A Fast New Algorithm for Training Feedforward Neural Networks

Robert S. Scalero and Nazif Tepedelenlioglu, *Member, IEEE*

***Abstract*—A fast new algorithm is presented for training multilayer perceptrons as an alternative to the backpropagation algorithm. The number of iterations required by the new algorithm to converge is less than 20% of what is required by the backpropagation algorithm. Also, it is less affected by the choice of initial weights and setup parameters.**

The algorithm uses a modified form of the backpropagation algorithm to minimize the mean-squared error between the desired and actual outputs with respect to the inputs to the nonlinearities. This is in contrast to the standard algorithm which minimizes the mean-squared error with respect to the weights. Error signals, generated by the modified backpropagation algorithm, are used to estimate the inputs to the nonlinearities, which along with the input vectors to the respective nodes, are used to produce an updated set of weights through a system of linear equations at each node. These systems of linear equations are solved using a Kalman filter at each layer.

I. Introduction

MULTILAYER perceptrons are feedforward neural networks which are used in a number of applications such as speech and image recognition. Because of the hidden layers they have, they overcome many limitations of single layer perceptrons. These types of networks are trained ahead of time, using known input/output data. Once trained, the network weights are frozen and unknown data can be run through the network.

The classical method for training a multilayer perceptron is the backpropagation algorithm [1], [2] which is an iterative gradient algorithm designed to minimize the mean-squared error between the desired output and the actual output for a particular input to the net.

Although it is successfully used in many cases, the backpropagation algorithm suffers from a number of shortcomings. One such shortcoming is the rate at which the algorithm converges. Many iterations are required to train small networks for even the simplest problems. For large network structures and data sets, it may take days or weeks in order to train the network. A training algorithm that reduces this time would be of considerable value.

It is the purpose of this paper to present a new alternative algorithm which is considerably faster than the backpropagation algorithm and has the added advantage of being less affected by poor initial weights and setup parameters (another shortcoming of the backpropagation algorithm).

The new algorithm uses a modified form of the backpropagation algorithm to minimize the mean-squared error between the desired output and the actual output with respect to the summation outputs (inputs to the nonlinearities). This is in contrast to the standard algorithm which minimizes the mean-squared error with respect to the weights. Error signals, generated by the backpropagation algorithm, are used to estimate values at the summation outputs that will improve the total network error. These estimates along with the input vectors to the respective nodes, are used to produce an updated set of weights through a system of linear equations at each node. These systems of linear equations are solved using a Kalman filter at each layer. Training patterns are run through the network until convergence is reached.

In Section II we briefly discuss the standard backpropagation algorithm and then in Section III we proceed to the new algorithm where we discuss the modifications to the backpropagation algorithm, derive the system of linear equations needed to solve for the weights, and incorporate a Kalman filter for solving the linear system. Following this, in Section IV experimental results are given showing comparisons between the two algorithms and finally Section V presents the main conclusions of the paper.

II. Backpropagation Algorithm

The backpropagation algorithm has become the standard algorithm used for training multilayer perceptrons of the type shown in Fig. 1. It is a generalized LMS algorithm that minimizes the mean-squared error between the actual and desired outputs of the network.

Since the backpropagation algorithm is treated in the literature [1], [2], we will only state the equations and summarize the standard algorithm here with the steps below:

Calculate the error signal for the output layer, i.e., for $j = L$ from

$$e_{pLk} = f'(y_{pLK})(o_{pk} - x_{pLK}) \tag{1}$$

and the error signal for the hidden layers from

$$e_{pjk} = f'(y_{pjk}) \sum_i (e_{p,j+1,i} w_{j+1,i,k}). \tag{2}$$

Manuscript received May 26, 1989; revised June 11, 1990.

R. S. Scalero is with Grumman Melbourne Systems, Melbourne, FL 32904-2322.

N. Tepedelenlioglu is with the Electrical and Computer Engineering Department, Florida Institute of Technology, Melbourne, FL 32901.

IEEE Log Number 9104009.

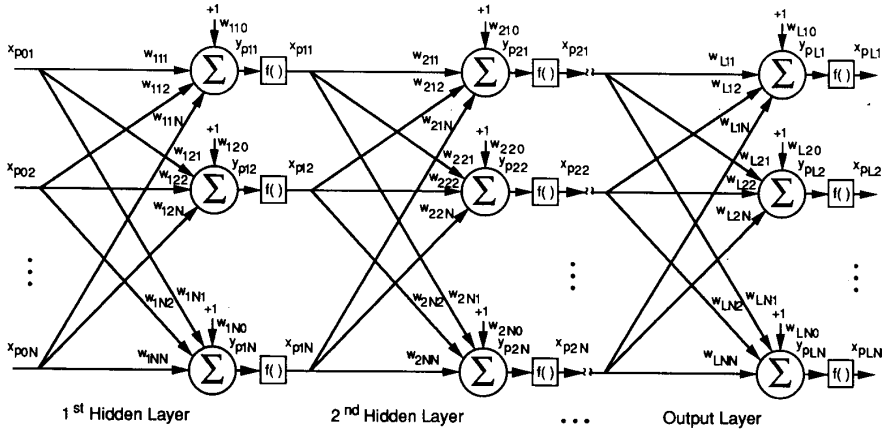


Fig. 1. Fully connected feedforward multilayer perceptron.

