Neural Networks, Vol. 3, pp. 671–692, 1990
Printed in the USA. All rights reserved.

0893-6080/90 $3.00 + .00
Copyright © 1990 Pergamon Press plc

ORIGINAL CONTRIBUTION

A Stochastic Reinforcement Learning Algorithm for Learning Real-Valued Functions

VIJAYKUMAR GULLAPALLI

Department of Computer and Information Science, University of Massachusetts

(*Received 26 September 1989; revised and accepted 31 January 1990*)

Abstract—*Most of the research in reinforcement learning has been on problems with discrete action spaces. 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 using a connectionist network. We define stochastic units that compute their real-valued outputs as a function of random activations generated using the normal distribution. Learning takes place by using our algorithm to adjust the two parameters of the normal distribution 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 stochastic real-valued units that learns to perform an underconstrained positioning task using a simulated 3 degree-of-freedom robot arm.*

Keywords—Neural networks, Associative reinforcement learning, Learning algorithm, Stochastic automata, Real-valued functions, Shaping, Robotics, Neurocontrol.

1. INTRODUCTION

Connectionist learning methods have been divided into three main paradigms: unsupervised learning; supervised learning; and reinforcement learning (e.g., Hinton, 1987). Unsupervised learning methods do not rely on an external teacher to guide the learning process. Instead, the teacher can be considered to be built into the learning method. Unlike the unsupervised learning paradigm, both the supervised and the reinforcement learning paradigms require an external teacher to provide training signals that guide the learning process. The difference between these two paradigms arises from the kind of training signals the teacher provides. In the supervised learning paradigm, the teacher provides the learning system with desired outputs for each given input. Learning involves memorizing these desired outputs by minimizing the discrepancy between the actual outputs of the system and the desired outputs. In contrast with the supervised learning paradigm, the role of the teacher in reinforcement learning is more *evaluative* than *instructional*. Sometimes called a critic because of this role, the teacher provides the learning system with a scalar evaluation of the system's performance of the task according to some performance measure. The objective of the learning system is to improve its performance, as evaluated by the critic, by generating appropriate outputs. Reinforcement learning therefore involves two operations: discovering the right outputs for a given input; and memorizing those outputs.

Since the scalar evaluation provided by the critic in reinforcement learning conveys much less training information than the desired outputs provided in supervised learning, reinforcement learning is often slower than supervised learning. Nevertheless, the ability to discover solutions to problems makes reinforcement learning important in situations where lack of sufficient structure in the task definition makes it difficult to define a priori the desired outputs for each input, as required for supervised learning. In such cases, reinforcement learning systems can be used to learn the unknown desired outputs by providing the system with a suitable evaluation of its performance.

Acknowledgements: 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, Richard Sutton, Rahul Simha, and Steve Bradtke for many useful comments and suggestions. This research was funded in part by the Air Force Office of Scientific Research, Bolling AFB, through grant AFOSR-87-0030.

Requests for reprints should be sent to Vijaykumar Gullapalli, COINS Dept., University of Massachusetts, Amherst, MA 01003.

Reinforcement learning has long been studied by psychologists interested in human and animal behavior (under the guise of operant or instrumental conditioning) (Skinner, 1938), and by mathematical learning theorists (see e.g., Bush & Mosteller, 1955; Bush & Estes, 1959; Atkinson, Bower, & Crothers, 1965). Other researchers who have investigated reinforcement learning problems include learning automata theorists (see e.g., Narendra & Thathachar, 1989; Narendra & Lakshmivarahan, 1977), control theorists (see e.g., Fu & Waltz, 1965; Mendel & McLaren, 1970), and researchers studying function optimization (see e.g., McMurtry, 1970). Connectionist approaches to reinforcement learning have been studied extensively by Barto and his colleagues (for example, Barto & Anandan, 1985; Sutton, 1984; Barto, Sutton, & Watkins, to appear), and by others (for example, Williams, 1986).

