Real-time fast learning hardware implementation

Ming Jun Zhang\textsuperscript{1,2,}\textsuperscript{*}, Samuel Garcia\textsuperscript{2}, and Michel Terre\textsuperscript{2,\textcircled{C}}

\textsuperscript{1} DGUT-Cnam Institute, Dongguan University of Technology, 1, Daxue Rd., Songshan Lake, Dongguan, Guangdong Province, PR China
\textsuperscript{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 of training examples, such as the MNIST [8], then in early 2010 s, larger data sets (Image Net...
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 consumption 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 $L$th layer.
- $y^L_i$ : the input of the activation function of the $i$th neuron of the $L$th layer.
- $z^L_i$ : the output of the $i$th neuron of the $L$th layer.
- $w^L_{ij}$ : the synaptic coefficient between the $j$th neuron of the $(L-1)$th layer and the $i$th neuron of the $L$th layer.
- $b^L_i$ : the constant coefficient of the $i$th neuron of the $L$th layer (bias).
- $t^L_i$ : the $i$th 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:

$$f(x) = \frac{e^x}{1 + e^x},$$  \hspace{1cm} (1)
or a linear function, especially for the last layer:
\[ f(x) = x. \] (2)

The first input layer is fed with \( s = (z_1^0, z_2^0, \ldots, z_N^0) \). The free parameters are all “synaptic coefficients”: \( w_{ij}^L \) and the constant parameters \( \theta_i^L \), 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 \( \theta_i^L \) 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 \( \frac{\partial E}{\partial y_j} \).

The equation for the **forward propagation**, for \( L = 1 \) to \( q \) is:
\[ y_i^L = \sum_{j=1}^{N_{i-1}} w_{ij}^L z_j^{L-1} + \theta_i^L, \] (3)
\[ z_i^L = f(y_i^L), \] (4)

The error calculation equation is:
\[ E(s) = \frac{1}{2} \sum_{j=1}^{N_q} (z_j^q - t_j)^2. \] (5)

Therefore the equations for the Backward propagation initiated with a sigmoid function for the last layer \( j \in [1, N_q] \) is:
\[ \frac{\partial E}{\partial y_j} = \left(z_j^q - t_j\right) z_j^q (1 - z_j^q). \] (6)

Or initiated with a linear function for the last layer is:
\[ \frac{\partial E}{\partial y_j} = \left(z_j^q - t_j\right). \] (7)

For \( L = q-1 \) to 1, for all \( j \) indexes of the concerned layer the equation is:
\[ \frac{\partial E}{\partial y_j^L} = z_j^L (1 - z_j^L) \sum_{k=1}^{N_{L+1}} w_{kj}^{L+1} \frac{\partial E}{\partial y_k^{L+1}}. \] (8)

Finally, the equations for the “synaptic coefficients” and the bias **Adaptation** are:
\[ w_{ij}^L = w_{ij}^L - \alpha \frac{\partial E}{\partial y_j^L} z_j^{L-1}. \] (9)
\[ \theta_i^L = \theta_i^L - \alpha \frac{\partial E}{\partial y_i^L}. \] (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 \((t + 15)\). 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 \( R_2 \) and \( R_3 \) that must calculate the sigmoid function, while others are loaded at 66% as \( R_4 \), \( R_5 \), \( R_6 \) and \( R_7 \). 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 of \( q \) and the processing time \( T \) is equal to:
\[ T = 8q + 1. \] (11)
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, $w(n)$ is the synaptic coefficient, $l(n)$ 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).