Update the weights for the tth iteration with

$$w_{jki}(t+1) = w_{jki}(t) + \mu e_{pjk} x_{p,j-1,i} + \Omega[w_{jki}(t) - w_{jki}(t-1)]. \quad (3)$$

The desired network output is expressed as o_{pk}, the summation and function outputs are y_{pjk} and x_{pjk}, respectively, μ is the step size, Ω is the momentum constant, $f'(y_{pjk})$ is the first derivative of $f(y_{pjk})$, and subscripts p, j, and k identify the pattern, layer, and node, respectively. The last term of (3) is the momentum term and is used to smooth out weight changes. Convergence is often faster with this.

Referring to Fig. 1, the algorithm proceeds as follows:

1) Initialize the weight vectors w_{jk} randomly.
2) Run a training pattern through the network.
3) Evaluate the error signals using (1) and (2).
4) Update the new weight vectors w_{jk} using (3).
5) Go to step 2 if the network has not converged.

As we note from (1)–(3), the standard backpropagation algorithm minimizes the error at the output of the nonlinearity with respect to the weights directly. The presence of a nonlinear function between the weights and the error is what makes the backpropagation algorithm different from the standard least squares adaptive filtering techniques which are known to have rapid convergence properties.

If the nonlinear neural network problem can be partitioned into linear and nonlinear parts, a wide range of techniques would be at our disposal for solving this problem. The next section develops a novel technique for minimizing the error with respect to the variable at the input of the nonlinearity (summation output), thereby rendering the determination of the weights using this variable a linear problem.

III. New Approach

A. Introduction

Referring to Figs. 1 and 2, one notices that if all node inputs $x_{p,j-1,k}$ and summation outputs y_{pjk} were specified, the problem would be reduced to a linear problem, i.e., a system of linear equations that relates the weight vector w_{jk} to the summation outputs y_{pjk} and the node inputs $x_{p,j-1,k}$.

The neuron inputs and outputs, however, are not totally defined for each layer in the network. Except for the inputs to first layer nodes and outputs from output layer nodes, which are specified by the training pattern and its associated desired response o_{pk}, respectively, none of the node I/O are known and must be estimated. Note that the desired summation outputs d_{pLk} for output layer nodes are also known through the inverse function of the desired response o_{pk}.

The estimates of the summation outputs y_{pjk} and node inputs $x_{p,j-1,k}$ will be designated as d^*_{pjk} and $x^*_{p,j-1,k}$, respectively. Since $x^*_{pjk} = f(d^*_{pjk})$, it is sufficient to estimate the desired summation outputs d^*_{pjk} only. This can be affected by using (1) and (2) in conjunction with

$$d^*_{pjk} = y_{pjk} + \mu e_{pjk}. \quad (4)$$

When using these estimates in the system of linear equations, the solution of the weight vector w_{jk} at each node will not be an exact one. It is then necessary to update our desired summation output estimates d^*_{pjk} with respect to the error produced at the output of the network. As training patterns are applied to the network, these estimates improve and consequently, so do the weight vectors.

To train the network, the weight vectors are first randomly initialized. The entire training set is run through the network and the error signals are calculated and used to update the desired summation output estimates d^*_{pjk}. At this point, new weights are calculated by using the system of linear equations at each node. The training set is then run through the network again and the process is continued until convergence is reached.

As we develop the algorithm, we will find that it is not necessary to wait for the entire training set to be run through the network before an update is performed, but

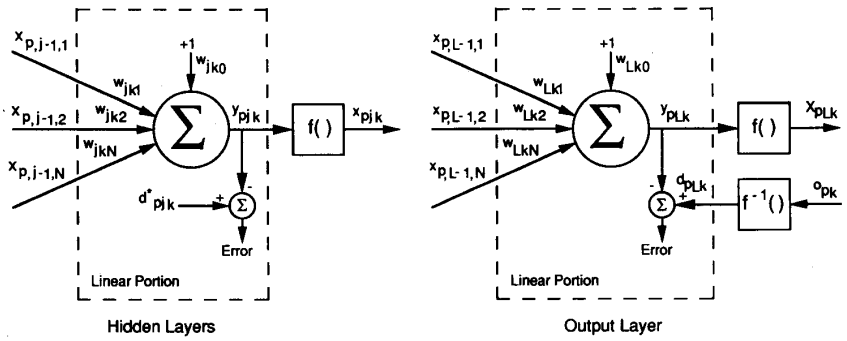


Fig. 2. Linear and error portion of the neurons in the hidden and output layers.

that updates may be performed for each training pattern (iteration).

