Automated Learning for Reducing the Configuration of a Feedforward Neural Network

Chin-Chi Teng and Benjamin W. Wah, *Fellow, IEEE*

***Abstract*—In this paper, we present two learning mechanisms for artificial neural networks (ANN's) that can be applied to solve classification problems with binary outputs. These mechanisms are used to reduce the number of hidden units of an ANN when trained by the cascade-correlation learning algorithm (CAS). Since CAS adds hidden units incrementally as learning proceeds, it is difficult to predict the number of hidden units required when convergence is reached. Further, learning must be restarted when the number of hidden units is larger than expected. Our key idea in this paper is to provide alternatives in the learning process and to select the best alternative dynamically based on run-time information obtained. Mixed-mode learning (MM), our first algorithm, provides alternative output matrices so that learning is extended to find one of the many one-to-many mappings instead of finding a unique one-to-one mapping. Since the objective of learning is relaxed by this transformation, the number of learning epochs can be reduced. This in turn leads to a smaller number of hidden units required for convergence. Population-based learning for ANN's (PLAN), our second algorithm, maintains alternative network configurations to select at run time promising networks to train based on error information obtained and time remaining. This dynamic scheduling avoids training possibly unpromising ANN's to completion before exploring new ones. We show the performance of these two mechanisms by applying them to solve the two-spiral problem, a two-region classification problem, and the Pima Indian Diabetes Diagnosis problem.**

I. Introduction

Among many artificial-neural-network (ANN) applications, pattern classifiers are considered very important and have been used widely in applications such as image analysis, image understanding, and speech understanding. Two tasks are involved in the design of an ANN-based pattern classifier. First, a proper ANN configuration, including the number of hidden units and the connections among ANN units, has to be determined. Second, the ANN-based pattern classifier has to be trained by supervised learning. Here, an application problem is specified by a set of input and desired output patterns (called training patterns), and training involves finding the weights of connections that correctly map each input pattern to the corresponding desired output pattern.

In general, it is desirable to have small ANN's in implementing a pattern classifier. This is true because increasing the number of hidden units in an ANN may improve its approximation quality with respect to its training patterns, but not always improves its generalization behavior to new patterns. An improperly chosen configuration may result in either overfitting of the training patterns or nonconvergence in learning. One way to improve the generalization behavior of an ANN is to reduce its number of hidden units when convergence is reached. In addition, smaller ANN's are faster when deployed.

Due to difficulties of existing learning methods in learning the weights of small networks, learning methods have been developed to incrementally change the network structure once learning fails in a small network. These methods include destructive, constructive, genetic-algorithm, and pattern-classification approaches [33].

Destructive or pruning methods start from a fairly large network and dynamically remove unimportant connections or units [3], [14], [21], whereas constructive or growth methods start from a small network and dynamically grow the network. Since the latter usually require less computations, extensive research has been carried out in this area [1], [4], [7], [8], [12], [13], [15]–[18], [22]. Another class of dynamic multilayer perceptrons is block-feedback neural networks [26], [27] which can be learned incrementally.

A constructive method that we study in detail in this paper is Fahlman and Lebiere's cascade-correlation learning algorithm (CAS) [9], [10]. This algorithm starts from a minimal ANN with an input layer and an output layer, adds new hidden units, and trains the corresponding weights until the ANN can map all the inputs to the corresponding outputs to within an error threshold. The main advantage in using CAS is that it can automatically find the size and the topology of the resulting ANN without specifying them before training begins. Also, the learning speed of CAS is comparable to other supervised learning algorithms.

One problem with CAS is that it modifies an ANN configuration dynamically; hence, the number of hidden units when CAS terminates is not bounded. Not bounding the number of hidden units in training may lead to overfitting the weights to the training patterns.

In this paper, we study two learning mechanisms for reducing the number of hidden units of an ANN when trained by CAS. The key idea of these mechanisms is to provide alternatives during learning and to select dynamically the best alternative that leads to convergence with a small number of hidden units.

- *Mixed-mode learning (MM)* is used to train a single ANN with an objective of minimizing the number of hidden units [29]. MM transforms CAS from finding

Manuscript received August 15, 1994; revised September 11, 1995 and March 12, 1996. This work was supported by National Science Foundation Grant MIP 92-18715 and Joint Services Electronics Program Contract JSEP N00014-90-J-1270.

C.-C. Teng is with the Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL 61801 USA.

B. W. Wah is with the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL 61801 USA.

Publisher Item Identifier S 1045-9227(96)06611-8.

a one-to-one mapping into one that finds one-to-many mappings. Since the learning objective is relaxed, it needs fewer training epochs than CAS. In general, the number of hidden units required in CAS for convergence is a monotonically nondecreasing function of learning epochs. Hence, a reduction in the number of learning epochs can lead to a reduction in the number of hidden units required for convergence.

- *Population-based learning for ANN's (PLAN)* is used to find a small network configuration under a user-specified time constraint. PLAN is a generate-and-test method that maintains a population of candidate ANN's, and selectively trains those that are predicted to require smaller configurations. Its goal is to find an ANN configuration with a small number of hidden units as compared to the alternative of applying MM + CAS repeatedly to train each ANN to completion before exploring new ANN's. To decide on a particular ANN to train, PLAN divides the time allowed into quanta, and picks the best ANN to be trained in each quantum based on dynamic information on training already performed on this and other ANN's in the population. As it is difficult to predict the exact number of hidden units required by an ANN when training is completed, PLAN compares two partially trained ANN's and predicts which one will converge with a smaller number of hidden units relative to the other. Our prediction mechanism is based on a comparator neural network (CANN) [32] that takes as inputs the total sum of squared errors (TSSE)-versus-time behavior of training performed already on two ANN's, and predicts which one will require a smaller number of hidden units when convergence is reached.

This paper is organized as follows. We first summarize CAS in Section II. In Section III, we present MM, our mixed-mode learning mechanism. Section IV describes PLAN, our population-based learning system. All the experiments reported in Sections III and IV were run on a Sun SparcStation 10/51 workstation. Conclusions are drawn in Section V.

II. Cascade-Correlation Learning Algorighm (CAS)

CAS was first proposed by Fahlman and Lebiere in 1990 [9], [10]. Starting from a minimal ANN with only an input layer and an output layer, it automatically adds new hidden units one by one to create a multilayered ANN structure, and trains the resulting network until convergence is reached. It has two training phases: *TRAIN_OUTPUT* and *TRAIN_INPUT*.

The *TRAIN_OUTPUT* phase trains the weights in the output layer by the *quick-prop* learning algorithm [6] in order to minimize TSSE. It stops when there is no significant reduction in TSSE after T_{out} (user-specified) training epochs. At this time, the next phase starts.

In the *TRAIN_INPUT* phase, N_{cand} (user-specified) candidate hidden units are first inserted into the ANN. The input of each candidate hidden unit is connected to the output of every input unit and every previously inserted hidden unit, and its output is left open. The weights on the input side of each candidate unit are then trained using a gradient-descent algorithm in order to maximize Y, the sum over all output units of the covariance between the candidate unit's value and the output errors, and

$$Y = \sum_{o=1}^{n} \left| \sum_{j=1}^{k} (V_j - \overline{V})(E_{j,o} - \overline{E}_j) \right| \tag{1}$$

where V_j is the output value of a candidate unit after applying the jth training pattern; $E_{j,o}$ is the output error at the oth output unit after applying the jth training pattern; $\overline{V} = 1/k \sum_{j=1}^{k} V_j$; $\overline{E}_j = 1/n \sum_{o=1}^{n} E_{j,o}$; k is the number of training patterns; and n is the number of output units.

The *TRAIN_INPUT* phase stops when there is no significant improvement in Y after T_{in} (user-specified) training epochs. Among the candidate units, the one with the maximum Y becomes a new hidden unit in the ANN, and the others are deleted. Once the new hidden unit has been added, the weights on its input side are frozen permanently, and links are created between its output side and all output units. At this time, the next *TRAIN_OUTPUT* phase is started. These two phases are then executed alternately until TSSE is below a user-specified threshold value.

Note that CAS works in a *batch learning* mode; that is, weight updates are made only after all the training patterns have been presented.

There are two advantages in using CAS. First, it can automatically find the size and the topology of the resulting ANN without specifying them before training begins. This avoids the problem of overspecifying the number of hidden units initially. Second, learning in CAS is fast as it updates only the weights for the new candidate hidden units added, and the weights of previously added hidden units are fixed after they were added. This allows efforts in previous training to be retained.

On the other hand, it is hard to predict the number of hidden units needed before training is completed. When training is finished and the number of hidden units is larger than what is expected, the entire training process has to be restarted. To prevent unnecessary training before obtaining a desired ANN, or to obtain a network with the smallest number of hidden units given a fixed amount of training time, it is necessary to identify unpromising configurations before their training is completed. In the next two sections, we present two methods to reduce the number of hidden units required for convergence in CAS.

III. Mixed-Mode Learning (MM)

In this section, we present our MM learning mechanism for improving CAS when applied to solve classification problems with binary outputs. We first present a noniterative learning algorithm based on linear programming for training single-layer neural networks. Using this algorithm, we then present our MM learning mechanism and its application in CAS.

