An End User Platform for FPGA-based Design and Rapid Prototyping of FeedForward Artificial Neural Networks with on-chip Back Propagation learning

Alin Tisan, *Member*, *IEEE* and Jeannette Chin, *Member*, *IEEE*

Abstract — The hardware implementation of an Artificial Neural Network (ANN) using field-programmable gate arrays (FPGA) is a research field that has attracted much interest and attention. With the developments made, the programmer is now forced to face various challenges, such as the need to master various complex hardware-software development platforms, hardware description languages and advanced ANN knowledge. Moreover, such an implementation is very time consuming. To address these challenges, the paper presents a novel neural design methodology using a holistic modelling approach. Based on the end user programming concept, the presented solution empowers end users by means of abstracting the low-level hardware functionalities, streamlining the FPGA design process and supporting rapid ANN prototyping. A case study of an ANN as a pattern recognition module of an artificial olfaction system trained to identify four coffee brands is presented. The recognition rate versus training data features and data representation was analysed extensively.

Keywords — FPGA; Artificial Neural Networks; End User Programming; HW / SW co-design and co-simulation; e-Nose;

I. INTRODUCTION

Hardware implementation of an ANN using field programmable gate arrays (FPGA) has been an interesting research field with applications in various domains since early nineties. At the beginning, the only generally accepted method was to design the application by means of Hardware Description Languages for VLSI (very large-scale integration) circuits, in particular VHDL or Verilog. Nowadays, engineers use modern Electronic Design Automation tools to create, simulate and verify a design, and, without committing to hardware, can quickly evaluate complex systems with high confidence in the "right first time" correct operation of the final product.

The FPGA reconfiguration capability and its parallel processing power are "hot topics", recognised in many papers focused in industrial applications: hardware implemented polar decoders [1], FPGA embedded controller of an n-Level DC–DC–AC inverter [2] or hardware implementation of predictive control algorithms for power converters [3]

With the newly emerged development environments for All Programmable Systems-on-Chip (SoCs) and multiprocessor Systems-on-Chip (MPSoCs) complex algorithms are now implemented in FPGA embedded processors [4]: FPGA/DSP-based digital controller with self-reconfiguration property for power quality compensation [5], FPGA embedded multiprocessor PLC that provides high execution speed, multiprocessing programming [6]. Despite ANNs being implemented in hardware for more than 25 years [7], it remains in the centre of attention for many researchers and a variety of methods to develop hardware implemented ANNs have been reported in the literature in the past decade [8, 9]. An overview of these achievements is given in [10] where the ANN theory and its hardware implementation are discussed extensively.

The main advantage in using the above methods is given by the fact that now the functional description of the design (the mathematical model) and its hardware implementation has been brought closer but the gap between them still exists. The pressing need to master different environments calls for a holistic approach in which the mathematical description and the electronic design implementation are simultaneously addressed in a unique environment. According to [11] the benefits of the holistic modelling approach are given by the possibility to evaluate increased system complexity at an early design stage in a unique platform. The time to market will be shortened, the use of automatic processes for hardware implementing the ANNs will be facilitated and therefore investigating different system topologies (ANN topologies) will be more eased. Combining the above-enumerated holistic modelling advantages with hardware description languages and FPGA capabilities, more complex neural networks can be modelled, simulated and implemented with an increased use of resource efficiency [12]. In this sense, an interesting approach is taken in [13] where the VHDL code of a Multilayer Perceptron ANN topology is generated by mean of a graphical user interface (GUI) designed in Matlab. The tool lifts the VHDL design burden from the user's shoulders, making the CAD environment to be more user-centred. Similar approaches are reported in the literature where automatic tools are developed to help the designer to exploit the dynamic partial reconfiguration of the FPGAs circuits [14] or to generate the VHDL code of complex fuzzy-logic systems [15].

This paper takes these steps further and presents a methodology based on the end user programming [16] concept, where end users are shielded from the need to know low-level technical hardware description languages. This is achieved by providing different layers of abstractions to represent in hardware the application functionality, such that end users are empowered by simply manipulating the abstractions via an intuitive and interactive GUI to support rapid prototyping. The system was tested as a pattern recognition module in an artificial olfaction system for identifying different coffee brands. An extended analysis regarding the recognition rates versus data representation has been carried out.

The paper is structured as follows: Section II – End User Programming on ANN Design Approach; Section III – Neural Libraries Design; Section IV – ANN Abstraction and EUP; Section V – Application and discussions; Section VI – Conclusions.

II. End User Programming (EUP) on ANN Design Approach

End user programming is characterised by the use of techniques that allow end users of an application to create "programs" themselves without needing to write any code [16]. A common way to achieve this goal is to create propriety types of "scripting languages"; abstracting conventional programming algorithms into some form of representations (e.g. graphical objects) and then to provide a platform for the users to manipulate these representations as the basis of learning how to create a program. Earlier work in this area was primarily focused on single desktop computing, allowing end users to create programs by manipulating abstract graphical objects. Recent developments have moved away from desktop computing systems to technology-rich ubiquitous environments where the EUP approach is no longer restricted to a single PC but leverages objects as a means to interact with the system [17]. Consequently, whilst some approaches still employ traditional graphical user interfaces on a single PC [18], others have been ported on mobile devices [19].

