FPGA Implementation of Digital Pre-Distortion Using a Neural Network

Matthew Morrow

Degree Stream : Master of Engineering

Supervisor : Dr. John Dooley

Ollscoil Mhá Nuad

Ollscoil na hÉireann Má Nuad

Submitted to the Department of Electronic Engineering
in Partial Fulfilment of the Requirements for the
Degree of Master of Engineering in Electronic Engineering at
Maynooth University

Department of Electronic Engineering
Maynooth University
Ireland

May 2023

Contents

Abstract

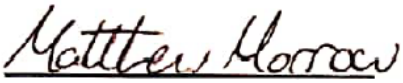
As the demand for greater efficiency in communication devices increases more solutions need to be found for achieving greater processing speeds and reducing power consumption. The field of machine learning and artificial intelligence is currently a major topic of interest and is currently driving the development of many technologies beyond their previous limits. In this report, pre-distortion methods through the means of neural networks will be examined and a hardware model for a pre-trained digital pre-distorter will be created. A brief overview is provided for machine learning and power amplifiers. The structure of neural networks is explored in order to inform the design. The Saleh model and it's corresponding pre-distortion model are presented. The process of creating a Time Delay Neural Network in Keras is documented. Various activation functions, network structures and data sets are tested during training. Finally a hardware model for a Time Delay Neural Network is created and Synthesised. The design is ultimately proven to be functional in simulation and potential for future work in this area is presented.

Decleration

I herby certify that this report and the project it describes are my original work and have not been taken from the work of others save and to the extent that such work has been cited and acknowledged in the report.

List of contributions from others:

Artix 7 FPGA provided by John Dooley

Signed: 

Student No: 18427114

Date: 05/05/2023

1 Introduction

As the world becomes more connected and more wireless devices enter our daily lives and grow in complexity the demand for better performance increases. In the wireless communication industry there are is an ongoing effort to increase the efficiency of these devices in terms of both time and energy. One such element of these devices are power amplifiers (PA).

These devices are typically non-linear in nature. For this reason, a PA's maximum output must be reduced by its peak to average power ratio (PAPR). This is not ideal considering that the maximum efficiency of a PA is achieved at its maximum power. For this reason, it is desirable to create a linear relationship between the input and output of a PA.

The linearisation of PA's has been achieved in multiple fashions such as feedforward, feedback and pre-distortion . Each of these methods has their strengths and weakness. Feedback systems are simple but lack stability. Feedforward linearisation techniques allow for a wide bandwidth coverage, but are not very efficient. Pre-distortion comes in two forms, analogue and digital. Analogue is easily implemented; however, its ability to linearise is lacklustre. Digital is the current standard for linearisation of PA's and achieves high efficiency and flexibility; however, at high frequencies, energy costs are high [1, 2] and for this reason, more efficient methods are currently under research.

Various methods of digital pre-distortion exist. The most basic method is to simply predistort the signal based on a previously attained model. Gupta et al. outlines a method of performing simple pre-distortion on a memory-less power amplifier [3]. Other digital predistortion methods are dynamic and can be altered based on feedback. Pham et al. suggests a method of dynamic feedback learning that uses a proposed dynamic orthonormal transformation matrix [4].

Recently, more research has investigated the potential use of neural networks for predistortion. Jiang et al. proposed the use of block-orientated time-delay neural networks for predistortion. They also propose a method for low-frequency sample rate feedback of power amplifiers to be implemented into the neural network [5].

In this report, neural network operation will be outlined along with various forms in which components of a neural network come. Various mathematical models for power amplifier will be discussed. This will ultimately guide a decision for the type of neural network to be used for pre-distortion on an FPGA.

2 Power Amplifier Modelling

In order to train a neural network to perform the operation of a predistorter, we need to provide it with training data. This training data can be generated using a mathematical model for a power amplifier. Two common types of power amplifier models are the Saleh model and the Volterra series. For nonlinear representation, the Saleh model uses AM/AM and AM/PM gain factors. Using four input parameters, the Saleh model can approximate a

Power Amplifiers amplitude and phase modulation error due to nonlinearity.

Volterra series models are said to have higher accuracy of representation for power amplifiers at the cost of higher computation requirements. The volterra series model is similar to the taylor series but is capable of capturing the memory effects of a power amplifier. Due to its convolutionsal structures, the volterra series is good at representing intermodulation and harmonic distortions [6].

Alternatively, a memory polynomial model could be used to replicate the behaviour of a power amplifier. Huadong et al. outlines a method of designing an envelope memory polynomial in order to model a 2W LDMOS power amplifier. This method efficiently captures both the non-linearity of the power amplifier and it's memory effect [7].

For this project, the Saleh model will be used due to its simplicity. Of the three models outlined, the Saleh model can most easily be modeled for predistortion and used to generate training data.

2.1 Saleh Model

The Saleh model, and its corresponding predistortion model (subsection 2.2), used were based on the findings presented by Jang et al.[8]. The authors described their method of predistortion of a Saleh model PA. As mentioned previously the AM/AM and AM/PM gain factors replicate the distorting effects of a power amplifier. These are shown below in Equation 1 and Equation 2.

$$(AMAM) \quad G[v(t)] = \frac{\alpha_a v(t)}{1 + \beta_a v(t)^2} \tag{1}$$

$$(AMPM) \quad \Phi[v(t)] = \frac{\pi}{3} \frac{\alpha_p v(t)^2}{1 + \beta_p v(t)^2} \tag{2}$$

In these equations, v(t) represents the absolute value of the PA input signal. α and β represent the parameters describing the distortion of the PA. The AM/AM and AM/PM values can then be used to reform a complex signal using Equation 3.

$$z(t) = G[v(t)]e^{j(\phi(t)+\Phi[v(t)])} \tag{3}$$

Here $\phi(t)$ represents the the angle of the complex input signal $v(t)$. The effect of the Saleh model on a 256-QAM signal (Figure 1) is shown below in Figure 2.

It is important to note that no gain value was included in the Saleh model as we are only interested in modelling the distortion effect of the system.

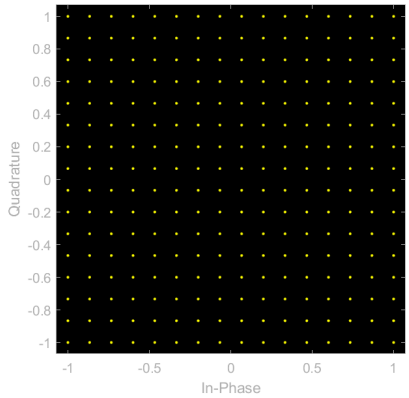


Figure 1: 256-QAM signal

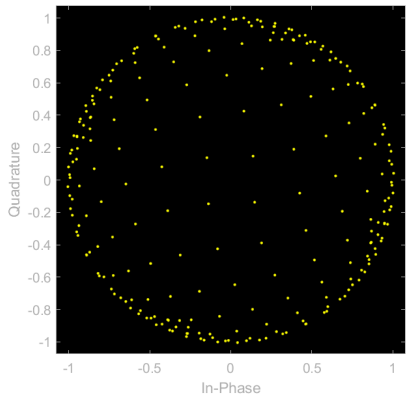


Figure 2: 256-QAM signal Saleh model output

2.2 Saleh pre-distortion

In order to linearise the output from the Saleh Model we will need to pre-distort the input signal. Where the magnitude of our output signal is $\rho(t)$ and the angle is $\psi(t)$, we can determine that Equation 4 and Equation 5 must be satisfied to achieve linearity.

$$\rho(t) = G[v(t)] \tag{4}$$

$$\psi(t) = \theta(t) + \Phi(t) \tag{5}$$

If $v(t)$ and $\theta(t)$ are the inputs of the PA, we can instantiate that they would be the outputs of the pre-distorter, thus the equations for pre-distortion are shown in Equation 6 and Equation 7.

$$v(t) = G^{-1}[u(t)] \tag{6}$$

$$\theta(t) = \phi(t) - \Phi(t) \tag{7}$$

We now must evaluate $G^{-1}[u(t)]$. This is shown in Equation 8. However, before predistortion we must limit the magnitude $u(t)$ to $\sqrt{\frac{\alpha_a^2}{4\beta_a}}$. This prevents the output magnitude of the power amplifier from clipping. Clipping only occurs when $u(t)$ exceeds this value. Clipping is visible by the circular shape formed by the signal in Figure 2.

$$G^{-1}[u(t)] = \frac{\alpha_a \sqrt{\alpha_a^2 - 4\beta_a u(t)^2}}{2\beta_a u(t)} \tag{8}$$

The resulting signal can be formed using Equation 3. The predistorted 256-QAM signal is shown in Figure 3 and the output of the Saleh model after pre-distortion is shown in Figure 4.

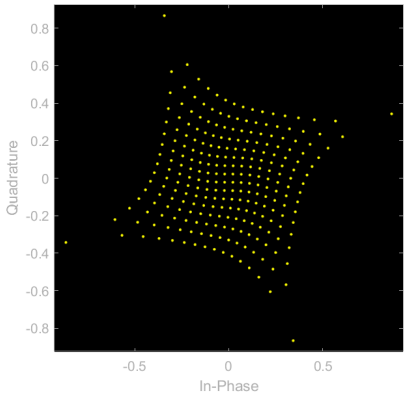


Figure 3: Predistorted 256-QAM signal

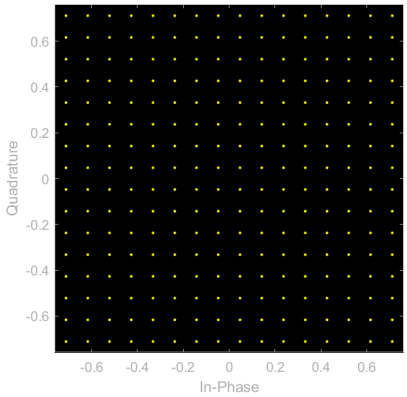


Figure 4: Saleh Model output from a Predistorted 256-QAM input

We can see that our Saleh model output in Figure 4 has maintained shape and matches the shape of the input shown in Figure 1. Now, with a pre-distortion model, we can examine how this can be implemented with a Neural Network.

3 Machine Learning

Machine learning is the name given to the process of devices automatically improving with experience. While unnamed at it's point of inception, the first known attempt at machine

learning was conceived by Donald Webb in the 1940s. Webb attempted to quantify the function of neurons in the brain and specifically how neurons contributed to learning. This was the foundation for what we call neural networks today [9].

The term Machine Learning first came to prominence in the mid-20^{th} century when Arthur Samuel, an IBM engineer, developed a software programme to play checkers [10]. The programme used a scoring function to determine moves and included a memory to remember these moves. Samuel later changed the programme so that the function for determining moves was adaptive so that the machine itself could determine better ways of playing rather than storing moves. This method of learning is called generalisation learning.

In the latter half of the 20^{th} century the field of machine learning continued to evolve with IBM developing an algorithm known as Perceptron. This was eventually implemented as the Perceptron Mark 1. This machine was created for image recognition and early versions could discriminate between shapes and characters [11].

The limitations of the Perceptron were soon realised in that classification could only be made through linear separability [12]. Through the implementation of feed-forward multi-layer Perceptacons this could be improved and non-linear data could be separated. This was the first implementation of an artificial neural network (ANN) [13].

4 Artificial Neural Networks

Neural networks are defined as computing systems that are inspired by animal neurological systems. While machine learning was first conceived in the late 1940s, the work towards creating a neural network began in 1943. McCulloch and Pitts began to develop a mathematical model of neurological systems and created a model for a neuron [14]. They suggested that a neuron would 'fire' when a combination of the inputs reached a threshold. This model was later incorporated into artificial neural networks which also implemented learning.

Unlike the uni-layer perceptron that came before, artificial neural networks contain a hidden layer, or multiple hidden layers, between the input and output of a system. Like a brain, a neural network consists of a chain of neurons that connect to each other. Each neuron in a neural network has trainable coefficients that allow it to improve it's estimations.

4.1 Artificial Neuron Model

An artificial neuron is made up of several parameters. First, we have the inputs, in which a neuron can have infinite amounts. For the input layer, the inputs will be the actual inputs to the system, for all other layers these inputs will come from the previous layer. The input(s) to a neuron are typically denoted by x.

The next component is the weights of the neuron. For each input to the neuron there exists one weight. This is the first of the adjustable variables within the neuron. As the network learns, it adjusts the values of these weights. The weight of a given input can be

viewed as the importance of that input to that neuron. Weights are identified as w.

The third component of a neuron is the bias, another trainable parameter, denoted as B. The bias, unlike weights, is not dependent on the inputs to the neuron. The bias acts as an offset in the neuron. Even if all the inputs to a neuron are zero, the bias can still feed a value to the output. These can all be combined into an equation as shown in Equation 9. x and w are arrays in the case of a multi-input neuron.

$$y = x * w + B \tag{9}$$

The final factor within an artificial neuron is the activation function. These activation functions are typically non-linear in nature. The introduction of non-linearity allows for the artificial neural network to learn complex patterns within data. Without the activation function, the network would only be able to identify linear relationships. The sum of weighted inputs and the bias is passed into the activation function, the result being the output of the artificial neuron.

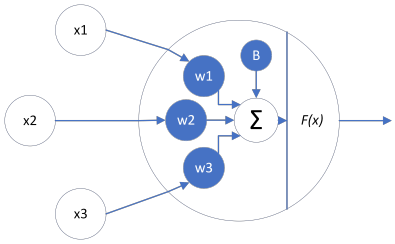


Figure 5: Neuron Diagram

4.2 Types of Activation Functions

There are many activation functions to choose from when designing a neural network. The most common are ReLU, hyperbolic tan and Sigmoid. The goal of these functions is to introduce some form of non-lienarity into the system.

The ReLU (Rectified Linear Unit) function is one that transforms any value less than or equal to zero into zero. Anything greater than zero remains unchanged and is passed directly through (Equation 10).

$$ReLU(x) = \begin{cases} x \ for \ x > 0 \\ 0 \ for \ x \leq 0 \end{cases} \tag{10}$$

Another common activation function is the hyperbolic tan. This function is centred at 0. It maps values into a range between -1 and 1, as shown in Figure 6 and Equation 11.

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{11}$$

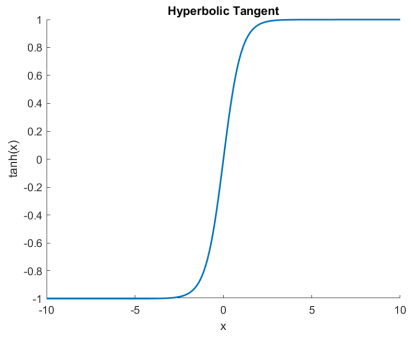


Figure 6: Hyperbolic Tangent

The Sigmoid function is similar to the hyperbolic tangent; however, it's output ranges from 0 to 1. This is shown in Figure 7 and Equation 12.

$$Sig(x) = \frac{1}{1 + e^{-x}} \tag{12}$$

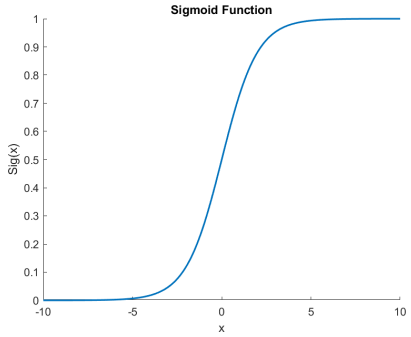


Figure 7: Sigmoid Function

4.3 Types of Neural Networks

Neural networks can be configured in a multitude of ways. The configuration of a neural network refers to the structure of neurons, input processing and interconnections between layers. Some common structures are Convolutional Neural Networks (CNN's), Recurrent Neural Networks (RNN's), and Time Delay Neural Networks (TDNN's).

