Journal of Universal Computer Science, vol. 14, no. 10 (2008), 1678-1694
submitted: 11/12/06, accepted: 30/4/08, appeared: 28/5/08 © J.UCS

An IP Core and GUI for Implementing Multilayer Perceptron with a Fuzzy Activation Function on Configurable Logic Devices

Alfredo Rosado-Muñoz
(GPDS. Dpt. Electronic Eng. University of Valencia, Burjassot, Valencia, Spain
rosado@uv.es)

Emilio Soria-Olivas
(GPDS. Dpt. Electronic Eng. University of Valencia, Burjassot, Valencia, Spain
emilio.soria@uv.es)

Luis Gomez-Chova
(GPDS. Dpt. Electronic Eng. University of Valencia, Burjassot, Valencia, Spain
luis.gomez-chova@uv.es)

Joan Vila Francés
(GPDS. Dpt. Electronic Eng. University of Valencia, Burjassot, Valencia, Spain
joan.vila@uv.es)

Abstract: This paper describes the development of an Intellectual Property (IP) core in VHDL able to implement a Multilayer Perceptron (MLP) artificial neural network (ANN) topology with up to 2 hidden layers, 128 neurons, and 31 inputs per neuron. Neural network models are usually developed by using programming languages, such as Matlab®. However, their implementation in configurable logic hardware requires the use of some other tools and hardware description languages, such as as VHDL. For easy migration, a Matlab Graphical User Interface (GUI) to automatically translate the ANN architecture to VHDL code has been developed. In addition, the use of an activation function based on fuzzy logic for the implementation of the MLP neural network simplifies the logic and improves the results. The environment was tested using a typical prediction problem, the Mackey-Glass series, where several ANN topologies were generated, tested and implemented in an FPGA. Results show the excellent agreement between the results provided by the software model and the hardware implementation.

Keywords: neural networks, multilayer perceptron, fuzzy logic, VHDL, FPGA, programmable logic, configurable hardware, IP core
Categories: B.6.3, B.7.1, C.1.3, I.2.3

1 Introduction

There exist different Neural Network types and different hardware implementations [Omondi, 06]. The Multilayer Perceptron (MLP) is the most frequently used artificial neural network (ANN) due to its ability to model non-linear systems and establish non-linear decision boundaries in classification or prediction problems. Furthermore, the MLP is a universal function approximator, which makes it a powerful tool in

several signal processing fields: pattern recognition, system modelling, control, etc [Haykin, 99], [Bishop, 96]. The MLP-ANN is formed by individual elements known as neurons, organized in highly interconnected layers, where all the neurons in one layer connect their output to all the neurons in the next layer.

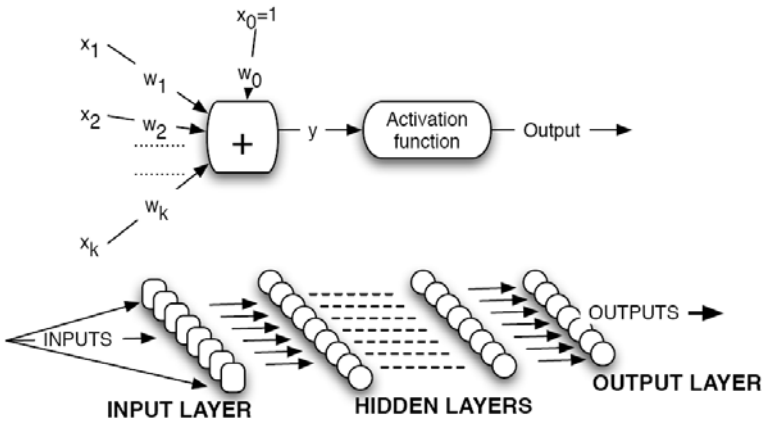

Figure 1: Scheme of a neuron and an MLP-ANN

Figure 1 (top) shows the internal structure of a neuron, in which the inputs are combined as follows:

$$y = w_0 + \sum_{i=1}^{N} w_i \cdot x_i \qquad (1)$$

where $\mathbf{x_i}$, i = 1,2,…,N, are the inputs to the neuron, $\mathbf{w_i}$ are the synaptic weights ($\mathbf{w_0}$ is called the bias, i.e. a threshold value), **y** is the linear combination of the inputs (multiplied by the synaptic weights) and the bias. The activation function (non-linear function) is the processing element mapping the **y** value to a pre-specified range. Figure 1 (bottom) shows the general view of an ANN, organized in different layers, introducing the inputs to all the neurons in the first layer and connecting all the outputs of one layer to all the neurons in the next layer.

The non-linear characteristics of the ANN come from the activation function, which is essential for proper operation of the neural network. The activation function must follow one basic property: it must be differentiable. Most non-linear functions satisfy this condition, but usually they are difficult to implement in hardware. Therefore, the main problem in hardware implementation of ANN is the non-linear activation function, because complex hardware needs to be developed to accurately represent such function. Many non-linear activation function implementations have been proposed in the bibliography, but most have problems. For example, the activation function could be approximated by several piecewise linear functions, but this solution is not differentiable. Another possibility is to use the Taylor series of the activation function [Mendil, 99], but its digital implementation leads to a loss of precision due to the finite bit-width of data.

