A Stochastic Algorithm for Learning Real-valued Functions via Reinforcement Feedback

Vijaykumar Gullapalli

September 1988

COINS Technical Report 88-91
University of Massachusetts
Amherst, MA 01003

This research was funded in part by the Air Force Office of Scientific Research, Bolling AFB, through grant AFOSR-87-0030.

A Stochastic Algorithm for Learning Real-valued Functions via Reinforcement Feedback

Vijaykumar Gullapalli

September 29, 1988

Abstract

Reinforcement learning is the process by which the probability of the response of a system to a stimulus increases with reward and decreases with punishment [19]. Most of the research in reinforcement learning (with the exception of the work in function optimization) has been on problems with discrete action spaces, in which the learning system chooses one of a finite number of possible actions. However, many control problems require the application of continuous control signals. In this paper, we present a stochastic reinforcement learning algorithm for learning functions with continuous outputs. Our algorithm is designed to be implemented as a unit in a connectionist network. We assume that the learning system computes its real-valued output as some function of a random activation generated using the Normal distribution. The activation at any time depends on the two parameters, the mean and the standard deviation, used in the Normal distribution, which, in turn, depend on the current inputs to the unit. Learning takes place by using our algorithm to adjust these two parameters so as to increase the probability of producing the optimal real value for each input pattern. The performance of the algorithm is studied by using it to learn tasks of varying levels of difficulty. Further, as an example of a potential application, we present a network incorporating these real-valued units that learns the inverse kinematic transform of a simulated 3 degree-of-freedom robot arm.

I would like to express my gratitude for the extensive help and support of Andrew G. Barto and Michael I. Jordan. I am also grateful to Judy Franklin and Richard Sutton for their useful comments on the draft of this report. This research was funded in part by the Air Force Office of Scientific Research, Bolling AFB, through grant AFOSR-87-0030.

1. Introduction

Reinforcement learning is the process by which the probability of the response of a system to a stimulus increases with reward and decreases with punishment [19]. This form of learning has long been studied by psychologists interested in human and animal behavior (under the guise of operant or instrumental conditioning) [24], and by mathematical learning theorists [9,8,3]. Other researchers who have investigated reinforcement learning problems include learning automata theorists [21,20], control theorists [12,19], and researchers studying function optimization [18].

Most of this research in reinforcement learning (with the exception of the work in function optimization) has been on problems with discrete action spaces, in which the learning system chooses one of a finite number of possible actions. However, many control problems require the application of continuous control signals. The importance of this requirement, especially when attempting to model human and animal motor control, was emphasized by Albus [2]. Albus examined the possibility of using reinforcement learning to train the Cerebellar Model Articulation Controller (CMAC) to produce a graded, instead of binary, response to stimulus patterns. To do so, he had to overcome the problem of response saturation, for which he had no immediate solution. The problem occurs because in a reinforcement learning system, learning takes place continuously. This is because even after the optimal response has been learned, the system continues to receive reinforcement every time it produces that response. Some mechanism is required to prevent such reinforcement from further strengthening the response until it gets saturated at a maximum possible value. Otherwise, the system comes to behave as if it has a binary action space, with the response to any stimulus being either the minimum or the maximum possible value.

In this paper, we present a stochastic reinforcement learning algorithm for learning functions with continuous outputs that does not suffer from the saturation problem discussed above. Our algorithm is designed to be implemented as a unit in a connectionist network. We assume that the learning system computes its real-valued output as some function of a random activation generated using the Normal distribution. The activation at any time depends on the two parameters, the mean and the standard deviation, used in the Normal distribution, which, in turn, depend on the current inputs to the unit. Learning takes place by using our algorithm to adjust these two parameters so as to increase the probability of producing the optimal real value for each input pattern. The algorithm does this by maintaining the mean of its activation as an estimate of the optimal activation (the activation that increases the expectation of the reinforcement from the environment) and using the standard deviation to control the amount of search around the current mean value of the activation.

Our algorithm has components analogous to two concepts from classical learning theory. The first is Hull's concept of *Behavior Oscillation* [14]. Hull postulated that the reaction potential (${}_sE_R$) (analogous to activation) varied with time, and he called the standard deviation of this variation the behavior oscillation (${}_sO_R$). He further postulated that the dispersion of ${}_sO_R$ changed with the number of reinforcements. This is akin to our adjusting the standard deviation of the activation based on the reinforcement received. The second related concept is Skinner's *successive approximations* [24], which he suggests as a mechanism for behavior modification. According to Skinner, the distribution of an operant response shifts in a manner such that it is centered around a value that is maximally reinforced. Hull [14] also suggested a similar mechanism for shifting the response intensity along a continuum through reinforcement training. The updating of the mean in our algorithm takes place in a manner similar to both these mechanisms. Sutton [25] also describes an algorithm in which the Normal distribution is used to generate real-valued outputs. However, in his algorithm, the output is controlled by varying the mean alone and the standard deviation is held constant. Harth et al. [13], in a somewhat equivalent approach, designed the Alopex algorithm to produce real values that are proportional to a bias with random noise added to it. Again, the noise had a fixed distribution. Farley et al. [11] describe a scheme in which the noise level is varied based on changes in performance over previous time steps. More recently, Williams [30] examined the possibility of designing stochastic learning automata that use multiparameter distributions to generate their outputs. As an example, Williams considered the Normal distribution and suggested guidelines as to how the mean and the variance of the distribution should be computed from the inputs. The design of our algorithm was based on similar considerations, although we do not use the logarithmic derivatives of the Normal distribution as suggested by Williams.

In developing the algorithm, we restricted our attention to a particular class of tasks from among the many studied by learning theorists. These tasks are extensions of the associative reinforcement learning tasks defined by Barto and Anandan [5]. In the next section, we present a brief overview of the stochastic learning paradigm and define the learning tasks in which we are interested. The algorithm itself is presented in Section 3, and its performance is evaluated in a series of simulations presented in Section 4. The development of this algorithm has been greatly influenced by the work of Barto et al. on A_{R-P} units [5,4]. These units also use stochastic reinforcement learning, albeit to produce binary outputs. A_{R-P} units could be easily modified to produce continuous outputs by removing the activation threshold. However, in such units there would be no control over the amount of noise that is added to the activation and hence they would continue to produce random output values regardless of the duration of training. Another possibility is to use an ensemble of binary units to generate an approximation of a real value. One such ensemble, which we use as a base-line to evaluate the performance of our algorithm,

is described in Section 4.

It should be noted that using the reinforcement learning paradigm might be advantageous when there are several "correct" responses to an input pattern (i.e., when the task is ill-defined). This is particularly true in control tasks for physical systems having excess degrees of freedom. In these tasks, it is difficult to determine *a priori* what the best input-output mapping is. Sometimes it is difficult to determine even a non-optimal input-output mapping for a task. In such cases, it would be more appropriate to let the learning system develop its own mapping, rather than imposing an arbitrary mapping on the system (as would be the case if we were to use supervised learning). In Section 5, we present an experiment that was motivated by our interest in exploring the utility of reinforcement learning in control tasks with excess degrees of freedom. We also compare this approach to an alternate one presented by Jordan [15] based on supervised learning techniques.

2. Stochastic Learning

Consider an abstract machine that randomly selects actions according to some stored probability distribution and receives feedback from the environment evaluating those actions. The machine then uses the feedback to update its distribution so as to increase the expectation of favorable evaluations for future actions. Such a machine is called a *stochastic learning automaton.* Learning automata have gained attention since the work of Tsetlin [27], and that of psychologists studying mathematical learning theory [8,3]. Narendra and Thathachar provide a good review of the theory in [21]. If we consider the task of learning a general input-output mapping using such an automaton, it is clear that the automaton needs to consider input other than the reinforcement signal. Barto et al. [7] call this kind of input the *context input.* For automata with context input, the preferred action may differ in different contexts. Since the automaton is trying to learn which actions to associate with which context inputs, Barto and Anandan [5] term such tasks as *associative reinforcement learning tasks.*

The tasks in which we are interested, and for which we have developed the algorithm presented in this paper, are extensions of associative reinforcement learning tasks. For these tasks, the interaction between the environment and the learning system takes place as follows. At time step t the environment provides the learning system with some pattern vector $x(t)$ from $X = \Re^n$. The learning system produces a random output $y(t)$ selected according to some internal probability distribution over the interval $Y = (0,1)$. The environment evaluates the output $y(t)$ in the context of the input $x(t)$ and sends to the learning system a reinforcement signal $r(t) \in R = [0,1]$, with $r(t) = 1$ denoting the

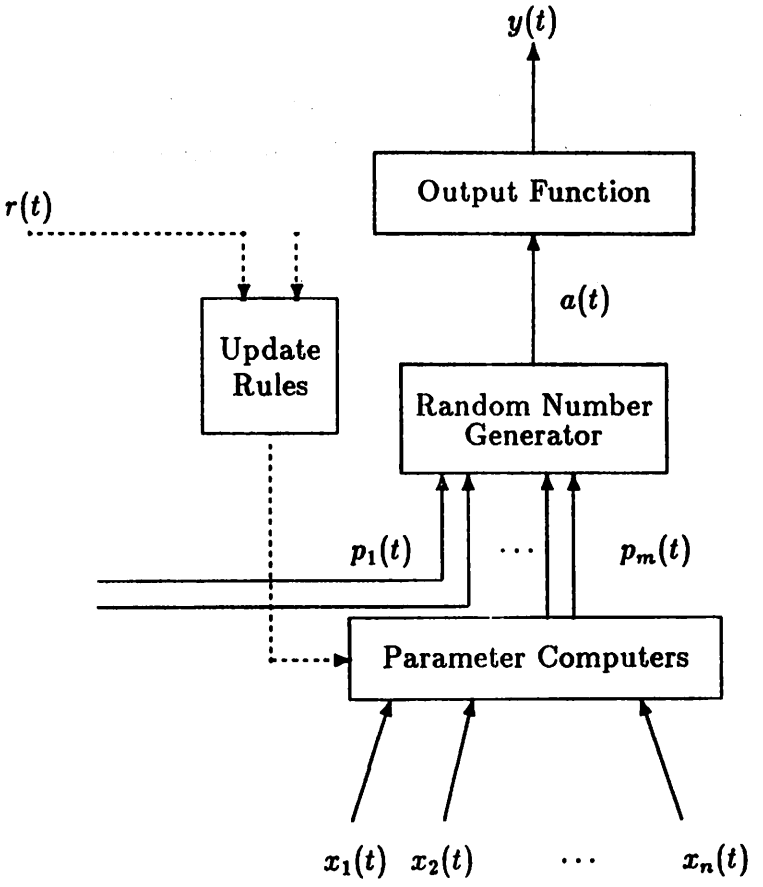

Figure 1: *Block diagram of a generalized stochastic unit*

maximum reinforcement. The environment determines the evaluation $r(t)$ according to some distribution $d : R \times X \times Y \rightarrow [0,1]$, where $d(r,x,y) = Pr\{r(t) = r \mid x(t) = x, y(t) = y\}$. Let r^x denote the maximum reinforcement that could be returned by the environment for a given input x (i.e., $r^x = \max_{r \in R}\{r \mid d(r,x,y) \neq 0 \text{ for some } y \in Y\}$). The objective of the learning system is to learn to respond to each input pattern $x \in X$ with the action y^x with probability 1, where y^x is such that $d(r^x,x,y^x) = \max_{y \in Y}\{d(r^x,x,y)\}$.

A stochastic automaton can be implemented as a stochastic unit that can be used as a component of a connectionist network. Figure 1 depicts such a stochastic unit. It computes its output as a function of its input in the manner described below. The inputs to the unit at time t are denoted $x_i(t), 1 \leq i \leq n$. These inputs are used by the unit to compute the parameters of the probability distribution function used to determine the random activation of the unit. The unit may have several parameters for its distribution, and it is not necessary that all the parameters be computed by the unit itself. In other words, some

of the parameters could be supplied as *inputs* to the unit (by some external agency).[1] We denote the parameters of the unit's distribution at time t by $p_1(t), p_2(t), ..., p_m(t)$. As seen typically in connectionist units, a weighted sum is used in the computation of each parameter, with a different set of weights for each parameter. Thus, if $w_i^j(t)$ is the weight associated with input i for parameter j, then the parameter $p_j(t)$ is computed as $p_j(t) = g_j(\sum_{i=1}^{n} w_i^j(t)x_i(t))$ for some function g_j. It may be mentioned here that a unit that does not use the inputs to determine any of its distribution parameters will, in general, behave like a stochastic learning automaton *without* context input.

The unit then computes its *activation*, $a(t)$, as a random variable drawn from the appropriate distribution (i.e., the distribution defined by $p_1(t), ..., p_m(t)$). Finally the unit computes its *output* as a function of its activation as $y(t) = f(a(t))$, where the function $f(\cdot)$ can be selected based on the kind of output desired. Barto and Anandan [5] use a threshold function for the A_{R-P} unit so that the unit's output values are restricted to either $+1$ or -1. Rumelhart, Hinton and Williams [23] use a nonlinear *squashing function* in their Back-propagation algorithm. An example of such a function is the *logistic function*

$$f(z) = \frac{1}{1 + e^{-z}}. \tag{1}$$

Such a function is useful for units with outputs restricted to the interval (0,1).

Given this framework for a stochastic unit, [2] we can build a unit that implements any specific stochastic learning algorithm by specifying (a) the distribution used by the unit to generate the random activation values, (b) the functions g_j used for computing the parameters of the distribution, (c) the output function $f(\cdot)$, and (d) the update rules for modifying the weights of the unit.

[1] An extreme example of this is when the computation of the parameter is external to the network itself, as in the case of temperature in the Boltzmann machine [1]. A more localized case is when the computation is done by specific units in the network and the parameters broadcast to other units in the network.

