Nile University

ECEN-409: Microprocessor Design

Instructor: Dr. Mohamed Ibrahim

“Electrocardiograph Arrythmia Disease Classification using Artificial Neural Network on FPGA ”

Fall 2018

Submitted By:

Youssef ElGendy	**1610190**
Mohamed Ayman	**1510162**
Al-Shaimaa Samir	**1610758**
Ahmed Yamout	**1510264**
Ahmed Samier	**1510217**

I. Abstract

Arrythmia is a well known medical condition which implies that a patients heart beats too slow, too fast, or in an irregular pattern. The condition can sometimes lead to serious problems and complications if ignored and not treated. While the machine learning problem of classifying ECG --electrocardiograph- into healthy and unhealthy individuals has been studied several times, however, we propose some modifications and different implementation architectures to implement the network. The reason why the network is implemented on FPGA is that the key feature that FPGA has over ASICs is its programmability. We aim to implement an artificial neural network that classifies healthy from arithmetic patients based on average psd and energy of their ECG signals. The network has 3 layers; input, hidden, output where each layer has 2,16, 2 perceptrons, respectively. After training the weights and biases of each layer are saved in their corresponding memories in their layer. The network implemented in VHDL makes use of those attributes, and propagates forward to obtain the final result, which is passed to a comparator to determine which perceptron has a higher output. The VHDL code was uploaded on ZYBO ZYNQ 7020 FPGA, and the output was as expected.

II. Contents

III. List of Figures

IV. Introduction

With the advances of technology and the usage of Artificial intelligence in nearly every application nowadays, thus the rise of special purpose hardware dedicated for AI computations. Moreover, major companies in the field started creating their own application specific integrated circuits -abbreviated as ASICS-. Google to start off created their own ASIC named Tensor processing unit -abbreviated as TPU-.

In addition to intel creating and releasing in 2019 the intel Nervana Neural Network Processor, this architecture basis provides flexibility to all deep learning applications. Although, this hardware is the most compatible for deep learning, but they are expensive for normal computer architects to use. Thus, the usage of field programmable gate arrays -abbreviated as FPGA- that fills that gap, by providing complex computations with a cheaper expense. Furthermore, FPGA's are rapidly reconfigurable for various verifications of ASIC design.

In this project we implement an Artificial Neural Network -abbreviated as ANN- on an FPGA, were hardware implementations, also referred to as accelerators are much faster than software based ANNs. Moreover, we implement a software based neural network, with forward propagation and back propagation to obtain the weights. Then we embed these weights on the hardware ANN, which implements forward propagation with higher speed and provides means of scalability for further implementations, as an FPGA is reconfigurable.

Well FPGA is an integrated circuit made up of thousands of Configurable Logic Blocks which is made of Multiplexers, Flip-Flops and Look-Up Tables. The circuit is made by connecting these CLBs with other blocks like RAM block, PPLs, External Memory Controllers, and DSP Blocks. However, the logic function is also written in HDL, but it can be reprogrammed again. It can be used as microprocessor then changed to graphics card and it does not cost a lot for NRE design.

V. Literature Review

I. The Classification Problem:

Classification is the problem of identifying to which of a set of categories (sub-populations) a new observation belongs, On the basis of a training set of data containing observations (or instances) whose category membership is known. Examples are assigning a given email to the "spam" or "non-spam" class, and assigning a diagnosis to a given patient based on observed characteristics of the patient (sex, blood pressure, presence or absence of certain symptoms, etc.). Classification is an example of pattern recognition.

In most cases, classification is a supervised learning, in which, there is a training set with known output. The process try to build a **classifier** that given a specific set of input parameters “features” it classify this observation “instance” into the category that match those features “class”. There are many classification methods across the fields. Statistical methodologies like linear regression is one of the methods used to build classifier, though, the most common methods is using ANN “Artificial Neural network**”**.

A. Artificial Neural network:

Inspired by the brain **Artificial neural networks** (ANN) are computing systems inspired by the biological neural networks that constitute animal brains. They are frameworks for many different machine learning algorithms to work together and process complex data inputs. Such systems "learn" to perform tasks by considering examples, generally without being programmed with any task-specific rules.

For example, in image recognition, they might learn to identify images that contain cats by analyzing example images that have been manually labeled as "cat" or "no cat" and using the results to identify cats in other images. They do this without any prior knowledge about cats, for example, that they have fur, tails, whiskers and cat-like faces. Instead, they automatically generate identifying characteristics from the learning material that they process.

ANN is one of the most common and most efficient ways to implement classifier. Different architectures and design of ANN each has its own benefits and drawbacks are being used in this field. Choosing the right architecture for the problem can be very tedious, yet, it’s very important to achieve the highest possible accuracy with the least computation power.

The original goal of the ANN approach was to solve problems in the same way that a human brain would. However, over time, attention moved to performing specific tasks, leading to deviations from biology. Artificial neural networks have been used on a variety of tasks, including computer vision, speech recognition, machine translation, social network filtering, playing board and video games and medical diagnosis.

B. Different Architectures:

There are so many types of networks to choose from and new methods being published and discussed every day. To make things worse, most neural networks are flexible enough that they work (make a prediction) even when used with the wrong type of data or prediction problem. Three of the most know architectures are:

- Multilayer Perceptron (MLPs)
- Convolutional Neural Networks (CNNs)
- Recurrent Neural Networks (RNNs)

These three classes of networks provide a lot of flexibility and have proven themselves over decades to be useful and reliable in a wide range of problems. They also have many subtypes to help specialized them to the quirks of different framings of prediction problems and different datasets.

Multi Layer Perceptron:

They are comprised of one or more layers of neurons. Data is fed to the input layer, there may be one or more hidden layers providing levels of abstraction, and predictions are made on the output layer, also called the visible layer. MLPs are suitable for classification prediction problems where inputs are assigned a class or label.

They are also suitable for regression prediction problems where a real-valued quantity is predicted given a set of inputs. Data is often provided in a tabular format, such as you would see in a CSV file or a spreadsheet.

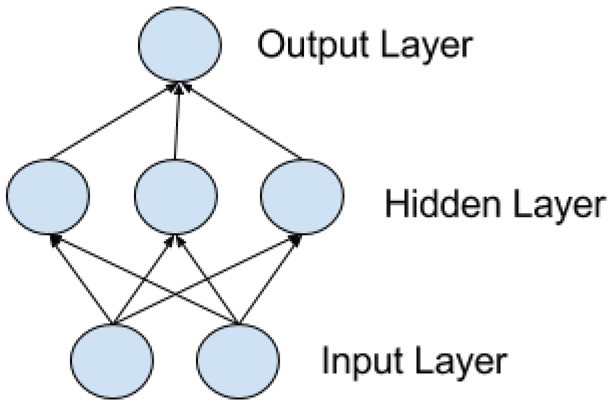

Figure 1 - Multi Layer Perceptron Structure

Some uses of MLP is in: Tabular datasets, Classification prediction problems, Regression prediction problems.