Usually, the development and simulation of new algorithms is carried out using Personal Computer (PC) platforms and high level programming languages such as Matlab®. This fact makes it difficult to migrate the developed algorithms to specific hardware platforms for real-time applications, low power systems, portable, or any other specific need for these algorithms to run in an optimized platform where a PC is not a possibility because of the execution speed, size, external signal interface, etc.

Artificial neural networks are intrinsically parallel, as all the neurons work at the same time and there exist massive interconnections, consuming a high number of logic resources for computing and interconnecting. As it is not an easy task, hardware implementation of ANN has been a hot topic for many years, mainly due to three aspects [Muthuramalingam, 08], [Granado, 06]:

1. Accuracy: Integer operations use less space and run faster than floating point. It is usual to force integer implementations at the expense of precision and accuracy. The main focus is obtaining similar results with integers than floating point.
2. Required space: There exists a trade-off between speed of operation and logic resources consumption. If used resources are low, mathematical operations must be done in a serial fashion, requiring numerous clock cycles to perform calculations.
3. Processing speed. For fast generation of results, it is desirable to perform operations in parallel. Thus, a few clock cycles are required at the expense of high logic resources usage.

Usually, only neural chips using microelectronic VLSI implementation were successful due to the amount of silicon area required, in some cases using analogue integrated circuits [Bo, 96]. However, during the last years, the capacity of reconfigurable devices has increased significantly [Zhu, 03], allowing the implementation of ANN structures in FPGA, which are widely used for rapid prototyping due to their excellent cost/performance ratio.

Usually, hardware design of ANN in FPGA is done for specific applications [Cheung, 06], [Krips, 02]. However, since the design of an ANN in hardware complex, it is desirable to be able to generate different ANN hardware systems with a few modifications in the code [Torres-Huitzil, 07]. This direction has been explored [Bougrain, 08] in order to ease the work of migrating an ANN to specific hardware, but making a general approach is difficult and is not easy to link the software side with the hardware implementation side.

In [Soria, 03], the authors proposed an expression of the activation function based on fuzzy logic and developed the algorithm for a single neuron system under Matlab. This paper develops an appropriate MPL-ANN hardware implementation taking advantage of the proposed activation function, which uses basic logic functions such as comparators, sign function, and arithmetic calculations like addition and multiplication. In particular, we focus on optimizing the performance leading to reduced logic usage and proposing an interface to easily migrate a Matlab-designed MLP neuronal system into VHDL code for straightforward hardware implementation. In order to achieve this goal, a generic hardware implementation of MLP neural networks using a parametric IP core and VHDL is proposed. This strategy guarantees implementation in any configurable hardware platform and provides further

integration with any other hardware functions and IP cores by using a simple data communication scheme. Using an IP core scheme, hardware independency and easy parameterization are achieved. This paper provides new results in terms of speed of operation, utilization of resources, and ease of use.

The paper is outlined as follows. Section 2 shows the theoretical basis of the proposed activation function and ANN to be used. Section 3 deals with the description of the VHDL-IP core module, including the Matlab Graphical User Interface (GUI) to convert the ANN model obtained with Matlab into VHDL code. Finally, section 4 presents the results of a specific problem of time-series prediction, a chaotic series, leading to a comparison of several topologies and showing the implementation results for an FPGA device.

2 Theoretical Development

The most usual activation function in an ANN is the hyperbolic tangent due to the fulfilment of certain characteristics:

1. Historically, the sign function has been widely used in pattern recognition to separate between two patterns. Hyperbolic tangent is an excellent approximation for the sign function.
2. Opposite to the sign function, hyperbolic tangent is differentiable. This fact allows the usage of an essential weight update rule, the delta rule [Haykin, 99].
3. The usage of the hyperbolic tangent derivative is used to update weights in an optimized way.
4. With respect to other functions, hyperbolic tangent provides a null mean value.

However, all the advantages of this function are difficult to implement in hardware due to its complexity [Mendil, 99]. One of the most important drawbacks of implementing digital neural networks lies in the activation function. Different approaches have been proposed to substitute the traditional hyperbolic tangent as activation function [Ferreira, 07]. The activation function proposed by the authors in [Soria, 03] is based on modelling the hyperbolic tangent function using linguistic variables [Klir, 97]. The linguistic variables used are shown in figure 2.

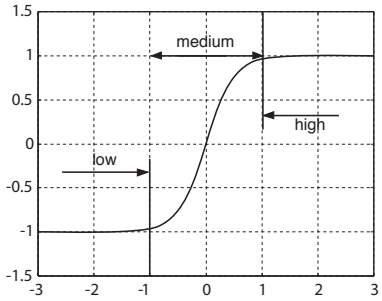

Figure 2: Linguistic variables used for modelling the hyperbolic tangent

According to figure 2 and the membership functions, the functions to be applied with x as a variable are expressed in equation 2,

$$f(x) = \begin{cases} -1 & when \quad x \ low \\ a \cdot x & when \quad x \ medium \\ 1 & when \quad x \ high \end{cases} \tag{2}$$

where *a* is a constant factor representing the smoothness of the sigmoid at origin.

