Implementation of Artificial neural network on Zynq-7000 SoC

Project Report

Submitted in partial fulfilment of the course EEE F366: Laboratory Project

By
Parth Kalgaonkar
ID No.: 2016A3PS0268P

Under the supervision of
Dr. Meetha V. Shenoy

Birla Institute of Technology and Science, Pilani, Pilani Campus
December 8, 2019

Contents

1 Introduction **2**

2 Implementation of a single neuron **2**
- 2.1 Fixed point represention of numbers . . . 2
- 2.2 Implementation of activation function . . . 2
  - 2.2.1 Taylor series expansion . . . 3
  - 2.2.2 Elliott approximations . . . 3
  - 2.2.3 Look-Up table based approach . . . 3

3 Integration of all neurons into complete network **4**
- 3.1 Processing system and system level signals . . . 4
- 3.2 Weights and bias memory . . . 6
- 3.3 Control logic . . . 6

4 Implementation of software interface **6**
- 4.1 Implementation of the UART interface . . . 7

5 Levenberg-Marquardt training on Zynq PS **7**
- 5.1 A brief Introduction to LM training . . . 7
- 5.2 Implementation of LM algorithm on Zynq PS . . . 9

6 Results **10**

7 Scope for future improvements **10**

List of Figures

- 1 Architecture of a single neuron . . . 2
- 2 Elliott approximation vs tanh . . . 3
- 3 Block diagram of neural network . . . 5
- 4 Typical application without training . . . 8
- 5 Flowchart for the training algorithm . . . 9

List of Tables

- 1 Resource utilization by neural network . . . 10
- 2 Time taken by neural network in various processes . . . 10
- 3 Power report summary . . . 10

1 Introduction

The need for localization of objects within closed environments is extremely widespread. Ultrasonic beacons are typically used for such applications. Translation of beacon data to spatial co-ordinates must be done in real time. Neural networks are used to approximate the non-linear relationship between delays and postions. Such a network must be implemented on a low power system with very low latency for real time application. An FPGA was chosen to implement the neural net. Additionally such systems must adapt to changing environments. To enable this, training must also be done on the board.

A Xilinx 7000 series SoC was chosen for this application. The on board DSP resources were used to implement multiplication operation on the programmable logic. The multiple available BRAM resource were also used to implement the tansigmoid transfer functions.

2 Implementation of a single neuron

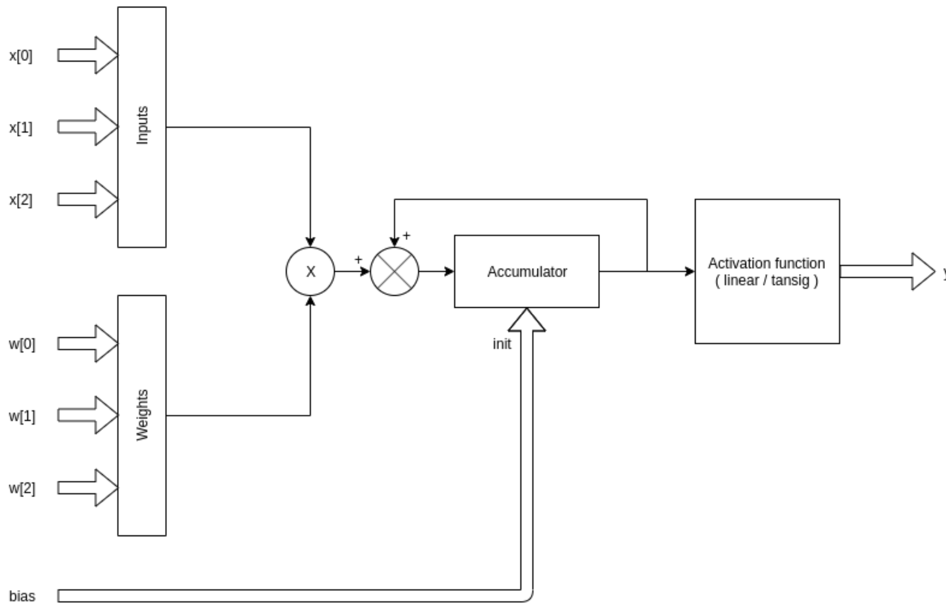


Figure 1: Architecture of a single neuron

2.1 Fixed point represention of numbers

The inputs, weights and biases of the network are real numbers and are represented in the hardware using fixed point notation. The number of bits used has the most impact on the accuracy of the system.

