

Automatic General Purpose Neural Hardware Generator

Fábio D. Baptista and Fernando Morgado-Dias

Abstract— Artificial Neural Networks have a wide range of applications. In some applications specific hardware is necessary when a PC cannot be connected or due to other factors such as speed, price and fault tolerance. The difficulty in producing hardware for Neural Networks is associated with price, accuracy and development time. Most users also prefer the network trained with a high level tool without reducing resolution and simplifying the activation function for hardware implementation. This paper proposes an Automatic General Purpose Neural Hardware Generator, simple to use, with adjustable accuracy that provides direct hardware implementation for Neural Networks with FPGAs without further development.

Index Terms— Artificial Neural Networks, Hardware Implementation, System Generator, Integrated Software Environment, Automatic implementation, Fixed Point Notation

I. INTRODUCTION

ARTIFICIAL Neural Networks (ANN) began as a mathematical model used to represent the behavior of animals. These models found their simplest habitat within the Personal Computer (PC). With a PC as the platform, new training algorithms and models can easily be tested, thus making it a convenient platform for development in many applications.

In some application areas of ANN, the use of a specific hardware implementation is required either because these implementations simply cannot be connected to a PC, as is the case with some applications working inside the human body, or due to other factors such as speed, price and fault tolerance, as can be found in some processing signal (speed), commercial (price) and critical (fault tolerance) applications.

A recent survey also shows that hardware implementation of ANN is in fact a necessity in the community [29].

Either as a result of this connectivity issue or due to commercial reasons, the need for a physical implementation of the ANN arose in the research conducted over the last three decades, which led to industrial solutions being proposed. Another important issue about ANN is that some of the

characteristics that leverage them cannot be achieved using sequential implementations. As stated in [1], ‘The greatest potential of neural networks remains in the high-speed processing that could be provided through massively parallel VLSI implementations’.

In 1993 the first commercial physical implementation of an ANN was developed by Nestor and Intel for an Optical Character Recognition (OCR) which uses Radial Basis Functions [2]. The need for physical implementations also arose in medical applications of ANN, which work with the human body. The work referred to in [3] contains many examples of neural vision systems, which likewise cannot be used attached to computers.

In spite of these examples, it is still an expensive and time consuming endeavor to build a hardware implementation of an ANN. This is mainly because each implementation must be tailored to the specific application in order to make good use of the hardware capacities and to respond to one’s effective needs.

ANN hardware implementations face some difficult problems, such as: implementing the activation function, the large amount of calculations utilized, and the choice of notation. The common use of the hyperbolic tangent and other sigmoid functions, though simple in terms of software, is very difficult to implement in hardware and has been the object of study of many researchers [4-11,27,30-31]. As a result of this difficulty, most of the implementations found in the literature use oversimplified activation functions [5-6, 8].

The lowest error solution can be found in [9], where the hyperbolic tangent was replaced by 25 polynomial sections of 3rd order with a Mean Square Error (MSE) of approximately 10^{-14} .

In [16] a hardware solution can be found where “the average and peak error for the sigmoid function in the domain $[-12, 12]$ are 4.82×10^{-5} and 3.36×10^{-4} ”.

The different proposals presented in [12-14] propose ANN with learning capabilities. These are relevant contributions but propose, nevertheless, simplified activation functions.

A more interesting approach in this direction was proposed in [24] where an algorithm is presented for compact neural-network hardware implementation by using a particular representation of weights and step activation functions. This solution is interesting and seems to be compact but, in its present form cannot deal with other activation functions.

In [25] an implementation for a particular ANN is presented using Simulink. This solution only allows activation function

Manuscript received June 9, 2012. The authors would like to acknowledge the Portuguese Foundation for Science and Technology for their support for this work through project PEst-OE/EEI/LA0009/2011.

F. D. Baptista and F. Morgado-Dias are with the Centro de Competências de Ciências Exatas e da Engenharia and Madeira Interactive Technologies Institute, Funchal, Portugal (phone: +351 291-705150/1; fax: +351 291-705199; email: morgado@uma.pt)

through a ROM, does not have a full parallel structure since it used Multiply and Accumulate blocks, does not allow a user to configure for an ANN and does not present any results concerning accuracy.

In [26] an implementation for Self-Organizing Maps with Simulink is proposed. This solution only applies to this type of ANN and is not configurable.

In [27] an implementation that uses Simulink can be found. This implementation uses logsig activation functions therefore avoiding the hyperbolic tangent that is the most common activation function and more difficult to implement in hardware. This solution is also not configurable.

These solutions presented in the literature, however, always have limitations when compared with their software counterparts. While the authors believe that the hardware implementation can be just another step in the design of the ANN.

New designs to implement ANN in hardware are being proposed by the research community. “Such new designs will be of use to industry if the cost of adopting them is sufficiently low” [23], as happened with the software counterparts of ANN.

As a solution to these problems, this paper proposes an Automatic General Purpose Neural Hardware Generator which is simple to use, in which the accuracy can be tailored according to the necessity or the hardware available. However the most important characteristic is that it provides a direct hardware implementation for feed-forward ANN without loss of accuracy and with several choices of solutions for the hyperbolic tangent and remaining hardware and uses a fully parallel architecture.

Experimental results show that the implementations developed, although using fixed-point notation, can replace a PC without introducing significant error.

The next section of the paper presents the motivation for this work. Section III shows the details of the neuron implementation, section IV gives the tool’s details, section V presents some of the results obtained and in section VI the conclusions are drawn.

II. MOTIVATION

It is the authors’ vision that the hardware implementation can be just another step in building an ANN, by using an automatic neural generator, as represented in figure 1.

Since none of the solutions found until now responds to this need, in this paper we present an easy bridge between the PC and the hardware implementation of ANN, namely: an Automatic general purpose Neural hardware Generator (ANGE). Briefly, this tool pursues the objective of an automatic implementation in hardware of a trained ANN, with minimal hardware knowledge. Therefore, the reader could ask himself the following: How can the user implement an ANN in hardware without knowing much of the hardware structure? The answer is very simple, ANGE has an user-friendly interface where the user inserts the structure of an ANN, number of bits, type of implementation of the activation function and uploads the files with the weights. After that,

ANGE will automatically produce the code necessary to synthesize hardware.

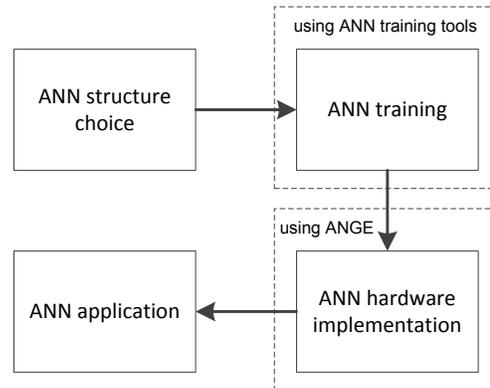


Fig. 1. ANN development tool chain.