B. Linear Equations

In this subsection we consider the linear portion of the neuron. This portion is shown in Fig. 2 for a neuron in the hidden layer as well as for one in the output layer. Since the following discussion will consider only a single neuron, we will omit the subscripts identifying individual neurons.

Once the desired summation output estimates d_p^* are calculated over the training set, we need to calculate the new vector $\boldsymbol{w}$ at each node. To do this, a system of linear equations must be developed that relates the weight vector $\boldsymbol{w}$ to the node's input vectors $\boldsymbol{x}_p$ and desired summation output d_p. These input vectors and desired outputs may be either known or estimated through (4) depending upon the position of the node in the network.

Since the network I/O is totally specified during training, the input vectors $\boldsymbol{x}_p$ to nodes in the first layer, and desired summation outputs d_p of nodes in the output layer (via the inverse function $d_p = f^{-1}(o_p)$), are totally known. All other interior inputs are not known and will be estimated by taking the function of d_p^* of the previous layer. For purposes of developing the system of linear equations, the estimation identifier will be dropped. Any input to a node, whether actual input vector $\boldsymbol{x}_p$ or its estimate $\boldsymbol{x}_p^*$, will be referred to as $\boldsymbol{x}_p$ and any actual desired summation output d_p or estimate d_p^*, as d_p.

The problem now is to minimize, with respect to the summation outputs y_p, the total mean-squared error E given by

$$E = \sum_{p=1}^{M} (d_p - y_p)^2 \tag{5}$$

where M is the number of training patterns, and y_p and d_p are the actual and the desired summation outputs for the pth training pattern. This error can be minimized by taking partial derivatives of E with respect to each weight and equating them to zero. The result will be a set of $N + 1$ linear equations where $N + 1$ is the number of weights in the neuron. Minimizing the error E with respect to a weight w_n produces

$$\frac{\partial E}{\partial w_n} = -2 \sum_{p=1}^{M} (d_p - y_p) \frac{\partial y_p}{\partial w_n} = 0 \tag{6}$$

for $n = 0$ through N. The summation output y_p is the inner product of the input vector $\boldsymbol{x}_p$ and weight vector $\boldsymbol{w}$. Taking the partial derivative of the summation output y_p with respect to a weight w_n we get the input x_{pn}. Making these substitutions, (6) can be written as

$$\frac{\partial E}{\partial w_n} = -2 \sum_{p=1}^{M} \left(d_p - \sum_{i=0}^{N} w_i x_{pi} \right) x_{pn} = 0 \tag{7}$$

or

$$\sum_{p=1}^{M} d_p x_{pn} = \sum_{p=1}^{M} \sum_{i=0}^{N} w_i x_{pi} x_{pn} \tag{8}$$

for $n = 0$ through N.

By rewriting the right-hand side of (8) as

$$\sum_{p=1}^{M} \sum_{i=0}^{N} w_i x_{pi} x_{pn} = \sum_{p=1}^{M} x_{pn} \sum_{i=0}^{N} w_i x_{pi} \tag{9}$$

and changing the right summation to a vector multiplication, we get

$$\sum_{p=1}^{M} x_{pn} \boldsymbol{w}^T \boldsymbol{x}_p = \left[\sum_{p=1}^{M} x_{pn} \boldsymbol{x}_p^T \boldsymbol{w} \right] \tag{10}$$

for $n = 0$ through N.

Defining

$$\boldsymbol{R} = \sum_{p=1}^{M} \boldsymbol{x}_p \boldsymbol{x}_p^T \tag{11}$$

and

$$\boldsymbol{p} = \sum_{p=1}^{M} d_p \boldsymbol{x}_p \tag{12}$$

(8) can be expressed in matrix form

$$\boldsymbol{p} = \boldsymbol{R}\boldsymbol{w}. \tag{13}$$

It must be noted that the matrix $\boldsymbol{R}$ can be interpreted as the correlation matrix of the training set, and the vector $\boldsymbol{p}$

as the cross correlation between the training patterns and the corresponding desired responses.

Equation (13) is referred to as the deterministic normal equation in the context of adaptive filtering [3].

The weight vector $\boldsymbol{w}$ in (13) may be solved for by using one of the standard techniques for solving a system of linear equations yielding

$$\boldsymbol{w} = \boldsymbol{R}^{-1}\boldsymbol{p}. \tag{14}$$

The derivation of (13) was based upon one neuron. During training, the matrix equation must be solved at each neuron in the network every time the weights are to be updated. While solving a matrix equation at each node seems to require an excessive amount of computations, a more serious problem, however, concerns the weight update.

Up to this point in the development, there has been the requirement that all the training patterns be run through the network before each update is performed. This poses no problem when the training set consists of a small number of patterns. However, for large training sets, the enormous amount of time required to pass the entire training set through the network before each weight update is performed would render the algorithm, in its present form, useless.

