A Constructive Algorithm for Feedforward Neural Networks With Incremental Training

Derong Liu, Tsu-Shuan Chang, and Yi Zhang

***Abstract*—We develop, in this brief, a new constructive learning algorithm for feedforward neural networks. We employ an incremental training procedure where training patterns are learned one by one. Our algorithm starts with a single training pattern and a single hidden-layer neuron. During the course of neural network training, when the algorithm gets stuck in a local minimum, we will attempt to escape from the local minimum by using the weight scaling technique. It is only after several consecutive failed attempts in escaping from a local minimum that will we allow the network to grow by adding a hidden-layer neuron. At this stage, we employ an optimization procedure based on quadratic/linear programming to select initial weights for the newly added neuron. Our optimization procedure tends to make the network reach the error tolerance with no or little training after adding a hidden-layer neuron. Our simulation results indicate that the present constructive algorithm can obtain neural networks very close to minimal structures (with the least possible number of hidden-layer neurons) and that convergence (to a solution) in neural network training can be guaranteed. We tested our algorithm extensively using a widely used benchmark problem, i.e., the parity problem.**

***Index Terms*—Constructive algorithm, feedforward neural networks, incremental training, linear programming, quadratic programming.**

I. Introduction

Many researchers have studied the neural network training problem, and many algorithms have been reported. Although there have been many successful applications, there are still a number of issues that have not been completely resolved. These include the determination of the number of hidden-layer neurons, and the convergence as well as the speed of convergence in training. We say that a training is convergent if the training algorithm can eventually find a solution (i.e., a trained neural network) to the problem at hand without human intervention. This implies, in many cases, that the training algorithm can escape from local minima which the algorithm may visit during the course of a neural network training. Techniques reported in the literature to deal with the local minimum problem (i.e., the convergence problem) include weight scaling [6], [13] and dynamic tunneling [14].

The number of hidden-layer neurons is one of the most important considerations when solving problems using multilayered feedforward neural networks. An insufficient number of hidden-layer neurons generally results in the network's inability to solve a particular problem, while too many hidden-layer neurons may result in a network with poor generalization performance. The required number of hidden nodes depends on the dimension of the input space and the number of separable regions required to solve a particular classification (mapping) problem [11], [16]. Choosing an insufficient number of hidden neurons leads to an overdetermined problem since there are not enough parameters to accurately model the required decision boundaries [11]. One method for overcoming the hidden neuron problem is to allow the neural network to grow during training [1]–[3], [5], [8], [10], [12], [15] (for a survey, see [9]). Some of these methods start with a network that contains a small number of hidden neurons (e.g., start with a single hidden-layer neuron) and attempt to train the network using variants of the backpropagation algorithm or other optimization methods. If the training fails to converge, additional hidden-layer neurons are added so that the network can learn in a higher dimensional space. This provides a means for accelerated learning and the possibility for guaranteed convergence. One of the problems which has not been addressed much [1]–[3] is how to determine an optimal set of initial weights, so that the expanded network converges to the desired error tolerance in the least number of training steps possible.

In this brief, we will develop a new constructive training algorithm for feedforward neural networks. In our approach, patterns are trained incrementally by considering them one by one. A single pattern is chosen to be trained in conjunction with previously trained patterns, until all patterns are trained. Initially, weights are chosen to achieve a specified error for a single pattern, which is always feasible. A determination is then made for an additional pattern from the training set. Usually, the newly added pattern will introduce error larger than the tolerance. A gradient-based algorithm or an optimization-based algorithm will then be used to bring the system error back to the specified error tolerance. If the error can be reduced to within the specified error tolerance, a third pattern can be incorporated in the same way. If not, the network is allowed to grow. In this case, optimization techniques (e.g., quadratic programming, linear programming, etc.) will be used to determine a set of weights which reduces the system error as close to zero as possible. In particular, we develop a method for selecting the initial weights associated with the newly added hidden-layer neuron, while keeping all the previously obtained weights.

II. New Constructive Learning Algorithm

Without loss of generality, consider a two-layer feedforward neural network. The network contains N_i inputs, N_h hidden-layer neurons, and N_o output layer neurons. Let w_{jk} be the weight for the connection between the kth input node and jth hidden-layer neuron. Let v_{ij} denote the connection weight between the jth hidden-layer neuron and the ith output layer neuron. The threshold (or bias) terms for the hidden-layer neurons and the output layer neurons are included in $W = [w_{jk}] \in R^{N_h \times (N_i+1)}$ and $V = [v_{ij}] \in R^{N_o \times (N_h+1)}$.

