USING NEURAL NETWORKS AND DYNA ALGORITHM FOR INTEGRATED PLANNING, REACTING AND LEARNING IN SYSTEMS

NAGW-1333
IN-63-CR
161922
P.27

by

Pedro Lima and Randal Beard

Rensselaer Polytechnic Institute
Electrical, Computer, and Systems Engineering Department
Troy, New York 12180-3590

August 1992

CIRSSE REPORT #122

Using Neural Networks and Dyna Algorithm for Integrated Planning, Reacting and Learning in Robotic Systems

Pedro Lima and Randal Beard

June 26, 1992

1 Introduction and Statement of the Problem

The traditional AI answer to the decision making problem for a robot is *planning*. However, planning is usually CPU-time consuming dependent on the availability and accuracy of a world model.

The Dyna system generally described in [1], uses trial and error to learns a world model which is simultaneously used to plan reactions resulting in optimal action sequences. It is an attempt to integrate planning, reactive and learning systems.

The architecture of Dyna is presented in figure 1. The different blocks are described in the following. For details, see [1].

There are three main components of the system. The first is the **world model** used by the robot for internal world representation. The input of the world model is the current state and the action taken in the current state. The output is the corresponding reward and resulting state.

The second module in the system is the **policy**. The policy observes the current state and outputs the action to be executed by the robot. At the beginning of program execution the policy is stochastic and through learning progressively becomes deterministic. The policy decides upon an action according to the output of an **evaluation function** which is the third module of the system.

The evaluation function takes as input, the current state of the system, the action taken in that state, the resulting state, and a reward generated by the world which is proportional to the current distance from the goal state.

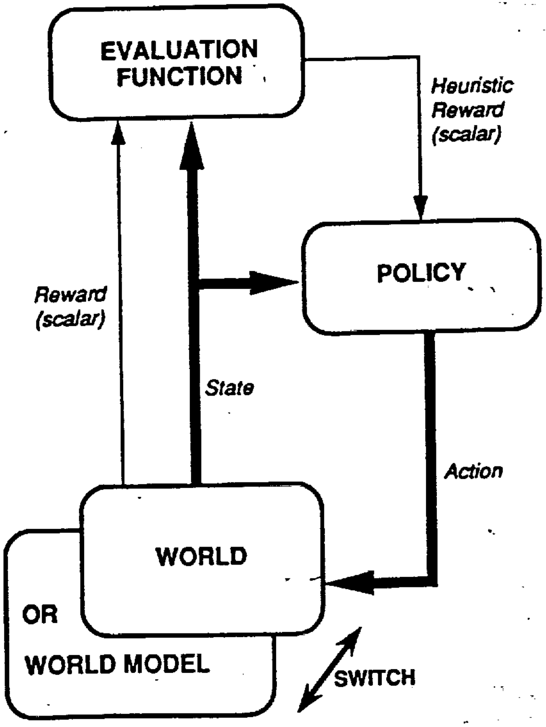


Figure 1: Dyna architecture. Reprinted from [1]

A slightly different version of this approach is the reinforcement learning method called *Q-learning* [2]. At each discrete time step $k = 1, 2, \cdots$ the controller (a mix of evaluation function plus policy) observes the state x_k of the world, selects action a_k, receives a reward r_k and observes the resultant state x_{k+1}. The objective is to maximize the expected discounted sum of future reward

$$E[\sum_{j=0}^{\infty} \gamma^j r_{k+j}] \ , \ 0 \leq \gamma \leq 1$$

The recursive relationship

$$Q(x, a) = E[r_k + \gamma \max_b Q(x_{k+1}, b) | x_k = x, a_k = a]$$

is satisfied by a function Q which is the expected discounted sum of future reward for performing action a in state x and performing optimally thereafter. An optimal control rule can be expressed in terms of Q. At each step the optimal action for state x is selected as the action a that maximizes $Q(x, a)$. Thus, given the observations x_k, a_k, r_k and x_{k+1}, the estimate $\hat{Q}$ of Q is updated at time step k:

$$\hat{Q}(x_k, a_k) = \hat{Q}(x_k, a_k) + \beta_k[r_k + \gamma \max_b \hat{Q}(x_{k+1}, b) - \hat{Q}(x_k, a_k)] \qquad (1)$$

If β_k is a gain sequence such that

$$0 < \beta_k < 1$$

$$\sum_{k=1}^{\infty} \beta_k = \infty$$

$$\sum_{k=1}^{\infty} \beta_k^2 < \infty$$

and all actions continue to be tried from all states

$$P(\lim_{k \to \infty} \hat{Q}_k = Q) = 1$$

Deterministically selecting the action that maximizes $\hat{Q}$ is analogous to a steepest descent search, and is prone to getting stuck in local minima. To correct this problem, the next action is chosen randomly where the probability of choosing action a in state x, $P(a|x)$ is a function of $\hat{Q}(x, a)$. Using this scheme the *estimated* best action has the greatest probability of being chosen, however the randomness allows the system to escape local minimum that will eventually be *smoothed* through learning. As time progresses these local minima are eliminated from Q, allowing the selection process to become increasingly deterministic.

The Q-learning method was actually used in our implementation instead of the policy and evaluation functions of the Dyna algorithm. The modified algorithm is

1. Decide if this is a real *experience* or a *hypothetical* experience. For each real experience, do N hypothetical experiences;

2. Pick a state x_k. If this is a real experience, use the *current state*. Otherwise, use a *random state*.

3. From the value of $\hat{Q}(x_k, a_{i_k})$, $\forall_i$ choose randomly action a for state x_k, using a Boltzmann Distribution

$$P(a_i|x) = \frac{\exp(\hat{Q}(x, a_i))}{\sum_j \exp(\hat{Q}(x, a_j))}$$

4. Do action a. From world (real experience) or world model (hypothetical experience) obtain next state x_{k+1} and reward r ;

5. Update $\hat{Q}(x, a_i)$ using (1);

6. Go to step 1.

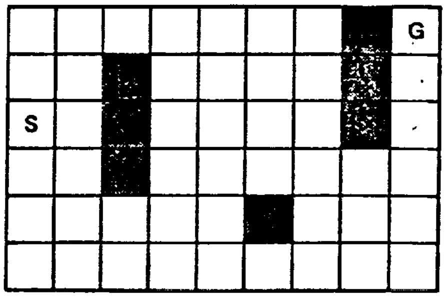


Figure 2: Simple world for a navigation problem. Reprinted from [1]

2 Proposed Work

Originally, the work proposed was:

1. to implement the simple 2-D world described in [1] where a "robot" is navigating around obstacles to learn the path to a goal, by using lookup tables as described in the paper. The purpose of this step was to demonstrate the convergence properties of the algorithm.

2. to substitute the world model and Q estimate function $\dot{Q}$ by neural networks.

3. to apply the algorithm to a more complex world where the use of a neural network would be fully justified. In a complex world the complete set of state/action pairs becomes prohibitively large and renders a lookup table method impractical. A neural network should be able to generalize to state/action pairs which have not yet been encountered by the robot, this property is very desirable in the planning stage of the algorithm.

In the next two sections, the system design and achieved results will be described. First we implement the world model with a neural network and leave Q implemented as a look up table. Next, we use a lookup table for the world model and implement the Q function with a neural net. Time limitations prevented the combination of these two approaches. The final section discusses the results and gives clues for future work.

3 Results Using Lookup Tables for the World Model and Q function

The problem consists of a 2-D world with obstacles where a robot is located at a starting point and must reach a goal state G with no *a priori* knowledge of the world or the goal. The robot receives a reward $r = 1$ when it reaches G, and zero reward otherwise.

The world can be modeled as a finite automaton. The Primitive Actions of the robot are move commands. The robot can move UP, DOWN, RIGHT, or LEFT, within the confines of the world shown in figure 2.

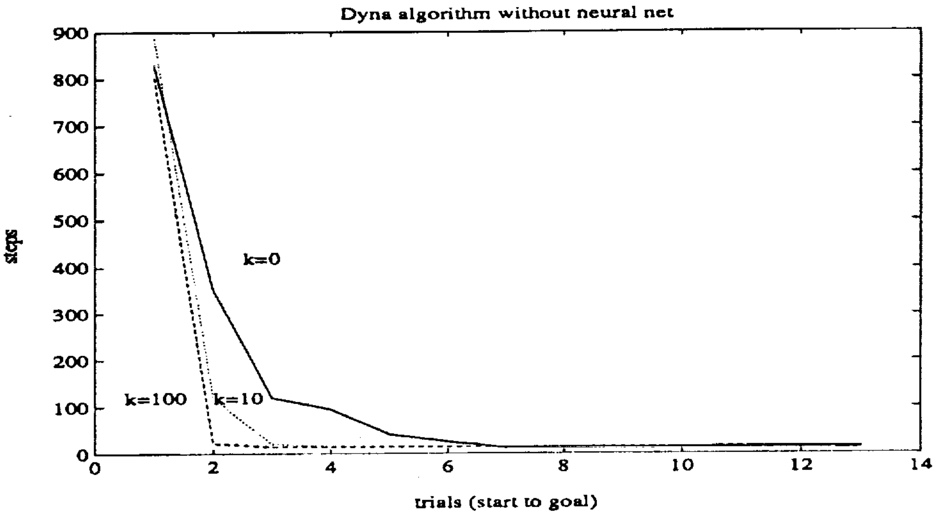


Figure 3: Results of Dyna Algorithm Using Lookup Tables

Given the low dimensionality of the problem, the world model and Q function were implemented as lookup tables. After an initial random walk first trial (until a reward $r = 1$ is received), the algorithm learns very quickly short paths to the goal. However, the shortest path is not always obtained, which is a result of the system getting stuck in local minima. Both the random and deterministic cases were successfully tested, as reported in figure 3.

