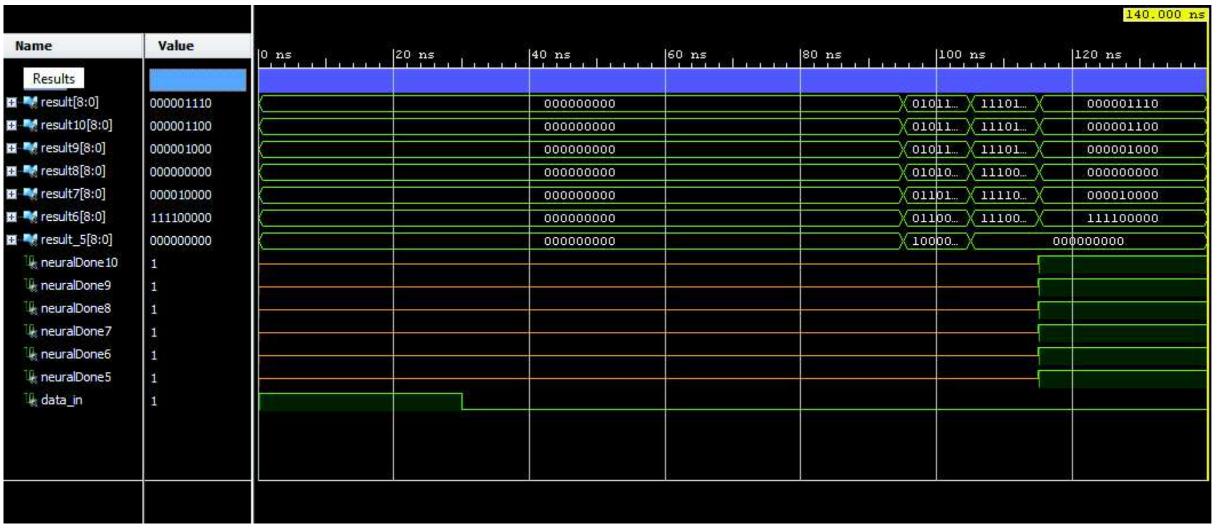
Name	Value
Results	
result[8:0]	011101110
result10[8:0]	011101100
result9[8:0]	011101000
result8[8:0]	011100000
result7[8:0]	011110000
result6[8:0]	011100000
result_5[8:0]	100000000
neuralDone10	1
neuralDone9	1
neuralDone7	1
neuralDone6	1
neuralDone5	1
data_in	0

Figure 17 – Three neuron ANN - 00 input



Name	Value
Results	
result[8:0]	000010100
result10[8:0]	000010100
result9[8:0]	000010100
result8[8:0]	000000000
result7[8:0]	000010000
result6[8:0]	111100000
result_5[8:0]	111000000
neuralDone10	1
neuralDone9	1
neuralDone8	1
neuralDone7	1
neuralDone6	1
neuralDone5	1
data_in	1

Figure 18 – Three neuron ANN - 11 input



Name	Value
Results	
result[8:0]	000001110
result10[8:0]	000001100
result9[8:0]	000001000
result8[8:0]	000000000
result7[8:0]	000010000
result6[8:0]	111100000
result_5[8:0]	000000000
neuralDone10	1
neuralDone9	1
neuralDone8	1
neuralDone7	1
neuralDone6	1
neuralDone5	1
data_in	1

Table 4 displays the results and compares them. In this table it is possible to compare the results obtained from the experiment with three hidden neurons. It shows that the networks with 11, 10 and 9 bit sigmoid look-up tables have their error within the acceptable margin, validating the implementation.

5.1.2 ANN topology with four hidden neurons

This subsection contains the results for the neural network topology with four neurons in the hidden layer. The following figures and tables in this subsection show the expected results obtained with the script, the obtained results from the hardware neural network, the sigmoid resolution with its different outputs.

The bit resolution for the weights and data set inputs consists of 3 bits for the integer part and four bits for the fractional part.

The weights from the input layer to the hidden layer are given in table 5, the weights from the hidden layer to the output layer are given in table 6. After the weights it is possible to see figures 19, 20, 21 and 22, where the most external signals of the neuron module are shown in the screen caption so that the module's behavior with the variation of hidden neurons can be seen. The signals given are: result, result10, result9, result8, result7, result6 and result5, where the signal result represents the final result of the neuron with a resolution of 11 bits, and the other result signals refer to the result of each resolution configuration, for example: result10 refers to the result of the neuron with 10 bits of resolution. The signal $data_in$ refers to the moment the data starts being inserted into the system, where a high level on the signal indicates that data is being introduced into the system and a low level indicates that the input of data to the network has finalized. Finally it is possible to see the last seven signals called: neuralDone, neuralDone10, neuralDone9, neuralDone8, neuralDone7, neuralDone6, neuralDone5, where its rising edge indicates