Denote an input pattern by $x = [x_0, x_1, \ldots, x_{N_i}]^T$, where $x_0 = 1$ is used to make w_{j0}, $j = 1, 2, \ldots, N_h$, the threshold terms of the hidden layer. Its corresponding desired output pattern is denoted by $d = [d_1, d_2, \ldots, d_{N_o}]^T$. The input–output relationship of the network is given by

$$y_j = f\left(\sum_{k=0}^{N_i} w_{jk} x_k\right), \qquad \text{for } j = 1, 2, \ldots, N_h \tag{1}$$

and

$$z_i = f\left(\sum_{j=0}^{N_h} v_{ij} y_j\right), \qquad \text{for } i = 1, 2, \ldots, N_o \tag{2}$$

where y_j is the output of the jth hidden neuron, z_i is that of the ith output neuron, and $f(\cdot)$ is a chosen activation function such as the sigmoidal function or the hyperbolic tangent function. In this brief, we assume that $f^{-1}(\cdot)$ exists, $f(0) = 0$, and $f(u) = -f(-u)$ for all $u \in R$. Note that $y_0 = 1$, and thus, v_{i0}, $i = 1, 2, \ldots, N_o$, are the

Manuscript received November 15, 2001; revised June 14, 2002, and August 4, 2002. The work of D. Liu was supported by the National Science Foundation under Grants ECS-9732785 and ECS-9996428. The work of T.-S. Chang was supported by MICRO under Grants 97-021 and 97-022. This paper was recommended by Associate Editor X. Yu.

D. Liu and Y. Zhang are with the Department of Electrical and Computer Engineering, University of Illinois, Chicago, IL 60607 USA (e-mail: dliu@ieee.org).

T.-S. Chang is with the Department of Electrical and Computer Engineering, University of California, Davis, CA 95616 USA (e-mail: chang@ece.ucdavis.edu).

Digital Object Identifier 10.1109/TCSI.2002.805733

threshold terms of the output layer. The error function E_p for a given pattern is defined as

$$E_p = \frac{1}{2}\sum_{i=1}^{N_o}(z_i - d_i)^2.$$

In a neural network learning problem, there are N_p patterns given as $x^p = \left[x_0^p, x_1^p, \ldots, x_{N_i}^p\right]^T$ and $d^p = \left[d_1^p, \ldots, d_{N_o}^p\right]^T$, $p = 1, 2, \ldots, N_p$. Note again that $x_0^p = 1$. The objective is to find a neural network such that the overall error $E(N_p)$ is within a specified small error tolerance $\epsilon > 0$, where

$$E(N_p) = \frac{1}{N_p}\sum_{p=1}^{N_p} E_p.$$

Given a trained neural network for $L-1 < N_p$ patterns with its system error strictly below a specified tolerance ϵ, assume that there comes a new pattern, called pattern number L, to be incorporated into our training problem. In other words, our goal now is to find a neural network such that its error $E(L) < \epsilon$. Starting from the original weights, the neural network with N_h hidden neurons either can be continuously trained to achieve the same specified error for the L patterns in the new problem (case 1) or ends up with a large system error and gets stuck in a local minimum (case 2). In case 1, by repeating the same process with other new patterns one by one, the neural network can be eventually trained to achieve the objective if case 2 never happens. In case 2, the neural network with only N_h hidden neurons cannot reduce the system error to within ϵ. In other words, the neural network needs to be expanded to have at least $(N_h + 1)$ hidden-layer neurons in order to further reduce the system error. In such a case, the training problem has a neural network with $(N_h + 1)$ hidden neurons and L patterns.

If the original neural network for $L-1$ patterns has a minimum number of hidden neurons, the resulting neural network for the L patterns should most likely have a minimum number of neurons as well. For any given single pattern, it can be proven that there always exists a set of weights $W \in R^{1\times(N_i+1)}$ and $V \in R^{N_o\times 2}$ for a neural network with one hidden node $(N_h = 1)$ to achieve zero system error. We can then consider training the second pattern with the first one together to achieve the system error tolerance. Repeat the process of training and expanding (adding a hidden-layer neuron when stuck in a local minimum) until all the patterns are trained together. In summary, this suggests the following learning algorithm which possibly gets a neural network with a minimum number of hidden neurons and whose convergence to a solution can be guaranteed.

Algorithm 2.1: New Framework for Constructive Learning:

Input: Desired output patterns associated with N_p input patterns.
Output: A trained neural network within a specified system error tolerance ϵ and with the number of hidden neurons as small as possible.

