Fast Learning Architecture for Neural Networks

ZHANG Ming Jun
CEDRIC/Laetitia
DGUT-CNAM
Dongguan, China
ming-jun.zhang@lecnam.net

GARCIA Samuel
CEDRIC/Laetitia
HESAM/CNAM
Paris, France
samuel.garcia@lecnam.net

TERRÉ Michel
CEDRIC/Laetitia
HESAM/CNAM
Paris, France
michel.terre@lecnam.net

***Abstract*—This paper proposes a solution to minimize the learning time of a fully connected neural network. The 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. Finally, the gain in processing time for the learning phase is analysed as a function of the network parameters chosen in this study.**

Keywords—Neural networks, Learning algorithms, Architecture, FPGA.

I. INTRODUCTION

Artificial intelligence algorithms based on neural networks [1] were introduced a long time ago. The founding paper is undoubtedly that of McCulloch and Pitts [2] entitled "A Logical Calculus of Ideas Immanent in Nervous Activity" which proposed the first mathematical model of an artificial neuron with a threshold response function. Since that time artificial intelligence has had an eventful history with cycles of significant advances and more stationary periods [3]. Currently, the surge in available labelled data, combined with the large computational capacities available, puts artificial intelligence back in the spotlight in many applications [4].

However, real-time learning remains a real challenge and in many of the applications currently proposed, 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. In this paper we focus on the learning phase and propose a processing architecture dedicated to this phase. We have chosen a perceptron type neural network with all layers fully connected. The analysed algorithm is the gradient backpropagation with a sigmoid activation function. Currently, the possible widely used processing architectures include Central Processing Unit (CPU), Graphics Processing Unit (GPU), Field Programmable Gate Array (FPGA), and Application Specific Integrated Circuit (ASIC) acceleration. Among them, GPU acceleration technology has frequently become the first choice for deep learning acceleration methods due to its powerful computing capabilities[5]. FPGA is composed of a logic cell array, which includes configurable logic modules, input and output modules, and internal connections. Compared with other acceleration methods, FPGA has the advantages of reconfigurability, high energy efficiency ratio, high performance, portability, and low latency. Based on these advantages, FPGA has developed rapidly in recent years and has gradually become a strong competitor of GPU in the field of algorithm acceleration. At present, there are many types of FPGA-based neural network accelerators with different design concepts [6][7] and acceleration schemes [8][9][10].

This article is organized as follows: in section II we introduce main notations and equations of the back propagation algorithm. In section III we propose the organisation of calculations and in section IV a pipeline FPGA implementation architecture is presented. A practical simulation case is described and given in section V, where a performance analysis is presented. Main conclusions are given in section VI.

II. BACKPROPAGATION ALGORITHM

A. Notations

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

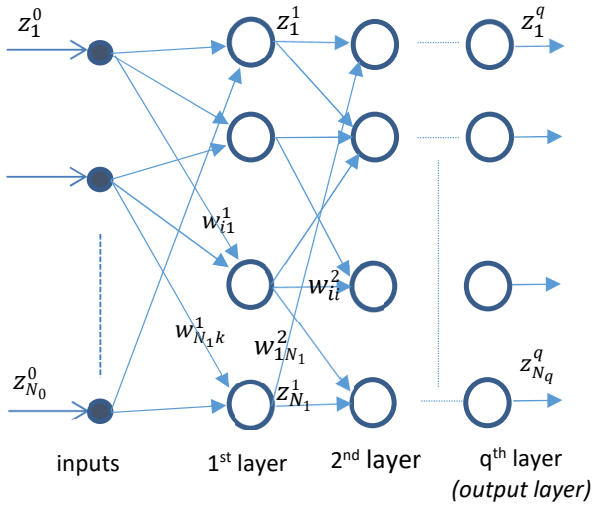

Fig. 1. Fully connected neural network.

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

B. Algorithm's equations

Concerning the activation function in the neuron, we consider either a sigmoid [11] function $f(x)$ defined as follows:

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

or a linear function, especially for the last layer:

$$f(x) = x \quad (2)$$

The first input layer is fed with $s = \left(z_1^0, z_2^0, \dots, z_{N_0}^0\right)$. 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 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 written as $\frac{\partial E}{\partial y_i^L}$ in the paper.