This problem can be eliminated by modifying the correlations in (11) and (12) and selecting training patterns to the network randomly.

We redefine (11) as

$$\boldsymbol{R}(t) = \sum_{k=1}^{t} \boldsymbol{x}(k)\boldsymbol{x}^{T}(k) \tag{15}$$

and (12) as

$$\boldsymbol{p}(t) = \sum_{k=1}^{t} d(k)\boldsymbol{x}(k) \tag{16}$$

where t is the number of the present iteration starting at $t = 1$. Thus (14) now becomes

$$\boldsymbol{w}(t) = \boldsymbol{R}^{-1}(t)\boldsymbol{p}(t). \tag{17}$$

Except for a factor of $1/t$, (15) and (16) are estimates of the correlation matrix and correlation vector, respectively, and they improve with increasing t. The weight at each node can now be updated at every iteration (training pattern) since one does not have to wait M training patterns for these estimates to be available.

In doing this, a new problem is created that may easily be fixed. With the exception of the first layer, estimates of one layer are based upon data that that layer receives from the previous layer. Since the previous layer is untrained at the beginning, the correlation estimates are not as good as they will be at or near the end of the training period. These inaccurate older estimates, however, are still strongly included in the estimation process later on when the network has gotten smarter. This would, at best, increase the training time of the network.

To remedy this problem, we insert a forgetting factor b into the correlations and rewrite (15) as

$$\boldsymbol{R}(t) = \sum_{k=1}^{t} b^{t-k}\boldsymbol{x}(k)\boldsymbol{x}^{T}(k) \tag{18}$$

and (16) as

$$\boldsymbol{p}(t) = \sum_{k=1}^{t} b^{t-k}\, d(k)\boldsymbol{x}(k). \tag{19}$$

The forgetting factor will allow the new information to dominate while the correlation estimates from earlier training will have negligible effect on the current estimates.

Both (18) and (19) may be written in their recursive form as

$$\boldsymbol{R}(t) = b\boldsymbol{R}(t-1) + \boldsymbol{x}(t)\boldsymbol{x}^{T}(t) \tag{20}$$

and

$$\boldsymbol{p}(t) = b\boldsymbol{p}(t-1) + d(t)\boldsymbol{x}(t). \tag{21}$$

Specifying the system correlations this way eliminates the problem associated with large numbers of training patterns. In fact, for very large training sets, convergence is often achieved long before all the training patterns have been presented to the network.

Although (20) and (21) are in recursive form, what one needs is a recursive equation for the inverse autocorrelation matrix $\boldsymbol{R}^{-1}(t)$ as required by (17). This can be achieved by using either the matrix inversion lemma [4] or what may be viewed as its compact form, the Kalman filter [5], [6].

C. Kalman Filter

A close examination of the last subsection will reveal that the problem at hand is a form of recursive least squares filtering. Some examples are treated in the literature especially within the context of adaptive channel equalization [5]–[7]. Therefore, we will not make an attempt to develop the theory here, but adapt the treatment in [5] to the solution in (17) iteratively. Thus, for the jth layer we have the following equations:

The Kalman gain vector:

$$\boldsymbol{k}_j(t) = \frac{\boldsymbol{R}_j^{-1}(t-1)\boldsymbol{x}_{j-1}(t)}{b_j + \boldsymbol{x}_{j-1}^{T}(t)\boldsymbol{R}_j^{-1}(t-1)\boldsymbol{x}_{j-1}(t)}. \tag{22}$$

The update equation for the inverse matrix:

$$\boldsymbol{R}_j^{-1}(t) = [\boldsymbol{R}_j^{-1}(t-1) - \boldsymbol{k}_j(t)\boldsymbol{x}_{j-1}^{T}(t)\boldsymbol{R}_j^{-1}(t-1)]b_j^{-1}. \tag{23}$$

The update equation for the weight vectors in the output layer:

$$\boldsymbol{w}_{Lk}(t) = \boldsymbol{w}_{Lk}(t-1) + \boldsymbol{k}_L(t)(d_k - y_{Lk}). \tag{24}$$

And the update equation for the weight vectors in the hidden layers:

$$\boldsymbol{w}_{jk}(t) = \boldsymbol{w}_{jk}(t-1) + \boldsymbol{k}_j(t)e_{jk}(t)\mu_j \tag{25}$$

where t is the present iteration number, and k is the node in the layer. Constants b_j and μ_j are the forgetting factor and backpropagation step size, respectively, in the jth layer.

It should be noted that the input vectors $\boldsymbol{x}_j(t)$ used in (24) and (25) are those which are present at the jth layer when the input pattern is propagated through the network. They are not modified using the backpropagated error. It is found experimentally that the Kalman filter works better this way.