![Figure 3](image_url)

**Table 1.** Clock time periods required for algorithm equations.

<table>
<thead>
<tr>
<th>Step time</th>
<th>Index $i, j, k$</th>
<th>Equations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\forall j \in [1, N_0] ; \forall i \in [1, N_1]$</td>
<td>$w_{ij}^0$</td>
</tr>
<tr>
<td>2</td>
<td>$\forall i \in [1, N_1]$</td>
<td>$y_i^1 = \sum_{j=1}^{N_0} w_{ij}^0 z_j^0 + \theta_i^1$</td>
</tr>
<tr>
<td>3,4,5</td>
<td>$\forall i \in [1, N_1]$</td>
<td>$z_i^1 = f(y_i^1)$</td>
</tr>
<tr>
<td>6</td>
<td>$\forall j \in [1, N_1] ; \forall i \in [1, N_2]$</td>
<td>$w_{ij}^1 z_j^1$ and $(1 - z_j^1)$</td>
</tr>
<tr>
<td>7</td>
<td>$\forall i \in [1, N_2]$</td>
<td>$y_i^2 = \sum_{j=1}^{N_1} w_{ij}^1 z_j^1 + \theta_i^2$ and $z_i^2 (1 - z_i^2)$</td>
</tr>
<tr>
<td>8,9,10</td>
<td>$\forall i \in [1, N_2]$</td>
<td>$z_i^2 = f(y_i^2)$</td>
</tr>
<tr>
<td>11</td>
<td>$\forall j \in [1, N_2]$</td>
<td>$(z_i^2 - t_j^2)$ and $(1 - z_i^2)$</td>
</tr>
<tr>
<td>12</td>
<td>$\forall j \in [1, N_2]$</td>
<td>$\frac{\delta E}{\delta y_i} = (z_i^2 - t_j^2) z_i^2 (1 - z_i^2)$</td>
</tr>
<tr>
<td>13</td>
<td>$\forall j \in [1, N_2]; \forall k \in [1, N_2]$</td>
<td>$\frac{\delta E}{\delta w_{ij}^0}$</td>
</tr>
<tr>
<td></td>
<td>$\forall i \in [1, N_2]$</td>
<td>$\theta_i^2 = \theta_i^2 - \delta \frac{\delta E}{\delta y_i}$</td>
</tr>
<tr>
<td></td>
<td>$\forall j \in [1, N_2]; \forall i \in [1, N_2]$</td>
<td>$\frac{\delta E}{\delta w_{ij}^0} m_i^1$</td>
</tr>
<tr>
<td>14</td>
<td>$\forall j \in [1, N_2]$</td>
<td>$\sum_{k=1}^{N_2} w_{kj}^0 \frac{\delta E}{\delta y_k} m_i^1$</td>
</tr>
<tr>
<td>15</td>
<td>$\forall j \in [1, N_2]; \forall i \in [1, N_2]$</td>
<td>$w_{ij}^2 = w_{ij}^2 - \delta \frac{\delta E}{\delta y_i} m_i^1$</td>
</tr>
<tr>
<td>16</td>
<td>$\forall j \in [1, N_2]$</td>
<td>$\frac{\delta E}{\delta w_{ij}^0} = z_i^1 (1 - z_i^1) \sum_{k=1}^{N_2} w_{kj}^2 \frac{\delta E}{\delta y_k}$</td>
</tr>
<tr>
<td>17</td>
<td>$\forall i \in [1, N_1] ; \forall j \in [1, N_0]$</td>
<td>$\theta_i^1 = \theta_i^1 - \delta \frac{\delta E}{\delta y_i}$</td>
</tr>
<tr>
<td></td>
<td>$\forall i \in [1, N_1] ; \forall j \in [1, N_0]$</td>
<td>$w_{ij}^1 = w_{ij}^1 - \delta \frac{\delta E}{\delta y_i} z_j^0$</td>
</tr>
</tbody>
</table>

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

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

$$\min_w e(n)^2,$$

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

$$e(n) = t(n) - f(w(n - 1)z(n)).$$

The equation

$$w(n) = w(n - 1) + \delta e(n)z(n),$$

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 resources involved.