The complete algorithm considered in this paper is given in the table I, hereafter:

TABLE I. BACKPROPAGATION EQUATIONS

inputs: $\left\{z_1^0, z_2^0, \dots, z_{N_0}^0\right\}$

Forward propagation,

for $L = 1\ to\ q$

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

$$z_i^L = f(y_i^L)$$

Error calculation

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

Backward propagation

Initialization with a sigmoid function for the last layer ($j \in \left[1, N_q\right]$):

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

Or initialization with a linear function for the last layer:

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

for $L = q - 1\ to\ 1$, for all *j* indexes of the concerned layer:

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

Adaptation, for $L = 1\ to\ q$, for all *i* and *j* indexes of the concerned layer:

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

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

III. ORGANIZATION AND GROUPING OF CALCULATIONS

We propose to use an architecture such that a set of calculations are performed on each clock period. The organization is presented in table II hereafter. Most operations can be performed by dedicated hardware resources on one clock period. Some functions, more complicated to obtain, will require several clock periods. This is typically the case for the sigmoid function which is often approximated using a Look Up Table (LUT) and requires 3 clock periods [12].

TABLE II. CLOCK TIME PERIODES REQUIRED FOR ALGORITHM EQUATIONS

inputs: $\left\{z_1^0, z_2^0, \dots, z_{N_0}^0\right\}$		
CLK		
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	$\forall i \in [1, N_1]$	$z_i^1 = f(y_i^1)$
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	$\forall i \in [1, N_2]$	$z_i^2 = f(y_i^2)$
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	$\forall j \in [1, N_1]$	$\sum_{k=1}^{N_2} \frac{\partial E}{\partial y_k^2} w_{kj}^2$
	$\forall j \in [1, N_1]$ $\forall i \in [1, N_2]$	$w_{ij}^2 = w_{ij}^2 - \delta \frac{\partial E}{\partial y_i^2} z_j^1$
15	$\forall j \in [1, N_1]$	$\frac{\partial E}{\partial y_j^1} = z_j^1\left(1 - z_j^1\right) \sum_{k=1}^{N_2} \frac{\partial E}{\partial y_k^2} w_{kj}^2$
16	$\forall i \in [1, N_1]$ $\forall j \in [1, N_0]$	$\frac{\partial E}{\partial y_i^1} z_j^0$
	$\forall i \in [1, N_1]$	$\theta_i^1 = \theta_i^1 - \delta \frac{\partial E}{\partial y_i^1}$
17	$\forall i \in [1, N_1]$ $\forall j \in [1, N_0]$	$w_{ij}^1 = w_{ij}^1 - \delta \frac{\partial E}{\partial y_i^1} z_j^0$

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 III. 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 $(\boldsymbol{w_{ij}^2})$ that are used in forward propagation at time $(t+5)$ and used in backward error updates at time $(t+13)$.

TABLE III. VARIABLES USED IN THE ARCHITECTURE AND RESSOURCES INVOLVED

inputs: $\{z_1^0(t), z_2^0(t), \dots, z_{N_0}^0(t)\}$	
CLK	
1	$A_{ij}(t+1) = \boldsymbol{w_{ij}^1(t)} z_j^0(t)$
2	$B_i(t+2) = \sum_{j=1}^{N_0} A_{ij}(t+1) + \boldsymbol{\theta_i^1(t+1)}$
5	$C_i(t+5) = f(B_i(t+2))$
6	$D_{ij}(t+6) = \boldsymbol{w_{ij}^2(t+5)} C_j(t+5)$ $F_j(t+6) = \big(1 - C_j(t+5)\big)$
7	$G_i(t+7) = \sum_{j=1}^{N_1} D_{ij}(t+6) + \boldsymbol{\theta_i^2(t+4)}$ $H_j(t+7) = C_j(t+5) F_j(t+6)$
10	$I_i(t+10) = f(G_i(t+7))$
11	$J_j(t+11) = \big(I_j(t+10) - \boldsymbol{t_j^2(t)}\big)$ $K_j(t+11) = \big(1 - I_j(t+10)\big)$
12	$L_j(t+12) = J_j(t+11) I_j(t+10) K_j(t+11)$
13	$M_{kj}(t+13) = L_k(t+12) \boldsymbol{w_{kj}^2(t+5)}$
	$\theta_i^2(t+13) = \theta_i^2(t+12) - \delta L_i(t+12)$
	$N_{ij}(t+13) = L_i(t+12) C_j(t+5)$
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	$w_{ij}^1(t+17) = w_{ij}^1(t+16) - \delta Q_{ij}(t+16)$

Considering table IIII it appears necessary to have dedicated hardware processing resources to calculate the 17 variables of the table. It's shown on table IV 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 [11], 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 clock cycles.

