2014 11th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE)

Multilayer perceptron network with integrated training algorithm in FPGA

A.N. Pérez-García[1], G.M. Tornez-Xavier[1], L.M. Flores-Nava[1],
F. Gómez-Castañeda[1], J.A. Moreno-Cadenas[1]
[1]Department of Electrical Engineering, CINVESTAV-IPN, Mexico D.F., Mexico
Phone (52) 55 5747 3800 ext. 6261
E-mail: {nperez, gtornez, lmflores, fgomez, jmoreno} @cinvestav.mx

***Abstract* — In this manuscript we present the implementation of an artificial neural network type Multilayer Perceptron (ANN-MP or NNMP) in Field-Programmable Gate Arrays (FPGA), including Back-Propagation training method based on descendent gradient. This network has 2 reconfigurable hidden layers, adjustable parameters (epochs and ratio learning) and batch learning. The proposed architecture aims to reduce the number of logical elements to be used, so serial processing is utilized. In order to test the performance of the trained network, a nonlinear function was approximated with satisfactory results.**

Keywords — Artificial neural network, back propagation, descendent gradient, FPGA.

I. INTRODUCTION

The study of artificial neural networks (ANN) and its implementation in hardware has become significant in engineering applications. ANN are used in plentiful variety of applications. The choice of one ANN depends on the problem to be solved, either for function approximation, classification, data fixing, pattern recognition or forecasting. An ANN architecture consists of a set of connections between simple nonlinear processing units (neurons) that team up to produce an output stimulus [1]. Multilayer Perceptron (MLP) is one of the most widespread architectures, and Back Propagation is the foremost training method for adjusting it. For creating a network is very important its topology, however there is not a clear methodology for develop and training it. A complex network is not always the most efficient to reduce the error, it depends on the application.

Hardware implementation of an ANN for a specific problem entails two essential stages:

Configuration and training: network type and configuration (inputs, number of layers, number of neurons per layer, connections between layers, etc.) are chosen, then a software tool is used for training. This training produces the weights and bias with which the network operates according to the task it was conceived.

Hardware implementation: optimized weights and bias as well as the best ANN configuration are used to build the hardware implementation. This information lets know the number of adders, multipliers, registers, etc., and connections among them in order to conform the hardware system.

Once there is a hardware implementation, if it were necessary to change the architecture, adding more inputs/outputs or using new training data it would be indispensable to reconfigure and train the network in software again and carry out a new hardware implementation.

FPGAs are the most utilized devices for an ANN hardware implementation due to its reconfigurable platform. These devices provide orders of magnitude of better performance compared to software simulation [3]. Hardware implementation of an ANN with training algorithm have resulted in various studies, the main focus of these studies is the implementation of parallel networks that do faster computations [4, 5, 6]. Other focus is the optimization of logic resources, networks working serially [8]. When training a network, the training data can be presented in three training protocols: stochastic, batch and on-line [2]. However, batch training not used, because of the complexity in the calculation of training algorithm.

In this work we present the hardware implementation of a NNMP that includes a training algorithm based on Back-Propagation (NNMP-BP) with batch training protocol, which allows training the network directly in hardware in real time. The NNMP-BP architecture was built using a generic VHDL description (Very high speed integrated circuit Hardware Description Language). For a better understanding of the implemented algorithm, Back Propagation and its mean features is introduced in Section II. The proposed architecture is presented in section III. Some representative results and conclusions are given in sections IV and V, respectively.

II. BACKPROPAGATION ALGORITHM

A. MULTILAYER PERCEPTRON

Multilayer Perceptron architecture (MP) is one of the most employed ANN. A generic MP is shown in Fig. 1 and consists of the following layers:

Input Layer: It has one or more inputs whose number depends on the application; each input is connected and multiplied by the synaptic weight of each neuron of the first hidden layer.

Hidden Layers: They consist of one or more layers of neurons; each neuron is connected to all the neurons in the next layer by a synaptic weight.

Output Layer: It contains the neurons matching the number of the network outputs.