ANGE makes use of Matlab and Simulink, System Generator and Integrated Software Environment (ISE). It implements the ANN in a Field Programmable Gate Array (FPGA), despite the fact that one of the intermediate results obtained can be used to implement the ANN in any type of platform (although not every type of manufacturer).

System Generator is a tool, which allows designs to be built up quickly while automatically generating Hardware Description Language (HDL) code, which is used for FPGA programming [17]. This solution was used because it produces code for fixed-point notation and can be integrated in an automatic tool such as ANGE, producing intermediate Simulink models.

System Generator supports FPGA hardware-in-the-loop verification for Xilinx FPGA boards. The HDL Verifier provides co-simulation interfaces, which connect MATLAB and Simulink [18]. As can be seen from figure 2 (which shows the development chain), an intermediate result is a HDL description of the circuit, which implements ANGE’s solutions. Although ANGE is targeted to FPGA, a solution written in a HDL is generic enough to allow it to be targeted to a different platform.

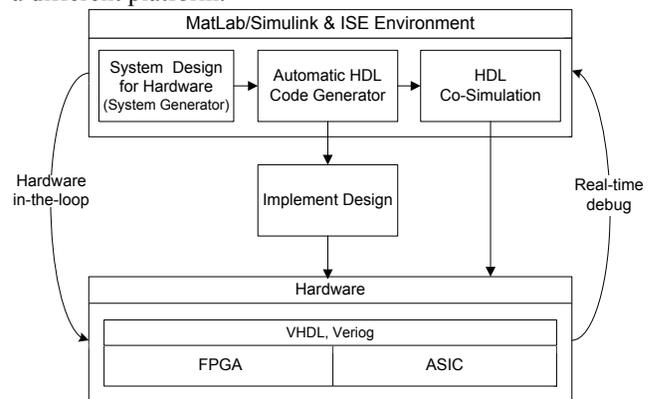


Fig. 2. FPGA development tool chain.

Matlab and Simulink tools are widely used in the ANN field whereas ISE and System Generator are tools used to work with programmable hardware. Programmable hardware is the preferred solution for a large portion of the ANN

implementations due to the acceptable price for a reduced number of copies [2]. The research described in this paper has been under development since 2009 and has produced several intermediate results, as well as a previous version of ANGE [19][28].

Comparing ANGE II to ANGE, ANGE had the simplest activation function possible, a Look Up Table (LUT). This was significantly improved in ANGE II, which offers several implementations of the activation function as well as several options regarding multipliers and binary point position.

To evaluate the quality of ANGE II, it has been tested in a control loop which illustrates its accuracy, as will be shown in section IV.

III. NEURON IMPLEMENTATION

ANGE implements Feed-Forward Artificial Neural Networks (FANN). The basic element of a FANN is the neuron, a logical-mathematical structure that seeks to simulate the shape, behavior and function of a biological neuron.

The input information is captured and processed by the sum function, and the new signal generated by the neuron is set by the activation function [20]. Hence, the neurons are based in a simple mathematical function, which can be translated analytically by the following form:

$$y = f\left(\sum_{i=1}^N I_i w_i\right) \quad (1)$$

where, I_i is the i th input of the network, w_i is the corresponding weight and f is the activation function.

In this new version, ANGE has two main modifications in the way in which equation 1 is implemented, when compared with the previous version. Figure 3 shows the principal modification operated in ANGE.

The first modification concerns the way in which the sum of products between the inputs and weights is calculated. The second modification consists of implementing different options to replace the hyperbolic tangent. These options are available in ANGE II and it is up to the user to choose the most convenient option.

A. 1st Modification – Implementation of the sum of products between the weights and inputs.

Instead of using the multiplier and adder blocks directly, this implementation uses MCode blocks.

The MCode block allows using a script similar to the Matlab language (but restricted to a small subset) inside System Generator, which is useful for implementing arithmetic functions, finite state machines and control logic [20].

Figure 4 shows a flowchart of the code's script for the Mcode block used to calculate the sum of the product of n inputs of a neuron with their respective weight. For this code, the resource use was minimized by using only one multiplier and one adder for each neuron while avoiding an additional multiplication by setting the initial sum value equal to the bias weight.

System Generator works only with fixed-point

representation. This representation reduces the resources used in the calculations; however, considering the same number of bits, it represents a smaller interval of values or can introduce a larger representation error.

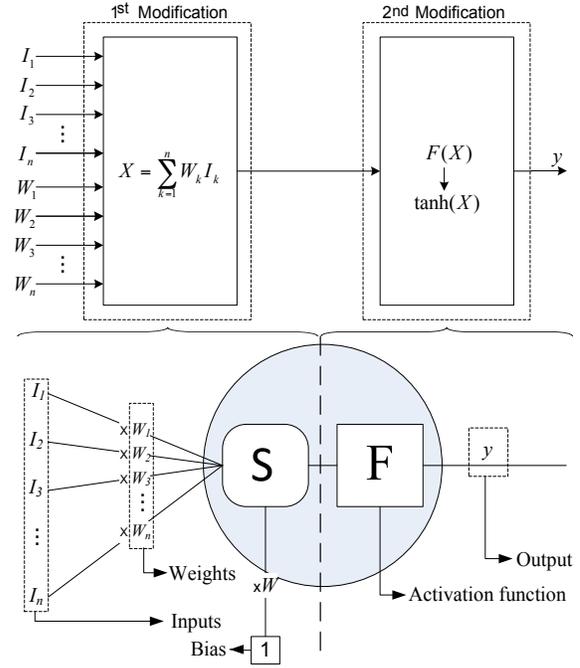


Fig. 3. Block Diagram of the changes made in the implementation of the neuron.

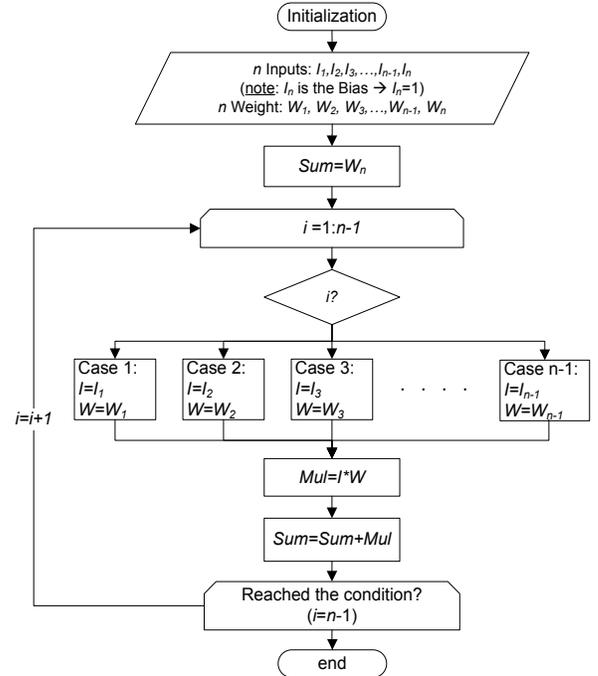


Fig. 4. Flowchart used to calculate the sum of the product of n inputs of a neuron with their respective weight.