Inputs and outputs can be normalized to values between -1 and 1. Representation of these is done in the hardware using 11 bit 2's complement numbers with 10 bits for fractional part and 1 sign bit. Whole number part is not required as inputs and outputs are assumed to be normalized.

Weights and biases can be any real numbers. Thus they are represented using 16 bit 2's complement numbers with 8 bits for fractional part.

Component	Resolution
Inputs/Outputs	1/1024
Weights/Biases	1/256

2.2 Implementation of activation function

The first layer of the network uses the Tansigmoid activation function.

$$tansig(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{1}$$

Implementing this on hardware poses a significant challenge as analytical computation would be extremely slow and hardware intensive. There are three methods that are usually used to implement this activation function. [3]

2.2.1 Taylor series expansion

The taylor series expansion for e^u can be given as:

$$e^u = 1 + u + \frac{u^2}{2!} + \frac{u^3}{3!} \cdots + \frac{u^n}{n!} + \cdots \tag{2}$$

Some implementations use the first few terms of the taylor series expansion to approximate *tansig*. An implementation using the first 5 terms is proposed by *Koyuncu*[3]. Such an implementation, however, is extremely complex and redundant for a simple system.

2.2.2 Elliott approximations

Elliott-93 approximation is a signum function that ranges between -1 and 1 and is given by:

$$tansig_{e93}(x) = \frac{x}{1 + |x|} \tag{3}$$

Another approximation is the Elliott-2 approximation which was developed from Elliott-93 and is given by:

$$tansig_{e2}(x) = sgn(x) * \frac{x^2}{1 + x^2} \tag{4}$$

Here $sgn(x)$ is the signum function.

Both of these approximations yield poor results as can be seen from the following comparison between the approximations and the exact function.

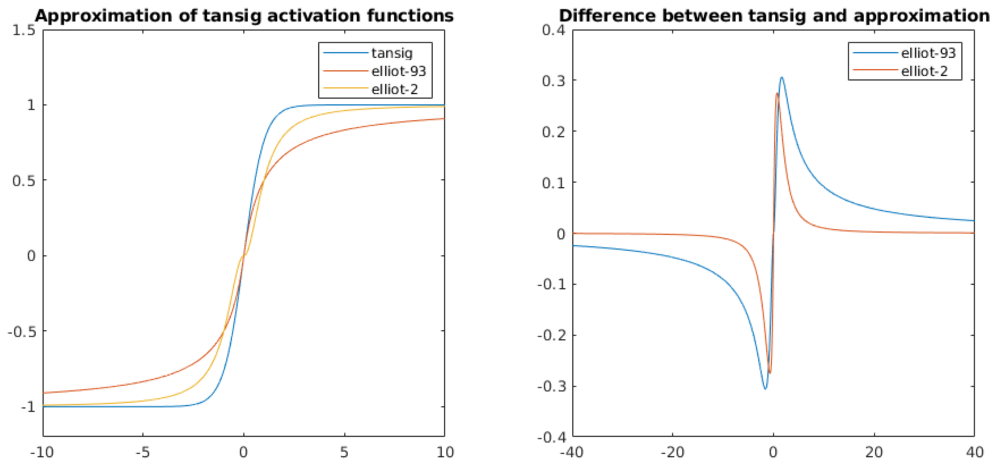


Figure 2: Elliott approximation vs tanh

Additionally, both approximations flatten out only for much larger inputs. Thus, their behavior is not as local as *tansig*.

2.2.3 Look-Up table based approach

Any function can be easily represented as a look-up table (LUT) of function values at various sample points. The LUT can then be stored in a BRAM. Then finding the value of the activation function becomes as simple as reading a value from the RAM at the specific address. The number of samples must be decided as this determines the size of the RAM required.

