FPGA IMPLEMENTATION OF ANN TRAINING USING LEVENBERG AND MARQUARDT ALGORITHMS

M.A. Cavuslu[*], *S. Sahin*[*]

Abstract: Artificial Neural Network (ANN) training using gradient-based Levenberg & Marquardt (LM) algorithm has been implemented on FPGA for the solution of dynamic system identification problems within the scope of the study. In the implementation, IEEE 754 floating-point number format has been used because of the dynamism and sensitivity that it has provided. Mathematical approaches have been preferred to implement the activation function, which is the most critical phase of the study. ANN is tested by using input-output sample sets, which are shown or not shown to the network in the training phase, and success rates are given for every sample set. The obtained results demonstrate that implementation of FPGA-based ANN training is possible by using LM algorithm and as the result of the training, the ANN makes a good generalization.

Key words: *Levenberg and Marquardt, FPGA, MLP and ANN training*

Received: October 15, 2017 **DOI:** 10.14311/NNW.2018.28.010
Revised and accepted: April 16, 2018

1. Introduction

Artificial neural networks (ANN), which can effectively model the nonlinear relationship between input and output in a system [1], are successfully used to solve problems in many areas [2], [3], [4], [5]. Back-propagation (BP) algorithm is commonly used in training of ANNs [6]. But, BP algorithm has some disadvantages such as lower training efficiency [7] and bad convergence speed [6].

Algorithms that require second derivative information such as Levenberg and Marquardt (LM) prominently increases ANN's learning speed [7]. LM algorithm, which combines the speed of Newton algorithm and stability of the steepest descent method, is used efficiently in network training [8].

The studies related to implementation of an ANN training and/or a trained ANN on FPGA are presented in the literature. Ferrer, Gonzalez, Fleitas, Acle and Canetti [9], Savich, Moussa and Areibi [10], Farmahini-Farahani, Fakhraie and Safari [11] have used fixed-point number format at various bit lengths. Nedjah, Silva, Mourelle and Silva [12], Çavuşlu, Karakuzu and Şahin [13], Çavuşlu, Karakuzu,

[*]Mehmet Ali Çavuşlu; Suhap Şahin; – Corresponding author; Computer Engineering, Kocaeli University, Izmit, Kocaeli, Turkey, E-mail: alicavuslu@gmail.com, suhapsahin@kocaeli.edu.tr

Şahin and Yakut [14], Çavuşlu, Karakuzu and Karakaya [15] have used floating-point number format at various bit-lengths. Won, on the other hand, has used integer format in his study [16].

Nedjah, Silva, Mourelle and Silva [12], Won [16], Çavuşlu, Karakuzu and Şahin [13], Çavuşlu, Karakuzu, Şahin and Yakut [14], Çavuşlu, Karakuzu and Karakaya [15], Savich, Moussa and Areibi [10] have used logarithmic sigmoidal approaches as activation function. Farmahini-Farahani, Fakhraie and Safari [11], Ferrer, Gonzalez, Fleitas, Acle and Canetti [9], Farmahini-Farahani, Fakhraie and Safari [11], Çavuşlu, Karakuzu and Karakaya [15] have used tangent hyperbolic activation function approaches as activation function.

Won [16],Farmahini-Farahani, Fakhraie and Safari [11] have used look-up table approach for hardware implementations of activation functions. Ferreira, Ribeiro, Antunes and Dias [17], Ferrer, Gonzalez, Fleitas, Acle and Canetti [9], Savich, Moussa and Areibi [10] have used piecewise linear approach.Nedjah, Silva, Mourelle and Silva [12] have used parabolic approach. Çavuşlu, Karakuzu and Şahin [13], Çavuşlu, Karakuzu, Şahin and Yakut [14], Çavuşlu, Karakuzu and Karakaya [15] have used mathematical functional approach.

Savich, Moussa and Areibi [10], Çavuşlu, Karakuzu, Şahin and Yakut [14] have used back-propagation algorithm for network training. Farmahini-Farahani, Fakhraie and Safari [11], Çavuşlu, Karakuzu and Karakaya [15] have used PSO algorithm in network training.

Implementation of ANN training with LM algorithm over FPGA in 32-bit floating-point number format is explained in this study with examples in system identification problems. Mathematical approach is used for neural cell activation functions. The difference of this approach from other studies is the use of division module in addition to the addition and multiplication modules used in the piecewise linear and parabolic approaches. The approach used in the study has advantages such as not requiring as much memory as in the reference table or control expressions as in the piecewise linear approach. Parallel RAM architecture, which utilizes parallel data processing capability of the FPGA, is implemented to ensure rapid parameter update, especially in systems that require online training. Parameter update operations are reduced in the implementation phase with this architecture, allowing the system to respond more quickly to updates.

2. Multilayer Perceptron

In MLP, the cells are organized as layers, and the outputs of the cells in a layer are weighted and given as input to the next layer. MLP consists of 3 layers including input layer, hidden layer, and output layer (Fig. 1). In Eq. 1, u_k^1 refers to the total value of k^{th} cell in hidden layer, m refers to total input number, x refers to input value, and ω refers to weight value. In the presentation of the weight values, upper index parameter indicates the layer number. In lower index value, the first parameter refers to the relevant weight input index, and the second parameter indicates the cell to which the weighted input value belongs. Output values obtained in the hidden layer are given as input to the cells in output layer. Transfer of these values to output after the weighting procedure is given in Eq. 2. In Eq. 2, n shows the cell number in the output layer.

$$\left.\begin{aligned} u_k^1 &= \sum_{i=1}^{m} \omega_{ik}^1 x_i + b_k^1 \\ s_k &= \varphi\left(u_k^1\right) \end{aligned}\right\} \quad k = 1, \ldots, q \tag{1}$$

$$\left.\begin{aligned} u_k^2 &= \sum_{i=1}^{q} \omega_{ik}^2 s_i + b_k^2 \\ y_k &= \varphi\left(u_k^2\right) \end{aligned}\right\} \quad k = 1, \ldots, n \tag{2}$$