978-1-4799-6230-3/14/$31.00 ©2014 IEEE

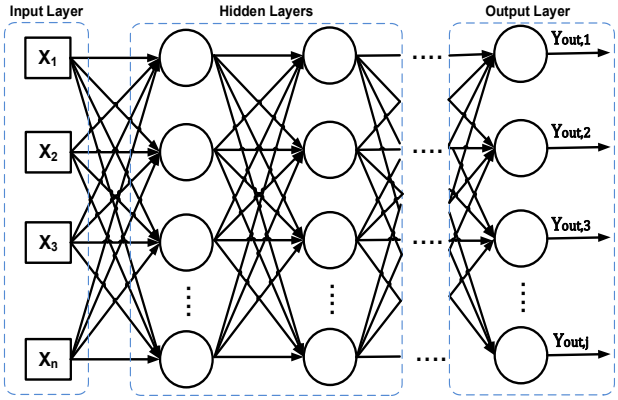

Fig. 1 Multilayer Perceptron Neural Network Model.

The use of NNMP goes through two stages: a *training phase*, where the network learns to realize a particular task and an *operation phase*, when the network performs the task it was trained for. Throughout the training phase there are two important steps: *Forward stage* and *Back-Propagation stage*.

In the Forward stage the output of the network is calculated from the input values, that is, input signals propagate through hidden layers until an output value is obtained.

In the *Back-Propagation stage* it is calculated an error between the expected value and the output obtained in the forward stage, this error is propagated to the neurons inside the network via synaptic weights, an error corresponding to each neuron is returned and the network weights and bias are updated.

B. FORWARD STAGE

In the Forward stage the output of the network is calculated. Let's assume *Xi = [X1, X2... Xn]* represents the inputs of the network, *Wij* is the synaptic weight of neuron j associated with an input i, $Y_{h,j}$ is the output of neuron j ($j = 1, 2... q$) in the layer h ($h = 1, 2... m$); b_{hj} is the bias. The input of a neuron of the first hidden layer ($h = 1$) is expressed by equation (1).

$$S_{1,j} = \sum_{i=1}^{n} \left(W_{i,j} * X_i + b_{1,j}\right) \quad \ldots \quad (1)$$

The accumulation performed in the input of a neuron is processed by a nonlinear activation function. Tangent sigmoid function is generally used, and it generates a continuous output value which varies between -1 and 1. In Fig. 2 a tangent sigmoid and its derivative are shown.

The output of a neuron in the first hidden layer is expressed by equation (2):

$$Y_{1,j} = f\left(S_{1,j}\right) = f\left(\sum_{i=1}^{n} \left(W_{i,j} * X_i + b_{1,j}\right)\right) \quad \ldots \quad (2)$$

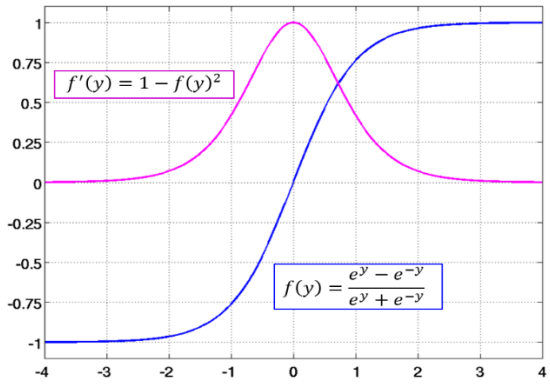

Fig. 2 Tangent sigmoid function and its derivative.

Calculations made by neurons in subsequent hidden layers and the output layer are expressed by equations (3) and (4).

$$S_{h,j} = \sum_{k=1}^{m} \left(W_{k,j} * Y_k + b_{h,j}\right) \quad \ldots \quad (3)$$

$$Y_{h,j} = f\left(S_{h,j}\right) = f\left(\sum_{k=1}^{m} \left(W_{k,j} * Y_k + b_{h,j}\right)\right) \ldots \quad (4)$$

Where $k=j^{(h-1)}$ denotes neuron j of previous layer (h-1); $W_{k,j}$ is the synaptic weight associated between neurons k and j.