[2] This definition of a stochastic unit is similar to that of the *Stochastic Quasilinear Unit* in [30], but there are significant differences. The first is that Williams defines the activation of the unit as a deterministic function of the inputs and uses this activation as the parameter (his stochastic quasilinear unit is designed for single parameter distributions) for the random number generator, which directly produces the output $y(t)$. We, on the other hand, consider the activation to be the value *produced* by the distribution of the unit and transformed by the function $f(\cdot)$ into the output of the unit. Since the units we consider can have more than one parameter for the distribution function, we feel this definition is more appropriate and more general. Another important difference is that in our framework the parameters for the unit's distribution can be computed externally and supplied as input to the unit.

3. A Learning Algorithm for Real-valued Units

There are two aspects that stochastic learning units producing real-valued outputs have to control. One is the value to be produced as output for a given input. The second is the amount of exploratory behavior the unit should manifest while learning in order to produce the best output values possible. The first can be regarded as estimating the mean value to output, while the second is equivalent to determining the variance to be exhibited in computing the activation of the unit. Ideally, a unit that learns real-valued outputs by means of reinforcement learning should have the following properties:

(a) it should be able to learn to associate with each input pattern an output value for which the reinforcement signal it receives indicates the highest degree of success;

(b) it should be able to improve its performance in cases where it is doing poorly by using greater degrees of exploratory behavior;

(c) it should be able to discriminate between cases in which it is doing poorly and those in which it is doing well, so that it does not degrade its performance in cases in which it is doing well by exhibiting behavior that is too random.

Clearly, a single parameter distribution cannot exhibit these properties because there is no way of independently controlling the mean and variance of real-valued random variables generated using a single parameter distribution. It is therefore necessary to consider using multi-parameter distributions to determine the output of the stochastic unit. It is interesting to note that the idea of controlling the exploratory behavior of the unit by using an additional parameter is exactly what has been achieved in the Boltzmann machine [1] by using the global temperature to control the variability in any unit's output. We now present a stochastic learning algorithm that can be used by a unit learning to produce real-valued outputs by estimating the mean and standard deviation of the Normal distribution, $\Psi(\mu,\sigma)$, it uses to generate its activation. We restrict attention to units whose outputs lie in the open real interval (0,1). Clearly, this does not restrict the power of these units because the output values can be scaled to cover any range of real values. The reinforcement received by the unit from the environment is also assumed to be in [0,1] with 1.0 denoting the maximum reinforcement.

The basic idea for this learning algorithm is very simple. Since we want the mean, μ, of the distribution generating the activation to be an estimate of the optimal output, it is natural to have the unit compute the mean in the "usual" manner of associative learning units. A simple way is to let $\mu(t)$ (the mean at time t) equal a weighted sum of the inputs

of the unit at time t:

$$\mu(t) = \sum_{i=1}^{n} w_i(t)x_i(t) + w_{thres}(t). \tag{2}$$

The determination of the standard deviation σ is more involved. The conditions stated above indicate that for a given input, the standard deviation should depend on how good the current expected output (i.e., the current value of the mean) is. Since the reinforcement input returned from the environment is a measure of this, what we are saying is that for a given input, the standard deviation used in computing the output should depend on the *expected reinforcement.* If the expected reinforcement is high, the unit is performing well for that input and so σ should be small. Conversely, if the expected reinforcement is low, σ should be large so that the unit explores a wider interval in its output range.

To accomplish this, the weights v_i are used to compute the *expected reinforcement*, $\hat{r}(t)$, as follows:

$$\hat{r}(t) = f(\sum_{i=1}^{n} v_i(t)x_i(t)), \tag{3}$$

where $f(z)$ is the *logistic function* described previously (Equation 1). This expected reinforcement is used to compute the standard deviation as

$$\sigma(t) = 1 - \hat{r}(t). \tag{4}$$

Based on $\mu(t)$ and $\sigma(t)$, the unit computes its *activation*, $a(t)$, which is a normally distributed random variable:

$$a(t) = \Psi(\mu(t), \sigma(t)). \tag{5}$$

Finally, the activation $a(t)$ is transformed into the output $y(t) \in (0, 1)$ by passing it through the logistic function (Equation 1):

$$y(t) = f(a(t)). \tag{6}$$

Equations 2–6 describe how the unit uses its inputs to compute its output at a given time step. We now give the learning rules, or, in other words, the equations used by the unit to update its weights. Assume the environment provides a reinforcement signal $r(t)$ that is its evaluation of the output of the unit at time t. The weights controlling the mean are updated at each time step as follows:

$$w_i(t+1) = w_i(t) + \alpha\Delta_w(t)x_i(t), \tag{7}$$

$$w_{thres}(t+1) = w_{thres}(t) + \alpha\Delta_w(t), \tag{8}$$

where α is the learning rate parameter and

$$\Delta_w(t) = (r(t) - \hat{r}(t))\frac{a(t) - \mu(t)}{\sigma(t)}. \tag{9}$$

We can view the fraction in Equation 9 as the *normalized noise* (or jitter) that has been added to the mean activation of the unit. If this noise has caused the unit to receive a reinforcement signal that is *more* than the expected reinforcement, then it is desirable for the unit to have an activation closer to the current activation $a(t)$. It should therefore update its mean output value in the direction of the noise. That is, if the noise is positive, the unit should update its weights so that the mean value increases. Conversely, if the noise is negative, the weights should be updated so that the mean value decreases. On the other hand, if the reinforcement received is *less* then the expected reinforcement, then the unit should adjust its mean in the direction *opposite* to that of the noise. It can be verified that the above equations have the desired effect on the mean.

The updating of the weights for the expected reinforcement is relatively straightforward. We want to associate with each input vector a corresponding reinforcement value, and since both of these are supplied to the unit, we can use the supervised learning paradigm here to learn this association. This can be done simply by using the LMS rule of Widrow and Hoff [28], as shown in the following equations:

$$v_i(t+1) = v_i(t) + \beta\Delta_v(t)x_i(t), \tag{10}$$

where β is the learning rate parameter and

$$\Delta_v(t) = r(t) - \hat{r}(t). \tag{11}$$

Figure 2 shows how the above algorithm can be implemented as a stochastic learning unit.

The basic learning cycle for a unit implementing these equations requires the following operations performed at each time step:

1. the input values $x_i(t)$ are presented to the unit by the environment;
2. the unit uses its input values to compute $\mu(t)$ using the weights $w_i(t)$ and w_{thres} as in Equation 2;
3. the unit computes $\hat{r}(t)$ using the weights $v_i(t)$ as in Equation 3 and uses it to compute $\sigma(t)$ using Equation 4;
4. the unit then computes its activation and its output using Equations 5 and 6;

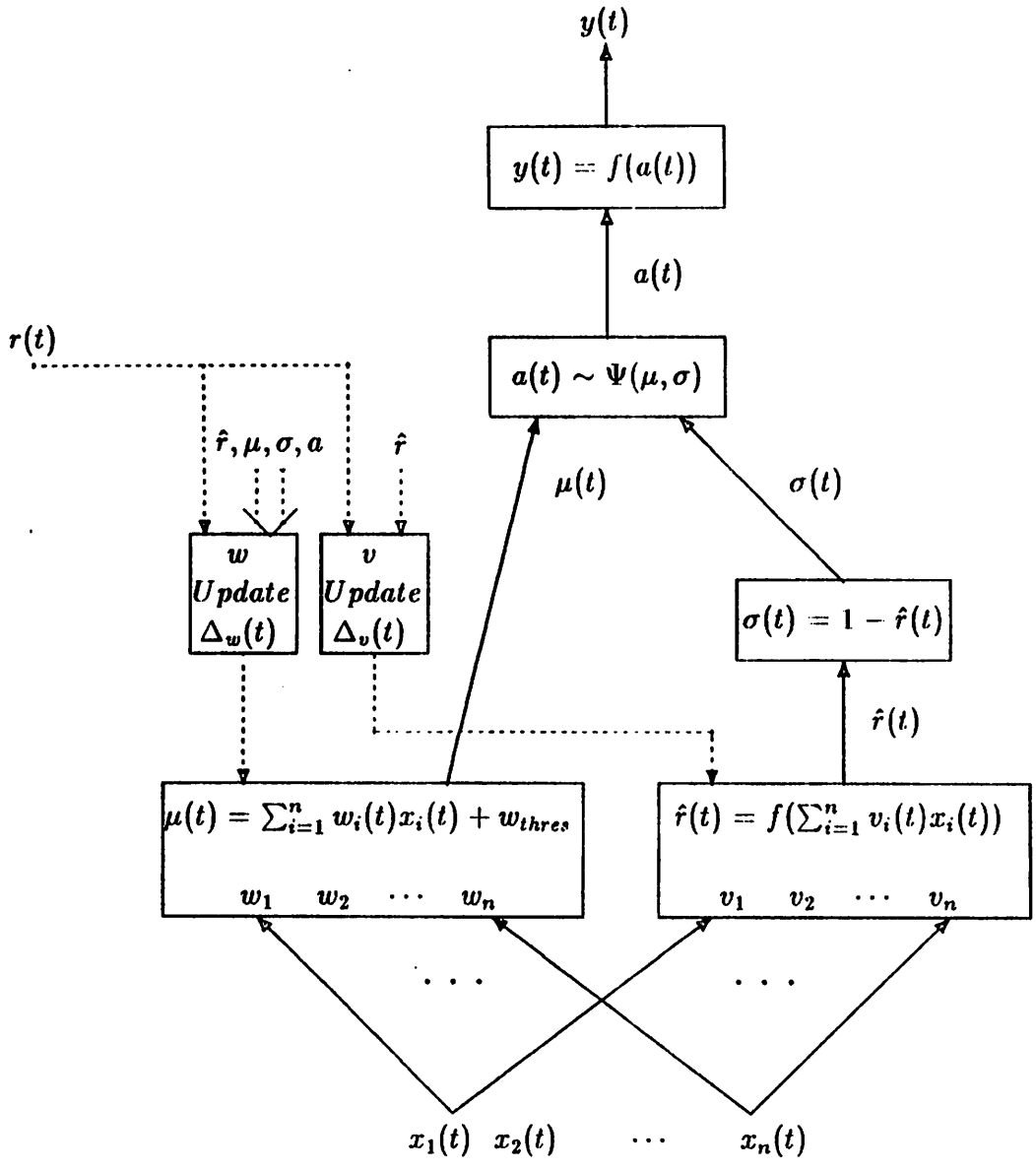

Figure 2: *Block diagram of a SRV unit showing the various computations being performed. The flow of signals used to compute the output of the unit is shown with solid lines, while the signals used for learning are shown with dotted lines.*

5. the environment evaluates the output produced for the inputs it supplied and sends a reinforcement signal $r(t)$ to the unit;

6. the unit uses this reinforcement to update all its weights using Equations 7–11.

In the case of a network of units, the inputs are presented to the input units in the network and the outputs of each succeeding layer from the first to the last are computed using steps 2 to 4 above. The environment then looks at the outputs of units in the output layer *only* and generates the reinforcement signal based on these outputs. This signal is then used by *all* the units to update their weights. We call units implementing the above algorithm *SRV* (Stochastic Real-Valued) units.

4. Performance of the Algorithm

In order to illustrate the learning behavior of SRV units, we present simulation results for units trained on five different tasks. All the tasks considered in this section are multiple input, single output tasks. For each task, the unit or network is presented with several stimulus vectors, for each of which it generates an output. The environment evaluates the output and returns a reinforcement based on the evaluation. The evaluation function is assumed to be known only to the environment. The reinforcement signal is then used by the unit or network to improve its performance of the task.

To keep things simple, the tasks presented in this section are defined by a set of stimulus-response pairs in $[0,1]^2 \times (0,1)$, but the desired response to any given stimulus is assumed to be known only to the environment and is used by it to determine the reinforcement. Such a situation is obviously contrived, and much faster learning could be obtained by having the environment provide the correct output for each input, i.e., by using supervised learning. However, it should be clear that we are using these tasks as a simple means of simulating cases in which the environment evaluates the output of an unit or network *without* knowing the correct response, so that we can test the learning behavior of the algorithm in such cases.

The tasks presented in this section are divided into two groups for which different configurations of the units are used. In the first group (Tasks 1, 2 and 3), a single unit is used to learn the task. In the second group (Tasks 4 and 5), networks of units are used. The stimulus-response vectors used for these tasks are presented in Table 1.