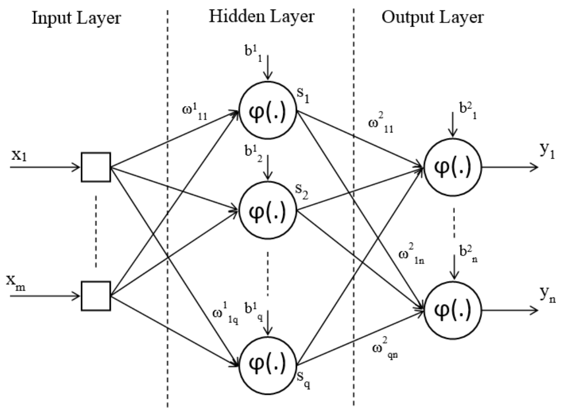


Fig. 1 *MLP architecture.*

2.1 Activation function

Logarithmic sigmoidal and tangent hyperbolic activation functions, which are widely used in literature, have been preferred in the study. As it is not possible to implement the exponential expression used in the functions directly on FPGA, mathematical approaches have been implemented on FPGA as hardware.

Mathematical approaches have been used in Eq. 3 and Eq. 4 respectively for hardware implementation of logarithmic sigmoidal and tangent hyperbolic functions [18].

$$Q_{logsig} = \frac{1}{2}\left[\frac{x}{1+|x|}\right] \tag{3}$$

$$Q_{tansig} = \frac{x}{1+|x|} \tag{4}$$

Gradient based training algorithm is realized in this study, derivatives of approaches related to logarithmic sigmoidal and tangent hyperbolic functions are shown in Eq. 5 and Eq. 6 respectively.

$$Q'_{logsig} = \begin{cases} \frac{0.5}{(1-x)^2}, & x < 0 \\ \frac{0.5}{(1+x)^2}, & x \geqslant 0 \end{cases} \tag{5}$$

$$Q'_{tansig} = \begin{cases} \frac{1}{(1-x)^2}, & x < 0 \\ \frac{1}{(1+x)^2}, & x \geqslant 0 \end{cases} \tag{6}$$

In Fig. 2, logarithmic sigmoid piecewise linear function used by Savich, Moussa and Areibi [10], and the Logarithmic sigmoid function mathematical approach given in Eq. 5 are comparatively illustrated within the range of $[-10, 10]$.

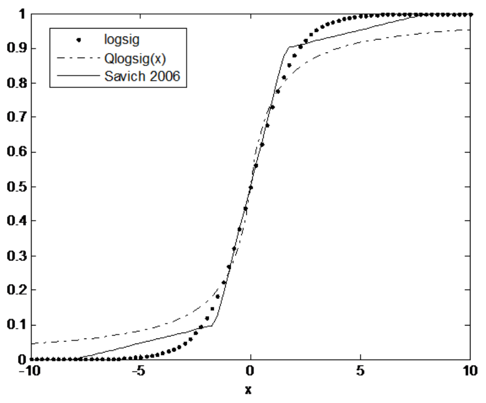


Fig. 2 *Comparison of logarithmic sigmoid function and its approaches.*

On the other hand, in In Fig. 3, the original logsig is shown with the suggested approach and the derivatives of piecewise linear approached used in Savich, Moussa and Areibi [10] are shown comparatively [19].

3. Levenberg and Marquardt algorithm

Parameter updating process related to LM algorithm derived from steepest descent and Newton algorithms is given by Eq. 7. In Eq. 7, ω represents weight vector, I represents unit matrix, and μ represents combination coefficient. J represents the Jacobian matrix in $[(P.n), N]$ dimensions, e represents error vector in $[(P.n), 1]$ dimensions, P refers to the training sample number, n refers to the output number and N refers to the weight number [7].

$$\Delta\omega = (J^T J + \mu I)^{-1} J^T e \tag{7}$$

LM algorithm performs the parameter update processes by using the error vector and Jacobian matrix created for sample values related to all inputs. Jacobian matrix described in Eq. 7 is obtained in Eq. 8, whereas the error vector is obtained as in Eq. 9.

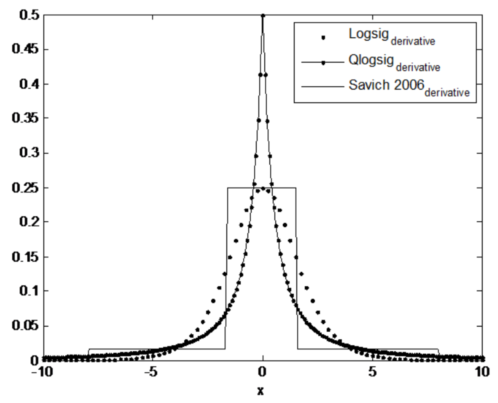


Fig. 3 *Comparison of logarithmic sigmoid function and its approaches derivatives.*

$$J = \begin{bmatrix} \frac{\partial e_{11}}{\partial \omega_1} & \frac{\partial e_{11}}{\partial \omega_2} & \cdots & \frac{\partial e_{11}}{\partial \omega_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{1n}}{\partial \omega_1} & \frac{\partial e_{1n}}{\partial \omega_2} & \cdots & \frac{\partial e_{1n}}{\partial \omega_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{P1}}{\partial \omega_1} & \frac{\partial e_{P1}}{\partial \omega_2} & \cdots & \frac{\partial e_{P1}}{\partial \omega_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{Pn}}{\partial \omega_1} & \frac{\partial e_{Pn}}{\partial \omega_2} & \cdots & \frac{\partial e_{Pn}}{\partial \omega_N} \end{bmatrix} \tag{8}$$

$$e = \begin{bmatrix} e_{11} \\ \cdots \\ e_{1n} \\ \cdots \\ e_{P1} \\ \cdots \\ e_{Pn} \end{bmatrix} \tag{9}$$

