Simultaneous Perturbation Learning Rule for Recurrent Neural Networks and Its FPGA Implementation

Yutaka Maeda, *Member, IEEE,* and Masatoshi Wakamura

***Abstract*—Recurrent neural networks have interesting properties and can handle dynamic information processing unlike ordinary feedforward neural networks. However, they are generally difficult to use because there is no convenient learning scheme. In this paper, a recursive learning scheme for recurrent neural networks using the simultaneous perturbation method is described. The detailed procedure of the scheme for recurrent neural networks is explained. Unlike ordinary correlation learning, this method is applicable to analog learning and the learning of oscillatory solutions of recurrent neural networks. Moreover, as a typical example of recurrent neural networks, we consider the hardware implementation of Hopfield neural networks using a field-programmable gate array (FPGA). The details of the implementation are described. Two examples of a Hopfield neural network system for analog and oscillatory targets are shown. These results show that the learning scheme proposed here is feasible.**

***Index Terms*—Field-programmable gate array (FPGA) implementation, Hopfield neural networks (HNNs), recurrent neural networks (RNNs), recursive learning, simultaneous perturbation.**

I. Introduction

NOWADAYS, neural networks (NNs) are widely used in many fields. At the same time, back-propagation (BP) has been widely adopted as a successful learning rule to find the appropriate values of the weights for NNs.

Unlike ordinary multilayered feedforward NNs, recurrent neural networks (RNNs) can handle dynamical information processing. It is well known that recurrent-type NNs have complicated properties compared with ordinary multilayered NNs. In spite of the great deal of interest in these RNNs, one of the reasons why RNNs cannot be easily used is the difficulty in setting up the values of the weights in the network for specific purposes. That is, a suitable learning scheme is essential for wider applications of RNNs.

For example, the Hopfield neural network (HNN) is a typical recurrent neural network with symmetrical fully connected weights [1]. HNNs are used to store patterns or to solve combinatorial optimization problems like the traveling salesman problem. For these problems, the weights in the network are typically determined by patterns to be memorized based on Hebbian learning rule [2] or an energy function based on the problem. If our patterns are analog or if we cannot find a proper energy function, it will be impossible to apply these techniques to find the optimal weight values of the network. Moreover, these techniques are difficult to realize as hardware systems.

Let us think about recursive-type learning rules for RNNs. The BP-type learning rule is not suitable for these RNNs. For example, the BP through time (BPTT) for RNNs is a learning rule based on the ordinary BP method [3]. In the BPTT method, an error defined by the difference between the outputs of the network and their desired outputs is propagated through time. This procedure is relatively complicated.

Hardware realization of NNs is an interesting issue [4], [5]. There are many approaches to implement NNs [6], [7]. In addition, realizing a learning scheme is intriguing [6], [8], [9]. However, it seems difficult to realize the learning rule described previously as a large-scale integrated (LSI) system. Thus, inventing a new learning rule for RNNs is crucial.

The simultaneous perturbation optimization method was introduced by Spall [10], [11]. Alespector *et al.* [12] and Cauwenberghs [13] described the same method. Maeda also independently proposed a learning rule using simultaneous perturbation and reported on the feasibility of the learning rule in control problems [14]–[16]. In these works, NNs with the simultaneous perturbation are realized by software. On the other hand, the merit of the learning rule was demonstrated in hardware implementation for many types of NNs [17]–[20]. Analog and pulse density types of NNs via the simultaneous perturbation learning rule were fabricated in these works.

The main advantage of the simultaneous perturbation method is its simplicity. The simultaneous perturbation can estimate the gradient of a function using only two values of the function itself. Therefore, it is relatively easy to implement as a learning rule of NNs compared with other learning rules such as the BP learning rule. At the same time, this learning rule is easily applicable to recurrent types of NNs, since only the final error values are required to estimate the gradient of the error function with respect to the weights.

The FPGA is a very useful device for realizing a specific digital electronic circuit in diverse industrial fields [21]. For example, Hikawa realizes an NN with on-chip BP learning using a field-programmable gate array (FPGA) [22], [23]. Successful

Manuscript received December 29, 2003; revised February 7, 2005. This work was supported by the Ministry of Education, Culture, Sports, Science and Technology of Japan under Grant-in-Aid for Scientific Research 16500142.

Y. Maeda is with the Department of Electrical Engineering and Computer Science, Faculty of Engineering, Kansai University, Osaka 564-8680, Japan (e-mail: maedayut@kansai-u.ac.jp).

M. Wakamura was with the Department of Electrical Engineering and Computer Science, Faculty of Engineering, Kansai University, Osaka 564-8680, Japan. He is now with Semiconductor Company, Matsushita Electric Industrial Co., Ltd., Nagaokakyo, 617-8520, Japan.

Digital Object Identifier 10.1109/TNN.2005.852237

digital implementation of support vector machines and some applications of FPGA-based NNs are reported [24]–[27].

In this paper, we considered an FPGA implementation of an analog HNN with the learning rule via the simultaneous perturbation. We would like to emphasize features of the research as follows.

1) We summarize the learning scheme using the simultaneous perturbation for RNNs.
2) We design an FPGA HNN system as a typical example of RNN systems. Then it is important to take inherent recurrent signal flows into account.
3) Learning capability via the simultaneous perturbation is realized using FPGA as a hardware system.
4) We consider an analog HNN. Then ordinary arithmetic operations are used to realize neural signal processing, instead of Boolean operations such as OR or AND used in a pulse stream type of implementation [19], [20].

The implementation is described in detail. The results of the FPGA HNN system are also shown.

II. Learning Scheme Using Simultaneous Perturbation

When we use an RNN for a specific purpose, we need to determine the proper values of the weights in the RNN. That is, the so-called learning of RNNs is very important. Now we consider a recursive learning scheme for RNNs.