Table 4 – Results obtained from XOR function for three neurons

x	y	res	Expected value	SW ANN	HW ANN	HW Quadratic error	SW Quadratic error
0	0	11	0	0.05995513	0.0781250	0.00610351562500	0.00359461761332
0	1	11	1	0.96801048	0.9296875	0.00494384765625	0.00102332938983
1	0	11	1	0.95758942	0.9296875	0.00494384765625	0.00179865729594
1	1	11	0	0.03774875	0.0546875	0.00299072265625	0.00142496812656
0	0	10	0	0.05995513	0.0781250	0.00610351562500	0.00359461761332
0	1	10	1	0.96801048	0.9218750	0.00610351562500	0.00102332938983
1	0	10	1	0.95758942	0.9218750	0.00610351562500	0.00179865729594
1	1	10	0	0.03774875	0.0546875	0.00299072265625	0.00142496812656
0	0	9	0	0.05995513	0.0781250	0.00610351562500	0.00359461761332
0	1	9	1	0.96801048	0.9062500	0.00878906250000	0.00102332938983
1	0	9	1	0.95758942	0.9062500	0.00878906250000	0.00179865729594
1	1	9	0	0.03774875	0.0312500	0.00097656250000	0.00142496812656
0	0	8	0	0.05995513	0.0781250	0.00610351562500	0.00359461761332
0	1	8	1	0.96801048	0.0000000	1.00000000000000	0.00102332938983
1	0	8	1	0.95758942	0.0000000	1.00000000000000	0.00179865729594
1	1	8	0	0.03774875	0.0000000	0.00000000000000	0.00142496812656
0	0	7	0	0.05995513	0.0781250	0.00610351562500	0.00359461761332
0	1	7	1	0.96801048	0.0625000	0.87890625000000	0.00102332938983
1	0	7	1	0.95758942	0.0625000	0.87890625000000	0.00179865729594
1	1	7	0	0.03774875	0.0781250	0.00610351562500	0.00142496812656
0	0	6	0	0.05995513	1.8750000	3.51562500000000	0.00359461761332
0	1	6	1	0.96801048	0.8750000	0.01562500000000	0.00102332938983
1	0	6	1	0.95758942	0.8750000	0.01562500000000	0.00179865729594
1	1	6	0	0.03774875	1.8750000	3.51562500000000	0.00142496812656
0	0	5	0	0.05995513	1.7500000	3.06250000000000	0.00359461761332
0	1	5	1	0.96801048	1.0000000	0.00000000000000	0.00102332938983
1	0	5	1	0.95758942	1.0000000	0.00000000000000	0.00179865729594
1	1	5	0	0.03774875	0.0000000	0.00000000000000	0.00142496812656

that the data processing for the neurons with 11, 10, 9, 8, 7, 6 and 5 bits of resolution respectively.

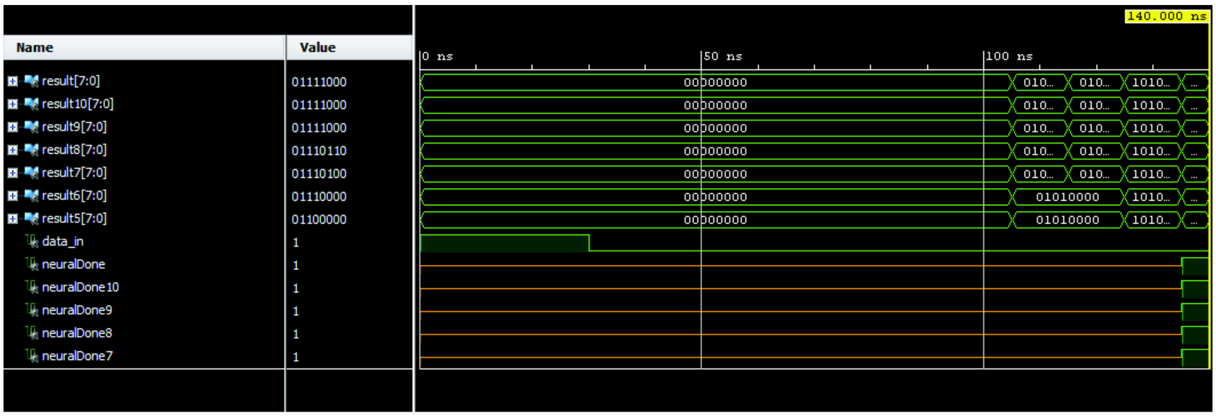
Table 5 – Table of weights for four neurons. Input layer to Hidden layer

Weight 1	**Weight 2**
-1.5370131	0.20571103
-0.55526946	2.24190836
1.25758021	0.0099828
3.41607776	3.60830287

Table 6 – Table of weights for four neurons. Hidden layer to Output layer

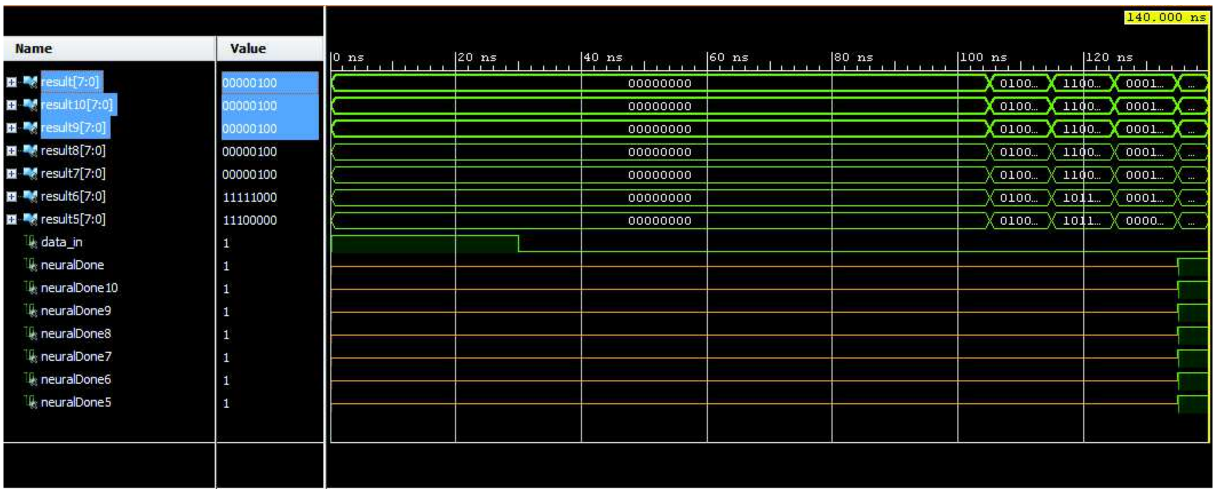
Weight
1.18471773
-2.20053965
-2.60886924
3.70095397