The parameter described in Eq. 7 is an adjustable parameter as seen in Eq. 10. If this parameter is too large, LM algorithm acts like steepest descent gradient method. If it is too small, it acts like the Newton method [7].

$$\mu(n) = \begin{cases} k\mu(n-1) & E[t] > E[t-1] \\ \mu(n-1)/k & E[t] \leq E[t-1] \end{cases} \tag{10}$$

In Eq. 10, $E[t]$ is used for fitness value calculation (Eq. 11).

$$E[t] = \sum_{i=1}^{P} e_i \tag{11}$$

4. Creating the Jacobian matrix

The placement structure of the row values of the Jacobian Matrix, which is defined in Eq. 8, used in the FPGA-based implementation is shown in Eq. 12. This structure is used for all input sets with reference to Fig. 4 (Eq. 13 – Eq. 18).

$$\begin{bmatrix}\nabla\omega^1_{11} & \dots & \nabla\omega^1_{1q} & \dots & \nabla\omega^1_{m1} & \dots & \nabla\omega^1_{mq} & \nabla b^1_1 & \dots\nabla b^1_q & \dots\end{bmatrix} \tag{12}$$

The v values shown in Fig. 4 are calculated as shown in Eq. 13. Update value of the bias parameters in the output layer is calculated as shown in Eq. 14. Parameter update value of the weighting process of the outputs of the cells in the hidden layer is calculated as shown in Eq. 15. In Eq. 15, s_i is i^{th} cell output at the hidden layer in the feed forward phase. The z values in Fig. 4 are calculated as shown in Eq. 16. Updating value of bias parameters in the hidden layer is calculated as shown in Eq. 17. Parameter updating value of the weighting process of inputs is calculated as shown in Eq. 18.

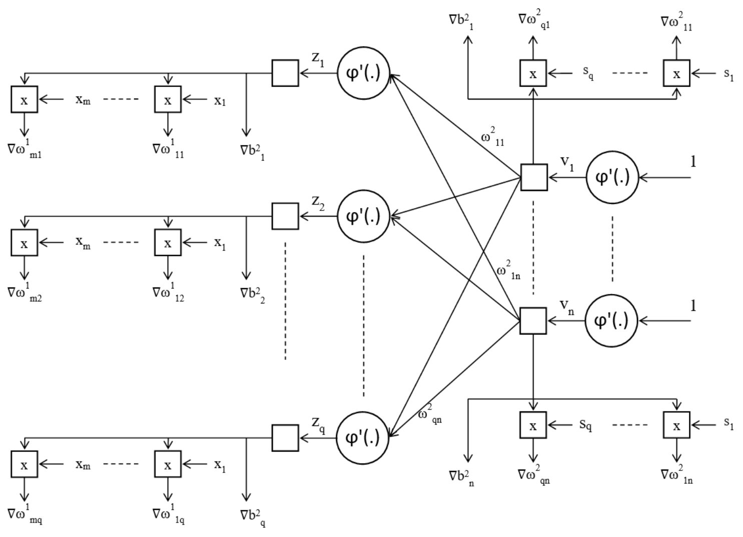


Fig. 4 *Block diagram of MLP parameters update by LM algorithm.*

$$v_k = \varphi^{'}_k; \quad k = 1, \dots, n \tag{13}$$

$$\nabla b^2_k = v_k; \quad k = 1, \dots, n \tag{14}$$

$$\nabla\omega^2_{ik} = s_i v_k; \quad k = 1, \dots, n; \quad i = 1, \dots, q \tag{15}$$

$$z_i = \varphi^{'}_k \textstyle\sum_{k=1}^{n} v_k\omega^2_{ik}; \quad i = 1, \dots, q \tag{16}$$

$$\nabla b_i = z_i; \quad i = 1, \dots, q \tag{17}$$

$$\nabla\omega^2_{ni} = x_k z_i; \quad i = 1, \dots, q; \quad n = 1, \dots, m \tag{18}$$

5. FPGA implementation of Levenberg and Marquardt algorithm

Levenberg and Marquardt Algorithm is implemented step by step in the phase of hardware implementation on FPGA as described in Eq. 7. These steps are summarized briefly as follows.

5.1 Creating of Jacobian matrix memories

The parallel data processing capability of FPGA is intended to be used at a high level during the hardware implementation phase, within the scope of the study. In line with this, separate memory units were created to retain the values related to each parameter in the phase of creating the Jacobian matrix. By using separate memory units, read and write processes can be performed simultaneously for each parameter value.

In the study, for N parameters to be optimized, same number of RAM blocks were created. The length of each RAM block is equal to the input sample number, P . The depths of memory units are adjusted as 32 bits (Fig. 5).

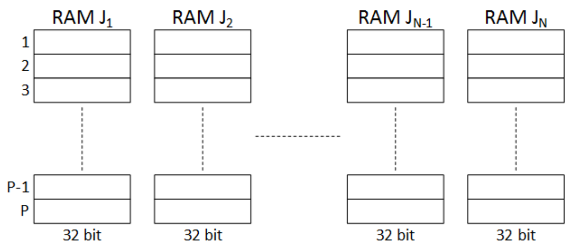


Fig. 5 *Block RAMs created for Jacobian matrix.*

5.2 Multiplying the transpose Jacobian matrix with the Jacobian matrix

In this step, a transpose Jacobian matrix in size of $N.P$ will be multiplied the Jacobian matrix in size of $P.N$. A matrix of $N.N$ will be obtained as the result of the multiplication. For the purpose of parallel data processing, multiplication results are stored in N number of memory units, in depth of 32 bits with a length of N (Fig. 6).

The Jacobian matrix implemented similarly in Section 5.1, by creating separate memory units for each parameter, saves time by simultaneous performance of read, multiplication and addition processes. Additionally, by use of the parallel memory unit, operation load is reduced from N^2P to NP in implementation.

