electronics

MDPI

Article

SoC Design Based on a FPGA for a Configurable Neural Network Trained by Means of an EKF

Juan Renteria-Cedano [1], Jorge Rivera [2,*], F. Sandoval-Ibarra [1], Susana Ortega-Cisneros [1] and Raúl Loo-Yau [1]

[1] Advanced Studies and Research Center (CINVESTAV), National Polytechnic Institute (IPN), Guadalajara Campus, Zapopan 45015, Mexico
[2] CONACYT–Advanced Studies and Research Center (CINVESTAV), National Polytechnic Institute (IPN), Guadalajara Campus, Zapopan 45015, Mexico
* Correspondence: riveraj@gdl.cinvestav.mx; Tel.: +52-33-3777-3600

Received: 30 May 2019; Accepted: 4 July 2019; Published: 8 July 2019

Abstract: This work presents a configurable architecture for an artificial neural network implemented with a Field Programmable Gate Array (FPGA) in a System on Chip (SoC) environment. This architecture can reproduce the transfer function of different Multilayer Feedforward Neural Network (MFNN) configurations. The functionality of this configurable architecture relies on a single perceptron, multiplexers, and memory blocks that allow routing, storing, and processing information. The extended Kalman filter is the training algorithm that obtains the optimal weight values for the MFNN. The presented architecture was developed using Verilog Hardware Description Language, which permits designing hardware with a fair number of logical resources, and facilitates the portability to different FPGAs models without compatibility problems. A SoC that mainly incorporates a microprocessor and a FPGA is proposed, where the microprocessor is used for configuring the the MFNN and to enable and disable some functional blocks in the FPGA. The hardware was tested with measurements from a GaN class F power amplifier, using a 2.1 GHz Long Term Evolution signal with 5 MHz of bandwidth. In particular, a special case of an MFNN with two layers, i.e., a real-valued nonlinear autoregressive with an exogenous input neural network, was considered. The results reveal that a normalized mean square error value of −32.82 dB in steady-state was achievable, with a 71.36% generalization using unknown samples.

Keywords: multilayer perceptrons; neural network hardware; kalman filters; field programmable gate arrays

1. Introduction

Artificial Neural Networks (ANNs) are architectures capable of learning and predicting the behavior of a system, as well as of data classification, where both functions can be carried out with proper training of the ANN [1]. The training process is basically an algorithm for setting the pondered values of all ANN interconnections, and for decreasing the error between real and estimated system outputs.

The most common training algorithms are Levenberg–Marquardt [2], which is a solution with a large number of mathematical operations, commonly feasible to be carried out on a personal computer, while backpropagation [3], and the Extended Kalman Filter (EKF) [4,5], a solution that requires a moderate number of mathematical operations and is suitable for implementation in hardware. A comparison between EKF and backpropagation is presented in [6], where the advantages of EKF over backpropagation are presented. Moreover, its effectiveness for assertive learning, its training speed, and its high convergence rate, are presented for the EKF in [7]. Recently, the Unscented Kalman

Filter (UKF) [8] has gained popularity with respect to the EKF due to its better stability and performance for the estimation of nonlinear systems, but its computational burden is considerable higher than that of the EKF. More recently [9], the Smooth Variable Structure filter (SVSF) [10] takes advantage of the sliding mode control theory [11]. The SVSF is characterized to be robust to model uncertainties, to provide a high rate of convergence, and to guarantee the convergence of the estimation error within a boundary layer. Moreover, it is an attractive method for training ANNs due to its convergence with a minimum number of epochs [10]. For an estimation example of a nonlinear system, the reader can refer to [12].

In the vast majority of applications, the training process is performed offline using measurements from the system under test, taking advantage of mathematical software and object-oriented programming languages for estimation of the optimal coefficients for the ANN [13]. However, in the case that the complete dataset is not available, and only samples are generated in real-time, an online training method is required. The online training method can easily be carried out with a personal computer (PC) or even with a digital signal processor (DSP). The use of a DSP can be restricted to moderate sized ANNs, and to data inputs with low sampling rates, while the use of a PC entails a lack of portability.

It is clear that there is a diversity of possible ANN configurations that can be implemented with software techniques in DSPs or PCs, where a hardware implementation is not suitable due to time consuming in design. Hardware implementation can be a suitable solution for an online learning based technique for portable real-time applications. Therefore, a hardware architecture that can be configured with the easiness of software is an appealing solution for dealing with a rapid hardware prototyping of ANNs. Despite the novelties of UKF and SVSF over EKF, the latter is still a popular training method for ANNs, as one can evidence in the recent literature [14–18]. Hence, the hardware implementation of ANNs trained with the EKF is an interesting problem. In this regard, there are some related works in the literature. In the work presented in [19], a glucose level controller for diabetes mellitus type 1 patients was designed. The algorithm was implemented with a FPGA that embeds a neural inverse optimal controller, in which the neural model is based on a Recurrent High Order Neural Network (RHONN). The RHONN was used to calculate the inverse optimal control law to obtain the insulin dose to be supplied. The architecture was proposed with fixed-point arithmetic operations with a 16-bit word length (1-bit to sign and 10-bit to fraction length), and the hyperbolic tangent function was developed through look-up tables, causing a precision error in the computation of matrix operations. In the work presented in [20], a RHONN identifier based on a Xilinx system using Virtex-7 FPGA was designed for a two-degree-of-freedom robot manipulator. Floating-point arithmetic operations were used, increasing in that way the accuracy in the mathematical computations; nevertheless, by using system generator software for hardware description of mathematical operations, more logical resources are used than those utilized with a HDL design, thus increasing hardware consumption and causing the clock frequency operation to be lower than that in a hardware developed using a hardware description language. In general, the above-mentioned works share a common disadvantage: if the neural network needs to be redesigned, then a new hardware design stage will be required.

At this point, it is evident that there are several interesting topics to be addressed by researchers, e.g., the training of ANN with the UKF or SVSF algorithms, where features as complexity of the algorithm, robustness, and convergence can be studied. Another topic for research is the implementation of ANNs and training algorithms in digital devices, which can include DSPs, PCs, FPGAs, System on Chip (SoC), or even graphical processing units, where features such as execution time, hardware resources, and fast prototyping can be explored.

In this work, the research interest is restricted to the fast prototyping of a configurable ANN based on a Multilayer Feedforward Neural Network (MFNN) scheme, and trained with the EKF for a FPGA implementation in a SoC environment. The SoC mainly relies on a microprocessor and a FPGA, where the configurability property can be carried out thanks to the microprocessor

that can configure the MFNN for reproducing several transfer functions for a particular choice of a MFNN without modifying the hardware implemented on the FPGA. This can be achieved by using a time-multiplexed perceptron, and by transmitting the number of inputs and neurons, and the type of threshold function to be synthesized to the FPGA via a serial communication link. The application domain of this proposal can range from the identification of dynamical systems, to the classification of objects in an image, and, due to the reduced size of SoCs, such applications can be carried out with an online training method, suitable for a real-time system. As a first stage of the proposed FPGA implementation, a Hardware In the Loop (HIL) simulation was carried out, where the corresponding forwardpropagation and Jacobian matrix subsystems were implemented on a FPGA, while the EKF training algorithm was implemented on a personal computer. This first stage is useful as a reference point for the second stage, or in the case of reduced resources. The second stage consists of a full FPGA implementation of the algorithm. Experimental tests were carried out for a black-box type identification of a Power Amplifier (PA).

The remainder of this work is organized as follows: In Section 2, the mathematical blocks involved, and their corresponding hardware implementation on FPGA are described. To validate the proposal, a HIL simulation with partial implementation results is shown in Section 3, as well as the full implementation on a FPGA. The experimental results for the HIL and the complete architecture are discussed in Section 4. Conclusions are presented in Section 5.

2. Configurable FPGA Implementation of MFNN

The designing of ANNs in FPGAs under the necessity of updating the designed architecture, for example, when an algorithm redesign is required, becomes in one of the most tedious tasks for electronic engineers. Hence, a hardware architecture capable of performing different ANN configurations without redesign is here proposed. This can be achieved by using a single time-multiplexed perceptron, and by transmitting the number of inputs and neurons, and the type of threshold function to be synthesized to the FPGA via a serial communication link. In this way, the perceptron is continuously configured for performing the functionality of each perceptron in the MFNN. Hence, in the case of implementing a different MFNN, instead of performing a hardware redesign, only the software instructions from the microprocessor are modified. Moreover, the perceptron itself is also time-multiplexed, i.e., the multiplications of the input signals with the corresponding weights are carried out with a single multiply–accumulate unit. An architecture with such capabilities must rely on a technique that uses multiplexers.

The main requirements are an architecture design using a standard Hardware Description Language (HDL), high numerical precision, and a simple communication protocol between the FPGA and a personal computer.