To improve the efficiency of the neuron, there is an adjustment of the number of bits used to represent the integer part, which depends on the activation function.

When the activation function is the hyperbolic tangent, the number of bits of the integer part has a maximum of 4 bits (3 bits + 1 sign bit). However, when it is a linear function, the number of bits changes in order to obtain the quantity of bits adequate to represent the integer part. This methodology gives a better representation and precision of the result of the sum of the product between the inputs and their respective weights.

B. 2nd Modification – Implementation of the Hyperbolic Tangent

The second modification concerns the representation of the activation function, specifically the hyperbolic tangent. The implementation of the hyperbolic tangent is not possible without previous treatment. For this reason, it is necessary to analyze its parity.

The hyperbolic tangent is an odd function because it is symmetric about the origin, i.e.

$$f(-x) = -f(x) \quad (2)$$

Therefore, its implementation is held only for the positive part. After studying the behavior of this odd function, it is possible to know what happens when $x < 0$, by using the arguments of symmetry.

Three different solutions are implemented to describe the hyperbolic tangent for the domain of $x \geq 0$. The different solutions were developed to allow the management of resources of the FPGA. While one solution uses more memory, another solution uses more multipliers and a third one more memory, giving the user freedom to select according to his needs.

The first implementation of the hyperbolic tangent consists of storing, in a Read Only Memory (ROM), the function's values for input values between 0-6. For input values greater than 6, the function is set equal to 1.

This implementation using System Generator is represented in figure 5 and was developed in the first ANGE version [19].

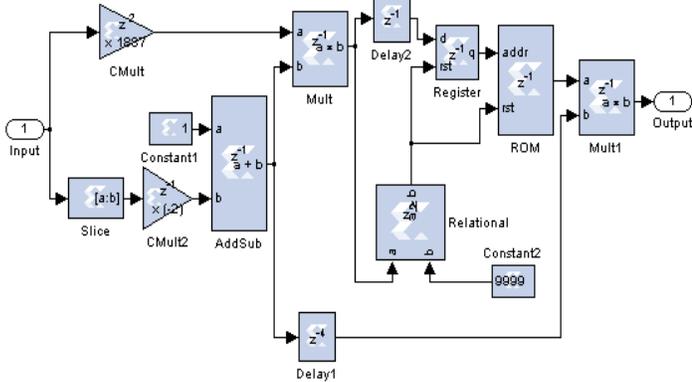


Fig. 5. Hyperbolic tangent implementation using a ROM.

The ROM is used as a Look Up Table (LUT) to store the values of the hyperbolic tangent. The inputs of the LUT are in fact the memory addresses instead of the inputs of the hyperbolic tangent. The outputs will be fixed point values and these numbers will be unsigned. For the LUT to perform the required function, several transformations are necessary. These changes are done by the subsystem represented in figure 5[19].

The second implementation of this function consists of using a polynomial interpolator. To determine these polynomials, several solutions were tested and the Chebyshev interpolation was chosen along with the implementation proposed in [9]. The domain of the function was split into smaller intervals to ensure a high accuracy. For this implementation 25 polynomials of third-degree were used, in order to describe the hyperbolic tangent with a Mean Square Error (MSE) equal to 1×10^{-14} [9]. Equation 3 describes the way in which the polynomial was implemented. This organization allows for the use of only 3 multiplications and 3 additions, which saves FPGA's resources.

$$f(x) = a + (b + (c + dx)x)x \quad (3)$$

The third implementation was achieved by using a piecewise linear approximation. This solution uses piecewise linear sections which can be described mathematically by:

$$f(x) = mx + b \quad (4)$$

where m is the slope and b is the point of intersection with the ordinate axis.

The piecewise linear implementation is not the regular implementation but is based in the solution proposed in [4].

In [4] the linear sections starts with a tangent from a point in the hyperbolic tangent function as an approximation to that function. The tangent is later replaced for another tangent starting in another point when it reaches the maximum error (see [4] for details).

In the current implementation an improvement was found: with the same maximum error instead of using a tangent, a secant can be used that starts and ends with the maximum error. It would start with the maximum allowed error, go through zero error twice and then end up with the maximum error with the inverse sign when compared to the starting point. This is illustrated in figure 6.

This solution reduces the number of linear sections, while keeping the same maximum error. Notice that the error is so small that it is difficult to distinguish the real function from the approximation and the vertical axis as such as small change that the values are depicted as the same.

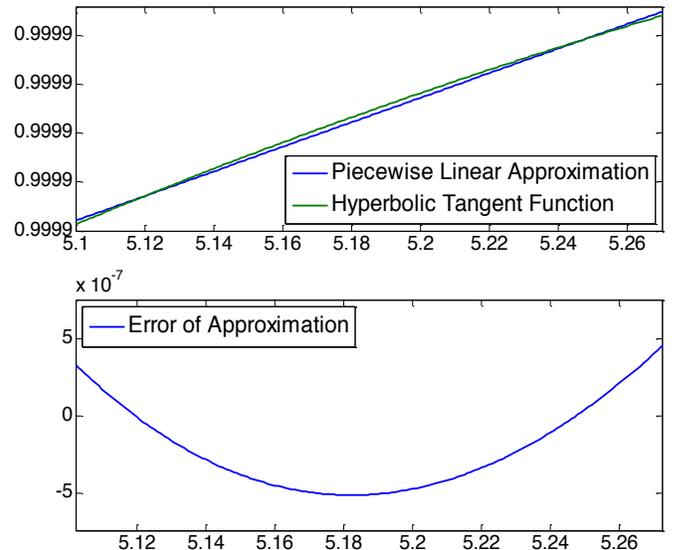


Fig.6. Approximation of an interval of hyperbolic tangent

using a piecewise linear approach.

The algorithm was also organized in such a way that the average error could be specified instead of the maximum error, and thus allowing for it to be directly compared with the polynomial approximations.

For this implementation 742 line sections were used, in order to describe the hyperbolic tangent with a MSE of approximately 1×10^{-14} .

