Implementation of Efficient Multilayer Perceptron ANN Neurons on Field Programmable Gate Array Chip

Emmanuel ADETIBA*[1], F.A. IBIKUNLE[2], S.A. DARAMOLA[3], A.T. OLAJIDE[4]
[1,3]Department of Electrical & Information Engineering, School of Engineering and Technology, College of Science and Technology, Covenant University, Ota, Ogun State, Nigeria.
[2] Department of Computer, Information and Telecommunications Engineering, College of Science and Technology, Botswana International University of Science and Technology,
Gaborone, Botswana.
[4]Department of Computer Science, Kwara State Polytechnics, Ilorin, Kwara State, Nigeria.
*Correspondence Author: emmanuel.adetiba@covenantuniversity.edu.ng

***Abstract*-- Artificial Neural Network is widely used to learn data from systems for different types of applications. The capability of different types of Integrated Circuit (IC) based ANN structures also depends on the hardware backbone used for their implementation. In this work, Field Programmable Gate Array (FPGA) based Multilayer Perceptron Artificial Neural Network (MLP-ANN) neuron is developed. Experiments were carried out to demonstrate the hardware realization of the artificial neuron using FPGA. Two different activation functions (i.e. tan-sigmoid and log-sigmoid) were tested for the implementation of the proposed neuron. Simulation result shows that tan-sigmoid with a high index (i.e. k >= 40) is a better choice of sigmoid activation function for the harware implemetation of a MLP-ANN neuron.**

***Index Term*-- ANN, ASIC, DSP, FPGA, MLP**

1.0 INTRODUCTION

An artificial neuron was inspired principally from the structure and functions of the biological neuron. It learns through an iterative process of adjustment of its synaptic weights and a neuron becomes more knowledgeable after each iteration of the learning process. The ultimate aim of learning by the neuron is to adjust the weights and update the output for a new actual output which coincides with the desired output. However, the capability of a single artificial neuron is very limited. For instance, the Perceptron (a threshold neuron) cannot learn non-linearly separable function [1]. To learn functions that cannot be learned by a single neuron, an interconnection of multiple neurons called Neural Network (NN) or Artificial Neural Network (ANN) must be employed. Apart from the artificial neuron which is the basic processing units in ANN, there are patterns of connections between the neurons and the propagation of data called network topology. There are two main types of ANN topology which are; feed-forward and recurrent network topologies. In feed-forward networks, the data flow from input to output strictly in a forward direction and there is no feedback of connections while in recurrent networks, there are feedback connections. A commonly used feed-forward network topology is Multi-Layer Perceptron (MLP). MLP caters for learning of non-linear functions and Figure 1.0 shows its architectural representation.

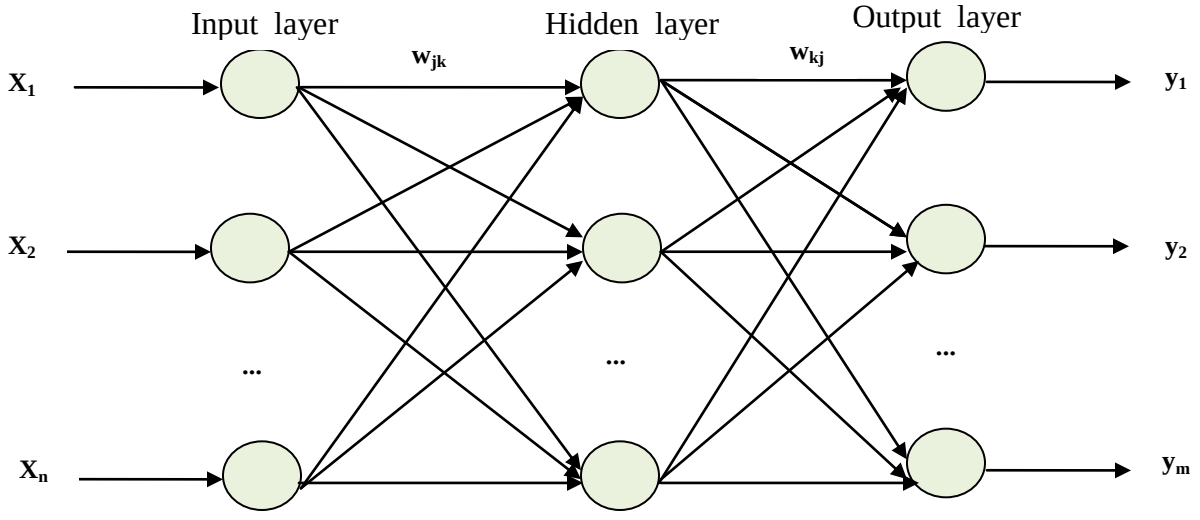

Fig. 1.0. Multi-Layer Perceptron (MLP) topology [2].

The MLP networks are typically trained with the training algorithm called the Backpropagation (BP) algorithm which is a supervised learning method that maps the process inputs to the desired outputs by minimizing the errors between the desired outputs and the calculated outputs [2]. BP is an application of the gradient method or other numerical

optimization methods to an ANN with feed-forward architecture in order to minimize the error function. The algorithm is the most popular method for performing supervised learning [3]. There are different variants of BP algorithm which include; conjugate gradient, Levenberg Marquardt (LM), gradient descent, quasi-Newton and etc.