The first requirement is solved using Verilog HDL [21], which allows synthesizing hardware in different FPGA families; the second requirement is satisfied using the numerical precision Standard IEEE 754-2008 [22] with a 64-bit floating-point; and the last requirement is solved using the RS-232 UART [23] protocol with a configurable baud rate.

With such capabilities, the proposed hardware can be operated in three different modes:

- *Validation mode*, allows replicating the transfer function of the previously configured MFNN.
- *Training mode* uses the first dataset to process the EKF.
- *Testing mode (debugger)* extracts information that was processed and stored by the hardware, i.e., sums and threshold functions. The information is correlated by a personal computer for validation of the MFNN functionality. This step is performed once.

The extracted features for the parametrization of the MFNN are the following:

- the number of layers (up to 256);
- the number of perceptrons per layer (up to 32);

- selection of input vectors up to 4, each one with dimension 16; and
- selection of threshold function per layer, where the hardware has embedded the five most common threshold functions: linear, hard-limit, sigmoid, tangent hyperbolic, and Gaussian.

In the following, the basic background of MFNN based on EKF training is presented, together with the FPGA implementation proposal as a configurable neural network. Figure 1 shows a block diagram of the algorithm to be implemented on a FPGA, which serves as a guideline for subsequent subsections. For a detailed description of MFNN trained by means of the EKF, the reader can refer to [24,25].

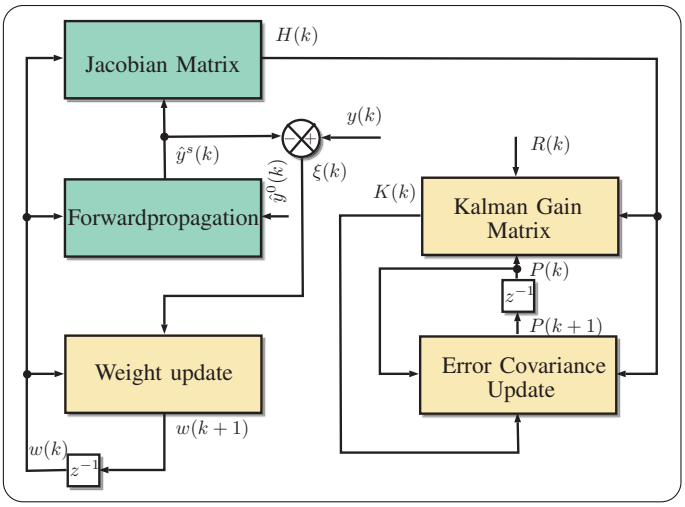

Figure 1. Block diagram of the EKF algorithm.

2.1. Forwardpropagation

The forwardpropagation block contains the mathematical equations for modeling linear or nonlinear systems, where the accuracy depends on model complexity. The ANN can model a physical phenomenon as a black-box by using I/O relations that yield results as low as -30 dB in a Normalized Mean Square Error (NMSE) metric, as shown in [26]. There are many ANN configurations with different neurons, layers, and threshold functions, with MFNN being the most widely used configuration for solving nonlinear problems.

The MFNN is a feedforward neural network, whose architecture is composed of two layers with external connections (input-output), where the rest of the layers (hidden) are used for mathematical processing for the extraction of features of its corresponding input signal. The common structure of the MFNN is shown in Figure 2, where the information flow between layers is evident.

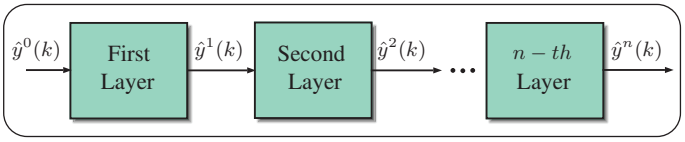

Figure 2. Block diagram of a basic MFNN architecture.

The operations involved in a given layer can be represented with a summation, which enters as an argument for a threshold function, allowing quantification of the neuron activation level in the corresponding layer. A general expression for the jth output in the nth layer is as follows:

$$\hat{y}_j^n(k) = f_j^n\left(\sum_{i=1}^{p_{n-1}}\left(w_{j,i}^n(k)\hat{y}_i^{n-1}(k)\right) + b_j^n\right) \tag{1}$$

where $\hat{y}^{n-1}(k)$ is an input vector for the nth layer, n can take values from the set {1,...,s} with s as the number of layers (hidden and output), and k is the time index. The weight vector is w^n, b^n is a bias vector, f^n defines a threshold function vector, i and j are indices that represent interconnections from perceptron i to perceptron j, and p_n identifies the number of perceptrons in the nth layer, with p_0 as the number of inputs to the network and p_s as the number of outputs of the network. It is common to multiply the bias vector by a weight matrix equal to $I_{j\times i}$, and to incorporate it into the summation. Hence, by analyzing Equation (1), it can be appreciated that each embedded perceptron in one layer performs the same number of mathematical operations (additions, products, and threshold functions). Therefore, Equation (1) can be implemented using a single processing unit (perceptron) resulting in a reduced hardware with complex mathematical capabilities.

In Figure 3, the forwardpropagation hardware is shown, where three main subsystems are identified in order to reduce design complexity. The first block is the serial communication hardware, the second is the routing hardware (composed of a finite state machine (FSM) for enabling multiplexers, demultiplexers and registers memories), and the last is a configurable neural network hardware composed of Controlpath and Datapath subsystems.

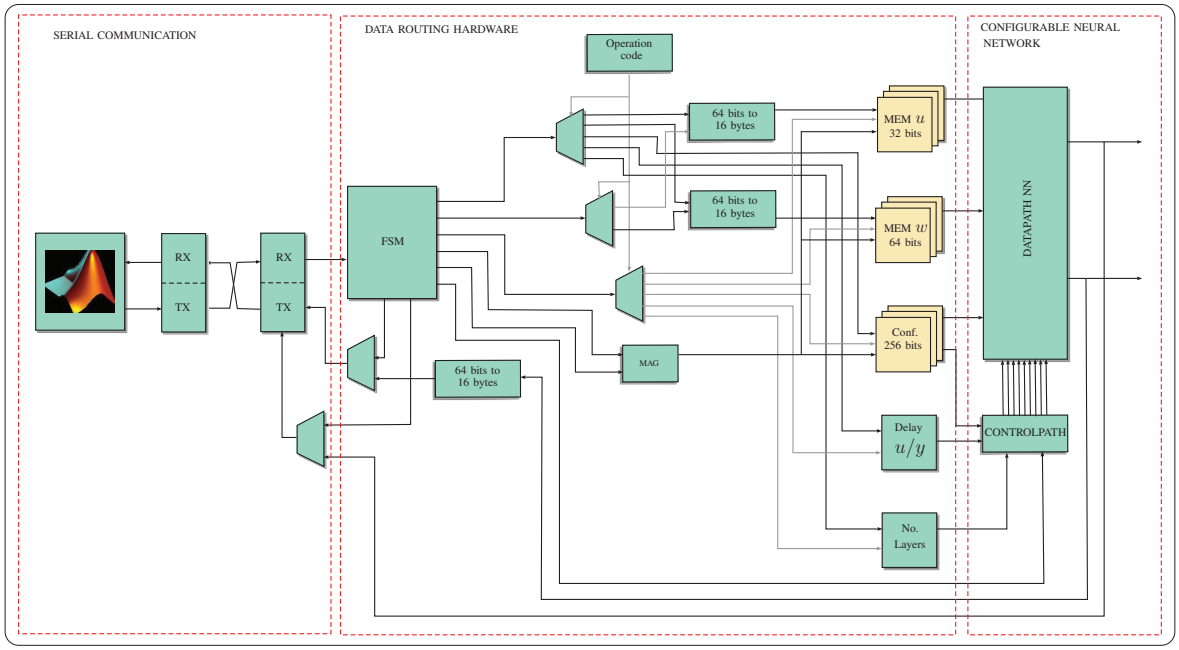

Figure 3. Configurable MFNN architecture.

The architecture setting process of the MFNN is performed using the RS-232 protocol with the following steps. First, the information transfer speed must be linked through a physical switch on the FPGA and through software on the personal computer. After that, the configuration parameters (extracted features of the MFNN) are encrypted in bytes and transmitted by means of a script written in Matlab software. When this process is successful, the Matlab software switches to idle while it waits for processed information from the FPGA.

The configurability feature of the MFNN is carried out by the FSM in charge of the routing hardware, where 80-bit information packages are received. These packages are characterized by head, body, and tail sections, as per usual. The head section contains an operation code, the body section contains the information to be stored, and the tail section identifies the end of the sequence. The operation code allows configuring the network connections, threshold functions, neurons, and the information to be processed. These parameters are used by the Controlpath subsystem, whose function is to limit the iterations and to enable the necessary routes for implementing the ANN transfer function defined by the user.