C. BACK-PROPAGATION STAGE

In this stage the error signal is propagated backwards within the network and the descendent gradient algorithm is used to update weights and bias; the update is performed in three main steps, namely, 1), 2) and 3).

1) Using equation (5) we get the error for neurons in the output layer, and equation (6) calculates the gradient of error.

$$\varepsilon_{out,j} = \left(t_{arget} - Y_{out,j}\right) \quad \ldots \quad (5)$$

$$\delta_{out,j} = \varepsilon_{out,j} * f'\left(Y_{out,j}\right) \ldots \quad (6)$$

Where $\varepsilon_{out,j}$ is the error between the target and the actual output of neuron j in output layer; $\delta_{out,j}$ is the gradient of error which is propagated into neurons in hidden layer via synaptic weights. Equations (7) and (8) respectively determine the propagated error and its local gradient.

$$\varepsilon_{h,j} = \delta_l * W_{j,l} \quad \ldots \quad (7)$$

$$\delta_{h,j} = \varepsilon_{h,j} * f'\left(Y_{h,j}\right) \quad \ldots \quad (8)$$

Where $l=j^{(h+1)}$ represents neuron j in subsequent layer of h; ; $W_{j,l}$ is the synaptic weight between neurons *j* and *l*.

2) Equations (9) and (10) define respectively the variation of weights and bias for the first layer and those for internal layers.

$$\Delta W_{i,j} = \alpha * \delta_{1,j} * X_i \qquad \Delta b_{1,j} = \alpha * \delta_{1,j} \quad \ldots \quad (9)$$

$$\Delta W_{k,j} = \alpha * \delta_{h,j} * Y_{h,j} \qquad \Delta b_{h,j} = \alpha * \delta_{h,j} \quad \ldots \quad (10)$$

Where α is the learning ratio that governs the degree of convergence of the network to learn.

3) Weights and bias updating.

$$W_{i,j}^{(n+1)} = W_{i,j}^{(n)} + \Delta W_{i,j}^{(n)}$$
$$b_{1,j}^{(n+1)} = b_{1,j}^{(n)} + \Delta b_{1,j}^{(n)} \quad \ldots \quad (11)$$

$$W_{k,j}^{(n+1)} = W_{k,j}^{(n)} + \Delta W_{k,j}^{(n)}$$
$$b_{h,j}^{(n+1)} = b_{h,j}^{(n)} + \Delta b_{h,j}^{(n)} \quad \ldots \quad (12)$$

Where (n) represents present values and $(n+1)$ are the values to be updated for use in the next iteration.

D. BATCH TRAINING

Calculation realized in the *forward* and the *back-propagation* stages for each presented pattern, embodies one learning iteration (epoch). When training a network with protocol stochastic or on-line, the gradient is calculated and the weights are updated for each iteration. In batch processing, the gradients obtained for each iteration are added and averaged, and the weights are updated when all the training patterns are presented.

III. HARDWARE IMPLEMENTATION

The network that we implemented in hardware has the following features: 2 inputs (X, Y), 2 hidden layers and 1 output layer to conform a 2-5-2-1 architecture (Fig. 3). Depending on the problem complexity, we use switches to activate/deactivate the neurons that conforms the hidden layers so we can evaluate the proposed training algorithm for different hidden layer structures.

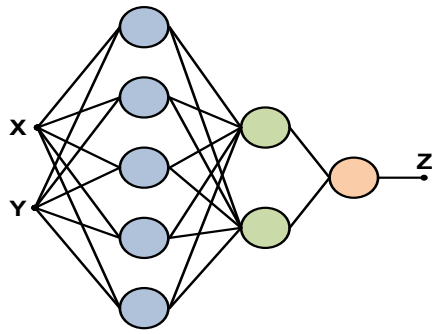

Fig. 3 Hardware reconfigurable network 2-5-2-1.