Task #	Stimuli Input → Output	Plot of performance on a single training run	Plot of average perf. over 20 training runs
1	(0.3, 0.8) → 0.8 (0.9, 0.2) → 0.4	Figure 5	Figure 6
2	(0.3, 0.1) → 0.8 (0.2, 0.7) → 0.5 (0.8, 0.5) → 0.2	Figure 7	Figure 8
3 (AND)	(0.1, 0.1) → 0.1 (0.1, 0.9) → 0.1 (0.9, 0.1) → 0.1 (0.9, 0.9) → 0.9	Figure 9	Figure 10
4 (AND) (Network)	(0.1, 0.1) → 0.1 (0.1, 0.9) → 0.1 (0.9, 0.1) → 0.1 (0.9, 0.9) → 0.9	Figure 12	Figure 13
5 (XOR)	(0.1, 0.1) → 0.1 (0.1, 0.9) → 0.9 (0.9, 0.1) → 0.9 (0.9, 0.9) → 0.1	Figure 14 (Normal reinf.) Figure 15 (Modified reinf.)	Figure 16

Table 1: *The five tasks used to test the performance of the SRV units. A single SRV unit was trained to perform Tasks 1 through 3, while a network of SRV and back-propagation units was used for Tasks 4 and 5. The table also specifies the figures showing performance plots relevant to each task. See text for details.*

Single Unit Tasks

In each of the three tasks in this group, a single SRV unit with two context inputs and an additional reinforcement input was used (Figure 3). For each of these tasks, the unit was started with all its weights set to zero. The learning rates used were $\alpha = 0.25$ and $\beta = 0.5$. The training was conducted as a sequence of *epochs*. In a single epoch, each of the input vectors specified for the task was presented to the unit and the learning cycle described in Section 3 was executed. At the end of each presentation, an error value was computed as the difference between the desired and actual output. Since both these values lie in (0,1), $|error| \leq 1$. This error was used to determine the reinforcement signal broadcast to the unit. The unit then used this signal to update its weights. Two different

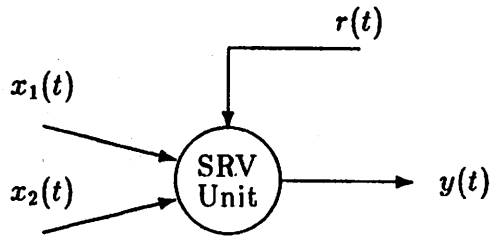

Figure 3: *A single real-valued unit with two context inputs and one output. The unit also has a reinforcement input.*

reinforcement signals were used: $r_1(t)$ and $r_2(t)$. The first of these, $r_1(t)$, is defined as

$$r_1(t) = 1 - |error|. \tag{12}$$

In order to test the tolerance of the unit to noisy evaluations from the environment, each task was also run with the reinforcement $r_1(t)$ corrupted by some noise. This was done by generating random numbers for the reinforcement $r_2(t)$ using the Normal distribution with mean $r_1(t)$ and standard deviation 0.005. Thus

$$r_2(t) = \Psi(r_1(t), 0.005). \tag{13}$$

It is possible to approximate a real-valued stochastic unit using a set of binary stochastic units. In order to provide a means of comparison between these two methods of learning, we present simulation results wherein an ensemble of A_{R-P} units is also used to learn these tasks. The ensemble is modeled after the Goore games discussed by Tsetlin [27]. As presented there, a Goore game is a collection of stochastic units *without context input*, that each contribute a fraction of the total output of the ensemble. We have elaborated on the basic definition of Goore games by considering units that use context inputs. Thus the ensemble of units used by us can be characterized as an *associative Goore game*. The ensemble is constructed as in Figure 4. Each binary A_{R-P} unit contributes a value of 0.0 or 0.1 to the output depending on whether its output is -1 or 1 respectively. There are nine units in all, thus permitting output values from 0.0 to 0.9 in steps of 0.1.

To summarize, each task described in Table 1 was run for 20 training runs using each combination of the two types of units (real-valued and binary) with the two types of reinforcement ($r_1(t)$ and $r_2(t)$), where a training run consisted of a single attempt at training the unit to perform the task. Each training run was continued until the average

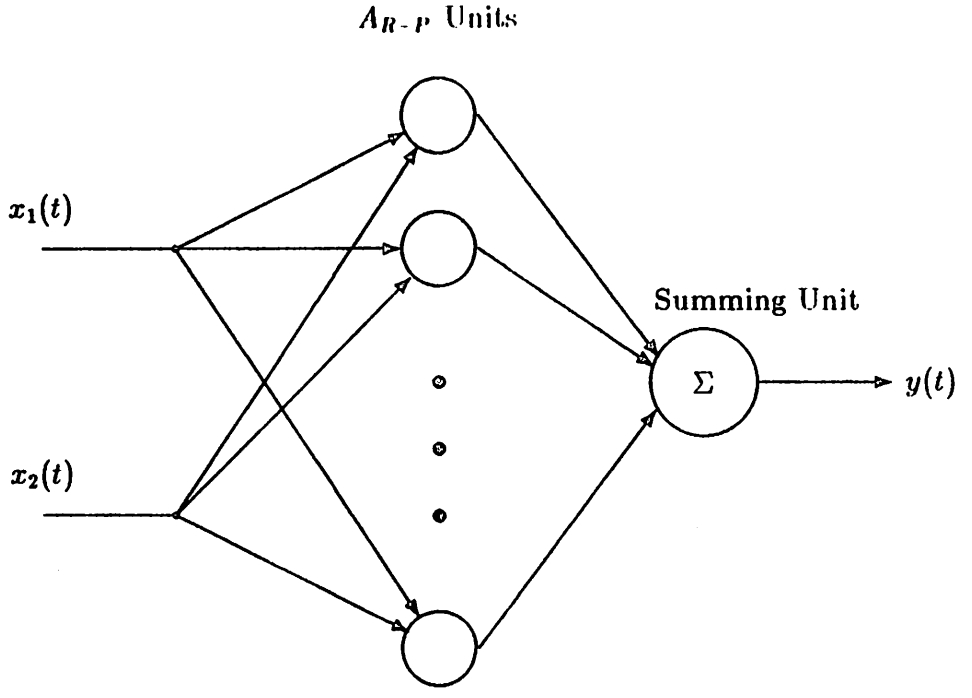

Figure 4: *An ensemble of binary A_{R-P} units used to simulate a real-valued unit. The output of the ensemble is the weighted sum of the outputs of each A_{R-P} unit, with each unit being assigned a weight of 0.1.*

error fell below a criterion or until a fixed number of epochs had elapsed. During a training run, at the end of each epoch, the average error over the last 100 epochs (200 epochs for Task 3) was computed. This error was used as a measure of the performance of the unit and is the value plotted in the graphs of the performance on the three tasks. For each task we present two graphs. The first is a plot of the performance of all four combinations over a single training run. The second is a plot of the average performance over all 20 training runs for each of the four combinations.

The graphs for Task 1 show that the SRV unit rapidly converges to the desired values, although in the case of a noisy reinforcement, the unit has a residual error of about 0.002 because the noisy reinforcement does not let the expected reinforcement of the unit become 1 (thus preventing the variance of the unit from becoming zero). In the case of the ensemble of units, the learning (with both kinds of reinforcement) was much slower, although the error was clearly being reduced. The relative performance figures for Task 2 are quite similar to those of Task 1, although the convergence rates for this task were slower than those of Task 1 in all four cases. In the case of Task 3 also, convergence is slow and there is still a sizable residual error even after 10,000 epochs. This is because for this task and

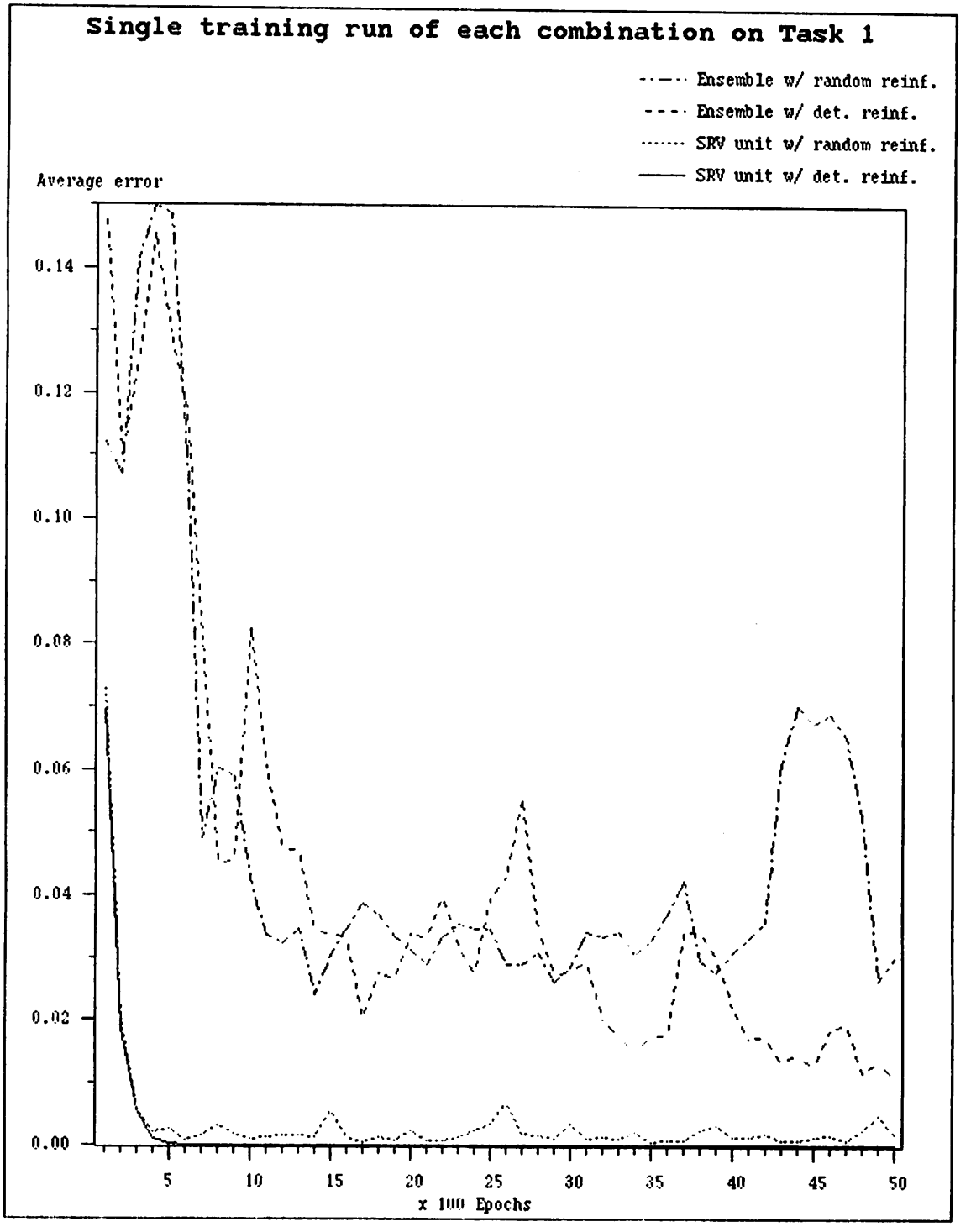

Figure 5: *Plot of the average error (over the last 100 epochs) at each epoch for each of the four combinations [real-valued and ensemble with r_1 and r_2] trained on Task 1. This plot is for a single training run.*

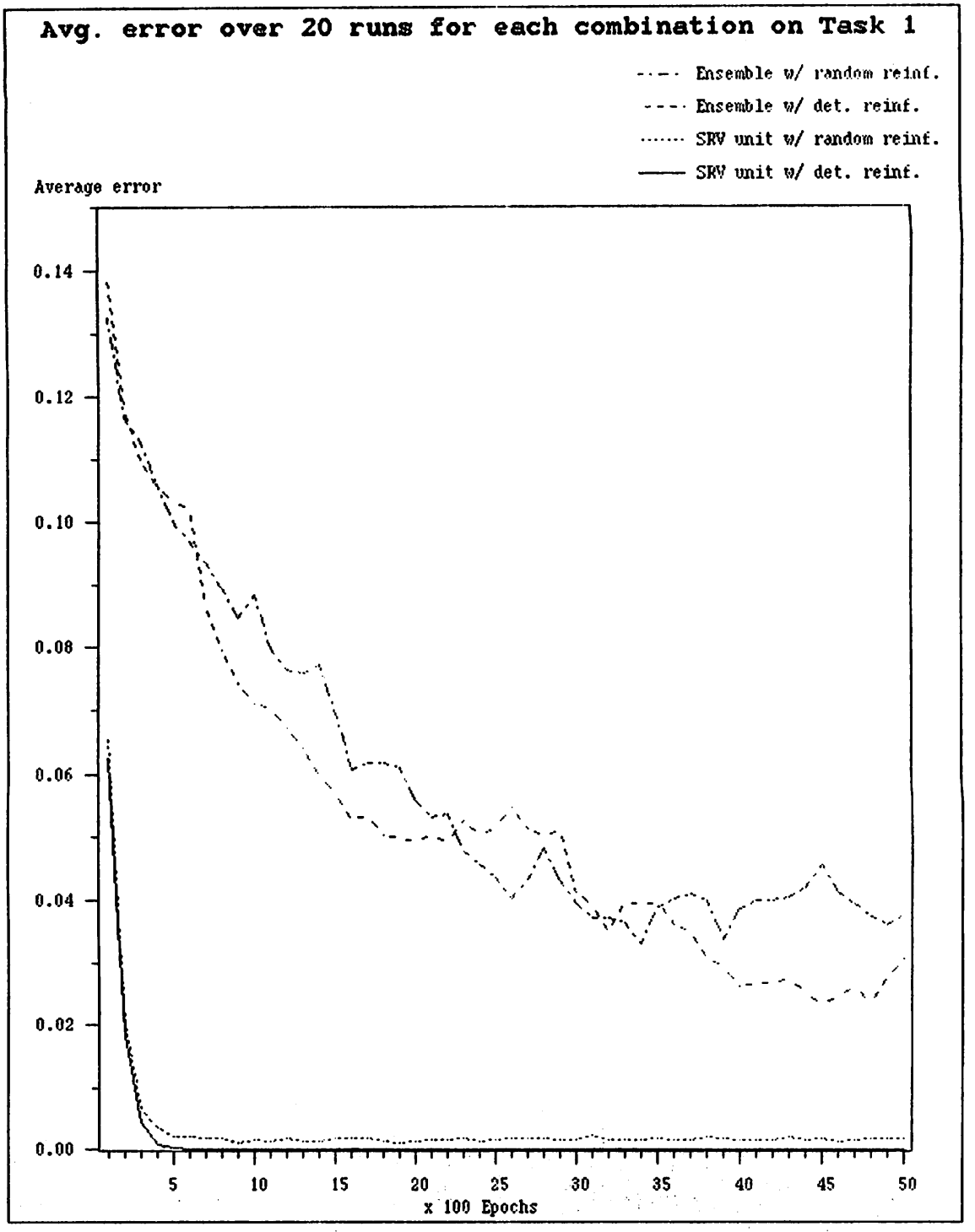

Figure 6: *Plot of the average error (over the last 100 epochs) at each epoch for each of the four combinations [real-valued and ensemble with r_1 and r_2] trained on Task 1. The error at each epoch is further averaged over 20 training runs.*

for a single unit, there is no set of weights that can compute the given mapping. However, the unit still learns a close approximation of the task (average error is below 0.75 after 2000 epochs), even though the output values do not exactly match the desired values.