The Datapath is a mathematical hardware composed of a perceptron, dual port memories, a Memory Address Generator (MAG), and data routing hardware. The architecture is shown in Figure 4. The transferring of information is carried out from left to right. Input vectors are stored in memory blocks labeled as u and w, and data are extracted by means of the MAG. The information is routed by the multiplexers towards the perceptron for information processing. The output of the perceptron is stored in memory blocks that feed the inputs of the threshold functions. The obtained results are stored in another memory block. At this point, the Controlpath evaluates if all the layers are processed; in such case, the process is stopped, and results can be transmitted to the output. On the contrary, new inputs are routed through multiplexers towards the perceptron for more data processing.

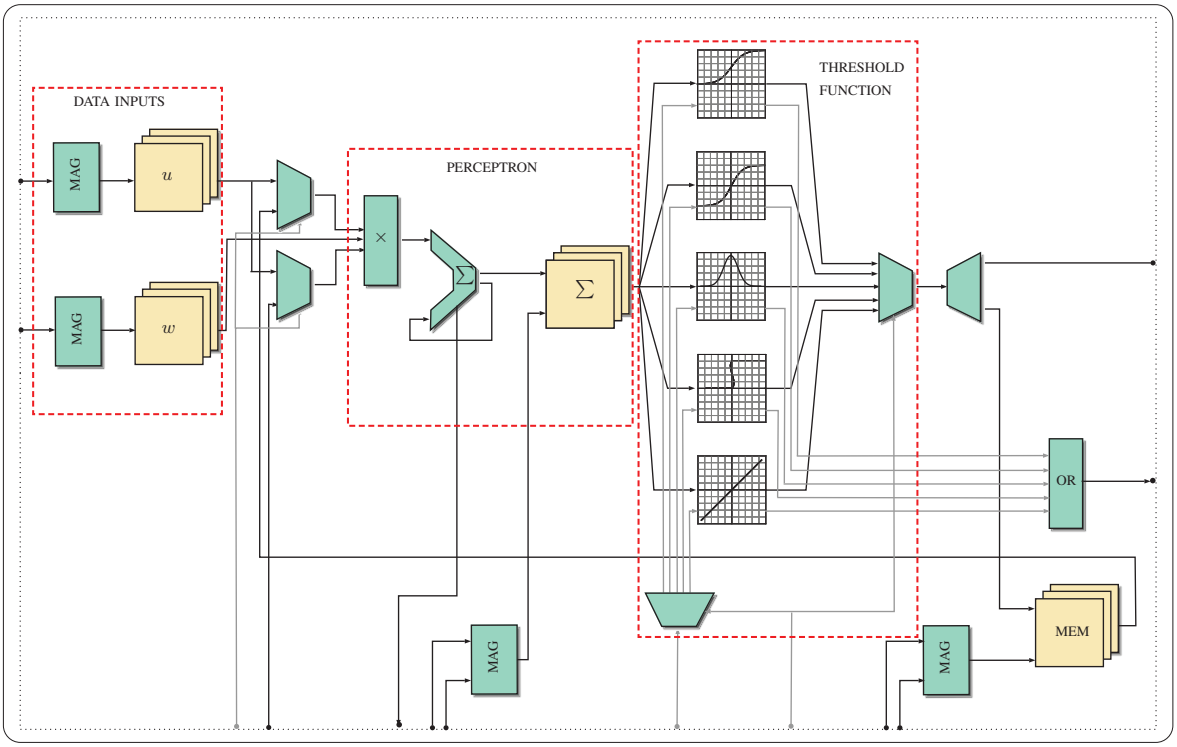

Figure 4. Hardware contained in Datapath block.

Implementing transcendental functions in FPGA is not a straightforward task. However, there are some alternative techniques for approximating transcendental functions, for example CORDIC [27], which is an iterative algorithm that approximates the results by means of vector rotations, where the accuracy of the approximation depends on the number of iterations executed. Another example is a Look-Up Table (LUT) [28], consisting of an array in which values for different inputs are stored, but the precision is compromised by size of the table. Another approximation method is carried out with linear segmentation, which is characterized by using linear functions with different slopes for minimizing the absolute error. A solution inspired by a piecewise polynomial approximation (PPA) method [29] is developed in this work. The PPA method relies on segmentation of the input values of the interval to be partitioned. Each of these segments is then approximated using a low-degree polynomial. In this work, Maclaurin series are applied to a segment, where the main nonlinearities of the functions are present, and the remaining segments are approximated in a linear fashion. The transcendental functions implemented by the PPA method are the following:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}},\ sigm(x) = \frac{1}{1 + e^{-x}},\ gauss(x) = e^{-x^2}$$

where the exponential function is approximated by the corresponding Maclaurin series:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!} \tag{2}$$

with $n = 9$. The approximated function for $tanh(x)$ is shown as an example below:

$$tanh(x) \approx \begin{cases} \dfrac{362880x + 60480x^3 + 3024x^5 + 72x^7 + x^9}{362880 + 181440x^2 + 15120x^4 + 504x^6 + 9x^8}, & |x| \leq 3.33 \\ 1, & x > 3.33 \\ -1, & x < 3.33. \end{cases} \tag{3}$$

The selection of the order in Equation (2) can be chosen according to the available FPGA resources and the desired accuracy, and the selection of the segments in Equation (3) can be easily identified with its corresponding graphic. Figure 5 shows a block diagram for the implementation of the function in Equation (3). Actually, the hardware shown in Figure 5 can evaluate any polynomial, since coefficients are stored in a memory block that feeds one input of a multiplier. The multiplier computes the powers of the input data, and then, it multiplies the result with the corresponding coefficient. This result is stored in an accumulator (there are two accumulators, one for the numerator and the other for the denominator). When the accumulators are finished, their outputs are processed by the divisor block and then stored in the output register. The logical resources reduction allows implementing Equation (3) with fewer processing elements.

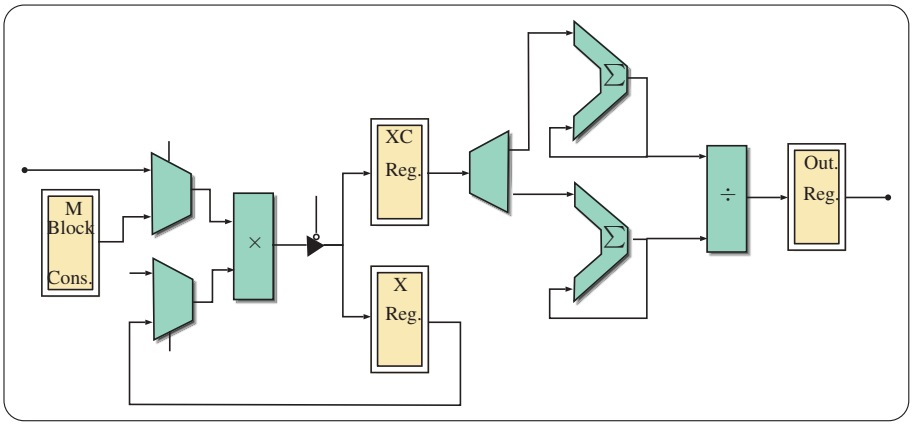

Figure 5. Architecture for the approximated hyperbolic tangent function.

The FSM in the Controlpath subsystem in Figure 3 for the configurable ANN is shown in Figure 6, where the loops and conditionals that limit the number of iterations required to process of each perceptron in each layer can be observed. When the iterations have finished, write signals for the output registers are enabled.

The output $\hat{y}^s(k)$ of the MFNN is an estimation of the real output $y(k)$, hence, the estimation error is defined as

$$\xi(k) = y(k) - \hat{y}^s(k).$$

Such estimation error is calculated for each dataset used to train the MFNN.

