Expert Systems with Applications 38 (2011) 673–681

Contents lists available at ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa

Implementation of a neuro-fuzzy network with on-chip learning and its applications

Cheng-Jian Lin [a,*], Chi-Yung Lee [b]

[a] Department of Computer Science and Information Engineering, National Chin-Yi University of Technology, Taichung County 411, Taiwan, ROC
[b] Department of Computer Science and Information Engineering, Nankai University of Technology, Nantou 542, Taiwan, ROC

ARTICLE INFO

Keywords:
Neural fuzzy network (NFN)
Field programmable gate array (FPGA)
Backpropagation (BP) method
Simultaneous perturbation
Gaussian function

ABSTRACT

The implementation of adaptive neural fuzzy networks (NFNs) using field programmable gate arrays (FPGA) is proposed in this study. Hardware implementation of NFNs with learning ability is very difficult. The backpropagation (BP) method in the learning algorithm is widely used in NFNs, making it difficult to implement NFNs in hardware because calculating the backpropagation error of all parameters in a system is very complex. However, we use the simultaneous perturbation method as a learning scheme for the NFN hardware implementation. In order to reduce the chip area, we utilize the traditional non-linear activation function to implement the Gaussian function. We can confirm the reasonableness of NFN performance through some examples.

1. Introduction

The uses of traditional neural fuzzy networks (NFNs) (Juang & Lin, 1998; Lin & Lee, 1996; Lin & Lin, 1997; Lin, Lin, & Shen, 2001; Paul & Kumar, 2002; Takagi & Sugeno, 1985; Zhang & Kandel, 1998) have been applied in many fields. Neural fuzzy networks (NFNs) bring the low-level learning and computational power of neural networks into fuzzy systems and give the high-level human-like thinking and reasoning of fuzzy systems to neural networks.

NFNs have been mostly implemented in software, but the disadvantage of the software method is the lack of real-time. Therefore, we propose implementing NFNs in hardware in this study. Neural networks have been successfully implemented before. Krips, Lammert, and Kummert (2002) proposed using field programmable gate arrays (FPGA) to implement neural networks through parallel computing in a real-time hand tracking system. Mohd-Yasin, Tan, and Reaz (2004) realized IRIS recognition for biometric identification employing neural networks on FPGA devices that allow for efficient hardware implementation. Hannan Bin Azhar and Dimond (2002) considered a RAM-based neural network using an alternative hardware solution to be implemented on FPGA devices for collision-free robot navigation. However, their implementation using hardware has resulted in a lack learning ability.

Implementing the learning mechanism of a NFN in hardware is a difficult issue (Sheu & Choi, 1995). We know that the backpropagation method is commonly used. The backpropagation method is difficult to implement using hardware because calculating the backpropagation error of all parameters in a system is very difficult. The backpropagation method requires a large number of logic gates in the hardware. From this point of view, we must use the easy learning capability of hardware to realize the implementation of NFNs.

In this study, we use the simultaneous perturbation method by Maeda and De Figueiredo (1997); Maeda and Kanata (1993). The advantage of the simultaneous perturbation optimization method is its simplicity. The method can estimate the gradient using only values of the error function. Therefore, implementation of this learning algorithm is relatively easier than the implementation of other learning algorithms. The simultaneous perturbation method does not have to take the backpropagation circuit into account.

This study adopts field programmable gate array (FPGA) devices to realize the hardware implementation of a NFN model. NFNs have been implemented using FPGAs because FPGAs are programmable and flexible. In recent years, FPGA have been used in many applications because the programmable logic element increases logic, speed, and memory capability and adds many extra functions. After the designing procedure is finished, the circuit can be easily and quickly implemented using a very high speed integrated circuit hardware description language (VHDL).

The advantages of this study are summarized below.

(1) The hardware implementation of a neural fuzzy network (NFN) is proposed for classification and prediction problems.
(2) We utilize the traditional non-linear activation function to replace the Gaussian function to reduce chip area.
(3) An online learning algorithm is applied to adjust the parameters of the NFN model in Section 3.
(4) We use a very high speed integrated circuit hardware description language (VHDL) to design a NFN with learning ability and implement using field programmable gate arrays (FPGAs).

* Corresponding author.
E-mail address: cjlin@ncut.edu.tw (C.-J. Lin).

0957-4174/$ - see front matter Crown Copyright © 2010 Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.eswa.2010.07.019

(5) As demonstrated in Section 5, the NFN model is demonstrated to be adaptive and effective for solving classic exclusive OR (XOR) problems and predicting chaotic signals.

The remainder of this study is organized as follows. Section 2 describes the structure of neural fuzzy networks. Section 3 describes the simultaneous perturbation algorithm. The basic architecture and implementation of a NFN is presented in Section 4. Section 5 presents the results of the simulation of several problems. Finally, the conclusions are given in the last section.

2. The structure of NFN

In this section, the structure of the a NFN is introduced. The standard four-layer network (Lin, Lin, & Shen, 2001) is used to realize a fuzzy model of the following form:

$$R_j : IFx_1 \text{ is } A_{1j} \text{ and } x_2 \text{ is } A_{2j} \ldots \text{ and } x_n \text{ is } A_{nj} \text{ THEN } y' \text{ is } w_j, \quad (1)$$

where x_i is an input variable, y' is an output variable, A_{nj} is a linguistic term of the precondition part, w_j is a constant consequent part, and n is the number of input variables.

The structure of the NFN is shown in Fig. 1, where the functions of the nodes in each layer of the NFN model are described as follows.

Layer 1: No computation is done in this layer. Each node in this layer is an input node, which corresponds to one input variable, and only transmits input values to the next layer directly.

$$u_i^{(1)} = x_i. \quad (2)$$

Layer 2: Nodes in this layer corresponds to one linguistic label of the input variables in Layer 1, i.e., the membership value specifying the degree to which an input value belongs to a fuzzy set is calculated in Layer 2. The Gaussian membership function, the operation performed in Layer 2, is

$$u_{ij}^{(2)} = \exp\left(-\frac{[u_i^{(1)} - m_{ij}]^2}{\sigma_{ij}^2}\right), \quad (3)$$

where m_{ij} and σ_{ij} are, respectively, the mean and variance of Gaussian membership function of the jth term of the ith input variable x_i.