A CNN is most commonly used for image processing. Unlike typical neural networks, the layers of a CNN are not fully connected, meaning that not every neuron at the input layer is connected to every neuron in the next layer. This allows for neuron's to be dependent only on specific inputs rather than the system as a whole. In terms of image processing this means processing the image in regions. CNN's also include pooling and fully connected layers. The difference between a fully connected neural network and a CNN is shown in Figure 8 [15].

An RNN is unique in that it can take a variable number of inputs and return a variable number of outputs. This is achieved through output feedback, allowing a large stream of

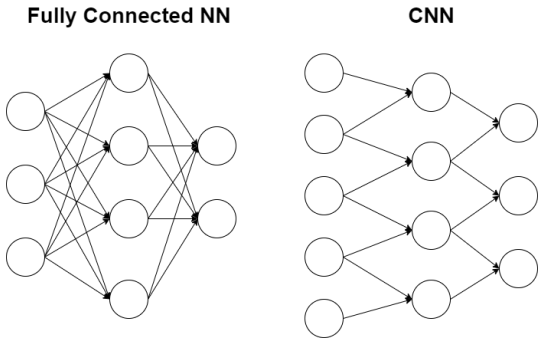


Figure 8: Fully Connected NN vs CNN

input data to be fed to the network. This is especially useful for applications such as language translation, speech recognition, and stock market predictions [16]. In an RNN the output of a given neuron could be stored and used as an input to the neuron again on the next input cycle.

TDNN's operate on the basis of taking a multitude of previous and future inputs. At the input layer a TDNN will consider a large time scale of the input, through subsequent layers this is filtered into a shorter timescale. Similar to CNN's, at the input layer, the neuron's will not be fully connected but instead focus on specific regions. This is where the term time delay comes from, in that a region will be a combination of samples at various times. In this context, TDNN's can be considered to be a one dimensional equivalent to CNN's. A variation of a TDNN that is densely connected at the input is known as a D-TDNN [17]. This means that each input and delayed input is connected to each neuron. TDNN's are widely used today in speech recognition to identify phones [18].

TDNN will be used as the primrary structure for the implementation in this report. Specifically, a D-TDNN will be used but will be referred to as a TDNN.

4.4 Training Methods

Neural Networks are typically trained through back-propagation. When the network is being trained, the output is compared with the desired output. This result allows for the calculation of error values. Using these errors, weights and biases within neurons can be adjusted. This adjustment is typically made based on an optimisation algorithm, for which many exist. For the purpose of this system two will be considered - Levenberg-Marquardt (LM) and Adam.

4.4.1 Levenberg-Marquardt

The LM optimisation algorithm was first conceived by Levenberg in 1944 and later built on by Marquardt in 1963 [19, 20]. First a loss function is found through the sum of squared errors (Equation 13).

$$e = y_{actual} - y_{predicted}$$

$$f = \sum e^2 \tag{13}$$

Next, the Jacobian matrix for the loss function is found for errors with respect to the network weights w (Equation 14). This Jacobian function provides the direction of minimisation.

$$J \;=\; \frac{\partial e}{\partial w} \tag{14}$$

With the Jacobian matrix, the Hessian can be estimated (Equation 15). The Hessian matrix provides the step size for minimisation.

$$Hf \approx 2\left(J^T\right) + (\lambda I) \tag{15}$$

The unique addition made in LM is the damping factor λ. Without λ this function would be equivalent to the Newton method; however, with a large λ it resembles gradient descent. For this reason, λ is initially set to a large value, but with each successful iteration λ is reduced. If an iteration is unsuccessful the value of λ is increased.

4.4.2 Adam

The Adam (adaptive moment estimation) optimisation algorithm is very common in modern neural networks. In 2015 Kingma and Ba proposed this new method of stochastic gradient descent, which would be much more computationally efficient than previous implementations [21]. Adam only requires first-order gradients, which means that it is simple and requires little memory.

The Adam algorithm begins with the computation of $\mathrm{m_t}$ and $\mathrm{v_t}$ which are the decaying averages of the past loss functions and the past squared loss functions. These are the same loss functions as shown in Equation 13. This is shown in Equation 16 and Equation 17 where β_1 and β_2 values are forgetting factors which are typically set to 0.9 and 0.999 respectively.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\,\nabla f \tag{16}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla f)^2 \tag{17}$$

As the values of m_t and v_t are initialised as 0 the authors noted that subsequent calculations are biased towards 0. For this reason additional bias correction equations are used, as shown in Equation 18 and Equation 19.

$$\hat{m}_t = \frac{m_t}{1 - \beta_1} f \tag{18}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2} \tag{19}$$

Finally, the weights can be adjusted using Equation 20. η represents the learning rate of the system and ϵ represents 10^{-8}, used to prevent division by 0.

$$w_{t+1} = w_t - \frac{\eta \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{20}$$

4.4.3 Loss Functions

For LM optimisation, the sum of squared errors (Equation 13), sometimes referred to as L2 loss, is typically used as the loss function. In fact, in some fields, such as image processing, L2 is the default loss function [22].

L1 loss is similar, however the error is not squared and instead the absolute value of the error is found. A 2017 paper explored several different loss functions and found that, depending on the application, different loss functions were preferable. Janocha and Czarnecki examined loss functions such as L1, L2, hinge, log and tan. In this paper, it is stated that many applications simply use log loss. They outline the benefits of L1 and L2 despite their simplicity. When tested on the MNIST dataset L2 and hinge were strong performers in terms of both speed and accuracy. Interestingly, these loss functions did not perform as well when presented with noisy data when compared to other loss functions such as L1 [23].

4.4.4 Training Data

Before implementing a neural network, training data must be gathered. The data needed to train a neural network are training data and test data. Both consist of a set of inputs and a set of outputs. When using the training data, the neural network is given the input data and the output is compared to the output training data. Using the loss function and the training algorithm, the weights and biases are modified. Once we have performed the desired number of epochs the test data is passed through the network and we can evaluate the accuracy of the network.

An Epoch refers to one cycle of training, on the entire training data-set. Multiple epochs can be performed in order to achieve greater accuracy. When carrying out multiple epochs, we run the risk of overtraining the network. Overtraining results in the network attempting to fit outputs too closely to the training data set. While the network may provide outputs accurate to the training data-set, for other inputs, the outputs may be incorrect.

Training data is typically split into batches. The batch size is the number of samples that the network is trained on at once. For a batch size of 10 on a dataset with 100 samples, each epoch will train on 10 batches of data. Batches are used so that the model can be trained on the entire dataset without having to use a large amount of memory. Batching can also help prevent overfitting of data and improve it's ability to generalise [24].

Another configurable parameter when training a neural network is the step size, also known as the learning rate. The step size is a hyperparameter that controls the rate at which the weights are updated. The step size must be carefully selected because if it is too small it will take a long time for the model to be trained. On the contrary if the step size is too large the system may fail to converge on a minima.

4.5 Hardware Implementation of Neural Networks

Typically Neural networks are implemented using software on a general-purpose processor; however, when there is a demand for higher processing speeds, hardware implementations are much more sensible. Misra and Saha outlined the main benefits of a hardware neural network (HNN) being speed, cost and graceful degradation [25].

Many new devices, even at the consumer level, have begun implementing specific hardware to increase the speed of neural networks and AI processing. Google has developed a new platform named TensorFlow aimed at improving the speed at which machine learning and, specifically, neural network tasks can be carried out. Alongside this new platform, there is a hardware system called Tensor Processing Units (TPUs) [26]. These processors include hardware dedicated to carrying out neural network related tasks. In 2015 Jouppi et al. carried out a test on this platform in a data centre to evaluate the benefit of such a system. It was found that not only performance was greatly improved but also energy efficiency [27].

Another approach to implementing a neural network in hardware is to pre-train the network in software and to take the network structure and implement that on hardware. This removes the need for back-propagation ability therefore reducing the amount of logic required. This is the approach that will be taken for the system outlined in this report.

5 Network Pre-training

In order to train the neural network parameters, Keras was used. Keras is a python based, neural network API developed by Google that utilises the Tensorflow library. Keras was chosen due to its ease of use and simplification of tasks. Using Keras the weights and biases of the system could be obtained to eventually put onto the hardware as pretrained values. Training values on a more powerful device will allow the final hardware implementation to be much simpler, allowing for lower power and logic usage.

All Python scripts related to this section can be found in the python notebook attached in subsection 10.3

5.1 Time delay function

While Keras can implement various types of network structure, it has no support for TDNN's. In order to implement a TDNN with Keras a python function was designed. This function took the input data set and produced several inputs for the network as time-delayed versions.

Since each input row N has both the real and imaginary value inside, we can simply expand every row to include the delayed values. To make the function scalable, the number of desired delays would be specified in the function call.

For a given sample in the data set at time n and 2 delays being specified in the function, the sample would be transformed from [real(n), imag(n)] to [real(n), imag(n), real(n-1), imag(n-1), real(n-2), imag(n-2)].

5.2 Keras Network structure

With the delay function generating the delayed inputs, the actual keras neural network begins with a dense input layer. This input layer has 10 neurons and is densely connected to the output layer which features 2 outputs and a linear activation function. To ensure that the weights and biases could be easily implemented in hardware they were constrained to a range of -1 to +1. This would allow for quantisation to an already constrained range of the values when being translated and implemented into hardware.

The network was compiled with the ADAM optimisation algorithm and the mean squared error loss function. It was also compiled to include the mean absolute error and accuracy of the output to monitor training performance.

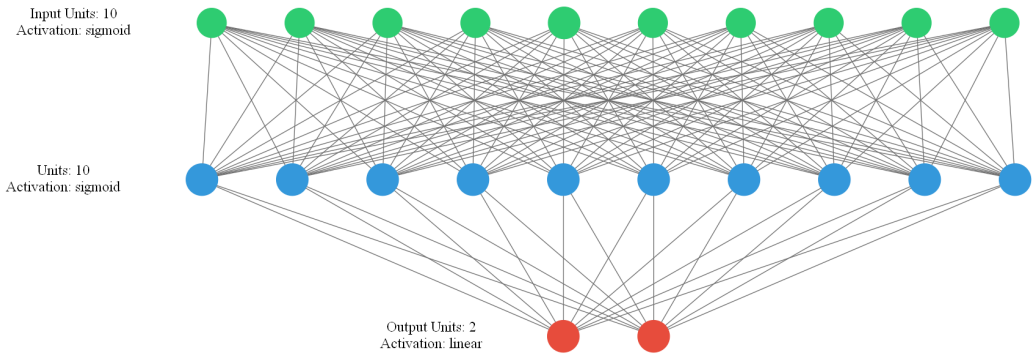


Figure 9: Visual Representation of the Keras neural network, generated using the Keras visualiser library

5.3 Training Data

The data on which the neural network would be trained was for the Saleh predistortion model, as described in subsection 2.2. This data was generated using Matlab and was then imported into Keras. The numpy library was used to process the imported data.

5.3.1 Generating Training Data

Using Matlab, code was created that was capable of performing Saleh distortion and pre-distortion. This script generated 256-QAM signals and performed saleh predistortion on

them. The training data-set consisted of every possible state transition for a 256 QAM signal. The test data-set imported an image, transformed the image to greyscale and encoded it in 256-QAM. Each of the data sets were pre-distorted, and the resulting signal would be used as the target output for the neural network. The data was saved in CSV files for tranferring to Keras. The Matlab script used to generate this data is provided in subsection 10.1.

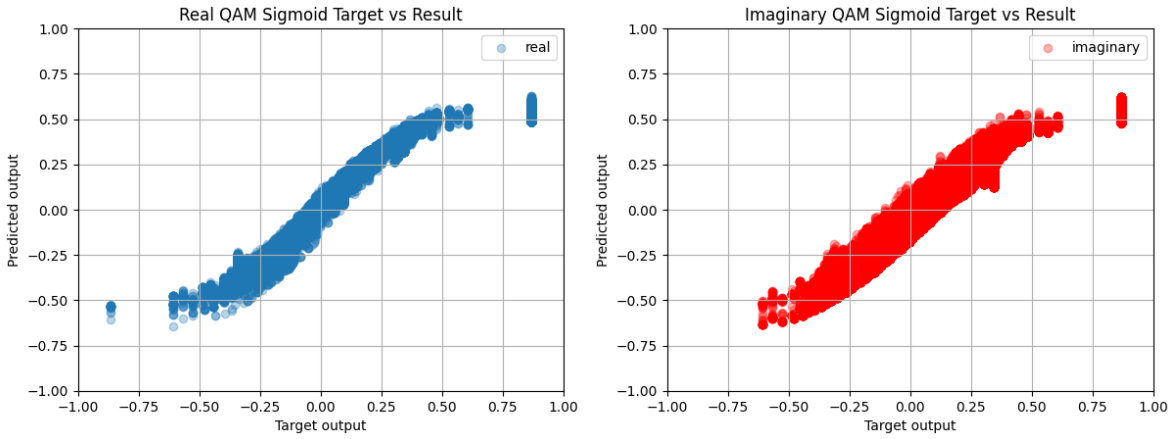
5.3.2 Importing Data

First, the data was imported from the CSV files, of which there were eight, divided into 3 catagories: real / imaginary, input / output, and training / test. Once imported to Python, the real and imaginary parts were combined and then transposed so that every sample would be a complex number with the real part at index 0 and the imaginary part at index 1. Finally, the input data was passed into the delay function outlined in subsection 5.1. The resulting data would be used as input to the neural network.

5.3.3 Training & Testing

The network was trained on the data-set over 10 epochs, with batch size of 32, a 10% validation split and shuffle enabled. The validation split takes part of the data for validating the accuarcy of the model during training. Shuffle mixes the order in which the data is trained.

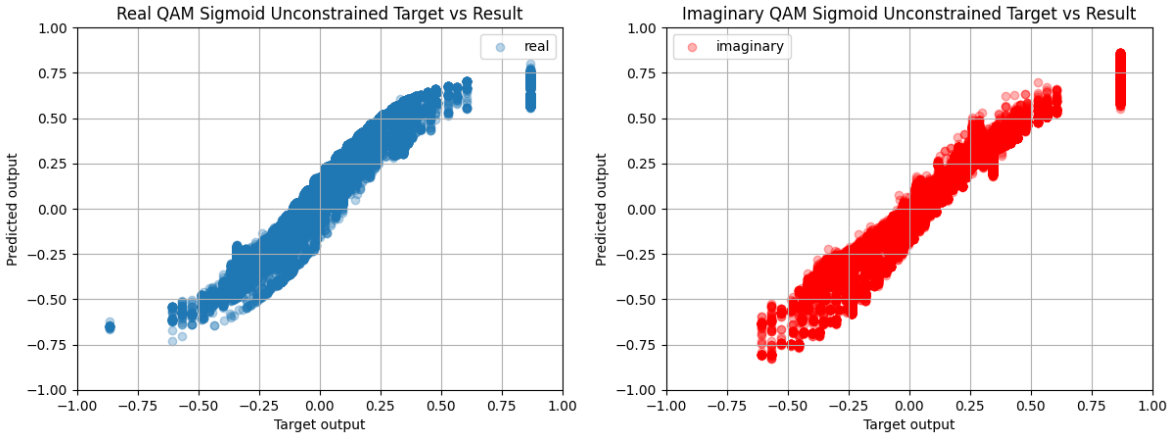
During training, using the sigmoid activation function on the first layer, a peak accuracy of 98.01% was achieved with a mean absolute error of 0.0244 on the validation data. The test evaluation yielded an accuracy of 99.85% and an absolute error of 0.1345. These values seem strange as a high accuracy is achieved but a large absolute error. On inspection of the relationship between expected outputs vs actual outputs, shown in Figure 10, we can see that there is a high correlation between target values and resulting values below 0.5, however above 0.5 we see that the output saturates around 0.6.