This flexibility allows them to be applied to other types of data. For example, the pixels of an image can be reduced down to one long row of data and fed into a MLP. The words of a document can also be reduced to one long row of data and fed to a MLP. Even the lag observations for a time series prediction problem can be reduced to a long row of data and fed to a MLP.

Convolution neural network:

Convolutional Neural Networks, or CNNs, were designed to map image data to an output variable.

They have proven so effective that they are the go-to method for any type of prediction problem involving image data as an input.

The benefit of using CNNs is their ability to develop an internal representation of a two-dimensional image. This allows the model to learn position and scale in variant structures in the data, which is important when working with images. Additionally, CNN is used for: Image data, Classification prediction problems, Regression prediction problems.

More generally, CNNs work well with data that has a spatial relationship.

The CNN input is traditionally two-dimensional, a field or matrix, but can also be changed to be one-dimensional, allowing it to develop an internal representation of a one-dimensional sequence.

This allows the CNN to be used more generally on other types of data that has a spatial relationship. For example, there is an order relationship between words in a document of text. There is an ordered relationship in the time steps of a time series.

Recurrent Neural Networks:

Recurrent Neural Networks, or RNNs, were designed to work with sequence prediction problems. Some examples of sequence prediction problems include:

- **One-to-Many**: An observation as input mapped to a sequence with multiple steps as an output.
- **Many-to-One**: A sequence of multiple steps as input mapped to class or quantity prediction.
- **Many-to-Many**: A sequence of multiple steps as input mapped to a sequence with multiple steps as output.

The Many-to-Many problem is often referred to as sequence-to-sequence, or seq2seq for short.

Recurrent neural networks were traditionally difficult to train.

The Long Short-Term Memory, or LSTM, network is perhaps the most successful RNN because it overcomes the problems of training a recurrent network and in turn has been used on a wide range of applications.

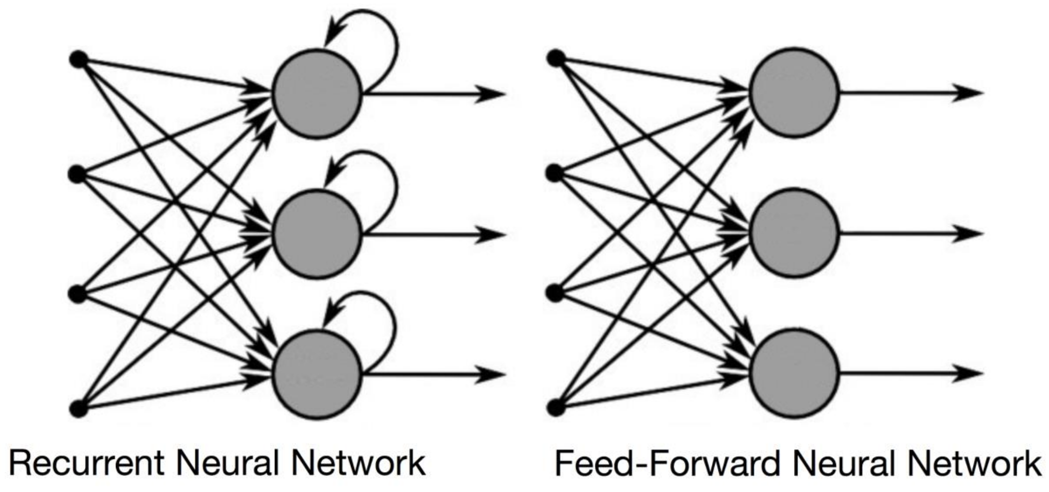

Figure 2 - RNN vs FFNN

RNNs in general and LSTMs in particular have received the most success when working with sequences of words and paragraphs, generally called natural language processing.

This includes both sequences of text and sequences of spoken language represented as a time series. They are also used as generative models that require a sequence output, not only with text, but on applications such as generating handwriting.

Recurrent neural networks are not appropriate for tabular datasets as you would see in a CSV file or spreadsheet. They are also not appropriate for image data input.

II. Activation Functions

In order to implement the network on FPGA, we have to pay special attention to the activation function, which will cause firing of the neurons. Since the output of the activation function is used as an input to the next layer, if an error occurs it will be propagated, amplified throughout the layers, and will cause misleading results. If we want our network to have a high accuracy, then we should implement an activation function of high precision. Sigmoid function –a non-linear function of exponential- is the most famous when it comes to implementing such networks due to the smooth transition it offers between inputs and output, however, its not the only valid activation function.

A. Sigmoid

$$y = \frac{1}{1 + e^{-x}}$$

Where y is the output, and x is the output, mapped between (0, 1)

However, in real hardware implementation, the sigmoid function cannot be implemented this way on FPGAs, as it requires division and exponentiation, which are two of the most demanding processes in terms of resources and clock cycles needed.

Therefore, approximation is done to overcome the complexity of calculating a sigmoid function on hardware.