A. Transforming Supervised Learning of a Single-Layer Network into Solving a Linear Programming Problem

An application suitable for supervised learning can be modeled as a mapping of an *input pattern matrix* P (with

k patterns, each with m values) into an *output pattern matrix* D (with k patterns, each with n values). P is, therefore, a $k \times m$ matrix, and D, a $k \times n$ matrix. Let P^T and D^T be the transposes of P and D, respectively. Further, let p_i be the ith input pattern, and d_i be the corresponding ith output pattern. We have

$$P^T = [p_0 \quad p_1 \cdots p_{k-1}]$$

where

$$p_i = [p_{i,0} \quad p_{i,1} \cdots p_{i,m-1}] \tag{2}$$
$$D^T = [d_0 \quad d_1 \cdots d_{k-1}]$$

where

$$d_i = [d_{i,0} \quad d_{i,1} \cdots d_{i,n-1}]. \tag{3}$$

The single-layer neural network to be learned performs a mapping from P to D.

Assume initially that the number of output units is one ($n = 1$). Since the classification problems that we study in this paper have binary outputs, we assume the 40-20-40 criterion; that is, an output is considered a logic ONE if it is larger than 0.6, and ZERO if smaller than 0.4.

When the sigmoid function S [$= 1/(1+e^{-ax})$] with gain a is used as the activation function, the problem of learning the weights of a single-layer single-output neural network is the same as that of getting a weight matrix $W_{m\times 1}$ where

$$W^T = [w_0 \quad w_1 \cdots w_{m-1}] \tag{4}$$

such that

$$\begin{aligned} p_i \cdot W &\geq S^{-1}(0.6) \quad \text{if} \quad d_i = 1 \\ p_i \cdot W &\leq S^{-1}(0.4) \quad \text{if} \quad d_i = 0 \\ &\text{for all} \quad i = 0, 1, 2, \cdots, k-1 \end{aligned} \tag{5}$$

where $S^{-1}(x)$ is the inverse sigmoid function. Since $S^{-1}(0.6) = -S^{-1}(0.4)$, (5) can be represented in matrix form as follows:

$$(\hat{P} \cdot W)^T \geq [S^{-1}(0.6) \quad S^{-1}(0.6) \cdots S^{-1}(0.6)]_{1\times k} \tag{6}$$

where

$$\hat{P}^T = [\hat{p}_0 \quad \hat{p}_1 \cdots \hat{p}_{k-1}]$$

and

$$\hat{p}_i = \begin{cases} p_i & \text{if } d_i = 1 \\ -p_i & \text{if } d_i = 0. \end{cases} \tag{7}$$

The process of obtaining W that satisfies (6) is very similar to that of finding a feasible solution of a linear program. The only difference is that the elements of W can be negative, whereas variables in linear programming problems have to be positive. To handle unrestricted variables, we transform the elements of W into positive variables x_j as follows:

$$x_j = w_j + \eta, \quad x_j \geq 0, \quad \eta \geq \quad j = 0, \cdots, m-1. \tag{8}$$

Assuming the following definitions:

$$A = [\hat{P} \quad Y]_{k\times(m+1)}$$

$$X = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{m-1} \end{bmatrix}_{m\times 1}$$
$$V = \begin{bmatrix} X \\ \\ \eta \end{bmatrix}_{(m+1)\times 1}$$
$$B = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{k-1} \end{bmatrix}_{k\times 1} \tag{9}$$

where

$$Y^T = [y_0, \cdots, y_{k-1}]$$
$$y_i = -\sum_{j=0}^{m-1} \hat{p}_{i,j}$$

and

$$b_i = S^{-1}(0.6).$$

Equation (6) becomes

$$A \cdot V \geq B. \tag{10}$$

Since variables x_j and η in (8) are positive, values of variables that satisfy (10) can be obtained by the Simplex method [24]. The following example illustrates the procedure for transforming a simple supervised learning problem into a linear programming problem.

Example 1: Consider a single-layer neural network for solving a problem with P and D as follows:

$$P = \begin{bmatrix} -15 & -5 \\ -1 & 1 \\ -1 & -3 \end{bmatrix}$$

and

$$D = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}. \tag{11}$$

The ANN needed has two input units and one output unit, and is to be trained by the three patterns. The sigmoid function is assumed to have gain of one. From (7) and (10), we have

$$\hat{P} = \begin{bmatrix} -15 & -5 \\ -1 & 1 \\ -1 & -3 \end{bmatrix} \tag{12}$$

and a set of inequalities

$$\begin{bmatrix} -15 & -5 & 20 \\ -1 & 1 & 0 \\ -1 & -3 & 4 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ \eta \end{bmatrix} \geq \begin{bmatrix} 0.41 \\ 0.41 \\ 0.41 \end{bmatrix}$$
$$\text{where} \quad x_0, x_1, \eta \geq 0. \tag{13}$$

Solving them by the Simplex method leads to the following feasible solution:

$$[x_0 \quad x_1 \quad \eta]^T = [0.0 \quad 0.41 \quad 0.41]^T. \tag{14}$$

From (8), weight matrix W is, therefore

$$W = \begin{bmatrix} x_0 - \eta \\ x_1 - \eta \end{bmatrix} = \begin{bmatrix} -0.41 \\ 0.0 \end{bmatrix}. \quad (15)$$

We can verify the result by computing

$$D_{real} = S(P \cdot W) = [1.0 \quad 0.6 \quad 0.6]^T \quad (16)$$

where S is the sigmoid function with gain one. ■

In the above example, we assume the number of output units to be one; i.e., D is a column vector. If the number of output units is n (that is, the desired output matrix D is a $k \times n$ matrix), we can decompose the learning problem into n subproblems. In each subproblem, one of the column vectors of D is used to get a matrix $\hat{P}$ by applying (7). $\hat{P}$ is then used to get the corresponding column vector of weight matrix W.

We define, for a specific desired output matrix $D_{k\times 1} = [d_0 \quad d_1 \quad \cdots \quad d_{k-1}]^T$, R_D^k to be the set of all column vectors $[\hat{d}_0 \quad \cdots \quad \hat{d}_{k-1}]^T$, where

$$\begin{matrix} \hat{d}_i \geq S^{-1}(0.6) & \text{if } d_i = 1 \\ \hat{d}_i \leq S^{-1}(0.4) & \text{if } d_i = 0 \end{matrix} \quad \text{for all} \quad i = 0, 1, 2, \cdots, k-1. \quad (17)$$

Assume span $\{P\}$ to be the column-vector space consisting of all linear combinations of column vectors in P [11]. Since $P \cdot W \in \text{span}\{P\}$, a weight matrix W exists such that $S(P \cdot W) \approx D$ iff

$$R_D^k \cap \text{span}\{P\} \neq \emptyset. \quad (18)$$

Note that the linear programming approach described above attempts to find a weight matrix W that results in all correct output patterns. This is not possible when there is no W that satisfies (18) (i.e., when (10) does not have any feasible solution). In this case, (18) does not have a feasible solution, and the method breaks down. To overcome this problem, we relax the condition in (18) and attempt to find a set of weights such that the number of correct output patterns is maximized. That is, we find a set of values for all variables in order to maximize the number of inequalities that are satisfied. This relaxed objective, therefore, coincides with the objective of other supervised learning algorithms.

To obtain weights such that the number of correct output patterns is maximized, we can formulate the corresponding optimization problem as follows:

$$\text{Maximize} \sum_{i=0}^{k-1} u_0 \left(\sum_{j=0}^{m} a_{i,j} v_j - b_i \right) \quad (19)$$

where $a_{i,j}$, v_j, and b_i are defined in (9), and $u_0(x)$ is a step function with transition at zero.

The overhead for solving the above nonlinear optimization problem is very high and is, therefore, not practical. To reduce the overhead, we use a heuristic to obtain a proper set of weights when there is no feasible solution. The heuristic is similar to Phase I of the Simplex method [24]. First, we add a slack variable s_i to every inequality constraint, changing every inequality into an equality as follows:

$$\sum_{j=0}^{m} a_{i,j} v_j - s_i - b_i = 0$$
$$s_i \geq 0, \quad i = 0, \cdots, k-1. \quad (20)$$

Next, we attach an artificial variable z_i to each constraint equation, where

$$z_i = b_i + s_i - \left(\sum_{j=0}^{m} a_{i,j} v_j \right)$$
$$z_i \geq 0, \quad i = 0, \cdots k-1 \quad (21)$$

in such a way that the set of equalities in (21) always has a feasible solution. Note that a trivial feasible solution is to set all v_j's and s_i's to zero. We then solve the following linear optimization problem using the Simplex method:

$$\text{Minimize} \sum_{i=0}^{k-1} z_i$$
$$\text{such that } z_i = b_i + s_i - \left(\sum_{j=0}^{m} a_{i,j} v_j \right), \quad i = 0, \cdots, k-1$$
$$\text{and } v_j, \ s_i, \ z_i \geq 0. \quad (22)$$

Since we minimize $\sum z_i$, the optimal solution of (22) should have a small value for each z_i. Therefore, it is very likely that the inequality $z_i \leq s_i$ is satisfied, leading to a larger objective value defined in (19). This is true because if $z_i \leq s_i$, then

$$\sum_{j=0}^{m} a_{i,j} v_j = s_i + b_i - z_i \geq b_i. \quad (23)$$

Note that the larger the objective value (19), the more correct output patterns are obtained in the output layer.

