eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea

ZIENTZIA ETA TEKNOLOGIA FAKULTATEA
FACULTAD DE CIENCIA Y TECNOLOGÍA

Final Degree Project
Degree in Electronic Engineering

Implementation of digital circuits for real-time classification of hyperspectral images

Author:
Avila Arenas, Iker
Director:
del Campo Hagelstrom, Ines Juliana

Leioa, the 22nd of June 2022

Contents

1 Introduction and objectives **1**
1.1 Introduction . . . 1
1.2 Objectives . . . 2
2 Theoretical Basis: FPGAs, Machine Learning and Hyperspectral Images **3**
2.1 Field Programmable Gate Arrays (FPGA) . . . 3
2.1.1 Brief history of digital electronics . . . 3
2.1.2 Field Programmable Gate Arrays (FPGA) . . . 3
2.1.3 Xilinx 7-series FPGAs . . . 4
2.1.4 Nexys A7 board . . . 5
2.2 Machine Learning . . . 6
2.2.1 Neural Networks . . . 6
2.2.2 Extreme Learning Machine . . . 8
2.3 Hyperspectral Images . . . 9
2.3.1 Electromagnetic spectrum: Visible and Infrared light . . . 9
2.3.2 Material Reflectance . . . 10
2.3.3 Hyperspectral Images for material detection . . . 11
3 Training ELM network for Hyperspectral Imaging **12**
3.1 The AeroRIT dataset: train and test sets . . . 12
3.2 Parameter design for ELM network . . . 13
3.3 ELM accuracy on the AeroRIT dataset . . . 14
3.4 Principal Component Analysis . . . 15
4 Implementation of ELM network in VHDL **17**
4.1 The individual neuron using DSP slices . . . 17
4.1.1 The DSP slice . . . 17
4.1.2 Neuron Design in VHDL and behavioral simulation . . . 18
4.1.3 Neuron hardware implementation and timing simulation . . . 19
4.2 Sigmoid function via ROM . . . 21
4.2.1 Sigmoid Function Hardware design . . . 21
4.2.2 Hidden Layer Neurons design . . . 21
4.2.3 Hidden layer neurons implementation . . . 22
4.3 Comparator . . . 23
4.4 Layout of a parallel ELM architecture . . . 24
4.4.1 10 hidden neurons neural network . . . 24
4.4.2 ELM hardware network application to HSI . . . 25
5 Power consumption analysis **28**
5.1 FPGA power consumption terminology . . . 28
5.2 Power analysis in the Vivado Design Suite . . . 29
5.2.1 Vectorless Vivado Power Report . . . 29
5.2.2 Vector (SAIF) Based Power Analysis . . . 30
5.3 Power for different amount of hidden neurons . . . 31
6 General Conclusions **32**
References **33**
Appendices: VHDL codes **36**

List of Figures

1 Field Programmable Gate Array (FPGA) architecture (D. Punia [1]). . . . 4
2 Artix-7 FPGA architecture overview by Xilinx [2] 5
3 Image of the Nexys A7 board by Digilent. 5
4 Scheme of a 2 hidden layer neural network with 2 output neurons. 6
5 Sigmoid function values $S(x)$ for input values x between -7.5 and 7.5. . . 7
6 Topology of an Extreme Learning Machine neural network, that is a single hidden-layer feed-forward neural network [3]. 8
7 Electromagnetic Spectrum and Visible spectrum. Image modified from Wikimedia commons[4]. 10
8 Reflectance signatures of grass and olive green paint in the light range between $430nm$ to $860nm$. Signatures the ECOstress spectral library [5]. . 10
9 Representation of a hyperspectral data cube and a pixel spectra. 11
10 The AeroRIT dataset is an airborne captured hyperspectral dataset that includes an RGB image 10a and a labelled ground-truth 10b that differentiates 5 classes. 12
11 ELM network test accuracy in AeroRIT test set over different number of neurons in the hidden layer. 14
12 Confusion matrix for ELM application over AeroRIT hyperspectral dataset. 15
13 Mean signatures of the 5 classes in AeroRIT for 3000 random pixels from each class 15
14 First 2 principal components of the different classes present in AeroRIT. . 16
15 Coefficients of the first 3 principal components in the AeroRIT dataset. . . 16
16 ELM network test accuracy over AeroRIT in the 15 principal components as a function of the number of neurons in the hidden layer. 16
17 Basic DSP48E1 Slice Functionality 17
18 RTL Schematic for a single Neuron. 18
19 Behavioral simulation of a single neuron for an input of 3 numbers. 19
21 Timing simulation of a single neuron for an input of 3 numbers. 19
20 Schematic for a single neuron after implementation, where it can be seen the use of a single DSP, the I/O buffers and the *sload* register. 20
22 RTL schematic for the Sigmoid function. 21
23 Schematic for the hidden neuron made by a neuron and the Sigmoid function. 22
24 Behavioral simulation of a neuron and the Sigmoid function. 22
25 Post implementation timing simulation for a neuron and the Sigmoid function. 23
26 RTL Schematic for a comparator if the output layer has 5 neurons. 23
27 10 hidden neurons ELM network post synthesis functional simulation. . . . 25
28 100 hidden neurons ELM network post synthesis functional simulation. . . 26
29 RTL schematic for an ELM neural network with 10 neurons in the hidden layer and 5 neurons in the output layer. 27
30 Vectorless power estimation report for a 10 hidden neurons ELM NN. . . . 29
31 vector based power estimation report for a 10 hidden neurons ELM NN using post implementation functional simulation for 100MHz clock.. 30
32 vector based power estimation report for a 10 hidden neurons ELM NN using post implementation functional simulation for 1MHz clock. 30
33 ELM network total power consumption (P) as a function of the number of neurons in the hidden layer (L). 31

List of Tables

1 Xilinx 7-series families hardware resources comparison. 5
2 Hardware resources available in the Artix-7CS324XC7A100T FPGA. . . . 5
3 Amount of pixels for the train set and test set for each class randomly selected from the AeroRIT dataset, and the total amount of pixels for each class in the dataset. The percentage of training pixels over the total is shown in parenthesis. 13
4 Hardware resources and latency for a single neuron. 19
5 Timing worst slacks for a single neuron. 19
6 Hardware resources and the clock delays for the Sigmoid function and a hidden neuron (made from a single neuron and the Sigmoid function). . . . 22
7 Timing slacks for a neuron followed by the Sigmoid function. 23
8 Hardware resources and the clock delays for a 5 output neurons comparator. 23
9 Hardware resources and latency for multiple ELM networks. 25
10 Timing worst slacks for a full ELM network with 100 neurons in the hidden layer. 26
11 Effective Thermal Resistance to Air for the Artix-7CS324XC7A100T depending on air velocity [6]. 28

List of Acronyms and Abbreviations

ΘJA	Effective Thermal Resistance to Air.
ASIC	Application specific integrated circuit.
CLB	Configurable logic block.
CPU	Central processing unit.
DSP	Digital signal processing (unit).
ELM	Extreme Learning Machine.
EM	Electromagnetic.
FF	Flip-Flop.
FPGA	Field programmable gate array.
HDL	Hardware description language.
HSI	Hyperspectral Imaging.
I/O	Input/Output.
IC	Integrated circuit.
IR	Infrared.
LUT	Look-up table.
ML	Machine Learning.
MUX	Multiplexer.
NIR	Near-Infrared.
NN	Neural Network.
PCA	Principal component analysis.
PCB	Printed circuit board.
PLA	Programmable logic array.
PLD	Programmable logic device.
PROM	Programmable read only memory.
RAM	Random access memory.
RGB	Red, green and blue.
ROM	read only memory.
RTL	Register transfer level.
SAIF	Switching activity values file.
SVD	Singular Value Decomposition.
VHDL	Very high speed hardware description language.
VIS	Visible.

1 Introduction and objectives

1.1 Introduction

In 1948 the first bipolar transistor was developed [7], this led to the creation of integrated circuits (ICs) made from them. ICs began to grow fast following the Moore's law [8]. Initially there were only full-custom devices completely designed and built for one specific task. Then semi-custom devices made it to the market, which allowed the engineers to design the applications for the device from previously manufactured standard silicon devices. Later programmable logic devices (PLD) were developed, these allowed to reuse the same IC for different tasks, thus saving resources. Field programmable gate arrays (FPGA) are the pinnacle of the semi-custom technology.

To design the FPGAs, as well as other ICs, various hardware description languages (HDL) were created. This hardware description languages allow the engineers to design the circuits from an algorithmic level. The most well known ones are Verilog and VHDL (Very high speed HDL), in fact they both are IEEE standards [9] and therefore are capable of describing any digital circuit internationally.

Together with ICs, and in part driven by the computational power they provided, algorithms were developed to analyze large amounts of data (big data). We know these algorithms as Machine Learning (ML) algorithms and neural networks (NN) are their highest exponent [10]. In this context in 2006 a paper proposing one new type of neural network called extreme learning machine (ELM) was published [11].

The ELM neural network differentiates itself from the rest of the neural networks by the fact that the neurons in its hidden layer are set randomly and never trained. This is ideal for hardware implementation since it allows to set the range of the weights in the hidden layer, thus making it viable to use fixed-point representation for the data.

In the early 20th century, hence at the same time as the ICs and ML, the development of hyperspectral images (HSI) started with remote sensing technology. This technology was originally developed for military applications [12], but it soon became clear that it had civilian applications as well. One of the first civilian applications of remote sensing was in agriculture [13], where it was used to map crop conditions. HSI are images that contain information about the spectral properties of the objects in the scene, thus are capable of detecting materials. Hyperspectral images are now used in a variety of fields, including environmental monitoring, mineral exploration, and medical diagnosis [14].

Recently a trend has begun to unite these three technologies, specifically to implement machine learning algorithms on FPGAs for the analysis of hyperspectral images in real time. Some examples of this are *Onboard target detection in hyperspectral image based on deep learning with FPGA implementation* by Sherin and Gayathri [15] or *Optimizing CNN-based Hyperspectral Image Classification on FPGAs* by Shuanglong et. al. [16].

However these designs encounter one of the biggest problems of FPGAs, their energy consumption. The increased integration of circuits has led to an increment of their required power to operate, making the capability to estimate the power required by a device a critical step in the design process.

1.2 Objectives

The main objective of this project is to develop a hardware design for a machine learning neural network capable of analyzing hyperspectral images in real time. This hardware design will be designed in the VHDL hardware description language, following what was learned in the *Digital Electronics* and *Design of Digital Systems* subjects.

For this another objective of the work is to learn the full process of implementing a digital circuit from the VHDL design from the most basic elements to the final simulation of a working device. This means that with the hardware designed we will validate its functionality with a test on a real time application for a given hyperspectral dataset. The hardware design will be simulated in the Vivado software and the hardware resources that a complete network and its components need will be analyzed. Finally the energy consumption of the device will be studied.

The neural network to be implemented will be that of an Extreme Learning Machine (ELM) algorithm. This will require to understand general concepts of neural networks, and specifically what an ELM network is and how it is trained. This NN will be trained to analyze hyperspectral images, thus an understanding of hyperspectral imaging will be needed.

The structure of this text is as follows, in the coming section the theoretical basis behind the three main technologies of the work will be studied, that is, digital circuits focusing on FPGA, the theory behind neural networks and extreme learning machine, and what HSI are and how can they be used to detect materials.

Next a section on how an ELM network can be trained to analyze hyperspectral images and how the main parameter of it can be chosen is explained. Here using a specific hyperspectral dataset how well the ELM algorithm can classify different materials is experimentally analyzed. In addition, the concept of principal component analysis is introduced, and how well it can complement the ELM network is studied.

After this, a section with the complete hardware description of the ELM network in VHDL will be presented. Here the work will start with most basic block of the network, the neuron, and it will escalate in complexity until a complete network is obtained. In this section the hardware requirements and the behaviour of each block are presented.

The fifth section will be an analysis of the energy consumption of the designed network. For this, first the main concepts related to power consumption in FPGA are introduced, and some of the possibilities to estimate the power consumption of any design are explained. This section is finished by analyzing how the power consumption of the designed ELM network grows when the number of neurons in the hidden layer increases. Finally, there will be a last section discussing the conclusions of the work.

2 Theoretical Basis: FPGAs, Machine Learning and Hyperspectral Images

2.1 Field Programmable Gate Arrays (FPGA)

2.1.1 Brief history of digital electronics

The history of digital electronics can be traced back to the 1920s and 1930s, when vacuum tubes were used to develop the first electronic digital computers. In 1948 the first bipolar transistor was developed [7] and led to the development of integrated circuits (ICs) in the 1950s. Here the miniaturization of the transistor began in the search for ICs with more transistors and therefore more capabilities. In particular, in 1965, Gordon Moore observed that the number of transistors in an integrated circuit doubled every two years, what is called the Moore's law [8]. This law has marked the development of this technology to our present day, with slight modifications.

The first integrated circuits were full-custom integrated circuits, that is, developed for one specific task. However when the complexity of the task increased, the number of transistors required, the design time, and the cost, also did so. In this context semi-custom devices made it to the market, these where standard cells or gate arrays that could be mass produced by the factories and then programmed by the engineers for the desired specific task, thus saving resources and money. Later the first commercial programmable logic device (PLD) appeared, the programmable read only memory (PROM), which allowed to further save resources. This was followed by the programmable logic array (PLA) and later the field programmable logic array (FPGA). This way full-custom Application Specific Integrated Circuits (ASICs) have been relegated to a place where they are only used for devices that are to be mass produced.

As it has been said the first ICs contained few transistors, therefore it was possible to design their operation at the electrical level. Then more complex circuits required to move to a logical level design through standardized logic gates, which work internally with transistors. In this way higher design levels were developed one after another to describe the circuits, after these two came the register transfer level (RTL), algorithmic level and finally the system level.

FPGAs are usually described at an algorithmic level and for this hardware description languages (HDL) are used, either Verilog or VHDL. Multiple software allow to then simulate the hardware designed, or to create the RTL or other lower levels schematics from the code. This tools are created to be used during the hole design process, from the coding of the modules in the HDL to estimating the power consumption of the final device. This power consumption estimation is specially important in FPGAs since their increased transistor integration has made their energy consumption grow, thus becoming one of their major issues.

2.1.2 Field Programmable Gate Arrays (FPGA)

Field Programmable Gate Arrays (FPGA) are semiconductor devices based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects [17], see Figure 1. FPGAs are chips designed to be reconfigured for different applications.

Commonly CLB blocks include Flip Flops (FF), Look Up Tables (LUT) and multiplexers (MUX). However nowadays it is common for FPGAs to include other embedded blocks to perform other specific tasks such as Random Access Memory (RAM) Blocks, or Digital Signal Processing (DSP) slices.

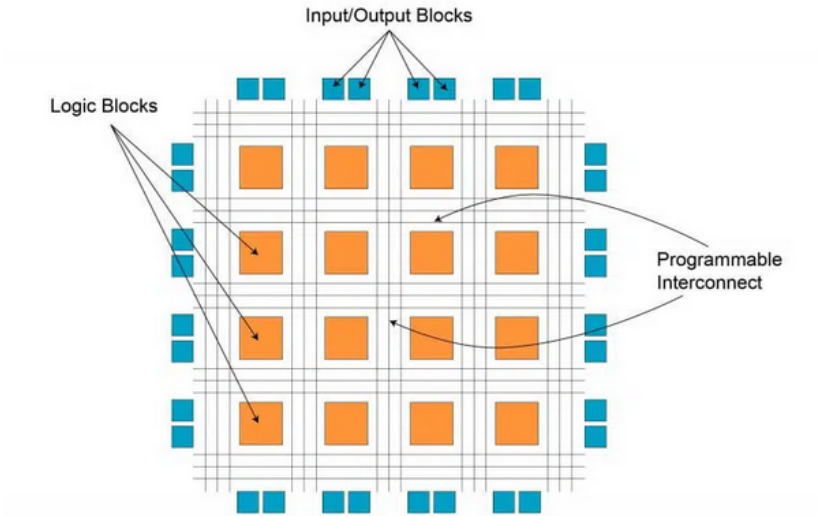

Figure 1: Field Programmable Gate Array (FPGA) architecture (D. Punia [1]).

