Int. J. Simul. Multidisci. Des. Optim. **14**, 1 (2023)
© M.J. Zhang et al., Published by EDP Sciences, 2023
https://doi.org/10.1051/smdo/2023001

Available online at:
https://www.ijsmdo.org

RESEARCH ARTICLE

OPEN ACCESS

Real-time fast learning hardware implementation

Ming Jun Zhang[1,2,*], Samuel Garcia[2], and Michel Terre[2]

[1] DGUT-Cnam Institute, Dongguan University of Technology, 1, Daxue Rd., Songshan Lake, Dongguan, Guangdong Province, PR China

[2] HESAM/CNAM/CEDRIC 292 rue Saint Matin, 75003 Paris, France

Received: 23 October 2022 / Accepted: 30 January 2023

Abstract. Machine learning algorithms are widely used in many intelligent applications and cloud services. Currently, the hottest topic in this field is Deep Learning represented often by neural network structures. Deep learning is fully known as deep neural network, and artificial neural network is a typical machine learning method and an important way of deep learning. With the massive growth of data, deep learning research has made significant achievements and is widely used in natural language processing (NLP), image recognition, and autonomous driving. However, there are still many breakthroughs needed in the training time and energy consumption of deep learning. Based on our previous research on fast learning architecture for neural network, in this paper, a solution to minimize the learning time of a fully connected neural network is analysed theoretically. Therefore, we propose a new parallel algorithm structure and a training method with over-tuned parameters. This strategy finally leads to an adaptation delay and the impact of this delay on the learning performance is analyzed using a simple benchmark case study. It is shown that a reduction of the adaptation step size could be proposed to compensate errors due to the delayed adaptation, then the gain in processing time for the learning phase is analysed as a function of the network parameters chosen in this study. Finally, to realize the real-time learning, this solution is implemented with a FPGA due to the parallelism architecture and flexibility, this integration shows a good performance and low power consumption.

Keywords: Neural networks / learning algorithms / deep learning / parallel architecture / FPGA / hardware accelerator

1 Introduction

In recent years, with the advent of the Big Data era, the amount of data created in just a few years has exploded [1], and thanks to the rise in data volume, increased computing power, and the advent of deep learning, artificial intelligence has begun to grow rapidly. One of the main research domains in artificial intelligence (AI) is artificial neural network (ANN) [2]. ANN algorithm is inspired by biological neurons, and it has been introduced in 1943 by Warren McCullough and Walter Pitts [3], however in 1969, M. Minsky and S. Papert, published the book "Perceptrons" [4], which has pointed out that single-layer Perceptron could not implement a heterogeneous gate (XOR), and multi-layer Perceptron could not give a learning algorithm, so it was useless. Given the status of the two men in the field of AI, the book generated a great response and ANN research fell into a slump. In 1985, Rumelhart et al. re-generated the Backpropagation Algorithm [5] for training the weights of multilayer perceptron, which fine-tunes the coefficients of the connection weights of each layer backwards to optimize the network weights by comparing the actual output with the error generated by the theoretical output, thus solving the problem that Minsky thought could not be solved. This has brought a second boom in ANN research. In 2006, Geoffrey Hinton, a professor at the University of Toronto and a leading figure in the field of machine learning, and his students published an article in "Science" that formally introduced the concept of deep learning [6] and started the wave of deep learning in academia and industry.

Since then, with the improvement of algorithms, computing speed and the emergence of big data, "deep learning" in NLP, image recognition, chess (AlphaGo, AlphaGo Zero [7]), autonomous driving and other applications have achieved remarkable results, ANN rejuvenated, and thus set off the third climax that continues to date.

In brief, there are two main reasons why deep learning techniques have made great breakthroughs in recent years;

– The first reason is due to the continuous expansion of the training data set. In the 80 s and 90 s, data sets contained tens of thousands training examples, such as the MNIST [8], then in early 2010 s, larger data sets (Image Net

* e-mail: ming-jun.zhang@lecnam.net

This is an Open Access article distributed under the terms of the Creative Commons Attribution License (https://creativecommons.org/licenses/by/4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

dataset, for instance) appeared with hundreds of thousands to tens of millions of training examples. The most used ImageNet dataset, ILSVRC 2012-2017, consists of approximately 1.5 million images.
– The second reason is the huge increase in computing power of hardware devices. Improvements in semiconductor devices and computing architectures have significantly reduced the time overhead of network computation, including the training process and the inference process.

There is no doubt that AI has made a very prominent progress in recent years, in many of the applications currently proposed [9–11], the networks are trained off-line, and the real-time focus is mainly on the processing of the data by the network and not on the learning phase with the adaptation of the free network parameters. Therefore, how to accelerate the training of neural networks, by software as well as hardware, how to do real-time training, and how to reduce the power consumption, especially during the training, are still big challenges for the researchers [12].

To accelerate the training of neural networks, we have proposed a solution to minimize the learning time of a fully connected neural network [13] and based on that, this paper presents a processing architecture in which the treatments applied to the examples of the learning base are strongly parallelized and anticipated, even before the parameters adaptation of the previous examples are completed. This strategy finally leads to a delayed adaptation and the impact of this delay on the learning performances is analysed through a simple replicable school case study. It is shown that a reduction of the adaptation step size could be proposed to compensate errors due to the delayed adaptation, then the gain in processing time for the learning phase is analysed as a function of the network parameters chosen in this study.

For the implementation of this processing architecture and considering the parallelism character of neuron network algorithm, GPU, FPGA and ASIC offer the possibilities to do so.

The GPU is widely used as hardware accelerators both for learning and for reference. GPU has high memory bandwidth and highly efficient matrix-based floating-point calculations [14]. However, GPU consumes considerably more power when it is operating compared with FPGA and ASIC [15].

FPGA has a form of parallelism with ASIC in development, they both achieve good performance with very lower power consummation compared with GPU, ASIC based accelerators demand a longer development cycle, a higher cost but with less flexibility than FPGA based accelerators. FPGA has now become another alternative hardware accelerator solution [16].

FPGAs are composed by a set of programmable logic units which are called Configurable Logic Blocks (CLB), a programmable interconnection network, on-die processors, transceiver I/O's, RAM blocks, DSP engines, etc. FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing. They have the following advantages compared to others hardware solutions: (1) Low power consumption; (2) Customizable; (3) Reconfigurable; (4) High performance [17–19].

FPGAs used to develop with the hardware description languages such as VHDL and Verilog. This was a hindrance for software developers. With the adaptation of software level programming framework such as the Open Computing Language (OpenCL) integrated in FPGAs developing tools, FPGAs are employed frequently in deep learning [20].

However FPGAs are often used as reference, not for real-time learning [21]. In this work, we focus on a full implementation (feed forward propagation and back propagation) on FPGA a Multi-Layer Perceptron neural network algorithm (MLPNN) which can be used as reference when the network is trained, and for real-time learning when the network is used for on-chip training. The following sections are organized as follows: in Section 2, we will introduce the main notations and equations of the algorithm. In Section 3, we will present the organization and grouping of calculations. In Section 4, we will demonstrate a theoretical study on the impact of a delayed adaptation of the parameters. A practical simulation case is observed in Section 5. In the last two sections, we will illustrate the hardware implementation and summarize the result of synthesis of this implementation, where an adequate explanation is presented.

2 Algorithm analysis

A Multi-Layer Perceptron neural network algorithm is composed by neurons and grouped by layers. The outputs of the neurons of the previous layer become the input of each neuron of the present layer. The notations and the equations are presented hereafter.

2.1 Notations

In the sequel of the paper, we consider a fully connected neural network Figure 1 with q layers numbered from 1 to q.

We introduce the following notations:
– N_L : the number of neurons of the Lth layer.
– y_i^L : the input of the activation function of the ith neuron of the Lth layer.
– z_i^L : the output of the ith neuron of the Lth layer.
– w_{ij}^L : the synaptic coefficient between the jth neuron of the $(L-1)th$ layer and the ith neuron of the Lth layer.
– θ_i^L : the constant coefficient of the ith neuron of the Lth layer (bias).
– t_i : the ith desired values at the output of the neural network.