B. The Algorithm

The objective of MM to reduce the number of training epochs can be achieved if the output matrix is flexible; that is, learning is faster if there is a large pool of desired output matrices, and learning can stop whenever any one of them is found. By exploiting this property, MM first transforms the original problem of finding a one-to-one mapping from P to D into one that finds a one-to-many mapping from P to one of a large set of possible output matrices I_{real}. It then transforms I_{real} to D by using the noniterative learning algorithms for single-layer ANN's described in Section III-A.

We use Fig. 1 to illustrate how MM works. Given an application problem with input matrix P and desired output matrix D, an existing supervised learning algorithm is used to train the original network. During training, a monitor is used to extract intermediate output matrix I_{real} periodically, and apply the linear programming method described in Section III-A to map I_{real} to D, where I_{real} is the set of output values of input and/or hidden units that are connected to the output units. An element $(I_{real})_{i,j}$ in matrix I_{real} is the output of

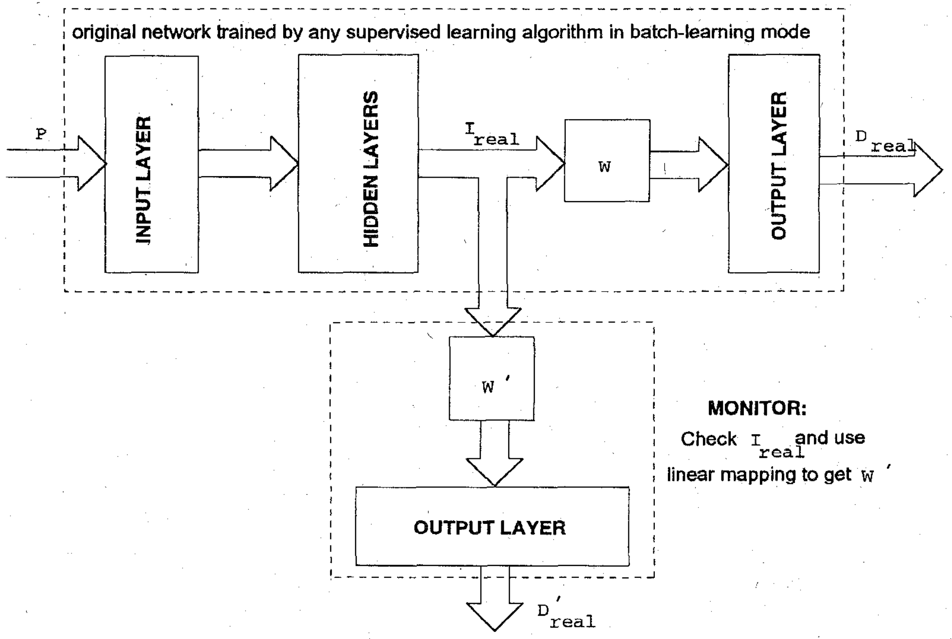

original network trained by any supervised learning algorithm in batch-learning mode
P
INPUT LAYER
HIDDEN LAYERS
I real
W
OUTPUT LAYER
D real
W '
MONITOR:
Check I real and use
linear mapping to get W '
OUTPUT LAYER
D' real

Fig. 1. Mixed-mode learning mechanism.

the jth unit that is connected to the output layer when the ith training pattern is applied. Note that the supervised learning algorithm used in Fig. 1 has to be in a batch learning mode, as I_{real} used by the monitor must correspond to a fixed set of weights. Patternwise learning cannot be used because the elements of I_{real} will change as training patterns are input.

MM requires a smaller number of training epochs since it has a relaxed objective. Learning in MM only tries to get an intermediate output matrix I_{real} that satisfies

$$R^k_{d_i} \cap \operatorname{span}\{I_{real}\} \neq \emptyset \quad \text{for all} \quad i = 0, 1, \cdots, n-1 \quad (24)$$

where d_i is the ith column vector of D. Since there are potentially many I_{real}'s that satisfy this criterion, learning involves finding one of the one-to-many mappings and is much easier. The additional overhead incurred involves finding a feasible solution of a set of linear inequalities.

C. Reduction in the Number of Hidden Units in Cascade Correlation

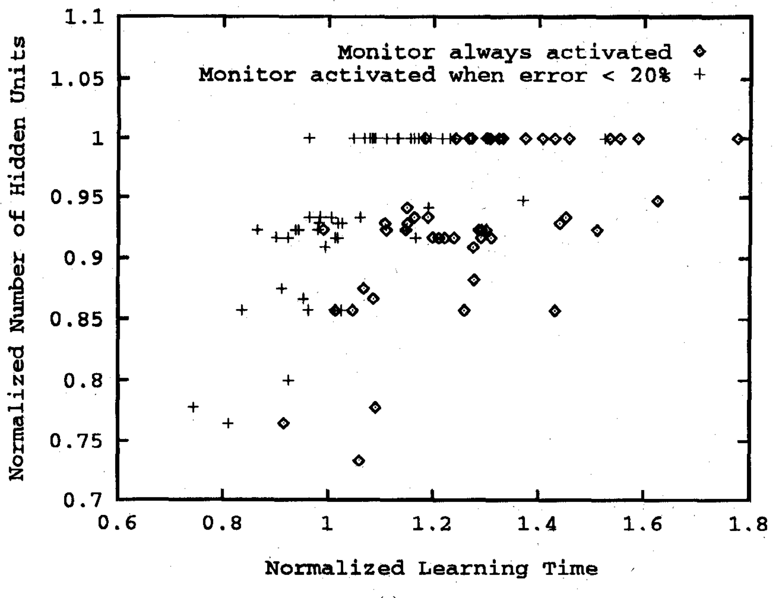
In CAS, the number of hidden units changes in a monotonically nondecreasing fashion as learning progresses. Hence, reducing the number of epochs by MM will lead to an equal or smaller number of hidden units in the resulting network.

As mentioned before, there are two training phases in CAS: *TRAIN_INPUT* for adding new hidden units, and *TRAIN_OUTPUT* for training the weights in the output layer. These two phases are executed alternately. If a monitor is added to CAS, we note that 1) I_{real} cannot be acquired in the *TRAIN_INPUT* phase, as the new hidden unit has not been decided upon, and 2) I_{real} is frozen in the *TRAIN_OUTPUT* phase. Consequently, we only have to use the monitor in the first epoch of each *TRAIN_OUTPUT* phase, resulting in a smaller overhead when MM is applied in CAS. The detailed procedure for applying MM in CAS is summarized as follows.

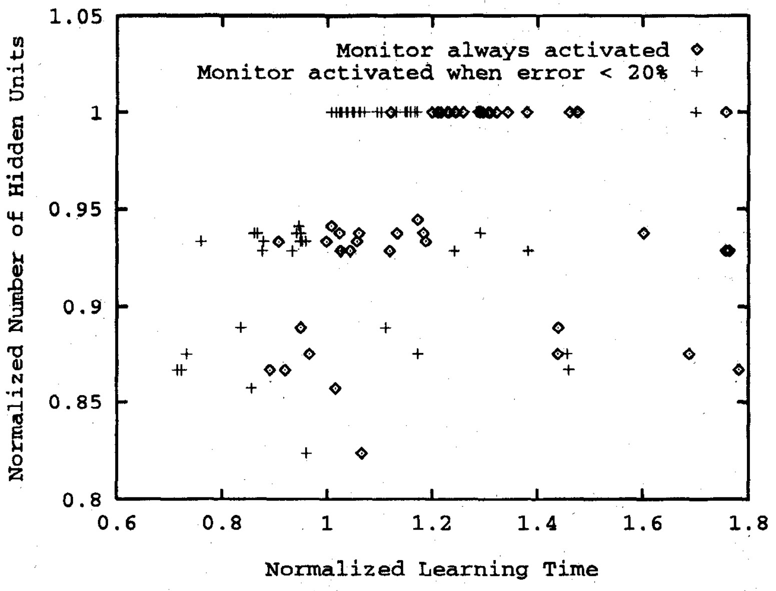
Procedure 1—Applying MM in CAS:

1) Train the initial network that has only the input units and the output units by quick-prop. If $\|D_{real} - D\|$ (see Fig. 1) is smaller than a prescribed threshold, then stop.
2) Execute the *TRAIN_INPUT* phase of CAS to add a new hidden unit.
3) Extract I_{real} from the network obtained in Step 2), where I_{real} is the input matrix to the output layer.
4) Using the method described in Section III-A, obtain the weights of connections to the output layer. If $\|D'_{real} - D\|$ is smaller than a prescribed threshold, then stop.
5) Execute the *TRAIN_OUTPUT* phase of CAS. If $\|D_{real} - D\|$ is smaller than a prescribed threshold, then stop; otherwise go to Step 2).

