Integrated Computer-Aided Engineering 24 (2017) 171–185
DOI 10.3233/ICA-170538
IOS Press

Layer multiplexing FPGA implementation for deep back-propagation learning

Francisco Ortega-Zamorano[a,b,*], José M. Jerez[a], Iván Gómez[a] and Leonardo Franco[a]
[a]*Department of Computer Languages and Computer Science, University of Málaga, Málaga, Spain*
[b]*School of Mathematics and Computer Science, University of Yachay Tech, San Miguel de Urcuquí, Ecuador*

Abstract. Training of large scale neural networks, like those used nowadays in Deep Learning schemes, requires long computational times or the use of high performance computation solutions like those based on cluster computation, GPU boards, etc. As a possible alternative, in this work the Back-Propagation learning algorithm is implemented in an FPGA board using a multiplexing layer scheme, in which a single layer of neurons is physically implemented in parallel but can be reused any number of times in order to simulate multi-layer architectures. An on-chip implementation of the algorithm is carried out using a training/validation scheme in order to avoid overfitting effects. The hardware implementation is tested on several configurations, permitting to simulate architectures comprising up to 127 hidden layers with a maximum number of neurons in each layer of 60 neurons. We confirmed the correct implementation of the algorithm and compared the computational times against C and Matlab code executed in a multicore supercomputer, observing a clear advantage of the proposed FPGA scheme. The layer multiplexing scheme used provides a simple and flexible approach in comparison to standard implementations of the Back-Propagation algorithm representing an important step towards the FPGA implementation of deep neural networks, one of the most novel and successful existing models for prediction problems.

Keywords: Hardware implementation, FPGA, supervised learning, deep neural networks, layer multiplexing

1. Introduction

Artificial Neural Networks (ANN) [12] are mathematical models inspired in the functioning of the brain that have been successfully applied to clustering and classification problems in several domains. The Back-Propagation algorithm (BP) introduced by Werbos in 1974 [46] and popularized through the work of Rumelhart et al. [38] is the most used learning procedure for training feed-forward neural networks (FFNN) architectures for its application to classification and regression problems. It is a gradient descent based method that minimizes the error between targets and network outputs, computing the derivatives of the error in an efficient way [25,36]. As a gradient descent algorithm the search for a solution can get stuck in local minima but in practice the algorithm is quite efficient, and as such it has been applied to a wide range of areas like pattern recognition [23], medical diagnosis [37], stock market prediction [35], etc.

Even if with the actual computational power it is possible to train neural networks models relatively fast, using large architectures and/or large patterns data sets may require the use of parallel strategies to speed up the training process. In particular, a recent popularized model known as Deep Learning and usually applied to large training data sets, relies in a training process that may take several days or even weeks to be completed [5,17]. In this sense alternatives based on cluster computing, GPUs and FPGAs are sensible strategies, each of them having their benefits and drawbacks [10,29,43,44]. In particular, Field Programmable Gate Arrays (FPGA) [18] are reprogrammable silicon chips, using prebuilt logic blocks and programmable routing resources that can be configured to implement

*Corresponding author: Francisco Ortega-Zamorano, Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Campus de Teatinos S/N, 29071, Málaga, Spain. E-mail: fortega@lcc.uma.es.

custom hardware functionality. Neuro-inspired models of computations have a very large degree of parallel processing of the information, and as such one the main advantages of FPGA over previously mentioned alternatives (cluster computing and GPUs) for implementing them is the fact of their intrinsic parallelism. On the other hand, programming FPGA is relatively more complex than the other models, and this fact might explain that they have not been much utilized yet for Deep Learning.

Several studies have analyzed the implementation of neural networks models in FPGAs [19–21,24,32], applying one of the two existing alternatives for their implementation: *off* and *on* chip. In off-chip learning implementations [13,30] the training of the neural network model is performed externally usually in a personal computer (PC) to which the FPGA is attached, and only the synaptic weights are transmitted to the FPGA that acts as a hardware accelerator. On the other hand on-chip learning implementations includes both training and execution phases of the algorithm [6,32,41] permitting the whole process to be carried out in the FPGA board independently of an external device. Existing specific implementations of the artificial neural network Back-Propagation algorithm in FPGA boards include the works of [9,29,31,39]. In all of these works, the neural network architecture is previously prefixed by the designer, as the number of neurons and hidden layers is limited by the FPGA resources available. Although recent advances on the computational power of these boards have permitted an increase in the size of the architectures, the number of layers that can be implemented is still limited, and as said above also this number should be prefixed before its application.

For the previously mentioned reason, in this work a layer multiplexing scheme for the on-chip implementation of BP algorithm in a VIRTEX-5 XC5VLX110T FPGA board is introduced. This scheme consists in implementing physically a single layer of neurons that can be reused any number of times in order to simulate architectures with any number of hidden layers [13]. The number of hidden layers that can be used in a neural architecture is only limited by the temporal constraint related to the execution time and to a maximum of 127 because of the memory resource design, although it has been observed previously [3,8] and confirmed in the present work that the performance of the BP algorithm is drastically reduced when the number of layers is too large. In this respect, a new promising field named Deep Learning has attracted the attention of several researchers and companies in recent years, due to the great success of deep neural networks architectures in several pattern recognition contests [14,22,40]. Deep learning schemes requires the use of additional strategies or modifications to the standard BP algorithm in order to be applied successfully [45], but so far all existing alternative requires heavy computational resources. The aim of this work is to build a simple and flexible implementation of the BP algorithm that may permit to simulate deep Back-Propagation neural networks efficiently, contributing to their study and application, and also opening new strategies towards the simulation of deep learning neural networks.

The organization of the present work is as follows: next section includes relevant implementation details about the BP algorithm. The FPGA implementation is described in Section 3, that contains the technical details of the implementation. The work continues with a result section where the implementation is tested and characterized, and finishes with the discussion and conclusions.

2. The Back-Propagation algorithm

The Back-Propagation 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 Back-Propagation algorithm, as they are important in order to understand the current work.

Let's 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 the previous equation, we have introduced h as the synaptic potential of a neuron. The activation function used, g, is the logistic function given by the following equation:

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

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. By 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:

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

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 from the last layer into the inner ones, that are computed according to Eqs (5) and (6).

The delta values for the neurons of the last of the N hidden layers are computed as:

$$\delta_j^N = (S_j^N)'[z_j - S_j^N] \tag{5}$$

The delta values for the rest of the hidden layer neurons are computed according to:

$$\delta_j^l = (S_j^l)'\sum w_{ij}\delta_i^{l+1}. \tag{6}$$

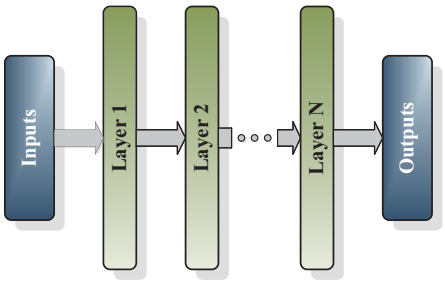
Training and validation processes

The training procedure is executed a certain number of times (epochs) using the training patterns. In one epoch, all training patterns are presented once in random ordering, adjusting the synaptic weights in an online manner. A well known and severe problem affecting all predictive algorithms is the problem of overfitting, caused by an overspecialization of the learning procedure on the training set of patterns [11]. In order to alleviate this effect, one straightforward alternative is to split the set of available training patterns in training, validation and test sets. From these sets by the application of Eq. (3) training, validation and generalization error measures are obtained, measures that will be denoted as E_{tr}, E_{val} and E_{gen} respectively. The training set will then be used to adjust the synaptic weights according to Eq. (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. The generalization ability (*Gen*) defined as $Gen = 1 - E_{gen}$ is a standard measure for the prediction accuracy of an algorithm, obtaining its optimal value for $Gen = 1$ when $E_{gen} = 0$.