(a) Real (b) Imaginary

Figure 10: QAM Sigmoid NN output Target vs Result

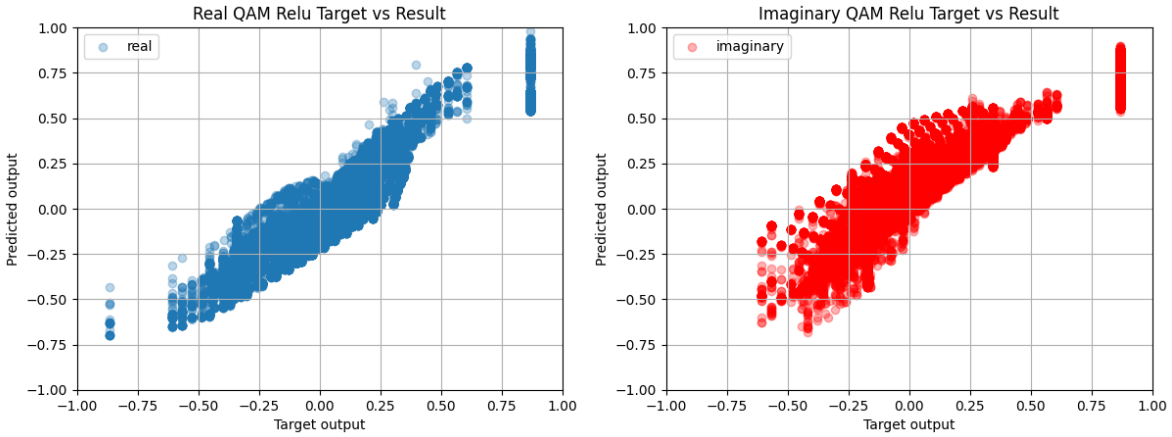
The first hypothesis for this saturation was the constraints set on the weights and biases. These constraints were removed and the network was retrained with identical training parameters. Upon retraining an accuracy of 99.81% is achieved on the test data with a mean absolute error of 0.0998. The accuracy is slightly lower, however a significant improvement is made in the mean absolute error. It is visible in Figure 11 that an improvement was made when these constraints were removed. This is a tradeoff that would need to be evaluated as there is an additional hardware cost in removing these constraints and allowing the weights and biases the be outside of the range of -1 to 1.



(a) Real (b) Imaginary

Figure 11: QAM Sigmoid NN output Target vs Result with Unconstrained Weights and Biases

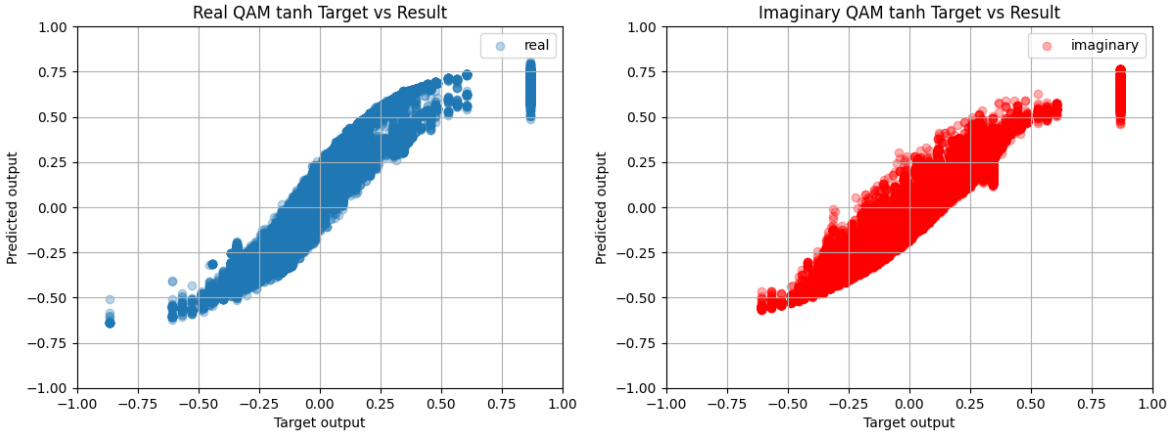
Next, the system was trained with constraints and the Relu activation function. This achieved an accuracy of 99.31% and an absolute error of 0.0938. These metrics seem promising, however on inspection of the expected outputs vs actual outputs shown in Figure 12 there is a large variance in outputs for a given input sample.



(a) Real (b) Imaginary

Figure 12: QAM Relu NN output Target vs Result

The same training process was also carried out for the tanh activation function. This resulted in an accuracy of close to 100% but an absolute mean error of 0.1648. In Figure 13 it is evident that there is a much larger variation than sigmoid however a slightly better linearity.



(a) Real (b) Imaginary

Figure 13: QAM tanh NN output Target vs Result

From the data presented, in termns of activation functions sigmoid seems to be the best option as it is linear for low input values and has a low variance.

The final test was to see if a different training data set would improve the linearity of the system when using a sigmoid function. For a QAM signal the power levels are not evenly distributed and edge values have a much larger magnitude than others. In an effort to correct for this a new data set was generated that would encode using APSK instead of QAM.

The APSK signal had 256 possible quantisation points. The APSK encoding was split into 8 radii with 32 points per radii, this is shown in Figure 14.

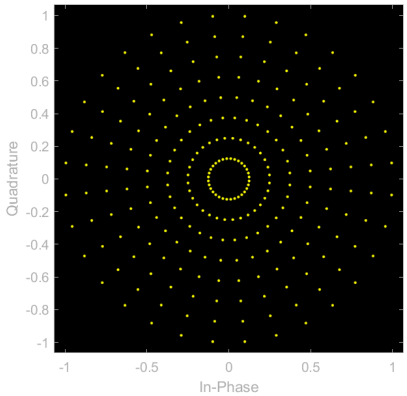
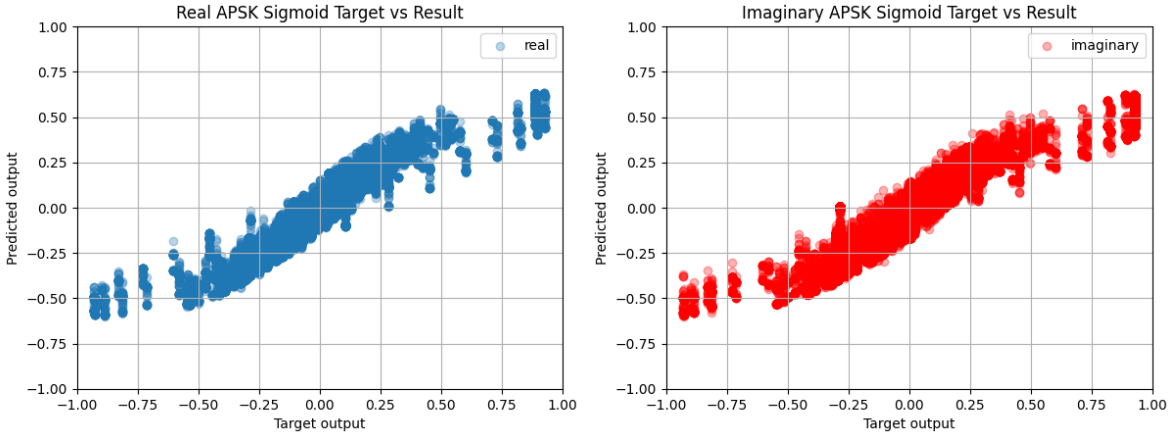


Figure 14: 256-APSK constellation diagram

This new data-set was used to train the sigmoid based network. An accuracy of 98.8% was achieved with an absolute mean error of 0.055. From Figure 15 it can be determined that no signifcant improvement was made in training. While the power distribution is definitely

more even it seems that the saturation issue is related to the network structure rather than the training data.



(a) Real (b) Imaginary

Figure 15: QAM tanh NN output Target vs Result

From these results it was decided that the QAM data set with the sigmoid function would be used as the model on which the hardware instantiation would be based.

5.4 Weight & Bias Extraction

In order to transfer weight and bias values from Keras to System Verilog the values had to be stored in .mem files so that they could be read into System Verilog with the ***$readmemh()*** command.

Conveiniently, Keras has a built-in function ***get_weights()*** that can extract weights and biases from a trained neural network model. Before any code was created to write these values to files, a function needed to be designed to convert floating point values into 16 bit hexadecimal fixed point. To convert a value to a 16 bit signed fixed point value, Equation 21 can be used.

$$f(x) = \begin{cases} 2^{15-1}x + 1 & \text{if } x > 0 \\ 2^{15-1}x + 2^{16} - 1 & \text{if } x < 0 \\ 0 & x = 0 \end{cases} \tag{21}$$

Next, a Python script was written to write the values of all the weights and biases to .mem files. The weights/bias matrices were transposed and then written to the file one by one. The weights were put in order of neuron and then in order of input (real no delay first, imaginary longest delay last). The biases were ordered by neuron. This was done for both the first and second layers and the files were saved as weights1.mem and weights2.mem, respectively.

6 Hardware Implementation

The target hardware chosen for the implementation of the design was the Digilent Nexys 4. This device features several hardware interfaces such as switches, displays and a 16MB Cellular RAM. The heart of the board is the Artix 7 FPGA designed by Xilinx. This features over 15,000 logic slices, a 450MHz clock and 4.8Kb Block Ram .

The design described in previous sections was created in system verilog. Using the Vivado design suite the design was incrementally tested, sythesised and implemented. Any fully routed design could be converted into a bitstream and programmed to the Artix 7 via the usb interface on the Nexys 4 board.

6.1 Data & Signals

When implementing a digital system, it is important to decide on data sizes. We know that for the system we are implementing, the input and output values will be between -1 and +1. With this we know that we will need to choose a data format capable of representing both signed values and fractions.

In System Verilog, the signed operator can be used to specify that a register or wire is transmitting a signed value. If an operand is signed the compiler will automatically assign signed arithmetic operations to that operand. If one operand is signed and another is not, the non-signed operand will be converted, and the operation will be carried out on the values.

For fractional values in binary, either floating point or fixed point notation could be used. Floating point, as the name suggests, has a moving binary point. This allows for the fractional size to scale based on the number of signifcant bits needed to represent values greater than 1. Fixed point values on the other hand have a set fractional bit size. Considering our input and output will be limited to a maximum absolute of 1, fixed point notation should be appropriate.

The bit size chosen for the system was 16. With signed fixed point values ranging from -1 to +1 we can achieve a minimum resolution of 0.000030517578125, which should be sufficient for this system.

6.2 Neuron

The first module implemented was a neuron. When designing the neuron it was important to ensure that values were scaleable. It was necessary to have parameters to control the size of inputs (number of bits) and the amount of inputs. With a scaleable input size, the number of weights the neuron has would also need to be scaled to match the number of inputs.

It was decided that weights and biases would be set as inputs into each neuron, allowing them to be initialised from memory outside of the neuron block. An enable input would be used to control when the weights and biases would be updated. In theory this would allow for memory values to be updated while the network is still operating and values would only be updated once the enable is set high.

The implemented neuron would operate off of a negative edge clock input. On every clock activation the following steps would need to occur within the neuron:

- Update weights/biases if enable is high.
- Multiply all weights and inputs.
- Sum all the products of the weights and biases.
- Pass the sum through an activation function.
- The output of the neuron is the output of the activation function.

This process is represented in the block diagram shown in Figure 16.

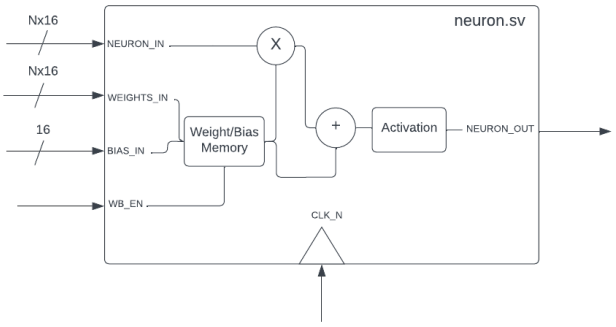


Figure 16: Neuron.sv Block Diagram

In the neuron module an addition and product (ixw) register was used. On every clock cycle the addition register would be set to the value of the bias. Next, each input/weight product would be calculated and the addition register would be incremented by each product value. The resulting sum of products and bias is passed into the activation function.

Since the system was implemented with fixed point notation in mind, the product of an input and a weight had to be bit shifted to the right 15 times. This also meant that the register in which the product would be stored would have to be increased in size to 32 bits, double the size of the input signal (0xFFFF $\times$ 0xFFFF = 0xFFFE0001).

Another issue that was encountered was that the addition size would be larger than the signal size. Considering the sum of all products plus the bias would result in a minimum of -11 and a maximum of +11 (each product being 1 from each of the inputs*weights), we would need 20 bits to store the sum. Because we would be adding a 16 bit signed value to a 20 bit signed value we would need to correct the result when the product is negative. System verilog would typically handle this itself, however becuase we perform a bitshift operation on the product, the signed bit is moved and we need to select only part of the product. For this reason when adding a negative value to the addtion register, we need to subtract 20'h10000 to ensure that the negativity is accounted for.

The System Verilog implementation for the neuron is shown in subsection 10.4.

6.2.1 Activation function

The activation function within the neuron can be interchanged. The two activation functions present in the design were the sigmoid function and the linear function. The linear function was very straightforward and simply connects the input to the output.

The sigmoid function was implemented using a look up table (LUT). The LUT values were generated using Python. First, all values representable by signed fixed point were generated. These values were then sorted by hex value (0x0000 - 0xfffff), which would allow for referencing by index when implemented in hardware. Next all values were passed into the sigmoid function. The resulting array was saved to a .mem file (sfp.mem), which would be implemented in system verilog using the ***$readmemh()*** function. The Python code used to generate values is shown in the jupter notebook in subsection 10.3.

In hardware the sigmoid activation module was not placed within the neuron module but instead outside. It was found that when placed within, the LUT was duplicated, which would result in inefficient utilisation. In the design each neuron would output the sum of products and biases. Whenever an input changed in value in the sigmoid LUT module, it would output the sigmoid value to the corresponding output wire. Each of these outputs are connected to the inputs of the next layer. This implementation is shown in subsection 10.6.

6.3 Delay

The delay function was relatively simple to implement. Like the neuron module, the input size would need to be scalable, which was achieved with the use of parameters. The module would trigger on the negative edge of a clock cycle. The block diagram for the delay module is shown in Figure 17 and the code is shown is shown in subsection 10.8.

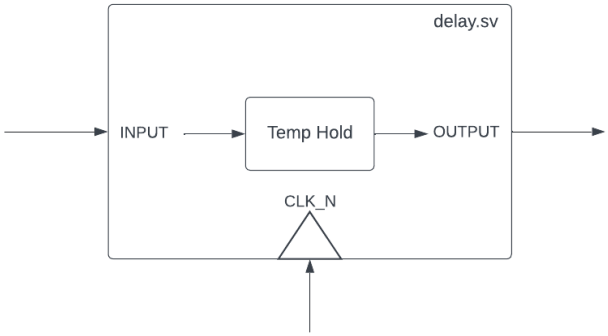


Figure 17: Delay.sv Block Diagram

6.4 TDNN container

The TDNN itself was constructed using a combination of the modules described above. The design of this module was based on the keras model shown in section 5. There were 2 in-

puts required by the network, one for the real component and the other for the imaginary component. The system also required a clock input to control both the neuron and the delay modules. The final input was the weight/bias enable. The output of the network was the real and imaginary components of the predistorted signal.

Like the keras model the network had 4 delay components for each input. When implementing this in system verilog each set of 4 delays were connected in series. The inputs and all of the delay outputs were densely connected to all of the neurons in layer 1 (Neruon1 - Neuron10). The outputs of all of the layer 1 neurons were all connected to the sigmoid function. The output of the sigmoid function was connected to the input of the second layer of neurons. The output of the second layer was connected to the output of the TDNN module.

Registers were created to drive the weight and bias inputs for all neurons. An initial block was used to extract all the weights and biases from the .mem files.