Figure 19 – Four neuron ANN - 01 input



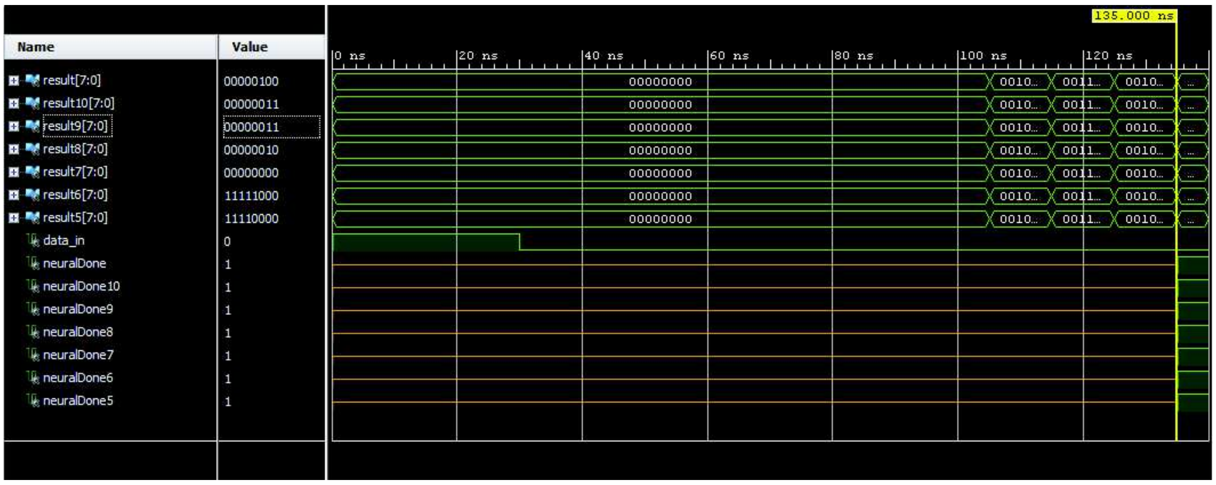
Name	Value
result[7:0]	01111000
result10[7:0]	01111000
result9[7:0]	01111000
result8[7:0]	01110110
result7[7:0]	01110100
result6[7:0]	01110000
result5[7:0]	01100000
data_in	1
neuralDone	1
neuralDone10	1
neuralDone9	1
neuralDone8	1
neuralDone7	1

Figure 20 – Four neuron ANN - 00 input



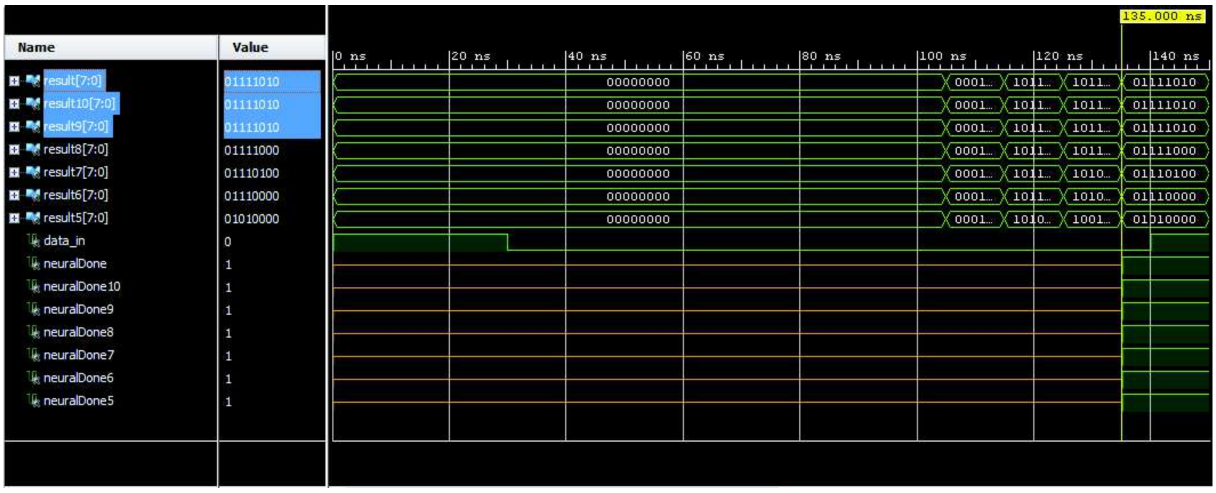
Name	Value
result[7:0]	00000100
result10[7:0]	00000100
result9[7:0]	00000100
result8[7:0]	00000100
result7[7:0]	00000100
result6[7:0]	11111000
result5[7:0]	11100000
data_in	1
neuralDone	1
neuralDone10	1
neuralDone9	1
neuralDone8	1
neuralDone7	1
neuralDone6	1
neuralDone5	1

Figure 21 – Four neuron ANN - 11 input



Name	Value
result[7:0]	00000100
result10[7:0]	00000011
result9[7:0]	00000011
result8[7:0]	00000010
result7[7:0]	00000000
result6[7:0]	11111000
result5[7:0]	11110000
data_in	0
neuralDone	1
neuralDone10	1
neuralDone9	1
neuralDone8	1
neuralDone7	1
neuralDone6	1
neuralDone5	1

Figure 22 – Four neuron ANN - 10 input



Name	Value
result[7:0]	01111010
result10[7:0]	01111010
result9[7:0]	01111010
result8[7:0]	01111000
result7[7:0]	01110100
result6[7:0]	01110000
result5[7:0]	01010000
data_in	0
neuralDone	1
neuralDone10	1
neuralDone9	1
neuralDone8	1
neuralDone7	1
neuralDone6	1
neuralDone5	1

Table 7 displays the results and compares them. In this table it is possible to compare the results obtained from the experiment with three hidden neurons. It shows that the networks with 11, 10 and 9 bit sigmoid look-up tables have their error within the acceptable margin, validating the implementation.