In many applications of RNNs, we know the ideal output or situation for the network. Using this information, we can evaluate how well the network performs. Such an evaluation function gives us a clue for optimizing the weights of the network.

In order to use the BPTT, the error quantity must propagate through time from a stable state to an initial state. This process is relatively complicated. It seems difficult to use such a method directly, because it takes a long time to compute the modifying quantities corresponding to all weights. At the same time, it seems practically difficult to realize the learning mechanism as a hardware system.

On the other hand, the simultaneous perturbation learning scheme is suitable for the learning of RNNs and their hardware implementation. The simultaneous perturbation optimization method requires only values of an evaluation function as mentioned. If we know the evaluation of a stable state, we can obtain the modifying quantities of all weights of the network without complicated error propagation through time.

The simultaneous perturbation learning rule for recurrent neural networks is described as follows:

$$\Delta w_t^i = \frac{J(\boldsymbol{w}_t + c\boldsymbol{s}_t) - J(\boldsymbol{w}_t)}{cs_t^i} \tag{1}$$

$$w_{t+1}^i = \begin{cases} w_{\max}, & \text{if } \left(w_t^i - \alpha\Delta w_t^i\right) > w_{\max} \\ -w_{\max}, & \text{if } \left(w_t^i - \alpha\Delta w_t^i\right) < -w_{\max} \\ w_t^i - \alpha\Delta w_t^i, & \text{otherwise} \end{cases} \tag{2}$$

where $\boldsymbol{w}_t$ and w_t^i denote the weight vector of a network and its ith element at the tth iteration, respectively. α is a positive constant and c is the magnitude of the perturbation. Δw^i represents the ith element of the modifying vector. $w_{\max}$ is the maximum value of the weight.

$\boldsymbol{s}_t$ and s_t^i denote a sign vector and its ith element that is 1 or -1, respectively. The sign of s_t^i is randomly determined. Moreover, the sign of s_t^i is independent of the sign of the jth element s_t^j of the sign vector. That is

$$E\left(s_t^i\right) = 0, \quad E\left(s_{t_1}^i s_{t_2}^j\right) = \delta_{ij}\delta_{t_1 t_2} \tag{3}$$

where E denotes the expectation. δ is Kronecker's delta. $J(\boldsymbol{w})$ denotes an error or an evaluation function, for example, defined by outputs of neurons in a stable state and a pattern to be embedded.

In many cases, the limitation on $w_{\max}$ for the weighting value is not necessary. However, when the behavior of the weight value is erratic, it is useful and efficient.

When we expand the right-hand side of (1) at the point $\boldsymbol{w}_t$, there exist $\boldsymbol{w}_{s1}$ such that

$$\Delta w_t^i = s_t^i \boldsymbol{s}_t^T \frac{\partial J(\boldsymbol{w}_t)}{\partial \boldsymbol{w}} + \frac{cs_t^i}{2} \boldsymbol{s}_t^{\mathrm{T}} \frac{\partial^2 J(\boldsymbol{w}_{\mathrm{S1}})}{\partial \boldsymbol{w}^2} \boldsymbol{s}_t. \tag{4}$$

We take an expectation of the previous quantity. From the conditions given by (3) of the sign vector $\boldsymbol{s}_t$, we have

$$E\left(\Delta w_t^i\right) = \frac{\partial J(\boldsymbol{w}_t)}{\partial w_t^i}. \tag{5}$$

That is, Δw_t^i approximates $\partial J(\boldsymbol{w}_t)/\partial w_t^i$. Since the right-hand side of (1) is an estimated value of the first-differential coefficient of the error function, the learning rule is a type of a stochastic gradient method [15], [16].

The learning rule described here is a simplified version of the one proposed in [15]. In the basic form of the simultaneous perturbation method [10], [15], different distributions can be used for the perturbation. In this paper, the Bernoulli distribution is adopted for the perturbations to simplify and reduce the complexity of the implementation.

An important point is that this learning rule requires only two values of an error function. Therefore, we can apply this learning scheme to not only the so-called feedforward-type of NNs but also to recurrent-type NNs like HNNs. At the same time, unlike the Hebbian learning, this scheme is applicable to analog problems.

We now explain the procedure of this learning scheme in detail.

1) Set the initial weights and the initial state of the network.
2) Run the RNN. The RNN then reaches a stable state.

If we would like to embed a certain pattern in the HNN, then the following procedure is more useful than the procedure outlined previously.

1') Set the initial weights and the desired pattern as the initial state.
2') Run the HNN once.

If the network has the optimal weight values for the desired pattern, the next state has to be the same as the desired pattern. Therefore, one operation of the HNN is sufficient to measure an error.

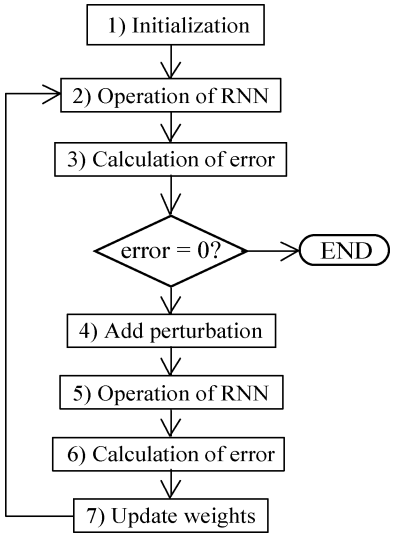

Fig. 1. Flowchart of the scheme for RNNs.

3) We calculate the value of the error which can be defined, for example, as follows:

$$J(\boldsymbol{w}) = \sum_i (o^i - d^i)^2 \tag{6}$$

where o^i and d^i denote a stable state of the RNN and its corresponding ideal output, respectively. This error is the difference between the ideal final pattern and the present final pattern of the network.