It is important to note that the proposed modelling is different from the hyperbolic tangent only in the mid-segment, where linear modelling is proposed. In the context of fuzzy logic, if we suggest *x* as a linguistic variable, this will be defined by a series of membership functions [Klir, 97]. We will consider triangular functions due to their simplicity (figure 3) [Soria, 03]

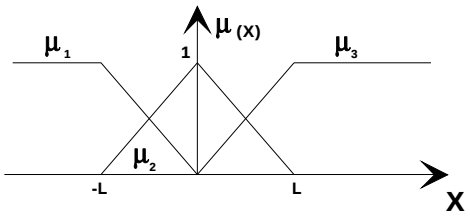

Figure 3: Membership functions used

The membership functions μ_1, μ_2 and μ_3 refer to the *low*, *medium*, and *high* concepts, respectively (figure 3). Applying a Sugeno model [Klir, 97], the value of the function at a specific point, x_o, is given by:

$$f(x_0) = (-1)\mu_1(x_0) + a\mu_2(x_0)x_0 + (+1)\mu_3(x_0) \tag{3}$$

The function given in (3) must meet the continuity property at the end of the linguistic variables defined in (2) and figure 3. Thus, in $\pm L$ points, due to the definition given in (2), the function values are ± 1 for the high and low values respectively, and additionally, the function in $\pm L$ must be $a \cdot (\pm L)$ for the medium values, leading to $a = (1/L)$. Using this expression, the fuzzy sets defined in (2), figure 3 and expression (3) [Soria, 03], expression (4) is obtained.

$$f(x) = \begin{cases} sign(x) & \text{if } |x| > L \\ \dfrac{-x \cdot |x|}{L^2} + \dfrac{2 \cdot x}{L} & \text{otherwise} \end{cases} \tag{4}$$

In terms of simplicity, the obtained function provides lower computational cost than the hyperbolic tangent with the same properties and results. This function leads to simpler hardware implementation, allowing more neurons to fit into the same

hardware capacity or incrementing the performance.

3 VHDL IP core for ANN architecture generation

Hardware implementation of the described neural network is carried out through an IP module specifically designed in VHDL for the task. The generated IP code is oriented to obtain a high parallelism among neurons maintaining moderate logic resources usage. In the proposed structure, every neuron performs internal calculations in a serial fashion so that logic resources do not increase dramatically, but every neuron makes calculations in parallel. This IP hardware module has evolved from previous work carried out by the authors in this field [Rosado, 98].

The IP core is composed of nine VHDL components corresponding to a hierarchical design where functional units are instantiated iteratively according to the parameters specified as *"generic"* in the VHDL code. This method allows dynamic generation of the hardware system according to the parameters entered in the IP core (figure 4).

```
entity ANN is
GENERIC (num_inputs_net  : integer := 2; -- number of inputs
       num_layers    : integer := 1; -- 1 or 2 (only hidden layers)
       num_neurons_layer1   : integer := 3; -- neurons of 1st layer
       num_neurons_layer2   : integer := 0; -- neurons of 2nd layer
       num_neurons_output   : integer := 1; -- neurons of out-layer
       width_weights_layer1 : integer := 2;
-- number of bits required to encode number of weights in layer 1
       width_weights_layer 2: integer := 0;
-- number of bits required to encode number of weights in layer 2
       width_weights_output : integer := 2        );
-- number of bits required to encode number of weights in outputs
PORT (NETIN      : in vector_inputs (num_inputs_net downto 1);
      WEIGHT     : in dat_int;  -- weight value, default 18 bit
       ID_NEURON : in std_logic_vector (13 downto 0);
    --(13..12)->Mode_neuron;(11..5)->num_neuron;(4..0)->num_weight;
       CLOCK, RESET  : in std_logic;
       ENA           : in std_logic;
       ENA_WEIGHT    : in std_logic;
    -- ENA_WEIGHT must be '1' during weight update
       ENDCAL        : out std_logic;
       NETOUT : out vector_inputs (num_neurons_output-1 downto 0));
end ANN;
```

Figure 4: Top Level entity structure of IP core allowing the definition of parameters for generation of different MLP-ANN

Due to the proposed parameters and corresponding ranges, the developed IP core can be fully customised in order to be adapted to almost any application requiring a Multilayer Perceptron neural network topology where real-time hardware is needed. A VHDL package has been defined in order to easily adapt the design to other needs; the modification of these values would allow changing the bit-width of input data and the maximum number of allowable inputs and neurons. Neural network configuration parameters are:

1. Number of data inputs, i.e. variables for solving the classification or regression problem. A maximum of 31 inputs is allowed. By default, 16 bit width is used.
2. Number of hidden layers. Typical problems where MLP-ANN are used do not need more than 2 hidden layers, thus only one or two hidden layers are allowed.
3. Number of outputs. In this case, any number of outputs (up to 31) can be selected.
4. Resolution of inputs and outputs. As a default, 16 bit width is used; this width is normalized to [-1, +1] values in order to be compatible with computer generated values for training and weighting.
5. Resolution of weights. Typically, weights might have normalized values greater than unity in absolute value. Then, 18 bit width is chosen as a default in order to allow values ranging from [-3, +3].
6. Specification of predefined weights and activation function for each neuron is possible but not compulsory. i.e., these values can be pre-programmed and/or fixed through the external interface once the system is running.

