Efficient Implementation of the Backpropagation Algorithm in FPGAs and Microcontrollers

Francisco Ortega-Zamorano, José M. Jerez, Daniel Urda Muñoz, Rafael M. Luque-Baena, and Leonardo Franco, *Senior Member, IEEE*

***Abstract*— The well-known backpropagation learning algorithm is implemented in a field-programmable gate array (FPGA) board and a microcontroller, focusing in obtaining efficient implementations in terms of a resource usage and computational speed. The algorithm was implemented in both cases using a training/validation/testing scheme in order to avoid overfitting problems. For the case of the FPGA implementation, a new neuron representation that reduces drastically the resource usage was introduced by combining the input and first hidden layer units in a single module. Further, a time-division multiplexing scheme was implemented for carrying out product computations taking advantage of the built-in digital signal processor cores. In both implementations, the floating-point data type representation normally used in a personal computer (PC) has been changed to a more efficient one based on a fixed-point scheme, reducing system memory variable usage and leading to an increase in computation speed. The results show that the modifications proposed produced a clear increase in computation speed in comparison with the standard PC-based implementation, demonstrating the usefulness of the intrinsic parallelism of FPGAs in neurocomputational tasks and the suitability of both implementations of the algorithm for its application to the real world problems.**

***Index Terms*— Embedded systems, field-programmable gate array (FPGA), hardware implementation, microcontrollers, supervised learning.**

I. Introduction

THE backpropagation (BP) algorithm is the most used learning procedure for training multilayer neural networks architectures. Even if the algorithm was originally proposed by Werbos in 1974 [1], it was not until 1986 that it become popularized through the work of Rumelhart *et al.* [2]. The BP algorithm is a gradient descent-based method that minimizes the error between target and actual network outputs, computing the derivatives of the error in an efficient way [3]–[5]. As a gradient descent algorithm, the search for a solution can get stuck in a local minima but in practice the algorithm is quite efficient, and as so it has been applied to a wide range of areas from pattern recognition [6], medical diagnosis [7], stock market prediction [8], and so on. Real-time applications require extra computational resources [9], involving in some cases also energy consumption restrictions, and thus the use of embedded (dedicated) systems [10] or low power consumption devices are needed, as in those cases a PC might not be the most adequate device for executing neural network models.

Field-programmable gate arrays (FPGAs) are reconfigurable hardware devices that can be reprogrammed to implement different combinational and sequential logic created with the aim of prototyping digital circuits, as they offer flexibility and speed. In recent years, the advance in technology have permitted to construct FPGAs with considerable the large amounts of processing power and memory storage, and as so they have been applied in several domains (telecommunications, robotics, pattern recognition tasks, infrastructure monitoring, and so on) [11]–[13]. In particular, FPGAs seem quite suitable for neural network implementations, as they are intrinsically parallel devices as is the processing of information in neural network models. Several studies have analyzed the implementation of neural networks models in FPGAs [14]–[19], but it is worth noting the difference between off and on chip implementations. In off-chip learning implementations [20], [21], the training of the neural network model is usually performed externally in a personal computer (PC), and only the synaptic weights are transmitted to the FPGA that acts as a hardware accelerator, while on-chip learning implementations includes both training and execution phases of the algorithm [18], [22], [23]. Existing specific implementations of the artificial neural network BP algorithm in FPGA boards include [24]–[26], noting that despite recent advances on the computational power of these boards, still the size of the neural architectures that can be implemented is quite limited. FPGA boards are predominantly programmed using hardware description languages, such as VHDL (VHSIC hardware description language) or Verilog and programming them is usually very time consuming.

Apart from FPGAs, other devices very much used in the neural network applications are microcontrollers, which are small and low-cost computers built on a single integrated circuit containing a processor core, memory, and programmable input/output peripherals built for dealing with specific tasks. These devices are commonly used in sensor nodes

Manuscript received May 7, 2014; revised October 24, 2014, January 26, 2015, and May 4, 2015; accepted July 21, 2015. Date of publication August 12, 2015; date of current version August 15, 2016. This work was supported by the Consejería de Economía, Innovación, Ciencia y Empleo, Junta de Andalucía under Grant P08-TIC-04026 and Grant P10-TIC-5770 and in part by the Ministerio de Economía y Competitividad through the Spanish National Science Foundation under Grant TIN2010-16556, including European Fund for Regional Development. *(Corresponding author: Leonardo Franco.)*

The authors are with the Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Málaga 29071, Spain (e-mail: fortega@lcc.uma.es; jja@lcc.uma.es; durda@lcc.uma.es; rmluque@unex.es; lfranco@lcc.uma.es).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNNLS.2015.2460991

(widely used in wireless sensor network [27]) usually under the environmental changing conditions, because they are an economic, small, and flexible solutions to interpret signals from the various sensors and take a decision according to the inputs received [19], [28]–[32]. An advantage of microcontrollers is that they can be easily programmed using standard programming languages, such as C, C++, and Java, while their main limitations are memory size and computing speed.

In this paper, we have implemented the BP algorithm in a VIRTEX-5 XC5VLX110T FPGA and an Arduino Due microcontroller. Our aim was twofold: 1) to obtain efficient implementations on both types of devices that permit its practical application in real-life problems and 2) to compare the efficiency between them and to a standard PC-based implementation. The organization of this paper is as follows. Section II includes details about the BP algorithm. The FPGA implementation is described in Section III, which contains four parts: the first three subsections describe each one of the three blocks used for the algorithm implementation, while the fourth subsection deals with specific implementation details. Section IV contains the microcontroller implementation of the algorithm. Section V presents the results of both implementations on a set of benchmark functions, together with a detailed analysis of the computational costs involved (number of cycles and execution times) and of the general functioning of the algorithm. Finally, the discussion and conclusions are drawn in Section VI.

II. Backpropagation Algorithm

The BP algorithm is a supervised learning method for training multilayer artificial neural networks, and even if the algorithm is very well known, we summarize in this section the main equations in relationship to the implementation of the BP algorithm, as they are important in order to understand the current work.

Let us consider a neural network architecture comprising several hidden layers. If we consider the neurons belonging to a hidden or output layer, the activation of these units, denoted by y_i, can be written as

$$y_i = g\left(\sum_{j=1}^{L} w_{ij} \cdot s_j\right) = g(h) \tag{1}$$

where w_{ij} are the synaptic weights between neuron i in the current layer and the neurons of the previous layer with activation s_j. In (1), we have introduced h as the synaptic potential of a neuron. g is a sigmoid activation function given by

$$g(x) = \frac{1}{1 - e^{-\beta x}}. \tag{2}$$

The objective of the BP supervised learning algorithm is to minimize the difference between given outputs (targets) for a set of input data and the output of the network. This error depends on the values of the synaptic weights, and so these should be adjusted in order to minimize the error. The error function computed for all output neurons can be defined as

$$E = \frac{1}{2}\sum_{k=1}^{p}\sum_{i=1}^{M}(z_i(k) - y_i(k))^2 \tag{3}$$

where the first sum is on the p patterns of the data set and the second sum is on the M output neurons. $z_i(k)$ is the target value for output neuron i for pattern k and $y_i(k)$ is the corresponding response output of the network. Using the method of gradient descent, the BP attempts to minimize this error in an iterative process by updating the synaptic weights upon the presentation of a given pattern. The synaptic weights between two last layers of neurons are updated as

$$\Delta w_{ij}(k) = -\eta\frac{\partial E}{\partial w_{ij}(k)} = \eta[z_i(k) - y_i(k)]g_i'(h_i)s_j(k) \tag{4}$$

where η is the learning rate that has to be set in advance (a parameter of the algorithm), g' is the derivative of the sigmoid function, and h is the synaptic potential previously defined, while the rest of the weights are modified according to similar equations by the introduction of a set of values called the deltas (δ), that propagate the error form the last layer into the inner ones.

