Rapid Learning Method for Multilayered Neural Networks Using Two-Dimensional Conjugate Gradient Search

TOSHINOBU YOSHIDA*

Back-propagation learning in multilayered neural networks is based on the principle of steepest descent. This method calculates the gradient of the error function in the reverse mode of automatic differentiation. The present paper first summarizes automatic differentiation and two-dimensional conjugate gradient search based on automatic differentiation. A new learning method using two-dimensional conjugate gradient search is then proposed for neural networks. The proposed method automatically controls the learning rate and the momentum factor of the back-propagation. It requires computation of the quadratic forms of the Hessian of the error function. The computation time and the memory storage are proportional to the square of the size of the neural network if all the components of the Hessian are computed. However, in the forward mode of automatic differentiation, the quadratic forms are computed at a cost proportional only to the size of the neural network. Numerical experiments show that the number of iterations is much smaller than for back-propagation, while the time taken for one iteration is about three times that in back-propagation.

1. Introduction

In 1964, Wengert [1] presented a procedure for automatic calculation of total/partial derivatives of arbitrary algebraic functions. Baur et al. [3], Kim et al. [4], Iri [5], and Sawyer [6] proposed the reverse mode of automatic differentiation, as opposed to Wengert's forward mode. This mode is called "fast automatic differentiation," because the reverse mode computes all components of the gradient of a scalar function at a cost proportional to the cost of computing the function itself. Several automatic differentiation tools are now available [7, 8, 10].

The back-propagation proposed by Rumelhart et al. [11] is a gradient calculation technique in the reverse mode of automatic differentiation. They presented a learning method for feed-forward neural networks based on back-propagation. Their method updates the weights of the connection by adding the gradient multiplied by $-\eta$ (η is called the "learning rate"). Since this method is essentially based on the principle of steepest descent, the convergence speed is slow. They also proposed an acceleration technique: the weights are changed by adding the sum of the gradient multiplied by $-\eta$ and the previous correction multiplied by the "momentum factor" α. Since the optimal η and α vary in the course of the iterations, it is difficult to fix these parameters beforehand. Vogl et al. [12] proposed a different acceleration method, controlling η and α dynamically.

Yoshida [13] proposed an optimization method using two-dimensional conjugate gradient search based on automatic differentiation. This is an iterative method for finding the minimum of a scalar function. In each iteration, the object function is approximated by a quadratic function defined on the plane spanned by the gradient direction and the previous search direction, and the search moves to the minimum of the estimated quadratic function. The method requires computation of the quadratic forms of the Hessian of the object function. The computation time and the memory storage are generally proportional to the square of the size of the object function, if all the components of the Hessian are computed. However, the method can compute quadratic forms by automatic differentiation in a time and with a storage proportional to the size of the object function.

Using this method, we propose in this paper a rapid learning method for multilayered feed-forward neural networks. The method automatically controls the parameters η and α. Sections 2 and 3 review automatic differentiation and two-dimensional conjugate gradient search, respectively. Section 4 presents notation and definitions related to multilayered neural networks, and describes the procedures for computing the gradient, inner products, and quadratic forms in detail. In section

This is a translation of the paper that appeared originally in Japanese in Transactions of IPSJ, Vol. 31, No. 7 (1990), pp. 988-995.

*Department of Computer Science, Gunma University, Kiryu, Gunma 376, Japan.

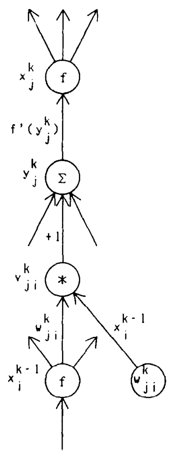


Fig. 1 Computational graph of Eq. (1).

5, we compare the proposed method with basic back-propagation, back-propagation with inertia, and Vogl's method.

2. Automatic Differentiation

The details of automatic differentiation are given in references 1 to 10. Here, only the basic ideas are reviewed.

Any computational sequence of functions can be represented as a directed acyclic graph. Let us call such a graph a "computational graph." For example, we represent the following functions in the computational graph as shown in Fig. 1:

$$x_j^k = f(\sum_i w_{ji}^k x_i^{k-1}). \tag{1}$$

We put a memory on every edge and every node. The memory on the edge from node u to node v has the value $\partial v/\partial u$, which we call the "elementary partial derivative" (see Fig. 1). Each node v is given a memory dv. We use the memories on nodes in both modes of automatic differentiation as follows:

Forward mode Let $e_1, \cdots, e_k$ be the edges entering node v. Each edge e_i exiting from node v_i has an elementary partial derivative de_i. The value dv at node v is defined from the values $dv_1, \cdots, dv_k$ at nodes $v_1, \cdots, v_k$ by

$$dv = \sum_{i=1}^{k} de_i dv_i. \tag{2}$$

This operation is performed in the computational graph from bottom to top, that is, in the order of function evaluation.[1]