The System Verilog code for this module is shown in subsection 10.9 and its block diagram is shown in Figure 18.

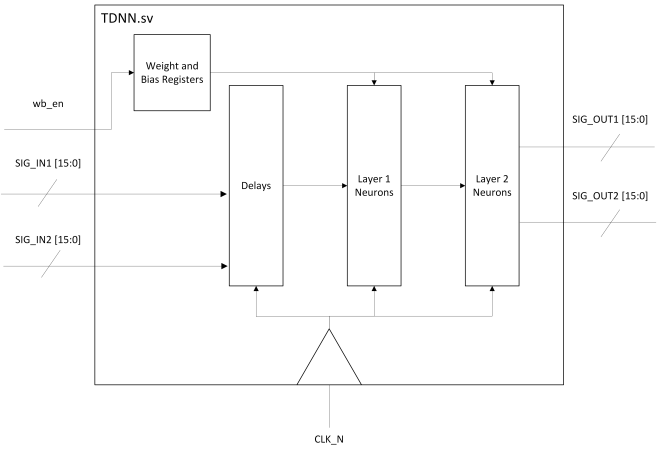


Figure 18: TDNN.sv Block Diagram

6.5 Testbench

A testbench was created in order to test whether the system works correctly. A series of 10 complex values were arbitrarily chosen and placed in an array. These values were inputted to the system and the outputs were compared to the output of the Keras neural network. The inputs and the outputs provided from the Keras model are shown below in Figure 19.

Real Input	Imag Input	Real Out	Imag Out
-0.55	-0.23	-0.26952	-0.0056
0.42	0.97	0.31524	0.30584
0.86	-0.34	0.35388	-0.30038
-0.99	-0.75	-0.52772	-0.14231
0.62	-0.12	0.27996	-0.15191
0.99	0.86	0.53818	0.15057
-0.39	-0.21	-0.21704	0.0491
-0.64	0.77	-0.1687	0.40365
0.91	-0.05	0.38737	-0.15875
0.12	-0.17	-0.01071	-0.10369

Figure 19: Inputs for Testing and the Outputs Provided by the Keras Model

The 10 inputs in the array were translated into the corresponding 16 bit signed fixed point hexadecimal values. These would be used as the inputs for the System Verilog testbench. The testbench is shown in subsection 10.10. The waveform generated is shown in Figure 20.

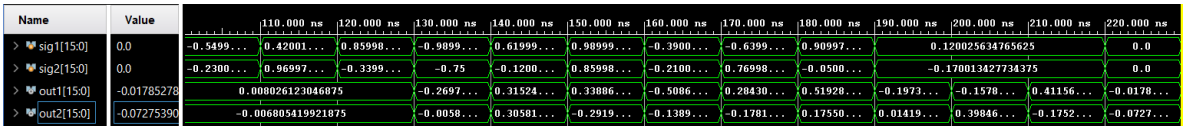


Figure 20: Testbench Waveform

The outputs from the simulation were compared to the outputs from the keras neural network, this is shown in Figure 21. In general both systems produced similar outputs. The maximum difference recorded was 0.03491. The slight differences in output could be attributed to the resolution of the hardware implementation. Large differences could be caused by compounded errors and amplification in products.

Verilog Real	Verilog Imag	Real Diff.	Imag Diff.
-0.2697	-0.0058	0.00018	0.0002
0.31524	0.30581	0	3E-05
0.33886	-0.2919	0.01502	0.00848
-0.5086	-0.1389	0.01912	0.00341
0.2843	-0.1781	0.00434	0.02619
0.51928	0.1755	0.0189	0.02493
-0.1973	0.01419	0.01974	0.03491
-0.1578	0.39846	0.0109	0.00519
0.41156	-0.1752	0.02419	0.01645
-0.0178	-0.0727	0.00709	0.03099

Figure 21: Verilog Outputs and Absolute Difference to Target

6.6 Nexys 4 container

A top level module was created to allow the TDNN system to interface with the I/O interface on the Nexys 4. The function of this system would be to simply test if the system functioned on hardware. The input to the system was the 16 switches that controls both real and complex inputs. The outputs were connected to the 8 value 7 segment displays. The system verilog file is shown in subsection 10.11.

Using the standard constraint file for the Nexys 4 provided by Digitlent, the clock frequency was set to 100MHz, the switch outputs were given port names and the 7 segment display inputs were given port names.

A 7 segment display module was created in order to display the 16 bit outputs as hexidecimal values. The display translated the input values into the 7 segment display format. The display updates at a rate 10,000 times slower than the system clock. The 7 segment display code is shown in subsection 10.12. The block diagram for the 7 segment display is shown in Figure 22.

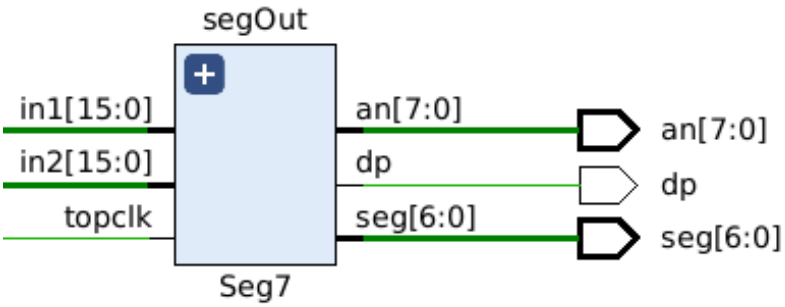


Figure 22: 7 segment display block diagram from Vivado

6.7 Synthesis And Implementation

Using Vivado the system was synthesised. Once syhnthesis was complete the resulting reports were analyed.

The design was found to be too large to be implemented on the Nexys 4. This is because there were not enough LUT units available on the Artix 7 chip. Unfortuneately, this meant the system could not be implemented on the board.

Name	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	DSPs (240)	Bonded IOB (210)	BUFGCTRL (32)
N TDNN_TOP	250424	3517	79534	4840	120	33	1

Figure 23: Nexys 4 Synthesis Utilisation Report

An attempt was made to synthesise the design for the Zynq 7000 ZC706 evaluation board;

however, the device on which the synthesis was performed did not have sufficient memory to compile the design. The device that was used had 16GB of RAM.

7 Ethics

Considering that the application of the system outlined in this project would be in wireless communications, several ethical considerations must be taken into account, such as security, safety, bias, and the use of intellectual property.

7.1 Security

Considering that the primrary application of this system is in communication systems the security of the signals are paramount. A predistortion system should accurately linearise a power amplifier's output, otherwise part of a transmitted signal may be lost which could lead to loss of communication and therefore posing risk to users.

It should be noted that the system presented in this paper is not ready for implementation and additional testing is required.

7.2 Safety

Since a predistortion system interacts with power amplifiers safety should be considered. Power amplifiers operate at high voltages and generate a large amount of heat. It is important that care is taken when interfacing with power amplifiers.

This design would need to be carefully evaluated and controlled if it is in contact with a power amplifier.

7.3 Bias

When training a neural network, it is important that the network is trained on a data set that is representative of all possible inputs that the implemented system might encounter. It must also be ensured that the system is not overtrained and is not biased towards the training data set. If a neural network is biased, it may produce incorrect outputs and could cause damage.

7.4 Intellectual Property

During the course of this project, the intellectual property (IP) of others was respected. Any nonoriginal material used was clearly referenced and not presented as the work of the author.

8 Future Work

Following the research and work carried out, this project could be expanded in the future to analyse and implement a neural network capable of predistorting a signal for a memory based

model. The primrary challenge in implementing a memory based predistortion system would be developing a mathematical predistortion model and generating sufficient data for training the model to properly pre-distort the signal. The TDNN structure provided in this report may be sufficient for implementing such a system. The memory-based volterra series model or polynomial model outlined in section 2 could be used.

Further work could be carried out to improve the accuracy of the Saleh model neural network. This report found that saturation occured at higher values for the training data set. Alternative network configurations could be explored along with alternative training methods.

There were several factors preventing the model from being implemented onto an FPGA. Work could be carried out to further optimise the design in order for it to fit onto the NEXYS 4 board. Alternatively, a more powerful device, than the one used for this project, could be used to Synthesise the model for the Zynq ZC706 evalulation board.

This project gave a detailed overview of how to translate a Keras-based neural network model into System Verilog. This could be used as a basis to implement a script that could automatically translate a Keras Python script into a functioning System Verilog project.

Lastly, future work could be carried out to minimise the number of neurons needed in the neural network in order to implement the Saleh model. The model itself could be modified or retrained on a different dataset in order to better implement Saleh predistortion.

9 Conclusion

In this report the process of creating a digital pre-distortion neural network for an FPGA was outlined. The general operation of neural networks was described to provide an understanding of the fundamentals on which the hardware model would be created. The Saleh model for Power Amplifiers was outlined and the corresponsing pre-distortion model was derived. This pre-distortion model would be the basis for generating the data-set used to train the neural network.

The process of creating a neural network in Keras was outlined and a model was created. Several activation functions were tested along with two different training data-sets. It was found that the network may be limited by the fact that weights and biases had been constrained to values between -1 and +1.

A System Verilog model was created, based on the keras TDNN model. The process of creating a neural network in hardware was described. The data structure was outlined along with the process of creating a sigmoid look up table. A test was carried out to validate that the Verilog Model matched the keras model. Finally the model was synthesised and a report for it's Utilisation was generated.

Ultimately, this report outlines the successful creation of a neural network design in System Verilog. The groundwork for implementing a working predistorter in system verilog is shown. The work shown in this report can alos be used as a guide on how to translate a neural

network from keras to System Verilog.

References

[1] Sonam Goyal and Jyoti Gupta. "Review Of Power Amplifier Linearization Techniques In Communication Systems". en. In: *International Journal of Engineering Research* 2.6 (2013), p. 8.

[2] Andžej Borel, Vaidotas Barzdėnas, and Aleksandr Vasjanov. "Linearization as a Solution for Power Amplifier Imperfections: A Review of Methods". en. In: *Electronics* 10.9 (May 2021), p. 1073. ISSN: 2079-9292. DOI: 10.3390/electronics10091073. URL: https://www.mdpi.com/2079-9292/10/9/1073 (visited on 12/10/2022).

[3] Rahul Gupta et al. "Adaptive Digital Baseband Predistortion for RF Power Amplifier Linearization". en. In: ().

[4] Quynh Anh Pham et al. "Dynamic Selection and Update of Digital Predistorter Coefficients for Power Amplifier Linearization". In: *2019 IEEE Topical Conference on RF/Microwave Power Amplifiers for Radio and Wireless Applications (PAWR)*. ISSN: 2473-4640. Jan. 2019, pp. 1–4. DOI: 10.1109/PAWR.2019.8708732.

[5] Chengye Jiang et al. "Block-Oriented Time-Delay Neural Network Behavioral Model for Digital Predistortion of RF Power Amplifiers". In: *IEEE Transactions on Microwave Theory and Techniques* 70.3 (Mar. 2022). Conference Name: IEEE Transactions on Microwave Theory and Techniques, pp. 1461–1473. ISSN: 1557-9670. DOI: 10.1109/TMTT.2021.3124211.

[6] Yufeng Wan, Tony J. Dodd, and Robert F. Harrison. "MODELING OF POWER AMPLIFIER NONLINEARITIES USING VOLTERRA SERIES". en. In: *IFAC Proceedings Volumes*. 16th IFAC World Congress 38.1 (Jan. 2005), pp. 785–790. ISSN: 1474-6670. DOI: 10.3182/20050703-6-CZ-1902.00132. URL: https://www.sciencedirect.com/science/article/pii/S1474667016361444 (visited on 04/22/2023).

[7] Wang Huadong et al. "An Memory polynomial model for power amplifiers". In: *2008 International Conference on Communications, Circuits and Systems*. May 2008, pp. 1346–1349. DOI: 10.1109/ICCCAS.2008.4658015.

[8] Kyeongsoo Jang et al. "Spectrum characteristics and predistortion gain in the nonlinear high power amplifier(HPA)". In: *2017 International Conference on Information and Communication Technology Convergence (ICTC)*. Oct. 2017, pp. 931–934. DOI: 10.1109/ICTC.2017.8190817.

[9] G. L. Shaw. "Donald Hebb: The Organization of Behavior". en. In: *Brain Theory*. Ed. by Günther Palm and Ad Aertsen. Berlin, Heidelberg: Springer, 1986, pp. 231–233. ISBN: 978-3-642-70911-1. DOI: 10.1007/978-3-642-70911-1_15.

[10] Arthur L Samuel. "Some Studies in Machine Learning Using the Game of Checkers". en. In: (), p. 20.

[11] Christopher M. Bishop. *Pattern recognition and machine learning*. en. Information science and statistics. New York: Springer, 2006. ISBN: 978-0-387-31073-2.

[12] Stuart J. Russell, Peter Norvig, and Ernest Davis. *Artificial intelligence: a modern approach*. en. 3rd ed. Prentice Hall series in artificial intelligence. Upper Saddle River: Prentice Hall, 2010. ISBN: 978-0-13-604259-4.

[13] Jaswinder Singh and Rajdeep Banerjee. "A Study on Single and Multi-layer Perceptron Neural Network". In: *2019 3rd International Conference on Computing Methodologies and Communication (ICCMC)*. Mar. 2019, pp. 35–40. DOI: 10.1109/ICCMC.2019.8819775.

[14] Warren S. McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". en. In: *The bulletin of mathematical biophysics* 5.4 (Dec. 1943), pp. 115–133. ISSN: 1522-9602. DOI: 10.1007/BF02478259. URL: https://doi.org/10.1007/BF02478259 (visited on 12/06/2022).

[15] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. "Understanding of a convolutional neural network". In: *2017 International Conference on Engineering and Technology (ICET)*. Aug. 2017, pp. 1–6. DOI: 10.1109/ICEngTechnol.2017.8308186.

[16] Oludare Isaac Abiodun et al. "State-of-the-art in artificial neural network applications: A survey". In: *Heliyon* 4.11 (Nov. 2018), e00938. ISSN: 2405-8440. DOI: 10.1016/j.heliyon.2018.e00938. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6260436/ (visited on 12/09/2022).

[17] Ya-Qi Yu and Wu-Jun Li. "Densely Connected Time Delay Neural Network for Speaker Verification". en. In: *Interspeech 2020*. ISCA, Oct. 2020, pp. 921–925. DOI: 10.21437/Interspeech.2020-1275. URL: https://www.isca-speech.org/archive/interspeech_2020/yu20b_interspeech.html (visited on 05/04/2023).

[18] Ankit Kumar and Rajesh Kumar Aggarwal. "Hindi speech recognition using time delay neural network acoustic modeling with i-vector adaptation". en. In: *International Journal of Speech Technology* 25.1 (Mar. 2022), pp. 67–78. ISSN: 1572-8110. DOI: 10.1007/s10772-020-09757-0. URL: https://doi.org/10.1007/s10772-020-09757-0 (visited on 12/10/2022).

[19] Kenneth Levenberg. "A method for the solution of certain non-linear problems in least squares". en. In: *Quarterly of Applied Mathematics* 2.2 (1944), pp. 164–168. ISSN: 0033-569X, 1552-4485. DOI: 10.1090/qam/10666. URL: https://www.ams.org/qam/1944-02-02/S0033-569X-1944-10666-0/ (visited on 12/15/2022).

[20] Donald W. Marquardt. "An Algorithm for Least-Squares Estimation of Nonlinear Parameters". In: *Journal of the Society for Industrial and Applied Mathematics* 11.2 (June 1963). Publisher: Society for Industrial and Applied Mathematics, pp. 431–441. ISSN: 0368-4245. DOI: 10.1137/0111030. URL: https://epubs.siam.org/doi/10.1137/0111030 (visited on 12/15/2022).

[21] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. arXiv:1412.6980 [cs]. Jan. 2017. DOI: 10.48550/arXiv.1412.6980. URL: http://arxiv.org/abs/1412.6980 (visited on 12/15/2022).