5.1.3 ANN topology with eight neurons

This subsection contains the results for the neural network topology with eight neurons in the hidden layer. The following figures and tables in this subsection show the expected results obtained with the script, the obtained results from the hardware neural network, the sigmoid resolution with its different outputs.

The bit resolution for the weights and data set inputs consists of three bits for the integer part and five bits for the fractional part, the same as the previously implemented topology.

The weights from the input layer to the hidden layer are given in table 8, the weights from the hidden layer to the output layer are given in table 9. After the weights it is possible to see figures 23, 24, 21 and 26, where the most external signals of the neuron module are shown in the screen caption so that the module's behavior with the variation of hidden neurons can be seen. The signals given are: result, result10, result9, result8, result7, result6 and result5, where the signal result represents the final result of the neuron with a resolution of 11 bits, and the other result signals refer to the result of each resolution configuration, for example: result10 refers to the result of the neuron with 10 bits of resolution. The signal $data_in$ refers to the moment the data starts being inserted into the system, where a high level on the signal indicates that data is being introduced into the system and a low level indicates that the input of data to the network has finalized. Finally it is possible to see the last seven signals called: neuralDone, neuralDone10, neuralDone9, neuralDone8, neuralDone7, neuralDone6, neuralDone5, where its rising edge indicates

Table 7 – Results obtained from XOR function for four neurons

x	y	res	Expected value	SW ANN	HW ANN	HW Quadratic error	SW Quadratic error
0	0	11	0	0.03813141	0.0156250	0.000244140625	0.00359461761332
0	1	11	1	0.95626049	0.9375000	0.003906250000	0.00102332938983
1	0	11	1	0.95923448	0.9531250	0.002197265625	0.00179865729594
1	1	11	0	0.05266151	0.0312500	0.000976562500	0.00142496812656
0	0	10	0	0.03813141	0.0156250	0.000244140625	0.00359461761332
0	1	10	1	0.95626049	0.9375000	0.003906250000	0.00102332938983
1	0	10	1	0.95923448	0.9531250	0.002197265625	0.00179865729594
1	1	10	0	0.05266151	0.0234375	0.000549316406	0.00142496812656
0	0	9	0	0.03813141	0.0156250	0.000244140625	0.00359461761332
0	1	9	1	0.95626049	0.9375000	0.003906250000	0.00102332938983
1	0	9	1	0.95923448	0.9531250	0.002197265625	0.00179865729594
1	1	9	0	0.05266151	0.0234375	0.000549316406	0.00142496812656
0	0	8	0	0.03813141	0.0156250	0.000244140625	0.00359461761332
0	1	8	1	0.95626049	0.9218750	0.006103515625	0.00102332938983
1	0	8	1	0.95923448	0.9375000	0.003906250000	0.00179865729594
1	1	8	0	0.05266151	0.0156250	0.000244140625	0.00142496812656
0	0	7	0	0.03813141	0.0156250	0.000244140625	0.00359461761332
0	1	7	1	0.95626049	0.9062500	0.008789062500	0.00102332938983
1	0	7	1	0.95923448	0.9062500	0.008789062500	0.00179865729594
1	1	7	0	0.05266151	0.0000000	0.000000000000	0.00142496812656
0	0	6	0	0.03813141	1.9531250	3.814697265620	0.00359461761332
0	1	6	1	0.95626049	0.8750000	0.015625000000	0.00102332938983
1	0	6	1	0.95923448	0.8750000	0.015625000000	0.00179865729594
1	1	6	0	0.05266151	1.9375000	3.753906250000	0.00142496812656
0	0	5	0	0.03813141	1.8906250	3.574462890620	0.00359461761332
0	1	5	1	0.95626049	0.7500000	0.062500000000	0.00102332938983
1	0	5	1	0.95923448	0.0625000	0.878906250000	0.00179865729594
1	1	5	0	0.05266151	1.8750000	3.515625000000	0.00142496812656

that the data processing for the neurons with 11, 10, 9, 8, 7, 6 and 5 bits of resolution respectively.