Training and Validation Processes: The training procedure is executed a certain number of times (epochs) using the training patterns. In one epoch, the training patterns are all presented once in random ordering, adjusting the synaptic weights in an on-line manner. A well known and severe problem affecting all predictive algorithms is the problem of overfitting, caused by an overspecialization of the training procedure on the training set of patterns [33]. In order to alleviate this effect, one straightforward alternative is to split the set of available training patterns in training, validation, and test sets. The training set will then be used to adjust the synaptic weights according to (4), while the validation set is used to control overfitting effects, storing in memory the values of the synaptic weights that have so far led to the lowest validation error, so when the training procedure ends, the algorithm returns the stored set of weights. The test set is used to estimate the performance of the algorithm in unseen data patterns.

III. FPGA Implementation of the BP Algorithm

FPGAs [34] are reprogrammable silicon circuits, using prebuilt logic blocks and modifiable routing resources that can be configured to implement custom hardware. Besides the fact that FPGAs can be completely reconfigured allowing to change its behavior almost instantaneously by loading a new circuitry configuration, they can also be used as hardware accelerators, in particular for the neural-based applications given their intrinsic parallel computational capabilities. FPGAs are usually programmed using a hardware description language (VHDL). For the current implementation, we used the Virtex-5 OpenSPARC Evaluation Platform (ML509) that includes a Xilinx Virtex-5 XC5VLX110T FPGA. The board was programmed using the Xilinx ISE Design Suite 12.4 environment within the ISim M.81d simulator. Fig. 1 shows a picture of a Virtex-5 OpenSPARC board, and Table I shows

Fig. 1. Picture of a Virtex-5 OpenSPARC Platform used for the implementation of the C-Mantec algorithm.

TABLE I
MAIN SPECIFICATIONS OF THE VIRTEX-5 XC5VLX110T FPGA RELATED TO ITS AVAILABLE SLICE LOGIC

Device	Slice Registers	Slice LUTs	Bonded IOBs	BR	DSP48
Virtex-5 XC5VLX110T	69,120	69,120	34	148	64

some characteristics of the Virtex-5 XC5VLX110T FPGA, indicating its main logic resources. The table indicates for the mentioned board the number of slice registers, lookup tables (LUTs), bonded input–output banks, block RAM (BR), and DSP48 cores. Given that every computation in the FPGA has to be defined from the first principles, they usually contain digital signal processors (DSPs) for helping to perform certain operations. A DSP is a specialized microprocessor that has an architecture optimized for the fast operational needs of digital signal processing. A DSP process data in real time, making it ideal for applications that cannot tolerate delays [35], [36].

The FPGA implementation of the BP algorithm was carried out using three blocks: 1) control; 2) pattern; and 3) architecture blocks. The control block organizes the whole information process by sending and processing the information from the architecture and pattern blocks. The pattern block manages the exchange of information between the PC and the FPGA for reading the set of patterns to be stored in BRs, and also is used to send a given pattern to the architecture block, that it will be in charge of the training process. Circuit computations have been programmed using fixed-point arithmetic, which is the standard way to work with FPGA boards. Floating-point operations can be codified in an FPGA but they tend to be inefficient in comparison with fixed-point representation [25]. We describe below the organization of each one of the three blocks in Sections III-A–III-C. Section III-D comments on the specific implementation details. Fig. 2 shows a diagram of the FPGA design, where the three blocks used are shown together with the information that is exchanged between them.

A. Pattern Block

The pattern block manages the data exchange between the PC and the FPGA board through the serial communication RS-232 port of the device. This port has been used because it can be easily implemented in VHDL and ported to other architectures.

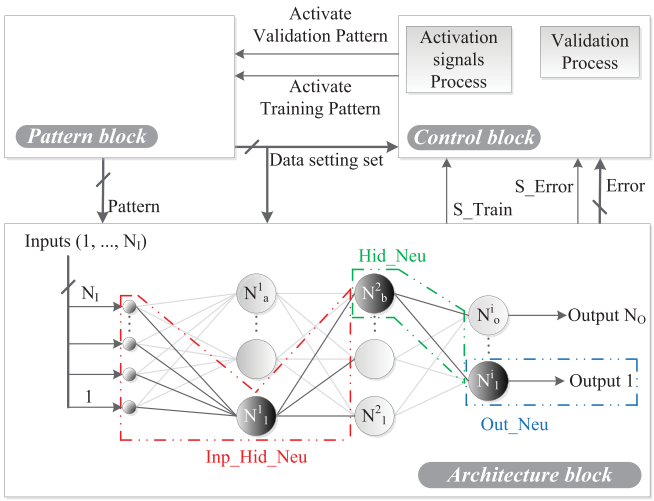

Fig. 2. Scheme of the FPGA design, where control, pattern, and architecture blocks are shown together with the information exchanged between them.

To start the process, the user sends the set of parameters of the algorithm and the training patterns. The set of parameters specifies the number of training (#Train) and validation patterns (#Valid), the number of neurons in each layer ($\#N_i$), the number of epochs (#Epoch), and the learning rate value η. The training and validation data sets are stored in the distributed-RAM block of the FPGA, storing the training set in the first positions and the validation in the following ones. Two bytes (1 byte = 8 bits) of memory are used for representing each attribute and each class of a pattern, and thus the total occupied memory of the data set is defined by the equation

$$\#\text{bytes} = 2 \cdot (N_I + N_O) \cdot (\#\text{Valid} + \#\text{Train}) \tag{5}$$

where N_I is the number of inputs (attributes) and N_O is the number of output classes.

During the execution of the algorithm, the pattern block might receive two different signals from the control block in order to send a random training or validation pattern. To avoid training several times a given pattern, the memory position of the last sent pattern is switched with the one corresponding to the final eligible memory position, while the number of eligible memory positions is reduced by 1. This action is repeated until the eligible memory is null, finishing the epoch at this moment and starting a following one. The same process is applied for both training and validation sets independently.

B. Control Block

The control block organizes the whole information flow process within the FPGA board by sending and processing the information from the architecture and pattern blocks. The structure of this block is organized into two main processes.

1) The main function of this block is to control two activation signals that indicate whether a training or a validation pattern should be sent to the architecture block. In order to perform this action, the control block receives a signal value from the pattern block that indicates the total number of training (#train) and

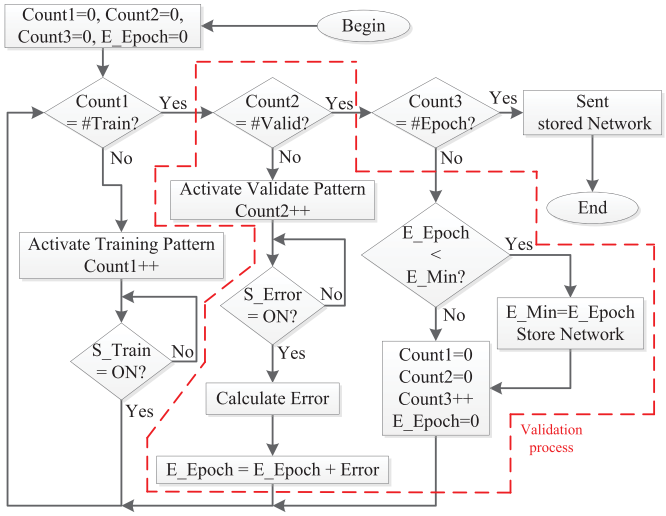

Fig. 3. Flowchart of the FPGA control block operations. The region inside the dashed line corresponds to the validation procedure implemented.

validation (#val) patterns sets for the whole learning procedure.

