AN AUTOMATED DESIGN SYSTEM FOR FINDING THE MINIMAL CONFIGURATION OF A FEED-FORWARD NEURAL NETWORK

Chin-Chi Teng and Benjamin W. Wah
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 West Main Street, Urbana, Illinois 61801, USA
wah@manip.crhc.uiuc.edu, (217) 333-3516

Abstract

In this paper, we present a method for finding the minimal configuration of a feed-forward artificial neural network (ANN) for solving a given application problem. We assume that the cascade-correlation (CAS) training algorithm is used to train the weights of the ANNs concerned. Under a given time constraint that is long enough to train tens of ANNs completely, we divide the time into quanta, and present a method for scheduling dynamically the ANN to be trained in each quantum from a pool of partially trained ANNs. Our goal is to find an ANN configuration with smaller number of hidden units as compared to the alternative of applying the CAS algorithm repeatedly to train each ANN to completion before exploring new ANNs. Our system is a population-based generate-and-test method that maintains a population of candidate ANNs, and that selectively train those that are predicted to require smaller configurations. Since it is difficult to predict the exact number of hidden units required when the CAS algorithm terminates, our system compares two partially trained ANNs 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) that takes as inputs the TSSE-versus-time behavior of training performed already on two ANNs, and that predicts which one will require a smaller number of hidden units when convergence is reached. We show that our CANN can predict correctly most of the time, and present experimental results on better configurations found in a given time limit for a classification problem and the two-spiral problem.

Research supported by National Science Foundation Grant MIP 92-18715 and Joint Services Electronics Program Contract JSEP N00014-90-J-1270.

Submitted to the 1994 IEEE International Conference on Neural Networks, June 1994.

1. Introduction

Among many artificial-neural-network (ANN) applications, *pattern classifiers* are considered very important and have been widely used in applications such as image analysis, image understanding, and speech understanding. A mechanism to train a pattern classifier is *supervised learning* in which the application problem is specified by a set of input and desired output patterns, and *training* involves finding the weights of an ANN that correctly maps each input pattern to the corresponding desired output pattern.

Many supervised learning algorithms have been developed in the past; among them, *back-propagation* is perhaps the most popular one. Convergence in traditional back-propagation algorithms may be slow. (In supervised learning, training converges when the TSSE — total-sum-of-squared errors — is smaller than a prescribed threshold.) To improve convergence time, Fahlman and Lebiere developed the *cascade-correlation training algorithm* (*CAS*) [1]. This algorithm starts with a minimal ANN that only has an input layer and an output layer, and 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. Experimental results have shown that CAS is faster than other algorithms.

One problem with the CAS algorithm is that it modifies the ANN configuration dynamically, and that the number of hidden units when the algorithm terminates is not bounded. Not bounding the number of hidden units in training may lead to overfitting of the weights with respect to the input-output patterns. In general, increasing the number of hidden units may increase the approximation quality of the ANN with respect to its training patterns, but not always improves its generalization behavior. Reducing the number of hidden units when convergence is reached is one way to improve the generalization behavior. An additional benefit of using smaller ANNs is that they are faster when deployed in applications.

0-7803-1901-X/94 $4.00 ©1994 IEEE

Mixed-mode learning [4] is a method we had developed to reduce the number of hidden units needed in the CAS algorithm. Our method transforms the CAS algorithm from one finding a unique one-to-one mapping to one finding one of the one-to-many mappings. Since the objective of learning is relaxed by this transformation, the number of learning epochs, and consequently, the number of hidden units, can be reduced significantly.

In this paper, we study the problem of reducing the number of hidden units of an ANN when trained by the CAS algorithm. Since there are infinitely many different configurations, and the time allowed for training is limited, we would like to schedule the time allowed so that only promising ANNs are trained. A *naive method* is to propose one configuration at a time and train it until convergence is reached. This does not utilize dynamic error information, which can be used to prune unpromising ANNs before they are trained to convergence. To maximize the number of different ANNs considered without having to train each to completion, we need to schedule time intelligently so that promising ANNs (those predicted to have smaller number of hidden units) are given more training time.

We propose in this paper a population-based learning system for designing ANNs for solving a given application problem. It aims at finding a feasible ANN with a small number of hidden units under a *time constraint* that is long enough to train tens of ANNs. Our system schedules the training of ANNs dynamically, allocating time to promising ANNs that have not been given adequate training. (It is not necessary to continue training for ANNs with a large number of hidden units, nor for ANNs that have a small number of hidden units and have received adequate training.)