We compare the performance of the original CAS with that of CAS with MM (MM + CAS) using the two-spiral problem [10] and the Pima Indian Diabetes Diagnosis problem [25] as benchmarks. The task of the two-spiral problem is to classify two sets of training points that lie on two distinct spirals in a two-dimensional plane, in which the dimension of input vectors is two and the number of training patterns is 194. The task of the Pima Indian Diabetes Diagnosis problem is to decide whether a Pima Indian individual is diabetic or not, based on personal data (such as age and number of times pregnant) and the results of medical examination (such as blood pressure and glucose tolerance test). The original Pima Indian diabetes database came from the National Institute of Diabetes and Digestive and Kidney Diseases. Although this benchmark does not require the largest number of training epochs in the PROBEN1 benchmark suite [25], it is one of the problems that has the largest classification error after training



(a)



(b)

Fig. 2. Performance of MM + CAS normalized with respect to the performance of the "original" CAS. (a) The two-spiral problem and (b) the Pima Indian Diabetes Diagnosis problem.

is completed. The dimension of its input vectors is eight and the number of training patterns is 384.

The experiments were repeated 50 times, each with different initial weights. Learning stopped when the network could correctly predict at least 98% of the training patterns. Fig. 2 shows the plots of normalized number of hidden units versus normalized learning time. Each point (x, y) in this figure is normalized with respect to the performance of the "original" CAS using identical initial configuration and random weights where

$$x = \frac{\#\text{ of hidden units for ANN trained by MM + CAS}}{\#\text{ of hidden units for ANN trained by CAS}}$$

$$y = \frac{\text{learning time (CPU s) for ANN trained by MM + CAS}}{\text{learning time (CPU s) for ANN trained by CAS}}. \tag{25}$$

Using this normalization method, point (1, 1) in Fig. 2 represents the performance of the "original" CAS.

In our experiments, we have found that it is not necessary to activate the monitor early in the learning process, since it is difficult for the monitor to find an I_{real} to map to D at that time. In Fig. 2, points indicated by diamonds were obtained when the monitor was always activated. In contrast, points indicated by crosses were obtained for cases in which the

TABLE I
AVERAGE PERFORMANCE OF MM + CAS FOR SOLVING THE TWO-SPIRAL AND DIABETES PROBLEMS. THE EXPERIMENTS WERE RUN ON A SUN SS 10/51

	Monitor Activated When Error <20%				
Benchmark Problem	**Number of Cases**	**Avg. # of Hidden Units**	**Avg. Norm. # of Hidden Units**	**Avg. CPU Time (s)**	**Avg. Norm. CPU Time**
Two-Spiral	21	13.58	1	28.04	1.159
Benchmark	29	12.40	0.898	23.92	0.988
Diabetes	24	15.31	1	94.16	1.119
Benchmark	26	13.88	0.909	83.56	0.993

	Monitor Always Activated				
Benchmark Problem	**Number of Cases**	**Avg. # of Hidden Units**	**Avg. Norm. # of Hidden Units**	**Avg. CPU Time (s)**	**Avg. Norm. CPU Time**
Two-Spiral	18	13.50	1	33.74	1.395
Benchmark	32	12.09	0.896	21.67	1.215
Diabetes	23	15.01	1	113.51	1.348
Benchmark	27	13.70	0.910	100.85	1.199

monitor was activated when the error in the original network was smaller than 20%. We see that the latter results in savings in learning time. Note that the number of hidden units of networks trained by MM + CAS is never larger than that trained by the "original" CAS, since learning completes when the "original" CAS completes or when the monitor in MM + CAS finds a suitable mapping.

Finally, Table I summarizes the average performance of MM + CAS. For the two-spiral problem, when the monitor is activated whenever the error in training is less than 20%, 29 out of 50 cases lead to 10.2% reduction in the number of hidden units (as compared to the "original" CAS), taking only 98.8% of the training time. There are 21 cases in which we found no improvement in the number of hidden units; the penalty in these cases is an additional 15.9% training time. On the other hand, when the monitor is always activated, 32 cases lead to 10.4% reduction in the number of hidden units (as compared to the "original" CAS), but taking 121.5% of the training time. Similar results were also obtained for the diabetes benchmark (see Table I).

Our empirical results show that in most trials, MM + CAS converges with a lower number of hidden units, although learning time is slightly longer. This reduction in the number of hidden units is important, since a trained ANN may have to be applied many times when deployed.

IV. POPULATION-BASED LEARNING FOR DESIGNING ANN'S (PLAN)

PLAN is our second approach to provide to the ANN design system alternative configurations in the training process as well as progress information in training. The objective of PLAN is to find an ANN with a small number of hidden units within a time limit that is long enough to train tens to hundreds of networks. Instead of training each ANN to convergence, like in CAS and MM + CAS, PLAN maintains a pool of promising but partially trained ANN's and selectively trains those that are likely to have small configurations. PLAN is a population-based design system that schedules the training of ANN's dynamically, allocating time to promising ANN's that have not been given adequate training [30]. (It is not necessary to continue training ANN's with a large number of hidden units and those that have a small number of hidden units and have received adequate training.) PLAN involves the dynamic scheduling of computational resources and the identification of promising ANN's based on partial TSSE-versus-time information.

PLAN has a key difference on how reinforcement in learning is done with respect to Janson and Frenzel's genetics-based machine learning system for designing ANN's [19]. Janson and Frenzel computed the fitness of a partially trained ANN as a function of its sum of squared errors (SSE), and used the fitness to rank all the ANN's. Since the instantaneous SSE is not monotonic with respect to training time, and many other factors (such as the rate of change of SSE and the number of epochs trained) may affect the network size, we develop a comparator artificial neural network (CANN) that takes into consideration these factors. Using as inputs partial training behavior of two ANN's, our CANN predicts which ANN will lead to a smaller configuration when training by MM + CAS is completed. In training our CANN, we collect complete training-error behavior of a number of ANN's for a few applications, and use this behavior to adjust the weights of our CANN.

In this section, we first present PLAN and our method of training a CANN. Experimental results are shown in Section IV-C.

A. The PLAN System

Learning heuristics for solving an application problem can be classified as point-based and population-based. In point-based methods, only one heuristic method (HM) is considered at a time, and the learning system switches to a new HM after discarding the previous HM. In this context, an ANN is an HM for solving an application problem, and the MM + CAS training algorithm discussed in Section III is a point-based learning mechanism.

Population-based learning methods [31], on the other hand, maintain a population of competing HM's and choose to modify a specific HM depending on partial performance results of all HM's tested. In designing ANN's, a population-based method maintains a pool of ANN's at any time. Within the time allowed, it divides the time into quanta, selects and trains one promising ANN for a quantum using a point-based method (MM + CAS), updates the performance obtained at the end of the quantum, discards an existing ANN when it is found to be inferior, and generates new ANN's to replace discarded ones. Fig. 3 shows the structure of our integrated population-based and point-based learning system for designing ANN's.

Recall that CAS has two training phases: *TRAIN_INPUT* and *TRAIN_OUTPUT*. Here, we define one learning episode as one *TRAIN_INPUT* phase followed by one *TRAIN_OUTPUT* phase.

The minimum number of hidden units required and the corresponding ANN configuration for solving an application problem are kept in incumbent N_{incum} in the internal critic.

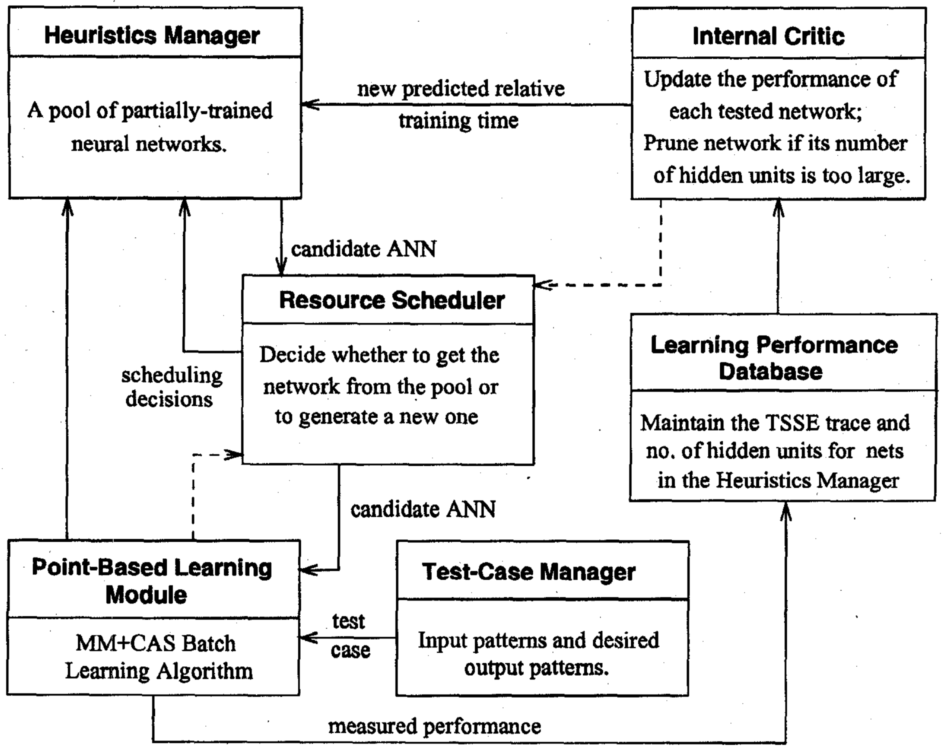

Heuristics Manager
A pool of partially-trained neural networks.
new predicted relative training time
Internal Critic
Update the performance of each tested network; Prune network if its number of hidden units is too large.
candidate ANN
Resource Scheduler
Decide whether to get the network from the pool or to generate a new one
scheduling decisions
Learning Performance Database
Maintain the TSSE trace and no. of hidden units for nets in the Heuristics Manager
candidate ANN
Point-Based Learning Module
MM+CAS Batch Learning Algorithm
test case
Test-Case Manager
Input patterns and desired output patterns.
measured performance

Fig. 3. PLAN system for designing ANN's. (Dashed arrows indicate completion signals sent to the resource scheduler.)