2) The secondary process of this block is to execute the validation process, included with the aim of avoiding overfitting effects.

In essence, this process computes an error value using the validation set of patterns to store the synaptic weight values that have led to the smallest validation error thus far as the training of the network proceeds. The implementation of the whole validation process in the FPGA is detailed in Section III-B.

When the computations starts, the set of patterns are loaded into the pattern block that sends a signal to the control block in order to start the execution of the algorithm. Fig. 3 shows a flowchart of the control block operations. At the beginning of the process, a set of counters related to the number of training patterns, the number of validation patterns, and the number of epochs are initialized to zero. If the number of actual training patterns has not reached the value #Train (set by the user), the training procedure starts by sending a signal to the pattern block indicating that a random chosen training pattern should be sent to the architecture block. The architecture block will then train the network, sending back a signal (*S*_Train) to the control block when the training of this pattern finishes, increasing the trained pattern counter Count1. When this value gets equal to the total number of training patterns, then the validation process start (this step is described in detail below). After the validation process, an epoch counter is used for checking whether the whole training-validation procedure should continue or not, as the previous steps are repeated until the maximum number of epochs (#Epoch) is reached.

Validation Process: The validation process, included to prevent overfitting problems, is executed after finishing a training epoch. This process requires the storing of the lowest validation error obtained so far (as the training procedure advances) together with the synaptic weights that led to this error. When this procedure is activated at the end of a training epoch, it computes the mean square error (MSE) for the validation set, and if this value is lower than the stored one, then it is saved together with all synaptic weights in a BR using a first-in first-out (FIFO) procedure. As the neurons included in the network architecture are indexed, the control block demands sequentially the set of synaptic weights associated with each neuron so they can be stored in a single FIFO RAM while preventing memory collision problems. The flowchart of the validation process is included in Fig. 3.

TABLE II
NUMBER OF EMPLOYED RESOURCES FOR THE REALIZATION OF EACH TYPE OF POSSIBLE NEURON

Resource	Resource for type of Neuron			
	Input	InpHid	Hidden	Output
LUTs	524	1126	923	502
Register	254	396	391	255
DSP48	1	1	1	1
Block RAM	1	1	1	1

C. Architecture Block

The architecture block is in charge of the physical implementation of the neural network architecture. The number of layers and the maximum number of neurons in each layer has to be predefined by the user before the execution of the algorithm.

Previous works [23], [37], [38] use three different types of neurons, corresponding to input, hidden and output layer neurons, as they all have different functionalities. Nevertheless, we decide to use a different approach in order to optimize further the FPGA resources, and thus the proposed implementation eliminate input layer neurons as they are included together with the first hidden layer neurons in a new module that we name input-hidden neurons. The definition of this new type of module is possible, mainly because the input layer neurons do not process the information as they simple act as input to the network. The implementation of the neurons consists of a group of LUTs with a specific functionality of the BP algorithm. The input-hidden neurons manage the input data, the synaptic weights between the input and the first hidden layer, the output computation of the first hidden neuron and also the synaptic weights connecting to the output or to a further hidden layer. The hidden layer modules (in case they are included) compute the neuron activation and store the values of the synaptic weights connecting to further neurons. Finally, the output modules evaluate the value of the output units in order to compute the error of the presented pattern. A scheme of a two hidden layer neural network is shown in Fig. 2 where the three different types of modules are indicated. Table II shows the number of employed resources by each type of neuron with a word size of 32 b, 16 for the integer part (N_1), and 16 to the decimal part (N_2). The election of the word size is described in Section III-D1.

To specify a given architecture the number of active neurons in each layer should be sent to the FPGA as part of the setting data set. However, the system is composed of a determined number of neurons and layers, so that the architecture of the network and the maximum number of neurons are

TABLE III
BOARD RESOURCES NEEDED FOR DIFFERENT SIZE NEURAL ARCHITECTURES FOR THE PROPOSED AND ALTERNATIVE METHODS

Architecture	*Type*	*Resource*			
		LUTs	Regis.	BR	DSP
$10-3-1$	Conv.	11044	6676	79	15
	Gomp.	8043	2243	–	70
	Prop.	6413	4151	69	5
$10-6-3-2$	Conv.	17084	9277	86	22
	Gomp.	20021	5342	–	169
	Prop.	13062	6767	76	12
$10-50-1$	Conv.	54425	25053	126	62
	Prop.	59335	22763	116	52
$30-30-10-2$	Conv.	56177	26478	137	73
	Prop.	46547	19008	107	43
$50-10-10-5$	Conv.	49703	24503	140	76
	Prop.	25533	11853	90	26
$60-15-10-5$	Conv.	59558	28998	155	91
	Prop.	33163	13833	95	31
Reduction mean		25.8%	35.2%	23.5%	50.1%

predetermined and delimited by the resources of the device. The novel layer (the first layer blocks), that is employed in this implementation, reduces the required resources for any architecture. Table III shows the board resources needed by the proposed method (Prop.) for different size neural architectures in comparison with a conventional implementation (Conv.), and also to the published results in [38] (Gomp.).

D. Implementation Details

We describe below details related to the choice of synaptic weights precision, for carrying out the implementation of products, and about the computation of the sigmoid function used as the transfer function of the neurons.

1) Synaptic Weights Precision: The representation of the synaptic weights can be chosen according to the available resources, considering that requiring higher accuracy may need a larger representation, leading also to an increase in the number of LUTs per neuron (consequently reducing the maximum number of available neurons), and a decrease in the maximum operation frequency of the board. On the other hand, synaptic weights accuracy cannot be much reduced, as a proper operation of the BP algorithm requires a certain level of precision [25], [39]. A synaptic weight is represented by a bit array with integer and fractional parts of length N_1 and N_2. N_1 determines the minimum and maximum values that can be represented $-2^{(N_1-1)}$ to $2^{(N_1-1)}$ while N_2 defines the accuracy $2^{(-N_2)}$. The number of bits needed to represent all possible discrete values within a certain range of positive values depends on the difference between the maximum and minimum of the interval, and can be obtained from the following equation:

$$\#\text{bits} = \log_2((1 + \max(w_{ij}))/(\min(w_{ij}))). \quad (6)$$

Table IV shows the number of LUTs needed to represent each type of neuron modules as a function of the number of bits used for representing the synaptic weights. N_1 and N_2 indicate the integer and fractional parts of the synaptic weight representation. The last column shows the maximum whole system operation frequency allowed for the chosen representation.