In order to fully benefit from the massive parallelism that is inherent in ANN, it is essential to implement it in hardware. ANNs can be implemented in hardware using either analog or digital electronics [4]. Analog electronics implementation of ANN is always very efficient with respect to space and processing speed, however, these are achieved by trading off the accuracy of the computation elements of the network. Digital electronics implementation of ANN can be classified into three groups; i.) DSP-based implementation ii.) ASIC–based implementation and iii.) FPGA-based implementation [5]. DSP-based implementations are sequential and do not preserve the parallel architecture of ANNs and ASIC implementations do not support reconfigurability after deployment. However, FPGA based implementation is very suitable for hardware realization of ANN. It not only preserves the parallel architecture of neural networks, but also, it offers flexibility in reconfiguration, modularity and dynamic adaptation for neural computation elements.

FPGA which is an acronym for Field Programmable Gate Array is described by Stephen and Jonathan [6] as an integrated circuit containing gate matrix which can be programmed by the user "in the field" without using expensive equipment. The manufacturers of FPGA include; Xilinx, Altera, Actel, Lattice, QuickLogic and Atmel. Majority of FPGAs are based on SRAM (Static RAM) and they store logic cells configuration data in the static memory organized as an array of latches. This class of FPGA must be programmed upon start because SRAM is volatile. Examples of SRAM based FPGAs are Virtex and Spartan families (from Xilinx) and Cyclone and Stratix (from Altera). SRAM based Altera Cyclone FPGA is the adopted technology for hardware implementation of the artificial neuron in this work.

2.0 MATERIALS AND METHODS

Generally, ANN implementation usually starts with the neuron because it is the basic unit of any neural network. Meanwhile, the hardware implementation of a neuron has two major parts. The first part is the basic functional units that realise the inner product and the second part is the implementation of the activation function. The architecture for the hardware implementation of an artificial neuron is shown in Figure 2.0