The major FPGA manufacturers are Xilinx and Altera, now part of AMD and intel respectively. Both companies offer powerful tools for logic design, and both have a very solid history in programming FPGAs. They are compatible with VHDL and Verilog hardware description languages. This work will focus on Xilinx devices and development tools since they are the most widely used.

2.1.3 Xilinx 7-series FPGAs

Xilinx is a technological company, now part of AMD, that produces many different silicon devices. It is best known for developing FPGAs. Xilinx has been building FPGA since 1984 and through various generations has been adding new architectural resources to them [17].

The latest Xilinx FPGA generation is the 7-series. This generation of FPGA includes on-chip memory, DSP engines, Different size LUTs, precise low jitter clocking, improved IO connectivity, and multiple size MUXes among others [18].

Each generation Xilinx introduces different families based on the same architecture to provide solutions that address different price, performance or power requirements. The 7-series includes 4 different families: Spartan-7, Artix-7, Kintex-7 and Virtex-7. In Figure 2 the architecture of the Artix-7 FPGA is shown. The other families of the generation use the same hardware resources but in a different quantity, this differences are shown in Table 1 for the principal commercial FPGA, however there are larger devices available.

Table 1: Xilinx 7-series families hardware resources comparison.

	Spartan-7	Artix-7	Kintex-7	Virtex-7
CLB	102K	251K	478K	1955K
Block RAM	4.2Mb	13Mb	34Mb	68Mb
DSP Slices	160	740	1920	3600
MicroBlaze CPU [1]	260 DMIPs	303 DMIPs	438 DMIPs	441 DMIPs
IO pins	400	500	500	1200

2.1.4 Nexys A7 board

The Nexys A7 is an FPGA development developed by Digilent [20] that we have available in the laboratory. A FPGA development board is a printed circuit board (PCB) that connects the FPGA to external resources such as switches, 7 segment leds, usb connectors, etc. It is a board for prototyping purposes.

The Nexys A7 is based in the Artix-7CS324XC7A100T, a specific model of the Artix-7 FPGA by Xilinx [21]. The board includes, among others: Ethernet, USB, UART, JTAG, and VGA connectors, built-in temperature sensor, microphone, accelerometer, 8 7-segment displays and 16 switches. An image of the board is shown in Figure 3.

More precisely the Nexys A7 is built with the Artix-7CS324XC7A100T FPGA, which is one specific model of the Artix-7, its hardware resources are shown in Table 2.

Table 2: Hardware resources available in the Artix-7CS324XC7A100T FPGA.

	LUT	FF	MUX	DSP
Neuron	63400	126800	147550	240

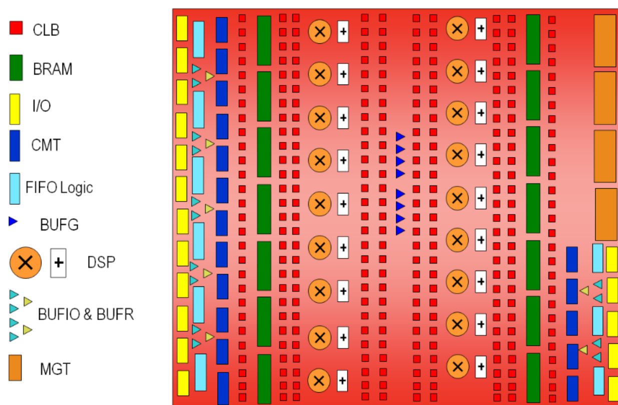

Figure 2: Artix-7 FPGA architecture overview by Xilinx [2]

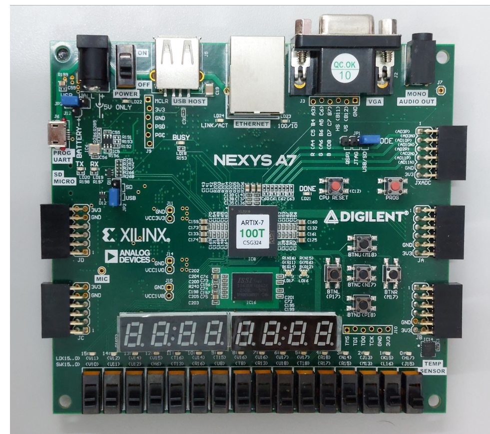

Figure 3: Image of the Nexys A7 board by Digilent.

[1]It is measured in DMIPs which stands for *Dhrystone Millions of Instructions* Per Second. Dhrystone is a benchmark for CPU performance measurement [19].

2.2 Machine Learning

Machine Learning (ML) is a part of artificial intelligence that develop computer algorithms to create good approximation models for large amount of data. The application of ML algorithms to the data is called data mining, since it finds useful material from a large amount of raw data. This algorithms are part of artificial intelligence because they have the ability to adapt, to learn based on previous experience [10].

One option is to create algorithms that can find patterns in the data without us telling them what to look for, this is called unsupervised ML. For example, in social media such unsupervised algorithms are used to generate clusters of videos, we don't know what exactly is being clustered but the algorithm finds patterns.

However this work will focus on supervised ML algorithms. This algorithms learn based on training data and the desired outcome for that data. In the social media analogy this would be like creating an algorithm to cluster cat videos, and if to do so we trained it by giving it example videos with the presence of cats in them.

Machine learning algorithms have multiple applications for regression calculation or for classification. Specifically they have been widely used for image recognition, for example for autonomous cars to recognize their environment.

2.2.1 Neural Networks

The ML algorithm that is going to be implemented in this work is a particular type of Neural Network (NN). Neural Networks are inspired by the human brain in a way that they mimic the neurons and their connections in it [22]. Their structure, as shown in Figure 4, is a set of layers composed of interconnected neurons.

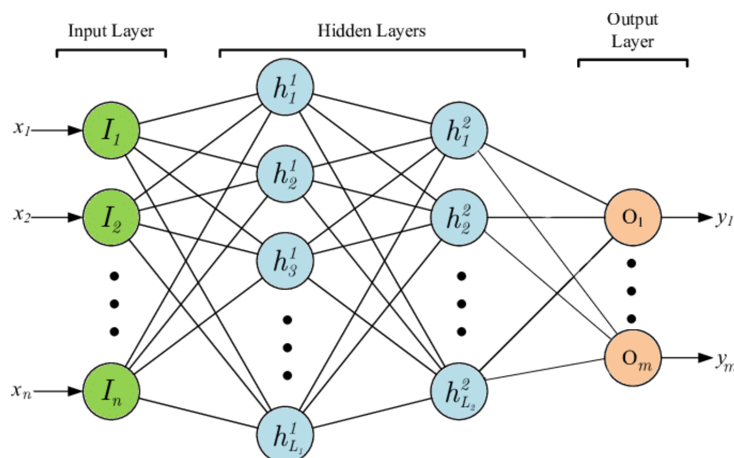

Figure 4: Scheme of a 2 hidden layer neural network with 2 output neurons.

The first layer, the input layer, is used to introduce the data into the NN. The middle layers, called hidden layers, are composed of neurons that perform linear combinations of the previous layer neurons followed by a non-linear activation function. The last layer, the output layer, is also made of neurons trained to perform linear combinations of the

previous, however it depends in weather the network is meant for classification or for regression if an activation function is used or nor. In the case of ELM for classification there is no activation function in the last layer, but a function to find the output neuron with the largest value.

Now let's see how the linear combination and the activation function work in each neuron if the information flows from left to right, what is called a feed-forward network. If $\mathbf{h}^{j-1} = (h_1^{j-1}, h_2^{j-1}, .., h_L^{j-1})$ is the output of the $(j-1)$-th layer, with h_i^{j-1} the output the i-th neuron on it, then, the neurons in the j-th layer perform the following linear combination:

$$\mathbf{w}_i^j \mathbf{h}^{j-1} + \mathbf{b}_i^j \tag{1}$$

where $\mathbf{w}_i^j$ is a matrix containing the trained coefficients for the linear combination, this coefficients are normally called the weights, and $\mathbf{b}_i^j$ are the trained bias, a vector of constants, of the i-th neuron in the j-th layer. If all the neurons performed just linear combinations, then the final result would be a linear combination of the input data and this structure would be nonsense. To avoid this an activation function is performed after each linear combination, it can be a threshold for example, but normally the Sigmoid (S) function is used. The Sigmoid function sends any real value to a real number between 0 and 1, and it is defined as:

$$S(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

One of the most important properties of the Sigmoid function is that $S(-x) = 1 - S(x)$. A graphical representation of this function can be seen in Figure 5.

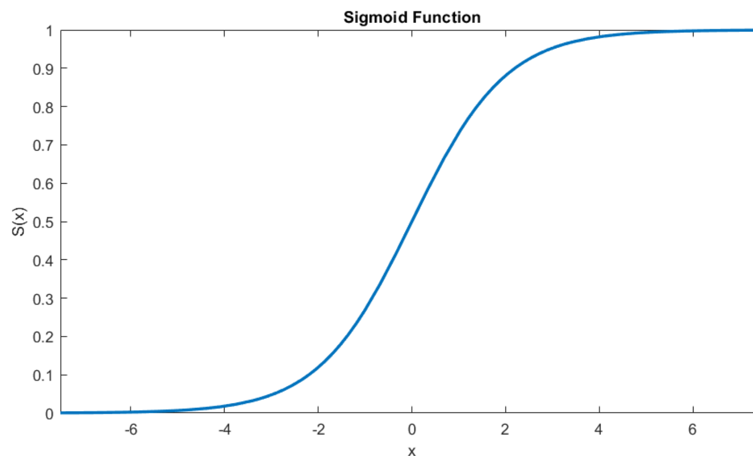

Figure 5: Sigmoid function values $S(x)$ for input values x between -7.5 and 7.5.

With this activation function the output of each i-th neuron in the j-th layer is:

$$h_i^j = S(\mathbf{w}_i^j \mathbf{h}^{j-1} + \mathbf{b}_i^j) \tag{3}$$

The core of a neural network algorithm is its training method, that is determining the weights and bias. One such training method is back-propagation, which computes the gradient-descent to find the weights that minimize the difference between the desired output and the output of the network for the training data [23].

This work will focus on one specific type of neural network that has its own training method, the Extreme Learning Machine neural network.

2.2.2 Extreme Learning Machine

Extreme Learning Machine (ELM) consists of a single hidden-layer feed-forward neural network, see Figure 6. What makes this network different is that the hidden layer is not trained but set randomly, that is, the weights and bias of the neurons in the hidden layer are random numbers in an specific range and only the output layer is trained [11]. This reduces considerably the training time, what lets to the possibility of generating multiple iterations of the network with different random weights, bias or number of neurons.

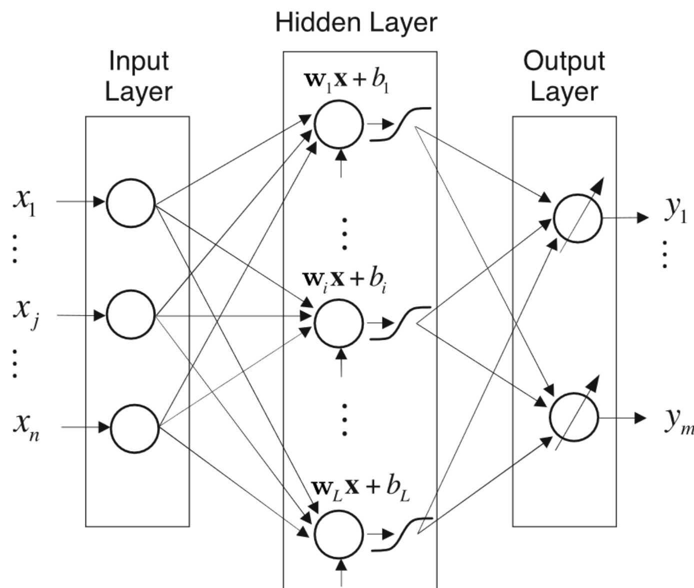

Figure 6: Topology of an Extreme Learning Machine neural network, that is a single hidden-layer feed-forward neural network [3].

The need to only train the output layer makes it possible to implement algebraic batch learning. If we call the output of the i-th neuron in the hidden layer for an input $\mathbf{x}$: $h_i(\mathbf{x}) = S(\mathbf{a}_i\mathbf{x} + \mathbf{b_i})$, note how we don't need the superscript anymore since we only have one hidden layer, and i can range between 1 and the number of hidden neurons L. If the network is trained with K samples we can build a matrix with all the results of the hidden layer, we call this $\mathbf{H}(\mathbf{x})$:

$$\mathbf{H} = \begin{bmatrix} \mathbf{h}(\mathbf{x}_1) \\ \vdots \\ \mathbf{h}(\mathbf{x}_K) \end{bmatrix} = \begin{bmatrix} h_1(\mathbf{x}_1) & \dots & h_L(\mathbf{x}_1) \\ \vdots & \vdots & \vdots \\ h_1(\mathbf{x}_K) & \dots & h_L(\mathbf{x}_K) \end{bmatrix}_{K\times L} \tag{4}$$

The weights of each neuron in the output layer $\boldsymbol{\beta}_j$ is what needs to be trained. We build a matrix $\mathbf{B}$ with this output weighs for each output neuron. Since the inputs are training samples we know their results in each output neuron, this will be a vector $\mathbf{t}$ containing the output targets. In the case of using ELM to classify between classes, the $\mathbf{t}$ will have

all zeros except for that position of the class that corresponds to the input that will be a one, we create a matrix $\mathbf{T}$ containing all these output vectors. Considering m output neurons this two matrices are:

$$\mathbf{B} = \begin{bmatrix} \boldsymbol{\beta}_1 & \dots & \boldsymbol{\beta}_m \end{bmatrix}_{L\times m} \qquad \mathbf{T} = \begin{bmatrix} \mathbf{t}_1 \\ \vdots \\ \mathbf{t}_K \end{bmatrix}_{K\times m} \tag{5}$$

The result $\mathbf{T}$ of the network is the matrix multiplication between $\mathbf{H}$ and $\mathbf{B}$, that is, $\mathbf{T} = \mathbf{H}(\mathbf{x})\mathbf{B}$ and if we know $\mathbf{H}$ and $\mathbf{T}$, it can be solved for $\mathbf{B}$ from here. Since $\mathbf{H}$ is generally non square its inverse can not be calculated and to solve the equation the Moore-Penrose generalized inverse $\mathbf{H}^\dagger$ of it is computed. Generally this generalized inverse is calculated via Singular Value Decomposition (SVD). Finally the output weights can be calculated by multiplying this matrix with the output vectors:

$$\mathbf{B} = \mathbf{H}^\dagger\mathbf{T} \tag{6}$$

It is important to note that when using the trained network as a classifier we will not get output vectors containing all zeros except for a one, so we will consider the index in the output vector with the largest value as the class predicted.

