IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received May 4, 2019, accepted May 29, 2019, date of publication June 5, 2019, date of current version June 25, 2019.

*Digital Object Identifier 10.1109/ACCESS.2019.2920885*

# A Novel Systolic Parallel Hardware Architecture for the FPGA Acceleration of Feedforward Neural Networks

**LEANDRO D. MEDUS, TARAS IAKYMCHUK, JOSE VICENTE FRANCES-VILLORA, MANUEL BATALLER-MOMPEÁN, AND ALFREDO ROSADO-MUÑOZ**

Group for Processing and Digital Design, Department of Electronic Engineering, Universitat de Valencia, 46100 Burjassot, Spain

Corresponding author: Alfredo Rosado-Muñoz (alfredo.rosado@uv.es)

**ABSTRACT** New chips for machine learning applications appear, they are tuned for a specific topology, being efficient by using highly parallel designs at the cost of high power or large complex devices. However, the computational demands of deep neural networks require flexible and efficient hardware architectures able to fit different applications, neural network types, number of inputs, outputs, layers, and units in each layer, making the migration from software to hardware easy. This paper describes novel hardware implementing any feedforward neural network (FFNN): multilayer perceptron, autoencoder, and logistic regression. The architecture admits an arbitrary input and output number, units in layers, and a number of layers. The hardware combines matrix algebra concepts with serial-parallel computation. It is based on a systolic ring of neural processing elements (NPE), only requiring as many NPEs as neuron units in the largest layer, no matter the number of layers. The use of resources grows linearly with the number of NPEs. This versatile architecture serves as an accelerator in real-time applications and its size does not affect the system clock frequency. Unlike most approaches, a single activation function block (AFB) for the whole FFNN is required. Performance, resource usage, and accuracy for several network topologies and activation functions are evaluated. The architecture reaches 550 MHz clock speed in a Virtex7 FPGA. The proposed implementation uses 18-bit fixed point achieving similar classification performance to a floating point approach. A reduced weight bit size does not affect the accuracy, allowing more weights in the same memory. Different FFNN for Iris and MNIST datasets were evaluated and, for a real-time application of abnormal cardiac detection, a ×256 acceleration was achieved. The proposed architecture can perform up to 1980 Giga operations per second (GOPS), implementing the multilayer FFNN of up to 3600 neurons per layer in a single chip. The architecture can be extended to bigger capacity devices or multi-chip by the simple NPE ring extension.

**INDEX TERMS** Feedforward neural networks - FFNN, systolic hardware architecture, FPGA implementation, neural network acceleration, deep neural networks.

## I. INTRODUCTION

Feed-Forward Neural Networks (FFNN) in different variations are one of the most used machine learning algorithms, with numerous applications typically running under PC-based software systems. However, when fast processing time is required in real-time applications or fast prediction, decision or classification, a PC-based system might not be able to provide enough throughput. Nowadays, this situation becomes very common since FFNN sizes are growing due to the complexity of the problems to be solved and big data applications, with an increasing number of inputs, neuron units, and the number of layers. Moreover, power consumption and computational speed is an important issue; CPUs and GPUs can process data at a high speed, but the use of power and resources is higher than FPGA and other custom embedded hardware platforms [1].

The computational resources and internal architecture possibilities of FPGA devices differ from classic Von Neumann PCs or even SIMD processing units as GPUs, CPUs or DSPs. FPGA are optimal for massive parallel and relatively simple processing units, rather than large universal computational blocks. This is the case of FFNN, which are composed of parallel inputs, parallel outputs and multiple neuron units

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



2169-3536 © 2019 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

arranged in layers. Thus, the FPGA device is a good candidate to be used as an independent device, receiving inputs directly from the process, computing them and sending the output to a real process. FPGA devices are one of the best options for the hardware implementation of FFNN in particular and artificial intelligence algorithms in general since required computations are based on the sum of products, which can fit very well into the FPGA internal slices (logic blocks, arithmetic units and RAM). Thus, the use of FPGA devices allows the parallelization of neural networks by using concurrent computing of multiple units, which can be massively interconnected and are able of be reconfigured with different weights and topologies depending on the target application. In addition, data representation can be tuned according to precision and accuracy requirements, as in [2]. Different applications can be found, as in the case of [3], where a fault tolerant Hopfield Neural Network was implemented in FPGA for space applications, or in [4], where a weightless neural network with Multi-valued Probabilistic Logic Nodes (M-PLN) was implemented and evaluated. Other similar practical FPGA implementations of neural networks can be found, as in MPPT controllers for solar charging applications [5], or in Software Defined Radio (SDR) modulation [6]. Alternatively, an FPGA accelerator can be connected to a PC in a Hardware In the Loop System (HILS), where input and output data are sent and received from the PC, guaranteeing a fixed processing time from the dedicated hardware [7], being independent on the load from the host PC.

Specific hardware implementation of artificial neural networks can be beneficial to speed up both training and online processes, as in [8], where the backpropagation learning algorithm was implemented, in [9], where a neural fuzzy chip with on-chip incremental learning ability was described, or in [10], where a fully pipelined acceleration architecture is designed aiming to alleviate the high computational demands of Restricted Boltzmann Machines (RBM). Further, the inclusion of artificial intelligence algorithms in embedded systems, targeting real-time applications, is common, as in [11], where an optimized streaming method for the hardware acceleration of deep convolutional neural networks is shown, or in [12], where the acceleration of Support Vector Machines (SVM) through a hybrid processing hardware architecture (optimized for object detection) was proposed. In addition, some other applications include modeling, as the case of a digital implementation of a modified astrocyte model [13].

The process of generating specific hardware from a versatile architecture can be tedious. In order to assist to inexperienced users in the hardware implementation of neural networks, some works propose neural network software design tools using user-friendly visual graphical interfaces, where the hardware configuration files are automatically generated according to the user options. This is the case in [14], where a complete design environment for migrating neural networks from software to FPGA hardware, including network training, was described, or [15], which describes an end-user design environment where any FFNN can be modeled, simulated, and later programmed on an FPGA.

Concerning hardware topologies, a straightforward neural unit architecture might consist of using separate hardware entities to perform each input by weight multiplication, and a parallel adder to add the multiplication products. Such design would be a fine-grained architecture. However, such architecture is unpractical due to the very high hardware requirements in occupation and interconnection lines, which leads to high power and high resource usage, along with low speed architectures [16]. A different approach might use neuron units with serial processing, which is more practical because every unit just requires one Multiply-And-Accumulate (MAC) block, time-multiplexing data into the same units [17]. However, despite serial or parallel computation, all fine-grained architectures implementing directly the neural units suffer from the connectionist problem: the number of interconnection synapses grows exponentially with the number of units in the FFNN, consuming a significant part of resources and reducing the operating frequency due to long lines delay. Thus, with every new unit in a fully-connected or feedforward network, the topology of connections becomes more complex and the synthesis software tries to create connections using logic cells instead of connection lines, inefficiently using logic resources, adding delay and power consumption to the device. Then, a fine-grained architecture can be used only for small size networks, limiting its range of applications.

When implementing large size networks, a more promising approach is offered by a coarse-grained architecture where a small number of processing elements perform time-multiplexed serial computation of the network units. In this case, performance is a trade-off between processing node complexity and working speed: the simpler the processing node, the faster, but requires more clock cycles. In this approach, the hardware implementation benefits from short point-to-point data lines and pipelined uniform operations, obtaining higher clock speed and lower resource usage at the expense of higher latency. As an example, in [18], the biggest FFNN layer was implemented and reused.

In this work, an unusual approach to design the proposed architecture was followed. Typically, custom computing architectures are defined according to the required algorithm calculations: after defining the control and data flow, the hardware architect makes use of required hardware blocks, which availability, resource occupation and performance might depend on the device used. However, in this work, the systolic architecture design considered the FPGA available resources as a premise: the existing FPGA on-chip resources were analyzed and then, the computational process was proposed. By doing this, an optimal use of resources is obtained, using short lines, reducing the use of logic resources, reducing local interconnections between blocks (avoiding delays due to long internal connections), increasing clock rate and, as a side effect, reducing power consumption, too. Thus, considering typical FPGA resources, the proposed

hardware architecture was proposed: it is a versatile and universal Systolic Massive Parallel Architecture (SYMPA) for feedforward neural networks, based on computationally independent Neural Processing Elements (NPE) having local weight memory, global data input, and command lines. The resulting hardware structure is a combination of fine-grained and coarse-grained with parallel input processing (all neuron units of an FFNN layer process the same input at the same time) and time-multiplexed input (a new input every clock cycle). The SYMPA architecture allows the implementation of arbitrary size and arbitrary type FFNN. The computational procedure and its hardware architecture are described, but also an analysis of the proposed FPGA-based implementation is conducted using different topologies, to assess the level of optimization achieved and the weak points of the proposed implementation. The main contributions of this paper are as follows:

- Proposal of an architecture able to adopt any type of FFNN. Thus, it can be used to solve different applications using FFNN such as the Multilayer Perceptron (MLP), autoencoder (AE), or logistic regression (LR).
- The proposed architecture can scale up to arbitrary size (inputs, units and layers), only limited by the available resources. Scaling up has no penalty on the operation clock frequency. It provides linear resource growth with the number of neuron units in the largest layer of the FFNN.
- A single Activation Function Block (AFB) is required for the whole FFNN (not one per neuron unit as usual), easily permitting to modify this block as different applications may require different AFB. Moreover, each FFNN layer may use a different AFB if more than one is defined.
- The proposed architecture achieves up to 550MHz of operation frequency, being the AFB the limiting block and thus requiring a careful design or selection of the activation function.
- The architecture provides great versatility: the output of intermediate layers can be available externally and the weights of the neural network can be modified during execution without device reprogramming.
- In a Virtex-7 FPGA implementation, SYMPA architecture accelerates up to x256 times with respect to the PC implementation and can perform up to 1980 GOPS when using 3600 neuron units per layer.

Despite some works demonstrate the feasibility of on-chip learning [19], [20], embedded learning is not considered in this work since weights are generally calculated using off-line procedures (backpropagation, ELM, etc.). Once calculated, the weight values are loaded to the FPGA internal memory.

Section II describes the types of FFNNs implemented in this work, followed by section III where the algorithm used for implementation is described. Section IV details the hardware implementation; section V describes the experimental results; section VI uses a real case application detecting anomalous heart rhythms to conduct a platform comparison, and, finally, sections VII and VIII deal with discussion and conclusions of the work.

## II. MULTILAYER PERCEPTRON (MLP), AUTOENCODER (AE) AND LOGISTIC REGRESSION (LR) AS FFNN TYPES

The main characteristics of any FFNN are the basic computation neuron units and the topology where they are arranged. Structured in layers, each unit of a layer is connected to all units in the next layer, never in a cyclic or recurrent form, imposing an ever forward flow of information. Each unit $j$ in a layer $i$ performs a sum of products for all the outputs of the previous layer, $\vec{Y}_{i-1}$, or the input layer, $\vec{X}$, with a weighted value $W_{jk}^{i}$ for each connection (synaptic weight), and adding a bias value $b_{ij}$. The result serves as the input to an activation function $F$ where the final output of the unit is generated. Eq. 1 shows the required computation for a single unit $j$ in a layer $i$ containing $N$ inputs; the same operation must be repeated for all units in an FFNN. The activation function is non-linear, typically a sigmoid, although different approximations simplifying calculations exist [21].

$$Y_{ij} = F\left(\sum_{k=1}^{N}(W_{jk}^{i} * Y_{(i-1)k}) + b_{ij}\right) \tag{1}$$

Fig. 1 shows the most common structure for an FFNN, the Multilayer Perceptron (MLP). In a MLP, units are arranged in layers and forward interconnections exist between inputs, layers and outputs.

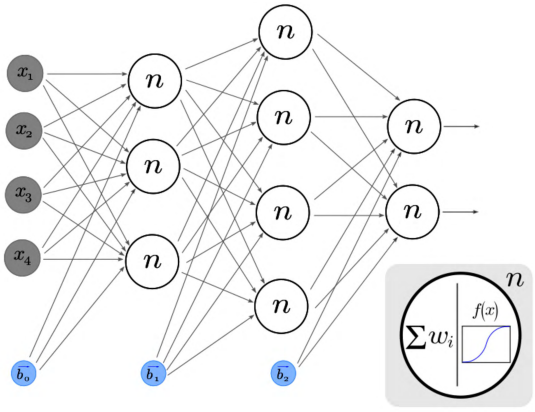


**FIGURE 1. Multilayer Perceptron (MLP) topology for 4 inputs, two hidden layers with 3 and 4 neuron units, respectively, and 2 outputs (4×3×4×2).**

Another type of FFNNs are autoencoders (AE), which aim to learn a compressed, distributed representation (encoding) of a dataset. An autoencoder network is a type of neural network whose main focus is to extract features that will help in reconstructing the original input signal back from those features efficiently. Autoencoders can be stacked (stacked autoencoders) [22], [23] to form deep networks and were first introduced in the 80s by Hinton [24]. The simplest form of an autoencoder is very similar to an MLP [25], with an input layer, an output layer and one or more hidden layers. An autoencoder can be seen as a type of MLP where the

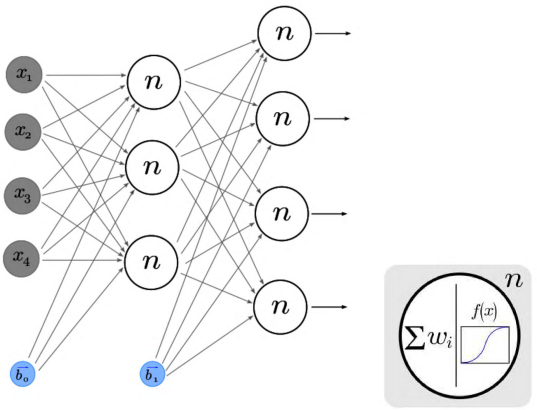


**FIGURE 2.** Diagram of an autoencoder (AE) with 4 inputs (and thus, 4 outputs), and 3 hidden units.

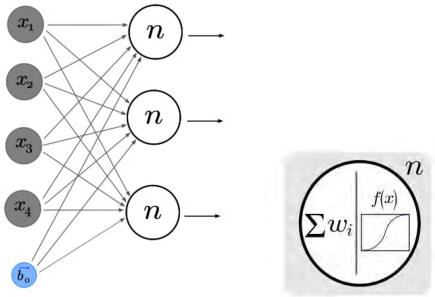


**FIGURE 3.** Logistic regression topology using 4 inputs and 3 neuron units.

output layer has the same number of nodes as the input layer, and instead of being trained to predict the target value $Y$ of given inputs $X$, autoencoders are trained to make the reconstruction $X'$ of their own inputs (Fig. 2). Therefore, autoencoders are unsupervised learning models.