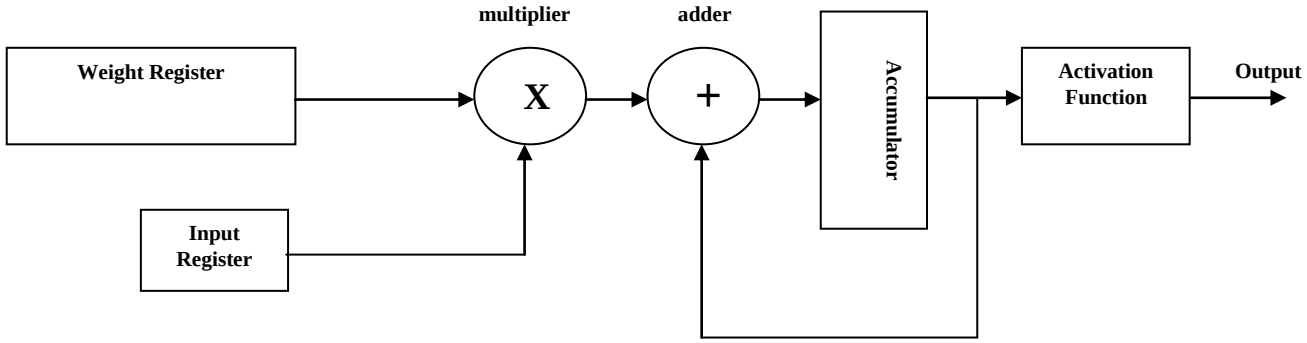

Fig. 2.0. Hardware architecture of an artificial neuron

2.1 Basic Functional Units

The basic functional units of a hardware neuron compute the inner product for the neuron and it is made up of the entities shown in Figure 2.0. The i**nput register** was implemented with a shift register for iterative entering of the input values into the neuron. The **weights register** was realized using a shift register and it serves the purpose of entering the corresponding weight of the current input value into the neuron. The **multiply accumulate (MAC) unit** of the neuron was realized with combinational circuits for full adder and multiplier. Appropriate number of bits were used for the input and output signals in the code so as to cater for the expected data range. These units were implemented with Very High-Level Description Language (VHDL) in Quartus II 9.0 Web Edition and the target was an Altera's DE2 board. This board contains an Altera Cyclone II 2C35 FPGA with a wide range of external memory, embedded multiplier, interfaces, I/O protocols and parameterizable IP cores [7]. The VHDL codes for the basic functional units are shown in Figure 3.0.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;
-- declare the entity
entity mac is
  generic (r : integer := 3;
         b : integer := 32);
  port( p : in signed (b-1 downto 0);
       w : in signed(b-1 downto 0);
       clk : std_logic;
       --w_out : out signed(b-1 downto 0);
       a : out signed (2* b-1 downto 0));
  end mac;
architecture Behavioral of mac is
   type weights is array (1 to r) of signed (b-1 downto 0);
   type inputs is array(1 to r) of signed (b-1 downto 0);
begin
process(clk,w,p)
variable weight : weights; variable input : inputs;
variable prod, acc : signed (2 * b-1 downto 0);
begin
if (clk'event and clk='1') then
weight := w & weight(1 to r-1); -- weights shift register
input := p & input(1 to r-1);
end if;
--input(1):= p1; input(2) := p2; input(3) := p3;
acc :=(Others =>'0');
--output weights
--multiply-accumulate(MAC)
L1: for j in 1 to r loop
 prod := input(j) * weight(j);
 acc := acc + prod;
end loop L1;
a <= acc; --linear output of the neuron
end process;
end Behavioral;
```

Fig. 3.0. VHDL codes for the basic functional units of a neuron

2.2 Neuron Activation Function

The commonly used activation functions in artificial neurons are linear, sigmoid and radial functions.
The linear activation has the form;

$$g(z) = z. \tag{1.0}$$

The sigmoid activation functions are *S* shaped and the ones that are mostly used are the logistic and the hyperbolic tangent (equations (2.0) and (3.0) respectively);

$$g(z) = \frac{1}{1+e^{-az}}, \tag{2.0}$$

$$g(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}. \tag{3.0}$$

There are different types of radial activation functions, but the one that is usually adopted uses Gaussian function;

$$g(z) = e^{-\frac{z^2}{b^2}}. \tag{4.0}$$

However, for the hardware implementation of the neuron in this work, the taylor series of the sigmoid activation functions (i.e. log-sigmoid and tan-sigmoid) which are the most commonly used activation functions in ANNs were analyzed since they cannot be implemented directly in hardware because they both contain exponential functions. With proper analysis, we were able to make an informed decision on the appropriate choice of sigmoid activation function for hardware neuron implementation. The analysis is reported in the subsequent sub-sections.

2.2.1 Taylor Series Approximations for Log-Sigmoid and Tan-Sigmoid

Restating equation (2.0), we have;

$$f(x) = \frac{1}{1+e^{-x}}, \qquad (5.0)$$

The power series of e^{-x} is;

$$e^{-x} = \sum_{n=0}^{\infty} (-1)^n \frac{x^n}{n!}, \qquad \forall\, x \in \mathbb{R}. \qquad (6.0)$$

For $0 \le n \le k$ let $z = \sum_{n=0}^{k} (-1)^n \frac{x^n}{n!}$ (7.0)

Putting (7.0) into (5.0), Taylor's series is obtained for (5.0) as:

$$f(z) = \frac{1}{1+z}. \qquad (8.0)$$

Equation (8.0) is the Taylor's series representation of log-sigmoid activation function.

Restating equation (3.0) for tan-sigmoid activation function gives;

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}, \qquad (9.0)$$

The power series for e^{x} is;

$$e^{x} = \sum_{n=0}^{\infty} \frac{x^n}{n!}, \qquad \forall\, x \in \mathbb{R}. \qquad (10.0)$$

For $0 \le n \le k$, (10.0) becomes;

$$y = \sum_{n=0}^{k} \frac{x^n}{n!}. \qquad (11.0)$$

Substituting equations (7.0) and (11.0) into (9.0) produces;

$$f(y,z) = \frac{y-z}{y+z}. \qquad (12.0)$$

Therefore, equation (12.0) is the Taylor series for tan-sigmoid activation function. The pseudocodes from these analysis are shown in figures 4.0 and 5.0.

```
LogSigmoid(X)
/* Initialize variables*/
y = 0: Prod = 1: LogSig = 0
Read k
For n = 0 To k
   Prod = ((-1) ^ n) * ((X ^ n)/Fact(n))
   y = y + Prod
EndFor
/* Compute the Taylor series LogSig */
LogSig = 1/(1 + y)
DISPLAY LogSig
End
Fact(n)
/* Initialize variable(s) */
 Factorial = n
For i = n To 2
   Factorial = Factorial * (i-1)
EndFor
Return Factorial
```

Fig. 4.0. Pseudocode for log-sigmoid Taylor series