Our design system has a key difference on how reinforcement is done with respect to Janson and Frenzel's *genetics-based machine learning system* for designing ANNs [2]. Janson and Frenzel compute the *fitness* of a partially trained ANN as a function of its sum-of-squared errors (SSEs), and use the fitness to rank all the ANNs. Since the instantaneous SSE is not monotonic with respect to training time, and many other factors, such as the rate of change of the SSE and the number of epochs trained, may indicate the network size, we develop a *comparator artificial neural network* (*CANN*) that takes into consideration these factors. Our CANN takes as inputs partial training behavior of two ANNs, and predicts which ANN will lead to a smaller configuration when training is completed. In training the CANN, we assume that we have determined ahead of time the complete training error behavior of a

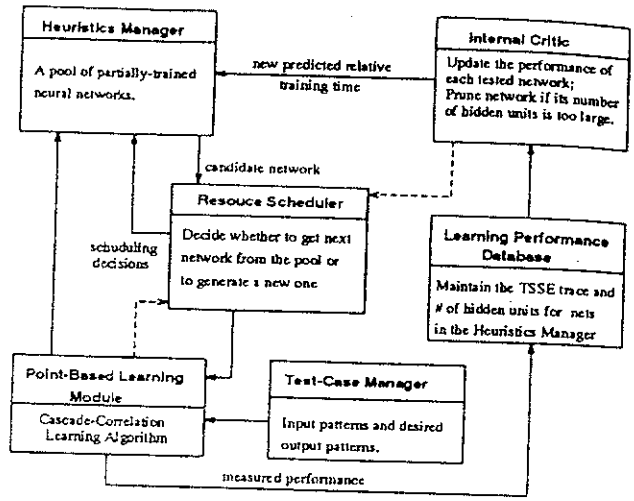

Figure 1. Population-based learning system for designing ANNs. (Dashed arrows indicate completion signals sent to the Resource Scheduler.)

number of ANNs for a given application. Consequently, errors in prediction can be used to adjust the weights of the CANN.

This paper is organized as follows. Section 2 presents our population-based learning system; Section 3, our method for training the CANN; and Section 4, our experimental results.

2. Population-Based Learning System for Designing Neural Networks

Learning of 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 paper, we consider an ANN as a HM for solving an application problem, and the CAS training algorithm as the point-based learning mechanism.

Population-based learning methods [5], on the other hand, maintain a population of completing HMs and choose to modify a specific HM depending on partial performance results of all HMs tested. In designing ANNs, a population-based method maintains a pool of ANNs at the same 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, updates performance obtained at the end of the quantum, generates new ANNs when none of the existing ANNs is promising, and discards an existing ANN when it is found to be inferior. In this paper, we use the

CAS algorithm as the point-based learning method. Figure 1 shows the structure of our integrated population-based and point-based learning system for designing ANNs.

There are two training phases in the CAS algorithm: the *TRAIN_INPUT* phase for adding new hidden units, and the *TRAIN_OUTPUT* phase for getting appropriate weights in the output layer. These two phases are applied alternately in the normal CAS algorithm. Here we define one *learning episode* as one TRAIN_INPUT phase followed by one TRAIN_OUTPUT phase. Note that the number of hidden units required when convergence is reached is unknown ahead of time.

The minimum number of hidden units required and the corresponding ANN configuration for solving the application problem are kept in the *incumbent* N_{incum} in the Internal Critic. Initially, the system trains a feasible ANN using the CAS algorithm, and stores the number of hidden units needed in N_{incum}.

The Heuristics Manager in Figure 1 is responsible for maintaining a fixed number of candidate ANNs, each partially trained, and for generating new candidate ANNs at the request of the Resource Scheduler. Initially, it generates a pool of candidate ANNs starting from a primary ANN (which includes 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 4.

In the design process, the time allowed is divided into quanta. 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 (a) there is a direct relationship between the training time and the number of hidden units needed, and (b) training time gives a better granularity to differentiate between promising and unpromising ANNs.

The candidate ANN chosen by the Resource Scheduler is trained by the Point-Based Learning Module for one training episode. The performance of training is then saved in the Learning Performance Database that maintains the history of performance for 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 a CANN to be is discussed in the next section. 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_{incumb} 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 the CANN and updates the Performance Database. Note that if a non-converged 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 pruned.

(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 candidate relative to another. This prediction is done by a CANN discussed in the next section.