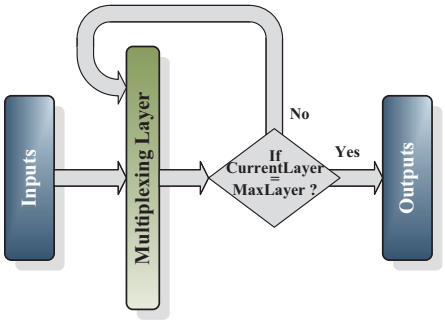
3. FPGA layer multiplexing scheme implementation of the BP algorithm

The hardware implementation of the Back-Propagation algorithm is divided in 3 different processes: the computation of the output of the neurons (S values), the calculation of the deltas of each neuron (δ), and the synaptic weight updating procedure. Given the logic of the Back-Propagation algorithm, in which the S values are obtained in a forward manner (from the input towards the output) while the deltas are computed backwards, and that finally the weights updating is executed with the values previously obtained, the three processes are sequentially implemented. It is possible to perform the weight updating phase at the same time that the deltas are computed but we have preferred to separate all three processes to obtain a clearer design.

The S values of every layer are obtained as a function of the S values of the previous layer neurons except for those from the first hidden layer which processes the information of the current input pattern. On the contrary, the δ values are computed backwardly, i.e., the δ values associated to a neuron belonging to a hidden layer are computed as a function of the δ values of the a deeper hidden layer, except for the last hidden layer which computes its δ values as a function of the error committed on the current input pattern (cf. Eqs 5–6). The updating process is carried out with the S and δ values of every layer, so it is necessary to store these values when they are computed to be used for the system when they are required. Thus, the structure of the Back-Propagation algorithm allows the whole process



(a) Standard Feed-forward neural network



(b) Layer multiplexing scheme

Fig. 1. (a) Standard feed-forward neural network architecture. (b) Layer multiplexing scheme for the simulation of deep feed-forward neural network architectures.

to be implemented using a layer multiplexing scheme but noting that forward and backward phases should be considered separately, as S and δ values cannot be computed in a single forward phase.

The deep design of the Back-Propagation algorithm is based on a layer multiplexing scheme in which only one layer is physically implemented being reused $3 \times N$ times in order to simulate a whole neural network architecture containing N hidden layers. Figure 1(a) shows the standard design of a feed-forward neural network architecture where it can be observed how the flow of information goes from the input towards the output, in a way that the input of a layer of neurons is the output of the previous layer. In a layer multiplexing scheme, as shown in Fig. 1(b) the same whole process is carried out but by reusing the structure of the single implemented layer.

The implementation of the layer multiplexing scheme requires a precise control of which layer is simulated in every moment, and for this reason a register called "*CurrentLayer*" is used. For each pattern, the process starts with the forward phase in which the output of the neurons are computed in response to the input pattern. This first phase starts by introducing an input pattern in the single multiplexing layer and by setting the variable "*CurrentLayer*" set to 1. Then the neurons' outputs are computed, stored in the distributed RAM memory and transmitted back to the input to calculate the following layer outputs, and thus the variable "*CurrentLayer*" is increased. The same process is repeated sequentially until the "*CurrentLayer*" value is equal to the maximum number of layers, previously defined by the user and stored in the "*MaxLayer*" register. When the last layer is reached the neurons output is computed together with the error committed in the pattern target estimation and these error values are stored in a register for its use in the second phase. The second phase involves the backward computation of the delta values, and the first computation involves the calculation of the delta values of the last layer. Once these values are obtained, they are backwardly transmitted to the previous layer in order to compute the delta values for these set of neurons, according to Eq. (6). With these delta values a recurrent process is used to obtain the delta values of the rest of the layers until the input layer values are obtained ("*CurrentLayer* $= 1$"). At this point the third phase is carried out in order to update the synaptic weights, and finishing one pattern iteration of the process.

In the following subsections details of the hardware implementation are given.

Hardware implementation design

Figure 2 shows the general structure of the hardware implementation of the Back-Propagation algorithm. The whole structure has been divided in two main modules in order to separate the communication protocol and the memory resources of the FPGA (external block) from the module where the implementation of the algorithm is carried out composed by the Architecture and Control blocks.

The External block logic depends specifically on the type of communication protocol chosen for receiving the pattern data set and transmitting the synaptic weights of the resultant model. The implementation of the present algorithm in different applications and boards require the use of a different external block, so instead of giving specific details about it, we have preferred to describe its functionality that it might be more helpful for future implementations.

In our case, the communication between the PC and the FPGA was handled using a serial communication protocol through the RS-232 port of the board in order to manage the exchange of information. The reason for this choice is that it can be implemented in VHDL and

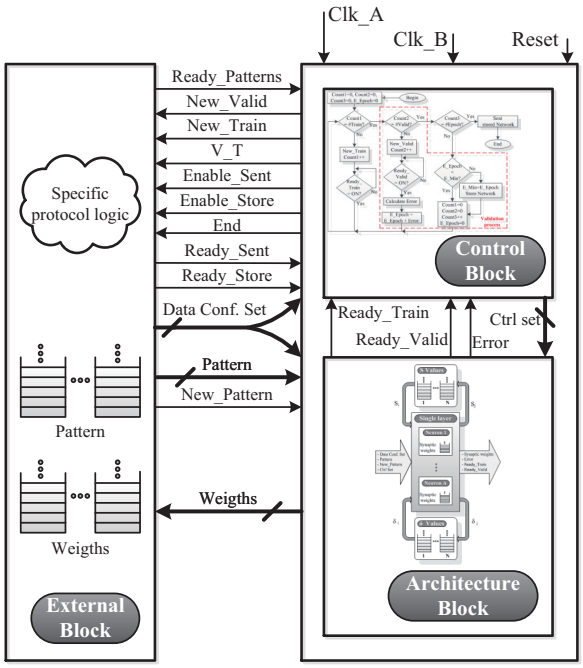


Fig. 2. Information flow exchange between the external module in charge of the communication protocols and memory resources management, and the module composed by the control and architecture blocks.

ported to other architectures quite easily in comparison to other possibilities.

The functionality of the external block is separated in two different processes: The first one was in charge of storing and managing the input training data set in order to present a different pattern every time that the control block requires it; while the second process was used for storing the synaptic weights once the learning process finishes (see Fig. 2). The hardware implementation of these two processes involves taking into account a series of signals between the blocks that are described in the appendix.

The internal module computes the neural model output and modifies the synaptic weights according to the training data presented to the network architecture. This module carries out the whole process of the algorithm and is composed by the control and architecture blocks described below in Sections 3.1 and 3.2 respectively.

3.1. 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

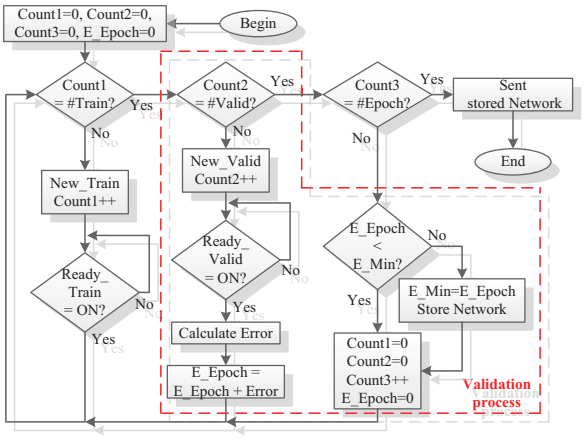


Fig. 3. Flow diagram for the operation of the control block for the BP algorithm. Two processes (network training and validation) are part of this block (see the text for more details).

pattern blocks. The structure of this block is organized around two main processes: i) Network Training: the main function of the control block is to manage 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 validation (#Val) patterns set for the training procedure. ii) Validation: a secondary process of the control block regards the use of a validation set for monitoring the training error, in order to control 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 while the training of the network proceeds. At the end of the training phase, this module retrieves the set of weights that had led to the minimum validation error. The implementation of the whole validation process in the FPGA is detailed in Section 2.