```
TanSigmoid(X)
/* Initialize variables*/
y = 0: z = 0: Prod1 = 1: Prod2 = 1: TanSig = 0
Read k
For n = 0 To k
    Prod1 = ((-1) ^ n) * ((X ^ n)/ LogSigmoid.Fact(n))
    y = y + Prod1
   Prod2 = (X ^ n)/LogSigmoid.Fact(n)
   z = z + Prod2
EndFor
/* Compute the Taylor series Tan-Sig */
TanSig = (y-z)/(y+z)
DISPLAY TanSig
End
```

Figure 5.0: Pseudocode for tan-sigmoid Taylor series

3. Experimental Results and Discussion

The VHDL codes shown in Figure 3.0 were simulated functionally in Quartus II 9.0 Web Edition environment on Altera's DE2 board that contains Altera Cyclone II 2C35 FPGA. The Register Transfer Logic (RTL) of the VHDL code for the basic functional units with 3 inputs and 3 weights is shown in Figure 6.0 and the simulation output is shown in Fig. 7.0.

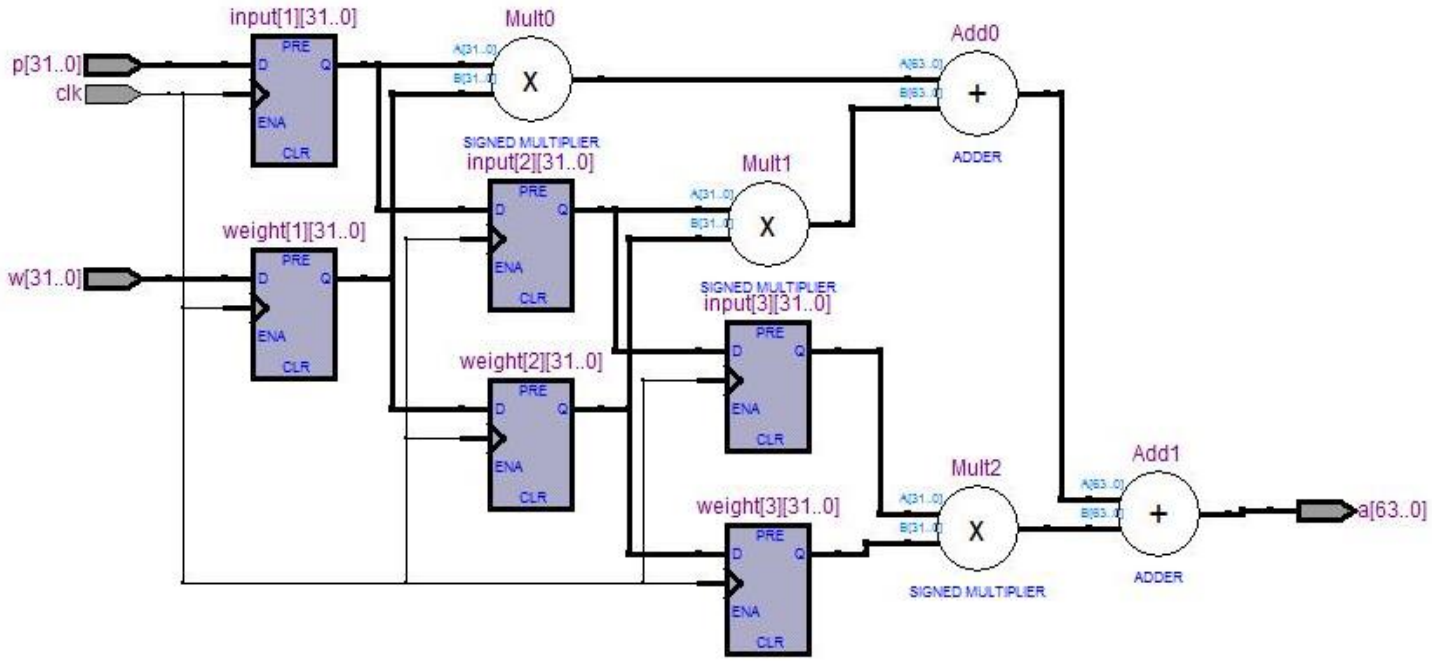

Fig. 6.0. RTL of the basic functional units of the artificial neuron

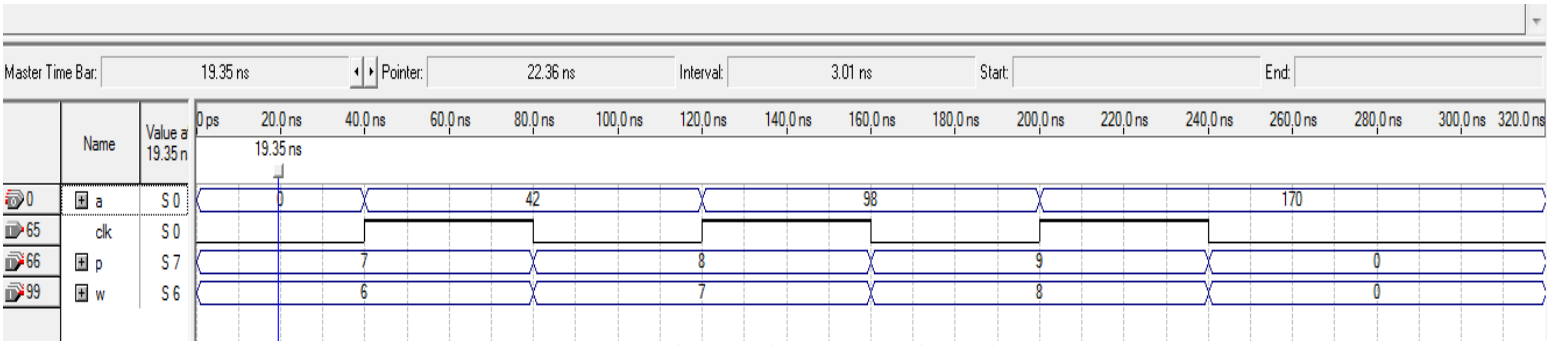

Fig. 7.0. Simulation output of the basic functional units of the artificial neuron