A. HARDWARE

When implementing a ANN in an FPGA one key thing to consider is the use of logical resources, particularly the multipliers included in DSP block, because digital implementation of neurons generally uses three multipliers per neuron, as will be explained in the next section, moreover the number of DSP's is limited for each FPGA family. For our system, we utilized an Atlys development board that uses a Spartan-6 XC6SL45 device of Xilinx.

For the implementation we worked with 16-bit words, utilizing a fix-point format [9] with two-complement signed numbers. The word was divided as shown in Fig. 4.

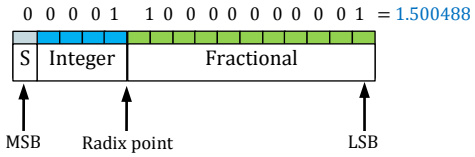

Fig. 4 16-bit fix-point number format.

B. BASIC NEURON

According with (2) and (4), a neuron executes two main processes:

1. The sum of products of the inputs by the weights plus their bias.
2. Evaluation of the activation function (tangent sigmoid). This was done in digital form using line segments from which the slope (a) and intercept (c) are obtained for each line [7].

In our case, an additional process is the estimation of the activation function derivative.

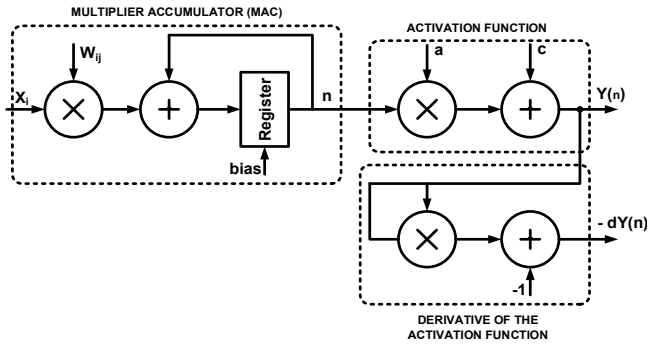

Fig. 5 Processes performed in a neuron.

Neuron digital implementation requires three multipliers to accomplish the processes described above (Fig. 5), this might consume most DSPs available in the FPGA; for reducing this amount we work with serial processing technique [4, 5, 6]. Fig. 6 depicts the use of a single multiplier for the three processes described above.

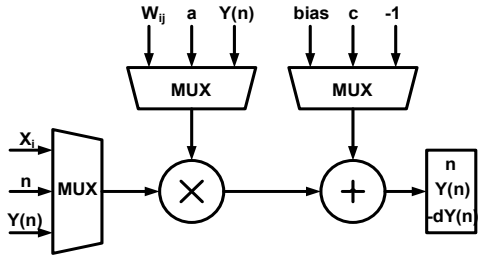

Fig. 6 Neuron serial processing.

Multiplexers are controlled by a selection signal that operates as follows:

1. For the first selection value, input (Xi), the synaptic weight (Wij) and bias are picked to get the output value (n) of MAC function; this value is stored in a register and fed back to the multiplexers.
2. When the selection signal changes to the second value, (n) is multiplied by the slope (a) of the selected line and added to its ordinate (c), in this way the activation function is evaluated to obtain the output of the neuron Y(n), which is stored in the register; this value is fed equally to the multiplexers.
3. With the next value of the selection signal we get the derivative of the activation function by multiplying Y(n) by itself and adding (-1).

C. GENERAL STRUCTURE

The NNMP-BP implemented in hardware is configurable and the network architecture to be used can be selected; the learning ratio (lr) as well as the number of training epochs can be set. In Fig. 7 the NNMP-BP architecture is shown. There are three main modules whose function is explained below.