[22] Hang Zhao et al. *Loss Functions for Neural Networks for Image Processing*. arXiv:1511.08861 [cs]. Apr. 2018. DOI: 10.48550/arXiv.1511.08861. URL: http://arxiv.org/abs/1511.08861 (visited on 04/30/2023).

[23] Katarzyna Janocha and Wojciech Marian Czarnecki. *On Loss Functions for Deep Neural Networks in Classification*. arXiv:1702.05659 [cs]. Feb. 2017. URL: http://arxiv.org/abs/1702.05659 (visited on 04/30/2023).

[24] Nitish Shirish Keskar et al. *On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima*. arXiv:1609.04836 [cs, math]. Feb. 2017. URL: http://arxiv.org/abs/1609.04836 (visited on 05/01/2023).

[25] Janardan Misra and Indranil Saha. "Artificial neural networks in hardware: A survey of two decades of progress". en. In: *Neurocomputing*. Artificial Brains 74.1 (Dec. 2010), pp. 239–255. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2010.03.021. URL: https://www.sciencedirect.com/science/article/pii/S092523121000216X (visited on 12/01/2022).

[26] Martin Abadi et al. "TensorFlow: A system for large-scale machine learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 265–283. URL: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf (visited on 12/09/2022).

[27] Norman P Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing UnitTM". en. In: (), p. 17.

10 Appendix

10.1 QAM Matlab script

```
%% QAM signal generation
M = 256;
% x will transition from each possible QAM state to each of the other QAM
% states, all possible transitions are captured
x = [];
for i = 0:M-1
    for j = 0:M-1
        x = [x,i,j];
    end
end

% generate QAM signal and normalise
CleanQamInput = qammod(x,M)/15;

%% Saleh distortion

% Saleh Coefficients
AMAM = [2.1587,        1.1517];
AMPM = [4.0033, 9.1040];

absy = abs(CleanQamInput);
AM = (absy    .* AMAM(1)) ./ ...
     (absy.^2 .* AMAM(2) + 1);
PM = (pi/3)*(absy.^2    .* AMPM(1)) ./ ...
     (absy.^2 .* AMPM(2) + 1);

PAQam = (AM.* exp(1i.*(angle(CleanQamInput) + PM)));

%% Pre-distortion

absy = abs(CleanQamInput)/(max(abs(CleanQamInput))/(sqrt((AMAM(1)^2)/(4*AMAM(2)))));

AMpre = zeros(1,length(absy));
count = 0;
for X = absy
    count = count + 1;
    if ((sqrt((AMAM(1)^2)/4*AMAM(2))>X) && (0 < X))
        new_val = ((AMAM(1) - sqrt(AMAM(1)^2 -(4 .* AMAM(2) .* X .^ 2)))./ (2 .* AMAM(2) .* X));
    else
        new_val = 1/AMAM(2);
    end
    AMpre(count)= new_val;
end

```

```
PMpre = - (pi/3) * (AMpre.^2    .* AMPM(1)) ./ ...
        ((AMpre.^2 .* AMPM(2)) + 1);

PAQamPre = (AMpre.* exp(1i.*(angle(CleanQamInput) + PMpre)));

%% applying saleh model post pre-distortion
absypre = abs(PAQamPre);

AMpost = (absypre    .* AMAM(1)) ./ ...
     (absypre.^2 .* AMAM(2) + 1);
PMpost = (pi/3)*(absypre.^2    .* AMPM(1)) ./ ...
     (absypre.^2 .* AMPM(2) + 1);

PAQamOutPredis = (AMpost.* exp(1i.*(angle(PAQamPre) + PMpost)));


%% Plots
% plot input, distorted signal, predistorted signal and distorted
% pre-distorted signal

scatterplot(CleanQamInput)
scatterplot(PAQam)
scatterplot(PAQamPre)
scatterplot(PAQamOutPredis)

%% Test predistortion
%% encode an image with 256 QAM

% Read PNG image
img = imread('TestImage.png');

% Convert image to grayscale
gray_img = rgb2gray(img);

% Reshape grayscale image into a vector
data = reshape(gray_img, [], 1);

% Modulate data using 256-QAM
modulated_data = qammod(data, 256)/15;

%%

absy = abs(modulated_data)/(max(abs(modulated_data))/(sqrt((AMAM(1)^2)/(4*AMAM(2)))));
absy = absy.';
AMpre = zeros(1,length(absy));
count = 0;
for X = absy
    count = count + 1;
    if ((sqrt((AMAM(1)^2)/4*AMAM(2))>X) && (0 < X))
```

```
        new_val = ((AMAM(1) - sqrt(AMAM(1)^2 -(4 .* AMAM(2) .* X .^ 2))) ./ (2 .* AMAM(2) .* X));
    else
        new_val = 1/AMAM(2);
    end
    AMpre(count)= new_val;
end

PMpre = - (pi/3) * (AMpre.^2    .* AMPM(1)) ./ ...
        ((AMpre.^2 .* AMPM(2)) + 1);

ImagePredis = (AMpre.* exp(1i.*(angle(modulated_data.') + PMpre)));
ImagePredis = ImagePredis.';

scatterplot(ImagePredis)

%% Apply saleh
absypre = abs(ImagePredis);

AMpost = (absypre    .* AMAM(1)) ./ ...
     (absypre.^2 .* AMAM(2) + 1);
PMpost = (pi/3)*(absypre.^2    .* AMPM(1)) ./ ...
     (absypre.^2 .* AMPM(2) + 1);

correct = (max(abs(modulated_data))/(sqrt((AMAM(1)^2)/(4*AMAM(2)))));

PAImageOut = ((AMpost*correct).* exp(1i.*(angle(ImagePredis) + PMpost)));
%% Demodulate the QAM signal

% Demodulate modulated data using 256-QAM
demodulated_data = qamdemod(modulated_data*15, 256);

% Reshape demodulated data into grayscale image
gray_img = reshape(demodulated_data, size(gray_img));

% Convert grayscale image to RGB image
rgb_img = cat(3, gray_img, gray_img, gray_img);

% Write RGB image to PNG file
imwrite(rgb_img, 'output.png');
%% Training & Test data output

% Training data input
real_train_input = real(CleanQamInput);
csvwrite("saleh_real_train_input.csv",real_train_input);
imag_train_input = imag(CleanQamInput);
csvwrite("saleh_imag_train_input.csv",imag_train_input);

% Training data output
real_train_output = real(PAQamPre);
```

```
csvwrite("saleh_real_train_output.csv",real_train_output);
imag_train_output = imag(PAQamPre);
csvwrite("saleh_imag_train_output.csv",imag_train_output);

% Test data input
real_test_input = real(modulated_data).';
csvwrite("saleh_real_test_input.csv",real_test_input);
imag_test_input = imag(modulated_data).';
csvwrite("saleh_imag_test_input.csv",imag_test_input);

% Test data output
real_test_output = real(ImagePredis).';
csvwrite("saleh_real_test_output.csv",real_test_output);
imag_test_output = imag(ImagePredis).';
csvwrite("saleh_imag_test_output.csv",imag_test_output);

```

10.2 APSK Matlab script

```
%% APSK signal generation
M = 256;
% x will transition from each possible APSK state to each of the other APSK
% states, all possible transitions are captured
x = [];
for i = 0:M-1
    for j = 0:M-1
        x = [x,i,j];
    end
end

% generate APSK signal and normalise
M = [32 32 32 32 32 32 32 32];
radii = [1/8 2/8 3/8 4/8 5/8 6/8 7/8 1];
modOrder = sum(M);
g = 0:modOrder-1;

CleanApskInput = apskmod(x,M,radii);

%% Saleh distortion

% Saleh Coefficients
AMAM = [2.1587,        1.1517];
AMPM = [4.0033, 9.1040];

absy = abs(CleanApskInput);
AM = (absy    .* AMAM(1)) ./ ...
```

```
     (absy.^2 .* AMAM(2) + 1);
PM = (pi/3)*(absy.^2    .* AMPM(1)) ./ ...
     (absy.^2 .* AMPM(2) + 1);

PAAPSK = (AM.* exp(1i.*(angle(CleanApskInput) + PM)));

%% Pre-distortion

absy = abs(CleanApskInput)/(max(abs(CleanApskInput))/(sqrt((AMAM(1)^2)/(4*AMAM(2)))));

AMpre = zeros(1,length(absy));
count = 0;
for X = absy
    count = count + 1;
    if ((sqrt((AMAM(1)^2)/4*AMAM(2))>X) && (0 < X))
        new_val = ((AMAM(1) - sqrt(AMAM(1)^2 -(4 .* AMAM(2) .* X .^ 2))./ (2 .* AMAM(2) .* X));
    else
        new_val = 1/AMAM(2);
    end
    AMpre(count)= new_val;
end

PMpre = - (pi/3) * (AMpre.^2    .* AMPM(1)) ./ ...
        ((AMpre.^2 .* AMPM(2)) + 1);

PAAPSKPre = (AMpre.* exp(1i.*(angle(CleanApskInput) + PMpre)));

%% applying saleh model post pre-distortion
absypre = abs(PAAPSKPre);

AMpost = (absypre    .* AMAM(1)) ./ ...
     (absypre.^2 .* AMAM(2) + 1);
PMpost = (pi/3)*(absypre.^2    .* AMPM(1)) ./ ...
     (absypre.^2 .* AMPM(2) + 1);

PAAPSKOutPredis = (AMpost.* exp(1i.*(angle(PAAPSKPre) + PMpost)));


%% Plots
% plot input, distorted signal, predistorted signal and distorted
% pre-distorted signal

scatterplot(CleanApskInput)
scatterplot(PAAPSK)
scatterplot(PAAPSKPre)
scatterplot(PAAPSKOutPredis)

%% Test predistortion
%% encode an image with 256 APSK
```

```
77
78  % Read PNG image
79  img = imread('TestImage.png');
80
81  % Convert image to grayscale
82  gray_img = rgb2gray(img);
83
84  % Reshape grayscale image into a vector
85  data = reshape(gray_img, [], 1);
86
87  % Modulate data using 256-APSK
88  modulated_data = apskmod(data,M,radii);
89
90  %%
91
92  absy = abs(modulated_data)/(max(abs(modulated_data))/(sqrt((AMAM(1)^2)/(4*AMAM(2)))));
93  absy = absy.';
94  AMpre = zeros(1,length(absy));
95  count = 0;
96  for X = absy
97      count = count + 1;
98      if ((sqrt((AMAM(1)^2)/4*AMAM(2))>X) && (0 < X))
99          new_val = ((AMAM(1) - sqrt(AMAM(1)^2 -(4 .* AMAM(2) .* X .^ 2)))./ (2 .* AMAM(2) .* X));
100     else
101         new_val = 1/AMAM(2);
102     end
103     AMpre(count)= new_val;
104 end
105
106 PMpre = - (pi/3) * (AMpre.^2    .* AMPM(1)) ./ ...
107         ((AMpre.^2 .* AMPM(2)) + 1);
108
109 ImagePredis = (AMpre.* exp(1i.*(angle(modulated_data.') + PMpre)));
110 ImagePredis = ImagePredis.';
111
112 scatterplot(ImagePredis)
113
114 %% Apply saleh
115 absypre = abs(ImagePredis);
116
117 AMpost = (absypre    .* AMAM(1)) ./ ...
118      (absypre.^2 .* AMAM(2) + 1);
119 PMpost = (pi/3)*(absypre.^2    .* AMPM(1)) ./ ...
120      (absypre.^2 .* AMPM(2) + 1);
121
122 correct = (max(abs(modulated_data))/(sqrt((AMAM(1)^2)/(4*AMAM(2)))));
123
124 PAImageOut = ((AMpost*correct).* exp(1i.*(angle(ImagePredis) + PMpost)));
125 %% Demodulate the APSK signal
```

```
126
127 % Demodulate modulated data using 256-APSK
128 demodulated_data = apskdemod(modulated_data,M, radii);
129
130 % Reshape demodulated data into grayscale image
131 gray_img = reshape(demodulated_data, size(gray_img));
132
133 % Convert grayscale image to RGB image
134 rgb_img = cat(3, gray_img, gray_img, gray_img);
135
136 % Write RGB image to PNG file
137 imwrite(rgb_img, 'output.png');
138 %% Training & Test data output
139
140 % Training data input
141 real_train_input = real(CleanApskInput);
142 csvwrite("saleh_real_train_input.csv",real_train_input);
143 imag_train_input = imag(CleanApskInput);
144 csvwrite("saleh_imag_train_input.csv",imag_train_input);
145
146 % Training data output
147 real_train_output = real(PAAPSKPre);
148 csvwrite("saleh_real_train_output.csv",real_train_output);
149 imag_train_output = imag(PAAPSKPre);
150 csvwrite("saleh_imag_train_output.csv",imag_train_output);
151
152 % Test data input
153 real_test_input = real(modulated_data).';
154 csvwrite("saleh_real_test_input.csv",real_test_input);
155 imag_test_input = imag(modulated_data).';
156 csvwrite("saleh_imag_test_input.csv",imag_test_input);
157
158 % Test data output
159 real_test_output = real(ImagePredis).';
160 csvwrite("saleh_real_test_output.csv",real_test_output);
161 imag_test_output = imag(ImagePredis).';
162 csvwrite("saleh_imag_test_output.csv",imag_test_output);
163
```

10.3 Python Jupyter Notebook

See next page for code and outputs for the Python Jupyter Notebook

TDNN1

May 4, 2023

1 TDNN for digital pre-distortion

Import modules

[ ]:
```
import tensorflow as tf
from tensorflow import keras
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
import pandas as pd
import csv
from numpy import genfromtxt
import pydot
```

1.1 Network Config

Enter values for network input size (number of inputs into the TDNN) and the number of desired delays within the TDNN.

