Deep Neural Network Architecture Implementation on FPGAs Using a Layer Multiplexing Scheme

Francisco Ortega-Zamorano, José M. Jerez, Iván Gómez and Leonardo Franco

Abstract In recent years predictive models based on Deep Learning strategies have achieved enormous success in several domains including pattern recognition tasks, language translation, software design, etc. Deep learning uses a combination of techniques to achieve its prediction accuracy, but essentially all existing approaches are based on multi-layer neural networks with deep architectures, i.e., several layers of processing units containing a large number of neurons. As the simulation of large networks requires heavy computational power, GPUs and cluster based computation strategies have been successfully used. In this work, a layer multiplexing scheme is presented in order to permit the simulation of deep neural networks in FPGA boards. As a demonstration of the usefulness of the scheme deep architectures trained by the classical Back-Propagation algorithm are simulated on FPGA boards and compared to standard implementations, showing the advantages in computation speed of the proposed scheme.

Keywords Hardware implementation · FPGA · Supervised learning · Deep neural networks · Layer multiplexing

1 Introduction

Neural Networks models have been successfully applied to a wide range of domains in clustering and classification problems in the last three decades [1, 2]. In particular, regarding supervised problems included in the broad area of pattern recognition,

F. Ortega-Zamorano(✉) · J.M. Jerez · I. Gómez · L. Franco
Department of Computer Science, ETSI Informática, Universidad de Málaga, Malaga, Spain
e-mail: {fortega,jja,ivan,lfranco}@lcc.uma.es

F. Ortega-Zamorano
School of Mathematics and Computer Science, University of Yachay Tech,
San Miguel de Urcuquí, Ecuador

S. Omatu et al. (eds.), *DCAI, 13th International Conference*,
Advances in Intelligent Systems and Computing 474,
DOI: 10.1007/978-3-319-40162-1_9

most of the strategies have been based on the utilization of feed forward neural network architectures (FFNN) trained by versions of the well known Back-Propagation algorithm (BP) [3, 4]. One important issue at the time of the implementation of FFNN models is the choice of an adequate architecture [5], that essentially consists of deciding how many hidden layers and neurons to include. It has been observed that the performance of the BP algorithm decreases when a large number of hidden layers are used and so the standard strategy before the irruption of Deep Learning strategies [6] have been to use single hidden layer architectures. Deep Learning is a relatively new technique belonging to the artificial intelligence and machine learning areas that have achieved state-of-the-art results in several recent competitions [7]. There are several approaches for their implementation, as training is a complex process, but in all cases the new characteristic in relationship to previous FFNN approaches is the fact that large (deep and wide) neural network architectures are used. Just to give some numbers, a typical deep learning architecture might include from 5 to 15 hidden layer of neurons with a number of neurons in each hidden layer in the order of the several hundreds or thousands [8]. Training these large networks using standard BP is computationally intensive but also faces the problem of the vanishing gradient problem [9] that makes the training process even slower. To improve the training performance under Deep Learning schemes several strategies have been developed, most of them based on some pre-training phase used to find good starting point synaptic weights from which apply the final supervised phase.

Current implementations of Deep Learning models require the use of parallel strategies to speed up the training process. In this sense alternatives based on cluster computing, GPUs and FPGAs are sensible strategies, each of them having their benefits and drawbacks [10, 11]. Field Programmable Gate Arrays (FPGA) [12] are reprogrammable silicon chips, using prebuilt logic blocks and programmable routing resources that can be configured to implement custom hardware functionality. The main advantage of FPGAs in comparison to PC implementations lies on their intrinsic parallelism but with the disadvantage over PCs and GPUs that they are programmed using VHDL that usually is harder and time consuming. FPGA implementations of neural networks have been analyzed in several studies [13, 14, 15]. Even if recent advances in the computational power of these boards have permitted an increase in the size of the architectures that can be implemented, they are still limited, and in general, the number of layers in the architecture should be prefixed before its application. For this reason, we introduce in this work a layer multiplexing scheme for the on-chip training of deep feedforward neural architectures using the BP algorithm, in which only a single layer of neurons is physically implemented , but this layer can be reused any number of times in order to simulate architectures with several hidden layers, the on-chip learning implementations includes both training and execution phases of the algorithm [15, 16]. Regarding this type of approach, Himavathi et al. [17] have used it previously for neural network training but under an off-chip learning scheme, in which only the synaptic weights of the final model are transmitted to the FPGA that acts as a hardware accelerator.

2 FPGA Layer Multiplexing Scheme Implementation of the BP Algorithm

We describe in this section the layer multiplexing scheme for the Back-Propagation algorithm, which will be divided in 3 different sequential processes: the computation of the neuron output values (S), the calculation of the deltas of each neuron (δ), and the update of synaptic weights. 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.