Step 1: Set $l = 1$. Choose one pattern from the training set. Train a neural network with one hidden node using the chosen pattern to achieve the system error tolerance ϵ.
Step 2: If $l < N_p$, then, choose the next pattern according to certain criteria, set $l = l + 1$, and go to Step 3 for training; otherwise, stop.
Step 3: If the training algorithm can reduce $E(l)$ to within ϵ, go back to Step 2; otherwise, go to Step 4 for growing.
Step 4: Restore the weights of the last successfully trained neural network. Increase the number of hidden neurons by one, and assign its initial weights using a chosen strategy. Go to Step 3.

In neural network learning for real-time applications, training data may be generated and collected one by one in real-time. In this case, the whole set of training data will not be available before the end of an experiment/trial. If learning is desired in parallel to the data collection process, the criteria for choosing the next pattern in Step 2 of the present constructive learning algorithm is simply the next available pattern.

One way to determine initial weights for the newly added hidden neuron in Step 4 is to use the quadratic programming/linear programming formulation below. Denote the weight matrices of the last successful training in Step 3 of Algorithm 2.1 as W and V. For notational convenience, (1) and (2) are rewritten for a given input pattern x^p in a matrix format, where we use the subscript a to indicate that z_a^p is the activation signal for the output z^p. We have

$$\begin{aligned} y^p &= \begin{bmatrix} 1 \\ \ldots \\ f(Wx^p) \end{bmatrix} = [1, f(W_1x^p), \ldots, f(W_{N_h}x^p)]^T \\ z_a^p &\triangleq f^{-1}(z^p) = Vy^p \end{aligned} \tag{3}$$

where $W_1, W_2, \ldots, W_{N_h}$ are the rows of matrix W. After a hidden neuron is added, the new weight matrices become

$$\overline{W} = \begin{bmatrix} W \\ \ldots \\ W_r \end{bmatrix} \text{ and } \overline{V} = \left[\begin{array}{c:c} V & V_c \end{array}\right]$$

where W_r is a row vector, and V_c is a column vector. For the expanded neural network, one can write the system equations for an input pattern x^p as

$$\begin{aligned} \overline{y}^p &= \begin{bmatrix} y^p \\ \ldots \\ f(W_rx^p) \end{bmatrix} \\ &= [1, f(W_1x^p), \ldots, f(W_{N_h}x^p), f(W_rx^p)]^T \end{aligned} \tag{4}$$

and

$$\begin{aligned} \overline{z}_a^p &= \overline{V}\,\overline{y}^p = \left[\begin{array}{c:c} V & V_c \end{array}\right]\overline{y}^p \\ &= Vy^p + V_cf(W_rx^p) = z_a^p + V_cf(W_rx^p) \end{aligned} \tag{5}$$

where y^p is given in (3). The goal now is to have $\overline{z}_a^p \approx d_a^p$ for all applicable p, where $d_a^p \triangleq f^{-1}(d^p)$, and d^p is the target output corresponding to x^p.

From (4) and (5), we can state the following two observations.

1) If the pattern x^p is the new pattern which caused the training to be trapped in a local minimum, we denote the pattern by x^L (thus, the previously trained patterns are denoted by x^p, $p = 1, 2, \ldots, L-1$). In this case, when determining W_r and V_c, we would like to have $\overline{z}_a^L = d_a^L$, or equivalently

$$V_cf\left(W_rx^L\right) = d_a^L - Vy^L. \tag{6}$$

Note that (6) is only required for one pattern denoted by x^L and $d_a^L - Vy^L \neq 0$.

2) If the pattern x^p is a previously trained pattern, we have $z_a^p \approx d_a^p$, and we now require in this case

$$V_cf(W_rx^p) = \overline{z}_a^p - z_a^p \approx 0, \qquad p = 1, 2, \ldots, L-1 \tag{7}$$

when determining W_r and V_c.

Clearly, after adding a hidden-layer neuron, if we can determine the weights associated with the new neuron, i.e., W_r and V_c, such that (6) and (7) are satisfied, the expanded neural network with weights given by $\overline{W}$ and $\overline{V}$ will be a solution for the problem with a total of L patterns. We note that (6) and (7) are the basis for our quadratic/linear programming described next.