Too few samples will lead to very high sampling noise. Too many samples will make the size of the RAM required very high and implementaion on the limited resources on the board will become

impossible. Thus it is very important to find the optimum number of samples. A method for this is suggested by *Himavathi et.al.*[2]

1. Let n be the number bits in output. In our case we have 10 fractional bits in the output (See 2.1).

$$\therefore n = 10 \tag{5}$$

2. Determine the range of samples required to cover full output range.

$$\begin{aligned} y_{min} &= -1 + 2^{-n} \\ &= \frac{-1023}{1024} \\ y_{max} &= 1 - 2^{-n} \\ &= \frac{1023}{1024} \end{aligned} \tag{6}$$

$$\begin{aligned} \therefore x_{min} &= \tanh^{-1}(y_{min}) \\ &= \tanh^{-1}(\frac{-1023}{1024}) \approx -3.8121 \\ \therefore x_{max} &= -x_{min} \approx 3.8121 \end{aligned}$$

3. Determine the point of maximum slope.

$$\frac{d\tanh(x)}{dx} = sech^2(x) \tag{7}$$

The slope is maximum at $x = 0$ and is equal to 1.

4. Smallest change in output is given by $\delta y = 2^{-10}$. The smallest step in input δx to cause a change of δy in output at the point of maximum slope is given by:

$$\delta x = \frac{\delta y}{\frac{d\tanh(x)}{dx}|_{x=0}} = \frac{1}{1024} \tag{8}$$

5. Number of samples can now be calculated as:

$$(LUT)_{min} = \frac{x_{max} - x_{min}}{\delta x} \approx 7807 \tag{9}$$

6. Number of bits of fractional part is determined by δx. In this case, 10 bits of fractional part are enough for a step size of 2^{-10}.

7. Atleast 13 bits are required to address a memory with 7807 locations. This design uses a memory with 14 bit addresses (ie 16384 locations) to allow for increase in the resolution of outputs in the future.

3 Integration of all neurons into complete network

This section describes the complete structure of the network on the FPGA. Fig 3 shows a top level block diagram of the system.

3.1 Processing system and system level signals

The Processing (PS) system is responsible for training the network on board when required. The capabilities of the hardware on the programmable logic (PL) are exposed to the outside world via UART. The PS recieves all data and control from UART and takes the necessary actions. The system level clock and reset signals are also provided by the PS. Functioning of PS is discussed further in 4