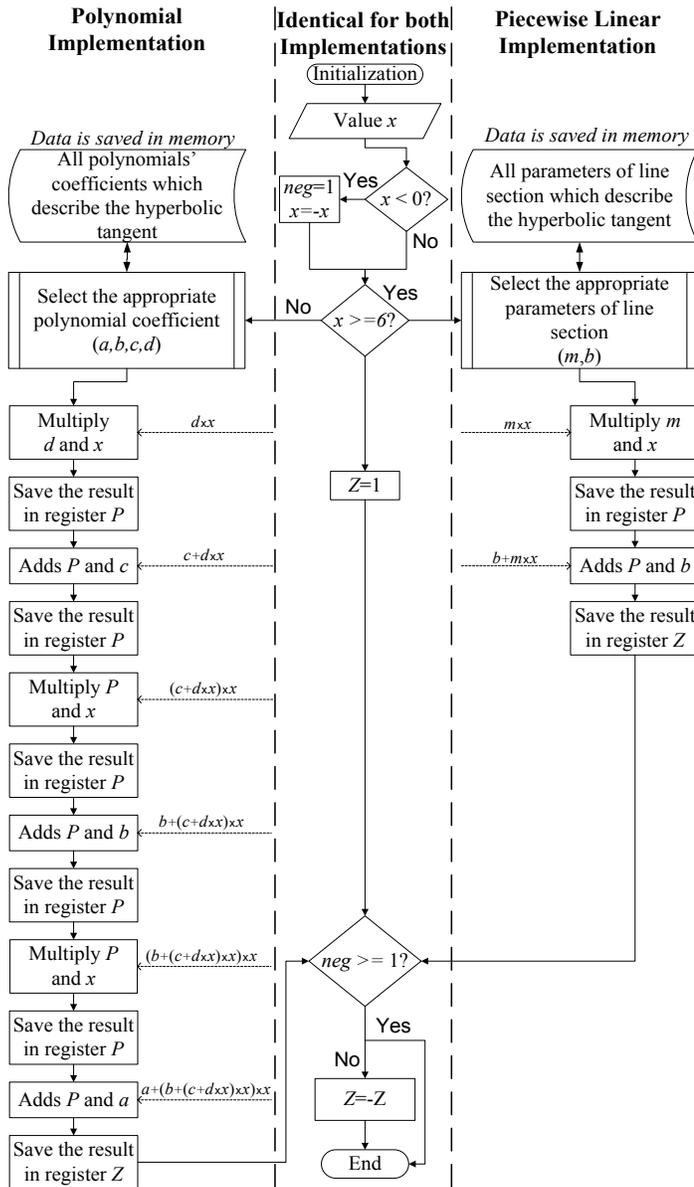


Fig. 7. Flowchart descriptive of the hyperbolic tangent function using the polynomial approximation.

The flowchart of figure 7 shows the difference between the polynomial and piecewise linear implementation.

On the one hand, it shows that to implement the polynomial, it is necessary to use 3 multiplications, 3 additions and 4 different coefficients for each polynomial. On the other

hand, in order for one to implement the piecewise linear, it is necessary to use only 1 multiplication, 1 addition and 2 coefficients for each straight line section.

The multiplication operation was implemented in three different forms: using a specific feature of the FPGA (DSP - Digital Signal Processor), using Booth's algorithm and by using a 2's complement multiplication [22].

IV. ANGE DETAILS

ANGE II is designed to work with FANN with linear or hyperbolic tangents activation functions.

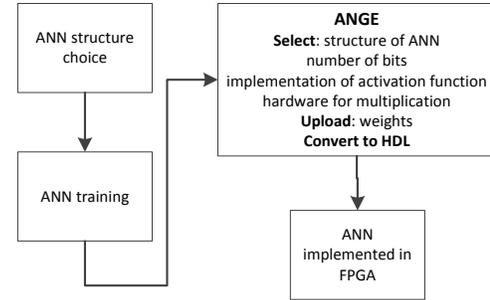


Fig. 8. Block diagram of ANGE use within the process of building an ANN.

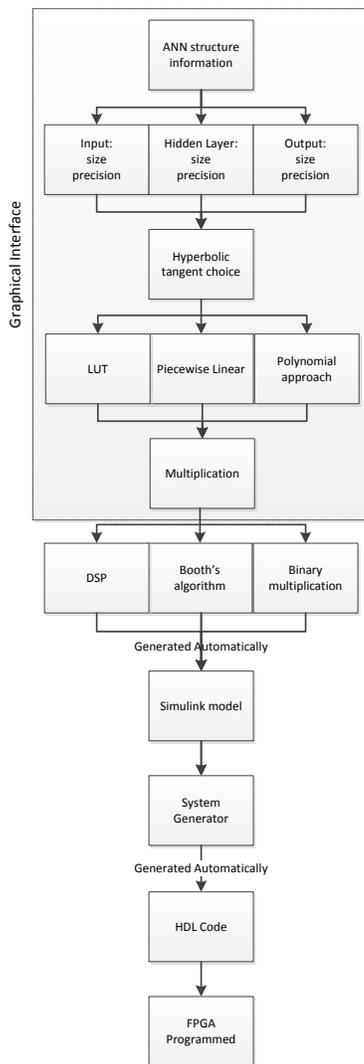


Fig. 9. Flowchart of ANGE's choice for the user.

ANGE II is presented through a simple graphical user interface, where the user can access the functional modules. Thus, the user only needs to take into account the problem to be solved, not which blocks and configurations are necessary to solve the problem.

By selecting a few parameters as shown in figure 8, the user obtains general purpose hardware implementation of its ANN.

The current version of ANGE runs over Matlab R2010a with System Generator 12.3 and ISE 12.3 and is capable of configuring hardware in a FPGA for an ANN as large as the FPGA available allows.

Figure 9 represents a flowchart of the process of programming the FPGA. The user must select a few options such as the size and precision for input, hidden layer and output, the type of replacement for the hyperbolic tangent and implementation of the multipliers. The following steps create a Simulink model that is processed by System Generator into HDL Code to program the FPGA.

In figure 10, the graphical interface is shown which illustrates the interaction between the user and the process.