In some applications of RNNs, the changes in the state of the network are important. In this case, we can consider the following evaluation:

$$J(\boldsymbol{w}) = \sum_t \sum_i \left(o_t^i - d_t^i\right)^2 \tag{7}$$

where o_t^i and d_t^i denote the state of a neuron and its corresponding desired state at the tth iteration, respectively. If the evaluation function is small, this means that the trajectory of the network is close to an ideal trajectory.

4) Add the perturbation to all weights in the network.

The perturbations $+c$ or $-c$, which are determined randomly, are added to all weights simultaneously.

5) We have the RNN to operate again the same as in procedure 2) or 2').

6) Obtain the error as in procedure 3). (Obtain $J(\boldsymbol{w} + c\boldsymbol{s})$.)

7) Using (1) and (2), update the weights of the network and go to procedure 2).

The flowchart of the procedure is depicted in Fig. 1. Note that only two error values for the RNN are used to update all weights in the network so the procedure is very simple and easy to implement.

We can summarize some of the advantages of the simultaneous perturbation learning rule for RNNs as follows:

1) applicability to analog problems;
2) applicability to oscillatory solutions;
3) applicability to trajectory learning;
4) energy function unnecessary;
5) error back-propagation through time unnecessary;
6) simplicity.

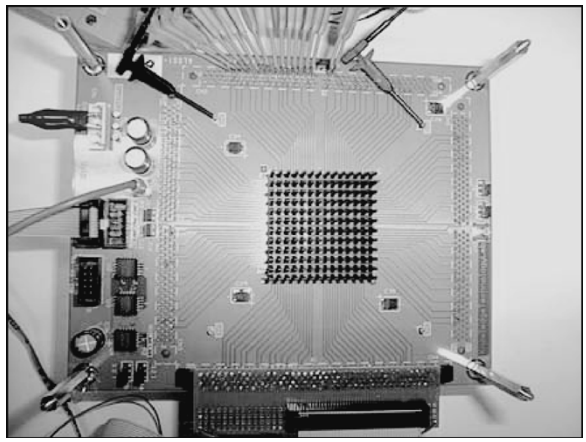

Fig. 2. FPGA board.

III. FPGA Implementation of HNN

As a typical example of RNNs, we handle an analog HNN to be implemented by FPGA.

There are some ways of realizing hardware HNNs with learning capability. For example, Lehmann *et al.* reported a VLSI system for a HNN with on-chip learning [28]. Also in our research, the FPGA implementation based on digital circuit design technology is used to realize the HNN. We consider an FPGA implementation of an analog HNN with a recursive learning capability using the simultaneous perturbation method. The HNN fabricated here contains 32 neurons. The number of neurons is prescribed by the FPGA device.

We adopted VHSIC Hardware Description Language (VHDL) in the circuit design for FPGA. The design result by VHDL is configured on FPGA through Leonardo and Quartus. FPGA Altera, EP20K400BC652-2V with 1 052 000 gates (16 640 logic elements) and 212 992 bits RAM is used (see Fig. 2).

When we consider the hardware implementation of an NN, it is beneficial to prepare the plural neuron elements for parallel processing. This requires a large number of gates in the FPGA. On the other hand, the number of gates is limited. It is crucial to economize on the number of gates used in FPGAs. We have to take this tradeoff into account. Therefore, if we design an HNN with so many neurons, it will be necessary to use the series processing of neurons from a practical point of view. Actually, it was impossible to realize totally parallel processing system of the HNN, because scale of the HNN with 32 neurons, that is, the number of gates required in the design is larger than the gate number in the FPGA device used here.

A. Technical Factors

When we design digital NN circuits, data handling and high-speed calculations are very important. Many approaches are possible to achieve these requirements. In this section, we describe in detail the technical factors of our HNN design.

1) Representation of Numerical Values and Operations: Our purpose is to construct the HNN efficiently,

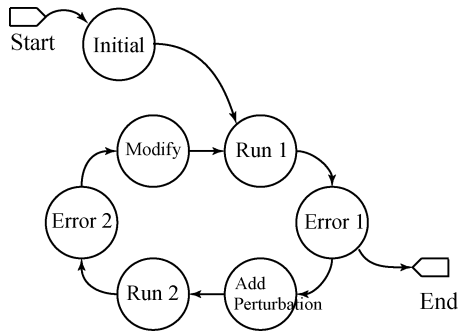

Fig. 3. State flow of the overall HNN system.

not to invent and evaluate new arithmetic representations, operations, or precision. In this implementation, we adopt a single precision floating-point representation of the numerical values. Thirty-two bits with one sign bit, eight exponent bits and 23 mantissa bits of IEEE 754 standard are used to represent a numerical value. This representation was sufficient to describe the HNN operation.

Arithmetic operations used here are standard procedures. We briefly describes about multiplication and addition. The adder operation consists of 1) comparison of the values, 2) alignment of the mantissas by equalizing their exponents, 3) addition of the mantissas, 4) normalization, and 5) truncation. Further, the multiplication procedure consists of 1) multiplication of the signs, 2) addition of the exponent, 3) multiplication of the mantissas, 4) normalization, and 5) truncation.

Henceforth, these operations will be referred as adder or multiplier.

2) Control Unit: The control unit plays a central role in the whole of the system for series processing. The unit outputs control signals to all other units.

The operation of the unit is described in Fig. 3 as a state flow graph. We design the unit based on the state flow graph.

At the initial state, we set up the initial values. Run 1 denotes an operation of the HNN without perturbation. Error 1 calculates a value $J(\boldsymbol{w})$ of the evaluation function without perturbation. Then, if $J(\boldsymbol{w})$ is sufficiently small, the learning process will terminate. Otherwise, the system gives the perturbation for all weights in Add perturbation state. A 64-bit shift register described in the following generates random numbers for the perturbation in the Add perturbation state. Run 2 is an operation of the HNN with the perturbation. Error 2 calculates another value of the evaluation function with the perturbation. In the Modify state in Fig. 3, we update all weights using these two values of the evaluation function, which are obtained in the states Error 1 and Error 2. This series of operations corresponds to one learning iteration.