From these simulations we can conclude that the algorithm works reasonably well for simple tasks and shows good convergence properties. It is also apparent that even for the simple tasks considered here, an ensemble of binary units is much slower than the algorithm.

Tasks using Networks of Units

As the next step in testing our algorithm, we attempted to train networks of SRV units to learn various tasks. However, our simulations indicate that such networks are very unstable. We then considered hybrid networks in which all the output units were SRV units and the rest of the units were *back-propagation* units. Back-propagation units are described by Rumelhart, Hinton and Williams [23]. The distinctive feature of networks of these units is the propagation of errors from the units in higher layers to those in lower layers. The units in each layer use these propagated errors to update their weights. The units in the output layer compute their errors by comparing their actual outputs to the desired outputs. Since the output units we used were stochastic and we did not want to provide the network with the correct output values (since the algorithm being tested was a reinforcement learning algorithm), we used the $\Delta_w(t)$ values (Equation 9) of the output units as approximations of the error at the output. On comparison with the back-propagation learning algorithm, we see that $\Delta_w(t)$ plays the same role in the SRV algorithm as the error term does for output units in the back-propagation algorithm. It therefore appears reasonable to use it as an approximation of the error. An additional step was added to the learning cycle described in Section 3, in which the errors were propagated backwards and the weights of all the back-propagation units were updated.

We present here results of simulations in which a hybrid network of SRV and back-propagation units is trained to perform two tasks. The network used is shown in Figure 11 and the tasks are described in Table 1. For both these tasks, the network was started with random weights and the learning rate β for the SRV unit was set to 0.1, while the learning rate α was set to 0.05 on the connections to the inputs and to 0.1 on the connection to the hidden unit. The learning rate for the back-propagation unit was also set to 0.1. As in the previous simulations, the training was done in epochs, during which each input pattern was presented to the network and the error computed from the output produced. The magnitude of this error was used to determine the reinforcement. The error also served as a measure of the performance of the algorithm.

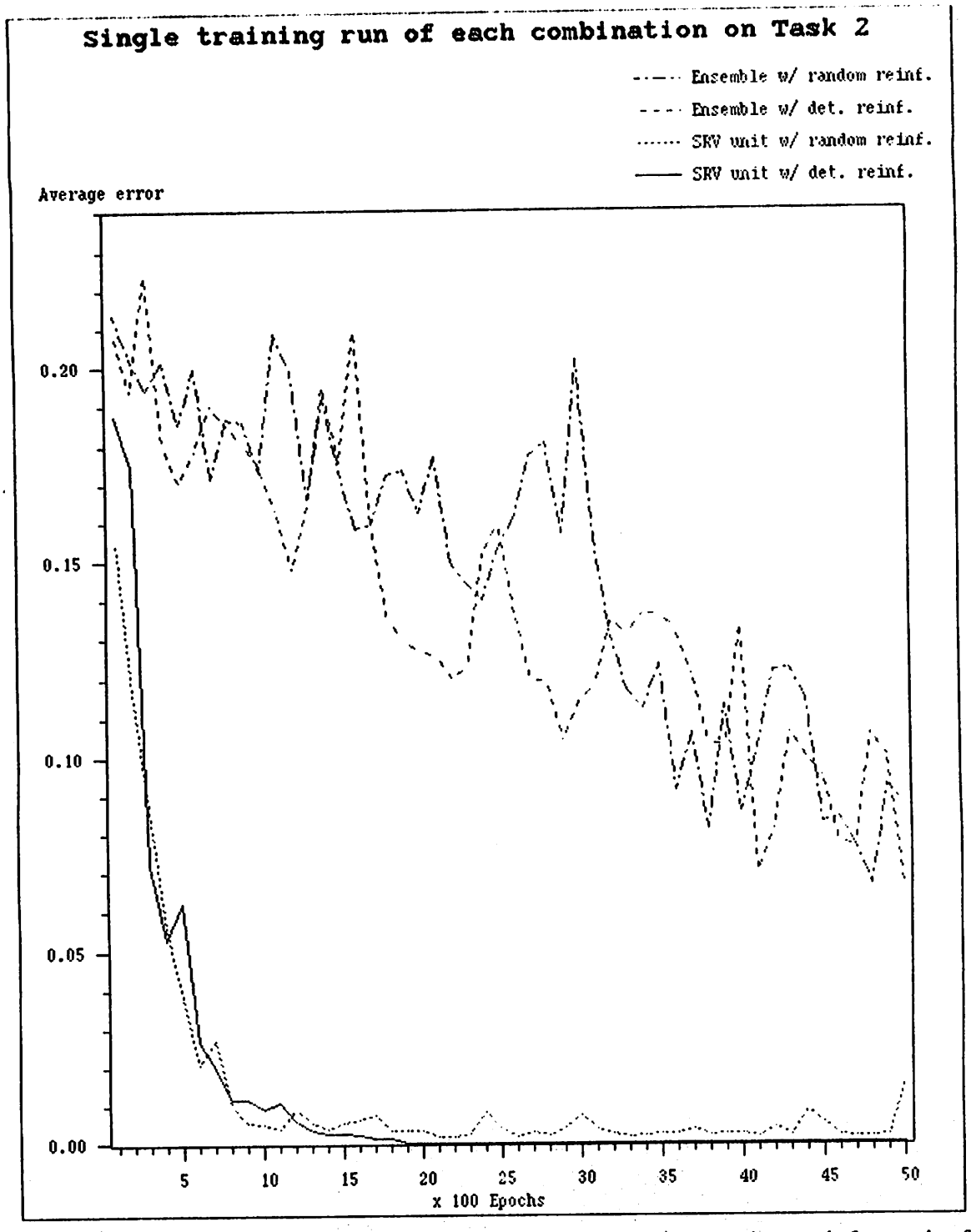

Figure 7: *Plot of the average error (over the last 100 epochs) at each epoch for each of the four combinations [real-valued and ensemble with r_1 and r_2] trained on Task 2. This plot is for a single training run.*

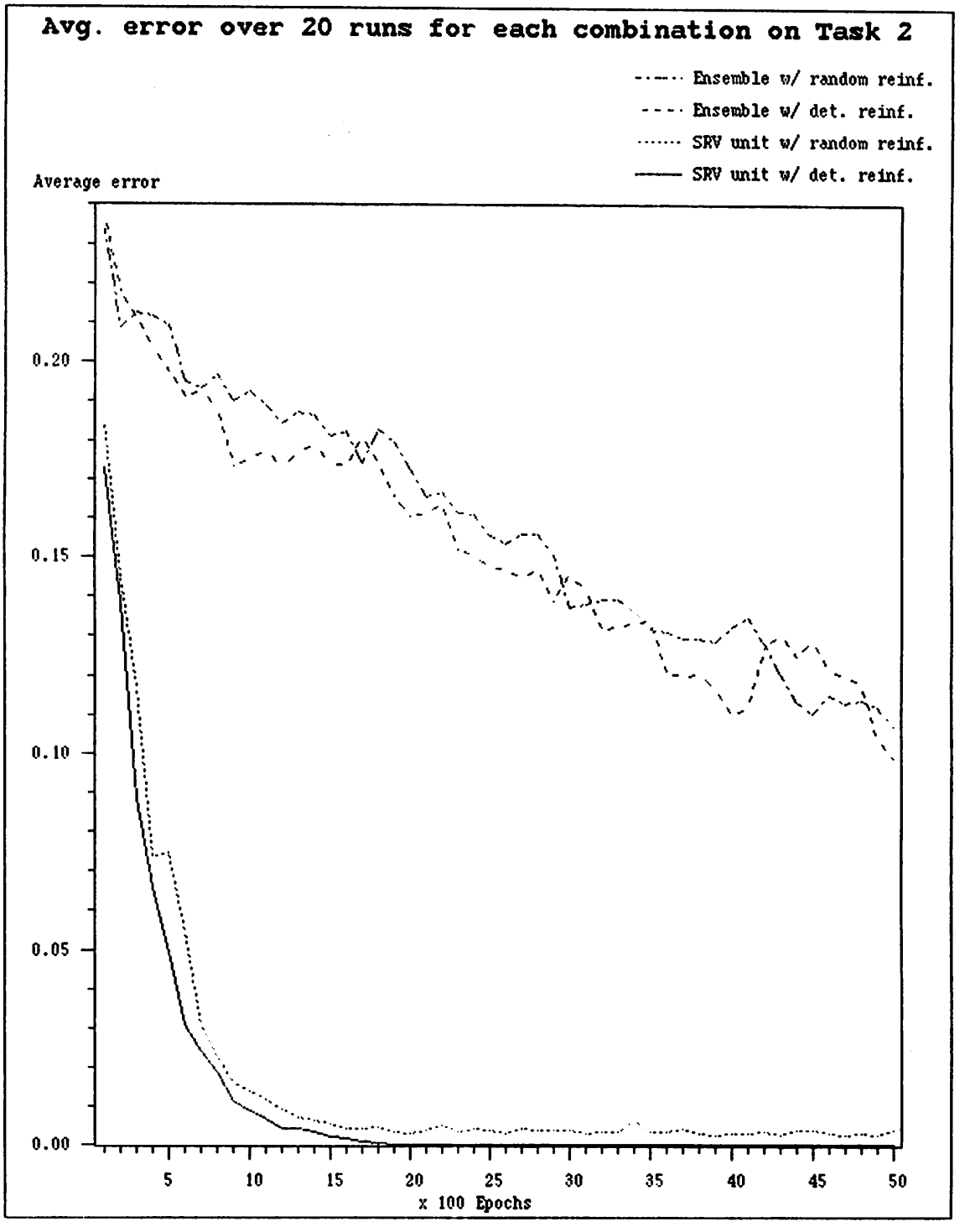

Figure 8: *Plot of the average error (over the last 100 epochs) at each epoch for each of the four combinations [real-valued and ensemble with r_1 and r_2] trained on Task 2. The error at each epoch is further averaged over 20 training runs.*

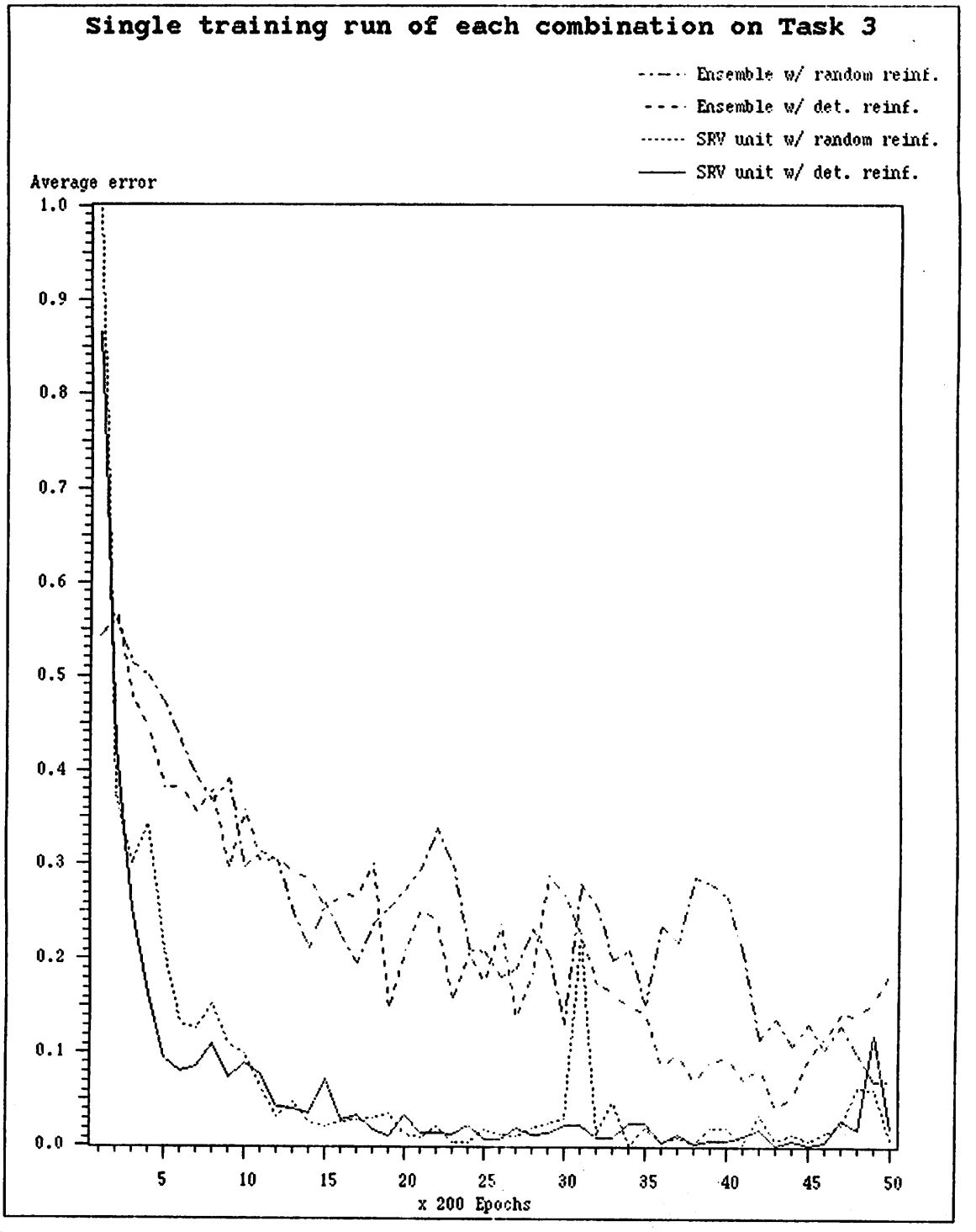

Figure 9: *Plot of the average error (over the last 200 epochs) at each epoch for each of the four combinations [real-valued and ensemble with r_1 and r_2] trained on Task 3. This plot is for a single training run.*

