Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Monterrey

School of Engineering and Sciences

Neural Network Circuit Implementation using Operational Amplifiers and Digital Potentiometers

A thesis presented by

Jacobo Posada Hoyos

Submitted to the
School of Engineering and Sciences
in partial fulfillment of the requirements for the degree of

Master of Science

in

Engineering Sciences

Monterrey, Nuevo León, June, 2021

Dedication

To God for be always my guide.

To my parents, Sandra and Diego, who gives me strength and wisdom to reach all my goals. For their support and for always give me their best. You are my main motivation.

To my siblings, Sebastián and Tania, for being there when I needed more support. Thank you for my nephews.

To my family and my friends, for all their confidence, support, and encouragement.

Acknowledgements

I would like to thank Dr. Alfonso Gómez Espinosa for the motivation, for giving me its knowledge, for his support, its experience, and time. For being my advisor and help me to define my thesis theme.

To the Dr. Benjamín Valdés Aguirre for accepting to be my co-advisor and teaching me about artificial neural networks.
To the Dr. Jesús Arturo Escobedo, the Dr. Agustín Domínguez Oviedo, and the Ms. Josué González García for being part of my advisory committee and for their advices.

To Tecnológico de Monterrey for the support on tuition and for giving me the opportunity to study in this excellent institution.

To CONACyT for supporting my maintenance during the past two years.

To the Dr. Esmeralda Uribe Lam for her advice and for the support in all aspects of academic life.

To Lucía Cortés, Olga García, Jorge Valencia, and Valentina Valencia for their advice and support, a great help in this process.

To my friends and colleagues for their friendship, company, and experiences. Specially to Alejandra, Andrés Barreto, Braian, Néstor, and Andrés Rumayor.

To every person who has helped me to have new dreams and to pursue them. Thank you for the encouragement and inspiration that allowed me to keep going and fulfill my dreams.

Neural Network Circuit Implementation using Operational Amplifiers and Digital Potentiometers

by

Jacobo Posada Hoyos

Abstract

Implementations of Artificial Neural Networks (ANN) have been advancing for almost three decades and their importance has been marked by the different methods used in their construction, their applications and comparisons in terms of speed, costs, and performance between implementations made by software and hardware. As analog implementations of ANN have been shown to have good levels of performance, high processing speed, low power consumption, small size, and low cost, they have played an important role in the development of new designs. This work presents a proposal to design a circuit implementation of an ANN by using Operational Amplifiers (Op-amps) and digital potentiometers to create a network that can be trained by using an external training system. This, based on circuit analysis and training algorithm by the back propagation (BP) approach.

The proposed design will be simulated in the circuit simulator Proteus. The circuit is tested using the logical gates benchmark problem to verify its performance with the BP learning algorithm.

The results of this work demonstrate that it is possible to create a neural network using analogous components. Furthermore, it shows good performance when implementing the training algorithm using digital potentiometers. As future work is expected to improve the performance of training to create a controller based on neural networks and thus, perform the control of a dynamic system.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

The new developments and progress toward Machine Learning and, specifically, on ANN have been a relevant topic of interest in computational sciences due to their great potential and wide range of applications. This because, as networks are a method of multidimensional regression analysis they are used to solve complex problems that other regression models cannot solve. For example, they can be used in prediction problems when high computational speed is required, when the system is difficult to express mathematically, and when the input and output data of a system is known.
This dissertation aims to present a novel implementation of an ANN by analog devices and digital potentiometers to take advantage of the capacity to change the synaptic weights of the network and make it capable of being trained by using the BP algorithm.

This chapter outlines the main motivations for the research, the specific problem to be addressed, the methodology proposed, the scope of the thesis, and an outline of the dissertation.

1.1 Motivation

This research is addressed to create a circuit of an ANN that has the ability to change their synaptic weights and thus, be able to implement a learning process. The learning process is needed to solve simple problems such as analog-to-digital converters (ADC) as well as complete NP problems such as the traveling salesman problem, routing, etc. It can also be used in image processing, dynamic system control, and networking.

1.2 Problem Statement and Context

The capability to learn is a significant challenge to be considered in an ANN. Such ability is what differentiates them from other approaches of control and prediction in computer science. Although schemes of analog implementations have been proposed, the use of digital potentiometers for the representation of synaptic weights has not been documented. Instead, devices such as transistors, fixed resistors, or other complex circuits have been used.
An important feature in ANN, is the ability of the network to be trained. This is a challenging process because the need of save and change parameters is a crucial implication. Solve this is not common in the proposed schemes and could be reached by using digital potentiometers changing its values. For this reason, the digital potentiometer arises as an option to update the weights of the ANN, keeping low the complexity of the circuit.
Then, the main questions in this research are the following:

- It is possible to create an ANN using analog devices and digital potentiometers?
- Can the ANN circuit have the capacity of being trained by using BP algorithm?

1.3 Solution Overview

To give an answer to the main questions of the research it is necessary to follow the following methodology:

- Design, implement, and test the circuit for a single neuron (perceptron).
- Design and implement a circuit for a complete network based in the perceptron circuit.
- Implement the BP algorithm that allows the ANN to perform the learning process (this by using the Arduino microcontroller as external training system).
- Test the complete circuit with the integration of the learning algorithm.

1.4 Thesis Structure

- Chapter 1. Introduction. This chapter presents the main scope of the research work, the motivation, the research questions, and how to answer them by using the methodology outlined in the solution overview.
- Chapter 2. Literature Review. In this section is presented the main implementations of ANN made by hardware. They are divided in digital, hybrid, and analog applications, giving special attention to this lasts approaches.

- Chapter 3. Theoretical Background. In this chapter, the main concepts related to ANN and electronic devices are presented. Starting with definitions of Op-Amps, limiting circuits, digital potentiometers, and also, the functioning of BP algorithm and the ANN-based controller.
- Chapter 4. Materials and Methods. In this chapter, are stated the main process and methods used to get the expected results. Starting with the process for the circuit implementation, followed by the BP learning algorithm guidelines, and the steps for the practical experiment. Also, the hardware and software resources are shown.
- Chapter 5. Results. Here are shown the main results of the solutions for the logical gates benchmark problems carried out by the circuit implementation with the learning process. The error and the learning parameters are also presented.
- Chapter 6. Conclusions. Finally, in addition to the main findings and contributions, some future work and possible improvements are presented.

Chapter 2

Literature Review

In the literature review of ANN implementations, although many of its applications involve software approaches [1, 2, 3], there are some applications where it is necessary to implement them at a hardware level. This because requirements imposed by handling large volumes of data to be handled and other performance requirements, such as streaming and object recognition for real-time applications [4]. Furthermore, in hardware implementations there are also several types of approaches that have different trade offs between characteristics like speed, cost, fault tolerance, among others [5].

In terms of speed, hardware implementations can offer great computational power due to features such as parallelism and the distribution of tasks through multiple components. For example, in the work presented by Erkmen et al. [6], where the forward propagation time of a Conic Section Function Neural Network circuitry can reach speeds five time faster than its software counterpart.
Furthermore, hardware implementations reduce computational and energy costs. This because they use fewer components and also have low power consumption, as shown by Maliuk et al. [7], where a low-cost and efficient ANN for on-chip integration is developed due to its compact area, non-volatile and dynamic weight storage.
In addition, it is necessary to have a system that has high fault tolerance since it depends on having low errors and faster and accuracy learning. Therefore, using parallelism and distribution of tasks in hardware implementations of ANN has been achieved to have a good performance in the presence of errors and has helped in the diagnosis of fault-tolerance control in various circuits. For example, Kumar et al. [8] presented an ANN PID controller designed to get a better and robust model, even in applications with time-critical needs.

Hardware implementations must also deal with issues such as computational errors due to components (mostly analogous), lack of precision in the results, and the non-linearity of the activation functions. Therefore, different technologies have been developed to achieve hardware implementations that can be robust, fast, precise, and ensure the proper functioning of neural networks.

Implementations such as digital, analog, and hybrid, have been the most developed by researchers in recent years. It is also important to note that the network architecture, the learning algorithm, and the activation function are features that have an important role in network design since some configurations are used in specific fields of development.

2.1 Digital Implementations

Digital implementations are characterized by storing the synaptic weights in registers, tables, or memories, where they are easily accessible. Also, they are characterized by the generation of linear and non-linear activation functions using Look Up Tables (LUT) to implement the learning algorithms with less difficulty. These characteristics allow for easy integration between other applications and systems [9].
Some networks can be implemented in Field Programmable Gate Array-based (FPGA) which are semiconductor devices that are based on a matrix of Configurable Logic Blocks (CLB) connected by programmable interconnects and can be reprogrammed to meet various applications.
On the other hand, there are also Application Specific-Integrated Circuits (ASICs) which are custom made for specific design tasks. Characteristics like lower consumption and speed are improved in many cases [4, 10]. Besides, in other cases, accuracy plays an important role and its value is high compared to the performance achieved by an ANN implemented by software [9, 11, 12].

2.2 Hybrid Implementations

Hybrid implementations which, as its name indicates, are combinations of analog and digital implementations to combine the best advantages of each system in a network. For example, some designs are based on the use of digital memories to save network weights and digital to analog converters for the conversion of analog network inputs [6]. Also, other designs can improve training time [13] even in the order of microseconds [14], and also achieve high levels of cost-effectiveness [15], low area, and low consumption, besides of non-volatile and dynamic storage [7].

2.3 Analog Implementations

The present research is focused on analog implementations which are constituted by electronic devices such as resistors, memristors, capacitors, op-amps and Field Effect Transistors (FET). The weights are saved by the passive elements and the active elements are used to perform activation functions and perform other operations. The use of these devices is due to the fact that analog implementations are recognized for its high-speed processing [16], low power consumption, and compact modules.

For example, using CMOS technology and Charge-Trap-Transistors (CTT) Du et al. [17] created an analog design of a Fully Connected Neural Network (FCNN) where CTTs are used as analog multipliers obtaining power and area reduction compared to digital multipliers but without any proof of training methods.

The use of new analog devices known as Memristor, whose resistance could change depending on the voltage/current applied, proposed by Chua [18] in 1971, and then developed by HP Labs [19], have been one of the most used analog devices to implement neural networks due to its operation similar to the process performed by a neuron.
Designs developed by [20, 21, 22, 23, 24] have proved the potential of this technology. For example, Choi et al. [20] presented a synapse array using memristors to control the weights and connections between layers, then used amplifiers and resistors to build a whole neuron. To demonstrate functionality, they applied their model to a pattern recognition system obtaining an accuracy of 91.3%. Adhikari et al. [21] created a neural network memristor bridge to perform the synapses in a neuron. This configuration allowed the neuron to have negative and positive values for weights as well as save them. The design has a sigmoid activation function and his learning is based on Chip-in-the-Loop, which means that the learning is performed by a host computer. This implementation also improved image processing and pattern recognition performance due to the small size of memristors. The use of this element also shown to improve the performances of image processing and pattern recognition [22, 23] in Cellular Neural Networks (CNN). Furthermore, Yakopcic et al. [24] created a crossbar array of memristors to simulate the operation of a CNN. This design achieved an accuracy of 91.8% compared to 92% of its software implementation, however it is a great achievement since it is the first design to attempt this configuration.

Alternatively to the use of memristors, there are different designs with analogous elements commonly used. For example, Chaudhuri [25] presented a model implemented by op-amps and resistors to create a neuron that can achieve a sigmoid activation function, but without learning or weight adjustments.
Rahman and Ansari [26] design a circuit to create an ANN using resistors, capacitors, and op-amps to solve linear equations that can be improved to solve quadratic equations. However, there was not physical implementation nor learning.
Kawaguchi et al [27] made a design of a Pulsed Neural Network (PNN) using sample hold circuits to change the weights since designs with fixed resistances could not do it. The circuit also has a short time for the learning process and it was scaleable, which is important for image processing applications. Also, the neural network presents training using a dynamic on-site learning system.
In addition, an amplifier-based artificial analog neuron was presented by Weber et al. [28] with an output voltage range and non-linearity added by diode-connected MOS transistors and also having positive and negative weights. The authors used the MOS transistor in its triode region (variable resistor) to adjust the weights of the network and trained it by an algorithm based on a variation of Simulated Annealing.
Sarwar et al. [29] presented a design to create linear and non-linear electrical models

of Hopfield Neural Networks (HPNN) using op-amps, resistances, diodes, and capacitors. They also compared similar models previously made where they demonstrated the advantages in terms of convergence time, lower level of circuit complexity, fewer components, and robustness. However, there was not a learning method.

Although the previous work shows the closest implementation to the proposed work, in this one a digital potentiometer will be used instead of fixed resistors to change the synaptic weights and, in this way, use a learning method such as backpropagation to have the possibility of network learning. As there is no evidence in the literature of this type of implementation, there is an opportunity for research work in this area. A summary between implementations is presented next in Table 2.1 where important characteristics such as complexity level, learning method, and components used can be seen.