The algorithm

1) Initialize.
   Equate the node offset $x_{j-1,0}$ of every node to some nonzero constant for layers $j = 1$ through L.
   Randomize all weights in the network.
   Initialize the inverse matrix $\boldsymbol{R}^{-1}$.
2) Select training pattern.
   Randomly select an input/output pair to present to the network. The input vector is $\boldsymbol{x}_0$ and desired network output vector is $\boldsymbol{o}$.
3) Run selected pattern through the network.
   For each layer j from 1 through L, calculate the summation output

$$y_{jk} = \sum_{i=0}^{N} (x_{j-1,i} w_{jki})$$

   and the function output

$$x_{jk} = f(y_{jk}) = \frac{1 - \exp(-ay_{jk})}{1 + \exp(-ay_{jk})}$$

   for every node k. N is the number of inputs (not including the offset) to a node, and constant a is the sigmoid slope.
4) Invoke Kalman filter equations.
   For each layer j from 1 through L, calculate the Kalman gain

$$\boldsymbol{k}_j = \frac{\boldsymbol{R}_j^{-1} \boldsymbol{x}_{j-1}}{b_j + \boldsymbol{x}_{j-1}^T \boldsymbol{R}_j^{-1} \boldsymbol{x}_{j-1}}$$

   and update the inverse matrix

$$\boldsymbol{R}_j^{-1} = [\boldsymbol{R}_j^{-1} - \boldsymbol{k}_j \boldsymbol{x}_{j-1}^T \boldsymbol{R}_j^{-1}] b_j^{-1}.$$

5) Backpropagate error signals.
   Compute the derivative of $f(y_{jk})$ using

$$f'(y_{jk}) = \frac{2a[\exp(-ay_{jk})]}{[1 + \exp(-ay_{jk})]^2}.$$

   Calculate error signals in the output layer, where $j = L$, by evaluating

$$e_{Lk} = f'(y_{Lk})(o_k - x_{Lk})$$

   for every node k.
   For the hidden layers, starting at layer $j = L - 1$ and decrementing through $j = 1$, find error signals by solving

$$e_{jk} = f'(y_{jk}) \sum_i (e_{j+1,i} w_{j+1,i,k})$$

   for every node in the jth layer.
6) Find the desired summation output.
   Calculate the desired summation output at the Lth layer by using the inverse function

$$d_k = \frac{1}{a} \ln \frac{1 + o_k}{1 - o_k}$$

   for every kth node in the output layer.
7) Update the weights.
   The weight vectors in the output layer L are updated by

$$\boldsymbol{w}_{Lk} = \boldsymbol{w}_{Lk} + \boldsymbol{k}_L(d_k - y_{Lk})$$

   for every kth node.
   For each hidden layer $j = 1$ through $L - 1$, the weight vectors are updated by

$$\boldsymbol{w}_{jk} = \boldsymbol{w}_{jk} + \boldsymbol{k}_j e_{jk} \mu_j$$

   for every kth node.
8) Test for completion.
   At this point we can either use the mean-squared error of the network output as a convergence test, or run the algorithm for a fixed number of iterations. Either way, if completion is not reached, go back to step 2.

IV. Experimental Results

To compare the performance of the new algorithm with that of the standard backpropagation algorithm, both algorithms were used to train networks to solve two problems. The first problem trains a network to determine if a set of coordinates are inside or outside a circle of a given radius (pattern classification). The second problem trains a network to perform pattern recognition. The network is presented a sequence of patterns together with the corresponding desired outputs and trained to identify the individual patterns. These experiments were programmed in Fortran and run on a Vax 8650.

A. Circle in the Square

The network is a 2 layer feedforward perceptron consisting of 2 inputs (x and y coordinates), 9 nodes in the first layer, and 1 node in the output layer. The input coordinates are selected randomly with values ranging from -0.5 through $+0.5$. If the distance of the training point from the origin is less than 0.35 the desired output is assigned the value -0.5 indicating that the point is inside the circle. A distance greater than 0.35 means that the point is outside the circle and the desired output becomes $+0.5$. Training patterns are presented to the network alternating between the two classes (inside and outside the circle).