Autoencoder networks have shown excellent properties for feature extraction, data compression or visualization [26], [27]. Autoencoders are popular [28]–[31] as they are used as a pre-training mechanism for deep supervised networks. Training deep neural networks is difficult as the magnitudes of gradients in the lower layers and in higher layers are different, it is difficult for stochastic gradient descent to find a good local optimum, then, as deep networks contain many parameters, they can remember training data and do not generalize well. With AE used for pre-training, the process of training a deep network is divided into some steps: training a sequence of autoencoders, one layer at a time using unsupervised data, train the last layer using supervised data and use backpropagation to fine-tune the entire network using supervised data.

Logistic Regression (LR) can also be considered a special architecture of neural networks [32]. The functional forms for logistic regression and artificial neural network models are quite different. However, an FFNN with only an output layer is identical to a logistic regression model if the logistic activation function is used (Fig. 3). Logistic regression is widely applied in medical applications [32], [33], topography [34] or social networks [35].

Additionally, other different variations of FFNN propose the use of a different activation function for each layer, or obtaining the output layer values without using any activation function, or using a SOFTMAX activation function for multiclass classification [36]. Note that, to obtain a more versatile architecture, it is important to be able to configure the activation function that calculates the output value in each layer.

## III. ALGORITHM PROPOSED FOR THE GENERAL FFNN COMPUTATION

Despite the number of inputs, outputs, hidden units and number of layers, which may vary according to the application, the topologies for the aforementioned types of FFNN only differ in some interconnection schemes. This fact makes it possible to propose a versatile hardware architecture defining the computation procedure is defined independently on the number of inputs, layers and neuron units in each layer.

All FFNN share the following properties:

1) No connections exist among the units in the same layer.
2) The output of a layer is a function of the previous layer inputs and a bias.
3) All units in a layer share the same activation function and can be computed independently. Different layers can have different activation functions, or even it can be skipped (as for the last layer in some FFNN).

In a general sense, an FFNN can be described as a parallel computation of Eq. 1 for each unit in the FFNN. The difference between units lies in its inputs and weight values. Thus, defining a vector $\vec{N} = (N_0, \ldots, N_i, \ldots, N_L)$ associated to the number of inputs ($N_0$), number of units in layer $i$ ($N_i$), and number of outputs ($N_L$), the generic computation of the unit $j = \{1, \ldots, N_i\}$ in the layer $i = \{1, \ldots, L\}$ ($L$ is the total number of layers including the output layer) can be described according to Eq. 2. In this equation, $\vec{Y}_i$ is the layer output obtained from the previous layer output $\vec{Y}_{i-1}$ (input values $\vec{X}$ are renamed as $\vec{Y}_0$), $\mathbf{W^i}$ and $\vec{b}_i$ are, respectively, the weight matrix and bias vector of a given layer $i$, and $F$ the activation function. The final FFNN outputs correspond to $\vec{Y}_L$ values.

$$Y_{ij} = F\left(\sum_{k=1}^{N_{i-1}} (W^i_{jk} * Y_{(i-1)k}) + b_{ij}\right) \tag{2}$$

Eq. 3 shows the computation case for the $N_1$ units in the first layer, which results in $\vec{Y}_1$.

$$\begin{aligned} Y_{11} &= F\left(\sum_{k=1}^{N_0} (W^1_{1k} * Y_{0k}) + b_{11}\right) \\ Y_{12} &= F\left(\sum_{k=1}^{N_0} (W^1_{2k} * Y_{0k}) + b_{12}\right) \\ &\ldots \\ Y_{1N_1} &= F\left(\sum_{k=1}^{N_0} (W^1_{N_1k} * Y_{0k}) + b_{1N_1}\right) \end{aligned} \tag{3}$$

To adapt the data computation flow in a regular form, the bias values of layer $i$ are renamed as $\vec{w}_{0_i} = \vec{b}_i$ and every

output vector $\vec{Y}_i$ from a layer is expanded by one element equal to 1 before entering the next layer (named $\vec{Y}'_i$, $N_{i-1}+1$ in size). Additionally, the bias column vector is concatenated with the weight matrix to create a layer matrix $\mathbf{W'^i} = [\vec{w}_{0_i} \mathbf{W^i}]$ with size $(N_i \times (N_{i-1}+1))$. Then, the computation $\mathbf{W'^i} \times \mathbf{Y'_{i-1}} = \vec{S}_i$ provides all sum of products corresponding to each unit in layer $i$. Repeating the process for all layers ($i = 1, \ldots, N_L$), it can be written as in Eq. 4. Finally, in order to obtain the output value for each FFNN unit, each $S_{ij}$ element must be applied the activation function F (Eq. 5).

Thus, Eqs. 4 and 5 show that computing the output of an FFNN requires two main operations: vector by matrix multiplication and activation function computation.

$$\begin{aligned}
\vec{S}_1 &= \mathbf{W'^1} \times \begin{bmatrix} 1 \\ \vec{Y}_0 \end{bmatrix} = \mathbf{W'^1} \times \mathbf{Y'_0} \\
&\cdots \\
\vec{S}_i &= \mathbf{W'^i} \times \begin{bmatrix} 1 \\ \vec{Y}_{i-1} \end{bmatrix} = \mathbf{W'^i} \times \mathbf{Y'_{i-1}} \\
&\cdots \\
\vec{S}_L &= \mathbf{W'^L} \times \begin{bmatrix} 1 \\ \vec{Y}_{L-1} \end{bmatrix} = \mathbf{W'^L} \times \mathbf{Y'_{L-1}}
\end{aligned} \tag{4}$$

$$\begin{aligned}
\vec{Y}_1 &= (F(S_{11}), \ldots, F(S_{1k}), \ldots, F(S_{1N_1})) \\
&\cdots \\
\vec{Y}_i &= (F(S_{i1}), \ldots, F(S_{ik}), \ldots, F(S_{iN_i})) \\
&\cdots \\
\vec{Y}_L &= (F(S_{L1}), \ldots, F(S_{Lk}), \ldots, F(S_{LN_L}))
\end{aligned} \tag{5}$$

In general, the matrix by vector multiplication $\mathbf{B} \times \vec{A}$ of a matrix $\mathbf{B}$ (size $M \times N$) by a column vector $\vec{A}$ (size $N$) can be parallelized in two forms:

- *Multiplying in parallel all elements of $\vec{A}$ by the matrix row vector $\vec{B}_i$.* This option requires $N$ multipliers and the process must be repeated for all $M$ rows in the matrix. All partial sums of a single unit are computed in one cycle, thus computing the partial sums of all units after $M$ clock cycles.
- *Multiplying in parallel a single element of the input vector $\vec{A}$ by each element of a matrix column $\vec{B}_j$.* This option requires $M$ multipliers and computes one partial weighted sum of all units in a layer in the same clock cycle. Thus, after $N$ clock cycles, all partial sums are obtained.

These two possibilities are different from the hardware architecture perspective. To obtain all partial sums in a FFNN unit, the first option requires the simultaneous multiplication of different pairs of data $A_k * B_{ik}$, whereas the second method requires the simultaneous multiplication of one fixed argument $A_i$ by all row elements $B_{ki}$, i.e. the problem is reduced to scalar-by-vector computation since, in each clock cycle, one element is the same for all multiplications.

The second method was the selected option for the algorithm in this work because it requires $N-1$ less data paths in hardware and the input vector $\vec{A}$ can be fed into the system element-wise instead of loading all elements at the same time. This feature is very useful when the FFNN is working with data stream in real time, i.e. a new data input value enters the FFNN per clock cycle. To illustrate this feature, Eq. 6 shows a computation example for an $N_0$-input FFNN with three units in a single hidden layer, being $\vec{X}$ the input vector and $\mathbf{W'^1}$ the weight matrix where each row represents the weights of one unit and the first column contains the bias values for each unit. After computation ($N_0+1$ clock cycles), the resulting vector $\vec{S}_1$ contains the sum of products for each unit in the layer and only the activation function evaluation is required to obtain the final units' output value (not included in this equation). Using as many multipliers as units in the layer working in parallel ($N_1 = 3$ in this case, $N_i$ with $i = 1, \ldots, N_L$ in general), the partial sums for all units will be simultaneously computed for each new input data $X_k$, every clock cycle (assuming that a multiplication operation takes place in one clock cycle). The bold elements marked in Eq. 6 illustrate a partial sum result obtained for each unit in a single clock cycle when using three multipliers to process the input $X_1$. Using three arithmetic accumulators ($N_i$ in general), the complete sum of products for each neuron unit in the layer are simultaneously obtained after $N_0 + 1$ clock cycles.

$$\begin{aligned}
\begin{bmatrix} S_{11} \\ S_{12} \\ S_{13} \end{bmatrix} &= \begin{bmatrix} w'^1_{00} & \boldsymbol{\omega'^1_{01}} & \cdots & w'^1_{0N_0} \\ w'^1_{10} & \boldsymbol{\omega'^1_{11}} & \cdots & w'^1_{1N_0} \\ w'^1_{20} & \boldsymbol{\omega'^1_{21}} & \cdots & w'^1_{2N_0} \end{bmatrix} \times \begin{bmatrix} 1 \\ \mathbf{X_1} \\ X_2 \\ \vdots \\ X_{N_0} \end{bmatrix} \\
&= \begin{bmatrix} w'^1_{00} + \boldsymbol{\omega'^1_{01}} * \mathbf{X_1} + w'^1_{02} * X_2 + \cdots + w'^1_{0N_0} * X_{N_0} \\ w'^1_{10} + \boldsymbol{\omega'^1_{11}} * \mathbf{X_1} + w'^1_{12} * X_2 + \cdots + w'^1_{1N_0} * X_{N_0} \\ w'^1_{20} + \boldsymbol{\omega'^1_{21}} * \mathbf{X_1} + w'^1_{22} * X_2 + \cdots + w'^1_{2N_0} * X_{N_0} \end{bmatrix}
\end{aligned} \tag{6}$$

Once the output values of units in a layer are obtained, they are used as inputs for the next layer. The layer structure and computation scheme described above can be repeated for all layers using the same computation elements, i.e. the same hardware architecture. Fig. 4 graphically shows the proposed, layer-wise, parallel feedforward architecture for a $N_0 \times N_1 \times N_2 = 5 \times 4 \times 4$ FFNN (L = 2). Initially, the inputs in a layer are serially processed by the layer computation blocks, feeding one input each clock cycle as described above. After obtaining the results of the units in one layer, its layer output values $\vec{S}_i$ (i = 1, ..., L) are stored in a memory (inter-layer memory) and the same hardware can be iteratively reused to compute all layers. It is important to note that the stored values $\vec{S}_i$ correspond to the sum of products result without activation function evaluation. It is only before entering into the next layer (or final output result) that they are evaluated by the activation function. In other words, the layer output values are stored before being evaluated by the activation function, and they are only evaluated when they are needed. Applying this mechanism, together with the serial input processing, a single activation function block can serve for all the FFNN structure since one value per clock cycle is used, no matter the FFNN size.

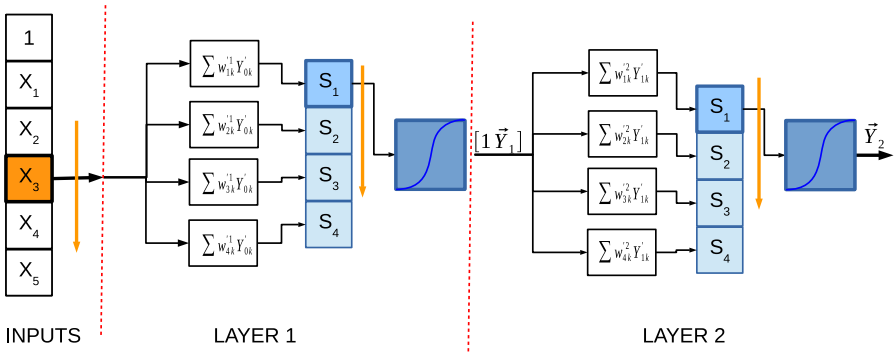


**FIGURE 4.** Computation procedure for the proposed layer-wise parallel feedforward architecture with serial input: An example of a $L = 2$ FFNN with $N_0 \times N_1 \times N_2 = 5 \times 4 \times 4$. The input vector $\vec{X}$ is serially entered and processed (orange arrow denotes sequentiality) by the array of multiply-accumulators to calculate the weighted sum of products. As the output layer values $\vec{S}$ are generated, they enter the activation function block, generating $\vec{Y}$ as the output of the layer. When one layer is finished, the computation is repeated for the next layer.