This result can be extended for any value for q and the processing time T is equal to:

$$T = 8q + 1 \tag{3}$$

TABLE IV. 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*)

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					0	w			w

IV. FPGA ARCHITECTURE

According to the previous section, a general fully connected neural network pipeline hardware architecture based on FPGA is proposed and presented in figure 2. All the inputs: $\{z_1^0, z_2^0, \dots, z_{N_0}^0\}$ will be charged in the memory "z-buffer", the initial "synaptic coefficients": w_{ij}^L and the constant parameters θ_i^L will be saved in register w_{ij} and

register θ and finally the desired values at the output of the neural network t_i will be charged to the memory "t-buffer".

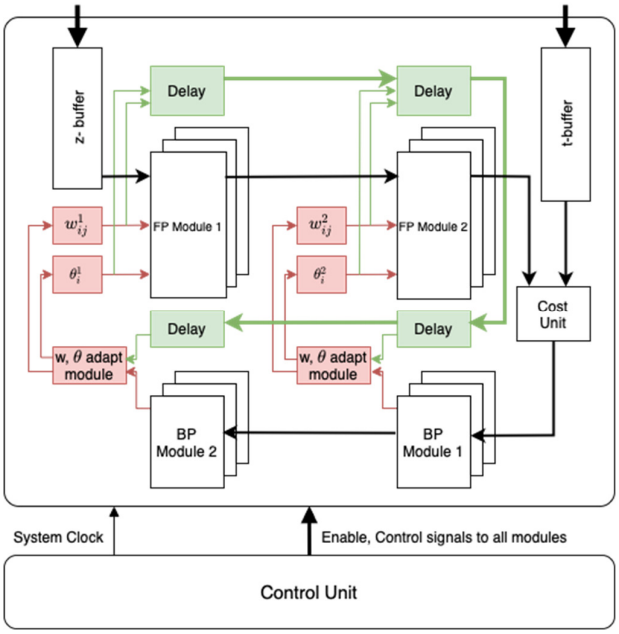

Fig. 2. FPGA Architecture.

A generic component "FP Module" must be created to realize the following operation:

$$y = f\left(\sum_{j=1}^{N_0} w_{ij} z_j + \theta_i\right) \quad (4)$$

For the forward propagation implementation, we need to implement one "FP Module" (component created in an FPGA) for each neuron of each layer. The inputs of each "FP Module" are the outputs of the neurons of the previous layer, the corresponding "synaptic coefficients" w_{ij}^L, the constant parameters θ_i^L. The "FP Module" output will be used as input of the neurons of the next layer. A "Delay module" is created to hold the values (such as w_{ij}^2 in clock time 6), to use them later (in clock time 13 for w_{ij}^2). The output of the last layer and the "t-buffer" are the inputs of the component "Cost Unit" and the output L_j of "Cost Unit" is connected to the first component "BP module" to start the process of backpropagation, it is connected also with the "w, θ adapt module" to adapt the coefficients w and bias θ of the last layer, the new coefficients and the bias will be saved in the register w_{ij} and θ of each layer respectively. The same structure will be repeated as many times as the number of layers. All modules are controlled by the "Control Unit" which provides the control signals to read or write weights (w) and bias (θ). In this pipeline structure, once the training process is starting, the training data can be charged one by one for each system clock. The specification of FPGA offers the possibility to hold the data for each system clock (T), the data will be transferred to the next component for next system clock (T+1), and free the register to receive the new data. Thus, from the 18th clock, all the operations from 1 to 17 of TABLE III will be running parallelly, in this way, we optimize the use of resources and accelerate the training process.