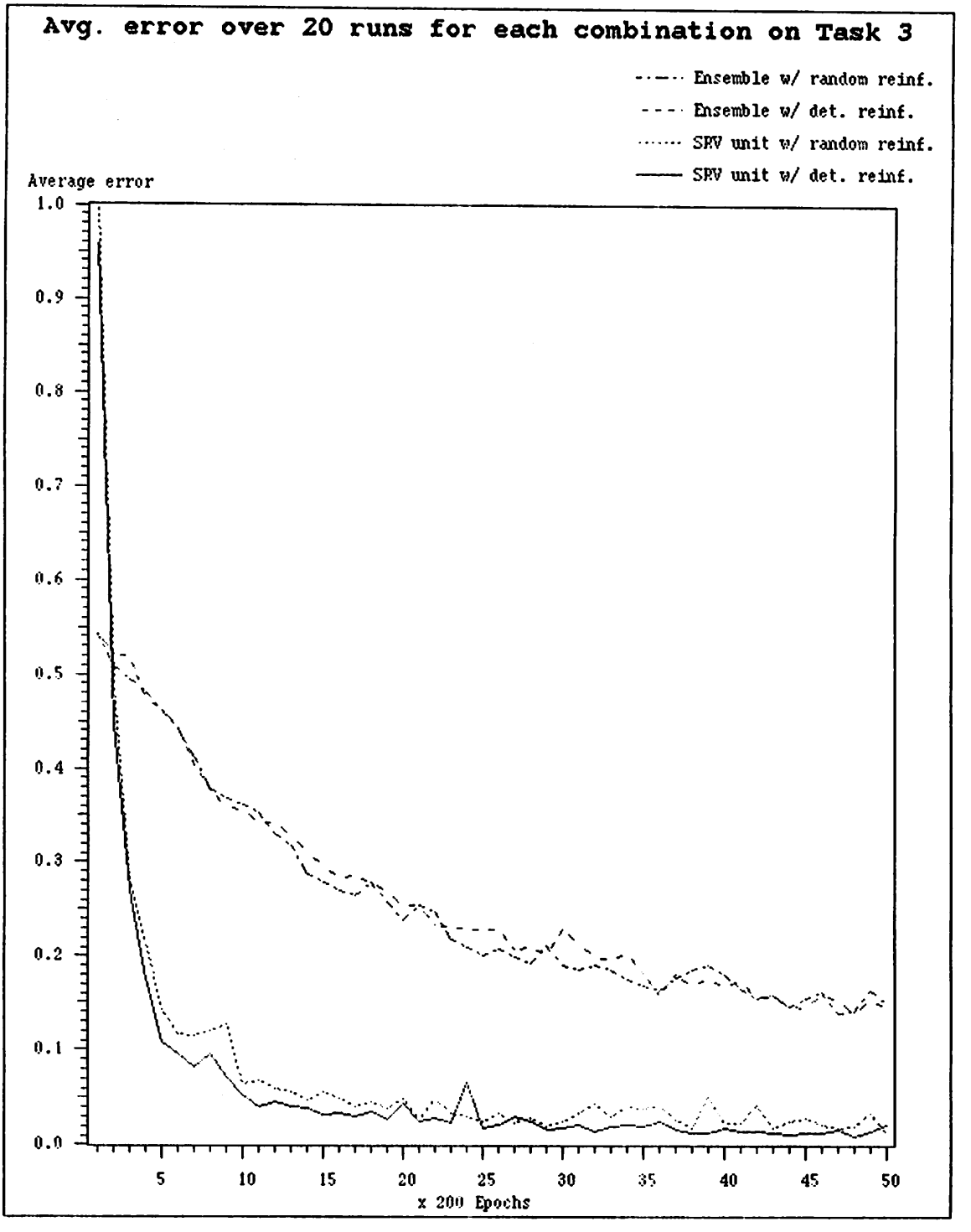

Figure 10: *Plot of the average error (over the last 200 epochs) at each epoch for each of the four combinations [real-valued and ensemble with r_1 and r_2] trained on Task 3. The error at each epoch is further averaged over 20 training runs.*

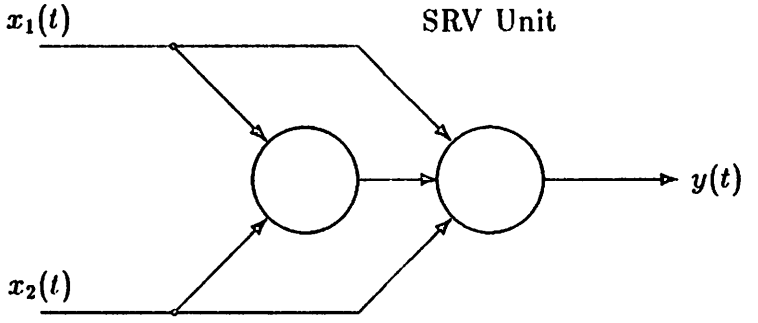

Figure 11: *The network used for the two tasks described in this section. Note that the output unit is a SRV unit, while the* hidden *unit is a back-propagation unit.*

On comparing Figure 10 and Figure 13 we can see that the network of SRV and back-propagation units learns to perform the AND task better than a single SRV unit. There is no residual error as in the case of the single unit. Furthermore, we can see from Figure 13 that the network does equally well with either kind of reinforcement (r_1 or r_2). Thus it appears that the network has a higher tolerance to noisy reinforcement as compared to a single unit.

Task 5 (the XOR task) is much harder than the previous ones because it is not a linearly separable task. Figure 14 shows the performance of the network for 10 training runs of this task. As is obvious from the figure, the network had much greater difficulty in learning this task. Unlike the other tasks, the performance did not improve uniformly as the training proceeded. Of the 20 training runs (of 50,000 epochs each) that we ran, on six training runs the network converged to the wrong weights or did not converge at all. On two others, the network had begun to converge to the right weights, although the error at the $50,000^{th}$ epoch was greater than 0.05. In the 14 training runs that the network learned the task, it took more than 39,000 epochs on the average for the error to drop below 0.05.

Since the performance on this task was not as good as that on previous tasks, we decided to see if a better reinforcement signal for the XOR task would improve the performance of the network. A rather crude definition of the XOR task would be the following statement: *The output produced when the two inputs in the stimulus are the same should be* less *than the value produced when they are different.* Based on this definition, we defined an additional factor r_{task}, that was added to the usual reinforcement, and which encoded the performance of the network over all four stimuli patterns used to define the task. For

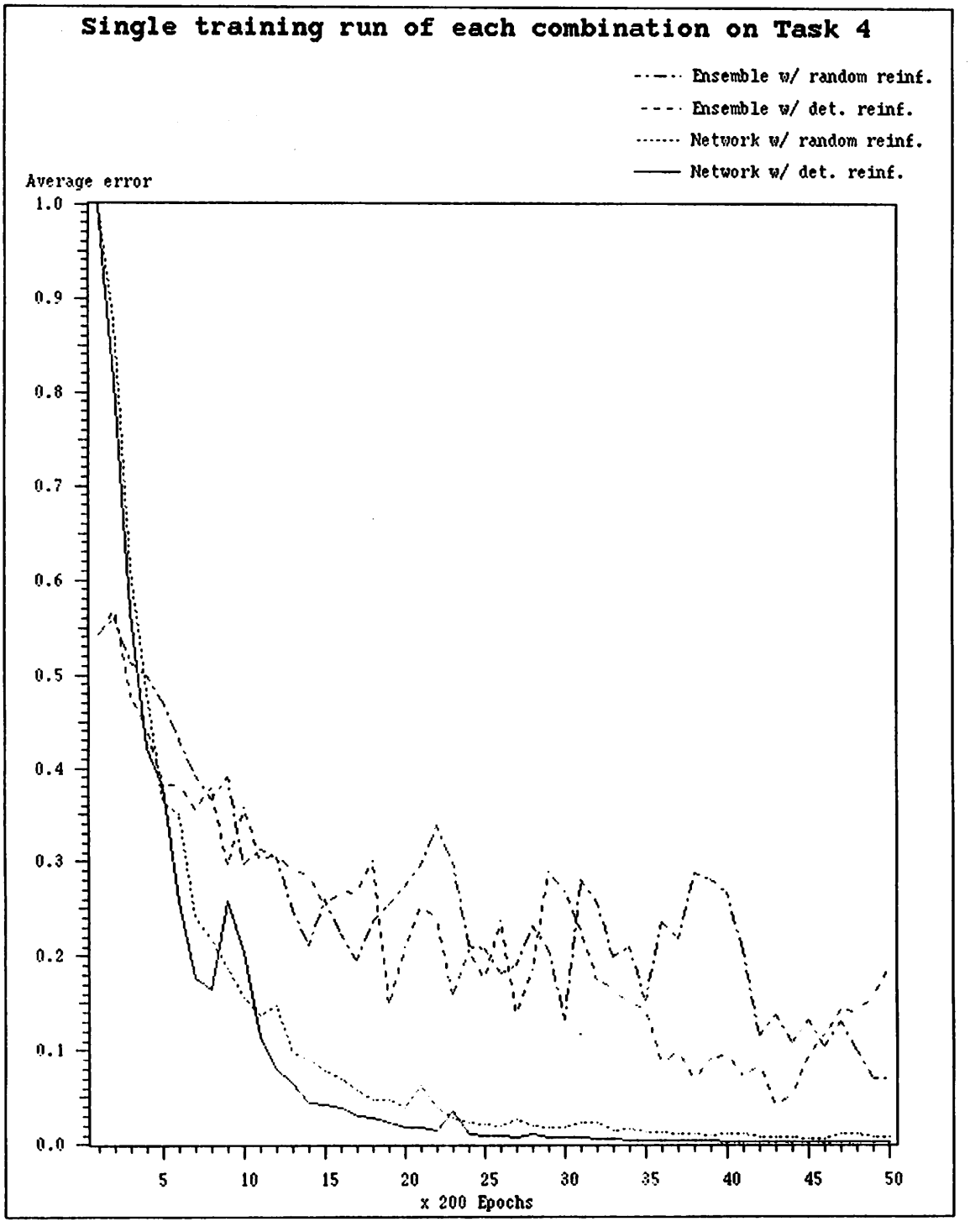

Figure 12: *Plot of the average error (over the last 200 epochs) at each epoch for the network of SRV and back-prop units trained on Task 4 using reinforcement signals r_1 and r_2. This plot is for a single training run. The corresponding plots for the ensemble of A_{R-P} units are included for comparison.*

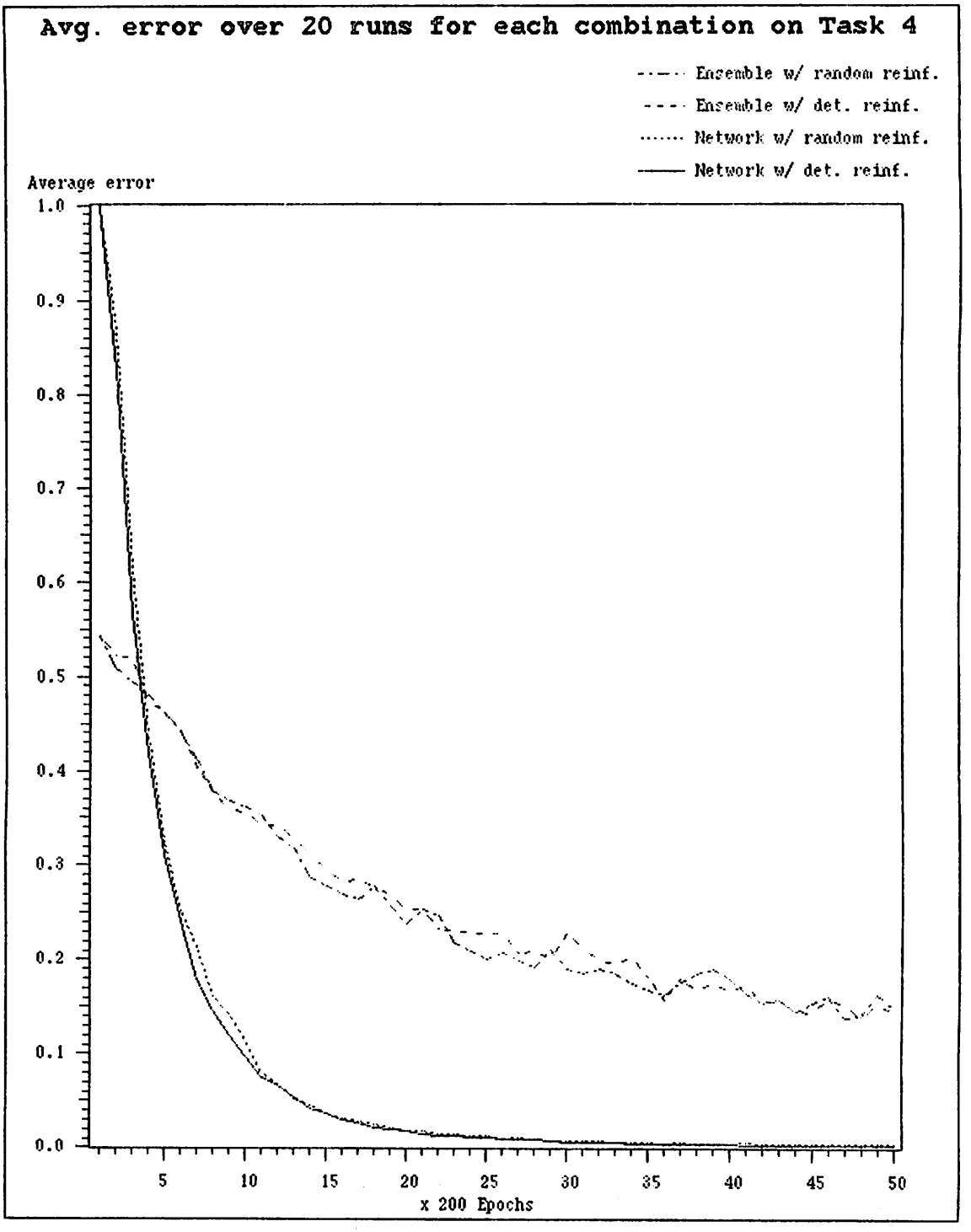

Figure 13: *Plot of the average error (over the last 200 epochs) at each epoch for the network of SRV and back-prop units trained on Task 4 using reinforcement signals r_1 and r_2. The error at each epoch is further averaged over 20 training runs. The corresponding plots for the ensemble of A_{R-P} units are included for comparison .*