As in FFNN architectures, every unit in a layer is connected with all units in the next layer, the layer-wise parallelism is very convenient as every layer output value is dependent upon the output values of the previous layer and the dataflow always goes in one direction.

As the number of units in each layer varies, the number of required computational blocks and associated control of computations changes from one layer to another. In order to the hardware architecture can fit all layers, the number of hardware processing units N is given by the highest number of units in a layer, considering all layers in the FFNN ($N = max(N_1, \ldots, N_L)$). Then, by careful design of the control flow, only the required number of computational units will be used in each layer computation. This architecture allows to process all units in a layer in parallel, with inputs serially processed in a pipelined fashion.

Taking into account all the aforementioned factors, the proposed hardware implementation of this versatile and universal SYstolic Massive Parallel Architecture (SYMPA) for FFNN is based on computationally independent Neural Processing Elements (NPE) having a Multiply-Accumulate (MAC) unit, local weight memory, global data input, and command lines. For the inter-layer communication memory implementation, all NPEs also include a scratchpad register connected in a daisy-chain, forming a scratchpad ring *SR*. In this architecture, all control signals but one are globals, and thus, the system has excellent scalability with linear dependency between the size of the network layer and hardware occupation. The hardware implementation benefits from short point-to-point data lines and pipelined uniform operations. An additional feature of this architecture is the external availability of intermediate result values after each layer computation, which can be used for network training algorithms or any other debugging purposes.

Algorithm 1 describes the sequence of operations required when computing an FFNN with $L$ layers. The loops involving k index are those performed in parallel by NPEs, the loops for j index are serially computed, and the loops for i index reuse the hardware computation architecture. To demonstrate the scratchpad daisy-chaining, it is presented as a vector (SR), common for all units. The activation function is denoted as *AFB*.

**Algorithm 1** Computation Process for a FFNN Network

$L \rightarrow$ Number of layers
$N_i \rightarrow$ Number of NPEs used for layer $i$ ($i = 1, \ldots, L$)
$w'^i(j)(k) \rightarrow w'^i_{jk}$ weight k from unit j in layer-$i$
*Processing the inputs, layer* 1
$Acc() = w'^1()(0)$; ▷ bias loaded in MAC unit
**for** j=1 to $N_1$ **do**
  **for** k=1 to $N_0$ **do**
    $Acc(j) = Acc(j) + X(k) * w'^1(j)(k)$; ▷ MAC
  **end for**
**end for**
$SR() = Acc()$; ▷ Flush results into the SR
*Processing layers* $i = \{2, ..., L\}$
**for** i=2 to L **do** ▷ layers
  **for** j=1 to $N_i$ **do** ▷ units
    $Acc(j) = w'^i(j)(0)$ ▷ bias load
    **for** k=1 to $N_{i-1}$ **do** ▷ unit output value
      $X(k) =$AFB$(SR(k))$;
      $SR(k+1) = SR(k)$;
      $Acc(j) = Acc(j) + X(k) * w'^i(j)(k)$; ▷ MAC
    **end for**
  **end for**
**end for**
$SR() = Acc()$;
*Processing the output layer*
**for** j=1 to $N_L$ **do**
  $Output(j) =$AFB$(SR(j))$;
  $SR(j+1) = SR(j)$;
**end for**

*(SR: Scratchpad Ring, AFB: Activation Function Block)*

The proposed algorithm consists of three main parts: accepting the network inputs, forward propagation of the signal through the layers in the network, and obtaining

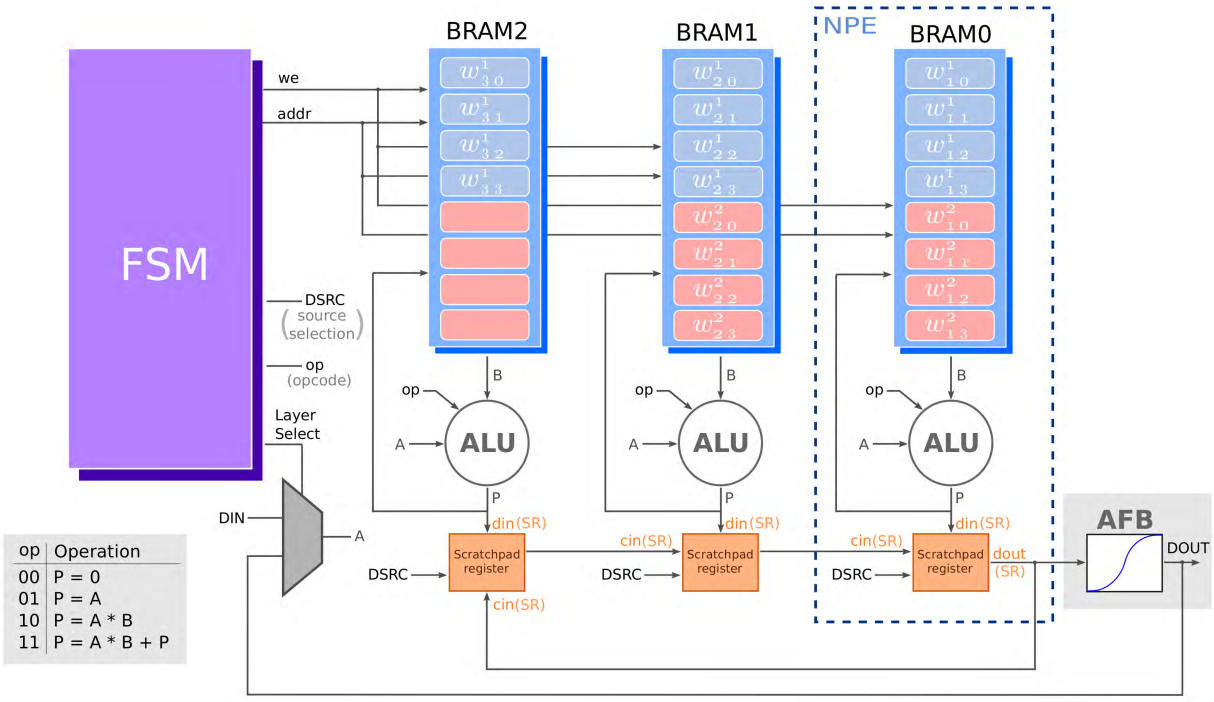


**FIGURE 5.** Hardware structure of a 3×3×2 MLP neural network. NPEs are created and connected according to the number of units existing in the biggest layer. In this case, three NPEs are required, the blue dashed line shows one NPE with BRAM memory, ALU, scratchpad register (SR) and all existing data and control lines. From top to bottom starting in the first layer, each NPE memory contains the unit weights. As two output units exist, the left-most NPE is empty for the bottom half memory since only one unit weights are stored. A single activation function (AFB) is used for the whole FFNN.

the outputs of the final layer. In the first part, the input data are externally taken from the FFNN inputs, which are sequentially introduced and concurrently multiplied by its corresponding weight $w'^{1}_{jk}$ of each unit $j$ in each NPE. The multiplication result of each NPE is added to the accumulator register by a MAC operation. As the bias value is stored in weight index zero ($w'^{i}_{j0}$) for each unit, it is loaded in the MAC unit of each NPE in the first clock cycle, before the inputs to the accumulator enter. After $N_0 + 1$ cycles, where $N_0$ is the number of network inputs, the accumulator of each unit $Acc(j)$ contains the sum of products of inputs by weights. Then, computation of the first layer is done and the values of the accumulators are stored in the scratchpad ring $SR$. Now, the NPEs are ready to compute the next layer. Note that, for those layers with a lower number of units than N, not all the NPEs of the hardware structure will be used.

The scratchpad ring $SR$ is a serially connected line of registers, similar to a shift register with a parallel load. After latched, data can be shifted out serially while NPEs compute a new sum of products. The $SR$ contains the computed sum of products values and they already have to be shifted through the AFB block to calculate the final output value.

As it can be seen in Algorithm 1, the input data are serially entered to the NPEs to compute the first layer; for the remaining layers, the layer input data are those obtained from the output of the previous layer. From the hardware perspective, the proposed architecture consists of a single layer of concurrent NPE units with common input, where each layer is computed using the weights of the corresponding layer, only using the number of units required for each layer.

After computing the sum of products value of the units in the output layer, the network output can be externally accessed by serially reading on the AFB output port, finishing the FFNN computation. Since the AFB output port is externally accessible and intermediate output unit values go through this module, it is also possible to read internal unit output values.

## IV. HARDWARE ARCHITECTURE

When developing the computation algorithm in the previous section, specific hardware blocks existing in FPGA were considered beforehand: distributed memories, arithmetic units, logic and different types of interconnections. By doing this, we guarantee that the proposed implementation uses standard blocks with the aim of obtaining an optimal and efficient implementation on existing commercial hardware. This favors the portability to any FPGA, regardless of the manufacturer, or even a VLSI device.

The proposed architecture, shown in Fig. 5, is a generic architecture that can be arbitrarily extended in number of layers and units per layer as long as hardware resources are available. Actually, since layers reuse the hardware, the main limiting factor is the number of units in the biggest layer, and the number of input. This is especially beneficial to deep multi-layer neural networks. Each NPE acts as a neuron unit of the FFNN for each layer computation. At most, one NPE

will be reused as many times as layers exist in the SFNN. Weight values corresponding to all units that will be computed by the NPE are stored in its internal local memory, i.e. each NPE contains the weight values of one unit per layer, at most. In case of computation of layers with a lower number of units than the biggest layer, some units will be unused and the corresponding local memory will not be fully filled. The NPE architecture is designed to accumulate the partial sum of products of the current unit in the ALU accumulator register, and, when finished, the resulting value is moved into the corresponding SR register so that they can be shifted through the SR daisy-chain into the Activation Function Block (AFB) and, simultaneously, the NPEs can compute the next layer values by feeding back the AFB output value to the NPEs, or providing the output values in case of the output layer. The order of operations is defined by a Finite State Machine (FSM) block. The FSM controls the weight loading into memories, and the addressing of local NPE memory depending on the layer computation being carried out, the NPE usage depending on the layer, and the SR latching.

An important feature of this architecture is the modular structure and minimization of connections: adding more NPEs gives linear growth in the hardware occupation with most of the connections being internal in the NPE. The only external connections in an NPE are the addressing, memory write enable (we), and SR register connection with the previous and next SR. As seen in Fig. 5, the NPEs must be placed with alignment to the right, i.e. the right-most NPE must contain the last unit of each layer and the rest of units will be arranged in NPEs from right to left. As not all layers contain the same number of units, the left-most NPEs will not be used for certain layers. Fig. 5 shows the placement for a 3×3×2 FFNN where the left-most NPE only contains weights from the hidden layer since the output layer contains two units. At least, weight values for one unit in a certain layer are stored in one NPE memory. In case a local NPE memory contains weights for units from several layers, the addition of an offset to the NPE memory addressing is the only modification required to reuse the hardware architecture for different layers. The data input in the system is serially performed through the *DIN* port, the control system allows weight update using the same data port (including bias values), at any time. This feature is very useful when weight values need to be modified without device reprogramming, once the hardware system is running.

The total number of weights in an FFNN is given by Eq. 7a, which corresponds to the memory size required for weights in the whole design. However, since weights are distributed in different memories, as many distributed memory blocks as NPEs are required. The size of each distributed memory block is the key factor to properly optimize the resources in the NPE implementation. The maximum distributed memory size required for each NPE is given by Eq. 7b. Despite some NPEs may require less memory, all NPEs are defined according to the value given in Eq. 7b in order to maintain a regular hardware structure which eases the memory addressing.

$$\begin{aligned} a.\ Total\_Weights &= \sum_{i=0}^{L-1}(N_i+1)*N_{i+1} \\ b.\ Max\_Weights\_in\_NPE &= \sum_{i=0}^{L-1}(N_i+1) \end{aligned} \tag{7}$$

For such a versatile architecture, it becomes very important to be able to customize the FFNN implementation according to parameters which can be easily modified to generate the hardware definition of any FFNN. Thus, a set of configuration parameters is defined. Using these parameters, the synthesized FFNN hardware will be obtained. The required information to properly generate the FFNN structure is the following:

- Number of NPEs. Is the maximum number of units in a layer, considering all layers, in the FFNN, i.e. $NPE\# = max(N_i)$ with $i = \{1, \ldots, L\}$.
- Memory size per NPE. This is the maximum memory size required by any NPE in the FFNN as indicated by Eq. 7b.
- Weight bit size. It is important for the estimation of memory requirements and must be determined according to the required accuracy.
- Fractional part bit size: Together with the total weight size, this value must be chosen according to the accuracy requirements.
- Number of layers $L$.
- Number of inputs $N_0$.
- Number of units in each hidden layer $N_i$ ($i = \{1, \ldots, L-1\}$).
- Number of outputs $N_L$.

### A. NEURAL PROCESSING ELEMENT (NPE) DESCRIPTION

As shown in Fig. 5, a single NPE consists of three blocks: a RAM block with single-cycle read/write access for weight storage, an ALU, and a scratchpad register *SR*.

Having each NPE its own distributed RAM block allows for concurrent NPE operation. The RAM block of the i-th NPE must contain several weights' banks, one for each i-th unit of each layer in the FFNN. In order to estimate the final RAM block size for the NPEs, the bit size of data is also required: reducing the weight data size allows lower memory size. This issue is discussed in detail in Section V-D.