Table 8 – Table of weights for eight neurons. Input layer to Hidden layer

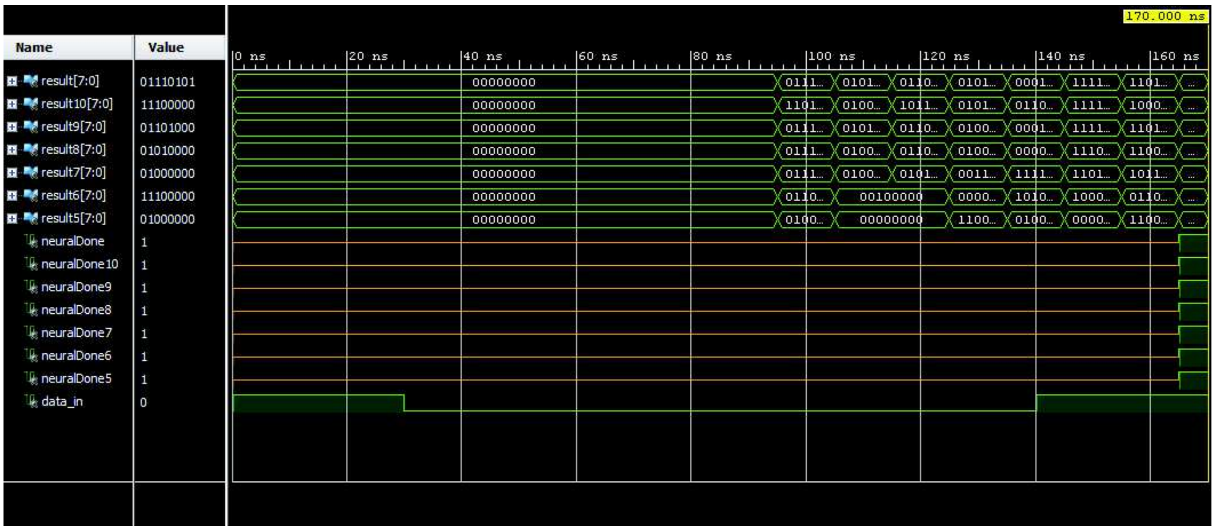
Weight 1	**Weight 2**
-0.55928744	1.7127462
-0.6146875	0.66299767
-0.86636916	2.338686
-1.41060649	-1.37237845
2.07637694	1.96512781
-3.17552737	1.25223065
1.63508328	-0.81111674
2.81203074	1.69237324

Table 10 displays the results and compares them. In this table it is possible to compare the results obtained from the experiment with three hidden neurons. It shows that the networks with 11, 10 and 9 bit sigmoid look-up tables have their error within the acceptable margin, validating the implementation.

Table 9 – Table of weights for eight neurons. Hidden layer to Output layer

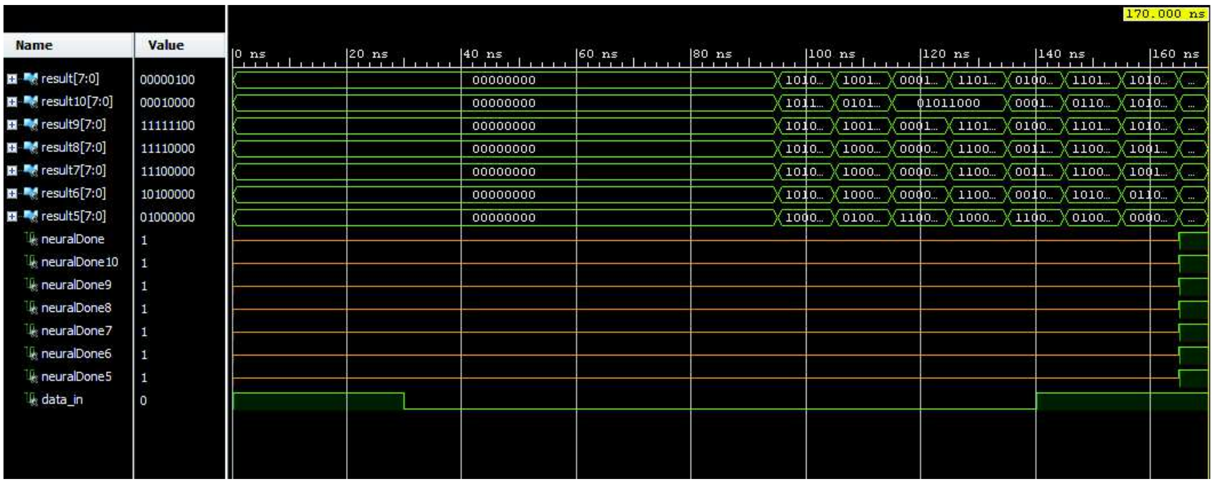
Weight
-1.28684406
-0.40672007
-1.95423893
-0.98740911
1.74449152
2.32356179
-0.7616997
1.4150074

Figure 23 – Eight neuron ANN - 01 input



Name	Value
result[7:0]	01110101
result10[7:0]	11100000
result9[7:0]	01101000
result8[7:0]	01010000
result7[7:0]	01000000
result6[7:0]	11100000
result5[7:0]	01000000
neuralDone	1
neuralDone10	1
neuralDone9	1
neuralDone8	1
neuralDone7	1
neuralDone6	1
neuralDone5	1
data_in	0

Figure 24 – Eight neuron ANN - 00 input



Name	Value
result[7:0]	00000100
result10[7:0]	00010000
result9[7:0]	11111100
result8[7:0]	11110000
result7[7:0]	11100000
result6[7:0]	10100000
result5[7:0]	01000000
neuralDone	1
neuralDone10	1
neuralDone9	1
neuralDone8	1
neuralDone7	1
neuralDone6	1
neuralDone5	1
data_in	0