4 Implementing the World Model by a Neural Network

In the above sections we have shown that when the system is given a perfect world model, it can learn an optimal path to the goal on the second trial. While this result is encouraging, the assumption of an exact world model is a very strong condition that most likely will not be satisfied by actual systems. Therefore the next step of the project was to implement the world model with a neural network.

The original system which is described above is modified in the following way. While the robot is moving around in the world during an actual trial, each state, action, next state tuple is stored to a file. When the robot has found the goal state, the trial ends. Before planning begins, the system recalls each state, action, next state tuple and after preprocessing uses this tuple as a training sample to the network. The test samples are shown to the network cyclically until some error criteria is met. Once the world model has been trained, the planning routine is executed. The planning routine, sends the network a tuple specifying the current state and action that is to be executed in that state. The network returns the next state and the reward associated with that state. It has been found (by experimentation) that it is sufficient to show each training sample to the network once. This has significance for on line implementation. The final algorithm essentially learns the world model on line.

There are several advantages to using a neural network for the world model. The first advantage is that it is highly improbable that all of the possible state-action pairs will be tried during the first trial run in the world. A system which explicitly stores state, action, next state tuples, and uses these tuples for the planning phase, would not be able to generalize to state-action pairs which had not been experienced by the automaton. However for planning to be successful, the result of all state-action pairs needs to be available to the routine. The generalization capabilities of neural networks are ideal because they can induce the result of a state-action pair that has not been shown to the network in its training set. Another advantage to using a neural network to approximate the world model, is that the a neural network is relatively insensitive to noisy data collected from the world. Another advantage is that a neural network should be able to model a world which is stochastic. We have designed experiments that demonstrate the ability of neural networks to perform adequately in the presence of data noise, and in the case of a stochastic world. These experiments will be described later in this section.

4.1 World Model Representation

The first issue in the design of a world model was how to represent the world with a neural network. The specifications for the world model are as follows. The world model is sent the x, y coordinates of the current location of the robot, and the action that the robot takes in that state. The world model returns the next state of the system and the reward associated with the action taken in the previous state. For this project a positive reward was given when the robot entered into the goal state, and a negative reward was given when the robot moved into an obstacle or into a wall. Four different approaches to the world model were tried and analyzed. Unfortunately data regarding the first two approaches was not saved and therefore is not available.

The first representation is to use three analog input units and three analog output units. The first two input units represent the x and y coordinates of the current position occupied by the robot. The third input unit is the action taken by the robot in the current state. The first two output units represent the x and y coordinates of the robot after executing the action. And the third output unit represents the value of the reward given to that action. Linear transfer functions were used at the output, and sigmoidal transfer functions were used at the hidden layer.

The second representation uses nine binary input units and nine binary output units. The meaning of the units is similar to the first representation except that the x, y coordinates and action are converted to binary. The presence of a binary reward is represented by the eighth output unit, and the presence of a negative reward is represented by the ninth output unit. Sigmoidal transfer functions were used for all units.

The third and fourth representations take advantage of the fact the world can be represented by a finite automata. For any given state-action pair the system will do one of two things; 1) it moves according to action, 2) it does not move due to the presence of a wall or barrier. The reward is similar; when the system moves, it either receives a reward (corresponding to the goal) or it does not. If the system does not move, it always receives a negative reward. Therefore the automata can be represented with two binary output units. The first indicates whether the robot moves or not, and the second signals the presence of a positive reward. The third representation uses binary inputs, similar to the second representation. The fourth uses analog inputs similar to the first representation. Sigmoidal transfer functions were used for all units in these representations

The third and fourth representations were analyzed and the training and test set error are plotted in figure 4.1. The plot labeled "Network 1" corresponds to the third representation, and the plot labeled "Network 2" corresponds to the forth representation. As can be seen from figure 4.1, training error is smaller for binary inputs. Surprisingly however, the test error is similar for both cases and is actually worse for binary inputs as the training set is repeatedly shown to the network. Test sample error is the most important evaluation of the performance of the network, therefore we decided to go with the forth representation. This representation also offers the advantage that it requires the fewest units and thus optimizes training and recall speeds.

Due to the reduced size of the output layer,the third and fourth representation are clearly better than the first and second. It is reasonable to expect that the performance of the first and second will be similar to the forth and third respectively. Therefore we feel that we are justified in choosing the forth representation over the first and second, even though we have not explicitly compared them experimentally.

4.2 Network Design

In this subsection, we will describe the design of the network. The following network parameters must be specified.

1. The number of hidden layers.
2. The number of hidden units.

3. The step size ETA.

4. The momentum gain MOM.

All parameters will be chosen according to experimental results. Network performance verses sample size is also considered. In our system, the number of samples shown to the network is determined by the (initially random) walk of robot. The number of samples has ranged from 120 to 3360. It will be shown later that the system is self correcting with respect to sample size.

The first tests that were performed were with the number of hidden layers. The squared error of randomly generated training and test samples were compared for one and two hidden layers. The result of these tests was that the training error decreased slightly for two hidden layers, but the test error increases. Also, the training time of the network with two hidden layers was significantly longer. Based on these results we decided that one hidden layer was sufficient for this project.

In order to analyze the effects of the four parameters listed above the following experiment was designed. A routine initially generates a random state. The robot, then randomly wanders around the world generating a specified number of samples. The number of samples generated is indicated in figures 4.2 - 4.9, by the field SAMPLES. The test set was generated in exactly the same fashion. Therefore the test set may contain samples in the training set, but will also contain many samples that are not present in the training set. (Although, most neural network application require that the training and test samples are disjoint, it was decided that because the actual usage of the network will be in exactly an analogous fashion, that this type of analysis would be more informative.) The number of test samples was always 500. Most cases shown in figures 4.2 - 4.9 have 100 training samples.

Figures 4.2 - 4.9 show plots of four types of error. "Train err", and "test err" are the squared error on the training set and the test set respectively. "train miss" and "test miss" are the percentage of the training set and the test set that are incorrect after the output unit has been rounded to plus or minus 1. "Train miss," and "test miss" give a better indication of the error that will be seen by our particular system.

There are comparison graphs which compare training and test error, for each of the cases considered. These figures will not give exact quantitatively accurate comparisons of the different cases. However, qualitatively the general behavior of plot should be accurate.

Number of Hidden Units. Figures 4.2 - 4.3 show error plots for varying hidden units. Hidden units of 3, 5, 10 and 30 were examined. Figure 4.4 directly compares the squared error for varying hidden units. In all cases, the training error is monotonically decreasing as expected. The results with the test error are very interesting. While 5, and 10 hidden units result in test error which is decreasing. 3 and 30 hidden units show test error which gets worse after a large number of training cycles. It was explained in class that we should expect this behavior if our training set had a noise component, since the network would eventually attempt to fit the noise. In this case, however, we do not have noise in our training set. One possible explanation of this behavior is that after a certain number of training cycles the value of the weights becomes so large that saturation begins to hinder generalization. In effect this is the same phenomena as the problem with noise. The the network learns to classify the training set, but progressively performs worse on any sample that is not in the training set. Figure 4.4 indicate that 3 hidden units is not sufficient and that relatively little improvement is observed by increasing the number of hidden units from 5 to 10. Therefore the number of hidden units used in the project was chosen to be 5.

Step Size ETA Figures 4.5 - 4.7 show error plots for variations in the step size ETA. As expected, the training error decreases more quickly for larger step size. Strange behavior is observed for ETA = 0.1, and ETA = 0.05 seemed to give adequate convergence, therefore ETA was chosen to be 0.05.

Momentum Gain MOM Figures 4.8 - 4.9 show error plots for variations in the momentum gain MOM. The results are exactly as we would expect. For no momentum the network converges very slowly. As momentum increases the convergence of the network increases. However, for large MOM the network does not converge at all. Due to these results we choose MOM=0.5.

Sample Size The size of the sample size is not a variable which we can chosen in this problem, since the world model is essentially trained "on line." However we did look at the absolute error levels to insure that the error decreases as the sample size increases. This behavior was observed as expected.

The design of the world model will now be summarized. The world model is implemented with a neural network with three analog input units and two binary output units. The network has one hidden layer with five units. The step size is 0.05, and the momentum gain is 0.5.

4.3 A Few Comments on Experimental Results

Several interesting effects were observed when the neural network explained above was used to implement the world model. During the planning phase after the first "real world" experience the behavior of the robot usually indicates that the system has not adequately learned the proper world model. The robot is oblivious to some of the obstacles and therefore begins planning a route that is obscured by an obstacle. During the next "real world" experience, the robot follows the planned path until it runs into the obstacle. The robot then essentially wanders around in the vicinity of the location which the world model had not previously learned. The robot repeatedly tries to move through the object, and repeatedly fails. Each of these action-failures are then taught to the world model. The next time the robot enters the planning phase, the world model has been corrected in the very locations which are the most critical to the optimal behavior of the system. Therefore the systems learns to avoid obstacles and paths which lead to these obstacles. This behavior provides a self correcting aspect to the problem. Normal behavior of the system that the second "real world" trial is usually longer than the previous trail, however the third trial is usually optimum, or close to optimum.

The world which we have considered in this project is rather simplistic. In order to get a feel for the systems ability to handle more difficult problems, we have added a random component to the world. When the policy function returns an action, and this action is given to the world, rather than deterministically applying this action, the world applies it with probability P. The randomness thus introduced can be thought of in several ways. P can be considered as a probability of communication failure, or P can be thought of as an unexpected occurrence in the world. The overall effect is to introduce incorrect training samples.

The introduction of noise has several effects. The most surprising effect for us was that the noise actually helps the system to converge. Figure 4.10 shows typical behavior of the system with several different values of P. P = 1 is the deterministic case, and it can be seen that this case is actually the worst. This seemingly odd behavior can be explained as follows. When the system is performing its random walk, the robot will sometimes try to go up, and actually go down. If down is a barrier the the world rewards a negative reward. The Q function will then learn not to go up when actually up is a desirable action. This "false" teaching, increases the initial walk of the robot. The increase in the initial walk, increases the number of training samples shown to the network, which improves the world model. Therefore planning is more complete, and the optimal path is learned quicker.