The ALU required for the arithmetic computations must perform the following operations: $P = 0$, $P = A * B$, $P = A*B+P$, $P = A$, where A and B are N-bit input data and P is the accumulator where resulting data are stored, also serving as data output of the ALU. The $P = A$ operation serves as an ALU bypass from *DIN* to the memory block required in a weight update operation.

The scratchpad register *SR* is the third component of the NPE. Each NPE scratchpad register is connected to the adjacent NPE scratchpad register forming a ring. The register contents can be loaded from the *din*(*SR*) input (through

the ALU), or from the adjacent scratchpad register on the left (using the *cin*(*SR*) port connected to the previous NPE). Data input source is selected by the *DSRC* input signal controlled by the FSM. The scratchpad register latches its content on the *dout*(*SR*) output port every clock cycle, acting as a rotating register in case of $DSRC = 1$. When the final sum of products of a certain unit is computed, the scratchpad register value is updated from ALU ($DSRC = 0$), which simultaneously occurs for all NPEs. After that, the FSM starts shifting them out to the Activation Function Block (AFB). In turn, the AFB output serves as input of the processing array to compute the next layer or provides final FFNN output values.

### B. ACTIVATION FUNCTION BLOCK (AFB) IMPLEMENTATION

The systolic nature of the proposed architecture makes it possible that just one Activation Function Block (AFB) is necessary to perform the neural computations of the whole FFNN. Despite its obvious impact in resource usage reduction, this fact enables an easy modification of the used AFB block, with a low impact on resource usage, opposite to hardware implementations where one activation function per unit is required. Thus, the hardware complexity of the single AFB block can be reasonably high with the sole consideration that its performance must be high enough to work at the same clock frequency that the NPE blocks, to avoid bottlenecking. Additionally, it is possible to implement FFNNs using different AFB for each layer by implementing as many AFBs as desired, and multiplexing them during the computation process. The following activation functions were implemented in this work: ReLU, Logistic Sigmoid and Hyperbolic Tangent.

#### 1) RELU

It is a relatively new type of activation function, becoming a trend in the last 10 years. Networks using the ReLU activation function can be trained faster and have sparser activations. The ReLU output is defined in the range $[0, \infty]$. As its implementation consists of a sign bit evaluation and one conditional signal assignment (Eq. 8), it is the simplest of the proposed activation functions and can be directly implemented in fixed-point arithmetic.

$$F(X) = max(0, X) = \begin{cases} X & \text{if } X \geq 0 \\ 0 & \text{if } X < 0 \end{cases} \tag{8}$$

#### 2) LOGISTIC SIGMOID

It is a classic differentiable function, used in networks trained with gradient descent methods. The logistic sigmoid is defined in the range $[0, 1]$. Two approximations of this activation function were implemented:

- *Classic piecewise-linear (PLA)* [37]. The PLA calculation procedure is shown in Fig. 6. It is based on shift and add operations, where every approximation is described by the line $y = Sx + B$, with coefficients $S$ chosen to be a power of two. As only comparison, addition and bitwise shift operations are used, the resulting implementation requires low hardware resource usage. A 9-line approximation was implemented due to its reduced MSE (as seen in Section V-C). It requires 16 constants to store, one array for comparison, and one array for constant addition. Due to the symmetrical nature of the sigmoid, the 9-line approximation has a precision of 18-line PLA. The implementation requires a 3 clock cycle pipeline.
- *Zhang second-order approximation* [38]. It requires a single-multiplication, as described in eq. 9, implemented using an ALU block and a bit shift (multiplications by the power of two are replaced by bit shift operation). This implementation can be obtained using a 4 clock cycle pipeline.

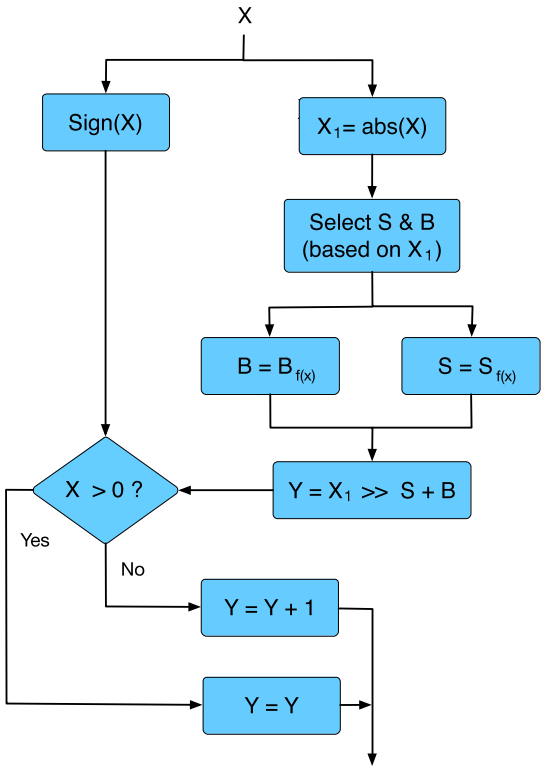


**FIGURE 6.** Shift and add approximation algorithm for the PLA approximation of the logistic sigmoid.

$$F(X) = \begin{cases} 2^{-1}(1 + 2^{-2} \cdot X)^2 & \text{if } -4 \leq X < 0 \\ 1 - 2^{-1}(1 - 2^{-2} \cdot X)^2, & \text{if } 0 \leq X < 4 \end{cases} \tag{9}$$

#### 3) HYPERBOLIC TANGENT

Defined in the range $[-1, 1]$, it is another classic differentiable function, used in networks trained with gradient descent methods. It was implemented using the second-order approximation function proposed by Kwan [39], based on the FPGA implementation by Rosado-Muñoz *et al.* [40]. The original expression for this approximation is described in Eq. 10, where V is the parameter controlling the slope of

approximation function.

$$F(X) = \begin{cases} sign(X) & \text{if } |X| > 2 \\ \dfrac{-X \cdot |X|}{V^2} + \dfrac{(2 \cdot X)}{V} & \text{if } |X| \leq 2 \end{cases} \quad (10)$$

To achieve maximum performance, this function has been slightly modified and parallelized using 2 ALU blocks with a 5-clock pipeline delay as depicted in Eq. 11.

$$F(X) = \begin{cases} sign(X) & \text{if } \quad |X| > 2 \\ \left(1 + \dfrac{X}{4}\right) \cdot X & \text{if } -2 < X < 0 \\ \left(1 - \dfrac{X}{4}\right) \cdot X & \text{if } \quad 0 < X < 2 \end{cases} \quad (11)$$

### C. NETWORK CONTROL SEQUENCES

According to the defined FFNN parameters (number of inputs, $N_0$, number of layers, $L$, number of neuron units in hidden layers, $N_i$ with $i \in \{1, \ldots, L-1\}$, and number of outputs, $N_L$), the Finite State Machine (FSM) was designed to automatically execute the required computation sequence.

**Algorithm 2** Control Sequence for Weight Loading

$A \rightarrow$ ALU Global Input
$ADDR \rightarrow$ Address bus
Set $P = A$ mode ( *All ALUs*) ▷ using *OPCODE*
A ( *All NPEs*) $\leftarrow$ DIN ▷ Weight from external input into A
Set *ADDR* ▷ Address of the weight
Wait for weight propagation ▷ Through ALU pipeline
*we* $\leftarrow 1$ ▷ Write Enable NPE
Wait one clock cycle

*(Repeat the sequence until all weights are stored in NPEs)*

For the sake of replicability, the control sequence of the load of weights is described in Algorithm 2, and the control sequence of the FFNN output computation is described in Algorithm 3. Note that, in both algorithms, some adjacent pseudocode steps are executed in parallel. Fig. 5 can be used as support to illustrate the use of the lines and buses referenced in both algorithms.

Most FSM operations are cyclically performed using two counters: one for the address generation, and another for the cycle count. Provided the FSM simplicity, with few continuously repeated states, the hardware occupation of the FSM is negligible compared to that of the NPEs.

### D. NUMBER OF CLOCK CYCLES OF EXECUTION

From the point of view of the number of clock cycles for execution, the SYMPA architecture presents a very efficient and deterministic behavior. Its systolic nature and mixed serial-parallel architecture permit to use pipelining efficiently: during the layer computation, input data are processed at a rhythm of one input per clock cycle, i.e. a $N_i$ input layer requires $N_i+1$ clock cycles to be computed. Depending on the implementation form of the hardware blocks, some additional cycles are needed for the interlayer delay: ALU ($T_{ALU}$), SR ($T_{SR}$) and AFB ($T_{AFB}$). Thus, after the last input of a layer $i$ has entered to the core, the next layer $i+1$ will be computed after $T_{ALU} + T_{SR} + T_{AFB}$ clock cycles. As an example, for a FFNN with $N_0$ inputs, $N_1$ units in the hidden layer and $N_2$ units in the output layer, the total computing time of the FFNN output of a input data pattern would be

**Algorithm 3** Output Computation Control Sequence

$A \rightarrow$ ALU Global Input
$ADDR \rightarrow$ Address bus
$AFB \rightarrow$ Activation Function Block output
$N_i \rightarrow$ Neuron units at layer $i$
*Computation of first layer*
Set $P = A * B$ mode ( *All ALUs*) ▷ using *OPCODE*
$ADDR \leftarrow 0$ ▷ bias reading for all units in first layer
$DIN \leftarrow 1$ ▷ All bias multiplicated by one
$P \leftarrow A * B$ ( *All ALUs*) ▷ bias in B loaded into P
Set $P = P + A * B$ mode ( *All ALUs*) ▷ using *OPCODE*
**for** each input data X(n) **do** ▷ $n = \{1, \ldots, N_0\}$
  Increment *ADDR* ▷ Adjust address to read weight
  A ( *All NPEs*) $\leftarrow$ DIN $\leftarrow$ X(n) ▷ Input to ALUs
  $P \leftarrow P + A * B$ ( *All ALUs*) ▷ MAC; B is the weight
**end for**
*DSRC* $\leftarrow 0$ ▷ Transfer ALU results to SR registers
*DSRC* $\leftarrow 1$ ▷ Set SR registers into Ring Mode

*(The next $N_1$ cycles the layer 1 outputs will be shifted out sequentially)*

*Computation of additional layers*
**for** each layer i **do** ▷ ( $i = \{2, \ldots, L\}$ )
  Set $P = A * B$ mode ( *All ALUs*) ▷ using *OPCODE*
  *ADDR* $\leftarrow 0$ ▷ bias reading for all units in first layer
  *DIN* $\leftarrow 1$ ▷ All bias multiplicated by one
  $P \leftarrow A * B$ ( *All ALUs*) ▷ bias in B loaded into P
  Set $P = P + A * B$ mode ( *All ALUs*) ▷ using *OPCODE*
  **for** each input $Y_{i-1}(n)$ **do** ▷ $n = \{1, \ldots, N_{i-1}\}$
    Increment *ADDR* ▷ Adjust address to read weight
    A ( *All NPEs*) $\leftarrow$ DOUT $\leftarrow$ AFB ▷ Feeding ALUs
    *(The output of activation function feeds $Y_{i-1}(n)$ to all ALUs)*
    $P \leftarrow P + A * B$ ( *All ALUs*) ▷ MAC; B is weight
  **end for**
  *DSRC* $\leftarrow 0$ ▷ ALU results to their SR registers
  *DSRC* $\leftarrow 1$ ▷ Set SR registers into Ring Mode
  *(Next $N_i$ cycles layer-i outputs will be shifted out sequentially)*
**end for**

*(In the next $N_L$ cycles the FFNN output will be shifted out sequentially)*

**TABLE 1.** Resource usage and maximum frequency of operation performance of several FFNN topologies.

| **Dataset** | Iris | Reduced MNIST | Reduced MNIST | Reduced MNIST | Full MNIST | Full MNIST |
|---|---|---|---|---|---|---|
| **Type of FFNN** | MLP | MLP | MLP | LR | AE | MLP |
| | 4×10×3 | 400×40×10 | 400×40×40×10 | 400×10 | 784×196×784 | 784×600×600×10 |
| **Activation Function** | Kwan HT [1] | Zhang LS [2] | Zhang LS [2] | Zhang LS [2] | ReLU | Kwan HT [1] + ReLU |
| **NPEs required** | 10 | 40 | 40 | 10 | 784 | 600 |
| **Slice registers** | 681 | 2481 | 2481 | 726 | 47871 | 36057 |
| **LUTs** | 681 | 2512 | 2512 | 668 | 45510 | 34838 |
| **BRAM36** | 5 | 20 | 20 | 5 | 392 | 600 |
| **DSP48E** | 12 | 41 | 41 | 11 | 784 | 602 |
| $f_{max}$ **(MHz)** | 490.849 | 498.691 | 498.691 | 498.691 | 550.570 | 490.849 |

[1] Kwan's second order approximation of the Hyperbolic Tangent activation function.
[2] Zhang's second order approximation of the Logistic Sigmoid activation function.

$(N_0+1)+(N_1+1)+N_2+2*(T_{ALU}+T_{SR}+T_{AFB})$. All layers account for bias calculation time adding one clock cycle, except the output layer, which does not need bias calculation. In general, the FFNN computation time can be described according to the number of clock cycles, C, described in Eq. 12. In case of the clock cycles required for weight loading, $C_{wload}$ provides the value.

$$C = \sum_{i=0}^{N_{L-1}} (N_i + 1) + N_L + (L \cdot (T_{ALU} + T_{SR} + T_{AFB}))$$
$$C_{wload} = \sum_{i=1}^{N_L} (N_i * (N_{i-1} + 1)) \quad (12)$$

## V. RESULTS

Once defined and characterized as shown in previous sections, the architecture was coded in VHDL. As a particular implementation case, we used Xilinx ISE Design Suite 14.7 for synthesis and implementation, using as target device the Xilinx XC7VX485T-2FFG1761 Virtex 7. The implementation was done using 18-bit word-length fixed point signed arithmetic (*DIN*, *DOUT* and *CIN*), with a fractional part of 12 bits. Nevertheless, different bit sizes were also tested.