Both algorithms were started from exactly the same ini-

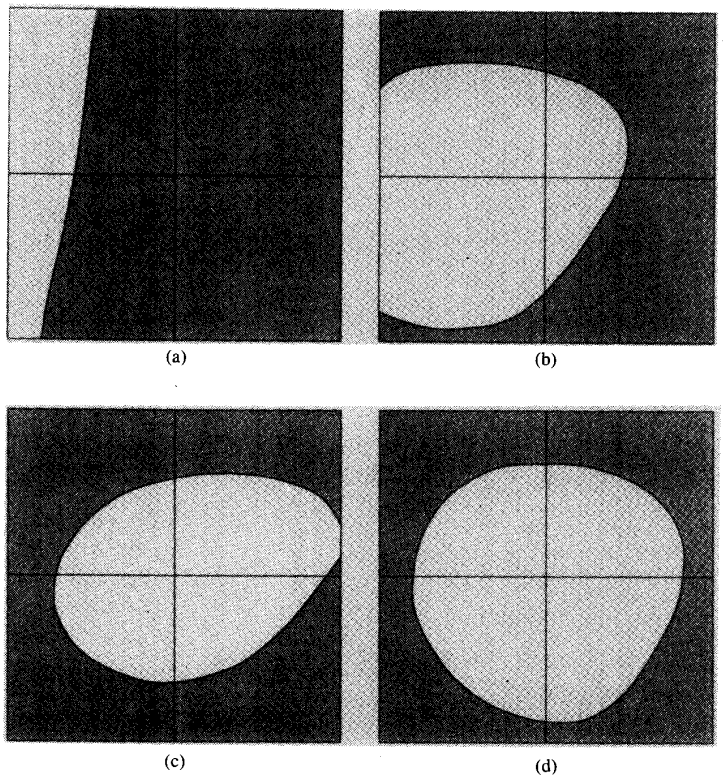


Fig. 3. New algorithm. (a) Iterations = 0. MSE = 0.801184. (b) Iterations = 100. MSE = 0.206176. (c) Iterations = 200. MSE = 0.115151. (d) Iterations = 300. MSE = 0.092668.

tial weights with values ranging from -20 through $+20$ and the training patterns were selected in exactly the same order. A sigmoid slope of 0.2 was chosen for all nonlinear function blocks in both algorithms.

The step size, forgetting factors, and momentum terms were selected to try and maximize the performance of each algorithm. However, an exhaustive search for the best possible parameters was beyond the scope of this paper and parameters may exist that may be slightly better for either algorithm. For the backpropagation algorithm, the step size and momentum term for both layers was 0.8 and 0.9, respectively. The new algorithm used forgetting factors of 0.99 for both layers and a first layer step size of 40. Although rather large, it must be remembered that this step size pertains to a different reference point (a summation output) than the backpropagation reference point (a weight).

The output of the network at various iteration numbers is shown in Fig. 3(a)–(d) for the new algorithm and Fig. 4(a)–(d) for the backpropagation algorithm. It is seen that the new algorithm forms a circle in 200 to 300 iterations whereas the backpropagation algorithm takes 1800 iterations.

It should be noted that this method of specifying the desired output forces the "training surface" to assume a top hat-like shape, a constant high value outside the circle and a constant low value inside the circle; which is not the best choice for the circle problem. Since during the training period, both algorithms try to minimize the mean-squared error between the actual and desired outputs, and since less error occurs when the output values are closer to -0.5 and $+0.5$, the algorithms tend to sacrifice the transition region in order to flatten the inside and outside regions. The result is a less than perfect circle. A training surface with continuous first derivatives and a gradual transition region would improve the appearance of the circle.

Fig. 5 shows the mean-squared error versus the iteration number for both algorithms during training. Here, 10 different training runs were averaged together. While the initial weights were changed from one run to the next, both algorithms started each run with identical weights and were both presented with the same training patterns. The new algorithm remains below a mean-squared error of 0.10 after 300 iterations (1.73 s of CPU time) as opposed to the backpropagation algorithm at 1800 iterations (4.84 s of CPU time. Furthermore, it can be calculated [8] that the new algorithm requires a total of 762 operations per iteration (mainly multiplications and additions) as opposed to 299 required by the backpropagation algorithm. Therefore, the total number of operations required by the backpropagation algorithm divided by that of the

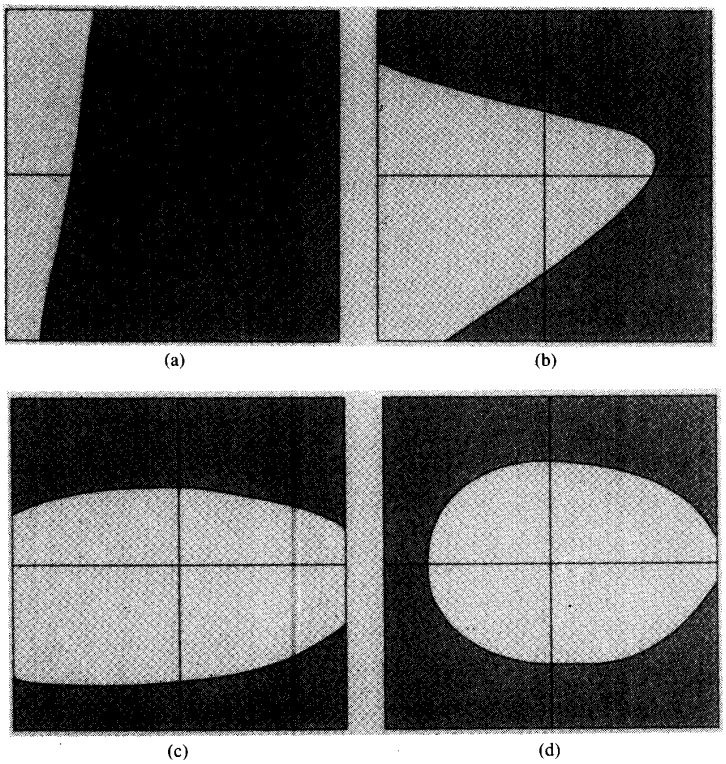


Fig. 4. Backpropagation. (a) Iterations = 0. MSE = 0.801184. (b) Iterations = 400. MSE = 0.228077. (c) Iterations = 1300. MSE = 0.156898. (d) Iterations = 1800. MSE = 0.099184.