Table 2.1: Summary and comparison between previous models and presented model

Article	ANN Type	Components	Synapse	Learning	Characteristics	Complexity
Du. et al [17]	FCNN	-Array of CTT -Analog-digital interfaces -Buffers -Sequential Analog Fabric (SAF)	Charge-Trap-Transistors	No	-Fully-CMOS nonvolatile device -Power reduction -Area reduction	Medium
Choi. et al [20]	FCNN	-Array of WOx-based resistive devices -Operational Amplifier -Resistors	Array of WOx-based Resistive Devices	BP	-RD for synapse applications -High accuracy	High
Adhikari et al [21]	MLNN	-Memristors -Transistors	Memristor Bridge	BP	-Synapse circuit using memristor bridge -Power reduction -Small Area	High
Yakopcic et al [24]	CNN	-Memristors -Operational Amplifiers -Resistors -DACs	Memristor Crossbar	Ex-situ	-Synapse circuit using memristor crossbar -Accuracy	High
Chaudhur et al [25]	FCNN	-Resistors -Operational Amplifiers	Resistors	No	-Sigmoid function implementation	Low
Rahman and Ansari [26]	FCNN	-Resistors -Operational Amplifiers -Capacitors	Resistors	No	-Solve linear equations -Hardware reduction -Time reduction	Low
Kawaguchi et al [27]	PNN	-Resistors -Operational Amplifiers -Capacitors	Sample hold circuits	In-situ	-Short learning time -Variable weights -Small elements -Circuit learning	Medium
Sarwar et al [29]	HPNN	-Resistors -Operational Amplifiers -Diodes -Capacitor	Resistors	No	-Low complexity -Linear/nonlinear characteristics -Stable	Low
Weber et al [28]	FCNN	-MOS transistor -Operational Amplifiers -Resistors	MOS transistor	Variation of simulated annealing	-Optimization of synaptic weights -Positive and negative weights -Output voltage range	Medium
This work	FCNN	- Digital Potentiometers - Resistors - Op-Amps - Diodes	Digital Potentiometers	BP	-Variable Weights -Low complexity -Nonlinear operation -Positive and negative weights	Low

Chapter 3

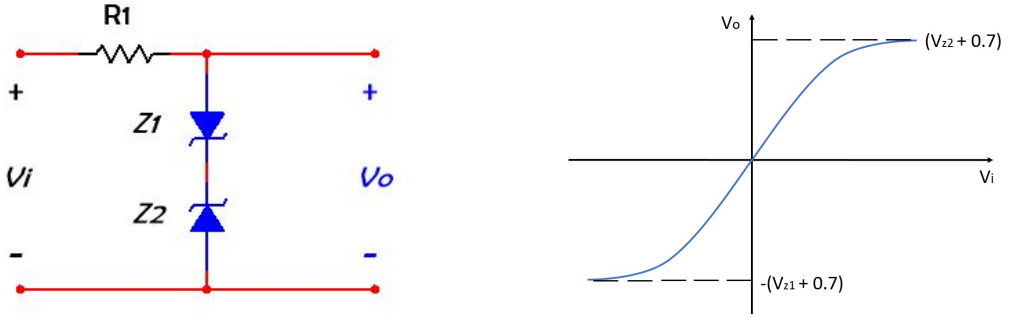
Theoretical Background

3.1 Operational Amplifier

Operational amplifiers (Op-Amps) are linear analog devices with all properties of a DC amplifier, being a close approximation to a perfect amplifier with characteristics such as infinite gain, infinite input impedance, and zero output impedance.
Op-Amps are used in signal processing, signal conditioning, filtering, or mathematical operations due to their different models with external feedback using basic components such as resistors and capacitors between the input and output terminals [26, 30]. In this way, many operations (inversion, sum, subtraction, integration, and differentiation) and configurations (inverting, non-inverting, voltage follower) can be made [13, 24, 27]. The most important feature here, is its ability to implement linear and non-linear functions.

3.1.1 Limiting Circuits

In many cases, diodes and op-amps are combined to create non-linear configurations to take advantage of the fact that diodes can operate closer to their ideal characteristics. A circuit representation can be seen in Figure 3.1a.



(a) (b)

Figure 3.1: (a) Limiting Circuit using two Zener diodes, (b) Relationship between Voltage and Current

The objective is to design a system that achieves a non-linear response output for a linear input signal [29], as seen in Figure 3.1b. In this case, the diodes constrain the input signal to be above or below a specific value and the output is constrained to be below or above the specific value (other values remain constant). These circuits are also called *Logarithmic Amplifier* because they use the logarithmic relationship between current and voltage in the diodes ($V_D \propto I_D$).

3.2 Artificial Neural Networks

Artificial Neural Networks (ANN) are a set of algorithms that are developed to emulate the human brain and are designed for tasks such as classification, clustering, prediction, among others [17]. Thus, the network inputs transform the data into linear separable spaces and each layer can encode different characteristics of the data to process them through the use of activation functions.

Algorithms such as Back-propagation or Real-Time Learning Algorithm (RTRL) [21] can be used for network learning. Some of the advantages of neural networks are their versatility, and the variety of models (convolutional, recurrent and their derivatives) with different structures and applications.

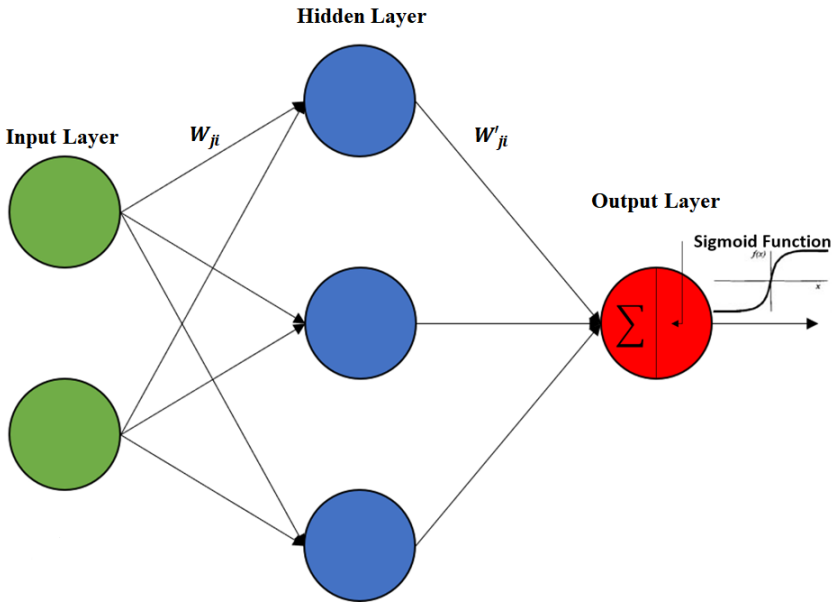


Figure 3.2: Artificial Neural Network Architecture

The ANN is built from 3 types of layers: the input layer, with the initial data, the hidden layers, which are the intermediate layers between the input layer and the output layer where all the calculation is done, and the output layer, which has the result for given inputs (Figure 3.2). As each of the nodes in the network has a synaptic weight at the junction with the next node, the node value and the synaptic weight generate a signal

that can be seen as the impact that connection has on the node in the next layer. The nodes can be either connected between layers or not, depending on the network architecture.

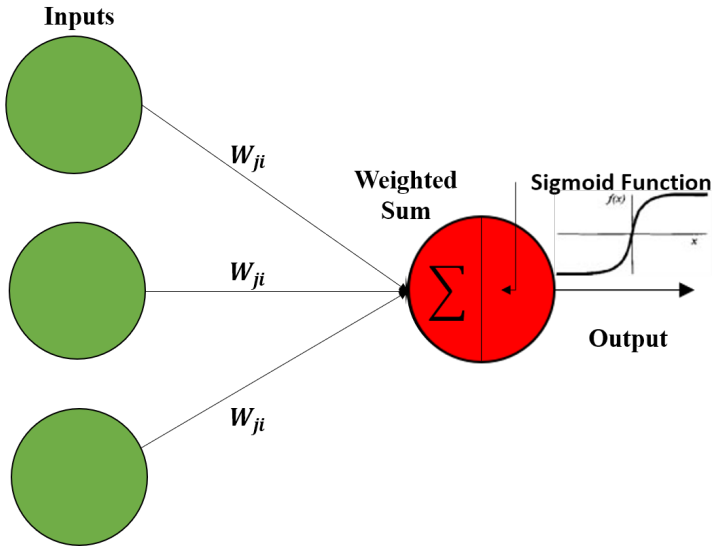


Figure 3.3: Node from an Artificial Neural Network - Perceptron

Therefore, if we take a look at a node it will look like Figure 3.3. This simple node is called *perceptron* which was the first basic unit of inference used for classification applications [29, 31].

3.2.1 Backpropagation Algorithm

Backpropagation is a method used in supervised learning algorithms to train ANN using a method called *gradient descent*. It works by partially deriving the cost (or error) function from any weight (or bias) in the network to know how quickly the cost changes when weights are changed and also gives details of how these changes modify the behavior of the network [32].

The weights change while the cost calculation is repeated until the desired value is reached. Then, the solution is the set of values of all the weights capable of achieving the minimum desired error. At this point, the network is trained and can deal with arbitrary input values responding with an active output if the input contains patterns similar to those that the network has already recognized in their training.

3.2.2 Gradient Descent

To change the weights of the inputs of an ANN, it is necessary to calculate the error function based on Gradient Descent using Equation 3.1.

$$\theta_i := \theta_i - \frac{\alpha}{m} \sum_{i}^{m} [(h_\theta(x_i) - y_i)x_i] \tag{3.1}$$

where

- θ_i are the weights of the hypothesis,
- $h_\theta(x_i)$ are the predicted values for the i^{th} input, and
- α is the learning rate (0,1).

Then, using the sigmoid function as activation function the Equation 3.2 is needed. Where c decides the steepness of the curve.

$$h_\theta(z) = \frac{1}{1 + e^{-cz}}, \tag{3.2}$$

where $z = \theta_0 x_0 + \theta_1 x_1 + ... + \theta_m x_m$ and c modifies the slope of the function.

3.2.3 Calculation of the Gradient

The gradient calculation in a neural network with one hidden layer and a single output can be done by following the process showed in [32] using chain rule and a control scheme as seen in Figure 3.4.

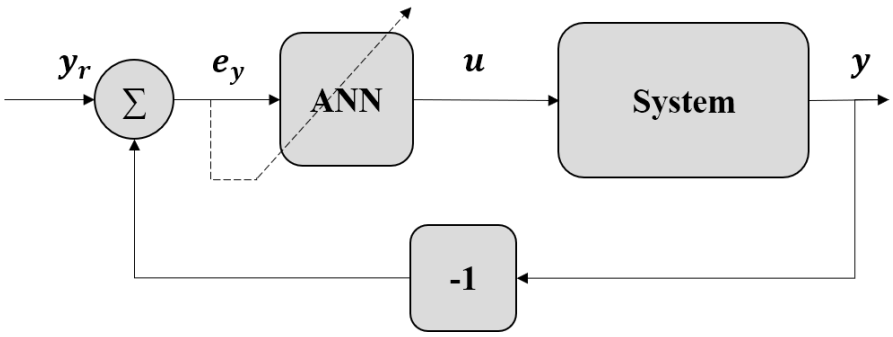


Figure 3.4: Block scheme of a control system [32]

Then, as the weights w_{ji} and v_j are the ones to be adjusted, the gradient is defined:

$$\nabla E = \begin{bmatrix} \frac{\partial E}{\partial v_j} \\ \frac{\partial E}{\partial w_{ji}} \end{bmatrix}, \tag{3.3}$$

with its partial derivatives,

$$\frac{\partial E}{\partial v_j} = \frac{\partial E}{\partial e_y}\frac{\partial e_y}{\partial e_u}\frac{\partial e_u}{\partial u}\frac{\partial u}{\partial r}\frac{\partial r}{\partial v_j} = -e_j u(1-u)\frac{\partial e_y}{\partial e_u}h_j, \tag{3.4}$$

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial e_y}\frac{\partial e_y}{\partial e_u}\frac{\partial e_u}{\partial u}\frac{\partial u}{\partial r}\frac{\partial r}{\partial h_j}\frac{\partial h_j}{\partial S_j}\frac{\partial S_j}{\partial w_{ji}} = -e_j u(1-u)v_j h_j(1-h_j)\frac{\partial e_y}{\partial e_u}x_i, \tag{3.5}$$

where e_u denotes the error between the current control and the control signal to operate the system and where r and S_j are,

$$r = \sum_j v_j h_j, \tag{3.6}$$

$$S_j = \sum_i w_{ji} x_i. \tag{3.7}$$

An important remark is that the error e_y can not be expressed in analytical terms. So, the partial derivatives $\partial e_y/\partial e_u$ in Equation 3.4 and 3.5 are unknown.
Now, the weights can be adjusted each time as follows:

$$v_j^{(t+1)} = v_j^{(t)} - \eta\frac{\partial E}{\partial v_j} = v_j^{(t)} + \eta h_j \delta^1 \frac{\partial e_y}{\partial e_u} \tag{3.8}$$

$$w_{ji}^{(t+1)} = w_{ji}^{(t)} - \eta\frac{\partial E}{\partial w_{ji}} = w_{ji}^{(t)} + \eta x_i \delta_j^2 \frac{\partial e_y}{\partial e_u}, \tag{3.9}$$

where η is the learning rate and $\delta_1 = e_y u(1-u), \delta_j^2 = \delta^1 v_j h_j(1-h_j)$. Considering the partial derivative

$$\frac{\partial e_y}{\partial e_u} = sgn\left(\frac{\partial e_y}{\partial e_u}\right)\cdot\left|\frac{\partial e_y}{\partial e_u}\right|, \tag{3.10}$$

and letting $\eta \cdot |\partial e_y/\partial e_u| \xrightarrow{\eta}$, Equations 3.8 and 3.9 simplify to,

$$v_j^{(t+1)} = v_j^{(t)} + sgn\left(\frac{\partial e_y}{\partial e_u}\right)\eta h_j \delta^1, \tag{3.11}$$

$$w_{ji}^{(t+1)} = w_{ji}^{(t)} + sgn\left(\frac{\partial e_y}{\partial e_u}\right)\eta x_i \delta_j^2, \tag{3.12}$$

where $sgn(\partial e_y/\partial e_u)$ can be found experimentally for the system.

3.3 Digital Potentiometer

A digital potentiometer is an electronic component that has the same functions as a mechanical potentiometer, instead of being mechanically activated, it uses digital signals and switches to change its value. It is commonly used for scaling and trimming analog signals in microcontrollers, instrumentation amplifiers, small-signal audio balance, and offset adjustment [33].