The technique adopted in this paper follows earlier published work on Pervasive Interactive Programming [20] that employs a show-me-by-example approach via natural interactions. The method further extends the use of modern EDA tools for the design, simulation and hardware implementation of an artificial neural network aiming to change the way in which user applications are defined. Instead of a classical solution, in which the application is defined using hardware description languages, it is more efficient (in terms of performances vs. hardware resource utilisation) and user friendly (the user does not need to know the neural algorithm or how to implement it in hardware) to create a pattern recognition system, in our case an ANN, by means of providing layers of abstractions to represent configurable modules, which are grouped into specific libraries that interface with the hardware. The abstractions are presented on an intuitive and interactive configurable graphical user interface (GUI), which end users can interact with and easily manipulate. The manipulations can be achieved by simple gestures such as pressing, tapping and "drag and drop". The GUI "listens" and "learns" the user's interactions on the screen and composes the program at the same time based on the desired requirements. For ambiguous actions detected,

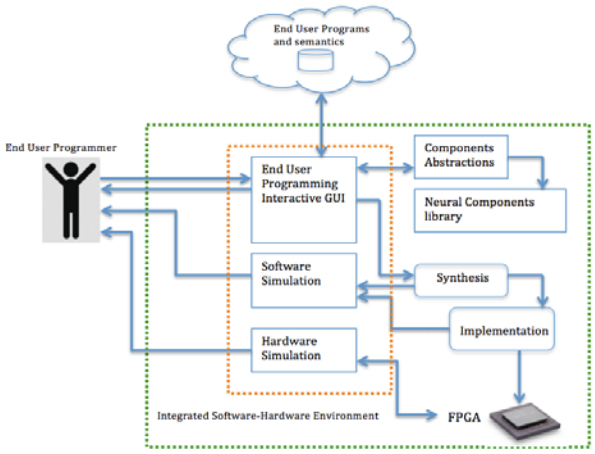


Fig. 1. Proposed method for hardware implemented ANN design

the system will help by providing suggestions. The immediate advantages of this approach are (1) to speed up the early phases of different ANNs' design and development process, (2) to allow the end users, who may not be familiar with the technologies, to create their programs without enduring a steep learning curve (Fig. 1).

The outcome is a configurable neural component library embedded into a design environment that allows considering simultaneously all the aspects of the system design. In this way, operational performance is maximized enabling high efficiency, which means high processing speed and minimum hardware resource utilization.

III. Neural Library Design

The ANN performance is heavily influenced by the topology chosen and its correlation with the application remains crucial. In this paper the neural network chosen to be modelled is the feed forward with back-propagation learning algorithm network (FFBP). As the main FFBP characteristics, such as the network topology, are selected by repetitive modifications, simulations and implementations of the project code, the availability of a hardware ready implementable ANN library would bring a plus in the effort to rapid design reliable pattern recognition systems in hardware.

The created neural network library, described in the next sections, contains extendable modules that comply with a generic FFBP architecture. It consists of processing units (neurons) organized in successive layers: one input layer, one or more intermediate hidden layers and one output layer. The network is fully connected, i.e. all the outputs of a layer are connected by synapses to all inputs of the following layer. Only the hidden and the output layers include processing units, whereas the input layer is used just for data feeding. The network uses the feedforward algorithm to push information forward from one layer to the next one and the back-propagation training algorithm for determining its weights: a repetitive algorithm that finds the minimum of the error function (the derivative of the sum-of-squares error with respects to the weights). The proposed software-hardware platform, underpinned by the ANN library and user interface, represents a viable way of designing and FPGA implement FFBP network topologies with on-chip learning as demonstrated in the Application section.

A. The FFBP neural network algorithm

The neural algorithm that emulates the FFBP ANN behaviour is described through equations (1)-(4). It starts with the computation of the output vector: MAC (multiply and accumulate) of all inputs with their corresponding weights (the net values) and fires the results with an activation function *f*, equation (1):

$$o_k \leftarrow f\left(bias + \sum_{k=1}^{K} w_k y\right) \tag{1}$$

The goal of the algorithm is to minimize the error function calculated in (2) by means of weights adjustment.

$$E \leftarrow E + \frac{1}{2}\left(d_k - o_k\right)^2, k = 1, 2, ..., K \tag{2}$$

For this, a corresponding partial derivative error with respect to its net output value is computed (3).

$$\delta_{ok} = -\frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial net_k} = -\frac{\partial\left[\frac{1}{2}\sum(t_k - o_k)^2\right]}{\partial o_k}\frac{\partial o_k}{\partial net_k} \Rightarrow$$
$$\delta_{ok} = (d_k - o_k)o_k(1-o_k)$$
$$\delta_{yj} = y_j(1-y_j)\sum_{k=1}^{K}\delta_{ok}w_{kj} \quad (3)$$

Next, an update stage follows, in which all the weights, hidden and output ones, will be adjusted (4).

$$w_{kj} \leftarrow w_{kj} + \eta\delta_{ok}y_j, \text{ for } k = 1,2,...,\text{K and } j = 1,2,...,\text{J}$$
$$v_{ji} \leftarrow v_{ji} + \eta\delta_{yj}x_i, \text{ for } i = 1,2,...,\text{I and } j = 1,2,...,\text{J} \quad (4)$$

The FFBP network will follow these computation steps until the calculated error will be less than a given threshold value.

In (1) to (4) were used the following abbreviations: *net* is the fired neuron output; w_k is the weight vector of neuron *k* from the output layer; *i, j, k* are the neuron's indexes (number); *I, J, K* are the number of neurons of the input, hidden and output layer, *E* is the error function, *o* is the output vector of the output layer; *y* is the output vector of the hidden layer, δ_{ok}, δ_{yj} are the gradient of the error signals of neuron *k* of the output layer and respectively neuron *j* from the hidden layer, v_j is the weight vector of the neuron *j* from a hidden layer.