TABLE IV
NUMBER OF LUTS PER EACH TYPE OF POSSIBLE NEURON MODULE ACCORDING TO THE NUMBER OF BITS USED FOR REPRESENTING THE SYNAPTIC WEIGHTS

N_1	N_2	LUTs per type of Neuron				f_{max} (MHz)
		Input	InpHid	Hidden	Output	
8	8	347	723	592	343	191.2
8	12	390	871	671	409	186.7
8	16	433	1028	763	448	183.7
12	12	430	913	740	425	183.7
12	16	476	1031	859	475	180.6
16	16	524	1126	923	502	177.3

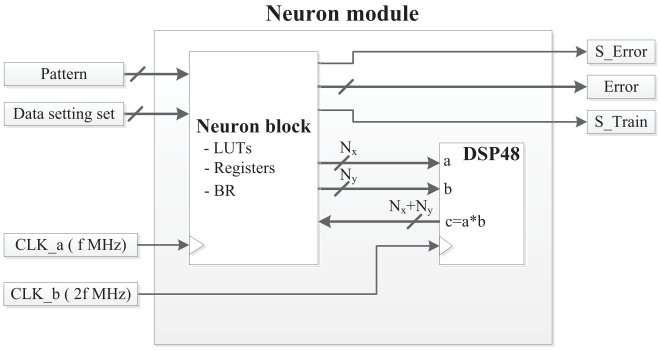

Fig. 4. Scheme of an implemented neuron that uses a time-division multiplexed strategy to execute several multiplications. Neuron and DSP blocks are synchronous but the DSP uses a frequency two times larger than the one used by the neuron block, so that a product operation could be completed in one operation cycle of the FPGA.

2) Product Implementation: The execution of the BP algorithm requires the computation of several products, mainly between neuron activations and synaptic weights values [see (1)–(4)], and so an efficient implementation of this operation is crucial in order to optimize board resources (affecting the number of LUTs per neuron required and the operation frequency of the FPGA). Multipliers can be implemented by shifters and adders, following the approach presented in [40] or by available specific DSP cores in the FPGA. The number of required LUTs for the first type of implementation is proportional to the bit size of the input data, as for example, for two vectors with N_a and N_b bits length, respectively, the product requires $N_a \times N_b$ LUTs while for the second type of implementation, one DSP for each neuron is needed (clearly this puts a limitation in the maximum number of neurons in the system). We decided to use the DSP-based strategy as the board frequency operation can be up to four times faster, as we measured the operation of the board without using the DSPs.

For an efficient use of the DSP resources, we implemented a time-division multiplexing scheme, using only one multiplier block per neuron and thus performing sequentially the computation of several products. This time-division multiplexing scheme is shown in Fig. 4. The neuron module comprises a first block (we named it neuron block) that includes several LUTs, registers and one BR, while a second block consists just of a DSP. Both blocks are synchronous but the DSP uses a frequency two times larger than the one used by the

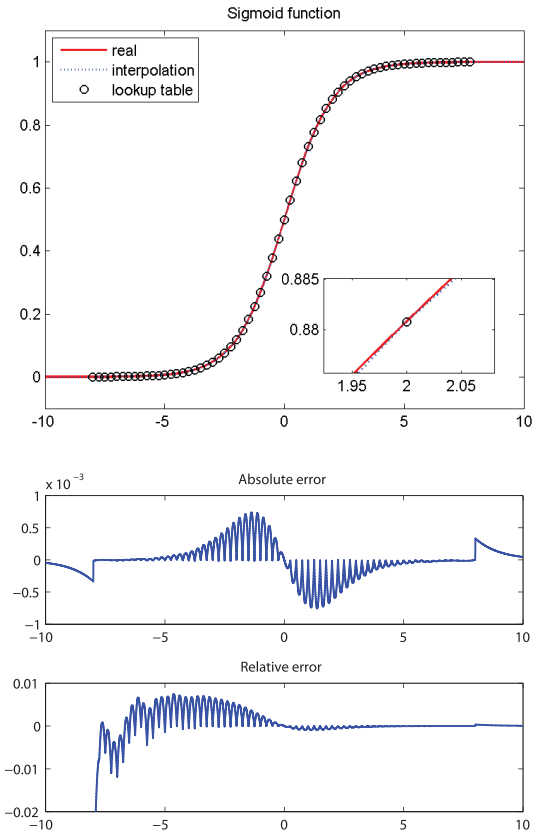

Fig. 5. Computation of the exponential function and its approximation based on an LUT plus a linear interpolation scheme (top graph). Absolute (middle graph) and relative errors (bottom graph) committed in the approximation of the function (see text for more details).

neuron block, so that a product operation could be completed in one operation cycle of the FPGA.

3) Implementation of the Sigmoid Function: The operation of the neurons involves the computation of sigmoid functions for obtaining the output value of the neurons. As we are using a fixed-point representation (as it is more efficient than the floating-point one), then the computation of the sigmoid function needs the use of an approximation. An LUT containing equispaced values of the function has been created to obtain certain number of output values. Neverthless, as high precision values are needed for the correct execution of the algorithm, the computation of the function approximation was further complemented by a linear interpolation procedure using the two adjacent tabulated values (lower and larger) with respect to the input. Storing table values requires large amounts of memory, and as one table per neuron is needed, this number should be optimized. The 64 tabulated values were used, as this ensures the obtention of absolute errors lower than 10^{-3}. These values start from -8 to 8 with 0.25 increasing steps (values lower than -8 and higher than 8 were set to 0 and 1, respectively). Fig. 5 (top) plots real, tabulated, and interpolated values for the sigmoid function, with the inset plot showing an enlargement of a portion of the curve. Fig. 5 (middle and bottom graphs) shows the absolute and relative errors involved in the computation of the sigmoid function in the range from -10 to 10.

Fig. 6. Picture of an Arduino DUE board used for the implementation of the C-Mantec algorithm.

IV. Microcontroller (μC) Implementation

We have further implemented the BP algorithm in an Arduino DUE microcontroller. We describe below in several sections all the details of the implementation process, highlighting the results of a comparison carried out between using a fixed-point representation or a floating-point one.

A. Arduino Board

Arduino is a single-board microcontroller designed to make the process of using electronics in multidisciplinary projects more accessible [41]. The hardware consists of a simple open source board designed around a 32-b Atmel ARM core microcontroller, and the software includes a standard programming language compiler that runs in a standard PC and a boot loader for loading the compiled code on the microcontroller. Arduino is a descendant of the open-source Wiring platform and is programmed using a Wiring-based language (syntax and libraries), similar to C++ with some slight simplifications and modifications, and a processing-based integrated development environment.

The Arduino DUE is based on the SAM3X8E ARM Cortex-M3 CPU [42], and it has 54 digital input/output pins (of which 12 can be used as PWM outputs), 12 analog inputs, 4 UARTs (hardware serial ports), an 84 MHz clock, an USB OTG capable connection, 2 DAC (digital to analog), and a reset and erase buttons. The SAM3X has 512 kB (2 blocks of 256 kB) of flash memory for storing code, it also comes with a preburned bootloader that is stored in a dedicated ROM memory. The available SRAM amounts to 96 kB in two contiguous banks of 64 and 32 kB. A picture of the Arduino DUE board is shown in Fig. 6.

B. Learning and Execution Phases of the Algorithm

The implementation of the BP neural network learning model comprises two phases: 1) the learning phase where the synaptic weights of the chosen architecture are adjusted according to the set of patterns presented to the network and 2) the execution phase in which the microcontroller outputs a signal in response to sensed input data according to the model previously adjusted.

The learning phase has been divided into two different processes: 1) loading of input patterns and 2) neural network training. Data can be loaded into the microcontroller memory on-line by I/O pins or by a serial communication USB port (this last option was the used one in the current implementation for simplicity reasons), but in both cases, the patterns have

to be stored into the memory board because the learning process works in cycles in which patterns are repeatedly used. The microcontroller memory has been further divided in two parts, one of 64 kB for the loading process, which is used for storing the inputs patterns, and the second part comprising 32 kB of the memory to be used for the system variables. This second part of memory is used for storing the value of the synaptic weights, and all other variables of the algorithm, like the activation value of the neurons, the deltas, the learning rate, and so on. The neural network training phase consists of the BP algorithm itself that was implemented with a validation phase to avoid overfitting effects. Once the training phase finishes the synaptic are stored in the memory block of 64 kB.