Initially, the system trains a feasible ANN using MM + CAS, and stores the number of hidden units needed in N_{incum}.

The heuristics manager in Fig. 3 is responsible for maintaining a fixed number of candidate ANN's, each partially trained, and for generating new candidate ANN's at the request of the resource scheduler. Initially, it generates a pool of candidate ANN's starting from a primary ANN (which has only the input and output layers) with random initial weights; this is followed by N_{incum}/k training episodes, where k is heuristically set as four. In our experiments, we have fixed the population size to be 25 at all times.

In the design process, the time allowed is divided into quanta. In our experiments, this is the time for training a selected ANN for one epoch. At the beginning of each quantum, the resource scheduler decides whether to generate a new candidate or to train an existing candidate for one episode. If an existing candidate is to be trained, then the resource scheduler chooses a candidate from the pool that requires the minimum predicted (relative) training time for convergence. We use a criterion based on the relative training time instead of the number of hidden units when convergence is reached because 1) the number of hidden units is monotonically nondecreasing with respect to training time, and 2) training time gives better granularity to differentiate between those ANN's that are promising and those that are not.

The candidate ANN chosen by the resource scheduler is trained by the point-based learning module for one training epoch. The training performance is then saved in the learning performance database which maintains the performance history of each candidate. Note that the learning performance of a candidate includes the number of hidden units and its temporal trace of TSSE.

The learning performance of the candidate trained in the last quantum is then evaluated by the internal critic, which is responsible for credit assignment that assigns credit/blame to the training result obtained. The internal critic predicts the relative convergence time of one candidate ANN with respect to another using the CANN discussed in Section IV-B. The prediction leads to the following alternative actions.

1) If the ANN selected has been trained to convergence and its number of hidden units is less than N_{incum}, then N_{incum} is updated. The resource scheduler then instructs the heuristics manager to generate a new candidate ANN, and schedules time to train the new ANN for $N_{incum}/4$ training episodes.
2) If the ANN has not been trained to convergence and the number of hidden units is smaller than $N_{incum} - 1$, then the internal critic will compute its new predicted relative convergence time using its CANN and updates the performance database. Note that if a nonconverged candidate ANN has $N_{incum} - 1$ hidden units, then this candidate will require at least N_{incum} hidden units when training converges, and hence can be discarded.
3) Otherwise, the candidate ANN is pruned, and the heuristics manager generates a new ANN.

The cycle of selection/training/generation is repeated until the time allowed is expended.

Our discussion in this section clearly identifies that the critical issue is to design a way to predict the convergence time of one partially trained ANN relative to another. This prediction is done by a CANN described in the next section.

B. Learning to Predict Relative Convergence Times Using a Comparator Neural Network

The time needed for convergence of a partially trained ANN is difficult to predict because it is an ill-defined function of many parameters, such as TSSE, slope of TSSE-versus-time trace, and epochs expended. Instead of predicting the actual convergence time, a more viable method is to 1) use relative convergence times rather than actual convergence times, and 2) develop a method to automatically learn this function based on actual behavior in training ANN's for a number of applications. In this section, we present the design of a CANN for implementing this function [32]. We assume that this function is primarily dependent on the behavior of TSSE versus time.

A simple method to predict the convergence time is to smooth (using a low-pass filter) the TSSE-versus-time trace (to remove high-frequency transients) and to extrapolate from the smoothed trace to see when it will reach the threshold TSSE. The difficulty with this approach is that there are many possible cutoff frequencies that can be used in the low-pass filter as well as many extrapolation methods. To overcome this difficulty, we develop a CANN whose inputs consist of TSSE traces from two partially trained ANN's (filtered by low-pass filters with different cutoff frequencies and extrapolated by different methods), and whose output predicts which of the two ANN's will converge faster. We have, therefore, simplified the original prediction problem to a problem of selecting good cutoff frequencies and extrapolation methods.

Specifically, we choose an application problem, train a number of ANN's using MM + CAS until convergence is reached, and record their TSSE traces. By taking segments of these traces (before convergence is reached), we then filter these segments using a set of low-pass filters with different cutoff frequencies, and extrapolate these filtered traces by different methods to obtain different predicted convergence times. These predicted convergence times, together with their actual convergence times, are then used to train the CANN, which predicts for any two partially trained ANN's which one will converge faster. Since we know the exact convergence times of these traces, errors in prediction can be used to update the weights of the CANN. In our current implementation, we use backpropagation (BP) to train the CANN.

Note that the training data of our CANN were derived from one application problem. To ensure that the CANN works for other applications, we need to normalize all the predicted convergence times before they are fed to the CANN. Our normalization procedure and its evaluation are presented in Section IV-C.

Our CANN architecture shown in Fig. 4 consists of two identical subnets: right subnet and left subnet. It is an extension of a CANN for predicting relative workload in a distributed computer system [20]. Given two input patterns, I_R and I_L, and the corresponding desired output patterns D_R and D_L, the objective of the CANN is to minimize the following error function:

$$\begin{aligned} E &= [O - O_D]^2 \\ &= [(O_L - O_R) - O_D]^2 \end{aligned} \tag{26}$$

where O_R and O_L are the actual outputs produced by I_R and I_L, respectively, and $O_D (= D_L - D_R)$ is the desired output difference. We minimize the error function using the following gradient-descent algorithm:

$$\begin{aligned} \frac{\partial E}{\partial O_L} &= \frac{\partial E}{\partial O}\frac{\partial O}{\partial O_L} \\ &= 2(O - O_D). \end{aligned} \tag{27}$$

The step size in our gradient-descent algorithm is set to η

$$\begin{aligned} \triangle O_L &= -\eta \frac{\partial E}{\partial O_L} \\ &= -2\eta(O - O_D) \end{aligned} \tag{28}$$

$$\begin{aligned} O_L^{teach} &= O_L + \triangle O_L \\ &= O_L - 2\eta(O_L - O_R - O_D). \end{aligned} \tag{29}$$

O_L^{teach} is then used to compute the error needed by BP to train the CANN. Note that a normal ANN training algorithm uses D_L rather than O_L^{teach} to compute the error. In deriving O_R^{teach}, we need to perform gradient ascent. This is

$$\begin{aligned} O_R^{teach} &= O_R + \triangle O_R \\ &= O_R - 2\eta(O_L + O_R - O_D). \end{aligned} \tag{30}$$

O_R^{teach} is then used to compute the error in the right subnet. To avoid any bias in the order of training patterns, we use I_L in the left subnet and I_R in the right subnet, and use I_L in the right subnet and I_R in the left subnet immediately in the next training instance.

C. Experimental Results

To implement our system for designing ANN's, we trained a CANN that can accurately predict, for any two partially trained candidate ANN's, which one would converge faster if both were trained to completion. We generated the training patters for this CANN as follows. First, we trained 15 ANN's using MM + CAS[1] for the two-spiral problem [10]. Let $T_{\max}$ and $T_{\min}$ be the maximum and minimum learning times, respectively, for these 15 ANN's. Second, we obtained 20 different segments from each TSSE-versus-time trace, the end of each segment being in the time interval $[0, t_i]$, where t_i was a random point in the trace. (These resulted in 300 different combinations.) We then filtered each segment by four different low-pass filters (the filters we used were seventh-order Butterworth filters [23] with cutoff frequencies Ω_c equal to 0.03π, 0.05π, 0.10π, and 0.15π, respectively) and extrapolated the smoothed trace using linear and exponential fitting methods in order to obtain eight predicted convergence times. These eight predicted times and the TSSE at the time when the segment was cut formed an input vector for our CANN. This input vector, when combined with the actual time needed for convergence, formed a training pattern.

[1]Any supervised learning algorithm besides MM + CAS can be used. However, the same learning algorithm should be used throughout PLAN.