When the computations start, the set of training patterns are loaded into the external block that sends a signal to the control block in order to start the execution of the algorithm. Figure 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 and validation patterns, and number of epochs are initialized to zero. While the number of training patterns for a given epoch is lower than the set value of training patterns (#Train), the training procedure keeps 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

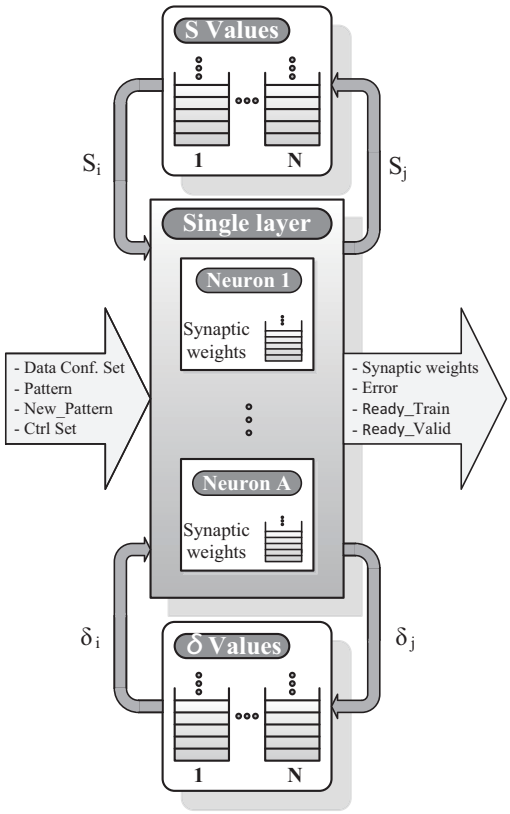


Fig. 4. Schematic representation of the layer multiplexing procedure used for the implementation of the BP algorithm.

network, sending back a signal (Ready_Train) to the control block when the training of this pattern finishes, increasing the trained pattern counter *Count1*. When the value of this counter gets equal to the total number of training patterns, then the validation process start. The previous steps belong to a loop so they are repeated until the maximum number of epochs (#Epoch) is reached.

3.2. Architecture block

The Fig. 4 shows a scheme of the architecture block that performs the layer multiplexing procedure by physically implementing a single layer of neurons. This single layer is composed of A neurons blocks in parallel implemented in order to compute the neuron's output (S) and the δ values, that will later be used for the update of the synaptic weights. The value of A (limited by the board resources) will be the maximum number of neurons for any hidden layer. The neuron blocks manage their own synaptic weights independently of the rest of the architecture, and thus they require a RAM block attached to them. Details of the neuron blocks are described below in Section 3.2. The architecture block also includes memory blocks to store the S and δ values computed for every layer and also for the different input and output signals that are described below.

The input signals are the pattern to be learned, the signal that indicates a new pattern is introduced (New_pattern), the configuration and control data sets, including also the S and δ values. The configuration data set includes the parameters set by the user to specify the neural network architecture, including the number of hidden layers, the number of neurons in each of these layers, learning parameters, etc. The control data set are signals that the control block needs for managing the process of the algorithm to activate the right procedure in every moment. The output signals comprise the output (S) and the δ values for every layer, the training error of the current pattern, and the ready signals for the validation and training processes which are integrated in the control data set.

The maximum number of layers has been determined by the size of the bus used to address the multiplexing layer scheme. In order to have a compromise between the number of layers and used resources, we have employed 7 bits in this bus, so the maximum number of layers is $2^7 - 1 = 127$. Also, the maximum number of layers is delimited for the resources needed to store the synaptic weights, according to the next equation:

$$N_i * N_1 + N_1 * N_2 + \ldots + N_{L-1} * N_{L-2} < \text{Available RAM} \quad (7)$$

where N_i is the number of inputs, N_1 is the number of neurons in the first layer, N_2 those corresponding to the second layer and so on.

Neuron block

Each of the A neuron blocks manages its own synaptic weights, computing the S and δ values involved in the Back-Propagation pattern updating procedure. The word length to be chosen for representing the synaptic weights would depend on the available resources, taking into account that obtaining a higher accuracy requires a larger representation, which will imply an increase in the number of LUTs per neuron (consequently a reduced number of available neurons) and a decrease in the maximum operation frequency of the board. A synaptic weight is represented by a bit array

Table 1
Board resources needed for the implementation of a neuron block

	Registers	LUTs	DSPs	RAM block
Neuron	428	1007	1	$n = \frac{Avail.RAM}{\#neurons}$

with integer and fractional parts of length N_1 and N_2. In our case the selected representation was $N_1 = 16$ and $N_2 = 16$, a representation that permits a relatively high accuracy as the errors generated in a layer are propagated to further ones. Board resources needed for the implementation of a neuron using the chosen representaion are shown in Table 1. The value for RAM block shown in the last column of the table and indicated by n is equal to the available RAM (a value that depends on the FPGA board specifications) divided by the maximum number of neurons allowed in the single implemented layer. For the implementation described in this work $n = 2 = int(\frac{148}{60})$ as the available RAM is 148 blocks as indicated in Table 2, while the maximum number of neurons is 60 (see Section 5).

The implementation of the neuron blocks has been performed by dividing all the involved processes in five main sub-blocks (*Multiplier*, *Weight* S, δ and *Update* blocks). We describe below the detailed implementation of each one of these sub-blocks.

3.2.1. Multiplier block

The multiplier block computes the multiplication operations involved in the Back-Propagation algorithm, mainly between neuron activations and synaptic weights values (see Eqs (1)–(6)). An efficient implementation of this operation is crucial in order to optimize the board resources. A time-division multiplexing scheme has been developed for an efficient use of the resources, using only one multiplier per neuron and thus performing sequentially the computation of several products [34]. Multipliers can be implemented by shifters and adders, following the approach presented in [4] or by available specific DSP cores in the FPGA. The DSP based strategy has been selected because the system frequency in the FPGA can be up to four times faster. The DSP uses a frequency two times larger than the used by the neuron block, so that a product operation could be completed in one operation cycle of the FPGA.

Figure 5 shows the multiplier block. A "*state*" signal will indicate which of the processes (S, δ or updating) is being executed at this moment, and two multiplexers will select the correct values to a DSP multiplier, which synchronized with a clock signal will send the multiplication result to the rest of the blocks.

3.2.2. Weight block

Each of the weight blocks attached to every neuron is in charge of writing and reading the synaptic weights using a single distributed RAM memory module. The memory module has three inputs (W/R, *Addr*, and *Value*) managed by two multiplexers controlled by the signal "*state*". The first input (W/R) decides which action to carry out (write or read), while the second input (Addr) specifies the memory address, and the third inputs is the value to store in case of a writing operation. The first multiplexer (the bottom one in the figure) allows writing ($W/R = 1$) only when the "*state*" signal is Update, otherwise only reading ($W/R = 0$) is possible. The second multiplexer (the one on top) selects the address which will be used for the read/write operation according to the "*state*" signal (S, δ, U). The memory module also uses a frequency two times larger than the used by the neuron block in order to complete the operation in one cycle of the FPGA.