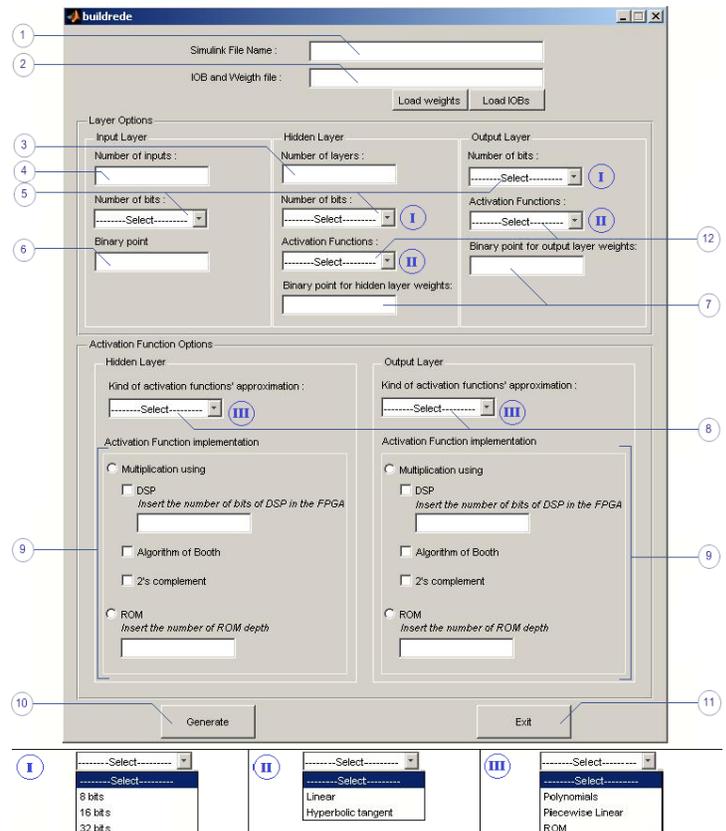


Fig 10. Graphical User Interface of the ANGE II

Each field is marked and a brief explanation of its use is presented:

1. Name of the Simulink model file in which the ANN will be generated.
2. Name of the .mat file, which contains the values of the Weights and Input Output Blocks (IOB) locations (the IOB locations indicate which pins of the FPGA will be used for the input/output of data).
3. Number of neurons to be implemented in the hidden layer of the ANN.
4. Number of inputs of the ANN.
5. Number of bits used to represent the activation function (available options: 8 bits; 16 bits; 32 bits).
6. Number of bits to be used in the fractional part of the fixed-point representation of the input values (Binary Point).
7. Number of bits to be used in the fractional part of the fixed-point representation of the weights values (Binary Point).
8. Type of implementation of activation function (available options: polynomial, piecewise linear, ROM).
9. Type of resources used to implement the activation function. On the one hand, if the activation function implemented by polynomial or piecewise linear is chosen, it is possible to select the type of multiplication to be used. On the other hand, if the activation function implemented by ROM is chosen, it is possible to insert the value of ROM depth.
10. Creates the ANN in a Simulink model file using the parameters indicated.
11. Exits the application.
12. Type of activation functions for each layer (available options: linear; hyperbolic tangent).

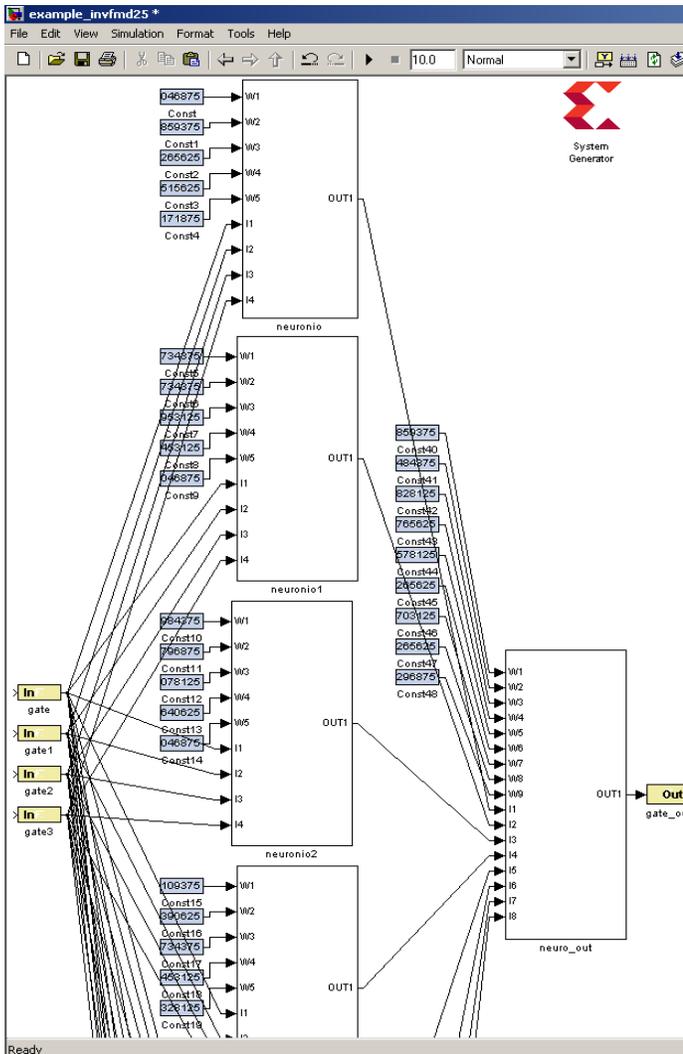


Fig. 11. Example of an ANN generated by ANGE with the weights loaded.

The large blocks represented in figure 11 are the neurons, the inputs are identified by the nomenclature *In* and the weights are introduced using the constant blocks. As can be seen, the ANN is implemented using full neurons and has a full parallel structure.

After selecting the configuration and characteristics of the ANN, ANGE will automatically generate a Simulink Model file, similar to the one represented in figure 11.

V. RESULTS

To test ANGE's implementation and evaluate the error introduced, models from a reduced scale kiln were used. This kiln was prepared for the ceramic industry where the temperature of the kiln is controlled by an inverse model of the system and further details can be seen in [23].

For these tests two sets of models which represent a direct and an inverse model of the system were used in order to construct a Direct Inverse Control loop, as represented in figure 12.

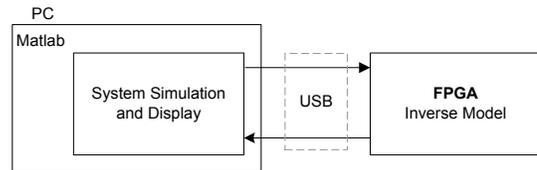


Fig. 12. MATLAB and FPGA connection

This type of loop requires that the models have a very good match and is, therefore, the best loop to be used when testing hardware implemented models. The reason for this is that, if the quality of the models is reduced by the implementation, it will be seen directly on the control results. Other loops (such as Internal Model Control) can compensate for the model's limitation and thus mask their imperfections [23]. The ANN used to test this tool has 4 inputs, 8 neurons in the hidden layer with hyperbolic tangent function and 1 neuron in the output layer with linear activation function.

The models were tested using two reference signals and in order to compare all the solutions used to implement the hyperbolic tangent, each system was compiled and a co-simulation block was created. These blocks can be inserted in a Simulink library and used in Simulink models, as shown in figure 13, inserting the FPGA in the loop and allowing the simulation to approach the real functioning of the system.

This co-simulation block was created for the FPGA Virtex 5 XC5VFX130T-1FF1738. The ANN implemented uses 32 bits for the gate-inputs and the weights. In the gate-inputs, 28 bit were used for the fractional part, and in the weights, 27 bits and 29 bits were used for the fractional part in the hidden layer and output layer, respectively.

