IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 12, 2019, accepted March 21, 2019, date of current version April 9, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2907261

Deep Neural Network Hardware Implementation Based on Stacked Sparse Autoencoder

MARIA G. F. COUTINHO[1], MATHEUS F. TORQUATO[2], AND MARCELO A. C. FERNANDES[1]
[1]Department of Computer Engineering and Automation, Federal University of Rio Grande do Norte (UFRN), Natal 59078-970, Brazil
[2]College of Engineering, Swansea University, Swansea SA2 8PP, U.K.

Corresponding author: Marcelo A. C. Fernandes (mfernandes@dca.urn.br)

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES)-Finance Code 001.

ABSTRACT Deep learning techniques have been gaining prominence in the research world in the past years; however, the deep learning algorithms have high computational cost, making them hard to be used to several commercial applications. On the other hand, new alternatives have been studied and some methodologies focusing on accelerating complex algorithms including those based on reconfigurable hardware has been showing significant results. Therefore, the objective of this paper is to propose a neural network hardware implementation to be used in deep learning applications. The implementation was developed on a field-programmable gate array (FPGA) and supports deep neural network (DNN) trained with the stacked sparse autoencoder (SSAE) technique. In order to allow DNNs with several inputs and layers on the FPGA, the systolic array technique was used in the entire architecture. Details regarding the designed implementation were evidenced, as well as the hardware area occupation and the processing time for two different implementations. The results showed that both implementations achieved high throughput enabling deep learning techniques to be applied for problems with large data amounts.

INDEX TERMS Deep learning, stacked sparse autoencoder, FPGA, systolic array.

I. INTRODUCTION

The use of Artificial Intelligence (AI) techniques for solving problems in several areas, such as Deep Learning (DL), also called Deep Neural Networks (DNN), have been gaining a great deal of attention in recent years. The DNNs are able to prove high computational power, concomitantly with the use of several hidden layers. Among the various Deep Learning techniques found in the literature, those ones based on Autoencoders (AEs) are mainly applied to prediction and classification problems [1]–[3].

The methodology of using multiple stacked autoencoders, forming a single DNN has proved to be effective in the training of deep learning networks, as can be seen in [4], [5]. Stacked Sparse Autoencoders (SSAE) have been used in classification problems as can be seen in [5]. In this approach, each hidden layer is composed of an individually trained sparse autoencoder, in an unsupervised way. The output of each hidden layer of each AE is used as input to the next AE, so that the input data characteristics are propagated through the network layer by layer, enabling the output layer to perform the data classification, after a supervised training.

However, deep neural networks have a high computational complexity due to the large number of hidden layers, which makes it difficult or impossible to apply this approach in several commercial applications. This problem is further aggravated by applying this technique in problem with large data amounts. With that in mind, several researches aiming the acceleration of complex algorithms have been conducted, and among them the use of reconfigurable computing have proved to be a great option.

The present paper is organized as follows: This first section presented a general introduction about the work explaining the motivation behind it. Section II discusses some related works and the state of the art. In Section III will be presented a theoretical foundation regarding the deep learning techniques, evidencing the Stacked Sparse Autoencoder. Section IV presents the hardware architecture details for the feedforward phase of two implementation. Section V will present the results of the proposed hardware validation and synthesis results, as well as comparisons with a state-of-the-art work. Finally, Section VI will present the final considerations regarding the obtained results.

The associate editor coordinating the review of this manuscript and approving it for publication was Fan Zhang.

II. RELATED WORKS

It is possible to find in the literature a great variety of works containing hardware implementations of artificial intelligence algorithms. As in the works developed in [6]–[9], in which it was possible to achieve high speed gains with the use of Field Programmable Gate Array (FPGA) reconfigurable computing, comparing to implementations using graphics processing units (GPUs) and general purpose processors used in High Performance Computers (HPC).

In [6] a hardware Radial Base Functios (RBF) Artificial Neural Network (ANN) was implemented and trained with the Least Mean Square (LMS) algorithm. The architecture implemented in Register-transfer level (RTL) was analyzed in terms of hardware occupancy rate, bit resolution and processing delay. The synthesis results for two distinct scenarios and several fixed point resolutions suggest the possibility of using the design in more complex practical situations. In the work of [7]–[9] a FPGA parallel implementation of genetic algorithm was proposed. The architecture was also implemented in RTL and performed tests with different parameters. The functions used by the genetic algorithm were stored in Lookup Tables (LUTs), thus eliminating the need for a dedicated circuit to perform each function and, consequently, enabling the reduction of area occupied in the FPGA, making the implementation more flexible. That approach resulted in a throughput increase, since building a complex circuit for these functions would produce longer hardware critical paths.

Some work involving implementations of DL techniques in FPGA motivated the development of this work. In the work of [10] it was proposed the implementation of a Convolutional Neural Network (CNN) in FPGA using fixed point. The circuit was built using the Vivado HLS tool. The hardware implementation, using a FPGA Virtex 7, managed to operate approximately 16.42 times faster than a CNN implemented in Matlab. Another proposal for the implementation of a CNN in FPGA is presented in [11] which also used high-level synthesis (HLS). Experiments showed that the FPGA implementation achieved up to 3× more energy efficiency than a CPU implementation, and a energy efficiency equivalent to the same CPU implementation with 16 threads operating in parallel. In addition, in respect to the implementation in SoC GPU for mobile applications, it was possible to obtain a gain of almost 15× in terms of speed and 16× in terms of energy efficiency. In [12] is proposed an optimized implementation to accelerate the CNNs on FPGA. For this, an analytical design scheme was adopted using the roofline model. However, the hardware implementation was developed using HLS. The CNN implemented had as target a Virtex 7 FPGA, and it obtained a speedup of 17.42× over the Intel Xeon 2.2 GHz Microprocessor.

The work [13] presented the effects of using data representation with limited precision in neural network training. The results showed that using stochastic rounding with a resolution of 16 bits fixed-point, it was possible to obtain a similar performance to the 32 bits in floating point. The work used the systolic matrix strategy to build multiplier arrays, thus contributing to the implementation throughput increase. Aiming the use of reconfigurable computing, [14] proposed a deep learning accelerator unit on FPGA, called DLAU. The proposed architecture was split into three processing units using a pipeline scheme and it can be applied to different network topologies. The hardware was developed using the Xilinx Zynq Zedboard development platform which features an ARM Cortex-A9 processor. Experiments showed that the DLAU achieved a speedup of 36.1× over Intel Core 2 2.3 GHz processors, in addition to achieving low power consumption compared to implementing an NVIDIA Tesla K40c GPU.

The systolic array has been studied for designing deep learning accelerators in recent years, for to reduce the area and power of the large matrix multiply unit [15], as also, to achieve high throughput [16]. The systolic array can provide low global data transfer and high clock frequency which is suitable for large-scale parallel design on FPGAs [16].

In [17] a DNN implementation in FPGA was proposed using VHSIC Hardware Description Language (VHDL) and floating point which enabled the use of a single layer of physical computation to execute the entire network feed-forward step. Synthesis were performed for several network architectures and the largest ones supported by each hardware used in the experiments were identified. The Xilinx Virtex-5 XC5VLX-110T FPGA was able to fit the $784-40-40-40-10$ architecture, whereas the Xilinx ZynQ-7000 XC7Z045 FPGA could fit the $784-126-126-126-10$ architecture. However, it was observed that with the $784-40-40-10$ and $784-126-126-10$ architectures, the best results were obtained in terms of recognition rate and processing time, reaching the maximum performance of 15810 and 15900 manuscript digits frames (from the MNIST database) per second, respectively. In the work of [18] an RTL compiler was proposed to accelerate FPGA Deep Learning algorithms that aimed to achieve similar performance to the implementations in RTL, since implementations that used high-level synthesis could not optimize the FPGA resources consumption.

Using Verilog, [19] implemented a Convolutional autoencoder (CAE) network in FPGA, which represents a type of CNN. The FPGA implementation was validated by running image compression algorithms. Comparisons with other hardware implementation such as CPU and GPU implementations were carried out. The speed comparison pointed that the FPGA was faster than the CPU and slower than the GPU. However, with regard to performance per watt the FPGA scored significantly higher than both others.

One of the first DNN implementations in FPGA using Stacked Sparse Autoencoders is presented in [5]. DNN was used to perform image classification of the data set called CIFAR-10. The network architecture used two hidden layers, containing 2000 neurons in the first hidden layer and 750 in the second one, in addition to 3072 inputs and 10 neurons in the output layer, called $3072-2000-750-10$ architecture. For the implementation, a high-level synthesis OpenCL

framework was applied, which might have contributed for the FPGA results being inferior in processing efficiency when compared to the GPU implementation. In [20], the advantages of using fixed point in autoencoder implementations are shown, considering that using a 4 bits resolution in the integer part, it was possible to obtain the same accuracy of using floating point. For the experiments, the $784-400-10$ network architecture and the MNIST database were used. However, no information was presented regarding the FPGA hardware resources occupancy rate and processing time.

In [21]–[23], proposals for traditional neural networks using autoencoders were presented. In [21] a sparse autoencoder architecture was implemented in VerilogHDL. The tests were carried out with a $196-100-196$ network architecture using the natural image data set of the Kyoto city in Japan. In [22], it was proposed, in addition to the traditional autoencoders architectures, a structure with two chained autoencoders, the $4-2-1-2-4$ architecture, however, it aimed to rebuild the input in the output, as in the conventional models. In this work, unlike the ones previously mentioned which implemented autoencoders techniques as well, the circuit was implemented in RTL, which allowed it to achieve superior efficiency comparing to the other autoencoder implementations mentioned here. The target FPGA was a Xilinx Virtex-6 xc6vlx240t.

Similar to the work structure showed in [22], in [23], autoencoders were used to implement the $32-16-8-16-32$ architecture. However, in this work the Xilinx Vivado HLS tool was used to perform the synthesis from the C language to RTL. A Xilinx Kintex-7 xc7k410tffg900-2 FPGA was used for the experiments. The implementation was able to achieve high throughput (about 200 Mega samples per second (MSPS)) and low latency (about 105 ns). However, it was observed that most of the FPGA hardware resources were occupied, which makes impracticable to design a network architecture larger than the one implemented.

In order to guarantee the FPGA implementation efficiency, through the optimum usage of resources, the implementation proposed in the present work was constructed in RTL, unlike [5]. The RTL implementation was also adopted in [22], however, in that work it was implemented autoencoders structures that were more similar to traditional ones than to DNNs. Thus, the present paper presents a hardware implementation proposal of a Deep Neural Network based on the Stacked Sparse Autoencoder technique. The hardware was developed for the feedforward phase adopting the systolic array technique, which allowed the use of multiple neurons and several layers. Data regarding hardware occupancy rate and processing time will be presented for a Virtex 6 XC6VLX240T-1FF1156 FPGA.

III. DEEP LEARNING

In the last years, several Deep Learning techniques have become the object of research of many academic works around the world. These techniques can be defined as a modernization of Multilayer Perceptron (MLP) networks, considering that one of the main differences between MLP networks and DNNs is the feasibility of the DNNs to train networks with numerous hidden layers, which is a major problem in conventional MLPs.

The term autoencoder precedes the Deep Learning advent. In its traditional structure, autoencoders were mainly applied to dimensionality reduction and patterns learning problems. In these networks the training occurs so that the output layer provides a reconstruction of the input layer, thus, both layers have the same size. At the same time, only one hidden layer is used and performs the characteristics extraction of the input data. Consequently, the autoencoder architecture is composed of three layers: an input, a hidden and an output. The input and hidden layers form the network encoder while the hidden and output layers make up the decoder [24]. The autoencoders have been used in many unsupervised learning problems where it is possible to represent a P-dimensional input vector (from input layer) in an M-dimensional vector (from output hidden layer), where $M < P$. In other words, the autoencoders can reduce the dimensionality of the input by extracting the essential information (hidden layer output). With autoencoders, it is also possible to recover the input information using the neurons of the output layer. Basically, the neurons of the hidden layer reduce the input information and the neurons of the output layer recover that information [25].

A. STACKED SPARSE AUTOENCODER (SSAE)

With the Deep Learning ascent, the autoencoders began to be used in a chained fashion, forming networks with numerous hidden layers, whose training was previously performed in each autoencoder in an unsupervised way. Among the autoencoder types the sparse are known for being heavily applied to classification problems. In the stacked sparse autoencoders, each hidden layer is composed of the hidden layer of an individually trained sparse autoencoder. Each sparse autoencoder receives as input the output of the hidden layer of the previous sparse autoencoder so that the characteristics of the input data are extracted along the network hidden layers, enabling the output layer to perform the classification after the supervised training. In this work, the network training was performed using this methodology. Figure 1 presents the proposed SSAE architecture with an input layer (P data inputs), two hidden layers (M and N inputs) and an output layer (H outputs). This architecture can be represented as a $P-M-N-H$ SSAE.