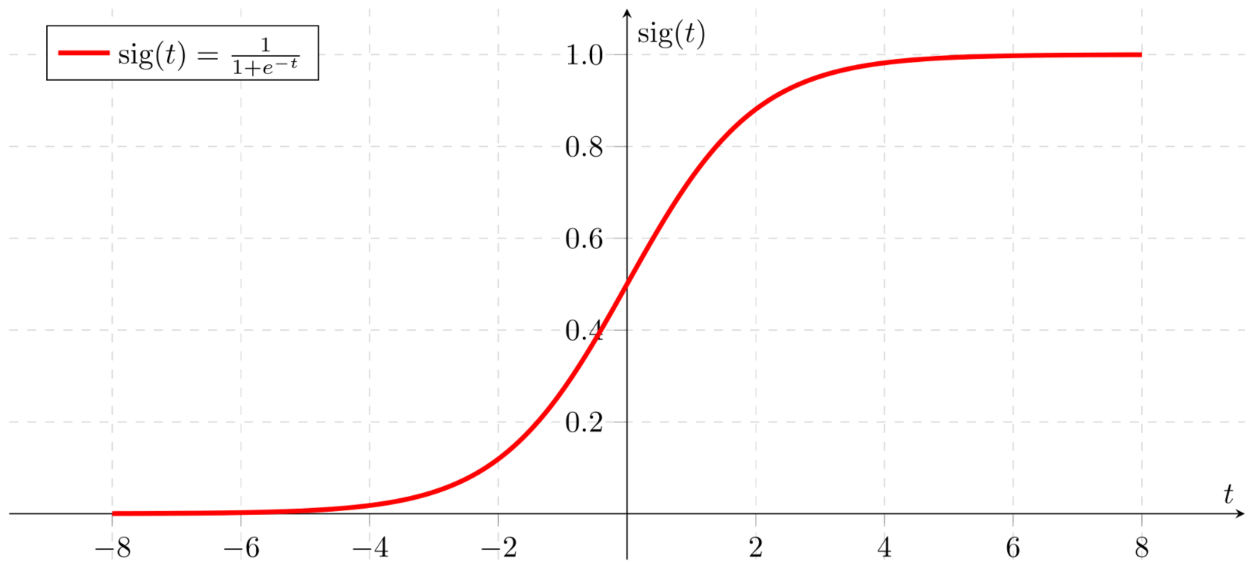

Figure 3 - Sigmoid Function

B. Sigmoid Approximation

Some sigmoid approximations include sampling the function and storing values (LUTS), and others include doing modifications to turn the traditional sigmoid function into another feasible form.

Lookup Tables (LUT)

Lookup tables are the simplest ways to approximate sigmoid. The function is broken into many segments, and each segment is stored in memory. Although this method is relatively fast and does not require any complex calculations, it requires the use of a big portion of memory, especially if a high precision is needed. The precision is directly proportional to the number of segments, and each of them is directly proportional to the amount of memory needed too.

However, a saturated region –where the output approaches 1 or 0- is ignored, and only the unsaturated region is segmented. In addition, we use the property of sigmoid that each value of input produces an output that is the compliment of the output produced by that input multiplied by -1, and so we cut down out region of interest to half the non-saturated region.

Deviation between (LUTs) and sigmoid function ranges from -0.005 to 0.005

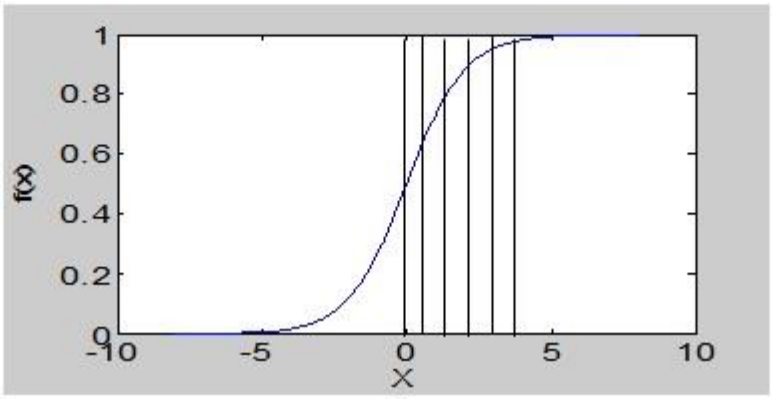

Figure 4 - Sigmoid LUT Sampling

Simple Implementation of Sigmoid

Another simpler way to compute sigmoid is to get a function that holds the same properties of no-linearity and continuous first derivative, in addition to approaching 1 at infinity and 0 at negative infinity. The new function should use less hardware than that required by the exponentiation and division in the original sigmoid function.

$$y = \frac{1}{2}\left(\frac{x}{1+|x|} + 1\right)$$

This new implementation requires two adders, a divider, a divide by 2, and absolute value circuits, which are far less complex and more available than the exponentiation circuit.

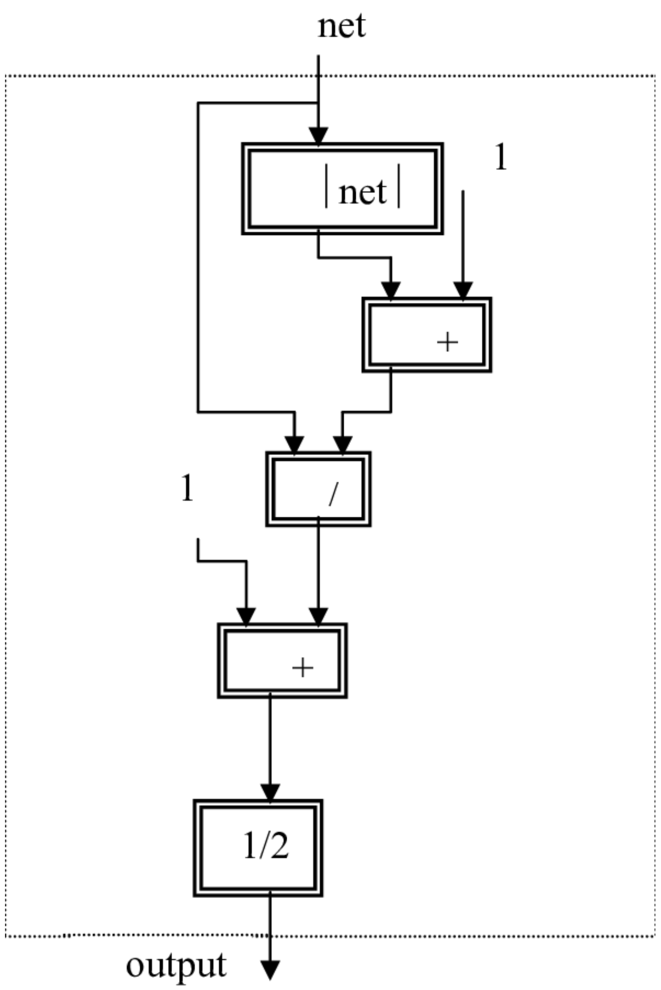

Figure 5 - Approximated Sigmoid

C. ReLU

It is another used activation function which stands for Rectified Linear Unit (***ReLU***). When the input is less than 0, the output is zero. However, as the input crosses the zero, the output becomes equal to the input, which means the line has a positive slope of 1.

$$y = \max(0, x)$$

Compared to other activation functions –mainly the sigmoid and tanh-, the ReLU is considered a low-cost function, and accelerates convergence due to its non-saturating linear form.

D. Tanh

$$y = \frac{e^{2x} - 1}{e^{2x} + 1}$$

The tanh function has got nearly similar characteristics to the sigmoid function. It provides a smooth transform between the input and output, in addition to approaching 1 at infinity. However, the function approaches -1 at negative infinity which means it maps the input to a value between [-1,1]. Like the sigmoid but unlike the ReLU, its activation saturates, and it is centered at the (0). As observed, it is symmetric about the origin (which means it is odd), and this means in practice, its non-linearity is preferable over the sigmoid. In fact, the tanh can be considered as a scaled sigmoid.