In order to validate the architecture, four datasets were used. Three standard datasets, and one additional dataset aimed at a real case application. The datasets are:

- *Iris*. Dataset using four parameters per input pattern and three output classes.
- *Full MNIST*. Dataset with 784 (28×28) grayscale 8-bit pixels per sample and 10 output classes.
- *Reduced MNIST*. Dataset with 400 (20×20) grayscale 8-bit pixels per sample and 10 output classes.
- *MIT-BIH & AHA*. The MIT-BIH Malignant Ventricular Arrhythmia [41] and AHA (2000 series) [42] databases were processed as in [43], [44] to obtain 15 features. One output class identify two different types of rhythms (normal and abnormal).

Each one of the above classification problems were trained for different test topologies (MLP, AE and LR) with the scaled conjugate gradient descent algorithm (MLP used backpropagation) in Matlab R2017b using the Deep Learning Toolbox [45]. LR and MLP performance was calculated as recognition error and, in case of the autoencoder, the cost function was the reconstruction error (MSE). Finally, the selected topologies were:

- *Iris*: 4×10×3 MLP (HT)
- *Full MNIST*: 784×196×784 AE (ReLU)
  784×600×600×10 MLP (HT+ReLU)
- *Reduced MNIST*: 400×40×10 MLP (LS)
  400×40×40×10 MLP (LS)
  400×10 LR (LS)
- *MIT-BIH & AHA*: 15×20×20×1 MLP (HT)

The list above also indicates the activation function used in each implementation. The Logistic Sigmoid (LS), Hyperbolic Tangent (HT), and ReLU activation functions were used. The use of different activation functions aims to analyze their resource usage and how they affect the performance of the whole design.

The 784×600×600×10 MLP implementation of the *Full MNIST* classification problem was selected to illustrate the versatility of the proposed architecture for large FFNN, which permits to use different activation functions by layer, without impacting performance. In this case, the hyperbolic tangent was used for all layers except the output layer, which used the ReLU activation function.

### A. HARDWARE RESOURCES

Table 1 shows the resource usage of six different FFNN implementations. The number of DSP48E blocks matches the number of NPEs (each NPE uses one DSP48E block in its internal ALU) plus the number of DSP48E blocks required for the AFB. Different requirements in DSP48E blocks for the used AFB are seen in the table. The table clearly shows that the number of used LUTs and slice registers are a linear function of the number of NPEs.

Concerning distributed RAM memory usage (BRAM36 memory blocks), the required number is $NPE/2$ since all implementations use 18-bit word length and BRAM36 can thus accommodate a 36-bits word-width which is shared by

two NPEs. Depending on the word-length and distributed RAM block used (it varies from one device family to another, or from device manufacturer), this value could change. Using word-length above 18 bits would imply dedicating a single BRAM block per NPE. Implementation in word-length divisors of 36 is preferred, especially 18 and 9 bits, natively supported by manufacturer cores. In this case, as we chose 18-bits word-length for weights and BRAM in the selected device is 1024 in size, each NPE can accommodate up to 2048 weights.

It is also important to consider the number of weights to be stored in memory since the size of distributed RAM block also varies from device and manufacturer; a large number of weights stored in a single NPE can imply an extra block RAM per NPE. This is the case of the 784×600×600×10 FFNN in Table 1, where one BRAM block per NPE is required.

Considering multilayer FFNN, the resource usage of the implemented architectures does not significantly change as long as new layers contain the same or less units than previous layers. As an example, 400×40×10 and 400×40×40×10 implementations show nearly the same hardware requirements in terms of DSP48E, LUT, and BRAM36 blocks.

### B. MAXIMUM FREQUENCY OF OPERATION

Table 1 clearly shows that the proposed design achieves a very high frequency of operation across implementations. In fact, the core architecture can work at 550MHz, which is the limiting frequency of operation specified by Xilinx for DSP and BRAM slices in Virtex7. In other words, as the proposed architecture requires a reduced amount of logic and block slices, along with short delays in interconnections, the maximum frequency of operation of its core design is only limited by the frequency of operation of the used FPGA device technology. This is why the maximum frequency of operation for a large FFNN (784×196×784) implementation is 550MHz.

However, it can also be seen that implementations using the hyperbolic tangent activation function have a maximum frequency of 490MHz and those using the logistic sigmoid activation functions have a maximum frequency of 498MHz. This is because, although the core architecture frequency is only limited by the frequency limitation of its slices, the clock speed limiting block in the whole architecture is the AFB. In case of the 784×196×784 implementation, the maximum frequency of operation achieves 550MHz due to the implementation simplicity of its ReLU activation function, which avoids the AFB bottleneck and permits the maximum frequency of operation to match the maximum frequency of operation of the device. On the other hand, implementations using the hyperbolic tangent and sigmoid logistic activation functions present lower maximum frequency of operation. This must be taken into consideration when maximum throughput is required.

Being $N$ the number of NPEs, the peak performance of the whole design (using the ReLU activation function) is $f_{max} * N$ synaptic Operations Per Second (OPS). Note that this benchmark depends on the selected topology (e.g. in case of the 784×196×784 AE using the ReLU activation function, the total performance is: 550 $MHz$×784 $NPEs$ = 431.2 $GOPS$. As the maximum estimated number of NPEs in a Virtex-7 family device is 3600 (maximum number of DSP48E blocks included in a device), the proposed architecture claims to perform up to 1980 billion operations per second (GOPS) on the biggest Virtex-7 FPGA device. However, using a multi-chip approach, the size of the FFNN could be enlarged by a simple interface between different devices as few lines are required to connect NPEs with the FSM and other NPEs.

### C. ACTIVATION FUNCTION

Four Activation Function Blocks (AFB) have been implemented according to the approximations described in Section IV-B:

- *PLA LS*: Classic piecewise-linear approximation of the Logistic Sigmoid activation function.
- *Zhang LS*: Zhang's $2^{nd}$ order approximation of the Logistic Sigmoid activation function.
- *Kwan HT*: Kwan's $2^{nd}$ order approximation of the Hyperbolic Tangent activation function.
- *ReLU*: Rectified linear unit.

It is important to analyze the impact of the approximations in the accuracy results. Fig. 7 (top left and top right) shows the similarity in shape of the real-valued non-approximated function and the proposed approximations for Hyperbolic Tangent (Kwan approximation) and Logistic Sigmoid (PLA and Zhang approximations). The bottom left and bottom right show the absolute error for the approximated functions, which is always under 4.3% in case of Kwan approximation, and 2.1% in case of PLA and Zhang approximations.

All four approximations were independently implemented in hardware in order to verify the required hardware resources. The summary of the FPGA implementations is presented in Table 2. The upper half of the table reports the resource usage, i.e., the amount of Look Up Tables (LUTs), Registers and DSP blocks (DSP48E); no memory is used in any implementation. As it can be seen, the PLA LS approach triples the LUT usage of the Zhang LS approximation, whereas avoids using DSP48E slices (because only uses shifts and adds are used). On the other side, the ReLU implementation shows a really low use of resources because of its simplicity. Nevertheless, taking into account that only one AFB is needed for the whole implementation, it can be considered that, in general, the resource usage of the Activation Function Block (AFB) is negligibly small in all cases and then, does not influences the design complexity. However, the performance is important in order to obtain the fastest clock operation as possible. The bottom half of Table 2 shows the maximum clock frequency ($f_{max}$), which is clearly dependent on the pipeline design (except for ReLU), less pipeline stages decreases clock frequency and increases the amount of logic used. The table also shows the MSE and maximum error for the approximated functions. The number

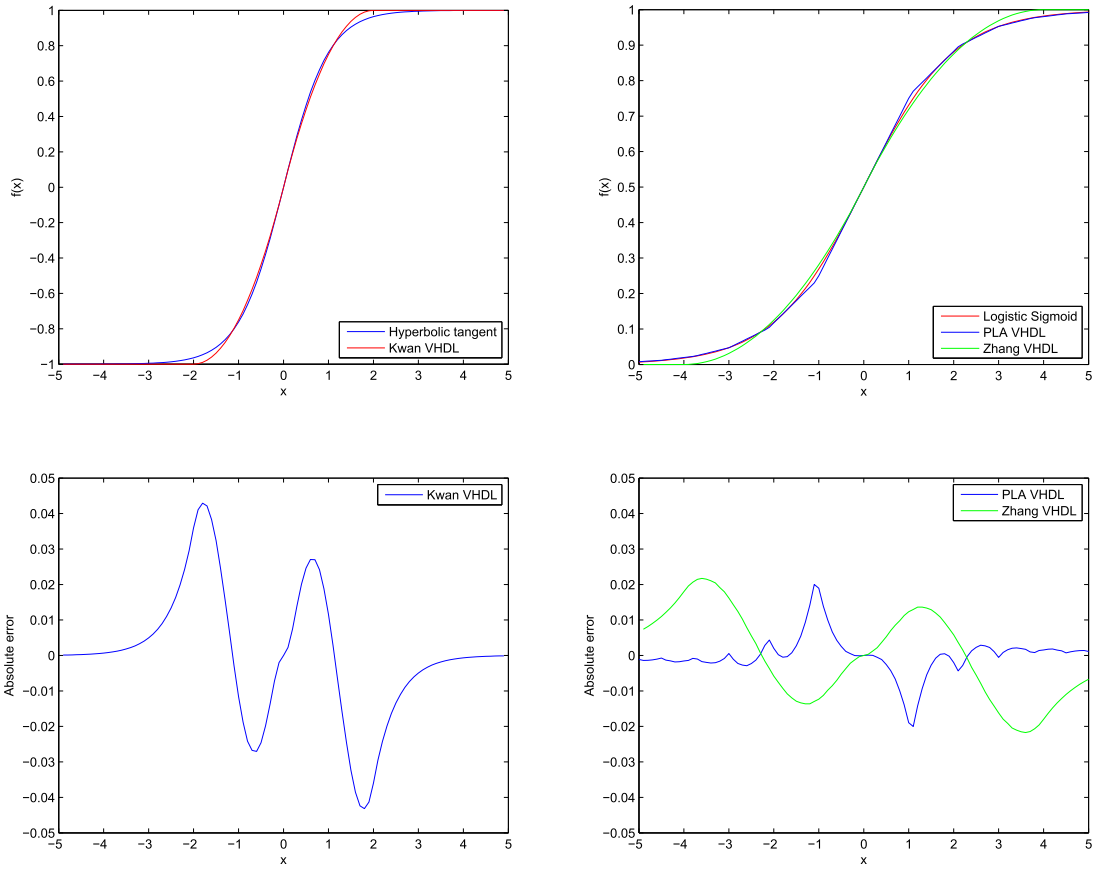


**FIGURE 7.** Top left: Comparison of hyperbolic tangent function (blue) and its Kwan approximation (red). Top right: Comparison of logistic sigmoid function (red) with its PLA (blue) and Zhang's second-order (green) approximations. Bottom left: Absolute error of the Kwan approximation to the hyperbolic tangent. Bottom right: Absolute errors of PLA and Zhang approximations to the logistic sigmoid.

**TABLE 2.** Resource usage, performance and error of proposed activation functions approximations. MSE and maximum errors compares to 64-bit floating-point with FPGA hardware implementation.

| | Activation function | | | |
|---|---|---|---|---|
| | **PLA LS** [1] | **Zhang LS** [2] | **Kwan HT** [3] | **ReLU** |
| **Registers** | 51 | 69 | 23 | 2 |
| **LUTs** | 165 | 53 | 26 | 1 |
| **DSP48E** | 0 | 1 | 2 | 0 |
| $f_{max}$ **(MHz)** | 270.8 | 498.7 | 490.8 | 550.6 |
| **Pipeline** $(T_{AFB})$ [4] | 3 | 4 | 5 | 1 |
| **MSE** | 3.10e-05 | 1.59e-04 | 3.27e-04 | 0 |
| **Max. error** | 0.020 | 0.021 | 0.043 | 0 |

[1] PLA approximation of the Logistic Sigmoid.
[2] Zhang's second order approximation of the Logistic Sigmoid.
[3] Kwan's second order approximation of the Hyperbolic Tangent.
[4] Pipeline length (number of pipeline stages).

of pipeline stages $T_{AFB}$ depending on the used AFB must be considered in the design of the FSM for proper data synchronization, as described in section IV-D.

Given the simplicity of the ReLU implementation, its clock frequency limitation comes from the delay in logic resources, which is 550MHz. On the other hand, Zhang LS and Kwan HT implementations show similar complexity, achieving an $f_{max}$ around 490MHz. Finally, an $f_{max}$ of 270MHz reveals that the PLA LS implementation is a hard bottleneck for the whole performance of the architecture. This illustrates the paramount importance of the AFB block design for achieving good performance in the proposed architecture. As a result, the PLA implementation was discarded for further analysis and not included in accuracy results.