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

The actual time needed for convergence for a partially-trained candidate 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 (a) use *relative* convergence times rather than actual convergence times, and (b) develop a method to automatically learn this function based on actual behavior in training ANNs for a number of applications. In this section, we present the design of a CANN for implementing this function. We assume that this function is primarily dependent on the behavior of TSSE versus time.

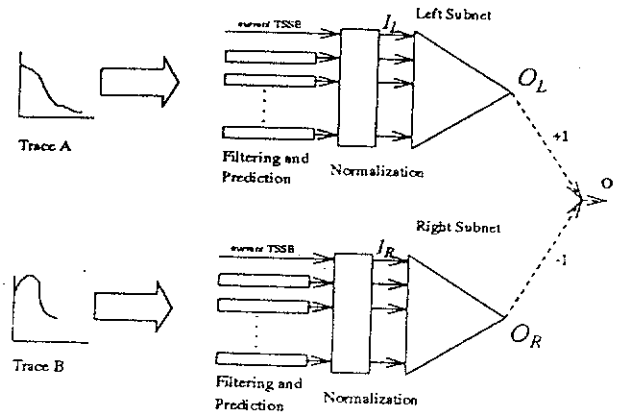

Figure 2. Comparator neural-network architecture.

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 reaches the threshold TSSE. The difficulty with this approach is that there are many possible cut-off 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 ANNs that are filtered by low-pass filters with different cut-off frequencies and extrapolated by different methods, and whose output predicts which of the two ANNs will converge faster. We have, therefore, simplified the original prediction problem to a selection problem.

Specifically, we choose an application problem, train a number of ANNs using the CAS algorithm 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 cut-off frequencies and extrapolate these filtered traces by different methods. These processed traces, together with their actual convergence times, are then used to train a CANN, which predicts for any two partially trained ANNs, 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.

The CANN developed here is used in the Heuristics Manager to order the set of partially trained ANNs so that the best ANN can be selected for training in the next quantum of time. The CANN, therefore, acts like a reinforcement unit in traditional machine learning methods.

Our CANN architecture shown in Figure 2 [3] consists of two identical subnets, right subnet and left subnet. 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.

$$E = [\,(O_L - O_R) - O_D]^2\,, \tag{1}$$

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. The error function is minimized using the following gradient-decent algorithm.

$$\frac{\partial E}{\partial O_L} = \frac{\partial E}{\partial O}\frac{\partial O}{\partial O_L} = 2(O - O_D), \tag{2}$$

where $O = O_L - O_R$. The step size in our gradient-decent algorithm is set to η.

$$\Delta O_L = -\eta\frac{\partial E}{\partial O_L} = -2\eta(O - O_D), \text{ and} \tag{3}$$

$$O_L^{teach} = O_L + \Delta O_L = O_L - 2\eta(O_L - O_R - O_D). \tag{4}$$

O_L^{teach} is then used to compute the error needed by back-propagation to train the CANN. Note that a normal ANN training algorithm uses D_L rather than O_L^{teach} to compute the error. Similarly, we obtain

$$O_R^{teach} = O_R + \Delta O_L = O_R + 2\eta(O_L - O_R - O_D). \tag{5}$$

O_R^{teach} is then used to compute the error to train the right subnet. To avoid 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.

4. Experimental Results

To implement our system for designing ANNs, we need to train a CANN that can accurately predict for any two partially trained candidate ANNs which one will converge faster if both are trained to completion.

We generate the training patters for this CANN as follows. First, we train 15 ANNs using the CAS algorithm for the two-spiral problem [1]. Let T_{max} and T_{min} be the maximum and minimum training times, respectively, for these 15 ANNs. Second, we obtain 20 different segments from each TSSE-versus-time trace, each in the time interval $[0, t_i]$, where t_i is a random point in the trace. (These result in 300 different combinations.) We then filter each segment by 4 different Butterworth filters and extrapolate the smoothed trace using linear and exponential fitting methods in order to obtain 8 predicted times needed for convergence. These 8 predicted times and the TSSE at the time when the segment is cut, form an input vector for the CANN. This input vector, when combined with the actual time needed for convergence, form a training pattern.

Since the training times in these 300 training patterns are unnormalized, and our CANN are trained to predict relative training times, we need to normalize the training times using the following equations.

Table 1. Comparing the accuracies of the CANN with those of simple prediction methods.