2.2 Algorithm's equations

Concerning the activation function in the neuron, we consider either a non-linear sigmoid function $f(x)$ defined as follows:

$$f(x) = \frac{e^x}{1 + e^x}, \tag{1}$$

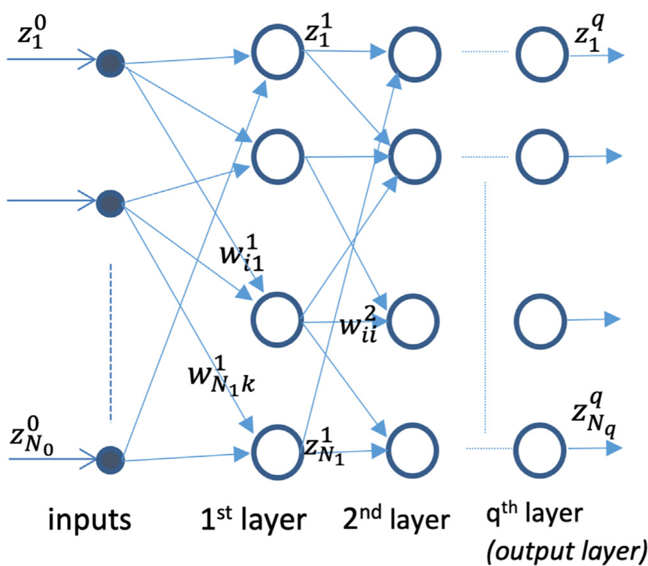

Fig. 1. Fully connected neural network.

or a linear function, especially for the last layer:

$$f(x) = x. \tag{2}$$

The first input layer is fed with $s = \left(z_1^0, z_2^0, \ldots, z_{N_0}^0\right)$. The free parameters are all "synaptic coefficients": w_{ij}^L and the constant parameters θ_i^L, for all layers $L \in [1,\ q]$.

The main objective in this part is to apply a gradient descent algorithm to find all w_{ij}^L and θ_i^L terms and, for that purpose, we will have to calculate the derivative of the error with respect to the synaptic coefficients and the constant parameters. This partial derivative will be represented as $\frac{\partial E}{\partial y_j^L}$.

The equation for the **forward propagation**, for $L = 1$ to q is:

$$y_i^L = \sum_{j=1}^{N_{L-1}} w_{ij}^L z_j^{L-1} + \theta_i^L, \tag{3}$$

$$z_i^L = f\left(y_i^L\right), \tag{4}$$

The error calculation equation is:

$$E(s) = \frac{1}{2}\sum_{j=1}^{N_q}\left(z_j^q - t_j\right)^2. \tag{5}$$

Therefore the equations for the Backward propagation initialed with a sigmoid function for the last layer $j \in [1, N_q]$ is:

$$\frac{\partial E}{\partial y_j^q} = \left(z_j^q - t_j^q\right) z_j^q \left(1 - z_j^q\right). \tag{6}$$

Or initialed with a linear function for the last layer is:

$$\frac{\partial E}{\partial y_j^q} = \left(z_j^q - t_j^q\right). \tag{7}$$

For $L = q$-1 to 1, for all j indexes of the concerned layer the equation is:

$$\frac{\partial E}{\partial y_j^L} = z_j^L\left(1 - z_j^L\right)\sum_{k=1}^{N_{L+1}} w_{kj}^{L+1}\frac{\partial E}{\partial y_k^{L+1}}. \tag{8}$$

Finally, the equations for the "synaptic coefficients" and the bias **Adaptation** are:

$$w_{ij}^L = w_{ij}^L - \delta\frac{\partial E}{\partial y_i^L} z_j^{L-1}. \tag{9}$$

$$\theta_i^L = \theta_i^L - \delta\frac{\partial E}{\partial y_i^L}. \tag{10}$$

3 Organization and grouping of calculations

We propose to use an architecture in which a set of calculations are performed on each clock period as described in Table 1. Most operations can be performed by dedicated hardware resources on one system clock. Some functions which are more complicated to obtain, will require several system clocks.

For the sake of simplicity for the paper presentation and without any loss of generalization we consider a 2 layers neural network ($q = 2$) in the sequel of the presented tables. To carry out the algorithm on a structure with calculations in parallel and with anticipation it is thus necessary to introduce new variables identified by letters in Table 2. These variables must be saved in dedicated memories. In our example, the worst case corresponds to $F_j(t+6)$ that will be calculated at $(t+6)$ and used at $(t+7)$ and $(t+15)$. It is also necessary to memorize the synaptic coefficients of layer 2 (w_{ij}^2) that are used in forward propagation at time $(t+5)$ and used in backward error updates at time $(t+13)$.

Considering Table 2, it appears that dedicated hardware processing resources are necessary to calculate the 17 variables listed in the table. As shown in Figure 2 that, thanks to a hardware resource allocation algorithm, the treatment can be done with 16 hardware dedicated processing resources, identified as R_1 to R_{16}. Some of them are loaded at 100%, it is typically the case for R_3 and R_6 that must calculate the sigmoid function, while others are loaded at 66% as R_1, R_2, R_4 and R_5. All other resources are loaded at 33%. The global load of hardware resources R_i being equal to 50.7%. The time necessary for the propagation of calculations is equal, for this $q = 2$ layers-network, to 17 step time. Each step time equals to one or several system clocks. This result can be extended for any value for q and the processing time T is equal to:

$$T = 8q + 1. \tag{11}$$

Table 1. Clock time periodes required for algorithm equations.

Inputs: $\left(z_1^0, z_2^0, \ldots, z_{N_0}^0\right)$		
Step time	Index i, j, k	Equations
1	$\forall j \in [1,\ N_0]$;$\forall i \in [1,\ N_1]$	$w_{ij}^1 z_j^0$
2	$\forall i \in [1, N_1]$	$y_i^1 = \sum_{j=1}^{N_0} w_{ij}^1 z_j^0 + \theta_i^1$
3,4,5	$\forall i \in [1, N_1]$	$z_i^1 = f\left(y_i^1\right)$
6	$\forall j \in [1,\ N_1]$;$\forall i \in [1,\ N_2]$	$w_{ij}^2 z_j^1$ and $\left(1 - z_j^1\right)$
7	$\forall i \in [1, N_2]$	$y_i^2 = \sum_{j=1}^{N_1} w_{ij}^2 z_j^1 + \theta_i^2$ and $z_j^1\left(1 - z_j^1\right)$
8,9,10	$\forall i \in [1, N_2]$	$z_i^2 = f\left(y_i^2\right)$
11	$\forall j \in [1, N_2]$	$\left(z_j^2 - t_j^2\right)$ and $\left(1 - z_j^2\right)$
12	$\forall j \in [1, N_2]$	$\frac{\partial E}{\partial y_j^2} = \left(z_j^2 - t_j^2\right) z_j^2 \left(1 - z_j^2\right)$
13	$\forall j \in [1,\ N_1]$; $\forall k \in [1,\ N_2]$	$\frac{\partial E}{\partial y_k^2} w_{kj}^2$
	$\forall i \in [1, N_2]$	$\theta_i^2 = \theta_i^2 - \delta \frac{\partial E}{\partial y_i^2}$
	$\forall j \in [1,\ N_1]$;$\forall i \in [1,\ N_2]$	$\frac{\partial E}{\partial y_i^2} z_j^1$
14	$\forall j \in [1, N_1]$	$\sum_{k=1}^{N_2} w_{kj}^2 \frac{\partial E}{\partial y_k^2}$
	$\forall j \in [1,\ N_1]$; $\forall i \in [1,\ N_2]$	$w_{ij}^2 = w_{ij}^2 - \delta \frac{\partial E}{\partial y_j^2} z_j^1$
15	$\forall j \in [1, N_1]$	$\frac{\partial E}{\partial y_j^1} = z_j^1 \left(1 - z_j^1\right) \sum_{k=1}^{N_2} w_{kj}^2 \frac{\partial E}{\partial y_k^2}$
16	$\forall i \in [1,\ N_1]$; $\forall j \in [1,\ N_0]$	$z_j^0 \frac{\partial E}{\partial y_i^1}$
	$\forall i \in [1, N_1]$	$\theta_i^1 = \theta_i^1 - \delta \frac{\partial E}{\partial y_i^1}$
17	$\forall i \in [1,\ N_1]$;$\forall j \in [1,\ N_0]$	$w_{ij}^1 = w_{ij}^1 - \delta \frac{\partial E}{\partial y_j^1} z_j^0$

4 A theoretical study on the impact of a delayed adaptation of the synaptic coefficient

Based on the analysis in the previous sections, there is a lag in the synaptic coefficient adaptation in the backpropagation algorithm for neural networks. In this paper, in addition to accelerating the training time by parallel operations, we further accelerate the training time by the synaptic coefficient adaptation without delay method in the backpropagation. In this section, a theoretical analysis is done on the synaptic coefficient adaptation without delay method.