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. 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 to be implemented using a layer multiplexing scheme but nothing 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. Fig. 1 shows as in a layer multiplexing scheme 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 the layer that 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 outputs 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

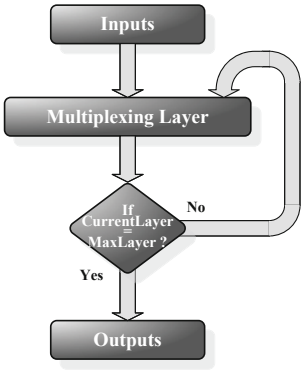

Fig. 1 Layer multiplexing scheme for the simulation of deep feed-forward neural network architectures.

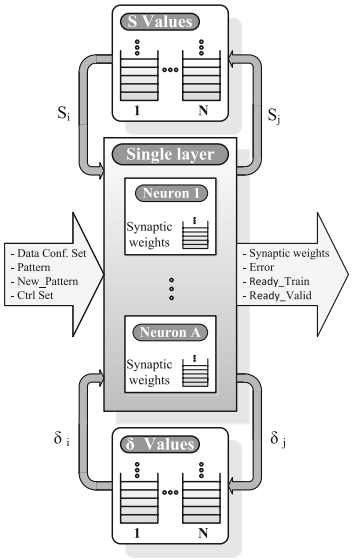

Fig. 2 Schematic representation of the layer multiplexing procedure used for the implementation of the BP algorithm.

values are obtained, they are backwardly transmitted to the previous layer in order to compute the delta values for these set of neurons. 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.

Table 1 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

The Fig. 2 shows a scheme of the architecture block that performs the layer multiplexing procedure for physically implementing a single layer of neurons. This single layer is composed of A neurons blocks 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. 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.

3 Results

We present in this section results from the implementation of both algorithms (BP and C-Mantec) in a Xilinx Virtex-5 board. Table 1 shows some characteristics of the Virtex-5 XC5VLX110T FPGA, indicating its main logic resources.

Several test cases were analyzed to verify the correct FPGA implementation of the model, comparing the results with those obtained from a PC and with previously published results. These 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 in all experiments, and using data from the well-known Iris set.

Table 2 shows the generalization ability obtained for several architectures with different numbers of hidden layers for PC and FPGA implementations. The first column indicates the number of hidden layer present in the architecture, the second column shows the generalization obtained using the PC implementation (mean computed over 100 independent runs), while third and fourth columns shows the results for

Table 2 Generalization ability for the Iris data set for neural network architectures with different numbers of hidden layers for PC and FPGA implementations.

Layers	Type Implementation		
	PC	FPGA	
		Layer Multiplexing	Fixed Layers
1	0.9376	0.9391	0.9406
2	0.9516	0.9442	0.9471
3	0.9518	0.9493	–
5	0.9333	0.9371	–
7	0.8702	0.8842	–
10	0.5273	0.5998	–
15	0.3064	0.3120	–

Table 3 Computation times expressed as a function of the number of hidden layers (X) in the neural architectures for the PC and layer multiplexing FPGA implementations for the cases of including 5 and 20 neurons in each of the layers.

Device	# Neurons	
	5	20
PC	$1.11 \cdot X + 6.25$	$1.26 \cdot X + 5.75$
FPGA	$0.044 \cdot X + 0.028$	$0.134 \cdot X + 0.029$

two different FPGA implementations: the layer multiplexing scheme proposed in this work and the fixed layer scheme utilized in Ref. [18] (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 maximum number of layers shown in the table is 15 because from this number of hidden layers on the obtained generalization is approximately one third, that is the expected value for random choices for a problem with three classes.

From the results shown in Table 2 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 start to decrease to reach the level expected for random choices for a number of layers equal to 15.

Fig. 3 shows the whole learning procedure execution time (in seconds and in logarithmic scale (right Y-axis)) for PC and FPGA implementations a function of the number of hidden layers present in the architecture with five (a) and twenty (b) neurons per layer. The graph also shows a third curve that indicates the number of times (#Times, in linear scale) that the FPGA implementation is faster than the PC one. The number of epochs used was of 1000.

Table 3 shows the results of a linear fitting for the computation time for a variable number of hidden layers, indicated by X in the equations shown. The fitted values were obtained from the cases shown in Fig. 3 for the FPGA and PC implementation in which the number of neurons in each of the hidden layers are fixed to five and twenty.

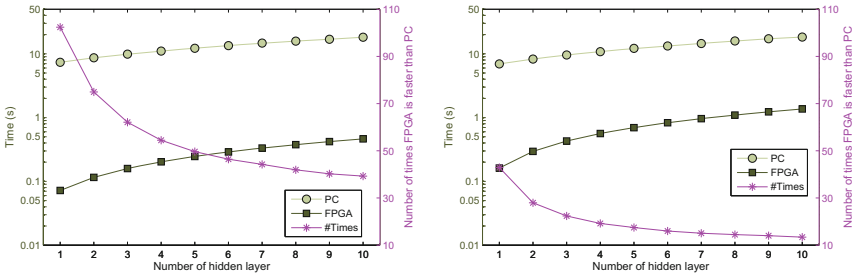

Fig. 3 Time and number of times that the FPGA is faster than the PC as a function of the number of hidden layers of the architecture (a) 5 neurons and (b) 20 neurons.