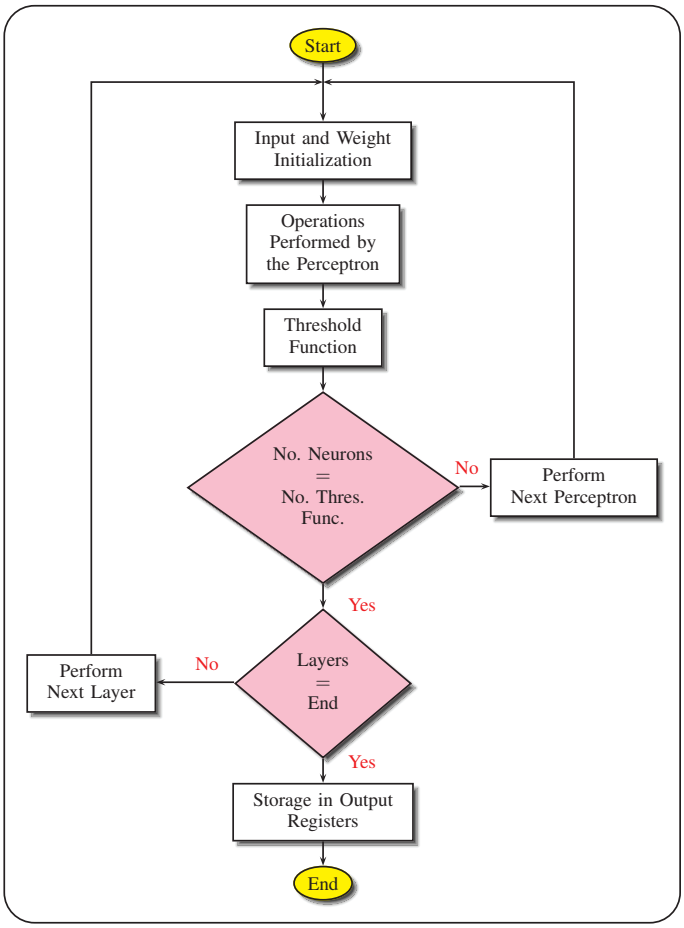

Figure 6. FSM designed flow diagram for the ANN.

2.2. Jacobian Matrix

The Jacobian Matrix block is calculated by partial derivatives of each individual output $\hat{y}_j^s(k)$ with respect to each weight w_j in the MFNN. The general expression is as follows:

$$H_{j,n}(k) = \left[\frac{\partial \hat{y}_j^s(k)}{\partial w_{l,r}^n(k)}\right]_{w(k)=w(k+1)} \tag{4}$$

where the weight vector in Equation (4) is evaluated with the predicted weight vector $w(k+1)$ given by Equation (8), $j = \{1,\ldots,p_s\}$, $n = \{1,\ldots,s\}$, with p_s as the total number of outputs, and l and r are defined in the following lines. Implementing algorithms in hardware with the ability to solve the Jacobian matrix for a MFNN with any configuration is extremely complex, since the number of equations to solve increases proportionally with the size of the MFNN. The proposed architecture is limited to solving MFNNs with up to $s = 3$ layers. For that, $w_{l,r}^n(k)$ is represented as $w_{l,r}^n(k) = \left[w_{l_1,r_1}^1(k)\ w_{l_2,r_2}^2(k)\ w_{l_3,r_3}^3(k)\right]$, with $w_{l,r}^n(k)$ as a weight vector containing all weights (including bias) between layers n and $n-1$, where the zero layer is the input layer. Moreover, $w_{l_1}^1(k) = \left[w_{l_1,0}^1(k)\ w_{l_1,1}^1(k)\ \ldots\ w_{l_1,p_0}^1(k)\right]$, $w_{l_2}^2(k) = \left[w_{l_2,0}^2(k)\ w_{l_2,1}^2(k)\ \ldots\ w_{l_2,p_1}^2(k)\right]$, $w_{l_3}^3(k) = \left[w_{l_3,0}^3(k)\ w_{l_3,1}^3(k)\ \ldots\ w_{l_3,p_2}^3(k)\right]$, $l_1 = \{0,1,\ldots,p_1\}$, $r_1 = \{0,1,\ldots,p_0\}$, $l_2 = \{0,1,\ldots,p_2\}$, $r_2 = \{0,1,\ldots,p_1\}$, $l_3 = \{0,1,\ldots,p_3\}$, $r_3 = \{0,1,\ldots,p_2\}$. A zero in the subindex of a weight

corresponds to a bias value. Hence, the Jacobian matrix in Equation (4) can be rewritten of the following form:

$$H_{j,n}(k) = \left[\frac{\partial \hat{y}_j^3(k)}{\partial w_{l_1,r_1}^1(k)} \; \frac{\partial \hat{y}_j^3(k)}{\partial w_{l_2,r_2}^2(k)} \; \frac{\partial \hat{y}_j^3(k)}{\partial w_{l_3,r_3}^3(k)} \right]_{w(k)=\hat{w}(k+1)} \tag{5}$$

The particular expressions for computing the Jacobian matrices in Equation (5) for a first, second, and third layer are as follows:

$$\begin{aligned}
H_{j,1}(k) &= \left[\frac{\partial \hat{y}_j^3}{\partial w_{l_1,r_1}^1(k)} \right] = \dot{f}_j^3 \left[\sum_{v=1}^{p_2} w_{j,v}^3 \dot{f}_v^2 w_{v,l_1}^2 \right] \dot{f}_{l_1}^1 f_{r_1}^0 \\
H_{j,2}(k) &= \left[\frac{\partial \hat{y}_j^3}{\partial w_{l_2,r_2}^2(k)} \right] = \dot{f}_j^3(\cdot) w_{j,l_2}^3 \dot{f}_{l_2}^2(\cdot) f_{r_2}^1(\cdot) \\
H_{j,3}(k) &= \left[\frac{\partial \hat{y}_j^3}{\partial w_{l_3,r_3}^3(k)} \right] = \begin{cases} \dot{f}_{l_3}^3(\cdot) f_{r_3}^2(\cdot), & j = l_3 \\ 0, & otherwise \end{cases}
\end{aligned}$$

where $\dot{f}$ is the derivative of the threshold function f. Figure 7 shows the hardware required for implementing the Jacobian Matrix block, where input X represents the summations of each perceptron.

Remark 1. *The derivative of the hard limit function is considered zero for practical purposes.*

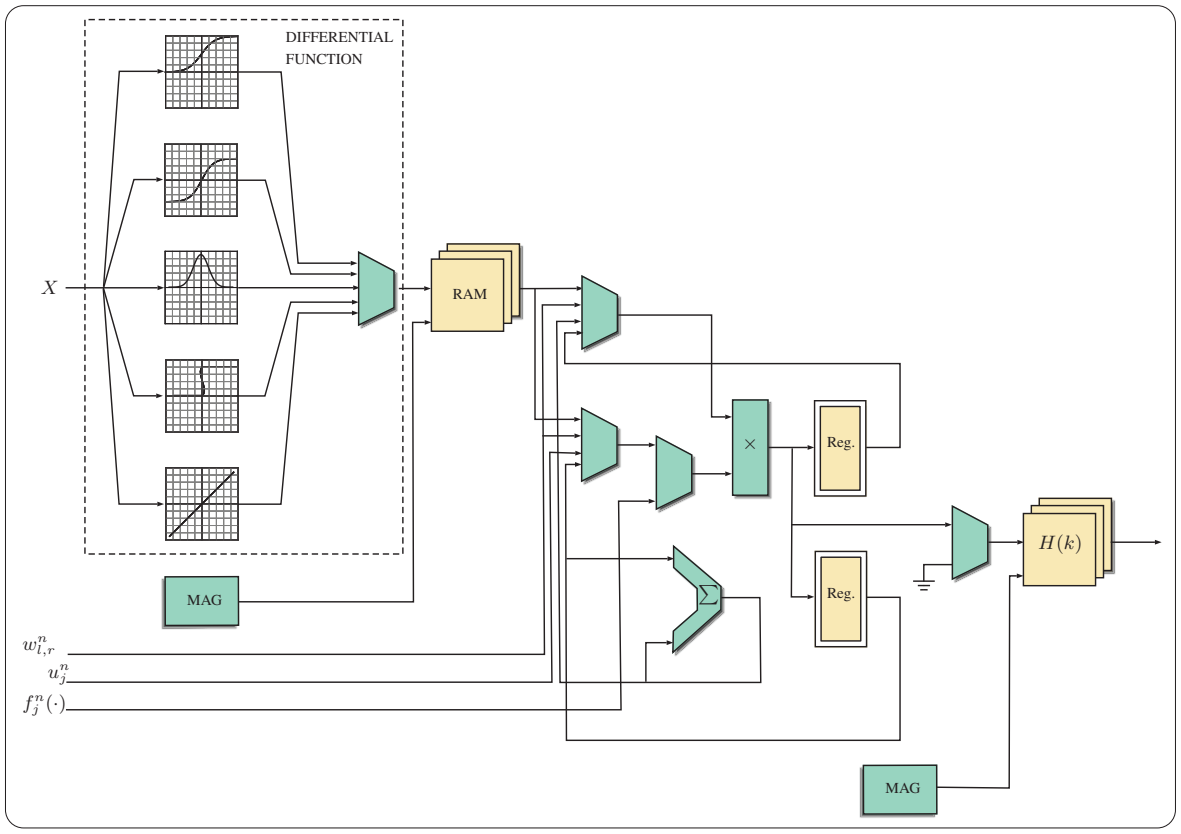

Figure 7. Jacobian matrix architecture.