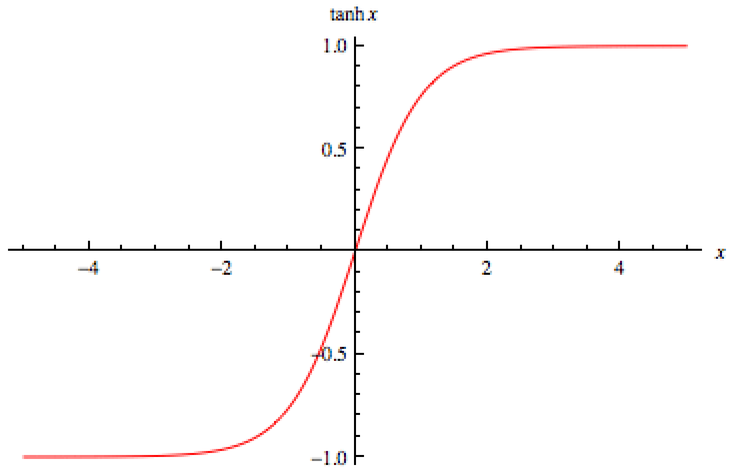

Figure 6 - Tanh Function

III. Data Sets

As mentioned earlier, an ECG records the electrical activity of the heart at relaxation state. It provides info about the rate of heart beats and wave oscillations. Moreover, for the classification of ECG, we basically need a dataset containing various heart beat signals for interpretation. Thus, we explored various data set sources, starting from Mendeley, Kaggle and Physionet.

They all provide reliable data set that are obtained from real life patients with clarified headers, that provide information about each patient. However, the researchers decided to use a dataset provided from Physionet found at https://physionet.org/physiobank/database/ , and to be exact the dataset provided by the MIT-BIH arrythmia for the positive tests (has a fast heart beat signal), which is found at https://physionet.org/physiobank/database/mitdb/ , and the MIT-BIH Normal Sinus Rhythm Database found at https://physionet.org/physiobank/database/nsrdb/ for the negative tests(has a normal heart beat signal).

IV. Preprocessing

Any ECG signal obtained from a dataset should go through a preprocessing stage, to extract the desired features that can be the input basis for an artificial neural network -abbreviated ANN- to classify the ECG into a specific class. Moreover, we will discuss various preprocessing techniques that can be used, to provide efficient feature extraction. However, before getting into the various preprocessing techniques, we firstly explain the main features in an ECG signal. A normal sinus rhythm ECG of a heart is shown in figure (1), starting with the P wave that indicates that the atria are contracting, pumping blood into the ventricles.

The QRS complex, typically beginning with a descending deflection, Q; a higher ascending deflection, a peak (R); and then a descending S wave. The QRS complex denotes ventricular depolarization and contraction. The PR interval denotes the transit time for the electrical signal to travel from the sinus node to the ventricles. T wave is normally a modest upwards waveform representing ventricular repolarization.

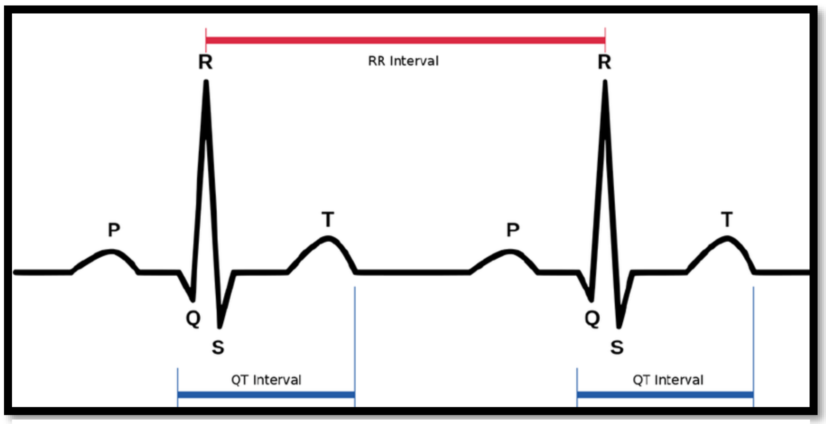

Figure 7 - ECG Signal

A. Interval Preprocessing

As mentioned earlier, an ECG signal is composed of various intervals that defines the wave, and these intervals are basis for any real time heart disease classification. Firstly, using as inputs the QRS interval and the PR. The combination of these two, allows a large sweep of the signal, which allows reasonable classification. Moreover, by increasing the inputs, we can also take into consideration the RR interval, which is the time between each R peak.

B. Power and Energy Preprocessing

Another approach is to look at the contents of an ECG signal from a general point, which is the power spectral density -abbreviated as PSD- and the energy of a signal. PSD displays the strength of the variations as a function of frequency. Moreover, it shows at which frequencies variations are solid and at which frequencies variations are weak. The unit of PSD is energy per frequency, in addition to the ability of obtaining energy within a specific frequency range by integrating PSD within that frequency range. Furthermore, energy is also a strength measurement of a signal, when the power of a signal is finite or is near zero.

It is obvious to anyone who thinks about the algebraic method for balancing chemical equations that the issue is to solve a system of homogeneous linear equations. Different writers have in this manner called attention to that matrices are important Instruments for balancing chemical equations. They didn't utilize all the most fitting scientific techniques (particularly module theory (II)) to analyze the solution, and gave short shrift to the topic of what happens when the adjusting of the first skeletal chemical equation make up a two (or more) parameter family. In any case, they opened up fascinating theoretical methodologies and their work might be influential over the long run.

The idea of the matrix method is to turn a chemical equation or a list of chemical species into a matrix. It is obtained by entering the number of atoms of a given element e occurring in a given type of reagent molecule m into the matrix in the row corresponding to e and the column corresponding to m.

We then use Gauss-Jordan elimination by row reduction to bring the original matrix to Hermite normal form. Then we use standard techniques to find a vector space basis for the collection of all column vectors A, this collection is called the kernel, or the null space.

Balancing of the original skeletal chemical equation correspond to column vectors A which Hermite normal form of the matrix right annihilates, i.e., column vectors, such that the matrix product of the Hermite form and the column vector is equal to the zero column vector 0. (Blakley,1982)

V. Fixed Point Representation

A fixed point binary number refers to a number that has a definite number of binary fraction bits, while the rest of the number is referred to as the integer part.

Unlike IEEE standard float representation -which uses a mantissa and some fraction bits-, fixed point representation is more flexible in calculations, as it can be dealt with as an ordinary integer.