3.2.3. S block

The S block (see the right part of Fig. 5) computes the output of a neuron as a function of the outputs of the previous layer which is introduced by the signal "*Vec_S*". The FSM (Finite State Machine) of the block manages the steps required by the process (see the top of the figure of the S-block). When the S process starts, the control block activates the signal "*Enable_S*" and then the FSM change its stand-by state ($A = 0$) to the $A = 1$ state.

In State $A = 1$ the block is in charge of computing the sum of the product of synaptic weights and input values (Eq. (1)). To perform this operation, two synchronous counters (indicated by #1 and #2 in the figure) are used together with a set of logic elements for performing the summation of product values. The signal "*Index*" of the adder #2 selects the corresponding S_j value and the memory address ("*MemAddr_S*") of the synaptic weight w_{ij} ("*Weight*"). These selected values are sent in each cycle to the Multiplier Block using the signals ("*Inp1Mul_S*") and ("*Inp2Mul_S*") for the w_{ij} and S_j values respectively. The result of the multiplication is returned through the signal "*OutMul*" and the counter #1 computes the summation of the multiplications until the "*Index*" signal is equal to the number of neurons in the previous layer (*Index* = *NuexLayer*), in this moment the "h" value (synaptic potential) is computed and the FSM changes to $A = 2$ state.

When the FSM is in the $A = 2$ state, the S block computes every neuron's output applying the transfer

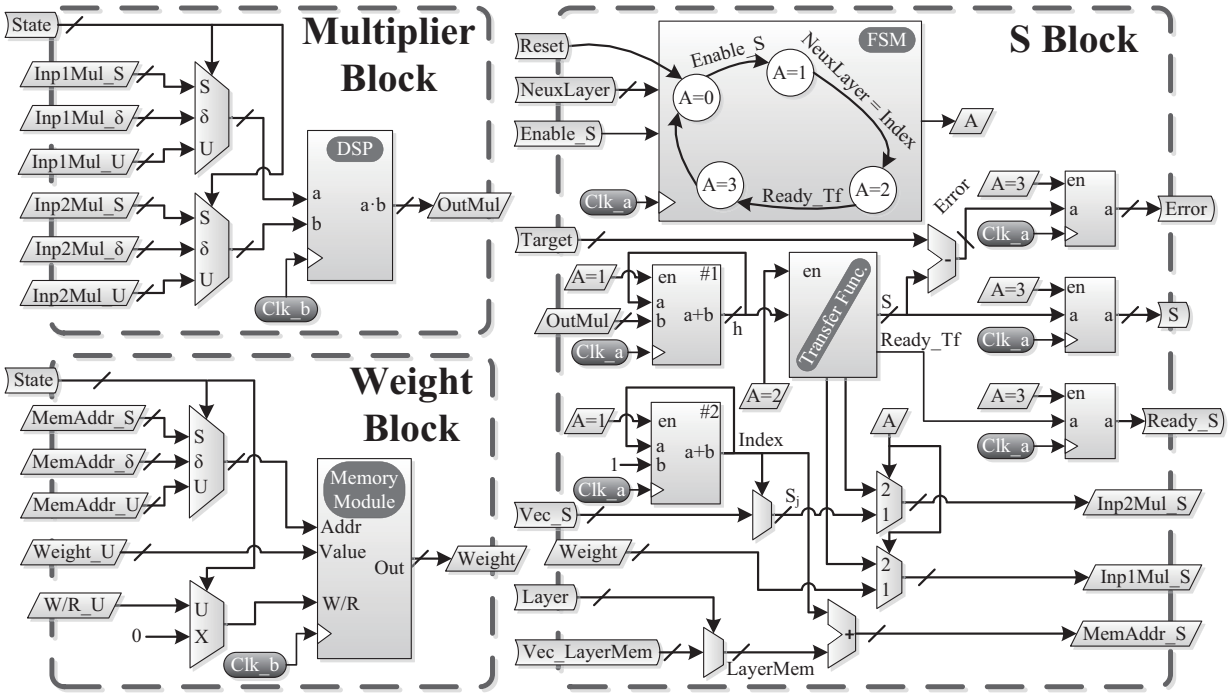


Fig. 5. Hardware details for the S-block, synaptic weights and multiplier blocks used for the implementation of a single neuron (See a note legend at the bottom of Fig. 6 for symbol reference).

function used (a sigmoid function in this case) to the synaptic potential previously obtained. This procedure, that for FPGA is not as straightforward as in a standard PC, is carried out using a lookup table containing tabulated values of the function plus a linear interpolation scheme. Further details of this procedure have been already explained in detail in Ref. [34]. Once the S value is obtained, the error for the estimation of the current pattern is computed, a ready signal ("*Ready_Tf*") is activated and the FSM change its state to $A = 3$. In the last state ($A = 3$), the output signals of the block ("*Error*", "*S*" and "*Ready_S*") are stored in three registers for their further use by other processes.

3.2.4. δ block

Figure 6 shows the δ block which is in charge of computing the δ values of the current layer. The δ process begins with the activation of signal "*Enable_δ*" and the FSM of the delta block switches from the initial inactive state ($A = 0$) to $A = 1$. In this state, the summation involved in the delta process Eqs (5) and (6) is computed. For this process it is necessary to design a crossed memory access since the δ block of a neuron requires the synaptic weight values of other neurons, and for this reason a decreasing counter (#1) is used. A second counter (#2) selects the memory address ("*MemAddr_δ*") using the "*Index*" signal, in a similar way as indicated for the case of the S block.

The corresponding δ values (δ_i^{l+1}) and the synaptic weight (w_{ij}) (see Eqs (5) and (6)) are chosen according to the value of the "*Index_δ*" signal of the counter #1 using the signals "*Vec_δ*" and "*Vec_weight*". The δ and synaptic weights values are sent to the multiplier block by the "*Inp1Mul_S*" and "*Inp2Mul_S*" signals respectively in order to compute the required multiplication for the sum of products which is carried out in the synchronous counter #3 by the "*OutMul*". This operation finishes when the "*Index*" signal is equal to the identifier of the neuron ("N"), and at this moment the FSM changes its state from $A = 1$ to $A = 2$.

When the FSM is in state $A = 2$, the δ block computes the derivative of the function ($S'_j = S_j \cdot (1 - S_j)$). The right value of S_j is selected using the N signal value from the "Vec_S" signals which contains the S value of all neurons. The computations on this state can be done in only one clock cycle, and thus in the next cycle the FSM switch its state to $A = 3$.

The state $A = 3$ calculates the multiplication between the result of the summation and the derivatives. This procedure is carried out by the multiplier block by the "*Inp1Mul_δ*" and "*Inp2Mul_δ*" signals and the multiplication is returned by the "*OutMul*$''$ signal. In