<table>
<thead>
<tr>
<th>Step</th>
<th>Equations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( A_{ij}(t+1) = w_{ij}^1(t)z_{ij}^0(t) )</td>
</tr>
<tr>
<td>2</td>
<td>( B_i(t+2) = \sum_{j=1}^{N_0} A_{ij}(t+1) + \theta_i^1(t+1) )</td>
</tr>
<tr>
<td>3,4,5</td>
<td>( C_j(t+5) = f(B_i(t+2)) )</td>
</tr>
<tr>
<td>6</td>
<td>( D_{ij}(t+6) = w_{ij}^2(t+5)C_j(t+5) ) ( F_j(t+6) = (1 - C_j(t+5)) )</td>
</tr>
<tr>
<td>7</td>
<td>( 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) )</td>
</tr>
<tr>
<td>8,9,10</td>
<td>( I_i(t+10) = f(G_i(t+7)) )</td>
</tr>
<tr>
<td>11</td>
<td>( J_j(t+11) = (I_j(t+10) - t_j^2(t)) ) ( K_j(t+11) = (1 - I_j(t+10)) )</td>
</tr>
<tr>
<td>12</td>
<td>( L_j(t+12) = J_j(t+11)I_j(t+10)K_j(t+11) )</td>
</tr>
<tr>
<td>13</td>
<td>( 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) )</td>
</tr>
<tr>
<td>14</td>
<td>( N_{ij}(t+13) = L_i(t+12)C_j(t+5) )</td>
</tr>
<tr>
<td>15</td>
<td>( O_i(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) )</td>
</tr>
<tr>
<td>16</td>
<td>( P_j(t+14) = F_j(t+6)O_j(t+14) )</td>
</tr>
<tr>
<td>17</td>
<td>( Q_{ij}(t+16) = P_i(t+14)z_{ij}^0(t) ) ( \theta_i^1(t+16) = \theta_i^1(t+15) - \delta P_i(t+14) )</td>
</tr>
<tr>
<td>18</td>
<td>( w_{ij}^1(t+17) = w_{ij}^1(t+16) - \delta Q_{ij}(t+16) )</td>
</tr>
</tbody>
</table>

For \( N_R + 1 \) input data set, the computation time for \( N_R + 1 \) iterations is:

\[
T_{NR} = (N_R + 1)T_i = (N_R + 1)(2T_x + 2T_+ + T_d).
\]  

For \( i = 0 \) to \( N_R \), when \( i = 0 \):

\[
e_{n-1}(n+i) = t(n+i) - w'(n-1)z(n+i).
\]

Hence the adaptation equation is:

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

For \( i = 0 \) to \( N_R \), when \( i = 0 \):

\[
S_{n-1}(n+i) = S_{n-1}(n+i-1) + e_{n-1}(n+i)z(n+i)
\]

Finally, when \( i = N_R \):

\[
w'(n + N_R) = w'(n-1) + \delta S_{n-1}(n + N_R).
\]

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 \( T'_{NR} \) for \( N_R + 1 \) iterations with a delayed adaptation parallelized structure is:

\[
T'_{NR} = (N_R + 1)T_x + T_x + 2T_+ + T_d
\]

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

\[
N_R(T_x + 2T_+ + T_d).
\]

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.
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 \( t + 4 \)).