At the Error state, the difference between the outputs of the HNN and their teaching signal is calculated. The difference denotes a value of the evaluation function.

3) Neuron Unit: The architecture of our neuron unit is depicted in Fig. 4. The neuron unit is realized using several types of calculations: adder, multiplier, register, and so on.

The output values of all neurons and corresponding weight values are stored in the memory unit.

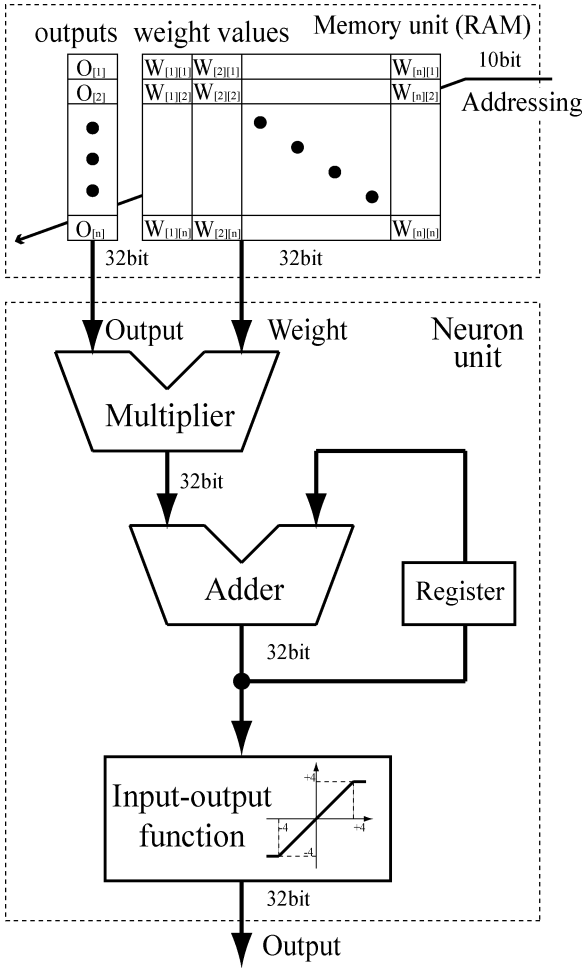

Fig. 4. Configuration of the neuron unit.

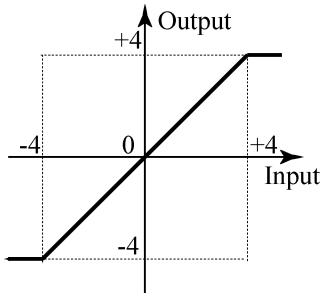

Fig. 5. Input–output characteristics of neurons.

The output of a neuron and the corresponding weight value in the memory are multiplied. First, this result is stored in a register. We repeat the multiplication for the other outputs of the other neurons with the corresponding weights. We add the result and the value in the register. Repeating this procedure, the weighted inputs for the neuron are summed up. The weighted sum becomes the output of the neuron through an input–output function.

The input–output characteristics of these neurons are depicted in Fig. 5. In order to simplify the circuit design, a linear function with a saturation of ± 4 is used here. That is, neurons in this network have analog state values from -4 to $+4$.

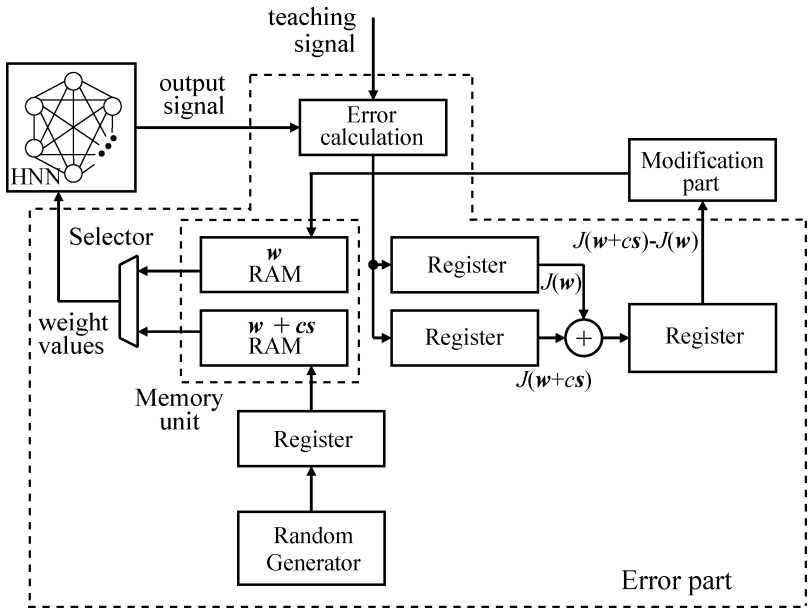

Fig. 6. Configuration of the error part.

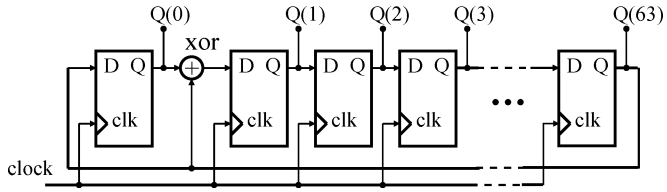

Fig. 7. Configuration of the linear feedback shift register.

As mentioned before, this single neuron unit is used serially to realize all neurons in the HNN.

4) Learning Unit: The leaning unit consists of two parts: the error part and the modification part. This single learning unit generates the modifying quantities and updates all weights.