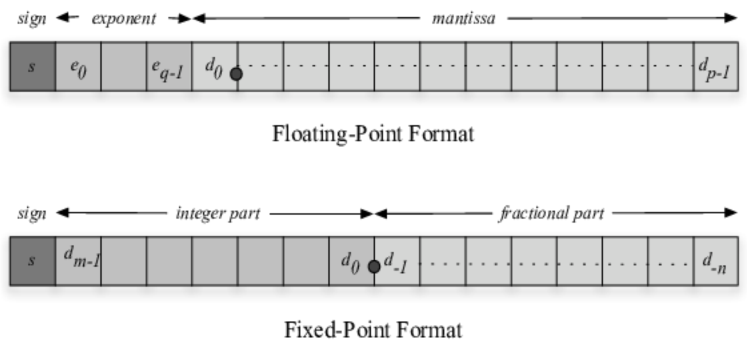

Figure 8 - Fixed Point vs IEEE Floating Point

Fixed Point Multiplication

In the multiplication of two fixed-point numbers, we deal with them as ordinary integers and perform integer multiplication. If both numbers have the same number of fraction bits, then the output will have exactly double that number of fraction bits. In order to keep up with the consistency of the fixed-point representation, we will have to reduce the number of fraction bits into half, either by truncation or rounding.

```
                 Fractional       Integer
               Interpretation  Interpretation
     0.10          1/2               2
x    0.11      x   3/4           x   3
 ---------
   . 010
   .010
  0.00
 ---------     ---------       ---------
  0.0110           3/8               6
```

VI. Methodology

VI. Preprocessing

We decided to take the approach of the PSD and energy preprocessing as with our current dataset, the interval preprocessing showed worse results than the power and energy preprocessing. To clarify, the approach of interval preprocessing is not always the worse, however given the fact that the data is compact, and the dataset is small, which does not give it enough capability to train and show its true potential. Firstly, the PSD of the ECG is computed using Welch's method. The method divides the time series into $k = \frac{N}{M}$ segments of M samples and their equations are:

$$x^i(n) = x\big(n + M(i-1)\big) \quad 0 < n < M-1, \;\; 1 < i < K \quad (1)$$

Moreover, a modified periodogram is shown below:

$$J_M^i(w) = \frac{1}{MU}\left|\sum_{n=0}^{n=M-1} x^i(n)w(n)e^{-jwn}\right|^2 \; i = 1 \to k \qquad (2)$$

$$U = \frac{1}{M}\sum_{n=0}^{n=M-1} w(n)^2 \qquad (3)$$

Equation (4) is basically the averaging of the PSD:

$$B_{xx}^{w}(w) = \frac{1}{k}\sum_{i=1}^{i=k} J_M^i(w) \qquad (4)$$

Then the energy is obtained as follows for an ECG sequence of length n:

$$E = \sum_{i=1}^{i=n} x^2 \quad (5)$$

The PSD and the energy are then fed to the input layer of the neural network. Moreover, they are used in the feedforward and backpropagation of the ANN.

C. Preprocessing Implementation

In the previous section we described the analytical procedure of preprocessing the PSD and energy to be prepared to be the input of our neural network. Moreover, in this section we will describe our implementation of the preprocessing techniques using Python programming language and external libraries.

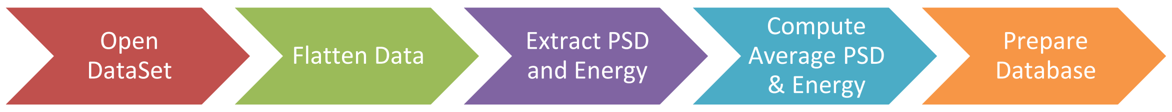

Figure 9: Preprocessing Procedure

Open Dataset

Firstly, we downloaded the database of the healthy patients (MIT-BIH Normal Sinus Wave) and the patients with arrythmia (MIT-BIH arrythmia) to our local directory using manual installation. Furthermore, we used the **wfdb library** to open the files which consisted of.dat, attr and .head files.

Flattening Data

Secondly, we accessed the p_signal, which is the signal in the attribute file and then we flatten the data, by selecting the first channel provided in each data sample for the next preprocessing stage.

PSD & Energy Extraction

In python there is a library called **scipy**, that provides a module that is suitable for signal processing, which is the (**signal**) and it has a built-in **periodogram** that takes as input the signal needed for computation and the frequency. Moreover, scipy.signal also provides a built-in **welch**, which is used to compute the power spectral density of a signal.

Compute average PSD & Energy

This stage is simple, the average PSD is obtained by the summation of the PSD over the total length of the signal. Additionally, the average energy is computed using the summation of the first ECG signal squared over the length of the total signal.