<table>
<thead>
<tr>
<th>CLK</th>
<th>R&lt;sub&gt;1&lt;/sub&gt;</th>
<th>R&lt;sub&gt;2&lt;/sub&gt;</th>
<th>R&lt;sub&gt;3&lt;/sub&gt;</th>
<th>R&lt;sub&gt;4&lt;/sub&gt;</th>
<th>R&lt;sub&gt;5&lt;/sub&gt;</th>
<th>R&lt;sub&gt;6&lt;/sub&gt;</th>
<th>R&lt;sub&gt;7&lt;/sub&gt;</th>
<th>R&lt;sub&gt;8&lt;/sub&gt;</th>
<th>R&lt;sub&gt;9&lt;/sub&gt;</th>
<th>R&lt;sub&gt;10&lt;/sub&gt;</th>
<th>R&lt;sub&gt;11&lt;/sub&gt;</th>
<th>R&lt;sub&gt;12&lt;/sub&gt;</th>
<th>R&lt;sub&gt;13&lt;/sub&gt;</th>
<th>R&lt;sub&gt;14&lt;/sub&gt;</th>
<th>R&lt;sub&gt;15&lt;/sub&gt;</th>
<th>R&lt;sub&gt;16&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>G</td>
<td>C</td>
<td>H</td>
<td>I</td>
<td>M</td>
<td>N</td>
<td>θ</td>
<td>Q</td>
<td>θ</td>
<td>O</td>
<td>w</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>C</td>
<td>K</td>
<td>I</td>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>C</td>
<td>F</td>
<td>P</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>A</td>
<td>G</td>
<td>C</td>
<td>H</td>
<td>I</td>
<td>M</td>
<td>N</td>
<td>θ</td>
<td>Q</td>
<td>θ</td>
<td>O</td>
<td>w</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>B</td>
<td>C</td>
<td>K</td>
<td>I</td>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>D</td>
<td>C</td>
<td>F</td>
<td>P</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>A</td>
<td>G</td>
<td>C</td>
<td>H</td>
<td>I</td>
<td>M</td>
<td>N</td>
<td>θ</td>
<td>Q</td>
<td>θ</td>
<td>O</td>
<td>w</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>B</td>
<td>C</td>
<td>K</td>
<td>I</td>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>D</td>
<td>C</td>
<td>F</td>
<td>P</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>G</td>
<td>C</td>
<td>H</td>
<td>I</td>
<td>M</td>
<td>N</td>
<td>θ</td>
<td>Q</td>
<td>θ</td>
<td>O</td>
<td>w</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>C</td>
<td>K</td>
<td>I</td>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>D</td>
<td>C</td>
<td>F</td>
<td>P</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>A</td>
<td>G</td>
<td>C</td>
<td>H</td>
<td>I</td>
<td>M</td>
<td>N</td>
<td>θ</td>
<td>Q</td>
<td>θ</td>
<td>O</td>
<td>w</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>B</td>
<td>C</td>
<td>K</td>
<td>I</td>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>D</td>
<td>C</td>
<td>F</td>
<td>P</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>A</td>
<td>G</td>
<td>C</td>
<td>H</td>
<td>I</td>
<td>M</td>
<td>N</td>
<td>θ</td>
<td>Q</td>
<td>θ</td>
<td>O</td>
<td>w</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>B</td>
<td>C</td>
<td>K</td>
<td>I</td>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>D</td>
<td>C</td>
<td>F</td>
<td>P</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>A</td>
<td>G</td>
<td>C</td>
<td>H</td>
<td>I</td>
<td>M</td>
<td>N</td>
<td>θ</td>
<td>Q</td>
<td>θ</td>
<td>O</td>
<td>w</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>B</td>
<td>C</td>
<td>K</td>
<td>I</td>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>D</td>
<td>C</td>
<td>F</td>
<td>P</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>A</td>
<td>G</td>
<td>C</td>
<td>H</td>
<td>I</td>
<td>M</td>
<td>N</td>
<td>θ</td>
<td>Q</td>
<td>θ</td>
<td>O</td>
<td>w</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>B</td>
<td>C</td>
<td>K</td>
<td>I</td>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
    z(n) & \rightarrow X \quad - \quad \int \quad + \quad t(n) \\
    w(n-1) & \quad e(n) = f(t(n) - w(n-1)z(n))
\end{align*}
\]

Fig. 3. Simplified error model.

\[
\begin{align*}
    w(n-1)z(n) & \quad \theta(n) \quad \theta(n)z(n) \quad \delta \quad w(n) \\
    T_x & \quad T_\theta \quad T_x \quad T_\delta \quad T_\theta
\end{align*}
\]

Fig. 4. Simplified error model.
Fig. 5. Calculation time for $N_R$ iterations.

Fig. 6. Illustration of adaptation with delay.

Fig. 7. Time comparison of operations with or without delay.
For each iteration of the training, we therefore randomly generate a vector of $N_{FFT}$ Gaussian random complex terms $\{x_i\} \in (0, N_{FFT} - 1)$. 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:

$$\{t_k\} \in (0, N_{FFT} - 1) = \text{FFT}\{\{x_i\} \in (0, N_{FFT} - 1)\}$$  \hspace{1cm} (24)

and use the terms of this Fourier Transform as the desired signal $\{t_1 - N_q\}$. 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 and the output vectors have respectively the size of $N_0 = 2 \times N_{FFT}$ and $N_q = 2 \times N_{FFT}$.