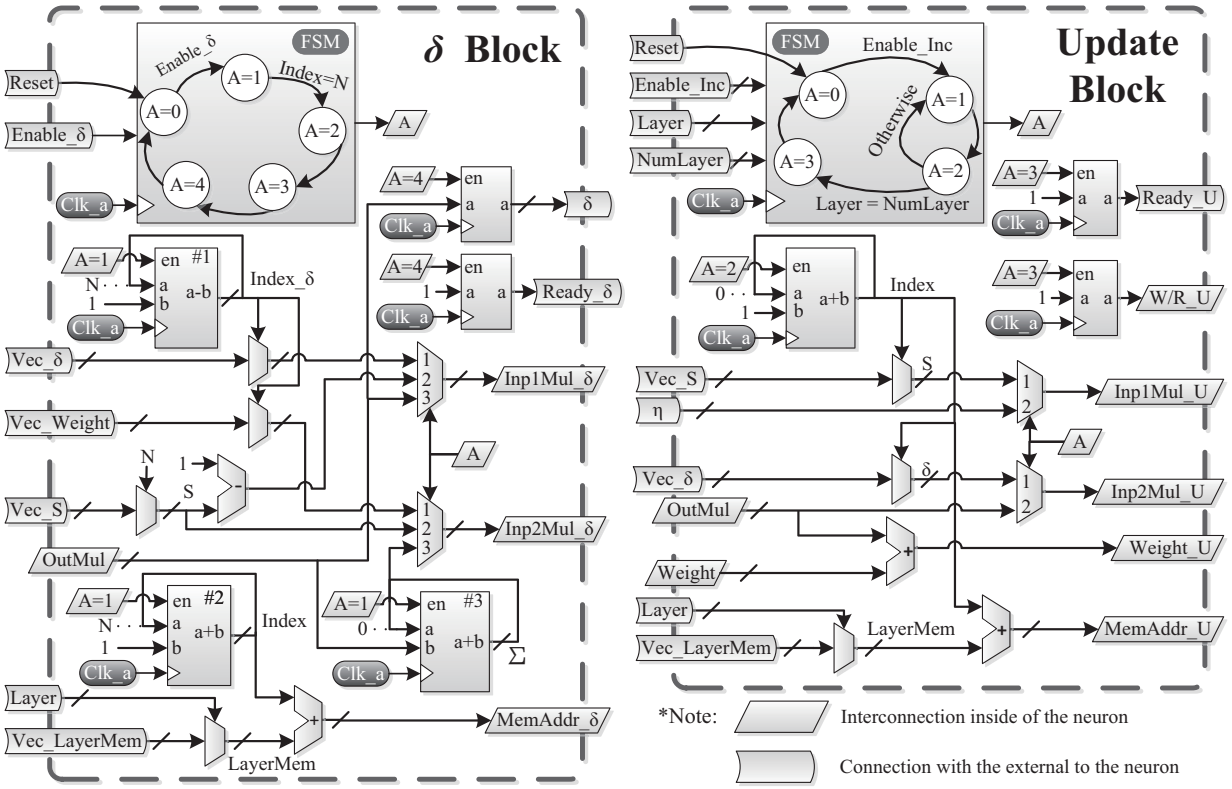


Fig. 6. Hardware details for the δ and Update blocks used for the implementation of a single neuron.

the last state ($A = 4$), the output signals of the block ("δ" and "*Ready_*δ") are stored in three registers for further usage.

3.2.5. *Update block*

The update block is in charge of modifying the synaptic weights for the whole architecture after an iteration of the algorithm has been carried out. Figure 6 shows the *Update* block in which it can be observed a 4-state FSM. The state $A = 0$ is the resting state that it is modified when the signal "*Enable_U*" is active. Inside a loop the states $A = 1$ and $A = 2$ of the FSM are used for reading the current synaptic weights and writing the updated ones for all simulated hidden layers (*Layer* $=$ *NumLayer*). The FSM switches to state $A = 3$ when the whole process is finished.

In state $A = 1$, the *Update* block carries out the request of every synaptic weight using a counter that generates the signal "*Index*" for reading the stored synaptic weights, in an analogous way as it has been explained before for the S Block. Also, in this state the multiplication between S_i and δ_i (see Eq. (4)) is performed. Both values are selected from their respective vectors "*Vec_S*" and "*Vec_*δ" using two multiplexers controlled for the "*Index*" signal.

When the FSM is in state $A = 2$, the multiplication between the learning rate (η) and the result of the multiplication obtained in the previous state is computed. This result is added to the current synaptic weight value to obtain the updated values. In order to store these values the Update block must activate a write/read signal ("W/R_U") to activate the input W/R of the memory module. In the last FSM state ($A = 3$), the Update block activate a signal indicating that the whole process is completed ("*Ready_U*").

4. Results

We present in this section results from the implementation of the Back-Propagation algorithm in a Xilinx Virtex-5 board. Table 2 shows some characteristics of the Virtex-5 XC5VLX110T FPGA, indicating its main logic resources. VHDL [2,4] (VHSIC Hardware Description Language) language was used for programming the FPGA, under the "Xilinx ISE Design Suite 12.4" environment using the "ISim M.81d"

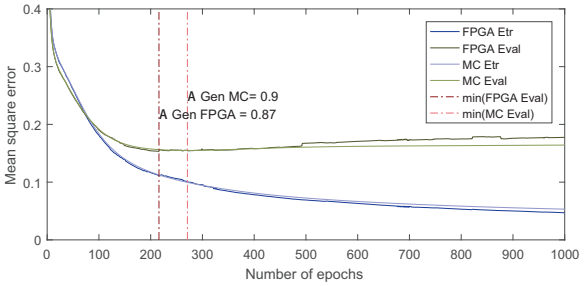
Table 2
Main specifications of the Xilinx Virtex-5 XC5VLX110T FPGA board

Device	Slice Registers	Slice LUTs	Bonded IOBs	Block RAM
Virtex-5 XC5VLX110T	69, 120	69, 120	34	148

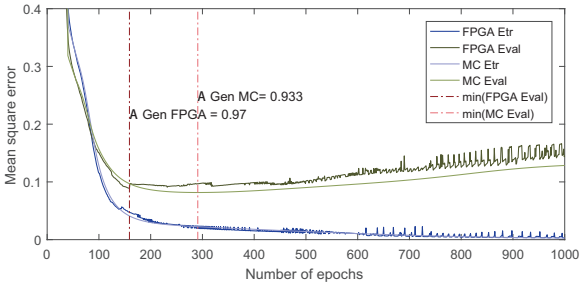
simulator. The operation system frequency was increased from the 100 MHZ board oscillator frequency to 200 MHZ through the use of a PLL, as the efficiency of the code allowed this configuration.

To verify the correct FPGA implementation of the model, several test cases were analyzed comparing the results with those obtained from C and Matlab implementations and with previously published results. In particular, to assess the advantages of using an FPGA board, we compare the results testing several network architectures under C and Matlab programming languages executed in the Picasso cluster that belongs to the Spanish Supercomputing Network.[1] The cluster is formed by a set of computation nodes unified behind a single Slurm queue system, consisting mainly of 7 HP DL980 nodes of 80 cores and 2 TB RAM computers, 32 HP SL230 nodes with 16 cores and 64 GB of RAM, 42 HP DL165 nodes with 24 cores and 96 GB of RAM, and 16 HP SL250 nodes with 2 GPUs each, totalling 63 TFLOP/s. Own generated code in C and Matlab languages were used for the comparison, noting that the C programming language is considered among the fastest that can be used in a PC [7,16] while Matlab is a language optimized for operations involving matrices and vectors useful for neural network implementations [27,42]. All the tests were carried out using a 50-20-30 splitting for the training, validation and generalization sets respectively, with a learning rate (η) value fixed to 0.2, and using data from the well-known Iris set [28]. The generalization set contains only patterns not used during the learning process, and it is used to test the prediction capacity of the algorithm, known as Generalization ability (Gen).

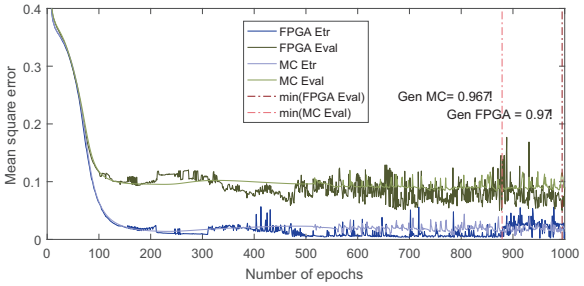
Figure 7 shows the evolution for the training (Etr) and validation (Eval) errors for the FPGA and the multicore (MC) cluster based implementation. The architectures used contained one hidden layer (a), two (b), and three (c), including five neurons in all hidden layers, and three neurons in the output that corresponds to the three classes of the Iris problem. In all three graphs, two vertical lines indicate the time at which the minimum of the validation error is obtained, point when the Generalization ability (Gen) is measured for both implementations (the obtained values are also indicated in the graph). It can be appreciated from the error curves that for the FPGA implementation case some larger oscillations appear, and this is due to rounding effects because of the size of the fixed point representation used. In terms of the level of prediction accuracy obtained these oscillations do not degrade it, and on the contrary in some cases even leads to larger values, as it has been observed previously in FPGA implementations [31,33], and in several works where it was concluded that certain level of noise might be beneficial for improving learning times, fault tolerance and prediction accuracy [1,15,26].