There are two types of digital potentiometers, the ones that have volatile memory, losing their position when disconnected, and the ones that have non-volatile memory (EEPROM) allowing the last value to be saved after disconnected. The number of positions and the resolution of each digital potentiometer has a range between 5 to 10 bits (32 to 1024 steps). Some advantages of digital potentiometers are that they can be controlled in a closed-loop and have different ways to communicate, such as I^2C or Serial Peripheral Interface (SPI) Bus for signaling or using some simpler up/down protocols.
In this way, there are two options to program the digital potentiometer, one as *Potentiometer Divider* (Voltage output operation), where the potentiometer is used to provide a variable voltage by adjusting the wiper position between the two endpoints, and as *Variable Resistor* (Rheostat operation), where the potentiometer is used as a two-terminal resistive element. The equations for adjusting the values change according to the manufacturer.

Chapter 4

Materials and Methods

To fulfill the implementation of the analog circuit of an ANN. First, it is necessary to describe the process to create the analog ANN. Then, know how to integrate the learning algorithm. Finally, verify its operation with a real experiment.
In the Section 4.1, a strategy proposed by Sarwar et al. [29] it is used to carry out the circuit implementation of both a single neuron and a complete network. Also, it is explained the implementation of the learning algorithm [32]. The materials used in the implementation of the circuit and the software resources are in Section 4.2.

4.1 Circuit Model of the Artificial Neural Network

The main goal of this section is to design the circuit used in the implementation of the neural network. The steps to carry out the implementation of the circuit are divided into three subsections. The first subsection is about the implementation of the circuits for a single neuron and for the complete neural network. The second subsection is about the implementation of the learning algorithm. The third subsection is about testing the ANN in a practical application with the logic gates benchmark problem.
The output of the perceptron is calculated by applying the sigmoid function to a weighted sum of the inputs and the synaptic weights. Then, an amplifier can perform this operation through a limiting circuit using Zener diodes.

4.1.1 Model of a single neuron - Perceptron

A basic model in Figure 4.1 is proposed [29]. The output of the circuit is designed to be a continuous and increasing logarithmic sigmoid function. Achieved by using the two Zener diodes (D1 - D2) in the feedback path of the first Op-Amp. Thus, the circuit is designed in three steps:

- Adder, to compute the weighted sum.
- Limiting circuit, to compute the sigmoid function activation.
- Adder, to normalize the output value.

For the first step the Equation 4.1 represent the adder configuration,

$$u_i = R_f \sum_{j=1, j\neq i}^{N} \frac{v_j}{R_{ji}}, \tag{4.1}$$

where u_i is the output of the adder, R_f is the feedback resistance, v_j are the values of the inputs and, the R_{ji} are the weight values.

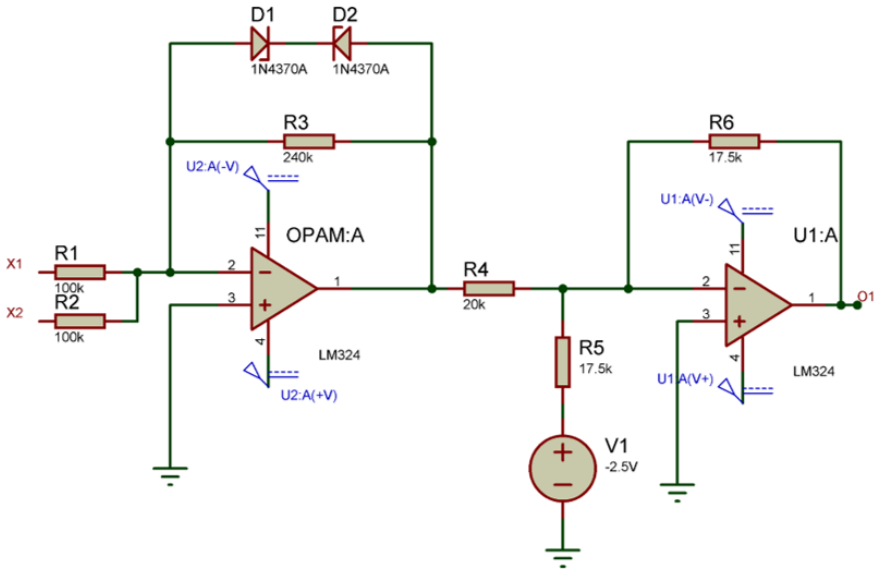


Figure 4.1: Proposed design of a single neuron

For the second step, an integration between the adder and the limiting circuit has to be done. This requires analyzing the feedback diodes using one polarity at a time. As the diodes are connected in series but in opposite directions, to manage positive and negative input values, if the Zener diode $D1$ is in forward-biased operation, the current ID_1 for one input v_j is:

$$I_{D1} = \frac{|v_j|}{R3} \xrightarrow{I}_s [exp\left(\frac{V_D}{nV_T}\right) - 1] = \frac{|v_j|}{R_{ji}}, \tag{4.2}$$

where I_s is the reverse bias saturation current of the diode, V_D is the voltage drop, V_T is the thermal voltage ($\approx 26mV$ at 25 C), and n is the quality factor of the diode (between 1 and 2) [29]. Simplifying to obtain V_D,

$$V_D = nV_T \ln\left(\frac{|u_i|}{R3I_s} + 1\right). \tag{4.3}$$

The process is the same using the D2 in forward-biased operation for negative values. Then, it is only necessary to add the Zener voltage V_z for each diode. The output for

the circuit lies in a range between $[-(V_z + V_D), Vz + V_D]$. Thus, the output for the first Op-Amp is going to be in the range:

$$u_{v_j} = \begin{cases} V_z + nV_T \ln\left(\frac{|u_i|}{R3I_s} + 1\right) & v_j < 0 \\ 0 & v_i = 0 \\ -\left(V_z + nV_T \ln\left(\frac{|u_i|}{R3I_s} + 1\right)\right) & v_j > 0. \end{cases} \tag{4.4}$$

For the third and last step, an inverse and an adder operation of an Op-Amps is required. Here, the output $u(v_j)$ needs to be added with the normalization value given by a DC source of value V_{dc}. The final equation for the second OpAmp is:

$$O_1 = -u_{v_j}\frac{R6}{R4} + V_{dc}\frac{R6}{R5}, \tag{4.5}$$

However, this equation must meet two conditions:

$$\frac{R6}{R4}V_{D_{max}} + V_z = 2.5, \frac{R6}{R5}V_{dc} = -2.5.$$

Integrating this circuit with the digital potentiometer and the Arduino microcontroller as shown in Figure 4.2 the circuit is complete.

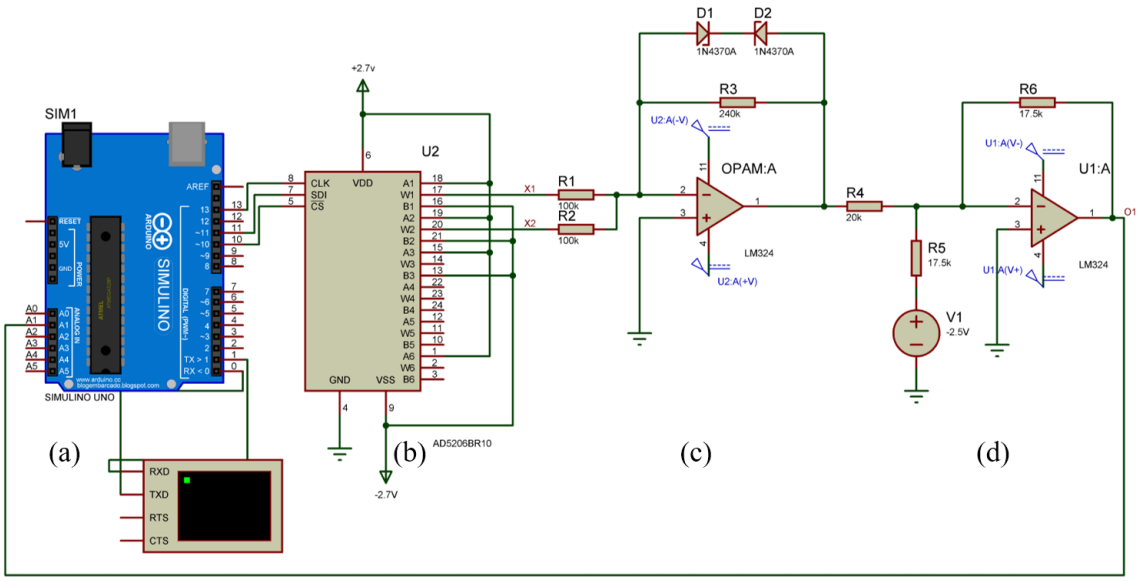


Figure 4.2: Circuit for the Perceptron model (a) Arduino microcontroller, (b) Digital potentiometer AD5206, (c) Summation and output function, (d) Output normalization.

In this circuit, the digital potentiometer is operating as a potentiometer divider and gives the synapse values to the network.

4.1.2 Model of the Artificial Neural Network

To create a complete ANN, it is first necessary to define its architecture. For this implementation the selected configuration is:

- Two inputs
- One hidden layer with three neurons
- One output

Having the basic model and the architecture defined, the circuit is done by integrating the design shown in Figure 4.2. First, the weight matrices for the hidden layer and the output layer are defined with dimensions $2x3$ and $1x3$. As each digital potentiometer (AD5206) has six independent channels, each potentiometer therefore can handle six weights. Then, for 9 weights, 2 potentiometers are used in total. The output of each neuron in the hidden layer goes to the analog inputs of the Arduino microcontroller, where each weight is multiplied by each input (or hidden value). This value goes then to the digital potentiometer via SPI communication. A flowchart diagram of the circuit is shown in the Figure 4.3. The complete circuit for the ANN can be found in A with the values of all the internally used resistances and sources.

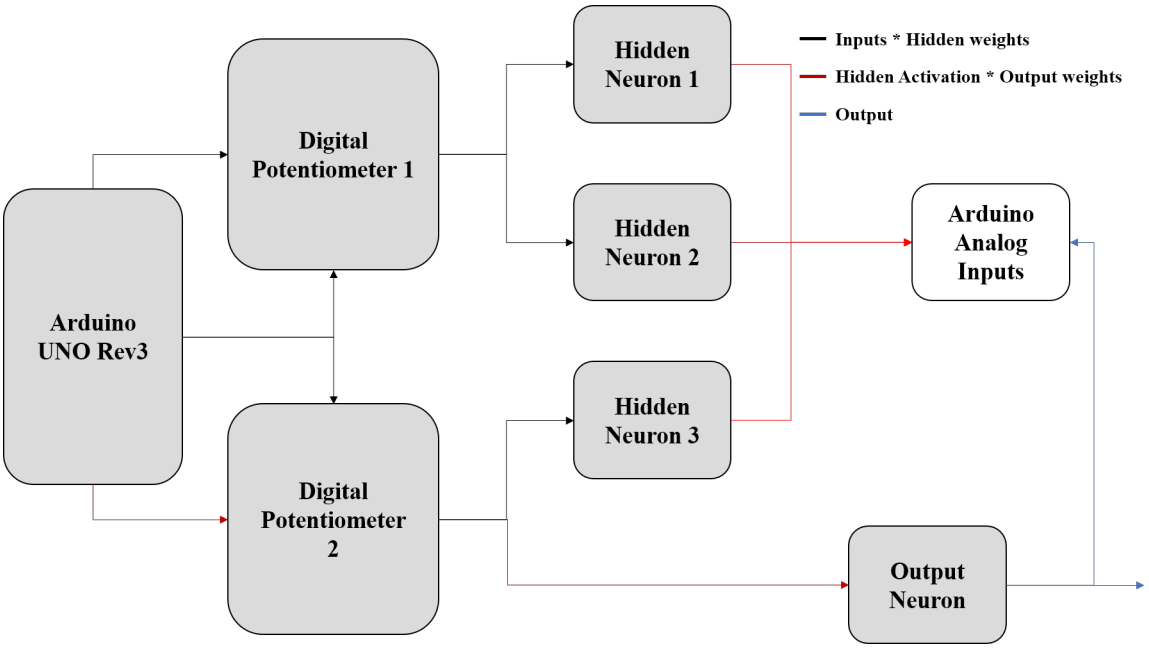


Figure 4.3: Flowchart diagram of the ANN circuit.

4.1.3 Implement the training algorithm to achieve the real time learning (Back-propagation - BP).

To realize the implementation of the training algorithm an Arduino® hardware development board will be used. The code is wrote in the Integrated Development Environment (IDE) of Arduino®. The pseudo code is shown in Appendix B and a flowchart of its functioning can be seen in Figure 4.4. The board version is the Arduino UNO Rev3.
The ANN weights are adjusted by Equations 3.8 and 3.9 for each iteration until an acceptable error is reached. When there is no necessary to adjust the weights the training stops and the network is ready to operate. It is important to remark that the range for the weights is limited to $[-1, 1]$.