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 $\delta$. Finally, at each iteration, we took the squared error $E$ which we integrated over an exponential window with a forgetting factor $\lambda$ by means of the following equation:

$$e(t + 1) = \lambda e(t) + (1 - \lambda) \sum_{j=1}^{N_q} (z_j - t_j)^2.$$  \hspace{1cm} (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.

![Fig. 8. Illustration of convergence with or without delay.](image)

<table>
<thead>
<tr>
<th>Table 3. Complex to real mapping.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{z_0}{1-N_0} = \text{Real}{x_0-(N_{FFT}-1)}$</td>
</tr>
<tr>
<td>$\frac{z_0}{N_0+1-N_0} = \text{Imag}{x_0-(N_{FFT}-1)}$</td>
</tr>
<tr>
<td>$\frac{t_1-N_q}{T} = \text{Real}{x_0-(N_{FFT}-1)}$</td>
</tr>
<tr>
<td>$\frac{t_{N_q+1-N_q}}{T} = \text{Imag}{x_0-(N_{FFT}-1)}$</td>
</tr>
</tbody>
</table>

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 $\delta$ adaptation step size of the gradient equals to $5 \times 10^{-4}$. 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 $5 \times 10^{-4}$ and 20 million examples with an adaptation step.
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

\[ T_{NCL} \approx T_x + (\log_2 N_{L-1} + 2) T_+ + T_{exp} + T_/ \]  

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:

\[ T_{BP_q} \approx 2T_x + T_+ \]  

\[ T_{BP_L} \approx 2T_x + (\log_2 N_{L-1} + 1) T_+ \]

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:

\[ T_{UP} = 2T_+ + T_x \]
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.
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:

\[
T_{Lat} = \sum_{L=1}^{q} T_{NC_L} + T_{BP_q} + \sum_{L=1}^{q-1} T_{BP_L} + T_{UP}
\]  

(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
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 4. Main simulation parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_{FFT}$</td>
<td>16</td>
</tr>
<tr>
<td>$q$</td>
<td>2</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>0.9999</td>
</tr>
<tr>
<td>$\delta$</td>
<td>$10^{-4}$ or $5 \times 10^{-4}$</td>
</tr>
<tr>
<td>1st layer</td>
<td>sigmoid</td>
</tr>
<tr>
<td>2nd layer</td>
<td>linear</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Delay</th>
<th>Normalized mean squared error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$1.8575 \times 10^{-3}$</td>
</tr>
<tr>
<td>11</td>
<td>$1.8735 \times 10^{-3}$</td>
</tr>
<tr>
<td>21</td>
<td>$1.8909 \times 10^{-3}$</td>
</tr>
<tr>
<td>31</td>
<td>$1.9099 \times 10^{-3}$</td>
</tr>
<tr>
<td>41</td>
<td>$1.9301 \times 10^{-3}$</td>
</tr>
<tr>
<td>101</td>
<td>$2.1437 \times 10^{-3}$</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>Module</th>
<th>FP1</th>
<th>FP2</th>
<th>BP2</th>
<th>BP1</th>
<th>UP2</th>
<th>UP1</th>
<th>Delay1</th>
<th>Delay2</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td>195</td>
<td>119</td>
<td>102</td>
<td>125</td>
<td>98</td>
<td>98</td>
<td>541</td>
<td>221</td>
<td>639</td>
</tr>
</tbody>
</table>

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

Fig. 16. Training data flow.
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:

\[
\text{Delay 1} = FP_1 + BP_2 + BP_1 + FP_2. \quad (31)
\]

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

\[
\text{Delay 2} = FP_2 + BP_2. \quad (32)
\]

Hence the total Latency is:

\[
\text{Total Latency} = FP_1 + FP_2 + BP_2 + BP_1 + UP_1. \quad (33)
\]

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 backpropagation 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.
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


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