Figure 25 – Eight neuron ANN - 11 input

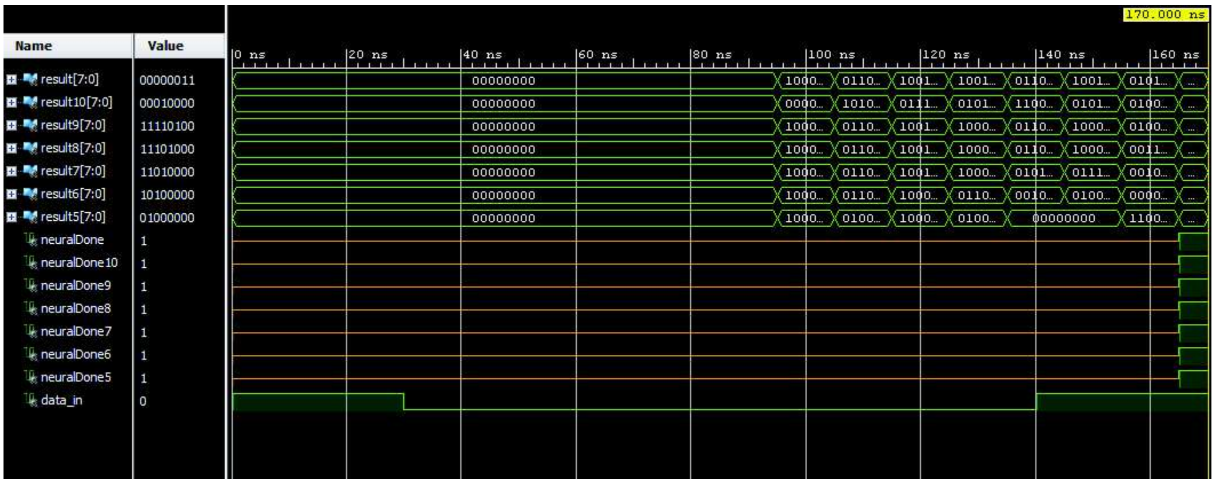
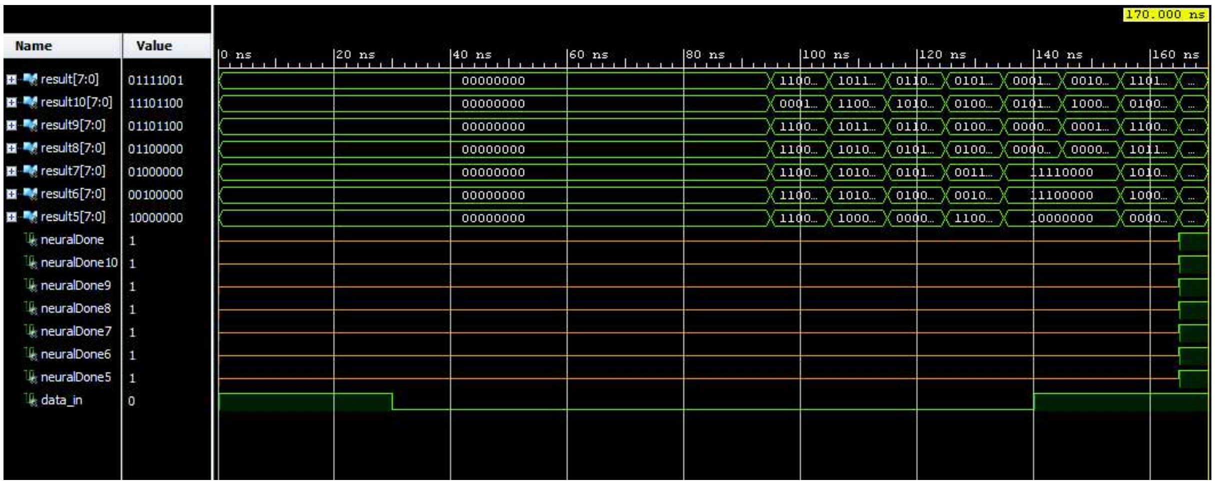


Figure 26 – Eight neuron ANN - 10 input



5.2 Discussion

This chapter intends to discuss the results obtained from the experiments, the limitations of the system and the difficulties encountered.

5.2.1 Discussion of results

Comparing the quadratic error from the previous tables it is possible o notice that the response given by the hardware ANNs are within the error margin of 0.01. It is possible to notice that these results are possible until the 9 bits of resolution is reached. After that when the resolution decreases the results obtained from the ANN are unpredictable and are not suited for use.

Despite using low resolution for the inputs and weights (8 bits) and a slightly higher resolution for the sigmoid look-up table (sigmoid activation functions with resolution from 9 to 11 bits are utilizable) the results obtained fall within the acceptable uncertainty, and if compared to the results obtained from the software ANNs, it is possible to notice that

Table 10 – Results obtained from XOR function for eight neurons

x	y	res	Expected value	SW ANN	HW ANN	HW Quadratic error	SW Quadratic error
0	0	11	0	0.04307439	0.0312500	0.0009765625000	0.00185540307387
0	1	11	1	0.95712946	0.9149625	0.0072313764062	0.00183788319989
1	0	11	1	0.95772737	0.9453125	0.0029907226562	0.00178697524712
1	1	11	0	0.04712844	0.0546875	0.0029907226562	0.00222108985683
0	0	10	0	0.04307439	0.1250000	0.0156250000000	0.00185540307387
0	1	10	1	0.95712946	1.7500000	0.5625000000000	0.00183788319989
1	0	10	1	0.95772737	1.8515625	0.7251586914060	0.00178697524712
1	1	10	0	0.04712844	0.1250000	0.0156250000000	0.00222108985683
0	0	9	0	0.04307439	1.9687500	3.8759765625000	0.00185540307387
0	1	9	1	0.95712946	0.8125000	0.0351562500000	0.00183788319989
1	0	9	1	0.95772737	0.8515625	0.0220336914062	0.00178697524712
1	1	9	0	0.04712844	1.9062500	3.6337890625000	0.00222108985683
0	0	8	0	0.04307439	1.8750000	3.5156250000000	0.00185540307387
0	1	8	1	0.95712946	0.6250000	0.1406250000000	0.00183788319989
1	0	8	1	0.95772737	0.7500000	0.0625000000000	0.00178697524712
1	1	8	0	0.04712844	1.8125000	3.2851562500000	0.00222108985683
0	0	7	0	0.04307439	1.7500000	3.0625000000000	0.00185540307387
0	1	7	1	0.95712946	0.5000000	0.2500000000000	0.00183788319989
1	0	7	1	0.95772737	0.5000000	0.2500000000000	0.00178697524712
1	1	7	0	0.04712844	1.6250000	2.6406250000000	0.00222108985683
0	0	6	0	0.04307439	1.2500000	1.5625000000000	0.00185540307387
0	1	6	1	0.95712946	1.7500000	0.5625000000000	0.00183788319989
1	0	6	1	0.95772737	0.2500000	0.5625000000000	0.00178697524712
1	1	6	0	0.04712844	1.2500000	1.5625000000000	0.00222108985683
0	0	5	0	0.04307439	0.5000000	0.2500000000000	0.00185540307387
0	1	5	1	0.95712946	0.5000000	0.2500000000000	0.00183788319989
1	0	5	1	0.95772737	1.0000000	0.0000000000000	0.00178697524712
1	1	5	0	0.04712844	1.0000000	1.0000000000000	0.00222108985683