2.3. Kalman Gain and Error Covariance Matrices

The operations involved in computing the Kalman gain create a large computational burden, which increases with the number of connections in the MFNN. The matrix operations are given by the following expressions

$$K(k) = P(k)H^T(k)\left[R(k) + H(k)P(k)H^T(k)\right]^{-1} \quad (6)$$

$$P(k+1) = P(k) - K(k)H(k)P(k) + Q(k) \quad (7)$$

where Equation (6) is the Kalman gain matrix, and Equation (7) is the update for covariance error.

Explicit implementation of matrix equations causes a waste of logical resources that generates a solution to a particular neural network. In this work, basic mathematical operations for computing the Kalman gain and the covariance error equations are used, resulting in a proposed hardware that is able to solve any matrix equation based on basic operations such as matrix multiplication, matrix adder–subtractor, and matrix inverse. Matrix multiplication is shown in Figure 8, in which an input interface allows controlling read–write access to memories or the multiplexing of inputs.

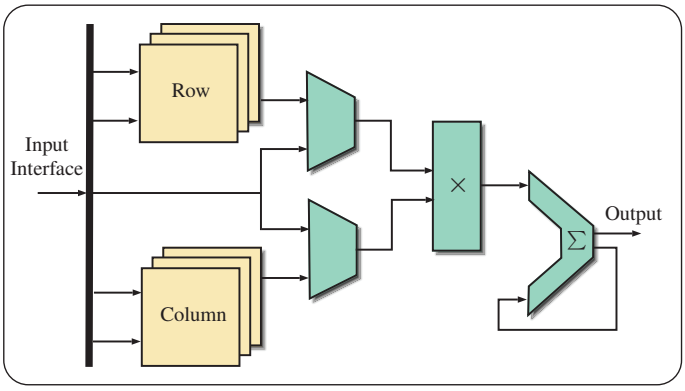

Figure 8. Matrix multiplier architecture.

The FSM for matrix multiplication allows configuring the operation mode. The first mode is a special case given by $H(k)P(k)H^T(k)$, i.e., a quadratic form. The second mode allows performing scalar product between two vectors, and the third mode solves matrix multiplication of two structures defined by the user, where the last two modes admit a maximum size of 4096 elements. The resulting matrix is organized in rows. Figure 9 shows an adder–subtractor hardware for a maximum size of 4096 elements.

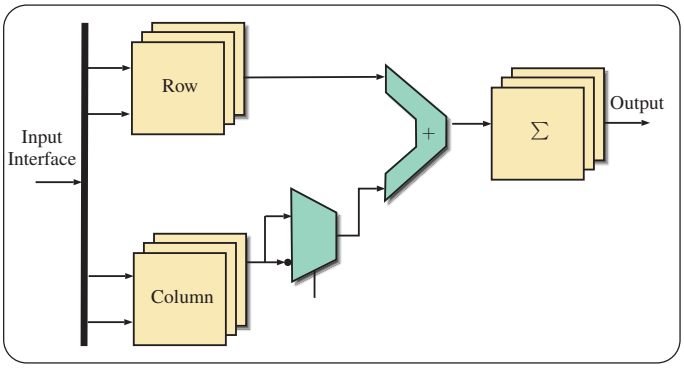

Figure 9. Matrix adder architecture.

Finally, hardware with the ability to obtain the inverse of a matrix is necessary for solving the Kalman gain in Equation (6). Analyzing the matrix operation performed in Equation (6), the matrix to be inverted, here denoted by $M(k)$, has the following structure:

$$M(k) = R(k) + H(k)P(k)H^T(k).$$

For an ANN with two outputs, this matrix results of dimension 2×2. The matrix inversion problem is solved by using the classical solution, i.e., the matrix adjoint divided by its determinant; for 2×2 matrices, this solution is well-known. The implementation of such a solution is shown in Figure 10, where the matrix elements are stored in register blocks with two opposite sign outputs. The multiplexers with two inputs, route the registers outputs to the multiplier block to perform the determinant of matrix $M(k)$ ($detM(k)$), storing the result in the adder. If the determinant is zero, the error flag is enabled, otherwise the enable signal of the divider block is set to one, allowing the matrix inversion to be performed. The multiplexer with four inputs takes the matrix elements to perform the adjoint of matrix $M(k)$ ($adjM(k)$). Finally, the hardware performs the operation $1/det(M(k)) \times adj(M(k))$, providing one matrix inverse value at a time.

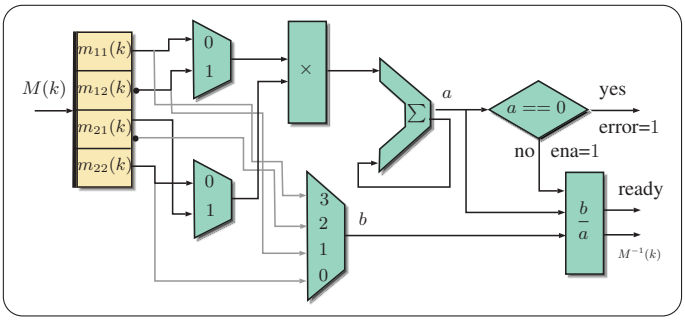

Figure 10. Matrix inverse architecture.

Employing the last three architectures, Equations (6) and (7) are processed by pairs of matrices, with their results stored in temporary memories to be employed later for the subsequent matrix operations until the equation is completed. The control of the matrix equation must respect the grouping signs and operand hierarchies.

2.4. Weight Update

The weight update consists of updating the new value for $w(k)$ according to the following equation

$$w(k+1) = w(k) + K(k)\xi(k). \tag{8}$$

The corresponding architecture is composed of an FSM that configures the forwardpropagation block for reading and writing the $w(k)$ matrix.

2.5. Stop Criteria

Stopping the training when a successful goal is achieved can be carried out with any of these methods: epoch ended, Mean Square Error (MSE), and NMSE. The first method stops the training when the epoch quantity is finalized, while the MSE and NMSE methods continue processing information and updating the weights until the error estimate is lower than the required error. An increased estimated error means that the neural network is being overtrained and the weight values will not converge, thus detecting a special case for stopping the training is detected. In this architecture,

the stop criterion for training is by means of the NMSE method, which is carried out in a validation process using a dataset different from the training process. The NMSE index is determined as follows:

$$NMSE = 10log\left[\frac{\sum_k |y(k) - \hat{y}^s(k)|^2}{\sum_k y^2(k)}\right].$$

The validation process stops the training to perform the NMSE using the actual value of $w(k)$.

Remark 2. *The index NMSE was used due to the application example. In the area of PA for radio frequency applications, the NMSE index is a commonly used metric, but, in general, any other index can be used for stopping the training phase.*

3. FPGA Implementations

A partial FPGA implementation was carried out by means of an HIL setup. The objective of this setup was to test the FPGA implementation of the forwardpropagation and Jacobian Matrix blocks, while the Kalman gain, the covariance error, and the weight update computations were performed in Matlab software. This first setup was useful as a reference point for the second setup, which consisted of the full FPGA implementation of the algorithm, as shown in Figure 1, which is also presented in this section.

The neural network under test for both implementations is a real-valued nonlinear autoregressive with exogenous input neural network (RVNARX) [30], which is a special case of a two-layer MFNN, and configurable with parallel or serial to parallel inputs. The parallel inputs mean that the architecture requires the estimated output similar to the extra input of the MFNN, and the serial to parallel inputs mean that the architecture uses input–output measurements from the PA as MFNN inputs. The serial to parallel inputs (16 neurons with tangent hyperbolic functions for the first layer, and two neurons with linear activation functions for the second layer) were configured for the RVNARX.

3.1. Implementation Based on HIL Technique

HIL [31] is a technique used for developing and testing embedded systems in real-time. The HIL is an effective platform for transferring information between the embedded system and a personal computer. The objective was to validate complex algorithms by splitting the execution of the algorithm across two or more devices. The HIL simulation was carried out with an FPGA Virtex-6 by Xilinx that executes the forwardpropagation and Jacobian Matrix blocks, and a personal computer that executes the Kalman gain, covariance error, and weight update. An illustration of the HIL setup is shown in Figure 11.

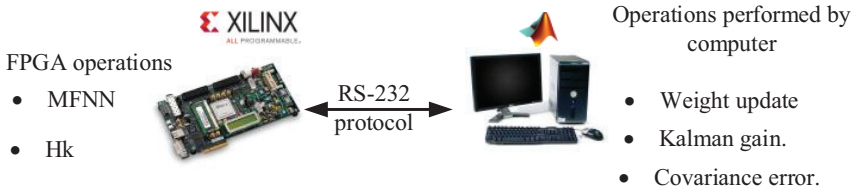

Figure 11. HIL performed by FPGA and computer.