Reverse mode Let $e_1, \cdots, e_k$ be the edges exiting from node v. Each edge e_i entering node v_i has an elementary partial derivative de_i. The value dv at node v is defined from the values $dv_1, \cdots, dv_k$ at nodes $v_1, \cdots, v_k$ by

$$dv = \sum_{j=1}^{k} de_j dv_j. \tag{3}$$

This operation is performed in the computational graph from top to bottom, that is, in the reverse order of function evaluation.[1]

Let $x_1, \cdots, x_n$ and $f_1, \cdots, f_m$, respectively, be the nodes corresponding to the input vector $\boldsymbol{x}$ and the output vector $\boldsymbol{f}$. Let us assign dx_i at node x_i. If the forward mode operations are applied, the values $df_1, \cdots, df_m$ at nodes $f_1, \cdots, f_m$ are

$$df_j = \sum_{i=1}^{n} (\partial f_j / \partial x_i) dx_i, \quad j = 1, \cdots, m. \tag{4}$$

In other words, the product of the Jacobi matrix $\nabla \boldsymbol{f}(\boldsymbol{x})$ and vector $\boldsymbol{dx} = (dx_1, \cdots, dx_n)'$ is computed. We note that the Jacobi matrix itself is not computed. For a scalar function $f(\boldsymbol{x})$, this bottom-up method computes the inner product $\nabla f(\boldsymbol{x}) \cdot \boldsymbol{y}$, if vector $\boldsymbol{y}$ is assigned to nodes $\boldsymbol{x}$. Since this computational sequence of $\nabla f(\boldsymbol{x}) \cdot \boldsymbol{y}$ can be represented in another computational graph, application of the bottom-up method computes the value of the quadratic form $\boldsymbol{z}' \nabla^2 f(\boldsymbol{x}) \boldsymbol{y}$, if vector $\boldsymbol{z}$ is assigned to nodes $\boldsymbol{x}$ of this computational graph.

Let us assign vector $\boldsymbol{y}$ to nodes $\boldsymbol{f}$, and apply the reverse-mode operations. Then, the values obtained at nodes $x_1, \cdots, x_n$ are

$$dx_i = \sum_{j=1}^{m} y_j (\partial f_j / \partial x_i), \quad i = 1, \cdots, n. \tag{5}$$

For a scalar function $f(\boldsymbol{x})$, this top-down method computes the gradient $\nabla f(\boldsymbol{x})$ by assigning $y = 1$ at node f. The computational complexity is at most a constant multiple of the complexity of computing the function itself, and does not depend on the number of variables. Back-propagation is simply a gradient evaluation technique based on this top-down method.

In summary, the gradient $\nabla f(\boldsymbol{x})$, inner product $\nabla f(\boldsymbol{x}) \cdot \boldsymbol{y}$, and quadratic form $\boldsymbol{z}' \nabla^2 f(\boldsymbol{x}) \boldsymbol{y}$ can be computed by automatic differentiation in a time and with a storage proportional to the size of the function.

3. Conjugate Gradient Method

Miele et al. [14] proposed a "memory gradient method" for finding the minimum of a scalar function $f(\boldsymbol{x})$. Their method approximates the function by a quadratic function of two variables η and α on the plane spanned by the steepest descent $\boldsymbol{g} = -\nabla f(\boldsymbol{x})$ and the previous correction $\boldsymbol{p}$ in the neighborhood of $\boldsymbol{x}$ as follows:

[1]We call this mode of differentiation "bottom-up."

[1]We call this mode of differentiation "top-down."

$$f(x+\eta g+\alpha p)\approx f(x)+b\cdot\xi+\frac{1}{2}\xi' A\xi, \tag{6}$$

where b, A, and ξ are defined by

$$\begin{aligned} dg&=\nabla f(x)\cdot g,\\ dp&=\nabla f(x)\cdot p,\\ ghg&=g'\nabla^2 f(x)g,\\ ghp&=g'\nabla^2 f(x)p,\\ php&=p'\nabla^2 f(x)p, \end{aligned} \tag{7}$$

$$b=\begin{pmatrix} dg\\ dp\end{pmatrix}, \tag{8}$$

$$A=\begin{pmatrix} ghg & ghp\\ ghp & php\end{pmatrix}, \tag{9}$$

$$\xi=\begin{pmatrix}\eta\\ \alpha\end{pmatrix}. \tag{10}$$

The minimizer $\xi=(\eta\ \alpha)'$ of the left-hand side of Eq. (6) is searched for in the two-dimensional plane, using the minimizer of the quadratic function on the right-hand side.