This work is focused on implementing the SSAE feedforward phase, in which the equation that defines the output of the i-th neuron from the k-th layer, $z_i^k(n)$, at the n-th instant, can be expressed as

$$z_i^k(n) = \sum_{j=1}^{U^l} w_{ij}^k(n) \times y_j^l(n) + wb_i^k(n) \times b \qquad (1)$$

where $w_{ij}^k(n)$ is the weight associated with the j-th input of the i-th neuron in the k-th layer at the n-th instant, $y_j^l(n)$ is the j-th input of the l-th layer, in which $l = k-1$, at the

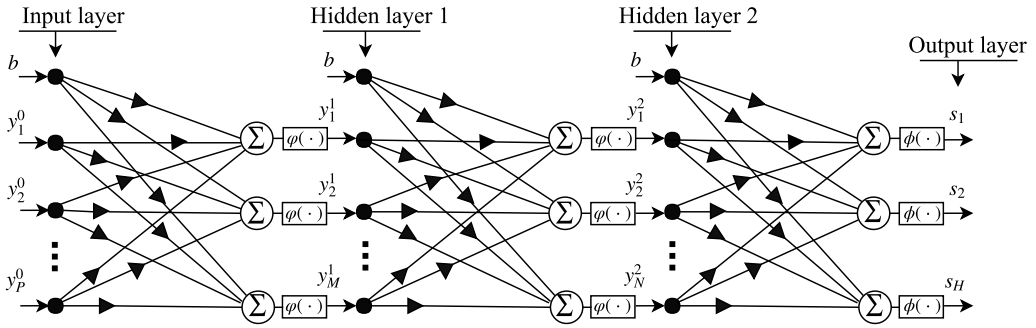

FIGURE 1. Proposed stacked sparse autoencoder architecture.

n-th instant, $wb_i^k(n)$ is the bias weight of the i-th neuron from the k-th layer at the n-th instant, b is the bias, which has a value of 1, and U^l is the number of inputs of the l-th layer, where $U^0 = P$, $U^1 = M$ e $U^2 = N$. In the hidden layers the sigmoid activation function was used, so the output associated with the i-th neuron from the k-th layer at the n-th instant, $v_i^k(n)$, can be expressed as

$$v_i^k(n) = \left(\frac{1}{1 + e^{-z_i^k(n)}} \right) \tag{2}$$

in which $v_i^k(n)$ will be the value of the j-th input to be used in the next layer at the n-th instant, $y_j^{l+1}(n)$, that is

$$y_j^{l+1}(n) = v_i^k(n) \tag{3}$$

in which $j = i$.

The softmax activation function was used in the output layer. This function has been adopted in neural classification networks as presented in [5], and can be characterized as

$$s_i(n) = \frac{e^{z_i^K(n)}}{\sum_{h=1}^{H} e^{z_h^K(n)}} \tag{4}$$

in which $s_i(n)$ consists of the i-th output of the last layer K with H neurons at the n-th instant. The H value is determined by the number of classes of the problem, since this function indicates the probability that each data belongs to a specific class.

IV. IMPLEMENTATION DESCRIPTION

In this work, two SSAE proposals were implemented for the feedforward phase using the systolic array technique. The main difference between them is in the way the network's synaptic weights are inserted into the hardware. In the following sections characteristics of each proposal will be present, as well as the details regarding the processing time of each implementation.

A. PROPOSAL 1

Based on the network structure detailed in Figure 1, the overall architecture of the first implementation proposed in this work is presented in Figure 2. This architecture is able to

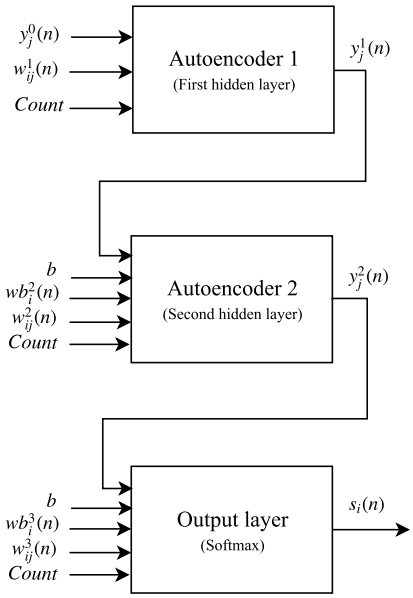

FIGURE 2. General architecture of proposal 1.

receive the network weights through weights streams, without the need of using memory resources to store them, unlike most implementations found in the literature [5], [22], [23]. The variables and constants of the implementation are represented in fixed point and for each j-th entry, $y_j^0(n)$, 1 bit is used in the integer part (without signal) and 12 bits in the decimal part, as the inputs are normalized between 0 and 1. For the synaptic weights of neurons from hidden layers, one and two, expressed as $w_{ij}^1(n)$ and $w_{ij}^2(n)$, as well as for the *bias* weights of all layers, $wb_i^k(n)$, 5 bits are used in the integer part (one bit for signal) and 12 bits in the decimal part. For the weights from the output layer neurons, $w_{ij}^3(n)$, 7 bits are used in the integer part (one bit for signal) and 12 bits in the decimal part.

For this proposal implementation, the systolic array technique was used and it acts as an intermediary approach between a completely parallel and fully serial methodology. This technique allows data to be received serially and the processing elements (PEs) perform their operations in parallel [26].

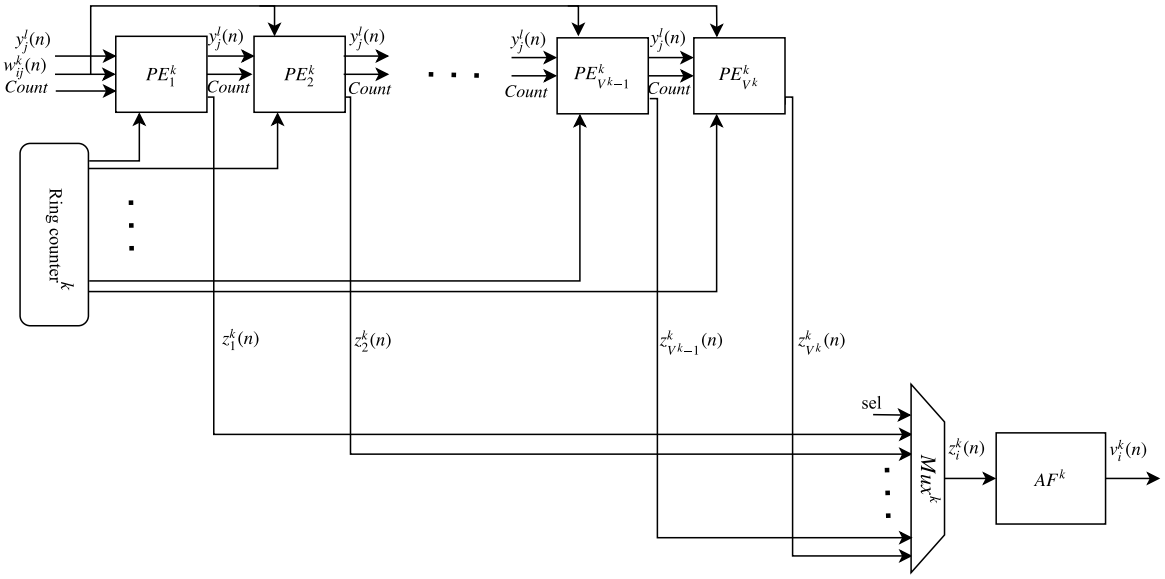

FIGURE 3. Architecture of the SSAE k-th hidden layer from proposal 1.

The network implemented in this work used two hidden layers, consisting of the 784-100-50-10 architecture. However, to add more hidden layers to the network it is necessary to replicate the inner block present in Figure 2, which represents the second hidden layer of the network. Another small adjust is necessary in order to set each layer with the amount of desired neurons.

1) SSAE LAYERS

The architecture of the k-th hidden layer, called k autoencoder in Figure 2, is presented in Figure 3 and it implements the systolic array technique. Each neuron of the k-th layer is represented by a i-th PE_i^k and the amount of PEs is defined by the value of V^k, where $V^1 = M$, $V^2 = N$ and $V^3 = H$. Thus, each i-th PE_i^k implements the Equation 1. The values computed in each i-th PE_i^k from the k-th hidden layer pass through an activation function, called here AF^k (see Equation 2), and are used as input to the next layer (see Equation 3).

The values computed in the output layer PEs go through the activation function defined in Equation 4 in order to generate the network output. The *sel* signal corresponds to the multiplexer selector, Mux^k, used to select the outputs of each i-th PE_i^k for the k-th layer input activation function, AF^k.

Through the systolic array, the inputs values flow between the PEs so that each PE starts its operations at the instant following the beginning of the operations of its preceding PE, causing these modules to operate in parallel. It is worth mentioning that it is from the second hidden layer, that is, when $k > 1$, that the *bias*, b, and the *bias* weights, $wb_i^k(n)$, are inserted as input of the k-th layer, as per the structure shown in Figure 2. Only in the first hidden layer, that is, when $k = 1$, these values are inserted together with the network inputs and the weights of the neurons layer, respectively.

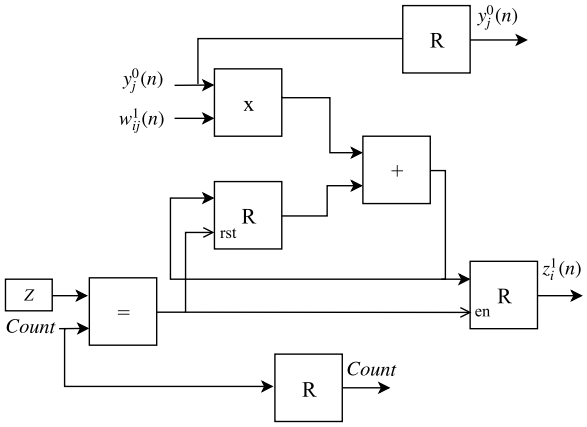

FIGURE 4. i-th PE_i^1 architecture from proposal 1.

An advantage of this first proposal is the insertion of the weights from each layer in a serial way through a single weight stream in the first hidden layer and two weights streams in the other network layers. This would be the least amount of weight streams required for the operation of this implementation, however, this architecture is easily scalable, which allow the weight streams parallelization for each layer. The amount of weight streams per layer may range from one to the amount of neurons in the layer, V^1, in the first network hidden layer, and from 2 up to $2 \times V^k$, in the others network layers, due to *bias* weights.

It is worth mentioning that the possibility of receiving the weights by streams allows the use of the same hardware for different problems, since the weights trained for a different problem can be inserted in the network, as well as the inputs of the problem in question, without the need to reconfigure the hardware.

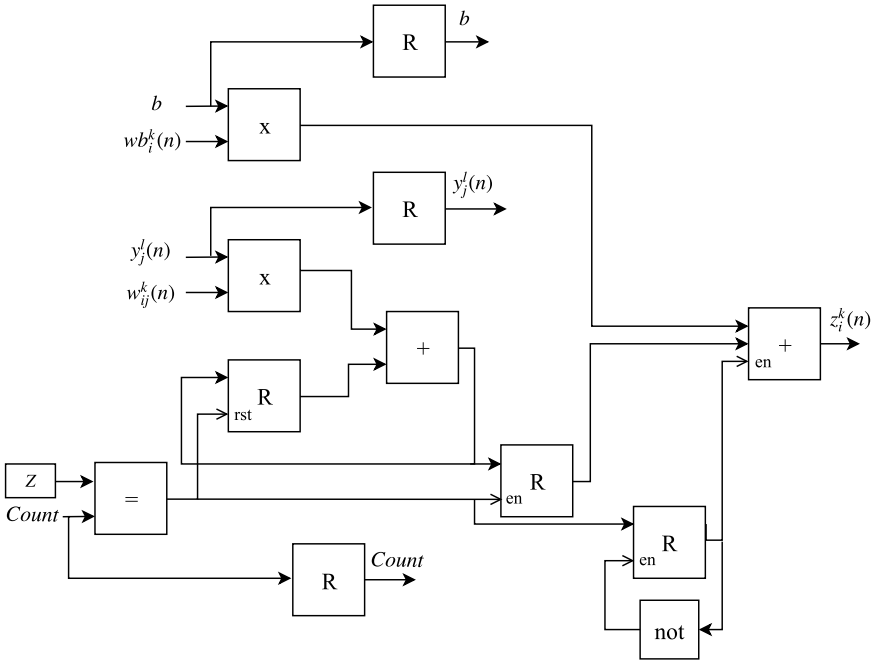

FIGURE 5. *i*-th PE_i^k architecture, for $k > 1$, from proposal 1.

2) PROCESSING ELEMENTS (PEs)

The architecture of the first hidden layer PEs differs from the PE architecture of the other layers. In the first hidden layer, the *bias*, b, and the *bias* weights, $wb_i^1(n)$, are inserted along with the network inputs and the neurons synaptic weights in the layer, respectively. Thus, the architecture of each i-th PE_i^1 from proposal 1 is detailed in Figure 4 and it's formed by a multiplier, an adder and four registers, R. The architecture of each i-th PE_i^k, for $k > 1$, from proposal 1, is detailed in Figure 5 and it's formed by two multipliers, two adders and six registers, R. Each i-th PE_i^k generates an output after Z samples, where for the first hidden layer, $Z = P$ and for the others, the Z variable must be greater than or equal to the number of entries ($Z \geq P$) and multiple of the number of neurons of the first hidden layer ($\text{mod}(Z, M) = 0$).

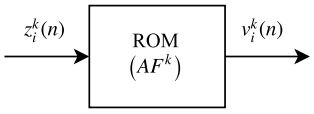

FIGURE 6. *k*-th activation function, AF^k, associated with the hidden layers.

3) ACTIVATION FUNCTIONS (AFs)

The implementation of each k-th activation function, AF^k, was designed using Lookup Tables (LUTs) as shown in Figure 6, which allows the approximation of functions through a L-values table. In order to implement the activation function of each hidden layer (Equation 2), ROM memories with 16 bits of depth were used, in which $L = 2^{16}$, storing words of 13 bits. For the implementation of the activation function from the output layer, the softmax, defined by Equation 4, a LUT was used to approximate the exponential function, only then the Equation 4 division was performed. The output layer LUT, AF^3, was set to a depth of $L = 2^{16}$, storing 57 bits.