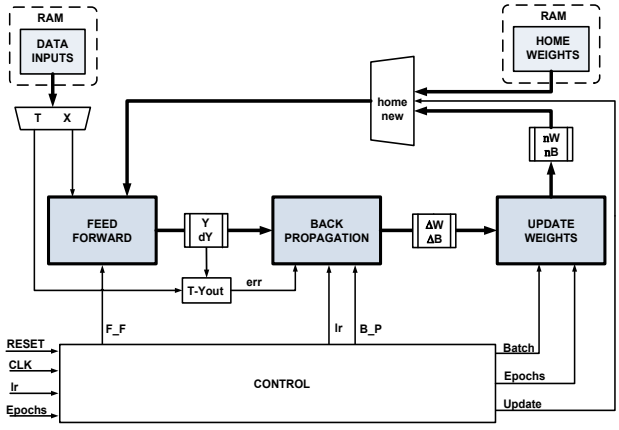

Fig. 7 NNMP-BP implementation.

a) *Feed forward*

This module contains the network architecture, which has three neuron layers, each neuron corresponds to a calculation block as previously explained.

The output of this module provides the calculated values of each neuron $Y_{h,j}$ and $Y_{out,j}$ which correspond to equations (2 and 4). The derivative of each block $dY_{h,j}$ y $dY_{out,j}$ is also obtained.

b) *Back Propagation*

This module calculates the deltas of weights and biases, corresponding to equations (6-10). Due to the large number of multipliers needed for these, serial processing technique is applied too.

c) *Weights Update*

Synaptic weights and biases are updated in this module for each training epoch, in accordance with (11) and (12), carrying out the following tasks:

1. Sum of synaptic weights deltas obtained for each training pattern.
2. Mean of calculated deltas (batch training) to obtain the new synaptic weights values, which will be transmitted to the module output.

Besides the 3 basic modules, there are other modules used for the neural network synchronization and autonomy.

d) *Control*

This module generates the signals to control the flow of information throughout the other modules and to enable or disable them for the realization of different processes:

Training

This process is performed using Feed Forward and Back Propagation modules which are activated using F_F and B_P signals, in this step we utilize 1024 vectors.

Neural network testing

Once the network has been trained, the system is set in *testing mode* using F_F signal, which activates only the Feed forward module and disables Back Propagation and weight updating processes. In this phase the neural network employs only the updated synaptic weights and biases, and the output produced by the network for the 128 test data is evaluated.

The training of an artificial network at software level (Matlab) uses different stop criteria (epochs, performance, validation checks, etc.), thus it is possible to monitor the performance function (mean square error MSE) to be minimized. In this work we use the number of epochs as stop condition which can be adjusted as described before.

e) *Data Inputs*

In this module the 1024 training data and the 128 test data are stored.

f) *Home Weights*

In this section initial synaptic weights and biases are stored. To obtain these values it is used Matlab that generates them randomly.

D. FPGA UTILIZED RESOURCES

One of the objectives proposed in this paper is the optimization of logic resources, so we had to limit the training and test data to be represented by power-of-two numbers in order to simplify the averaging by elimination division and just using arithmetic shift instead. Table 1 resumes FPGA logic resources used for implementing our NNMP-BP architecture.

Table 1. Resources used by the FPGA

Target Device: xc6slx45-3csg324			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	6,151	54,576	11%
Number of Slice LUTs	6,384	27,288	23%
Number of occupied Slices	2,538	6,822	37%
Number of RAM B16BWERs	3	116	2%
Number of RAM B8BWERs	3	232	1%
Number of DSP48A1s	11	58	18%
Number of PLL_ADVs	1	4	25%

IV. TESTING AND RESULTS

In order to evaluate the implemented architecture, the task of approximating a nonlinear function was proposed (Fig. 8). We tested 2-5-1 and 2-5-2-1 architectures.

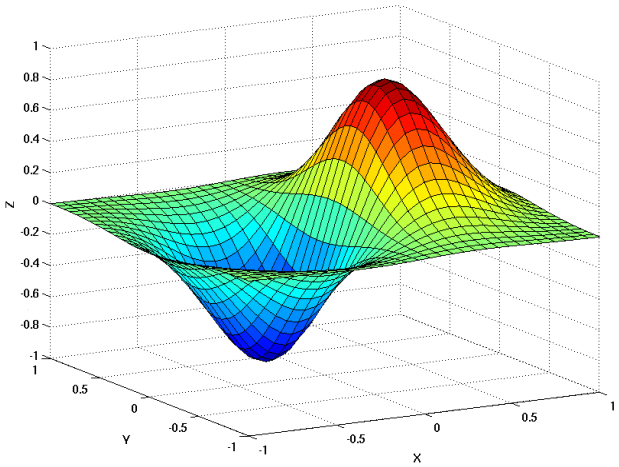

Fig. 8 Test function $z = 4xe^{-4(x^2+y^2)}$

The training was made for different learning rates and different epochs. In Fig. 9 we can see the graphic of MSE during the training of the 2-5-2-1 network architecture for different learning rates; we observe that a learning rate lr = 0.7 produced a MSE of 0.00143.