5 Implementing the Q function by a Neural Network

The reason why the Q function was used instead of the Eval and Policy functions was due to the difficulty of figuring out how to update the policy neural net, as well as because the Q function provides a more general framework [2].

In [1] it is suggested that a lookup table indexed by states x and actions a should be updated by $w_{x_k a_k} \leftarrow w_{x_k a_k} + \alpha(r + \gamma\ EVAL(x_{k+1}) - EVAL(x_k)$. If we use x_k, a_k as inputs of a multi-layer perceptron, backpropagation will not work here, clearly, since the target is be $r + \gamma\ EVAL(x_{k+1})$ but $EVAL(x_k)$ is not the output of the network.

Using a neural net to represent Q, the inputs are x_k, a_k and, as in [2], the target is $r + \gamma \max_b \hat{Q}(x_{k+1}, b)$. A feedforward net with 1 hidden layer was implemented. The non-linearity at the hidden unit is a sigmoid, while a linear function is used in the output. Backpropagation was used to update the network.

Different number of hidden units (usually below 20), different η, momentum gains and input implementations (integer, binary) were tested. γ was made equal to 0.9.

The major problem found is that, when a reward is received, the weights are largely increased for the corresponding action and subsequent recalls from the net will reflect the reinforcement of that action, whatever the state is. The opposite case happens when, after a random walk by states far from the goal, all actions are penalized, and the net "unlearns" all that was previously learned.

This happens since no previous training of the neural net is possible. The method gives as target for a state a value based on the value of $\hat{Q}$ for the next state. Thus, it is not possible to anticipate a training set! Another problem is that, due to the asymmetry of the problem — the best action from the states before the goal is going UP — actions like LEFT and DOWN are rarely experienced and the network can't learn how good they are for some states as fast as it learns RIGHT and UP actions, for example.

When the number of hypothetical steps of the algorithm was increased, a choice of small η (= 0.05) was made to prevent oscillations in the convergence to $\hat{Q}$ and the inputs were binary coded to prevent influence of bigger integers in the range, the states closer to the goal learned and stabilized at the right preferred actions after a few steps. However, the other states in "free-space" randomly changed their evaluation and most promising action along time and converged after a considerable amount of iterations (4 hours running in a Sun workstation). The success was due to the increase of the number of times that a state was visited, together with a slower but safer convergence of $\hat{Q}(x_k, a_k)$.

The following is a copy of the screen after almost complete convergence of the algorithm. The numbers in the lower matrix are the values of the probability of the best action for each state, and the character after them represents the best action at that state. It is noticeable that a shortest path of 17 steps has been achieved, and the algorithm only fails to improve since the DOWN action from the start state has not been learned yet.

```
. . . . . . . X O
. . X . . . . X .
. . X . . . . X .
O . X . . . . . .
. . . . . X . . .
. . . . . . . . .
```

```
 0.41U   0.35R   0.37U   0.39R   0.38R   0.44R   0.46R           0.99U
 0.32R   0.34R           0.37R   0.37R   0.42R   0.44R           0.99U
 0.47R   0.48R           0.51R   0.51R   0.55R   0.58R           0.94U
 0.39R   0.40R           0.44R   0.44R   0.48R   0.51R   0.55R   0.85U
 0.41R   0.41R   0.42R   0.45R   0.45R           0.51R   0.56R   0.75U
 0.35R   0.37R   0.38R   0.41R   0.42R   0.46R   0.48R   0.53R   0.51U
```

```
Shortest Path so far =    17 steps

Real Steps =    4

Total Trials =   161
```

6 Discussion and Future Work

The results presented in this report are encouraging and suggest that the feasibility of implementing both the world model and the Q function by neural networks. This would become necessary if we were working with complex world which could not be represented by lookup tables. The results show that generalization occurs and that the use of the neural network for a simple world improve the performance of the learning algorithms with respect to lookup tables. Also, Q function implementation using a neural net, even though hard to tune and slow to converge, suggest strategies for similar implementation in more complex problems. The lookup table version would prevent the application of the Q-learning method to problems where generalization is strongly needed. The self correcting nature of the system when the world model is implemented with a neural network suggest methods that could be used for error recovery within planning systems.

Future work should thus proceed in the following directions:

- Solve the 2-D world problem presented above using neural networks for both the Q function and world model;
- Model more complex problems, such as robotic tasks, in a similar way to the modeling of the "Robot in a Maze" problem. Robotic tasks executed on CIRSSE testbed are excellent candidates for this type of modeling. We also think that Dyna planning is a possible approach for the Organization Level of an Intelligent Machine. A world model would be required though, for the hypothetical experiments.
- Another approach may be to have a Markov process as the world. This would probably require a net with stochastic units, and/or nets which would learn the transition probabilities, such as a Boltzmann Machine, to model the world.
- Make a detailed study of the tuning procedure and convergence process for the neural network implementing the Q function.

References

[1] Sutton, R.S. "First Results with Dyna, an Integrated Architecture for Learning, Planning, and Reacting," in *Neural Networks for Control,* Miller ed. The MIT Press, 1990.

[2] Sutton, R.S., Barto, A.G., Williams, R.J., "Reinforcement Learning in Direct Adaptive Optimal Control," IEEE Control Systems Mag., Vol. 12, No. 2, April 1992.

[3] Hertz, J., Krogh, A., Palmer, R., "Introduction to the Theory of Neural Computation", Addison-Wesley 1991