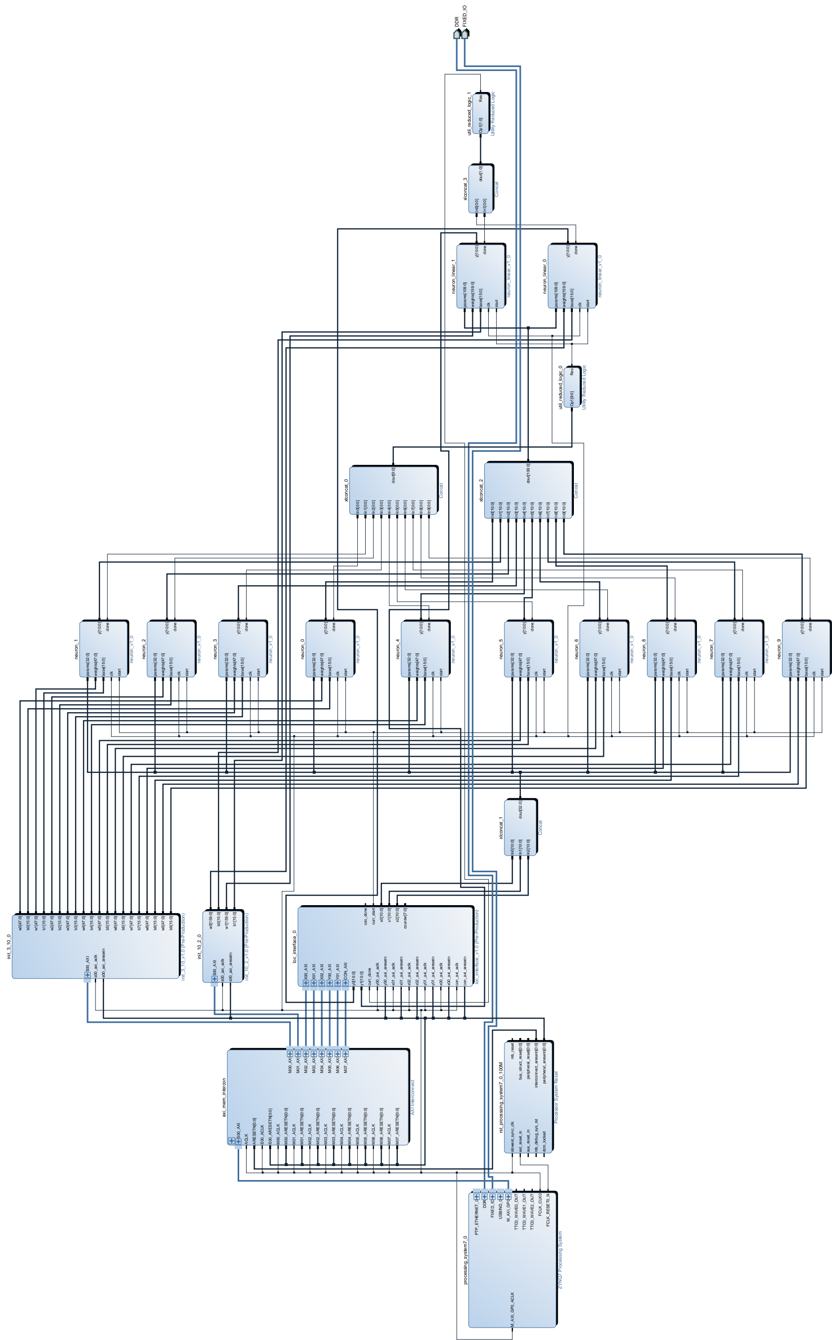


Figure 3: Block diagram of neural network

3.2 Weights and bias memory

These act as buffers, one for each layer, where values of weights and biases for the layer are stored. At the beginning of every transaction, each neuron reads these values into it's local registers. These blocks are connected to the PS through an AXI Full (memory-mapped) interface. Weights and biases can be written or read in burst for faster operation through DMA resources.

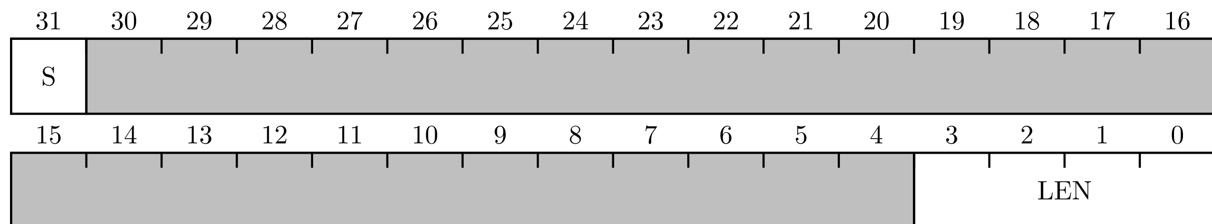
3.3 Control logic

This block acts as the interface between the PS and the network on the PL. The block is made up of 6 AXI interfaces as described below.

1. X00_AXI, X01_AXI and X02_AXI are three AXI-Full interfaces for each of the three inputs with a maximum burst length of 16. Starting at the base address for each interface, 11 bit inputs (sign extended to 32-bits) can be written in consecutive memory locations. Inputs for one transaction should be written at the same offsets.

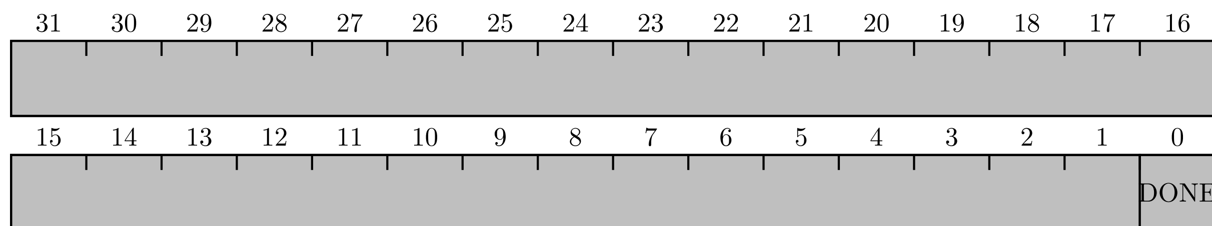
2. Y00_AXI and Y01_AXI are two AXI-Full interfaces for each of the two outputs. Outputs of the network are not sign extened but are padded with zeroes to make the output size 32 bits. Read bursts can be of maximum length 16.