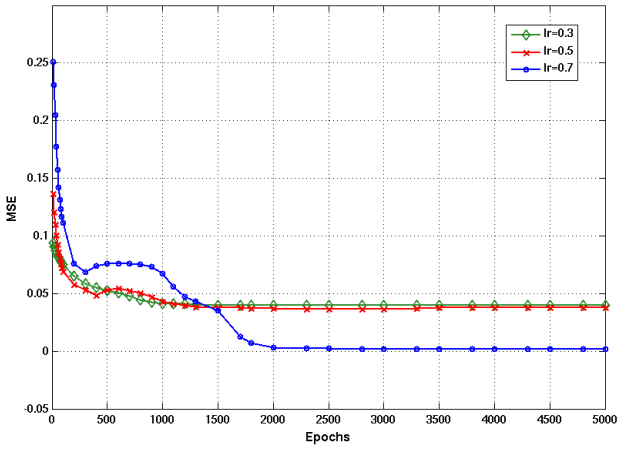

Fig. 9 MSE for different learning ratios in 2-5-2-1 architecture.

To evaluate the implementation it was created a module that calculates the MSE for the test data. The values obtained for the two selected architectures are presented in Table 2.

Table 2. Results of testing both architectures

	Architecture (2-5-1)	Architecture (2-5-2-1)
MSE	0.01434	0.00143
R	0.9441	0.9936
lr	0.55	0.7
Epochs	10000	10000

From the hardware implementation, FPGA data were extracted for analysis in Matlab. Fig. 10 shows the MSE for the 128 test data; it represents the squared difference between the output data produced by the FPGA and the targets. We note that for this application our system has a maximum square error of about 0.01.

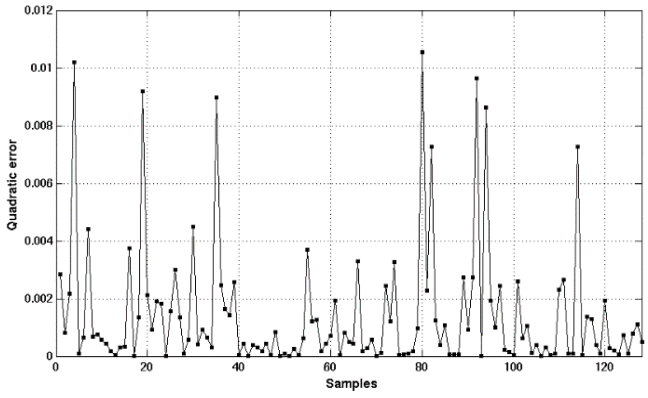

Fig. 10 Quadratic error for 2-5-2-1 architecture.

In Fig. 11 is presented a scatter plot with the regression line and the line equation that will help us to make future predictions about the output that would produce our digital system from any input value. Additionally, it is shown the Pearson correlation coefficient, which indicates the degree of dependence between expected test data and the output data of the digital system and it has a value R = 0.9936.

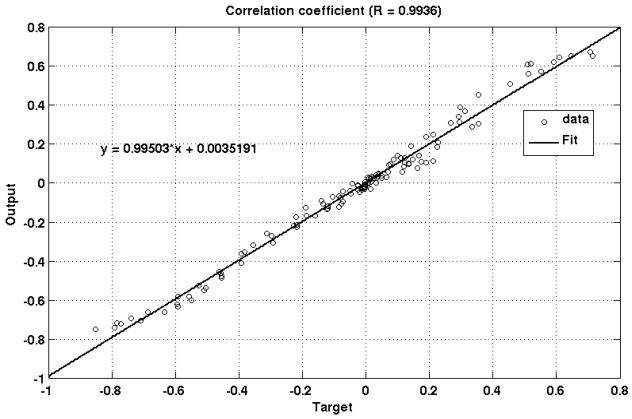

Fig. 11 Correlation coefficient (R) for the test set and 2-5-2-1 architecture.