Once the aforementioned parameters are chosen, the hardware implementation is carried out using the synthesis tool for the target hardware, generating the specified number of neurons and their associated interconnections. This is done due to a general VHDL description based on the "*generate*" instruction, allowing automatic and iterative generation of neurons and corresponding interconnections. Figure 5 shows an example of layer 1 automatic generation according to parameters. Doing this allows the generation of all the neurons in parallel, which means that calculations of every neuron will be performed at the same time, increasing global performance.

```
-- Layer 1 generation
G2: if (k < num_neurons_layer1) generate begin
 neur_in : neuron generic map (num_inputs => num_inputs_net,
                               width_weights => width_weights_layer1)
                  port map(  INPUTS => NETIN,
                             WEIGHT => WEIGHT,
                             MODE_NEURON => mode_neuron,
                             NUM_WEIGHT => num_weight,
                             CLOCK => CLOCK,
                             RESET  => RESET,
                             ENA => ENA,
                             ENA_WEIGHT => ena_weight_int(k),
                             ENDCAL => endcal_int(k),
                             OUTPUT => out_int(k)  );
end generate G2;
```

Figure 5: VHDL code for the automatic generation of first layer in an ANN

The neural network has two modes of operation: Configuration and Operation. When working in *operation mode*, data inputs are read, calculations performed and the output is obtained. During *configuration mode*, weights and activation function for each neuron are programmed, overriding previous values and allowing to solve a different problem with the implemented ANN. Through this interface (the *port*

description in figure 4), the activation function of each neuron can also be specified according to three predefined functions: fuzzy, sign, and transparent function.

The external inputs and outputs of the neural network interface are shown in figure 4. These signals can be divided into two groups: configuration and operation mode interface.

1. *Configuration mode interface*. The proposed system does not include the possibility of self-training for weight updating, requiring external connection in case of modification of weights. This is done using a group of signals containing weight and neuron configuration information by an address and data. This set of signals is intended for the connection of a host system in charge of downloading the weight values and activation function of each neuron. The host system can be a microprocessor, a PC, or an external memory (RAM or ROM) where values are stored and can also be easily modified. The configuration interface consists of a 32-bit bus as default, allowing easy interfacing with standard 32-bit microprocessors. This group of signals can be divided into two sets: a) WEIGHT (18 bit), which specifies the value of certain weight, and b) ID_NEURON (14 bit), containing different information:
   i) The activation function for every neuron (MODE_NEURON, 2 bit, selecting fuzzy “00”, sign “01” or transparent function “11”).
   ii) Selected neuron (NUM_NEURON, 7 bit) to address the neuron to be configured.
   iii) Weight number address for the selected neuron (NUM_WEIGHT, 5 bit).

   This data structure means that a maximum of 128 neurons can be specified, and a maximum of 31 inputs per neuron is allowable. The IP core assigns a number to each generated neuron (numbers are automatically assigned from the input layer to the output layer) so that the user knows how to address the neurons of the neural network when transferring configuration. Control signals required for configuration mode are CLOCK and ENA_WEIGHT. CLOCK latches the 32-bit value into the device, storing the new weight value in an internal RAM according to the ID_NEURON indications. While ENA_WEIGHT is active, only configuration mode is enabled, ignoring all the operation mode signals.
2. *Operation mode interface*. All the inputs to the ANN are parallel, using 16 bit per input as default. Inputs are synchronized by the ENA signal, which indicates that input values (NETIN_1,…,NETIN_n) are valid and can be latched into the circuit in the next rising edge of the clock to start calculations. The ENA signal must be active during at least one clock cycle, and the neural network will not admit any other input values until the calculation of the first layer is finished. The end of calculation is indicated by the ENDCAL signal, which is active during one clock cycle, and indicates that the ANN output value NETOUT is valid; this value remains valid until a new ENA signal starts a new calculation process. All the neurons of the same layer work in parallel and each layer is calculated separatedly, allowing pipelining.

Due to pipelining, it is not necessary to wait until the end of the neural network calculation to launch a new calculation process (taking into account that a latency time appears). Once the first layer has finished and passed the values to the next layer, new data can be accepted. The delay for each layer is equal to the number of inputs plus three clock cycles; equation (5) shows the clock cycle delay Δ according to the number of layers (given in clock cycle units). Thus, due to pipelining, the minimum time for the next input to be applied would be the maximum among (n_i+3), $(n_{h1}+3)$ and $(n_{h2}+3)$, where n_i is the number of inputs to the ANN, n_{h1} is the number of neurons in the first hidden layer, and n_{h2} is the number of neurons in the second hidden layer. Therefore, a maximum sampling frequency of $1/\Delta$ could be achieved.

$$\Delta = \begin{cases} \max((n_i+3),(n_{h1}+3)) & \text{for 1 hidden layer} \\ \max((n_i+3),(n_{h1}+3),(n_{h2}+3)) & \text{for 2 hidden layers} \end{cases} \quad (5)$$

3.1 Neuron description

The neural network is based on a flexible neuron description. The neuron can be configured in multiple ways so that it can be adapted to any position in the network or to any network topology. Its input and output signals are shown in figure 6. Internally, every neuron contains a state machine to control the math calculations, an arithmetic unit consisting of a single multiplier and an addition unit, a RAM memory for weight storage and some additional logic to control and synchronize the system. As only one multiplier and one addition unit are used, serial calculations are performed internally, which greatly simplifies the logic occupation, allowing the implementation of a complex neural network in a relatively low logic area. On the other hand, several clock cycles are needed to perform calculations. With regard to the external interface, apart from INPUTS and OUTPUT, all neurons receive the same CLOCK, RESET, WEIGHT, ID_NEURON (only MODE_NEURON and NUM_WEIGHT parts) configuration signal. During configuration, only ENA_WEIGHT of each neuron is activated individually when the neuron is addressed. NUM_NEURON (the last part of ID_NEURON) is not fed into each neuron because it remains only in the main ANN structure for neuron addressing. ENA signal is the same for every neuron in the same layer; for the first layer, ENA corresponds to the external ENA input signal. For the second and output layers, the ENA signal is connected to the ENDCAL signal of the previous layer in a chained way. This procedure makes possible to have a pipeline in the system.