2.3 Hyperspectral Images

Common color cameras capture Reed, Green and Blue light with three relative wide filters. Multispectral Cameras extend the number of filters so that electromagnetic waves from the infrared spectrum are also captured. Hyperspectral cameras go one step further and capture light in multiple narrow adjacent bands so that an spectra is captured for each image pixel [13]. A pixel is the smallest possible element in an image, or the smallest solid angle that a camera detection element can capture.

Hyperspectral Images (HSI) are the images captured with hyperspectral cameras, also called optical spectrometers. Thus each pixel in a hyperspectral image contains information about multiple wavelength lights.

2.3.1 Electromagnetic spectrum: Visible and Infrared light

When talking about imaging it is important to remember that light is an electromagnetic wave. The range of all wavelengths of electromagnetic radiations is called the electromagnetic spectrum (EM spectrum).

We, humans, can only perceive electromagnetic waves, that is, light, with a wavelength between $400nm$ and $700nm$. Therefore we call that range of the EM spectrum the visible spectrum (VIS spectrum). Light with wavelengths grater than $700nm$ (but less than 10^6nm) is called Infrared light (IR) and if it is really close to the visible spectrum it is referred to as Near-Infrared light (NIR).

The VIS and IR spectrum ranges are the most used in hypersectral imaging, however, the hole spectrum is much larger as it can be seen in Figure 17

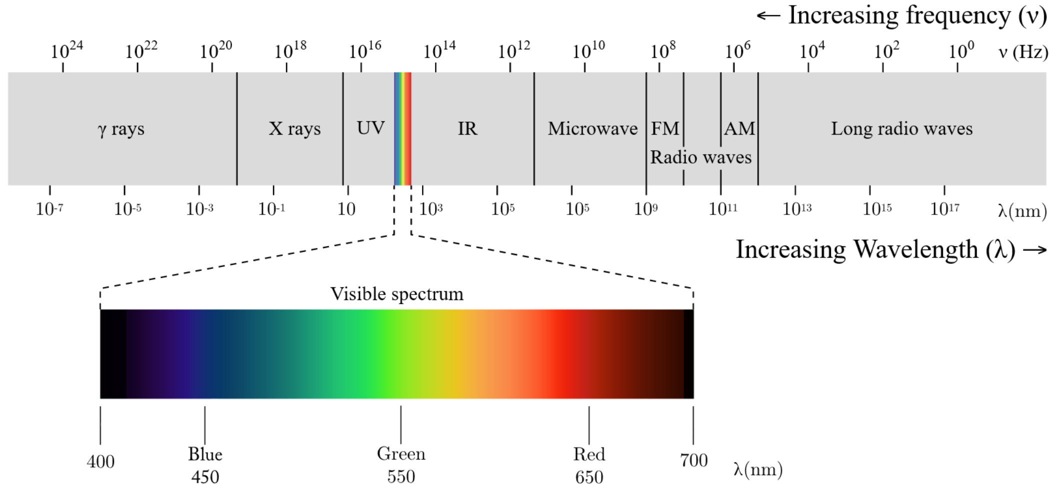

Figure 7: Electromagnetic Spectrum and Visible spectrum. Image modified from Wikimedia commons[4].

2.3.2 Material Reflectance

When light impacts against a material a part of it is absorbed and the rest is reflected, what light is absorbed depends on the light wavelength and the materials properties. For example, chlorophyll 'a' absorbs almost all the red light of about $675nm$, and part of the blue light as well [24], so, when light reaches vegetation with chlorophyll 'a' non light of $675nm$ gets reflected: the reflectance of vegetation in the band of $675nm$ is 0%. If we note down the percentage of light that gets reflected for a material in each wavelength we obtain the materials reflectance signature, which sometimes is also called reflectance spectra.

In Figure 8 we can see the reflectance signature of grass and an olive green paint, even though both materials are green we can clearly see the total absorbance of chlorophyll in grass. This shows how reflectance signatures can be used to distinguish between materials.

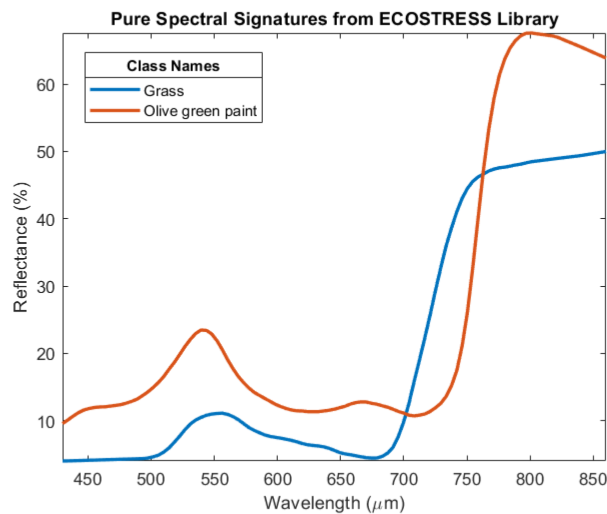

Figure 8: Reflectance signatures of grass and olive green paint in the light range between $430nm$ to $860nm$. Signatures the ECOstress spectral library [5].

2.3.3 Hyperspectral Images for material detection

The idea is to use hyperspectral cameras to generate this reflectance signatures for each pixel in the scene, so that via previously trained machine learning algorithms it can differentiate what materials are present at an image pixel by pixel.

This way each pixel has as many values as electromagnetic wavelengths are captured, normally referred to as bands. Each band value ranges between 0% and 100%, or usually between 0 and 1. This way a three-dimensional array of data can be made, with two space dimension and a third dimension for the wavelength, this is called the hyperspectral data-cube, see Figure 9.

It must be said that the information captured by a hyperspectral camera is not directly a reflectance spectra, some preprocessing is needed [25]. This preprocessing is called radiometric modelling and there are various methods to do it.

In airborne HSI it has been common to use the black body radiation pattern of the sun to calculate the amount of light that reaches the earth, so that the captured amount of light is divided by the emission light in each wavelength. This is called radiative transfer modelling.

Another technique to preprocess the image includes knowing the reflectance pattern of an element in the image so that via linear regressions the reflectance patterns of the rest of the elements are calculated. This is called the empirical line method for radiometric modelling.

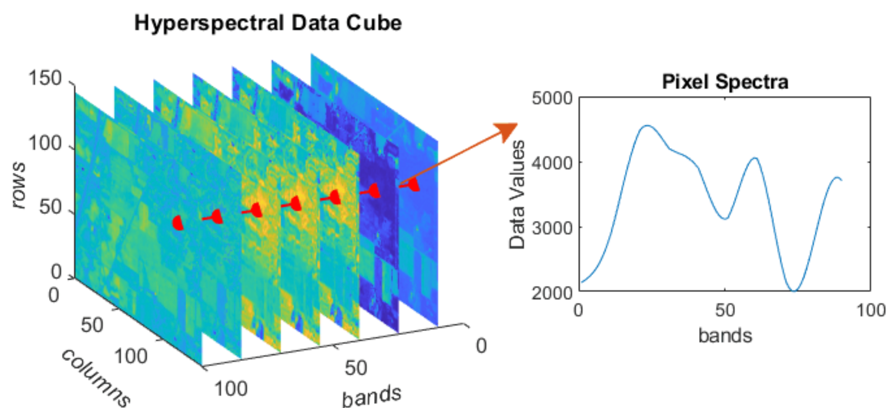

Figure 9: Representation of a hyperspectral data cube and a pixel spectra.

3 Training ELM network for Hyperspectral Imaging

The ELM training is done for a specific type of input, in this case for a specific hyperspectral dataset. The AeroRIT dataset [26] has been chosen here because it is a new dataset that provides all the data needed for the analysis.

3.1 The AeroRIT dataset: train and test sets

The AeroRIT dataset is an image taken in Rochester Institute of Technology's university campus from a flying aircraft. Each pixel captures 51 wavelengths from $400nm$ to $900nm$ each one separated by $10nm$. The image has in total a size of 1973×3975 pixels, a RGB representation of it can be seen in Figure 10a.

Together with the hyperspectral dataset Rangnekar et. al. [26] provide a ground-truth image where 5 different classes are labelled. This way each pixel in the dataset corresponds to one of 5 classes, this is shown in Figure 10b.

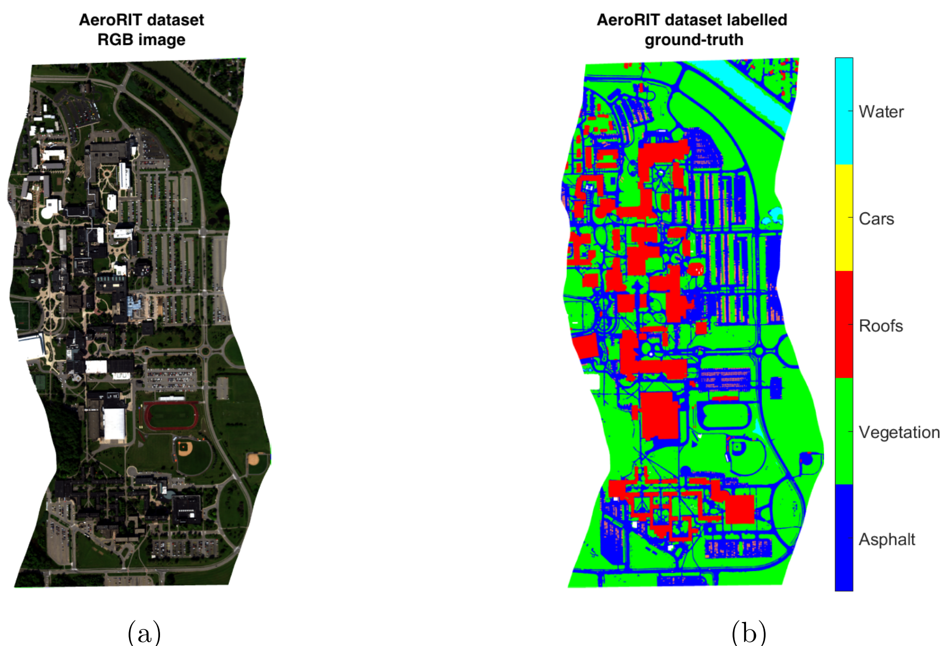

Figure 10: The AeroRIT dataset is an airborne captured hyperspectral dataset that includes an RGB image 10a and a labelled ground-truth 10b that differentiates 5 classes.

When testing classification algorithms it is important to separate a set of inputs to train the classification model and a set of inputs to test the model. This is done so that the model is tested in a set of inputs that it has never seen before, and therefore it is also important that the sets are chosen randomly. This has been done with the AeroRIT dataset, the amount of pixels in each set for each labelled class can be seen in Table 3.

Table 3: Amount of pixels for the train set and test set for each class randomly selected from the AeroRIT dataset, and the total amount of pixels for each class in the dataset. The percentage of training pixels over the total is shown in parenthesis.

	Total Pixels	**Train Pixels**	**Test Pixels**
Asphalt (1)	1946453	3000 (0.15%)	1943453
Vegetation (2)	3191657	3000 (0.09%)	3182657
Roofs (3)	917830	3000 (0.33%)	914830
Cars (4)	132093	3000 (2.27%)	129093
Water (5)	118664	3000 (2.53%)	115664
Total	6306697	15000 (0.24%)	6303697

3.2 Parameter design for ELM network

Before the training of the network is performed the random weights for the neurons in the hidden layer must be set. When generating this random numbers it has been decided to keep them in the $[-1, 1]$ range for simplicity in the posterior hardware implementation.

The number of neurons in the output layer is fixed by the number of classes that the networks needs to distinguish, in the case of AeroRIT, 5 neurons are needed.

On the other hand the number of neurons in the hidden layer is not fixed, hence it is a parameter that needs to be selected. To select this we can iterate over the number of neurons until we find the desired precision for the network.

Increasing the amount of neurons in the hidden layer increases the amount of time needed to train the network. In fact to train ELM (Equation 6) the most time consuming function that needs to be performed is the Moore-Penrose pseudo inverse, if this is done via the SVD decomposition we can estimate a time complexity about $\mathcal{O}(L^3)$, with L the amount of neurons in the hidden layer [27]. Anyway the training is only done once so this time complexity does not affect the networks performance when it is in real-time feed-forward operation.

It is important to note that the amount of neurons in the hidden layer can also increment the amount of time that the network needs to obtain an output for a given input. However, since the objective of this network is to be implemented in a parallel hardware network, this is not such a big issue, instead the hardware resources in the FPGA are the limitation.

The precision over the test set for each trained network for AeroRIT dataset has been represented, in Figure 11 the mean accuracy between 5 ELM network for every 5 neurons added is shown. We can see the precision of the ELM network rapidly increases when the neuron number is low, and an overall accuracy of over 75% is obtained for 20 neurons. With 100 neurons an accuracy of around 84% is obtained and from here the added accuracy grows slowly with each added neuron until it gets saturated in the 87% accuracy, at about 350 neurons in the hidden layer. For a bigger number of hidden neurons a drop in the accuracy is expected, since this is a known behaviour for all neural networks.

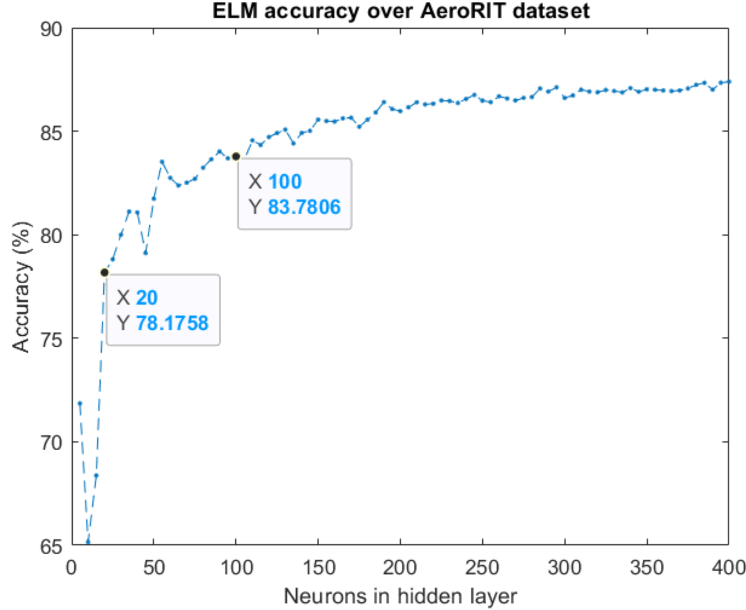

Figure 11: ELM network test accuracy in AeroRIT test set over different number of neurons in the hidden layer.

3.3 ELM accuracy on the AeroRIT dataset

We have decided to stick with 100 neurons in the hidden layer, with their weights in the [-1, 1] range and trained with 3000 pixels for each class in the AeroRIT dataset.

The overall precision of this network over the rest of the pixels in the set is around 85%, however we can analyze how well it behaves for each class. For doing so we compare each classified class with the real labelled class and build a confusion matrix, which shows how many pixels have been classified as each class and the percentage of correctness, Figure 12.

The way to interpret this confusion matrix is the following, from the pixels classified in the ground truth as class 1 (first row) 1536831 have been classified as class 1, 46368 have been classified as class 2 and so on. Then 79.0% of the class 1 pixels have been correctly identified. The lower 2x5 matrix is telling that from all the pixels that the algorithm has classified as class 1 86.8% where of class 1.