We take a simple case below Figure 3, where $z(n)$ is the output of the neuron, $w(n-1)$ is the synaptic coefficient, $t(n)$ is the desired values at the output of the neural network, f is the active function, $e(n)$ is the error between the output of neuron and the desired output (Fig. 3).

The prime objective of our study focuses on minimizing errors. Considering the following adaptation equations:

$$\min_w e(n)^2, \tag{12}$$

which is used to calculate the minimum of the sum of the squared errors, with

$$e(n) = t(n) - f(w(n-1)z(n)). \tag{13}$$

The equation

$$w(n) = w(n-1) + \delta e(n) z(n), \tag{14}$$

is used to adapt the synaptic coefficient. In order to simplify the description and not to affect the convergence results, we will use a linear function as activation function in the rest of the exposition in this section.

Table 2. Variables used in the architecture and ressources involved.

Inputs: $\left(z_1^0, z_2^0, \ldots, z_{N_0}^0\right)$	
Step time	Equations
1	$A_{ij}(t+1) = w_{ij}^1(t) z_j^0(t)$
2	$B_i(t+2) = \sum_{j=1}^{N_0} A_{ij}(t+1) + \theta_i^1(t+1)$
3,4,5	$C_j(t+5) = f(B_i(t+2))$
6	$D_{ij}(t+6) = w_{ij}^2(t+5) C_j(t+5)$
	$F_j(t+6) = \left(1 - C_j(t+5)\right)$
7	$G_i(t+7) = \sum_{j=1}^{N_1} D_{ij}(t+6) + \theta_i^2(t+4)$
	$H_j(t+7) = C_j(t+5)) F_j(t+6)$
8,9,10	$I_i(t+10) = f(G_i(t+7))$
11	$J_j(t+11) = \left(I_j(t+10) - t_j^2(t)\right)$
	$K_j(t+11) = \left(1 - I_j(t+10)\right)$
12	$L_j(t+12) = J_j(t+11) I_j(t+10) K_j(t+11)$
13	$M_{kj}(t+13) = L_k(t+12) w_{kj}^2(t+5)$
	$\theta_i^2(t+13) = \theta_i^2(t+12) - \delta L_i(t+12)$
	$N_{ij}(t+13) = L_i(t+12) C_j(t+5)$
14	$O_j(t+14) = \sum_{k=1}^{N_2} M_{kj}(t+13)$
	$w_{ij}^2(t+14) = w_{ij}^2(t+13) - \delta N_{ij}(t+13)$
15	$P_j(t+14) = F_j(t+6) O_j(t+14)$
16	$Q_{ij}(t+16) = P_i(t+14) z_j^0(t)$
	$\theta_i^1(t+16) = \theta_i^1(t+15) - \delta P_i(t+14)$
17	$w_{ij}^1(t+17) = w_{ij}^1(t+16) - \delta Q_{ij}(t+16)$

Where the additional operation and the multiplication operation time are denoted respectively with T_+ and $T_\times$. The time of multiplication operation with δ is denoted with T_δ.

The time for a full iteration (Fig. 4) is:

$$T_i = 2T_\times + 2T_+ + T_\delta. \tag{15}$$

For $N_R + 1$ input data set, the computation time for $N_R + 1$ iterations is:

$$T_{NR} = (N_R + 1)T_i = (N_R + 1)(2T_\times + 2T_+ + T_\delta). \tag{16}$$

For $i = 0$ to N_R where we keep the "old" coefficient, and then we try to minimize the sum of the squares of the errors made:

$$\min_{w'} \sum_{i=0}^{N_R} e_{n-1}(n+i)^2. \tag{17}$$

with

$$e_{n-1}(n+i) = t(n+i) - w'(n-1)z(n+i). \tag{18}$$

Hence the adaptation equation is:

$$w'(n+N_R) = w'(n-1) + \delta \sum_{i=0}^{N_R} e_{n-1}(n+i) z(n+i) \tag{19}$$

For $i = 0$ to N_R, when $i = 0$:

$$S_{n-1}(n+i) = S_{n-1}(n+i-1) + e_{n-1}(n+i) z(n+i) \tag{20}$$

Finally, when $i = N_R$:

$$w'(n+N_R) = w'(n-1) + \delta S_{n-1}(n+N_R). \tag{21}$$

The whole delay of the calculation is presented as Figure 5. Thus we can parallelize and start the products input by coefficient before adaptation. The parameters will be only adapted at the end of sum of the errors as described in Figure 6.

Hence, the computation time T'_{N_R} for $N_R + 1$ iterations with a delayed adaptation parallelized structure is:

$$T'_{N_R} = (N_R + 1)T_\times + T_\times + 2T_+ + T_\delta \tag{22}$$

The considerable gain of learning time $\left(T_{N_R} - T'_{N_R}\right)$ with a delayed adaptation parallelized structure is:

$$N_R(T_\times + 2T_+ + T_\delta). \tag{23}$$

On all tested examples, such as illustrated in Figure 8, the adaptation with delay converges as fast as without delay.

5 Simulation results

To test the algorithm, we have chosen to ask the neural network to learn how to calculate a Discrete Fourier Transform. The advantage of this option is to have a perfect replicable simulation without referring to training or a generalization base. Because the training vectors can be generated randomly in a quasi-infinite way, the simulation is not dependent on the size of the existing database, either.

CLK	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	R_{12}	R_{13}	R_{14}	R_{15}	R_{16}
1	A	G	C		H	I			M	N	θ			Q	θ	
2		B	C	K		I	J					O	w			w
3	D		C	F	P	I		L								
4	A	G	C		H	I			M	N	θ			Q	θ	
5		B	C	K		I	J					O	w			w
6	D		C	F	P	I		L								
7	A	G	C		H	I			M	N	θ			Q	θ	
8		B	C	K		I	J					O	w			w
9	D		C	F	P	I		L								
10	A	G	C		H	I			M	N	θ			Q	θ	
11		B	C	K		I	J					O	w			w
12	D		C	F	P	I		L								
13	A	G	C		H	I			M	N	θ			Q	θ	
14		B	C	K		I	J					O	w			w
15	D		C	F	P	I		L								
16	A	G	C		H	I			M	N	θ			Q	θ	
17		B	C	K		I	J					O	w			w
18	D		C	F	P	I		L								
19	A	G	C		H	I			M	N	θ			Q	θ	
20		B	C	K		I	J					O	w			w
21	D		C	F	P	I		L								
22	A	G	C		H	I			M	N	θ			Q	θ	
23		B	C	K		I	J					O	w			w

Fig. 2. Temporal implementation on dedicated hardware processing resources (green: data of time t, red: data of time $t+1$, blue: data of time $t+2$, brown: data of time $t+3$, yellow: data of time $t+4$).

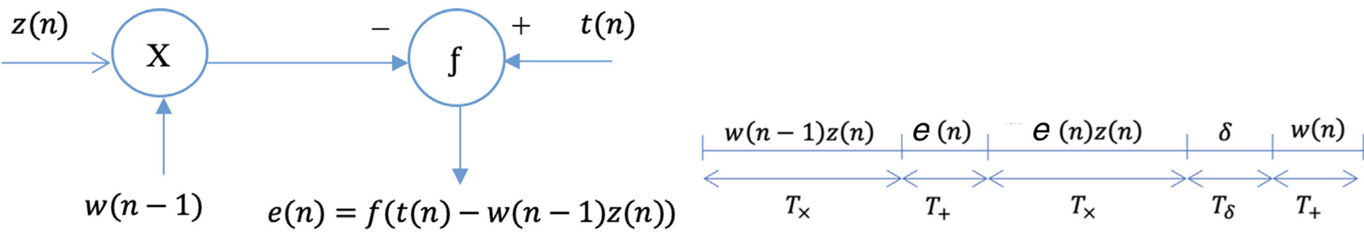

Fig. 3. Simplifed error model.

Fig. 4. Simplifed error model.