An important fact associated with this implementation strategy is the use of a single activation function only, or LUT, per layer. This feature significantly reduces the space occupied in the FPGA (see [6]).

4) PROCESSING TIME

In proposal 1, a ring counter in each k-th layer, as shown in Figure 3, is used to enable receipt of the weights in each i-th PE_i^k from the k-th layer. Since the least amount of weight streams per layer is being used, the operation of the k-th layer's ring counter must be V^k times faster than the operation of each PE_i^k. In order to maintain the implementation timing, $V^k > V^{k+1}$ is considered. Thus, the execution time of each PE circuit is determined by the ring counter run-time of the first hidden layer, t_{rc}. Therefore, the PE time, t_{PE}, can be defined as

$$t_{PE} = (V^1 \times t_{rc}) \tag{5}$$

where $(V^1 \times t_{rc}) \geq t_c$ and t_c is the system critical path time.

The proposed architecture has an initial delay that can be expressed as

$$d = (Q \times K + D) \times t_{PE} \tag{6}$$

in which Q represents the initial delay of the first layer, and it is a number greater than or equal to the number of neural network inputs ($Q \geq P$) and a multiple of the number of

neurons from the first hidden layer (mod$(Q, M) = 0$). As the first layer is greater than other layers ($V^1 > V^2 > \ldots > V^{K-1} > V^K$), the variable Q can be used to estimate the delay for all layers. K is the number of network layers with neurons and, D is the delay in number of samples which is a result of the hidden and output layers activation functions. The neural network throughput, th_{ff}, in frames per second (FPS) or in inputs per second (IPS) can be expressed as

$$th_{ff} = \frac{1}{Q \times t_{PE}}. \tag{7}$$

It is important to mention that the throughput achieved is independent from the number of layers of the network, as it only depends on the amount of inputs and neurons used in the first hidden layer. The amount of layers with neurons, K, will only impact in the system initial delay.

Based on Equation 7, the execution time of the proposed SSAE, called here as feedforward time, t_{ff}, after the initial delay of d seconds (Equation 6), can be expressed as

$$t_{ff} = Q \times t_{PE} = \frac{1}{th_{ff}} \tag{8}$$

thus, in every t_{ff}, it is possible to obtain the output of all the H neurons from the last layer, that is, the network output for a given input.

B. PROPOSAL 2

The general architecture of the second proposal is presented in Figure 7 and it is based on the structure shown in Figure 1. The main differences regarding the first proposal are related to the way in which the synaptic weights are inserted into the network. In this second proposal, the weight values are stored in ROM memories and they are no longer received serially as in the first implementation. The project variables and constants are represented in fixed point, using the bit resolutions presented in Subsection IV-A, for proposal 1.

1) SSAE LAYER

The Figure 3 presents the general architecture of the k-th hidden layer, called k autoencoder (Figure 2). In this second proposal, there are no streams for the synaptic weights, since these values are stored in LUTs using ROM memories. This is possible since the network training has been previously finished, thus there is no need to change the weights once the network has already been trained for a specific problem. However, in the need to use the network for a new problem, the hardware must be reconfigured, storing the new weights, obtained after the previous network training for the new problem in the lookup tables.

2) PROCESSING ELEMENTS (PEs)

In this proposal, the PEs architecture from the first hidden layer differs from the PE architecture from the other layers. The architecture of each i-th PE_i^1 of proposal 2 is detailed in Figure 9 and consists of a multiplier, an adder, four registers, R, and a LUT used to store the weights of the i-th PE_i^1, called W_i^1. The architecture of each i-th PE_i^k, for $k > 1$

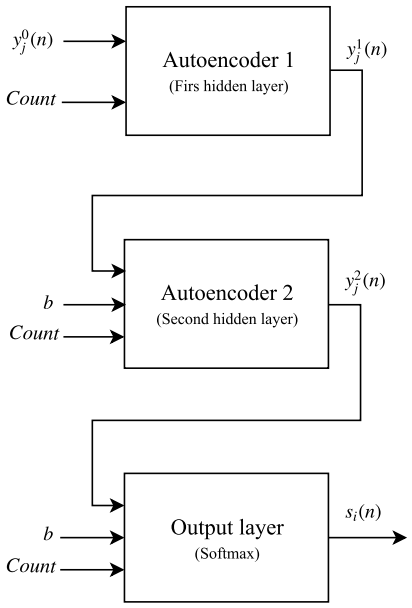

FIGURE 7. General architecture of proposal 2.

from proposal 2, is detailed in Figure 10 and consists of two multipliers, two adders, six registers, R, one LUT used to store the weights of the i-th PE_i^k, called the W_i^k, and a constant for the *bias* weight, wb_i^k. Each i-th PE_i^k generates an output after Z samples, as shown in Subsection IV-A2 from the proposal 1. The LUTs of each PE_i^k and W_i^k, use a ROM memory with a depth of $L = Z$, storing words of 17 bits in the hidden layers, and 19 bits in the output layer. In addition, the constants used for the *bias* weight of each i-th neuron, wb_i^k, were configured with 5 bits in the integer part (using one for signal) and 12 bits in the decimal part.

3) ACTIVATION FUNCTIONS (AFs)

The implementation of each k-th activation function, AF^k, from the hidden layers as well as the activation function of the output layer of proposal 2 corresponds to what was already presented for proposal 1 in Subsection IV-A3.

4) PROCESSING TIME

In this second proposal, the PE execution time corresponds to the system critical path time, t_c, i.e.

$$t_{PE} = t_c. \tag{9}$$

However, the other equations from subsection IV-A4 can be considered for the processing time calculation from proposal 2, taking into account the Equation 9. The initial delay, d, can be defined by Equation 6. In addition, the throughput, th_{ff}, can be expressed by Equation 7, as well as the execution time of the SSAE, t_{ff}, can be expressed by Equation 8 .

V. RESULTS

A. HARDWARE OCCUPATION ANALYSIS

This subsection will present the hardware occupation analysis associated with the PEs on FPGA. For all synthesis results,

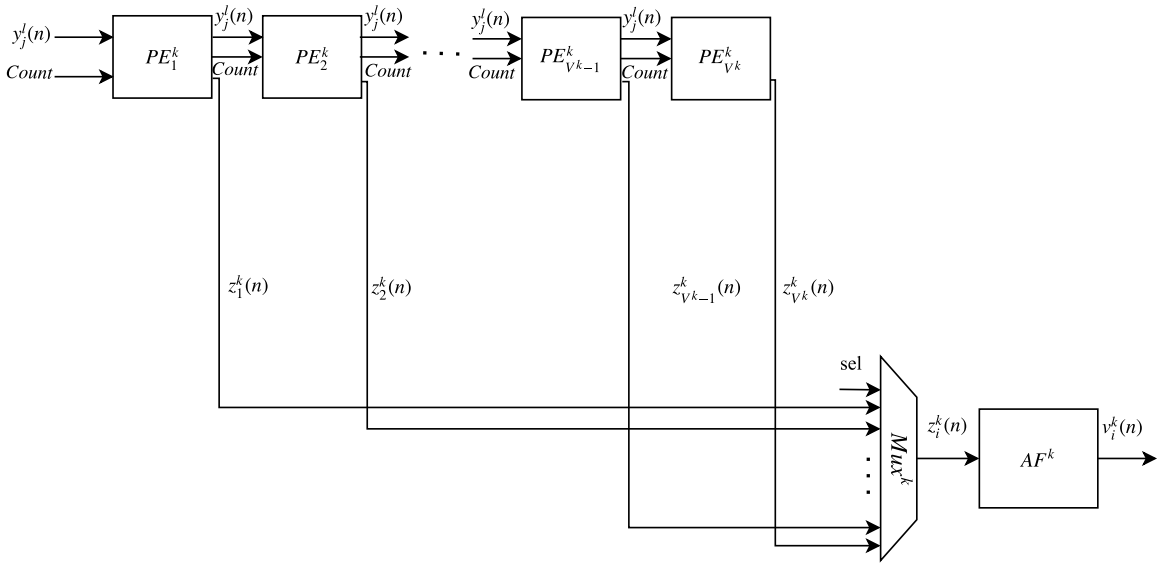

FIGURE 8. Architecture of the SSAE k-th hidden layer from proposal 2.

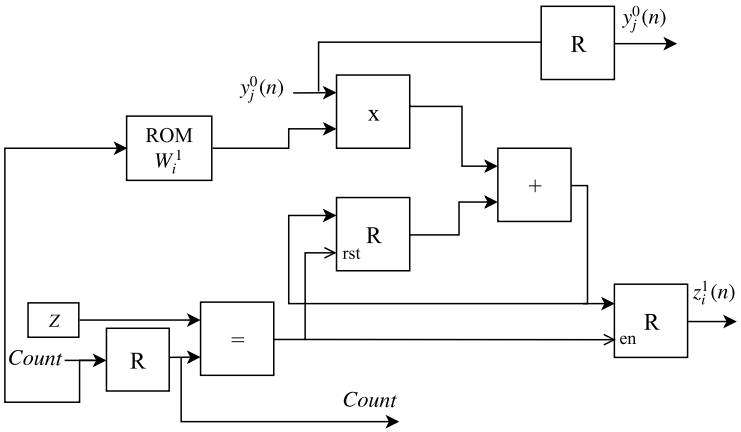

FIGURE 9. Architecture of the i-th PE_i^1 from proposal 2.

TABLE 1. Hardware occupation (logic cells and multipliers) per PE for proposal 1.

Layer	n_{PE}	n_{LC}	n_{Mult}
First layer (n_{PE}^1)	1	141 (0.06%)	1 (0.13%)
	5	853 (0.35%)	5 (0.65%)
	10	1664 (0.69%)	10 (1.30%)
	25	4200 (1.74%)	25 (3.26%)
	50	9397 (3.90%)	50 (6.51%)
	100	22162 (9.19%)	100 (13.02%)
Other layers ($n_{PE}^{k\geq 2}$)	1	269 (0.11%)	2 (0.26%)
	2	599 (0.25%)	4 (0.52%)
	5	1349 (0.56%)	10 (1.30%)
	10	2610 (1.08%)	20 (2.60%)
	25	6511 (2.70%)	50 (6.51%)
	50	14621 (6.06%)	100 (13.02%)

the Virtex 6 XC6VLX240T-1FF1156 target FPGA was used. Table 1 shows the hardware occupation (logic cells and multipliers) per PE (n_{PE}) for proposal 1. As each PE_i^k represents the i-th neuron in the k-th layer, table 1 shows the occupation per neuron.

Figures 11 and 12 show the linear regression curve for the first layer ($k = 1$) and other layers ($k \geq 2$), respectively. The measurement points were obtained from Table 1, and the equation associated with the regression analysis for the first layer can be expressed as

$$n_{LC}^1 = \left\lfloor 212.2 n_{PE}^1 \right\rfloor \tag{10}$$

where n_{LC}^1 and n_{PE}^1 are the number of logic cells and PEs in the first layer, respectively.

For other layers the equation is

$$n_{LC}^{k\geq 2} = \left\lfloor 285.1 n_{PE}^{k\geq 2} \right\rfloor \tag{11}$$

where $n_{LC}^{k\geq 2}$ and $n_{PE}^{k\geq 2}$ are the number of logic cells and PEs in the other layers, respectively. The total number of PEs, n_{PE}

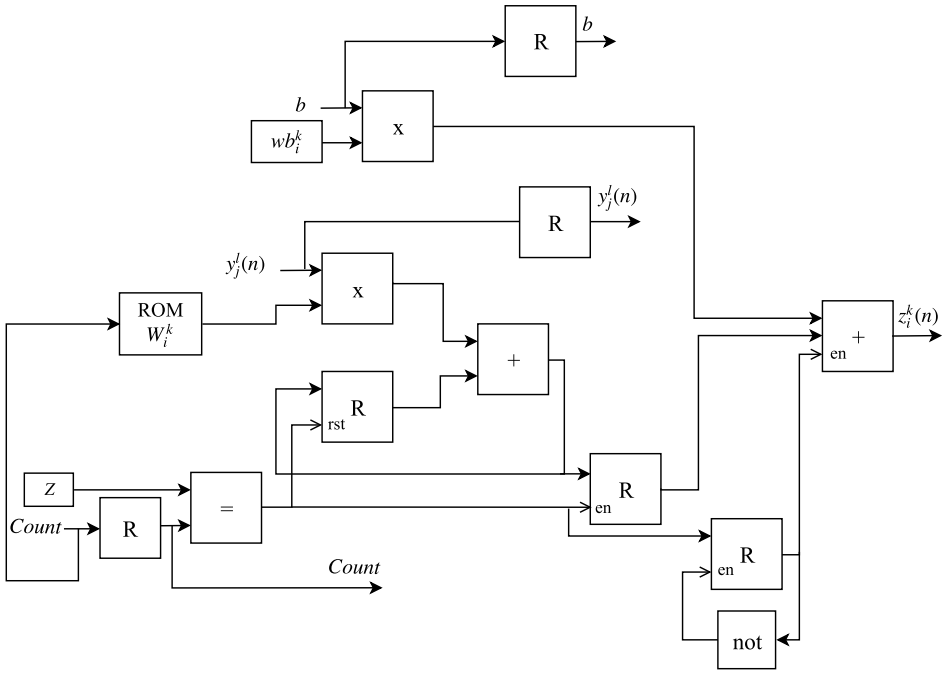

FIGURE 10. Architecture of the i-th PE_i^k, for $k > 1$, from proposal 2.