The results can not to be compared directly with other previous studies, as each study takes a different focus, using different architectures and implemented in different technologies. However, we can mention that in this work focused on the optimization of resources (one DSP by neuron) and train the network using a protocol type batch. Table 3 compares these features with other studies.

V. CONCLUSIONS

In this paper, we presented the implementation of an ANN architecture with Back Propagation training algorithm. It is reconfigurable, allowing us to change its learning rate and training epochs in hardware to train the network in real time without the need of reprogramming the FPGA for each change of these parameters.

Serial processing implementation allows significant saving of FPGA logical resources, but the main drawback of this technique is in reducing its performance, increasing the network response time, which was 1.96 ms per epoch for a 50 MHz clock signal. In this first version the number of training epochs was implemented as stopping criterion; for an improved version we want to add more stop criteria to achieve a better performance of the training algorithm. Another point to consider is to include a routine for generating random initial weights and biases within the implemented system.

ACKNOWLEDGMENTS

To my advisers for their thought assistance, and Dr. Oliverio Arellano Cardenas for his technical support for the elaboration of this manuscript.

REFERENCES

[1] M.T. Hagan, H. B. Demuth, M. Beale, “Neural Network Design”, PWS Publishing Company, 1996.

[2] Richard O. Duda, Peter E. Hart, David G. Stork, “Pattern Classification”, John Wiley & Sons, 2001.

[3] Scott Hauck "The Roles of FPGAs in Reprogrammable Systems". *Proceedings of the IEEE, Vol. 86, No. 4, pp. 615-639, April, 1998*.

[4] Rafael Gadea, Joaquín Cerdá, Francisco Ballester, Antonio Mocholí, "Artificial Neural Network Implementation on a single FPGA of a Pipelined On-Line Backpropagation". *Proceedings of The 13th International Symposium on System Synthesis, IEEE, 2000*.

[5] Vijay Pandya, Shawki Areibi, Medhat Moussa, "A Handel-C Implementation of the Back-Propagation Algorithm On Field Programmable Gate Arrays". *Proceedings of the 2005 International Conference on Reconfigurable Computing and FPGAs (ReConFig), IEEE, 2005*.

[6] Mohammed Bahoura, Chan-Wang Park, "FPGA-Implementation of High-Speed MLP Neural Network". *Proceedings of The 18th International Conference on Electronics, Circuits and Systems (ICECS), IEEE, 2011*.

[7] G.M Tornez, F.Gómez, J.A. Moreno, L.M. Flores "FPGA development and implementation of a solar panel emulator". *In 10th International on Electrical Engineering, Computing Science and Automatic Control CCE 2013*.

[8] Liu Shcushan, Chen Yan, Xu Wenshang, Zhang Tongjun, "A single layer architecture to FPGA implementation of BP artificial neural network ". *2nd International Asi Conference on Informatics in Control, Automation and Robotics (CAR), 2010*.

[9] A. Savich, M. Moussa, S. Areibi "The impact of arithmetic representation on implementing MLP-BP on FPGAs: A study". *IEEE Transactions on Neural Networks, Vol.18, No.1, January 2007*.

Table 3. Performance comparison

Reference	Architecture	Precision	MSE	Protocol	Focus	DSP’s
This work	2-5-2-1	16-bit fixed-point	0.00143	Batch	Serie	11
[5]	5-3-3	16-bit fixed-point	0.03	stochastic	Parallel	--
[4]	2-6-3-2	16-bit fixed-point	0.05	On-line	Pipelined	--
[6]	1-2-1	18-bit fixed-point	--	On-line	Pipelined	14