Application		2-Region Class.	2-Spiral Problem
CANN		67.72%	73.62%
Linear Fitting	Filter 1	57.32%	56.25%
	Filter 2	56.43%	52.14%
	Filter 3	61.38%	58.31%
	Filter 4	58.33%	53.22%
Exponential Fitting	Filter 1	59.21%	59.55%
	Filter 2	57.15%	53.23%
	Filter 3	59.54%	61.44%
	Filter 4	57.98%	53.29%

$$\text{Normalized time} = \begin{cases} 1 & \text{if } T > T_{max} \text{ or } T < 0 \\ \dfrac{(T - T_{min})}{(T_{max} - T_{min})} & \text{if } T_{max} > T > T_{min} \\ 0 & \text{if } T < T_{min} \end{cases} \quad (6)$$

$$\text{Normalized TSSE} = \frac{TSSE}{TSSE_{max}}, \quad (7)$$

where T is the unnormalized training time, and $TSSE_{max}$ is the maximum TSSE in the 15 ANNs trained.

Using the normalized training patterns, we then trained a CANN to differentiate between any two training patterns which one will have a smaller convergence time (see Figure 2). The configuration of each subnet in the CANN is 9-15-1. We stopped training when we reached 80% accuracy.

We test the CANN developed using a simple classification problem and the two-spiral problem. In the classification problem, the goal is to identify which one of the two regions (Figure 3) that a point belongs, whereas in the two-spiral problem, the goal is to identify which one of the two spirals that a point belongs. The procedure of generating test patterns is the same as that of generating training patterns. Table 1 compares the accuracies of the CANN with those of simple prediction methods based on filtering and extrapolation alone. We see that the accuracy of prediction is improved by using the CANN.

Figure 4 shows two contour graphs to depict the accuracy of the 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 the CANN, and the corresponding z value is the average accuracy that the CANN can predict the ANN with a shorter convergence time. The 5 grey levels in these graphs represent the following accuracy intervals: [0, 0.525), [0.525, 0.621), [0.621, 0.717), [0.717, 0.813), and [0.813, 1.0], and darker colors mean lower accuracies. These contour graphs indicate that the CANN is

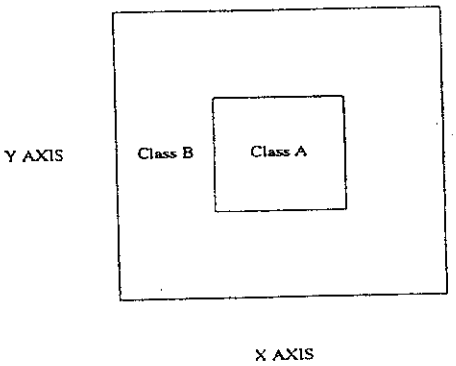

Figure 3. Two-region classification problem.

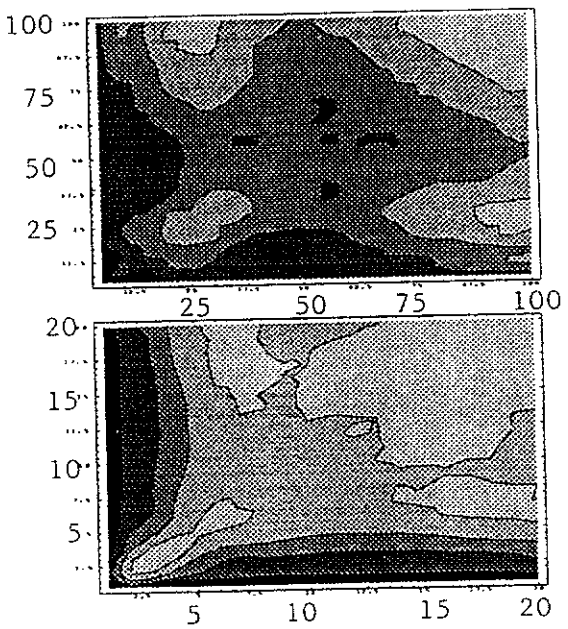

Figure 4. Average accuracies of the CANN in identifying which one of two ANNs has shorter convergence time. (The first figure is for the two-region classification problem, and the second is for the two-spiral problem.)

accurate when both ANNs have been trained to a reasonable amount, and is the least accurate when one or both ANNs 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 first generated. Hence, in our design system, all the input TSSE traces to the CANN are from ANNs that have been trained for some time. The average accuracies of the CANN are, therefore, higher than those reported in Table 1. For instance, the average accuracy in the first contour graph in Figure 4 is 71.59% when we consider only the part

of the graph where $x > 20\%$ and $y > 20\%$.

Finally, we compare in Figure 5 the average performance of our design system with those of two other methods. The first method is the naive method stated in Section 1, and the second method, called improved naive method, is similar to the first method except that it stops training when the number of hidden units is larger than $N_{incum}-1$. Each curve in Figure 5 represents an average of 15 runs.