The execution phase is programmed to be carried out from external data, as usually the microprocessor will be used as an independent sensor. In this mode, a pattern would be read from a sensor connected to one of the ports of the board, and the previously trained model will be executed to obtain the neural network output.

C. Pattern Storage

The number of bits used for representing each of the inputs of a pattern has to be decided in advance of the implementation. In the present case, 8 b have been used to represent each variable, considering that these input values have to be previously normalized between 0 and 255. Using this representation for the patterns, the maximum number of samples that can be stored in a 64-kB memory is given by the following equation:

$$N_P \cdot (N_I + N_O) \leq 65\,536 \tag{7}$$

where N_P is the number of patterns, N_I is the number of input variables (the dimension of the patterns), and N_O is the number of outputs of the patterns that determines the number of output of the neural architecture.

D. Data Type Representation

The microcontrollers are devices with limited computing power, so in order to speed up the learning process, we decided to utilize a fixed-point data representation. We note that floating point is the usual data type representation used in this kind of device but this representation is not always the most efficiency. This paradigm shift involves important changes in the way the BP algorithm is programmed but in return offers a faster learning process and a smaller size representation of variables. The following list gives the details of the type of representation used for variables related to the implementation of the neurons:

1) *Deltas (δ):* 2 byte integers;
2) *Synaptic Weights (w):* 2 byte integers;
3) *Outputs (y):* 2 byte integers.

The previous choice for the representation of the neural network-related variables affects the maximum network size that can be utilized. The total number of neurons (N_N) in the whole architecture can be expressed as the sum of the number of neurons in each layer ($N_N = N_{N1} + N_{N2} + \cdots$), so the maximum number of neurons used in each layer should verify the following constraint:

$$\begin{aligned} &2 \cdot (N_{N1} + N_{N2} + \cdots) \\ &\quad + 2 \cdot (N_I \cdot N_{N1} + N_{N1} \cdot N_{N2} + N_{N2} \cdot N_{N3} + \cdots) \\ &\quad + 2 \cdot (N_{N1} + N_{N2} + \cdots) \leqslant 32\,768 \end{aligned} \tag{8}$$

where the first term relates to the variable storage space for the δs, the second term account for the synaptic weights between all the layers of neurons, and the last term is related to the output value of the neurons in each layer. (N_{N1} represents the number of neurons in the first layer, N_{N2} of the second layer and so on, while N_I is used for the number of inputs).

From the 2 bytes used for representing the variables of the system, 10 b were used for the decimal part, and the remaining 6 for the integer part, and thus the value of the system variables ranges between 32 and -32. A special case was the representation used for the variable computing the summation of the synaptic potential, because in order to avoid saturation effects a 4 bytes representation was used.

E. Computation of the Sigmoid Function

The computation of the sigmoid function can be implemented using the specific arithmetic and logic unit for resolving the exponential function. The previous computation involves two different variable conversions, the first related to the input values of the sigmoid (casting from integer to a floating-point representation) and the second conversion is done to the output value in a reverse casting. The computational cost of implementing the previous method is high, with an approximate time of operation of 62 μs, and thus an alternative method based on an LUT plus linear interpolation of adjacent values, similar to the one used in the FPGA implementation, was chosen. The method is explained in detail in Section III-D3, and in this case the computation time employed is reduced to 2 μs (97% reduction in comparison with the first mentioned method).

F. Fixed-Point Versus Floating-Point Representation Comparison

In Fig. 7, top, middle, and bottom show the number of times that the implementation based on integer data type is faster in comparison with the floating-point representation as a function of the number of neurons in the different layers of the neural architecture. An architecture with only one hidden layer has been used to compute the values represented in the figure, where N_N represents the number of neurons, N_I the number of inputs, and N_O is used for the number of outputs. Fig. 7 top shows the comparison for variable values of N_N and N_I (keeping fixed N_O equal to 1), Fig. 7 middle is computed for different values of N_N and N_O with N_I equals to 10, and finally Fig. 7 bottom represents the values obtained as a function of N_I and N_O for N_N equals to 5.

V. Results

We analyze in this section several aspects in relationship to the two implementations of the BP algorithm carried out

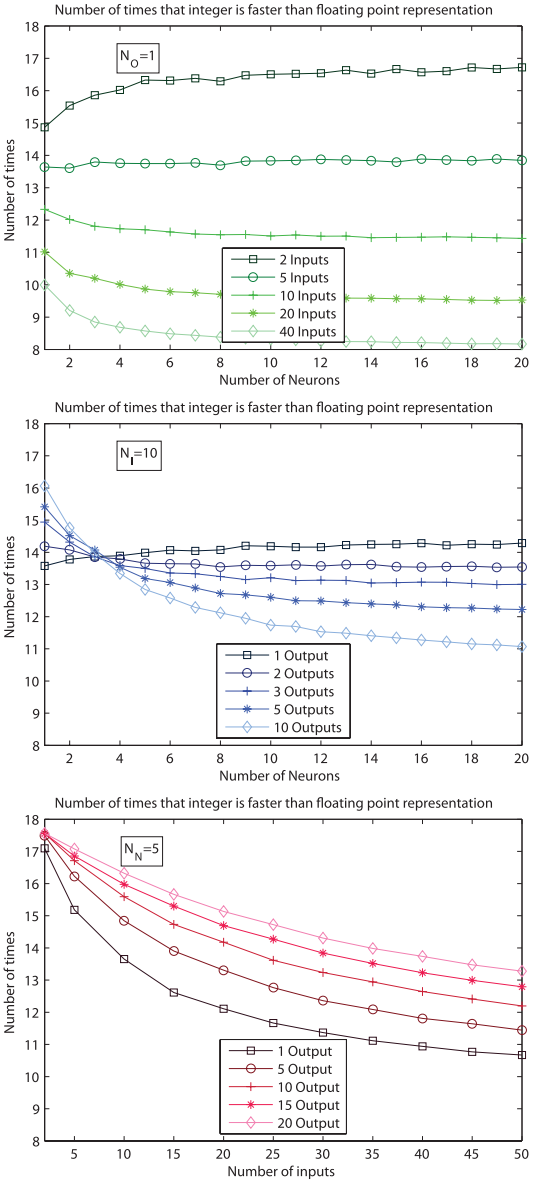

Fig. 7. Number of times that the fixed-point representation used in the microcontroller is faster than the floating point one for different number of neurons in the layers of a one hidden layer architecture (see text for more details).

in an FPGA board and in an Arduino DUE microcontroller, considering also a third implementation of the algorithm in PC for comparison purposes. The PC implementation of the algorithm has been executed under MATLAB code and run in an Intel(R) core (TM) i5-3330 CPU at 3 GHz with 16 GB of RAM memory. All three implementations of the algorithm follow the same operation steps and the only evident differences between them are the random number generator used for the initialization of the synaptic weights, the type of data representation used in each case, and the computation of the sigmoid function. The FPGA implementation uses an LFSR random number generation routine, while the microcontroller and the MATLAB code use the built-in random and randn functions, respectively.