3. CON_AXI is an AXI-Lite interface for control information. It exposes functionality in form of two 32-bit registers.



CON_AXI_slv_reg0: Control register (Base + 0x00)

First is a control register. only 5 bits of this register are significant. `LEN` is the number of transactions to be done. `S` is the start bit. Writing a 1 to this bit starts the operation of the neural network. Programmer must make sure that values are written in the X00, X01, and X02 interfaces before starting the neural net. Any pending transactions are silently terminated.



CON_AXI_slv_reg1: Status register (Base + 0x04)

Status register is the second 32 bit register. Bit 0 in this register is set to 1 when the last burst of transactions has completed. Other bits can be used to signinfy other status data in future.

4 Implementation of software interface

The software running on the Processing system has 3 responsibilities.

1. Getting input data from UART, sending it to the neural network, getting the network outputs back and sending it back through the UART interface in form of UTF-8 encoded decimal numbers.

2. Setting the weights of the network layers to values recieved from UART.
3. Training the neural network using the Levenberg-Marquardt training algorithm to a given set of inputs and targets.

4.1 Implementation of the UART interface

Every transaction on the UART interface is initiated by the controlling device. The system and Zynq PS act as a slave.

Every transaction begins with a single integer (in `UTF-8` encoding) to indicate the mode of operation. Currently, the following modes have been implemented.

Mode `'-1'`: Update the weights for layer 1. The system expects 40 whitespace seperated integers to follow. The system will wait for these silently and indefinitely.

Mode `'-2'`: Update the weights for layer 2. The system expects 22 whitespace seperated integers to follow. The system will wait for these silently and indefinitely.

Mode `'0'`: Retrain the network to a new set of inputs and targets. This is discussed in greater detail in section 5

Default : Give a set of inputs for the network to compute. The value of the command word is assumed to be the number of inputs to be computed. This number must be positive and less than 16. The system then expects those inputs in form of triplets of comma seperated integers (in the format `" %d, %d, %d\n"`).

Figure 4 shows a UML diagram that describes a typical application where the network has been pre-trained:

5 Levenberg-Marquardt training on Zynq PS

5.1 A brief Introduction to LM training

The Levenberg-Marquardt algorithm provides a numerical solution to minimizing a non-linear function. Most commonly used method for neural networks training is the steepest descent algorithm, also known as the error backpropagation algorithm(EBP). It is widely known for being inefficient due to it's slow convergence.

Other second order algorithms such as the Gauss-Newton algorithm improve upon the speed of EBP by more This method is only valid when the quadratic approximation of the error function is valid. Otherwise, GN is mostly divergent.

The basic idea of the Levenberg-Marquardt algorithm is to use a combined training process around a complex error surface. Far away from the minima, the algorithm behaves similar to the steepest descent algorithm. But as it approaches the target, it approximately becomes the GN algorithm which can result in a great speed up in convergence.[1]

$$\boldsymbol{e} = \begin{bmatrix} e_{1,1} \\ e_{1,2} \\ \vdots \\ e_{1,M} \\ \vdots \\ e_{P,M} \end{bmatrix} \tag{10}$$

where $e_{i,j}$ is the error in the ith output in the jth set of input parameters and targets.