### D. ACCURACY

In addition to performance and resource occupation, a very important issue lies in the accuracy of the proposed computing architecture since fixed-point arithmetic is used. An analysis of six different FFNN implementations was carried out, comparing the output of the neuronal network implementations with its 64-bits floating point PC Matlab-based counterpart. Using Matlab, it was found that Iris MLP implementation obtained 98.67% classification accuracy, both the MLP and LR implementations of the Reduced MNIST dataset obtained 95.2% classification accuracy, and the Full MNIST autoencoder reconstruction MSE was 0.21. Table 3

**TABLE 3.** Accuracy results for several FFNN classifiers with variable data size and using three different datasets. The MSE was calculated against 64-bit floating point Matlab implementation. Default format is Q6.12 unless other fractional part stated.

| FFNN | AFB | MSE | Max diff |
|---|---|---|---|
| Iris MLP 4×10×3 | HT | 3.86e-08 | 2.40e-03 |
| Full MNIST AE 784×196×784 | ReLU | 4.61e-09 | 4.73e-04 |
| Full MNIST AE 784×196×784 (Q9.9) | ReLU | 6.70e-07 | 3.60e-03 |
| Full MNIST AE 784×196×784 (Q12.6) | ReLU | 3.97e-05 | 2.99e-02 |
| Full MNIST MLP 784×600×600×10 | HT + ReLU | $<$ 1e-06 | 6.29e-03 |
| Full MNIST MLP 784×600×600×10 (Q9.9) | HT + ReLU | 6.00e-06 | 4.64e-02 |
| Full MNIST MLP 784×600×600×10 (Q12.6) | HT + ReLU | 4.77e-04 | 3.5e-01 |
| Reduced MNIST MLP 400×40×10 | LS | 2.06e-08 | 4.00e-04 |
| Reduced MNIST LR 400×10 | LS | 5.59e-06 | 2.00e-02 |

shows the MSE error of the reached classification accuracy for different FPGA implementations when compared to Matlab.

In order to evaluate the influence of fixed-point arithmetic, the Full MNIST dataset using both a 784×196×784 AE and a 784×600×600×10 MLP were implemented with 18-bit word-length and three different fractional part sizes: Q6.12, Q9.9 and Q12.6. As expected, the MSE error increases when bit size decreases (Table 3) but the MSE error is still negligible and thus, it can be considered that fixed-point arithmetic is not affecting the neural network results.

Note that the largest FFNN with ReLU units has a low MSE due to the ReLU activation function which provides mathematically exact results regardless of its fixed/floating point representation. This network was trained with regularization and dropout to obtain small weights ($[-1, 1]$) and avoid overflow problems with easy fixed-point implementation.

Concerning the Logistic Sigmoid, it is a very efficient data range limiter, limiting data amongst the layers into the $[-1, 1]$ range. Thus, when the implementation uses saturated arithmetic, the numeric data overflow does not become a problem and fixed-point arithmetic is valid. In fact, Table 3 shows that MSE error is mostly linked to the AFB implementation approximations done, rather than the fixed-point implementation versus floating-point implementation.

One of the interesting and useful properties of neural networks is their robustness to weight rounding. This fact can be used to optimize the hardware resources by reducing the memory size. Table 4 gathers the classification accuracy for different sizes of fractional parts using three implementations, including the large 3-layered 784×600×600×10 for the full MNIST. The obtained results show the fact that the FFNN using between 6 and 10 bits of fractional part have comparable performance to the FFNN using double precision floating-point weights. This is in the line with some studies [46], [47] showing that weight precision can be drastically reduced without compromising the network accuracy.

**TABLE 4.** Dependency between fractional part size of the weights and overall network accuracy for different FFNN implementations.

| | Classification Accuracy (in %) | | |
|---|---|---|---|
| **Bits** (fractional part) | **Iris MLP** 4×10×3 | **rMNIST** [1] 400×40×10 | **fMNIST** [2] 784×600×600×10 |
| **17** | 98.67 | 95.2 | 98.62 |
| **12** | 98.67 | 95.2 | 98.62 |
| **10** | 98.67 | 95.18 | 98.63 |
| **8** | 98.67 | 95.2 | 98.60 |
| **6** | 98.67 | 95.2 | 98.64 |
| **4** | 98.67 | 95.42 | 96.39 |
| **2** | 97.33 | 94.88 | 9.80 |
| **1** | 96.67 | 93.06 | 9.00 |

[1] Reduced MNIST dataset.
[2] Full MNIST dataset.

## VI. REAL CASE APPLICATION

To test the performance achieved by the proposed architecture on the FPGA against other platforms, a real case application is proposed. The aim of this application is to discern between the normal function of the heart and several pathologies as Ventricular Tachycardia (VT) and Ventricular Fibrillation (VF), amongst others. To feed the classifier, the input data (ECG signal) were preprocessed in several stages [43], [44]. The first step consisted of a baseline wandering removal (denoising), Fig. 8, using an 8th order IIR Butterworth bandpass filter with a response range from 1 Hz to 45 Hz. In the following stage, previous to a time-frequency Pseudo Wigner-Ville representation, a window signal alignment was required. The result was a bidimensional matrix image, which dimensionality was reduced with a kernel average, and, finally, the smoothed image was subsampled obtaining 15 values used as input data to the FFNN. Thus, the classification phase executed by the neural network is the last step to identify the normal/non-normal behavior of a human heart. For comparison purposes, we only analyzed the neural network.

A multilayer perceptron (MLP) was proposed, using the MIT-BIH Malignant Ventricular Arrhythmia [41] and

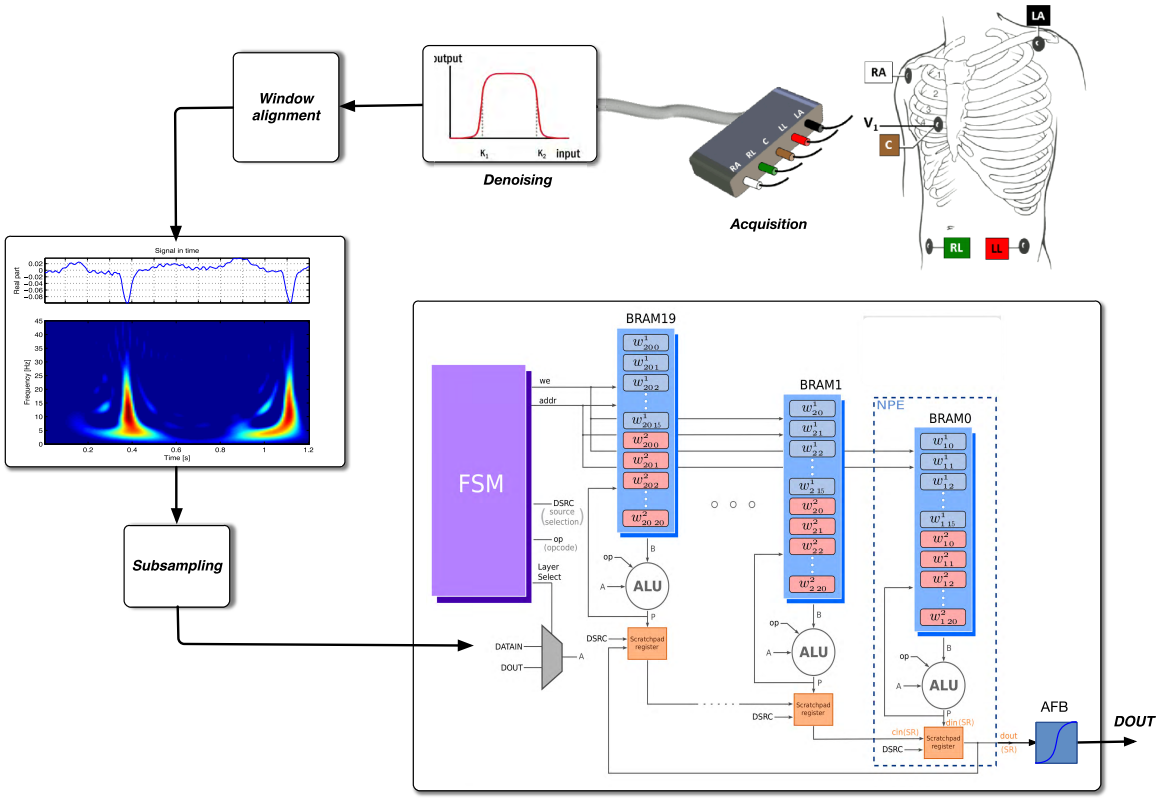


**FIGURE 8.** Block diagram of the real case application to classify between normal and pathological rhythms of the heart (especially, VT and VF).

AHA (2000 series) [42] database for training and testing. Two-thirds of the data were randomly chosen for training whereas the rest of the data were used for testing. The classifier was designed and trained using back-propagation using the Matlab 'Neural Networks' Toolbox. The hyperbolic tangent was selected as activation function. Finally, an MLP with 15 inputs, 20 neurons in the $1^{st}$ hidden layer, 20 neurons in the $2^{nd}$ hidden layer and one single output was obtained (15×20×20×1).

The neural network was implemented in the FPGA using the $2^{nd}$ order Kwan approximation of the hyperbolic tangent as activation function [43]. The complete architecture, including the AFB block, was configured to operate in Q6.12 fixed-point format.

The resource usage and performance is detailed in Table 5. It also shows the number of cycles required to process all 15 inputs. Results in the table show a reduced use of memory, high performance (490MHz) and low number of clock cycles (84) required for each 15 input processing, which means that 5.83 Msamples/second could be processed (171ns processing time).

To evaluate the results of the FPGA-implemented FFNN classifier, Table 6 details the Sensitivity (Se), Specificity (Sp), and Accuracy (Acc), as defined in Eqs. 13, 14 and 15, [44], where TP are the True Positives, FN the False Negatives, TN the True Negatives, and FP the False Positives. As seen in the table, results for FPGA fixed-point implementation slightly differ from floating-point implementations, showing that AFB approximation and fixed-point weights do not alter the classification results.

**TABLE 5.** Resource usage and performance for the FFNN MLP implementation for normal or pathological classification of the heart.

| | |
|---|---|
| **Datasets** | MIT-BIH Malignant and AHA 2000 |
| **Type of FFNN** | MLP 15×20×20×1 |
| **Activation Function** | Kwan HT |
| **Slice registers** | 1267 |
| **LUTs** | 1198 |
| **RAMB18** | 10 |
| **DSP48E** | 22 |
| $f_{max}$**(MHz)** | 490.849 |
| **#Clocks** | 84 |

$$Sensitivity(\%) = \frac{TP}{TP + FN} \times 100 \quad (13)$$

$$Specificity(\%) = \frac{TN}{TN + FP} \times 100 \quad (14)$$

$$Accuracy(\%) = \frac{TP + TN}{TP + FN + TN + FP} \times 100 \quad (15)$$

**TABLE 6. Performance comparison amongst platforms.**

| | **Matlab** [1] 64-bit floating | **LPC4337** [2] 32-bit floating | **FPGA** [3] Q6.12 fixed |
|---|---|---|---|
| Accuracy | 98.63 | 98.63 | 98.53 |
| Specificity | 98.88 | 98.88 | 98.82 |
| Sensitivity | 97.92 | 97.92 | 97.67 |
| Computation time | 43.79 $\mu$s | 3.30 $\mu$s (@120MHz) | 171 ns (@490MHz) |

[1] Intel Core i7-7700HQ CPU @2.8GHz, 64-bit Linux, 8 GB RAM.
[2] LPC4337 32-bit ARM Cortex-M4 microcontroller.
[3] Virtex 7 XC7VX485T-2FFG1761 FPGA.

Obviously, this FFNN processing can be implemented under different hardware technologies, being the main differences: speed, workload and power consumption. For comparison purposes, Table 6 also includes the performance of the same classifiers implemented in Matlab on a PC (Intel Core i7-7700HQ CPU @ 2.80GHz, 64 bit Linux OS, 8 GB RAM), and an LPC4337 microcontroller unit (MCU 32-bit ARM Cortex-M4 with floating point hardware arithmetic unit, running at 120MHz). As can be seen in the table, when implemented on an LPC4337 MCU running at 120MHz, the output is ready after approximately 400 clock cycles (implementing the algorithm in assembler) [48]. In this case, the processing time would be 3.3 $\mu$s @120 MHz. However, the same computation running in Matlab needs an average of 43.79 $\mu$s per input pattern. In any case, the FPGA implementation is able to perform the computation in a much shorter time, using a reduced hardware.

## VII. DISCUSSION

The proposed architecture provides great versatility: it can implement an arbitrary number of layers without hardware increase except in the RAM for weight storage, which is extremely useful in case of deep multi-layer neural networks. The outputs of intermediate layers can be externally accessed as the output of the AFB block where all units output are evaluated is connected to an external port. This feature can be used for network training algorithms or any other debugging purpose. Additionally, the weight values can be modified during execution by writing in the RAM memories, without device reprogramming.

Another relevant characteristic of this architecture is the use of a single Activation Function Block (AFB) for the whole FFNN. By serial feeding, this AFB block evaluates the non-linear neuron output function for the sum of products generated in each FFNN neuron unit. At first glance, it may appear that having a unique AFB block for the whole neural network may affect performance, but it allows to maximize performance. As only one block is necessary, the required amount of resources for this block is negligibly small (Section V-C) compared with other approaches using one activation function per neuron unit. Moreover, it is possible to implement several AFB blocks which can be switched in different layer computations; as an example, Table 1 shows the results of implementing the 784×600×600×10 MLP using the Kwan HT activation function in all layers except the output layer, which uses the ReLU activation function.