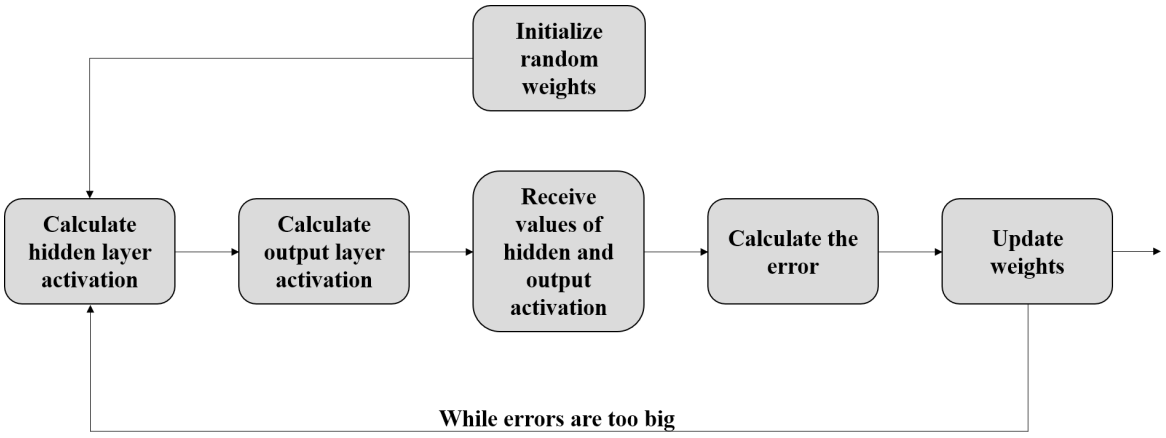


Figure 4.4: Flowchart diagram of the Back Propagation algorithm.

To verify the functioning of the ANN the logic gates benchmark problem was proposed. This part is detailed in Chapter 5.

4.2 Resources

Software and hardware resources used in this work are listed below:

- Software for the training System (Arduino® IDE).
- Software for the simulation of the ANN (Python).
- Software for the simulation of the circuit (Proteus Lab™).
- Arduino Library for Proteus.
- Laptop Acer Nitro AN515-52 Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz RAM @ 8.00GB

- Zener diode (1N4370).
- Digital Potentiometer (AD5206) 8-bit.
- Microcontroller Board Arduino® UNO Rev3.
- Operation Amplifier $LM324$.

Chapter 5

Results

In the previous chapter, the problem and proposed solution of how to create the circuit for the ANN was discussed. To prove the performance of the circuit, each of the methods had to be developed to determine the best configuration of parameters and values for each device.
The circuit was simulated using the circuit simulator Proteus (8.8). An Arduino microcontroller (Rev3) is used for the implementation of the BP algorithm and to control the values sent to digital potentiometers. The outputs of the digital potentiometers are connected to four Op-Amps which are the hidden and output activations of the network. All the outputs of each neuron are sent back to the Arduino to the learning process.
It is important to remark that the supply voltage of the digital potentiometer (AD5206) is $\pm 2.7V$. As the potentiometer has 255 possible values (255 being $+2.7V$, 127 being 0, and 0 being $-2.7V$) the increment of each step is about $\approx \pm 0.021V$. The complete circuit can be seen in Appendix B.

5.1 Preliminary Results

This section shows the first attempts to verify the performance of the circuit by testing different resistor and gain values on the two Op-Amps.

5.1.1 Functioning of the Perceptron Circuit

First, it is necessary to check if it is working well and if it is giving the required output. For this purpose, two sine voltage sources where connected to the inputs of the circuit to test all the possible values the network can work with. The source values of the inputs, resistors, and other elements used were: 2 sinusoidal voltage sources each of $3V$ with $1Hz$ of frequency, two Zener Diodes (1N4370A - $2.4V$), and the resistances $R1 = R2 = R4 = 10k\Omega$. The Figure 5.1 shows the circuit and Figure 5.2 shows the performance of the activation part for the first Op-Amp.

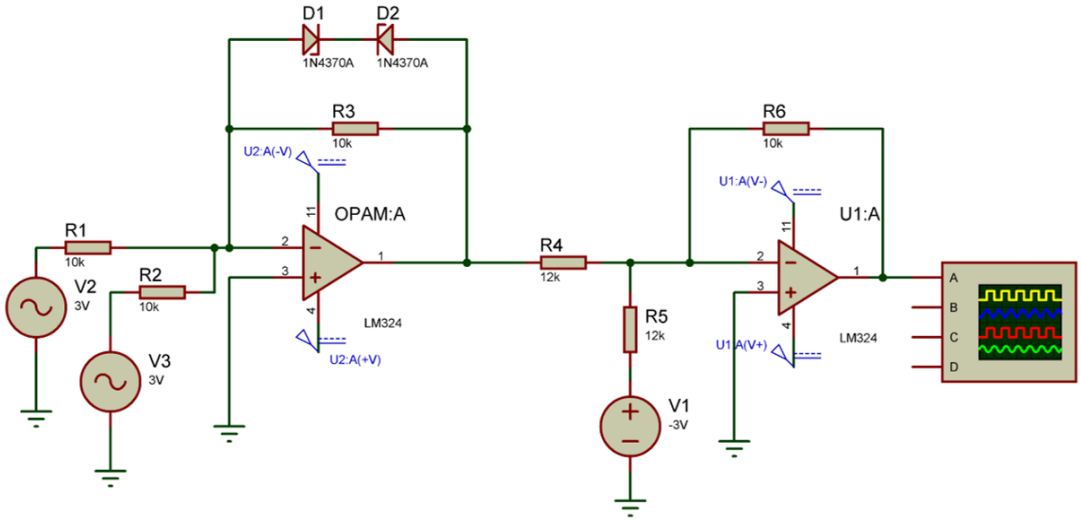


Figure 5.1: Perceptron circuit using Sine waves

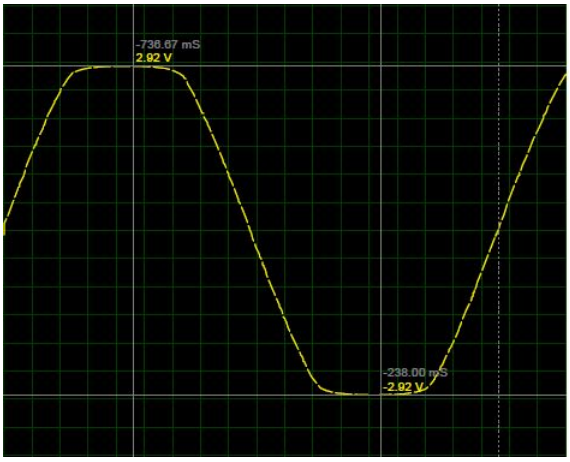


Figure 5.2: Perceptron Activation Output

As the Zener Diodes have a value of $2.4V$ the output of the signal meet the output value given by the Equation 4.4.
The normalization circuit was done by using the following values, $R4/R6 = R5/R6 = 2 * (3V/5V) = 1.2$ and $V_{dc} = 3V$. The Figure 5.3 shows the normalization output for the perceptron.

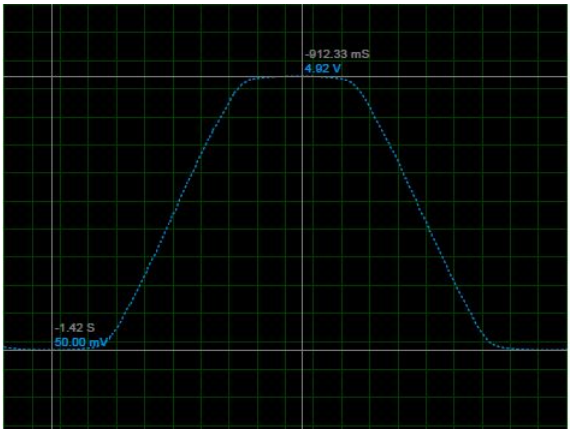


Figure 5.3: Perceptron Normalized Output

As can be seen, the circuit for the perceptron model shows that the respective outputs in both the calculation of the activation function and the normalization of the output are between the corresponding values.
Furthermore, if the network training require a fast convergence, a way to do that is by modifying the slope *c* of the sigmoid function (Equation 3.2). To test this, the slope of the sigmoid is modified by a factor of 1.5. As the resistors $R1$ and $R2$ have a value of $10k\Omega$, the feedback resistor $R4$ must be modified to a value of $15k\Omega$. The sigmoid response and the normalized output for this change are shown in Figure 5.4 and Figure 5.5.

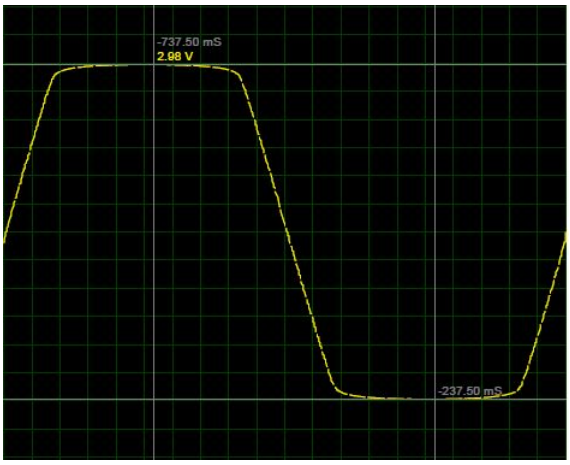


Figure 5.4: Perceptron Activation Output with *c* = 1.5

The normalized output should also be changed to a 1.5 factor. This is done by modifying the values, $R6/R8 = 12k\Omega/20k\Omega = 0.6$ and $R7/R8 = 24k\Omega/20k\Omega = 1.2$.

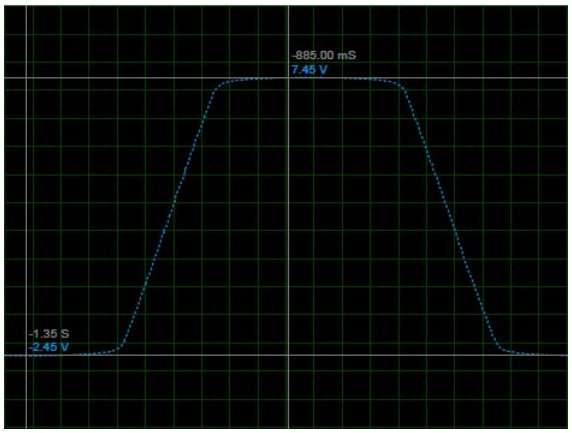


Figure 5.5: Perceptron Normalized Output with *c* = 1.5

The result of the perception shows the modification of the slope and makes the limits to be reached in less time.

5.1.2 Functioning of the Artificial Neural Network Circuit

To test the circuit, the logical gates benchmark problem was used. For each test, the output for each truth table, the value of the synaptic weights, and an error graph is showed. Each error value is shown each 1000 iterations. In addition, to check the performance of the circuit in Feed Forward operation, the last test was carried out using previously calculated weights.

Logical Disjunction (OR) Test

In this experiment, two test were made. A test with the slope equal to 1 for the sigmoid activation function and a test with a slope equal to 1.5. The truth table for the operation between the two logical values can be found in Table 5.1.

Table 5.1: Truth table of Logical Disjunction

X	Y	Output
0	0	0
0	1	1
1	0	1
1	1	1

- **Sigmoid Function with Slope $c = 1$**
  In an attempt to get a response for this problem, the circuit result shows a great performance for this test with a learning rate of 0.05 and a tolerance of 0.1. The

output of the test is shown in Table 5.2 and the weights are shown in Table 5.3. Also, other test was realized but after 55000 iterations the output get stuck in what we think could be a local minimum at 0.06848.

Table 5.2: Output OR gate [V] - Slope c = 1

Input		Target	Output
0	0	0	0.32747
0	1	1	0.76540
1	0	1	0.81427
1	1	1	0.97263
MSE			0.09876

Table 5.3: Weights for the OR gate - Slope c = 1

Inputs	Hidden Weights			Output Weights		
X1	-0.88642	0.82161	-0.58703	-0.99867	0.99999	-0.96681
X2	-0.80931	0.86860	-0.35521			

Trying to solve the problem in local minimum, it was decided to change the value for the learning rate from 0.05 to 0.01 when a value of error less than 0.1 was found. However, the circuit responds slow without reaching a lower error value than the local minimum. The graph of the error can be seen in Figure 5.6.

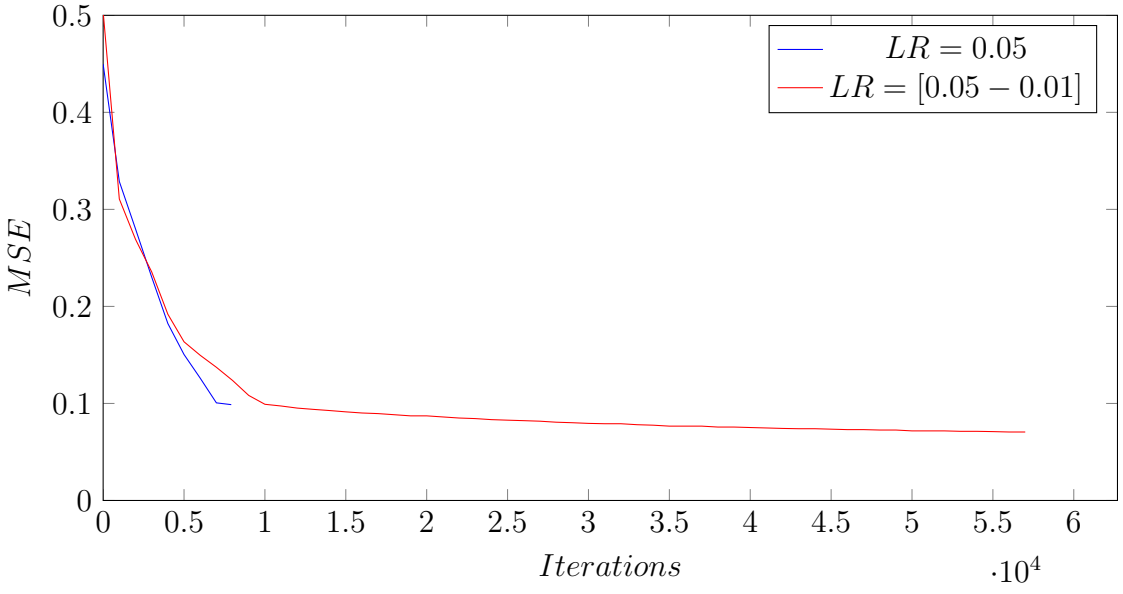


Figure 5.6: Error graph for OR gate (2 examples)

Although the error was low, it was decided to try the same test but this time changing the slope of the activation function. This is presented in the next subsection.