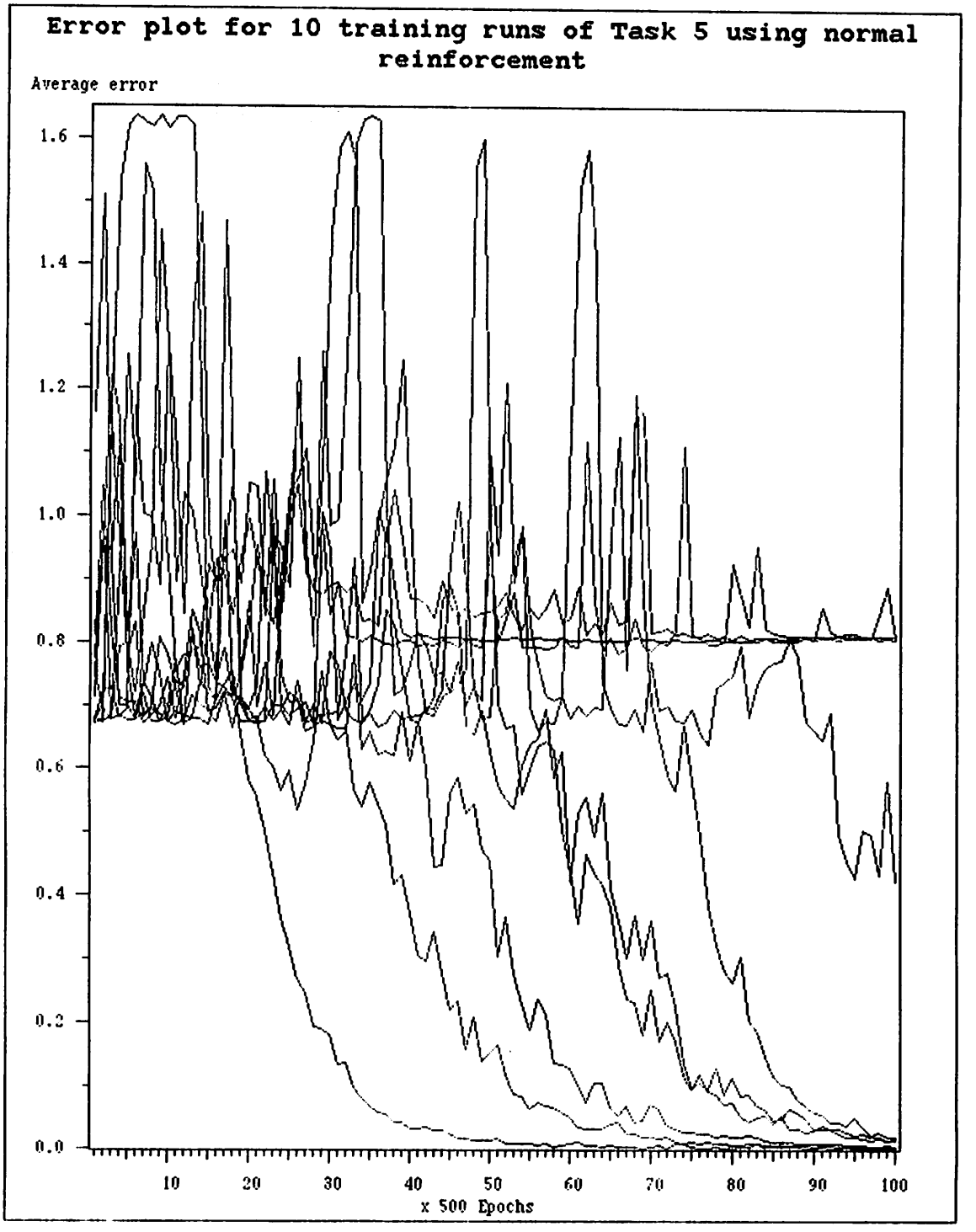

Figure 14: *Plot of the average error (over the last 500 epochs) at each epoch for 10 training runs on Task 5. The reinforcement used was* r_1.

computing r_{task}, we stored the latest outputs produced by the network for each of the four inputs. After each presentation of a stimulus, we checked to see if the condition stated above was satisfied, i.e., each of the latest outputs on similar inputs was less than each of the latest outputs on dissimilar inputs. If this was true, r_{task} was set to 0.5 and if it was not, then r_{task} was set to -0.5. The reinforcement was now computed as:

$$r_3 = \frac{r_1 + r_{task}}{2}, \tag{14}$$

where r_1 is as defined in Equation 12.

With this simple modification of the reinforcement, the performance of the network improved tremendously. Figure 15 shows a plot of 10 training runs of the XOR task using the modified reinforcement. There were two improvements in the performance of the network. Out of 20 training runs, the network now converged to the correct weights in 17 training runs. Moreover, convergence of the network was speeded up so that now the error fell below 0.05 in less than 12,000 epochs on the average, as compared to more than 39,000 with the normal reinforcement. The plot of the average error over 20 training runs for each of the two kinds of reinforcement used is presented in Figure 16. From the individual training runs with the modified reinforcement, it was also observed that the network rapidly found the right "feature" needed for solving the task and thereafter spent a lot of time in tuning the weights so as to produce the exact output values (0.1 or 0.9) required.

The performance of the SRV units, on their own and in conjunction with the back-propagation units, in the five tasks described above indicates that these units are indeed able to learn various tasks, and perform especially well in the case of linearly separable tasks. This encouraged us to test these units on a more elaborate task. We present one from the domain of robot arm control in the next section.

5. An Experiment in Robot Arm Control

A domain that presents many challenges for connectionist approaches is robotics. This field has presented a number of difficult problems for which no completely satisfactory solutions have been found. In this section we will address one problem that is of great significance in robot arm control. Most tasks in robotics involve the manipulation of objects specified in terms of world coordinates. However, a robot arm is usually controlled by adjusting the angles of the various joints of the arm. Thus, in order to perform a specified task, it becomes necessary for the controller to compute from a given position in

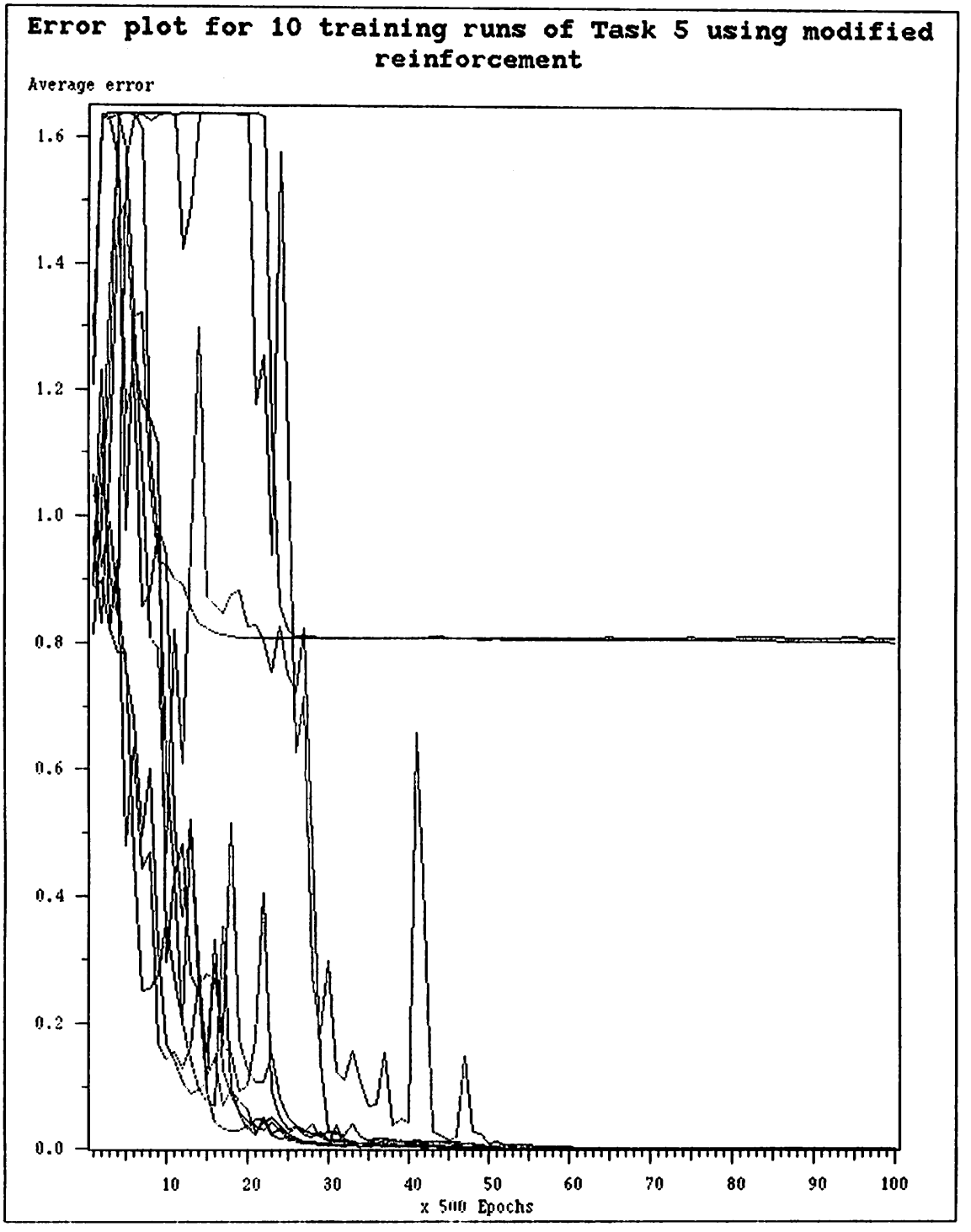

Figure 15: *Plot of the average error (over the last 500 epochs) at each epoch for 10 training runs on Task 5. The reinforcement used was* r_3.

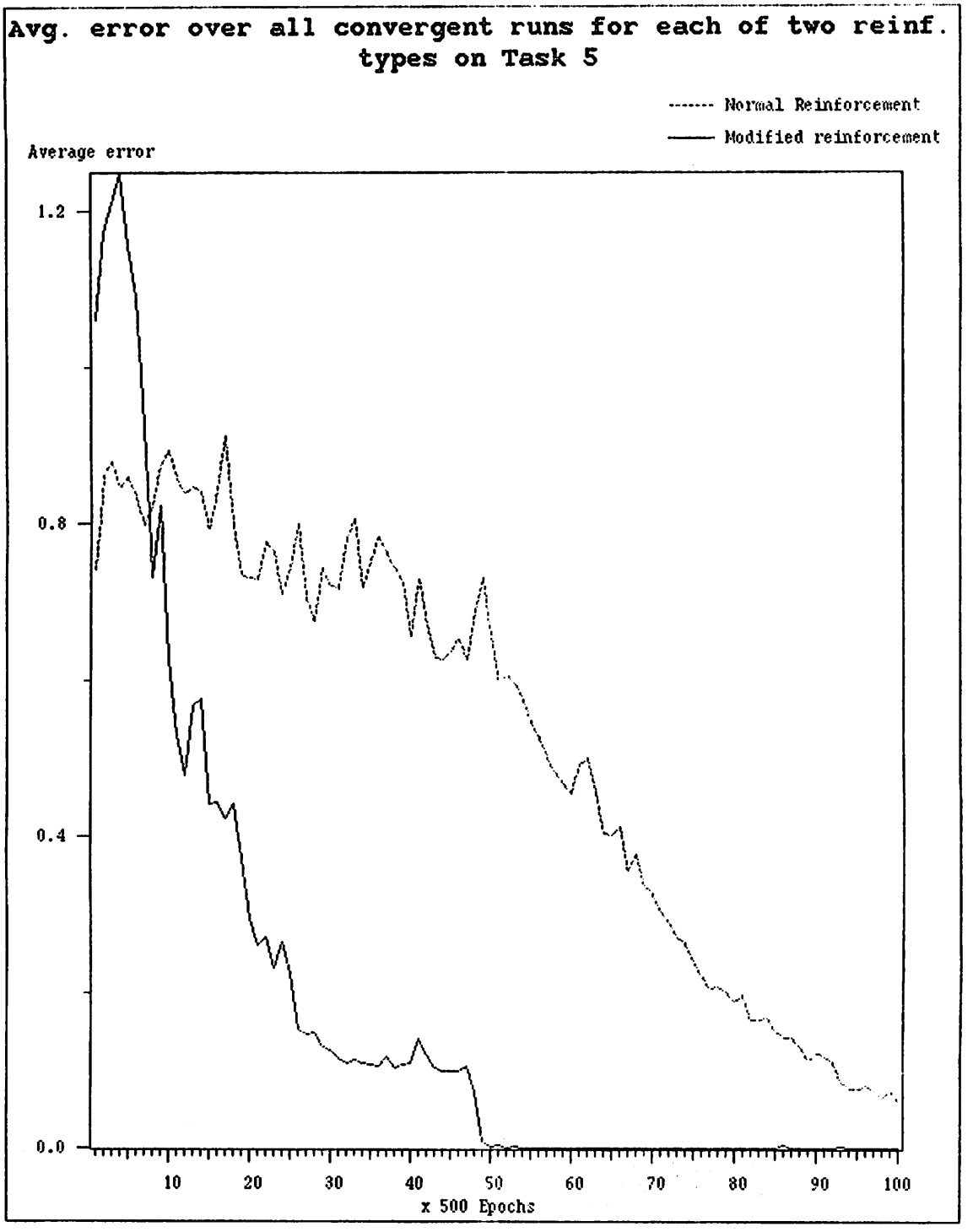

Figure 16: *Plot of the average error (over the last 500 epochs) at each epoch for 20 training runs on Task 5. The plot shows the relative performances of the network when using the normal reinforcement r_1 and when using the modified reinforcement r_3.*

world coordinates, a set of joint angles that can achieve that position. This is the problem of inverse kinematics.