We find that the model has great accuracy with classes 1,2 and 5 (Asphalt, vegetation and water) and struggles with classes 3 and 4 (cars and roofs). The biggest problem is found to be in the cars class, 87% of the pixels classified as cars did not correspond to that class and even with this over-classification the model was not capable to find half of the real pixels of class 4. One reason for this might be that roofs and cars are both made from the same metallic materials and therefore the algorithms mixes them during the classification. However, there is no apparent reason for mixing cars and asphalt, since the mean reflectance signatures for 3000 pixels of each class are separated by 5% reflectance, even if both are flat, see Figure 13 where this mean signatures for all the classes are shown.

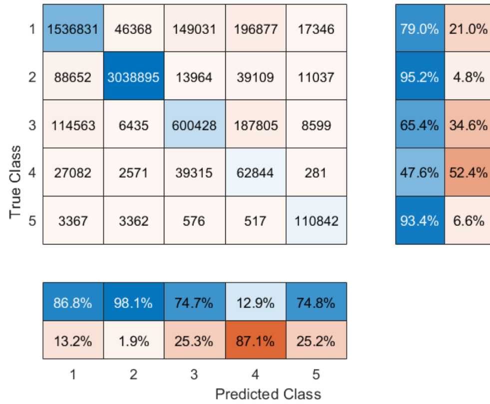

Figure 12: Confusion matrix for ELM application over AeroRIT hyperspectral dataset.

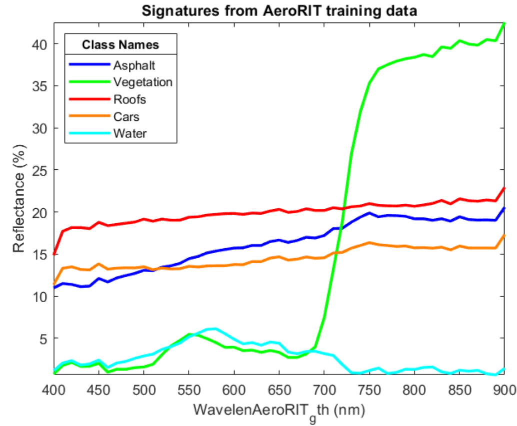

Figure 13: Mean signatures of the 5 classes in AeroRIT for 3000 random pixels from each class

3.4 Principal Component Analysis

A huge problem with hyperspectral images is that the algorithms to analyze them, therefore this also applies to ELM, have computational complexities highly dependant on the number of bands. For this reason it is important to see if all of them are necessary or the dimension of the problem can be reduced without loosing important information.

A way to reduce the dimension is to find a new basis for the data so that in some axis it is spread out maximizing distance, this axis carry most of the information and are called principal components of the data. To find the axis where the data is more distant the most common method is called Singular Value Decomposition (SVD). The new axis obtained are characterized by their variability as a percentage, so that the ones with higher variability can tell apart better between data points. The axis with the least variability can be discarded since they are not useful to differentiate between pixels.

When applying Principal Component Analysis (PCA) to reduce the dimensionality of the AeroRIT dataset, we find that the first principal component holds 75.85% of the variability of the reflectance signatures, and the second holds 23.35%, so that both combined hold 99.20% of the variability. If we plot this components for 1000 pixels of each class, Figure 14, it is found how clusters for each class appear.

To see what bands contribute the most to these principal components we plot the coefficients of the application that transform the original 51 bands to the first principal components, Figure 15 shows this coefficients for the first 3 principal components. All of the bands are used in the new basis, and therefore it can be concluded that some bands are not more important than others. However it is clear that first principal component gives more significance to the infrared region, whereas the second component does it for the visible spectrum.

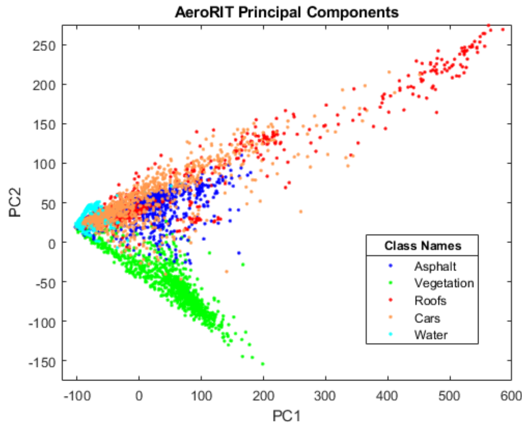

Figure 14: First 2 principal components of the different classes present in AeroRIT.

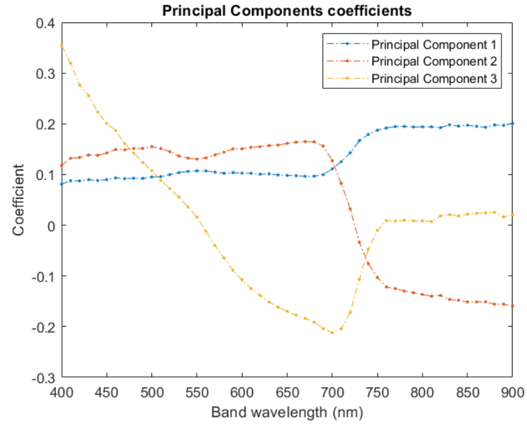

Figure 15: Coefficients of the first 3 principal components in the AeroRIT dataset.

To find how well ELM performs using PCA to reduce the dimensionality, we have calculated the transformation with the train set and applied that same transformation to the test set. Then it has been calculated that 15 components are needed to hold 99.9% of the variability in the train set, so the other 36 components have been discarded. With the resulting train and test sets the ELM algorithm has been applied iterating over the number of neurons in the hidden layer. In Figure 16 the accuracy of the network for a number of hidden neurons multiple of 5 is shown, where the mean between 5 random networks have been taken as the accuracy. It is found that to obtain an accuracy over 80% the number of neurons in the hidden layer needed is 10 neurons only, from the more than 20 that were necessary using the original data. In fact, the amount of neurons for an accuracy of around 85% gets reduced to half of the number needed without PCA, from 100 neurons to 50 neurons.

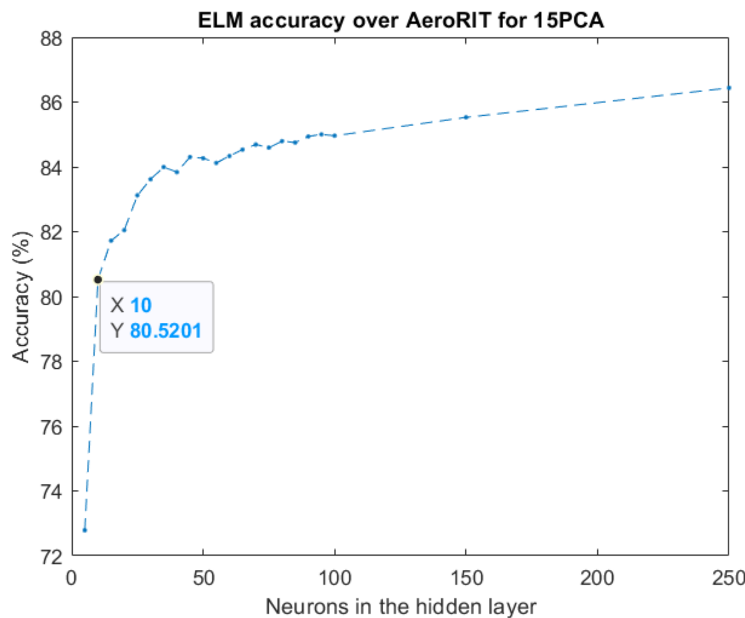

Figure 16: ELM network test accuracy over AeroRIT in the 15 principal components as a function of the number of neurons in the hidden layer.

4 Implementation of ELM network in VHDL

As explained in the previous section, an ELM network is a feed-forward network with a single hidden layer. The Hardware implementation therefore needs to implement the neurons in the hidden layer and in the output layer, the activation function for the hidden layer, and a comparison module to select the highest output from the output layer.

This section shows a design for each of those components and a full ELM network design in VHDL, this codes can be seen in the appendices at the end of the document. However, the hardware necessary to perform the PCA will not be included, and the design only focuses on the ELM network. Furthermore the hardware resources from the Nexys A7 board used by each component and their simulations are analyzed.

The VHDL code is designed so that the lengths of the signals, the number of neurons, the size of the memories, etc. are not fixed and can be changed via a parameter list in a package. The final objective is a parallel ELM network architecture for high-speed real-time classification. Specifically the design will be tested for hyperspectral pixels classification.

4.1 The individual neuron using DSP slices

The most basic block in a neural network is the neuron. A neuron receives an input signal **x** and multiplies it by its weights **w** (Equation 1).

4.1.1 The DSP slice

All Artix 7 series FPGAs include many dedicated, low power digital signal processing (DSP) slices (Table 1). Specifically they include multiple DSP48E slices. This DSP slices include, among others, a 25x18 two's-complement multiplier and an accumulator. The basic functionality scheme of the DSP48E can be seen in Figure 17. Since the DSP is a synchronous device it needs a clock a signal and a clock enable signal.

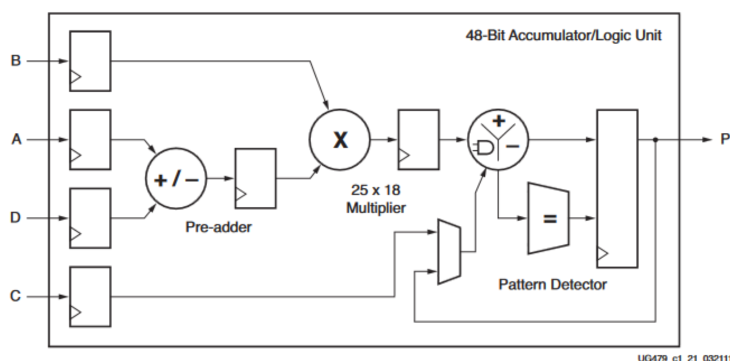

Figure 17: Basic DSP48E1 Slice Functionality

The multiplier has a fixed amount of bits for the inputs. Therefore the precision on the inputs is defined by their number of bits. For this hardware, the numbers will be set in the fixed-point representation in the 2's complement. For this reason the random weights are set in the [-1,1] range, thus, when the sum of products is carried out negative numbers will compensate the positive ones and the final result won't be too large.

4.1.2 Neuron Design in VHDL and behavioral simulation

The neuron calculates the product between two vectors. In a serial architecture this product can be obtained by a sum of products. The DSP is designed to perform a sum of products and therefore is ideal to be used as a Neuron. Following the scheme of the DSP functionality we build the Neuron in VHDL so that when implemented it uses the DSP slice, this code can be seen in Apendix B. The resulting RTL schematic provided by Vivado can be seen in Figure 18. This block will be referred to as *Neuron*.

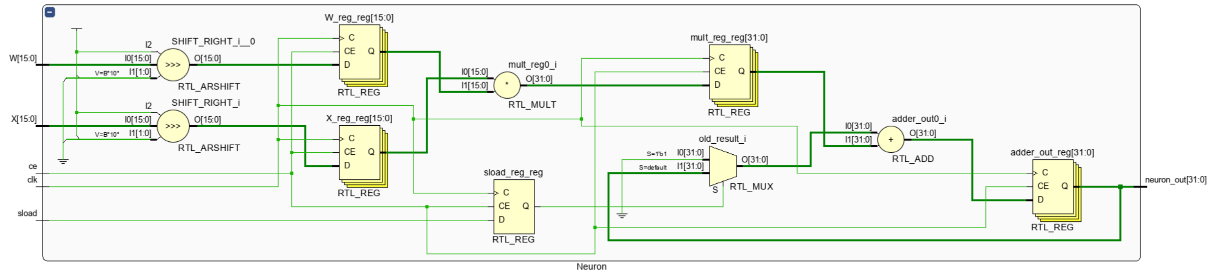

Figure 18: RTL Schematic for a single Neuron.

This Neuron hardware implementation can be used for both the neurons in the hidden layer and the neurons in the output layer. In both cases their weights will be stored in ROMs. However there is one difference between the neurons in the hidden layer and in the output layer, the hidden layer neurons have to add a bias to the result. To use the same neuron in all cases we consider that the bias is just the first weight stored in the ROM and start the input with a '1' so that it gets added to the result.

To verify the functionality of this neuron we performed a behavioral simulation in Vivado for an input of $n = 3$ numbers, this would be a hyperspectral pixel of 2 bands, see Figure 19. For this specific application we have chosen 16-bit 2's complement representation of the weights W and the input X with all the bits except the sign bit for the fractional part in the input, which means a decimal precision of 0.00003. The resulting multiplication and the accumulator (*neuron_out* in the image) are 32-bit long 2's complement numbers with 26 bits for the fractional part and 4 for the integer part, thus 2 decimal bits of precision is lost in the inputs. The amount of integer bits can be chosen for each application.

In the simulation waves we see that the *Neuron* has a delay of 3 clock cycles, that is, each number of the serial input needs 3 clock cycles to get reflected in the accumulator. This is for the 3 registers that can be seen in Figure 18. Thus to get the final result of a neuron 3 clock cycles plus the number of bands of the input are needed. So for an input of n bands, $n + 3$ clock cycles are needed, this is the *Neuron's* latency. The bias would add another cycle delay, however it has been assumed as another weight, and an extra one in the input has been introduced. In fact this is the case for the test bench presented in Figure 19, where the first weight is the bias and therefore it is multiplying the '1' representation in the 2's complement.

Note in the simulation the presence of the *clock_enable* (ce) signal, which must be '1' during the hole process.

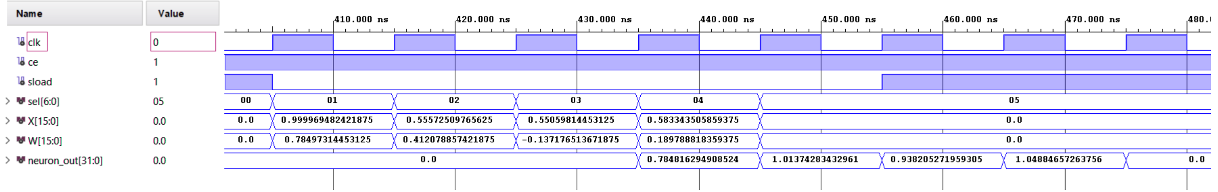

Figure 19: Behavioral simulation of a single neuron for an input of 3 numbers.

4.1.3 Neuron hardware implementation and timing simulation

Since the neuron follows the DSP scheme, when we implement it in VHDL for the Nexys A7 we get a new schematic that uses a single DSP for the neuron, see this in Figure 20 in the next page, where apart from the DSP we find the input/output (I/O) buffers used for reliability and a register *sload* that is used to control weather the neuron is activated for accumulation or not. See the resources used for a single neuron in Table 4.

Table 4: Hardware resources and latency for a single neuron.

	LUT	FF	MUX	DSP	Latency
Neuron	0	1	0	1	m+3

Once the hardware has been implemented there is the possibility to simulate its behaviour as if it was being executed in the FPGA, this is the post place and rout timing simulation. This simulation for the same 3 values that have been simulated previously can be seen in Figure 21. In this simulation we find that apart from the cycle delay commented previously, an added combinational delay appears in the same clock cycle .

The clock cycle used for the simulation has a 10ns symmetric period, this corresponds to a 100MHz clock. with this clock Vivado gives a design timing summary indicating the timing slack, the margin by which a timing requirement is met. This slack must have positive values for a good design and can be seen in Table 5. The Worst Negative Slack gives an idea about how larger the frequency of the clock can get, in this case the period can be reduced to about 4ns which corresponds to a 250MHz clock, the use of DPS allows this fast frequencies.

Table 5: Timing worst slacks for a single neuron.