B. FFBP neural library design

Designing the neuron of a multilayer feed-forward neural network with on-chip learning must consider not only the requested computations in the propagation phase, when the neural network is already trained and performs the recognition task (1), but also the learning phase, when the neural weights are updated according with the error minimization, as in (2) to (4). The neuron designed by the authors is built using Xilinx System Generator library blocks and consists of MAC unit, RAM memory module, multiplexor and a register for bias values initialization, and one firing function block. When designing the MAC unit two approaches may be adopted: using distributed resources, (Fig. 2), or dedicated modules that can replace the accumulator, multiplier and multiplexor components such as XtremeDSP or BRAM blocks. The number of dedicated modules (which ensure the best neuronal processing performances) differs from one FPGA family to another. Therefore for finding the neuron with the highest performance, related to the hardware resources available in the targeted FPGA, four possible optimization scenarios were considered:

- DL_AO: optimized for minimizing the occupied area and multiplications are done using distributed logic resources;

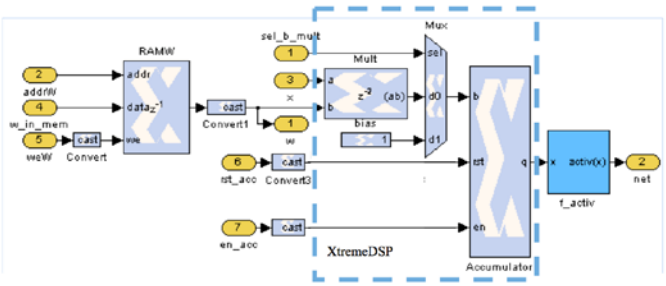


Fig. 2. Hardware architecture of the neuron

Table I
Hardware resources utilization for neuron implementation

Model	Slices	LUTs	Frequency (MHz)
DL_AO	36	49	151,676
DL_SO	36	49	154,369
DSP_AO	5	0	253,485
DSP_SO	5	0	267,237

- DL_SO: optimized for speed processing and multiplications are done using distributed logic resources;
- DSP_AO: optimized for minimizing the occupied and multiplications are done using dedicated resources
- DSP_SO: optimized for speed processing and multiplications are done using dedicated resources.

The synthesis of the maximum processing frequency and the hardware resources utilization were generated with the ISE Xilinx report generator tool and are presented in Table I.

The results analysis shows that the neuron based on the XtremeDSP block has the highest processing frequency and uses the fewest hardware resources in terms of slices or LUTs (as expected). Nevertheless, as the XtremeDSP blocks are limited, (128 for 4VSX35), to extend the number of neurons implemented, distributed logic can be used instead.

Another component of the neuronal library is the activation function. Its role is to map the neuron output values to a range of values given by the function chosen as a firing function, in this case the sigmoid function (5).

$$output(net) = \frac{1}{1+e^{-\left(bias+\sum_{k=1}^{N}w_k x_k\right)}} \quad (5)$$

Implementing the sigmoid function in hardware requires advanced hardware description language knowledge. Moreover, once implemented, it acts as a bottleneck for the neuron speed performance demanding considerable hardware resources in the same time. In order to reducing the hardware cost, different approximations of the sigmoid function can be adopted. The main classical methods to digitally implement an activation function are Look-up tables and truncation of the Taylor series expansion. Taylor expansion can further be implemented in various ways: sum-of-steps, piece-wise linear, combination of the previous, or others. The best results reported in the literature show errors of 8% to 13.1% for sum-of-steps approximations and ± 2.45% to ± 1.14% for piece-wise linear approximation. Also, there are approximations with smaller errors, but they use floating-point multiplications, thus practical implementation becomes too complex [30].

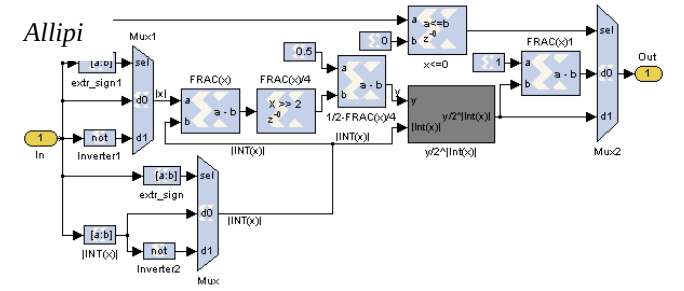
The firing function library, created by the authors, consists of ready hardware implementable modules of functions chosen to approximate the sigmoid function: A-low (F1), Allipi (F2), PLAN (F3) and Zhang (F4) [21]. Their mathematical and hardware implementation are summarised in Table II and their hardware resources utilization in Table III. As shown, the approximation functions were implementing using minimum of the hardware resources.

Table IV shows the resources utilized by the entire neuron using different approximation functions, revealing that each of them has drawbacks and strengths in terms of speed processing and hardware utilisation.

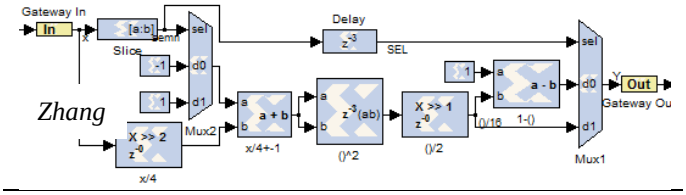
Table II

MATHEMATICAL AND HARDWARE IMPLEMENTATION OF THE APPROXIMATION FUNCTIONS

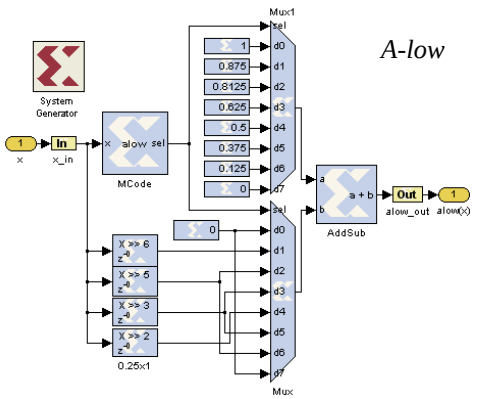
$$Alippi(x) = \begin{cases} 1 - \dfrac{1/2 + FRAC(-x)/4}{2^{INT(x)}} & \text{for } x > 0 \\ \dfrac{1/2 + FRAC(x)/4}{2^{INT(x)}} & \text{for } x \le 0 \end{cases}$$



$$y = \begin{cases} \dfrac{1}{2}\left(\dfrac{x}{2^2} - 1\right)^2 & \text{for } -4 > x < 0 \\ 1 - \dfrac{1}{2}\left(\dfrac{x}{2^2} + 1\right)^2 & \text{for } 4 > x \ge 0 \end{cases}$$



x	y
-8	0
-4	0.0625
-2	0.12
-1	0.25
1	0.75
2	0.87
4	0.937
8	1



X	*PLAN*(X)
|X| ≥ 5	1
2,375 ≤ |X| < 5	0,03125 · |X| + 0,84375
1≤ |X| < 2.375	0,0125 · |X| + 0,625
0 ≤ |X| < 1	0,25 · |X| + 0,5