VII. Proposed Architectures

A. Proposed Architecture I:

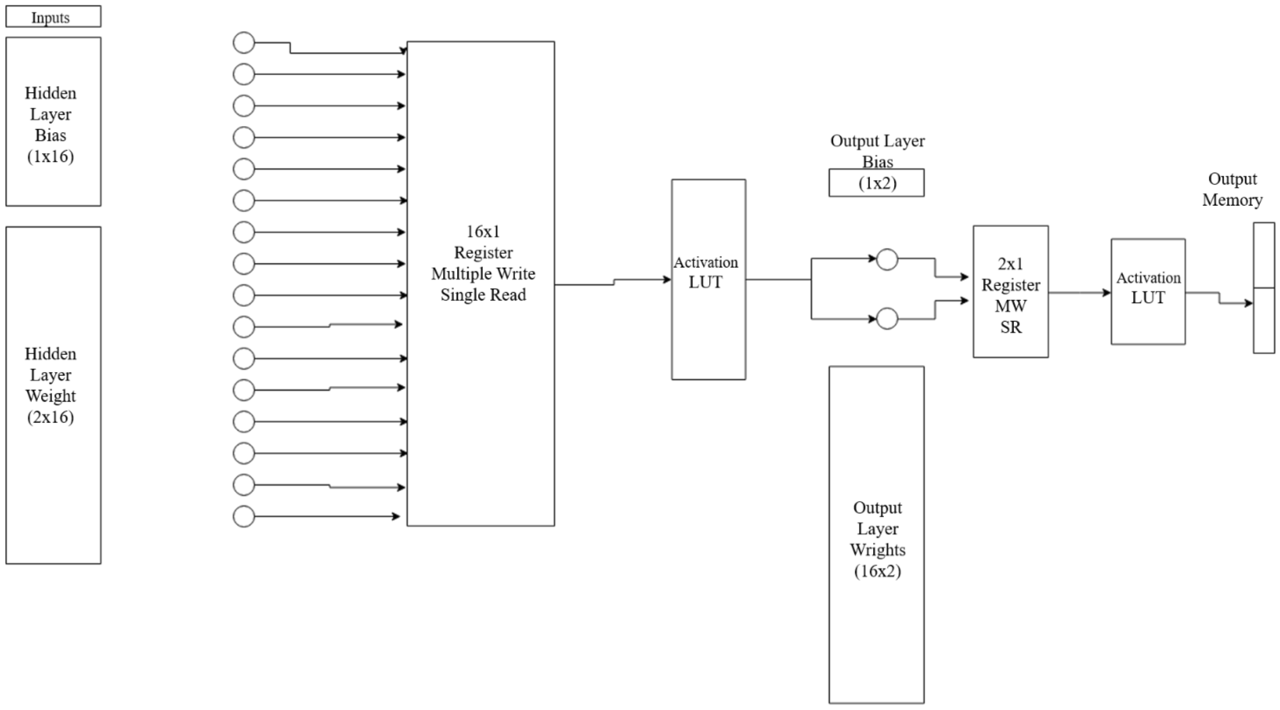

Figure 10 - Proposed Architecture I

In this proposed structure, the output of the perceptrons of the hidden layer -after 2 cycles- would be passed to a multiple-write single-read register file, which will allow all the perceptrons to write their data at once, but will output one perceptron at a time. The output of the register will be passed to the activation function lookup table, which will output the perceptrons output after activation.

Output layer perceptrons will receive their input directly from the output of the hidden layer's activation function, and will write their output to registers after exactly 16 cycles. Like the hidden layer, the output of the register file is the input to the activation function unit, which will output perceptron's output after activation. The output of the activation function is the final output of the system, and therefore stored in two registers.

However, we never applied this proposed architecture for the following: The delay caused by writing and reading from the register file at the output of each layer of perceptrons makes no sense, and leads to 16 clock cycle delay in the hidden layer, and 2 clock cycles in the output layer.

B. Proposed Architecture II

After proposing the first architecture, it was obvious that the register files at the output of the perceptrons did not make any sense, as they allow multiple writes but not reads, causing each reading to be delayed one clock cycle. Therefore, some modifications were made to improve the previous architecture, and to obtain a more flexible fast network.

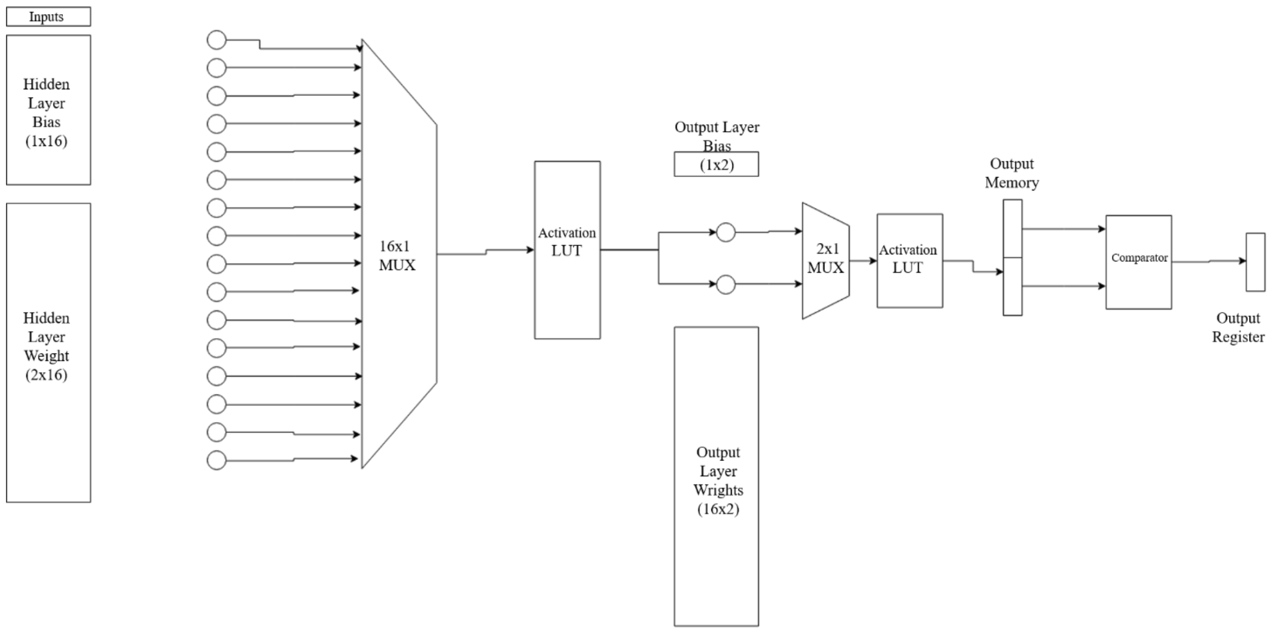

Figure 11 - Prposed Architecture II

Modifications included:

1. Replacing the register file with a multiplexer, 16x1 in the hidden layer, and 2x1 in the second.
2. Adding a comparator after the output registers to make one perceptron output 1 and the other 0, making difference much more noticeable.

C. Detailed Implementation:

The network consists of a large classifier that includes a hidden layer controller and an output layer controller, in addition to an input memory (2x32) bits.

Hidden Layer

The hidden layer controller has a hidden layer, bias memory for the hidden layer (16x32) bits, weights memory for the hidden layer (16x32) bits.

The hidden layer has exactly 16 perceptrons, a multiplexer (16x1), an activation function lookup table, in addition to a layer counter that keeps track of the selected neuron by the multiplexer. Perceptrons of the hidden layer take exactly two cycles to calculate

$$output(perceptron_n) = I_1 * w_{n1} + I_2 * w_{n2} + Bias_n$$

Implicitly, the perceptron has a MAC -Multiply and accumulate- which is a DSP -Digital Signal Processor-, in addition to some other modules. One of these modules is the DSP counter used to control the addition of the bias in the first cycle of operation, but adds the previous result (accumulate) in later cycles.

The system is parallelized across the perceptrons, but sequential across the layer. This means all layer perceptrons start together, and finish their work together, because each perceptron is implemented as an independent MAC unit. However, since the activation function takes only one input at a time, and the multiplexer is 16x1, we need to pass the perceptrons output one by one to the next layer.

Once the perceptrons of the hidden layer finish computing their output, a signal that starts the 4-bit counter is fired. This counter controls that the multiplexer chooses a perceptron and passes it to the activation function, and the activation function outputs the activated perceptron output, every clock cycle.

This neuron select counter will also be passed to the output layer to control the DSP bias addition.

The hidden layer now passes to the output layer one output per cycle for 16 consecutive cycles, in addition to the counter that selects the what perceptrons output is passed on the output layer.