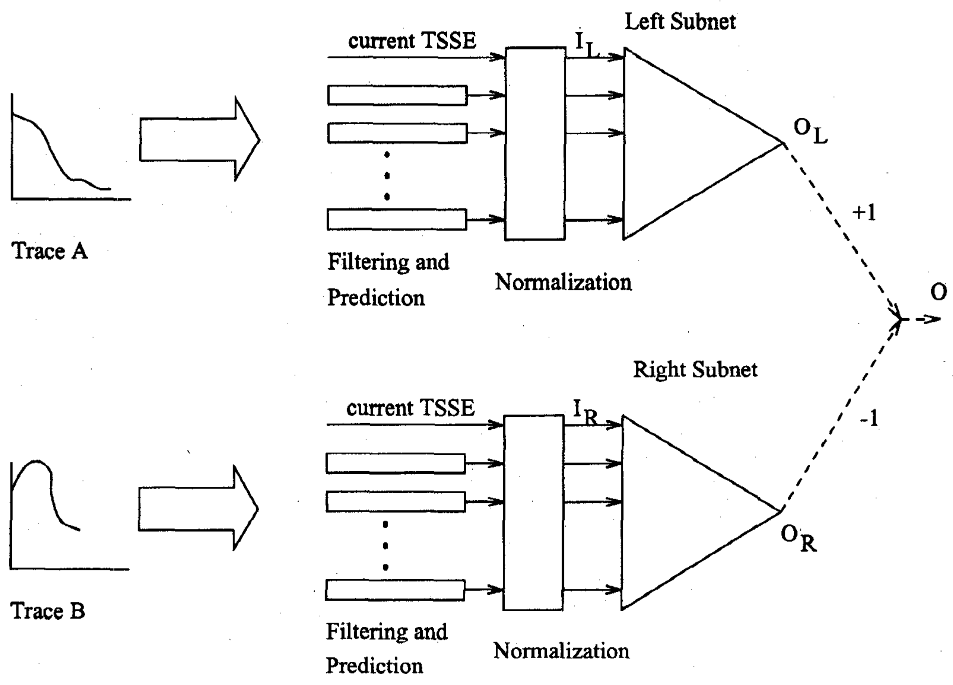


Fig. 4. Comparator neural-network (CANN) architecture.

The training times in these 300 training patterns were unnormalized. Since our CANN should be problem-independent in predicting relative training times, we normalize training times using the following equations:

$$\text{Normalized time} = \begin{cases} 1 & \text{if } T > T_{\max} \\ \dfrac{T - T_{\min}}{T_{\max} - T_{\min}} & \text{if } T_{\max} > T > T_{\min} \\ 0 & \text{if } T_{\min} > T \end{cases} \tag{31}$$

$$\text{Normalized TSSE} = \frac{\text{TSSE}}{\text{TSSE}_{\max}} \tag{32}$$

where T is the unnormalized training time and $\text{TSSE}_{\max}$ is the maximum TSSE in the 15 ANN's trained. Note that $T_{\min}$ and $T_{\max}$ are not the real minimum and maximum learning times. Rather, they represent the extreme learning times of ANN's in the pool observed so far in the learning process. For instance, when a network is trained to convergence and its learning time is larger than $T_{\max}$, then we update $T_{\max}$ to this new value. On the other hand, when a completely trained network has learning time smaller than $T_{\min}$, then we update $T_{\min}$. Using this normalization method, all learning times are normalized between zero and one. This method avoids choosing ahead of time constants to normalize the learning times. Note that without this normalization, the CANN we learn will be too dependent on the application we use in training it.

Using normalized training times, we then trained our CANN to differentiate between any two partially trained ANN which one would have a smaller (normalized) convergence time. In our experiments, our CANN has a configuration of 9-15-1 neurons in each subnet. We stopped training when we reached 80% accuracy. In general, training a CANN is very time-consuming and took about four days on a Sun SparcStation 10/51 workstation. Currently, we use plain BP to train CANN, although we can improve its learning speed by using other variants of BP [2], [5], [28], [34].

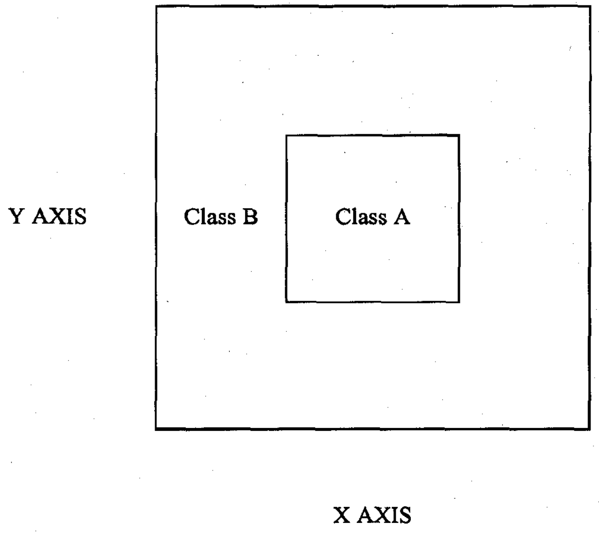
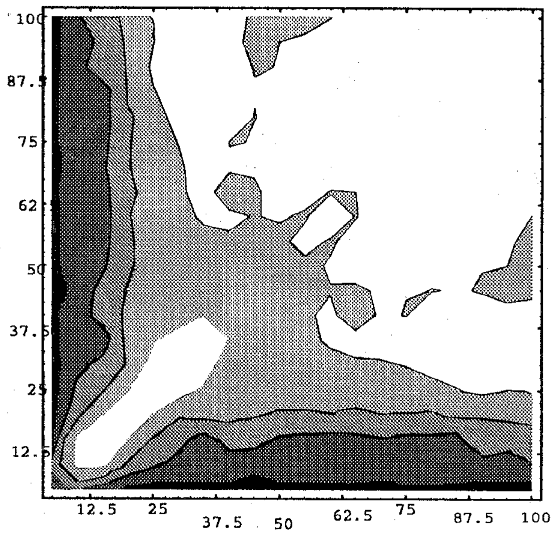
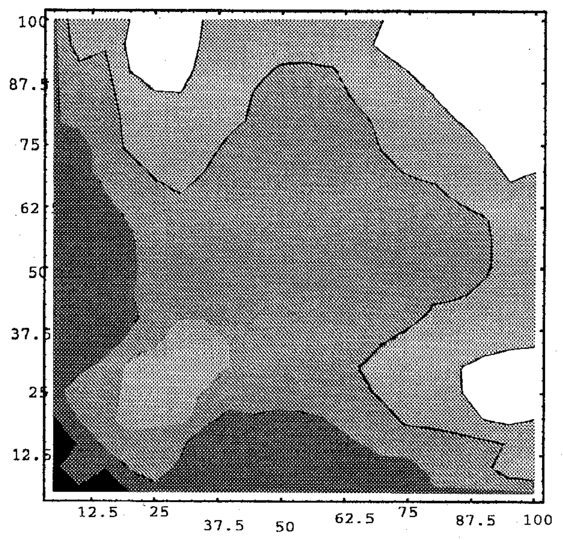


Fig. 5. A two-region classification problem.

As described before, the training patterns of our CANN were obtained from the two-spiral problem. To verify that our CANN can generalize to new applications, we tested it using the two-spiral problem as well as a simple two-region classification problem. In the classification problem, the goal is to identify which one of two regions (Fig. 5) to which a point belongs. The procedure of generating test patterns is the same as that of generating training patterns. Table II compares the accuracies of our CANN with those of simple prediction



(a)



(b)

Fig. 6. Average accuracies of our CANN in identifying one of the two ANN's to have shorter convergence time. The five grey levels in these graphs represent the following accuracy intervals: [0, 0.5), [0.5, 0.6), [0.6, 0.7), [0.7, 0.8), and [0.8, 1.0]. Further, darker colors mean lower accuracies: (a) two-spiral problem and (b) two-region classification problem.

TABLE II
COMPARING THE ACCURACIES OF CANN WITH THOSE OF SIMPLE PREDICTION METHODS

Application		2-Region Class.	2-Spiral Problem
Prediction Accuracies Using CANN		67.72%	72.88%
Linear Fitting	Seventh-order Butterworth Filter ($\Omega_c = 0.03\pi$)	57.32%	55.83%
	Seventh-order Butterworth Filter ($\Omega_c = 0.05\pi$)	56.43%	56.12%
	Seventh-order Butterworth Filter ($\Omega_c = 0.10\pi$)	61.38%	60.19%
	Seventh-order Butterworth Filter ($\Omega_c = 0.15\pi$)	58.33%	57.21%
Exponential Fitting	Seventh-order Butterworth Filter ($\Omega_c = 0.03\pi$)	59.21%	61.22%
	Seventh-order Butterworth Filter ($\Omega_c = 0.05\pi$)	57.15%	54.31%
	Seventh-order Butterworth Filter ($\Omega_c = 0.10\pi$)	59.54%	61.92%
	Seventh-order Butterworth Filter ($\Omega_c = 0.15\pi$)	57.98%	52.65%

methods based on filtering and extrapolation alone. We see that the accuracy of prediction is improved by using our CANN.

Fig. 6 shows two contour graphs that depict the accuracy of our CANN. A point (x, y) represents the case when $x\%$ of the TSSE trace of one ANN and $y\%$ of the TSSE trace of another ANN are fed into our CANN, and the corresponding z value is the average prediction accuracy of our CANN. These contour graphs indicate that our CANN is accurate when both ANN's have been trained to a reasonable amount, and is the least accurate when one or both of them have received very little training.