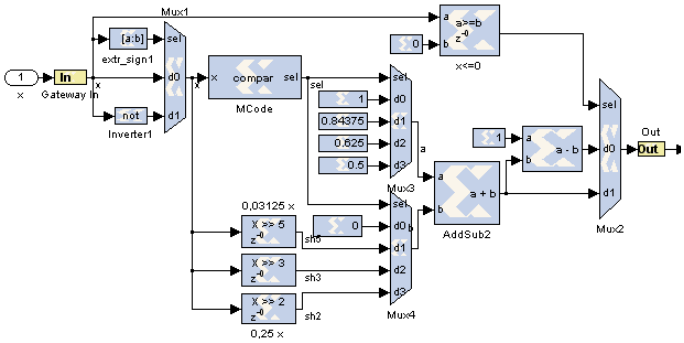


Table III

RESOURCE DISTRIBUTIONS FOR HARDWARE IMPLEMENTATION OF DIFFERENT FIRING FUNCTION WITH DIFFERENT BITS REPRESENTATION

Func	LUT			DSP			BRAM		
	(32,16)	(16,8)	(8,4)	(32,16)	(16,8)	(8,4)	(32,16)	(16,8)	(8,4)
F1	185	74	23	0	0	0	0	0	0
F2	127	67	36	0	0	0	0	0	0
F3	109	44	24	0	0	0	0	0	0
F4	93	29	18	1	1	1	0	0	0

Table IV

RESOURCE DISTRIBUTIONS FOR HARDWARE IMPLEMENTATION OF ARTIFICIAL NEURON WITH DIFFERENT FIRING FUNCTION USING (3,10) BITS REPRESENTATION

Resources distribution	Neuron Lookup table	Neuron Zhang	Neuron Allipi	Neuron A-low	Neuron PLAN
Slices	5	28	59	29	12
LUTs	0	25	89	31	10
RAMBs	2	1	1	1	1
DSPs	1	2	1	1	1
Max frequency (MHz)	227.790	255.860	290.613	268.168	234.467

It can be concluded that the best approximation method, in terms of resources utilized and errors introduced, is the PLAN function, when the number of the neurons that use sigmoid function is larger than the number of the BRAM blocks available in the FPGA circuit. When the number of neurons is lower than the total BRAM blocks available in the FPGA circuit, the best way to approximate the sigmoid function is the Lookup Tables method. The resolution used was (3,10) where 3 bits were allocated for the integer part and 10 bits for binary part. The errors introduced by the implemented functions are summarized in Table V.

Table V

ERRORS AND RESOURCE UTILIZATION OF THE 4VSX35 FPGA CIRCUIT FOR HARDWARE IMPLEMENTATION OF THE SIGMOID APPROXIMATION

Approximation function	Maximum error (%)	Mean error (%)	Total equivalent gates count for design
Lookup Table	0	0	131.072
A-low	5.63	0.63	411
Allipi	1.89	1.11	877
Plan	1.89	0.63	351
Zhang	2.16	1.10	314

C. The control neural library with on-chip learning

The control of the neuronal processing components is done through specialised blocks designed by the authors. These units are designed to accommodate the on-chip BP learning algorithm and the parallelism at the neuron level, all the neurons within the same layer are controlled at the same time (in parallel), taking advantage of the massive parallel processing supported by the FPGAs.

The blocks that control the ANN processing units consist of a general counter, used to provide the time base for the entire neural network according to the ANN's phase: propagation (when the network is already trained and performs recognition) or learning (when the network is on-chip trained to recognize the patterns) and ANN layer specific command signal generator blocks (two in the example given: one for each neuronal layer), Fig. 3.

The *General counter* block calculates, function of neurons architecture, network topology and processing phase (propagation or learning), the counter's maximum value and generates the reference time to set/reset the neuronal control

signals. Eq (6) gives the algorithm for calculating the number of clock cycles necessary to complete the processing and weights updating tasks (the counter max value), *clk_cycles*, and where *n1*, *n2* are the neurons in the input and respectively output layer; *t* is the t^{th} layer of network and *ceil* is the Matlab function that approximates a real number up to the next integer. Eq 6 gives also the ANN processing speed in the learning (PL = 1) or propagation (PL = 0) phase.

$$clk\_cycles = [n1+n2+6t-2]+PL[14+n2+(t-1)(n2ceil(((n1+n2)+ +6t+12)/n2)-((n1+n2)+6t+12)+3+n2)] \quad (6)$$

The *Command signals generator* block generates the controlling signals for all the processing elements of the neurons at specific moments (counter values). For this, the block calculates the values at which commands have to be given using (7), according to the neuron's architecture, where *t*: the layer number; *n*: the number neurons in layer *t*; *set_rst_acc:* the counter's value at which the accumulator's reset signal is set (the neuron's accumulator is reset); *en_acc_start*: the counter value at which the accumulator enable signal is set (the neuron accumulator is enabled); *en_acc_stop*: the time at which the accumulator enable signal is reset (the accumulator is disable); *propag_start*: the counter's value at which the neuron's propagation phase starts; *propag_stop*: the counter's value at which the propagation phase stops; *update_layer*(*t*)*_start*: counter's value at which the t^{th} layer weights start to be updated (the memory write enable signal is set); *update_layer*(*t*)*_stop*: counter's value at which the weights of the t^{th} layer updating process is stopped (the memory write enable signal is reset).