their quadratic errors are quite similar, indicating that a high resolution was not needed to solve the problem presented.

5.2.2 Difficulties encountered

Developing hardware for FPGAs is a challenging task, especially for software developers. When developing software the developer may create a sequential approach for developing ANNs, which means that everything is executed in an expected manner, first the inputs are introduced into the system, then they are multiplied by the weights, and so on. And when developing hardware descriptions for FPGAs this approach is not possible, due to the fact that FPGAs possess a parallel nature, and this means that all the components in the described hardware are executing their functions simultaneously. In order to solve a problem of this magnitude in FPGAs a well devised datapath is necessary, and this datapath must contain control signals to allow the components to perform their functions. And this is the main difficulty experienced during the development of this project, which is a modification of the developing paradigm.

Developing for this SoC presented a few more challenges than the ones described above. The architecture of the Zynq family of SoC is quite complex, and so are the components of the system. In order to reach all the objectives it was necessary to learn how this specific SoC works and how all of its components can be used, which is not an easy task due to the fact that many components require a thorough knowledge of how they work and very difficult control logic for them. This was the main reason that the learning method was not implemented. The idea of the project was to develop the learning method in software, which is allowed by the AMBA AXI bus, which was presented previously, and it was decided that it should be developed in software due to the lack of hardware resources available. However, due to the complexity of the available resources and specially the complexity of the AMBA AXI bus, it was decided that this project would not be able to learn. The learning method could have been implemented in hardware in order to increase the speed of processing, but doing so would drastically reduce the maximum number of connections that can be implemented in the hardware, turning it inviable as a product.

5.2.3 Limitation analysis

After developing the project some limitations were noted. The main limitation being the hardware resources given by the selected SoC. The SoC used allows for a maximum of 110 connections to be implemented in hardware. Considering a topology similar to the one presented above, it allows for up to two input neurons, fifty five hidden neurons and one output neuron. This indicates that this specific hardware is only suitable for ANNs ranging from simple to medium complexity. A further of the available SoCs in the Zynq family has shown that there are more powerful SoCs available, which would allow for incredibly complex topologies, however, their price is a main concern since this project is intended to be turned into a product and it is expected to be used instead of using a notebook/workstation (the price of the high-end SoCs in the Zynq family can be as expensive as high-end workstations, making them unusable for the type of problems this project is trying to solve).

5.3 Comparison between a hardware approach and a software approach

As can be seen in the previous chapters, the implementation process differs greatly between a hardware implementation and a software implementation. A hardware implementation must be carefully planned and have a well structured datapath so that it avoids pipeline stalls and the maximum throughput of data can be reached. Such is not the case for software implementation of ANNs. With software implementations the manner in which the arithmetic operations are implemented do not matter to the end result, as much as in a hardware implementation, as long as it follows the flux of data through the network.

However, with the abstractions developed in this project it is possible to automate the entire process of developing an ANN, which means that when this project is turned into a product, it can be used by anyone without a software or hardware background, allowing the user to focus entirely on the solution of the presented problem. As this project was designed with the intention of geological applications, the only thing necessary to take this device to the field is to define an ANN topology, train it and use it.

6 Conclusion

The objective of this project was to create a generic hardware architecture for a Neuron so that it will have a response similar to the one obtained from a software ANN. From the results obtained in the previous chapter it is possible to tell that it is possible to create a neuron in programmable logic with a lower resolution than the ones made in software and have a good response, despite the difference in quantization and rounding between both systems. Such errors are introduced in the system due to its low resolution nature seem to have little effect in the output for the ANN implemented to validate the neuron.

When the results and difficulties obtained by the papers cited in this work are compared to the ones obtained in this, it is possible to tell that the implementation of hardware neural networks is now viable. What once was a time where only high-end FPGAs had the minimum necessary resources for simple neural networks, now is entirely viable. The number of hardware resources available to the developer are up to 100 times the ones available 10 years ago, and even several times more complex than FPGAs created approximately 5 years ago. The low resolution used in this project indicates that high bit resolution is not necessary for these problems, as even the papers that used 32 bits of resolution obtained similar responses to the one developed in this project.

Despite having a good output response compared to the software implementation, it is important to note that the type of system tested benefited by the implemented neuron's nature, however it is not possible to validate the neuron's response for more systems, since there may exist systems that do not benefit from the neuron's nature, despite having external projects indicating that this implementation is viable for more complex systems. To validate the system for a wider range of systems it is necessary to perform more tests and perform a more thorough static analysis of the implemented neuron.