V. Simulation Results

To test the algorithm, we have chosen to ask the neural network to learn how to calculate a Discrete Fourier Transform. The interest of this choice is to have a perfectly replicable simulation without the need to refer to a training base and to a generalization base. The simulation is also not dependent on the size of the existing databases because the training vectors can be generated randomly in a quasi-infinite way. 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. Then, for each iteration, we compute the Fourier Transform of this Gaussian vector:

$$\{X_k\}_{k\in[0,N_{FFT}-1]} = FFT\{\{x_i\}_{i\in[0,N_{FFT}-1]}.\} \quad (5)$$

and use the terms of this Fourier Transform as the desired signal $(t_{1\to N_q})$. 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 vector and the output vector are thus of size $N_0 = 2 \times N_{FFT}$ and $N_q = 2 \times N_{FFT}$.

TABLE V. Complex to Real Mapping

$z^0_{1\to\frac{N_0}{2}} = Real\{x_{0\to N_{FFT}-1}\}$
$z^0_{\frac{N_0}{2}+1\to N_0} = Imag\{x_{0\to N_{FFT}-1}\}$
$t_{1\to\frac{N_q}{2}} = Real\{X_{0\to N_{FFT}-1}\}$
$t_{\frac{N_q}{2}+1\to N_q} = Imag\{X_{0\to N_{FFT}-1}\}$

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:

$$E(t+1) = \lambda E(t) + (1-\lambda)\sum_{j=1}^{N_q}\left(z_j^q - t_j\right)^2 \quad (6)$$

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 VI.

TABLE VI. Main Simulation Parameters

$N_{FFT} = 16$
$q = 2$
$\lambda = 0{,}9999$
$\delta = 10^{-4}$ *or* 5.10^{-4}
1st layer: sigmoid, 2nd layer: linear

It appears on figure 3 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 cycles. However, the simulation results presented in figure 4 were obtained with a δ adaptation step size of the gradient equals to 5.10^{-4}. It can be seen, in figure 4, that reducing this step size to 10^{-4} is sufficient to improve the performances very significantly, to solve the non-convergence problem and to allow the adaptation with delay to have performances very close to 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 size of 10^{-4}. In the simple example presented, 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. Globally without pipeline the time of an iteration is equal to 17 clock cycles compared to one clock cycle with the pipeline. We must therefore compare 17 times 5 million to 1 times 20 million, which leads to an overall gain in processing time equal to 4,25 for the example presented.

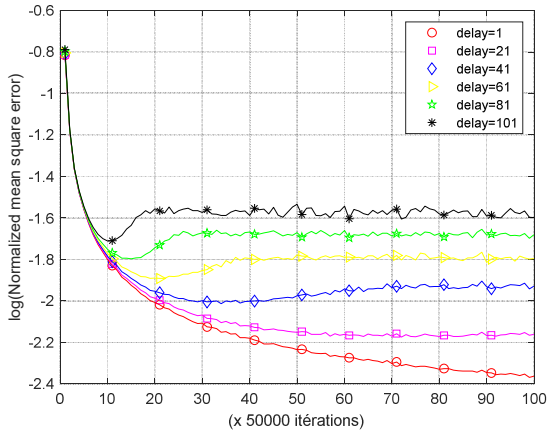

Fig. 3. Logarithm of the normalized mean squared error vs number of iterations with $\delta = 5.10^{-4}$ (N_{FFT}=16)

TABLE VII. NMSE WITH AN ADAPTATION STEP SIZE EQUALS TO 5.10^{-4}

delay	Normalized mean squared error
1	4,337 10^{-3}
21	6,90 10^{-3}
41	11,775 10^{-3}
61	16,0375 10^{-3}
81	20,875 10^{-3}
101	25,212 10^{-3}