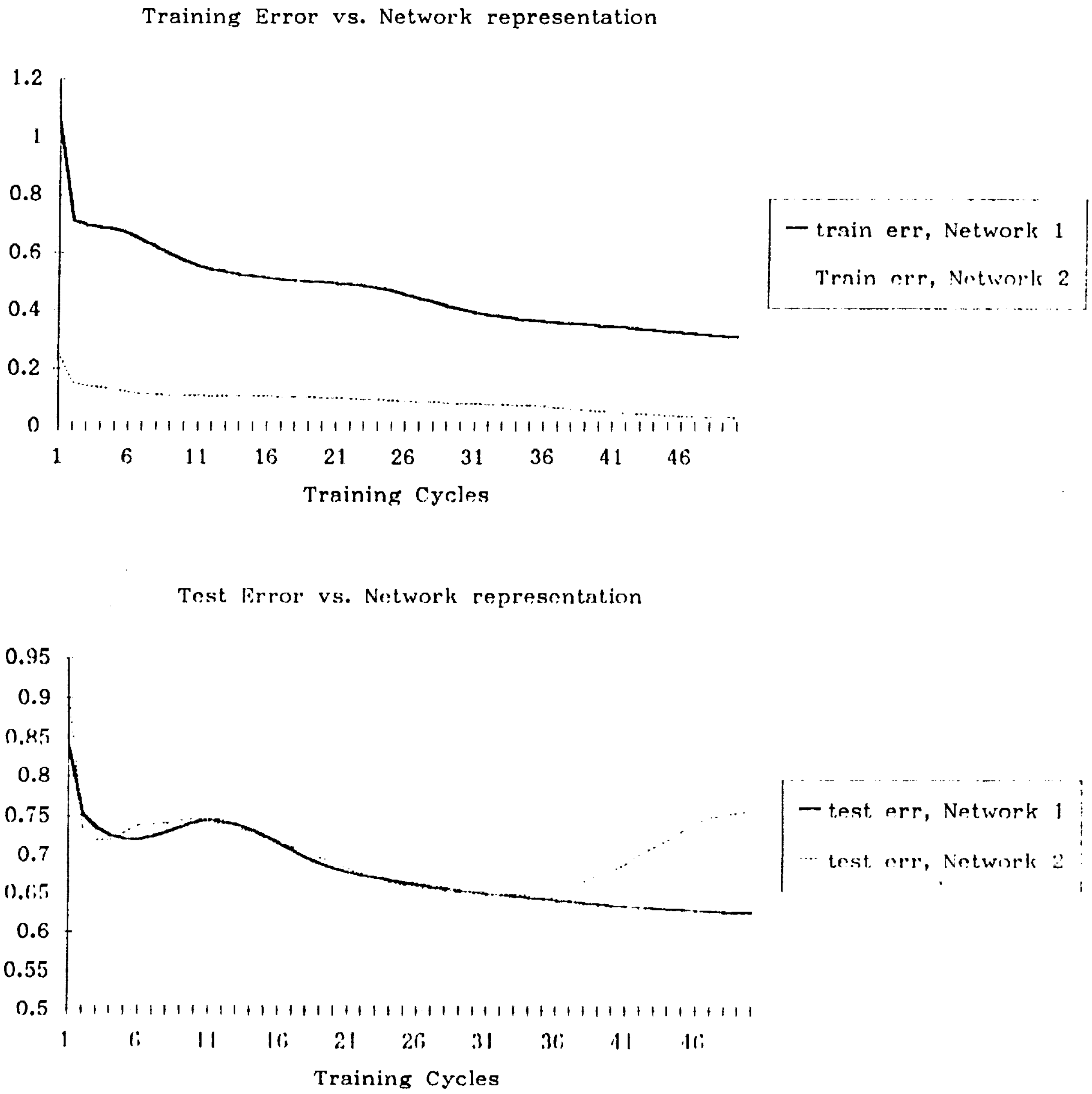
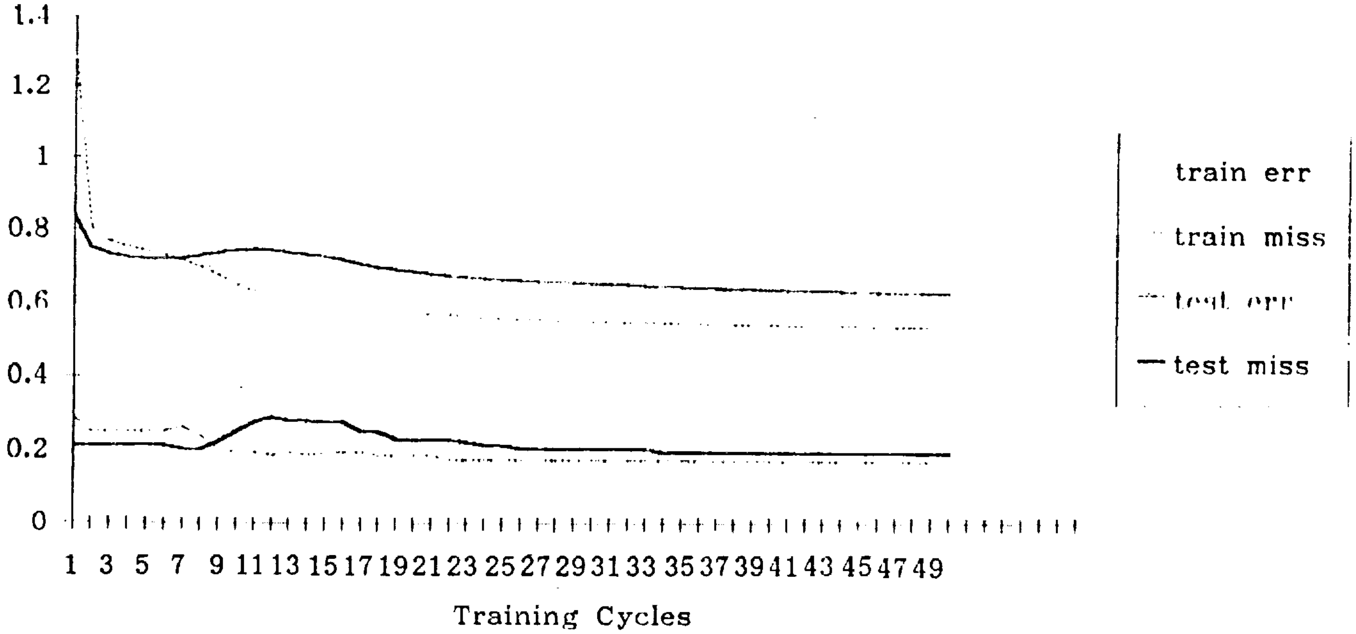
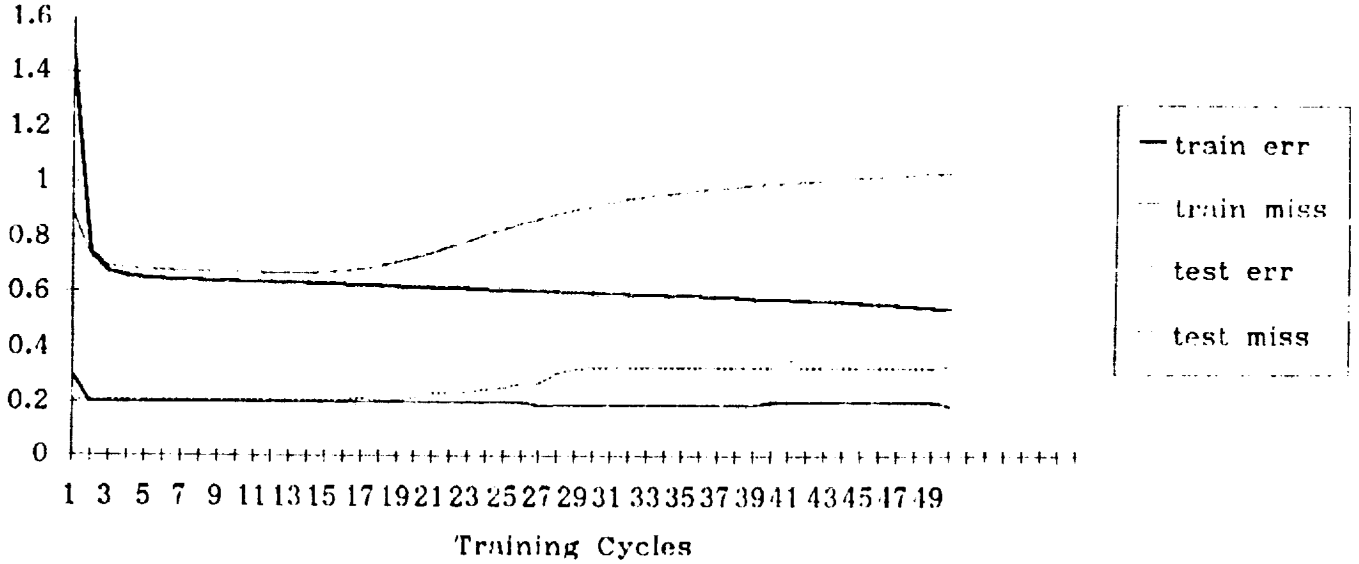

Training Error vs. Network representation
1.2
1
0.8
0.6
0.4
0.2
0
1
6
11
16
21
26
31
36
41
46
Training Cycles
train err, Network 1
Train err, Network 2
Test Error vs. Network representation
0.95
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
1
6
11
16
21
26
31
36
41
46
Training Cycles
test err, Network 1
test err, Network 2