Layer 3: Nodes in this layer represents the precondition part of one fuzzy logic rule. They receive the one-dimensional membership degrees of the associated rule from the nodes of a set in Layer 2. This layer is denoted by I, which multiplies the incoming signals and outputs the product result. As a result, the output function of each inference nodes is

$$u_j^{(3)} = \prod_i u_{ij}^{(2)}, \quad (4)$$

where $u_j^{(3)}$ is the output of the jth rule node.

Layer 4: This layer acts a defuzzifier. The single node in this layer is labeled $\sum$, and its sums all incoming signals to obtain the final inferred result

$$u^{(4)} = \sum_j u_j^{(3)} w_j, \quad (5)$$

where the weight w_j is the output action strength associated with the jth rule, and $u^{(4)}$ is the output of the NFN.

3. Simultaneous perturbation algorithm

When this study utilizes the gradient descent method as a learning algorithm to train a NFN, the parameters are updated iteratively. First, this study defines a cost function as follows

$$J(u) = \frac{1}{2}(y - y_d)^2, \quad (6)$$

where u represents the adjusted parameter of the NFN, and y_d and y are the desired output and the current output, respectively. The

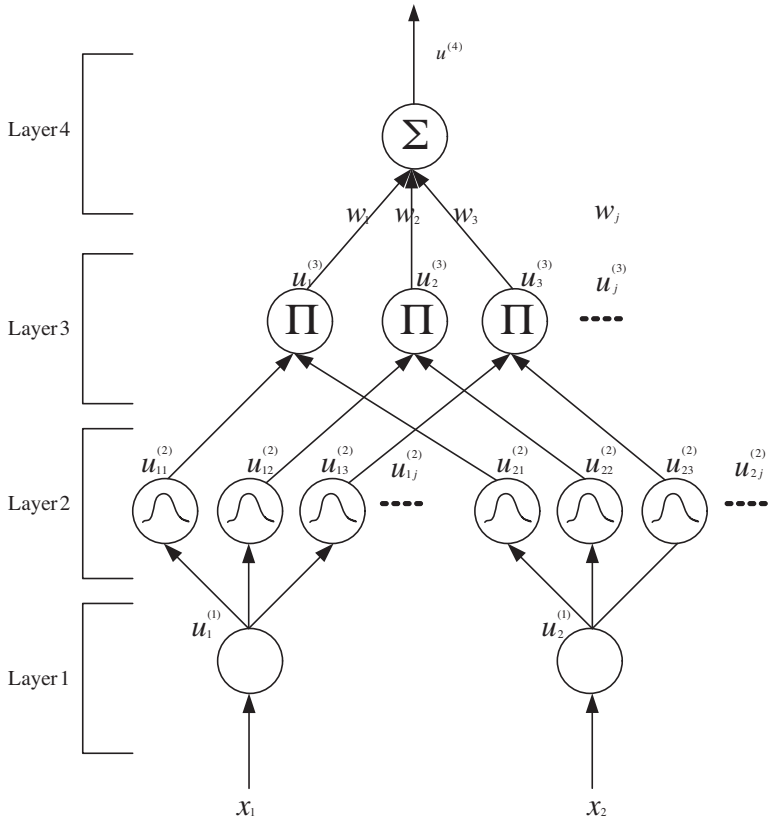

Fig. 1. Structure of a NFN model.

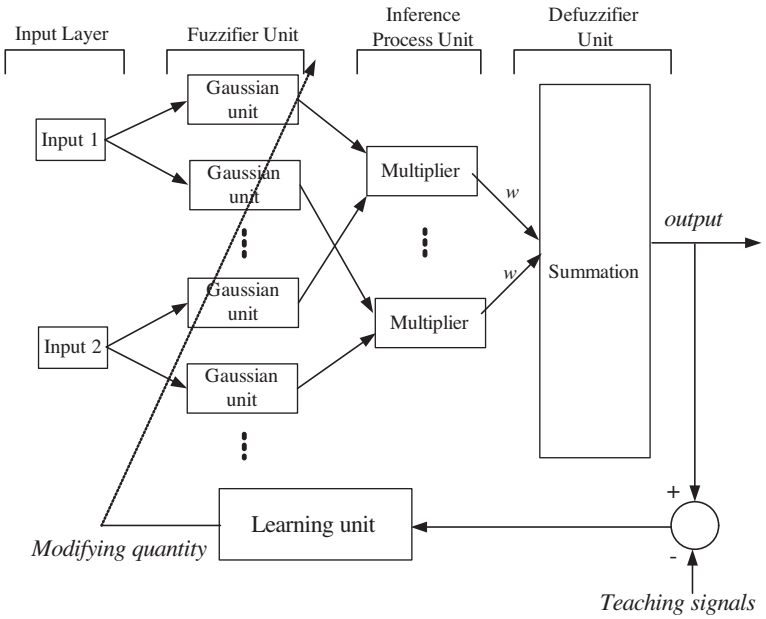

Fig. 2. The overall component of the NFN model with learning ability.

Table 1
Data representation.

Value	Data representation Format: s4·5
8	0 1000 00000
4	0 0100 00000
2	0 0010 00000
1	0 0001 00000
0.5	0 0000 10000
0.25	0 0000 01000
0.0125	0 0000 00100
0.0625	0 0000 00010
0.03125	0 0000 00001

Table 2
Representing values.

Value	Data representation format: s4·5	Encoded value
0.2849	0 0000 01001	0.28125
1.7482	0 0001 10111	1.71875
−4.6741	1 0100 10101	−4.65625

optimization target is characterized to minimize the following error function with respect to the adjusted parameters u of the NFN. Then, according to the gradient descent method, the parameters are updated using

$$\Delta u = -\eta \frac{\partial J(u)}{\partial u}, \tag{7}$$

where η is the learning rate. We use the chain rule for $\partial J(u)/\partial u$

$$\Delta u = -\eta \frac{\partial J(u)}{\partial u} = -\eta \frac{\partial J(u)}{\partial y}\frac{\partial y}{\partial u} = -\eta (y - y_d)\frac{\partial y}{\partial u} = -\eta e \frac{\partial y}{\partial u}. \tag{8}$$

In Eq. (8), e, the error between the desired output and the current output, can be calculated easily, but $\partial y/\partial u$ requires more complicated computation.