For the HIL simulation, the devices must be prepared by pairing the communication speed and transmitting the initialization hardware parameters through Matlab scripting. After that, the communication frame consisted of 66 bits, where 64 bits were for data, 1 bit for start, and 1 bit for end, yielding to an overhead of $3.\overline{03}\%$.

3.2. Full FPGA Implementation

The complete architecture implemented in a Zynq-7 ZC706 evaluation kit is presented in Figure 12; the ARM microprocessor was incorporated to take advantage of the high-level programming for controlling the DDR3 memory, and the process flow for matrix operations. The ARM microprocessor can communicate with hardware blocks by means of the AMBA AXI4 interface, which is based on handshakes, where the AXI4 protocol was adapted to be compatible with the interface. The AXI4 interface allows connecting hardware with different operating frequencies, thus increasing the execution speed of less complex algorithms implemented in hardware. Therefore, the AXI4 interconnects different operating frequencies without loss of information.

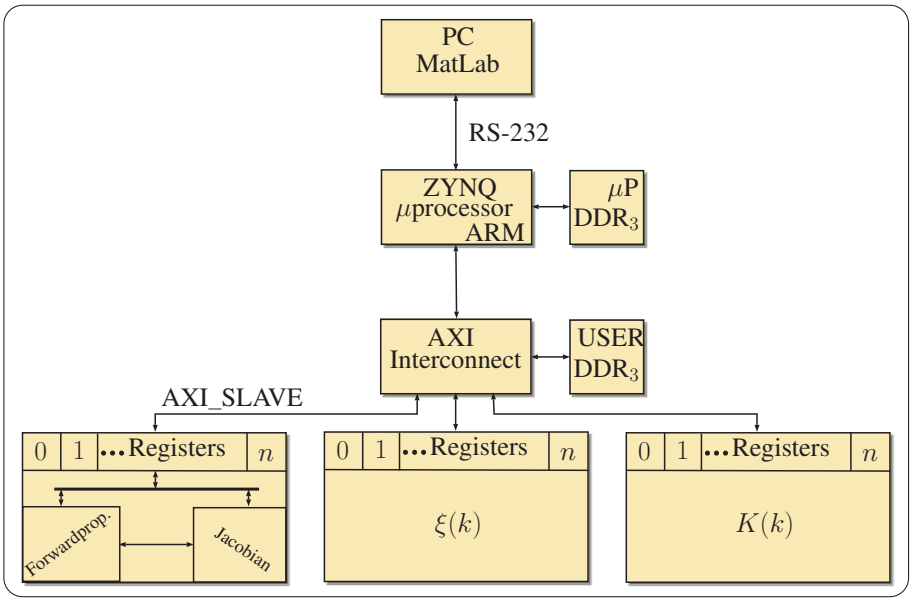

Figure 12. Full architecture implemented in FPGA.

Transmitting information to slave hardware based on AXI4 requires a driver to configure the interface in order to transmit a message. Hence, a driver was developed in C++ language and executed by the ARM microprocessor. The driver allows configuring any hardware architecture, reading and writing memories, and controlling the execution of mathematical blocks by means of the architecture. The configuration process was carried out in two stages: The first stage consisted in loading a file with extension hdf, which contained a system level configuration that incorporated the hardware and connections to the microprocessor. The second stage was carried out by the SDK of Xilinx, which programed the scripts written in C for the ARM microprocessor. The EKF process was executed through the block diagram process presented in Figure 13.

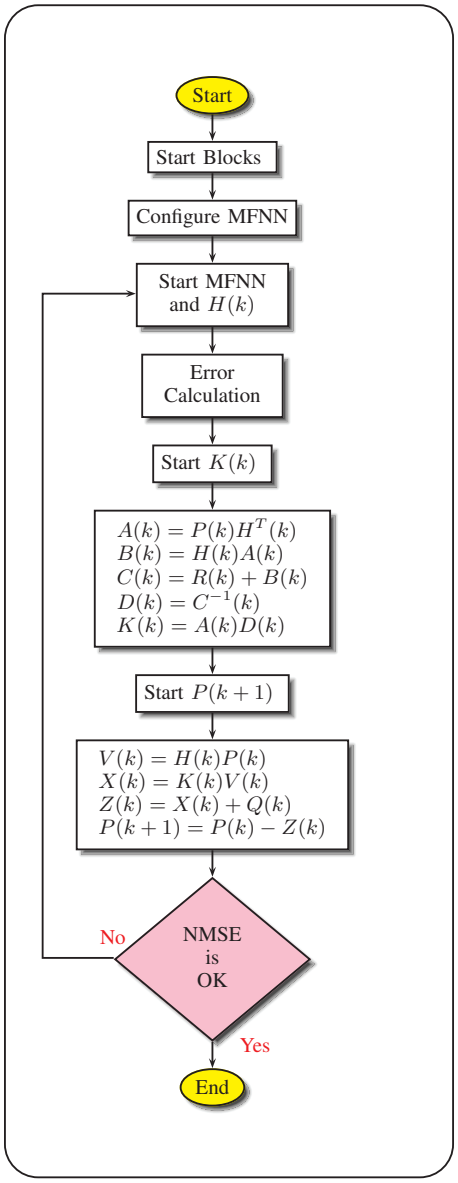

Figure 13. Training algorithm performed by ARM microprocessor.

4. Experimental Results

To validate the proposed architecture, a test was carried out for a black-box type identification of a PA. A dataset of 5000 complex numbers taken from the measurements of a GaN class F PA using a 2.1 GHz Long Term Evolution (LTE) signal with a 5 MHZ of bandwidth was considered [32]. For taking the memory effects [33] of PAs into account, the input signal was delayed by six samples. For the HIL simulation, the NMSE in steady state was fixed at a value of -35.6 dB, with a 220.15 µs execution time for the RVNARX and of 190.23 µs for the Jacobian Matrix, making a total execution time in the FPGA of 410.38 µs for each input dataset. The architecture requires 2.049 s to process 5000 complex numbers. Figure 14 shows a comparison of the estimated and the measured output where the good performance of the HIL setup can be appreciated.

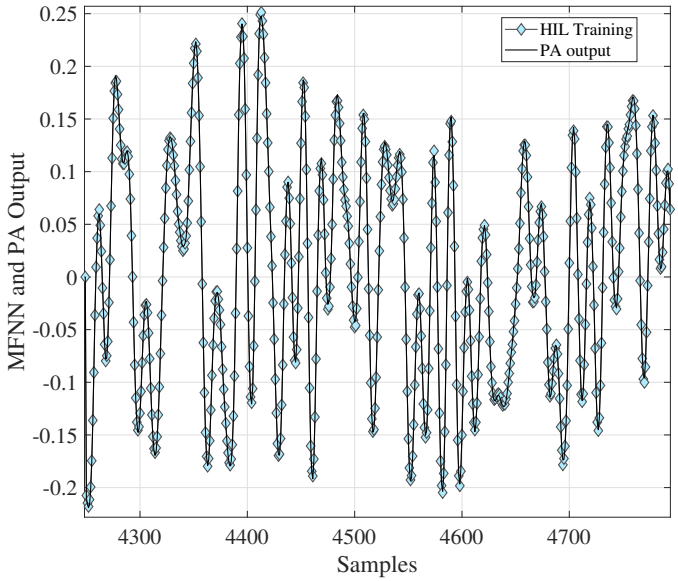

Figure 14. Comparison of results obtained between HIL simulation and PA output.

The complete architecture was tested using the same neural network and PA measurements as those used in the HIL simulation. In this experiment, the execution combined the training process given in Figure 13, and the training-validation process shown in Figure 15, which is a training based on epochs containing training–validation loops. The NMSE evolution is shown in Figure 16.