	Wosrt Negative Slack	Worst Hold Slack	Worst Pulse Width Slack	Maximum Frequency
Neuron	6.647 ns	0.647 ns	4.500 ns	250 MHz

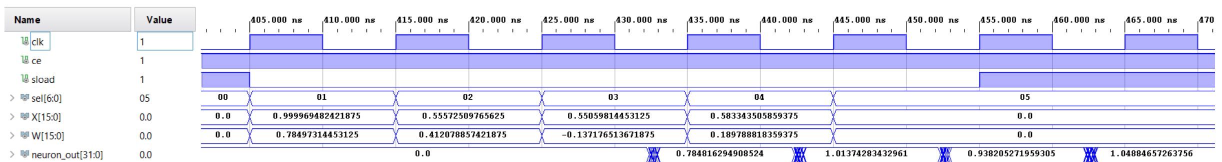

Figure 21: Timing simulation of a single neuron for an input of 3 numbers.

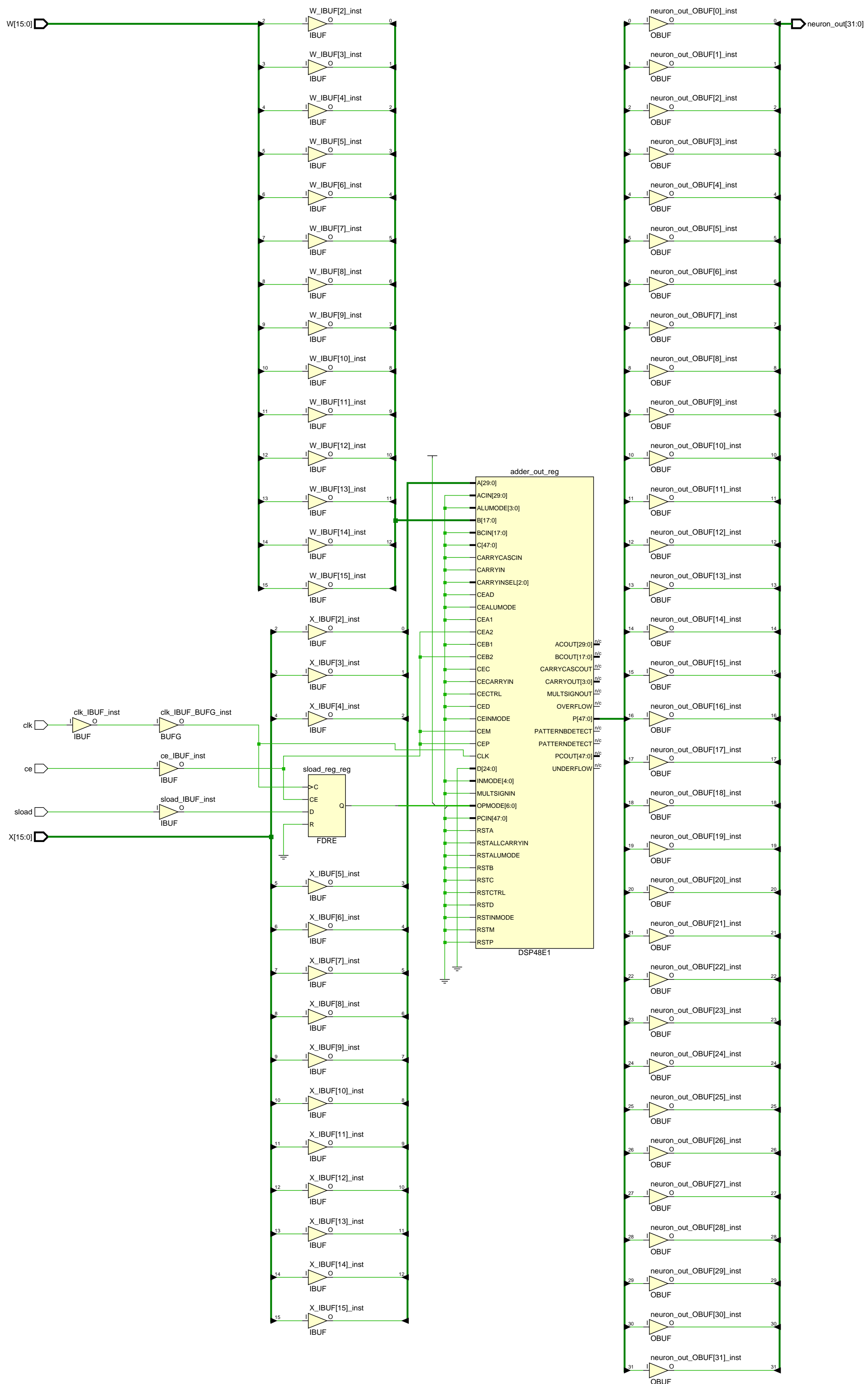

Figure 20: Schematic for a single neuron after implementation, where it can be seen the use of a single DSP, the I/O buffers and the *sload* register.

4.2 Sigmoid function via ROM

The second step, after the sum of products in the hidden neurons, is the activation function. As we have seen in ELM this is usually the Sigmoid function, Equation2.

4.2.1 Sigmoid Function Hardware design

To implement the Sigmoid via hardware one option is to pre-calculate its values and store them in a ROM. Then the output value of the neuron is used as the selection vector in the ROM to obtain the Sigmoid function's value. A ROM is characterized by two parameters, the number of words that it stores, and the number of bits of each word (the word-length). The *Neuron*'s output is 32 bit long, this means that the ROM needs $2^{32} \approx 4.3 \cdot 10^9$ words stored. To reduce this number the neuron output will be cropped to 16 bits.

We choose a precision for the Sigmoid input and calculate in MATLAB the results for the expected range. To reduce the size of the ROM we use the property shown in Equation (7) for the negative values of the Sigmoid function.

$$S(-x) = 1 - S(x) \tag{7}$$

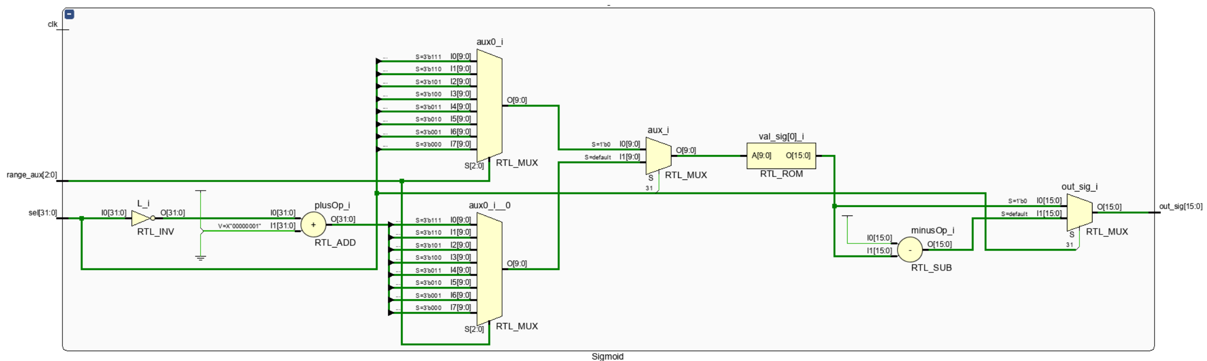

Figure 22: RTL schematic for the Sigmoid function.

Note that the Sigmoid function is not synchronized with the clock signal, see Figure 22 were the resulting RTL from the VHDL code in appendix C is shown. This means that it does not add clock cycle delays, however, combinational delays are still present.

The hardware resources needed to implement one Sigmoid function can be seen in Table 6. Since the Sigmoid always appears with a neuron block, and never by itself, the simulation will be analyzed by considering a new block made by the neuron and the Sigmoid function: the hidden neuron.

4.2.2 Hidden Layer Neurons design

We can therefore build a new VHDL block, appendix D, for the neurons in the hidden layer with a neuron followed by the Sigmoid, see the RTL in Figure 23, we call this new block *Neuron_Sigmoid*. The Sigmoid range specification and a final register are added to the block so that the output is synchronized with the clock, this adds one cycle delay.

The functional behavioral simulation for the same input that has been used for the simple

neuron simulations is shown in Figure 24. In this simulation we can see the new cycle delay from the final register. In this simulation we can also see how for the output of the neuron the correct Sigmoid values is obtained ($S(1.04885) = 0.740036$).

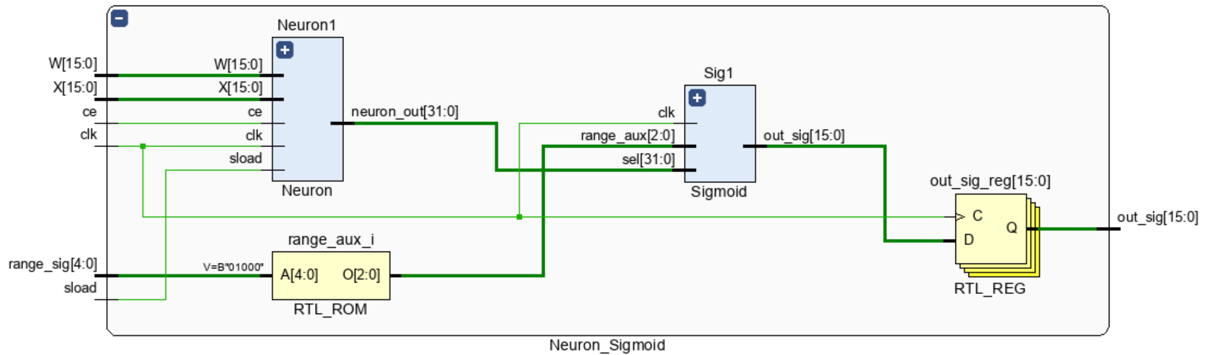

Figure 23: Schematic for the hidden neuron made by a neuron and the Sigmoid function.

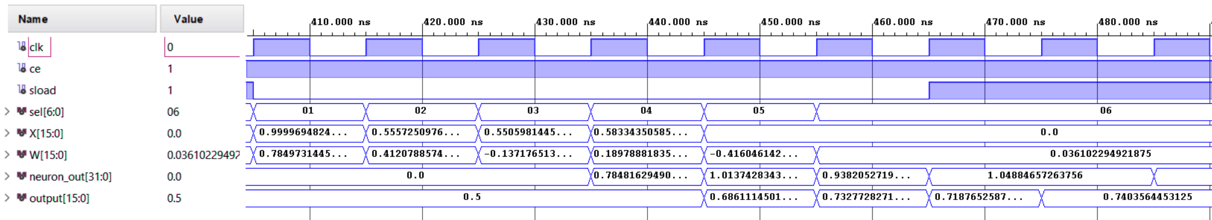

Figure 24: Behavioral simulation of a neuron and the Sigmoid function.

4.2.3 Hidden layer neurons implementation

As said, this new block connects the output of the hidden neuron to the input of the Sigmoid function. It has been chosen that it is in this new block where the range of the Sigmoid is specified, so the resources of the final block are not the sum of the resources in a neuron and in the Sigmoid. The hardware resources once this full block is implemented can be seen in Table 6 as well as the latency of the block, which depends on the number of signals n in the input.

A post implementation timing simulation has also been carried out, see Figure 25. In this simulation we once again find the combinational delays inside the clock cycles. The worst timing slacks are shown in Table 7 where we find that for a $100MHz$ clock they are all positive and therefore should perform good. However introducing a ROM considerably reduces the slack and the maximum frequency of *Neuron_Sigmoid* is 111 MHz.

Table 6: Hardware resources and the clock delays for the Sigmoid function and a hidden neuron (made from a single neuron and the Sigmoid function).

	LUT	FF	MUX	DSP	Latency
Sigmoid	261	67	89	0	0
Hidden Neuron	264	17	89	1	$n+4$

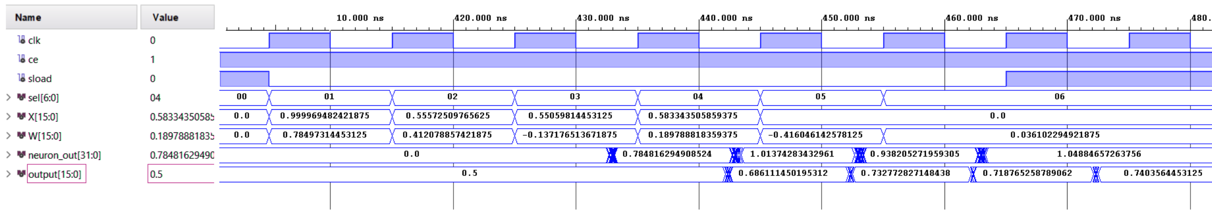

Figure 25: Post implementation timing simulation for a neuron and the Sigmoid function.

Table 7: Timing slacks for a neuron followed by the Sigmoid function.

	Worst Negative Slack	Worst Hold Slack	Worst Pulse Width Slack	Maximum Frequency
Neuron	1.586 ns	0.710 ns	4.500 ns	111 MHz

4.3 Comparator

The final step in the network is to compare the outputs of the output layer neurons to find the largest one, so that the class represented by this neuron is assigned to the input.

This is the only block in the network design that won't be general, that is, it will need to be designed for an specific amount of output neurons. This is because it is assumed a low number of output neurons and therefore a free design can result in a faster non synchronized comparator.

Figure 26 shows the RTL schematic for a 5-neuron comparator. This final VHDL block is called *Comparator* and it is not synchronized with the clock, i.e. it is a combinational circuit. From the post place and route simulations we can estimate a combinational delay for this comparator to be less than $2ns$. This delay can get larger for larger comparators.

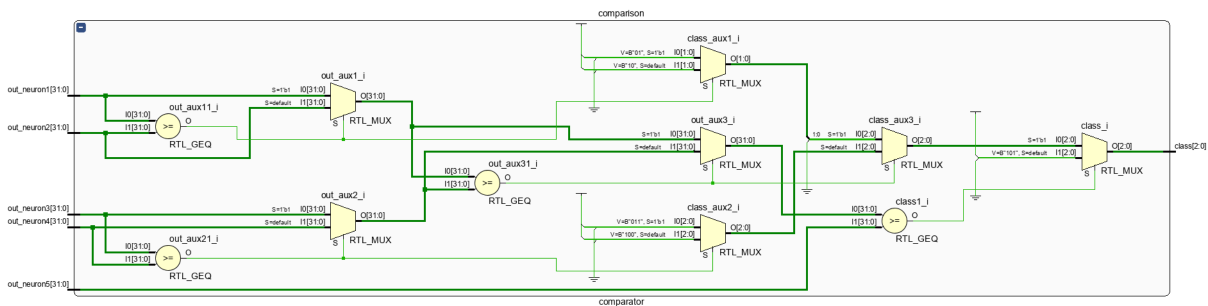

Figure 26: RTL Schematic for a comparator if the output layer has 5 neurons.

Table 8: Hardware resources and the clock delays for a 5 output neurons comparator.

	LUT	FF	MUX	DSP	Latency
Comparator	178	0	0	0	0

4.4 Layout of a parallel ELM architecture

With all the individual blocks designed in VHDL the final step is to build a full ELM Neuronal Network. This is just an iterative call for the blocks needed and their connections.

The complete ELM architecture here proposed is designed so that the input enters the network sequentially and that the neurons in each layer operate in parallel. The general architecture is as follows: The values for an input enter the network sequentially and go directly to all the neurons in the hidden layer at the same time. The hidden neurons compute their results in parallel and the final results get stored in a RAM. The values stored in this RAM are sequentially taken as the inputs for the output neurons, which again compute their results in parallel. The final results in the output neurons get to the combinational comparator which gives a final class for the input.