In order to improve the above-mentioned complicated computation problem, many researchers (Maeda, 1993; Spall, 1987) used the different approximation approach to obtain a derivation of a function. They added a small perturbation factor c to the ith parameter of the parameter vector, u_i, which is defined in Eq. (9):

$$u_i = (u_1, \ldots, u_i + c, \ldots, u_k), \tag{9}$$

where k is the number of adjustable parameters. They utilized the different approximation approach to derive $\partial y/\partial u$. This can calculate the amount of parameter modification for the NFN:

$$\Delta u = -\eta \frac{\partial J(u)}{\partial u} \approx -\eta \frac{J(u_i) - J(u)}{c}. \tag{10}$$

On the other hand, they also utilized the different approximation approach to derive the $\partial y/\partial u$ as follows:

$$\frac{\partial y}{\partial u} \approx \frac{f(u_i) - f(u)}{c}. \tag{11}$$

Although the different approximation approach is simple, it requires many forward operations in the NFN. When the number of adjustable parameters is k, k-times forward operations are required to complete the amount of parameter modifications for all parameters. Therefore, when the number of adjustable parameters in the network is large, this approach is not suitable.

In order to improve the above-mentioned disadvantages, we adopt the simultaneous perturbation method proposed by Maeda, Hirano, and Kanata (1995). First, we define a perturbation vector that is added to all parameter of the NFN as follows:

$$c_l = (c_l^1, \ldots, c_l^n), \tag{12}$$

where the l denotes an iteration. The perturbation vector c_l is an uniformly random number in the interval $[-c_{\max}, c_{\max}]$ except the interval $[-c_{\min}, c_{\min}]$.

The simultaneous perturbation learning rule with the parameters is updated and the parameter modifying quantity is described as follows:

$$u_{l+1} = u_l + \Delta u_l, \tag{13}$$

$$\Delta u_l^i = -\eta e_l \frac{f(u_l + c_l) - f(u_l)}{c_l^i}, \tag{14}$$

where u_l is the adjustable parameter of the NFN, η is a positive the learning rate of the parameters of the NFN, and Δu_l is the update of the modifying quantity.

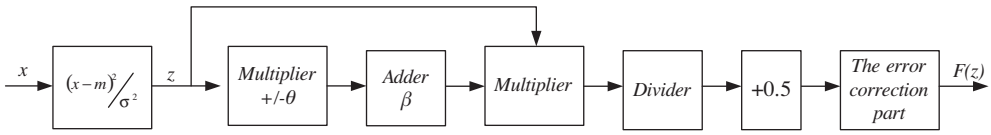

Fig. 3. A block diagram of a hardware implementation of the Gaussian approximation.

4. Hardware implementation of NFN

In this section, we introduce the hardware implementation of the NFN structure, the data representation, and the learning method. The overall component of the NFN model with learning ability is shown in Fig. 2. It consists of four main parts: (1) a fuzzifier unit; (2) an inference processing unit; (3) a defuzzifier unit; and (4) a learning unit. Detailed descriptions are given in Section 4.2–4.5 and 4.6, respectively. In Section 4.1, we describe the data representation of fixed-point numbers. In Section 4.2, the traditional non-linear activation function is adopted to replace the Gaussian function in the NFN model. In Section 4.3, we implement the inference processing unit that calculates the result using the designed multiplier. Hardware implementation of the defuzzifier unit is described in Section 4.4. All modified quantities of parameters are calculated in Section 4.5. Finally, in Section 4.6, we include a limiter to prevent overflow in the parameter modification parts.

4.1. Data representation

In this study, we use a fixed-point number for NFNs to maintain consistency and effectiveness of the representation of data which are the same. The encoding technique uses digital values as a means to represent the respective data (Blake & Maguire, 1998; Tommiska, 2003). The fixed-point format is defined as follows:

$$[s]a \cdot b, \tag{15}$$

where the optional s denotes a sign bit with 0 for positive numbers and 1 for negative numbers, a is the number of integer bits, and b is the number of fractional bits.

Table 1 shows the principal values with the above-mentioned technique. Table 1 shows that the fixed-point numbers are easily accommodated in this system. Therefore, the smallest value can be represented by −15.86875 (1 1111 11111) and the largest value by 15.86875 (0 1111 11111). Table 2 shows the corresponding values by data representation. It shows some errors between the principal values and the encoded values. For all practical systems it is possible to choose a word-length long enough to reduce the finite precision effects in a negligible level, and it is often desirable to use as few bits as possible while achieving user-defined output error conditions in order to optimize area, power, or speed (Inacio & Ombres, 1996; Ewe, Cheung, & Constantinides, 2004). This study uses 10 bits as the number of word-length for all operations.

4.2. Fuzzifier unit

This module implements the fuzzification operator in Eq. (3). In this module, the Gaussian function is the main part of the structure of the NFN for the fuzzy rules. In Eq. (3), we know that the operator of the Gaussian function is very complicated using the traditional non-linear activation functions. The Taylor series with a look-up table (LUT) (Lin & Tsai, 2007) has been adopted to approximate the implementation of the Gaussian function. The method requires a quite large number of hardware resources. Therefore, it is unsuitable for direct digital implementation in hardware. A reasonable approximation of a non-linear function can be implemented directly using digital techniques. The following equation is a second order non-linear function (Blake & Maguire, 1998):

$$F(z) = \begin{cases} (z \cdot (\beta - \theta \cdot z) + 1)/2 & \text{for} \quad 0 \leqslant z \leqslant L, \\ (z \cdot (\beta + \theta \cdot z) + 1)/2 & \text{for} \quad -L \leqslant z \leqslant 0, \end{cases} \tag{16}$$

where β and θ represent the slope and the gain of the non-linear function $F(z)$ between the saturation regions $-L$ and L. Taking the upper and lower saturation regions to be equal to 2, we get the following expressions for θ and β:

$$\theta = \pm\frac{1}{4} \quad \text{and} \quad \beta = 1.$$

Fig. 3 shows a block diagram of the hardware implementations of the Gaussian approximation. It uses two multipliers, an adder, a divider, and an error correction part. The flow diagrams of the multiplier and divider algorithms are shown in Figs. 4 and 5, respectively. In Fig. 6 we compare the representative Gaussian function and the implemented Gaussian function using the second order non-linear function without the error correction part. Fig. 7 shows a comparison between the practical Gaussian function and the implemented Gaussian function using the second order non-linear function with the error correction part. The errors in the above two methods for the representative Gaussian function approximation are shown in Figs. 8 and 9. The resource costs for a multiplier