The number of inputs will be dependent on the dataset setup. For the PA dataset below there are 2 inputs - real and imaginary parts of the signal

[ ]:
```
number_of_delays = 4; # This defines the amount of delays that are in the TDNN
number_of_inputs = 2; # The amount of inputs for the nueral network
APSK = False;
QAM = not APSK;
```

[ ]:
```
NN_input_size = number_of_inputs*(number_of_delays+1)
```

Creating the Dataset

[ ]:
```
if(QAM == True):
    #training data
    real_input = np.genfromtxt('./saleh_real_train_input.csv', delimiter=',')
    real_output = np.genfromtxt('./saleh_real_train_output.csv', delimiter=',')
    imag_input = np.genfromtxt('./saleh_imag_train_input.csv', delimiter=',')
    imag_output = np.genfromtxt('./saleh_imag_train_output.csv', delimiter=',')

    #test data
```

```
    real_test_input = np.genfromtxt('./saleh_real_test_input.csv',␣
 ↪delimiter=',')
    real_test_output = np.genfromtxt('./saleh_real_test_output.csv',␣
 ↪delimiter=',')
    imag_test_input = np.genfromtxt('./saleh_imag_test_input.csv',␣
 ↪delimiter=',')
    imag_test_output = np.genfromtxt('./saleh_imag_test_output.csv',␣
 ↪delimiter=',')

if (APSK == True):
    #training data
    real_input = np.genfromtxt('./saleh_real_train_input_apsk.csv',␣
 ↪delimiter=',')
    real_output = np.genfromtxt('./saleh_real_train_output_apsk.csv',␣
 ↪delimiter=',')
    imag_input = np.genfromtxt('./saleh_imag_train_input_apsk.csv',␣
 ↪delimiter=',')
    imag_output = np.genfromtxt('./saleh_imag_train_output_apsk.csv',␣
 ↪delimiter=',')

    #test data
    real_test_input = np.genfromtxt('./saleh_real_test_input_apsk.csv',␣
 ↪delimiter=',')
    real_test_output = np.genfromtxt('./saleh_real_test_output_apsk.csv',␣
 ↪delimiter=',')
    imag_test_input = np.genfromtxt('./saleh_imag_test_input_apsk.csv',␣
 ↪delimiter=',')
    imag_test_output = np.genfromtxt('./saleh_imag_test_output_apsk.csv',␣
 ↪delimiter=',')
```

Combine real and imaginary parts into a signle input

[ ]:
```
train_input = np.array([real_input,imag_input])
train_output = np.array([real_output,imag_output])
test_input = np.array([real_test_input,imag_test_input])
test_output = np.array([real_test_output,imag_test_output])
```

Transpose matrix

[ ]:
```
train_input = np.transpose(train_input)
train_output = np.transpose(train_output)
test_input = np.transpose(test_input)
test_output = np.transpose(test_output)
```

Delay function to add addtional columns onto the input, acting as the delays for the TDNN

[ ]:
```
def delay(X, no_delay):
    """
```

```
    This function will expand each row of a given matrix by input no_delay

    Given a Nx2 matrix, and no_delay = 4, the output will be a Nx10

    The additional columns will be based on the previous rows
    n-1, n-2, n-3 and n-4.
    """
    input_samples = X.shape[0]
    input_width = X.shape[1]
    blank_delay = np.zeros(input_width)
    Y = np.empty([input_samples,(input_width*(no_delay+1))])
    for n in range(input_samples):
        delay_array = []
        for i in range(no_delay):
            if((n-(i+1))>=0):
                delay_array = np.append(delay_array, X[n-(i+1)])
            else:
                delay_array = np.append(delay_array,blank_delay)
        Y[n] = np.r_[X[n],delay_array]

    return(Y)
```

[ ]:
```
train_input_delayed = delay(train_input,number_of_delays)
test_input_delayed = delay(test_input,number_of_delays)
```

1.1.1 Create the network structure:

The following creates a NN keras strcture and Compiles it and trains it on the dataset

[ ]:
```
TDNN = keras.Sequential([
    keras.layers.
 ↪Dense(10,input_shape=(NN_input_size,),activation='sigmoid',kernel_constraint␣
 ↪= keras.constraints.MinMaxNorm(-1,1), bias_constraint= keras.constraints.
 ↪MinMaxNorm(-1,1)),
    keras.layers.Dense(2,activation='linear',kernel_constraint = keras.
 ↪constraints.MinMaxNorm(-1,1),bias_constraint = keras.constraints.
 ↪MinMaxNorm(-1,1))
])

TDNN.compile(
    optimizer= keras.optimizers.Adam(),
    loss='mse',
    metrics=['mean_absolute_error', 'accuracy'],
    steps_per_execution= 1
)

TDNN.fit(train_input_delayed,train_output,
```

```
        batch_size=32,
        epochs=10,
        validation_split=0.1,
        shuffle = True)
```

```
Epoch 1/10
3687/3687 [==============================] - 5s 1ms/step - loss: 0.0220 -
mean_absolute_error: 0.0606 - accuracy: 0.9480 - val_loss: 0.0018 -
val_mean_absolute_error: 0.0281 - val_accuracy: 0.9494
Epoch 2/10
3687/3687 [==============================] - 4s 1ms/step - loss: 0.0023 -
mean_absolute_error: 0.0276 - accuracy: 0.9820 - val_loss: 0.0013 -
val_mean_absolute_error: 0.0240 - val_accuracy: 0.9648
Epoch 3/10
3687/3687 [==============================] - 4s 1ms/step - loss: 0.0021 -
mean_absolute_error: 0.0269 - accuracy: 0.9799 - val_loss: 0.0015 -
val_mean_absolute_error: 0.0274 - val_accuracy: 0.9461
Epoch 4/10
3687/3687 [==============================] - 4s 1ms/step - loss: 0.0020 -
mean_absolute_error: 0.0266 - accuracy: 0.9797 - val_loss: 0.0014 -
val_mean_absolute_error: 0.0270 - val_accuracy: 0.9492
Epoch 5/10
3687/3687 [==============================] - 4s 1ms/step - loss: 0.0020 -
mean_absolute_error: 0.0265 - accuracy: 0.9796 - val_loss: 0.0014 -
val_mean_absolute_error: 0.0263 - val_accuracy: 0.9471
Epoch 6/10
3687/3687 [==============================] - 4s 1ms/step - loss: 0.0020 -
mean_absolute_error: 0.0264 - accuracy: 0.9801 - val_loss: 0.0013 -
val_mean_absolute_error: 0.0248 - val_accuracy: 0.9464
Epoch 7/10
3687/3687 [==============================] - 4s 1ms/step - loss: 0.0020 -
mean_absolute_error: 0.0262 - accuracy: 0.9797 - val_loss: 0.0014 -
val_mean_absolute_error: 0.0257 - val_accuracy: 0.9613
Epoch 8/10
3687/3687 [==============================] - 4s 1ms/step - loss: 0.0020 -
mean_absolute_error: 0.0261 - accuracy: 0.9799 - val_loss: 0.0015 -
val_mean_absolute_error: 0.0274 - val_accuracy: 0.9478
Epoch 9/10
3687/3687 [==============================] - 4s 1ms/step - loss: 0.0020 -
mean_absolute_error: 0.0260 - accuracy: 0.9800 - val_loss: 0.0012 -
val_mean_absolute_error: 0.0235 - val_accuracy: 0.9667
Epoch 10/10
3687/3687 [==============================] - 4s 1ms/step - loss: 0.0020 -
mean_absolute_error: 0.0259 - accuracy: 0.9801 - val_loss: 0.0013 -
val_mean_absolute_error: 0.0257 - val_accuracy: 0.9471
```

[ ]: <keras.callbacks.History at 0x145a35810c0>

Evalutate the system on the test data

[ ]:
```
TDNN.evaluate(test_input_delayed,test_output)
```

```
36450/36450 [==============================] - 34s 931us/step - loss: 0.0418 -
mean_absolute_error: 0.1371 - accuracy: 0.9984
```

[ ]:
```
[0.04176266863942146, 0.13712984323501587, 0.9983898997306824]
```

Generate all of the outputs from the test data

[ ]:
```
Predicted = TDNN.predict(test_input_delayed)
Predicted
```

```
36450/36450 [==============================] - 26s 723us/step
```

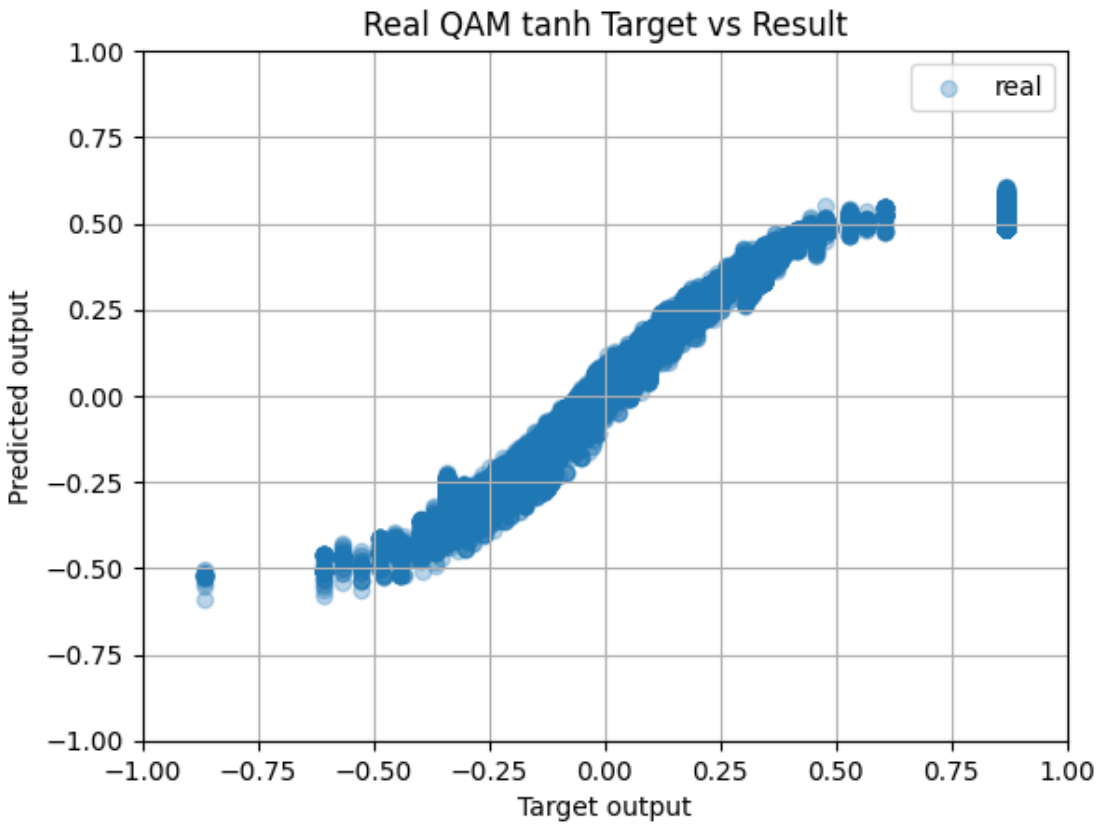
[ ]:
```
array([[-0.30208015,  0.55623484],
       [-0.30253452,  0.5574628 ],
       [-0.28809822,  0.52184594],
       …,
       [-0.25255996,  0.48147568],
       [-0.25256002,  0.48147568],
       [-0.25256002,  0.48147568]], dtype=float32)
```

Create a figures to compare the network output and the target output

[ ]:
```
plt.figure()
fig, ax = plt.subplots()
plt.xlabel("Target output")
plt.ylabel("Predicted output")
plt.title('Real QAM tanh Target vs Result')
plt.xlim([-1,1])
plt.ylim([-1,1])
plt.grid()
ax.scatter(test_output[:,0],Predicted[:,0],alpha=0.3,label='real')
ax.legend()
```

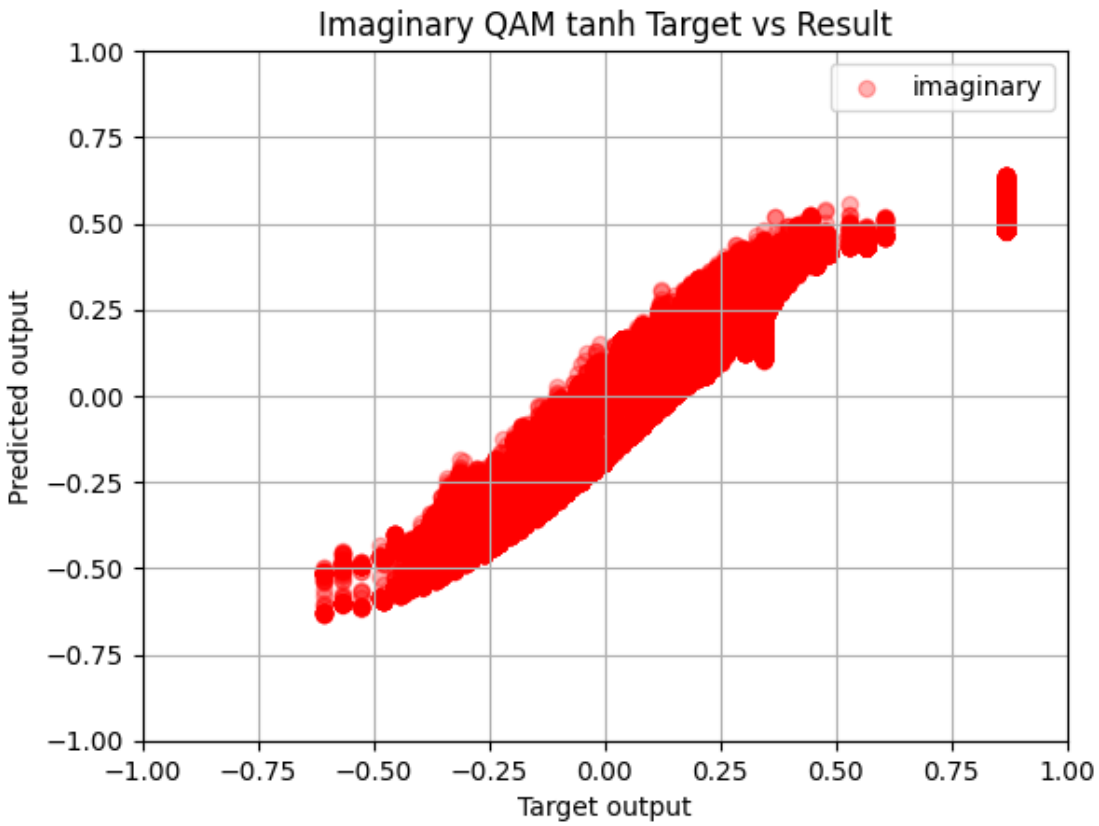
[ ]:
```
<matplotlib.legend.Legend at 0x145466af8e0>
```

```
<Figure size 640x480 with 0 Axes>
```



[ ]:
```
plt.figure()
fig, ax = plt.subplots()
plt.xlabel("Target output")
plt.ylabel("Predicted output")
plt.title('Imaginary QAM tanh Target vs Result')
plt.xlim([-1,1])
plt.ylim([-1,1])
plt.grid()
ax.scatter(test_output[:,1],Predicted[:,1],alpha=0.
 ↪3,color='red',label='imaginary')
ax.legend()
```

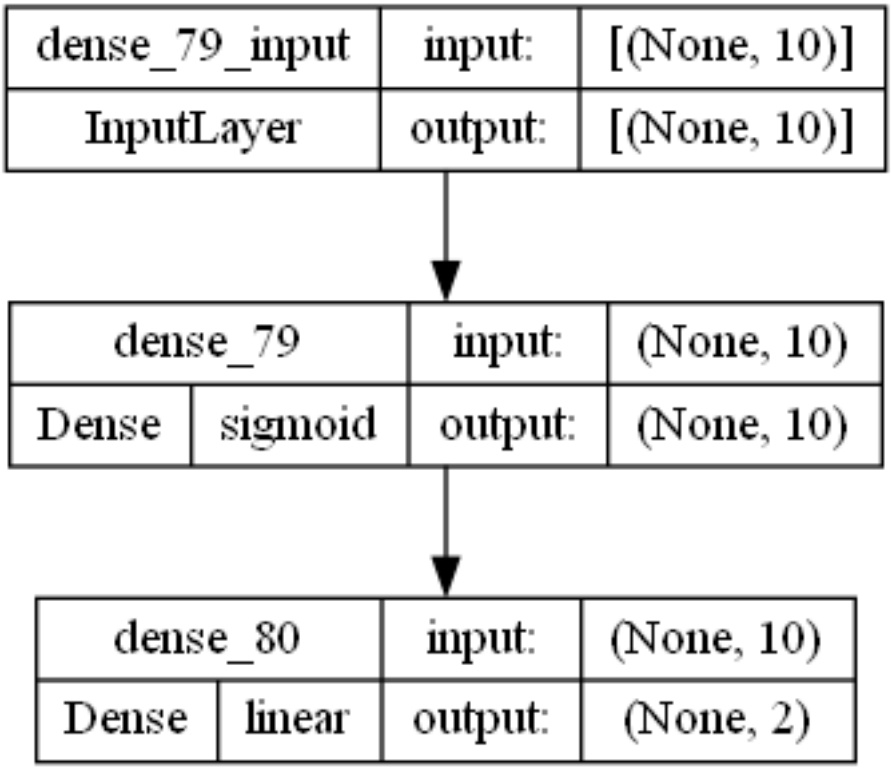
[ ]:
```
<matplotlib.legend.Legend at 0x1454684cb80>

<Figure size 640x480 with 0 Axes>
```



Display the network structure

[ ]:
```
import pydot
from keras.utils.vis_utils import plot_model
plot_model(TDNN, to_file='model_plot.png', show_shapes=True,␣
  ↪show_layer_names=True, show_layer_activations=True)
```