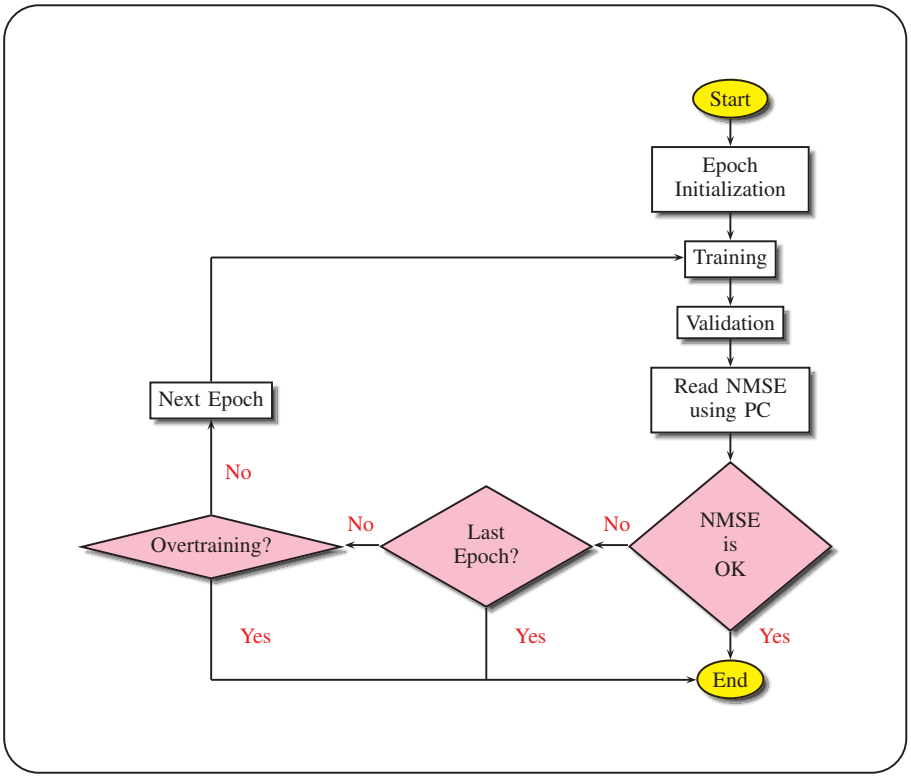

Figure 15. Training and validation algorithm.

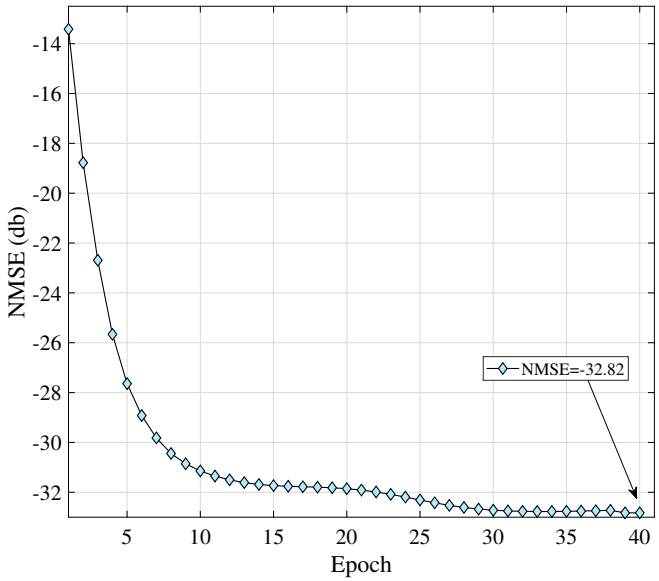

Figure 16. NMSE evolution by means of EKF training.

This test was carried out with 40 epochs with 10,000 complex numbers for training and 2000 complex numbers for validation. The NMSE value obtained in steady state was −32.82 dB. This value is different from that obtained with the HIL simulation, as the samples sets for training and validation were different.

The result obtained after the 200-samples training process is shown in Figure 17, where the estimated values obtained with the MFNN are practically the same as those provided by the real system.

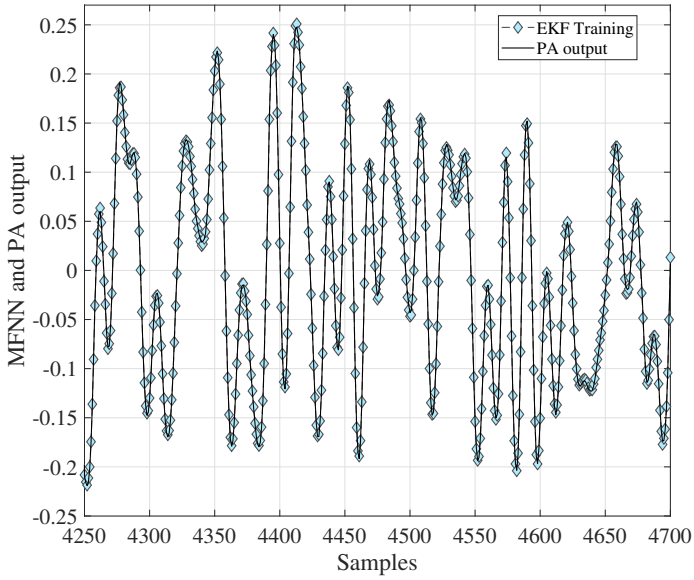

Figure 17. Comparison of results obtained between MFNN based on EKF training and PA output.

Table 1 compares logical resource consumption for three different architectures. The second column presents resources occupation for a custom architecture, in which each perceptron was implemented in hardware with the idea of a parallel processing of data, increasing in that way the operating frequency by 11.5% with respect to the proposed HIL architecture (third column). On the other hand, one disadvantage of parallel processing is that resources consumption was increased by

35% with respect to the proposed architecture. The proposed HIL configurable architecture has proven to be the implementation with the lowest consumption of logical resources, as it can only perform forwardpropagation and Jacobian Matrix processes; a personal computer was required to train the neural network. The fourth column corresponds to the complete EKF hardware, where it is clear that more resources were utilized with respect to the custom architecture presented in [32] and the HIL setup, but the advantage relies on the configurable architecture feature that adds the ability to train an MFNN without a PC, positioning this solution as a powerful mathematical hardware for modeling and identifying nonlinear systems in a reconfigurable device, and that can easily be used for portable applications. Another advantage relies on the fact that, if the MFNN increases in size, the required resources by a custom implementation will also increased but, in our proposal, the number of resources will not be incremented (with the exception of memory blocks), i.e., it will always keep practically the same number of resources.

Table 1. Utilization of Logical Resources.

Logical Resources	Custom Architecture [32]	Configurable Architecture (HIL)	Extended Kalman Filter	Logical Resources Available
Slice Registers	38,572	28,800	52,186	301,440
Slice LUTs	29,057	24,149	36579	150,720
Block RAM	225	277	316	416
DSP	64	156	180	768
Maximum operation frequency	95.511 MHz	85.6 MHz	85.6 MHz	—
Execution time	(RVNARX) 6.23 µs	(RVNARX and Jacobian matrix) 410.38 µs	(EFK) 41.76 ms	
Power Consumption	1.24 W	2.1 W	2.75 W	—

5. Conclusions

The training of neural networks with the EKF algorithm is a popular combination, but it represents a considerable computational burden. To reduce the computational execution time, there are some works dealing with the implementation of neural networks trained with the EKF algorithm in reconfigurable devices such as FPGAs. One drawback of this approach is the time required for algorithm redesign. Hence, this work presents a novel configurable architecture with highly complex computing capacities, which allows replicating the transfer function of a MFNN without the need to restructure the implemented hardware. All of this is possible by using a single perceptron multiplexed in time configured by means of a microprocessor in a SoC environment. Nevertheless, one of the main attractive features of this work is the full implementation of the EKF in hardware, designed for processing high-precision numbers for the training of configurable MFNNs of two or three layers, developed under a modular architecture strategy with multiple clocks.

The design was subjected to a stress test using an MFNN configured under the guidelines of an RVNARX in serial-parallel configuration with six delays per input and 16 neurons with hyperbolic tangent, and two neurons with the linear threshold function. The input data were obtained from a GaN class F PA using a 2.1 GHz LTE signal with 5 MHz of bandwidth, applying 10,000 complex number samples for training, and 2000 complex number samples for validation, obtaining an NMSE of −32.82 dB in 40 training epochs. The generalization of the RVNARX obtained for unknowing inputs was 71.36% of assertion, which is considered an excellent result for a stand-alone device.

A comparison with a custom implementation was carried, out as shown in Table 1. The custom implementation of the MFNN implements each perceptron of each layer. Although our proposal, as shown in the fourth column of Table 1, requires of more resources than those required by the custom implementation (second column), the advantage relies on the fact that, if the MFNN increases in size, the required resources by a custom implementation will be increased too, but in our proposal the number of resources will not be incremented (with the exception of memory blocks), i.e., it will always keep practically the same number of resources.

Some issues remain, such as the general implementation of the Jacobian matrix for the maximum number of layers (256), the general implementation of matrix inversion, and the extension of this proposal for the training of ANN based on the UKF or the SVSF.