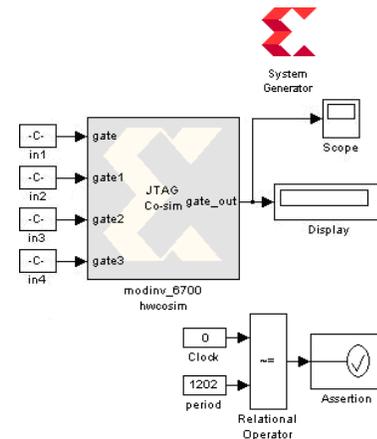


Fig. 13. An ANN generated by ANGE used a co-simulation block.

In order to determine the limitations introduced by the FPGA, a test to compare the same ANN implemented and executed in a FPGA and MatLab was performed. For the FPGA, it was also necessary to test the different implementations to compare their performances.

In table 1 this comparison is summarized using the MSE between the MatLab simulation and the FPGA co-simulation as measurement. The models were tested using two different

signals.

Table 2 shows the percentage of error between the Matlab Simulation and FPGA co-simulation, using the three different implementations of hyperbolic tangent. To calculate this percentage, the following equation was used:

$$E_{\%} = \left| \frac{Error_{Co-simulation} - Error_{Matlab\ simulation}}{Error_{Matlab\ simulation}} \right| \times 100 \quad (4)$$

TABLE 1
MSE IN MATLAB SIMULATION AND FPGA CO-SIMULATION USING DIFFERENTS HYPERBOLIC TANGENT IMPLEMENTATIONS

		Mean Square Error (MSE)	
		Signal Ref1	Signal Ref2
FPGA Co-simulation	ROM (10000 values)	0.01427558	8.156278
	Polynomials	0.01334391	8.153297
	Piecewise Linear	0.01333980	8.153300
Matlab Simulation		0.01307	8.1542

TABLE 2
ERROR BETWEEN THE MATLAB SIMULATION AND FPGA CO-SIMULATION

Error between Matlab Simulation and FPGA Co-simulation using	Signal	
	Ref1	Ref2
ROM (10000)	9.22%	0.0254%
Polynomials	2.1%	0.0110%
Piecewise Linear	2.1%	0.0110%

Reference signals 1 and 2 (that can be seen in figures 14 and 15) represent two different test used verify the implementation. The first is a signal rich in frequency terms while the second represents a typical profile behavior for certain ceramic parts' treatment.

As can be seen in table 2, the maximum error introduced in the hardware implementation is when using the ROM, where the maximum error introduced for the Ref1 was of 9.22% and for the Ref2 was of 0.0254%. The error introduced results from the use of a LUT, which contains 10000 values, with 32 bits to represent the hyperbolic tangent. Its maximum value can be derived from the number of bits truncated and the maximum step between consecutive values in the LUT. As a result, the error introduced should be bounded and small, and can be reduced with an increase in the number of bits used in the solution.

Furthermore with regards to table 2, it is possible to find the two best solutions for implementing an ANN. These two solutions are the implementations of the hyperbolic tangent with polynomials and the piecewise linear, where both solutions present the same error. For the second reference, the error obtained is smaller than the one obtained with Matlab. This may occur because in a control simulation, the effect of truncation can cancel small errors.

The solutions which use 25 polynomials and 742 piecewise linear have their coefficients represented with a limited number of bits (32 bits) and have fixed point notation. In these coefficients, represented with 32 bits, 2 bits are for the integer part and 30 bits are for the fractional part.

In order to calculate the polynomials and the piecewise linear sections, it is necessary to use two different operations (multiplication and addition). The result of each operation is

truncated with 32 bits, where 2 bits are used for the integer part and 30 bits are used for the fractional part.

Figure 14 and 15 show the results of the response from the block co-simulation, where the hyperbolic tangent was implemented using a ROM.

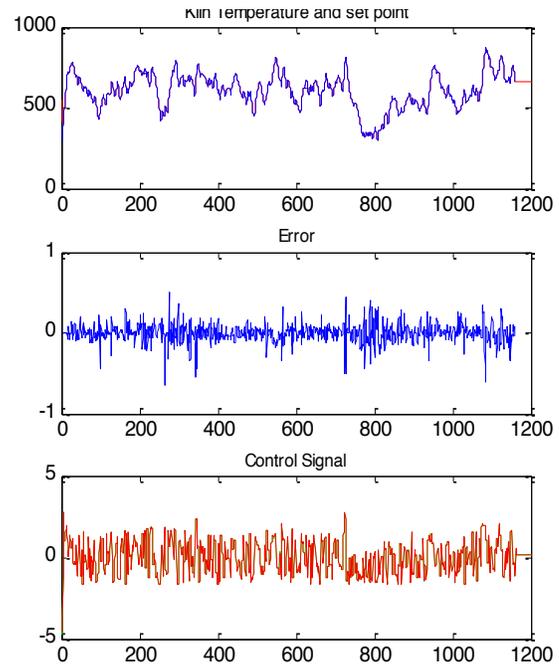


Fig. 14. The response to signal Ref1 and its associated error

TABLE 3
THE FPGA RESOURCES USED FOR EACH OF THE SYSTEM DESCRIBED ABOVE (DATA PROVIDED BY ISE)

Device Utilization	Kind of implementation of hyperbolic tangent						
	ROM (10000 values)	Polynomial using multiplication with			Piecewise linear using multiplication with		
		DSP's	2's Complement	Booth's Algorithm	DSP's	2's Complement	Booth's Algorithm
Number of BSCANs (4)	1 (25%)	1 (25%)	1 (25%)	1 (25%)	1 (25%)	1 (25%)	1 (25%)
Number of BUFGs (32)	2 (6%)	2 (6%)	2 (6%)	2 (6%)	2 (6%)	2 (6%)	2 (6%)
Number of BUFGCTRLs (32)	2 (6%)	1 (3%)	1 (3%)	1 (3%)	1 (3%)	1 (3%)	1 (3%)
Number of DSP48Es (320)	80 (25%)	176 (55%)	80 (25%)	80 (25%)	112 (35%)	80 (25%)	80 (25%)
Number of External IOBs (840)	1 (1%)	1 (1%)	1 (1%)	1 (1%)	1 (1%)	1 (1%)	1 (1%)
Number of LOCed IOBs (1)	1 (100%)	1 (100%)	1 (100%)	1 (100%)	1 (100%)	1 (100%)	1 (100%)
Number of RAMB36_EXPs (298)	58 (19%)	2 (1%)	2 (1%)	2 (1%)	2 (1%)	2 (1%)	2 (1%)
Number of RAMB18X2s (298)	24 (8%)	-	-	-	-	-	-
Number of Slices (20480)	1614 (19%)	1773 (8%)	8468 (41%)	19848 (96%)	17200 (83%)	18390 (89%)	20480 (100%)
Number of Slice Registers (81920)	2257 (2%)	520 (1%)	520 (1%)	603 (1%)	520 (1%)	520 (1%)	523 (1%)
Number used as Flip Flops	2249	520	520	520	520	520	520
Number used as Latches	0	0	0	0	0	0	0
Number used as LatchThrus	8	0	0	83	0	0	3
Number of Slice LUTs (81920)	4838 (5%)	6334 (7%)	29981 (36%)	74867 (91%)	58405 (71%)	66231 (80%)	80984 (98%)
Number of Slice LUT-Flip Flop pairs (81920)	5324 (6%)	6516 (7%)	30219 (36%)	74923 (91%)	58465 (71%)	66271 (80%)	80984 (98%)