$$\begin{aligned} set\_rst\_acc &= (t-1)(n+6)+2 \\ en\_acc\_start &= (t-1)(n+6)+4 \\ en\_acc\_stop &= (t-1)(n+6)+4+n-1 \\ propag\_start &= (t-1)(n+6) \\ propag\_stop &= (t-1)(n+6)+n-1 \\ update\_layer(t)\_start &= t(n+6)+12 \\ update\_layer(t)\_stop &= t(n+6)+12+n-1 \end{aligned} \quad (7)$$

The blocks are described using VHDL language and implemented using Black box modules, a block that converts a VHDL design into a System Generator block. The computational tasks from eq. (2) to (4), which describe the algorithms for updating on-chip the ANN weights, have been implemented with three computing blocks: i) the errors computing block, ii) the output layer weights computing block and iii) the hidden layer weights computing block. The errors and layer weights computing blocks calculate the accumulated error, the gradient of the error (δ) and the value that the weights should be changed with (Δw) to decrease the accumulated error, fig. 4

The weights of the hidden layer are calculated last (due to back propagation error algorithm). For this, a series of processing blocks that calculate according to (3) the new weights were designed, fig. 5. The processing time, expressed in clock cycles, is given in (8) and is used in delaying the weights updating task (the delay introduced to permit the calculation of the new weights to be completed).

$$Delay = n_2 \cdot ceil(((n_1+n_2)+6t+12)/n_2)-(n_1+n_2)+6t+15 \quad (8)$$

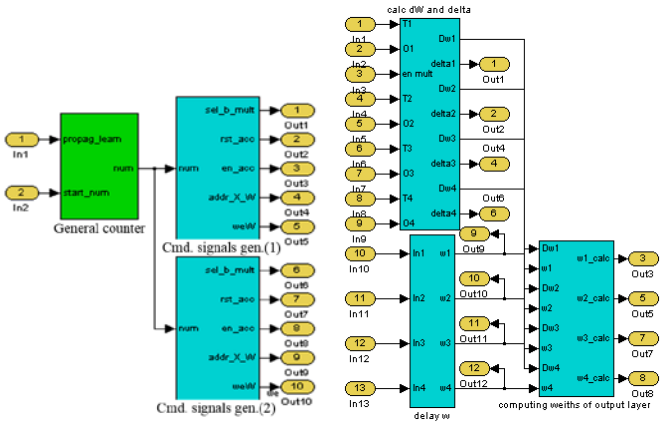


Fig. 3. Control block architecture

Fig. 4. Output layer errors and weights calculus blocks

An overall view of the main blocks involved in designing an ANN architecture with a 7-7-4 topology (7 neurons in the input layer, 7 neurons in the hidden layer and 4 neurons in the output layer) is shown in Fig.6

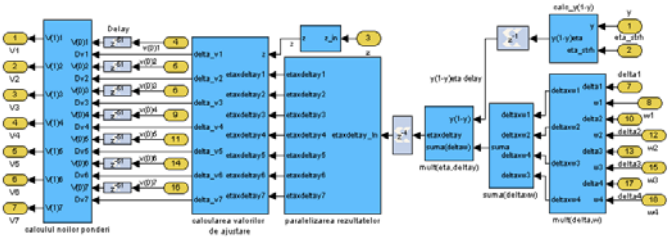


Fig. 5. The weights of the hidden layer calculus blocks

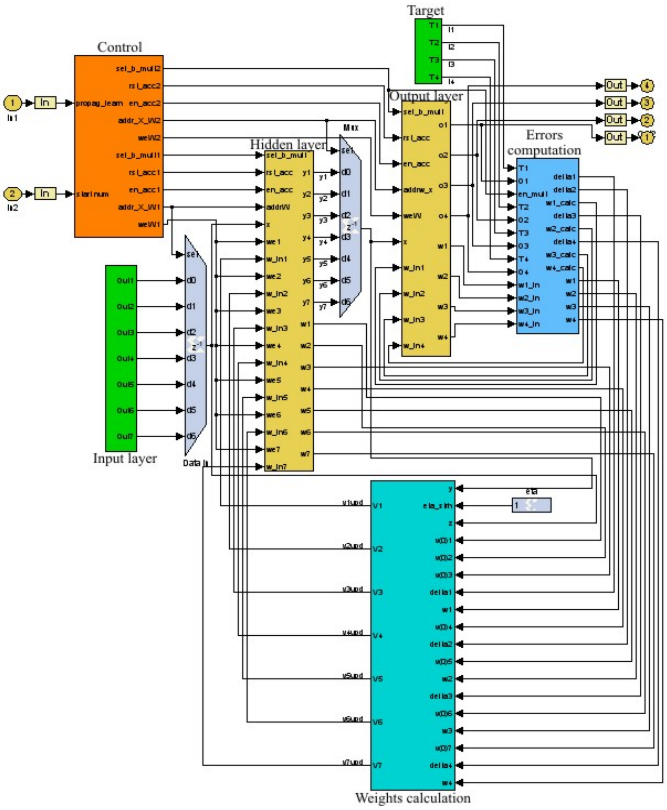


Fig. 6. Architecture of the 7-7-4 FFBP neural network topology

IV. ANN Abstraction and EUP

The idea behind the ANN abstraction is to shield the complexity from the end user while allowing them to create their own desired ANN program without incurring a steep learning curve, thus promote rapid prototyping. To achieve this goal we employed rule-based technique that often found in many AI systems. Rules are first created according to the design and requirement of each of ANN blocks (Fig 7) and store in the "End User Programs and Semantics" component (i.e. the knowledge space) (Fig 1). These rules are then used as the basis of ANN programming abstraction presented as graphical representations. The relationships between each rule are described in a form of semantic manner. The end user will then create their own ANN programs by simply manipulating the graphical "Rules" representations and develop their own new "Rules" via an interactive GUI (Fig 7). The GUI is implemented using event-based architecture. Using JavaScript API, various UI events (such as drag, drop, move and click) have been developed mapping the rules requirements that store in the main knowledge space (thus can "trigger" other events based on those rules) to "listen" to the user's activities that is happening on this panel. To understand the user interactions, a learning feature is implemented such that each activity will be captured and interpreted/inference according to the rules store in rule-based or be learned as new rule. Based on rules, an "expert system" is implemented to guide the end user to create their ANN program via a series of dialogs. User is able to configure the "rule" by simply clicking the graphical representations. The newly created "Rules" will be recorded as instances and stored back in knowledge space, which can retrieve and amend later. The contributions of this paper in terms of software design are: (1) the semantic rules based on ANN design, (2) the abstractions and representations, (3) the expert system including the learning feature and (4) GUI event-based architecture. The EUP GUI is implemented using Python language and JavaScript, together with a pre-installed MATLAB Engine that enable Simulink functions to be called through the provided APIs.