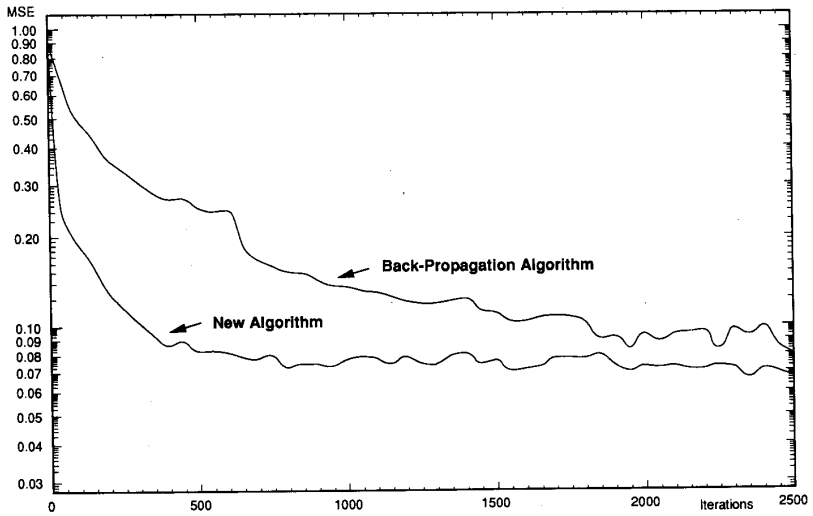


Fig. 5. Learning curves for the new algorithm versus backpropagation algorithm for the circle in the square problem. A two layer network with 9 nodes in the hidden layer and 1 node in the output layer was used.

new algorithm yields an improvement ratio of 2.35. It is interesting to note that the ratio of CPU times is 2.8 which indicates that the performance of the new algorithm is even better than the theoretical ratio of 2.35. This is because the new algorithm converges in fewer iterations, thus requiring less total overhead.

TABLE I
IMPROVEMENT RATIO OF THE NEW ALGORITHM OVER BACKPROPAGATION WITH THE MSE CONVERGENCE SET AT 0.25

7 × 7 Patterns 16 Input Nodes	4 Patterns (2 Outputs)	8 Patterns (3 Outputs)	12 Patterns (4 Outputs)	16 Patterns (4 Outputs)
New algorithm iterations	8	10	25	28
Backprop iterations	63500	166500	20500	52500
Iteration ratio	7937.50	16650.00	820.00	1875.00
Comp. ratio	0.3391	0.3459	0.3526	0.3526
Total improve.	2691.58	5758.74	289.10	661.06

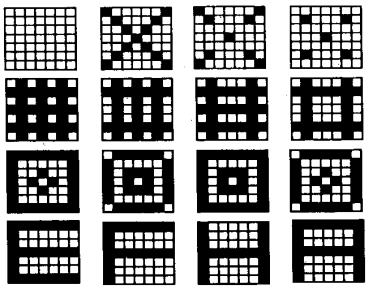
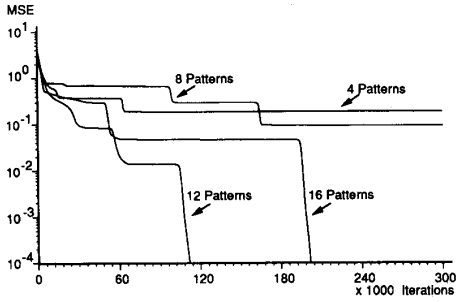
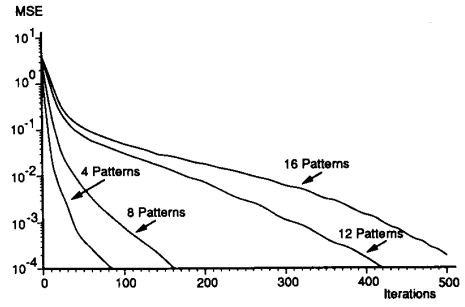


Fig. 6. Seven-by-seven patterns used for training.



(a)



(b)

Fig. 7. Learning curves for a 2 layer pattern recognition network with 16 nodes in the hidden layer. Seven-by-seven patterns were used for training. (a) Backpropagation algorithm. (b) New algorithm.

B. Pattern Recognition

The second experiment we consider is the training of a neural network to recognize patterns presented to its input. Although many different experiments were performed with various data sets and different network structures, we present here only a limited number because of space considerations. For additional results, the reader is referred to [8].