Table 2 illustrates the paramount importance of the AFB design in this architecture. Here, the differences in the maximum frequency of operation are exclusively due to the AFB implementation. ReLu implementation is the fastest, enabling the FFNN to work at 550MHz (in this case, the maximum frequency of operation of the FPGA modules: DSP48E and BRAM). On the other side, the PLA implementation of the Logistic Sigmoid makes the overall speed to fall down to 270MHz, which means that this block is bottlenecking the system. In turn, the Zhang LS and Kwan HT implementations show better performance, achieving around 490MHz, which indicates that both are bottlenecking the system but can maintain a high clock frequency. It is also important to consider the pipeline-delay for a different AFB ($T_{AFB}$).

**TABLE 7. Clock cycles required for different FFNN architectures. Weight loading and single sample computation times are shown.**

| **Topology** | **Weight loading** | **Output computation** |
|---|---|---|
| MLP 4×10×3 | 83 | 39 |
| MLP 400×40×10 | 16450 | 472 |
| LR 400×10 | 4010 | 411 |
| AE 784×196×784 | 308308 | 1786 |

The number of cycles of execution is very deterministic in this architecture. It is described by Eq. 12, where $T_{ALU}$, $T_{SR}$ and $T_{AFB}$ are additional cycles due to the propagation time in the ALU, Scratchpad Register and AFB block, respectively. As an example, Table 7 shows the number of cycles achieved for four different implementations, with $T_{ALU} + T_{SR} = 8$. The output computation time also takes into account the clock frequency. As an example, in case of the 784×196×784 AE using the ReLU activation function, the total performance is 550 *MOPS per NPE*×784 *NPEs* = 431.2 *GOPS* and the output computation time is $(1/550 \cdot 10^6) \cdot 1786 = 3.24\mu s$.

The comparison amongst hardware platforms reports a considerable acceleration when using the FPGA implementation of the proposed architecture. This study was conducted using a real case application and the same computation algorithm on all platforms, Table 6. Thus, when the FFNN is implemented in an FPGA, the output (the result of classifying the inputs) is generated after 84 clock cycles, which is 171ns *@490MHz*. In turn, a careful assembler codification of the algorithm in an LPC4337 32-bit ARM Cortex-M4 microcontroller requires 3.3 $\mu$s *@120MHz*. In this case, the sequential nature of the execution and the Von Neumann architecture restricts the efficiency of the computation,

**TABLE 8.** Comparison of the proposed FFNN hardware implementation with other approaches: Clock system frequency and normalized to the number of neurons in the FFNN (considering it as the sum of the hidden and output neurons). [1] Normalized values per neuron unit.

| Work | $f_{max}$ (MHz) | Clock [1] Cycles | Speed-up | DSP [1] blocks | Memory bits [1] | Registers [1] | LUTs [1] | FFNN type & size |
|---|---|---|---|---|---|---|---|---|
| **This work** | 490.8 | 2.05 | 256.1 | 0.53 | 4.4kb | 30.9 | 29.2 | MIT-BIH. 15×20×20×1 |
| **This work** | 490.8 | 1.02 | | 0.50 | 17.8kb | 29.8 | 28.8 | Full-MNIST. 784×600×600×10 |
| **This work** | 490.8 | 3.00 | | 0.92 | 0.4kb | 52.4 | 52.4 | Iris. MLP 4×10×3 |
| **Ferreira et al. [49]** | 300.0 | | 26.7 | 0.85 | 13.0kb | 699.3 | 627.3 | Iris. MLP 4×8×3×3 |
| **Vranjkovic et al. [50]** | 113.0 | | 48.3 | 1.00 | 3.1kb | 151.0 | | Average of several MLP-ANNs |
| **Suzuki et al. [51]** | 231 | | | 5.00 | | 1047.3 | 1033.0 | Autoencoder 4×2×4 |
| **Nedjah et al. [52]** | 20.3 | 10.47 | | | 6.9kb | | | MLP 220×24×10 |
| **Oliveira et al. [53]** | 77.8 | 6.43 | | | 2.0kb | 429.1 | | Iris. MLP 4×8×3×3 |
| **Huynh et al.(1) [18]** | 178.0 | 91.82 | | 0.31 | 11.1kb | 338.8 | 489.2 | Full-MNIST. 784×40×40×40×10 |
| **Huynh et al.(2) [18]** | 216.0 | 35.06 | | 0.32 | 11.7kb | 359.2 | 593.8 | Full-MNIST. 784×126×126×126×10 |
| **Zhai et al. [54]** | | | | 7 | 0.50kb | 715.7 | 1008 | MLP 12×3×1 |

in front of the parallel FPGA computation. Finally, the execution of the FFNN in Matlab running in a PC (Intel Core i7-7700HQ CPU) needs an average computation time of 43.79 $\mu$s *@2.80GHz*. As it can be seen, the FPGA accelerates the computation ×20 times than the LPC4337 MCU and ×256 times than a PC, not considering power consumption, which is generally much lower in an FPGA than MCU or CPU.

To enable hosting very large FFNN with reduced memory occupation, the proposed architecture uses fixed-point for arithmetic and weight storage. Nevertheless, it have been demonstrated (section V-D) that using 18-bit word-length for weights achieves a classification performance comparable to double precision floating-point weight values and computation (Table 6). Furthermore, a value between 6 and 10 bits for the fractional part reveals to be enough to achieve similar classification accuracy than floating point. This fact is very important due to the high number of weights existing in large FFNN, thus requiring a large memory for storage. However, this architecture allows to include more weights in the same number of NPEs by using their BRAM.

A direct comparison of the proposed architecture to other works in the bibliography is difficult. Different works take different approaches to the architectural solutions and authors tend to use relativistic metrics. In a search for similar works, Table 8 shows three implementations using the architecture proposed in this work and eight approaches made by other authors. In order to have comparable values, the hardware resources used in each implementation are normalized to the total number of units in the FFNN, e.g. a 4×8×3×3 MLP requires 14 neuron units, which means that reported total resource values are divided by 14 in order to obtain the normalized resources per unit.

In case of Ferreira and Barros [49], they achieved a ×36 speedup over a GCC compilation on a Linux PC, using a Intel Xeon @1.6GHz, for a 4×8×3×3 MLP (14 units). The use of memory, LUT and registers are significantly higher than our equivalent implementation of a 4×10×3 MLP (13 units).

Vranjkovic and Struharik [50] reported a coarse-grained accelerator on the same Virtex 7 platform as this work, they propose several FFNN implementations and provide average resource occupation results. The report 113MHz of maximum operating frequency and an average of ×48 speedup over Weka/PC software implementation. Note that our proposal works at 490MHz using the logistic sigmoid activation function, and accelerates by ×256. The proposal uses more DSP blocks and registers, while a slightly lower value for memory.

Suzuki *et al.* [51] showed a 4×2×4 autoencoder architecture achieving 231MHz of maximum clock frequency. In this case, all reported occupation values are remarkably higher than our proposal. Furthermore, we use a single clock data processing rate for the whole system, whereas [51] uses several clock signals.

Nedjah *et al.* [52] computed a 220×24×10 MLP (34 units) in 356 clock cycles, and our implementation would solve it in 276 clock cycles (33% less clock cycles). The normalized memory usage for our similar 15×20×20×1 MLP (41 units) is also lower.

Oliveira *et al.* [53] implemented a 4×8×3×3 MLP Iris problem using 90 clock cycles and 77.8MHz, whereas our architecture would do it using 51 cycles at 490MHz at lower memory usage.

Huynh [18] propose different FFNN implementation in their work. They used a similar approach to that of our work: they implement all neuron units of the largest layer and serial processing. For a 784×126×126×126×10 and 784×40×40×40×10 MLP for Full-MNIST dataset classification, i.e. 388 and 130 neuron units, respectively. They obtain a much lower clock frequency and clock cycle number, while being slightly better in memory and DSP, but using more registers and LUTs when comparing to our 784×600×600×10 (1230 units).

Finally, Zhai *et al.* [54] propose a 12×3×1 MLP (4 units) in a Xilinx Zynq SoC to detect and classify the gas sensor data with a processing time of 540ns. Our architecture uses less resources and achieves a ×7 speedup.

As summary, obtained results show remarkable benefits of using the proposed architecture to accelerate FFNN computation, providing a high-end computing platform with superior speed performance, being able to compute the output of large FFNNs much faster than other works in the bibliography. Furthermore, FPGA devices typically require less power than PC or MCU and require a small board to work, providing integration of online FFNN computation in multiple applications. This is especially important in our proposal, where a single chip solution is given, without any additional external memory or additional device which may act as bottleneck.

## VIII. CONCLUSIONS

The proposed SYMPA architecture exploits the fact that different types of FeedForward Neural Networks (FFNN) differ only in its interconnection schemes. With this, the computational procedure can be generally defined, no matter the number of inputs or outputs, hidden layers or hidden neurons per layer. It provides a modular procedure for the single chip FPGA implementation of any fully connected FFNN (MLP, AR, LR), no matter of the number of input, outputs, layers or neurons by layer, with the only limitation of the available resources in a device. Its systolic nature and pipelined design make it possible to obtain linear scalability in resource occupation when increasing the number of units in the FFNN, with no resource increase when adding more layers, except for weight memory storage. The architecture uses a single activation function for the whole FFNN, apart from the obvious resource savings, this fact allows customization options for the activation function model and intermediate unit outputs result readout. It is also possible to update weights during normal operation (no device reprogramming required). By using a mixed serial-parallel architecture based on Neural Processing Units (NPE) containing an ALU and a RAM block each, the resulting computation time is a linear function of the number of layers and number of inputs. However, the maximum clock speed is fixed and independent of the FFNN size, it is the number of clock cycles the changing value for different FFNN. Thus, gathering versatility, simplicity and high-performance, the proposed architecture design becomes a clearly viable candidate for its use in practical implementations with standard off-the-shelf hardware.

The hardware architecture combines concepts from matrix computation fundamentals, mixed serial-parallel computer architecture, and specific hardware availability in current FPGA devices as ALUs and distributed RAM. This architecture presents excellent scalability by replicating Neural Processing Elements (NPE), providing local interconnection among adjacent NPEs and reduced global control signals, thus reducing delays and optimizing clock frequency operation. The resource usage has a linear dependency with respect to the size of the largest network layer, i.e. the NPE number (section V-A). Thus, the system can be easily scaled by adding or removing NPE elements connected to a systolic ring with adequate modification of the FSM. Scalability is only limited by the availability of hardware resources though it is possible to create multi-chip FFNN by simple inter-chip connections.

A practical implementation in a Xilinx Virtex 7 FPGA device can host multiple-layer FFNN with up to 3600 units per layer without using external memory, obtaining a high concurrency in computation reaching up to 1980 Giga Operations Per Second (GOPS). The maximum clock frequency achieved is 550MHz using the ReLU activation function, and 490MHz using logistic sigmoid or hyperbolic tangent. It is important to remark that the maximum clock frequency only depends on the activation function used, since NPEs work (independently of the neural network size) at the maximum possible device speed. It is not the normal situation in other designs, where increasing the complexity means a decrease in clock speed. An important analysis of this work is related to the result that a reduced bit word-length for weights is valid for proper FFNN operation. Thus, since memory size can be a limiting factor in large FFNN, using reduced bit size for weights will allow storing more weight values in the same memory size. Other authors use external memory to store weights. However, by utilizing the on-chip memory, there is no RAM interface bottleneck, thus accelerating the whole design. In general, the architecture proposed in this work is significantly different from the aforementioned approaches due to the combination of matrix algebra and resource optimization.

Current research on similar architectures for matrix operations [55] suggests that the proposed design can be easily adapted for Recurrent Neural Networks and Restricted Boltzmann Machines, and additionally, used in combination with backpropagated-based on-chip learning methods.

One area of future work will be the adaptation of the architecture to work with layers containing more units than the number of available NPEs. It can be done by storing partial layer results in additional BRAM block memory and repeating the input feeding to the NPEs so that the result is obtained after several iterations (as a 'time vs. size' trade-off). It is also straightforward to adapt the architecture for very large FFNN by using multiple devices since the communication between chips would be very simple, expanding the application to any deep learning application where FFNN contain multiple layers with a large number of units per layer. Since the speed of operation is limited by the maximum chip frequency, future FFNN implementations in other devices would increase the performance.

## REFERENCES

[1] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong, "Energy-efficient CNN implementation on a deeply pipelined FPGA cluster," in *Proc. Int. Symp. Low Power Electron. Design*, New York, NY, USA, Aug. 2016, pp. 326–331.

[2] J. Zhu and P. Sutton, *FPGA Implementations of Neural Networks—A Survey of a Decade of Progress*. Berlin, Heidelberg: Springer, 2003, pp. 1062–1066.

[3] J. A. Clemente, W. Mansour, R. A. Ayoubi, F. Serrano, H. Mecha, H. Ziade, W. El Falou, and R. Velazco, "Hardware implementation of a fault-tolerant hopfield neural network on FPGAs," *Neurocomputing*, vol. 171, pp. 1606–1609, Jan. 2016.