The algorithm functionality is based on the Jacobian matrix like Gauss-Newton. The jacobian matrix is given by:

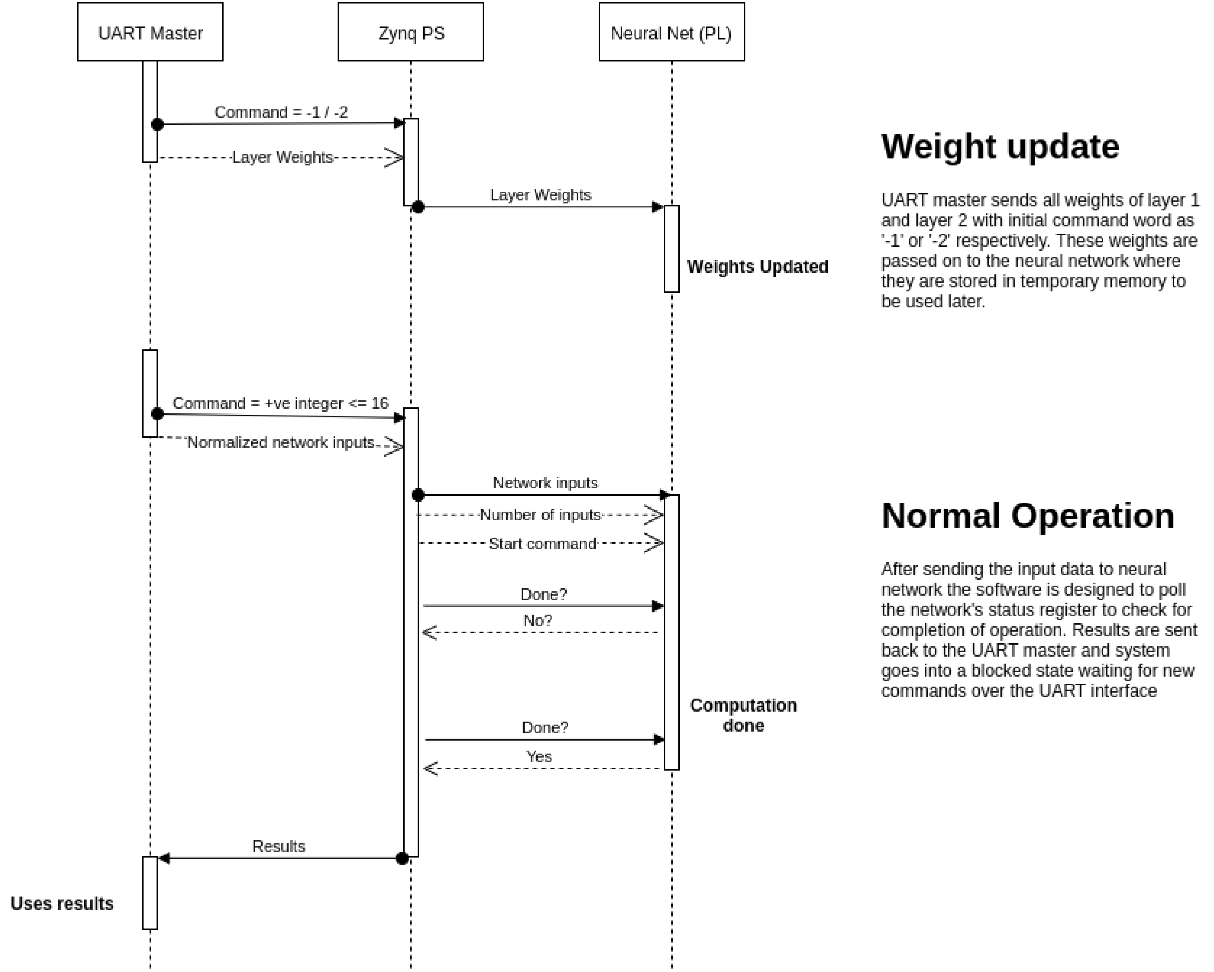


Figure 4: Typical application without training

$$\boldsymbol{J} = \begin{bmatrix} \frac{\partial e_{1,1}}{\partial W_1} & \frac{\partial e_{1,1}}{\partial W_2} & \cdots & \frac{\partial e_{1,1}}{\partial W_N} \\ \frac{\partial e_{1,2}}{\partial W_1} & \frac{\partial e_{1,2}}{\partial W_2} & \cdots & \frac{\partial e_{1,2}}{\partial W_N} \\ \vdots & \vdots & & \vdots \\ \frac{\partial e_{P,M}}{\partial W_1} & \frac{\partial e_{P,M}}{\partial W_2} & \cdots & \frac{\partial e_{P,M}}{\partial W_N} \end{bmatrix} \tag{11}$$

where N is the total number of weights and biases in the neural net.