Output Layer

The output layer, like the hidden layer, has its own weight memory (2x16) words too, in addition to its bias memory (2x16) words. The perceptrons of this layer fetch two weights (for each neuron) every cycle, and two biases (one for each neuron) at the first operating cycle.

The neuron select that chooses which perceptron output is passed from the hidden to the output layer chooses also the weights of that specific input (relative to each neuron). The output of each perceptron in the output layer is given by:

$$output(perceptron_n) = Bias_n + \sum_{i=0}^{15} I_i * w_{ni}$$

When perceptrons finish, they also fire a signal that chooses which perceptron output to be passed on to the activation function, and to be written in a register.

At the end of the output layer, the output of each perceptron is written to memory, and once they are both written, a comparator takes both values and fires if the second is greater than the first, which means the neuron is fired.

D. Fixed Point Manipulation

In this project, all numbers, entering or leaving the system or even stored in memory use the fixed point representation. The number of fraction bits is held at 8, which means we have a step of $\frac{1}{2^{fraction\_bits}}$ (0.00390625).

E. Sigmoid LUT

The activation function used in this project either in the hidden layer or the output layer is sigmoid. The lookup table was generated by a python program that used a step of $\frac{1}{2^{fraction\_bits}}$. The saturated range was (-4,4), and the size of LUT output was 9 bits only (Sigmoid output lies in range(0,1), no need for more than one integer bit). The LUT had a total size of

$$4 * 2^8 * 8\ bits = 1K * 8\ bits.$$

F. DSP

The DSP multiplies two number (25, 18) bits respectively, and adds them to a third number of 48 bits. Although all numbers used in the system are 32 bits with 8-bit fractions, the numbers here still follow the convention, but have abandoned their left most significant bits. Since the biases, weights and inputs ranges do not exceed an integer part of 10, therefore ignoring the most significant bits will not make a difference. However, since all biases are 32 bits only, extending them to 48 bits is not done using the conventional way. The fixed point have to be exactly 16 bits away from the right, to match the result of the DSP. Therefore, we concatenate (x00) to the right of the bias, and sign-extend the bias to reach 48 bit.

MAC Unit:

The neuron is represented as MACC (multiplier and accumulate). Each neuron takes multiple inputs each associated with different weights. The neuron multiplies those weights with the inputs and sums them together. We implemented MACC as a single multiplier and single adder due to limited resources on the FPGA. Each neuron goes through the MACC, which basically multiplies the weights with the inputs and accumulates the output of this operation. Moreover, the multiply operation could be implemented with various ways, starting with the simple multiplication design or with even more complex designs that can provide concurrency, which enhances the performance of the whole system.

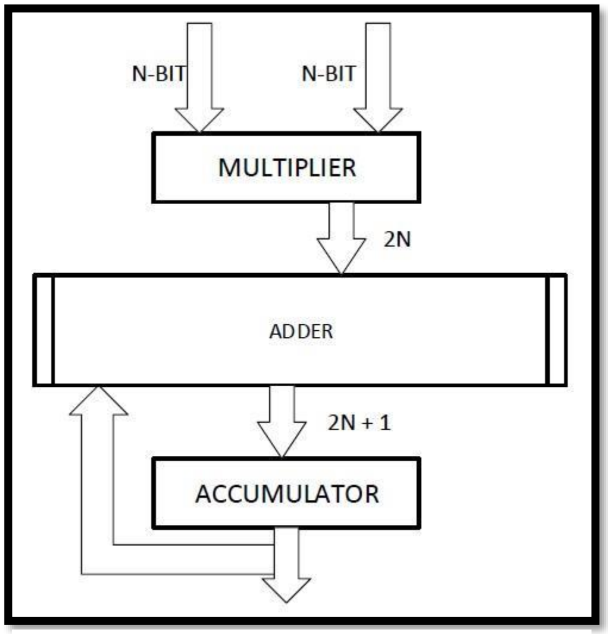

Figure 12 - MAC Unit

VII. Discussion

I. Uploading on FPGA

Proceeding with the sections mentioned earlier, of proposed architectures on software and on hardware. Thus, the need of a prototyping device that would be used for testing and validation. The used board is a ZYBO ZYNQ 7020 FPGA, which provides large resources with lower cost compared to special purpose devices for deep learning and related fields.

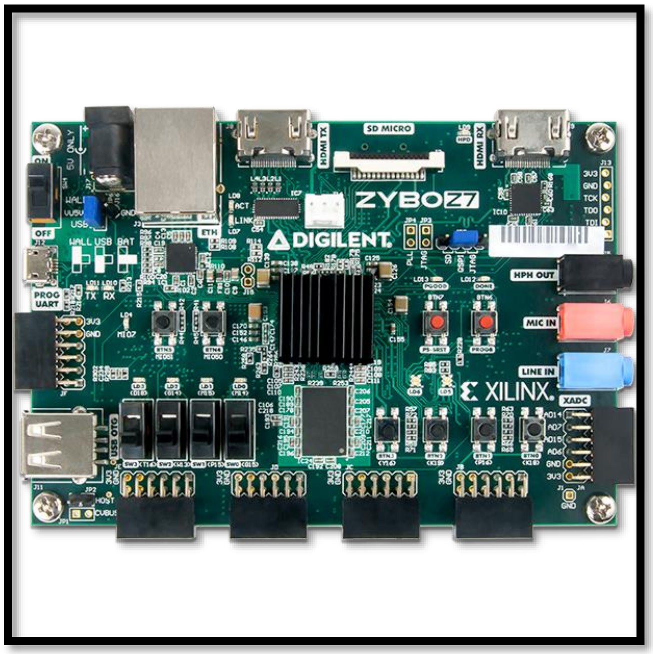

Figure 13: ZYBO Zynq 7020

The Z7020 provides more capabilities than the 7010 in mostly every resource block. The demonstration of the difference between the 2 versions is in the below table.

	Zybo Z7-10	**Zybo Z7-20**
FPGA part	*XC7Z010-1CLG400C*	*XC7Z020-1CLG400C*
1 MSPS On-chip ADC	*Yes*	*Yes*
Look-up Tables (LUTs)	*17,600*	*53,200*
Flip-flops	*35,200*	*106,400*
Block RAM	*270 KB*	*630 KB*
Clock Management Tiles	*2*	*4*
Available Shield I/O	*32*	*40*
Total Pmod Ports	*5*	*6*
Fan Connector	*No*	*Yes*
Zynq Heat Sink	*No*	*Yes*
HDMI CEC Support	*TX port only*	*TX and RX ports*
RGB LEDs	*1*	*2*

Figure 14: Comparison Table

To start the programming of the FPGA, we need to setup our .ucf file, which is the configuration file that configures the ports of the board with the system's input and output ports. The network architecture runs with a counter that synchronizes the data transportation between layers and the feedforward process. Thus, this counter was chosen to be outputted on the 4 main LEDs of the board, $LED1 \rightarrow LED4$. Moreover, 2 RGB LEDs were chosen, LED5 & LED6 for the Done signal; to indicate that the network has finished forward propagation and Result signal; that indicates that the current test has arrythmia or if he has a normal heart beat.