The interpretation of the graphs in Figure 5 is as follows. When the total time allowed is small, the two naive methods are slightly better in finding ANNs with smaller number of hidden units. In this case, our design system spends more time to generate new candidate ANNs and does not have enough time to train any ANN to completion. In contrast, when the time allowed is large, our design system is superior in identifying ANNs with small number of hidden units. For instance, in the first graph in Figure 5, our design system finds an ANN with an average of 5.5 hidden units using only 21% of the time needed by the naive method to find a similar ANN, and 57% of the time needed by the improved naive method. Similarly, in the second graph in Figure 5, our design time takes 18% and 55%, respectively, of the average times needed by the naive and improved naive methods to obtain an ANN with 10.5 hidden units. It is important to note that the improvement is even greater when the time allowed is large.

References

[1] S. E. Fahlman and Christian Lebiere, The Cascade-Correlation Learning Architecture (CMU-CS-90-100), School of Computer Science, Carnegie Mellon Univ., Pittsburgh, PA, Feb. 1990.

[2] D. J. Janson and J. F. Frenzel, "Training Product Unit Neural Networks with Genetic Algorithms," *IEEE Expert: Special Issue on Intelligent Systems and their Applications*, vol. 8, no. 5, pp. 26-33, Oct. 1993.

[3] P. Mehra and B. W. Wah, "Adaptive Load-Balancing Strategies for Distributed Systems," *Proc. 2nd Int'l Conf. on Systems Integration*, pp. 666-675, IEEE Computer Society, Morristown, NJ, June 1992.

[4] C.-C. Teng and B. W. Wah, "Mixed-Mode Learning: A Method for Reducing the Number of Hidden Units in Cascade Correlation," *Proc. Int'l Symposium on Artificial Neural Networks*, National Chiao Tung University, Hsinchu, Taiwan, (accepted to appear) Dec. 1993.

[5] B. W. Wah, "Population-Based Learning: A New Method for Learning from Examples under Resource Constraints," *Trans. on Knowledge and Data Engineering*, vol. 4, no. 5, pp. 454-474, IEEE, Oct. 1992.

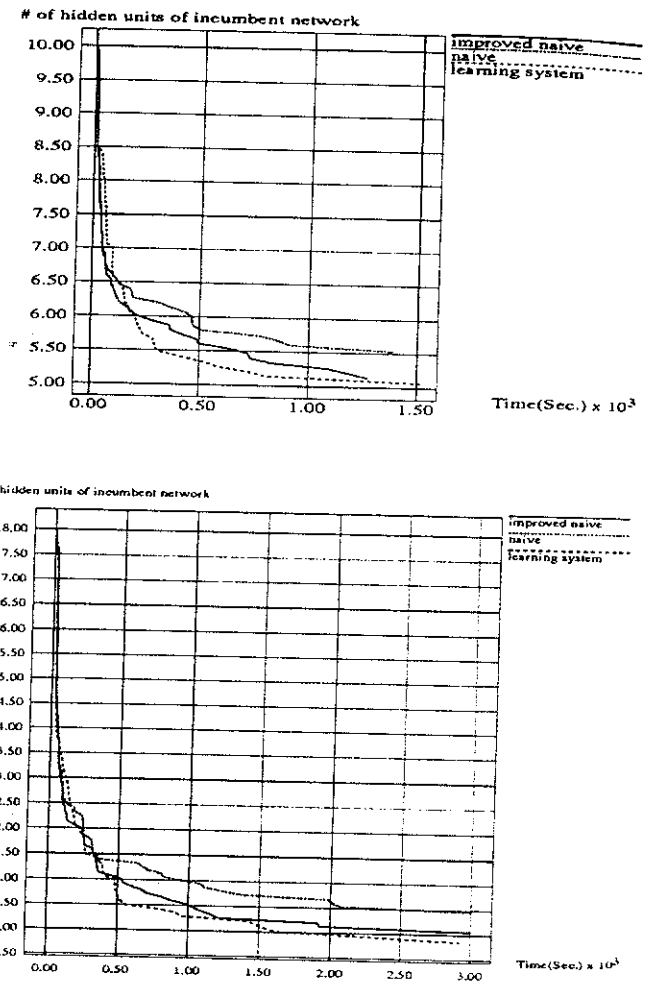

Figure 5. Performance of population-based learning. (The first figure is for the two-region classification problem, and the second is for the two-spiral problem.)