Fig. 7. ANN Design EUP GUI

V. Application and analysis

The developed FFBP neural network library was used to create a pattern recognition module for an artificial olfactory system trained to recognize different types of coffee. The olfactory system consists of: seven gas sensors chosen to react to a wide spectrum of odours (TGS842, TGS826_1, TGS826_2, TGS2600, TGS2601, TGS2602, TGS2620), temperature sensor (LM35), humidity sensor (SY-HS-230), mounted into a gas test chamber, test chamber, three gas pumps, circuits for sensors conditioning and pumps command, data acquisition board, pattern recognition module hardware implemented in FPGA (Virtex-4 SX 4VSX35), user interface.

A. Data acquisition and processing

The data acquisition module was customized to control the gas pumps (used to transport the smell to and from the test chamber), acquire data generated by all 9 sensors and pre-process the acquired signals (filtering, drift cancellation). The data has been extracted from the measurement over a defined absorption/ desorption time of the voltage drop on sensors resistance when the enriched odour is applied/removed. Data acquired constitutes the fingerprint of the smell and to process it, dimensional reduction techniques are applied. In most cases, this is performed by extracting a single parameter (e.g. steady-state, final or maximum response) from each sensor, disregarding the initial transient response, which may be affected by the dynamics of the odour delivery system. In some situations, transient analysis may significantly improve the performance of the gas sensor arrays and should be taken in consideration. Considering the feature extraction methods reported in literature [22], a heuristic method has been adopted with the following selected features: average value (A1), maximum value (A2), function integral (A3), integral of the absorption time (A4), maximum slope of the absorption (A5), maximum slope of the desorption function (A6), time at which maximum slope of absorption function occurs (A7) and time at which maximum slope of desorption function occurs (A8).

B. ANN performance analysis

For determining the best FFBP network implementable with a minimum of resources, a series of different FFBP NN topologies have been tested. In addition, for each topology, fixed-point binary representation with different resolutions have been investigated. Fig 8 shows the recognition rate vs. data representation for a topology of 56-56-4 neurons, which processes an input vector with 56 components: 8 features per sensors (A1 to A8) and 7 sensors. The recognition rate varies from 100%, for (16,16) bits representation (16 bits for integer part and 16 bits for binary part), to 50% for (7,8) bits representation and 0% for (2,3) bits representation. A major drop of the recognition rate occurs, 96% to 49%, when one bit of the integer part: (8,8) → (7,8) is changed. The recognition rate remains constant for a major drop of data resolution (16,16) → (8,8). These observations may be very useful when choosing the data representation resolution. Fig 9 and 10 are plotted in order to highlight the influence of data representation resolution over the recognition rate for a given training set. First, a training set with features (A1, A2, A3) is shown in Fig 9 and (A2) in Fig 10. It can be concluded, there is no perfect FFBP network

topology for every purpose, but it can be adapted to fulfil the most important requirements of a given application. For example, if the chip area occupation is an important issue, then a 21-21-4 FFBP network with a (5,5) bits representation and a theoretically recognition rate of 90% could be more than acceptable. However, for obtaining a higher recognition rate, a 56-56-4 FFBP network with a (16,16) bits representation might be a better option. Consequently, as demonstrated in the above discussion, the accuracy of the ANN is massively determined by the data representation adopted. Similar reports are shown in [10].

C. ANN hardware implementation results

To implement in FPGA the above ANN topologies requires specific hardware resources, which can be priory calculated. Having a formula to estimate the hardware resources needed for implementing a specific ANN topology would let the user choose the right ANN size and FPGA circuit.

By analysing the hardware implementation reports presented in Table VI, where HL denotes the hidden layer and OL the output layer, it can be concluded that:

- each neuron added to the hidden layer increases by 32 LUTs and 1 multiplier the overall resource utilization.
- each neuron added to the output layer increases by 40 LUTs and 4 multipliers the output neurons weights computation block and with 49 LUTs and 1 multiplier the hidden neurons weights computation block;