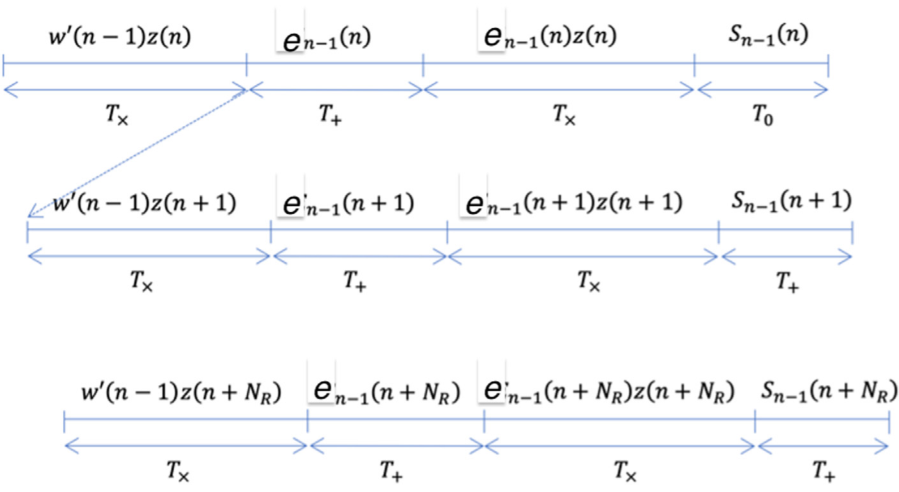

Fig. 5. Calculation time for N_R iterations.

0	$T_\times$	T_+	$T_\times$	T_0						
1		$T_\times$	T_+	$T_\times$	T_+					
'2			$T_\times$	T_+	$T_\times$	T_+				
⋮				⋱	⋱	⋱	⋱			
N_R					$T_\times$	T_+	$T_\times$	T_+	T_δ	T_+

Fig. 6. Illustration of adaptation with delay.

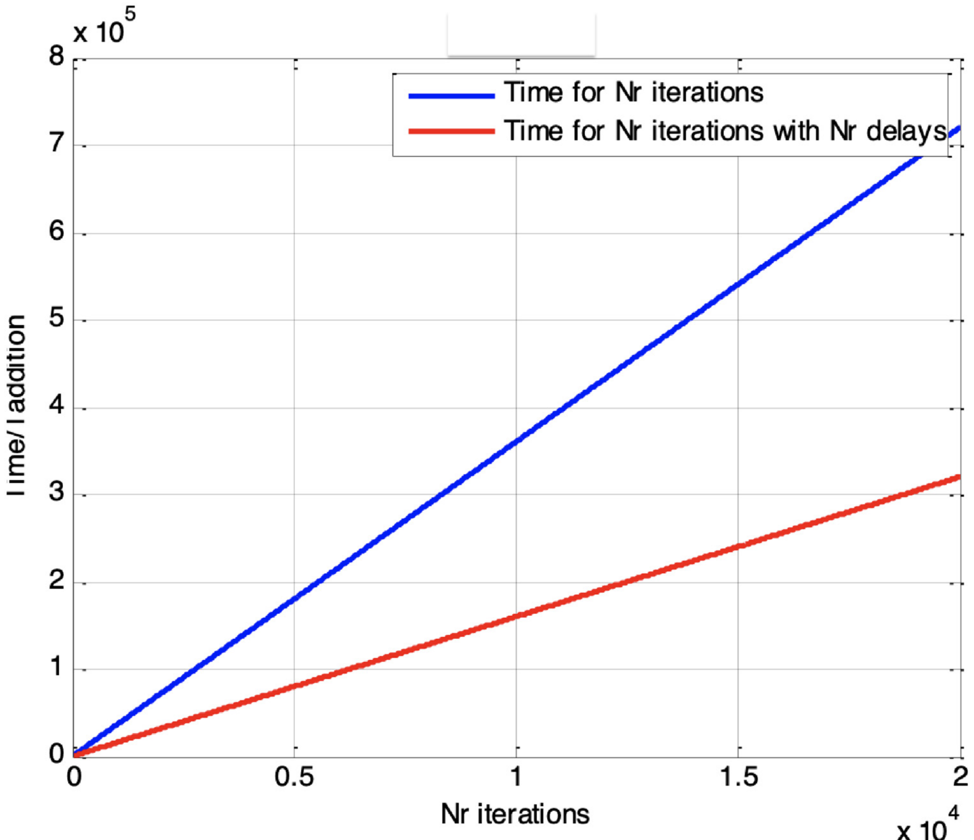

Fig. 7. Time comparison of operations with or without delay.

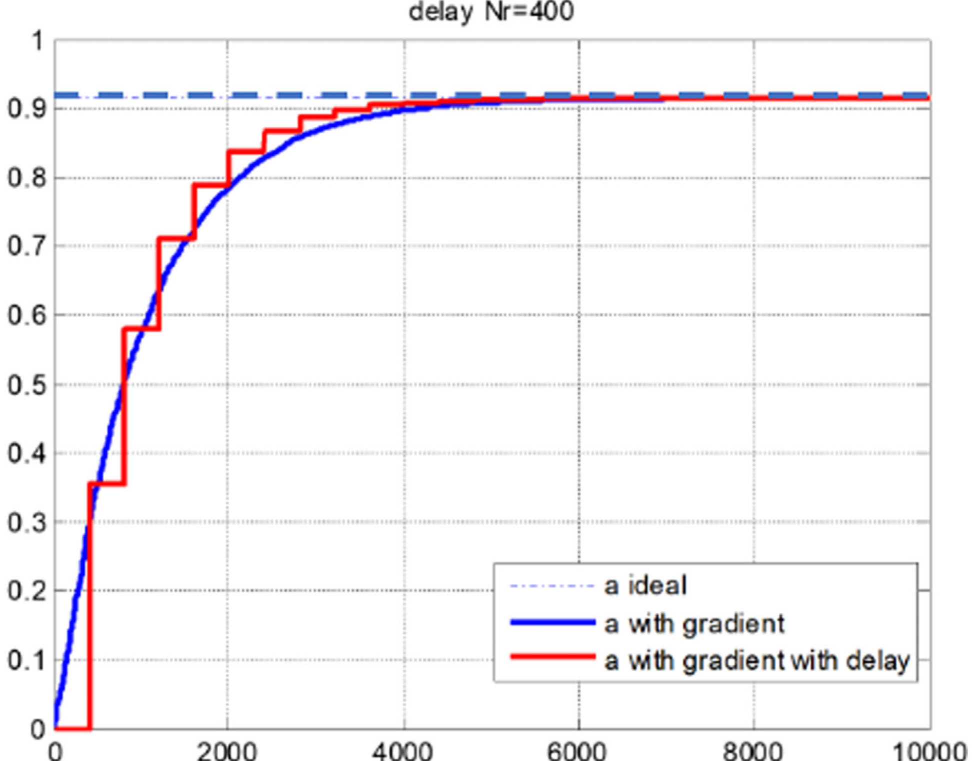

Fig. 8. Illustration of convergence with or without delay.

For each iteration of the training, we therefore randomly generate a vector of N_{FFT} Gaussian random complex terms $\{x_i\}_i \in (0, N_{FFT} - 1)$. Each term having zero mean and unity variance for its real and imaginary parts (Table 3). Then, for each iteration, we compute the Fourier Transform of this Gaussian vector as follows:

$$\{t_k\}_k \in (0, N_{FFT} - 1) = FFT\{\{x_i\}_i \in (0, N_{FFT} - 1)\} \quad (24)$$

and use the terms of this Fourier Transform as the desired signal $\left(t_{1\to N_q}\right)$. The input and output signals being Gaussian we present in input a vector made up of the real parts then the imaginary parts. The input and the output vectors have respectively the size of $N_0 = 2 * N_{FFT}$ and $N_q = 2 * N_{FFT}$.

We let the training run for several million examples (between 50 and 5 million depending on the simulations) for different values of the training delay and we varied the gradient adaptation step size δ. Finally, at each iteration, we took the squared error E which we integrated over an exponential window with a forgetting factor λ by means of the following equation:

$$e(t+1) = \lambda e(t) + (1-\lambda)\sum_{j=1}^{N_q}\left(z_j^q - t_j\right)^2. \quad (25)$$

We then divided this error by the power of the desired signal to obtain a normalized mean squared error (NMSE) that we plotted in logarithm in base 10. Main simulation parameters are summarized in Table 4.