Yoshida [13] showed that the steepest descent g, the inner products dg and dp, and the quadratic forms ghg, ghp, and php can be computed efficiently by automatic differentiation. He also presented another two-dimensional search technique, which inexactly searches for the minimizer ξ as follows:

1. If A is positive definite, the quadratic function on the right-hand side of Eq. (6) is minimized by $\xi=-A^{-1}b$.
2. Else if A is negative definite, the quadratic function has a maximum, and $\xi=-A^{-1}b$ gives the maximum point. Hence, if ξ is oriented in the opposite direction $A^{-1}b$, the quadratic function can be reduced.
3. Else if A is non-singular, the quadratic function has a saddle point, that is, A has a positive eigenvalue and a negative eigenvalue. The function has a minimum along the line corresponding to the positive eigenvalue, and a maximum along the line corresponding to the negative eigenvalue. Vector $-A^{-1}b$ is the composition of the vector to the minimum point and the vector to the maximum point. Hence, the value of the quadratic function can be reduced by setting vector ξ as the composition of the vector to the minimum point and the opposite vector to the maximum point. Vector ξ is given by the following equations:

$$c=\sqrt{(ghg-php)^2+4ghp^2} \tag{11}$$

$$\xi=-\frac{1}{c}\begin{pmatrix} ghg-php & 2ghp\\ 2ghp & php-ghg\end{pmatrix}A^{-1}b, \tag{12}$$

4. Otherwise, α is set at 0, and the minimum is searched for along the steepest descent.

The following is an optimization algorithm based on these ideas.

Algorithm CG (Two-Dimensional Conjugate Gradient Search)

```
CG1 Initialize $x\in R^n, p:=o, f_{old}:=$sufficiently large number, and $r:=N$.
CG2 Compute $f:=f(x)$ and $g:=-\nabla f(x)$.
CG3 If $\|g\|<\varepsilon$, stop.
CG4 If $f<f_{old}$, go to CG5.
      Otherwise, set $p:=p/2$, $x:=x-p$, $r:=N$ and return to CG2.
CG5 Compute $b$ and $A$.
CG6 If det $A=0$ or $r\geq N$, go to CG7.
    If $A$ is positive definite, set $\xi:=-A^{-1}b$, $r:=r+1$ and go to CG8;
    else if $A$ is negative definite, set $\xi:=A^{-1}b$ and go to CG8;
    otherwise, set $\xi$ by using Eqs. (11) and (12), and go to CG8.
CG7 Set $r:=0$, $\alpha:=0$.
    If $ghg>0$, set $\eta:=-dg/ghg$;
    else if $ghg<0$, set $\eta:=dg/ghg$;
    otherwise, set $\eta:=1$.
CG8 Set $p:=\eta g+\alpha p$, $x:=x+p$, and return to CG2.
```

Here ε is the stopping parameter and N is the restart parameter. That is, α is set at 0 every N iterations.

Let the object function f be a quadratic function of an n-dimensional vector x defined by

$$f(x)=c+b\cdot x+\frac{1}{2}x'Hx, \tag{13}$$

where H is a positive definite matrix of size $n\times n$, b is an n-dimensional vector, and c is a scalar. For this object function, it can be proved that the above algorithm generates search steps that are mutually conjugate with respect to H. These are exactly the vectors generated by the conjugate gradient method by Hestenes-Stiefel-Daniel's rule [13]. In other words, if the object function is a quadratic function, Algorithm CG is the usual conjugate gradient method and finds a local minimum in at most n iterations.

4. Multilayered Neural Networks

The neural network model considered here is a multilayered feed-forward analog network with one input layer (layer 0), $n-1$ hidden layers (from layer 1 to layer $n-1$), and one output layer (layer n). Layer k has m_k cells. Let us denote the output of the jth cell in layer k by x_j^k, and the weight of the connection from the ith cell of layer $k-1$ to the jth cell of layer k by w_{ji}^k. The value w_{j0}^k is regarded as the threshold value, and x_0^{k-1} is always 1. The output function f is assumed to be a twice continuously differentiable increasing function. The output of the jth cell in layer k is given by

$$y_j^k=\sum_{i=0}^{m_{k-1}} w_{ji}^k x_i^{k-1}, \tag{14}$$

$$x_j^k=f(y_j^k). \tag{15}$$

The training data is a collection of pairs (x_a, c_a) of input and desired output, where

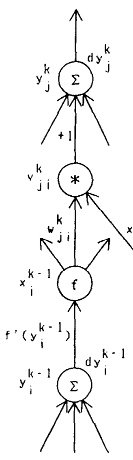


Fig. 2 Computational graph for the equations in Step TD5 of Algorithm TD.

$$\begin{aligned} \boldsymbol{x}_a &= (x_{a1}, x_{a2}, \cdots, x_{a,m_0}), \\ \boldsymbol{c}_a &= (c_{a1}, c_{a2}, \cdots, c_{a,m_n}), \\ a &= 1, 2, \cdots, N. \end{aligned} \tag{16}$$

We define the error function $R(\boldsymbol{w})$ by

$$R(\boldsymbol{w}) = \frac{1}{2} \sum_{a=1}^{N} \sum_{j=1}^{m_n} (x_{aj}^n - c_{aj})^2, \tag{17}$$

where $\boldsymbol{w}$ is a vector consisting of all the weights w_{ji}^k that include the thresholds w_{j0}^k, and x_{aj}^n is the output for the input $\boldsymbol{x}_a$.