In general, the inverse kinematics for a robot arm is nonlinear and configuration dependent. Traditional approaches to solving this problem involve modeling the kinematics of the arm through a set of equations. The inverse transformation is then obtained by computing a closed-form solution to these equations [22]. However, only certain restricted classes of robots allow closed-form inverse kinematic solutions. Another approach is to use iterative procedures for solving a system of nonlinear equations. The main draw-back with these procedures is that they are heavily computation bound and tend to have points of singularity, i.e., there are points in the solution space that cannot be computed by these methods. Moreover, depending on the kinds of degrees of freedom available for a task and especially in the presence of excess degrees of freedom, infinitely many solutions may exist for the inverse transform. Traditional methods usually do not specify which solution to use and in procedures that do pick a single solution, the selection is usually based on arbitrary criteria or on the idiosyncrasies of the method. It therefore becomes even more difficult to ensure that the excess degrees of freedom are effectively utilized in performing a given task [10].

In recent times, several researchers in connectionism have presented networks that have been trained to compute the inverse kinematics of robot arms. Most of these researchers ([16,17,29] among others) use supervised learning methods wherein the network is trained using pairs of vectors specifying the target coordinates and the corresponding "desired" joint angles. However, in most control tasks, it is difficult to determine *a priori* what the optimal output vector should be for a given input vector. This is especially true in the presence of excess degrees of freedom. Thus, the mapping learned by these networks quite often does not effectively utilize the degrees of freedom in the system. An approach to solving this problem that is within the supervised learning paradigm is presented by Jordan [15]. His approach involves specifying several additional constraints on the task and also introducing don't-care conditions in the output vector. These additional constraints and the less restrictive conditions on the output vector make the task well-defined once again, making it possible to use supervised learning techniques to learn these tasks. However, knowing what constraints are suitable for a task and knowing what outputs do not matter in a given situation are again non-trivial problems in many tasks. As opposed to this, it might be easier to *evaluate* the performance of a task and deliver a reinforcement value that is appropriate. Thus, reinforcement learning may be suitable in situations where the learning system's performance can be evaluated easily but the task itself cannot be specified in great detail.

In this section we present an experiment in reinforcement learning in which a network of units was trained to learn the inverse kinematic transform of a simulated 3-DOF robot

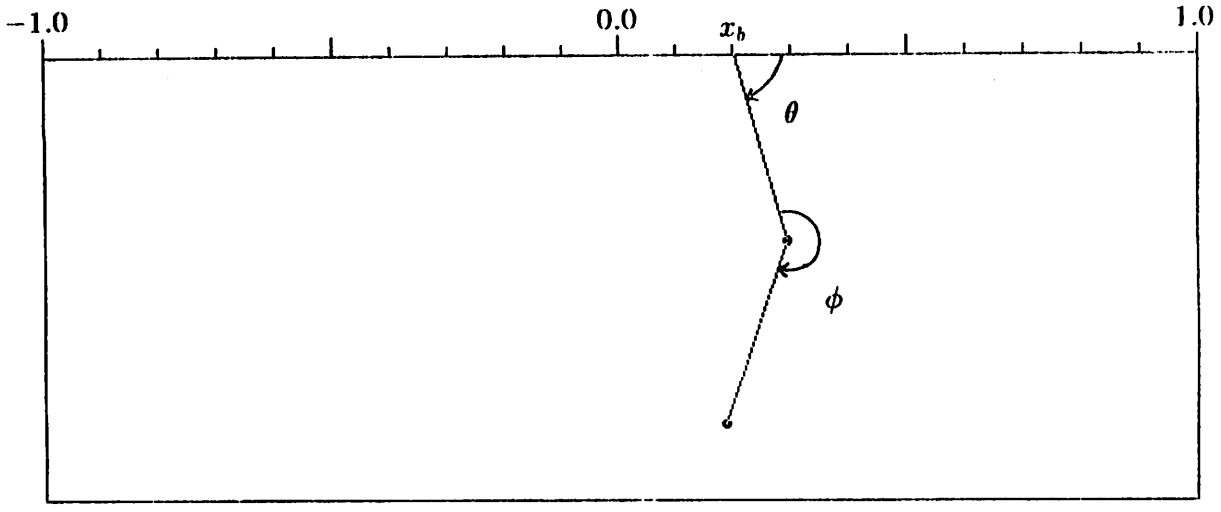

Figure 17: *The simulated robot arm in its workspace. The three degrees of freedom of the arm are the translation along the x-axis x_b, the rotation of the first link θ, and the rotation of the second link ϕ.*

arm. The goal of the experiment and the setup used for conducting the simulation are described first, followed by a description of the performance obtained.

The Task and the Setup

For this task, we used a simulated robot arm with three degrees of freedom. The arm had a planar workspace and consisted of two links of equal length mounted an a base. Figure 17 shows the arm in its workspace. The workspace is described by x-y coordinates in $[-1.0, 1.0] \times [0.0, 1.0]$. The three degrees of freedom of the arm are (1) the base of the arm can move along the x-axis between -0.33 and 0.33, (2) the first link of the arm can be positioned at any angle between 0° and 180° from the x-axis, and (3) the second link can be positioned at any angle between 90° and 270° relative to the first link.

Thus the position of the arm at any instant can be specified by giving the values x_b, the location of the base, θ, the angular displacement of the first link and ϕ, the angular displacement of the second link. These three together form the joint-space coordinates of the arm. The arm itself was simulated using the equations for its forward kinematics. These equations are relatively easy to derive and enable us to compute the x-y coordinates of the endpoint of the arm, given its coordinates in joint space. Although it is possible, given the above degrees of freedom, for the endpoint of the arm to be located at a point outside the specified workspace, for the purposes of this experiment we only use points

inside the workspace.

The task to be learned is: *Given the current position of the arm in joint-space coordinates and a desired spatial location x_d, determine the new joint-space coordinates for the arm that will result in the endpoint of the arm having x_d as its x-coordinate.* It should be noted that the task above is underconstrained, as no restriction is made regarding the y-coordinate of the endpoint of the arm. Thus there are two excess degrees of freedom for this task (for most of the workspace). A further goal is to learn to perform the task as efficiently as possible. By "efficiently" we mean "in a manner that involves as little change in the joint-space coordinates as possible". Thus, the final configuration of the arm should depend on the initial configuration, and the network has to be able to produce several arm configurations for a given target location. This requires the excess degrees of freedom of the arm to be effectively utilized in performing the task. We felt that the use of stochastic units with reinforcement learning and the training scheme described below might enable the network to learn the task in such a manner.

The network used for this task is shown in Figure 18. As in the case of the network tasks described in Section 4, the network consists of two types of units. The output units are SRV units, while all the other units are back-propagation units. As described before, the SRV units send back their $\Delta_w(t)$ values as errors, which are used by the back-propagation units to update their weights. The training was performed as a sequence of epochs. For each epoch, a random configuration for the arm and a random value for x_d were chosen. The values for x_d were restricted to the interval $(-0.9, 0.9)$. The epoch started with the current joint coordinates of the arm and x_d being given as input to the network. A new set of joint coordinates was then computed as output by the network. The outputs of the SRV units were scaled from (0,1) to the appropriate intervals of values for each of x_b, θ and ϕ. These values were fed to the simulator for the arm to determine the x-y coordinates of its endpoint. The network then received a reinforcement from the environment based on how close the endpoint was to the desired x location. As the last step of each epoch, the network updated its weights using the reinforcement value, and the training cycle was repeated with a new x_d and a new set of joint coordinates for the arm.

The reinforcement was computed as a function of the current and previous distances (dist and prevdist) of the endpoint from the desired x-coordinate, x_d, as follows:

$$r(t) = \begin{cases} \left.\begin{array}{ll} 1 & \text{if prevdist} = 0 \\ 0.5 + \frac{prevdist - dist}{2 * prevdist} & \text{otherwise} \end{array}\right\} & \text{if dist} \leq \text{prevdist} \\ \left.\begin{array}{ll} 1 & \text{if dist} = 0 \\ 0.5 - \frac{dist - prevdist}{2 * dist} & \text{otherwise} \end{array}\right\} & \text{otherwise} \end{cases} \tag{15}$$

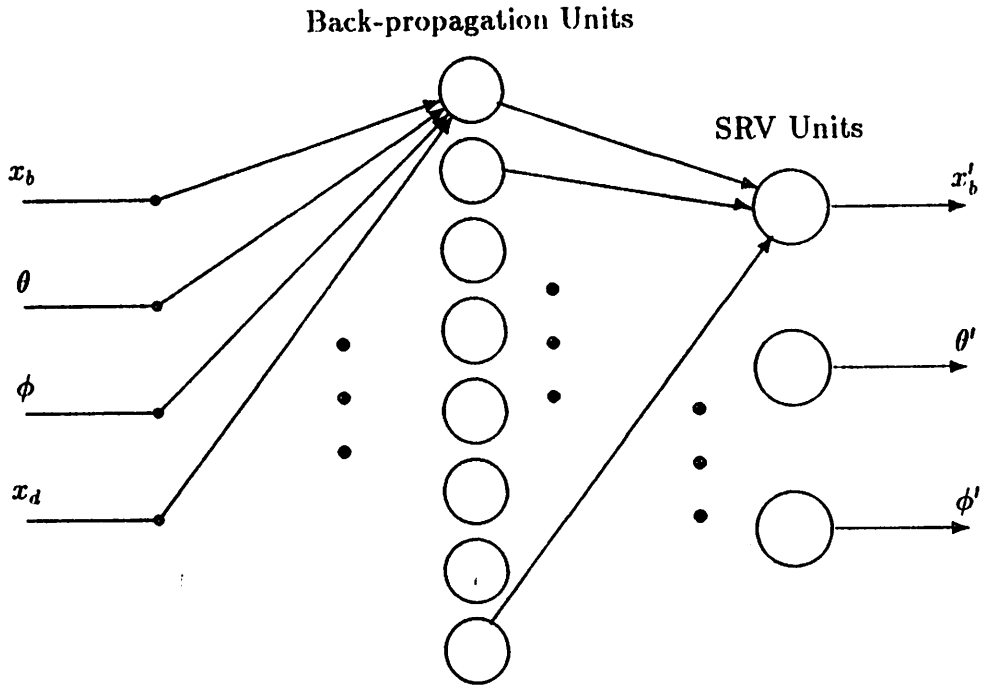

Figure 18: *The network used to learn the task of controlling the simulated robot arm. The output layer of the network is composed of SRV units and the rest of the units are back-propagation units. Each of the eight back-propagation unit gets all four inputs and is connected to all three SRV output units.*

Thus, the magnitude of reward (punishment) was proportional to the fraction of the distance towards (away from) the target location covered in a single step.

As an illustration of the learning process, Figure 19 shows the arm positions tried over 50 epochs using the *same* target location. The desired x location is indicated by the vertical line ($x_d = -0.384$). The network gets the maximum reinforcement when the endpoint is anywhere on this line. Note that the stochastic nature of the output units quite literally causes the network to "explore" the space of joint coordinates and the reinforcement enables it to determine which direction to move the coordinates so that the task goal is achieved.