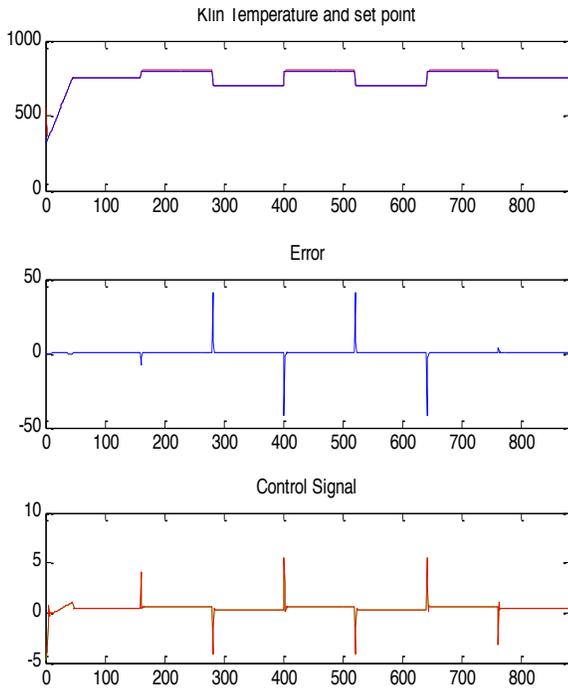


Fig. 15. The response to signal Ref2 and its associated error

Similarly to what is shown in Figures 14 and 15 tests were conducted using the polynomial implementation and the piecewise linear approach. Those figures are similar in shape to 14 and 15 but with a smaller error as presented in Table 2. Within each of these solutions the three implementations for the multiplier would yield the same results and are therefore not shown.

Table 3 shows the device utilization for the ANN in each implementation of the hyperbolic tangent. Analyzing table 3, it can be seen that among the solutions, which use polynomials, the one, which uses DSPs, saves other resources of the FPGA. However, two other solutions were tested where the multiplication applicable can be done without DSPs namely: Booth's algorithm and 2's complement. The idea behind implementing three different ways in order to perform a multiplication lies in finding the best possible balance between performance and resource use as well as allow the user select different resource allocations.

After analyzing the implementations it can be stated that: using DSPs frees other resources; 2's complement is less resource demanding than Booth's algorithm, while both these solutions do not use DSPs.

By analyzing the three solutions which use the piecewise linear sections, it is possible to conclude the same as for the solutions which use the polynomial approximation. Thus, to summarize: the solution, which uses DSP's, saves other resources.

And thus, the following question emerges: among the three different implementations for the multiplication, which are the choices to obtain a more effective implementation?

One plausible answer could be found if the structure of the intended ANN is analyzed. If the ANN has many inputs or many neurons in the hidden layer, it is advisable that one uses

the algorithm of 2's complement. This implementation allows saving DSPs, which are necessary to calculate the sum of the products between the weights and the inputs. Otherwise, it is better for one to use the multiplication's implementation with DSPs.

The solution using the ROM is a solution, which allows saving more resources of the FPGA. However, this solution presents a disadvantage because it presents a higher MSE. It is important to note that this solution uses 10000 values and that the representation for each of these values is done with 32 bits. If one wishes to decrease the MSE, the number of values in the ROM must necessarily be increased. But one must also bear in mind not to exceed the memory and resources of the FPGA.

Table 3 shows another interesting result: as a result of the larger number of parameters to be stored in the piecewise linear approach (742 sections), there is a higher device utilization rate for this solution. As can be seen from the last two rows, the increase in resources use rises dramatically. The rise can go from almost 10% (slice LUTs in Booth's algorithm from 91% to 98%) to 900% (slice LUTs in DSP's from 7% to 71%).

VI. CONCLUSION

This paper presents the second version of the ANGE tool, an Automatic General Purpose Neural Hardware Generator, and shows some of the results obtained with it. A test is presented with a small network fitted in a Virtex 5 XC5VFX130T-1FF1738 FPGA.

This version of the tool presents different implementations of the hyperbolic tangent. The first implementation consists of storing the values of the hyperbolic tangent in a LUT. The second implementation consists of using the Chebyshev interpolation. For a specified error, 25 polynomials of third order were needed. To implement the polynomials in the FPGA, adders and multipliers were needed. The multiplication operation was implemented in three different forms: with a specific feature of the FPGA (DSP), Booth's algorithm and 2's complement multiplication.

When comparing the solutions, the LUT holds the highest error, whereas the piecewise linear solution, with a new approach, and the polynomial approach hold almost the same error. When analyzing the resources used for both solutions, it was found that the polynomial approach can be advantageous, since the requirements may be significantly smaller in all the implementation alternatives.

The low error obtained, the diversity of solutions for the hyperbolic tangent and multiplication, the possibility to adjust the resolution (and resources used) and the possibility of directly implementing the ANN trained with a high level tool make ANGE II an unique and precious tool for hardware implementation of ANN regardless of their use for regression or classification.

ANGE will be distributed freely and will allow for fast prototyping while using the preferred hardware target for the neural community and is expected to contribute to a new growth of hardware implementations of ANN.

ACKNOWLEDGEMENTS

The authors would like to acknowledge the Portuguese Foundation for Science and Technology for their support through project PEst-OE/EEI/LA0009/2011.