The purpose of the learning in the neural network model is to realize the input-output relation specified by the training data. This is done by searching for the weights $\boldsymbol{w}$ that minimize the error function $R(\boldsymbol{w})$. We adopt Algorithm CG for the search. We present concrete algorithms for computing the steepest gradient, inner products, and quadratic forms required in Algorithm CG.

The details of the algorithm for computing the steepest descent $\boldsymbol{g} = -\nabla R(\boldsymbol{w})$ by the top-down method, that is, by back-propagation, are as follows:

Algorihm TD (Computation of the Steepest Descent Direction)

TD1 Set $R := 0$, $\boldsymbol{g} := \boldsymbol{o}$, and repeat following steps from TD2 to TD6 for all $a = 1, \cdots, N$.

TD2 Set $x^0 := \boldsymbol{x}_a$. For $k = 1$ to n, do

$$y_j^k := \sum_{i=0}^{m_{k-1}} w_{ji}^k x_i^{k-1}, \quad x_j^k := f(y_j^k).$$

TD3 Compute $R := R + \frac{1}{2} \sum_{j=1}^{m_n} (x_j^n - c_{aj})^2$.

TD4 Compute $dy_j^n := (x_j^n - c_{aj}) f'(y_j^n)$.

TD5 For $k = n$ down to 2, do

$$dy_i^{k-1} := \left(\sum_{j=1}^{m_k} w_{ji}^k dy_j^k \right) f'(y_i^{k-1}),$$
$$dw_{ji}^k := dy_j^k x_i^{k-1}, \quad g_{ji}^k := g_{ji}^k - dw_{ji}^k.$$

TD6 Compute $dw_{ji}^1 := dy_j^1 x_i^0$, $g_{ji}^1 := g_{ji}^1 - dw_{ji}^1$.

The equations in Step TD5 are derived from the computational graph shown in Fig. 2.

The following are the details of the algorithm for computing the inner products dg and dp and the quadratic forms ghg, ghp, and php by the bottom-up method.

Algorithm BU (Computation of Inner Products and Quadratic Forms)

BU1 Set $dg := 0$, $dp := 0$, $ghg := 0$, $ghp := 0$, $php := 0$, and repeat the following steps for all $a = 1, \cdots, N$.

BU2 Set $x^0 := \boldsymbol{x}_a$, $\boldsymbol{gx}^0 := \boldsymbol{o}$, $\boldsymbol{px}^0 := \boldsymbol{o}$, $\boldsymbol{ggx}^0 := \boldsymbol{o}$, $\boldsymbol{gpx}^0 := \boldsymbol{o}$, $\boldsymbol{ppx}^0 := \boldsymbol{o}$.

For $k = 1$ to n, do BU3.

BU3 For $j = 1$ to m_k, do

$$y := \sum_{i=0}^{m_{k-1}} w_{ji}^k x_i^{k-1},$$
$$gy := \sum_{i=0}^{m_{k-1}} (g_{ji}^k x_i^{k-1} + w_{ji}^k gx_i^{k-1}),$$
$$py := \sum_{i=0}^{m_{k-1}} (p_{ji}^k x_i^{k-1} + w_{ji}^k px_i^{k-1}),$$
$$ggy := \sum_{i=0}^{m_{k-1}} (2 g_{ji}^k gx_i^{k-1} + w_{ji}^k ggx_i^{k-1}),$$
$$gpy := \sum_{i=0}^{m_{k-1}} (g_{ji}^k px_i^{k-1} + p_{ji}^k gx_i^{k-1} + w_{ji}^k gpx_i^{k-1}),$$
$$ppy := \sum_{i=0}^{m_{k-1}} (2 p_{ji}^k px_i^{k-1} + w_{ji}^k ppx_i^{k-1}),$$
$$x_j^k := f(y), \quad gx_j^k := f'(y) gy, \quad px_j^k := f'(y) py,$$
$$ggx_j^k := f'(y) ggy + f''(y) gy^2,$$
$$gpx_j^k := f'(y) gpy + f''(y) gy\, py,$$
$$ppx_j^k := f'(y) ppy + f''(y) py^2.$$

BU4 For $j = 1$ to m_n, do

$$u := x_j^n - c_{aj},$$
$$dg := dg + u\, gx_j^n,$$
$$dp := dp + u\, px_j^n,$$
$$ghg := ghg + (gx_j^n)^2 + u\, ggx_j^n,$$
$$ghp := ghp + gx_j^n px_j^n + u\, gpx_j^n,$$
$$php := php + (px_j^n)^2 + u\, ppx_j^n.$$

The equations in Step BU3 are derived from the computational graph shown in Fig. 3 as follows:

The inner product $gx_j^k = \nabla x_j^k \cdot \boldsymbol{g}$ is obtained at node x_j^k in the middle part of Fig. 3 by the bottom-up method, if gx_i^{k-1} and g_{ji}^k are assigned to nodes x_i^{k-1} and w_{ji}^k, respectively. The right part of Fig. 3 represents the process of computing gx_j^k. The computational graph of the inner product px_j^k shown in the left part of Fig. 3 is obtained

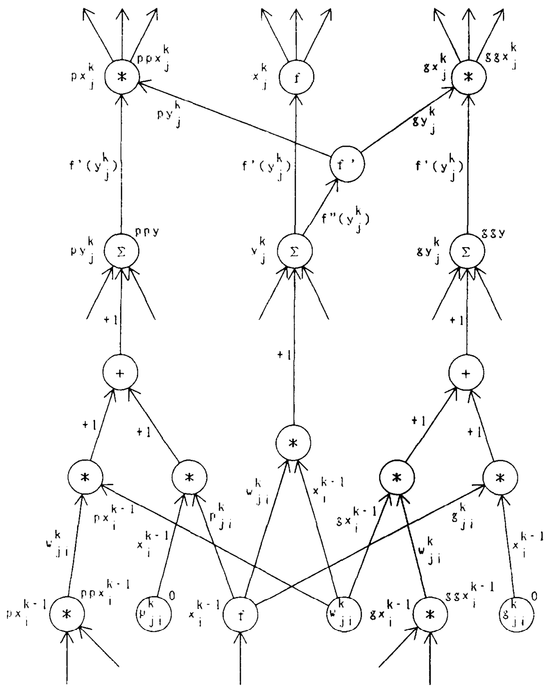


Fig. 3 Computational graph for the equations in Step BU3 of Algorithm BU.

in the same way. The bottom-up method computes the quadratic form $ggx_j^k = \boldsymbol{g}^t\nabla^2 x_j^k \boldsymbol{g}$ at node gx_j^k by assigning gx_i^{k-1}, g_{ji}^k, ggx_i^{k-1}, and 0 to nodes x_i^{k-1}, w_{ji}^k, gx_i^{k-1}, and g_{ji}^k, respectively. The quadratic form $gpx_j^k = \boldsymbol{g}^t\nabla^2 x_j^k \boldsymbol{p}$ is obtained at node gx_j^k by assigning px_i^{k-1}, p_{ji}^k, gpx_i^{k-1}, and 0 to nodes x_i^{k-1}, w_{ji}^k, gx_i^{k-1}, and g_{ji}^k, respectively. The quadratic form ppx_j^k is obtained in the same way as ggx_j^k.

Let us estimate the amount of computation and storage in algorithms TD and BU. Let $N(\boldsymbol{v})$ denote the number of elements of $\boldsymbol{v}$. According to the definition of this network model, the number of cells $N(\boldsymbol{x})$ and the number of connections $N(\boldsymbol{w})$ including threshold are

$$N(\boldsymbol{x}) = \sum_{k=0}^{n} m_k, \tag{18}$$

$$N(\boldsymbol{w}) = \sum_{k=1}^{n} m_k(m_{k-1}+1). \tag{19}$$

Algorithm TD uses the storage assigned to $\boldsymbol{w}$ and $\boldsymbol{x}$ for the evaluation of R, and the storage assigned to $\boldsymbol{w}$, $\boldsymbol{g}$, $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{dy}$ for the evaluation of R and $\boldsymbol{g}$. If the output function f is

$$f(y_i^k) = 2/(1+\exp(-y_i^k)) - 1, \tag{20}$$

its derivative and second derivative are

$$f'(y_i^k) = (1-(x_i^k)^2)/2, \tag{21}$$

$$f''(y_i^k) = -x_i^k f'(y_i^k). \tag{22}$$

In this case, $f'(y_i^k)$ and $f''(y_i^k)$ are computed from x_i^k, and there is no need to store the vector $\boldsymbol{y}$. Hence, the amount of storage needed for computing R and $\boldsymbol{g}$ is $2N(\boldsymbol{w})+2N(\boldsymbol{x})$, which is twice the amount of storage needed for computing R. The numbers of multiplications and additions in Step TD2 are both $N(\boldsymbol{w})$, and the number of operations for computing f is $N(\boldsymbol{x})-m_0$. Hence, R is computed by these operations and a few operations in Step TD3, namely, m_n+1 multiplications and $2m_n$ additions. Steps TD4, TD5, and TD6 involve $2N(\boldsymbol{w})+m_n-m_1(m_0+1)$ multiplications, $2N(\boldsymbol{w})+2m_n-N(\boldsymbol{x})-m_1m_0$ additions, and $N(\boldsymbol{x})-m_0$ f's. Hence, R and $\boldsymbol{g}$ are computed by almost three times the number of operations for R.