Assume first that V_c is a scaler, i.e., the network has only one output neuron. The approach described here can easily be extended to networks with more than one output neurons. From (6), we see that V_c can be chosen as

$$V_c = \alpha \left(d_a^L - V y^L \right) \tag{8}$$

where α is a scalar such that $|\alpha| > 1$ for the case when $|f(\cdot)| < 1$ (e.g., the sigmoidal function or the hyperbolic tangent). We note that in practice, most activation functions used for feedforward neural networks satisfy the property that $|f(\cdot)| < 1$. With this choice for V_c, (6) can now be written as $f\left(W_r x^L\right) = 1/\alpha$, or equivalently

$$W_r x^L = f^{-1}\left(\frac{1}{\alpha}\right). \tag{9}$$

In order to determine W_r through optimization techniques, we rewrite (7) as

$$-\delta \le V_c f(W_r x^p) \le \delta, \qquad p = 1, 2, \ldots, L-1 \tag{10}$$

where $\delta > 0$ should be small. We can write (10) as

$$-f^{-1}\left(\frac{\delta}{|V_c|}\right) \le W_r x^p \le f^{-1}\left(\frac{\delta}{|V_c|}\right)$$

or equivalently

$$-\lambda \le W_r x^p \le \lambda, \qquad p = 1, 2, \ldots, L-1 \tag{11}$$

where $\lambda > 0$ is small.

We can now use quadratic programming to minimize

$$W_r W_r^T + k\lambda^2 \tag{12}$$

subject to (9) and (11), where k is a parameter used to emphasize (11) in the optimization. For example, we choose $k = 2$ in our simulation studies. The larger the value of k, the more we emphasize (11) in the minimization of the expression in (12). We apply Matlab's implementation of quadratic programming `qp` in our simulation studies. Note that linear programming can also be used to solve W_r if we express (9) as an inequality, i.e., as

$$-\lambda + f^{-1}\left(\frac{1}{\alpha}\right) \le W_r x^L \le f^{-1}\left(\frac{1}{\alpha}\right) + \lambda \tag{13}$$

and if we minimize $\lambda > 0$ subject to (11) and (13). The linear programming problem in standard inequality form is given as [17]

$$\text{maximize } c^T \xi \text{ subject to } A\xi \le b$$

where A is an $m \times n$ coefficient matrix, b is an $m \times 1$ column vector, c is an $n \times 1$ vector, and the decision variables ξ_j, $j = 1, \ldots, n$, are contained in the $n \times 1$ column vector ξ. Using Matlab's implementation of linear programming `lp`, we choose $c^T \xi = -\lambda$, and we choose A and b accordingly. ξ in this case is formed of W_r and λ.

We choose to use the objective function as in (12) in our quadratic programming to prevent exceedingly large values for the components of W_r. It has been observed in our simulation that the linear programming as described above will sometimes result in large magnitude (e.g., exceeds 300 in magnitude) for the components of W_r. Our simulation results show that quadratic programming performs better than linear programming in some cases.

During the course of neural network training, the algorithm may get stuck in a local minimum. Techniques reported in the literature to deal with the local minimum problem (i.e., the convergence problem) include weight scaling [6], [13] and dynamic tunneling [14]. In [6] and [13], weight scaling is employed when the training process gets stuck in local minima. Assume that the weight vector obtained is given by W and the training algorithm is stuck at a local minimum. Using the weight scaling process, we will choose an initial weight vector as $W/\|W\|$ for the network training, where $\|\cdot\|$ is a vector norm. As pointed out in [6] and [13], weight scaling after the training gets to a local minimum will effectively reduce the degree of saturation of the activation function and thus keep a relatively large derivative of the activation function. After getting stuck in a local minimum, using weight scaling will resume a relative large weight updates which may eventually lead the training algorithm out of the local minimum. It is noted that weight scaling [6], [13] can also increase the speed of convergence in neural network training due to similar reasoning. In the present training algorithm, when the algorithm gets stuck in a local minimum, we will attempt to escape from the local minimum by using the weight scaling technique described here. Only after several consecutive failed attempts in escaping from a local minimum, will we allow the network to grow by adding a hidden-layer neuron. We note that the weight scaling process employed here may be substituted by any other techniques for attempting to escape from a local minimum, e.g., the dynamic tunneling technique [14]. However, the weight scaling process is much simpler to implement and works very well in our simulation.