- **Sigmoid Function with Slope *c*** $= 1.5$

  The results shown in the Table 5.4 were obtained by changing the slope of the sigmoid activation function. The final weight values can be found in Table 5.5. The learning rate chosen was $LR = 0.05$ and the tolerance value was 0.01.

Table 5.4: Output OR gate [V] - Slope c = 1.5

Input		Target	Output
0	0	0	0.11046
0	1	1	0.93060
1	0	1	0.95894
1	1	1	1.00000
MSE			0.00935

Table 5.5: Weights for the OR gate - Slope $c = 1.5$

Inputs	Hidden Weights			Output Weights		
X1	-0.50211	-0.51257	0.31342	-0.99985	0.84040	-0.71314
X2	0.40238	-0.27818	-0.05334			

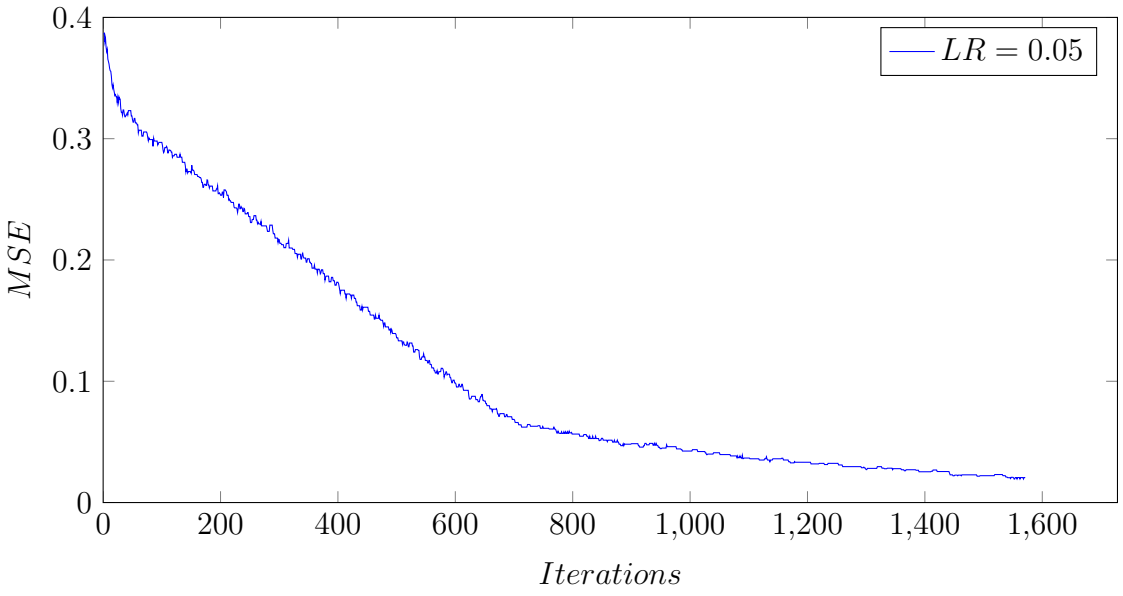


Figure 5.7: Error graph for OR gate

The Figure 5.7 shows how the MSE error decrease through time, meaning that the network is getting a good result in training. It can be seen that the program reaches the error goal in 8132, improving the convergence of the system compared to the attempts in the past experiment with the slope $c = 1$.

Logical Conjunction (AND) Test

To test the logical conjunction, the same two slopes were used. The truth table for the operation between the two logical values can be found in Table 5.6.

Table 5.6: Truth Table of Logical Conjunction

X	Y	Output
0	0	0
0	1	0
1	0	0
1	1	1

- **Sigmoid Function with Slope $c = 1$**

  The results of this experiment can be seen in Table 5.7 and a graph of the error can be seen in Figure 5.8. The final weight values can be found in Table 5.8. The learning rate chosen was $LR = 0.05$ and the tolerance value was 0.01.

Table 5.7: Output AND gate [V] - Slope c = 1

Input		Target	Output
0	0	0	0.12805
0	1	1	0.32063
1	0	1	0.32356
1	1	1	0.50635
MSE			0.23379

Table 5.8: Weights for the AND gate

Inputs	Hidden Weights			Output Weights		
X1	-0.82737	0.46449	-0.11019	-0.99919	0.16818	-0.99923
X2	-0.12759	0.29415	-0.84350			

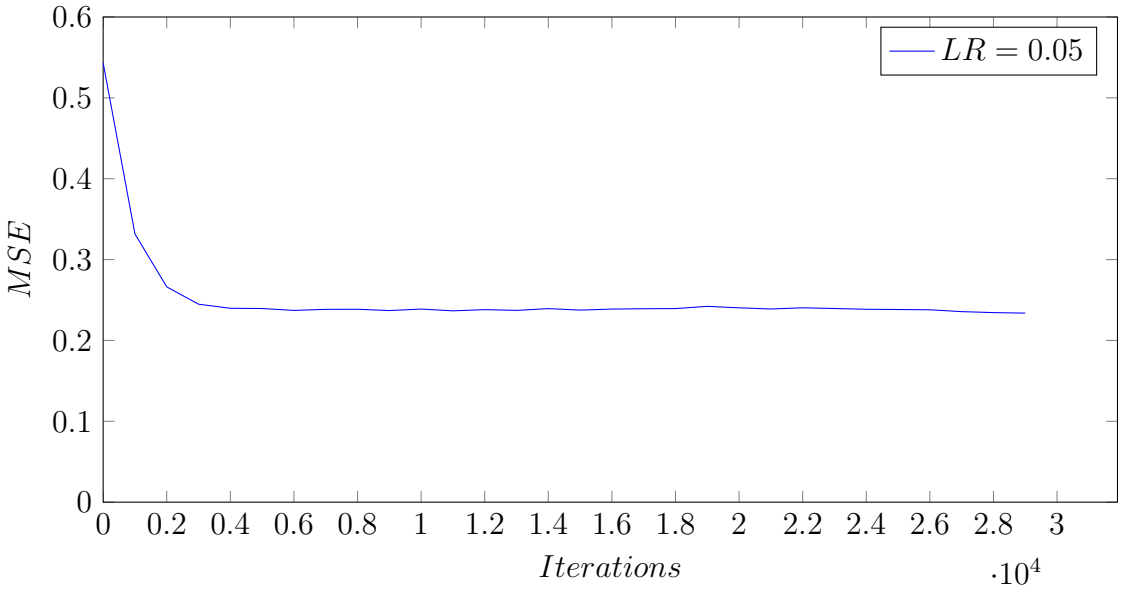


Figure 5.8: Error graph for AND gate

- **Sigmoid Function Slope *c*** $= 1.5$

  The results of this experiment can be seen in Table 5.9 and a graph of the error can be seen in Figure 5.9. The final weight values can be found in Table 5.10. The learning rate chosen was $LR = 0.05$ and the tolerance value was 0.01.

Table 5.9: Output AND gate [V] - Slope c = 1.5

Input		Target	Output
0	0	0	0.00000
0	1	1	0.07038
1	0	1	0.07331
1	1	1	0.90518
MSE			0.00966

Table 5.10: Weights for the AND gate

Inputs	Hidden Weights			Output Weights		
X1	-0.60638	0.25790	0.03741	-0.99995	0.58523	-0.99987
X2	0.29719	0.13650	-0.49865			

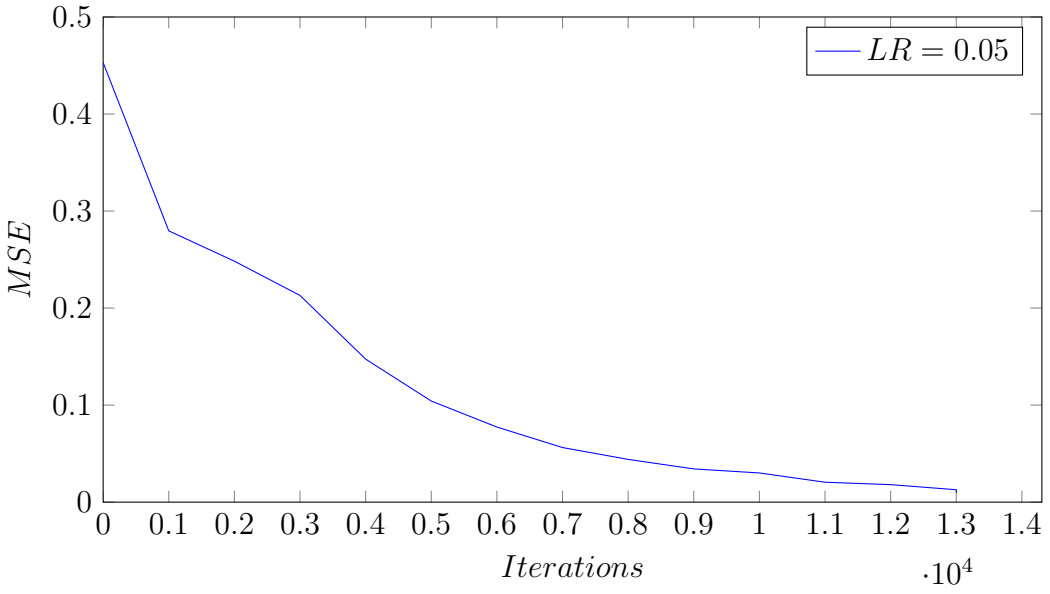


Figure 5.9: Error graph for AND gate

The Figure 5.9 shows how the MSE error decrease through time, meaning that the network is obtaining a good result in training. Reaching a convergence in the 13004 iteration.

Exclusive Disjunction (XOR) Test

The results for the test for the XOR test are shown next. One test was made using the slope $c = 1.5$ with backpropagation. Also, one test was made in feed forward operation to prove its performance without training. The truth table of the XOR gate is shown in Figure 5.11.

Table 5.11: Truth table of Exclusive Disjunction

X	Y	Output
0	0	0
0	1	1
1	0	1
1	1	0

- **Sigmoid Function Slope *c*** $= 1.5$

  The results of this experiment can be seen in Table 5.12. A graph of the error can be see in Figure 5.10. The computed weights values can be found in Table 5.13. The learning rate chosen was $LR = 0.05$ and the tolerance value was 0.01.

Table 5.12: Output XOR gate [V] - Slope c = 1.5

Input		Target	Output
0	0	0	0.27175
0	1	1	0.82405
1	0	1	0.82405
1	1	1	0.47312
MSE			0.17980

Table 5.13: Weights obtained for the XOR gate - Proteus Lab

Inputs	Hidden Weights			Output Weights		
X1	-0.80709	-0.43701	0.44922	-0.32086	-0.99996	0.32087
X2	0.45556	-0.43701	-0.80531			

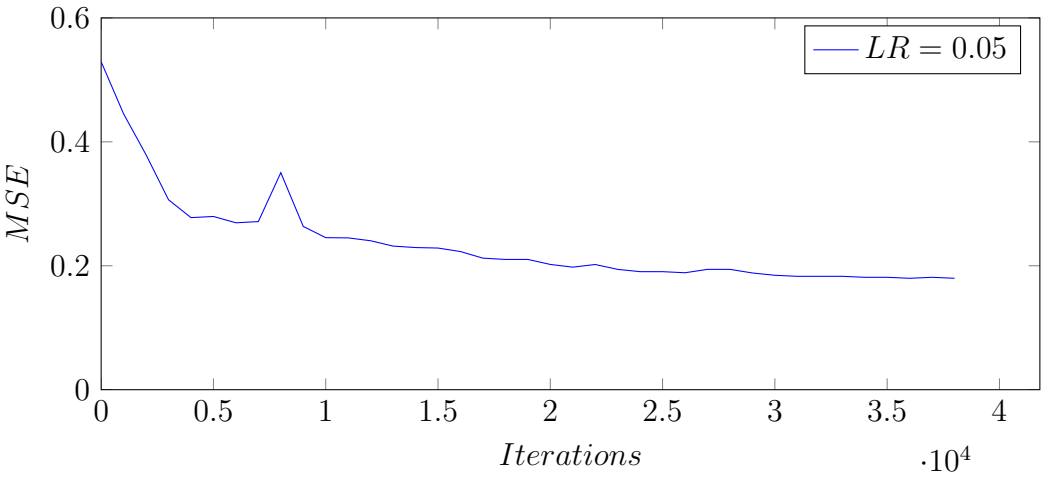


Figure 5.10: Error graph for XOR gate

- **Feed Forward Operation**

  As could be observed in the previous test, the error is still high to give as solved the problem. To prove the performance of the ANN circuit to deal with this problem an algorithm in Python was made to obtain the synaptic weights that can be tested in the model to obtain the desired output. The code can be seen in Appendix D. The weights obtained can be seen in Table 5.15 and the output of these is shown in Table 5.14.