Fig. 8 shows the estimated number of clock cycles that each implementation employs for the learning of an input pattern

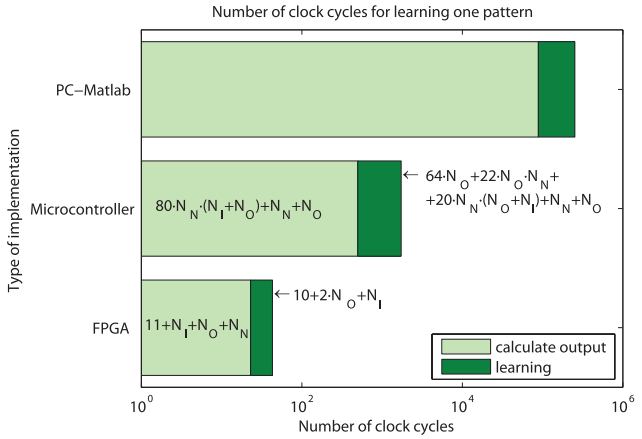

Fig. 8. Number of clock cycles involved in the learning process of a single pattern according to the type of implementation used for the case of a 4–5–3 neural network architecture.

for a single hidden layer neural network architecture with a 4–5–3 structure. Each bar in the graph is further divided in two parts: 1) computation of the output of the network in response to the input pattern (clear part of the bar) and 2) cycles involved in the modification of the synaptic weights including the backward phase of the BP algorithm (dark part of the bar) (note the logarithmic scale of the graph). The number of clock cycles in the FPGA implementation has been measured straightforwardly using the ISIM simulator. The estimation of the number of cycles for the microcontroller has been done by computing the time that takes the computation of each instruction of the algorithm, multiplying this value by the clock frequency of the microcontroller (80 MHz) and then summing over all the instructions involved in the algorithm. The number of cycles used in the PC-MATLAB implementation cannot be computed directly and has been estimated by measuring the total computation time and multiplying this value by the CPU frequency (3 GHz), even if a strict evaluation of the number of cycles should involve taking into account other factors like instructions of the operative system, libraries, and so on. The method used in the FPGA and microcontroller implementations permits to estimate the number of operation cycles as a function of the number of neurons in one hidden layer architecture (N_I, N_N, and N_O) and is represented inside the bars in Fig. 8. It is worth noting that even if in principle microcontroller and computer codes used are quite similar, there are differences regarding the implementations as the data representation used is different (fixed and floating point, respectively), and the computation of the sigmoid function values is done in a different way (tabulated values plus interpolation for the microcontroller versus ALU in the case of a computer). To test the correct implementation of the BP algorithm in the FPGA and microcontroller devices, we tested the training, validation, and test errors on a set of benchmark problems from the UCI database [43] frequently used in the literature. Table V shows the accuracy (generalization ability) values obtained for the three implementations of the algorithm for eleven benchmark problems. The first three columns indicate the data set name, number of inputs and outputs, respectively, while the

TABLE V
ACCURACY

Function	$\#I$	$\#O$	PC	FPGA	μC
Diabetes	8	2	78.31	79.35	79.13
Cancer	9	2	95.63	95.73	95.60
Statlog (Heart)	13	2	78.52	78.27	78.26
Climate	18	2	93.27	94.14	94.51
Ionosphere	34	2	88.21	87.57	87.14
HeartC	35	2	78.80	80.11	80.22
Iris	4	3	92.22	92.77	90.89
Balance Scale	4	3	87.93	87.82	87.61
Seeds	7	3	97.62	96.51	96.66
Wine	13	3	88.89	87.04	86.67
Glass	10	6	93.85	91.54	92.31
Average			88.48	88.26	88.09

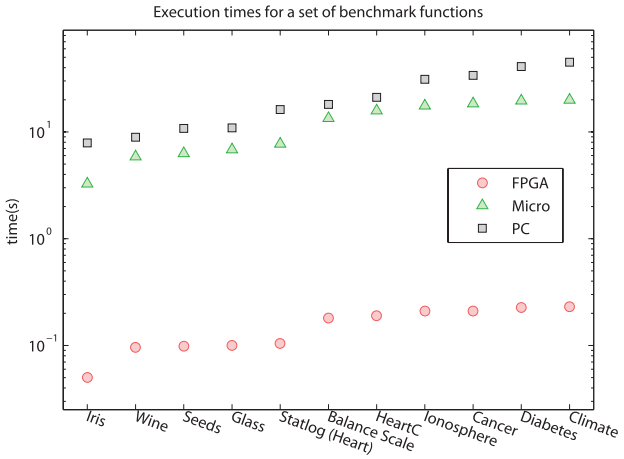

Fig. 9. Execution times (in seconds) for the whole learning process (training and validation) for the set of benchmark functions used for verifying the correct implementation of the algorithm in the FPGA and microcontroller using neural network architectures with a single hidden layer (see text for more details).

last three columns shows the accuracy obtained using neural network architectures with five neurons in the single hidden layer, as the number of inputs and outputs is determined by the problems themselves. We have not optimized the neural architecture for each problem, as our aim is to demonstrate the correct implementation of the algorithm and not to obtain optimal values of prediction accuracy. For carrying out the simulations, a training, validation, and test sets splitting was used in a 50%–20%–30% scheme; in which the validation set was used to find the number of epochs for evaluating the test error, the maximum number of epochs was set to 1000, and the learning rate was equal to 0.2.

We further computed execution times for the whole learning process (training and validation) for the same set of benchmark functions mentioned above, and the results are shown in Fig. 9 for the FPGA, microcontroller and PC versions of the BP algorithm (note the logarithmic scale used in the y-axis of the figure). We also perform an analysis to see how FPGA and microcontroller performance behaves in comparison with the PC implementation as the complexity of the functions grow. We have used the execution time needed in the PC implementation time_{PC} as an estimation for the complexity of the benchmark functions, to obtain that the number of times that the FPGA implementation is faster than the PC

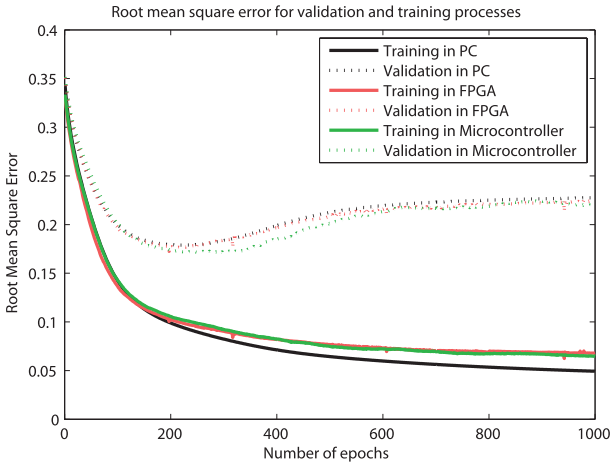

Fig. 10. RMSE for the training and validation process when learning the Iris data set using a 4–5–3 architecture for the three different implementations used.

($\#\text{times}_{\text{FPGA}}$) grows as $\#\text{times}_{\text{FPGA}} = 94 + 2 \cdot \text{times}_{\text{PC}}$ (Pearson correlation coefficient equals to 0.753), while the analysis for the microcontroller shows not significant performance increase: $\#\text{times}_{\mu\text{C}} = 1.6 + 0.0083 \cdot \text{times}_{\text{PC}}$ (Pearson correlation coefficient equals to 0.25).