Additionally, the neuron description has been developed using a generic definition so that the same VHDL code can generate neurons with different input number, and thus, different weight number. The generic parameters NUM_INPUTS and WIDTH_WEIGHTS are used for this purpose as can be observed in the example given in figure 5.

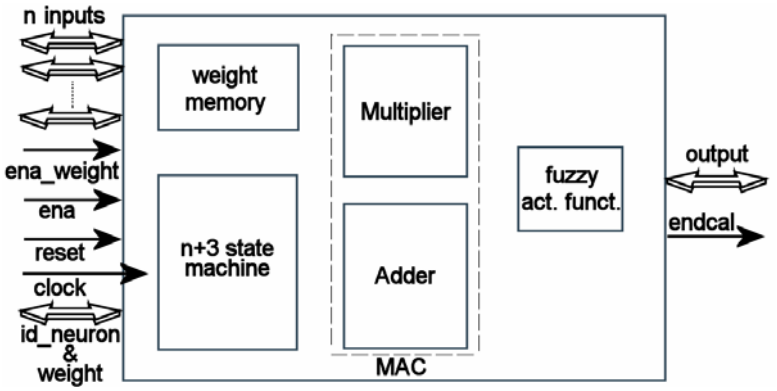

Figure 6: Input and output signals of a single neuron

3.2 Graphical User Interface

All the network configuration parameters for the MLP-ANN generation can be introduced by hand in a text file by modifying the *generic* part of the code shown in figure 4. However, in order to automate the generation of VHDL code for different neural network structures and ease the migration of Matlab designed ANN structures and weights, a Matlab Graphical User Interface (GUI) [Marchand, 02] has been built (figure 7), assisting the user in specifying the ANN structure and generating the VHDL code for both the neural network implementation (by automatic fulfilment of the generic part of figure 4), weight initialisation (this part is optional) and the VHDL simulation test bench. Basically, The GUI helps the user in setting the appropriate parameters of the IP core for the generation of the VHDL neural network architecture, number of inputs, number of layers, number of neurons per layer, weight values, etc. The GUI can also generate a data test file including initialization weights and activation function for each neuron, which can be used for VHDL simulation purposes and initial ANN values. The GUI can generate the VHDL parameters for the neural network structure and configuration values from a Matlab *.mat* file containing a previously trained neural network or from a neural network generated by the user through the graphical interface. Furthermore, for testing purposes, the general VHDL code is able to read input data files generated from the Matlab GUI, allowing VHDL simulation to test the automatically generated VHDL code. Once simulated, the results can also be compared with Matlab generated results in order to evaluate the accuracy of the hardware implementation (see section 5).

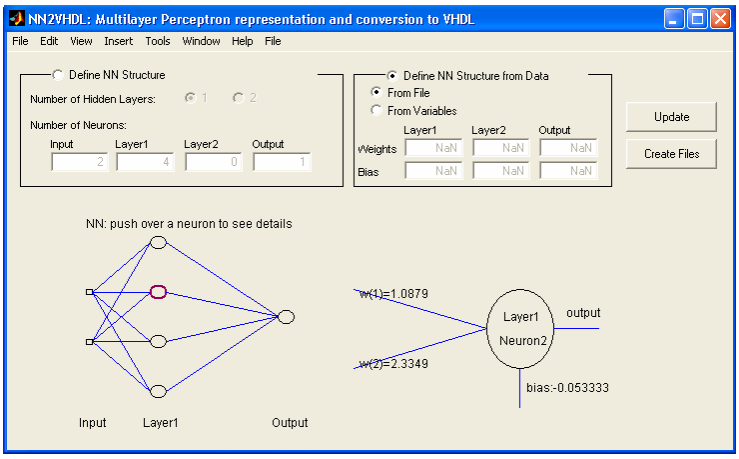

Figure 7: Matlab GUI interface for the VHDL code generation

4 Implementation results

For comparison purposes, different neural network structures were synthesized for a Xilinx Spartan 3 FPGA using XST under Xilinx ISE 9 software. Table 1 shows implementation results for different ANN structures containing several neurons, from a single neuron with two inputs to a complex ANN with eight inputs, two hidden layers with 28 neurons each one and a single output neuron, all of them implemented in the same Xilinx Spartan 3 device XC3s4000fg900-4. Table 1 gives a comparison between different ANN structures in logic occupation and speed of operation, where ***ANN W-X-Y-Z*** stands for W:number of inputs, X:number of neurons in the first layer, Y:number of neurons in the second layer, Z:number of neurons in the output layer.