Table 5.14: Output XOR gate [V] - Python

Input		Target	Output
0	0	0	0.22385
0	1	1	0.93255
1	0	1	0.93255
1	1	1	0.05474
MSE			0.03110

Table 5.15: Weights obtained by Python - XOR gate

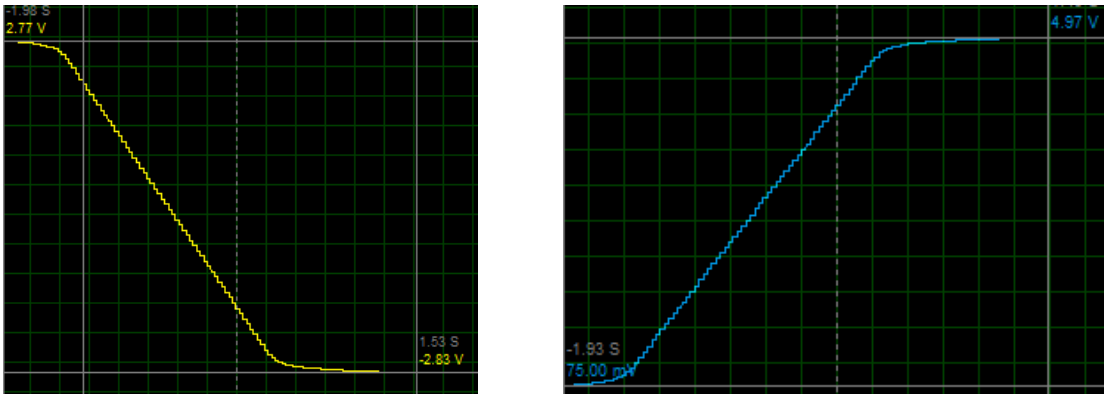
Inputs	Hidden Weights			Output Weights		
X1	-0.647914	0.833005	0.999998	-0.731207	0.999937	-0.731206
X2	0.999998	0.832988	-0.647998			

5.2 Final Result

After obtaining the preliminary results, errors in the gains given by each of the amplifiers were corrected and modified as shown in equation 4.5. Thus, the circuit converged quickly and with low errors even with the XOR gate problem. The results are shown below.

5.2.1 Functioning of the Perceptron Circuit

For each circuit stage, the limiting phase and the output phase of the sigmoid function are shown in Figure 5.11. It can be seen that the behavior of the function has the ”S” shape required for the sigmoid function and also has the desired operating range $\approx (0V - 5V)$. Different from what was shown in the preliminary results, the signal input values are provided by the Arduino microcontroller and not by sinusoidal signals.



(a) (b)

Figure 5.11: (a) Output Limiting Circuit, (b) Sigmoid Function

5.2.2 Functioning of the Artificial Neural Network Circuit

Logical Disjunction (OR) Test

Only one experiment was used to evaluate the performance of the circuit for this problem using BP algorithm.

The value for the learning rate of the neural network was set to $\eta = 0.08$ and the tolerance was set in 0.02. Results for the lower error is presented in Table 5.16. The values of the respective weights are shown in Table 5.17. A graphic showing the MSE error is presented in Figure 5.12.

Table 5.16: Output OR gate [V] - Slope c = 2

Input		Target	Output
0	0	0	0.15249
0	1	1	0.96285
1	0	1	0.88661
1	1	1	0.99022
MSE			0.01879

Table 5.17: Weights obtained by Proteus Lab - OR gate

Inputs	Hidden Weights			Output Weights		
X1	0.32084	0.01384	-0.36001	0.47250	0.17958	-0.99985
X2	0.73457	-0.37768	-0.99998			

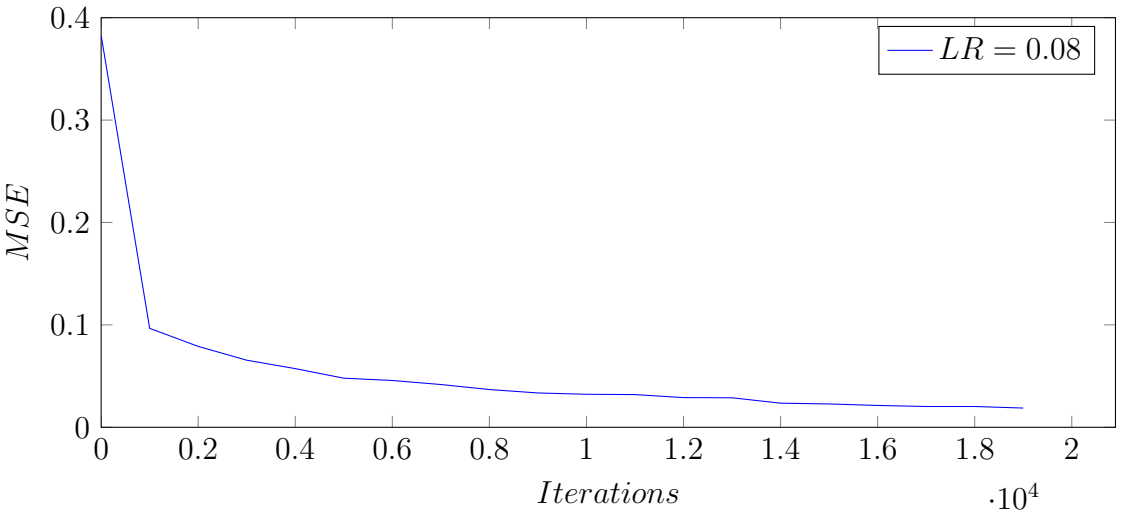


Figure 5.12: Error graph for OR gate

Logical Conjunction (AND) Test

Only one experiment was used to evaluate the performance of the circuit for this problem using BP algorithm.

The results of this experiment can be seen in Table 5.18. A graph of the error can be seen in Figure 5.13. The final weight values can be found in Table 5.19. The learning rate chosen was $LR = 0.09$ and the tolerance value was 0.05.

Table 5.18: Output AND gate [V] - Slope c = 2

Input		Target	Output
0	0	0	0.01173
0	1	1	0.18377
1	0	1	0.12219
1	1	1	0.78788
MSE			0.04765

Table 5.19: Weights for the AND gate

Inputs	Hidden Weights			Output Weights		
X1	0.14292	0.14352	-0.43414	0.50355	-0.97308	-0.99991
X2	0.19001	-0.53933	0.10511			

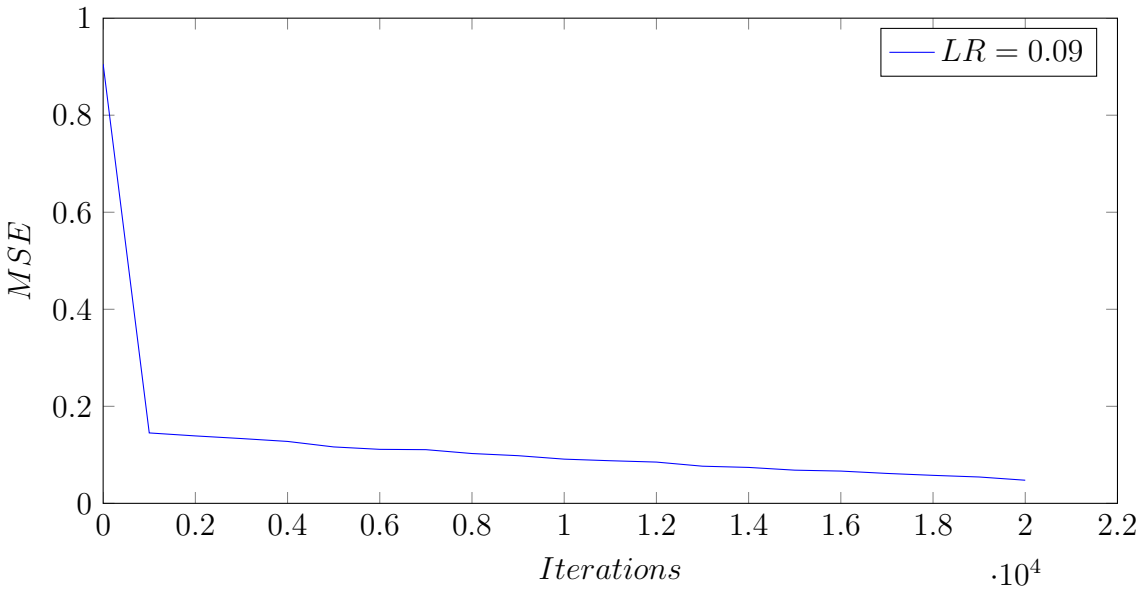


Figure 5.13: Error graph for AND gate

Exclusive Disjunction (XOR) Test

The logical gate XOR benchmark problem was used to evaluate the performance of the circuit. Two experiments were made: one with the BP algorithm made in the Arduino microcontroller and other using python script to calculate the weights.

- **Backpropagation Algorithm**

  For the first experiment, the value for the learning rate of the neural network was set to $\eta = 0.05$. In this case, three experimental tests were made to find the best behavior and convergence of the network. Results for the lower error is presented in Table 5.20 and the values of the respective weights are shown in Table 5.21. Also, a graphic showing the MSE error is presented in Figure 5.14.

Table 5.20: Output XOR gate [V] - Proteus Lab

Input		Target	Output
0	0	0	0.25024
0	1	1	0.80254
1	0	1	0.80254
1	1	0	0.09482
MSE			0.07480

Table 5.21: Weights obtained for the XOR gate - Proteus Lab

Inputs	Hidden Weights			Output Weights		
X1	0.86810	0.88683	-0.49518	-0.71736	0.99945	-0.71747
X2	-0.46674	0.91707	0.90635			

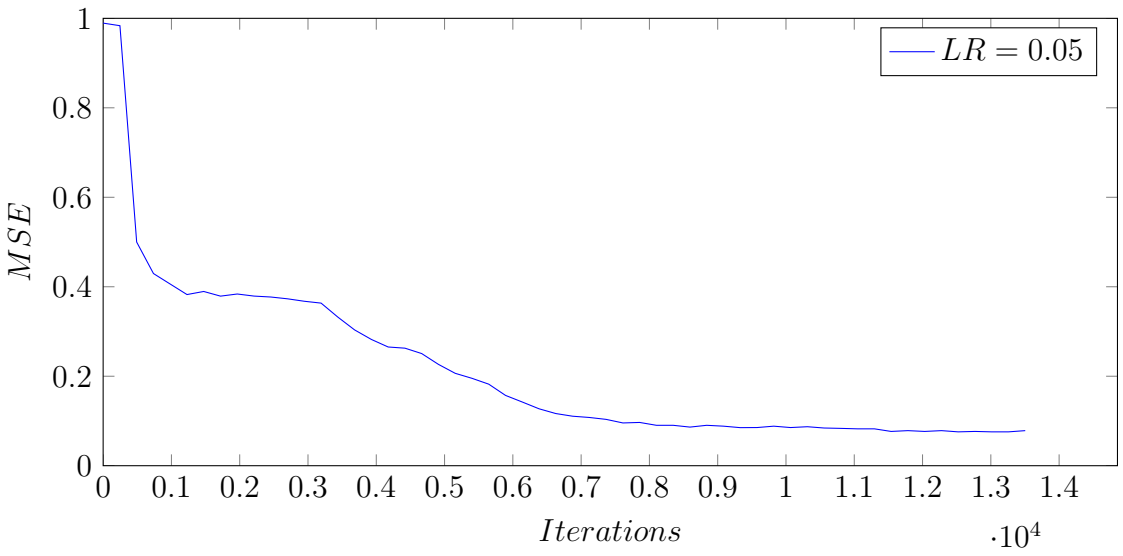


Figure 5.14: Error graph for XOR gate

- **Feed Forward Operation**

  The second experiment validate and proved the performance of the circuit in feed forward operation using the synaptic weights obtained by the Python script. The results of the experiment are shown in Table 5.22 and the obtained weights can be seen in Table 5.23.

Table 5.22: Output XOR gate [V] - Python

Input		Target	Output
0	0	0	0.22385
0	1	1	0.77615
1	0	1	0.77615
1	1	0	0.13978
MSE			0.08493

Table 5.23: Weights obtained by Python - XOR gate

Inputs	Hidden Weights			Output Weights		
X1	-0.64788	0.99999	0.82856	-0.73135	-0.73115	0.99994
X2	0.99999	-0.64787	0.99999			

The output of both experiments have an accuracy greater than 95% and for practical uses the values greater than $0.7V$ can be seen as 1 and values lower than $0.3V$ can be considered as 0.

Chapter 6

Conclusions

A novel method to implement a circuit of an ANN by hardware was proposed. This circuit uses Op-Amps for processing and digital potentiometers to change the synaptic weights in the ANN. The use of this combination results in having the possibility of training the network through the BP algorithm and obtaining a circuit to solve diverse problems with a low complexity design.

In this research, some questions were proposed that must be answered:

- It is possible to create an ANN using analog devices and digital potentiometers?
- Can the ANN circuit have the capacity of being trained by using BP algorithm?

Thus, the results shown in Chapter 5 give the reasons to address the first question, since the analog devices and the digital potentiometer evidenced good performance in Feed Forward operation obtaining accuracy levels greater than 95% and demonstrated the potential of the circuit to implement the ANN.

Regarding to the second question, the integration with the BP algorithm and the Arduino microcontroller was successful. The experimental results shows that the performance in the BP learning method achieved a level of accuracy greater than 95% in 13750 iterations with a circuit easy to implement and using few and low cost components, proving to have advantages in terms of synaptic weight adjustment and learning ability compared to designs already seen in literature.

As future work, a digital potentiometer with high resolution could be used to reach higher values of accuracy. The implementation of other types of architectures, such as Recurrent Neural Networks or LSTM can be performed to evaluate the training ability. Also, changes in the control system could be made to improve the accuracy and training convergence.