In Algorihm BU, if f is given by Eq. (20), $\boldsymbol{y}$ need not be stored. The amount of storage used in this algorithm is $3N(\boldsymbol{w})$ for $\boldsymbol{w}$, $\boldsymbol{g}$, and $\boldsymbol{p}$, and $6N(\boldsymbol{x})$ for $\boldsymbol{x}$, $\boldsymbol{gx}$, $\boldsymbol{px}$, $\boldsymbol{ggx}$, $\boldsymbol{gpx}$, and $\boldsymbol{ppx}$. Since $N(\boldsymbol{x})$ generally is much smaller than $N(\boldsymbol{x})$, this algorithm computes inner products and quadratic forms in a little more than three times the storage used for R. The numbers of multiplications and additions in this algorithm are approximately $14N(\boldsymbol{w})+11N(\boldsymbol{x})$ and $12N(\boldsymbol{w})+3N(\boldsymbol{x})$, respectively. The number of operations for computing each of f, f', and f'' is $N(\boldsymbol{x})$. Hence, inner products and quadratic forms can be evaluated in about 14 times the number of operations for R. The operations in one iteration of Algorithm CG are mostly the operations in Algorithms TD and BU. Therefore, the number of operations in an iteration is at most six times the number of operations required for back-propagation.

5. Computer Simulations

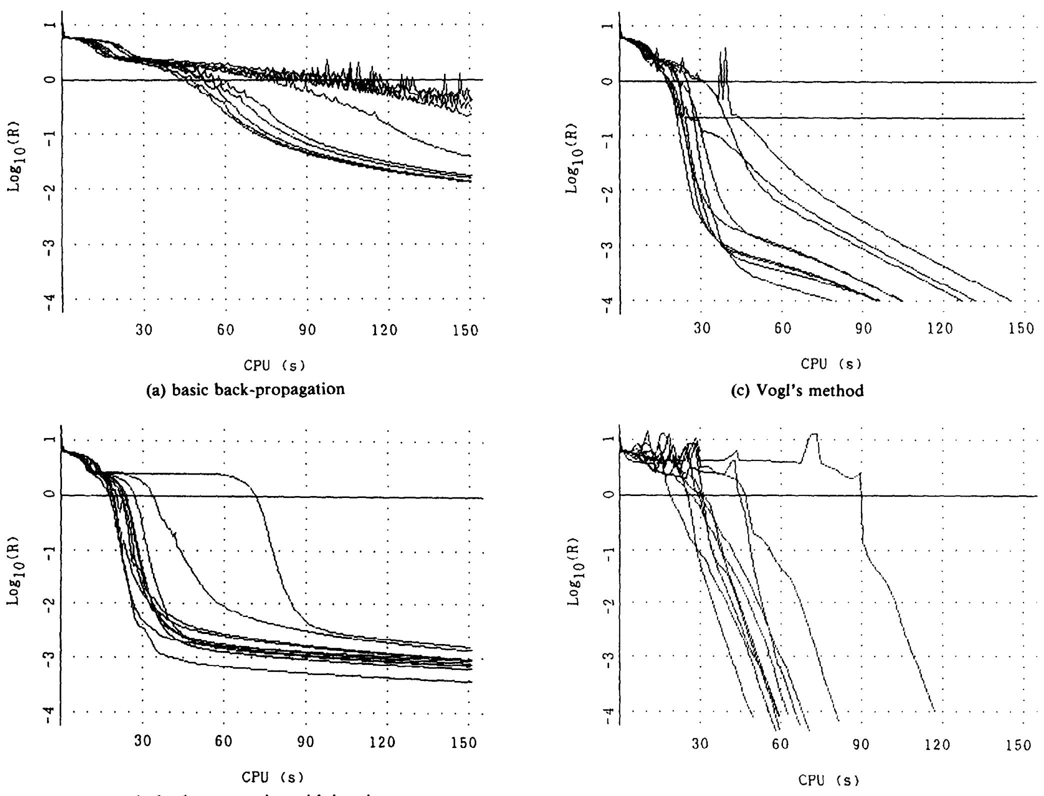
In this section, the proposed method is compared with basic back-propagation, back-propagation with inertia, and Vogl's method. The following simulations were done on a personal computer.[1] The programs for the simulations were written in C language.

5.1 Neural Network Model and Its Learning Subjects

We used a neural network model with four layers. The input layer has 20 cells, the first and the second layers have 10 cells each, and the output layer has 2 cells. The output function is given by Eq. (20). The network distinguishes signals from two autoregressive models (AR1 and AR2) of six degrees. For example, if a signal from AR1 is given to the input layer as a 20-dimensional vector, the output cell corresponding to AR1 outputs the value $+1$, and the other output cell the value -1.

In Training A, three signals each from AR1 and AR2 are used as training data. In Training B, 10 signals each from AR1 and AR2 are used as training data.

[1]CPU: 80286, arithmetic co-processor: 80287, clock: 10 MHz.