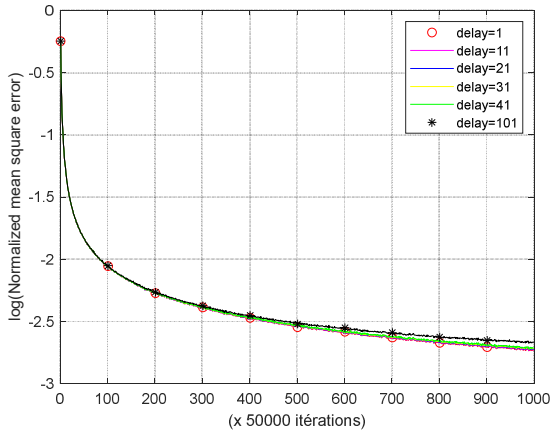

Fig. 4. Logarithm of the normalized mean squared error vs number of iteration ith $\delta = 10^{-4}$ (N_{FFT}=16)

TABLE VIII. 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}

VI. CONCLUSION

In this paper we have proposed a hardware decomposition of the computations of a learning algorithm based on the gradient backpropagation algorithm. We have shown, in the case of a simple and perfectly replicable example, that a complete step of coefficient adaptation could be performed in 23 clock cycles. 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 on this simple example 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 with a positive overall result.

REFERENCES

[1] G. G. Towell et J. W. Shavlik, "Knowledge-based artificial neural networks," *Artificial Intelligence*, Elsevier, vol. 70, n° 1, p. 119-165, oct. 1994, doi: 10.1016/0004-3702(94)90105-8.

[2] W. S. McCulloch, W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bull. Math. Biophys.*, vol. 5, n° 4, p. 115-133, déc. 1943, doi: 10.1007/BF02478259.

[3] N. Muthukrishnan, F. Maleki, K. Ovens, C. Reinhold, B. Forghani, et R. Forghani, "Brief History of Artificial Intelligence," *Neuroimaging Clin. N. Am.*, vol. 30, n° 4, p. 393-399, nov. 2020, doi: 10.1016/j.nic.2020.07.004.

[4] D. Nahavandi, R. Alizadehsani, A. Khosravi, U. R. Acharya, "Application of artificial intelligence in wearable devices: Opportunities and challenges," *Comput. Methods Programs Biomed.*, vol. 213, p. 106541, janv. 2022, doi: 10.1016/j.cmpb.2021.106541.

[5] W. Jeon, G. Ko, J. Lee, H. Lee, D. Ha, W. W. Ro, "Chapter Six - Deep learning with GPUs," *Advances in Computers*, vol. 122, S. Kim et G. C. Deka, Éd. Elsevier, 2021, p. 167-215. doi: 10.1016/bs.adcom.2020.11.003.

[6] K. Guo, S. Zeng, J. Yu, Y. Wang, H. Yang, "A Survey of FPGA-based Neural Network Inference Accelerators," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 12, n° 1, p. 1-26, avr. 2019, doi: 10.1145/3289185.

[7] R. Xie, H. Huttunen, S. Lin, S. S. Bhattacharyya, J. Takala, "Resource-constrained implementation and optimization of a deep neural network for vehicle classification," *2016 24th European Signal Processing Conference (EUSIPCO)*, 2016, pp. 1862-1866, doi: 10.1109/EUSIPCO.2016.7760571.

[8] A. Shawahna, S. M. Sait, A. El-Maleh, " FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review," *IEEE Access*, vol. 7, p. 7823-7859, 2019, doi: 10.1109/ACCESS.2018.2890150.

[9] A. Tisan, J. Chin, "An End-User Platform for FPGA-Based Design and Rapid Prototyping of Feedforward Artificial Neural Networks With On-Chip Backpropagation Learning," *IEEE Trans. Ind. Inform.*, vol. 12, n° 3, p. 1124-1133, juin 2016, doi: 10.1109/TII.2016.2555936.

[10] A. R. Omondi, J. Rajapakse, "Neural Networks in FPGAs", *in Proc. of the 9th International Conference on Neural Information Processing (ICONIP)*, Singapore, 18-22 November, 2002, pp. 954-959.

[11] M. Zhang, S. Vassiliadis, J. G. Delgado-Frias, "Sigmoid Generators for Neural Computing Using Piecewise Approximations," *IEEE Trans. Comput.*, vol. 45, 1996, pp. 1045-1049.

[12] S. Ngah, R. Abu Bakar, "Sigmoid Function Implementation Using the Unequal Segmentation of Differential Lookup Table and Second Order Nonlinear Function," *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)*,vol. 9, n° 2-8, 2017, pp 103-108.