Fig. 4.1

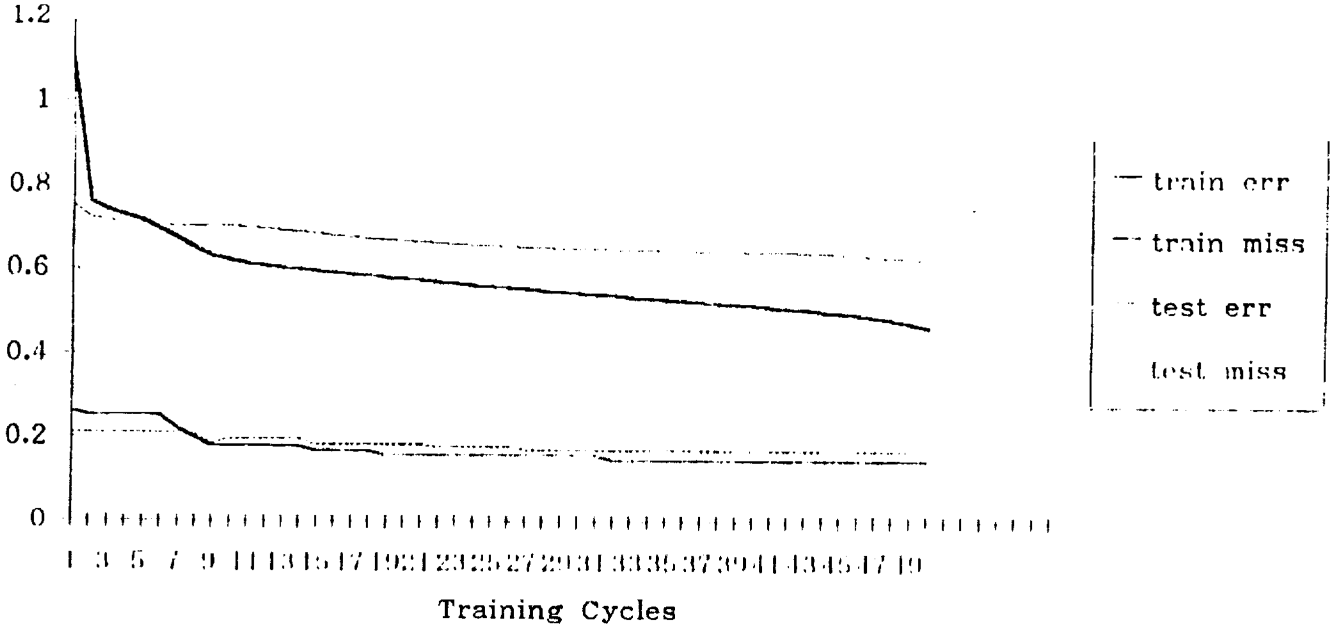


HIDDEN = 5
SAMPLES = 100
MOMENTUM = 0.500000
ETA = 0.050000

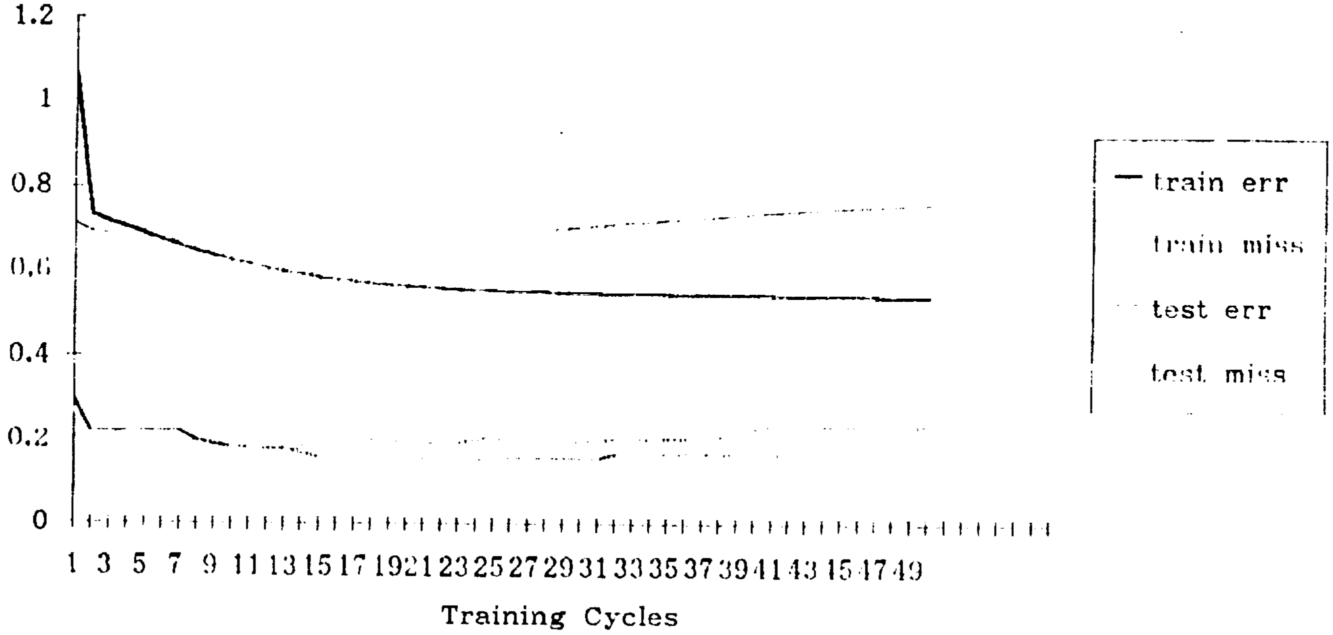


HIDDEN = 3
SAMPLES = 100
MOMENTUM = 0.500000
ETA = 0.050000

Fig. 4.2



HIDDEN = 10
SAMPLES = 100
MOMENTUM = 0.500000
ETA = 0.050000



HIDDEN = 30
SAMPLES = 100
MOMENTUM = 0.500000
ETA = 0.050000

Fig. 4.3

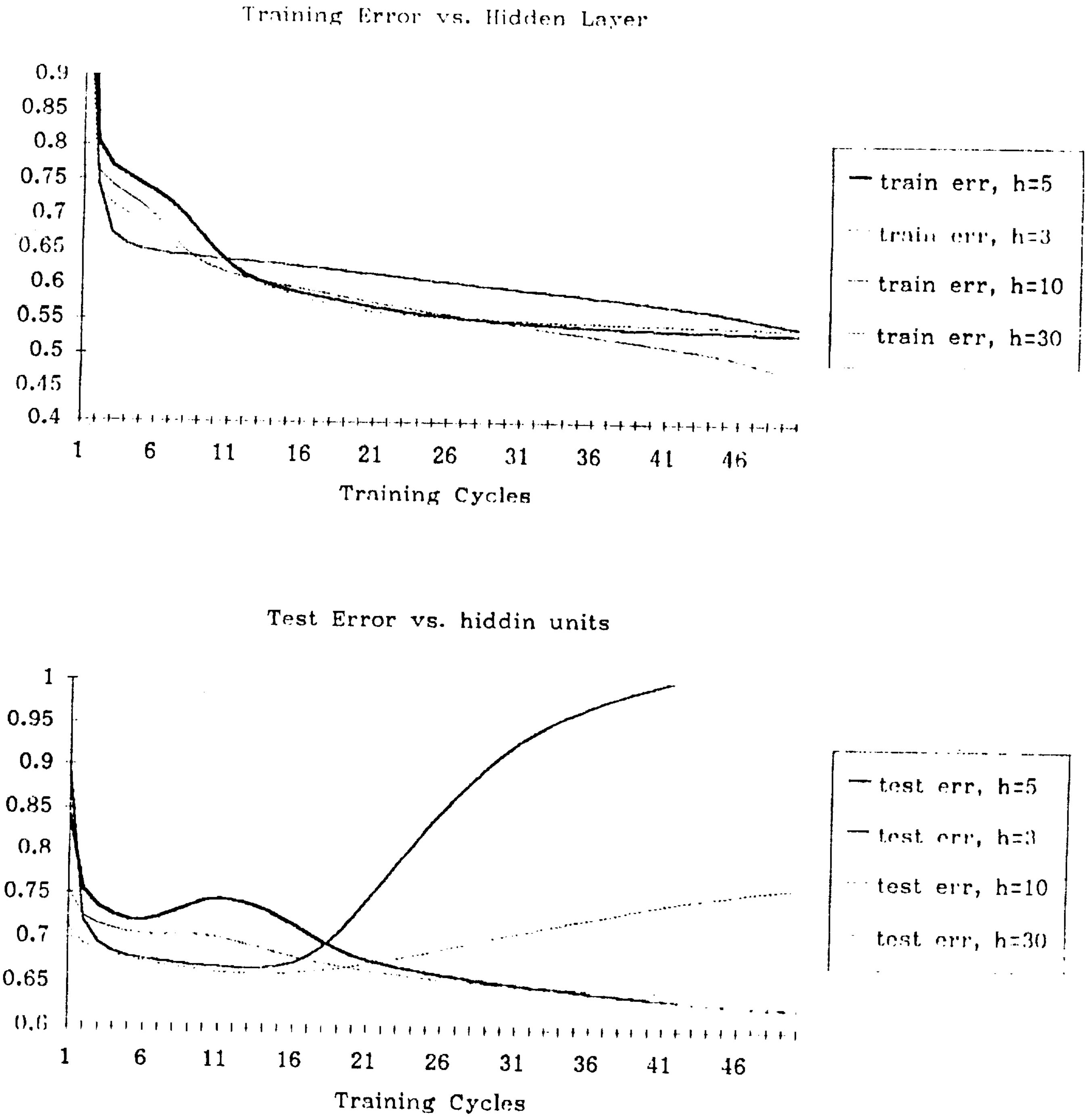
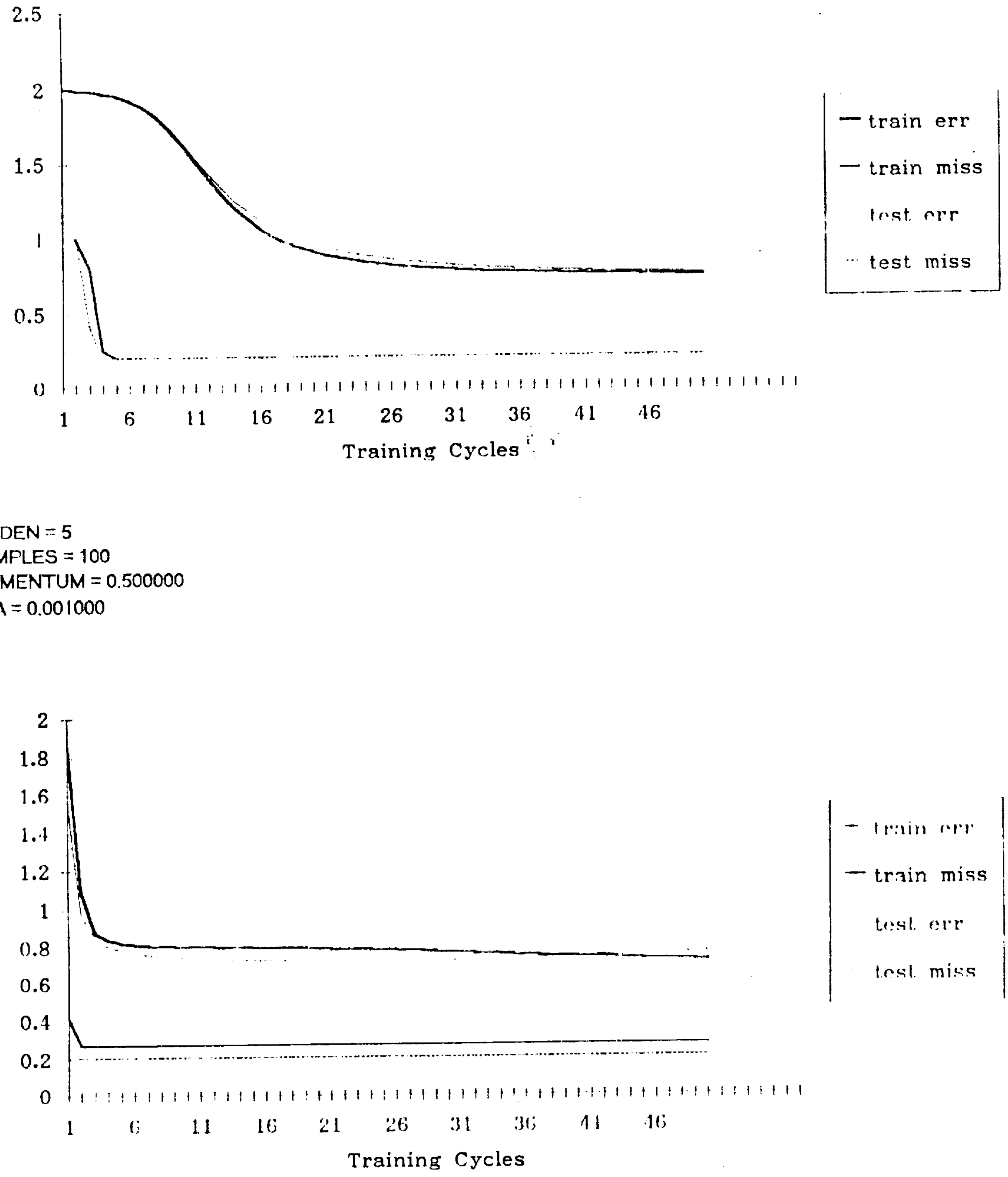
Training Error vs. Hidden Layer
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
0.45
0.4
1
6
11
16
21
26
31
36
41
46
Training Cycles
train err, h=5
train err, h=3
train err, h=10
train err, h=30
Test Error vs. hiddin units
1
0.95
0.9
0.85
0.8
0.75
0.7
0.65
0.6
1
6
11
16
21
26
31
36
41
46
Training Cycles
test err, h=5
test err, h=3
test err, h=10
test err, h=30