Appendix A

Appendix

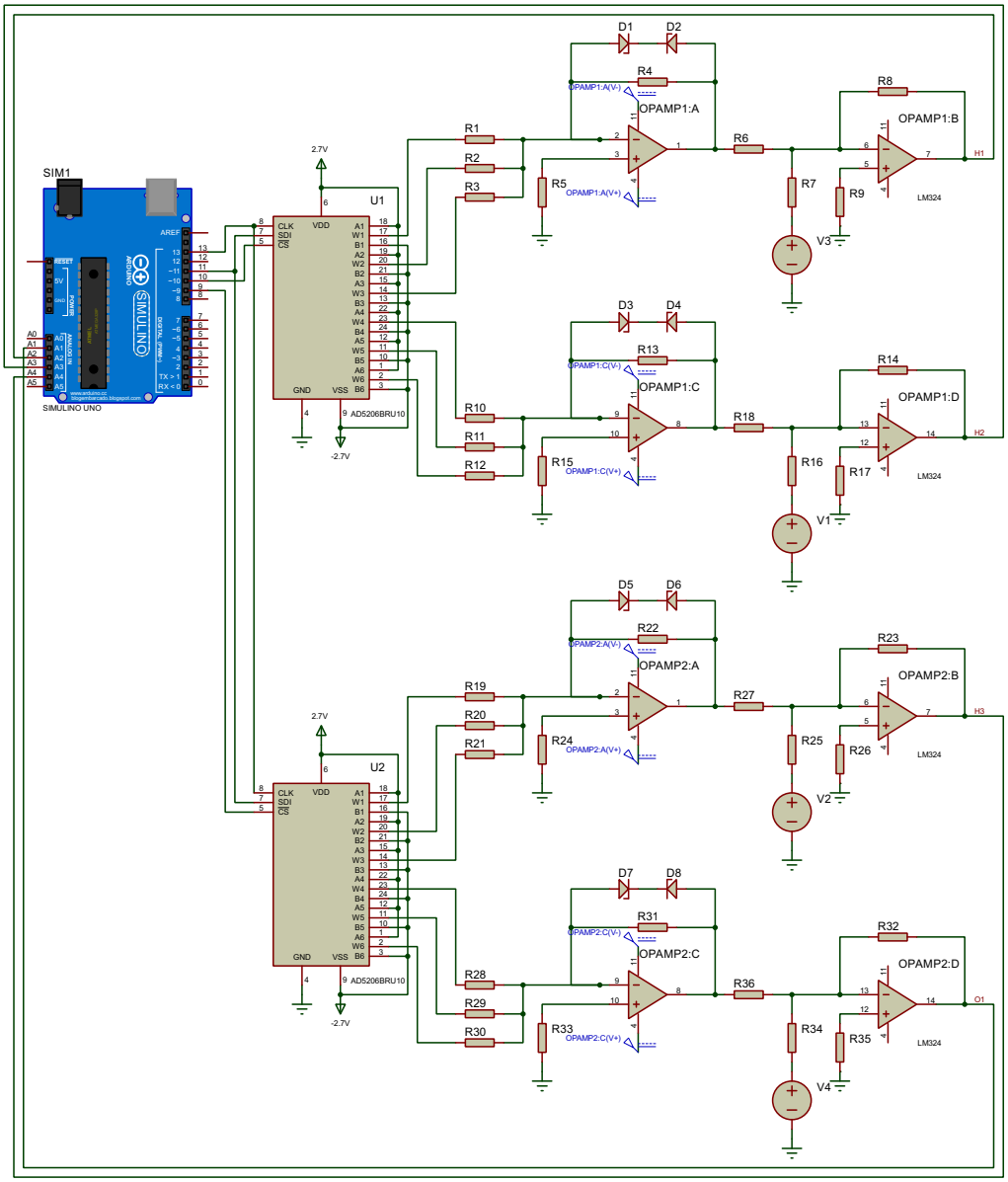


Figure A.1: Artificial Neural Network Circuit Scheme

Appendix B

Appendix

```
// Network Configuration
const int inputSize;
const int hiddenSize;
const int outputSize;
const int valuesSize;
float learningRate;
const float tolerance;
// Inputs
float inputValues[valuesSize][inputSize] = {};
// Desired Output
float targetValues[valuesSize][outputSize] = {};
// Constants
float inputWeights[hiddenSize][inputSize];
float hiddenWeights[outputSize][hiddenSize];
float outputs[valuesSize];
float error;
boolean done;

// Initialize random weights "inputWeights" and "hiddenWeights"

// Training
while (not done)
{
  // Send Input Values * Input Weights to Digital Potentiometer
  synapseH = inputValues * inputWeights;
  // Values to the hidden layer
  digitalPotWrite1(i, synapseH);

  // Reading the hidden weights values and scalling to 0-1 value.
  h_j = analogRead(j) / 1023.0;


  // Send Hidden Values * Hidden Weights to Digital Potentiometer
  synapseO = h_j * hiddenWeights;
  // Values to the hidden layer
  digitalPotWrite1(i, synapseH);

  // Reading the hidden weights values and scalling to 0-1 value.
  h_j = analogRead(j) / 1023.0;
  // Output value
  outputs = analogRead() / 1023.0;


  // Calculating the error
  error = outputs - targetValues;
  errors += error * error;

  // Updating input weights
  inputWeights[i][j] -= learningRate * error * outputs * (1.0 - outputs) * h_j *  (1 - h_j) * hiddenWeights *
      inputValues;

  //Updating hidden weights
  changeHiddenWeights[i][j] = learningRate * error * outputs * (1.0 - outputs) * h_j ;

  if (errors < tolerance)
  {
    done = true;
    Serial.println("Training done!");
  }
}
```

Listing B.1: Back propagation pseudo code.

Appendix C

Appendix

```
/*
  ARTIFICIAL NEURAL NETWORK CODE W/ BP
  Jacobo Posada Hoyos
  21/11/2020
*/

// include the SPI library:
#include <SPI.h>
// Network Configuration
const int inputSize = 2;
const int hiddenSize = 3;
const int outputSize = 1;
const int valuesSize = 4;
float learningRate = 0.05;
const float tolerance = 0.05;
// Inputs
float inputValues[valuesSize][inputSize] =
{
  {0, 0},
  {0, 1},
  {1, 0},
  {1, 1}
};
// Desired Output
float targetValues[valuesSize][outputSize] =
{
  {0},
  {1},
  {1},
  {1}
};
// Constants
float inputWeights[hiddenSize][inputSize];
float hiddenWeights[outputSize][hiddenSize];
float changeInputWeights[hiddenSize][inputSize];
float changeHiddenWeights[hiddenSize][outputSize];
float hiddenValues[hiddenSize];
float outputValues;
float outputs[valuesSize];
float errors;
float error;
int synapseH[hiddenSize][inputSize];
int synapseO[outputSize][hiddenSize];
int control = 0;
boolean done = false;
int ReportEvery1000;
int test;
int epoch = 0;

// Pins to select the Digital Potentiometers
const int slaveSelectPin1 = 10;
const int slaveSelectPin2 = 9;

// Outputs of hidden layer
int H1 = A2;
int H2 = A3;
int H3 = A4;

// Output Value
int O1 = A1;

void setup() {
  // set the slaveSelectPin as an output:
  pinMode(slaveSelectPin1, OUTPUT);
  pinMode(slaveSelectPin2, OUTPUT);
  // Report
```

```
67    ReportEvery1000 = valuesSize;
68    // initialize SPI:
69    SPI.begin();
70    // initialize Serial:
71    Serial.begin(9600);
72    // Initialize Random Values
73    randomSeed(analogRead(A3));
74  }
75
76  void loop() {
77    initialWeights(control);
78    toTerminalFirst(control);
79    control = 1;
80    test = 0;
81    while (not done)
82    {
83      error = 0.0;
84      errors = 0.0;
85
86      for (int q = 0 ; q < valuesSize ; q++ ) {
87
88        // Input Values * Input Weights
89        //Serial.println("Input W");
90        for (int i = 0; i < hiddenSize; i++)
91        {
92          for (int j = 0; j < inputSize; j++)
93          {
94            if (inputWeights[i][j] < 0)
95            {
96              synapseH[i][j] = int(abs(ceil((-inputWeights[i][j] * inputValues[q][j] * 127.0) - 127.0)));
97            }
98            else
99            {
100             synapseH[i][j] = int(ceil((inputWeights[i][j] * inputValues[q][j] * 128.0) + 127.0));
101           }
102         }
103       }
104       // Values to the hidden layer
105       digitalPotWrite1(0, synapseH[0][0]);
106
107       digitalPotWrite1(1, synapseH[0][1]);
108
109       //digitalPotWrite1(2, synapseH[0][2]);
110
111       digitalPotWrite1(3, synapseH[1][0]);
112
113       digitalPotWrite1(4, synapseH[1][1]);
114
115       //digitalPotWrite1(5, synapseH[1][2]);
116
117       digitalPotWrite2(0, synapseH[2][0]);
118
119       digitalPotWrite2(1, synapseH[2][1]);
120
121
122       // Reading the hidden weights values and scalling to 0-1 value.
123       hiddenValues[0] = analogRead(H1) / 1023.0;
124       //delay(10);
125       hiddenValues[1] = analogRead(H2) / 1023.0;
126       //delay(10);
127       hiddenValues[2] = analogRead(H3) / 1023.0;
128       //delay(10);
129
130       for (int i = 0; i < outputSize; i++)
131       {
132         for (int j = 0; j < hiddenSize; j++)
133         {
134           if (hiddenWeights[i][j] < 0)
135           {
136             synapseO[i][j] = int(abs(ceil((-hiddenWeights[i][j] * hiddenValues[j] * 127.0) - 127.0)));
137           }
138           else
139           {
140             synapseO[i][j] = int(ceil((hiddenWeights[i][j] * hiddenValues[j] * 128.0) + 127.0));
141           }
142         }
143       }
144       // Values to the output layer
145       digitalPotWrite2(3, synapseO[0][0]);
146
147       digitalPotWrite2(4, synapseO[0][1]);
148
149       digitalPotWrite2(5, synapseO[0][2]);
150
151       // Output value
152       outputValues = analogRead(O1) / 1023.0;
153       outputs[q] = outputValues;
154
155       // Calculating the error
156       error = outputValues - targetValues[q][0];
157       errors += error * error;
158       // Updating input weights
```

```
159       for (int i = 0; i < hiddenSize; i++)
160       {
161         for (int j = 0; j < inputSize; j++)
162         {
163           changeInputWeights[i][j] = learningRate * error * outputValues * (1.0 - outputValues) * hiddenValues[i] *
          (1 - hiddenValues[i]) * hiddenWeights[0][i] * inputValues[q][j];
164           if ((inputWeights[i][j] - changeInputWeights[i][j] > 1) or (inputWeights[i][j] - changeInputWeights[i][j]
        < -1))
165           {
166             inputWeights[i][j] = inputWeights[i][j];
167           }
168           else
169           {
170             inputWeights[i][j] -= changeInputWeights[i][j];
171           }
172         }
173       }
174
175       //Updating hidden weights
176       for (int i = 0; i < outputSize; i++)
177       {
178         for (int j = 0; j < hiddenSize; j++)
179         {
180           changeHiddenWeights[i][j] = learningRate * error * outputValues * (1.0 - outputValues) * hiddenValues[j]
        ;
181           if ((hiddenWeights[i][j] - changeHiddenWeights[i][j] > 1) or (hiddenWeights[i][j] - changeHiddenWeights[i
        ][j] < -1))
182           {
183             hiddenWeights[i][j] = hiddenWeights[i][j];
184           }
185           else
186           {
187             hiddenWeights[i][j] -= changeHiddenWeights[i][j];
188           }
189         }
190       }
191
192       // Report
193
194       ReportEvery1000 = ReportEvery1000 - 1;
195       if (ReportEvery1000 == 0)
196       {
197         Serial.println(errors / 2.0);
198         ReportEvery1000 = 1000;
199         epoch = epoch + 1;
200       }
201
202       if (q == (valuesSize - 1))
203       {
204
205         errors = errors / 2.0;
206         if (errors < 0.1)
207         {
208           learningRate = 0.01;
209         }
210         if (errors < tolerance)
211         {
212           test = test + 1;
213           if (test == 2)
214           {
215             done = true;
216             Serial.println("");
217             toTerminal();
218             Serial.println("");
219             Serial.println("Training done!");
220           }
221         }
222
223       }
224     }
225   }
226 }
227
228
229 // Initializing the weigths values for the input and hidden layers.
230 void initialWeights(int control)
231 {
232   if (control == 0)
233     Serial.println("Initial");
234   {
235     for (int i = 0; i < hiddenSize; i++)
236     {
237       for (int j = 0; j < inputSize; j++)
238       {
239         inputWeights[i][j] =  (float(random(100)) / 100) - 0.5;
240       }
241     }
242     for ( int i = 0; i < outputSize; i++)
243     {
244       for (int j = 0; j < hiddenSize; j++)
245       {
246         hiddenWeights[i][j] = (float(random(100)) / 100) - 0.5;
```

```
      }
    }
  }
}

// Sending the values for each Digital Potentiometer

void digitalPotWrite1(int address, int value) {
  // take the SS pin low to select the chip:
  digitalWrite(slaveSelectPin1, LOW);
  delay(10);
  //  send in the address and value via SPI:
  SPI.transfer(address);
  SPI.transfer(value);
  delay(10);
  // take the SS pin high to de-select the chip:
  digitalWrite(slaveSelectPin1, HIGH);
}

void digitalPotWrite2(int address, int value) {
  // take the SS pin low to select the chip:
  digitalWrite(slaveSelectPin2, LOW);
  delay(10);
  //  send in the address and value via SPI:
  SPI.transfer(address);
  SPI.transfer(value);
  delay(10);
  // take the SS pin high to de-select the chip:
  digitalWrite(slaveSelectPin2, HIGH);
}

// Printing values of interest
void toTerminal()
{
  for (int q = 0; q < valuesSize; q++)
  {
    Serial.println();
    Serial.print ("  Input ");
    for (int i = 0 ; i < inputSize ; i++ )
    {
      Serial.print (inputValues[q][i]);
      Serial.print (" ");
    }
    Serial.print ("  Target ");
    Serial.print (targetValues[q][0]);
    Serial.print (" ");

    Serial.print ("  Outputs = ");
    Serial.print (outputs[q], 5);
    Serial.print (" ");


  }
  Serial.println();
  Serial.println();
  Serial.print ("  Error = ");
  Serial.print (errors, 5);
  Serial.print (" ");
  Serial.println();
  Serial.print("  Input Weights = ");
  for ( int i = 0; i < hiddenSize; i++)
  {
    for (int j = 0; j < inputSize; j++)
    {
      Serial.print(inputWeights[i][j], 5);

      Serial.print (" ");
    }
  }
  Serial.println();
  Serial.print("  Hidden Weights = ");
  for ( int i = 0; i < outputSize; i++)
  {
    for (int j = 0; j < hiddenSize; j++)
    {
      Serial.print(hiddenWeights[i][j], 5);
      Serial.print (" ");
    }
  }
  Serial.println((epoch - 1) * 1000 + ReportEvery1000);
  Serial.println();
}


void toTerminalFirst(int control)
{
  if (control == 0)
  {
    for (int q = 0; q < valuesSize; q++)
    {
      Serial.println();
      Serial.print ("  Input ");
```

```
339         for (int i = 0 ; i < inputSize ; i++ )
340         {
341           Serial.print (inputValues[q][i]);
342           Serial.print (" ");
343         }
344         Serial.print ("  Target ");
345         Serial.print (targetValues[q][0]);
346         Serial.print (" ");
347       }
348
349     }
350 }
```