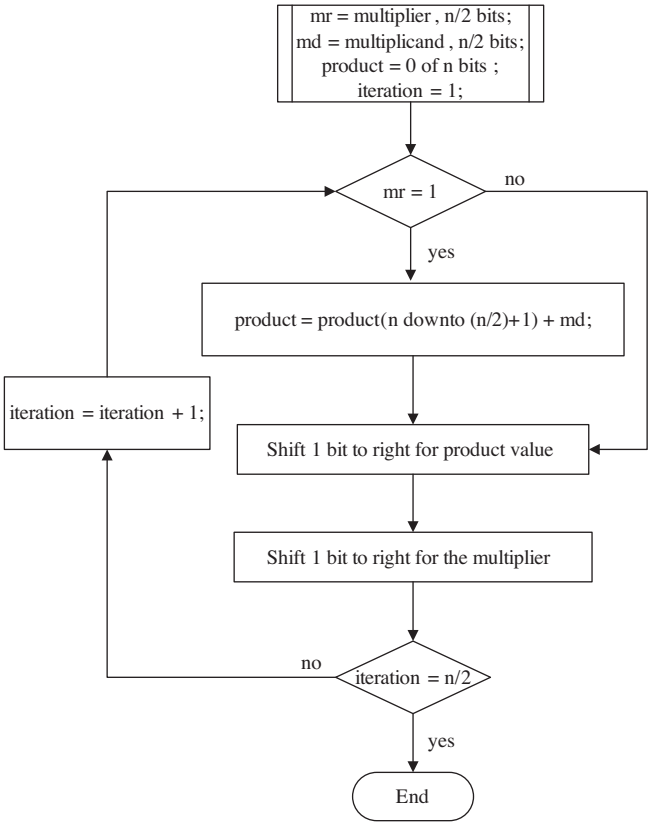

Fig. 4. A flow diagram of the multiplier algorithm.

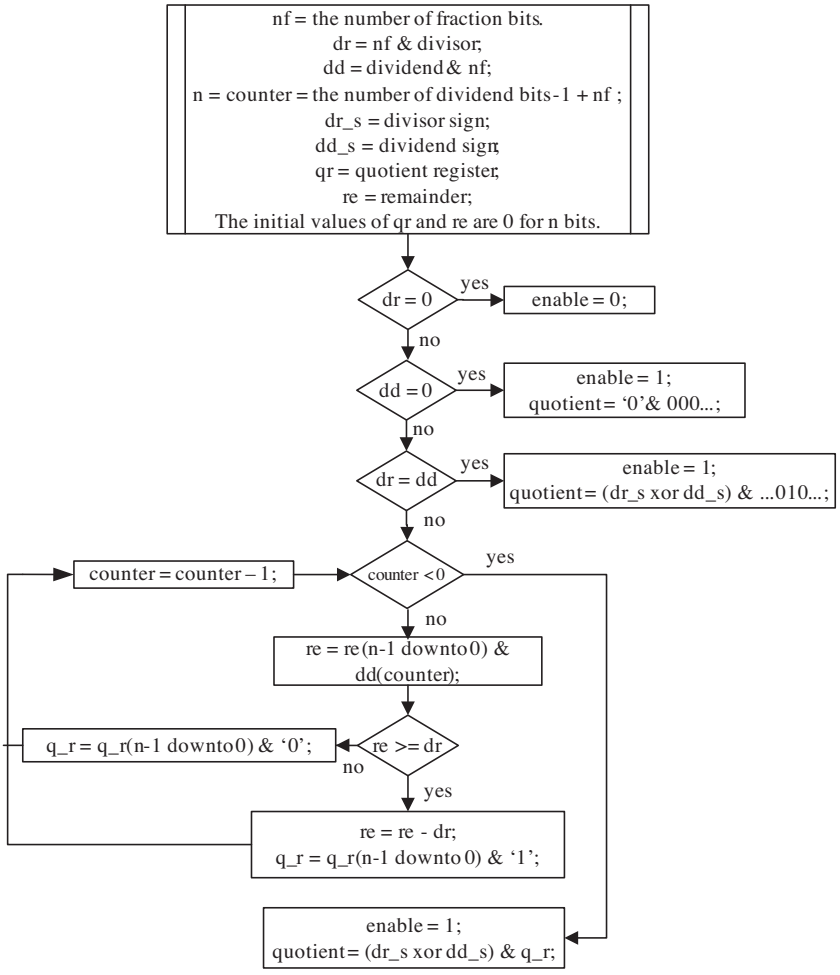

Fig. 5. A flow diagram of the divider algorithm.