Table 3. Complex to real mapping.

$$\left(\quad z^0_{1\to\frac{N_0}{2}} = \mathrm{Real}\{x_{0\to(NFFT-1)}\}\right.$$

$$\left(z^0_{\frac{N_0}{2}+1\to N_0} = \mathrm{Imag}\{x_{0\to(NFFT-1)}\}\right.$$

$$\left(t_{1\to\frac{N_q}{2}} = \mathrm{Real}\{x_{0\to(NFFT-1)}\}\right.$$

$$\left(t_{\frac{N_q}{2}+1\to N_q} = \mathrm{Imag}\{x_{0\to(NFFT-1)}\}\right.$$

It appears in Figure 9 that the adaptation delay degrades the performances rather quickly and can even block the convergence of the adaptation. This is very noticeable as soon as this delay exceeds about ten clock times. However, the simulation results presented in Figure 10 were obtained with a δ adaptation step size of the gradient equals to $5 * 10^{-4}$. It can be seen, in Figure 10, that reducing this step size to 10^{-4} would sufficiently improve the performances significantly Tables 5. In the meantime, it tends to solve the non-convergence problem and allow the adaptation with delay to have performances closely resembling the adaptation without delay. However, this reduction of the adaptation step increases the convergence time of the learning. We can see that for an objective of the logarithm of the normalized mean squared error equals to -2.4, we need to present 5 million examples with an adaptation step size of $5 * 10^{-4}$ and 20 million examples with an adaptation step

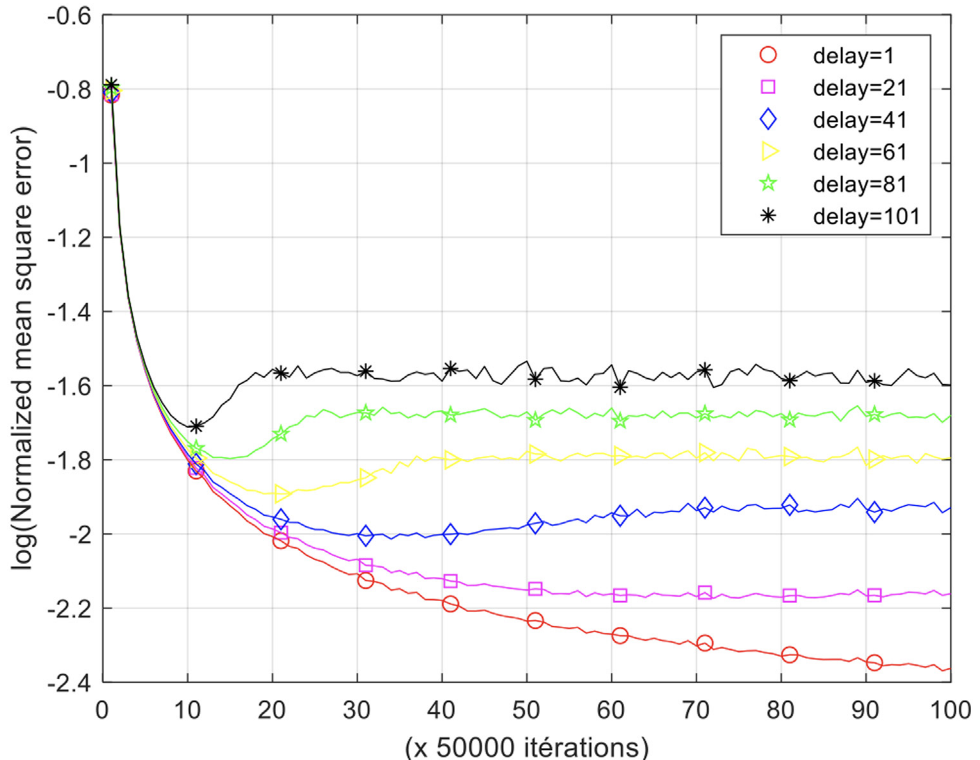

Fig. 9. Logarithm of the normalized mean squared error vs number of iterations with $\delta = 5 \times 10^{-4}$ ($N_{FFT} = 16$).

size of 10^{-4}. Looking through the basic example we have used, we can therefore conclude that the implementation architecture proposed in this article optimizes the processing time but at the cost of an increase in convergence time by a factor of 4.

6 The design of real-time learning architecture

In this section, we have designed the hardware level in two parties: computation unit level and system level. The objective is to parallelize the calculations as much as possible and specially to anticipate them without waiting for the adaptation of the coefficients linked to the previous examples. This allows to optimize the use of hardware resources at the cost of a latency for the adaptation of the coefficients. We introduce the time of an addition T_+, a multiplication T_x, the calculation of the exponential function T_{exp} and a division $T_/$.

Based on the algorithm features described previously and considering the characteristic of FPGA, the computation unit level of our implementation consists of three parties which are the forward propagation, back propagation, and adaptation.

6.1 Forward propagation

Denoted FP, implements the two forward propagation equations (3) and (4) of Section 2. It is represented in Figure 11. Parallelizing all the multiplications, we arrive, for the FP component, at a latency T for the forward propagation step of the L layer, equal to:

$$T_{NC_L} \cong T_x + (\log_2 N_{L-1} + 2)T_+ + T_{exp} + T_/. \tag{26}$$

6.2 Backpropagation

For the backpropagation equations of Section 2, we implement a component noted BP_q for the last layer or BP_L for an inner layer. The two components are represented in Figures 12 and 13.

Parallelizing all the calculations for the output layer, we arrive at latency times:

$$T_{BP_q} = 2T_x + T_+. \tag{27}$$

$$T_{BP_L} \cong 2T_x + (\log_2 N_{L-1} + 1)T_+. \tag{28}$$

6.3 Adaptation

For the adaptation equations of Section 2 we implement a component denoted UP represented in Figure 14.

For the UP module, the latency of backpropagation in L^{th} layer is:

$$T_{UP} = 2T_+ + T_x. \tag{29}$$

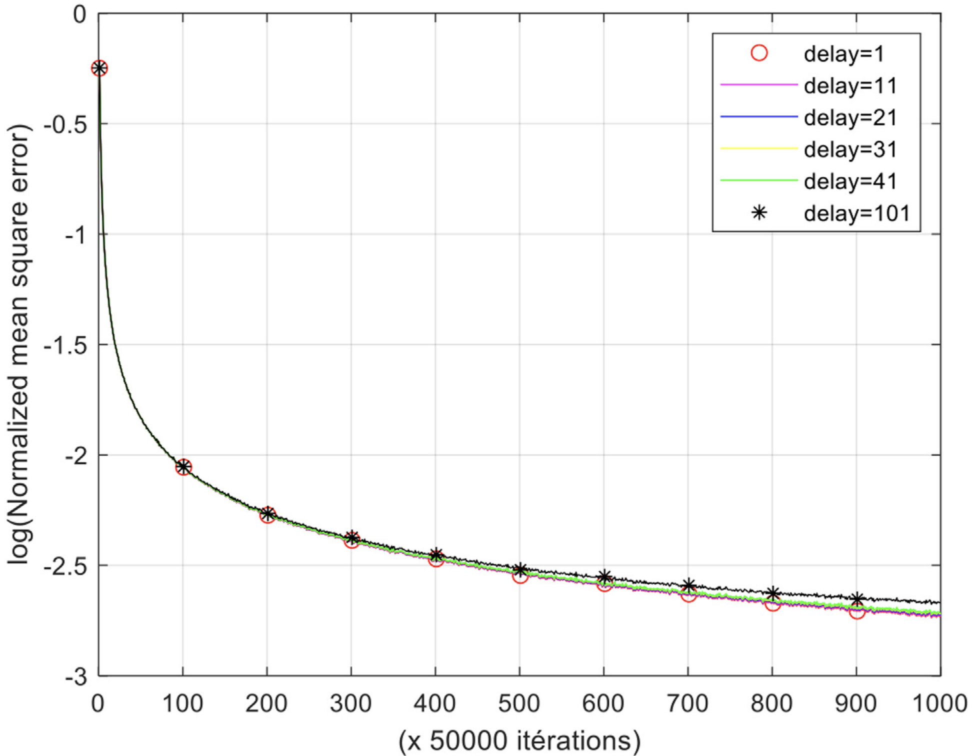

Fig. 10. Logarithm of the normalized mean squared error vs number of iteration i^{th} $\delta = 10^{-4}$ $(N_{FFT} = 16)$.