The simulation output in Figure 7.0 which shows the implementation result of multiply and accumulate (MAC) operation on the content of the input and weight registers illustrates a perfect output for the inputs. This is an attestation to the correctness of our VHDL codes (Figure 3.0) for the basic functional units part of the artificial neuron implemented in this work.

Also, experiments were carried out so as to ascertain the appropriate sigmoid activation function between log-sigmoid and tan-signed for the hardware realization of artificial neurons. The pseudocode for log-sigmoid and tan-sigmoid and their respective Taylor series' approximations (Figures 4.0 and 5.0) were implemented in MATLAB R2008a. Experimental trials for k=10, 20 and 40 and for values of X ranging from -20 to +20 (in Equations 7.0 and 11.0) for the two activation functions were performed. The results obtained from these experiments were graphically plotted in order to aid our comparative analysis. These plots are shown in Figures 8.0, 8.1, 8.2, 9.0, 9.1 and 9.2.

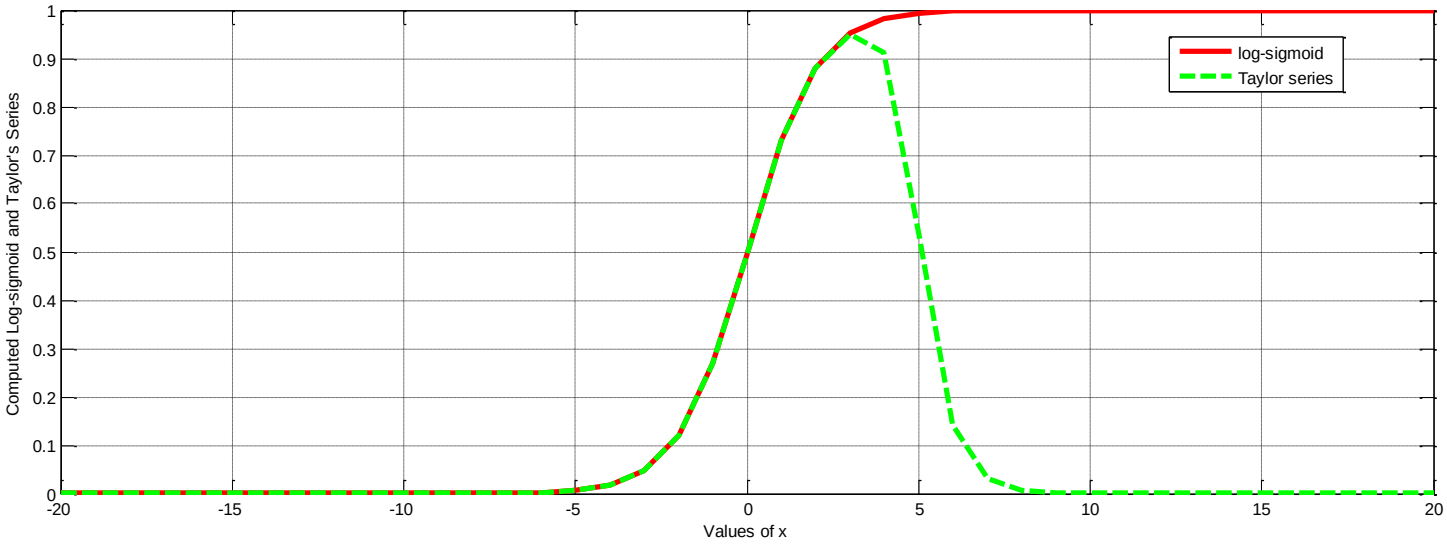

Fig. 8.0. Log-sigmoid and it's Taylor series approximation for $0 \leq k \leq 10$

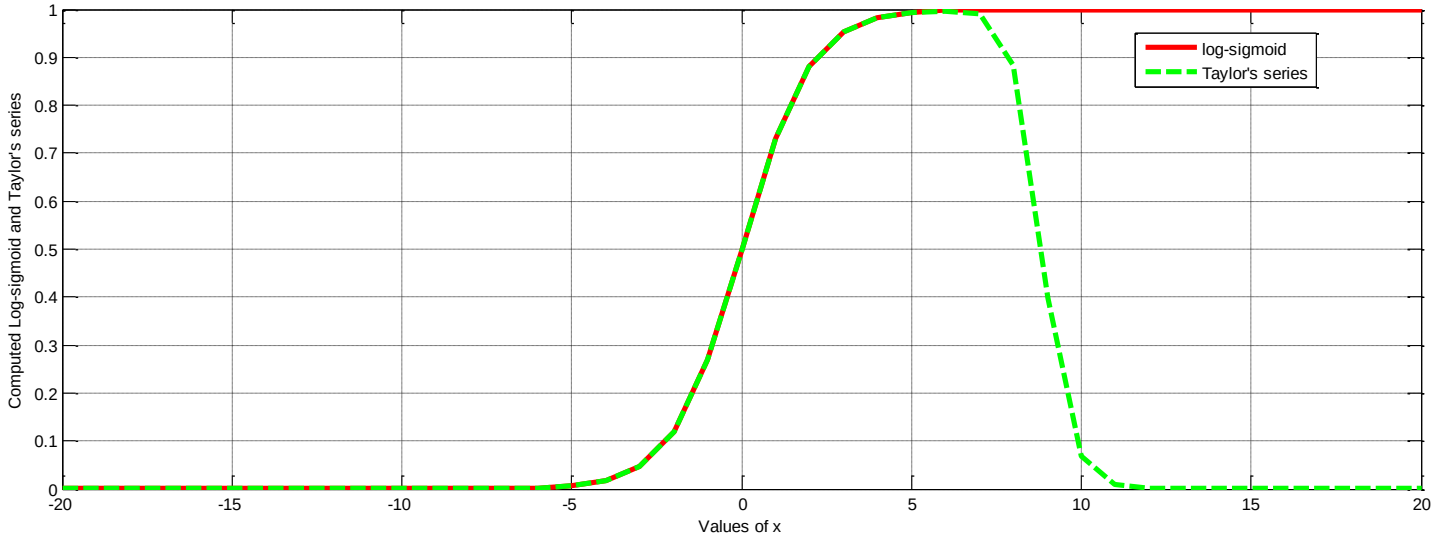

Fig. 8.1. Log-sigmoid and it's Taylor series approximation for $0 \leq k \leq 20$