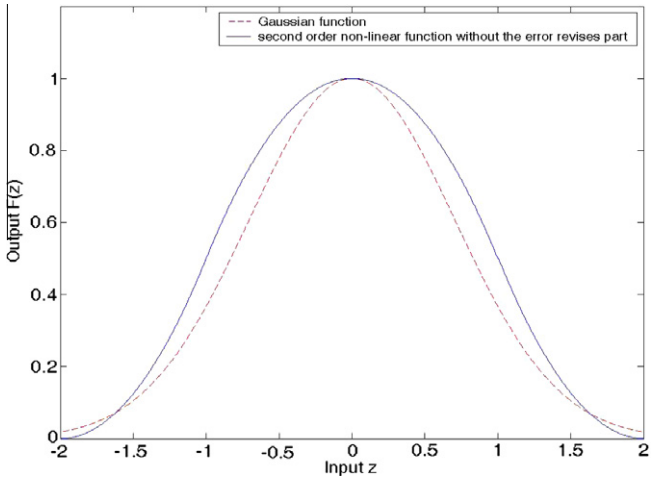

Fig. 6. The representative Gaussian function and the implemented Gaussian function using a second order non-linear function without the error correction part.

and a divider are tabulated in Tables 3 and 4, respectively. We compare the resource requirement for our utilization method with an adopted Taylor series method which has a look-up table (Lin & Tsai, 2007). The comparison results are tabulated in Table 5. The result shows that our utilization method costs fewer resources than other methods (Lin & Tsai, 2007).

4.3. Inference processing unit

The main work of the inference processing unit is to perform the multiplication operation in Eq. (4). Fig. 10 shows a block diagram of the inference processing unit in which each product module is made up of one or more multipliers. The hardware implementation of the multiplier is shown in Fig. 11.

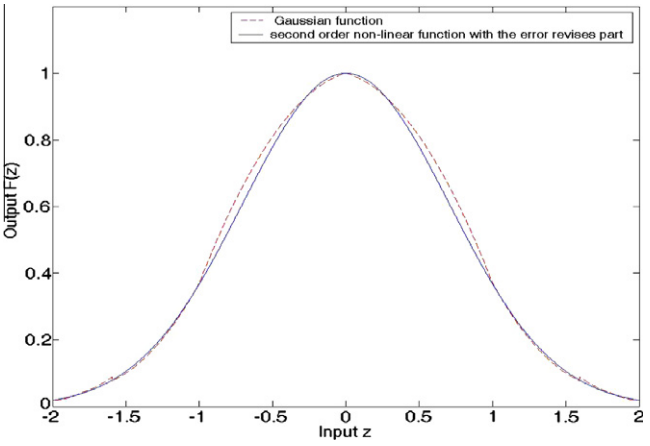

Fig. 7. The representative Gaussian function and implemented the Gaussian function using a second order non-linear function with the error correction part.

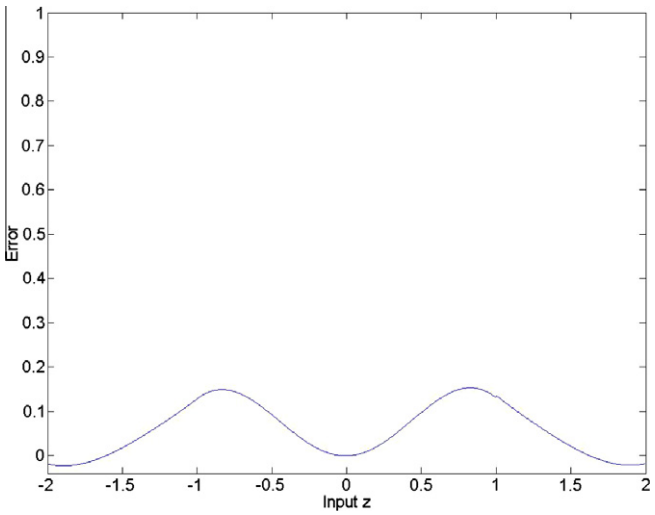

Fig. 8. Error between the representative Gaussian function and an approximation method of a second order non-linear function without the error correction part.

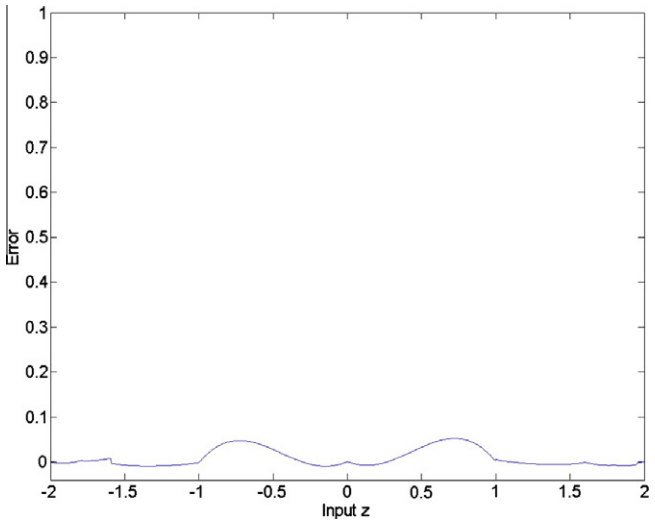

Fig. 9. Error between the representative Gaussian function and an approximation method of a second order non-linear function with the error correction part.

Table 3
Resource requirement for a multiplier implementation.

Device selected	XC2V6000		
Features	Supply	Utilized	%
Number of Slices	33792	17	0.05
Number of Slice Flip Flops	67584	30	0.04
Number of bonded IOBs	824	62	7.52
Number of MULT18X18s	144	1	0.69
Number of GCLKs	16	1	6.25

Table 4
Resource requirement for a divider implementation.

Device selected	XC2V6000		
Features	Supply	Utilized	%
Number of Slices	33792	821	2.43
Number of Slice Flip Flops	67584	24	0.36
Number of 4 input LUTs	67584	1455	2.15
Number of bonded IOBs	824	72	8.74
Number of GCLKs	16	1	6.25

4.4. Defuzzifier unit

The defuzzifier unit implements Eq. (5) and is shown in Fig. 12. First, signal $u_j^{(3)}$ and all w_j parameters are multiplied by multipliers. All initial wj parameters are random using a linear feedback shift register (LFSR). After multiplication, an adder is still needed to sum all the values. If the number of rules is R, there will be R multipliers and one accumulator in total.

4.5. Learning unit module

The learning unit is shown in Fig. 13. This unit achieves the learning ability using simultaneous perturbation. The equation is represented in Eq. (14). The different perturbation factors are added in each parameter, so each parameter has different modified values. The perturbation c is an uniformly random number in the interval $[-c_{\max}, c_{\max}]$ and in the interval$[-c_{\min}, c_{\min}]$. Using LFSR counters to address the RAM makes the design even simpler. An n-bit LFSR counter has a maximum sequence of $2^n - 1$ states.