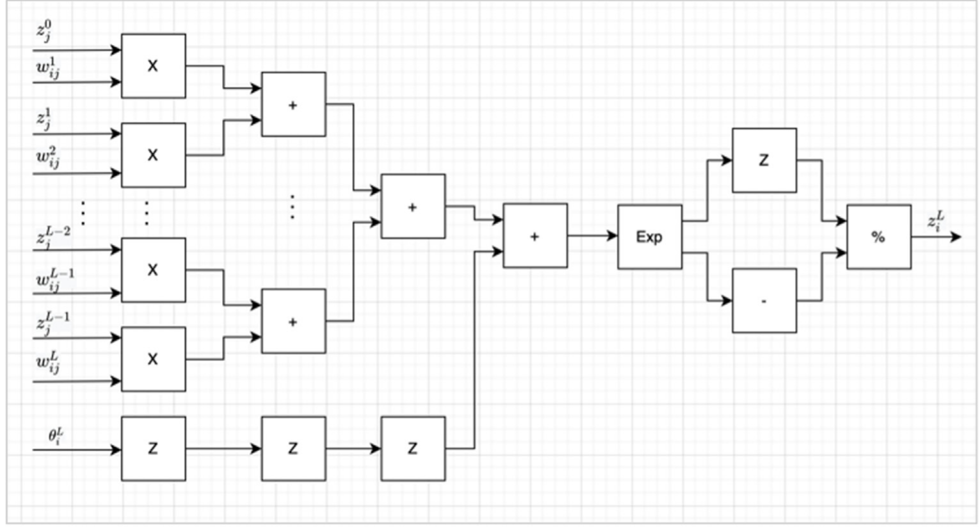

Fig. 11. Architecture of FP.

The structures of these modules of each layer are the same with the equivalent numbers of inputs and one output. Once we know the number of neurons, and layers, we can generate and implement the neural network quickly. Because of the reconfigurability of FPGA, the neural network can be regenerated as many times as we want the neural networks in the same chip depending on the requirements.

We form the above computation units as IP cores in FPGA and use them to design the System level. The whole architecture of the system level design is presented in Figure 15.

In system level, a control unit was designed to send the control signals for each component and synchronize the whole system with the same system clock. To thin the flow of data, the delay modules were added to hold the signals.

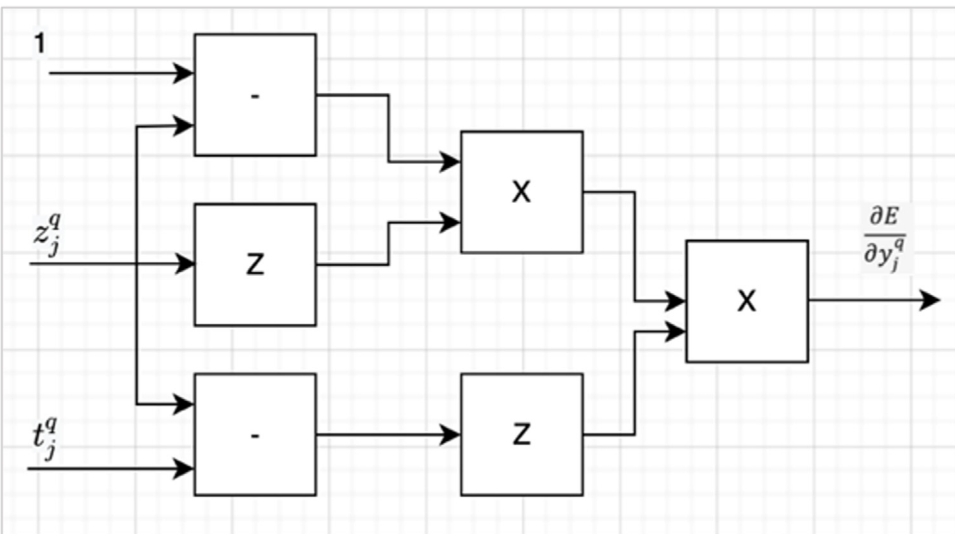

Fig. 12. Architecture of BP_q for for the calculation of the backpropagation equations for the last layer.

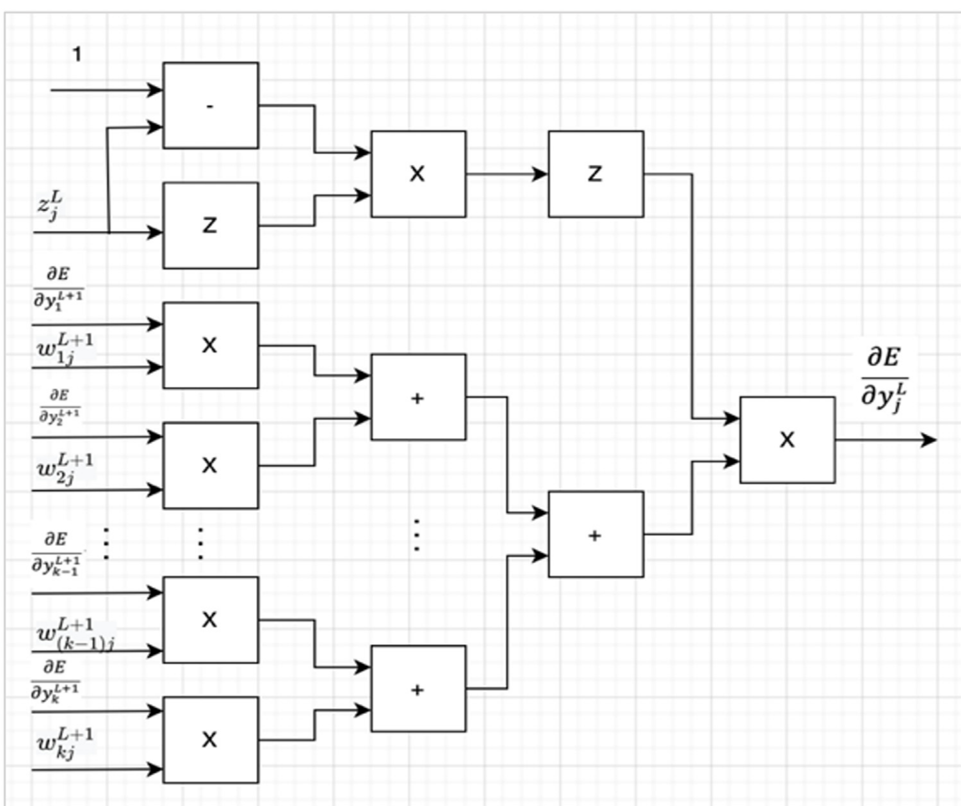

Fig. 13. Architecture of BP_L for the calculation of the backpropagation equations for the L layer.

Among all the controlled signals, the "mode" signal is used to choose between the "reference" mode and the "learning" mode. With this pipeline parallelism system level architecture, one training data is received for each system clock, and all the operations run simultaneously from the first Total Latency which is:

$$T_{Lat} = \sum_{L=1}^{q} T_{NC_L} + T_{BP_q} + \sum_{L=1}^{q-1} T_{BP_L} + T_{UP} \tag{30}$$

The system repeats the same operation for each system clock until the valid signal becomes '0'. Figure 16 illustrates the flowing of the training data. The advantages of this