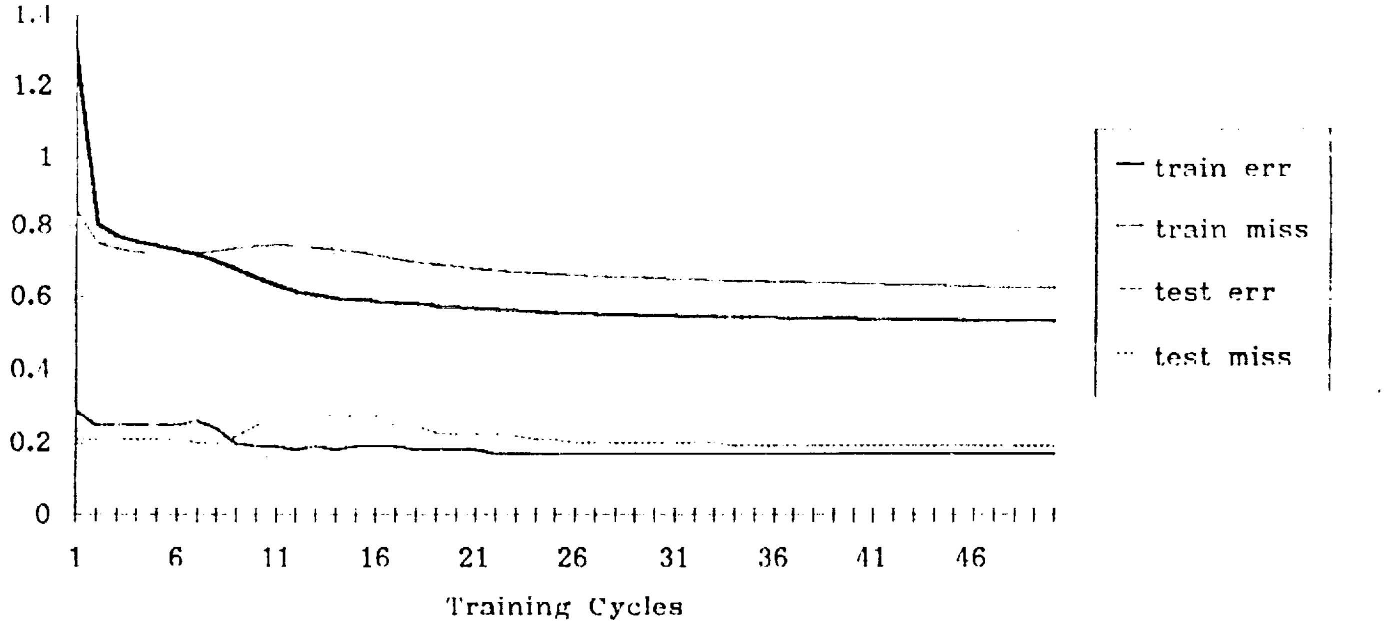
Fig. 4.4



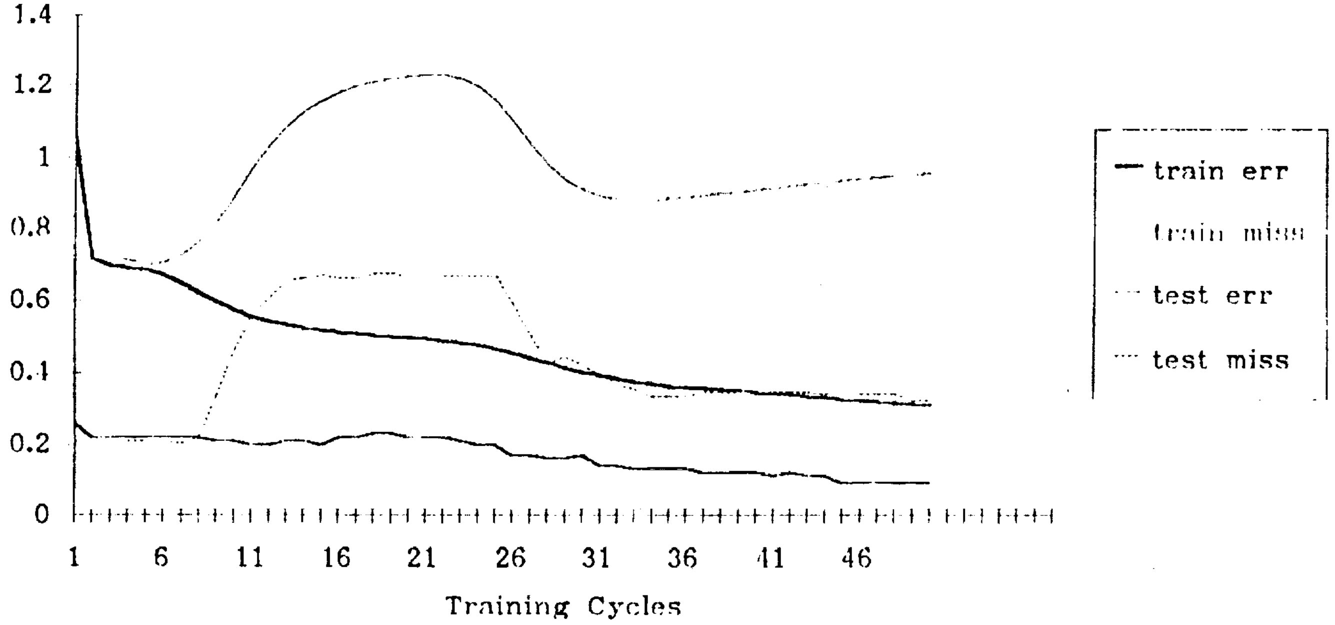
IDDEN = 5
AMPLES = 100
IOMENTUM = 0.500000
TA = 0.001000