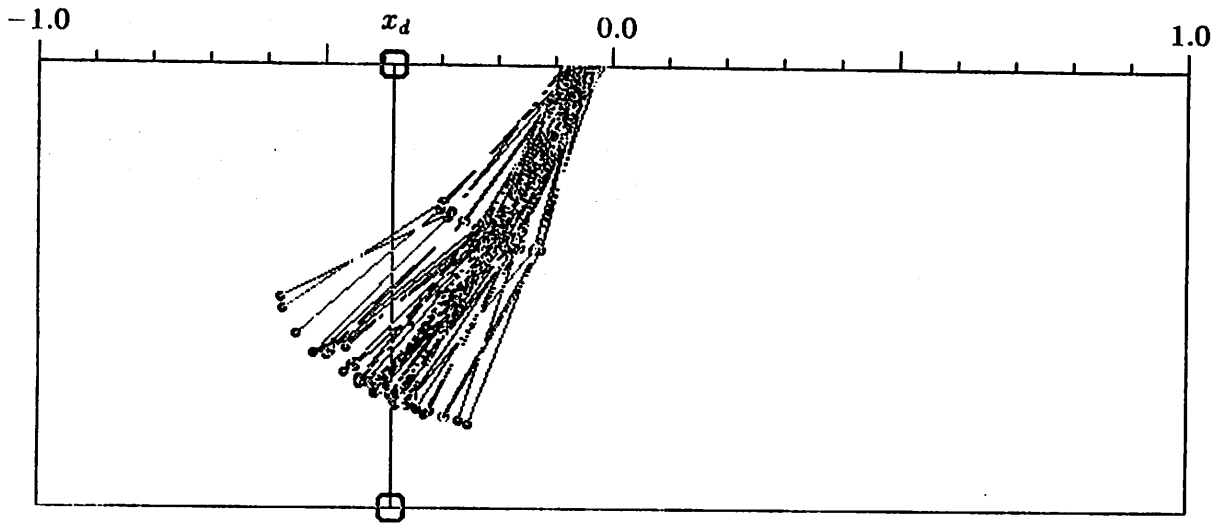

Figure 19: *Illustration of the learning process of the network used to control the arm. The vertical line shows the desired x location. The arm was trained for 50 learning epochs using this x location. The figure shows the arm positions at the end of each of those 50 epochs.*

Performance

In order to determine the performance of the network, we conducted a series of tests at various stages of learning. In each test, we presented the network with 2000 random target locations and arm configurations and allowed it to control the movement of the arm. When the arm reached its final position (usually in 2 to 3 cycles through the network), we recorded the x-coordinate of the endpoint. The tests were run after 10,000, 20,000, and 30,000 training epochs.

This data is presented in the following fashion. The x-coordinate range was divided into 16 intervals, $[-1, -0.7)$, $[-0.7, -0.6)$, $\cdots$ $[0.6, 0.699)$, $[0.7, 1]$. For each interval, for each test, all the data points with the target location in the interval were collected and the average error, $(target - x)$, was computed. This average error for the three tests is plotted in Figure 20. From the figure, it is clear that for most of the range of the target location, the error between the desired location and the actual location achieved using the network to control the arm is less than 0.05. It is also clear that in the extremities of the range, the arm overshoots the target location. For example, when the target is less than -0.7, the error is about 0.2, indicating that the actual x location is around -0.9. Similarly, when the target location is greater than 0.7, the error is -0.2 indicating that the arm location is about 0.9.

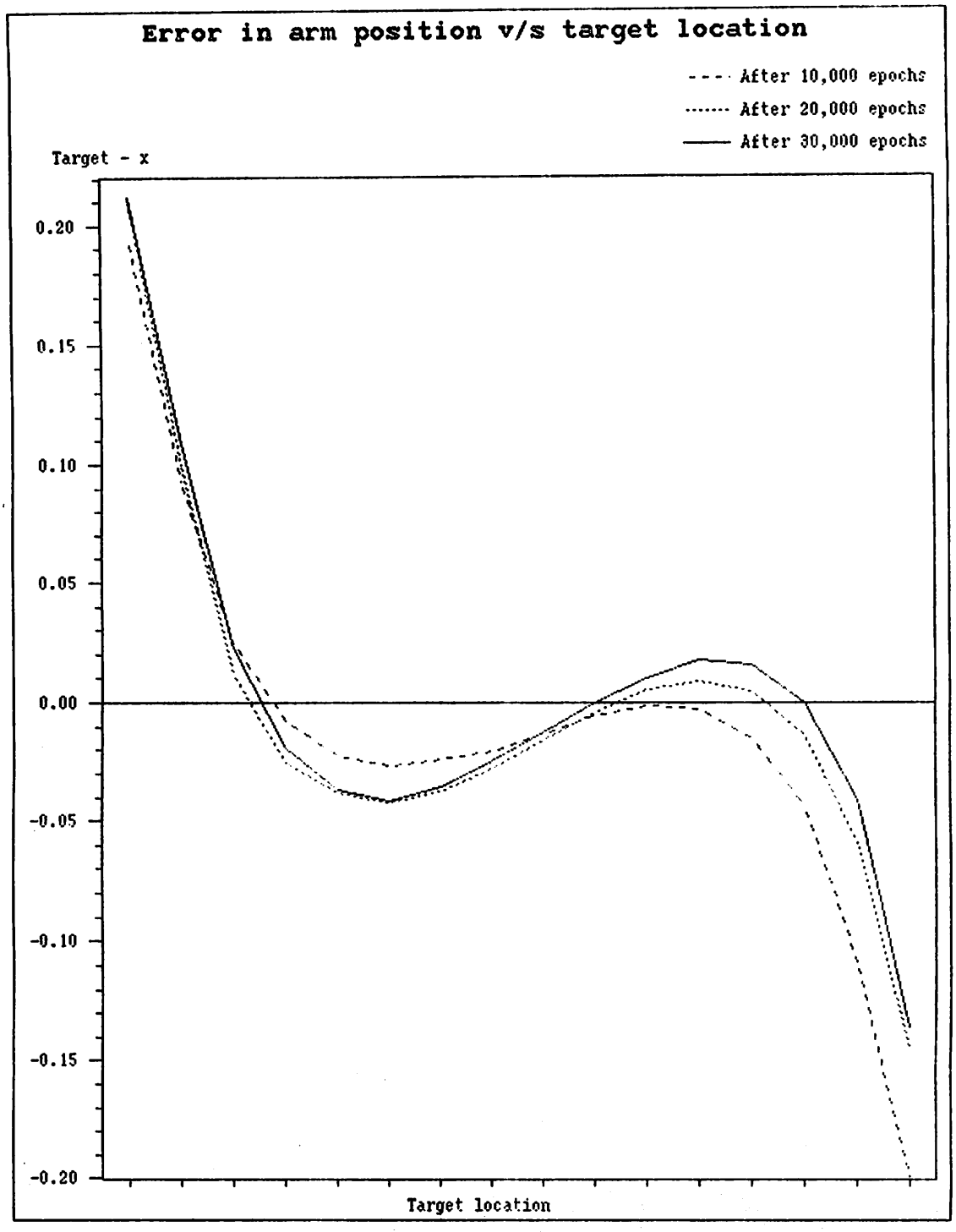

Figure 20: *Performance of the arm after 10,000, 20,000, and 30,000 training epochs. The figure shows the difference between the target location and the arm position achieved by the network, averaged over all target locations in an interval, for 16 intervals covering the range of possible target locations from* -1.0 *to 1.0.*

From this we conclude that the network has been able to successfully learn the inverse kinematics of the arm, albeit with some error in the positioning of the endpoint. We feel that this is an important step towards developing a general purpose mechanism for learning the inverse kinematics of arbitrary robot arms. We also feel that with further training, the error in positioning could be kept below any specified level. A shortcoming of the network's performance, however, is the lack of any effect of the initial arm location on the final configuration computed for a desired x-location. As we have mentioned before, our hope was that the network would learn a *position-dependent transform* for the arm so that it can move from the current arm location to the desired arm location as *efficiently* as possible. By efficiently, we mean in a manner so that the arm coordinates are altered as little as possible. However, from observations of the arm in tests, we find that this has not happened in the simulation described. This means the network always uses a fixed arm configuration in order to achieve a particular target location and hence does not effectively utilize the excess degrees of freedom. Any reasonable mapping should take advantage of the excess degrees of freedom. We feel, however, that we have not experimented enough with this approach and we hope that modifications to the network and to the reinforcement might make it possible to learn this kind of behavior.

6. Conclusions and Comments

The simulations described in the previous sections indicate that the algorithm developed for stochastic reinforcement learning in real-valued units performs well in simple learning tasks. The algorithm also does not suffer from the saturation problem described by Albus (see page 2). In the case of the more difficult XOR task, we found that the algorithm can learn the task, albeit slowly. Analysis of the learning process suggested that the use of a more informative reinforcement signal, viz. one that encoded the performance on the task *as a whole*, should yield better performance in terms of speed of learning and robustness. This was verified empirically for the XOR task for which we showed a speedup of learning by a factor of 3 when a simple modification was made in the reinforcement signal.

As for the experiment in learning the inverse kinematics, we feel that we have not investigated this approach well enough to be able to draw any major conclusions. The preliminary results are very encouraging, as we could get the network to learn the inverse kinematics for the arm to a reasonable degree of accuracy. However, the problem of effective utilization of excess degrees of freedom remains. There are several directions in which we could carry out future investigations. These include trying different types of network architecture, reinforcement signals, and state variables for the arm.

Possible directions of future work on real-valued units include modifications of the algorithm to incorporate better predictors of the reinforcement and "batching" the gradient computation, i.e., averaging the $\Delta(t)$ over several epochs for the same input pattern as has been done by Barto et al. [6]. It remains to be seen how much can be gained from using real-valued units that can perform stochastic reinforcement learning.

References

[1] Ackley, D. H., Hinton, G. H., & Sejnowski, T. J. (1985). A learning algorithm for Boltzmann machines. *Cognitive Science*, 9, 147-169.

[2] Albus, J. S. (1975). *Brains, Behavior, and Robotics.* Peterborough, N. H.: BYTE books.

[3] Atkinson, R. C., Bower, G. H., & Crothers, E. J. (1965). *An Introduction to Mathematical Learning Theory.* New York: Wiley.

[4] Barto, A. G. (1985). Learning by statistical cooperation of self-interested neuron-like computing elements. *Human Neurobiology*, 4, 229-256.

[5] Barto, A. G., & Anandan, P. (1985). Pattern recognizing stochastic learning automata. *IEEE Transactions on Systems, Man and Cybernetics*, 15, 360-374.

[6] Barto, A. G., & Jordan, M. I. (1987). Gradient following without back-propagation in layered networks. *Proceedings of the IEEE First Annual Conference on Neural Networks*, San Diego, California.

[7] Barto, A. G., Sutton, R. S., & Brouwer, P. S. (1981). Associative search network: A reinforcement learning associative memory. *Biological Cybernetics*, 40, 201-211.

[8] Bush, R. R., & Estes, W. K. (Eds.). (1959). *Studies in Mathematical Learning Theory.* Stanford, California: Stanford University Press.

[9] Bush, R. R., & Mosteller, F. (1955). *Stochastic Models for Learning.* New York: Wiley.

[10] Fenton, R. G., Benhabib, B., & Goldenberg, A. A. (1985). Optimal point to point control of robots with redundant degrees of freedom. *Proc. ASME Winter Annual Meeting*, 13, 93-109.

[11] Farley, B. G., & Clark, W. A. (1954). Simulation of self-organizing systems by digital computer. *I.R.E. Transactions on Information Theory*, PGIT-4.

[12] Fu, K. S., & Waltz, M. D. (1965). A heuristic approach to reinforcement-learning control systems. *IEEE Transactions on Information Theory*, 9, 390-398.

[13] Harth, E., & Tzanakou, E. (1974). Alopex: A stochastic method for determining visual receptive fields. *Vision Research*, 14, 1475-1482.

[14] Hull, C. L., (1952). *A behavior system: An introduction to behavior theory concerning the individual organism.* New Haven: Yale University Press.

[15] Jordan, M. I. (1988). *Supervised learning and systems with excess degrees of freedom.* COINS Tech. Report 88-27. Amherst: Dept. of Computer and Info. Sciences, University of Massachusetts.

[16] Kawato, M. (1987). Adaptation and learning in control of voluntary movements in the central nervous system. To appear in *Advanced Robotics*, Vol. 2.

[17] Kuperstein, M. (1987). Adaptive visual-motor coordination in multijoint robots using parallel architecture. *Proc. of the IEEE Conf. on Robotics and Automation*, 1595-1602.

[18] McMurtry, G. J. (1970). Adaptive optimization procedures. In J. M. Mendel & K. S. Fu (Eds.), *Adaptive, Learning and Pattern Recognition Systems: Theory and Applications.* New York and London: Academic Press.

[19] Mendel, J. M. & McLaren, R. W. (1970). Reinforcement-learning control and pattern recognition systems. In J. M. Mendel & K. S. Fu (Eds.), *Adaptive, Learning and Pattern Recognition Systems: Theory and Applications.* New York and London: Academic Press.

[20] Narendra, K. S., & Lakshmivarahan, S. (1977). Learning automata - A critique. *Journal of Cybernetics, and Info. Science*, 1, 53-65.

[21] Narendra, K. S., & Thathachar, M. A. L. (1974). Learning automata - A survey. *IEEE Transactions on Systems, Man, and Cybernetics*, 4, 323-334.

[22] Paul, R. P., Shimano, B., & Mayer, G. E. (1981). Kinematic control equations for simple manipulators. *IEEE Transactions on Systems, Man and Cybernetics*, 11, 456-460.

[23] Rumelhart, D. E., Hinton, G. E. , & Williams, R. J. (1986). Learning internal representations by error propagation. In D. E. Rumelhart & J. L . McClelland (Eds.), *Parallel Distributed Processing: Explorations in the microstructure of cognition. Vol. 1: Foundations.* Cambridge: MIT Press/Bradford Books.

[24] Skinner, B. F. (1938). *The behavior of organisms: an experimental analysis.* New York: D. Appleton Century.

[25] Sutton, R. S. *Ballistic bug.* Unpublished working paper, June, 1982.

[26] Sutton, R. S. *Temporal aspects of credit assignment in reinforcement learning.* Ph.D. dissertation, University of Massachusetts, Amherst, 1984.

[27] Tsetlin, M. L. (1973). *Automata theory and modeling of biological systems.* New York: Academic Press.

[28] Widrow, B., & Hoff, M. E. (1960). Adaptive switching circuits. *1960 WESCON Convention Record Part IV*, 96-104.

[29] Widrow, B., & Stearns, S. D. (1985). *Adaptive signal processing.* Englewood Cliffs, NJ: Prentice-Hall.

[30] Williams, R. J. (1986). *Reinforcement learning in connectionist networks: A mathematical analysis.* (Tech. Report 8605). La Jolla: University of California, San Diego, Institute for Cognitive Science.