Fig. 6 shows the error part. This part generates the difference in the error values with and without the perturbation, that is, $J(\boldsymbol{w}+c\boldsymbol{s})-J(\boldsymbol{w})$. This quantity is commonly used to modify all weights.

A random number generator is used in this part. A linear feedback shift register with 64 flip-flops is adopted as shown in Fig. 7.

Fig. 8 shows a modification part in the learning unit. In the memory unit, all weight values and the corresponding sign information of one bit are stored. Based on the sign and the modifying quantity generated in the error part, the part modifies the weight values.

First, common quantity $J(\boldsymbol{w}+c\boldsymbol{s})-J(\boldsymbol{w})$ is given from the error part. Then the sign of this difference is changed according to the sign information for a weight in the memory.

We set $\alpha/c=1/2^m$ (m is an adequate integer.). Therefore, shifting the bits of the exponent part of the result means a multiplication of the value α/c with the previous result. This result denotes a modifying quantity for a weight. Using an adder, we can update the weight.

We have to repeat this procedure for all weights in the network. However, the differences in the error values $(J(\boldsymbol{w}+c\boldsymbol{s})-J(\boldsymbol{w}))$ and α/c are the same for all weights. The point is that only different weights have different sign information. Thus, the complete learning procedure is very simple and compact.

5) Memory: The memory stores the weight values of the neurons, the output and weight values of the neurons, the sign information, and the teaching signals. These data are read or restored according to the command of the control unit.

Since we have 32 neurons in the HNN, there exist $1024(=32\times 32)$ weight values. Therefore, we need 10 bits addressing $(2^{10}=1024)$. HNNs have a weights matrix whose symmetrical elements are equal. Thus, we take this condition on weights into account.

B. HNN System

The total operation of the system is divided into two functions: the operation of the HNN and a learning function.

1) Operation of HNN: Fig. 9 shows configuration of the HNN from an arithmetic operation point of view. Moreover, control signals are described in detail. The state machine in Fig. 9 controls the two RAMs: the counter and the selector. The counter decides which weight values and output in the RAMs should be selected through the lookup table.

The operation of the HNN is realized by the neuron unit and the memory. The neuron unit carries out sum-of-products operations in the neurons at the Run mode. Then the neuron unit performs the sum-of-products operation serially. That is, a single actual neuron realizes for 32 imaginary neurons. Fig. 10 shows the flow of operations. From time t to $t+1$, the arithmetic logic unit (ALU) works as the tth neuron. From $t+1$ to $t+2$, the ALU works as the $t+1$th neuron, and so on. The weight values, inputs, and outputs are on memories. These data are read from or written to the memories.

2) Learning Function: The function is realized by the learning unit.

As mentioned before, the single learning unit generates all modifying quantities for all weights in the network. The basic quantity $\alpha(J(\boldsymbol{w}+c\boldsymbol{s})-J(\boldsymbol{w}))/c$ for the modification of weights

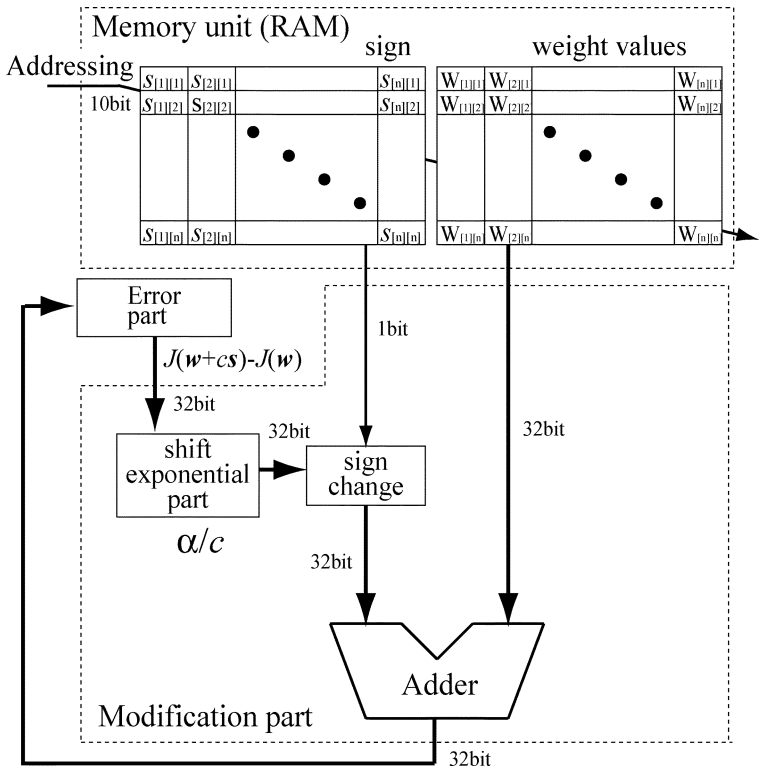

Fig. 8. Configuration of the modification part.

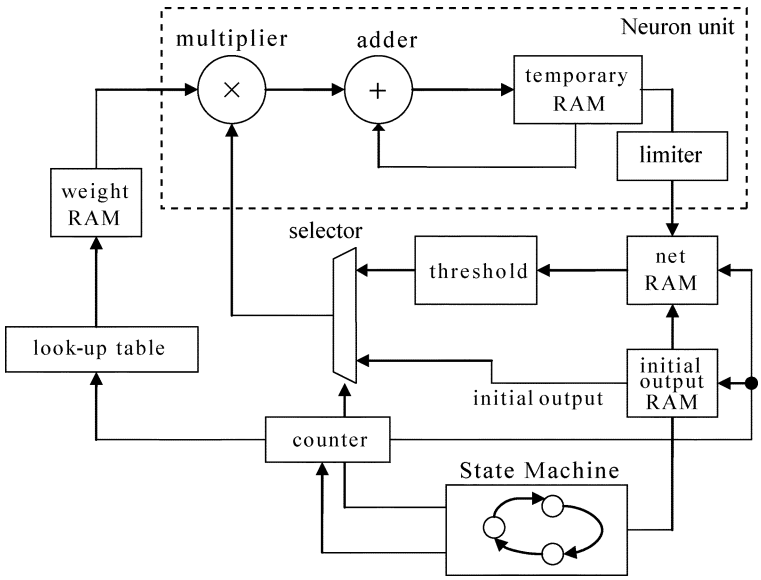

Fig. 9. Configuration of the HNN from arithmetic operation point of view.

is common. The only difference is the sign of the quantity. By scanning all of the weights and their sign information shown in Fig. 8, the unit updates the weight value.