Remark 1: Our constructive algorithm developed here can be combined with a neural network pruning algorithm as described in the framework of [3] so that tighter network structure can be obtained. We will show in Section III, however, that even without the pruning process, our constructive algorithm can usually obtain neural networks with minimal structure (least possible number of hidden-layer neurons) or obtain neural networks very close to its minimal structure. ■

III. Simulation Studies

In this section, we present our simulation results. A widely used benchmark problem is considered, i.e., the parity problem. The goal of our simulation is to show, using two-layer feedforward neural networks, how many hidden-layer neurons our algorithm can obtain for various parity problems. We will compare our algorithm using quadratic programming and using linear programming in selecting initial weights for newly added hidden-layer neurons. In our simulation studies, we choose to use hyperbolic tangent activation function in our simulation.

We have run our algorithm extensively for the parity problems. The results are displayed in Table I for 3–8-bit parity problems (see the numbers in the first row for each problem). In each case, the total number of patterns is given by 2^N, where N indicates N-parity problem. Table I shows the results when quadratic programming is used for determining the initial weights of the added hidden-layer neurons. All the results shown are for 50 test runs starting from different random initial conditions. In our simulation studies, our algorithm was successful in finding a solution for each problem in all of the 50 runs, i.e., all 50 runs for each problem tested were convergent in training. We also conducted the same experiments using linear programming (not shown in the Table); it normally results in more hidden-layer neurons than using quadratic programming. For example, in the 50 runs for the 7-bit parity problem, there are 19 runs where we obtained four hidden-layer nodes using quadratic programming while there are only ten runs where we obtained four hidden-layer nodes using linear programming. It has been shown in [16] that the N-bit parity problems can be solved using two-layer feedforward neural networks with

TABLE I
NUMBER OF HIDDEN-LAYER NEURONS OBTAINED VERSUS NUMBER OF RUNS OUT OF 50 RUNS FOR SEVERAL PARITY PROBLEMS WHEN QUADRATIC PROGRAMMING IS USED

Prob.	Number of nodes							Test set	
	2	3	4	5	6	7	8	A	B
3-bit	50 50	0	0	0	0	0	0	3	2.4
4-bit	0	37 38	13 12	0	0	0	0	5	3.9
5-bit	0	32 38	18 12	0	0	0	0	11	8.3
6-bit	0	0	28 24	22 26	0	0	0	21	16.4
7-bit	0	0	19 13	30 28	1 9	0	0	43	33.5
8-bit	0	0	0	14 7	33 23	3 14	0 6	85	68.7

$\lceil (N+1)/2 \rceil$ hidden sigmoidal units, where $\lceil \cdot \rceil$ denotes rounding toward $+\infty$. This implies that, for example, the 5- and 8-bit parity problems can be solved using two-layer feedforward neural networks with three and five hidden nodes, respectively. Our simulation results confirmed this finding in [16]. In Step 3 of our algorithm, we used the Levenberg–Marquardt algorithm [7] implemented in Matlab (the function `trainlm`) [4], where batch learning was used.

Finally, we repeated the experiments shown in Table I using two thirds of the data as training data and using the other one third of data as test data. To test our algorithm in a more realistic environment, we add noise uniformly distributed in $[-0.5, 0.5]$ to our training data. The results of experiments with noisy training data are shown in the second row for each problem in Table I. Our experiments show that adding noise to training data tend to increase the number of hidden-layer neurons required for solving the same parity problem. On the other hand, in experiments with only two thirds of the whole data set as training data, we would usually obtain neural networks with less number of hidden neurons if we do not add noise to the training set. As a result, in our experiments with noisy data using two thirds of the whole data set as training data, we obtain slightly higher number of hidden-layer neurons in each case than the ones with no noise and with the whole training data set for training. For example, for 7-bit parity problem, we obtained 13, 28, and nine runs with four, five, and six hidden neurons, respectively, in a total of 50 runs. The validation results of the test data are also shown in Table I (the last two columns). The numbers shown in the column under A are the total number of patterns in the test set and the numbers under B are the average numbers of test data patterns that are successful during validation among 50 experiments. Our results show an accuracy of 75%–80% in all the parity problems tested.

IV. CONCLUSION

We have developed in this brief a constructive learning algorithm for feedforward neural networks. In our approach, patterns are trained incrementally by considering them one by one. Our algorithm starts with a single training pattern and with a single hidden-layer neuron. The initial network training is guaranteed to converge to a solution. We then incorporate another pattern into the training process which may lead to one of the two possibilities, i.e., a successful training or getting stuck in a local minimum. For the former case, we continue the training process by incorporating more training patterns one by one until the training process gets stuck in a local minimum. For the latter case, we will allow the network to grow by adding a hidden-layer neuron. We have developed an optimization procedure based on quadratic/linear programming to determine initial values for the weights associated with the newly added hidden neuron. By using this optimization procedure, our algorithm can most likely bring the system error to either within the error tolerance (requiring no further training) or very close to it (requiring a few steps of further training).