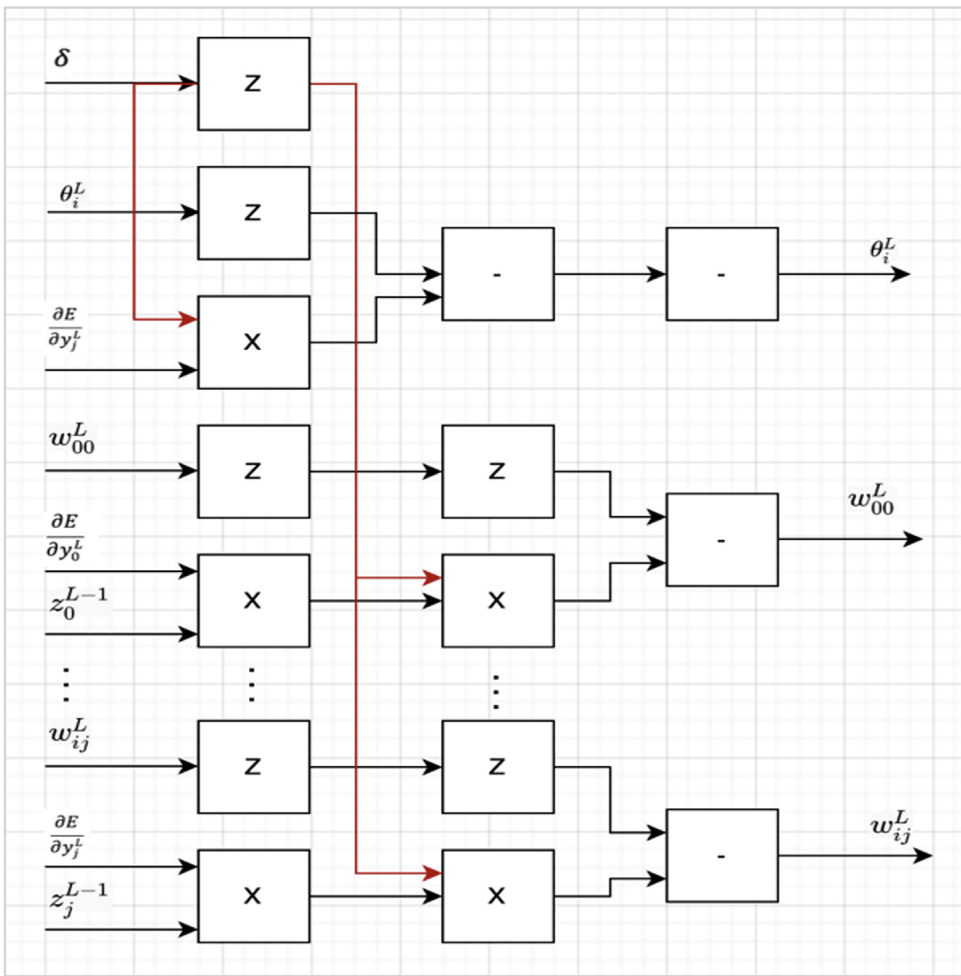

Fig. 14. Architecture of UP for the calculation of the adaptation equations.

Table 4. Main simulation parameters.

$N_{FFT} = 16$
$q = 2$
$\lambda = 0.9999$
$\delta = 10^{-4}$ *or* 5×10^{-4}
1st layer : sigmoid, 2nd layer : linear

design are: firstly, no timeout for the inputs – to charge the training data sets for training. Secondly, all the neurons in each layer are calculated at the same time, whatever the number of neurons in a layer, the latency of this layer in question is the same for one neuron or for n neurons. Thirdly, the computation units are operating at the same time, the using of on-chip resources are optimized. Thus, our approach and contribution can expedite the learning of the neural network. Even the number of neurons and layers are important, this work offers a remarkable reward for training time.

Table 5. NMSE with an adaptation step size equals to 10^{-4}.

Delay	Normalized mean squared error
1	1.8575×10^{-3}
11	1.8735×10^{-3}
21	1.8909×10^{-3}
31	1.9099×10^{-3}
41	1.9301×10^{-3}
101	2.1437×10^{-3}

7 Synthesis results

In this paper, we have designed a neural network with 32 neurons as inputs, one hidden layer with 32 neurons and one output layer with 32 neurons to calculate a Discrete Fourier Transform. The input and the desired output are generated from a Matlab program. The data used is 32-bit

Table 6. Latency summary.

Module	FP1	FP2	BP2	BP1	UP2	UP1	Delay1	Delay2	Total
Latency	195	119	102	125	98	98	541	221	639

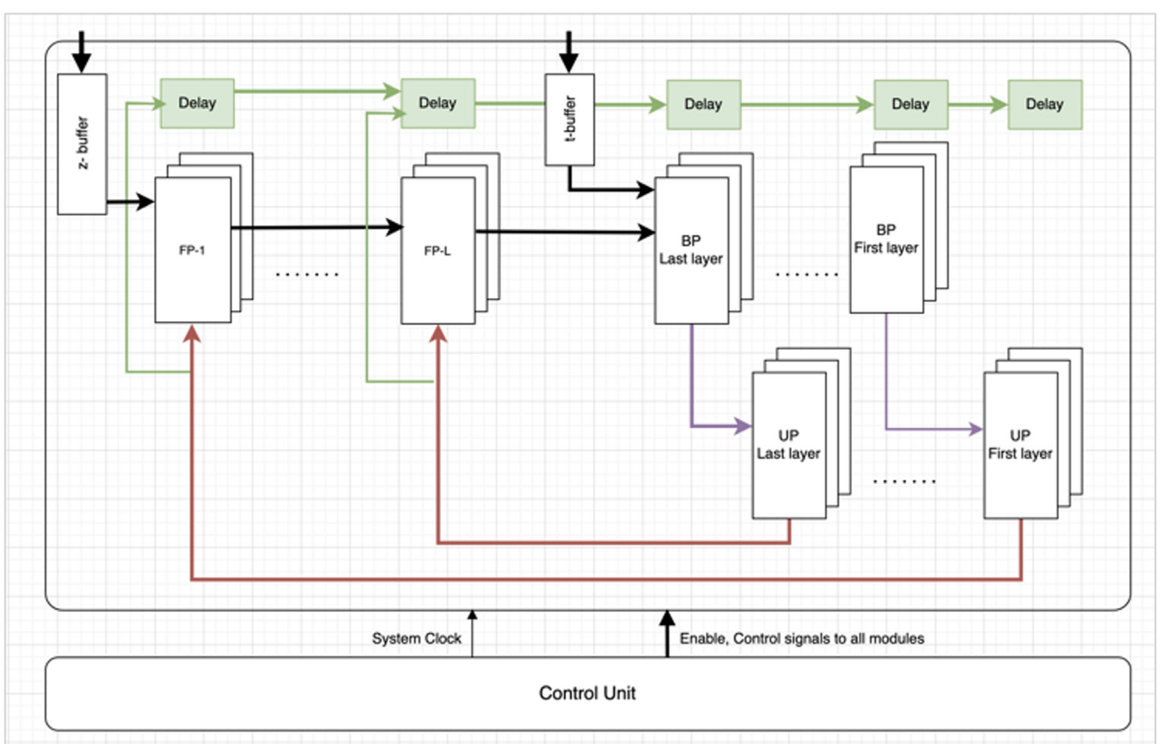

Fig. 15. Whole architecture of a real-time learning implementation.

Clock	Data Set 1	Data Set 2	Data Set 3	Data Set 4	Data Set 5	Data Set 6	Data Set 7	Data Set 8
1	FP1							
2	FP2	FP1						
...	...	FP2	FP1					
...	FPq	...	FP2	FP1				
...	FPq	FPq	...	FP2	FP1			
...	BPq-1; UPq	BPq	FPq	...	FP2	FP1		
...	...; UPq-1	BPq-1; UPq	BPq	FPq	...	FP2	FP1	
...	BP1;...	...; UPq-1	BPq-1; UPq	BPq	FPq	...	FP2	FP1
Total Latency	UP1	BP1;...	...; UPq-1	BPq-1; Upq	BPq	FPq	...	FP2
Total Latency + 1		UP1	BP1;...	...; UPq-1	BPq-1; UPq	BPq	FPq	...
Total Latency + 2			UP1	BP1;...	...; UPq-1	BPq-1; UPq	BPq	FPq
Total Latency + 3				UP1	BP1;...	...; UPq-1	BPq-1; UPq	BPq
Total Latency + 4					UP1	BP1;...	...; UPq-1	BPq-1; UPq
Total Latency + 5						UP1	BP1;...	...; UPq-1
Total Latency + 6							UP1	BP1;...
Total Latency + 7								UP1

Fig. 16. Training data flow.

Table 7. Resource utilization summary for each cell.

Cell	FP1	FP2	BP2	BP1	UP2	UP1	Delay1	Delay2
ALUT	2691	1203	190	1364	2185	2185	0	0
REG	5106	2889	382	3340	5428	5428	18835456	7694336
MLAB	10	4	2	6	2	2	0	0
RAM	10	3	3	3	3	3	0	0
DSP	42.5	33	2	36	33	33	0	0

Table 8. Resource utilization summary for each layer.

Layer	FP1	FP2	BP2	BP1	UP2	UP1	Delay1	Delay2
ALUT	86112	38496	6080	43648	69920	69920	0	0
REG	163392	92448	12224	106880	173696	173696	18835456	7694336
MLAB	320	128	64	192	64	64	0	0
RAM	320	96	96	96	96	96	0	0
DSP	1360	1056	64	1152	1056	1056	0	0

data of floating point. To implement this neural network, we have done an analysis in the first place about the latency for each component to develop the delay modules.