The weight update equation then becomes

$$\boldsymbol{W_{k+1}} = \boldsymbol{W_k} - (\boldsymbol{J}^T\boldsymbol{J} + \mu\boldsymbol{I})^{-1}\boldsymbol{J}^T\boldsymbol{e} \tag{12}$$

where $\boldsymbol{W_k}$ is a column vector of all weights at the kth iteration.

When the combination coefficient μ is very small, equation 12 approaches the weight update equation for the Gauss-Newton method. When μ is very large, it approximates the EBP algorithm with the equivalent learning rate α given by:

$$\alpha = \frac{1}{\mu} \tag{13}$$

The combination coefficient μ is adapted as the training progresses. For every successful iteration, μ is decreased so the system approaches a Gauss-Newton approximation and vice-versa.

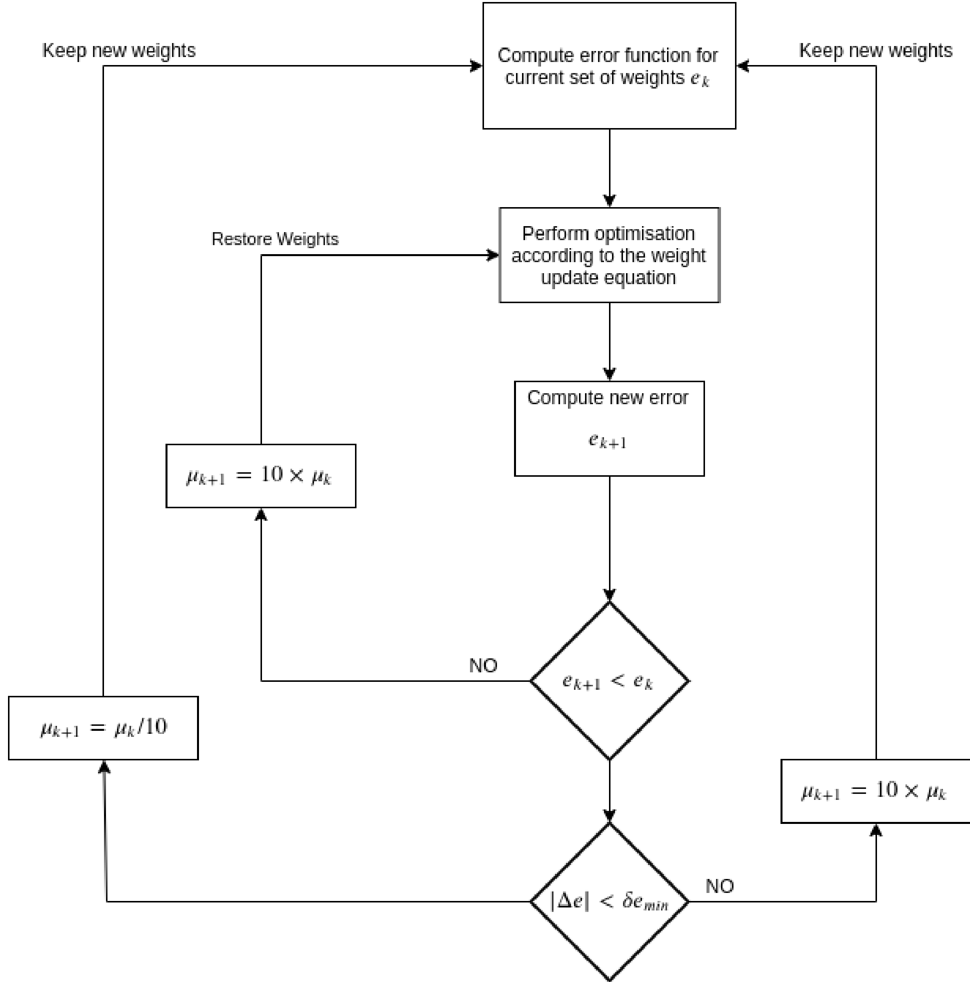


Figure 5: Flowchart for the training algorithm

5.2 Implementation of LM algorithm on Zynq PS

In order to avoid communication overheads between the PS and the PL during the training phase, a digital twin of the neural network is created using software. This twin reads the initial weights and biases back from the neural network. Training is performed on the twin and the final values of the weights are then written back to the hardware on the PL.