IIDDEN = 5
AMPLES = 100
MOMENTUM = 0.500000
ETA = 0.010000

Fig. 4.5

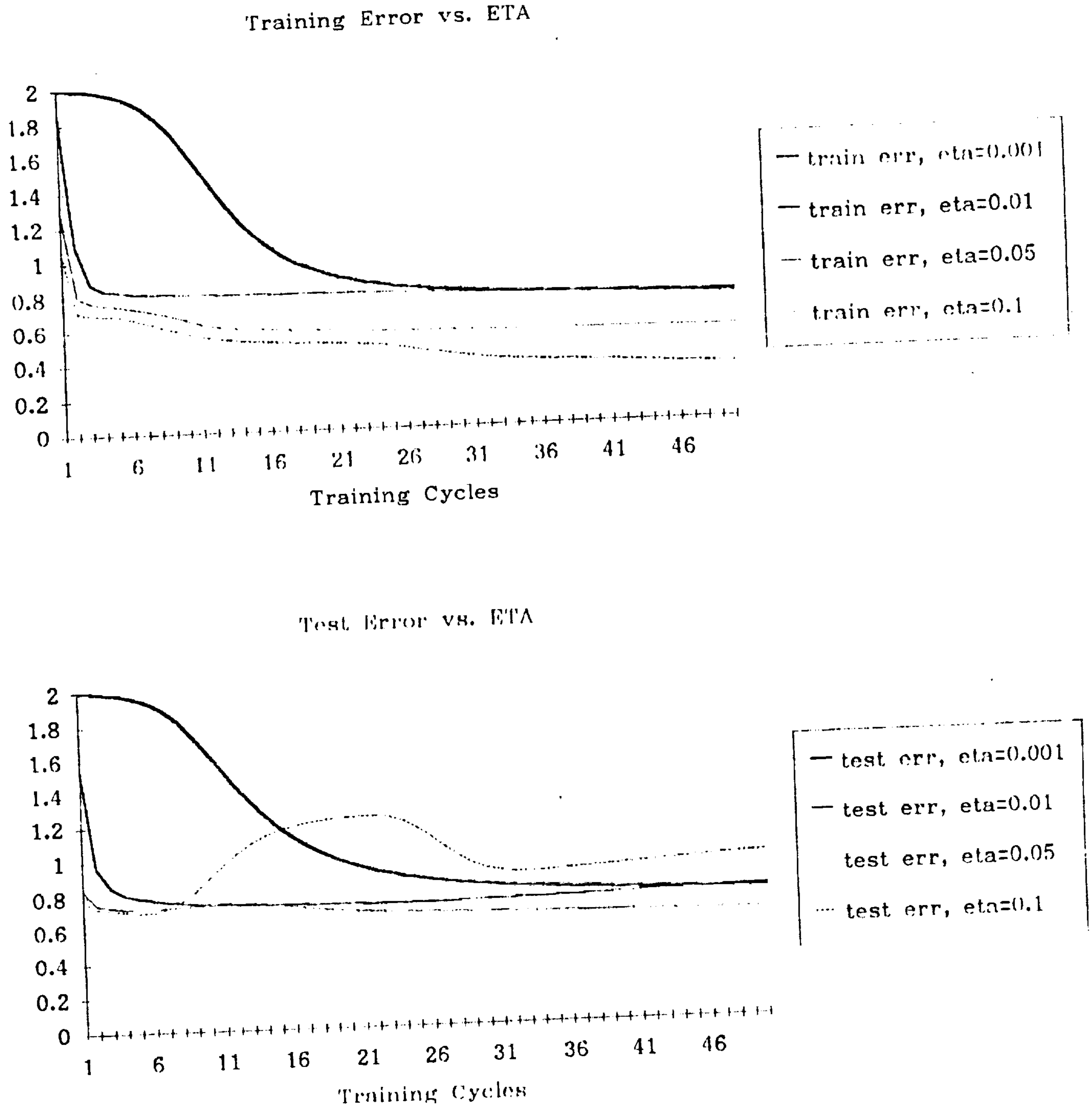


IDDEN = 5
AMPLES = 100
IOMENTUM = 0.500000
TA = 0.050000



IDDEN = 5
AMPLES = 100
IOMENTUM = 0.500000
TA = 0.100000

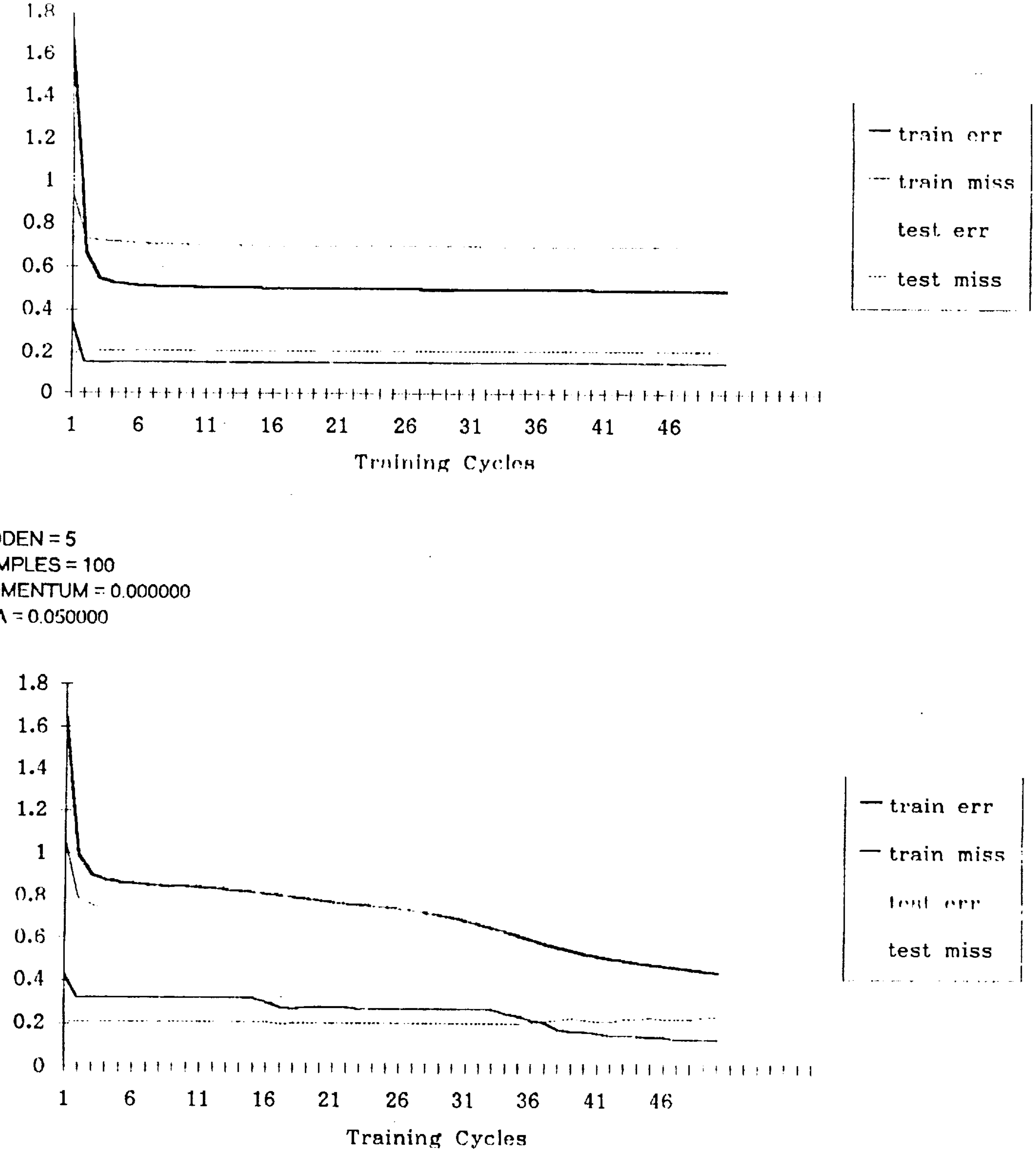
Fig. 4.6



Training Error vs. ETA

Test Error vs. ETA

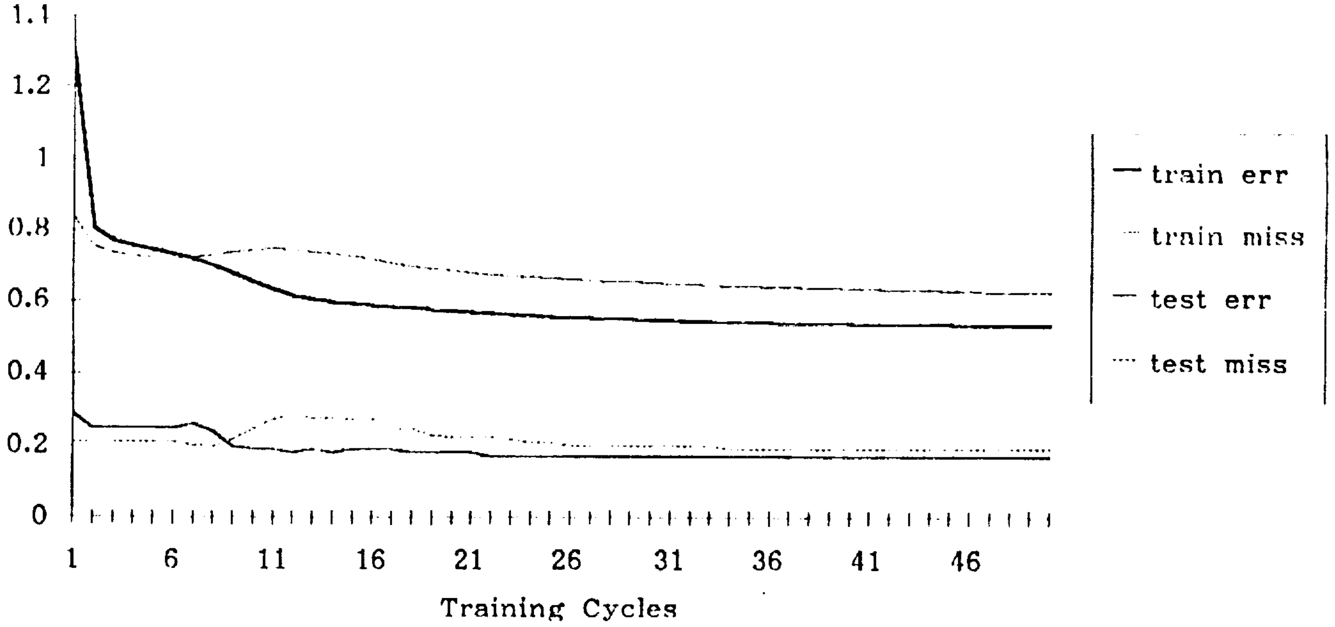
Fig. 4.7



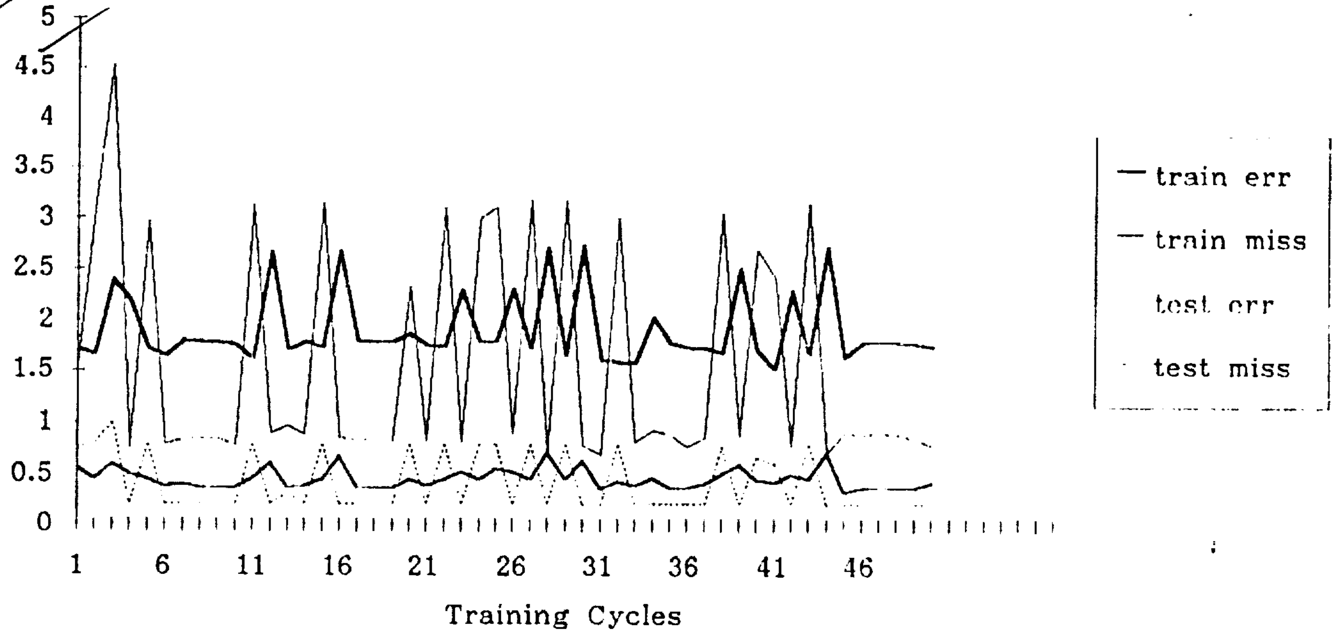
HIDDEN = 5
SAMPLES = 100
MOMENTUM = 0.000000
ETA = 0.050000

HIDDEN = 5
SAMPLES = 100
MOMENTUM = 0.100000
ETA = 0.050000

Fig. 4.8



HIDDEN = 5
SAMPLES = 100
MOMENTUM = 0.500000
ETA = 0.050000



HIDDEN = 5
SAMPLES = 100
MOMENTUM = 0.900000
ETA = 0.050000

Fig. 4.8 (cont'd)