(a) basic back-propagation

(b) back-propagation with inertia

(c) Vogl's method

(d) the proposed conjugate gradient method

Fig. 4 Convergence behaviors for Training A using four learning methods.

5.2 Learning Methods

In the basic back-propagation method, the weights w are changed according to the rule

$$w := w + \eta g, \tag{23}$$

where $g = -\nabla R(w)$. In Training A and B, the learning rate η was 0.3 and 0.05, respectively.

Back-propagation with inertia changes w by the following equations:

$$\begin{aligned} p &:= \eta g + \alpha p, \\ w &:= w + p. \end{aligned} \tag{24}$$

In Training A and B, the learning rate η was 0.3 and 0.05, respectively, while the momentum factor α was 0.8 in both. In these methods, if overrun occurs, that is, if R increases, the values of w backtrack as described in Step CG4 of Algorihm CG.

Vogl's method controls the parameters η and α in Eq. (24) as follows:

If R is greater than the previous value of R, then w backtracks, and set $\eta := \beta\eta$, $\alpha := 0$.
Otherwise, set $\eta := \phi\eta$, $\alpha := \alpha_0$.

In Training A and B, the parameters ϕ, β, and α_0 were 1.05, 0.7, and 0.8, respectively.

In the proposed conjugate gradient method, the restarting parameter N was 100.

5.3 Results

The convergence behaviors for Training A and B in 10 trials with different initial values are shown in Figs. 4 and 5, respectively.

In Training A, the basic back-propagation iterated 100 to 160 times in 150 seconds. Its convergence was very show, as shown Fig. 4(a), and the error R did not decrease below 10^{-2}. The computation time for one iteration was 0.9 second. If η is less than 0.3, the occurrence of overrun decreases, but the convergence becomes slower. In the opposite case, if η greater than 0.3, the overrun occurs more often, so the convergence becomes slower. The back-propagation method with inertia and Vogl's method converged to $R < 10^{-2}$ in about 30 seconds. However, the error R decreased very slowly after 40 seconds. The computation time for one iteration was 0.95 second. The proposed method converged

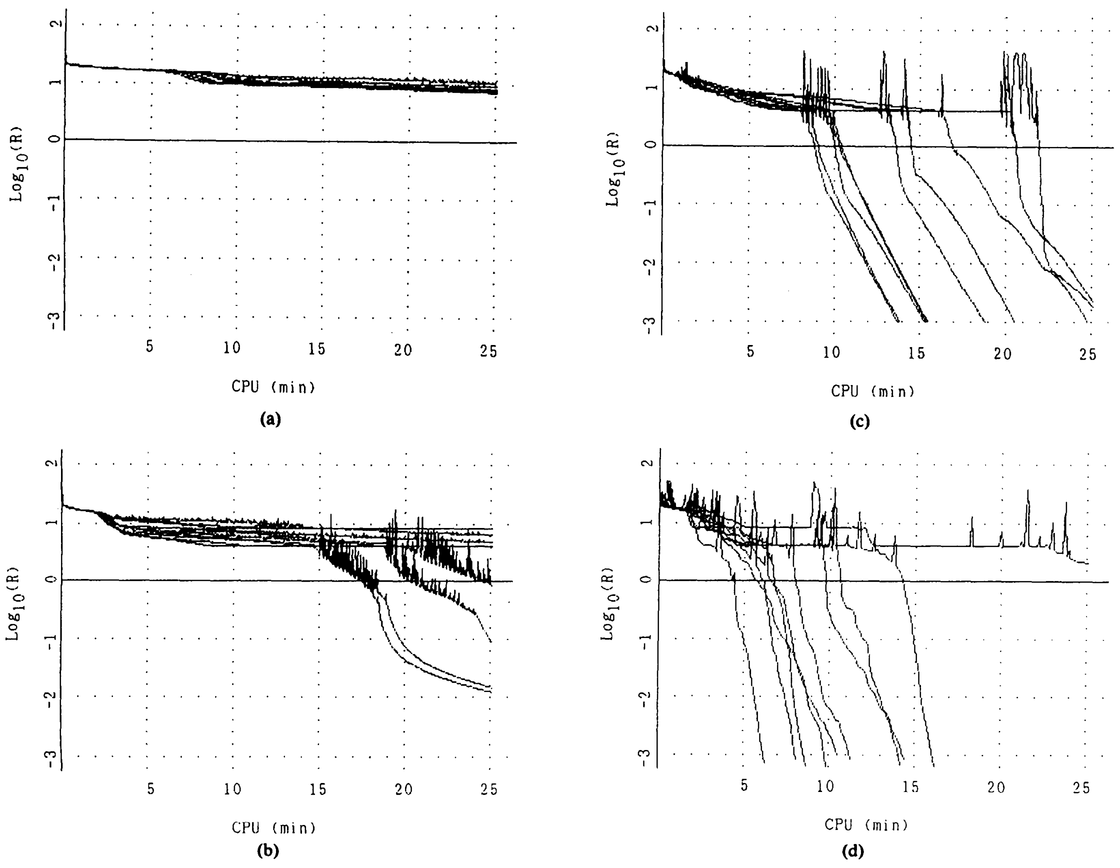


Fig. 5 Convergence behaviors for Training B using the four learning methods: (a)-(d) are the same methods as in Fig. 4.

to $R < 10^{-4}$ in 16 to 36 iterations. The computation time for one iteration was 2.8 seconds, which was about three times that in back-propagation.