[4] N. Nedjah, F. P. da Silva, A. O. de Sa, L. de Macedo Mourelle, and D. A. Bonilla, "A massively parallel pipelined reconfigurable design for M-PLN based neural networks for efficient image classification," *Neurocomputing*, vol. 183, pp. 39–55, Mar. 2016.
[5] S. Messalti, A. Harrag, and A. Loukriz, "A new variable step size neural networks MPPT controller: Review, simulation and hardware implementation," *Renew. Sustain. Energy Rev.*, vol. 68, Part 1, pp. 221–233, Feb. 2017.
[6] Z.-L. Tang, S.-M. Li, and L.-J. Yu, "Implementation of deep learning-based automatic modulation classifier on FPGA SDR platform," *Electronics*, vol. 7, no. 7, p. 122, Jul. 2018.
[7] E. Iranpour and S. Sharifian, "An FPGA implemented brain emotional learning intelligent admission controller for SaaS cloud servers," *Trans. Inst. Meas. Control*, vol. 39, no. 10, pp. 1522–1536, Oct. 2017.
[8] F. Ortega-Zamorano, J. M. Jerez, D. U. Muñoz, R. M. Luque-Baena, and L. Franco, "Efficient implementation of the backpropagation algorithm in FPGAs and microcontrollers," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 27, no. 9, pp. 1840–1850, Sep. 2016.
[9] C.-F. Juang and C.-Y. Chen, "An interval type-2 neural fuzzy chip with on-chip incremental learning ability for time-varying data sequence prediction and system control," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 25, no. 1, pp. 216–228, Jan. 2014.
[10] L.-W. Kim, "Deepx: Deep learning accelerator for restricted Boltzmann machine artificial neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 29, no. 5, pp. 1441–1453, May 2017.
[11] A. Dundar, J. Jin, B. Martini, and E. Culurciello, "Embedded streaming deep neural networks accelerator with applications," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 28, no. 7, pp. 1572–1583, Jul. 2017.
[12] C. Kyrkou, C. S. Bouganis, T. Theocharides, and M. M. Polycarpou, "Embedded hardware-efficient real-time classification with cascade support vector machines," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 27, no. 1, pp. 99–112, Jan. 2016.
[13] H. Soleimani, M. Bavandpour, A. Ahmadi, and D. Abbott, "Digital implementation of a biological astrocyte model and its application," *IEEE Trans. Neural Netw.*, vol. 26, no. 1, pp. 127–139, Jan. 2015.
[14] M. Bataller-Mompeán, J. M. Martínez-Villena, A. Rosado-Muñoz, J. V. Francés-Víllora, J. F. Guerrero-Martínez, M. Wegrzyn, and M. Adamski, "Support tool for the combined software/hardware design of on-chip ELM training for SLFF neural networks," *IEEE Trans Ind. Informat.*, vol. 12, no. 3, pp. 1114–1123, Jun. 2016.
[15] A. Tisan and J. Chin, "An end-user platform for FPGA-based design and rapid prototyping of feedforward artificial neural networks with on-chip backpropagation learning," *IEEE Trans. Ind. Informat.*, vol. 12, no. 3, pp. 1124–1133, Jun. 2016.
[16] Y. Zhou, W. Wang, and X. Huang, "FPGA design for PCANet deep learning network," in *Proc. IEEE 23rd Annu. Int. Symp. Field-Program. Custom Comput. Mach.*, May 2015, p. 232.
[17] R. Wang, C. S. Thakur, G. Cohen, T. J. Hamilton, J. Tapson, and A. van Schaik, "Neuromorphic hardware architecture using the neural engineering framework for pattern recognition," *IEEE Trans. Biomed. Circuits Syst.*, vol. 11, no. 3, pp. 574–584, Jun. 2017.
[18] T. V. Huynh, "Deep neural network accelerator based on FPGA," in *Proc. 4th NAFOSTED Conf. Inf. Comput. Sci.*, Nov. 2017, pp. 254–257.
[19] J. V. Frances-Villora, A. Rosado-Mu noz, M. Bataller-Mompean, J. Barrios-Aviles, and J. F. Guerrero-Martinez, "Moving learning machine towards fast real-time applications: A high-speed FPGA-based implementation of the OS-ELM training algorithm," *Electronics*, vol. 7, no. 11, p. 308, Nov. 2018.
[20] J. V. Frances-Villora, A. Rosado-Muñoz, and J. M. Martínez-Villena, M. Bataller-Mompean, J. F. Guerrero, and M. Wegrzyn, "Hardware implementation of real-time Extreme Learning Machine in FPGA: Analysis of precision, resource occupation and performance," *Comput. Electr. Eng.*, vol. 51, pp. 139–156, Apr. 2016.
[21] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Netw.*, vol. 61, pp. 85–117, Jan. 2015.
[22] 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, no. 12, pp. 3371–3408, Dec. 2010.
[23] C. Xing, L. Ma, and X. Yang, "Stacked denoise autoencoder based feature extraction and classification for hyperspectral images," *J. Sensors*, vol. 2016, Jun. 2016, Art. no. 3632943.
[24] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. 1, D. E. Rumelhart, J. L. McClelland, and C. PDP Research Group, Eds. Cambridge, MA, USA: MIT Press, 1986, pp. 318–362.
[25] D. P. Kingma and M. Welling, "Auto-encoding variational Bayes," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, Apr. 2014, pp. 14–16.
[26] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2007, pp. 153–160.
[27] C. C. Tan, *Autoencoder Neural Networks: A Performance Study Based on Image Reconstruction, Recognition and Compression*. Berlin, Germany: LAP Lambert Academic, 2009.
[28] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Comput.*, vol. 18, no. 7, pp. 1527–1554, 2006.
[29] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006.
[30] Y. Bengio and Y. LeCun, "Scaling learning algorithms towards AI," *Large-Scale Kernel Mach.*, vol. 34, no. 5, pp. 1–41, Aug. 2007.
[31] D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio, "Why does unsupervised pre-training help deep learning?" *J. Mach. Learn. Res.*, vol. 11, pp. 625–660, Feb. 2010.
[32] S. Dreiseitl and L. Ohno-Machado, "Logistic regression and artificial neural network classification models: A methodology review," *J. Biomed. Inform.*, vol. 35, nos. 5–6, pp. 352–359, 2002.
[33] G. Harshvardhan, N. Venkateswaran, and N. Padmapriya, "Assessment of glaucoma with ocular thermal images using glcm techniques and logistic regression classifier," in *Proc. Int. Conf. Wireless Commun., Signal Process. Netw. (WiSPNET)*, Mar. 2016, pp. 1534–1537.
[34] A. Maas, F. Rottensteiner, and C. Heipke, "Using label noise robust logistic regression for automated updating of topographic geospatial databases," *ISPRS Ann. Photogramm., Remote Sens. Spatial Inf. Sci.*, vol. III-7, pp. 133–140, 2016. [Online]. Available: https://www.isprs-ann-photogramm-remote-sens-spatial-inf-sci.net/III-7/133/2016/. doi: 10.5194/isprs-annals-III-7-133-2016.
[35] X. Zhu, Y. Nie, S. Jin, A. Li, and Y. Jia, "Spammer detection on online social networks based on logistic regression," in *Proc. Int. Conf. Web-Age Inf. Manage.*, Cham, Switzerland: Springer, 2015, pp. 29–40.
[36] C. Bishop, *Pattern Recognition and Machine Learning*, 1st ed. New York, NY, USA: Springer-Verlag, 2006.
[37] P. D. Reynolds, *Algorithm Implementation in FPGAs Demonstrated Through Neural Network Inversion on the SRC-6e*. Waco, TX, USA: Baylor Univ., 2005.
[38] M. Zhang, S. Vassiliadis, and J. G. Delgado-Frias, "Sigmoid generators for neural computing using piecewise approximations," *IEEE Trans. Comput.*, vol. 45, no. 9, pp. 1045–1049, Sep. 1996.
[39] H. K. Kwan, "Simple sigmoid-like activation function suitable for digital hardware implementation," *Electron. Lett.*, vol. 28, no. 15, pp. 1379–1380, Jul. 1992.
[40] A. Rosado-Muñoz, E. Soria-Olivas, L. Gomez-Chova, and J. Vila Francés, "An ip core and gui for implementing multilayer perceptron with a fuzzy activation function on configurable logic devices," *J. Universal Comput. Sci.*, vol. 14, no. 10, pp. 1678–1694, May 2008.
[41] S. D. Greenwald, "The development and analysis of a ventricular fibrillation detector," M.S. thesis, Dept. Elect. Eng. Comput. Sci., Massachusetts Inst. Technol., Cambridge, MA, USA, 1986. [Online]. Available: http://hdl.handle.net/1721.1/92988
[42] A. L. Goldberger, L. A. N. Amaral, L. Glass, J. M. Hausdorff, P. C. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley, "PhysioBank, PhysioToolkit, and PhysioNet: components of a new research resource for complex physiologic signals," *Circulation*, vol. 101, no. 23, pp. e215–e220, Jun. 2000. doi: 10.1161/01.CIR.101.23.e215 .
[43] A. Cortina, A. Mjahad, A. Rosado, M. Bataller, J. V. Francés, M. K. Dutta, and G. Vyas, "Ventricular fibrillation detection from ECG surface electrodes using different filtering techniques, window length and artificial neural networks," in *Proc. Int. Conf. Emerg. Trends Comput. Commun. Technol. (ICETCCT)*, Nov. 2017, pp. 1–5.
[44] A. Mjahad, A. Rosado-Muñoz, M. Bataller-Mompeán, J. V. Francés-Víllora, and J. F. Guerrero-Martínez, "Ventricular fibrillation and tachycardia detection from surface ECG using time-frequency representation images as input dataset for machine learning," *Comput. Methods Programs Biomed.*, vol. 141, pp. 119–127, Apr. 2017.

[45] R. B. Palm, "Prediction as a candidate for learning deep hierarchical models of data," M.S. thesis, Dept. Inform. Math. Model., Tech. Univ. Denmark, Lyngby, Denmark, 2012.
[46] J. Ott, Z. Lin, Y. Zhang, S.-C. Liu, and Y. Bengio, "Recurrent neural networks with limited numerical precision," *CoRR*, vol. abs/1608.06902, 2016. [Online]. Available: http://arxiv.org/abs/1608.06902
[47] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proc. 32nd Int. Conf. Mach. Learn. (ICML)*, Lille, France, vol. 37, 2015, pp. 1737–1746. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045118.3045303
[48] *ARM Cortex M4 Processor Technical Reference Manual Revision r0p1 Sec. 3.3.1*, ARM Limited, Cambridge, U.K., 2015.
[49] A. P. do A. Ferreira and E. N. da S. Barros, "A high performance full pipelined arquitecture of MLP neural networks in FPGA," in *Proc. 17th IEEE Int. Conf. Electron., Circuits Syst.*, Dec. 2010, pp. 742–745.
[50] V. Vranjković and R. Struharik, "Coarse-grained reconfigurable hardware accelerator of machine learning classifiers," in *Proc. Int. Conf. Syst., Signals Image Process. (IWSSIP)*, May 2016, pp. 1–5.
[51] 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. doi: 10.1371/journal.pone.0194049.
[52] N. Nedjah, R. M. da Silva, and L. de Macedo Mourelle, "Compact yet efficient hardware implementation of artificial neural networks with customized topology," *Expert Syst. Appl.*, vol. 39, no. 10, pp. 9191–9206, Aug. 2012.
[53] J. G. M. Oliveira, R. L. Moreno, O. de Oliveira Dutra, and T. C. Pimenta, "Implementation of a reconfigurable neural network in FPGA," in *Proc. Int. Caribbean Conf. Devices, Circuits Syst. (ICCDCS)*, Jun. 2017, pp. 41–44.
[54] X. Zhai, A. A. S. Ali, A. Amira, and F. Bensaali, "MLP neural network based gas classification system on Zynq SoC," *IEEE Access*, vol. 4, pp. 8138–8146, 2016.
[55] T. Iakymchuk, A. Rosado-Muñoz, M. B. Mompéan, J. V. F. Víllora, and E. O. Osimiry, "Versatile direct and transpose matrix multiplication with chained operations: An optimized architecture using circulant matrices," *IEEE Trans. Comput.*, vol. 65, no. 11, pp. 3470–3479, Nov. 2016.

**LEANDRO D. MEDUS** received the degree in bioengineering from the National University of Entre Rios, Argentina, in 2016. He is currently pursuing the Ph.D. degree in electronic engineering with the Universitat de Valencia, Spain. His research interests include digital design in FPGA, biomedical signal processing, machine learning, and embedded systems.

**TARAS IAKYMCHUK** received the M.Sc. Diploma degree from the Wroclaw University of Technology, Wroclaw, Poland, in 2011, and the Ph.D. degree from the Universitat de Valencia, Spain, in 2017, under the Group for Processing and Digital Design. He was in collaboration with research groups from Sevilla, Manchester, and Institute of Neuroinformatics, Zurich. His main research interests include embedded systems, neural networks, hardware learning, and bio-inspired computation. He created PlumerAI, an startup aimed at creating small devices incorporating cutting-edge artificial intelligence reasoning.

**JOSE VICENTE FRANCES-VILLORA** received the M.Sc. and Ph.D. degrees in physics from the Universitat de Valencia, Valencia, Spain, in 1994 and 2000, respectively, where he is currently an Associate Professor and a Researcher with the Department of Electronic Engineering. His work is related to digital signal processing, and its industrial and biomedical applications, real-time systems, and intelligent transport systems.

**MANUEL BATALLER-MOMPEÁN** received the M.Sc. degree in physics and the Ph.D. degree in electronic engineering from the Universitat de Valencia, Valencia, Spain, in 1984 and 1989, respectively. Since 1984, he has been with the Department of Electronic Engineering, Universitat de Valencia, where he is an Associate Professor with the Group for Digital Design and Processing. His research interest includes digital signal processing and its hardware implementation.

**ALFREDO ROSADO-MUÑOZ** received the M.Sc. and Ph.D. degrees in physics from the Universitat de Valencia, Spain, in 1994 and 2000, respectively. He is a member of the International Federation of Automatic Control. He is currently a Professor with the Department of Electronic Engineering, Universitat de Valencia. His work is related to digital hardware design (embedded systems) for digital signal processing, artificial intelligence and control systems, especially targeted for biomedical engineering, and bio-inspired systems. He also works on neuromorphic hardware and automation systems.