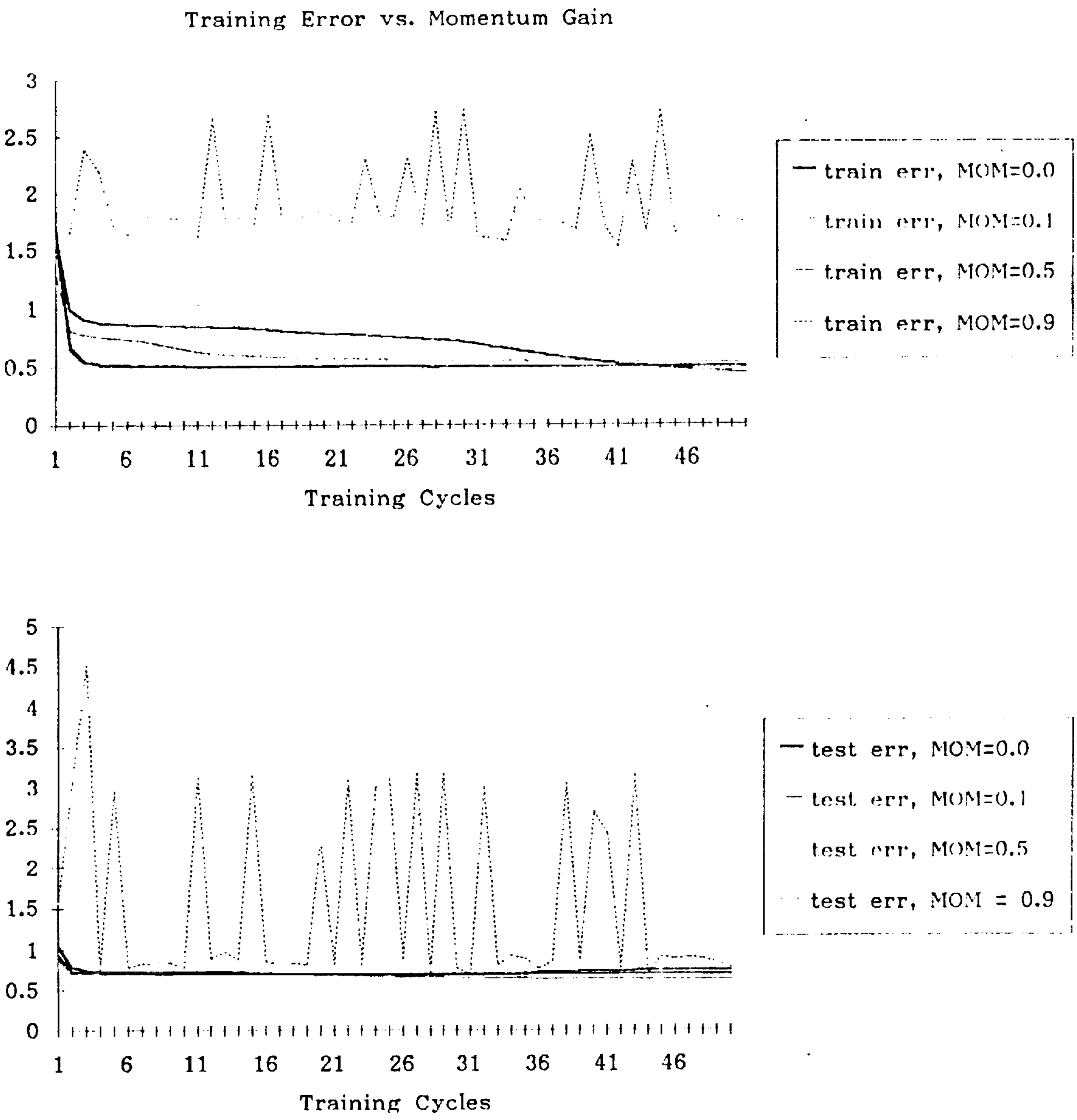

Training Error vs. Momentum Gain

Fig 4.3

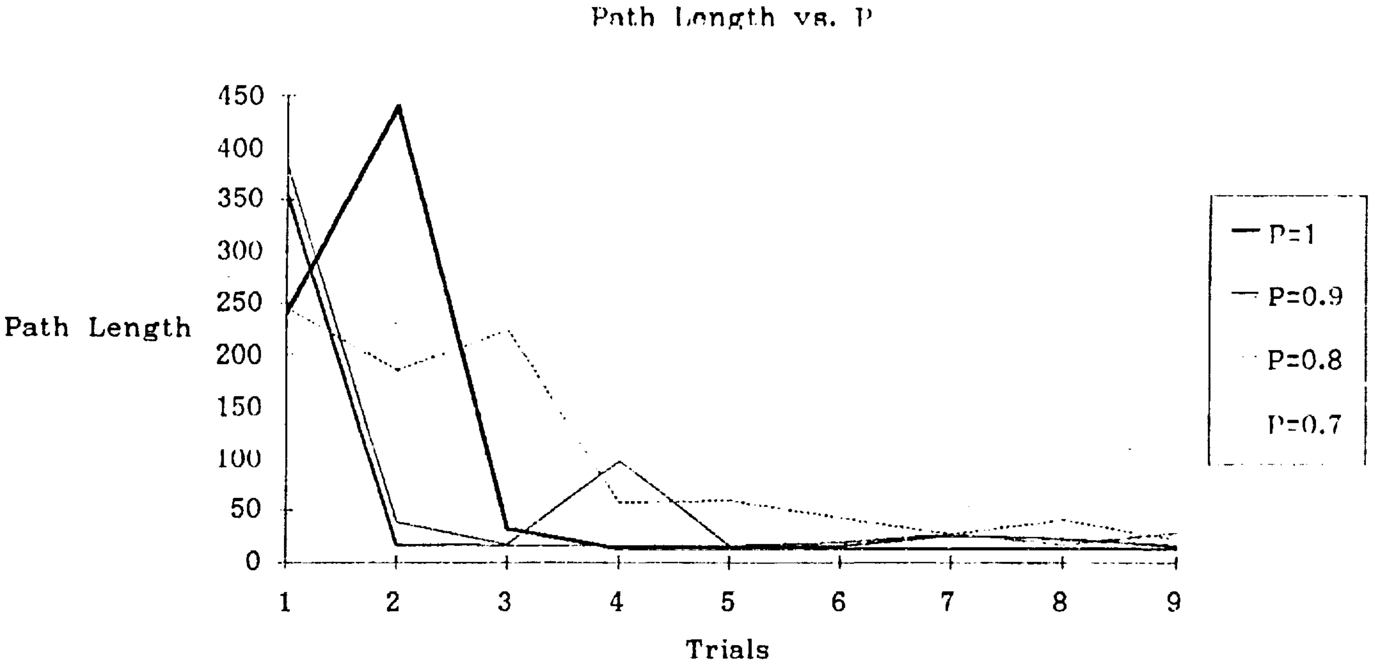

Path Length vs. P
Path Length
450
400
350
300
250
200
150
100
50
0
1
2
3
4
5
6
7
8
9
Trials
P=1
P=0.9
P=0.8
P=0.7

Fig. 4.10

```
*     *     *     *     *     *     *     *     *
*     *     *     *     *     *     *     *     100U
*     *     *     *     *     *     *     *     *
*     *     *     *     *     *     *     *     *
*     *     *     *     *     *     *     *     *
*     *     *     *     *     *     *     *     *
```

After 1 trial.

```
*     *     *     93D   *     *     91D   100R  *
*     *     *     100D  *     100R  100R  100U  100U
*     *     100R  100D  *     *     100D  *     100U
*     76D   42D   100R  100R  100R  100R  100R  100U
91R   100R  100R  100U  100U  *     45U   100U  *
*     42R   28R   100U  100L  85L   *     100U  *
```

After 2 trials.

```
*     41D   57R   100D  *     91D   91D   100R  *
43R   100D  *     100D  100L  100R  100R  100U  100U
100D  100R  100R  100D  100U  100U  100D  *     100U
100D  100D  42D   100R  100R  100R  100R  100R  100U
100R  100R  100R  100U  100U  *     100U  100U  *
100R  100U  100R  100U  100L  99L   46L   100U  *
```

After 5 trials.

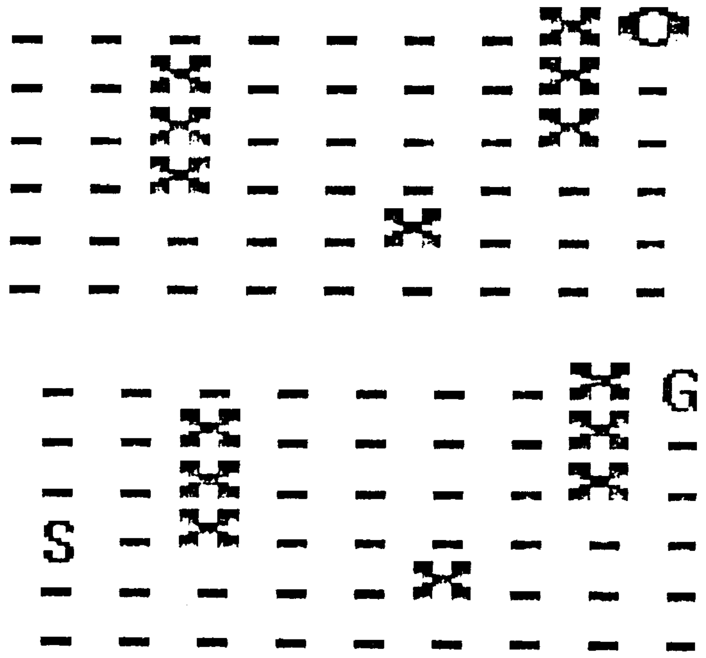


Fig. 4.11