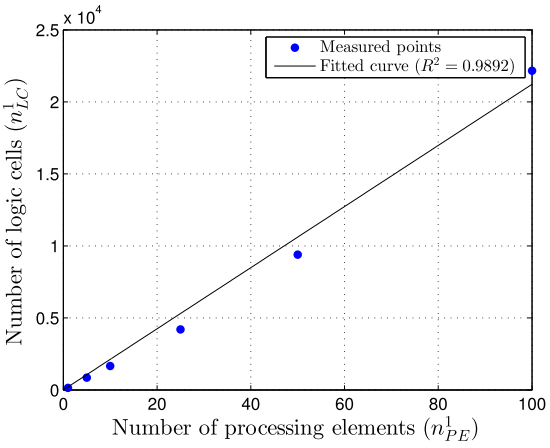

FIGURE 11. Linear regression curve for the hardware occupation (logic cells) per PE for proposal 1 in the first layer, n_{LC}^1.

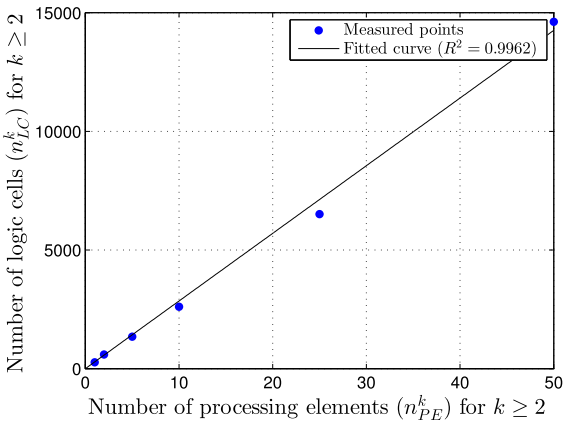

FIGURE 12. Linear regression curve for the hardware occupation (logic cells) per PE for proposal 1 in the other layers, n_{LC}^k ($k \geq 2$).

can be expressed as

$$n_{PE} = n_{PE}^1 + n_{PE}^{k\geq 2} = n_{PE}^1 + \sum_{k=2}^{K} n_{PE}^k = V^1 + \sum_{k=2}^{K} V^k \quad (12)$$

where n_{PE}^k is the number of PEs in the k-th layer and the total of logic cells, n_{LC}, is expressed as

$$n_{LC} = n_{LC}^1 + n_{LC}^{k\geq 2} = n_{LC}^1 + \sum_{k=2}^{K} n_{LC}^k. \quad (13)$$

The regression analysis is not necessary for the multipliers, and the equations can be expressed as

$$n_{Mult}^1 = n_{PE}^1, \quad (14)$$
$$n_{Mult}^{k\geq 2} = 2n_{PE}^{k\geq 2}, \quad (15)$$

and

$$n_{Mult} = n_{Mult}^1 + n_{Mult}^{k\geq 2} = n_{Mult}^1 + 2\sum_{k=2}^{K} n_{Mult}^k \quad (16)$$

where the n_{Mult}^k is the number of multipliers in the k-th layer.

The hardware occupation results concerning proposal 2 are presented in Table 2, and the regression analysis for the

TABLE 2. Hardware occupation (logic cells and multipliers) per PE for proposal 2.

Layer	n_{PE}	n_{LC}	n_{Mult}
First layer (n^1_{PE})	1	141 (0.06%)	1 (0.13%)
	5	746 (0.31%)	5 (0.65%)
	10	1384 (0.57%)	10 (1.30%)
	25	3375 (1.40%)	25 (3.26%)
	50	6682 (2.77%)	50 (6.51%)
	100	13432 (5.57%)	100 (13.02%)
Other layers ($n^{k\geq 2}_{PE}$)	1	229 (0.09%)	2 (0.26%)
	2	570 (0.24%)	4 (0.52%)
	5	1151 (0.48%)	10 (1.30%)
	10	2056 (0.85%)	20 (2.60%)
	25	4860 (2.02%)	50 (6.51%)
	50	9735 (4.04%)	100 (13.02%)

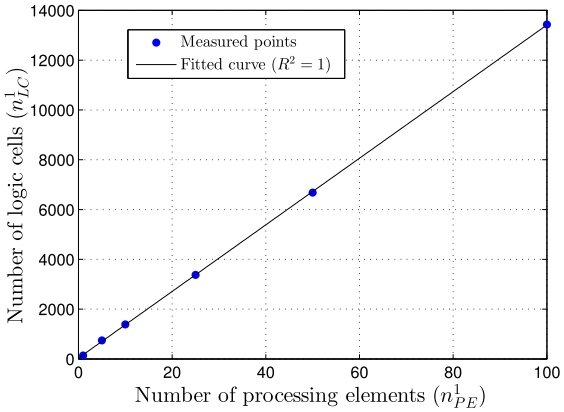

FIGURE 13. Linear regression curve for the hardware occupation (logic cells) per PE for proposal 2 in the first layer, n^1_{LC}.

first and others layers are illustrated in Figures 11 and 12, respectively. The regression analysis equations for logic cells can be expressed as

$$n^1_{LC} = \left\lfloor 133.8 n^1_{PE} + 34.17 \right\rfloor \tag{17}$$

and

$$n^{k\geq 2}_{LC} = \left\lfloor 191.7 n^{k\geq 2}_{PE} + 129.1 \right\rfloor. \tag{18}$$

The multipliers occupation can be described by Equations 14, 15 and 16.

It is essential to observe that in Equations 10, 11, 17 and 18, it was considered that the maximum number of DSP multipliers was sufficient for all PEs ($n^{max}_{Mult} \geq n_{Mult}$); otherwise ($n_{Mult} > n^{max}_{Mult}$), the Equations 10 and 11 could be rewritten as

$$n^1_{LC} = \left\lfloor (212.2 + N_{LCM})\alpha^1 + 212.2 n^{1_{max}}_{Mult} \right\rfloor \tag{19}$$

and

$$n^{k\geq 2}_{LC} = \left\lfloor (285.1 + 2N_{LCM})\alpha^{k\geq 2} + 285.1 \frac{n^{k\geq 2_{max}}_{Mult}}{2} \right\rfloor \tag{20}$$

where

$$\alpha^1 = n^1_{PE} - n^{1_{max}}_{Mult} \tag{21}$$

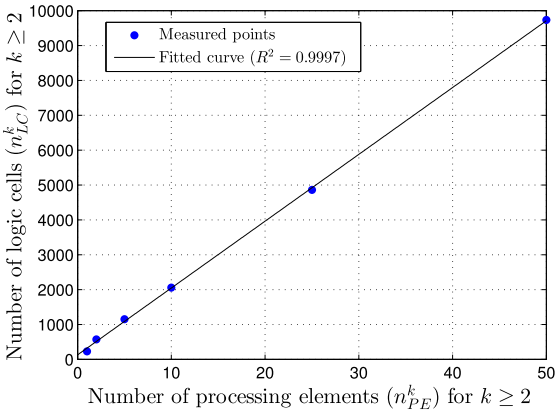

FIGURE 14. Linear regression curve for the hardware occupation (logic cells) per PE for proposal 2 in the other layers, n^k_{LC} ($k \geq 2$).

and

$$\alpha^{k\geq 2} = n^{k\geq 2}_{PE} - \frac{n^{k\geq 2_{max}}_{Mult}}{2} \tag{22}$$

where the N_{LCM} represents the number of logic cells needed to build a multiplier. The $n^{1_{max}}_{Mult}$ and $n^{k\geq 2_{max}}_{Mult}$ are the maximum numbers of the DSP multiplier used in first and other layers, respectively.

Following the same idea, Equations 17 and 18 can be rewritten as

$$n^1_{LC} = \left\lfloor (133.8 + N_{LCM})\alpha^1 + \beta^1 \right\rfloor \tag{23}$$

and

$$n^{k\geq 2}_{LC} = \left\lfloor (191.7 + 2N_{LCM})\alpha^{k\geq 2} + \beta^{k\geq 2} \right\rfloor \tag{24}$$

where

$$\beta^1 = 34.17 + 133.8 n^{k^{1max}}_{Mult} \tag{25}$$

and

$$\beta^{k\geq 2} = 129.1 + 191.7 \frac{n^{k\geq 2_{max}}_{Mult}}{2}. \tag{26}$$

Figures 15 and 16 show the estimate number of PEs for some commercial FPGAs [27]–[29]. The estimation was based on Equations 10, 11, 19 and 20, for proposal 1, and Equations 17, 18, 23 and 24, for proposal 2. Based on Xilinx datasheet of the Multiplier IP Core [30], [31], the variable $N_{LCM} = 200$. For all values, it was considered

$$n^1_{PE} = \left\lfloor \frac{\gamma \times n_{PE}}{100} \right\rfloor \tag{27}$$

where γ is the percentage of the total number of PEs, n_{PE}, (or neurons) in the first layer ($k = 1$). Table 3 shows the values of the maximum number of PEs, n^{max}_{PE} for each proposal. Figures 15 and 16 show results for $\gamma = 20\%$ and Table 3 presented results for several values of γ.

The results from Figures 15 and 16 and Table 3 show the proposal viability for real problems with several layers and neurons (each PE^k_i represents the i-th neuron associated with the k-th layer).

TABLE 3. The estimate of the maximum number of PEs, n_{PE}^{max}, for some commercial FPGAs.

FPGA		n_{LC}^{max}	n_{Mult}^{max}	n_{PE}^{max}					
				Proposal 1			Proposal 2		
Label	Name			$\gamma = 20\%$	$\gamma = 40\%$	$\gamma = 60\%$	$\gamma = 20\%$	$\gamma = 40\%$	$\gamma = 60\%$
FPGA1	Virtex 6 XC6VLX240T	241152	768	620	680	750	720	800	890
FPGA2	Virtex 6 XC6VLX760	758784	864	1470	1610	1780	1720	1900	2120
FPGA3	Virtex 7 XC7V2000T	1954560	2160	3780	4140	4510	4410	4880	5450
FPGA4	Virtex UltraScale VU440	5540850	2880	9700	10000	10000	10000	10000	10000

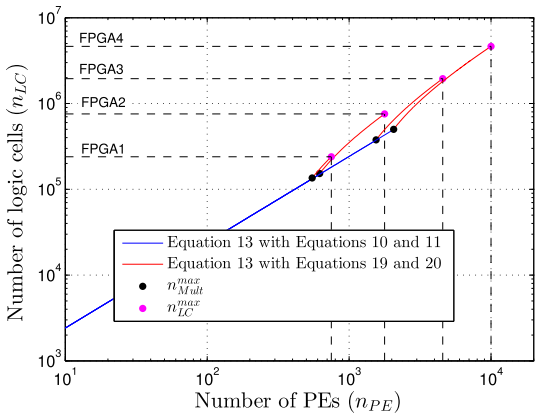

FIGURE 15. Scalability of Proposal 1 for the number of PEs (n_{PE}), $\gamma = 20\%$.

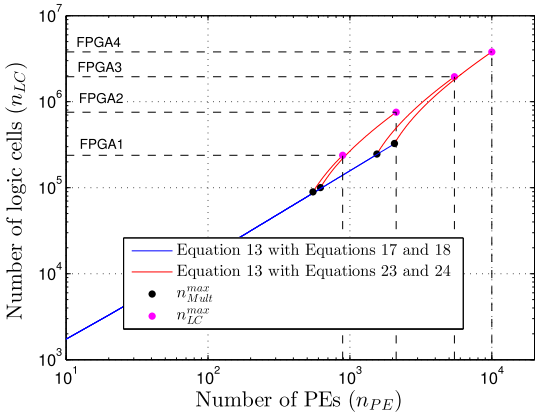

FIGURE 16. Scalability of Proposal 2 for the number of PEs (n_{PE}), $\gamma = 20\%$.

B. TIME PROCESSING ANALYSIS

1) PROPOSAL 1 - TIME PROCESSING ANALYSIS

In the proposal 1 (see Figure 4), the weights are input streams per neural network layer, and this reduces the hardware memory needed to store the weights at the same time it reduces the throughput (see Equations 5 and 7). One way to address this drawback is by creating a parallelization in the weights streams per layer, that is,

$$t_{PE} = \frac{(V^1 \times t_{rc})}{G} = \frac{(n_{PE}^1 \times t_{rc})}{G} \tag{28}$$

where G is the parallelization degree of the weights streams, in other words, for the k-th layer, there are G group of PEs, called GPE, and each GPE has $pr = V^k/G$ PEs. Figure 17 illustrates the parallelization of the weight stream where each g-th GPE_g^k drives a weight stream. The parallelization degree value is $1 \leq G \leq n_{PE}^1$ and

$$\frac{(n_{PE}^1 \times t_{rc})}{G} \geq t_c \tag{29}$$

which can be described as

$$G = \left\lfloor \left(n_{PE}^1 - 1\right)\eta + 1 \right\rfloor = \left\lfloor \left(\left\lfloor \frac{\gamma \times n_{PE}}{100} \right\rfloor - 1 \right)\eta + 1 \right\rfloor \tag{30}$$

where $0 \leq \eta \leq 1$.