As it can be seen from the number of slices (basic measure of logic occupation in Xilinx devices), the occupation level of a neural network is proportional to the number of neurons; a neural network with 5 neurons occupies 899 slices, which gives a ratio of 179 slices per neuron, a value similar to the occupation level of a single neuron (177 slices). Every neuron requires a single multiplier; it can be seen in the "multiplier" column of Table 1, giving the total number of neurons in every ANN. As the number of internal neurons and inputs grow, more slices are required and the clock speed decreases, mainly due to the exponential increment of interconnections. This table also shows that an increment in the number of inputs requires more slice usage, increasing the slices per neuron rate.

The obtained results vary slightly depending on the selected device and family. Spartan 3 is a low cost device, with the simplest device (XC3s50) containing 768 slices, allowing a 4 neuron ANN to be implemented. If we consider that a typical classification or regression problem requires no more than 30 neurons, a medium size, moderate cost device in Spartan 3 family such as XC3s1000 could fit the design running at a 65MHz clock rate. Other FPGA families such as Virtex 4 or Virtex 5 contain up to 89,000 slices, meaning that an approximate number of 300 neurons could be implemented into a single FPGA (the exact number of neurons depends on

the number of inputs and outputs). However, the present implementation can use a maximum of 128 neurons, requiring less logic area than those offered by devices.

ANN W-X-Y-Z	device slices	multipliers	clock freq.	sampling freq.
Single neuron	177	1	71.8 MHz	14.36 MHz
ANN 2-4-0-1	899	5	71.8 MHz	10.26 MHz
ANN 4-8-8-1	3538	17	71.8 MHz	6.55 MHz
ANN 6-15-15-1	7973	31	65.7 MHz	3.65 MHz
ANN 8-28-28-1	19810	57	61,8 MHz	501 kHz

Table 1: Occupied resources and speed of operation for different ANN structures implemented in a Spartan3 XC3s4000 device

Regarding speed of operation, Table 1 shows the clock speed that could be achieved by different ANN structures. As observed, a data input sampling frequency in the order of MHz can be used for most of the ANN structures; however, most of the ANN applications do not require such high frequencies.

Additionally, if implementation were made in last generation Xilinx Virtex5 device, a complex ANN implementation with 30 inputs, two layers 31 neurons each, and 31 outputs would require a medium-size device (for example the xc5vlx110ff1760). This ANN implementation would run at a 73MHz clock frequency (2.1 MHz sampling rate).

5 Application results

As stated before, logic occupation, performance, and precision are the most important topics when implementing algorithms in specific hardware. Logic occupation and performance were analyzed in section 4. In order to test the precision of the proposed system, several neural network topologies were generated to solve a typical time series problem: the one-step forward prediction of the Mackey-Glass series generated according to the non-linear differential equation given in equation 6.

$$\frac{dx(t)}{dt} = -0.1 \cdot x(t) + \frac{0.2 \cdot x(t-\tau)}{1 + x^{10}(t-\tau)} \qquad \text{for } \tau = 17 \qquad (6)$$

The problem consists in predicting next input value *x(t+1)* from actual and previous values *x(t)*, *x(t-1)*, *x(t-2)*, etc. This problem was chosen because Mackey-Glass is a chaotic series with a high non-linearity, which is a non-simple prediction problem. Optimized non-linear models, like neural networks, may give a prediction error that is three orders of magnitude lower than the error given for an optimized linear model [Svarer, 83].

The main aim of using this prediction series was the comparison of results between the same ANN model for an implementation in software (MATLAB) and hardware (FPGA using VHDL) to evaluate accuracy. The input time series has been processed so that it meets zero mean and a standard deviation of 1. The number of inputs was taken as two; thus, the problem lies in predicting *x(t+1)* from *x(t)* and *x(t-*

1). A set of 1200 points, without the validation set, was taken. For comparison purposes, different ANN consisting of two input neurons, a variable number of hidden neurons and a single output neuron were implemented.

The quality of the modelling can be deduced from figure 8, which shows the Mean value of the Absolute Error (MAE) and the Root Mean Square of the Error (RMSE) of the model for different number of neurons in the hidden layer. These error estimations are calculated between the theoretical Mackey-Glass series (equation 6) and the Matlab simulated ANN model prediction output. As it can be seen, the prediction problem is not easy and even using the floating point precision calculus made with Matlab in a PC, the ANN shows significant errors.

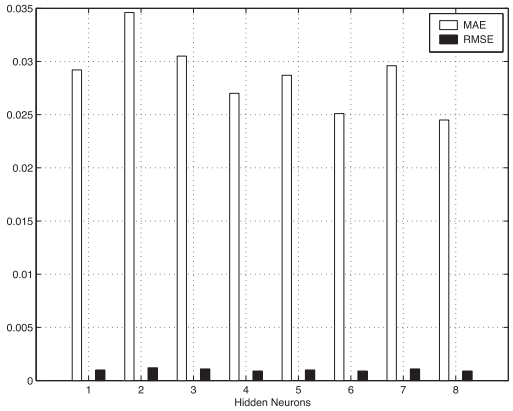

Figure 8: MAE and RMSE of the difference between the theoretical Mackey-Glass series and the ANN model in Matlab.