Fig. 10 shows the root MSE (RMSE) obtained for the training and validation processes when the BP algorithm is implemented in the FPGA, μC, and PC as a function of the number of epochs for the Iris data set using a 4–5–3 architecture (similar results were observed for all data sets). It can be seen that the training error always decreases as training advances being lower for the PC implementation than for other two, indicating that a more precise representation (32-b floating point) helps to adjust the synaptic weights during training. However, for the validation RMSE, the three curves are quite similar noting also that the values increase at certain point of the process (approximately at 200 epochs), indicating overfitting effects and justifying the use of a validation set. As in general the interest on the application of supervised neural networks to practical problems is related to prediction, validation, and test errors are the important features and thus the results confirm that the representation used for the FPGA and μC (16-b fixed point) is adequate.

VI. Discussion and Conclusion

The BP algorithm has been successfully implemented in an FPGA board and Arduino microcontroller, in a learning paradigm that includes a validation scheme in order to prevent overfitting effects. The implementation of the algorithm in the FPGA board involved several challenges as hardware programming is a totally different paradigm approach in comparison with standard software programming, and as such, we have first introduced a new neuron representation that permits to increase the efficiency of the traditional implementation, obtaining an average reduction of 25.8% in the total number of LUTs needed to implement different architectures, as shown in Table III. Further improvements are related to a time-division multiplexing scheme for carrying out product computations

taking advantage of the FPGA DSP built-in cores, and an LUT plus linear interpolation scheme for computing the transfer function of the neurons (the sigmoid function). At the time of a real implementation, the limitation in terms of the size of the neural network architectures that can be simulated would come from the specific FPGA board used, and this analysis can be done from the results shown in Table II. In our case in which we are using a Xilinx Virtex V XC5VLX110T board the main limitation comes from the number of available DSP cores, as the mentioned board includes 64 DSP cores, and thus this factor limits the maximum number of neurons in the architecture to 63, as an extra core is needed in the validation process. If we compare these new results with the previous published works, for example those appearing in [38], we see a significant efficiency increase that will permit the utilization of much larger neural architectures. Nevertheless, the efficiency increase for a particular case would depend on the values of the combination of neural network architecture parameters and resources of the FPGA board used.

Considering the microcontroller implementation, the standard floating-point data type representation has been changed to a more efficient one based on a fixed-point scheme, reducing system memory variable usage and an increase in computation speed, obtaining that the fixed-point representation is $\sim$10 times faster than the floating point one (see Fig. 7).

The results shown in Fig. 10 indicate that the representation used for the FPGA and microcontroller was adequate as validation and test errors were similar to the PC implementation of the algorithm. The use of a lower precision representation affects the learning error (it is lower for the PC implementation), but it is not relevant regarding prediction accuracy. We hypotethize that this fact might be related to the effect observed when noise is added both to input and synaptic weight values [44], that instead of having negative effects, it can help to improve generalization by preventing the overfitting effects.

An estimation of the computation time (Fig. 9 involved in relationship with the three implementations of the BP algorithm (FPGA, μC, and PC) shows the potential advantages of using an FPGA board as a hardware accelerator device for neurocomputing applications, obtaining a speedup of hundred of times with an improvement increasing with the complexity of the problem, highlighting the intrinsic parallelism of the device.

As an overall conclusion, this paper shows the potential advantages of using FPGA boards as hardware accelerator devices for neurocomputing applications giving their intrinsic parallel capabilities, while in relationship to the use of neural networks in microcontrollers, we highlight the on-chip characteristic of the presented implementation that will permit its use in remote sensors using a standalone operation mode.

REFERENCES

[1] P. J. Werbos, "Beyond regression: New tools for prediction and analysis in the behavioral sciences," Ph.D. dissertation, Dept. Appl. Math., Harvard Univ., Cambridge, MA, USA, 1974.

[2] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[3] K. Mehrotra, C. K. Mohan, and S. Ranka, *Elements of Artificial Neural Networks*. Cambridge, MA, USA: MIT Press, 1997.

[4] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed. Upper Saddle River, NJ, USA: Prentice-Hall, 1998.

[5] R. D. Reed and R. J. Marks, II, *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. Cambridge, MA, USA: MIT Press, 1998.

[6] J. Li, K. Ouazzane, H. B. Kazemian, and M. S. Afzal, "Neural network approaches for noisy language modeling," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 24, no. 11, pp. 1773–1784, Nov. 2013.

[7] K. Misra, S. Chattopadhyay, and D. Kanhar, "A hybrid expert tool for the diagnosis of depression," *J. Med. Imag. Health Informat.*, vol. 3, no. 1, pp. 42–47, 2013.

[8] H. Mo and J. Wang, "Volatility degree forecasting of stock market by stochastic time strength neural network," *Math. Problems Eng.*, vol. 2013, Sep. 2013, Art. ID 436795.

[9] G.-B. Huang and C.-K. Siew, "Real-time learning capability of neural networks," *IEEE Trans. Neural Netw.*, vol. 17, no. 4, pp. 863–878, Jul. 2006.

[10] S.-M. Baek and J.-W. Park, "Hessian matrix estimation in hybrid systems based on an embedded FFNN," *IEEE Trans. Neural Netw.*, vol. 21, no. 10, pp. 1533–1542, Oct. 2010.

[11] E. Monmasson, L. Idkhajine, M. Cirstea, I. Bahri, A. Tisan, and M. W. Naouar, "FPGAs in industrial control applications," *IEEE Trans. Ind. Informat.*, vol. 7, no. 2, pp. 224–243, May 2011.

[12] D. Bacon, R. Rabbah, and S. Shukla, "FPGA programming for the masses," *Queue*, vol. 11, no. 2, pp. 40–52, 2013.

[13] P. Conmy and I. Bate, "Component-based safety analysis of FPGAs," *IEEE Trans. Ind. Informat.*, vol. 6, no. 2, pp. 195–205, May 2010.

[14] D. L. Ly and P. Chow, "High-performance reconfigurable hardware architecture for restricted Boltzmann machines," *IEEE Trans. Neural Netw.*, vol. 21, no. 11, pp. 1780–1792, Nov. 2010.

[15] Q. N. Le and J.-W. Jeon, "Neural-network-based low-speed-damping controller for stepper motor with an FPGA," *IEEE Trans. Ind. Electron.*, vol. 57, no. 9, pp. 3167–3180, Sep. 2010.

[16] W. Mansour, R. Ayoubi, H. Ziade, R. Velazco, and W. El Falou, "An optimal implementation on FPGA of a Hopfield neural network," *Adv. Artif. Neural Syst.*, vol. 2011, Jan. 2011, Art. ID 7.

[17] L.-W. Kim, S. Asaad, and R. Linsker, "A fully pipelined FPGA architecture of a factored restricted Boltzmann machine artificial neural network," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 1, Feb. 2014, Art. ID 5.

[18] F. Ortega-Zamorano, J. M. Jerez, and L. Franco, "FPGA implementation of the C-Mantec neural network constructive algorithm," *IEEE Trans. Ind. Informat.*, vol. 10, no. 2, pp. 1154–1161, May 2014.

[19] F. Ortega-Zamorano, J. M. Jerez, J. L. Subirats, I. Molina, and L. Franco, "Smart sensor/actuator node reprogramming in changing environments using a neural network model," *Eng. Appl. Artif. Intell.*, vol. 30, pp. 179–188, Apr. 2014.

[20] S. Himavathi, D. Anitha, and A. Muthuramalingam, "Feedforward neural network implementation in FPGA using layer multiplexing for effective resource utilization," *IEEE Trans. Neural Netw.*, vol. 18, no. 3, pp. 880–888, May 2007.

[21] T. Orlowska-Kowalska and M. Kaminski, "FPGA implementation of the multilayer neural network for the speed estimation of the two-mass drive system," *IEEE Trans. Ind. Informat.*, vol. 7, no. 3, pp. 436–445, Aug. 2011.