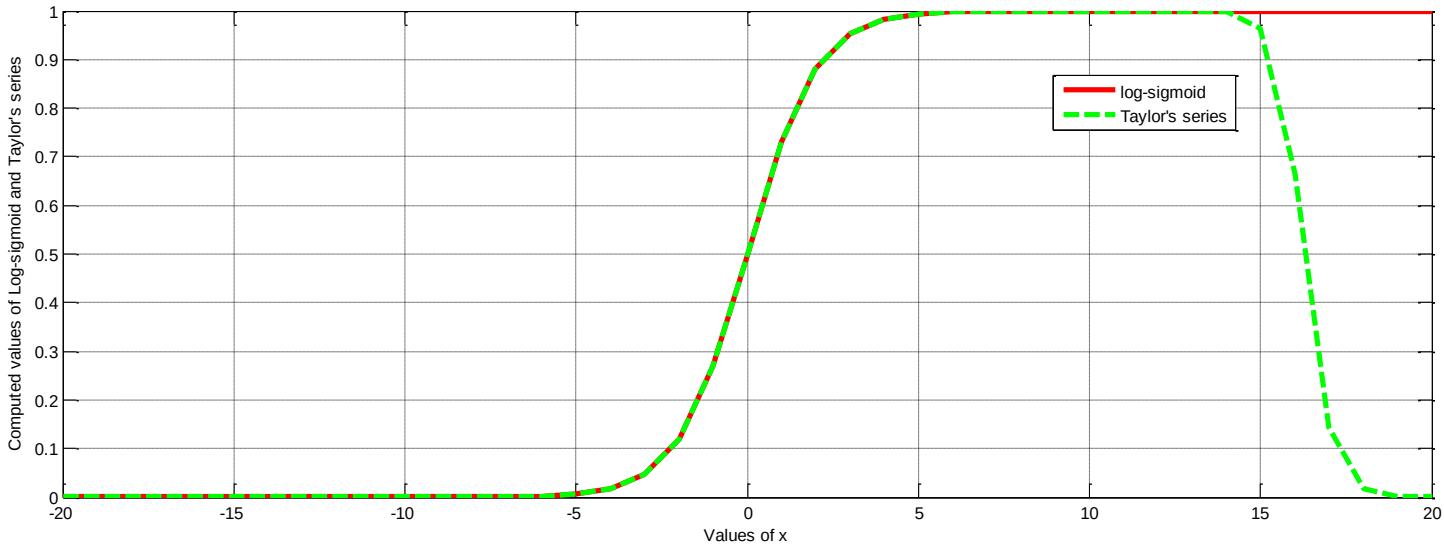

Fig. 8.2. Log-sigmoid and it's Taylor series approximation for $0 \leq k \leq 40$

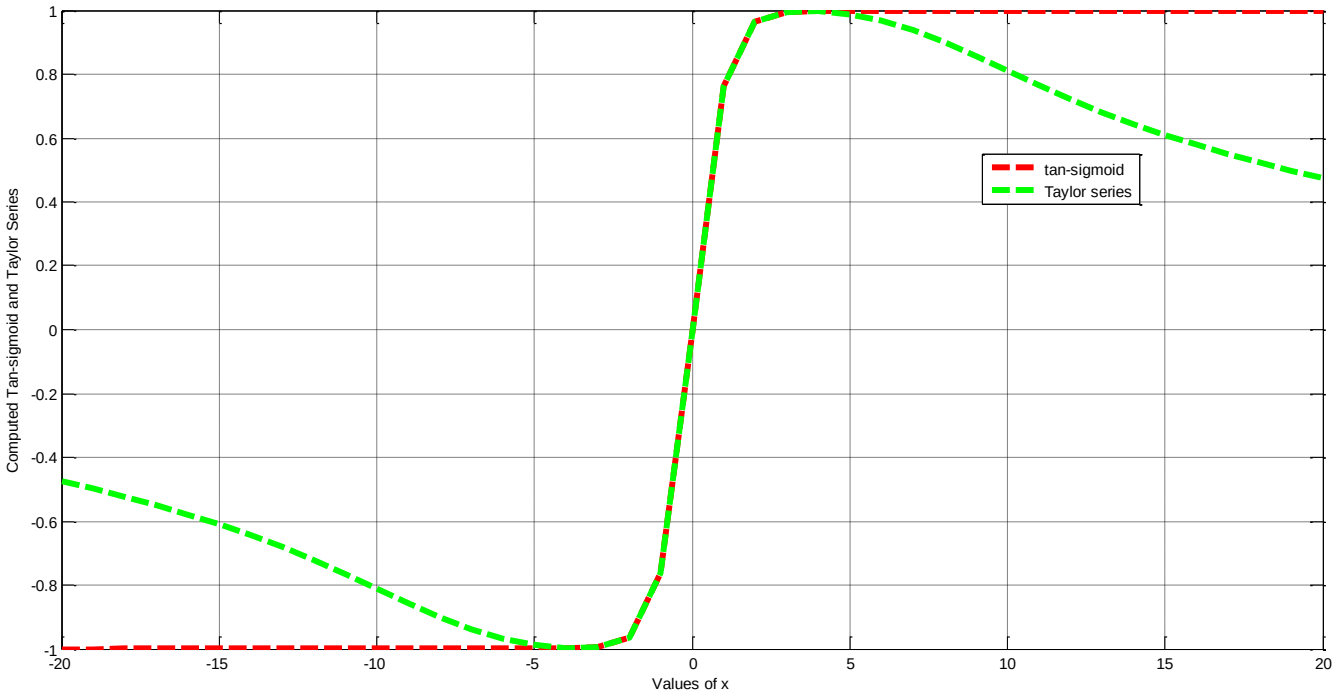

Fig. 9.0. Tan-sigmoid and it's Taylor series approximation for $0 \leq k \leq 10$

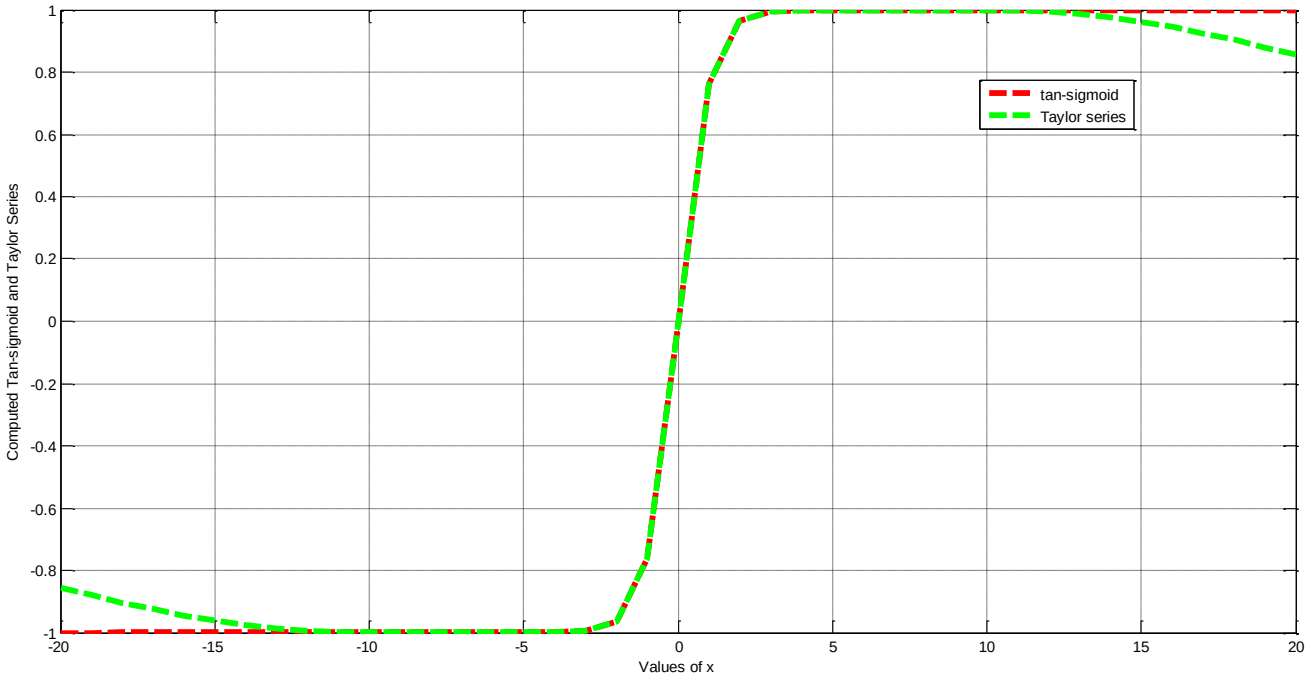

Fig. 9.1. Tan-sigmoid and it's Taylor series approximation for $0 \leq k \leq 20$

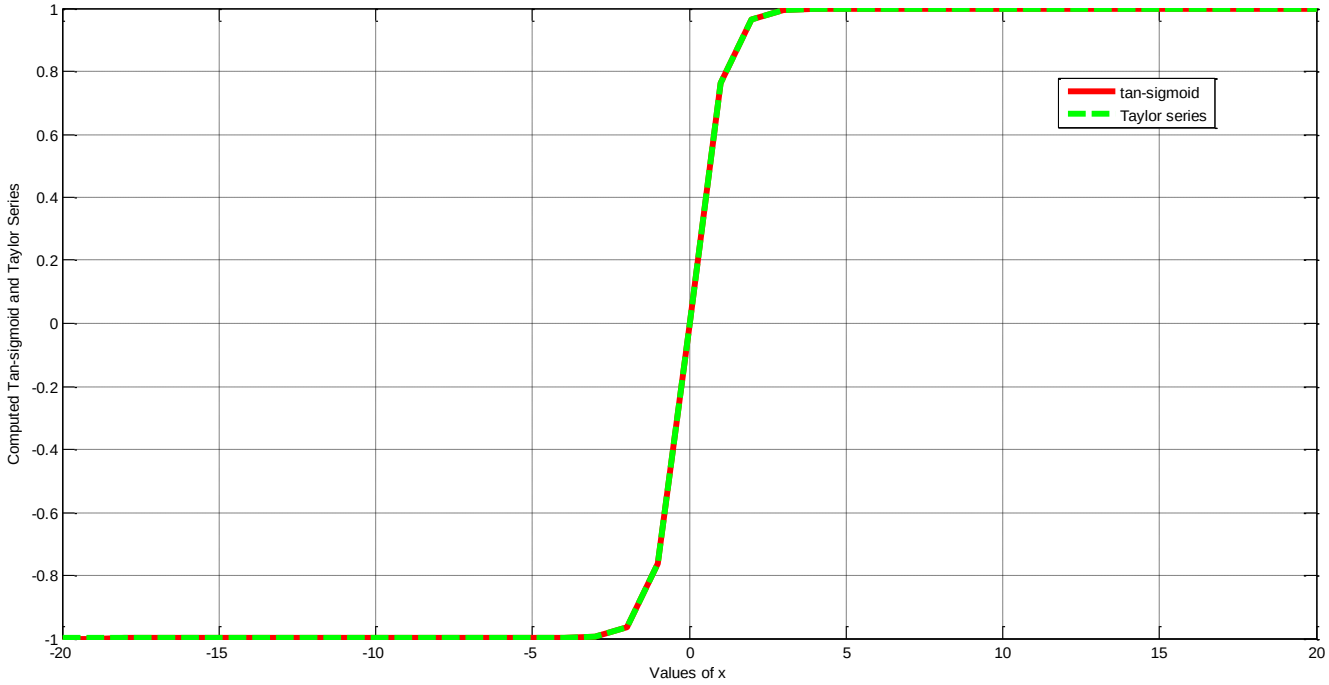

Fig. 9.2. Tan-sigmoid and it's Taylor series approximation for $0 \leq k \leq 40$

Figures 8.0, 8.1 and 8.2 show that for the various values of k (i.e. 10, 20 and 40) for $-20 \leq X \leq 20$, there are little convergences between the actual log-sigmoid function and its Taylor series approximations. Meanwhile, k = 40 was anticipated to give a good convergence but the plot in Figure 8.0 shows that for higher positive values of X, the deviation between the actual function and the Taylors series approximation was getting more pronounced. However, from Figures 9.0, 9.1 and 9.2, the convergences between tan-sigmoid function and its Taylor series approximations improve as the values