In the multiplication processes of the transposed Jacobian matrix and the Jacobian matrix, the data from the memory units created for the Jacobian matrix needs to be read first. For this aim, in the 1st step, k index-data is read from the relevant memory unit for each parameter. In the 2nd step, m index-memory unit

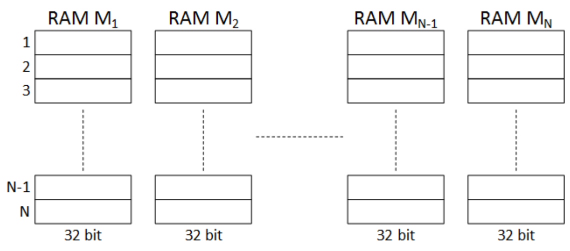


Fig. 6 *Block RAMs created for multiplication results.*

element is selected from the values read from memory units, to be used in the next multiplication process. In the 3rd step, all values read from memory units in the 1st step are multiplied with the value selected in the 2nd step. In the 4th step, the results of multiplications obtained in the 3rd step are added with the $S(k)$, the total value obtained in the previous k index value, to obtain the new total value $S(k+1)$. Initial total value is zero ($S(0) = 0$) for $k = 0$ situation. In the 5th step, k value is incremented by 1, if k value is lower than input sample value (P), the process returns back to the 1st step, otherwise it moves on to the 6th step (Fig. 7).

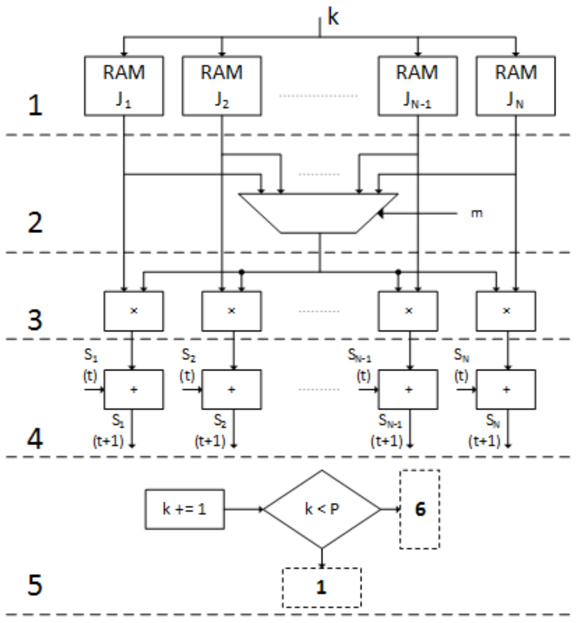


Fig. 7 *Block diagram of the multiplication of transposed Jacobian matrix with Jacobian matrix – 1.*

In the 6th step, the total values obtained in the 4th step are written to the memory units (RAM M) created for storing the matrix multiplication results. In the 7th step aims the implementation of $(J^T J + \mu I)$ expression described in Eq. 7.

For this reason, the diagonal values of the matrix obtained at the result of the multiplication are recorded in the memory unit. For these processes to be performed, the matrix has (m, m) diagonal values in the m^{th} step, (m^{th} element of the m^{th} memory unit is taken), the diagonal value found in the 8th step is written to m^{th} value of memory unit (RAM S). In the 9th step, the m value is incremented by one. If this value is smaller than the N value, process returns to the 1st step. Otherwise, it proceeds to the 10th step (Fig. 8).

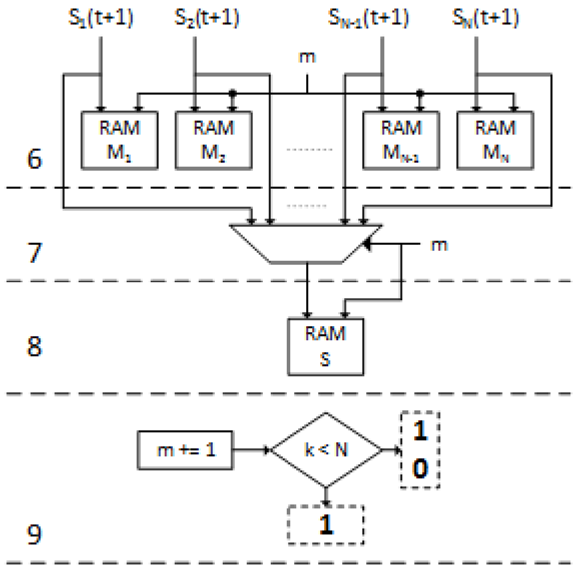


Fig. 8 *Block diagram of the multiplication of transposed Jacobian matrix with Jacobian matrix – 2.*

In 10th step, the value stored in the *nth* address of memory unit (RAM S), in which the matrix diagonal values are stored, is read. In the 11th step, the value read in the 10th step is directed to the 12th step, depending on the diagonal value, and in the 12th step, μ parameter is added. In the 13th step, it is written at the *nth* address of the memory units (RAM M), in which matrix multiplications are stored. In the 14th step, n value is incremented by one. If this value is smaller than the N value, process returns to the 1st step again. Otherwise, it proceeds to the next step (Fig. 9).

5.3 Inverse matrix calculation

Suppression methods such as adjoint, LU, QR, Hermitian, Analytic, Blockwise, Gauss-Jordan are suggested in literature for taking reciprocal matrix. In the study, it was performed on hardware implementation by using the Gauss-Jordan suppression method without any restriction with regards to matrix dimensions.