From the tables presented in the previous chapter it is possible to tell that the system's outputs for higher resolution sigmoid tables tend to follow the outputs obtained by the software implementation of the neural networks, however, it is possible to tell that the quantization error tends to accumulate with the number of calculations and with the weight's value.

Comparing the networks with three, four and eight neurons with 11 bit sigmoids, it is possible to see that the quantization error doesn't play as important role as the weight's values. This is possible to see by comparing the top results. While the 4 neuron topology tends to follow the software implementation closely, the other networks tend to deviate from the software implementation, even if by a small deviation.

In order to test this hypothesis, a higher resolution network was implemented, where the input data set has 2 bits of resolution for the integer part and 14 bits for the fractional part and the weights have 4 bits for the integer part and 12 bits for the fractional part of the value. And then a 12 bit sigmoid table was created for the sake of comparing the previously obtained results to the new ones. From the obtained result, the obtained results tend to follow the software implementation closely indicating that 16 bits of resolution could be enough for most problems. This indicates that the number of neurons tends to propagate the rounding error, so having a higher resolution should improve the results for a larger range of neurons.

The difference between the bit resolution of both implementations (floating-point arithmetic and fixed point 8 bit arithmetic for the software and hardware implementation respectively) has no major implications in the resulting output.

6.1 Future work

There are still many other implementations that could be tested, as well as there are more features that could be implemented in future works. Among them are:

- **The training of the neuron so it is more independent from PCs**

  The next step of the implementation is to close the loop and implement the back-propagation algorithm. For the backpropagation algorithm it is intended to use the ARM's NEON co-processor. The reason for using this co-processor instead of the processor is because the calculation time is improved by a factor of 2 approximately, according to ARM's benchmark, due to the fact that it is possible to perform up to 16 parallel multiplications. And the reason for not using the programmable logic portion of the SoC is due to the fact that it is rather small compared to higher end FPGAs and has lower hardware resources than higher end DSP FPGAs. However, a proper comparison between the three implementations is necessary.

- **Tests with wider range of bit resolution**

  This work focused on 8 bit and 9 bit implementations. It is important to test a wider range of resolutions in order to get the optimal size for most implementations.

- **Implementation of a hardware module for the sigmoid function**

  As the importance of the sigmoid's resolution increases, so does its size in memory, by a factor of 2 every bit increase. For real implementations it is impractical to implement sigmoid functions with a resolution of more than 13 bits, so a sigmoid function approximation must be implemented for higher resolutions networks. This will lead to an increase of a few clock cycles, but the results could be greatly improved.

- **Complete independence from PC**

  At the moment, all the sigmoid functions and the networks are implemented by software, which increases the system's independence on PCs, since the linux running on one, or both, processors can easily generate its own architecture. However it is important to create a visual software that is capable of generating the network according to the input provided by the user (on the zedboard). Finally, it is important to install Xilinx's Vivado on the ARM platform (not available yet) and implement a hardware reconfiguration module, so that the Zynz SoC can synthetize its own hardware and program itself according to any given input.

- **Different input implementations**

  This system is to be used in field applications. So several modules must be implemented in order to get the input data from its surroundings, with or without the need of the processor for this.

Bibliography

ANN wikibooks. Disponível em: <https://en.wikibooks.org/wiki/Artificial_Neural_Networks/Boltzmann_Learning>. Cited in page 32.

ARM. *ARM A9 Architecture*. Disponível em: <http://www.arm.com/images/Cortex_A9_large.png>. Cited in page 25.

CORIC, S.; LATINOVIC, I.; PAVASOVIC, A. A neural network fpga implementation. *Neural Network Applications in Electrical Engineering, 2000. NEUREL 2000. Proceedings of the 5th Seminar on*, Belgrado, 2001. Cited in page 35.

ENGELBRECHT, A. P. *Computational intelligence An Introduction*. [S.l.]: Wiley, 2007. Cited in page 26.

FASCICOLA TOM, O. S. et al. Adaptive interfaces based on fpga implemented artificial neural network. *SYMPOSIUM OF ELECTRONICS AND TELECOMMUNICATIONS*, 2004. Cited in page 36.

GADEA, R. et al. Artificial neural network implementation on a single fpga of a pipelined on-line backpropagation. Madrid, Espanha, 2000. Cited 2 times in pages 35 and 37.

GLOVER, F. Tabu search: a tutorial. University of Boulder, 1990. Cited in page 26.

GLOVER, F. Tabu search fundamentals and uses. University of Colorado, 1995. Cited in page 26.

HANSSON, S. O. *Decision Theory: A Brief Introduction*. [S.l.]: Royal Institute of Technology (KTH), 1994. Cited in page 26.

HAYKIN, S. et al. *Neural Networks A comprehensive foundation*. [S.l.]: Prentice Hall, 1999. Cited 4 times in pages 17, 27, 32, and 39.

PINJARE, S. L.; M, A. K. *Implementation of Neural Network Back Propagation Training Algorithm on FPGA*. 2012. Cited in page 36.

SUMATHI, S. *Computational intelligence Paradigms: Theory and Applications using Matlab*. [S.l.]: Chapman & Hall, 2010. Cited 2 times in pages 31 and 32.

SUN, R. Artificial intelligence: Connectionist and symbolic approaches. Madrid, Espanha, 2000. Cited in page 26.

TSAI, M.-S.; FU, Y.-H. Implementation of high performance hardware based toroidal neural network with learning capability. *Advanced Intelligent Mechatronics, 2009. AIM 2009. IEEE/ASME International Conference on*, Singapura, 2009. Cited in page 36.

VENKATESAN, R.; BALAMURUGAN, B. A real-time hardware fault detector using an artificial neural network for distance protection. *Power Delivery, IEEE Transactions on*, 2001. Cited in page 35.