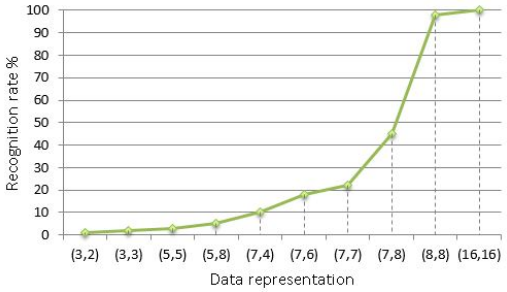


Fig. 8. Recognition rates vs. data representation for 56-56-4 FFBP

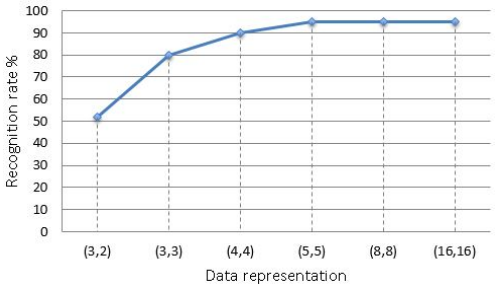


Fig. 9. Recognition rate vs. data representation for 21-21-4 FFBP

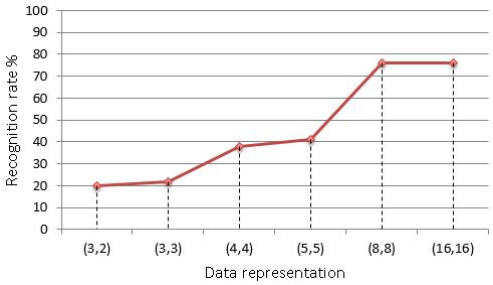


Fig. 10. Recognition rate vs. data representation for with 7-21-4 FFBP

Table VI
DSP, SLICES (SL) AND RAM DISTRIBUTION OVER FFBP COMPONENT BLOCKS FOR DIFFERENT ANN TOPOLOGIES

ANN	Neuronal block			HL block			OL block			Control block		
	DSP	SL	RAM	DSP	SL	RAM	DSP	SL	RAM	DSP	SL	RAM
1-1-1	1	5	4	5	64	0	4	24	0	12	105	0
7-2-4	6	76	12	9	190	0	12	100	0	12	114	0
7-7-7	14	105	28	17	256	0	28	215	0	12	108	0

Based on the reports presented, 3 equations have been generated to estimate the hardware resources utilized to implement a given FFBP topology, prior to an actual hardware implementation, (9) - (11). These permit choosing the right FPGA circuit for a given ANN topology/size in the very early ANN design stages, saving time and costs.

$$RAMs = 2(N_h + N_o) \quad (9)$$

$$DSPs = 15 + 2N_h + 6N_o \quad (10)$$

$$LUTs = 200 + 89N_o + 32N_h \quad (11)$$

where: - N_o is the number of neurons in the output layer,
- N_h is the number of neurons in the hidden layer.

Applying (9)-(11) to the circuit targeted in this paper, a Virtex4 with 15.360 slices, 30.720 LUTs, 192 BRAMs, 192 DSPs, the maximum number of neurons that can be implemented using strictly the dedicated BRAMs and XtremeDSP blocks (for ensuring the maximum processing speed) is 60, organized as: 45 in the hidden layer and 15 in the output layer. However, using the distributed multipliers and BRAMs available in the circuit, 26 more neurons, 20 in the hidden layer and 6 in the output layer, can be implemented. These will utilize 6878 LUTs and 76 BRAMs, leaving 22657 LUTs unused. The unused LUTs can be further converted into 20 neurons in the hidden layer and 10 in the output layer. Therefore, the maximum number of neurons that can be hardware implemented (on the expense of the processing speed) is approximated to 120 (double than the number of neurons that use only dedicated BRAMs and XtremeDSP blocks).

To illustrate the FPGA implementation performance, a report in terms of hardware resources utilization, and maximum processing frequency is presented in Table VII.

Table VII
FPGA implementation reports

Resource distribution	FF(1-1-1)	FF(7-7-7)	FF(7-2-4)
LUTs	322	1167	612
RAMB16s	4	28	12
DSP48s	23	71	43
Max frequency (MHz)	122.489	96.516	106.09

D. ANN performance comparisons

A direct comparison of the data presented in table VII with others reported in the literature is not always relevant due to the lack of common referencing in reporting the hardware resources per ANN performances. These depend on the type of the resources available in the FPGA (4 or 6 inputs LUTs, multipliers or XtremeDSPs, etc.) the depths of the ANN parallelism adopted (synapse, neuron or layer), the firing function (sigmoid, hardlim, etc.) processing speed, data representation, use of dedicated or distributed resources, on or off chip learning, number of hidden layers to nominate the most important ones. In [10] for implementing the 10-3-1 FFBP topology with a synaptic parallelism, 70 DSPs and 8043 LUTs were used. In [11] the hardware utilization is reported per neuron with

1299 LUTs / neuron. In [23] for a 2-5-1 topology 11 DSPs and 6384 LUTs were consumed. In this paper for a similar topology of 7-2-4, 43 DSPs and 412 LUTs were used.

As shown above, the hardware utilisation depends on factors which vary from one ANN topology, and FPGA, to another but they are all reflected in the recognition rate (RR) and processing speed (PS) supported by the chosen FPGA. Hence, reporting RR and PS, along with the hardware utilisation, would indicate better the level of success in using a particular ANN topology in a specific FPGA circuit.

Choosing the right FPGA circuit for a given ANN or the ANN size for a given FPGA circuit is not straightforward. As shown in [10] for selecting the right FPGA circuit, the designer is forced to implement the design first and then interpret the hardware resources used vs. the ANN topology. Therefore, being able to estimate the hardware resources needed for implementing an ANN before to an actual implementation would shorten the development time and consequently save costs. This is addressed for a given FPGA family by the equations (9)-(11).

VI. CONCLUSIONS AND FUTURE WORK

A novel neural design strategy has been developed, which benefits of reduced design time over classical field orientation approaches, leading to a low complexity and easy to implement pattern recognition module. A particular application of the pattern recognition system for an olfactory system is investigated and results presented show efficient hardware implementation in FPGA circuit. The achievement presented in this paper refers to a holistic modelling / design method, using modules created into hardware-software co-design environment (Matlab-System Generator–ISE) and grouped in a specific NN library. These modules emulate in hardware any FFBP network topology behaviour, giving the opportunity to design hardware implementable FFBP neural networks, at a higher level, via an intuitive and interactive EUP interface.

The proposed methodology takes advantage of the FPGA parallel processing power preparing the ground for an auto-adaptive reconfigurable device ready to respond - read auto-reconfigure - to any pattern recognition challenge. It is hoped that, through the proposed method, it would be possible to make steps towards a “more like brain” computational machine, in terms of adaptability and quick response, a system that makes its own choices (upon an implemented algorithm), i.e. intelligence.

As the components are entirely designed using System Generator blocks, the created library is technology dependent to the software used. For increasing the portability, future work will consider having the blocks designed using hardware description languages, generated from System Generator.