of k range from 10 to 20 to 40 for $-20 \leq X \leq 20$. Infact, as shown in Figure 9.2, at k = 40, there is a perfect convergence between the actual tan-sigmoid function and its Taylor series approximation for the range $-20 \leq X \leq 20$.

4.0 CONCLUSION

The result in Figure 7.0 shows that our VHDL code in this work is very accurate and can be reliably loaded into an FPGA (i.e. Altera's DE2 board that contains Altera Cyclone II 2C35 FPGA) to realize the basic functional units of any artificial neuron. Also, the plot in Figure 9.2 shows that tan-sigmoid with a high index (i.e. k >= 40) is a better choice of sigmoid activation function for the hardware implementation of an artificial neuron. A Multi-Layer Perceptron (MLP) neural network can therefore be implemented on FPGA by aggregating several of the hardware neurons in this work based on the required MLP configuration for a given area of application. Our next direction for this work is to adapt an FPGA-based MLP neural network to realize the classifier sub-module of a genomics-based diagnostic system for lung cancer. However, FPGA-based MLP neural network hardware can be applied in other areas such as communications, control, next-generation sequencing, biometrics and biomedical devices.

REFERENCES

[1] Tredennick, N., (1996). Microprocessor-based computers, IEEE Computer: 50 years of computing, 27-37.
[2] Huang, Y. (2009). Advances in Artificial Neural Networks: Methodological Development and Application, Algorithms, 2: 973-1007
[3] Werbos, P. J., (1994). The Roots of Backpropagation: From ordered derivatives to Neural Networks and Political Forecasting, John Wiley and Sons, New York.
[4] Fiesler E., and Beale, R., (1997). Handbook of Neural Computation, E1.2:1-13, Institute of Physics Publishing and Oxford University Publishing, New York.
[5] Pedro F., Pedro, R., Ana A., and Fernando M. D., (2007). A high bit resolution FPGA implementation of a FNN with a new algorithm for the activation function, Neurocomputing, 71:71–77.
[6] Stephen B., and Jonathan R., (2011). Architecture of FPGAs and CPLDs: A Tutorial, Department of Electrical and Computer Engineering, University of Toronto.
[7] Altera (2012). DE2 Development and Education Board User Manual, Version 1.6, Altera Corporation, 4-6.