The input pixels are set to a level of -0.5 or $+0.5$. It is important to note that these levels are considered to be analog values rather than digital binary values. Although we present here experiments with patterns of two levels, similar results are obtained with patterns of various gray levels.

The output is likewise treated as analog. Therefore, the network is considered trained not only when the output agrees in sign with the desired output but also in absolute value.

The 7×7 pixel patterns in Fig. 6 are the inputs to a 2 layer feedforward perceptron with 16 nodes in the hidden layer. The desired output of the network is a 2, 3, or 4 b binary word depending upon the number of patterns used to train the network.

Fig. 7 shows the mean-squared error versus the iteration number for both algorithms during training for the 7×7 example. Table I presents numerically the performance comparison of the two algorithms shown in Fig. 7. This comparison takes into account the computational efficiency of each algorithm as well as the number of iterations required for the algorithm to reach a specified mean-squared error. The result is a time ratio of the two algorithms when run on a sequential machine. A mean-squared error convergence of slightly less than 0.25 was chosen since this value is the maximum that can be used and still produce correct results, assuming that the outputs are eventually passed through a hard limiter to produce a binary word.

V. CONCLUSIONS

We have presented in this paper a new algorithm which is faster than the standard backpropagation algorithm in training multilayer perceptrons.

The algorithm presented here converges in less than 20% of the time it takes for the backpropagation algorithm. Testing performed on 3 layer networks and net-

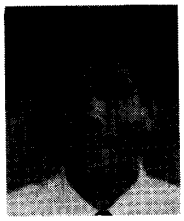
works with more neurons per layer, had equally impressive results. In one experiment (not shown) with 39 neurons in the first and second layers and 1 neuron in the output layer, the backpropagation algorithm took approximately 20 000 iterations to reach the same mean-squared error that the new algorithm achieved in 400.

Also, the new algorithm is more predictable in its training. In Fig. 7, we notice that the backpropagation algorithm tends to reach a certain mean-squared error and remain there for quite a while making little or no progress. At some point, it either rapidly converges, or jumps to a new level where it would again make little or no progress for quite a while. In contrast, the new algorithm continues to make steady progress toward improving the mean-squared error throughout the training period.

Finally, the convergence of the backpropagation algorithm depends heavily on the magnitude of the initial weights. If chosen incorrectly, the algorithm takes a long time to converge. The new algorithm seems to be less sensitive to the initial weight setting. Furthermore, the adaptive nature of the Kalman gain makes the new algorithm much less likely to get caught in a state other than the global minimum.

References

[1] R. P. Lippmann, "An introduction to computing with neural networks," *IEEE ASSP Mag.*, vol. 4, no. 2, Apr. 1987.
[2] D. E. Rumelhart and J. L. McClelland, *Parallel Distributed Processing*, vol. 1. Cambridge, MA: M.I.T. Press, 1986.
[3] S. S. Haykin, *Adaptive Filter Theory*. Englewood Cliffs, NJ: Prentice-Hall, 1986, pp. 312–314.
[4] B. D. O. Anderson and J. B. Moore, *Optimal Filtering*. Englewood Cliffs, NJ: Prentice-Hall, 1979, pp. 138–139.
[5] J. G. Proakis, *Digital Communications*. New York: McGraw-Hill, 1983, pp. 412–416.
[6] S. S. Haykin, *Adaptive Filter Theory*. Englewood Cliffs, NJ: Prentice-Hall, 1986, pp. 381–390.
[7] D. Godard, "Channel equalization using a Kalman filter for fast data transmission," *IBM. J. Res. Develop.*, vol. 18, pp. 267–273, 1974.
[8] R. S. Scalero, "A fast new algorithm for training feedforward neural networks," Ph.D. dissertation, Florida Institute of Technology, 1989.



Robert S. Scalero was born in Brooklyn, NY, on October 19, 1953. He received the B.S., M.S., and Ph.D. degrees in electrical engineering from the Florida Institute of Technology, in 1975, 1979, and 1989, respectively.

He joined Grumman Melbourne Systems Division (a division of Grumman Corporation) in 1987 and is currently a Technical Specialist in a research and development group. His research includes neural networks, radar, digital signal processing, and adaptive filtering.

Nazif Tepedelenlioglu (M'84) received the B.S. and M.S. degrees in electrical engineering in 1962 and 1963, respectively, from the Middle East Technical University, Turkey, and the Ph.D. degree in 1969 from the Polytechnic Institute of Brooklyn.

He returned to the Middle East Technical University where he taught until 1983. Since 1983 he has been an Associate Professor of Electrical Engineering at the Florida Institute of Technology. He is coauthor of the chapter "Fast Algorithms for Training Multilayer Perceptrons" in *Neural and Intelligent Systems Integration* (John Wiley and Sons, 1991). His research is in the areas of neural networks, adaptive systems, digital signal processing, and error control coding.