In Training B, basic back-propagation with fixed η converged very slowly. This is because the optimal η varies in the course of the iterations. In the fastest trial of back-propagation with inertia, the error decreased to nearly 10^{-2} in 483 iterations and 70 overruns in about 1500 seconds. In the fastest trial of Vogl's method, the error decreased to below 10^{-3} in 276 iterations and 27 overruns in 820 seconds. The computation time for one iteration was 2.7 seconds. In the proposed method, nine trials converged to $R < 10^{-3}$. The fastest trial converged in 40 iterations and 6 overruns in 375 seconds, and the slowest trial converged in 98 iterations and 30 overruns in 970 seconds. The computation time for one iteration was 9.0 seconds, which was about three times that in back-propagation.

6. Conclusion

In this paper, we have proposed a rapid learning method for multilayered neural networks based on two-dimensional conjugate gradient search. This method gives optimal control of the learning rate and the momentum factor. The gradient, inner products, and quadratic forms are computed by automatic differentiation. The storage required for these computations is about three times the storage required for representing the network, and the number of operations needed during one iteration of the proposed method is at most six times that in back-propagation. Computer simulations have shown that the number of iterations for the proposed method is much less than that in back-propagation, while the time needed for one iteration is about three times that in back-propagation. If a parallel computer is used, the ratio of the computing time during one iteration of the proposed method to the back-propagation will be reduced, and more rapid learning can be expected. Subjects for future study include examination of Miele's exact two-dimensional search and development of learning methods using higher-order derivatives.

Acknowledgements

The author is greatly indebted to Professor Kenichi Kanatani of Gunma University for his helpful advice on ways to improve the manuscript.

References

1. WENGERT, R. E. *A Simple Automatic Derivative Evaluation Program, Comm. ACM*, 7, 8 (1964), 463-464.
2. RALL, L. B. *Automatic Differentiation: Techniques and Applications, Lecture Notes in Computer Science*, **120**, Springer-Verlag, Berlin (1981).
3. BAUR, W. and STRASSEN, V. *The Complexity of Partial Derivatives, Theor. Comput. Sci.*, **22** (1983), 317-330.
4. KIM, K. V., NESTEROV, YU. E., SKOKOV, V. A., and CHERKASSKII, B. V. *An Efficient Algorithm for Computing Derivatives and Extremal Problems, Ekonomika i matematicheskie metody*, **20**, 2 (1984), 309-318.
5. IRI, M. *Simultaneous Computation of Functions Partial Derivatives and Estimates of Rounding Errors—Complexity and Practicality, Jpn. J. Appl. Math.*, **1**, 2 (1984), 223-252.
6. SAWYER, J. W., JR. *First Partial Differentiation by Computer with an Application to Categorical Data Analysis, American Statistician*, **38**, 4 (1984), 300-308.
7. IWATA, N. *Automatization of the Computation of Partial Derivatives* (in Japanese), Master's Thesis, Information Engineering, Graduate School, University of Tokyo (March 1984).
8. KUBOTA, K. and IRI, M. *Formulation of Fast Automatic Differentiation and Its Implementation System* (in Japanese), The Institute of Statistical Mathematics Cooperative Research Report, Application of Graph Theory to Numerical Computation, 61-CR-14 (1987), 154-163.
9. KUBOTA, K. and IRI, M. *Formulation and Analysis of Computational Complexity of Fast Automatic Differentiation* (in Japanese), *Trans. IPS Japan*, **29**, 6 (1988), 551-560.
10. YOSHIDA, T. *Automatic Derivative Derivation System* (in Japanese), *Trans. IPS Japan*, **30**, 7 (1989), 799-806.
11. RUMELHART, D. E., HINTON, G. E., and WILLIAMS, R. J. *Learning Representations by Back-Propagating Errors, Nature*, **323** (1986) 533-536.
12. VOGL, T. P., MANGIS, J. K., RIGLER, A. K., ZINK, W. T., and ALCON, D. L. *Accelerating the Convergence of the Back-Propagation Method, Biological Cybernetics*, **59** (1988), 257-263.
13. YOSHIDA, T. *Conjugate Gradient Method with Two-Dimensional Search Using Automatic Differentiation, Journal of Optimization Theory and Applications*, in preparation.
14. MIELE, A. and CANTRELL, J. W. *Study on a Memory Gradient Method for the Minimization of Functions, Journal of Optimization Theory and Applications*, **3**, 6 (1969), 459-470.