The reciprocal process for matrix consists of the following phases. In the 1st phase, first the multiplication values stored in RAM Ms and the values at m^{th} address of RAM I's created as a unit matrix, are read. In the 2nd phase, the value

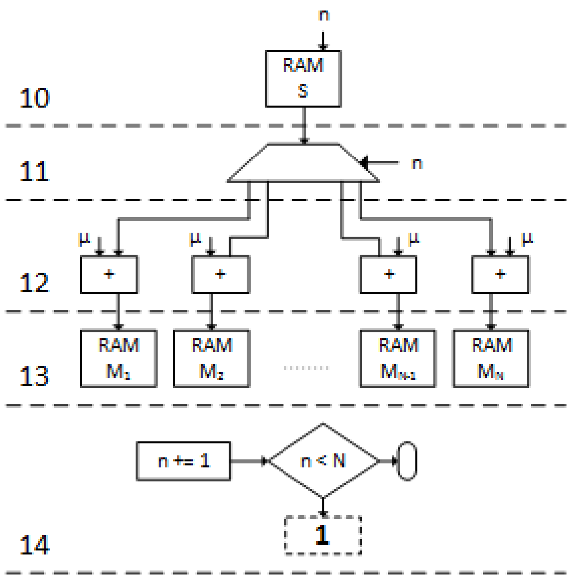


Fig. 9 *Block diagram of the multiplication of transposed Jacobian matrix with Jacobian matrix – 3.*

in m^{th} RAM M is selected. In the 3rd phase, all values read in the 1st phase are divided into the value obtained in the 2nd step. The results of the division are written again on their locations at the m^{th} addresses of RAM Ms and RAM Is, in the 4th phase (Fig. 10).

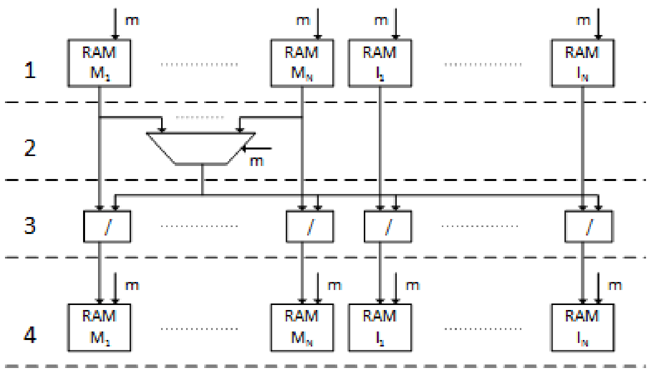


Fig. 10 *Inverse matrix calculation – 1.*

In the 5th step, whether the x value is equal to the m value is checked. If equal, process passes on to the 13th step by incrementing the m value by one. Otherwise, it passes to the 6th step. In the 6th step, x^{th} values are read from the memory units, in which the multiplication values (RAM M), and the inverse values (RAM I) are stored. And in the 7th step, the value related to the m^{th} RAM is selected as index value, from the RAM M's in which multiplication values are stored. In the 8th step, all values read from the RAM's are stored in the memory (Fig. 11).

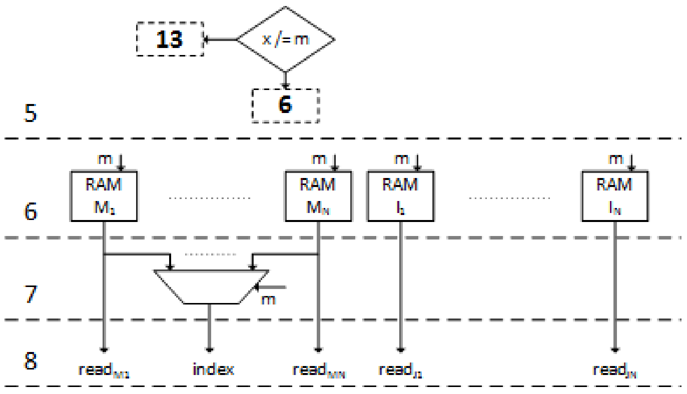


Fig. 11 *Inverse matrix calculation – 2.*

In the 9th step, *xth* values are read from the memory units, in which the multiplication values and their reciprocal values are stored. In the 10th step, the values read are multiplied with the index value. In the 11th step, the values, stored in memory in the 8th step, are subtracted from the multiplication values obtained in the 10th step. And in the 12th step, the result of the subtraction is written at the x^{th} addresses in the RAM M's and RAM I's. In the 13th step, x value is incremented by 1. If the x value is less than the number of parameters , then the process returns to the 5th step, otherwise it passes on to the 14th step. In the 14th step, the m value is incremented by 1. If this value is smaller than the number of parameters, process returns to the 1st step, otherwise the process is ended (Fig. 12).

5.4 Multiplication with error matrix

In the 1st step, m^{th} values are read from the memory units, in which the reciprocal matrix values are stored. Whereas in the 2nd step, x^{th} values are read from the memory units, in which the Jacobian matrix parameters are stored. In the 2^{rd} step, the values read in the 1st and 2nd are multiplied with vector. In the 4th step the result of vector multiplication is multiplied with the error value obtained for the *xth* input samples. In the 5th step, the result of multiplication is added to the multiplication result obtained in the previous step. In the 6th step, x value is incremented by 1. If the x value is smaller than the number of samples, the process returns to the 2nd step, otherwise it passes to the 7th step. In the 7th step, the total of the multiplications of error and matrix for all input samples is stored as the update value of the m^{th} parameter. In the 8th step, m value is incremented by 1. If this value is smaller than the number of parameters, then the process returns to the 1st step, otherwise the process is ended (Fig. 13).