According to the pipeline architecture of a whole architecture of a Real-Time Learning Implementation demonstrated in Figure 15, we need two modules to manage the delays. The latency of the first one, denoted Delay 1, is:

$$\text{Delay1} = FP1 + FP2 + BP2 + BP1. \quad (31)$$

And the second one, denoted Delay 2, of which latency is:

$$\text{Delay2} = FP2 + BP2. \quad (32)$$

Hence the total Latency is:

$$\text{Total Latency} = FP1 + FP2 + BP2 + BP1 + UP1. \quad (33)$$

We may draw from this theoretic analysis that the first adaptation of synaptic, and bias finished after the 639th system clock Tables 6 then there was one adaptation for each system clock from the $640th$ one.

In order to choose the suitable target FPGA device family, we did follow analysis using various registers, DSP, RAM, MLAB and ALUT for each component and each layer, with summaries presented in Tables 7 and 8.

In Tables 7 and 8, there are a considerably large number of registers depending on the FPGA board which has been chosen. This can be transformed in part into RAM after the placement and routing. Comparing with the Intel Agilex product table, we can see that even a basic family of Agilex offers a large capacity to receive this implementation. Therefore, we chose the Intel Agilex AGFB022R31C2E1V as selected device in our project with Quartus 21.4.

We have developed six components individually with the Intel® HLS Compiler: FP1 for the neuron of forward propagation of the first hidden layer, FP2 for the neuron of forward propagation of output layer, BP2 for the back-propagation of the neuron of output layer, BP1 for the backpropagation of the neuron of hidden layer, UP2 for synaptic and bias adaptation of the neuron of output layer and UP1 for synaptic and bias adaptation of the neuron of hidden layer.

Finally, we have programmed a top-level file to integrate all the components to create our neural network. In addition, the basic components previously developed previous can be reused to regenerate easily the new neural networks far more easily for any number of neurons and layers.

After the synthesis, in terms of power utilization, the logic units used 10.768 watts, the RAM blocks used 3.195 watts, the DSPs used 9.022 watts the clock used 8.392 watts, and the power static was 3.210 watts. The summary is described in Figure 17. From architectural point of view, this implementation tends to save tremendous training time and power. In our case, the frequency adapted is only 400 Mhz, however with the new generation of FPGA, the frequency may achieve 1.5 Ghz. Thus, the process of learning can be faster.

8 Conclusion

This paper proposes we have proposed a hardware decomposition of the computations of a whole architecture for a Real-Time Learning Implementation based on the gradient backpropagation algorithm. It has shown, in our case, that a complete step of coefficient adaptation could be performed in 23 step times. The proposed parallelization leads to an anticipation of the computations and to a delayed update of the free coefficients of the network. It was shown that the effects of this adaptation delay could be compensated by a reduction of the adaptation step size. We finally, conclude that by slightly slowing down the adaptation phase, we can propose a parallelized architecture which will significantly accelerate the processing time

Power Summary

Resource Type	Power (W)
Logic	10.768
RAM	3.195
DSP	9.022
Clock	8.392
PLL	0.852
IO	9.920
Transceiver	1.360
HPS	0
Crypto	0
HBM	0
Miscellaneous	1.247
Total Dynamic Power	**44.757**
Static Power (Before Savings)	5.129
Static Power Savings	-1.919
Total Static Power	**3.210**
SmartVID Power Savings	**-0.917**
Total Power	**47.051**

Fig. 17. Summary of power utilization.

with a positive overall result. Hence, we did also a hardware implementation with FPGA that can easily integrate this kind of design, and the pipeline structure design in FPGA. It suggests a solution to receive one training data set for each system clock, not only can it reduce the training time, but also maximize the utilization of the resource on chip.

References

1. Z. Xu, N. Tang, C. Xu, X. Cheng, Data science: connotation, methods, technologies, and development, Data Sci. Manag. **1**, 32–37 (2021)
2. Z. Yan, Z. Jin, S. Teng, G. Chen, D. Bassir, Measurement of Bridge Vibration by UAVs Combined with CNN and KLT Optical-Flow Method. Appl. Sci. **12**, 1581 (2022)
3. W.S. McCulloch, W. Pitts, A logical calculus of the ideas immanent in nervous activity, Bull. Math. Biophys. **5**, 115–133 (1943)
4. H.D. Block, A review of perceptrons: An introduction to computational geometry, Inf. Control **17**, 501–522 (1970)
5. D.E. Rumelhart, J.L. McClelland, Learning Internal Representations by Error Propagation (1987), pp. 318–362
6. G.E. Hinton, R.R. Salakhutdinov, Reducing the dimensionality of data with neural networks, Science (New York, N.Y.) **313**, 504–507 (2006)

7. D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou et al., Mastering the game of Go without human knowledge, Nature **550**, 354–359 (2017)
8. H. Gao, S. Tomić, J. Nikolić, Z. Perić, D. Aleksić, Performance of post-training two-bits uniform and layer-wise uniform quantization for MNIST dataset from the perspective of support region choice, Math. Probl. Eng. **2022**, 1463094 (2022)
9. A. Van Biesbroeck, F. Shang, D. Bassir, CAD model segmentation via deep learning, Int. J. Comput. Methods 18 (2020), 10.1142/S0219876220410054
10. V. Shelar, S. Subramani, D. Jebaseelan, R-tree data structure implementation for Computer Aided Engineering (CAE) tools, Int. J. Simulat. Multidiscipl. Des. Optim. **12**, 6 (2021)
11. A. Ignatov, G. Malivenko, R. Timofte, S. Chen, X. Xia, Z. Liu, Y. Zhang, F. Zhu, J. Li, X. Xiao, Y. Tian, X. Wu, C. Kyrkou, Y. Chen, Z. Zhang, Y. Peng, Y. Lin, S. Dutta, S. Das, S. Siddiqui, Fast and Accurate Quantized Camera Scene Detection on Smartphones, Mobile AI 2021 Challenge: Report (2021), pp. 2558–2568
12. Y. Hu, Y. Liu, Z. Liu, A survey on convolutional neural network accelerators: GPU, FPGA and ASIC, *2022 14th International Conference on Computer Research and Development (ICCRD)* (2022), pp. 100–107
13. M.J. Zhang, S. Garcia, M. Terre, Fast Learning Architecture for Neural Networks, in *2022 30th European Signal Processing Conference (EUSIPCO)* (2022), pp. 1611–1615
14. L. Ravaglia, M. Rusci, A. Capotondi, F. Conti, L. Pellegrini, V. Lomonaco, D. Maltoni, L. Benini, Memory-latency-accuracy trade-offs for continual learning on a RISC-V extreme-edge node, *2020 IEEE Workshop on Signal Processing Systems (SiPS)* (2020), pp. 1–6
15. Y. Li, S.E. Li, X. Jia, S. Zeng, Y. Wang, FPGA accelerated model predictive control for autonomous driving, J. Intell. Connect. Veh. **5**, 63–71 (2022)
16. K. Guo, S. Zeng, J. Yu, Y. Wang, H. Yang, DL A survey of FPGA-based neural network inference accelerators, ACM Trans. Reconfig. Technol. Syst. **12**, 1–26 (2019)
17. S. Xiong, G. Wu, X. Fan, X. Feng, Z. Huang, W. Cao, X. Zhou, S. Ding, J. Yu, L. Wang, Z. Shi, MRI-based brain tumor segmentation using FPGA-accelerated neural network, BMC Bioinfor. **22**, 421 (2021)
18. C.-C. Sun, A. Ahamad, P.-H. Liu, SoC FPGA accelerated sub-optimized binary fully convolutional neural network for robotic floor region segmentation, Sensors **20**, 6133 (2020)
19. L. Wang, Y. Zhao, X. Li, An automatic conversion tool for caffe neural network configuration oriented to openCL-based FPGA platforms, 2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC) (2019), pp. 195–198
20. M. Carreras, G. Deriu, L. Raffo, L. Benini, P. Meloni, Optimizing temporal convolutional network inference on FPGA-based accelerators, IEEE J. Emerg. Selected Top. Circ. Syst. 1–1 (2020), 10.1109/JETCAS.2020.3014503
21. H. Park, C. Lee, H. Lee, Y. Yoo, Y. Park, I. Kim, K. Yi, Work-in-progress: optimizing DCNN FPGA accelerator design for handwritten hangul character recognition, in *2017 International Conference on Compilers, Architectures and Synthesis For Embedded Systems (CASES)* (2017), pp. 1–2

Cite this article as: Ming Jun Zhang, Samuel Garcia, Michel Terre, Real-time fast learning hardware implementation, Int. J. Simul. Multidisci. Des. Optim. **14**, 1 (2023)