For the network to start the processing a clock enable signal is added to the architecture. This signal is active high, this means that its default value is 0 and needs to be set to 1 when the first value of the input sequence enters the network.

4.4.1 10 hidden neurons neural network

We have seen that for AeroRIT approximately 100 neurons are needed in the hidden layer. However, to describe the architecture, and to show a legible schematic, a network with 10 neurons in the hidden layer is presented. As it has been stated, the network has been coded in a way that the amount of neurons in the hidden layer can be changed with a parameter and the ROM for the hidden weights.

The resulting schematic for the 10 hidden neurons network can be seen in Figure 29 at the end of this section. To the left of the schematic we find the clock, clock enable signal and the input. These travel through the combinational path to the ROMs with the weights for the hidden layer and to the neurons in the hidden layer. The output of the neurons in the hidden layer go to a block of registers followed by a multiplexer, this block acts like a RAM. Then the output of the multiplexer goes to the output neurons, where the stored values get multiplied with the weights stored in other ROMs. Finally we see how the final results of the output neurons get stored in some registers that are used as the input for the comparator.

The hardware resources that are needed for such a network to be implemented in an FPGA of the 7-series family of Xilinx can be found in Table 9. Comparing these to the hardware resources availabe in the Artix-7CS324XC7A100T (Table 2) we can confirm that this network can in fact be implemented in the Nexys A7 board.

From the schematic we can also calculate the latency for any network. Apart from the counter there are 3 layers of registers in the network, adding each of them one cycle delay. Taking into account the latencies for the *Hidden_Neuron* and the *Neuron* in Tables 4 and 6, and if the number of elements in the input is n and the number of hidden neurons is L, then the total delay is:

$$\text{Network Delay} = L + 3 + n + 4 + 3 = L + n + 10 \qquad (8)$$

This latency for a 52 values input in this 10 hidden neurons design can be seen in Figure 27, which is a post-synthesis functional simulation for the 10 hidden neurons network. In this waves simulation we can also see the presence of the *clock_enable* signal, this signal must be a '0' when the network is not operating and turned to a '1' when the first data of the input is sent to the device. This signal is not a reset, so, it would be useful to add a reset signal in future versions of the network.

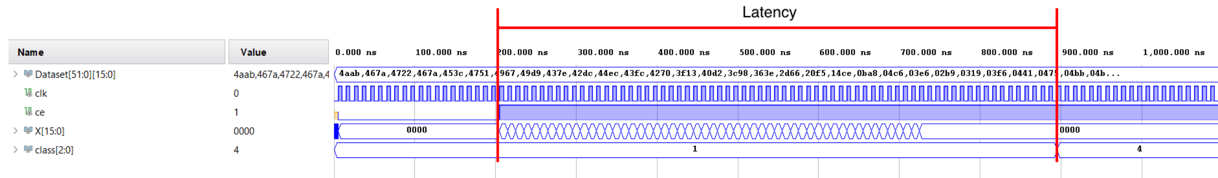

Figure 27: 10 hidden neurons ELM network post synthesis functional simulation.

4.4.2 ELM hardware network application to HSI

With this general ELM network architecture, see the VHDL in appendix E, the final step is to test for an HSI application. For this the model trained for the AeroRIT dataset with 100 neurons in the hidden layer has been used, Section 3.3.

After the synthesis has been performed, when showing the resources the design needs, see Table 9, we find that the amount of Look Up Tables (LUT) needed is 258% of the total LUTs available in the Artix-7CS324XC7A100, see Table 2. Consequently the implementation can not be carried out for this FPGA and a larger one, such as the Artix-7xcvu13p-fhga2104-3-e would be needed for such a network. It has been found that the largest number of hidden neurons that can be implemented in the Artix-7CS324XC7A100 is 60, with 99% of the LUTs in usage.

Table 9: Hardware resources and latency for multiple ELM networks.

	LUT	FF	MUX	DSP	Latency
10 hidden neurons NN	4505	471	1020	15	72 clock cycles
100 hidden neurons NN	163708	3082	72726	105	162 clock cycles

A post-synthesis simulation for an input of a vegetation test pixel (class 2) is shown in Figure 28, which, in fact, gives as a result a 2.

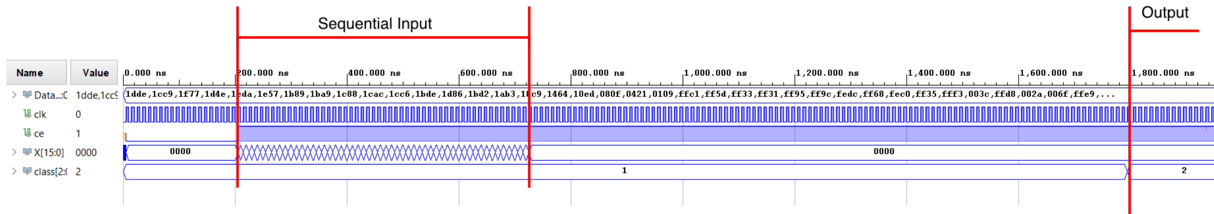

Figure 28: 100 hidden neurons ELM network post synthesis functional simulation.

When analyzing the timing slacks (Table 10) it is found that the worst negative slack is less than $1ns$. This suggest that the clock can not be much faster than the current 100 MHz. In fact, if we tried to minimize the Worst Negative Slack, thus reducing the clock's period to $9.3096ns$, the resulting frequency would be of 106 MHz but there would be great risk of the device not working properly. This means there would be great risk for little speed gained.

Table 10: Timing worst slacks for a full ELM network with 100 neurons in the hidden layer.

	Wosrt Negative Slack	**Worst Hold Slack**	**Worst Pulse Width Slack**	**Maximum Frequency**
Neuron	0.604 ns	0.260 ns	4.500 ns	106 MHz

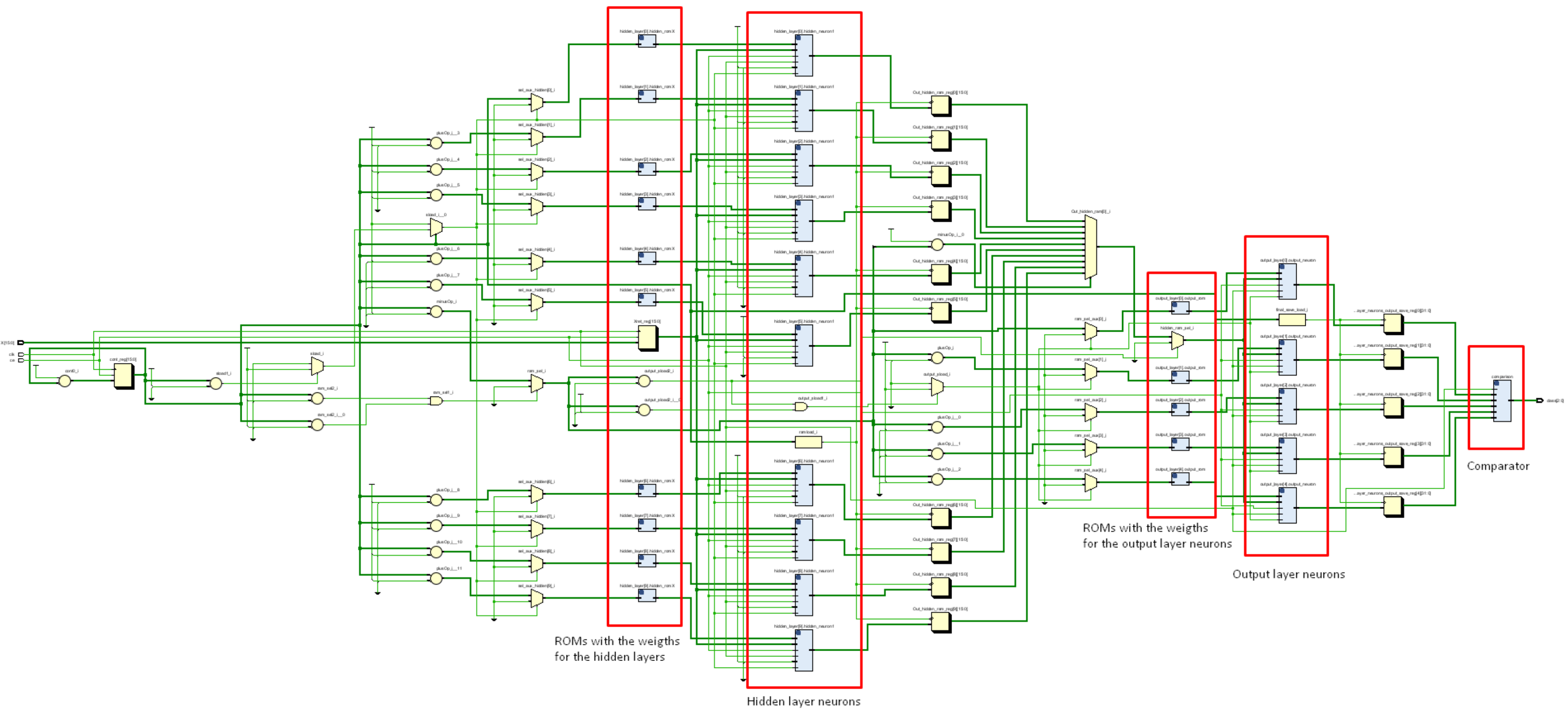

Figure 29: RTL schematic for an ELM neural network with 10 neurons in the hidden layer and 5 neurons in the output layer.

5 Power consumption analysis

Since FPGAs where first introduced in 1984 by Xilinx [28] clock frequency and logic gates density has continuously been increasing [29]. This has led to an increase in the power consumed by these circuits, which increases operating temperature. As a consequence the annual expenses in cooling systems of data centers and telecommunication operators has increased [30]. Furthermore, in the global context of climate change the capability to analyze and optimize ICs power consumption has become critical.

The Vivado Design Suit by Xilinx is one of the few FPGA designing software with the capability to analyze and optimize the power consumption of the circuit that is being designed. Here the power consumed by the neural network will be analyzed, but first the main terminology and concepts of FPGA power consumption are introduced.

5.1 FPGA power consumption terminology

The power consumption of an FPGA is mainly due to two factors [31]. On the one hand **Device Static Power** represents the intrinsic power of a circuit due to its transistors leakage currents, it is the power consumed in the standby mode of the programmed circuit. On the other hand **Dynamic Power** is the power consumed when the hardware is operating, that is when the signals are toggling between states, and therefore varies with time. The sum of both of these powers is called **Total On-Chip Power**.

In Electronics power consumption is highly related with temperature. Any FPGA manufacturer guarantees for each device a range of temperatures for the device to operate as expected. Out of this range proper operation is not guaranteed and the device can be damaged. To track the temperature of the device in operation the **Junction Temperature** (T_j) is used, measured in Celsius degrees (°C) and calculated as:

$$T_j = T_{amb} + P_{on-chip} * \Theta JA \tag{9}$$

where T_{amb} is the ambient temperature, $P_{on-chip}$ the Total On-Chip Power and ΘJA is the **Effective Thermal Resistance to Air** (°C/Watt) which defines how power is dissipated from the device silicon to the environment.

The difference between the maximum Junction Temperature supported by the device and the actual Junction Temperature is the **Thermal Margin**, which can be given both in Celsius degrees or in power units.

ΘJA usually depends on the air flow velocity, and can be reduced with the use of a heat-sink. Thermal data defining ΘJA for Xilinx devices for different air velocities can be found using the *Package Thermal Data Query Tool* [6]. This data for the Artix-7CS324XC7A100T is shown in Table 11. As expected, the higher the air velocity the lower the Effective Thermal Resistance.

Table 11: Effective Thermal Resistance to Air for the Artix-7CS324XC7A100T depending on air velocity [6].

Air Velocity (m/s)	0	1.27	2.54	3.81
ΘJA (°C/Watt)	18.2	14.1	13	12.3

5.2 Power analysis in the Vivado Design Suite

It is challenging for the software tools to estimate the power consumed accurately. Consequently, guiding the tools as much as possible can minimize the assumptions made by the software, thus obtaining a more accurate estimation. Here two power simulation approaches are presented [32] for the synthesized design of a 10 hidden neurons NN with 5 output neurons and in the next section for other networks.

For all the power estimations in this work it will be specified that the clock is running at $100MHz$, at an ambient temperature of 25°C, with no heat-sink and null air velocity. This is done as a guide for the software to obtain more accurate results.

5.2.1 Vectorless Vivado Power Report

The first approach is the vectorless propagation engine. This engine predicts the switching activity of the designed elements where no activity is provided from simulation results. Thus this simulation stage only needs the implemented netlist. However it is common for the users to specify the clocks frequency and the expected ambient temperatures for more accurate results.

The estimated power consumption for the 10 hidden neurons network given by the power report provided by Vivado is shown in Figure 30. We find that most of the $0.144W$ power consumed is estimated to be Static Power. It is worth noting how the 15 DSP slices combined consume less than $0.001W$, in fact, most part of dynamic power (30%) is due to the toggling of the signals. Logic Slices consume 30% of the dynamic power and the Clocks 28% of it.

With all it is found that the Junction Temperature is only 2.1°C bigger than the ambient temperature, and that there is 4 extra Watts to be consumed by the device before its performance is compromised. However the confidence level of this estimation is low, and is just an approximation.

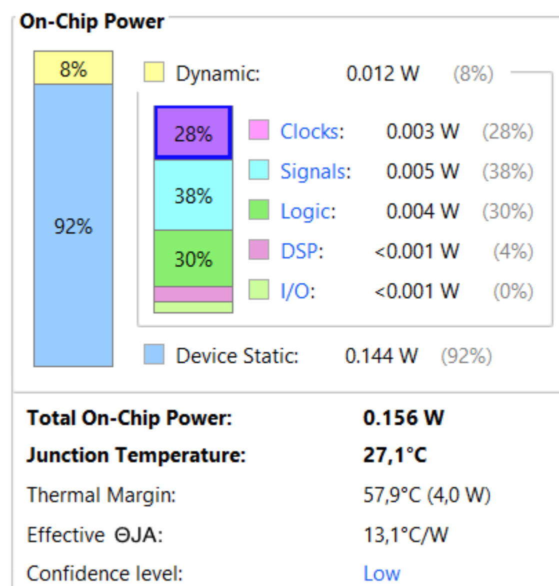

Figure 30: Vectorless power estimation report for a 10 hidden neurons ELM NN.

5.2.2 Vector (SAIF) Based Power Analysis

The second approach for a power estimation is to introduce the results of a simulation for the design. Any simulation in Vivado generates a switching activity values file (SAIF) that can be used to improve the power simulations.

The .saif file has been created for the simulation shown in Figure 27, this file is provided as the switching activity for the signals in the circuit. The resulting power report provided by Vivado can be seen in Figure 31. It is found that Static Power has a similar value to that obtained in the vectorless mode, but Dynamic Power is nearly 5 times that first estimation. Furthermore with the simulation data the power consumed by the DSP slices increases to 0.004W.

Nevertheless, for both vectorless and .saif aided power estimations there is a Thermal Margin of more than 57°C, or a margin on the total On-Chip Power of 4.0W which is 20 times the power being consumed currently.

To see the effect of the clock frequency the same power estimation has been performed for a 1MHz clock. The power report is shown in Figure 32, and it is found that the Dynamic Power is reduced from 0.053W to 0.005W. It is found that the frequency change in the clock does not affect the static power.