(a) One hidden layer



(b) Two hidden layers



(c) Three hidden layers

Fig. 7. Training (Etr) and validation (Eval) errors evolution for the two implementations (FPGA, MC) when the Iris data set is learned depending on the number of layer of the neural architecture with five neurons in each hidden layer.

[1] http://www.scbi.uma.es/.

Table 3
Generalization ability for the Iris data set for neural network architectures with different numbers of hidden layers for MC and FPGA implementations

Lay.	Type implementation		
	MC	FPGA	
		Layer multiplexing	Fixed layers
1	0.937 ± 0.058	0.939 ± 0.056	0.940 ± 0.057
2	0.951 ± 0.031	0.944 ± 0.036	0.947 ± 0.035
3	0.951 ± 0.029	0.949 ± 0.039	–
5	0.933 ± 0.100	0.937 ± 0.039	–
7	0.870 ± 0.201	0.884 ± 0.080	–
10	0.527 ± 0.294	0.599 ± 0.265	–
15	0.306 ± 0.055	0.312 ± 0.096	–
20	0.305 ± 0.046	0.310 ± 0.102	–
127	0.307 ± 0.049	0.309 ± 0.093	–

Table 3 shows the generalization ability obtained for several architectures with different numbers of hidden layers for the FPGA and MC implementations. The first column indicates the number of hidden layer present in the architecture, the second column shows the generalization obtained using the MC implementation (mean and standard deviation computed over 100 independent runs using C code), while third and fourth columns shows the results for two different FPGA implementations: the layer multiplexing scheme proposed in this work and the fixed layer scheme utilized in Ref. [31] (only available for architectures with one and two hidden layers). The number of neurons in each of the hidden layers was fixed to five and the number of epochs set to 1000. The results clearly show that for architectures with 15 or more hidden layers the generalization ability gets much reduced, reaching a random expected value for a problem with three classes.

From the results shown in Table 3 it can be seen that the obtained values for generalization are approximately similar for the three implementations considered, and that regarding the number of hidden layers present in the neural architectures the performance of the BP algorithm is relatively stable for architectures with up to 5 hidden neuron layers point from which the generalization accuracy starts to decrease to reach the level expected for random choices for a number of layers equal to 15.

Figure 8 shows the computation times (in μs and in logarithmic scale) and number of clock cycles ($\#cc$) involved in the three processes related to the operation of the BP algorithm for FPGA, MC-C, and MC-Matlab implementations: Output, Delta and Updating processes when learning a single pattern. The graph shows the results for one, ten and twenty hidden layers with five neurons per layer. The expressions shown on top of the figure for the number of clock cycles involved in the three phases are valid for any architecture when using the FPGA implementation, where the values of N_j should be replaced by the number of neurons in each of the layers.

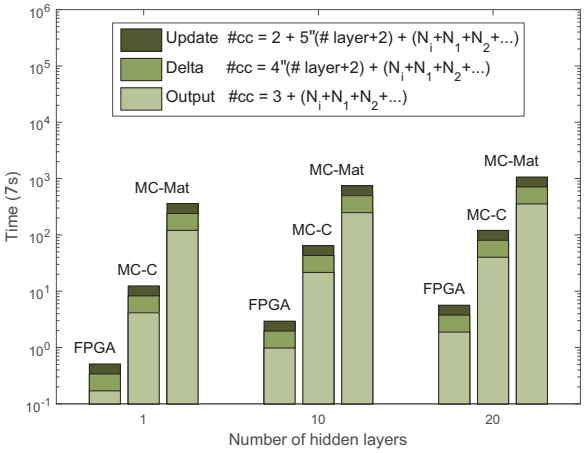
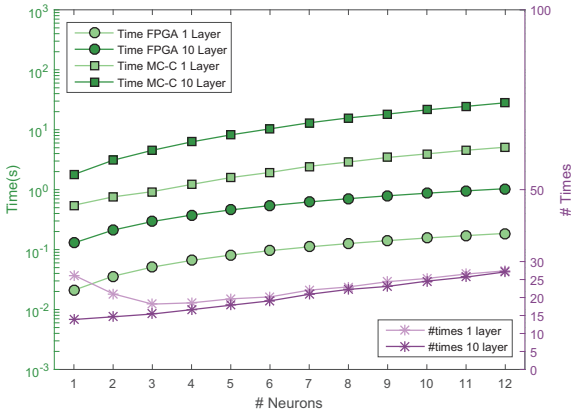


Fig. 8. Computation times and number of cycles involved in the three processes related to the operation of the BP algorithm for the FPGA, PC-C and PC-Matlab implementations: Output, Delta and Updating processes.

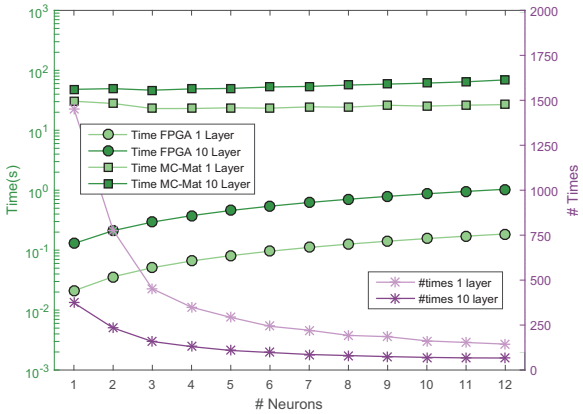
Figures 9(a) and (b) show execution times for the complete BP learning procedure (in seconds in a logarithmic scale for 1000 epochs) for the FPGA implementation and MC-C (Fig. 9(a)) and MC-Matlab (Fig. 9(b)) as a function of the number of neurons for one and ten hidden layers architectures (this number of neurons is fixed for all layers). In both graphs it is also shown the number of times that the FPGA implementation is faster in comparison to the MC one for C and Matlab code respectively (see the right y-axis scale). We also show in Fig. 10 the number of times that the FPGA implementation is faster than the MC-C (Fig. 10(a)) and MC-Matlab (Fig. 10(b)) codes as a function of the number of layers in the deep architectures for different number of neurons in each of these layers.

In order to obtain a fair comparison between the FPGA and PC the computation time of the FPGA has been measured without taking account the communication time due to this time could change depending on the type of protocol used in the communication. So, the computation time is calculated from the input data are completely sent to the neural computation model is computed in the FPGA.