Based on Equations 28 and 30, Equation 7 can be rewritten as

$$th_{ff} = \frac{G}{Q \times n_{PE}^1 \times t_{rc}} = \frac{\left\lfloor \left(\left\lfloor \frac{\gamma \times n_{PE}}{100} \right\rfloor - 1 \right)\eta + 1 \right\rfloor}{Q \times \left\lfloor \frac{\gamma \times n_{PE}}{100} \right\rfloor \times t_{rc}}. \tag{31}$$

The number of operations (adds and multipliers) per PE are 2 and 4 for the first ($k = 1$) and other ($k \geq 2$) layers, respectively. Thus using Equation 28, the performance, for proposal 1, pf, in operations per seconds (OPS) can be expressed as

$$pf = \frac{G\left(2n_{PE}^1 + 4n_{PE}^{k\geq 2}\right)}{n_{PE}^1 \times t_{rc}}. \tag{32}$$

Using Equations 12, 27, and 30, the Equation 32 can be rewritten as

$$pf = \frac{G\,(400 - 2\gamma)}{\gamma \times t_{rc}} = \frac{\left\lfloor \left(\left\lfloor \frac{\gamma \times n_{PE}}{100} \right\rfloor - 1 \right)\eta + 1 \right\rfloor (400 - 2\gamma)}{\gamma \times t_{rc}}. \tag{33}$$

After several synthesis results (see Table 1), the median values of t_{rc} and t_c were about 10 ns and 12.31 ns, respectively. The system critical path time can be increased regarding the number of bits and other operations such as the softmax activation function (see Equation 4) in the last layer, however, as the number of PEs is high, the constraint presented in Equation 29 can be easily satisfied. Figures 18, 19 and 20 show the throughput curve, th_{ff}, in IPS, and performance, pf, in Giga OPS (GOPS) to several values of Q, G and η for n_{PE} values presented in Table 3 with $\gamma = 40\%$.

Figures 18 and 19 show the throughput, th_{ff}, with different values of $\eta = \{0, 1/8, 1/4, 1/8\}$ and it is possible to observe that the th_{ff} decreases with Q for $\eta = 0$, however it is possible

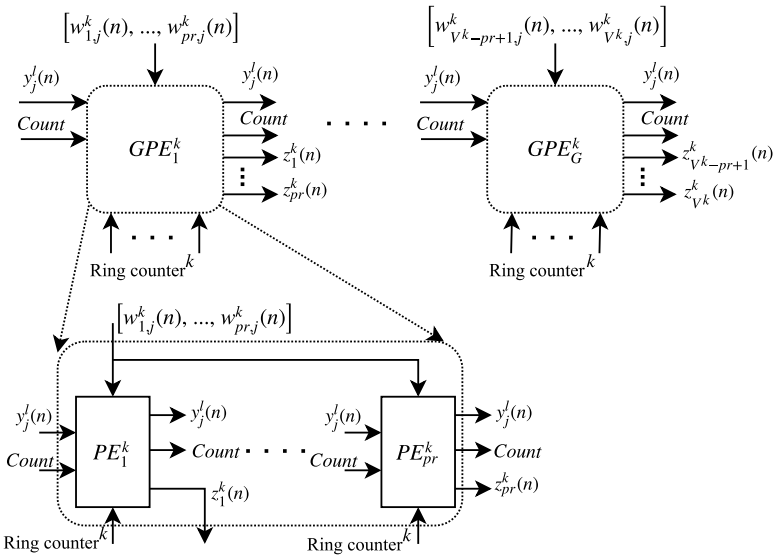

FIGURE 17. Architecture of the GPEs in the k-th hidden layer from proposal 1.

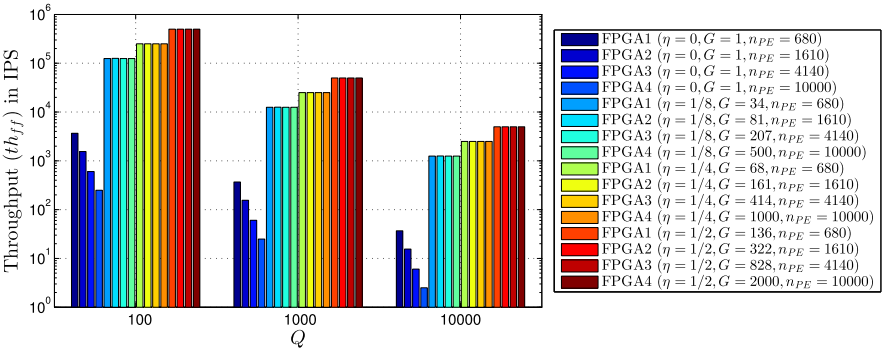

FIGURE 18. Throughput, th_{ff}, (in IPS) for several values of Q and η for n_{PE} values presented in Table 3 with $\gamma = 40\%$.

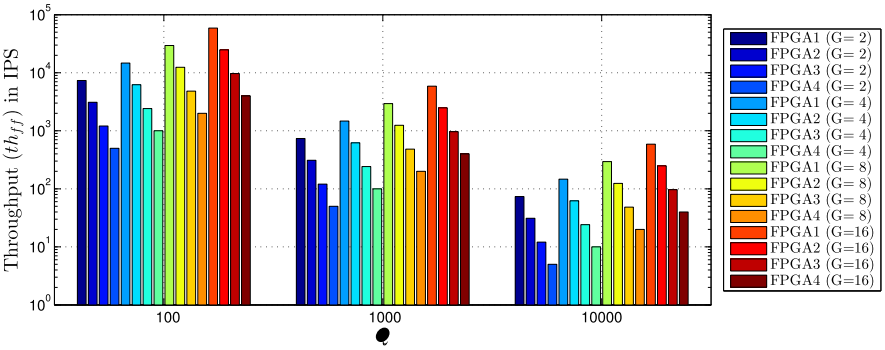

FIGURE 19. Throughput, th_{ff}, (in IPS) for several values of Q and G for n_{PE} values presented in Table 3 with $\gamma = 40\%$.

to compensate the reduction increasing η (or G). Figures 20 shows the performance of the proposal 1 for several values of η and, it could achieve more than 1000 GOPS or 1 tera OPS (TOPS) on FPGA 4 with $\eta \geq 0.3$. Table 4 shows the values of the th_{ff} in $\times 1000$ IPS (KIPS), and pf illustrated in Figures 18, 19 and 20.

2) PROPOSAL 2 - TIME PROCESSING ANALYSIS

For proposal 2, the throughput, th_{ff}, depends just on the system critical path time, t_c, and using the Equations 7 and 9, the throughput, in IPS, can be expressed as

$$th_{ff} = \frac{1}{Q \times t_c}. \tag{34}$$

TABLE 4. Values of the throughput, th_{ff}, (in KIPS) and performance, pf, (in GOPS) shown in Figures 18, 19 and 20 ($\gamma = 40\%$).

	n_{PE}^{max}	FPGA1 680			FPGA2 1610			FPGA3 4140			FPGA4 10000		
	η	1/8	1/4	1/2	1/8	1/4	1/2	1/8	1/4	1/2	1/8	1/4	1/2
	G	34	68	136	81	161	322	207	414	828	500	1000	2000
$Q = 100$	th_{ff} (KIPS)	125	250	500	125	250	500	125	250	500	125	250	500
$Q = 1000$	th_{ff} (KIPS)	12.5	25.0	50.0	12.5	25.0	50.0	12.5	25.0	50.0	12.5	25.0	50.0
$Q = 10000$	th_{ff} (KIPS)	1.25	2.50	5.00	1.25	2.50	5.00	1.25	2.50	5.00	1.25	2.50	5.00
	pf (GOPS)	27.2	54.4	108.8	64.8	128.8	257.6	165.6	331.2	662.4	400	800	1600

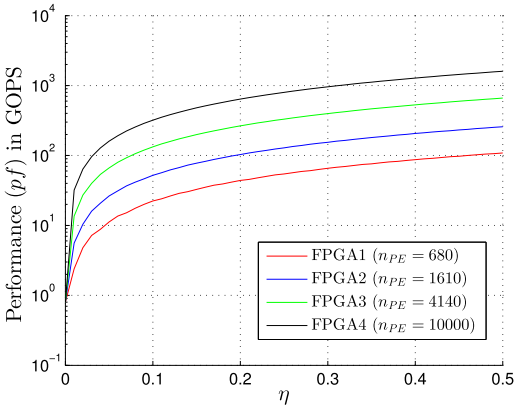

FIGURE 20. Performance, pf, (in GOPS) for several values of η (or G) for proposal 1 and using n_{PE} values presented in Table 3 with $\gamma = 40\%$.

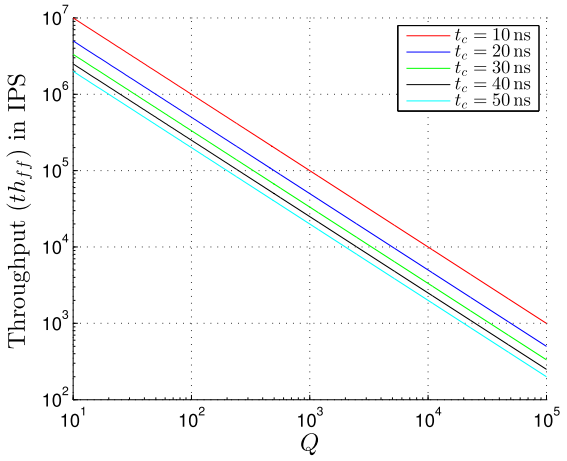

FIGURE 21. Throughput, th_{ff}, (in IPS) to several values of Q and t_c for proposal 2.

Figure 21 shows the curves of the th_{ff} for various values of Q and t_c. Based on the values presented in Table 2, the t_c achieved a median value of approximately 12 ns. However, the t_c value can increase with the number of bits and the other circuit elements such the softmax activation function implementation (see Equation 4). It is interesting to notice that the value of t_c is similar for the two proposals since the critical path of PE is similar for both.

Similarly of proposal 1, the number of operations (adds and multipliers) per PE for proposal 2 are 2 and 4 for the first ($k = 1$) and other ($k \geq 2$) layers, respectively. Thus using

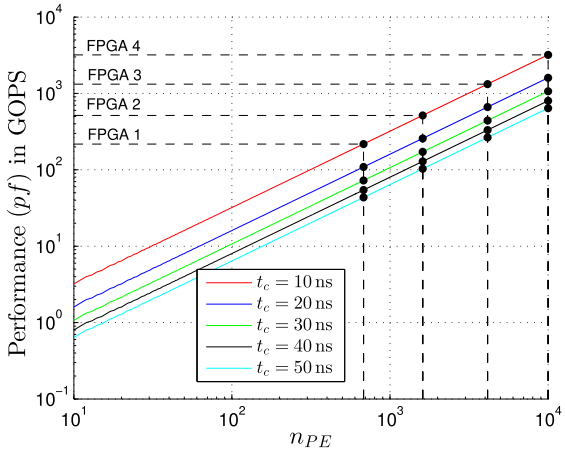

FIGURE 22. Proposal 2 performance, pf, (in GOPS) to several values of n_{PE} and t_c for some FPGAs. with $\gamma = 40\%$.

TABLE 5. Values of the throughput, th_{ff}, (in MIPS) and performance, pf, (in GOPS) shown in Figures 21 and 22 ($\gamma = 40\%$).

	Q	100	1000	10000	100000
$t_c = 10$ ns	th_{ff} (KIPS)	1000	100	10	1
$t_c = 50$ ns	th_{ff} (KIPS)	200	20	2	0.2
		FPGA1	FPGA2	FPGA3	FPGA4
	n_{PE}^{max}	680	1610	4140	10000
$t_c = 10$ ns	pf (GOPS)	217	515	1324	3200
$t_c = 50$ ns	pf (GOPS)	43	103	264	640

Equation 9, the performance, for proposal 2, pf, in operations per seconds (OPS) can be expressed as

$$pf = \frac{\left(2n_{PE}^{1} + 4n_{PE}^{k\geq 2}\right)}{t_c}. \tag{35}$$

Using Equations 12, 27, and

$$n_{PE}^{k\geq 2} = n_{PE} - \left\lfloor \frac{\gamma \times n_{PE}}{100} \right\rfloor, \tag{36}$$

Equation 35 can be rewritten as

$$pf = \frac{\left(4n_{PE} - 2n_{PE}^{1}\right)}{t_c} = \frac{\left(4n_{PE} - 2\left\lfloor \frac{\gamma \times n_{PE}}{100} \right\rfloor\right)}{t_c}. \tag{37}$$

Figure 22 shows the performance, pf, using several values of n_{PE} and t_c for some FPGAs. Table 5 resumes the values illustrated in Figures 21 and 22. Similar to proposal 1, proposal 2 can also achieve a performance of thousands of Giga operations per second or TOPS.

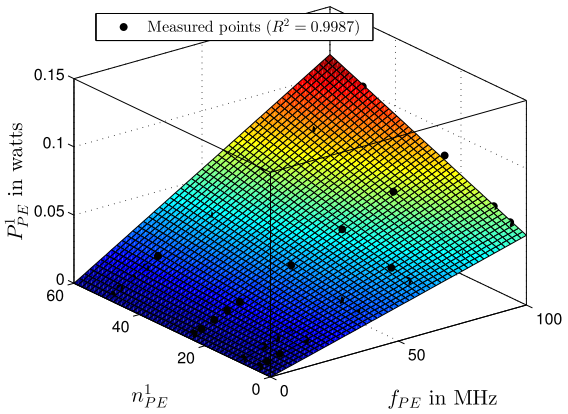

FIGURE 23. Surface of regression analysis of dynamic power consumption, P^1_{PE}, per number of PEs, n^1_{PE}, and frequency of PE, f_{PE}, in MHz for the first layer of proposal 1.