Author Contributions: Conceptualization, S.O.-C. and R.L.-Y.; Methodology, F.S.-I.; Software, J.R.-C.; Validation, J.R. and F.S.-I.; Formal Analysis, S.O.-C. and R.L.-Y.; Writing—Original Draft Preparation, J.R.-C.; Writing—Review and Editing, J.R.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Turajlic, E.; Begović, A.; Škaljo, N. Application of Artificial Neural Network for Image Noise Level Estimation in the SVD domain. *Electronics* **2019**, *8*, 163. [CrossRef]
2. Reynaldi, A.; Lukas, S.; Margaretha, H. Backpropagation and Levenberg-Marquardt Algorithm for Training Finite Element Neural Network. In Proceedings of the 2012 Sixth UKSim/AMSS European Symposium on Computer Modeling and Simulation, Valetta, Malta, 14–16 November 2012; pp. 89–94.
3. Vo, H.M. Implementing the on-chip backpropagation learning algorithm on FPGA architecture. In Proceedings of the 2017 International Conference on System Science and Engineering (ICSSE), Ho Chi Minh City, Vietnam, 21–23 July 2017; pp. 538–541.
4. Haykin, S.; Haykin, S. *Adaptive Filter Theory*; Pearson: Essex, UK, 2014.
5. Mercorelli, P. A Motion-Sensorless Control for Intake Valves in Combustion Engines. *IEEE Trans. Ind. Electron.* **2017**, *64*, 3402–3412. [CrossRef]
6. Li, S. Comparative analysis of backpropagation and extended Kalman filter in pattern and batch forms for training neural networks. In Proceedings of the IJCNN'01 International Joint Conference on Neural Networks. Proceedings (Cat. No. 01CH37222), Washington, DC, USA, 15–19 July 2001; Volume 1, pp. 144–149.
7. Mathews, V.J. Adaptive polynomial filters. *IEEE Signal Process. Mag.* **1991**, *8*, 10–26. [CrossRef]
8. Rigatos, G.G. A Derivative-Free Kalman Filtering Approach to State Estimation-Based Control of Nonlinear Systems. *IEEE Trans. Ind. Electron.* **2012**, *59*, 3987–3997. [CrossRef]
9. Afshari, H.; Gadsden, S.; Habibi, S. Gaussian filters for parameter and state estimation: A general review of theory and recent trends. *Signal Process.* **2017**, *135*, 218–238. [CrossRef]
10. Ahmed, R.; El Sayed, M.; Gadsden, S.A.; Tjong, J.; Habibi, S. Artificial neural network training utilizing the smooth variable structure filter estimation strategy. *Neural Comput. Appl.* **2016**, *27*, 537–548. [CrossRef]
11. Young, K.D.; Utkin, V.I.; Ozguner, U. A control engineer's guide to sliding mode control. *IEEE Trans. Control Syst. Technol.* **1999**, *7*, 328–342. [CrossRef]
12. Mercorelli, P. An Adaptive and Optimized Switching Observer for Sensorless Control of an Electromagnetic Valve Actuator in Camless Internal Combustion Engines. *Asian J. Control* **2014**, *16*, 959–973. [CrossRef]
13. Frances-Villora, J.; Rosado-Munoz, A.; Bataller-Mompean, M.; Barrios-Aviles, J.; Guerrero-Martinez, J. Moving Learning Machine towards Fast Real-Time Applications: A High-Speed FPGA-Based Implementation of the OS-ELM Training Algorithm. *Electronics* **2018**, *7*, 308. [CrossRef]
14. Wang, L.; Lu, D.; Liu, Q.; Liu, L.; Zhao, X. State of charge estimation for $LiFePO_4$ battery via dual extended kalman filter and charging voltage curve. *Electrochim. Acta* **2019**, *296*, 1009–1017. [CrossRef]
15. Lu, C.; Wu, S.; Jiang, C.; Hu, J. Weak harmonic signal detection method in chaotic interference based on extended Kalman filter. *Digit. Commun. Netw.* **2019**, *5*, 51–55. [CrossRef]

16. neda, C.E.C.; López-Mancilla, D.; Chiu, R.; na Rauda, E.V.; Orozco-López, O.; Casillas-Rodríguez, F.; Sevilla-Escoboza, R. Discrete-time neural synchronization between an Arduino microcontroller and a Compact Development System using multiscroll chaotic signals. *Chaos Solitons Fractals* **2019**, *119*, 269–275.
17. Joukov, V.; Bonnet, V.; Karg, M.; Venture, G.; Kulić, D. Rhythmic Extended Kalman Filter for Gait Rehabilitation Motion Estimation and Segmentation. *IEEE Trans. Neural Syst. Rehabil. Eng.* **2018**, *26*, 407–418. [CrossRef] [PubMed]
18. Ergen, T.; Kozat, S.S. Online Training of LSTM Networks in Distributed Systems for Variable Length Data Sequences. *IEEE Trans. Neural Networks Learn. Syst.* **2018**, *29*, 5159–5165. [CrossRef] [PubMed]
19. Romero-Aragon, J.C.; Sanchez, E.N.; Alanis, A.Y. Glucose level regulation for diabetes mellitus type 1 patients using FPGA neural inverse optimal control. In Proceedings of the 2014 IEEE Symposium on Computational Intelligence in Control and Automation (CICA), Orlando, FL, USA, 9–12 December 2014; pp. 1–7.
20. Dávalos, U.; Castaneda, C.; Esquivel, P.; Jurado, F.; Morfín, O.A. Recurrent Neural Identification on Xilinx system generator using V7 FPGA for a 2DOF robot manipulator. In Proceedings of the 2016 International Joint Conference on Neural Networks (IJCNN), Vancouver, BC, Canada, 24–29 July 2016; pp. 2359–2365.
21. *IEEE Standard for Verilog Hardware Description Language*; IEEE Std 1364–2005; IEEE: Piscataway, NJ, USA, 2006.
22. *IEEE Standard for Floating-Point Arithmetic*; IEEE Std 754–2008; IEEE: Piscataway, NJ, USA, 2008.
23. Wakhle, G.B.; Aggarwal, I.; Gaba, S. Synthesis and Implementation of UART Using VHDL Codes. In Proceedings of the 2012 International Symposium on Computer, Consumer and Control, Taichung, Taiwan, 4–6 June 2012; pp. 1–3.
24. Iiguni, Y.; Sakai, H.; Tokumaru, H. A real-time learning algorithm for a multilayered neural network based on the extended Kalman filter. *IEEE Trans. Signal Process.* **1992**, *40*, 959–966. [CrossRef]
25. Ruck, D.W.; Rogers, S.K.; Kabrisky, M.; Maybeck, P.S.; Oxley, M.E. Comparative analysis of backpropagation and the extended Kalman filter for training multilayer perceptrons. *IEEE Trans. Pattern Anal. Mach. Intell.* **1992**, *14*, 686–691. [CrossRef]
26. Rawat, M.; Rawat, K.; Ghannouchi, F.M. Adaptive Digital Predistortion of Wireless Power Amplifiers/Transmitters Using Dynamic Real-Valued Focused Time-Delay Line Neural Networks. *IEEE Trans. Microw. Theory Tech.* **2010**, *58*, 95–104. [CrossRef]
27. Qian, M. Application of CORDIC Algorithm to Neural Networks VLSI Design. In Proceedings of the Multiconference on "Computational Engineering in Systems Applications", Beijing, China, 4–6 October 2006; Volume 1, pp. 504–508.
28. Alimohammad, A.; Fard, S.F.; Cockburn, B.F. Hardware Implementation of Nakagami and Weibull Variate Generators. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2012**, *20*, 1276–1284. [CrossRef]
29. Trejo-Arellano, J.; Castillo, J.V.; Longoria-Gandara, O.; Carrasco-Alvarez, R.; Gutiérrez, C.; Atoche, A.C. Adaptive segmentation methodology for hardware function evaluators. *Comput. Electr. Eng.* **2018**, *69*, 194–211. [CrossRef]
30. Aguilar-Lobo, L.M.; Loo-Yau, J.R.; Rayas-Sánchez, J.E.; Ortega-Cisneros, S.; Moreno, P.; Reynoso-Hernández, J.A. Application of the NARX neural network as a digital predistortion technique for linearizing microwave power amplifiers. *Microw. Opt. Technol. Lett.* **2015**, *57*, 2137–2142. [CrossRef]
31. Marks, N.D.; Kong, W.Y.; Birt, D.S. Stability of a Switched Mode Power Amplifier Interface for Power Hardware-in-the-Loop. *IEEE Trans. Ind. Electron.* **2018**, *65*, 8445–8454. [CrossRef]
32. Renteria-Cedano, J.A.; Aguilar-Lobo, L.M.; Loo-Yau, J.R.; Ortega-Cisneros, S. Implementation of a NARX neural network in a FPGA for modeling the inverse characteristics of power amplifiers. In Proceedings of the 2014 IEEE 57th International Midwest Symposium on Circuits and Systems (MWSCAS), College Station, TX, USA, 3–6 August 2014; pp. 209–212.
33. Barradas, F.M.; Nunes, L.C.; Cunha, T.R.; Lavrador, P.M.; Cabral, P.M.; Pedro, J.C. Compensation of Long-Term Memory Effects on GaN HEMT-Based Power Amplifiers. *IEEE Trans. Microw. Theory Tech.* **2017**, *65*, 3379–3388. [CrossRef]