With respect to the hardware implementation, once the ANN was trained in Matlab, weight values and ANN structure were translated into VHDL code using the proposed GUI tool. Figure 9 shows Modelsim VHDL simulation results for a two input (*x(t)* and *x(t-1))* ANN with one hidden layer containing three neurons and one single output (*x(t+1)).* All the external signals and configuration parameters are shown in this figure. In the simulation diagram, a 10 MHz clock is used. After the initial reset, predefined weights are introduced in the system through the external configuration interface (ENA_WEIGHT signal active), followed by the start of the operation mode, first input set is sampled (ENA signal active) and the output result for the prediction is generated. The procedure of entering new inputs and obtaining the corresponding prediction is repeated. Figure 9 also shows the first output result generated 12 clock cycles after ENA signal is activated. This time is reduced to 6 cycles after the second sample, due to the pipeline system.

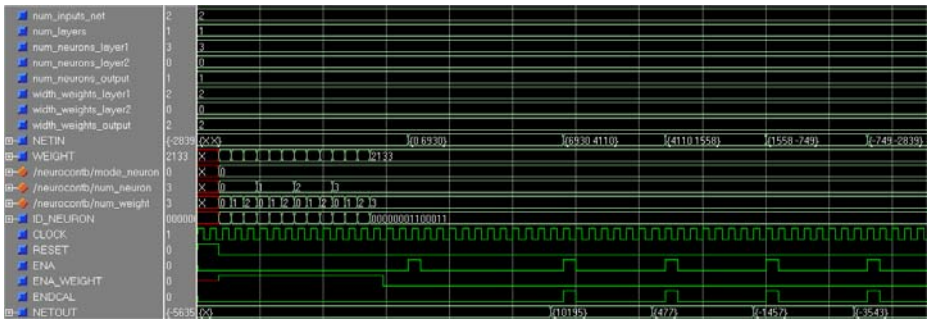

Figure 9: VHDL Simulation results for the initialisation of weights and output calculation for a 2 input, 3 neuron, 1 output ANN

For the specific neural network consisting of two inputs, one hidden layer with 3 neurons and one output neuron, figure 10 shows the results obtained for a test set of 200 input samples: the ideal signal for the Mackey-Glass series according to equation 6 (Ideal), the prediction obtained with the proposed ANN in Matlab floating point precision (Matlab), and the prediction result of the hardware implemented network in 16 bit precision (FPGA). Simulation results are stored in a file so that they can be compared with the Matlab® generated results.

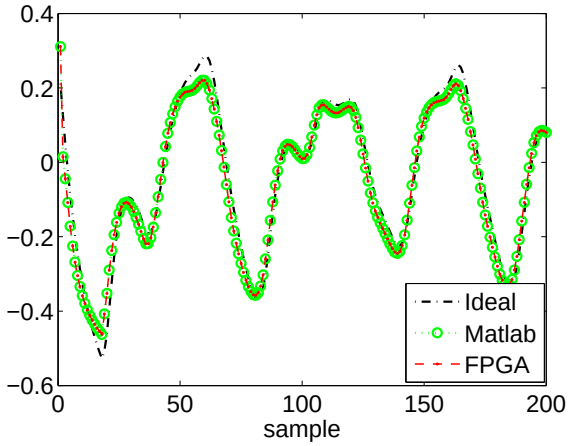

Figure 10: Comparison among Mackey-Glass series equation (Ideal), prediction results obtained by the ANN in Matlab (Matlab) and ANN hardware implementation (FPGA)

In order to evaluate the error obtained by the hardware implementation, some parameters were measured. Table 2 shows the MAE and RMSE of the difference between results obtained by Matlab using floating point precision and the hardware implementation of the ANN using signed 16 bit integer.

Hidden Neurons	3	4	5	6	7	8	9
MAE	6,29E-06	4,45E-06	2,20E-06	1,03E-05	9,91E-06	4,96E-06	9,32E-06
RMSE	2,08E-04	1,47E-04	6,16E-05	3,46E-04	3,29E-04	1,62E-04	3,03E-04

Table 2: Mean value of the MAE and RMSE of the difference between the results obtained by Matlab and the hardware implementation ANN prediction

The hardware implementation used signed 16 bit precision, which means that the quantization error is 3.052e-5. However, as shown in figure 8 and table 2, the errors due to hardware implementation (table 2) can be neglected compared with the difference due to the ANN modelling and Ideal series (figure 8). It is noticeable that the error of the hardware implementation is about 2 orders of magnitude lower than the error of the modelling.

Regarding the behaviour of the modelling for different number of neurons, the error magnitude is quite similar for all the structures, despite the number of hidden neurons. It is also important to note that a greater number of neurons do not necessarily means better results (lower error) for the prediction. In that sense, table 2 shows that independently of the number of neurons, the comparison error between hardware configurable logic and Matlab is always kept at a low value.

6 Conclusions

The proposed system offers new advantages and eases the hardware implementation procedure of a complex system such as neural networks. Due to the new proposed fuzzy activation function, it is possible to keep hardware consumption low, allowing a parallel calculation system for the neurons in the same layer and increasing performance without sacrificing logic occupation.

The proposed system also includes a graphical interface to translate Matlab generated neural networks into VHDL making use of the automatic IP core parameter configuration. Very little knowledge of VHDL is needed to generate fully functional hardware neural networks, because the developed IP core automates the conversion.