First, the learning unit is a calculation that calculates part of $f(u_l + c_l)$ and e_l. We can calculate the error between the output of the NFN and the teaching signal. The next multiplication in Fig. 13 is $f(u_l + c_l)^* e_l^* \alpha$. It is worth mentioning that in our circuit, and α is chosen to be $0.9 \sim 0.001$. Finally, the signal is divided by a randomly generated number and then the modified value can be calculated using Eq. (14).

4.6. Parameter modification part

The updated parameter values are calculated in the parameter modification part. The block diagram of the parameter modification for the NFN is shown in Fig. 14. The sum of the parameters (w, m, σ) and modified quantity $(\Delta w, \Delta m, \Delta\sigma)$ uses an adder for the NFN. In Fig. 14, after the parameters are refreshed, the refreshed parameters may result in an overflow of the data representation. Therefore, we must use a limiter in the refresher. The limited numerical value of the parameters is in the range [−16, +16]. Finally, the output of this part is equal to the new parameter value.

5. Illustrative examples

In this study, we present the implementation of the XOR problem and a prediction of a chaotic signal problem. Neural fuzzy networks are designed to solve these two problems. The FPGA chip design is checked and configured using the software XILINX ISE6.2i. The chip circuit of the FPGA uses Xilinx Virtex-II XC2V6000-4FF1152C, which contains 6,000,000 logic gates. The perturbation value c is an uniformly random number in the interval $[-0.01, 0.01]$ except an interval $[-0.001, 0.001]$.

Example 1 (*Exclusive OR*). The decision-plane dichotomization mentioned above does not always exist for a given set of patterns. A famous example is the XOR problem. The desired output is 1 when one of the inputs is 1, and the desired output is 0 when both inputs are 1 or 0. The input patterns and the corresponding desired output are

$$\left(x^{(1)} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, d^{(1)} = 0\right), \quad \left(x^{(2)} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, d^{(2)} = 1\right),$$
$$\left(x^{(3)} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, d^{(3)} = 1\right), \quad \left(x^{(4)} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, d^{(4)} = 0\right).$$

Table 5
Resource requirement for our utilization method and an adopted Taylor series method with look-up table.

Device selected	XC2V6000				
Features	Supply	Utilized	%	Taylor + LUT (Lin & Tsai, 2007)	%
Number of Slices	33792	935	2.77	1682	4.98
Number of Slice Flip Flops	67584	133	0.19	887	1.31
Number of 4 input LUTs	67584	1725	2.6	3110	4.6
Number of bonded IOBs	824	109	13.2	106	12.9
Number of MULT18X18s	144	2	1.4	9	6.25
Number of GCLKs	16	1	6.25	1	6.25

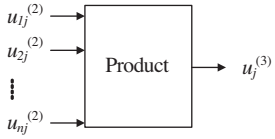

Fig. 10. Inference processing unit module.

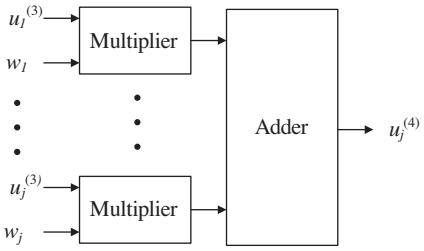

Fig. 12. Defuzzifier unit module.

In this example, the NFN contains only two input nodes and two output nodes using four configured fuzzy rules to perform the XOR problem. The hardware implementation of the NFN uses about 511,000 logic gates. The maximum frequency is 12.618 MHz. The learning rate of $\eta = 0.046875$ was set in the hardware. Initial parameters are random in the interval [0,1]. The random number generation part uses a linear feedback shift register (LFSR). Table 6 shows a comparison between the results of the XOR problem using hardware and software implementation. In the table, we can see the expected output result of the NFN implemented in hardware and software using the simultaneous perturbation method. Table 6 shows that the results of the NFN implementation in hardware and software are similar.

Example 2 (*Prediction of a Chaotic Signal*). In this example, the implemented NFN model was used to predict a chaotic signal. The classical time series prediction problem is an one-step-ahead prediction (Lin & Lee, 1996). The following equations describe the logistic function:

$$x(k+1) = ax(k)(1 - x(k)).$$

The behavior of the time series generated by this equation depends critically upon the value of the parameter a. If $a < 1$, the system has a single fixed point at the origin, and from a random initial value between [0,1] the time series collapses to a constant value. For $a > 3$, the system generates a periodic attractor. Beyond the value $a = 3.6$, the system becomes chaotic. In this study, we set the parameter value a to 3.8. The first 60 pairs (from $x(1)$ to $x(60)$), with the initial value $x(1) = 0.001$, were the training data set, while the remaining 100 pairs (from $x(1)$ to $x(100)$), with initial value $x(1) = 0.9$, were the testing data set used for validating the proposed method.

The learning rate $\eta = 0.0396875$ was used in the hardware implementation of the NFN. Initial parameters are random using LFSR in the interval [0,1]. The implementation of the NFN in hardware and the software uses five fuzzy logic rules. The NFN implementation requires approximately 377,000 logic gates on a FPGA chip. The maximum frequency is 12.679 MHz.

Fig. 15 shows the prediction results from the software implementation of the NFN and desired output to learning process for 1000 generations. The "+" represents the desired output of the time series, and the notation "$*$" represents the NFN output of the software implementation. The prediction errors of both outputs above are shown in Fig. 16. The experimental results demonstrate the prediction capability of the NFN.