4 Discussion and Conclusions

We have introduced in this work an FPGA implementation for deep neural network architectures using a layer multiplexing scheme. The layer multiplexing scheme used permits to simulate a neural network with several hidden layers by only implementing physically a single hidden layer of neurons. Main advantages of this approach are that very deep neural network architectures can be analyzed through a simple and flexible framework with a very efficient FPGA resource utilization. The implementation has been tested and compared to an existing PC one, obtaining that for a large number of hidden layers the FPGA implementation is approximately 20 to 30 times faster than the PC one. The layer multiplexing scheme used permits in principle the simulation of very deep networks with any number of hidden layers, but memory resource constraints limit the current implementation to approximately hundred hidden layers, that from the point of view of existing Deep Learning models is quite large. The on-chip implementation carried out includes also a validation phase to avoid overfitting effects. Using the Back-Propagation algorithm for training the several hidden layers architectures shown that the performance of the standard BP algorithm starts to degrade when 10 or more hidden layers are present in the architectures, so additional strategies are needed in order to improve the training. In this sense, we believe that the present implementation will facilitate the study of this and related issues, helping to understand very deep neural networks.

Acknowledgements The authors acknowledge support from Junta de Andalucia through grants P10-TIC-5770, from CICYT (Spain) through grant TIN2014-58516-C2-1-R, and from the Universidad de Málaga, Campus de Excelencia Internacional Andalucía Tech (all including FEDER funds). And thanks Yachay Tech for financial support for science research.

References

1. Haykin, S.: Neural Networks: A Comprehensive Foundation, 2nd edn. Prentice Hall PTR, Upper Saddle River (1998)
2. Reed, R.D., Marks, R.J.: Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks. MIT Press, Cambridge (1998)
3. Werbos, P.J.: Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. PhD thesis, Harvard University (1974)
4. Rumelhart, D., Hinton, G., Williams, R.: Learning representations by back-propagating errors. Nature **323**(6088), 533–536 (1986)
5. Gómez, I., Franco, L.: Neural network architecture selection: Can function complexity help? Neural Processing Letters **30**, 71–87 (2009)
6. Hinton, G.E., Osindero, S., Teh, Y.W.: A fast learning algorithm for deep belief nets. Neural Comput. **18**(7), 1527–1554 (2006)
7. Schmidhuber, J.: Deep learning in neural networks: An overview. Neural Networks **61**, 85–117 (2015)
8. Ciresan, D.C., Meier, U., Gambardella, L.M., Schmidhuber, J.: Deep, big, simple neural nets for handwritten digit recognition. Neural Computation **22**(12), 3207–3220 (2010)
9. Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. In: Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS 2010). Society for Artificial Intelligence and Statistics, pp. 249–256 (2010)
10. Suresh, S., Omkar, S.N., Mani, V.: Parallel implementation of back-propagation algorithm in networks of workstations. IEEE Trans. Parallel Distrib. Syst. **16**(1), 24–34 (2005)
11. Huqqani, A.A., Schikuta, E., Ye, S., Chen, P.: Multicore and {GPU} parallelization of neural networks for face recognition. Procedia Computer Science **18**, 349–358 (2013). 2013 International Conference on Computational Science
12. Kilts, S.: Advanced FPGA Design: Architecture, Implementation, and Optimization. Wiley-IEEE Press (2007)
13. Le Ly, D., Chow, P.: High-performance reconfigurable hardware architecture for restricted boltzmann machines. IEEE Transactions on Neural Networks **21**(11), 1780–1792 (2010)
14. Kim, L.W., Asaad, S., Linsker, R.: A fully pipelined fpga architecture of a factored restricted boltzmann machine artificial neural network. ACM Trans. Reconfigurable Technol. Syst. **7**(1), 5–23 (2014)
15. Ortega-Zamorano, F., Jerez, J., Franco, L.: Fpga implementation of the c-mantec neural network constructive algorithm. IEEE Transactions on Industrial Informatics **10**(2), 1154–1161 (2014)
16. Dinu, A., Cirstea, M., Cirstea, S.: Direct neural-network hardware-implementation algorithm. IEEE Transactions on Industrial Electronics **57**(5), 1845–1848 (2010)
17. Himavathi, S., Anitha, D., Muthuramalingam, A.: Feedforward neural network implementation in fpga using layer multiplexing for effective resource utilization. IEEE Transactions on Neural Networks **18**(3), 880–888 (2007)
18. Ortega-Zamorano, F., Jerez, J., Urda Munoz, D., Luque-Baena, R., Franco, L.: Efficient implementation of the backpropagation algorithm in fpgas and microcontrollers. IEEE Transactions on Neural Networks and Learning Systems **PP**(99), 1–11 (2015)