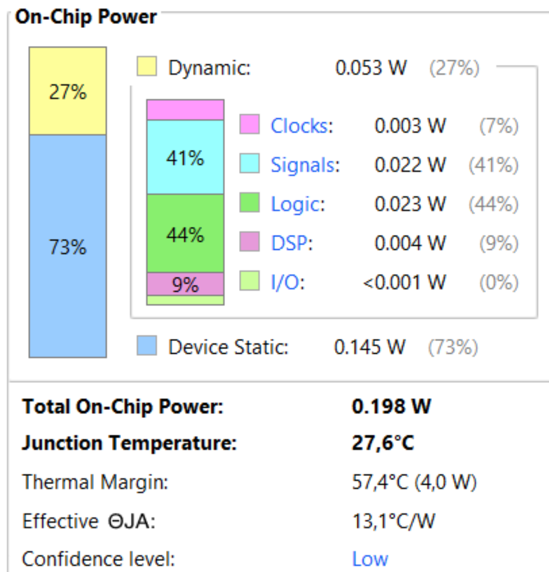

Figure 31: vector based power estimation report for a 10 hidden neurons ELM NN using post implementation functional simulation for 100MHz clock..

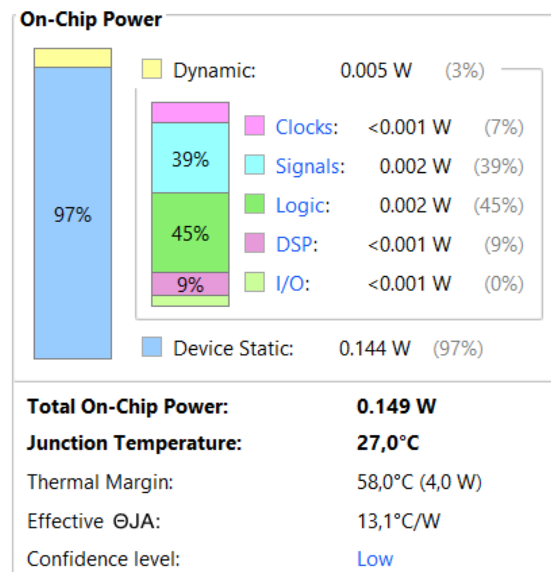

Figure 32: vector based power estimation report for a 10 hidden neurons ELM NN using post implementation functional simulation for 1MHz clock.

5.3 Power for different amount of hidden neurons

To find out how the amount of neurons in the hidden layer affects the power consumption of a network, we have performed the Vector Based Power Analysis for the 5 output neurons with different amount of neurons in the hidden layer.

All the power estimations shown in this section are using the post synthesis netlist and functional simulations. For all the cases the clock's frequency is $100MHz$, the ambient temperature has been set to 25°C, without heat-sink and with null air velocity.

The total power consumption for 10, 25, 50, 60, 75,100 and 200 neurons in the hidden layer can be seen in Figure 33. We find that the Total Power (P) consumed increases linearly with the amount neurons in the hidden layer (L), in fact the slope is $(0,015 \pm 0,002)$ extra Watts per neuron added. However this increase in power is not caused by the extra DSP that each neuron adds, but it is mostly due to the power consumed by other logic devices and by the signals' toggling.

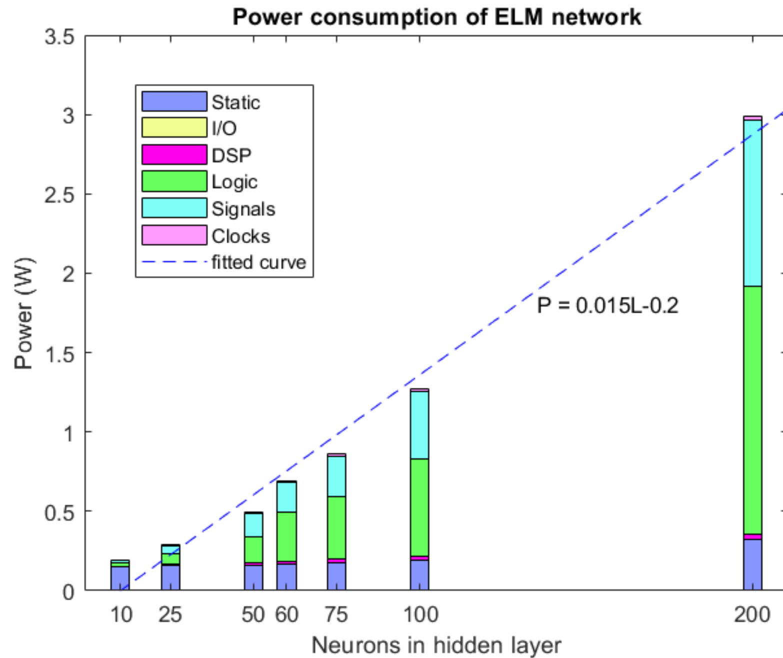

Figure 33: ELM network total power consumption (P) as a function of the number of neurons in the hidden layer (L).

6 General Conclusions

The main objective of this work has been to implement a digital circuit for real-time classification of hyperspectral images. Thus, some digital technologies have been studied from the full-custom devices to the semi-custom FPGAs that are used in the work. The use of hyperspectral images for material detection has been introduced and the usage of Machine Learning (ML) for such a duty explained. In particular emphasizing in the Neural Networks (NN) paradigm.

Specifically the Machine Learning algorithm that has been implemented has been an Extreme Learning Machine (ELM) Neural Network. This is a one hidden layer feed-forward network whose coefficients are set randomly and not trained. ELM has proved to be capable of identifying more than 75% of the pixels with 20 neurons and around 85% of them with 100 neurons when applied to the AeroRIT dataset. However, it is let for future studies to find out why it only detects half of the pixels corresponding to cars in the scene, and how this could be improved, maybe using Depp Learning Machine algorithms.

The use Principal Component Analysis (PCA) has proved to be useful to reduce the amount of neurons in the hidden layer to half of what had previously been needed for the same accuracy. Thus reducing the dimensionality of the problem could be reduced to a third of the values needed for the analysis.

The ELM digital architecture has been designed in VHDL, starting from the smallest element in a network, which is a neuron. If well designed for the Xilinx 7-series family FPGAs, when the design is implemented each neuron uses a single Digital Signal Processing (DSP) unit. The behaviour of the neuron has been simulated, and we have found that it has a $3 + n$ cycles latency if n is the number of elements in the input. It must be said that the resulting neuron can be used for any neural network and not only for ELM. Then the activation function, which follows the hidden neurons, has been designed using a ROM to store its values.

Later, we have designed a 5 input comparator to complete an ELM architecture for a 1 hidden layer network, which has been tested for the AeroRIT dataset. The network has been designed to receive a serial input and process it in all the neurons in parallel. Thus, if the serial input is n elements long and the network has L hidden neurons the total latency of the network is $L + n + 10$. The resources needed for the network have been analyzed to found that in the Artix-7CS324XC7A100T the maximum number of neurons that can be implemented is 60, for its limited amount of LUTs.

Finally, the power consumption of the designed architecture in the Xilinx 7-series FPGAs has been analyzed, after introducing the main concepts related this. It has been found that the power consumed by the FPGA increases linearly with the amount of neurons in the hidden layer as 0.015 Watts per neuron on it. Most of the consumption has been in the signals toggling and in the logic devices, this might be for the large use of LUTs but further research is needed to prove this and, in all cases, to find a solution.

With all, in this project, a working VHDL code for an ELM Neural Network has been achieved, which has proved to be reliable for material detection via Hyperspectral Im-

ages.

References

[1] D. Punia, "FPGA design, architecture and applications," Dec 2021. [Online]. Available: https://www.logic-fruit.com/blog/fpga/fpga-design-architecture-and-applications/

[2] "7-series architecture overview - xilinx.eetrend.com," 2013. [Online]. Available: http://xilinx.eetrend.com/files-eetrend-xilinx/forum/201509/9204-20390-7_series_architecture_overview.pdf

[3] I. del Campo, M. V. Martinez *et al.*, "A versatile hardware/software platform for personalized driver assistance based on online sequential extreme learning machines," *Neural Computing and Applications*, vol. 31, 12 2019.

[4] P. Ronan, "EM Spectrum," 2007. [Online]. Available: https://commons.wikimedia.org/wiki/File:EM_spectrum.svg

[5] S. K. Meerdink, S. J. Hook *et al.*, "The ecostress spectral library version 1.0," *Remote Sensing of Environment*, vol. 230, p. 111196, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0034425719302081

[6] Xilinx, "Package thermal data query." [Online]. Available: https://www.xilinx.com/cgi-bin/thermal/thermal.pl

[7] J. Bardeen and W. H. Brattain, "The transistor, a semi-conductor triode," *Phys. Rev.*, vol. 74, pp. 230–231, Jul 1948. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRev.74.230

[8] G. E. Moore, "Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff." *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 33–35, 2006.

[9] V. Berman, "Standard verilog-vhdl interoperability," in *International Verilog HDL Conference*, 1994, pp. 2–9.

[10] E. Alpaydin, *Introduction to Machine Learning*, 2nd ed., ser. Adaptive Computation and Machine Learning Series. London, England: MIT Press, Dec. 2010.

[11] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, "Extreme learning machine: Theory and applications," *Neurocomputing*, vol. 70, no. 1, pp. 489–501, 2006, neural Networks. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231206000385

[12] A. F. Goetz, "Three decades of hyperspectral remote sensing of the earth: A personal view," *Remote Sensing of Environment*, vol. 113, pp. S5–S16, 2009, imaging Spectroscopy Special Issue. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S003442570900073X

[13] R. B. Smith, *Introduction to: Hyperspectral Imaging*, MicroImages Inc., https://www.microimages.com/documentation/Tutorials/hyprspec.pdf, 2012.

[14] L. Giannoni, F. Lange, and I. Tachtsidis, "Hyperspectral imaging solutions for brain tissue metabolic and hemodynamic monitoring: Past, current and future developments," *Journal of Optics*, vol. 20, p. 044009, 03 2018.

[15] S. S. C and G. R, "Onboard target detection in hyperspectral image based on deep learning with FPGA implementation," *Microprocessors and Microsystems*, vol. 85, p. 104313, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933121004749

[16] S. Liu, R. S. W. Chu *et al.*, "Optimizing cnn-based hyperspectral image classification on FPGAs," in *Applied Reconfigurable Computing*, C. Hochberger, B. Nelson *et al.*, Eds. Cham: Springer International Publishing, 2019, pp. 17–31.

[17] "What is an FPGA? field programmable gate array." [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html

[18] "Xilinx 7 series FPGAS." [Online]. Available: https://www.xilinx.com/publications/prod_mktg/7-Series-Product-Brief.pdf

[19] R. P. Weicker, "Dhrystone benchmark: Rationale for version 2 and measurement rules," Aug 1988. [Online]. Available: https://www.netlib.org/benchmark/dhry-c

[20] "Nexys A7: FPGA trainer board." [Online]. Available: https://digilent.com/shop/nexys-a7-fpga-trainer-board-recommended-for-ece-curriculum/

[21] *Nexys A7™ FPGA Board Reference Manual*, Digilent, 7 2019. [Online]. Available: https://digilent.com/reference/_media/reference/programmable-logic/nexys-a7/nexys-a7_rm.pdf

[22] IBM Cloud Education, "What are neural networks?" [Online]. Available: https://www.ibm.com/cloud/learn/neural-networks

[23] C. Li, G. Chen *et al.*, "A novel high-performance deep learning framework for load recognition: Deep-shallow model based on fast backpropagation," *IEEE Transactions on Power Systems*, vol. 37, no. 3, pp. 1718–1729, 2022.

[24] K. Ballschmiter and J. J. Katz, "Long wavelength forms of chlorophyll," *Nature*, vol. 220, no. 5173, pp. 1231–1233, Dec. 1968. [Online]. Available: https://doi.org/10.1038/2201231a0

[25] M. Vidal and J. M. Amigo, "Pre-processing of hyperspectral images. essential steps before image analysis," *Chemometrics and Intelligent Laboratory Systems*, vol. 117, pp. 138–148, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0169743912001220

[26] A. Rangnekar, N. Mokashi *et al.*, "Aerorit: A new scene for hyperspectral image analysis," *IEEE Transactions on Geoscience and Remote Sensing*, vol. PP, pp. 1–9, 04 2020.

[27] X. Li, S. Wang, and Y. Cai, "Tutorial: Complexity analysis of singular value decomposition and its variants," *arXiv: Numerical Analysis*, 2019.

[28] "Xilinx, inc. history." [Online]. Available: http://www.fundinguniverse.com/company-histories/xilinx-inc-history/

[29] R. W. Keyes, "Miniaturization of electronics and its limits," *IBM Journal of Research and Development*, vol. 44, no. 1.2, pp. 84–88, 2000.

[30] Lattice Semiconductor, "Power considerations in FPGA systems," Feb 2009. [Online]. Available: https://www.latticesemi.com/-/media/LatticeSemi/Documents/WhitePapers/NZ/PowerConsiderationsinFPGADesignLatticeECP3.ashx?document_id=32410

[31] "Vivado design suite user guide: Power analysis and optimization (ug907)," Jun 2020. [Online]. Available: https://docs.xilinx.com/v/u/2020.1-English/ug907-vivado-power-analysis-optimization

[32] "Vivado design suite tutorial power analysis and optimization (ug997)," Jul 2020. [Online]. Available: https://docs.xilinx.com/v/u/2020.1-English/ug997-vivado-power-analysis-optimization-tutorial

Appendices: VHDL codes

In the following appendices the VHDL codes that have been developed are shown:

- **Appendix A** is a package with the constants that define the network, such as, the number of elements of each input, the number of neurons in the hidden layer and the number of neurons in the output layer. This package also contains the word-length of the main signals.
- **Appendix B** shows the VHDL code for the *Neuron* block following the DSP scheme.
- **Appendix C** Contains the ROM implementation of the *Sigmoid* block for the implementation of the Sigmoid activation function.
- **Appendix D** is the VHDL code for the *Neuron_Sigmoid* block.
- **Appendix E** contains the VHDL code for the complete implementation of an ELM neural network.
- **Appendix F** is the general structure of the ROMs for the weights, this structure is the same for the hidden layer and the output layer. Here the structure is presented with the ROM for the output layer.