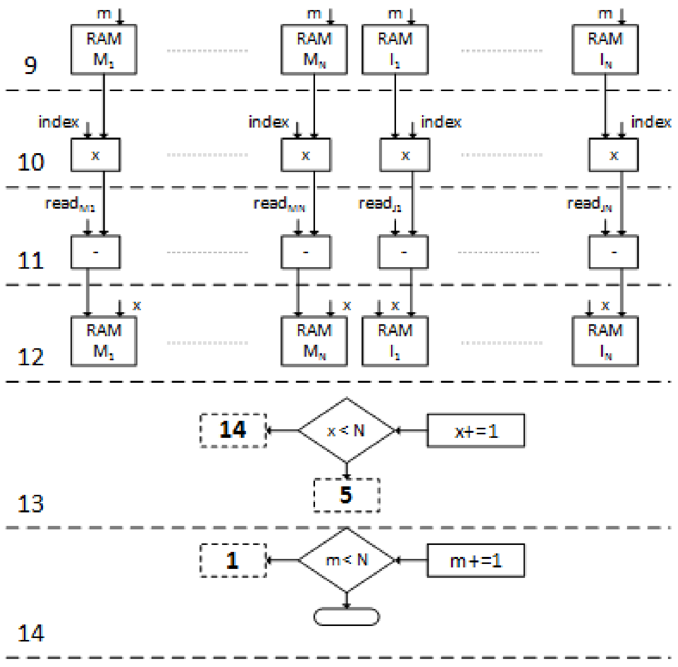


Fig. 12 *Inverse matrix calculation – 3.*

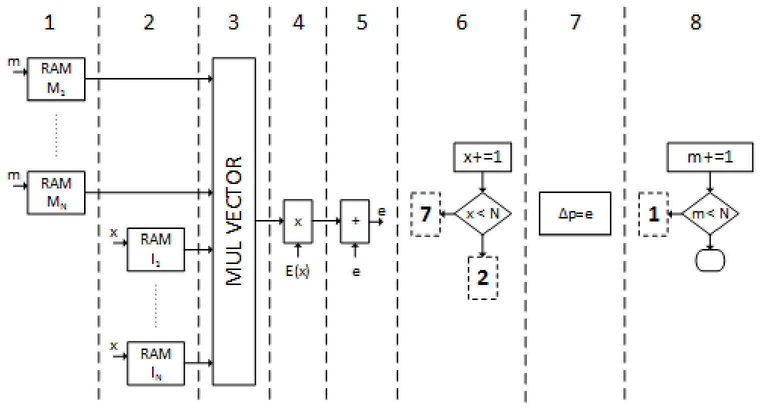


Fig. 13 *Multiplication with error matrix.*

6. Experimental results

ANN training implementation on FPGA by using LM algorithm is experimentally tested in two sample system identification problem. Experimental studies have been implemented by using ISE Design Suite 14.6 program over Xilinx Kintex 7 xc7k325tffg900-2 FPGA.

6.1 System identification problem – 1

Dynamic system identification problem is the process of adjustment of the parameters, which will represent the system, in a suitable manner. This adjustment process is implemented by optimization of the success function established on the error between the actual output of the system to be identified and the output of the model to be selected for identification. For this sample, the system given in Eq. 19 has been identified [5]. In Eq. 19, $u[k]$ has been identified as in Eq. 20.

$$y[k+1] = \frac{y[k]}{1+y^2[k]} + u^3[k] \tag{19}$$

$$u[k] = \cos[\frac{2k\pi}{100}]; k = 1, 2, \cdots, 100 \tag{20}$$

For system identification, the MLP with the inputs of $u[k]$ and $y[k]$, and with 3 cells in its hidden layer and 1 cell in output layer has been used. Logarithmic sigmoidal function has been used as activation function for hidden layer cells, and linear activation has been used for function output cells.

Fig 14a illustrates the hardware outputs related to LM algorithm and dynamic system identification process for Example 1. Fig. 14b shows the errors related to the hardware implementation of LM algorithm and dynamic system identification for Example 1.

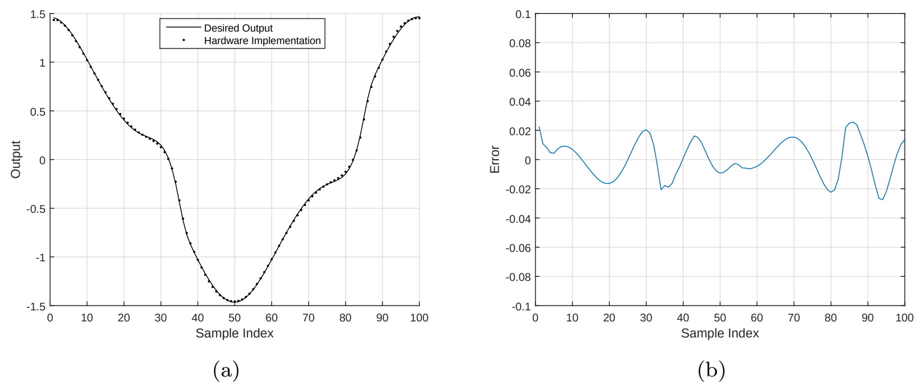
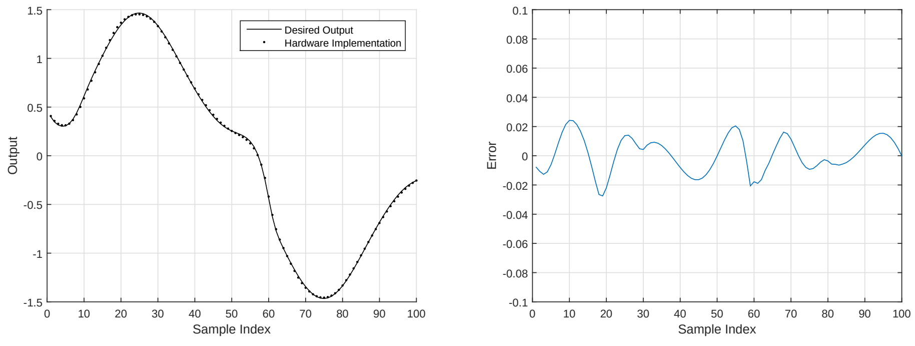


Fig. 14 *Experimental results of the training phase for the Example 1.*

Input series given in Eq. 21 are used to test the ANN training according to input series given in Eq. 20.

$$u[k] = \sin[\frac{2k\pi}{100}]; k = 1, 2, \cdots, 100 \tag{21}$$

Fig. 15a shows the outputs related to dynamic system identification for input series given in Eq. 21 , for Example 1. Fig. 15b exhibits the hardware implementation errors related to dynamic system identification process for the network trained with the LM algorithm for input series given in Eq. 21.



(a) (b)

Fig. 15 *Experimental results of the test phase for the Example 1.*

Tab. I shows the fitness values of the network trained according to the input series, and the fitness values of the input series that are not shown to the network. Fitness values are given as the average of fitness values obtained from 10 different trainings.