TABLE 6. Measured dynamic power consumption per number of PEs, n^1_{PE}, and frequency of PE, f_{PE}, in MHz for the first layer of proposal 1.

f_{PE}	P^1_{PE}			
	$n^1_{PE} = 5$	$n^1_{PE} = 10$	$n^1_{PE} = 25$	$n^1_{PE} = 50$
2 MHz	0.001 W	0.001 W	0.002 W	0.002 W
5 MHz	0.003 W	0.003 W	0.004 W	0.005 W
10 MHz	0.006 W	0.006 W	0.008 W	0.010 W
15 MHz	0.008 W	0.009 W	0.012 W	0.015 W
20 MHz	0.011 W	0.012 W	0.016 W	0.021 W
40 MHz	0.022 W	0.024 W	0.032 W	0.041 W
60 MHz	0.033 W	0.037 W	0.048 W	0.062 W
80 MHz	0.044 W	0.049 W	0.065 W	0.082 W
100 MHz	0.055 W	0.061 W	0.081 W	0.103 W

C. POWER CONSUMPTION ANALYSIS

1) PROPOSAL 1 - POWER CONSUMPTION

Figure 23 shows the surface of regression analysis using the measurement points presented in Table 6. The equation obtained from regression analysis ($R^2 = 0.9987$) can be expressed as

$$P^1_{PE} = \frac{f_{PE}\left(505 + 0.0592 f_{PE} + 10.65 n^1_{PE}\right)}{1 \times 10^6} \tag{38}$$

where P^1_{PE} is the dynamic power consumption in watts (W) of the first layer, and f_{PE} is the frequency of PE in MHz, that is

$$f_{PE} = \frac{1 \times 10^{-6}}{t_{PE}} = \frac{G \times 10^{-6}}{n^1_{PE} \times t_{rc}}. \tag{39}$$

The dynamic power measurement for other layers ($k \geq 2$) is presented in Table 7. For this case, the surface of regression analysis ($R^2 = 0.9984$) can be expressed as

$$P^{k\geq 2}_{PE} = \frac{f_{PE}\left(565.3 + 0.0429 f_{PE} + 12.39 n^{k\geq 2}_{PE}\right)}{1 \times 10^6}. \tag{40}$$

where $P^{k\geq 2}_{PE}$ is the dynamic power consumption in watts (W) for other layers ($k \geq 2$). The Figure 24 shows the curve of regression analysis of dynamic power consumption, $P^{k\geq 2}_{PE}$, per number of PEs, $n^{k\geq 2}_{PE}$, and frequency of PE, f_{PE}, in MHz for other layers of proposal 1 ($k \geq 2$).

TABLE 7. Measured dynamic power consumption per number of PEs, $n^{k\geq 2}_{PE}$, and frequency of PE, f_{PE}, in MHz for other layers of proposal 1 ($k \geq 2$).

f_{PE}	$P^{k\geq 2}_{PE}$		
	$n^{k\geq 2}_{PE} = 5$	$n^{k\geq 2}_{PE} = 10$	$n^{k\geq 2}_{PE} = 25$
2 MHz	0.001 W	0.001 W	0.002 W
5 MHz	0.003 W	0.003 W	0.004 W
10 MHz	0.007 W	0.007 W	0.009 W
15 MHz	0.01 W	0.01 W	0.013 W
20 MHz	0.013 W	0.013 W	0.018 W
40 MHz	0.026 W	0.027 W	0.036 W
60 MHz	0.039 W	0.04 W	0.053 W
80 MHz	0.052 W	0.053 W	0.071 W
100 MHz	0.065 W	0.067 W	0.089 W

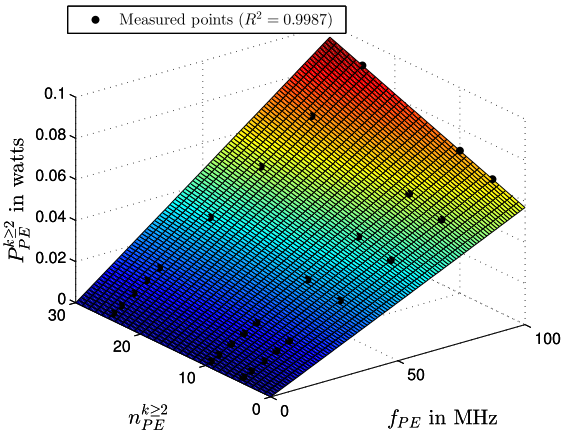

FIGURE 24. Surface of regression analysis of dynamic power consumption, $P^{k\geq 2}_{PE}$, per number of PEs, $n^{k\geq 2}_{PE}$, and frequency of PE, f_{PE}, in MHz for other layers of proposal 1 ($k \geq 2$).

Using Equations 27 and 39, the dynamic power consumption, P^1_{PE}, for the first layer can be expressed as

$$P^1_{PE} = \frac{G\left(505 + \frac{0.0592 G \times 10^{-6}}{\left\lfloor \frac{\gamma \times n_{PE}}{100} \right\rfloor \times t_{rc}} + 10.65 \left\lfloor \frac{\gamma \times n_{PE}}{100} \right\rfloor\right)}{\left\lfloor \frac{\gamma \times n_{PE}}{100} \right\rfloor \times t_{rc} \times 10^{12}} \tag{41}$$

and the dynamic power consumption, $P^{k\geq 2}_{PE}$, for other layers can be expressed by

$$P^{k\geq 2}_{PE} = \frac{G\left(565.3 + \frac{0.0429 G \times 10^{-6}}{\left\lfloor \frac{\gamma \times n_{PE}}{100} \right\rfloor \times t_{rc}} + 12.39 n^{k\geq 2}_{PE}\right)}{\left\lfloor \frac{\gamma \times n_{PE}}{100} \right\rfloor \times t_{rc} \times 10^{12}}. \tag{42}$$

Finally, the total of dynamic power consumption, P_{PE}, is expressed as

$$P_d = P^1_{PE} + P^{k\geq 2}_{PE}. \tag{43}$$

Figures 25 and 26 show curves associated with power consumption using Equations 41, 42, and 43. Some curves

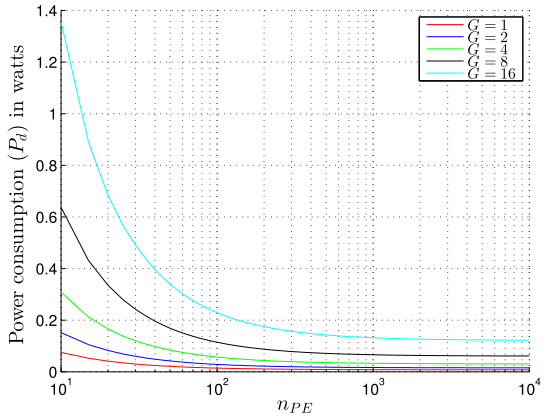

FIGURE 25. Total dynamic power consumption, P_d, per number of PEs, n_{PE}, for several values of G of proposal 1 ($\gamma = 40\%$).

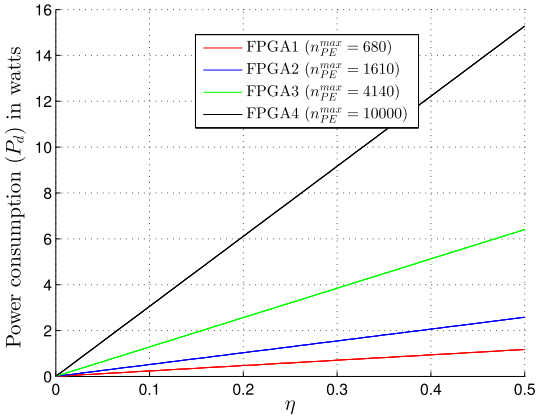

FIGURE 26. Total dynamic power consumption, P_d, per number of η for some FPGAs of proposal 1 ($\gamma = 40\%$).

values are presented in Table 8. All values were calculated with $\gamma = 40\%$.

2) PROPOSAL 2 - POWER CONSUMPTION

Figures 27 and 28 show the surface of regression analysis using the measurement points presented in Tables 9 and 10, respectively. The equation obtained from regression analysis ($R^2 = 0.9964$) for the first layer, P_{PE}^1 can be expressed as

$$P_{PE}^1 = \frac{f_{PE}\left(569.0849 + 24.4342 n_{PE}^1\right)}{1 \times 10^6} \tag{44}$$

where f_{PE} in MHz can be expressed by

$$f_{PE} = \frac{1 \times 10^{-6}}{t_{PE}} = \frac{1 \times 10^{-6}}{t_c} \tag{45}$$

and the equation obtained from regression analysis ($R^2 = 0.9931$) for other layer, $P_{PE}^{k\geq 2}$ can be expressed a

$$P_{PE}^{k\geq 2} = \frac{f_{PE}\left(639.3527 + 20.7985 n_{PE}^{k\geq 2}\right)}{1 \times 10^6}. \tag{46}$$

TABLE 8. Some curves values presented in Figures 25 and 26.

	FPGA1 ($n_{PE}^{max} = 680$)			
η	0	0.0037	0.0111	0.0258
G	1	2	4	8
P_d (W)	0.0086	0.0171	0.0342	0.0684
η	0.0554	1/8	1/4	1/2
G	16	34	68	136
P_d (W)	0.1369	0.2931	0.5864	1.1737
	FPGA2 ($n_{PE}^{max} = 1610$)			
η	0	0.0016	0.0047	0.0109
G	1	2	4	8
P_d (W)	0.0080	0.0160	0.0319	0.0639
η	0.0233	1/8	1/4	1/2
G	16	81	161	322
P_d (W)	0.1278	0.6484	1.2891	2.5790
	FPGA3 ($n_{PE}^{max} = 4140$)			
η	0	0.0006	0.0018	0.0042
G	1	2	4	8
P_d (W)	0.0077	0.0155	0.0309	0.0619
η	0.0091	1/8	1/4	1/2
G	16	207	414	828
P_{Pd} (W)	0.1237	1.6015	3.2031	6.4071
	FPGA4 ($n_{PE}^{max} = 10000$)			
η	0	0.0003	0.0008	0.0018
G	1	2	4	8
P_d (W)	0.0076	0.0153	0.0305	0.0611
η	0.0038	1/8	1/4	1/2
G	16	500	1000	2000
P_{Pd} (W)	0.1222	3.8184	7.6370	15.2749

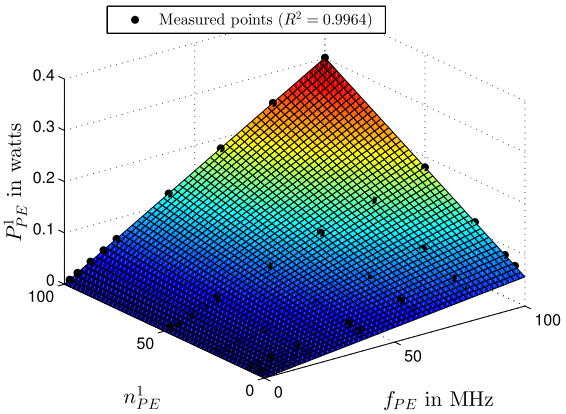

FIGURE 27. Surface of regression analysis of the dynamic power consumption, P_{PE}^1, per number of PEs, n_{PE}^1, and frequency of PE, f_{PE}, in MHz for the first layer of proposal 2.

Using Equations 27, 36 and 45, the Equations 44 and 46 can be characterized as

$$P_{PE}^1 = \frac{569.0849 + 24.4342 \left\lfloor \frac{\gamma \times n_{PE}}{100} \right\rfloor}{t_c \times 10^{12}} \tag{47}$$

and

$$P_{PE}^{k\geq 2} = \frac{639.3527 + 20.7985 \left(n_{PE} - \left\lfloor \frac{\gamma \times n_{PE}}{100} \right\rfloor\right)}{t_c \times 10^{12}}. \tag{48}$$

TABLE 9. Measured dynamic power consumption per number of PEs, n^1_{PE}, and frequency of PE, f_{PE}, in MHz for the first layer of proposal 2.

f_{PE}	P^1_{PE}				
	$n^1_{PE}=5$	$n^1_{PE}=10$	$n^1_{PE}=25$	$n^1_{PE}=50$	$n^1_{PE}=100$
2 MHz	0.001 W	0.002 W	0.003 W	0.004 W	0.006 W
5 MHz	0.003 W	0.004 W	0.007 W	0.010 W	0.015 W
10 MHz	0.006 W	0.008 W	0.014 W	0.021 W	0.030 W
15 MHz	0.010 W	0.012 W	0.021 W	0.031 W	0.045 W
20 MHz	0.013 W	0.016 W	0.028 W	0.042 W	0.060 W
40 MHz	0.026 W	0.033 W	0.056 W	0.083 W	0.120 W
60 MHz	0.039 W	0.049 W	0.084 W	0.125 W	0.180 W
80 MHz	0.052 W	0.065 W	0.112 W	0.167 W	0.241 W
100 MHz	0.064 W	0.081 W	0.140 W	0.207 W	0.298 W

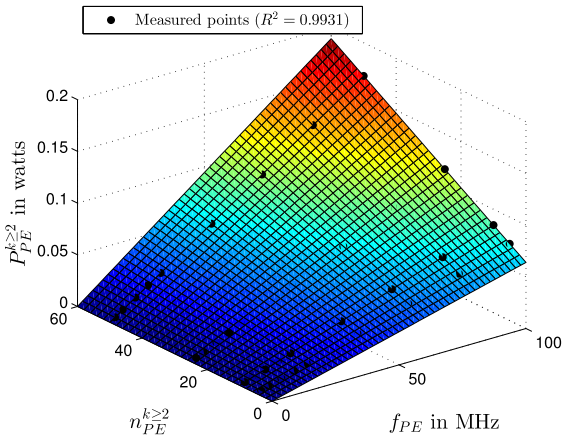

FIGURE 28. Surface of regression analysis of the dynamic power consumption, $P^{k\geq2}_{PE}$, per number of PEs, $n^{k\geq2}_{PE}$, and frequency of PE, f_{PE}, in MHz for the first layer of proposal 2.