In conclusion, the paper shows that any FFBP topology may be built using predefined neural blocks with the following characteristics: i) holistic modelling and optimisation, ii) behavioural analysis, and iii) easy hardware prototyping on an FPGA development platform via an intuitive EUP interface. In addition, it has been developed a set of equations to estimate: i) the hardware resources needed to implement an FFBP ANN with on-chip learning in a given FPGA circuit (eq. 9-11) and ii) the processing speed of the implemented ANN topology (eq. 6). Moreover, design concepts introduced in [20] and [24] are brought further with contributions in developing an ANN design platform based on semantic rules, abstractions and representations, expert system and GUI event-based architecture.

VII. REFERENCES

[1] P. Giard, G. Sarkis, C. Thibeault and W.J. Gross, Electronics Letters Vol. 51 No. 10 pp. 762–763, 2015

[2] B. Dastagiri Reddy, Anish N. K., et al, Embedded Control of n-Level DC–DC–AC Inverter, IEEE Trans. on Power Electronics, 30(7), 2015

[3] Z. Zhang, He Xu, et all, Predictive Control with Novel Virtual-Flux Estimation for Back-to-Back Power Converters, IEEE Transactions on Industrial Electronics, vol. 62, no. 5, 2015

[4] A. Malinowski, Y. Hao, Comparison of embedded system design for industrial applications, Trans. Ind. Informatics 7 (2), pp 244-254, 2011.

[5] Man-Chung Wong, Yan-Zheng Yang, Chi-Seng Lam et all, Self-Reconfiguration Property of a Mixed Signal Controller for Improving Power Quality Compensation During Light Loading, IEEE Transactions On Power Electronics, vol. 30, no. 10, 2015

[6] Z. Hajduk, B. Trybus and J. Sadolewski, Architecture of FPGA Embedded Multiprocessor Programmable Controller, IEEE Transactions on Industrial Electronics, vol. 62, no. 5, 2015

[7] J. Misra, I. Saha, Artificial neural networks in hardware: A survey of two decades of progress, Neurocomputing 74, pp 239-255, 2010.

[8] A. Gomperts, A. Ukil and F. Zurfluh, Development and Implementation of Parameterized FPGA-Based General Purpose Neural Networks for Online Applications, IEEE Transactions On Industrial Informatics, Vol. 7, No. 1, February 2011

[9] F. Ortega-Zamorano, J. M. Jerez and L. Franco, FPGA Implementation of the C-Mantec Neural Network Constructive Algorithm, IEEE Trans. on Ind.Informatics, vol. 10, No. 2, May 2014.

[10] A. Omondi, R. Amos, J. Rajapakse, FPGA Implementations of Neural Networks, Edided by Springer, ISBN-10 0-387-28485-0, 2006.

[11] M. Cirstea, A. Dinu, A VHDL Holistic Modeling Approach and FPGA Implementation of a Digital Sensorless Induction Motor Control Scheme, IEEE Trans. on Ind. Electronics, vol. 54, (4), 1853 - 1864, 2007

[12] A. Dinu, M.N. Cirstea, S.E. Cirstea: Direct Neural Networks Hardware Implementation Algorithm, *IEEE Trans. on Ind. Electronics*, vol. 57, no. 5, pp.1845-1848, May 2010.

[13] A. Rosado-Muñoz, E. Soria-Olivas et al., An IP Core and GUI for Implementing Multilayer Perceptron with a Fuzzy Activation Function on Configurable Logic Devices, J. of Universal Comp. Sc. vol. 14, no10. pp 1678-1694, 2008.

[14] E. Vansteenkiste et all, TPaR: Place and Route Tools for the Dynamic Reconfiguration of the FPGA’s Interconnect Network, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 33, no. 3, pp 370-383, 2014

[15] M. Brox et all, CAD Tools for Hardware Implementation of Embedded Fuzzy Systems on FPGAs, IEEE Transactions on Industrial Informatics, vol. 9, vo. 3, pp 1635-1644, 2013

[16] A, Cypher Halbert DC, Kurlander D, et al, “Watch What I Do: Programming by Demonstration” The MIT Press, England 1993.

[17] N,.Elumeze, et all Serious Programming Made Cuddly: A Fully End-User-Programmable Stuffed Toy, Digital Game and Intelligent Toy Enhanced Learning, Third IEEE Int. Conf., pp.146-150, 2010

[18] J.,Lincke, , Krahn, R., et al, Lively Fabrik A Web-based End-user Programming Environment, Creating, Connecting and Collaborating through Computing, 7th International Conference on , pp.11-19, 2009

[19] Mateo, C.; Brunete, A.; et al, Hammer: An Android based application for end-user industrial robot programming, Mechatronic and Embedded Systems and Applications,10th Int. Conference on , pp.1,6, Sept. 2014

[20] Chin, V. Callaghan, G. Clarke, End-user Customisation of Intelligent Environments". In the handbook of Ambient Intelligence and Smart Environments, Springer, 2010, Spring, pp. 371-407,

[21] M.T. Tommiska: Efficient digital implementation of the sigmoid function for reprogrammable logic, *IEE Proceedings – Computers and Digital Techniques*, number 6, pp. 403-411, 2003.

[22] R. Gutierrez-Osuna, H. T. Nagle, and S.,S Schiffman, Transient response analysis of an electronic nose using multi-exponential models, *Sensors and Actuators B*, 1999, 61(1-3), 170-182.

[23] A.N. Pérez-García1 et all, Multilayer perceptron network with integrated training algorithm in FPGA, 11th Int. Conf. on Electrical Engineering, Computing Science and Automatic Control, pp 1-6, 2014.

[24] A. Tisan, M. Cirstea, S. Oniga, A. Buchman, Artificial olfaction system with hardware on-chip learning neural networks, 12th International Conference on Optimization of Electrical and Electronic Equipment (OPTIM), pp884-889, 2010.