Training	Test	Iteration
0.0081	0.0073	20

Tab. I *Fitness values of training and test data for Example 1.*

Tab. II exhibits the synthesis results for the implementation of ANN trained with LM algorithm on FPGA.

Logic Utilization	Used	Available	Utilization
Number of Slice Registers	19314	407600	4%
Number of Slice LUTs	54858	203800	26%
Number of Block RAMs	22	445	4%
Number of DSP48E1s	150	840	17%
Max. Freq. (MHz)		68.036	

Tab. II *Synthesis results for Example 1.*

6.2 System identification problem – 2

System identification given in Eq. 22 has been implemented for this example [20].

$$
\begin{aligned}
y(k+1) = & -1.17059y[k-1] + 0.606861y(k) + \\
& 0.679190y^2[k]y[k-1] - 0.136235y^4[k]y[k-1] + \\
& 0.165646y^3[k]y[k-1] - 0.00711966y^6[k-1] + \\
& 0.114719y^5[k]y[k-1] - 0.0314354y[k]y[k-1] + \\
& 0.0134682y^3[k]
\end{aligned}
\tag{22}
$$

For system identification, the MLP with the inputs of $y[k]$ and $y[k-1]$, and with 3 cells in its hidden layer and 1 cell in output layer has been used. Tangent hyperbolic function has been used as activation function for hidden layer cells, and linear activation function has been used for output cells. In obtaining input values, initial values have been selected as $y[k] = y[k-1] = 0.1$.

Fig. 16a, illustrates the hardware outputs related to LM algorithm and dynamic system identification process for Example 2. Fig. 16b shows the errors related to the hardware implementation of LM algorithm and dynamic system identification for Example 2.

Fig. 16a, the hardware outputs related to LM algorithm and dynamic system identification process are shown for Example 2. Fig. 16b shows the errors related to the hardware implementation of LM algorithm and dynamic system identification for Example 2.

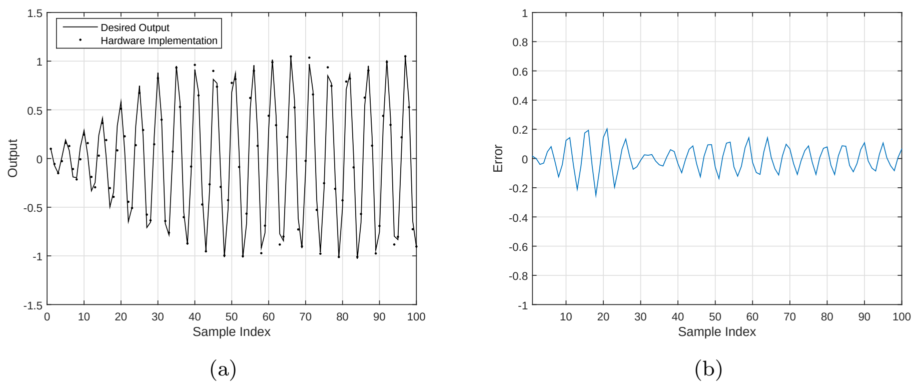


Fig. 16 *Experimental results of the training phase for the Example 2.*

Fig. 17a shows the hardware outputs related to the dynamic system identification process for input series created with initial values of $y[k] = y[k-1] = -0.1$. Fig. 17b shows the hardware output errors related to the dynamic system identification process for input series created with initial values of $y[k] = y[k-1] = -0.1$ for the network trained with LM algorithm.

Tab. III shows the fitness values according to the algorithms used in the training of the network, trained with the given input series, and the fitness values obtained

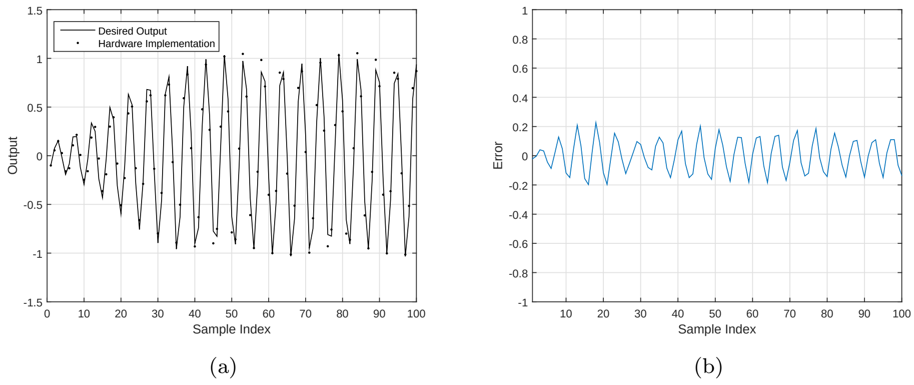


Fig. 17 *Experimental results of the test phase for the Example 2.*

for input series that are not shown to the network. Fitness values are given as the average of fitness values obtained from 10 different trainings.

Training	Test	Iteration
0.3844	0.6155	20

Tab. III *Fitness values of training and test data for Example 2.*

Tab. IV exhibits the synthesis results for the implementation of ANN training with LM algorithm on FPGA.

Logic Utilization	Used	Available	Utilization
Number of Slice Registers	19314	407600	4%
Number of Slice LUTs	53709	203800	26%
Number of Block RAMs	22	445	4%
Number of DSP48E1s	150	840	17%
Max. Freq. (MHz)		68.036	

Tab. IV *Synthesis results for Example 2.*

7. Conclusion

The most important advantage of ANNs is their parallel data processing feature. Therefore it is very important for ANN to be implemented within a hardware environment that will emphasize its parallel data processing feature. In this respect, FPGAs are one of the most suitable platforms used to implement this feature of ANNs. In the study, the implementation of ANN training over FPGA by using Levenberg and Marquardt algorithm is presented. Hardware implementation is tested by using dynamic system identification problems, and the experimental results obtained are shown in Fig. 14, Fig. 15, Fig. 16 and Fig. 17, and are given comparatively in Tab. I and Tab. III. As a result of hardware implementation for the system identification problems, illustrated also in the Tables and Figures, the test results obtained with samples shown or not shown to the network reveal that the ANN trained by using LM algorithm makes a successful generalization. As seen also in Tab. II and Tab. IV, the synthesis results obtained demonstrate that ANN with LM training can be successfully implemented on FPGA hardware.