If we adopt the BPTT or other gradient types of learning rules, the learning unit and the learning operation become extremely complicated. In contrast, using the simultaneous perturbation method, we could simplify the mechanism, since only two stable states are used to calculate the error and to update the weights.

For this design, 9225 logic elements (about 55%) and 45 056 bits of RAM(about 21%) are used. Moreover, 38 out of 502 pins are assigned to observe states of the HNN.

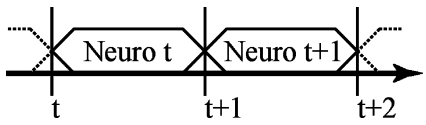

Fig. 10. Time-division-multiplexing processing.

C. Result

1) Analog Pattern: We show a result for the learning of analog values. The desired outputs of each neuron in the HNN are analog quantities. The teaching pattern to be memorized for

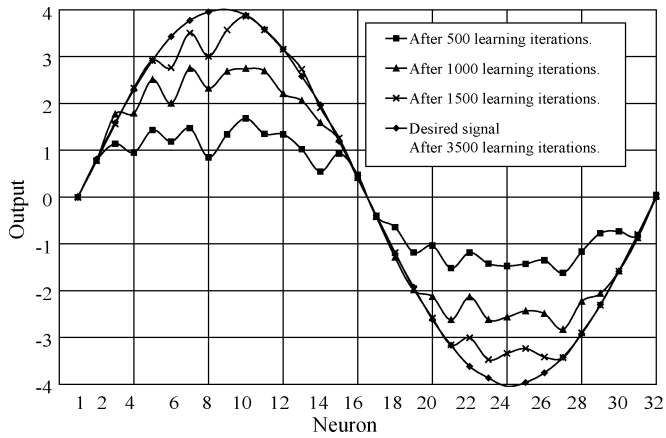

Fig. 11. Result for analog problem.

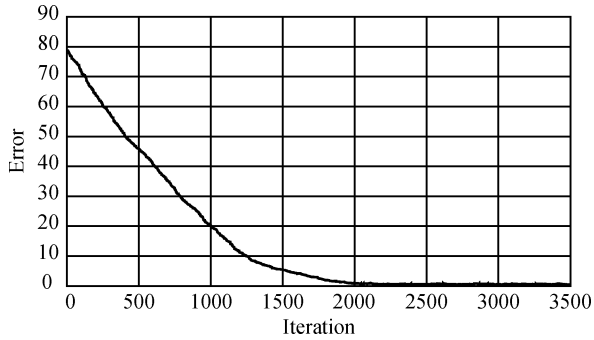

Fig. 12. Change in error for the analog problem.

all neurons is the sinusoidal analog value shown in Fig. 11. The waveform itself does not have any specific meanings.

The horizontal axis of the figure denotes the neuron number. The vertical axis shows an output of the neuron in a stable state. For example, the output of the first neuron is depicted in the leftmost side of the chart. The 32nd neuron is on the rightmost side. Each neuron has its desired analog value, which is shown as the desired signals in Fig. 11. That is, the desired output of the first neuron is zero, that of the second neuron is about 0.8, and so on. Connecting these ideal values for all neurons shows a sinusoidal waveform.

We can see the learning process in Fig. 11. The more the learning process proceeds, the more the actual outputs resemble the teaching pattern. After about 3500 iterations of learning, the outputs of the neurons are exactly the same as the desired ones.

In this experiment, the coefficient $\alpha/c = 1/2^5$ and $c = 0.000\,15$.

Fig. 12 shows the change in error with increasing number of iteration. The error decreases as the learning proceeds. After about 2500 learning iterations, the error is sufficiently decreased. With 10 MHz clock frequency, the learning has been accomplished within about 2 s.

2) Oscillatory Solution: The analog HNN can learn a kind of oscillatory state. We handle the problem of an oscillatory pattern in which two analog patterns occur alternately in a stable state of the network. That is, in a proper stable state, two analog patterns appear in turn.

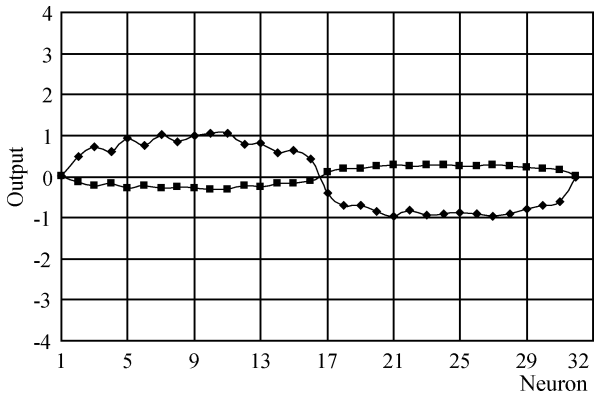

Fig. 13. Result for an oscillatory pattern after 1000 learning iterations.