In addition, due to the pipelining and parallelization of neuron calculation in the same layer, the implemented hardware system is able to run at clock frequency rates up to 70.1 MHz, which for the described example would allow a 10.7 MHz data sampling rate. Thus, using the IP core proposed in this paper, MLP neural network systems can be easily migrated from software design environments such as Matlab into specific hardware platforms (configurable hardware) for real time applications in a few steps, achieving an excellent ratio between logic resources and speed of operation, maintaining the accuracy.

This work is self contained and can be used as it is. Currently, the work intends to provide the same tool for other types of ANN, not only Multilayer Perceptron, and moreover, introduce the possibility of incorporating the learning process in the hardware, which would increase the range of applications where an ANN hardware system could be used.

References

[Bellis, 04] Bellis, S., Razeeb, K. M., Pounds-Cornish, A., Argyropoulos, C., et al.: FPGA Implementation of Spiking Neural Networks - an Initial Step towards Building Tangible Collaborative Autonomous Agents, International Conference on Field Programmable Technology (FPT'04), 2004.

[Bishop, 96] Bishop, C.M.: Neural Networks for Pattern Recognition, Oxford: Clarendon Press, 1996.

[Bo, 96] Bo, G.M., Caviglia, D.D., Valle, M.: Current Mode CMOS Multi Layer Perceptron Chip, 5th International Conference on Microelectronics for Neural Networks and Fuzzy Systems (MicroNeuro '96), pp 103, 1996.

[Bougrain, 08] Bougrain, A., Mozzati, C., Dondelinger, E., Tonnelier, M.: DynNet and GINNet projects: Library and Graphical Interface for knowledge extraction from data, INRIA-LORIA, France, 2008, http://ginnet.gforge.inria.fr/#overview

[Cheung, 06] Cheung, O.Y.H., Leong, P.H.W., Tsang, E.K.C., Shi, B.E.: A Scalable FPGA Implementation of Cellular Neural Networks for Gabor-type Filtering, 2006 International Joint Conference on Neural Networks, pp 15-20, 2006.

[Ferreira, 07] Ferreira, P., Ribeiro, P., Antunes, A., Morgado-Dias, F.: A high bit resolution FPGA implementation of a FNN with a new algorithm for the activation function, Neurocomputing, vol 71, no 1-3, pp 71-77, 2007.

[Granado, 06] Granado, J.M., Vega, M.A., Pérez, R., Sánchez, J.M., Gómez, J.A.: Using FPGAs to Implement Artificial Neural Networks, 13th IEEE International Conference on Electronics, Circuits and Systems, ICECS'06, pp 934-937, 2006.

[Haykin, 99] Haykin, S.: Neural Networks: A Comprehensive Foundation, New Jersey: Prentice Hall, 1999.

[Klir, 97] Klir, G.J., Clair, U.H., Yuan, B.: Fuzzy Set Theory, New Jersey: Prentice Hall, 1997.

[Krips, 02] Krips, M., Lammert, T., Kummert, A.: FPGA Implementation of a Neural Network for a Real-Time Hand Tracking System, Proceedings of the First IEEE International Workshop on Electronic Design, Test and Applications (DELTA'02), pp. 1453-1457, 2002.

[Marchand, 02] Marchand, P., Holland, O.T.: Graphics and GUIs with MATLAB, Boca Raton: CRC Press, 2002.

[Mendil, 99] Mendil, B., Benmahammed, K.: Simple Activation Functions for Neural and Fuzzy Neural Networks. Circuits and Systems. ISCAS' 99, vol 5, pp 347-350, 1999.

[Muthuramalingam, 08] Muthuramalingam, A., Himavathi, S., Srinivasan, E.: Neural Network Implementation Using FPGA: Issues and Application, International Journal of Information Technology, vol 4, no 2, pp 86-92, 2008.

[Omondi, 06] Omondi, A.R., Rajapakse, J.C.: FPGA implementations of Neural Networks, Springer Netherlands, 2006.

[Rosado, 98] Rosado, A., Bataller, M., Soria, E., Calpe, J., Francés, J.V.: A new hardware architecture for a general-purpose neuron based on distributed arithmetic and implemented on FPGA devices, Engineering of Intelligent Systems Conf. EIS'98, pp 321-326, 1998.

[Soria, 03] Soria, E., Martín, J.D., Camps, G., Serrano, A.J., Calpe, J., Gómez, L.: A Low Complexity Fuzzy Activation Function for Artificial Neural Networks, IEEE Transactions on Neural Networks, vol. 14, no. 6, pp. 1576-1579, October 2003.

[Svarer, 83] Svarer, C., Hansen, L.K., Larsen, J., Rasmussen, C.E.: Designer networks for time series processing, Proceedings of the III IEEE Workshop on Neural Networks for Signal Processing, pp 78–87, 1983.

[Torres-Huitzil, 07] Torres-Huitzil, C., Girau, B., Gauffriau, A.: Hardware/Software Codesign for Embedded Implementation of Neural Networks, Lecture Notes in Computer Science. Springer Berlin/Heidelberg, vol 4419/2007, pp 167-178, from the book: "Reconfigurable Computing: Architectures, Tools and Applications", 2007.

[Zhu, 03] Zhu, J., Sutton, P.: FPGA implementations of Neural Networks, a survey of a decade of progress, 13th International Conference on Field Programmable Logic and Applications, 2003.