It is important to point out that our design system tries to avoid the dark regions in the contour graphs because it trains each candidate ANN by $N_{incum}/4$ learning episodes when the ANN is generated. Hence, all the input TSSE traces to our CANN in PLAN are from ANN's that have been trained for some time. The average accuracies of our CANN are, therefore, higher than those reported in Table II. For instance, the average accuracy in Fig. 6(b) is 71.59% when we consider only the part of the graph where $x > 20\%$ and $y > 20\%$.

Finally, we compare in Fig. 7 the average performance of 15 runs of our design system with that of the following naive design methods.

- *Naive Method.* This proposes one ANN at a time and trains it by MM + CAS until convergence is reached before switching to a new ANN.
- *Improved Naive Method.* This trains an ANN by MM + CAS either when convergence is reached or when the number of hidden units is larger than $N_{incum} - 1$. In the latter case, training should stop as there is already a better configuration with N_{incum} hidden units.

The interpretation of the graphs in Fig. 7 is as follows. When the total time allowed is small, the two naive methods are slightly better in finding ANN's with a lower number of hidden units. In this case, our design system takes more time to generate new candidate ANN's and does not have enough time to train any one to completion. In contrast, when the total time allowed is large, our design system is superior in identifying ANN's with a small number of hidden units. For instance, in the first graph in Fig. 7, our design system finds an ANN with an average of 10 hidden units using only 55% (respectively, 97%) of the time needed by the naive (respectively, improved naive) method to find a similar ANN. (The exact learning times are listed in Table III.) Similarly, in the second graph in Fig. 7, our design system takes 37 and 66%, respectively, of the average times needed by the two naive methods to obtain designs with an average of 5.5 hidden units. It is important to point out that even though the curves shown are close to each other, the parts we are interested in are the horizontal distances between two curves (indicating the difference in

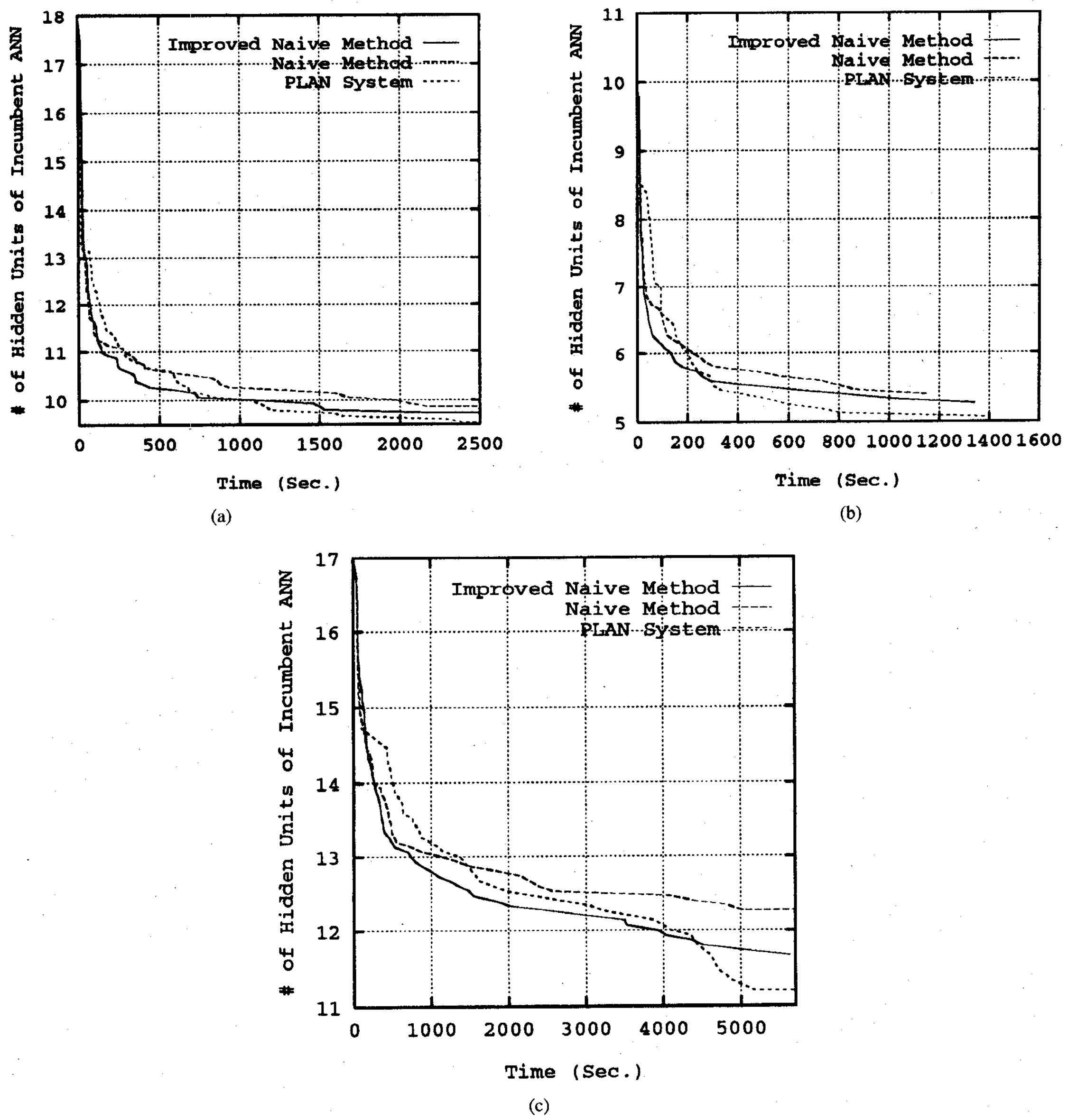


Fig. 7. Performance of the PLAN system: (a) two-spiral problem, (b) two-region classification problem, and (c) Pima Indian Diabetes Diagnosis problem.

training times). Since the curves flatten out when the time allowed is large, improvements due to our design system are more prominent in these cases.

We have also applied our design system to the Pima Indian Diabetes Diagnosis problem discussed in Section III. Since the behavior of this problem was not used in designing the CANN, it serves as a verification of our design system. Fig. 7(c) and Table III show that our system takes an average of 4680 CPU s to find designs with an average of 11.5 hidden units, whereas the two naive methods found designs that have an average of around 12 hidden units after 5700 CPU s. The experiments were not continued at this point due to limited computational resources because each point in the curve was averaged over 15 runs. However, we expect that the two naive methods will take a substantial amount of CPU time to find designs with an average of 11 hidden units.

TABLE III
AVERAGE PERFORMANCE OF PLAN AND THE NAIVE METHODS OVER 15 RUNS

Problem	Average # of Hidden Units Achieved	Learning Time (CPU s on Sun SS 10/51)		
		Naive	Improved Naive	PLAN
Two-spiral	10	1965.64	1107.73	1073.45
Two-region	5.5	803.57	454.30	298.32
Diabetes	11.5	>5700	>5700	4679.96

V. CONCLUSIONS

In this paper, we have presented two mechanisms for reducing the number of hidden units required for convergence in the cascade-correlation learning algorithm. The key idea of these mechanisms is to provide alternatives in the learning process and to select the best alternative dynamically based on information obtained. In general, our approach may lead to longer learning time, but the smaller networks found are faster when deployed in target applications and may generalize better.

The first mechanism, mixed-mode learning, is based on finding an intermediate output matrix that can be mapped to the desired output matrix. The validity of this mapping is verified by linear programming, which finds a feasible solution for a set of linear inequalities. Our experimental results show that MM + CAS leads to reduced number of hidden units, but increased computation time.

The second mechanism is PLAN, a learning system that aims to find a neural network with a small number of hidden units under a given time constraint. Here, we assume that the time allowed is long enough to completely train tens to hundreds of neural networks. PLAN uses a comparator-neural-network predictor that predicts, for two partially trained ANN's, which one will have a smaller convergence time. Based on the prediction, PLAN dynamically schedules partially trained ANN's for further training. Our experimental results demonstrate that PLAN can find smaller ANN's under similar time constraints as compared to those found by repeated applications of MM + CAS.

Although we have demonstrated our learning mechanisms with respect to the cascade correlation learning algorithm, the mechanisms can be used in conjunction with other learning algorithms that operate in batch-learning mode. These variations as well as the use of various activation functions will be investigated in the future.

Note Added in Proof: Since the writing of this paper, we have developed a new optimization method [35] for learning the weights of artificial neural networks. Our new approach formulates the learning problem as a continuous unconstrained nonlinear optimization problem that searches for the optimal assignment of weights to minimize the nonlinear TSSE function. By relying on an external force to lead the research trajectory out of local minima, we were able to find converged networks with four hidden units to solve the two-spiral problem. The methods presented in this paper, although worse than the results in our new approach, are still significant as they represent the best of what genetic algorithm-based design methods can offer.

References

[1] T. Ash, "Dynamic node creation in backpropagation networks," *Connection Sci.*, vol. 1, no. 4, pp. 365–375, 1989.

[2] H. S. M. Beigi and C. J. Li, "Learning algorithms for neural networks based on quasi-Newton methods with self-scaling," *J. Dynamic Syst., Meas., Contr., Trans. ASME,* vol. 115, pp. 38–43, Mar. 1993.