From the results shown in Figs 9(a) and 10(a), it can be seen that the number for times that the FPGA implementation is faster than the MC-C increases linearly with the number of neurons in each of the lay-



(a)



(b)

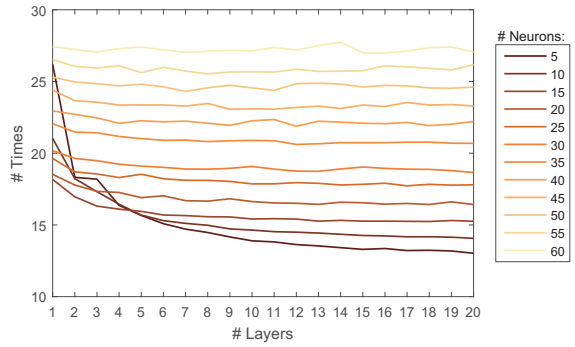
Fig. 9. Computational time and number of times that the FPGA implementation is faster than the MC-C (a) and MC-Matlab (b) as a function of the number of neurons in each layer for the case of one and ten hidden layers architectures (see text for details).

ers, reaching 27 times for the case of using 60 neurons in each layer, noting that these values are kept constant for different number of hidden layers. In relationship to the computational times between the FPGA and the MC-Matlab implementation the advantage of using the FPGA decreases as the number of neurons in each layer increases, and this effect can be explained because Matlab uses matrix-based computations that are more efficient for heavier computations, but noting that the number of times that the FPGA is faster than MC-Matlab converges asymptotically to 60 times approximately.

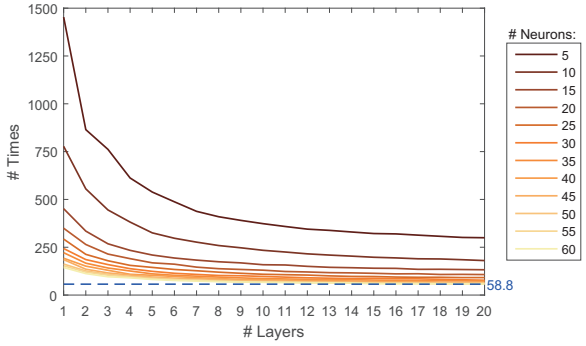
To test the correct implementation of the deep learning scheme of the BP algorithm in the FPGA board, we measured training, validation and test errors on a set of benchmark problems from the UCI database [28] frequently used in the literature. Table 4 shows the accuracy (generalization ability) values obtained for both implementations of the algorithm for eleven bench-

Table 4
Generalization ability for PC and FPGA implementations obtained for eleven benchmark data sets

Function	I	O	PC	FPGA
Diabetes	8	2	0.784 ± 0.028	0.793 ± 0.023
Cancer	9	2	0.958 ± 0.013	0.954 ± 0.011
Statlog	13	2	0.784 ± 0.023	0.776 ± 0.022
Climate	18	2	0.933 ± 0.012	0.944 ± 0.015
Ionosphere	34	2	0.874 ± 0.001	0.864 ± 0.001
HeartC	35	2	0.789 ± 0.035	0.801 ± 0.028
Iris	4	3	0.923 ± 0.019	0.926 ± 0.020
Bal.Sca.	4	3	0.869 ± 0.011	0.873 ± 0.012
Seeds	7	3	0.976 ± 0.017	0.965 ± 0.016
Wine	13	3	0.886 ± 0.022	0.880 ± 0.024
Glass	10	6	0.938 ± 0.025	0.914 ± 0.022
Average			0.8830 ± 0.0187	0.8808 ± 0.0176



(a)



(b)

Fig. 10. Number of times that the FPGA implementation is faster than the MC-C (graph a) and MC-Matlab (graph b) as a function of the number of layers in the deep architectures for different values for the number of neurons in each of these layers.

mark problems. The first three columns indicate the data set name, number of inputs and outputs respectively, while the last two columns shows the generalization ability obtained using neural network architectures with 5 neurons in the single hidden layer. This choice of number of neurons permits the comparison with published results [31]. 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. 100 independent runs were computed for each benchmark data set and the average and standard deviation of the obtained results are reported in the table. The results indicate a correct functioning of the algorithm, noting that the small observed differences can be related to the methodology of computation and to the different number representation used in the two analyzed cases.

5. Discussion and conclusions

We have successfully implemented the Back Propagation algorithm in an FPGA board using a novel layer multiplexing on-chip learning scheme that includes a validation procedure in order to prevent overfitting effects. The layer multiplexing scheme utilized permits to simulate a several hidden layer neural network with only implementing physically a single hidden layer of neurons. The main advantage of this approach is that very deep neural network architectures can be analyzed through a simple and flexible framework with very efficient resource utilization. A modular design has been utilized in a hardware implementation that incorporates strategies like multiplexing of the multipliers, optimized memory access and efficient data type representation, with the aim of producing a flexible and resource efficient tool for the study of multi-layer neural architectures.

In terms of computational times, the implementation has been tested and compared to multicore (MC) C and Matlab codes executed in a 97 nodes supercomputer. In comparison to the MC-C code, the number of times that the FPGA implementation is faster increases linearly as the number of neurons in each of the layers increases, while being almost constant for different number of hidden layers, reaching a value of 27 when 60 neurons are included in each of these hidden layers. The same comparison but for the case of FPGA and MC-Matlab implementations shows a different behaviour as the advantage of the FPGA decreases as the number of neurons in each layer increases. This advantage also has a slight decrease as the number of layers is increased, but in all analyzed cases being larger than 58.8 times.

The layer multiplexing scheme used permits in principle the simulation of networks with any number of hidden layers, but due to memory resource design the maximum number of layers in the current implementation is 127. Regarding the maximum number of neurons allowed in each of the hidden layer, hardware resources of the FPGA board used (VIRTEX-5 XC5VLX110T) pose a limit of 60 neurons. The results obtained confirm the degradation of the Back-Propagation algorithm for very deep architectures comprising 15 or more hidden layers, as almost a random behavior is obtained for deeper networks (see Table 3). Understanding and improving the training of deep architectures is a big present challenge, and we believe that the present work may contribute to their understanding as we have introduced a flexible tool for carrying this analysis that we plan to tackle in the near future. Further, it is worth noting that so far FPGAs have not been much applied to Deep Learning approaches, and we believe that high developing times are to blame. In this sense, we hope that this work can help other researchers on the applicaction of FPGA based approaches, as the intrinsic parallelism of these devices makes them a suitable technology for implementing neuroinspired models.

Acknowledgments

The authors acknowledge support from Junta de Andalucia through grant P10-TIC-5770, and from CICYT (Spain) through grants TIN2010-16556 and TIN2014-58516-C2-1-R (all including FEDER funds). The authors thankfully acknowledge the computer resources, technical expertise and assistance provided by the SCBI (Supercomputing and Bioinformatics) center of the University of Málaga, Spain.

Appendix

Detail of signals used between blocks (see Fig. 2 and related text)

- Ready_Patterns: Active (‘1’) when the pattern data set is ready to be learned.
- New_Valid: A pulse (one clock cycle active) when a validation pattern is required for the architecture.
- New_Train: A pulse when a training pattern is required for the architecture.
- V_T: A signal which defines the process (validation or training) that is being executed.
- Enable_Sent: A pulse when the stored synaptic weights in the external block must be send from the external block to the user.

- Enable_Store: A pulse when the synaptic weights of the neurons must be stored in the external block.
- End: Active when the process is finished.
- Ready_Sent: A pulse that is sent when the external block has finished sending the synaptic weights to the external device.
- Ready_Store: A pulse that is sent when the external block has finished storing the synaptic weights of every neuron.
- Data_Conf_Set: Variables of configuration of the model (number of layers, number of neuron for layer, number of patterns, etc).
- Pattern: Variable that introduces a new training or validation pattern when New_Train or New_Valid is active. respectively.
- New_Pattern: Pulse when a pattern is introduced.
- Weights: Synaptic weights of every neuron to be stored.

References

[1] G. An, The effects of adding noise during backpropagation training on a generalization performance, *Neural Computation* **8**(3) (Apr 1996), 643–674.

[2] P. Ashenden, *The Designer's Guide to VHDL, Volume 3, Third Edition (Systems on Silicon) (Systems on Silicon)*, 3 edition, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[3] Y. Bengio, P. Simard and P. Frasconi, Learning long-term dependencies with gradient descent is difficult, *IEEE Transactions on Neural Networks* **5**(2) (Mar 1994), 157–166.