A

```
---------------------------------------------------------------
        PACKAGE: DEFINES THE PARAMETERS
---------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
package sizes is
   -- Entry signal
   constant INPUT_LENGTH: integer := 52;    -- Defines the number of elements in the input
   -- Hidden Layer
   constant N_HIDDEN_NEURONS: integer :=10;  -- Number of neurons in the hidden layer
   -- Output Layer
   constant N_OUTPUT_NEURONS: integer := 5;  -- Number of neurons in the output layer
   -- DSP
   constant SIZEIN: integer := 16;          -- Length of X, W and Bias
   constant SIZE_NEURON_OUT: integer := 32; -- Out of the DSP, entry of the sigmoid function
   constant SIZE_INT_IN: integer := 4;      -- Length of the integer part in the DSP output.
   -- SIGMOID
   constant int_sel_len: integer := 10;     -- Defines the length of the selection vector.
   constant int_out_len: integer := 16;     -- Defines the length of the output.
   constant N_BIT_OP: integer := 32;        -- Length of the input vector.
   constant N_BIT_FRAC: integer := 26;      -- Length of the fractional part of the input vector.
   constant N_BIT_RANGE_SIG: integer := 5;  -- Fixes the sampling range in the X axis.
end package;
```

B

```
--------------------------------------------------------------------------------
                        Module Name: Neuron
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use work.sizes.all;

entity Neuron is
    Port (
        signal clk        : in  std_logic;  -- clock
        signal ce         : in  std_logic;  -- clock enable
        signal sload      : in  std_logic; -- synchronous load
        signal X          : in  signed (SIZEIN-1 downto 0);  -- 1st input to MACC
        signal W          : in  signed (SIZEIN-1 downto 0);  -- 2nd input to MACC
        signal neuron_out : out signed (SIZE_NEURON_OUT-1 downto 0) -- MACC output
        );
end Neuron;

architecture Behavioral of Neuron is

-- Declare registers for intermediate values
signal X_reg, W_reg           : signed (SIZEIN-1 downto 0) := (others => '0');
signal sload_reg              : std_logic := '1';
signal mult_reg               : signed (2*SIZEIN-1 downto 0) :=(others => '0');
signal adder_out, old_result  : signed (SIZE_NEURON_OUT-1 downto 0):= (others => '0');


begin

process (adder_out, sload_reg)
 begin
  if sload_reg = '1' then
      old_result <= (others => '0');
  else
      -- 'sload' is now active (=low) and opens the accumulation loop.
      -- The accumulator takes the next multiplier output in the same cycle.
      old_result <= adder_out;
  end if;
end process;

process (clk)
 begin
  if rising_edge(clk) then
    if ce = '1' then
        X_reg     <= shift_right(X, SIZE_INT_IN/2); -- leave space for intger bits
        W_reg     <= shift_right(W, SIZE_INT_IN/2); -- leave space for integer bits
        mult_reg  <= X_reg * W_reg;
        sload_reg <= sload;
        -- Store accumulation result into a register
        adder_out <= old_result + mult_reg;
    end if;
  end if;
end process;

 -- Output accumulation result
neuron_out <= adder_out;

end Behavioral;
```

C

```vhdl
----------------------------------------------------------------------------------------
                              Module Name: Sigmoid
----------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
--use IEEE.STD_LOGIC_SIGNED.ALL;
use work.sizes.all;

entity Sigmoid is
    Port (sel : in  STD_LOGIC_VECTOR (N_BIT_OP-1 downto 0);   -- Output of the hidden neuron.
          range_aux: in STD_LOGIC_VECTOR (2 downto 0);        -- Sampling range of the sigmoid.
          clk: in STD_LOGIC;
          out_sig : out  STD_LOGIC_VECTOR (int_out_len-1 downto 0));
end Sigmoid;

architecture Behavioral of Sigmoid is

signal aux: STD_LOGIC_VECTOR(int_sel_len-1 downto 0);
signal sign : STD_LOGIC;
signal neg: STD_LOGIC_VECTOR(N_BIT_OP-1 downto 0);
signal one : STD_LOGIC_VECTOR (int_out_len-1 downto 0):=(others=>'1');

-- The matrix containing the sampled values is filled,
-- its number of elements will be determined by the length of the selection vector (int_sel_len).
type memory is array (0 to 2**int_sel_len-1) of STD_LOGIC_VECTOR (int_out_len-1 downto 0);
constant val_sig: memory:= ("1000000000000000",
"1000000010000000",
"1000000100000000",
"1000000110000000",
        :
        :
"1111111111101001",
"1111111111101001",
"1111111111101010");

begin

sign <= sel(N_BIT_OP-1);
neg <= std_logic_vector(signed(not(sel)) + "000000000000000000001"); -- Two's complement.

with sign select
            aux <= sel(N_BIT_FRAC-1 + to_integer(unsigned(range_aux))
                      downto N_BIT_FRAC + to_integer(unsigned(range_aux)) - int_sel_len) when '0',
                   neg(N_BIT_FRAC-1 + to_integer(unsigned(range_aux))
                      downto N_BIT_FRAC + to_integer(unsigned(range_aux)) - int_sel_len) when others;

with sign select
    out_sig <= val_sig(to_integer(unsigned(aux))) when '0',
              std_logic_vector("0000000000000001" - unsigned(val_sig(to_integer(unsigned(aux)))))
                                                  when others; -- f(-x) = 1 - f(x)
end Behavioral;
```

D

```
--------------------------------------------------------------------------------------------
                              Module Name: Neuron_Sigmoid
--------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
--use IEEE.STD_LOGIC_SIGNED.ALL;
use work.sizes.all;


entity Neuron_Sigmoid is
    Port (
        clk        : in std_logic;  -- Clock
        ce         : in std_logic;  -- clock enable
        sload      : in  std_logic; -- synchronous load
        X          : in  signed (SIZEIN-1 downto 0);  -- 1st input to MACC
        W          : in  signed (SIZEIN-1 downto 0);  -- 2nd input to MACC
        range_sig: in STD_LOGIC_VECTOR (N_BIT_RANGE_SIG-1 downto 0);
        out_sig : out  STD_LOGIC_VECTOR (int_out_len-1 downto 0)
        );
end Neuron_Sigmoid;

architecture Behavioral of Neuron_Sigmoid is

component Neuron is
    Port (
        signal clk        : in std_logic;  -- Clock
        signal ce         : in std_logic;  -- clock enable
        signal sload      : in  std_logic; -- synchronous load
        signal X          : in  signed (SIZEIN-1 downto 0);  -- 1st input to MACC
        signal W          : in  signed (SIZEIN-1 downto 0);  -- 2nd input to MACC
        signal neuron_out : out signed (SIZE_NEURON_OUT-1 downto 0) -- MACC output
        );
end component;

component Sigmoid is
    Port (sel : in  STD_LOGIC_VECTOR (SIZE_NEURON_OUT-1 downto 0); -- Output of the hidden neuron.
          range_aux: in STD_LOGIC_VECTOR (2 downto 0);   -- Sampling range of the sigmoid.
          clk: in STD_LOGIC;
          out_sig : out  STD_LOGIC_VECTOR (int_out_len-1 downto 0));
end component;

signal neuron_out_sig_in : signed (SIZE_NEURON_OUT-1 downto 0);
signal range_aux: STD_LOGIC_VECTOR (2 downto 0); -- Its range will vary depending on N_BIT_RANGE_SIG
signal out_aux: STD_LOGIC_VECTOR (int_out_len-1 downto 0);

begin

Neuron1: Neuron port map(
    clk => clk,
    ce => ce,
    sload => sload,
    X => X,
    W => W,
    neuron_out => neuron_out_sig_in
);

-- The sampling range will determine the amount of integer bits of the input that will be taken.
with range_sig select
range_aux <= "001" when "00010", --Precision 1*10^-1
             "010" when "00100", --Precision 1*10^-2
             "011" when "01000", --Precision 1*10^-3
             "100" when "10000", --Precision 1*10^-4 --> Maximum precision with 4 integer bits.
             "000" when others;
```

```
Sig1: Sigmoid port map(
    sel => std_logic_vector(neuron_out_sig_in),
    range_aux => range_aux,
    clk => clk,
    out_sig => out_aux
);

process (clk)
begin
if rising_edge(clk) then
    out_sig <= out_aux;
end if;
end process;

end Behavioral;
```

E

```
------------------------------------------------------------------------------------------
                          Module Name: Neural Network
------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
--use IEEE.STD_LOGIC_SIGNED.ALL;
use work.sizes.all;

entity NeuralNetwork_Nneurons is
  Port (
    clk         : in std_logic;   -- Clock
    ce          : in std_logic;   -- Clock Enable
    X           : in  std_logic_vector (SIZEIN-1 downto 0);  -- 1st input to MACC
    class       : out std_logic_vector(2 downto 0) := (others=>'0') --output class for pixel
    );
end NeuralNetwork_Nneurons;

architecture Behavioral of NeuralNetwork_Nneurons is

signal cont : unsigned (15 downto 0) := (others=>'0');
signal Xret : signed (SIZEIN-1 downto 0) := (others=>'0');

-- Hidden ROMs
component ROM_InputW is
    Port (sel : in  STD_LOGIC_VECTOR (15 downto 0);
          out0 : out  STD_LOGIC_VECTOR (15 downto 0));
end component;
type ROM_hidden is array (0 to N_HIDDEN_NEURONS-1) of std_logic_vector (15 downto 0);
signal W_hidden : ROM_hidden;
signal sel_aux_hidden : ROM_hidden;

signal sload : std_logic := '1';
-- Hidden Neurons
component Neuron_Sigmoid is
    Port (
        clk         : in std_logic;   -- Clock
        ce          : in std_logic;   -- clock enable
        sload       : in  std_logic; -- synchronous load
        X           : in  signed (SIZEIN-1 downto 0);   -- 1st input to MACC
        W           : in  signed (SIZEIN-1 downto 0);   -- 2nd input to MACC
        range_sig   : in STD_LOGIC_VECTOR (N_BIT_RANGE_SIG-1 downto 0);
        out_sig     : out  STD_LOGIC_VECTOR (int_out_len-1 downto 0)
        );
end component;
```

```
type Hidden_output is array (0 to N_HIDDEN_NEURONS-1) of std_logic_vector (int_out_len-1 downto 0);
signal Hidden_Neuron : Hidden_output;
signal Out_hidden_ram : Hidden_output := (others=>(others=>'0'));
signal hidden_ram_sel : std_logic_vector (int_out_len-1 downto 0);

-- RAM control
signal ramload : std_logic := '1';
signal ram_sel : unsigned (15 downto 0) := (others=>'0');
signal output_sload : std_logic := '1';
signal ram_out: std_logic_vector (int_out_len-1 downto 0) := (others=>'0');

-- Ouput ROMs
component ROM_OutputW is
Port (sel : in  STD_LOGIC_VECTOR (15 downto 0);
      out0 : out  STD_LOGIC_VECTOR (15 downto 0));
end component;
type ROM_output is array (0 to N_OUTPUT_NEURONS-1) of std_logic_vector (15 downto 0);
signal W_output : ROM_output;
signal ram_sel_aux : ROM_output;

-- Nueron out outputs
component Neuron is
    Port (
        signal clk        : in  std_logic;  -- Clock
        signal ce         : in  std_logic;  -- clock enable
        signal sload      : in  std_logic; -- synchronous load
        signal X          : in  signed (SIZEIN-1 downto 0);  -- 1st input to MACC
        signal W          : in  signed (SIZEIN-1 downto 0);  -- 2nd input to MACC
        signal neuron_out : out signed (SIZE_NEURON_OUT-1 downto 0) -- MACC output
        );
end component;
type Outlayer_output is array (0 to N_OUTPUT_NEURONS-1) of signed(SIZE_NEURON_OUT-1 downto 0);
signal outlayer_neurons_output : Outlayer_output;
signal outlayer_neurons_output_save : Outlayer_output := (others=>(others=>'0'));

--Final save control
signal final_save_load : std_logic := '1';

----Comparison for 5 classes

component comparator is
    Port (
    clk : in std_logic;
    out_neuron1 : in signed (SIZE_NEURON_OUT-1 downto 0);
    out_neuron2 : in signed (SIZE_NEURON_OUT-1 downto 0);
    out_neuron3 : in signed (SIZE_NEURON_OUT-1 downto 0);
    out_neuron4 : in signed (SIZE_NEURON_OUT-1 downto 0);
    out_neuron5 : in signed (SIZE_NEURON_OUT-1 downto 0);
    class       : out std_logic_vector(2 downto 0) := (others=>'0') --output class for pixel
    );
end component;

begin

process (clk)
begin
    if (ce = '1' and clk = '1' and clk'event) then
        cont <= cont + "0000000000000001";
        Xret <= signed(X);
    end if;
end process;

sload <= '1' when cont = "0000000000000000" else
         '0' when cont < INPUT_LENGTH+3 else
         '1';
```

```
hidden_layer:
    for i in 0 to N_HIDDEN_NEURONS-1 generate
    sel_aux_hidden(i) <= std_logic_vector(cont + INPUT_LENGTH*i) when sload='0' else (others=>'0');
    hidden_romX: ROM_InputW port map(sel_aux_hidden(i), W_hidden(i));
    hidden_neuron1: neuron_sigmoid port map(clk, ce, sload, Xret, signed(W_hidden(i)),
                    "01000", Hidden_Neuron(i));
end generate;

ramload <= '0' when cont = (INPUT_LENGTH+4) else '1';
process(ramload)
begin
    if (ramload = '0' and ramload'event) then
    Out_hidden_ram <= Hidden_Neuron;
    end if;
end process;

ram_sel <= (cont-(INPUT_LENGTH+5)) when (cont > (INPUT_LENGTH+4) and
                                        cont <(INPUT_LENGTH+N_HIDDEN_NEURONS+6)) else (others=>'0');
output_sload <= '0' when (ram_sel >0 and ram_sel<(N_HIDDEN_NEURONS+5)) else '1';
hidden_ram_sel <= Out_hidden_ram(to_integer(ram_sel-1)) when ram_sel>0 else (others=>'0');

output_layer:
    for i in 0 to N_OUTPUT_NEURONS-1 generate
    ram_sel_aux(i) <= std_logic_vector(ram_sel+i*N_HIDDEN_NEURONS) when output_sload='0' else (others=>'0');
    output_rom: ROM_OutputW port map (ram_sel_aux(i), W_output(i));
    output_neuron: Neuron port map(clk, ce, output_sload, signed(hidden_ram_sel),
                                signed(W_output(i)), outlayer_neurons_output(i));
end generate;

final_save_load <= '0' when cont = (INPUT_LENGTH+N_HIDDEN_NEURONS+7) else '1';
process (final_save_load)
begin
if (final_save_load='1' and final_save_load'event) then
    outlayer_neurons_output_save <= outlayer_neurons_output;
end if;
end process;

---- COMPARISON FOR 5 OUTPUT CLASSES

comparison: comparator port map (clk, outlayer_neurons_output_save(0),
                                outlayer_neurons_output_save(1), outlayer_neurons_output_save(2),
                                outlayer_neurons_output_save(3), outlayer_neurons_output_save(4), class);


end Behavioral;
```

F

```
--------------------------------------------------------------------------------
                         MODULE: Output ROM
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
use work.sizes.all;

entity ROM_OutputW is
Port (sel : in  STD_LOGIC_VECTOR (15 downto 0);
      out0 : out  STD_LOGIC_VECTOR (15 downto 0));
end ROM_OutputW;

architecture Behavioral of ROM_OutputW is
type memoria is array (0 to N_HIDDEN_NEURONS*N_OUTPUT_NEURONS+1) of STD_LOGIC_VECTOR (15 downto 0);
-- in the hidden layer's ROM N_OUTPUT_NEURONS is changed with INPUT_LENGTH in all the code
constant pesos_InputW: memoria:= (
"0000000000000000", --starts at 0
"1111011111101101",
"1111011010110000",
        .
        .
        .
"1101011011011100",
"1111111110101100",
"0000000000000000"
);

type memoria_ext is array(0 to N_HIDDEN_NEURONS*N_OUTPUT_NEURONS+1) of STD_LOGIC_VECTOR (15 downto 0);
signal pesos_InputW_ext: memoria_ext;
begin
generador: FOR i IN 0 TO N_HIDDEN_NEURONS*N_OUTPUT_NEURONS+1 GENERATE
    pesos_InputW_ext(i) <= pesos_InputW(i) WHEN i <= N_HIDDEN_NEURONS*N_OUTPUT_NEURONS+1
                                                                    ELSE "0000000000000000";
END GENERATE generador;

out0 <= pesos_InputW_ext(to_integer(unsigned(sel)));

end Behavioral;
```