After the initial command of `0`, the system expects an integer from UART to indicate the number of data points for which training is to be performed. Following this, the training inputs and targets are expected as quintuplets of comma seperated values in the form `" %d, %d, %d, %d, %d"`. Here, the last two integers are the targets and the first three integers are the inputs. All inputs and targets must be scaled up by a factor of `1024` and rounded down to the nearest integers.

Any values greater than `1023` and less than `-1024` are invalid and result in undefined behaviour as inputs and targets are assumed to be normalized to a value between `-1` and `1`.

All neurons are simulated in software as structures containing their respective weights. These neurons are used first in a forward propagation to compute each activation which is required in the back-propagation stage. The backpropagation phase computes the jacobian term for each weight of the neurons. These terms are then collected to form the Jacobian matrix.

An open source library by Martensson [4] is used for the Matrix operations on the Zynq PS. This is a wrapper of the LAPACK library [5] designed keeping the low memory requirements of embedded systems in mind. It typically works upto 7 times faster compared to traditional matrix computational systems such as Octave.

6 Results

Table 1 shows the resource utilization by the neural network on the programmable logic. Table 2 reports the time taken by the neural network and the PS interface for various activities. Table 3 reports the expected power consumption of various design components.

Resource	Utilization	Available	Utilization(%)
FF	4939	106400	4.641917
LUT	4085	53200	7.678571
Memory LUT	167	17400	0.9597701
BRAM	59	140	42.142857
DSP48	12	220	5.4545455
BUFG	1	32	3.125

Table 1: Resource utilization by neural network

Process	Time taken
Weight update (Layer 1)	≈15ms
Weight update (Layer 2)	≈8ms
Training (verbose)	≈2s
Training (non-verbose)	≈1.7s
Simulation (single)	≈1.5ms
Simulation (burst of 15)	≈28ms

Table 2: Time taken by neural network in various processes

On-Chip	Power (W)	Used	Utilization (%)
Clocks	0.016	3	—
Slice Logic	0.017	10623	—
LUT as Logic	0.016	3918	7.36
Register	<0.001	4939	4.64
CARRY4	<0.001	174	1.31
LUT as Distributed RAM	<0.001	99	0.57
F7/F8 Muxes	<0.001	288	0.54
LUT as Shift Register	<0.001	68	0.39
Others	0.000	342	—
Signals	0.034	8215	—
Block RAM	0.064	59	42.14
DSPs	0.013	12	5.45
PS7	1.529	1	—
Static Power	0.167	—	—
Total	1.840	—	—

Table 3: Power report summary

7 Scope for future improvements

1. Using more of the available Multiplication resources, design of single neurons can be completely pipelined in order to improve performance many fold.

2. PS interface can be improved to use DMA resources so that AXI burst functionality can be better utilized.

3. PS interface can be modified to accept interrupts from the network. The current interrupt sent by the neural network is ignored in favour of simpler design.

4. AXI slave interface can be modified to allow for much larger bursts of data.

References

[1] Yu Hao and BM. Wilamowski. Levenberg-marquardt training. *Industrial electronics handbook 5*, 12, 2011.

[2] S. Himavathi, D. Anitha, and A. Muthuramalingam. Feedforward neural network implementation in fpga using layer multiplexing for effective resource utilization. *IEEE Transactions on Neural Networks*, 18(3):880–888, May 2007.

[3] Ismail Koyuncu. Implementation of high speed tangent sigmoid transfer function approximations for artificial neural network applications on FPGA. *Advances in Electrical and Computer Engineering*, 18:79–86, Aug 2018.

[4] Daniel Martensson. EmbeddedLAPACK. `github.com/DanielMartensson/EmbeddedLapack`, Feb 2019.

[5] Lapack team. LAPACK. `www.netlib.org/lapack`, 2000 - 2019.