[3] Y. Q. Chen, D. W. Thomas, and M. S. Nixon, "Generating-shrinking algorithm for learning arbitrary classification," *Neural Networks,* vol. 7, no. 9, pp. 1477–1489, 1994.

[4] J. Diederich, "Connectionist recruitment learning," in *Proc. 8th Eur. Conf. Artificial Intelligence.* London: Pitman, 1988, pp. 351–356.

[5] S. Ergezinger and E. Thomsen, "An accelerated learning algorithm for multilayer perceptrons: Optimization layer by layer," *IEEE Trans. Neural Networks,* vol. 6, pp. 31–42, Jan. 1995.

[6] S. E. Fahlman, "Faster learning variations on back-propagation: An empirical study," in *Proc. Connectionist Models Summer School.* San Mateo, CA: Morgan Kaufmann, 1988, pp. 38–51.

[7] ——, "The recurrent cascade-correlation architecture," in *Advances in Neural Information Processing Systems 3,* R. Lippmann, J. Moody, and D. S. Touretzky, Eds. San Mateo, CA: Morgan Kaufmann, 1991, pp. 190–196.

[8] S. E. Fahlman and C. Lebiere, "The cascade-correlation learning architecture," in *Advances in Neural Information Processing Systems 2,* D. S. Touretzky, Ed. San Mateo, CA: Morgan Kaufmann, 1990, pp. 524–532.

[9] ——, "The cascade-correlation learning architecture," Carnegie Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-CS-90-10, Feb. 1990.

[10] ——, "The cascade-correlation learning architecture," in *Advances in Neural Information Processing Systems 2.* San Mateo, CA: Morgan Kaufmann, 1990, pp. 524–532.

[11] S. H. Friedberg, A. J. InSel, and L. E. Spence, *Linear Algebra.* Englewood Cliffs, NJ: Prentice-Hall, 1979.

[12] B. Fritzke, "Growing cell structures—A self-organizing network for unsupervised and supervised learning," *Neural Networks,* vol. 7, no. 9, pp. 1441–1460, 1994.

[13] C. L. Giles, D. Chen, G. Sun, H. Chen, Y. Lee, and M. W. Goudreau, "Constructive learning of recurrent neural networks: Limitations of recurrent casade correlation and a simple solution," *IEEE Trans. Neural Networks,* vol. 6, pp. 829–836, July 1995.

[14] B. Hassibi and D. G. Stork, "Second order derivatives for network pruning: Optimal brain surgeon," in *Advances in Neural Information Processing Systems 5,* S. Hanson, J. Cowan, and C. Giles, Eds. San Mateo, CA: Morgan Kaufmann, 1993.

[15] M. Heywood and P. Noakes, "A framework for improved training of Sigma-Pi networks," *IEEE Trans. Neural Networks,* vol. 6, pp. 893–903, July 1995.

[16] ——, "Directed product term selection in Sigma-Pi networks," in *Proc. IEEE Int. Conf. Neural Networks,* Orlando, FL, 1994.

[17] J. Hwang, S. Lay, M. Maechler, R. D. Martin, and J. Schimert, "Regression modeling in back-propagation and projection pursuit learning," *IEEE Trans. Neural Networks,* vol. 5, pp. 1–24, May 1994.

[18] J. Hwang, S. You, S. Lay, and I. Jou, "From a cascaded correlation learning network to a projection pursuit learning network," in *Int. Symp. Artificial Neural Networks,* National Chiao Tung Univ., Taiwan, 1993, pp. E11–E20.

[19] D. J. Janson and J. F. Frenzel, "Training product unit neural networks with genetic algorithms," *Expert: Special Issue Intell. Syst. Applicat.,* vol. 8, pp. 26–33, Oct. 1993.

[20] P. Mehra and B. W. Wah, "Adaptive load-balancing strategies for distributed systems," in *Proc. 2nd Int. Conf. Syst. Integration,* IEEE Computer Society, June 1992, pp. 666–675.

[21] M. C. Moze and P. Smolensky, "Using relevance to reduce network size automatically," *Connect. Sci.,* vol. 1, no. 1, pp. 3–16, 1989.

[22] T. M. Nabhan and A. Y. Zomaya, "Toward generating neural network structures for function approximation," *Neural Networks,* vol. 7, no. 1, pp. 89–99, 1994.

[23] A. V. Oppenheim and R. W. Schafer, *Digital Signal Proccessing.* Englewood Cliffs, NJ: Prentice-Hall, 1975, pp. 211–218.

[24] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity.* Englewood Cliffs, NJ: Prentice-Hall, 1982, pp. 55–58.

[25] L. Prechelt, "PROBEN1—A set of neural network benchmark problems and benchmarking rules," Univ. of Karlsruhe, Karlsruhe, Germany, Tech. Rep. 21/94, 1994.

[26] S. Santini and A. D. Bimbo, "Recurrent neural networks can be trained to be maximum *a posteriori* probability classifiers," *Neural Networks,* vol. 8, no. 1, pp. 25–29, 1995.

[27] S. Santini, A. D. Bimbo, and R. Jain, "Block-structured recurrent neural networks," *Neural Networks,* vol. 8, no. 1, pp. 135–147, 1995.

[28] R. S. Scalero and N. Tepedelenlioglu, "A fast new algorithm for training feedforward neural networks," *IEEE Trans. Signal Processing,* vol. 40, pp. 202–210, Jan. 1992.

[29] C.-C. Teng and B. W. Wah, "Mixed-mode learning: A method for reducing the number of hidden units in cascade correlation," in *Proc. Int. Symp. Artificial Neural Networks,* National Tsing Hua Univ., Hsinchu, Taiwan, Dec. 1993, pp. I.01–I.07.

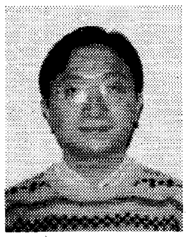
[30] ——, "An automated design system for finding the minimal configuration of a feed-forward neural network," in *Proc. Int. Conf. Neural Networks,* IEEE, vol. 3, pp. 1295–1300, June 1994.

[31] B. W. Wah, "Population-based learning: A new method for learning from examples under resource constraints," *IEEE Trans. Knowledge Data Eng.,* vol. 7, pp. 454–474, Oct. 1992.

[32] B. W. Wah, P. Mehra, and C.-C. Teng, "Comparator neural network for dynamic prediction," in *Proc. Int. Symp. Neural Networks,* National Cheng Kung Univ., Tainan, Taiwan, Dec. 1994, pp. 571–580.

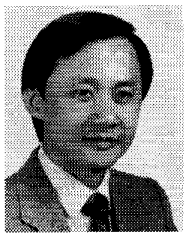
[33] N. Weymaere and J. Martens, "On the initialization and optimization of multilayer perceptrons," *IEEE Trans. Neural Networks,* vol. 5, pp. 738–751, Sept. 1994.

[34] X.-H. Yu, G.-A. Chen, and S.-X. Cheng, "Dynamic learning rate optimization of the backpropagation algorithm," *IEEE Trans. Neural Networks,* vol. 6, pp. 669–677, May 1995.
[35] Y. Shang and B. W. Wah, "Global optimization for neural-network training," *IEEE Comput.*, vol. 29, pp. 45–54, March 1996.



Chin-Chi Teng received the B.S. degree (1987) in electrical engineering from National Taiwan University, Taipei, Tiawan, and the M.S. degree (1993) in electrical and computer engineering from the University of Illinois at Urbana-Champaign. Currently, he is working toward the Ph.D. degree.

Since 1990, he has been a Research Assistant at the Coordinated Science Laboratory at the University of Illinois at Urbana-Champaign. He has had summer internships at Advanced Micro Devices, Inc., Sunnyvale, CA, and LSI Logic, Inc., Santa Clara, CA. His current research interests are in the area of computer-aided design of VLSI circuits and systems, the design for IC reliability, and neural networks.



Benjamin W. Wah (S'74–M'77–SM'85–F'91) received the Ph.D. degree in computer science from the University of California, Berkeley, in 1979.

He has served on the faculty of Purdue University, West Lafayette, IN (1979 to 1985), as a Program Director at the National Science Foundation (1988 to 1989), as Fujitsu Visiting Chair Professor of Intelligence Engineering, University of Tokyo, Japan (1992), and as the McKay Visiting Professor of Electrical Engineering and Computer Science, University of California, Berkeley (1994). He is currently a Professor in the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory of the University of Illinois at Urbana-Champaign. His current research interests are in the areas of parallel and distributed processing, knowledge engineering, and optimization.

Dr. Wah is Editor-in-Chief of the IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, and serves on the editorial boards of *Information Sciences*, the *International Journal on Artificial Intelligence Tools*, and the *Journal of VLSI Signal Processing*. He has chaired a number of conferences and is the General Chair of the 1996 International Conference on Neural Networks. He is currently serving in the IEEE Computer Society as a member of its Governing Board, Publications Board, and Press Activities Board, and as a Vice-Chair of its Fellows Committee. In 1989, he received a University Scholar Award from the University of Illinois.