[ ]:



Visualise the network structure

[ ]:
```
import keras_visualizer as kv

kv.visualizer(TDNN, file_name='visualTDNN', file_format= 'png')
```

Float to fixed point hex conversion

[ ]:
```
def float_to_fixed_point(value):
    """
    This function takes in a floating point value
    and converts it to signed 16 bit fixed point hex
    """
    fixed_point = int(value * (2 ** 15 - 1))+1
    if fixed_point < 0:
        fixed_point = (1 << 16) + fixed_point - 2
    elif value == 0:
        fixed_point = 0
    return hex(fixed_point)[2:]
```

Float to fixed point binary conversion

[ ]:
```
def float_to_bin_fixed_point(value):
    '''
    This function takes in a floating point value
    and converts it to signed 16 bit fixed point binary
    '''
    fixed_point = int(value * (2 ** 15 - 1)) + 1
    print(fixed_point)
    if fixed_point < 0:
        fixed_point = (1 << 16) + fixed_point - 3
        print(fixed_point)
    return bin(fixed_point)[2:].zfill(16)
```

1.1.2 get weights and biases for layer 1

[ ]:
```
bias = open('bias1.mem', 'w')

for x in np.transpose(TDNN.layers[0].get_weights()[1]):
    data = "{}\n".format(float_to_fixed_point(x)) #string with hex value
    data = data.replace('0x', '') #remove any 0x
    bias.write(data) #write to file

bias.close()

weights = open('weights1.mem', 'w')

for x in np.transpose(TDNN.layers[0].get_weights()[0]):
    for i in x:
        data = "{}\n".format(float_to_fixed_point(i))
        data = data.replace('0x', '')
        weights.write(data)

weights.close()
```

1.1.3 get weights and biases for layer 2

[ ]:
```
#get weights and biases for layer 2
bias = open('bias2.mem', 'w')

for x in np.transpose(TDNN.layers[1].get_weights()[1]):
    data = "{}\n".format(float_to_fixed_point(x)) #string with hex value
    data = data.replace('0x', '') #remove any 0x
    bias.write(data) #write to file

bias.close()

weights = open('weights2.mem', 'w')
```

```
for x in np.transpose(TDNN.layers[1].get_weights()[0]):
    for i in x:
        data = "{}\n".format(float_to_fixed_point(i))
        data = data.replace('0x', '')
        weights.write(data)

weights.close()
```

[ ]:
```
x = delay(np.array([[0,0],[0,0]]), 4)
TDNN.predict(x)
```

```
1/1 [==============================] - 0s 49ms/step
```

[ ]:
```
array([[ 0.00821954, -0.00660184],
       [ 0.00821954, -0.00660184]], dtype=float32)
```

Generate Sigmoid values for the network

[ ]:
```
#gen sigmoid LUT values
sigmoid_size = 2**20;
#All 16-bit signed fixed point values between -1 and 1
sig_index = range(0,(sigmoid_size)-1)
signed_value = [float(-15)]
for x in sig_index:
    #each subsequent value is a step larger than the last
    signed_value.append(signed_value[-1] + 0.000030517578125)

#sorted from smallest hex value to largest: 0x0 - 0x1fffff
sorted_values = signed_value[491520:] + signed_value[0:491520]

sig = []
for sfp in sorted_values:
    sig.append(1/(1+np.exp(-sfp)))

sfp = open('sfp.mem', 'w')
for x in sig:
    write_str = "{}\n".format(float_to_fixed_point(x))
    sfp.write(write_str)

sfp.close()
```

Generate Outputs for the Neural Network to compare with System Verilog Simulation

[ ]:
```
A = delay(np.array([[-0.55,-0.23],
                    [0.42,0.97],
                    [0.86,-0.34],
                    [-0.99,-0.75],
                    [0.62,-0.12],
                    [0.99,0.86],
```

```
                    [-0.39,-0.21],
                    [-0.64,0.77],
                    [0.91,-0.05],
                    [0.12,-0.17]]),
          number_of_delays)



for x in (A):
    print(float_to_fixed_point(x[0]),end=', ')
    print(float_to_fixed_point(x[1]))

TDNN.predict(A)
```

```
b99a, e28f
35c3, 7c28
6e14, d47b
8148, a000
4f5c, f0a3
7eb8, 6e14
ce14, e51e
ae15, 628f
747a, f999
f5d, ea3d
1/1 [==============================] - 0s 23ms/step
```

[ ]:
```
array([[-0.26952857, -0.00560638],
       [ 0.31524706,  0.3058448 ],
       [ 0.35388988, -0.3003893 ],
       [-0.52772814, -0.14231658],
       [ 0.2799639 , -0.15191057],
       [ 0.53818667,  0.1505712 ],
       [-0.21704316,  0.04910013],
       [-0.16870686,  0.40365633],
       [ 0.38737702, -0.15875062],
       [-0.01071233, -0.10369855]], dtype=float32)
```

10.4 Neuron without activation

```
module NEURON #(
    parameter SIG_SIZE = 16,
    parameter WEIGHT_SIZE = 16,
    parameter ADDITION_SIZE = 20,
    parameter NUM_INPUTS = 3
  )(
    input signed [SIG_SIZE-1:0] NEURON_IN[NUM_INPUTS-1:0],
    input signed [WEIGHT_SIZE-1:0] WEIGHTS_IN[NUM_INPUTS-1:0],
    input WB_EN,
    input signed [WEIGHT_SIZE-1:0] BIAS_IN,
    input CLOCK_N,
    output reg signed [ADDITION_SIZE-1:0] NEURON_OUT
  );
  logic signed [WEIGHT_SIZE-1:0] weights[NUM_INPUTS-1:0];
  logic signed [WEIGHT_SIZE-1:0] bias;
  logic signed [SIG_SIZE*2-1:0] ixw [NUM_INPUTS-1:0];
  logic signed [ADDITION_SIZE-1:0] addition;
  integer i;

  assign NEURON_OUT = addition;

  initial
  begin
    addition <= 0;
  end

  always @(negedge CLOCK_N)
  begin
    if(WB_EN)
    begin
      for(i = 0; i<NUM_INPUTS;i++)
      begin
        weights[i] <= WEIGHTS_IN[i];
      end
      bias <= BIAS_IN;
    end
    addition = bias;
    //multiply inputs by weights
    //add product to addition register
    for(i=0;i<NUM_INPUTS;i++)
    begin
      ixw[i] <= ((NEURON_IN[i]*weights[i])>>15);
      addition = addition + ixw[i][15:0];
      if(ixw[i] > 16'h7fff)
        addition = addition - 20'h10000;
    end
```

```
  end

endmodule
```

10.5 Neuron with linear activation

```
module NEURON_LIN #(
    parameter SIG_SIZE = 16,
    parameter WEIGHT_SIZE = 16,
    parameter ADDITION_SIZE = 20,
    parameter NUM_INPUTS = 3
  )(
    input signed [SIG_SIZE-1:0] NEURON_IN[NUM_INPUTS-1:0],
    input signed [WEIGHT_SIZE-1:0] WEIGHTS_IN[NUM_INPUTS-1:0],
    input WB_EN,
    input signed [WEIGHT_SIZE-1:0] BIAS_IN,
    input CLOCK_N,
    output reg signed [SIG_SIZE-1:0] NEURON_OUT
  );
  logic signed [WEIGHT_SIZE-1:0] weights[NUM_INPUTS-1:0];
  logic signed [WEIGHT_SIZE-1:0] bias;
  logic signed [SIG_SIZE*2-1:0] ixw [NUM_INPUTS-1:0];
  logic signed [ADDITION_SIZE-1:0] addition;
  logic sigmoid_flag = 0;
  integer i;

  linear activation(.x(addition),
                    .y(NEURON_OUT));

  initial
  begin
    addition <= 0;
  end

  always @(negedge CLOCK_N)
  begin
    if(WB_EN)
    begin
      for(i = 0; i<NUM_INPUTS;i++)
      begin
        weights[i] <= WEIGHTS_IN[i];
      end
      bias <= BIAS_IN;
    end
    addition = bias;
    //multiply inputs by weights
```

```
    //add product to addition register
    for(i=0;i<NUM_INPUTS;i++)
    begin
      ixw[i] <= ((NEURON_IN[i]*weights[i])>>15);
      addition = addition + ixw[i];
      if(ixw[i] > 16'h7fff)
        addition = addition - 20'h10000;
    end
  end

endmodule
```

10.6 Sigmoid module

```
module SIGMOID_LUT(
    input [19:0] x [9:0],
    output reg signed [15:0] y [9:0]
  );

  reg signed [15:0] lut [(2**20)-1:0];

  initial
  begin
    $readmemh("sfp.mem",lut);
  end

  always@(x[0]) y[0] <= lut[x[0]];
  always@(x[1]) y[1] <= lut[x[1]];
  always@(x[2]) y[2] <= lut[x[2]];
  always@(x[3]) y[3] <= lut[x[3]];
  always@(x[4]) y[4] <= lut[x[4]];
  always@(x[5]) y[5] <= lut[x[5]];
  always@(x[6]) y[6] <= lut[x[6]];
  always@(x[7]) y[7] <= lut[x[7]];
  always@(x[8]) y[8] <= lut[x[8]];
  always@(x[9]) y[9] <= lut[x[9]];

endmodule
```

10.7 linear module

```
module linear(
    input [19:0] x,
    output reg signed [15:0] y
```

```
  );

  //4 MSB's are not used as input should never exceed 0xFFFF
  assign y = x[15:0];

endmodule
```

10.8 Delay module

```
module DELAY#(
    parameter SIG_SIZE = 16,
    parameter WEIGHT_SIZE = 16,
    parameter ADDITION_SIZE = 32,
    parameter NUM_INPUTS = 3
  )(
    input signed [SIG_SIZE-1:0]INPUT,
    input CLK_N,
    output reg signed [SIG_SIZE-1:0]DELAY_OUTPUT
  );

  reg signed [SIG_SIZE-1:0]TEMP_HOLD;

  always @(negedge CLK_N)
  begin
    DELAY_OUTPUT <= TEMP_HOLD;
    TEMP_HOLD <= INPUT;
  end

  initial
  begin
    DELAY_OUTPUT = 0;
    TEMP_HOLD = 0;
  end

endmodule
```

10.9 TDNN module

```
module TDNN#(
    parameter SIG_SIZE = 16,
    parameter WEIGHT_SIZE = 16,
    parameter ADDITION_SIZE = 20,
    parameter LAYER1_IN_SIZE = 10,
    parameter LAYER2_IN_SIZE = 10,
```

```
    parameter NUM_SIGS = 2
  )(
    input signed [SIG_SIZE-1:0] SIG_IN1,
    input signed [SIG_SIZE-1:0] SIG_IN2,
    input signed CLOCK_N,
    input signed wb_en,
    output signed [SIG_SIZE-1:0] SIG_OUT1,
    output signed [SIG_SIZE-1:0] SIG_OUT2
  );
  wire signed [SIG_SIZE-1:0] delay_out_1;
  wire signed [SIG_SIZE-1:0] delay_out_2;
  wire signed [SIG_SIZE-1:0] delay_out_3;
  wire signed [SIG_SIZE-1:0] delay_out_4;
  wire signed [SIG_SIZE-1:0] delay_out_5;
  wire signed [SIG_SIZE-1:0] delay_out_6;
  wire signed [SIG_SIZE-1:0] delay_out_7;
  wire signed [SIG_SIZE-1:0] delay_out_8;

  wire signed [ADDITION_SIZE-1:0] n1_out;
  wire signed [ADDITION_SIZE-1:0] n2_out;
  wire signed [ADDITION_SIZE-1:0] n3_out;
  wire signed [ADDITION_SIZE-1:0] n4_out;
  wire signed [ADDITION_SIZE-1:0] n5_out;
  wire signed [ADDITION_SIZE-1:0] n6_out;
  wire signed [ADDITION_SIZE-1:0] n7_out;
  wire signed [ADDITION_SIZE-1:0] n8_out;
  wire signed [ADDITION_SIZE-1:0] n9_out;
  wire signed [ADDITION_SIZE-1:0] n10_out;
  wire signed [SIG_SIZE-1:0] l1_sigmoid_out [9:0];
  wire signed [SIG_SIZE-1:0] n11_out;
  wire signed [SIG_SIZE-1:0] n12_out;

  reg signed [(WEIGHT_SIZE-1):0] WEIGHTS1 [(10*10)-1:0];
  reg signed [(WEIGHT_SIZE-1):0] WEIGHTS2 [(2*10)-1:0];

  wire signed [WEIGHT_SIZE-1:0] N1W[LAYER1_IN_SIZE-1:0];
  wire signed [WEIGHT_SIZE-1:0] N2W[LAYER1_IN_SIZE-1:0];
  wire signed [WEIGHT_SIZE-1:0] N3W[LAYER1_IN_SIZE-1:0];
  wire signed [WEIGHT_SIZE-1:0] N4W[LAYER1_IN_SIZE-1:0];
  wire signed [WEIGHT_SIZE-1:0] N5W[LAYER1_IN_SIZE-1:0];
  wire signed [WEIGHT_SIZE-1:0] N6W[LAYER1_IN_SIZE-1:0];
  wire signed [WEIGHT_SIZE-1:0] N7W[LAYER1_IN_SIZE-1:0];
  wire signed [WEIGHT_SIZE-1:0] N8W[LAYER1_IN_SIZE-1:0];
  wire signed [WEIGHT_SIZE-1:0] N9W[LAYER1_IN_SIZE-1:0];
  wire signed [WEIGHT_SIZE-1:0] N10W[LAYER1_IN_SIZE-1:0];
  wire signed [WEIGHT_SIZE-1:0] N11W[LAYER2_IN_SIZE-1:0];
  wire signed [WEIGHT_SIZE-1:0] N12W[LAYER2_IN_SIZE-1:0];