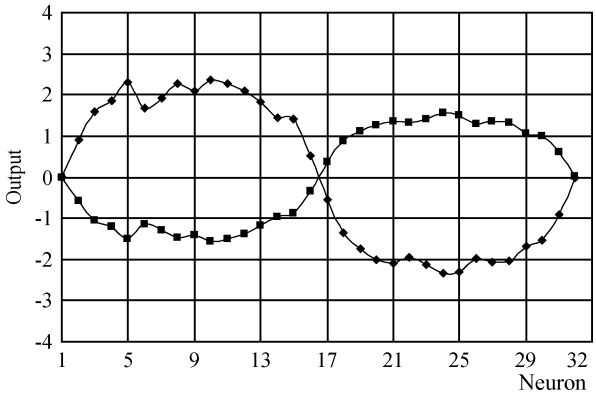

Fig. 14. Result for an oscillatory pattern after 2000 learning iterations.

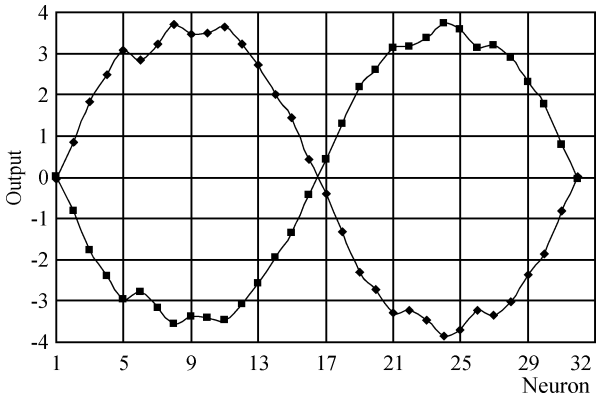

Fig. 15. Result for an oscillatory pattern after 3000 learning iterations.

In this experiment, we handle two sinusoidal waveforms with 180 different phases as the target patterns, which are exactly the same as the patterns shown in Fig. 16.

Figs. 13 –16 exhibit outputs of the neurons of the network in a stable state after a certain learning period. In the very early stage, as in Fig. 13, the outputs are relatively small. After 2000 learning iterations, the outputs respond to the target pattern. In Fig. 15, the output pattern is rather close to the desired pattern. After 5000 learning iterations, there is a perfect agreement with the wanted pattern as in Fig. 16. As shown in Fig. 17, the error decreases to zero after around 3800 learning iterations. It took about 2 s with 10 MHz clock for the learning.

In this experiment, the coefficient α/c is $1/2^6$, $c = 0.0001$.

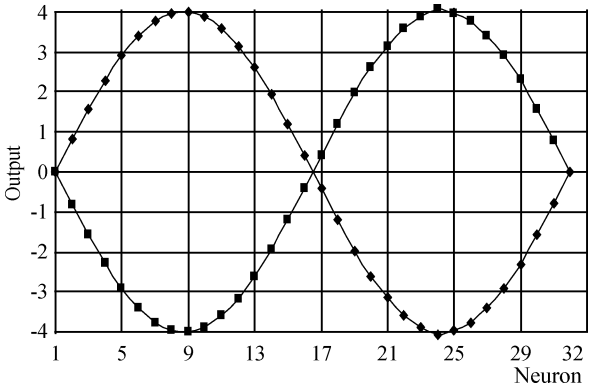

Fig. 16. Result for an oscillatory pattern after 5000 learning iterations.

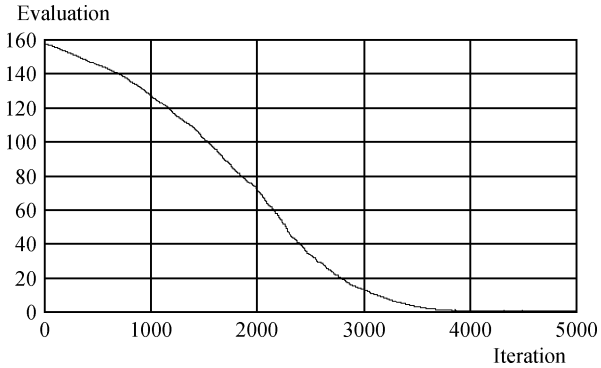

Fig. 17. Change in error for the oscillatory analog problem.

We would like to emphasize that these two examples had all analog targets. In these cases, Hebbian learning, which is widely used for HNNs, would be unsuitable since the outputs considered in its scheme are binary.

Moreover, the latter example of learning of oscillatory state is impossible to realize using Hebbian learning. However, the simultaneous perturbation learning rule makes it possible.

IV. Conclusion

This paper proposed a recursive learning scheme for recurrent NNs. The learning scheme is based on the simultaneous perturbation method.

As an example, we implement the HNN with this learning scheme by FPGA. Learning examples for analog targets and an oscillatory pattern are shown.

This scheme requires only two values of an error function. Therefore, it was relatively easy to implement, especially as a hardware system.

Moreover, it is possible to utilize this scheme for analog problems or the learning of oscillatory patterns in contrast to ordinary Hebbian learning.

Investigation on capability of the simultaneous perturbation learning rule for such problems is our future work.

References

[1] J. J. Hopfield, "Neurons with graded response have collective computational properties like those of two-state neuron," in *Proc. Nat. Acad. Sci.*, vol. 81, 1984, pp. 3088–3092.

[2] B. Kosko, *Neural Networks and Fuzzy Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1992.

[3] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing*. Cambridge, MA: MIT Press, 1986, vol. 1.

[4] B. J. Sheu and J. Choi, *Neural Information Processing and VLSI*. Norwell, MA: Kluwer, 1995.

[5] S. M. Fakhraie, "Scalable closed-boundary analog neural networks," *IEEE Trans. Neural Netw.*, vol. 15, pp. 492–504, Mar. 2004.

[6] L. M. Reyneri, "Implementation issues of neuro-fuzzy hardware: going toward HW/SW codesign," *IEEE Trans. Neural Netw.*, vol. 14, pp. 176–194, Jan. 2003.

[7] R. Genov and G. Cauwenberghs, "Kerneltron: support vector "machine" in silicon," *IEEE Trans. Neural Netw.*, vol. 14, pp. 1426–1434, Sep. 2003.