Now we discuss the performance of the NFN output from the hardware implementation and compare it with the performance of the NFN output from the software implementation. Fig. 17 shows the prediction results of the desired output and the NFN output from hardware implementation. The prediction errors of both outputs above are shown in Fig. 18. From the error value,

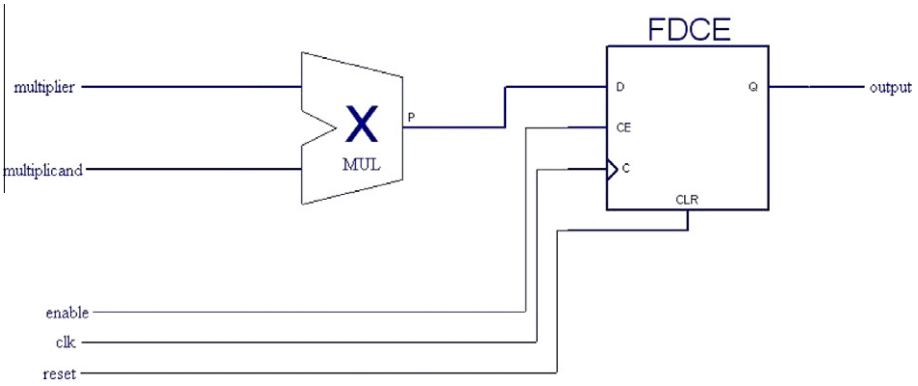

Fig. 11. Multiplier module.

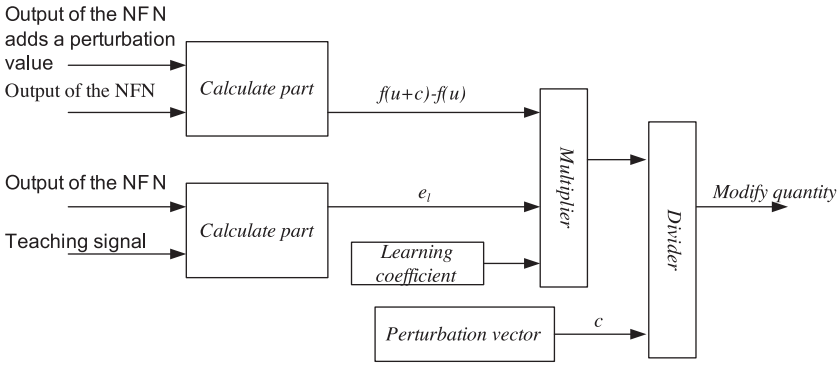

Fig. 13. Learning unit.

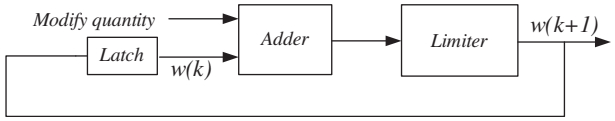

Fig. 14. Parameter modification part.

Table 6
Simulation results with hardware implementation and software implementation.

Input 1	Input 2	Desired output	The hardware implementation of NFN output	The software implementation of NFN output
0	0	0	0.02493	0.00235
0	1	1	0.97348	0.99865
1	0	1	0.98535	0.99933
1	1	0	0.01934	0.00549

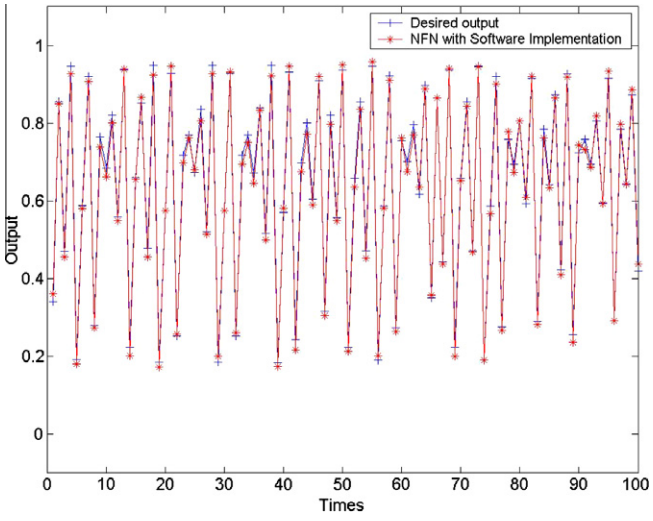

Fig. 15. Prediction results of the NFN output from software implementation.

the differences between the NFN output from the hardware implementation and the NFN output from the software implementation are similar.

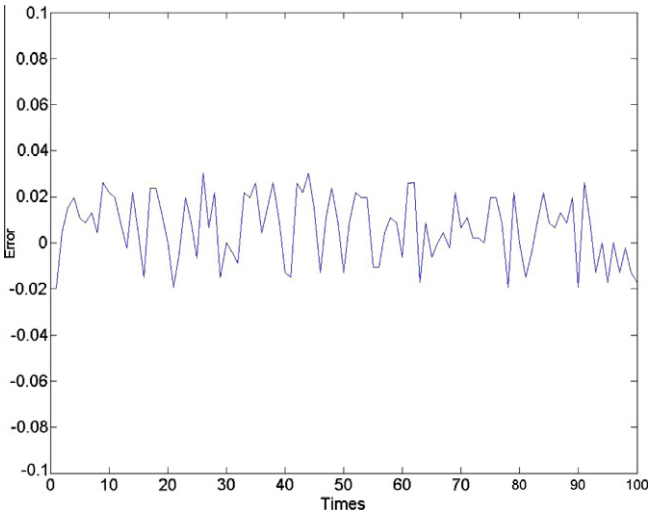

Fig. 16. Error between the NFN output from software implementation and the desired output.

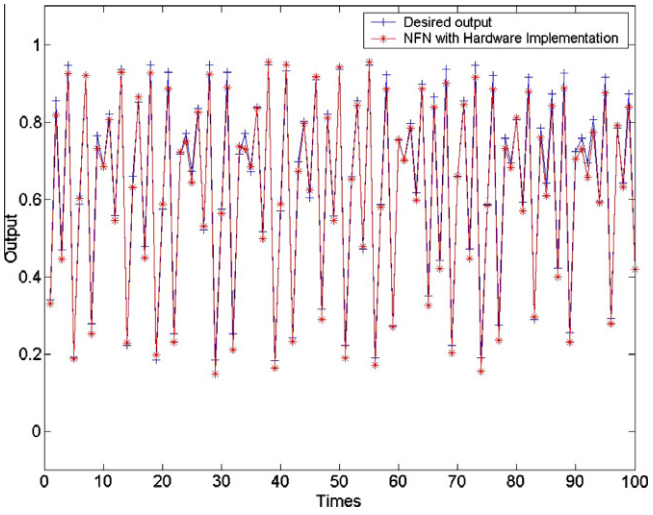

Fig. 17. Prediction results of the NFN output from hardware implementation.