[4] P.P. Chu, *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*, John Wiley & Sons, 2006.

[5] C. Clark and A.J. Storkey, Training deep convolutional neural networks to play go, in: *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of JMLR Proceedings, JMLR.org (2015), 1766–1774.

[6] A. Dinu, M. Cirstea and S. Cirstea, Direct neural-network hardware-implementation algorithm, *IEEE Transactions on Industrial Electronics* **57**(5) (May 2010), 1845–1848.

[7] B. Fulgham, The computer language benchmarks game, http://benchmarksgame.alioth.debian.org/.

[8] X. Glorot and Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, in: *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10), Society for Artificial Intelligence and Statistics*, (2010), 249–256.

[9] 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* **7**(1) (Feb 2011), 78–89.

[10] B. Guthier, S. Kopf, M. Wichtlhuber and W. Effelsberg, Parallel implementation of a real-time high dynamic range video system, *Integrated Computer-Aided Engineering* **21**(2) (2014), 189–202.

[11] D.M. Hawkins, The problem of overfitting, *Journal of Chemical Information and Computer Sciences* **44**(1) (2004), 1–12.

[12] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd edition, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998.

[13] S. Himavathi, D. Anitha and A. Muthuramalingam, Feedforward neural network implementation in fpga using layer multiplexing for effective resource utilization, *IEEE Transactions on Neural Networks* **18**(3) (May 2007), 880–888.

[14] G.E. Hinton, S. Osindero and Y.-W. Teh, A fast learning algorithm for deep belief nets, *Neural Comput* **18**(7) (July 2006), 1527–1554.

[15] L. Holmstrom and P. Koistinen, Using additive noise in backpropagation training, *Neural Networks, IEEE Transactions on* **3**(1) (Jan 1992), 24–38.

[16] R. Hundt, Loop recognition in c++/java/go/scala, in: *Proceedings of Scala Days 2011*, (2011).

[17] F. Iandola, K. Ashraf, M. Moskewicz and K. Keutzer, Firecaffe: Near-linear acceleration of deep neural network training on compute clusters, in: *Proceeedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[18] S. Kilts, *Advanced FPGA Design: Architecture, Implementation, and Optimization*, Wiley-IEEE Press, 2007.

[19] 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* **7**(1) (Feb 2014), 5–23.

[20] Q.N. Le and J.-W. Jeon, Neural-network-based low-speed-damping controller for stepper motor with an fpga, *IEEE Transactions on Industrial Electronics* **57**(9) (Sept 2010), 3167–3180.

[21] D. LeLy and P. Chow, High-performance reconfigurable hardware architecture for restricted boltzmann machines, *IEEE Transactions on Neural Networks* **21**(11) (Nov 2010), 1780–1792.

[22] Y. LeCun, Y. Bengio and G. Hinton, Deep learning, *Nature* **521**(7553) (May 2015), 436–444.

[23] J. Li, K. Ouazzane, H. Kazemian and M. Afzal, Neural network approaches for noisy language modeling, *Neural Networks and Learning Systems, IEEE Transactions on* **24**(11) (Nov 2013), 1773–1784.

[24] W. Mansour, R. Ayoubi, H. Ziade, R. Velazco and W.E. Falouh, An optimal implementation on fpga of a hopfield neural network, *Advances in Artificial Neural Systems* **2011** (2011), 1–9.

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

[26] A. Murray and P. Edwards, Synaptic weight noise during multilayer perceptron training: Fault tolerance and training improvements, *Neural Networks, IEEE Transactions on* **4**(4) (Jul 1993), 722–725.

[27] J. Nazari and O.K. Ersoy, Implementation of back-propagation neural networks with matlab, Technical report, Purdue University School of Electrical Engineering (01 1992).

[28] U. of California Irvine, Machine learning repository, http://archive.ics.uci.edu/ml/.

[29] A. Omondi and J. Rajapakse, *FPGA Implementations of Neural Networks*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[30] T. Orlowska-Kowalska and M. Kaminski, Fpga implementation of the multilayer neural network for the speed estimation of the two-mass drive system, *IEEE Transactions on Indus-*

trial Informatics **7**(3) (Aug 2011), 436–445.

[31] F. Ortega-Zamorano, J.M. Jerez, D. Urda Muñoz, R.M. Luque-Baena and L. Franco, Efficient implementation of the backpropagation algorithm in fpgas and microcontrollers, *IEEE Transactions on Neural Networks and Learning Systems* **27**(9) (2016), 1840–1850.

[32] F. Ortega-Zamorano, J.M Jerez and L. Franco, Fpga implementation of the c-mantec neural network constructive algorithm, *IEEE Transactions on Industrial Informatics* **10**(2) (May 2014), 1154–1161.

[33] F. Ortega-Zamorano, J.M Jerez, G. Juarez and L. Franco, Fpga implementation comparison between c-mantec and backpropagation neural network algorithms, in: *Advances in Computational Intelligence*, volume 9095, Springer International Publishing, (2015), 197–208.

[34] F. Ortega-Zamorano, J.M Jerez, G. Juarez, J.O Pérez and L. Franco, High precision fpga implementation of neural network activation functions, in: *Intelligent Embedded Systems (IES), 2014 IEEE Symposium on*, (Dec 2014), 55–60.

[35] T. Pinto, Z. Vale, T.M. Sousa, I. Praça, G. Santos and H. Morais, Adaptive learning in agents behaviour: A framework for electricity markets simulation, *Integr Comput-Aided Eng* **21**(4) (Oct 2014), 399–415.

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

[37] M. Rizzi, M. D'Aloia and B. Castagnolo, A supervised method for microcalcification cluster diagnosis, *Integr Comput-Aided Eng* **20**(2) (Apr 2013), 157–167.

[38] D. Rumelhart, G. Hinton and R. Williams, Learning representations by back-propagating errors, *Nature* **323**(6088) (1986), 533–536.

[39] A. Savich, M. Moussa and S. Areibi, The impact of arithmetic representation on implementing mlp-bp on fpgas: A study, *IEEE Transactions on Neural Networks* **18**(1) (Jan 2007), 240–252.

[40] J. Schmidhuber, Deep learning in neural networks: An overview, *Neural Networks* **61** (2015), 85–117.

[41] J. Shawash and D. Selviah, Real-time nonlinear parameter estimation using the levenberg-marquardt algorithm on field programmable gate arrays, *IEEE Transactions on Industrial Electronics* **60**(1) (Jan 2013), 170–176.

[42] S. Shrestha, Z. Bochenek and C. Smith, Artificial neural network (ann) beyond cots remote sensing packages: Implementation of extreme learning machine (elm) in matlab, in: *Geoscience and Remote Sensing Symposium (IGARSS), 2012 IEEE International*, (July 2012), 6301–6304.

[43] S. Sun, Z. Yan and J. Zambreno, Demonstrable differential power analysis attacks on real-world fpga-based embedded systems, *Integr Comput-Aided Eng* **16**(2) (Apr 2009), 119–130.

[44] N. Sundararajan and P. Saratchandran, *Parallel Architectures for Artificial Neural Networks: Paradigms and Implementations*, 1st edition, IEEE Computer Society Press, Los Alamitos, CA, USA, 1998.

[45] H. Wen, W. Xie and J. Pei, A pre-radical basis function with deep back propagation neural network research, in: *Signal Processing (ICSP), 2014 12th International Conference on*, (Oct 2014), 1489–1494.

[46] P.J. Werbos, Beyond regression: New tools for prediction and analysis in the behavioral sciences, Ph.D. thesis, Harvard University, 1974.