```

```
generate
  genvar i;
  for(i=0;i<10;i++)
  begin
    assign N1W[i] = WEIGHTS1[0+i];
    assign N2W[i] = WEIGHTS1[10+i];
    assign N3W[i] = WEIGHTS1[20+i];
    assign N4W[i] = WEIGHTS1[30+i];
    assign N5W[i] = WEIGHTS1[40+i];
    assign N6W[i] = WEIGHTS1[50+i];
    assign N7W[i] = WEIGHTS1[60+i];
    assign N8W[i] = WEIGHTS1[70+i];
    assign N9W[i] = WEIGHTS1[80+i];
    assign N10W[i] = WEIGHTS1[90+i];
    assign N11W[i] = WEIGHTS2[0+i];
    assign N12W[i] = WEIGHTS2[10+i];
  end
endgenerate

reg signed [WEIGHT_SIZE-1:0] bias1 [9:0]; //Layer 1 bias
reg signed [WEIGHT_SIZE-1:0] bias2 [1:0]; //Layer 2 bias

//Delays for Sig_in_1
DELAY delay1(SIG_IN1,CLOCK_N,delay_out_1);
DELAY delay2(delay_out_1,CLOCK_N,delay_out_2);
DELAY delay3(delay_out_2,CLOCK_N,delay_out_3);
DELAY delay4(delay_out_3,CLOCK_N,delay_out_4);

//Delays for Sig_in_2
DELAY delay5(SIG_IN2,CLOCK_N,delay_out_5);
DELAY delay6(delay_out_5,CLOCK_N,delay_out_6);
DELAY delay7(delay_out_6,CLOCK_N,delay_out_7);
DELAY delay8(delay_out_7,CLOCK_N,delay_out_8);

//Layer 1
NEURON #(.NUM_INPUTS(LAYER1_IN_SIZE)) neuron1(
         .NEURON_IN(
           {delay_out_8, delay_out_4, delay_out_7, delay_out_3, delay_out_6,
            delay_out_2, delay_out_5, delay_out_1, SIG_IN2, SIG_IN1}),
         .WEIGHTS_IN(N1W),
         .WB_EN(wb_en),
         .BIAS_IN(bias1[0]),
         .CLOCK_N(CLOCK_N),
         .NEURON_OUT(n1_out));
NEURON #(.NUM_INPUTS(LAYER1_IN_SIZE)) neuron2(
         .NEURON_IN({delay_out_8, delay_out_4, delay_out_7, delay_out_3, delay_out_6,
                     delay_out_2, delay_out_5, delay_out_1, SIG_IN2, SIG_IN1}),
         .WEIGHTS_IN(N2W),
         .WB_EN(wb_en),
```

```
105             .BIAS_IN(bias1[1]),
106             .CLOCK_N(CLOCK_N),
107             .NEURON_OUT(n2_out));
108     NEURON #(.NUM_INPUTS(LAYER1_IN_SIZE)) neuron3(
109             .NEURON_IN({delay_out_8, delay_out_4, delay_out_7, delay_out_3, delay_out_6,
110                         delay_out_2, delay_out_5, delay_out_1, SIG_IN2, SIG_IN1}),
111             .WEIGHTS_IN(N3W),
112             .WB_EN(wb_en),
113             .BIAS_IN(bias1[2]),
114             .CLOCK_N(CLOCK_N),
115             .NEURON_OUT(n3_out));
116     NEURON #(.NUM_INPUTS(LAYER1_IN_SIZE)) neuron4(
117             .NEURON_IN({delay_out_8, delay_out_4, delay_out_7, delay_out_3, delay_out_6,
118                         delay_out_2, delay_out_5, delay_out_1, SIG_IN2, SIG_IN1}),
119             .WEIGHTS_IN(N4W),
120             .WB_EN(wb_en),
121             .BIAS_IN(bias1[3]),
122             .CLOCK_N(CLOCK_N),
123             .NEURON_OUT(n4_out));
124     NEURON #(.NUM_INPUTS(LAYER1_IN_SIZE)) neuron5(
125             .NEURON_IN({delay_out_8, delay_out_4, delay_out_7, delay_out_3, delay_out_6,
126                         delay_out_2, delay_out_5, delay_out_1, SIG_IN2, SIG_IN1}),
127             .WEIGHTS_IN(N5W),
128             .WB_EN(wb_en),
129             .BIAS_IN(bias1[4]),
130             .CLOCK_N(CLOCK_N),
131             .NEURON_OUT(n5_out));
132     NEURON #(.NUM_INPUTS(LAYER1_IN_SIZE)) neuron6(
133             .NEURON_IN({delay_out_8, delay_out_4, delay_out_7, delay_out_3, delay_out_6,
134                         delay_out_2, delay_out_5, delay_out_1, SIG_IN2, SIG_IN1}),
135             .WEIGHTS_IN(N6W),
136             .WB_EN(wb_en),
137             .BIAS_IN(bias1[5]),
138             .CLOCK_N(CLOCK_N),
139             .NEURON_OUT(n6_out));
140     NEURON #(.NUM_INPUTS(LAYER1_IN_SIZE)) neuron7(
141             .NEURON_IN({delay_out_8, delay_out_4, delay_out_7, delay_out_3, delay_out_6,
142                         delay_out_2, delay_out_5, delay_out_1, SIG_IN2, SIG_IN1}),
143             .WEIGHTS_IN(N7W),
144             .WB_EN(wb_en),
145             .BIAS_IN(bias1[6]),
146             .CLOCK_N(CLOCK_N),
147             .NEURON_OUT(n7_out));
148     NEURON #(.NUM_INPUTS(LAYER1_IN_SIZE)) neuron8(
149             .NEURON_IN({delay_out_8, delay_out_4, delay_out_7, delay_out_3, delay_out_6,
150                         delay_out_2, delay_out_5, delay_out_1, SIG_IN2, SIG_IN1}),
151             .WEIGHTS_IN(N8W),
152             .WB_EN(wb_en),
153             .BIAS_IN(bias1[7]),
```

```
            .CLOCK_N(CLOCK_N),
            .NEURON_OUT(n8_out));
  NEURON #(.NUM_INPUTS(LAYER1_IN_SIZE)) neuron9(
            .NEURON_IN({delay_out_8, delay_out_4, delay_out_7, delay_out_3, delay_out_6,
                        delay_out_2, delay_out_5, delay_out_1, SIG_IN2, SIG_IN1}),
            .WEIGHTS_IN(N9W),
            .WB_EN(wb_en),
            .BIAS_IN(bias1[8]),
            .CLOCK_N(CLOCK_N),
            .NEURON_OUT(n9_out));
  NEURON #(.NUM_INPUTS(LAYER1_IN_SIZE)) neuron10(
            .NEURON_IN({delay_out_8, delay_out_4, delay_out_7, delay_out_3, delay_out_6,
                        delay_out_2, delay_out_5, delay_out_1, SIG_IN2, SIG_IN1}),
            .WEIGHTS_IN(N10W),
            .WB_EN(wb_en),
            .BIAS_IN(bias1[9]),
            .CLOCK_N(CLOCK_N),
            .NEURON_OUT(n10_out));

  //Layer 1 Activation
  SIGMOID_LUT sigAct(.x({n10_out,n9_out,n8_out,n7_out,n6_out,n5_out,n4_out,n3_out,n2_out,n1_out}),
                     .y(l1_sigmoid_out));

  //Layer 2
  NEURON_LIN #(.NUM_INPUTS(LAYER1_IN_SIZE)) neuron11(.NEURON_IN(l1_sigmoid_out),
              .WEIGHTS_IN(N11W),
              .WB_EN(wb_en),
              .BIAS_IN(bias2[0]),
              .CLOCK_N(CLOCK_N),
              .NEURON_OUT(SIG_OUT1));
  NEURON_LIN #(.NUM_INPUTS(LAYER1_IN_SIZE)) neuron12(.NEURON_IN(l1_sigmoid_out),
              .WEIGHTS_IN(N12W),
              .WB_EN(wb_en),
              .BIAS_IN(bias2[1]),
              .CLOCK_N(CLOCK_N),
              .NEURON_OUT(SIG_OUT2));

  initial
  begin
    $readmemh("bias1.mem", bias1);
    $readmemh("weights1.mem", WEIGHTS1);
    $readmemh("bias2.mem", bias2);
    $readmemh("weights2.mem", WEIGHTS2);
  end

endmodule
```

10.10 TDNN testbench

```
module TDNN_TB();
    reg signed [15:0] sig1, sig2;
    reg topclk;
    reg WEIGHTBIAS_EN;
    wire signed [15:0] out1, out2;

    //design under test instanstiation
    TDNN DUT(   .SIG_IN1(sig1),
                .SIG_IN2(sig2),
                .CLOCK_N(topclk),
                .SIG_OUT1(out1),
                .SIG_OUT2(out2),
                .wb_en(WEIGHTBIAS_EN)
                );

    //toggle clock every 5 sim cycles
    always #5 topclk = ~topclk;

    //dump signals to a dumpfile
    initial begin
        $dumpfile("dump.vcd");
        $dumpvars();
    end

    initial begin
        //initialise values of registers
        topclk = 0;
        sig1 = 0;
        sig2 = 0;
        WEIGHTBIAS_EN = 0;
        #10
        //enable in order to set values of Weight &
        //Bias registers
        WEIGHTBIAS_EN = 1;
        //allow time for weight and bias values to set
        #10
        WEIGHTBIAS_EN = 0;
        #80
        //run through 10 signal inputs
        //to be compared to keras output
        sig1 = 16'hb99a;
        sig2 = 16'he28f;
        #10
        sig1 = 16'h35c3;
        sig2 = 16'h7c28;
        #10
```

```
        sig1 = 16'h6e14;
        sig2 = 16'hd47b;
        #10
        sig1 = 16'h8148;
        sig2 = 16'ha000;
        #10
        sig1 = 16'h4f5c;
        sig2 = 16'hf0a3;
        #10
        sig1 = 16'h7eb8;
        sig2 = 16'h6e14;
        #10
        sig1 = 16'hce14;
        sig2 = 16'he51e;
        #10
        sig1 = 16'hae15;
        sig2 = 16'h628f;
        #10
        sig1 = 16'h747a;
        sig2 = 16'hf999;
        #10
        sig1 = 16'h0f5d;
        sig2 = 16'hea3d;
        //wait 30 sim cycles at the end
        //system has 2 clock cycle delay between input and output
        #30
        sig1 = 0;
        sig2 = 0;
        #10

        $finish;
    end

endmodule
```

10.11 Top level module

```
module TDNN_TOP#(
    parameter SIG_SIZE = 16,
    parameter WEIGHT_SIZE = 16,
    parameter ADDITION_SIZE = 32,
    parameter NUM_INPUTS = 3,
    parameter NUM_SIGS = 2
  )(
    // Slide switch inputs
    input [15:0]sw,
```

```
    input topclk,
    output [6:0] seg,
    output [7:0] an,
    output dp
  );
  reg WB_ENABLE;
  wire [15:0] sig_out1, sig_out2;

  TDNN TDNN_DES(.SIG_IN1(sw),
                .SIG_IN2(sw),
                .CLOCK_N(topclk),
                .wb_en(WB_ENABLE),
                .SIG_OUT1(sig_out1),
                .SIG_OUT2(sig_out2));

  Seg7 segOut(.in1(sig_out2),
              .in2(sig_out1),
              .topclk(topclk),
              .seg(seg),
              .an(an),
              .dp(dp));

  initial
    WB_ENABLE = 1;

endmodule
```

10.12 7 segment display module

```
module Seg7(
    input [15:0] in1,
    input [15:0] in2,
    input topclk,
    output reg [6:0] seg,
    output reg [7:0] an,
    output reg dp
  );
  reg [3:0] seg_in;
  reg [2:0] state;
  reg [16:0] timer;
  assign dp = 1;



  seg7Decode(seg_in, seg);

```

```
18  always@(posedge topclk)
19  begin
20    if(timer  == 99_999)
21    begin
22      timer <= 0;
23      case(state)
24        0:
25        begin
26          seg_in <= in1[3:0];
27          an <= 8'b11111110;
28        end
29        1:
30        begin
31          seg_in <= in1[7:4];
32          an <= 8'b11111101;
33        end
34        2:
35        begin
36          seg_in <= in1[11:8];
37          an <= 8'b11111011;
38        end
39        3:
40        begin
41          seg_in <= in1[15:12];
42          an <= 8'b11110111;
43        end
44        4:
45        begin
46          seg_in <= in2[3:0];
47          an <= 8'b11101111;
48        end
49        5:
50        begin
51          seg_in <= in2[7:4];
52          an <= 8'b11011111;
53        end
54        6:
55        begin
56          seg_in <= in2[11:8];
57          an <= 8'b10111111;
58        end
59        7:
60        begin
61          seg_in <= in2[15:12];
62          an <= 8'b01111111;
63        end
64      endcase
65      state <= state+1;
66    end
```

```
    else
      timer <= timer+1;
  end

endmodule


module seg7Decode(
    input  [3:0]x,
    output reg [6:0]z
  );
  always @*
  case (x)
    4'b0000 :              //Hexadecimal 0
      z = 7'b1000000;
    4'b0001 :                   //Hexadecimal 1
      z = 7'b1111001;
    4'b0010 :                  // Hexadecimal 2
      z = 7'b0100100;
    4'b0011 :                  // Hexadecimal 3
      z = 7'b0110000;
    4'b0100 :                 // Hexadecimal 4
      z = 7'b0011001;
    4'b0101 :                 // Hexadecimal 5
      z = 7'b0010010;
    4'b0110 :                 // Hexadecimal 6
      z = 7'b0000010;
    4'b0111 :                 // Hexadecimal 7
      z = 7'b1111000;
    4'b1000 :                      //Hexadecimal 8
      z = 7'b0000000;
    4'b1001 :                   //Hexadecimal 9
      z = 7'b0010000;
    4'b1010 :                  // Hexadecimal A
      z = 7'b0001000;
    4'b1011 :                  // Hexadecimal B
      z = 7'b0000011;
    4'b1100 :                 // Hexadecimal C
      z = 7'b1000110 ;
    4'b1101 :                 // Hexadecimal D
      z = 7'b0100001;
    4'b1110 :                 // Hexadecimal E
      z = 7'b0000110;
    4'b1111 :                 // Hexadecimal F
      z = 7'b0001110;
  endcase

endmodule
```

10.13 Github Repository

All files related to this report can be found in the Github repository at the following link:
`https://github.com/MatthewPicchu/TDNN-predis-MM23`

Health and Safety Risk Assessment and Guidelines.

Every project student or group of students must perform a health and safety risk assessment of their project before commencing any work.

This should be done in consultation with their project supervisor. The Risk Assessment form must be assessed, along with any required material such as Material Safety Data Sheets, and signed off by a member of the Health and Safety Committee.

The form is designed to identify any hazards that the proposed project might present to the student or people that may come into contact with the project.

Issues to consider include

- **Hazardous materials, for example flammable materials, epoxies, cements, paints, aerosols, carcinogens, poisons, toxins…**
- **Working alone, at height, in confined space, in a dangerous area**
- **Working with electricity**
- **Working with machines and tools**
- **Working with bio materials**
- **Working with heat or flame (e.g. soldering or hot air guns)**
- **Working with lasers**
- **Working with heavy or large items**
- **Working with fast spinning objects (e.g. fans or propellers)**
- **Disposable / Recycling of hazardous materials eg batteries, plastics, metal and those mentioned above.**

Where risks are identified, the student must discuss how the risk will be managed by listing what controls are in place and what further controls may be required.

The Risk Assessment Form should be included in your project report.

The Risk Assessment Form should be available for when requested by any staff member.

If this form is NOT available, then all work on the hardware aspects of your project MUST STOP until the form is signed off.

H&S committee members are: Mr. Andrew Meehan, Mr. John Maloco, Ms. Ann Dempsey, and Dr Bryan Hennelly.

I confirm, I have no medical condition that impedes working on my Project, and I will advise a Health & Safety Officer if this status changes.

Signed: Matthew Morrow

Department of Electronic Engineering, Maynooth University

PROJECT RISK ASSESSMENT FORM Rev. 3

NAME AND STUDENT NUMBER: **Matthew Morrow 18427114**	PROJECT NAME: **FPGA Implementation of Parallel Processor**
SUPERVISOR: **John Dooley**	PROJECT LOCATION: **Laboratory**

BRIEF DESCRIPTION OF PROJECT

Reducing the length of time that it takes for a calculation requires additional effort in planning how that function will be mapped to the hardware. This project will explore the ways in which polynomial functions are performed on FPGAs.

Hazards, Risk [High(H) Medium (M) Low (L)], and Control Measures

HAZARD	Risk	Controls
Ever present risk of accidents due to lack of vigilance	L	• No Smoking, eating, drinking • No one should be alone • Pathways should be kept clear of obstacles • Accidents must be reported • Be familiar with locations of safety equipment
Fire risk – damage of property or harm to people	L	• Ensure Electronics are connected and used correctly.
Electrical Risk – Electrocution, Burns, damage to property	L	• Ensure electronics are grounded correctly • Ensure Cables do not cross walkways • Ensure there are no exposed wires • Do not use damaged equipment
Personal Health	L	• Take regular breaks • Do not use computer screen for extended periods • Ensure correct seating position as to not cause long-term physical pain

Identified risks should be discussed with your supervisor and a safe system of work agreed. A more in-depth risk assessment may be required after initial review. Do not proceed until form is signed off.

WEEE Compliance & Recycling: Indicate below how you intend to manage and dispose of your project's material during and post project work.

Further Controls Required

SIGNATURE OF STUDENT: Matthew Morrow DATE: 27/10/2022

SIGNATURE OF SUPERVISOR: _____ DATE: _____

DEPT HEALTH AND SAFTEY COMMITTEE MEMBER: John Mahon DATE: 09/12/22