6. Conclusion

In this study, the NFN was successfully implemented in hardware on a FPGA chip using the simultaneous perturbation algorithm. We replaced the difficult to implement MP method with the simple and easy simultaneous perturbation method. In order to reduce chip area, we utilized the traditional non-linear activation function to implement the Gaussian function. The re-

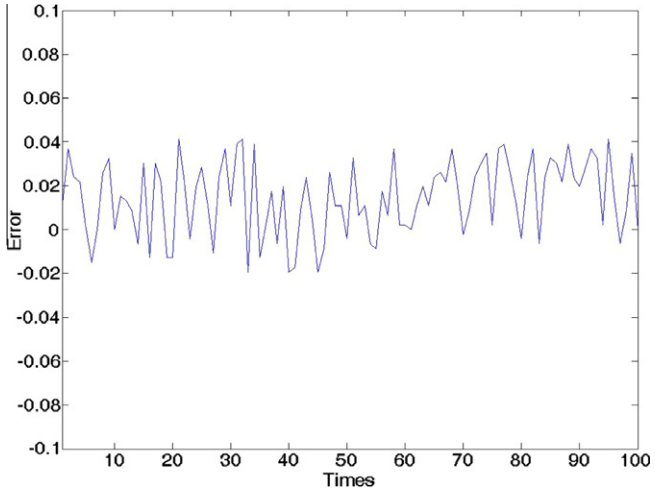

Fig. 18. Error between the NFN output from hardware implementation and the desired output.

sults of experiments with samples show that hardware implementation of the NFN with simultaneous perturbation algorithm using FPGA was successful. In the future, we will develop other techniques to implement the NFN on a FPGA chip with fewer chip area, and extend the methodology to implement recurrent neural networks.

References

Blake, J. J., & Maguire, L. P. (1998). The implementation of fuzzy systems, neural networks and fuzzy neural networks using FPGAs. *Information Sciences, 112*(Dec.), 151–168.

Ewe, C. T., Cheung, P. Y. K., & Constantinides, G. A. (2004). Dual fixed-point: An efficient alternative to floating-point computation. *Lecture Notes in Computer Science, 3203*, 200–208.

Hannan Bin Azhar, M. A., & Dimond, K. R. (2002). Design of an FPGA based adaptive neural controller for intelligent robot navigation. *Proceedings of the euromicro symposium on digital system design*, 283–290.

Inacio, C., & Ombres, D. (1996). The DSP decision: Fixed point or floating? *IEEE Spectrum, 33*(9), 72–74.

Juang, C. F., & Lin, C. T. (1998). An on-line self-constructing neural fuzzy inference network and its applications. *IEEE Transactions on Fuzzy Systems, 6*(1), 12–31.

Krips, M., Lammert, T., & Kummert, A. (2002). FPGA implementation of a neural network for a real-time hand tracking system. *Proceedings of the first IEEE international workshop on electronic design, test and applications*, 313–317.

Lin, C. T., & Lee, C. S. G. (1996). *Neural fuzzy systems: A neuro-fuzzy synergism to intelligent systems*. NJ: Prentice-Hall.

Lin, C. J., & Lin, C. T. (1997). An ART-based fuzzy adaptive learning control network. *IEEE Transactions on Fuzzy Systems, 5*(4), 477–496.

Lin, F. J., Lin, C. H., & Shen, P. H. (2001). Self-constructing fuzzy neural network speed controller for permanent-magnet synchronous motor drive. *IEEE Transactions on Fuzzy Systems, 9*(5), 751–759.

Lin, C. J., & Tsai, H. M. (2007). FPGA implementation of a wavelet neural network with particle swarm optimization. *Mathematical and Computer Modelling: An International Journal*.

Maeda, Y. (1993). Learning rule of neural networks for inverse systems. *Electronic Communication of Japan, 76*, 17–23.

Maeda, Y., & De Figueiredo, R. J. P. (1997). Learning rules for neuro-controller via simultaneous perturbation. *IEEE Transactions on Neural Networks, 8*(5).

Maeda, Y., Hirano, H., & Kanata, Y. (1995). A learning rule of neural networks via simultaneous perturbation and its hardware implementation. *Neural Networks, 8*, 251–259.

Maeda, Y., & Kanata, Y. (1993). Learning rules for recurrent neural networks using perturbation and their application to neuro-control. *Transactions on Institute of Electronic Engineering of Japan, 113-C*, 402–408.

Mohd-Yasin, F., Tan, A. L., & Reaz, M. I. (2004). The FPGA prototyping of IRIS recognition for biometric identification employing neural network. *Proceedings of the 16th international conference on microelectronics*, 458–461.

Paul, S., & Kumar, S. (2002). Subsethood-product fuzzy neural inference system (SuPFuNIS). *IEEE Transactions on Neural Networks, 13*(3), 578–599.

Sheu, B. J., & Choi, J. (1995). *Neural information processing and VLSI*. Boston, MA: Kluwer.

Spall, J.C. (1987). A stochastic approximation technique for generating maximum likelihood parameter estimates. In *Proceedings of 1987 American control conference* (pp. 1161–1167).

Takagi, T., & Sugeno, M. (1985). Fuzzy identification of systems and its applications to modeling and control. *IEEE Transactions on System, Man, and Cybernetics, SMC-15*, 116–132.

Tommiska, M. T. (2003). Efficient digital implementation of the sigmoid function for reprogrammable logic. *Computers and Digital Techniques, 150*(Nov.), 403–411.

Zhang, Y. Q., & Kandel, A. (1998). Compensatory neuro-fuzzy systems with fast learning algorithms. *IEEE Transactions on Neural Networks, 9*(1), 83–105.