Our algorithm is especially useful for real-time learning problems where training patterns are generated one by one in real-time, while neural network learning is required in parallel. Examples of this type of problems include stock market prediction and real-time learning control systems, among others. In real-time learning problems, we may from time to time encounter a new training pattern. Using our algorithm, if timely learning is desired, a neural network can learn the new pattern with no training or little training through the addition of a hidden node. This is achieved through the use of quadratic/linear programming for determining the initial weights associated with the new hidden neuron. If time permits, we can first try to see if our algorithm can learn the new pattern and then decide whether to add a hidden-layer neuron.

ACKNOWLEDGMENT

The authors would like to thank Dr. M.E. Hohil and Dr. S. H. Smith for participating in the early stages of this project.

REFERENCES

[1] M. R. Azimi-Sadjadi, S. Sheedvash, and F. O. Trujillo, "Recursive dynamic node creation in multilayer neural networks," *IEEE Trans. Neural Networks*, vol. 4, pp. 242–256, Mar. 1993.
[2] T.-S. Chang and K. A. S. Abdel-Ghaffar, "A universal neural net with guaranteed convergence to zero system error," *IEEE Trans. Signal Processing*, vol. 40, pp. 3022–3031, Dec. 1992.
[3] T.-S. Chang and J. N. Hwang, "A novel framework for neural net learning," in *Intelligent Engineering Systems Through Artificial Neural Networks*. New York: ASME, 1996, vol. 6, pp. 169–174.
[4] H. Demuth and M. Beale, *Neural Network Toolbox User's Guide, Version 3*. Natick, MA: MathWorks, 1998.
[5] M. Frean, "The upstart algorithm: A method for constructing and training feedforward neural networks," *Neural Comput.*, vol. 2, no. 2, pp. 198–209, 1990.
[6] Y. Fukuoka, H. Matsuki, H. Minamitani, and A. Ishida, "A modified back-propagation method to avoid false local minima," *Neural Netw.*, vol. 11, no. 6, pp. 1059–1072, 1998.
[7] M. T. Hagan and M. B. Menhaj, "Training feedforward networks with the Marquardt algorithm," *IEEE Trans. Neural Networks*, vol. 5, pp. 989–993, Nov. 1994.
[8] Y. Hirose, K. Yamashita, and S. Hijiya, "Back-propagation algorithm which varies the number of hidden units," *Neural Netw.*, vol. 4, no. 1, pp. 61–66, 1991.
[9] T.-Y. Kwok and D.-Y. Yeung, "Constructive algorithms for structure learning in feedforward neural networks for regression problems," *IEEE Trans. Neural Networks*, vol. 8, pp. 630–645, May 1997.
[10] P. G. Maghami and D. W. Sparks, "Design of neural networks for fast convergence and accuracy: Dynamics and control," *IEEE Trans. Neural Networks*, vol. 11, pp. 113–123, Jan. 2000.
[11] G. Mirchandani and W. Cao, "On hidden nodes for neural nets," *IEEE Trans. Circuits Syst.*, vol. 36, pp. 661–664, May 1989.
[12] R. Parekh, J. Yang, and V. Honavar, "Constructive neural-network learning algorithms for pattern classification," *IEEE Trans. Neural Networks*, vol. 11, pp. 436–451, Mar. 2000.
[13] A. K. Rigler, J. M. Irvine, and T. P. Vogl, "Rescaling of variables in back propagation learning," *Neural Netw.*, vol. 4, no. 2, pp. 225–229, 1991.
[14] P. RoyChowdhury, Y. P. Singh, and R. A. Chansarkar, "Dynamic tunneling technique for efficient training of multilayer perceptrons," *IEEE Trans. Neural Networks*, vol. 10, pp. 48–55, Jan. 1999.
[15] R. Setiono and L. C. K. Hui, "Use of quasi-Newton method in a feedforward neural network construction algorithm," *IEEE Trans. Neural Networks*, vol. 6, pp. 273–277, Jan. 1995.
[16] E. D. Sontag, "Feedforward nets for interpolation and classification," *J. Comput. Syst. Sci.*, vol. 45, no. 1, pp. 20–48, 1992.
[17] G. E. Thompson, *Linear programming: An Elementary Introduction*. New York: Macmillan, 1971.