Listing C.1: Code for the Artificial Neural Network with BP algorithm

Appendix D

Appendix

```
Artificial Neural Network #
# Original code retrieved from: https://towardsai.net/building-neural-nets-with-python
# Import libraries
import numpy as np
from numpy.random import rand, seed
# Define a seed to generate the same random values
seed(2)
# Define input values :
input_features = np.array([[0,0],[0,1],[1,0],[1,1]])
print (input_features.shape)
print (input_features)
# Define target output :
target_output = np.array([[0,1,1,0]])
# Reshaping our target output into vector :
target_output = target_output.reshape(4,1)
print(target_output.shape)
print (target_output)
# Define weights :
# 6 for hidden layer
# 3 for output layer
# 9 total
# Generation of initial weights
weight_hidden = np.random.rand(2,3)
weight_output = np.random.rand(3,1)
# Learning Rate :
lr = 0.05
# Sigmoid function :
def sigmoid(x):
 return 1/(1+np.exp(-x))
# Derivative of sigmoid function :
def sigmoid_der(x):
 return sigmoid(x)*(1-sigmoid(x))
# Training
for epoch in range(200000):
 # Input for hidden layer :
 input_hidden = np.dot(input_features, weight_hidden*10)

 # Output from hidden layer :
 output_hidden = sigmoid(input_hidden)

 # Input for output layer :
 input_op = np.dot(output_hidden, weight_output*10)

 # Output from output layer :
 output_op = sigmoid(input_op)
#============================================================
 # Phase1

 # Calculating Mean Squared Error :
 error_out = ((1 / 2) * (np.power((output_op - target_output), 2)))
 print(error_out.sum())
 # Derivatives for phase 1 :
 derror_douto = output_op - target_output
 douto_dino = sigmoid_der(input_op)
 dino_dwo = output_hidden
 derror_dwo = np.dot(dino_dwo.T, derror_douto * douto_dino)
#============================================================
 # Phase 2
 # derror_w1 = derror_douth * douth_dinh * dinh_dw1
 # derror_douth = derror_dino * dino_outh

 # Derivatives for phase 2 :
 derror_dino = derror_douto * douto_dino
 dino_douth = weight_output*10
 derror_douth = np.dot(derror_dino , dino_douth.T)
 douth_dinh = sigmoid_der(input_hidden)
```

```
 dinh_dwh = input_features
 derror_wh = np.dot(dinh_dwh.T, douth_dinh * derror_douth)
# Update Weights
 for i in range(2):
   for j in range(3):
     if weight_hidden[i][j] - lr * derror_wh[i][j] > 1:
       weight_hidden[i][j] = weight_hidden[i][j]
     elif weight_hidden[i][j] - lr * derror_wh[i][j] < -1:
       weight_hidden[i][j] = weight_hidden[i][j]
     else:
       weight_hidden[i][j] -= lr * derror_wh[i][j]
 for i in range(3):
   if weight_output[i] - lr * derror_dwo[i] > 1:
     weight_output[i] = weight_output[i]
   elif weight_output[i] - lr * derror_dwo[i] < -1:
     weight_output[i] = weight_output[i]
   else:
     weight_output[i] -= lr * derror_dwo[i]
# Final hidden layer weight values :
print (weight_hidden)
# Final output layer weight values :
print (weight_output)

#============================================
# Predictions :
#Taking inputs :
single_point = np.array([1,1])
#1st step :
result1 = np.dot(single_point, weight_hidden*10)
#2nd step :
result2 = sigmoid(result1)
#3rd step :
result3 = np.dot(result2,weight_output*10)
#4th step :
result4 = sigmoid(result3)
print(result4)
#============================================
```

Listing D.1: ANN code in Python

Bibliography

[1] C. Yao, Y. Qu, B. Jin, L. Guo, C. Li, W. Cui, and L. Feng, "A convolutional neural network model for online medical guidance," *IEEE Access*, vol. 4, pp. 4094–4103, 2016.

[2] Q. Do, T. C. Son, and J. Chaudri, "Classification of asthma severity and medication using tensorflow and multilevel databases," *Procedia Computer Science*, vol. 113, pp. 344 – 351, 2017. The 8th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2017) / The 7th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2017) / Affiliated Workshops.

[3] P. Bawane, S. Gadariye, S. Chaturvedi, and A. Khurshid, "Object and character recognition using spiking neural network," *Materials Today: Proceedings*, vol. 5, no. 1, Part 1, pp. 360 – 366, 2018. International Conference on Processing of Materials, Minerals and Energy (July 29th – 30th) 2016, Ongole, Andhra Pradesh, India.

[4] V. A. Torres, B. R. Jaimes, E. S. Ribeiro, M. T. Braga, E. H. Shiguemori, H. F. Velho, L. C. Torres, and A. P. Braga, "Combined weightless neural network fpga architecture for deforestation surveillance and visual navigation of uavs," *Engineering Applications of Artificial Intelligence*, vol. 87, p. 103227, 2020.

[5] C. S. Lindsey and T. Lindblad, "Review of hardware neural networks: A User's perspective," in *3rd Workshop on Neural Networks: From Biology to High-energy Physics Marciana Mariana, Italy, September 26-30, 1994*, pp. 0215–224, 1994.

[6] B. Erkmen, R. A. Vural, N. Kahraman, and T. Yildirim, "A mixed mode neural network circuitry for object recognition application," *Circuits, Systems, and Signal Processing*, vol. 32, pp. 29–46, Feb 2013.

[7] D. Maliuk and Y. Makris, "An experimentation platform for on-chip integration of analog neural networks: A pathway to trusted and robust analog/rf ics," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 26, pp. 1721–1734, Aug 2015.

[8] V. Kumar, P. Gaur, and A. Mittal, "Ann based self tuned pid like adaptive controller design for high performance pmsm position control," *Expert Systems with Applications*, vol. 41, no. 17, pp. 7995 – 8002, 2014.

[9] D. Ma, J. Shen, Z. Gu, M. Zhang, X. Zhu, X. Xu, Q. Xu, Y. Shen, and G. Pan, "Darwin: A neuromorphic hardware co-processor based on spiking neural networks," *Journal of Systems Architecture*, vol. 77, pp. 43 – 51, 2017.

[10] D. Baptista, S. Abreu, C. Travieso-González, and F. Morgado-Dias, "Hardware implementation of an artificial neural network model to predict the energy production of a photovoltaic system," *Microprocessors and Microsystems*, vol. 49, pp. 77 – 86, 2017.

[11] H. Phan-Xuan, T. Le-Tien, and S. Nguyen-Tan, "Fpga platform applied for facial expression recognition system using convolutional neural networks," *Procedia Computer Science*, vol. 151, pp. 651 – 658, 2019. The 10th International Conference on Ambient Systems, Networks and Technologies (ANT 2019) / The 2nd International Conference on Emerging Data and Industry 4.0 (EDI40 2019) / Affiliated Workshops.

[12] H. V. H. Ayala, D. M. Muñoz, C. H. Llanos, and L. dos Santos Coelho, "Efficient hardware implementation of radial basis function neural network with customized-precision floating-point operations," *Control Engineering Practice*, vol. 60, pp. 124 – 132, 2017.

[13] L. Yang, Z. Zeng, and X. Shi, "A memristor-based neural network circuit with synchronous weight adjustment," *Neurocomputing*, vol. 363, pp. 114 – 124, 2019.

[14] D. Chabi, Z. Wang, C. Bennett, J. Klein, and W. Zhao, "Ultrahigh density memristor neural crossbar for on-chip supervised learning," *IEEE Transactions on Nanotechnology*, vol. 14, pp. 954–962, Nov 2015.

[15] Z. Tang, R. Zhu, P. Lin, J. He, H. Wang, Q. Huang, S. Chang, and Q. Ma, "A hardware friendly unsupervised memristive neural network with weight sharing mechanism," *Neurocomputing*, vol. 332, pp. 193 – 202, 2019.

[16] P. Masa, K. Hoen, and H. Wallinga, "70 input, 20 nanosecond pattern classifier," in *Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94)*, vol. 3, pp. 1854–1859 vol.3, 1994.

[17] Y. Du, L. Du, X. Gu, J. Du, X. S. Wang, B. Hu, M. Jiang, X. Chen, S. S. Iyer, and M. F. Chang, "An analog neural network computing engine using cmos-compatible charge-trap-transistor (ctt)," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, pp. 1811–1819, Oct 2019.

[18] L. Chua, "Memristor-the missing circuit element," *IEEE Transactions on Circuit Theory*, vol. 18, pp. 507–519, Sep. 1971.

[19] D. R. Stewart and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, pp. 80–83, May. 2008.

[20] W. Choi, K. Moon, M. Kwak, C. Sung, J. Lee, J. Song, J. Park, S. A. Chekol, and H. Hwang, "Hardware implementation of neural network using pre-programmed resistive device for pattern recognition," *Solid-State Electronics*, vol. 153, pp. 79 – 83, 2019.

[21] S. P. Adhikari, C. Yang, H. Kim, and L. O. Chua, "Memristor bridge synapse-based neural network and its learning," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 23, pp. 1426–1435, Sep. 2012.

[22] H. Kim and C. Yang, "Design of cellular neural network architecture using memristors," in *2015 International SoC Design Conference (ISOCC)*, pp. 207–208, Nov 2015.

[23] E. Bilotta, P. Pantano, and S. Vena, "Speeding up cellular neural network processing ability by embodying memristors," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, pp. 1228–1232, May 2017.

[24] C. Yakopcic, M. Z. Alom, and T. M. Taha, "Memristor crossbar deep network implementation based on a convolutional neural network," in *2016 International Joint Conference on Neural Networks (IJCNN)*, pp. 963–970, July 2016.

[25] B. RayChaudhuri, "Prototype neural network hardware with analog electronic circuit," *Indian Journal of Physics*, vol. 84, pp. 1435–1440, Oct 2010.

[26] S. A. Rahman and M. S. Ansari, "A neural circuit with transcendental energy function for solving system of linear equations," *Analog Integrated Circuits and Signal Processing*, vol. 66, pp. 433–440, Mar 2011.

[27] M. Kawaguchi, N. Ishii, and M. Umeno, "Analog neural circuit and hardware design of deep learning model," *Procedia Computer Science*, vol. 60, pp. 976 – 985, 2015. Knowledge-Based and Intelligent Information & Engineering Systems 19th Annual Conference, KES-2015, Singapore, September 2015 Proceedings.

[28] T. Oliveira Weber, D. da Silva Labres, and F. L. Cabrera, "Amplifier-based mos analog neural network implementation and weights optimization," in *2019 32nd Symposium on Integrated Circuits and Systems Design (SBCCI)*, pp. 1–6, Aug 2019.

[29] F. Sarwar, S. Iqbal, and M. Hussain, "Linear and nonlinear electrical models of neurons for hopfield neural network," *Zeitschrift für Naturforschung A*, vol. 71, pp. 995–1002, Oct 2016.

[30] O. Krestinskaya, K. N. Salama, and A. P. James, "Learning in memristive neural network architectures using analog backpropagation circuits," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, pp. 719–732, Feb 2019.

[31] F. Rosenblatt, "Perceptron simulation experiments," *Proceedings of the IRE*, vol. 48, pp. 301–309, March 1960.

[32] A. Gómez-Espinosa, R. Castro Sundin, I. Loidi Eguren, E. Cuan-Urquizo, and C. D. Treviño-Quintanilla, “Neural network direct control with online learning for shape memory alloy manipulators,” *Sensors*, vol. 19, no. 11, 2019.

[33] M. Ermini, J. Dhanasekar, and V. K. Sudha, “Memristor emulator using mcp 3208 and digital potentiometer,” in *ICTACT JOURNAL ON MICROELECTRONICS*, 2018.

Curriculum Vitae

Jacobo Posada Hoyos was born in Manizales, Caldas, Colombia on October 28, 1996. He earned his B.S. degree in Electronic Engineering from the Universidad Nacional de Colombia, Manizales Campus in August 2019. He was accepted in the graduate programs in Engineering Sciences in August 2019.

This document was typed in using LaTeX 2_ε[a] by Jacobo Posada Hoyos.

[a]The style file `phdThesisFormat.sty` used to set up this thesis was prepared by the Center of Intelligent Systems of the Instituto Tecnológico y de Estudios Superiores de Monterrey, Monterrey Campus