TABLE 10. Measured dynamic power consumption per number of PEs, $n^{k\geq2}_{PE}$, and frequency of PE, f_{PE}, in MHz for other layers of proposal 2 ($k \geq 2$).

f_{PE}	$P^{k\geq2}_{PE}$			
	$n^{k\geq2}_{PE}=5$	$n^{k\geq2}_{PE}=10$	$n^{k\geq2}_{PE}=25$	$n^{k\geq2}_{PE}=50$
2 MHz	0.001 W	0.002 W	0.003 W	0.003 W
5 MHz	0.004 W	0.004 W	0.007 W	0.008 W
10 MHz	0.007 W	0.009 W	0.014 W	0.017 W
15 MHz	0.011 W	0.013 W	0.021 W	0.025 W
20 MHz	0.015 W	0.017 W	0.029 W	0.034 W
40 MHz	0.030 W	0.034 W	0.057 W	0.067 W
60 MHz	0.045 W	0.051 W	0.086 W	0.101 W
80 MHz	0.060 W	0.068 W	0.114 W	0.135 W
100 MHz	0.074 W	0.085 W	0.142 W	0.167 W

Based of Equations 43, 47 and 48, the total dynamic power consumption per PEs, P_d, for proposal 2 can be expressed as

$$P_d = \frac{1208.4376 + 20.7985 n_{PE} + 3.6357 \left\lfloor \frac{\gamma \times n_{PE}}{100} \right\rfloor}{t_c \times 10^{12}}. \quad (49)$$

Figure 29 shows the total dynamic power consumption per PEs, P_d, using Equation 49 and Table 11 presented some values for FPGA1, FPGA2, FPGA3 and FPGA4.

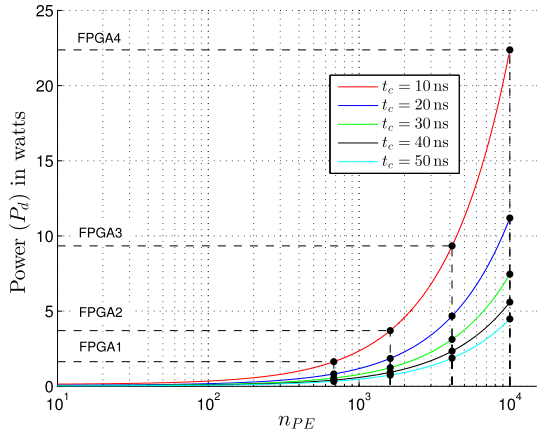

FIGURE 29. Proposal 2 total dynamic power consumption, P_d, (in watts) to several values of n_{PE} and t_c for some FPGAs with $\gamma = 40\%$.

TABLE 11. Values of the dynamic power consumption, P_d, (in watts) shown in Figure 29 ($\gamma = 40\%$).

		FPGA1	FPGA2	FPGA3	FPGA4
	n^{max}_{PE}	680	1610	4140	10000
$t_c = 10$ ns	P_{PE} (W)	1.634	3.704	9.334	22.374
$t_c = 50$ ns	P_{PE} (W)	0.327	0.740	1.867	4.475

D. VALIDATION

In order to validate the implementation proposed in this article, a data set of manuscript digits called MNIST [32], containing 60000 images in the training set and 10000 images in the test set was used for the experiments. Each image contains 28 × 28 pixels, totaling 784 inputs for the SSAE. The SSAE implemented in this validation used two hidden layers, consisting of the 784-100-50-10 architecture. The experiments were performed with the 10000 images from the MNIST test set. The network training was previously performed using the 60000 images from the train set in the Matlab/Simulink [33] simulation platform (License number 1080073) with the scaled conjugate gradient (SCG) algorithm and the target FPGA was a Virtex 6 XC6VLX240T-1FF1156 [27]. Figure 30 shows the hardware setup for the experiments. It was used the Virtex-6 FPGA ML605 Evaluation Kit by Xilinx [34]. A video demonstration of the proposal 2 executing the MNIST test set is presented in [35].

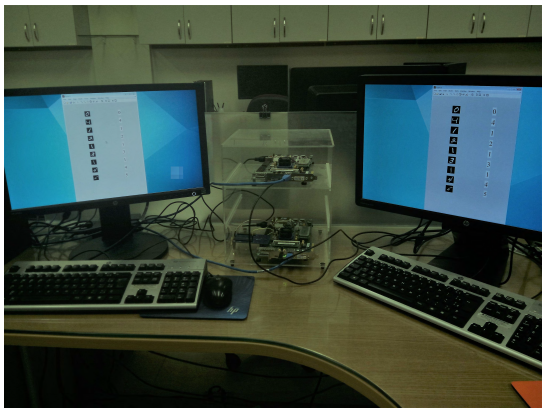

FIGURE 30. Hardware setup for the experiments with the Xilinx Virtex-6 FPGA ML605 evaluation kit.

1) PROPOSAL 1 VALIDATION

For the hardware validation, a comparison was made between the results obtained by the implementation on the Matlab/Simulink platform and the results obtained by the hardware implementation. Thus, the mean square error (MSE) between the output of the Matlab implementation, $s_i^{ref}(n)$, and the output of the hardware implementation was calculated. The MSE calculation for this experiment is defined as

$$MSE = \frac{1}{H \times 10000} \sum_{i=1}^{H} \sum_{n=1}^{10000} \left(s_i^{ref}(n) - s_i(n) \right)^2. \quad (50)$$

It was found that the MSE between the implementation results in Matlab, which used floating point (64 bits), and the fixed-point hardware implementation was only 2.1×10^{-6}, which is a fairly acceptable result, since the weights associated with the hardware implementation used only 12 bits in the decimal part. Among the 10000 images used in the validation, Matlab's hit percentage was 93.4%, while the proposed hardware implementation reached a 93.38% hit. The small percentage error in the classification of the implementation in FPGA comparing to the implementation in Matlab is related to the bit resolution used for the network synaptic weights. However, this result is still quite relevant since it indicates that a higher bit resolution (such as 64 bits, for example) is not required to achieve meaningful results, showing that by using only 12 bits in the decimal part, in a fixed point representation, it is possible to guarantee the reduction in the hardware occupation area, besides promoting the throughput increase [6], [20].

TABLE 12. Proposal 1 - hardware area occupation.

	Multipliers (n_{Mult})	Logic cells (n_{CL})
Synthesis	220 (28.65%)	37796 (15.67%)
Estimated (Equations 10, 11, 13, and 16)	220 (28.65%)	38326 (15.89%)

After the validation of the hardware implementation using Matlab, the synthesis was performed in order to obtain the FPGA resources allocation report. The Table 12 details the data related to the hardware area occupation of the circuit implemented in the FPGA. The first column shows the number of multipliers which are used mainly by the PEs of each k-th layer. The second column displays the number of registers and the third column shows the amount of logic cells used throughout the circuit.

The data presented in Table 12 show the feasibility of the implementation of proposal 1. It turns out that only 16% of the target FPGA logic cells were occupied, indicating that there is still plenty of room to additional layers and neurons. One of the elements that were most used was the multipliers, around 29%, since each PE from the first layer utilizes one, and the PEs of the other layers consume two due to the bias. Despite this, it is noted that it is still possible to greatly increase the number of layers and neurons of the SSAE.

TABLE 13. Proposal 1 - processing times and delays ($G = 1$ and $Q = 800$).

t_{rc}	d	t_{ff}	th_{ff}	P_d Measurement	P_d Estimated (Eqs. 41, 42, and 43)
10 ns	2.4 ms	0.8 ms	1.25 KFPS	0.003 W	0.0047 W

TABLE 14. Proposal 2 - hardware area occupation.

	Multipliers (n_{Mult})	Logic cells (n_{CL})
Synthesis	220 (28.65%)	24399 (10.12%)
Estimated (Equations 13, 16, 17, and 18)	220 (28.65%)	25045 (10.39%)

TABLE 15. Proposal 2 - processing times and delays ($Q = 800$).

t_c	d	t_{ff}	th_{ff}	P_d Measurement	P_d Estimated (Eq. 49)
45 ns	0.1 ms	0.04 ms	25 KFPS	0.112 W	0.110 W

The Table 13 presents information about the processing time of proposal 1, considering the least amount of required weight streams. The first column exposes the ring counter time for the first hidden network layer, t_{rc}. The second column shows the architecture initial delay, d, defined by Equation 6. The third column presents the network feedforward execution time, t_{ff}, expressed by Equation 8, after the initial delay (Equation 6), the fourth column shows the network throughput, th_{ff}, determined by Equation 7, which in this particular experiment consists of the number of images classified per second and the last column shows the dynamic power consumption, P_d, for this implementation.

2) PROPOSAL 2 VALIDATION

The hardware validation results with Matlab for proposal 1, presented in Section V-D1 for a Virtex FPGA 6 XC6VLX240T-1FF1156, also applies for proposal 2.

TABLE 16. State of the art comparison associated with throughput, th_{ff}, and power consumption, P_d, for two different deep neural network size.

References				Proposals				
Ref.	Device	Network architecture	Value	Proposal 1			Proposal 2	
[5]	Intel (Altera) Stratix V	$3072-2000-750-10$		$Q = 4000$, $t_{rc} = 10$ ns, and $t_c = 45$ ns				
			th_{ff} (FPS)	G	th_{ff} (FPS)	Speedup	th_{ff} (FPS)	Speedup
			45	1	12.5	–	5208	$\approx 116\times$
				2	25	–		
				4	50	$\approx 1.11\times$		
				8	100	$\approx 2.22\times$		
			P_d (W)	G	P_d (W)	Power saving	P_d (W)	Power saving
			16	1	0.0023	$\approx 7081\times$	1.46	$\approx 11\times$
				2	0.0045	$\approx 3540\times$		
				4	0.0090	$\approx 1770\times$		
				8	0.0181	$\approx 885\times$		
[14]	Xilinx XC7Z020	$784-256-256-10$		$Q = 1000$, $t_{rc} = 10$ ns, and $t_c = 45$ ns				
			th_{ff} (FPS)	G	th_{ff} (FPS)	Speedup	th_{ff} (FPS)	Speedup
			12.45	1	390.62	$\approx 31\times$	22222	$\approx 1784\times$
				2	781.25	$\approx 62\times$		
				4	1562.5	$\approx 125\times$		
				8	3125	$\approx 251\times$		
			P_d (W)	G	P_d (W)	Power saving	P_d (W)	Power saving
			1.814	1	0.0059	$\approx 309\times$	0.29	$\approx 6\times$
				2	0.0117	$\approx 154\times$		
				4	0.0234	$\approx 77\times$		
				8	0.0468	$\approx 38\times$		

However, new reports were obtained regarding the hardware resources occupation and the processing time, after the synthesis of the implementation of the second proposal.

Table 14 presents the synthesis data related to the FPGA occupation area from proposal 2. The columns of this table are organized according to what was already presented for Table 12 in the Subsection V-D1 for proposal 1.

The data presented in Table 14 show that the proposal 2 was able to further optimize the use of FPGA hardware resources comparing to proposal 1. This was possible due to the use of LUTs to store the synaptic weights in each i-th PE_i^k, dispensing with the implementation of some modules used in proposal 1. It can be seen that only 11% of the FPGA logic cells were occupied. The amount of multipliers used remained the same in relation to the previous proposal, around 29%, since there was no change in the way calculations were performed in each PE_i^k. Thus, the data presented for proposal 2 reaffirm the possibility of significantly increasing the number of layers and neurons of the network, indicating even better results than the previous proposal.

The Table 15 presents information about the circuit processing time for the second proposal. The first column exposes the PE processing time, t_{PE}, and the other columns are organized according to what was presented to the table 13 in the subsection V-D1, referring the proposal 1.

The data presented in Table 15, for proposal 2, are quite expressive. The PE execution time achieved was only 48 ns, the initial delay, 0.1 ms, and the feedforward time achieved 0.03 ms. With this, it was possible to reach a throughput of 26000 classified images per second, which is a value approximately 20× higher than the value obtained in proposal 1, using the least possible amount of weight streams. In this way, it was verified that the use of LUTs to store the network neuron weights resulted in the throughput increase in comparison with the previous proposal. However, when using the maximum weights streams limit per layer, proposal 1 tends to reach the results of proposal 2. Thus, both proposals proved feasibility in massive data problems.

E. STATE OF THE ART COMPARISON

From the results presented in the previous sections, this section is intended to make a comparison of the results obtained by the implementation of each proposal here presented with the results of works from the state of the art. Table 16 shows a comparison with works presented in [5] and [14]. In those works, it is possible to compare results associated with throughput, th_{ff}, and power consumption, P_d, for two different deep neural network size.

The related work presented in [5] proposes an SSAE FPGA implementation and it uses an equivalent FPGA in terms of processing for the validation. As presented in Table 16, the proposal 1 has speedup gains for the values of $G \geq 4$, however implementation has an expressive power saving (about 1000×) for several values of G ($1 \leq G \leq 8$). When G increases, the speedup also increases (see Figures 18 and 19) and the power saving decreases (see Figures 25 and 26). Regarding the proposal 2, the speedup was about 116× over the architecture presented in [5] and the power saving was about 11×. The work presented [5] was implemented with the high-level synthesis OpenCL framework and, this method reduces the customization capacity implying a reduction in performance.