[22] A. Dinu, M. N. Cirstea, and S. E. Cirstea, "Direct neural-network hardware-implementation algorithm," *IEEE Trans. Ind. Electron.*, vol. 57, no. 5, pp. 1845–1848, May 2010.

[23] J. Shawash and D. R. Selviah, "Real-time nonlinear parameter estimation using the Levenberg–Marquardt algorithm on field programmable gate arrays," *IEEE Trans. Ind. Electron.*, vol. 60, no. 1, pp. 170–176, Jan. 2013.

[24] A. R. Omondi and J. C. Rajapakse, Eds., *FPGA Implementations of Neural Networks*. Dordrecht, The Netherlands: Springer-Verlag, 2006.

[25] A. W. Savich, M. Moussa, and S. Areibi, "The impact of arithmetic representation on implementing MLP-BP on FPGAs: A study," *IEEE Trans. Neural Netw.*, vol. 18, no. 1, pp. 240–252, Jan. 2007.

[26] A. Gomperts, A. Ukil, and F. Zurfluh, "Implementation of neural network on parameterized FPGA," in *Proc. AAAI Spring Symp., Embedded Reason.*, 2010, pp. 45–51.

[27] J. Yick, B. Mukherjee, and D. Ghosal, "Wireless sensor network survey," *Comput. Netw.*, vol. 52, no. 12, pp. 2292–2330, Aug. 2008.

[28] M. Sayed-Mouchaweh and E. Lughofer, Eds., *Learning in Non-Stationary Environments: Methods and Applications*. New York, NY, USA: Springer-Verlag, 2012.

[29] D. Urda, E. Cañete, J. L. Subirats, L. Franco, L. Llopis, and J. M. Jerez, "Energy-efficient reprogramming in WSN using constructive neural networks," *Int. J. Innov. Comput., Inf. Control*, vol. 8, no. 11, pp. 7561–7578, 2012.
[30] E. Cañete, J. Chen, R. M. Luque, and B. Rubio, "NeuralSens: A neural network based framework to allow dynamic adaptation in wireless sensor and actor networks," *J. Netw. Comput. Appl.*, vol. 35, no. 1, pp. 382–393, 2012.
[31] S. Mahmoud, A. Lotfi, and C. Langensiepen, "Behavioural pattern identification and prediction in intelligent environments," *Appl. Soft Comput.*, vol. 13, no. 4, pp. 1813–1822, Apr. 2013.
[32] M. A. Rassam, A. Zainal, and M. A. Maarof, "An adaptive and efficient dimension reduction model for multivariate wireless sensor networks applications," *Appl. Soft Comput.*, vol. 13, no. 4, pp. 1978–1996, Apr. 2013.
[33] D. M. Hawkins, "The problem of overfitting," *J. Chem. Inf. Comput. Sci.*, vol. 44, no. 1, pp. 1–12, 2004.
[34] S. Kilts, *Advanced FPGA Design: Architecture, Implementation, and Optimization*. New York, NY, USA: Wiley, 2007.
[35] U. Meyer-Baese, *Digital Signal Processing With Field Programmable Gate Arrays*, 3rd ed. New York, NJ, USA: Springer-Verlag, 2007.
[36] *Virtex-6 FPGA DSP48E1 Slice User Guide (v1.3)*, Xilinx Inc., San Jose, CA, USA, 2011.
[37] C.-J. Lin and C.-Y. Lee, "Implementation of a neuro-fuzzy network with on-chip learning and its applications," *Expert Syst. Appl.*, vol. 38, no. 1, pp. 673–681, Jan. 2011.
[38] A. Gomperts, A. Ukil, and F. Zurfluh, "Development and implementation of parameterized FPGA-based general purpose neural networks for online applications," *IEEE Trans. Ind. Informat.*, vol. 7, no. 1, pp. 78–89, Feb. 2011.
[39] M. Moussa, S. Areibi, and K. Nichols, "On the arithmetic precision for implementing back-propagation networks on FPGA: A case study," in *FPGA Implementations of Neural Networks*, A. R. Omondi and J. C. Rajapakse, Eds. New York, NY, USA: Springer-Verlag, 2006.
[40] P. P. Chu, *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*. New York, NY, USA: Wiley, 2006.
[41] J. Oxer and H. Blemings, *Practical Arduino: Cool Projects for Open Source Hardware*. Berkeley, CA, USA: Apress, 2009.
[42] Atmel. *Datasheet Atmel SAM3X8E ARM Cortex-M3 CPU*. [Online]. Available: http://www.atmel.com/Images/doc11057.pdf, accessed May 7, 2014.
[43] University of California, Irvine. *Machine Learning Repository*. [Online]. Available: http://archive.ics.uci.edu/ml/, accessed May 7, 2014.
[44] G. An, "The effects of adding noise during backpropagation training on a generalization performance," *Neural Comput.*, vol. 8, no. 3, pp. 643–674, Apr. 1996.

Francisco Ortega-Zamorano was born in 1980. He received the M.Sc. and Ph.D. degrees in computer science from the Universidad de Málaga, Málaga, Spain, in 2013 and 2015, respectively.

He joined the Department of Computer Languages and Computer Science, Universidad de Málaga, in 2011, where he is currently a Post-Doctoral Researcher. His current research interests include field-programmable gate array hardware implementation, neurocomputational real-time systems, smart sensor neworks, and deep learning.

José M. Jerez received the Ph.D. degree in computer science from the Universidad de Málaga, Málaga, Spain, in 2003.

He has participated in more than 15 research projects funded by international and national boards. He is currently an Associate Professor with the Universidad de Málaga. In particular, he is developing prediction software for biomedical problems using artificial intelligence techniques in collaboration with Málaga University hospital. His current research interests include computational intelligence, image analysis, and bioinformatics.

Dr. Jerez belongs to several reviewing scientific committees.

Daniel Urda Muñoz was born in Málaga, Spain, in 1982. He received the Degree in computer sciences, the master's degree in software engineering and artificial intelligence, and the Ph.D. degree from the Universidad de Málaga, Málaga, in 2008, 2010, and 2015, respectively. His Ph.D. thesis focused on developing predictive models based on genetic profiles to identify robust genetic signatures with high prediction capabilities for applications of personalized medicine.

He is currently a Marie Curie Post-Doctoral Researcher with Pharmatics Ltd., Edinburgh, U.K., where he is involved in machine learning for personalized medicine.

Rafael M. Luque-Baena received the M.S. and Ph.D. degrees in computer engineering from the Universidad de Málaga, Málaga, Spain, in 2007 and 2012, respectively.

He moved to Mérida in 2013, and is currently a Lecturer with the Department of Computer Engineering, Centro Universitario de Mérida, University of Extremadura, Badajoz, Spain. He also keeps pursuing research activities in collaboration with other universities. His current research interests include visual surveillance, image/video processing, neural networks, and pattern recognition.

Leonardo Franco (M'06–SM'13) received the M.Sc. and Ph.D. degrees from the National University of Córdoba, Córdoba, Argentina, in 1995 and 2000, respectively.

He was then a Post-Doctoral Researcher with the International School for Advanced Studies, Trieste, Italy, and Oxford University, Oxford, U.K. Since 2005, he has been with the Department of Computer Science, Universidad de Málaga, Málaga, Spain, where he is currently an Associate Professor. He has authored over 60 publications in journals and international conferences proceedings. His current research interests include neural networks, computational intelligence, biomedical applications, and computational neuroscience.