REFERENCES

- [1] R.P. Lippmann, "An introduction to computing with neural nets", IEEE ASSP Mag. (1987) 4–22.
- [2] F. Morgado Dias, A. Antunes and A. Mota, "Artificial neural networks: a review of commercial hardware", in Engineering Applications of Artificial Intelligence, Vol.17, pp. 945-952, 2004.
- [3] T. Stieglitz and J. Meyer, "Biomedical Microdevices for Neural Implants", BIOMEMS Microsystems, Vol. 16, pp. 71-137, 2006.
- [4] P. Ferreira, P. Ribeiro, A. Antunes, F. Morgado, "A high bit resolution FPGA implementation of a FNN with a new algorithm for the activation function", Neurocomputing, Vol. 71, Issues 1-3, Pages 71-77, 2007.
- [5] M. Leon, A. Castro, R. Ascencio, "An artificial neural network on a field programmable gate array as a virtual sensor", Proceedings of the Third International Workshop on Design of Mixed-Mode Integrated Circuits and Applications, Puerto Vallarta, Mexico, 1999, pp. 114–117.
- [6] J.L. Ayala, A.G. Lomena, M. López-Vallejo, A. Fernández, "Design of a pipelined hardware architecture for real-time neural network computations", IEEE Midwest Symposium on Circuits and Systems, USA, 2002.
- [7] Soares, A.M., Pinto, J.O.P., Bose, B.K., Leite, L.C., da Silva, L.E.B., Romero, M.E., "Field Programmable Gate Array (FPGA) Based Neural Network Implementation of Stator Flux Oriented Vector Control of Induction Motor Drive", IEEE International Conference on Industrial Technology, 2006.
- [8] Xi Chen¹, Gaofeng Wang, Wei Zhou, Sheng Chang, and Shilei Sun, "Efficient Sigmoid Function for Neural Networks Based FPGA Design", ICIC 2006, LNCS 4113, pp. 672 – 677, Springer-Verlag Berlin Heidelberg 2006.
- [9] D. Baptista, F. Morgado-Dias, "A hyperbolic tangent replacement by third order polynomial approximation", CONTROLO'12-10th Portuguese Conference on Automatic Control, 2012.
- [10] Ghariani M., Kharrat M.W., Masmoudin N., Kamoun L., "Electronic implementation of a Neural Observer in FPGA technology: application to the control of electric vehicle", 16th International Conference on Microelectronics, 2004.
- [11] Meng Qian, "Application of CORDIC Algorithm to Neural Networks VLSI Design", IMACS Multiconference on Computational Engineering in Systems Applications, 2006.
- [12] A. R. Ormondi, J. Rajapakse, FPGA Implementations of Neural Networks. New York: Springer, 2006.
- [13] A. Gomperts, A. Ukil, F. Zurfluh, "Development and Implementation of Parameterized FPGA Based General Purpose Neural Networks for Online Applications," IEEE Transactions on Industrial Informatics, vol. 7, issue 1, pp. 78–89, 2011.
- [14] T. Orłowska-Kowalska, M. Kaminski, "FPGA Implementation of the Multilayer Neural Network for the Speed Estimation of the Two-Mass Drive System," IEEE Transactions on Industrial Informatics, vol. 7, issue 3, pp. 436–445, 2011.
- [15] A. Dinu, M.N. Cirstea, S.E. Cirstea, "Direct Neural-Network Hardware-Implementation Algorithm," IEEE Transactions on Industrial Electronics, vol. 57, issue 5, pp. 1845–1848, 2010.
- [16] D. Le Ly, P. Chow, "High-Performance Reconfigurable Hardware Architecture for Restricted Boltzmann Machines," IEEE Transactions on Neural Network, vol. 21, issue 11, pp. 1780–1792, 2010.
- [17] Maxfield, C. "The Design Warrior's Guide to FPGAs", Elsevier, ISBN 0750676043, New York, USA, 2004.
- [18] MathWorks. URL: <http://www.mathworks.com/products/hdl-verifyer/description3.html>, August 2012
- [19] L. Reis, L. Aguiar, D. Baptista, F. Morgado-Dias, "ANGE-Automatic Neural Generator", International Conference on Artificial Neural Network – ICANN'11, Espoo, Finland, 2011.
- [20] Wright S., Marwala, T. Artificial Intelligence Techniques for Steam Generator Modelling. School of Electrical and Information Engineering, P/Bag x3, Wits, South Africa, 2007.
- [21] Xilinx, "System Generator for DSP reference Guide", UG638, v 13.2, July, 2011.
- [22] D. Baptista, F. Morgado-Dias, "On the implementation of different hyperbolic tangent solutions in FPGA", 10th Portuguese Conference on Automatic Control – CONTROLO'12, Funchal, Portugal, 2012.
- [23] F. Morgado Dias, A. Mota: "Direct Inverse Control of a Kiln", 4th Portuguese Conference on Automatic Control, 2000.
- [24] A. Dinu, M. Cirstea, and S. Cirstea: "Direct Neural-Network Hardware-Implementation Algorithm", IEEE Transactions on Neural Network, Vol. 57, No. 5, 2010.
- [25] S. Oniga, "A New Method for FPGA Implementation of Artificial Neural Network Used in Smart Devices", International Computer Science Conference microCAD, 31-36, 2005.
- [26] A. Tisan, M. Cirstea: "SOM neural network design - A new Simulink library based approach targeting FPGA implementation", Mathematics and Computers in Simulation 91: 134-149, 2013
- [27] ST Pérez-Suárez, CM Travieso-González, JB Alonso-Hernández. Design Methodology of an Equalizer for Unipolar Non Return to Zero Binary Signals in the Presence of Additive White Gaussian Noise Using a Time Delay Neural Network on a Field Programmable Gate Array. Sensors 13 (12), pp. 16829-16850, 2013.
- [28] "A Software Tool for Automatic Generation of Neural Hardware", L. Reis, L. Aguiar, F. D. Baptista and F. Morgado-Dias, The International Arab Journal of Information Technology, Vol. 11, No. 3, May 2014.
- [29] "A survey of software and hardware use in Artificial Neural Networks", S. Abreu, F. Freitas, F. D. Baptista, R. Vasconcelos and F. Morgado-Dias, Neural Computing and Applications, Volume 23, Issue 3-4 (2013), pp 591-599.
- [30] "Low-resource hardware implementation of the hyperbolic tangent for artificial neural networks", F. D. Baptista and F. Morgado-Dias, Neural Computing and Applications, Volume 23, Issue 3-4 (2013), pp 601-607.
- [31] "Hyperbolic tangent implementation in hardware: a new solution using polynomial modeling of the fractional exponential part", Ivo Nascimento, Ricardo Jardim and F. Morgado-Dias, Neural Computing and Applications, Vol. 23, Issue 2, pp. 363-369, 2013.

Fábio Darío V. Baptista received his engineering degree in Electronics and Telecommunications in 2007 and his Master's Degree in Telecommunications and Network in 2009, both from the University of Madeira, Portugal. His main research field is artificial neural networks.

Fernando Morgado-Dias received his Diplôme D'Études Approfondies in Microelectronics from the University Joseph Fourier in Grenoble, France in 1995 and his PhD from the University of Aveiro, Portugal, in 2005 and is currently an Assistant professor at the University of Madeira. His research interests include Artificial Neural Networks and their applications, especially regarding their hardware implementations.