TABLE 17. State of the art comparison associated with performance, *pf*, in GOPS.

References			Proposals				
Ref.	Device	Value	Proposal 1			Proposal 2	
[36]	Xilinx Ultrascale KU060 FPGA	pf (GOPS)	$n_{PE}^{max} = 10000, \gamma = 40\%, t_{rc} = 10\,\text{ns}$, and $t_c = 45\,\text{ns}$				
		173	η	pf (GOPS)	Speedup	pf (GOPS)	Speedup
			0.1	320	$\approx 1.8\times$	711	$\approx 4.11\times$
			0.2	640	$\approx 3.7\times$		
			0.3	960	$\approx 5.5\times$		
			0.4	1280	$\approx 7.4\times$		
[12]	Xilinx Virtex7 485t FPGA	pf (GOPS)	$n_{PE}^{max} = 10000, \gamma = 40\%, t_{rc} = 10\,\text{ns}$, and $t_c = 45\,\text{ns}$				
		61.62	η	pf (GOPS)	Speedup	pf (GOPS)	Speedup
			0.1	320	$\approx 5.2\times$	711	$\approx 11.5\times$
			0.2	640	$\approx 10.4\times$		
			0.3	960	$\approx 15.6\times$		
			0.4	1280	$\approx 20.8\times$		
[13]	Xilinx Kintex K325T FPGA	pf (GOPS)	$n_{PE}^{max} = 10000, \gamma = 40\%, t_{rc} = 10\,\text{ns}$, and $t_c = 45\,\text{ns}$				
		260	η	pf (GOPS)	Speedup	pf (GOPS)	Speedup
			0.1	320	$\approx 1.2\times$	711	$\approx 2.7\times$
			0.2	640	$\approx 2.5\times$		
			0.3	960	$\approx 3.7\times$		
			0.4	1280	$\approx 4.9\times$		
[37]	ASIC	pf (GOPS)	$n_{PE}^{max} = 10000, \gamma = 40\%, t_{rc} = 10\,\text{ns}$, and $t_c = 45\,\text{ns}$				
		452	η	pf (GOPS)	Speedup	pf (GOPS)	Speedup
			0.2	640	$\approx 1.4\times$	711	$\approx 1.53\times$
			0.3	960	$\approx 2.1\times$		
			0.4	1280	$\approx 2.8\times$		
			0.5	1600	$\approx 3.5\times$		

In [14], the speedup was more expressive than the work presented in [5], between 31× and 251× for proposal 1 and about 1784× for proposal 2, however the power consumption was lower. It is interesting to observe that the power saving is greater for larger architectures, on the other hand the speedup is lower. The FPGA accelerator proposed in [14], called DLAU, computes just 32 neurons with 32 weights every clock cycle and the implementations here proposed can compute several neurons (or PEs) every clock cycle and which increases the performance when comparing with the DLAU.

Table 17 compares the performance results, *pf*, in GOPS with works presented in [36], [12], [13] and [37]. The proposals presented in [36] and [12] were implemented with HLS tools. In [36] was proposed an HLS engine called Caffeine and in [12] was proposed an implementations with Xilinx Vivado HLS. In [13] was presented the effects of the data precision in neural network implementations and in [37] was presented an application-specific integrated circuit (ASIC) proposal.

As presented in Table 17, the speedup is higher with respect to cases based on HLS, [36] and [12]. The proposal 1 achieved a speedup about 7.4× and 20.8× over works presented in [36] and [12], respectively and proposal 2 achieved a speedup about 4.11× and 11.5× over the same works. Finally, it is important to note that the implementations here proposed (proposal 1 and 2) presented a speedup even over ASIC implementations, as presented in [37].

VI. CONCLUSION

This paper presented a hardware implementation proposal of the Stacked Sparse Autoencoder Deep Learning technique. The DNN feedforward phase was implemented using fixed point and design in RTL. Throughout the implementation the systolic array technique was applied, which allowed the use of many neurons in the various network layers, besides allowing the results to be obtained in a short processing time. All the implementation details were presented as well as results related to the syntheses for occupation and processing time of two distinct implementations targeting a FPGA Virtex 6 XC6VLX240T-1FF1156.

The results showed that both implementations could achieve high throughputs. Particularly, the use of LUTs to store the synaptic weights of each neuron in proposal 2 allowed the reduction of the area occupied in the FPGA, besides guaranteeing the increase of the throughput obtained, compared to the lower bound value of the proposal 1 throughput. In spite of that, the throughput value reached by proposal 1 could be increased in line with the increase in the number of weight streams per network layer and may tend to the throughput value obtained by proposal 2. On top of that, the proposal 1 allows, in practical applications, the same hardware to be used for different problems by enabling new weights to be added to the network through weight streams after previous network training. In addition, the comparison with related works also justified the feasibility of the proposed implementations, demonstrating the possibility of

reaching high speedups. Finally, the results indicated the suitability of using this Deep Learning technique in massive data problems.

REFERENCES

[1] P. Baldi, "Autoencoders, unsupervised learning, and deep architectures," in *Proc. ICML Workshop Unsupervised Transf. Learn.*, 2012, pp. 37–49.

[2] L. Deng and D. Yu, "Deep learning: Methods and applications," *Found. Trends Signal Process.*, vol. 7, nos. 3–4, pp. 197–387, Jun. 2014.

[3] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Netw.*, vol. 61, pp. 85–117, Jan. 2015.

[4] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion," *J. Mach. Learn. Res.*, vol. 11, pp. 3371–3408, Dec. 2010.

[5] J. Maria, J. Amaro, G. Falcao, and L. A. Alexandre, "Stacked autoencoders using low-power accelerated architectures for object recognition in autonomous systems," *Neural Process. Lett.*, vol. 43, no. 2, pp. 445–458, Apr. 2016.

[6] A. C. D. de Souza and M. A. C. Fernandes, "Parallel fixed point implementation of a radial basis function network in an FPGA," *Sensors*, vol. 14, no. 10, pp. 18223–18243, 2014.

[7] M. F. Torquato and M. A. C. Fernandes, "High-performance parallel implementation of genetic algorithm on FPGA," *Circuits, Syst., Signal Process.*, pp. 1–26, Jan. 2019.

[8] A. L. X. Da Costa, C. A. D. Silva, M. F. Torquato, and M. A. C. Fernandes, "Parallel implementation of particle swarm optimization on FPGA," *IEEE Trans. Circuits Syst., II, Exp. Briefs*, to be published.

[9] L. M. D. Da Silva, M. F. Torquato, and M. A. C. Fernandes, "Parallel implementation of reinforcement learning Q-learning technique for FPGA," *IEEE Access*, vol. 7, pp. 2782–2798, 2018.

[10] Y. Zhou and J. Jiang, "An FPGA-based accelerator implementation for deep convolutional neural networks," in *Proc. 4th Int. Conf. Comput. Sci. Netw. Technol. (ICCSNT)*, vol. 1, Dec. 2015, pp. 829–832.

[11] M. Bettoni, G. Urgese, Y. Kobayashi, E. Macii, and A. Acquaviva, "A convolutional neural network fully implemented on FPGA for embedded platforms," in *Proc. New Gener. CAS (NGCAS)*, Sep. 2017, pp. 49–52.

[12] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays (FPGA)*, New York, NY, USA, 2015, pp. 161–170. doi: 10.1145/2684746.2689060.

[13] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. (2015). "Deep learning with limited numerical precision." [Online]. Available: https://arxiv.org/abs/1502.02551

[14] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie, and X. Zhou, "DLAU: A scalable deep learning accelerator unit on FPGA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 36, no. 3, pp. 513–517, Mar. 2017.

[15] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2017, pp. 1–12.

[16] X. Wei *et al.*, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *Proc. 54th Annu. Design Automat. Conf.*, 2017, Art. no. 29.

[17] T. V. Huynh, "Deep neural network accelerator based on FPGA," in *Proc. 4th NAFOSTED Conf. Inf. Comput. Sci.*, Nov. 2017, pp. 254–257.

[18] Y. Ma, N. Suda, Y. Cao, S. Vrudhula, and J.-S. Seo, "ALAMO: FPGA acceleration of deep learning algorithms with a modularized RTL compiler," *Integration*, vol. 62, pp. 14–23, Jun. 2018.

[19] W. Zhao, Z. Jia, X. Wei, and H. Wang, "An FPGA implementation of a convolutional auto-encoder," *Appl. Sci.*, vol. 8, no. 4, p. 504, 2018.

[20] J. F. Jiang, R. D. Hu, D. S. Wang, J. W. Xu, and Y. Dou, "Performance of the fixed-point autoencoder," *Tehnicki Vjesnik-Tech. Gazette*, vol. 23, pp. 77–82, Feb. 2016.

[21] Y. Jin and D. Kim, "Unsupervised feature learning by pre-route simulation of auto-encoder behavior model," *Int. J. Comput., Elect., Automat., Control Inf. Eng.*, vol. 8, no. 5, pp. 668–672, 2014.

[22] A. Suzuki, T. Morie, and H. Tamukoh, "A shared synapse architecture for efficient FPGA implementation of autoencoders," *PLoS ONE*, vol. 13, no. 3, Mar. 2018, Art. no. e0194049.

[23] D. J. M. Moss, D. Boland, P. Pourbeik, and P. H. W. Leong, "Real-time FPGA-based anomaly detection for radio frequency signals," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2018, pp. 1–5.

[24] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.

[25] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 8, pp. 1798–1828, Aug. 2013.

[26] H. T. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," in *Sparse Matrix Proceedings*, vol. 1. Philadelphia, PA, USA: SIAM, 1978, pp. 256–282.

[27] XILINX Virtex-6. (2018). *Virtex-6 Family Overview*. Accessed: Dec. 28, 2018. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds150.pdf

[28] XILINX Virtex-7. (2018). *7 Series FPGAs Data Sheet: Overview*. Accessed: Dec. 28, 2018. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf

[29] XILINX Virtex Ultra. (2018). *UltraScale Architecture and Product Data Sheet: Overview*. Accessed: Dec. 28, 2018. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf

[30] XILINX IP Core. (2018). *LogiCORE IP Multiplier v11.2*. Accessed: Dec. 28, 2018. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/mult_gen_ds255.pdf

[31] XILINX IP Core. (2018). *LogiCORE IP Multiplier v12.0*. Accessed: Dec. 28, 2018. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/mult_gen/v12_0/pg108-mult-gen.pdf

[32] Y. LeCun, C. Cortes, and C. J. Burges. (Jan. 2018). *Yann LeCun's Home Page*. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[33] The MathWorks. (Jan. 2018). *Matlab/Simlink*. [Online]. Available: https://www.mathworks.com/

[34] XILINX ML605. (2019). *Virtex-6 FPGA ML605 Evaluation Kit*. Accessed: Jan. 29, 2019. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/ek-v6-ml605-g.html

[35] G. F. Maria Coutinho and M. A. C. Fernandes. (2019). *SSAE Video Demonstration—MNIST*. Accessed: Jan. 29, 2019. [Online]. Available: https://drive.google.com/drive/folders/1vKUv3LW04caC5ZLONmHqqt0gg85IgWLE?usp=sharing

[36] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2016, pp. 1–8.

[37] T. Chen *et al.*, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 269–284, 2014.

MARIA G. F. COUTINHO was born in Natal, Brazil. She received the B.S. degree in computer science from the State University of Rio Grande do Norte, Natal, in 2017, and the M.Sc. degree in electrical and computer engineering from the Federal University of Rio Grande do Norte, Natal, in 2019, where she is currently pursuing the Ph.D. degree in electrical and computer engineering and a Team Member of the Research Group on Embedded Systems and Reconfigurable Computing. Her main research topic is the acceleration of deep learning algorithms through reconfigurable computing in FPGA. Her research interests include artificial intelligence, embedded systems, and reconfigurable hardware.

MATHEUS F. TORQUATO was born in Natal, Brazil. He received the B.Sc. degree in science and technology, the B.E. degree in computer engineering, and the M.Sc. degree in computer engineering from the Federal University of Rio Grande do Norte, Natal, in 2013, 2015, and 2017, respectively. He is currently a part of the Research Group on Embedded Systems and Reconfigurable Hardware, where his main research topic is the acceleration of artificial intelligence (AI) algorithms through reconfigurable computing (RC) on FPGA. Apart from his main research topic of AI and RC at his home university, he has other research experiences such as human–computer interaction at the Future Interaction Technology Lab, Swansea University, Wales, U.K., and computer vision at the Sensing and Machine Vision for Automation and Robotic Intelligence Lab, University of Ottawa, ON, Canada. His research interests include AI, embedded systems, reconfigurable hardware, human–computer interaction, and the tactile Internet.

MARCELO A. C. FERNANDES was born in Natal, Brazil. He received the B.S. and M.S. degrees from the Federal University of Rio Grande do Norte, Natal, in 1997 and 1999, respectively, and the Ph.D. degree from the University of Campinas, Campinas, Brazil, in 2010, all in electrical engineering. From 2015 to 2016, he was a Visiting Researcher with the Centre Telecommunication Research (CTR), King's College London, London, U.K. He is currently an Adjunct Professor with the Department of Computer Engineering and Automation, Federal University of Rio Grande do Norte. He is also the Leader of the Research Group on Embedded Systems and Reconfigurable Computing (RESRC) and a Coordinator of the Laboratory of Machine Learning and Intelligent System (LMLIS). He has authored or co-authored many scientific papers and practical studies with reconfigurable computing on FPGA to accelerate artificial intelligence algorithms. His research interests include artificial intelligence, digital signal processing, embedded systems, reconfigurable hardware, and the tactile Internet.