[8] T. Morie and Y. Amemiya, "An all-analog expandable neural network LSI with on-chip backpropagation learning," *IEEE J. Solid-State Circuits*, vol. 29, pp. 1086–1093, Sep. 1994.

[9] T. Morie, O. Fujita, and K. Uchimura, "Self-learning analog neural network LSI with high-resolution nonvolatile analog memory and a partially-serial weight-update architecture," *IEICE Trans. Electron.*, vol. E80-C, no. 7, pp. 990–995, 1997.

[10] J. C. Spall, "A stochastic approximation technique for generating maximum likelihood parameter estimates," in *Proc. Amer. Control Conf.*, 1987, pp. 1161–1167.

[11] ——, "Multivariable stochastic approximation using a simultaneous perturbation gradient approximation," *IEEE Trans. Autom. Control*, vol. 37, pp. 332–341, Mar. 1992.

[12] J. Alespector, R. Meir, B. Yuhas, A. Jayakumar, and D. Lippe, "A parallel gradient descent method for learning in analog VLSI neural networks," in *Advances in Neural Information Processing Systems 5*, S. J. Hanson, J. D. Cowan, and C. Lee, Eds. San Mateo, CA: Morgan Kaufmann, 1993, pp. 836–844.

[13] G. Cauwenberghs, "A fast stochastic error-descent algorithm for supervised learning and optimization," in *Advances in Neural Information Processing Systems 5*, S. J. Hanson, J. D. Cowan, and C. Lee, Eds. San Mateo, CA: Morgan Kaufmann, 1993, pp. 244–251.

[14] Y. Maeda and Y. Kanata, "Learning rules for recurrent neural networks using perturbation and their application to neuro-control," *Trans. Inst. Elect. Eng. Jpn.*, vol. 113-C, no. 6, pp. 402–408, 1993.

[15] ——, "A learning rule of neural networks for neuro-controller," in *Proc. World Congr. Neural Networks*, vol. 2, 1995, pp. II-402–II-405.

[16] Y. Maeda and R. J. P. de Figueiredo, "Learning rules for neuro-controller via simultaneous perturbation," *IEEE Trans. Neural Netw.*, vol. 8, pp. 1119–1130, Sep. 1997.

[17] Y. Maeda, H. Hirano, and Y. Kanata, "A learning rule of neural networks via simultaneous perturbation and its hardware implementation," *Neural Netw.*, vol. 8, no. 2, pp. 251–259, 1995.

[18] G. Cauwenberghs, "An analog VLSI recurrent neural network learning a continuous-time trajectory," *IEEE Trans. Neural Netw.*, vol. 7, pp. 346–361, Mar. 1996.

[19] Y. Maeda, A. Nakazawa, and Y. Kanata, "Hardware implementation of a pulse density neural network using simultaneous perturbation learning rule," *Analog Integr. Circuits Signal Process.*, vol. 18, no. 2, pp. 1–10, 1999.

[20] Y. Maeda and T. Tada, "FPGA implementation of a pulse density neural network with learning ability using simultaneous perturbation," *IEEE Trans. Neural Netw.*, vol. 14, pp. 688–695, May 2003.

[21] S. Hauck, "The role of FPGA's in reprogrammable systems," *Proc. IEEE*, vol. 86, no. 4, pp. 615–638, Apr. 1998.

[22] H. Hikawa, "Frequency-based multilayer neural networks with on-chip learning and enhanced neuron characteristics," *IEEE Trans. Neural Netw.*, vol. 10, pp. 545–553, May 1999.

[23] ——, "A new digital pulse-mode neuron with adjustable activation function," *IEEE Trans. Neural Netw.*, vol. 14, pp. 236–242, Jan. 2003.

[24] D. Anguita, A. Boni, and S. Ridella, "A digital architecture for support vector machines: Theory, algorithm and FPGA implementation," *IEEE Trans. Neural Netw.*, vol. 14, pp. 993–1009, Sep. 2003.

[25] F. Yang and M. Paindavoine, "Implementation of an RBF neural network on embedded systems: Real-time face tracking and identity verification," *IEEE Trans. Neural Netw.*, vol. 14, pp. 1162–1175, Sep. 2003.

[26] C. T. Yen, W. Weng, and Y. T. Lin, "FPGA realization of a neural-network-based nonlinear channel equalizer," *IEEE Trans. Ind. Electron.*, vol. 51, no. 2, pp. 472–479, 2004.

[27] H. S. Ng and K. P. Lam, "Analog and digital FPGA implementation of BRAIN for optimization problem," *IEEE Trans. Neural Netw.*, vol. 14, pp. 1413–1425, Sep. 2003.

[28] C. Lehmann, M. Viredaz, and F. Blayo, "A generic systolic array building block for neural networks with on-chip learning," *IEEE Trans. Neural Netw.*, vol. 4, pp. 400–407, May 1993.

Yutaka Maeda (M'92) received the B.E., M.E., and Ph.D. degrees in electronic engineering from Osaka Prefecture University, Osaka, Japan, in 1979, 1981, and 1990, respectively.

He joined the Faculty on Engineering, Kansai University, Osaka, in 1987, where he is currently a Professor and a Vice Dean. In 1993, he was a Visiting Researcher in the Automatic Control Center, Northeastern University, China. From 1995 to 1996, he was a Visiting Researcher in the Electrical and Computer Engineering Department, University of California, Irvine. His research interests are in the areas of control theory, artificial neural networks, and biomedical engineering.

Masatoshi Wakamura received the M.E. degree from Kansai University, Osaka, Japan, in 2004.

He is with Semiconductor Company, Matsushita Electric Industrial Co., Ltd., Nagaokakyo, Japan. He was engaged in research on FPGA implementation of neural networks, while in Kansai University.