Furthermore, the switches of the board were utilized for more functionality. Firstly, the first switch was used for the reset; to reset the whole system and the second switch is used to choose between 2 inputs stored in the memory of the FPGA. Given the fact that the built-in clock of the FPGA is 125 MHZ, which is fast for a user to see the counter and the output of the system, we created a module to decrease the speed of the clock, to give a chance for the outputs to be seen.

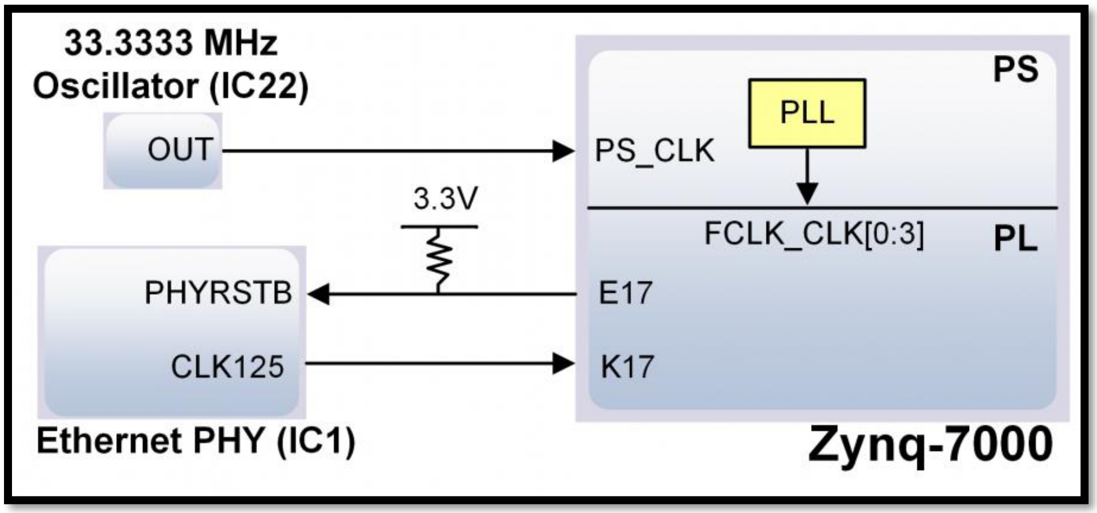

Figure 15: Clock Modules in the FPGA

II. Important Precautions

Some important precautions should be taken into consideration, when burning the program on the FPGA, firstly to make sure that are no liquids or anything that can harm the board. Secondly, to discharge any current that might be in the user’s hand, to prevent the ICs on the board from malfunctioning. Furthermore, every input and output of the program should be assigned to a standard IO port on the FPGA, with specification of voltage and further information if needed. In addition to, making sure that all the required drivers are installed, for the connection between the board and the PC during programming phase.

Lastly, the VHDL/VERLILOG program should not exceed the time constrains; this is a common warning that we faced, which is considered a warning the mapping/routing phase, although it is a warning, but it will produce an error in the “Generating Programming File” phase. Thus, the provided clock should be suitable for the program and the program itself should work efferently to meet those constrains provided by the system’s clock. If the clock is too fast, then dividing the clock is a must, especially if it is a standard dedicated clock.

VIII. Future Suggestions

Quantization

Machine learning programs tend to use floating-point precision in the backward and forward network propagation to make use of the extra precision, and although the numbers still have a limited precision this representation still is resource-expensive and why some FPGA boards have special digital signal processing units to deal with float-point precision numbers.

However, to generalize machine learning applications on FPGA boards; even those who do not have special DSPs can be done using quantization, which is a non-linear mapping function that we implement to reduce resource requirement to store and operate on neuron weights. After the operation is done the values are restored back to a close value of the original value in a process known as dequantization.

Quantization is traditionally done in the form of mapping 32 bits to 8 unsigned bits, where the minimum and maximum values are mapped to 0 and 255 respectively to squeeze out the most accuracy out of the quantization process. Although this mapping operation seems very inefficient due to the accuracy loss that will occur during the quantization and dequantization the optimization itself still works since neural networks are very immune to noise which in our case could occur from input as well as accuracy loss from the quantization process.

This immunity generally comes from the objective of creating neural networks that try to not over-fit, which in case of noise would not be able to generalize or recognize a certain object, and in case of under-fitting would create false positive results thus leaving us with a neural network that can over-look noise to certain extent and can generalize over detected/generated object. This property of neural networks still does not give us the right to perform quantization without testing our network for an acceptable accuracy drop off.

A more formal description of the suggested affine quantization method is as follows:

$$y = (x - a)\frac{d - c}{b - a} + c \qquad (6)$$

Where y is the quantized number

X are numbers from range [a, b] (a is the minimum float-point precision number we obtain, and b is the maximum)

Quantization from range [c, d] (in our case the numbers will be 0 ,255). As for dequantization we plug in the output obtained back in the equation with ranges [a, b] and [c, d] exchanged to obtain the rounded value after being operated on.

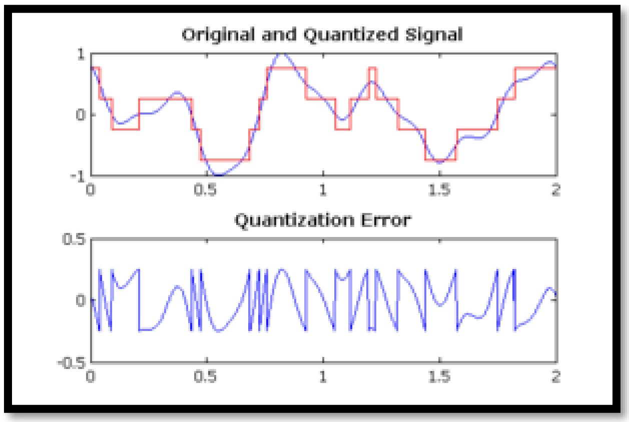

Figure 16: Quantization

IX. Conclusion

To summarize, an artificial neural network of three layers was implemented, and after training, the weights and biases were converted to a fixed-point binary representation. They were stored on memories to be used in the VHDL implementation of the network. The VHDL code was uploaded on ZYBO ZYNQ 7020 FPGA, and the output was as expected.