References

[1] MERCHANT S., PETERSON G., KONG S. Intrinsic Embedded Hardware Evolution of Block-based Neural Networks, *International Conference on Engineering of Reconfigurable Systems & Algorithms*, 2006.

[2] KARAKUZU C., ÖZTÜRK S. A Comparison of fuzzy, neuro and classical control techniques based on an experimental application, *Journal of Quafquaz University*, 6, pp. 189–198, 2000, doi: 10.1109/EEEI.2000.924354.

[3] ÇAVUŞLU M. A., KARAKAYA F., ALTUN H. ÇKA Tipi Yapay Sinir Ağı Kullanılarak Plaka Yeri Tespitinin FPGA'da Donanımsal GerCeklenmesi, In: *Proc. Akıllı Sistemlerde Yenilikler ve Uygulamalar Sempozyumu*, 2008.

[4] LI X., AREIBI S. A Hardware Software Co-design Approach for Face Recognaiton, *Microelectronics, ICM 2004 Proceedings. The 16th International Conference on*, 2004, pp. 55–58, doi: 10.1109/ICM.2004.1434204.

[5] NARENDRA K.S., PARTHASARATY K. Identification and Control of Dynamical Systems Using Neural Network, *IEEE Transactions on Neural Networkworks*, 1990, 1(1), pp. 4–27, doi: 10.1109/72.80202.

[6] FERRARI S., JENSENIUS M. A constrained optimization approach to preserving prior knowledge during incremental training, *IEEE Trans. Neural Netw.*, 2008, 19(6), pp. 996–1009, doi: 10.1109/TNN.2007.915108.

[7] WILAMOWSKI B.M., CHEN Y., MALINOWSKI A. Efficient algorithm for training neural networks with one hidden layer, In: *Proceedings of the International Joint Conference on Neural Networks*, 1999, 3, pp. 1725–1728, doi: 10.1109/IJCNN.1999.832636.

[8] DOHNAL J. Using of Levenberg-Marquardt Method in Identification by Neural Networks, In: *Student EEICT 2004. Student EEICT 2004. Brno: Ing. Zdenk Novotn CSc.*, 2004, pp. 361–365.

[9] FERRER D., GONZALEZ R., FLEITAS R., ACLE J.P., CANETTI R. NeuroFPGA – Implementing Artificial Neural Networks on Programmable Logic Devices, *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, 2004, 3, pp. 218–223, doi: 10.1109/DATE.2004.1269233

[10] SAVICH A.W., MOUSSA M., AREIBI S. The Impact of Arithmetic Representation on Implementing MLP-BP on FPGAs: A Study, *IEEE Transactions on Neural Networks*, 18:1(2007), pp. 240–256, doi: 10.1109/TNN.2006.883002.

[11] FARMAHINI-FARAHANI A., FAKHRAIE S. M., SAFARI S. Scalable Architecture for on-Chip Neural Network Training using Swarm Intelligence, *Proc. of the Design, Automation and Test in Europe Conf. (DATE'08), Munich, Germany*, 2008, pp. 1340–1345, doi: 10.1109/DATE.2008.4484865.

[12] NEDJAH N., SILVA R.M.D., MOURELLE L.M.M., SILVA M.V.C.D. Dynamic MAC-based architecture of artificial neural networks suitable for hardware implementation on FPGAs, *Neurocomputing*, 2009, 72(10–12), pp. 2171–2179, doi: 10.1016/j.neucom.2008.06.027.

[13] ÇAVUŞLU M.A., KARAKUZU C., ŞAHİN S. Neural Networkwork Hardware Implementation Using FPGA, In: *ISEECE 2006 3rd International Symposium on Electrical, Electronic and Computer Engineering Symposium Proceedings, Nicosia, TRNC*, 2006, pp. 287–290.

[14] ÇAVUŞLU M.A., KARAKUZU C., ŞAHİN S., YAKUT M. Neural Network Training Based on FPGA with Floating Point Number Format and It's Performance, *Neural Computing & Application*, 2011, 20(2), pp. 195–202, doi: 10.1007/s00521-010-0423-3.

[15] ÇAVUŞLU M.A., KARAKUZU C., KARAKAYA F., Neural identification of dynamic systems on FPGA with improved PSO learning, *Applied Soft Computing*, 2012, 12(9), pp. 2707–2718, doi: 10.1016/j.asoc.2012.03.022

[16] WON E. A hardware implementation of artificial neural networks using field programmable gate arrays, *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 2007, 581(3), pp. 816–820, doi: 10.1016/j.nima.2007.08.163

[17] FERREIRA P., RIBEIRO P., ANTUNES A., DIAS F.M. A high bit resolution FPGA implementation of a FNN with a new algorithm for the activation function, *Neurocomputing*, 2006, 71(1-3), pp. 71-77, doi: 10.1016/j.neucom.2006.11.028

[18] ELLIOT D.L. A Better Activation Function for Artificial Neural Networks, *Technical Research Report T.R. 93-8, Institute for Systems Research, University of Maryland*, (1993).

[19] ÇAVUŞLU M.A., KARAKUZU C. Nöral ve Bulanık Sistem Hücre Aktivasyon Yaklaşımları ve FPGA'da Donanımsal Gerceklenmesi, Sakarya Üniversitesi Fen Bilimleri Enstitüsü Dergisi, 2011, 15(1), pp. 8–16.

[20] CHEN S., BILLINGS S.A. Neural networks for nonlinear dynamic system modelling and identification, *Int. J. Control*, 1992, 56(2), pp. 319–346, doi: 10.1080/00207179208934317.