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 (1975). 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, in general, 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 simulus 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 has the maximum expectation of 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* (Hull, 1952). 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* (Skinner, 1938), 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 (1952) 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 (1982) 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 and Tzanakou (1974), in a somewhat equivalent approach, designed an algorithm, called Alopex, to produce real values that are proportional to a bias with random noise added to it. In the original algorithm, the noise had a fixed distribution, although in more recent versions feedback is used to adjust the noise distribution. Farley and Clark (1954) also describe a scheme in which the noise level is varied based on changes in performance over previous time steps. More recently, Alspector, Allen, Hu, & Satyanarayana (1987) described a reinforcement learning algorithm based on the Boltzmann machine learning algorithm (Ackley, Hinton, & Sejnowski, 1985), in which the noise in the activation of their stochastic units is controlled so as to keep the units active for a reasonable fraction of the time. In addition to the frequency of activation of the units, they also used measures such as the sum of the magnitudes of weights into a unit and the past history of reinforcement received by the network to determine the amplitude of the noise. Finally, the possibility of designing stochastic learing automata that use multiparameter distributions to generate their outputs was examined by Williams (1986). As an example, Williams considered the normal distribution and suggested guidelines as to how the mean and the standard deviation 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 & Anandan (1985). 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 as been greatly influenced by the work of Barto et al. on A_{R-P} units (see e.g., Barto & Anandan, 1985; Barto, 1985). These units also use stochastic reinforcement learnings, 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 nonoptimal 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.

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 (1973), and that of psychologists studying mathematical learning theory (see e.g., Bush & Estes, 1959; Atkinson, Bower, & Crothers, 1965). Narendra and Thathachar (1989) provide a good review of the theory of learning automata. 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, Sutton, & Brouwer (1981) 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 (1985) 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 $\mathbf{x}(t)$ from $X = \Re^n$, where $\Re$ is the set of real numbers. The learning system produces a random output $y(t)$ selected according to some internal probability distribution over some interval $Y \subseteq \Re$. The environment evaluates the output $y(t)$ in the context of the input $\mathbf{x}(t)$ and sends to the learning system a reinforcement signal $r(t) \in R = [0, 1]$, with $r(t) = 1$ denoting the maximum reinforcement. The environment determines the evaluation $r(t)$ according to some conditional probability distribution $D : R \times X \times Y \rightarrow [0, 1]$, where $D(r, \mathbf{x}, y) = \Pr\{r(t) \leq r|\mathbf{x}(t) = \mathbf{x}, y(t) = y\}$. The objective of the learning system is to learn to respond to each input pattern $\mathbf{x} \in X$ with the action $y^{\mathbf{x}}$ with probability 1, where $y^{\mathbf{x}}$ is such that $E(r|\mathbf{x}, y^{\mathbf{x}}) = \max_{y\in Y}\{E(r|\mathbf{x}, y)\}$.

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 $x_i(t) \in \Re$, $1 \leq i \leq n$. 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 probability distribution function used by the unit could have several parameters, $p_1(t)$, $p_2(t)$, . . . , $p_m(t)$, not all of which need be computed by the unit itself. In other words, some of the parameters could

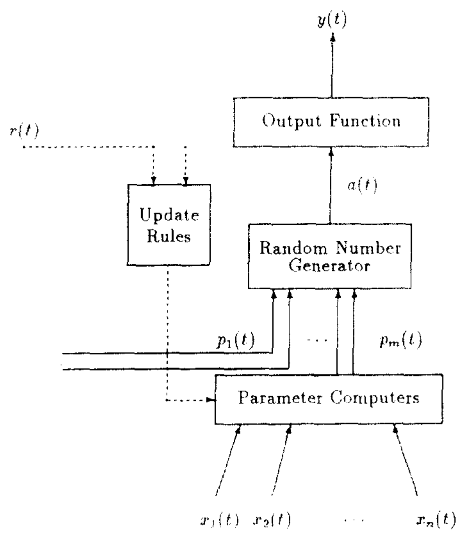

FIGURE 1. Block diagram of a generalized stochastic unit. The flow of signals used to compute the output of the unit is shown with solid lines, while the signals used for learning are shown with dashed lines.

be supplied as *inputs* to the unit by some external agency.[1] The parameters computed internally by the unit are obtained as weighted sums of inputs, 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 $p_j(t) = g_j(\Sigma_{i=1}^n w_i^j(t)x_i(t))$ 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 $p_1(t), \ldots, p_m(t)$). 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 Jordan (1987) use a threshold function for the A_{R-P} unit so that the unit's output values are restricted to either 0 or 1. Rumelhart, Hinton, and Williams (1986) use a nonlinear *squashing function* in their back-propagation algorithm. An example of such a function is the *logistic function*

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

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.

3. A LEARNING ALGORITHM FOR REAL-VALUED UNITS

There are two aspects of their behavior 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 units should manifest while learning 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 in these cases that is too random.

Clearly, a single parameter distribution cannot exhibit these properties because there is no way of

[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 (Ackley, Hinton, & Sejnowski, 1985). 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* of Williams (1986), 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.

independently controlling the mean and variance of real-valued random variables generated using a single parameter distribution. It is therefore necessary to consider using multiparameter distributions to determine the output of the stochastic unit. We now present a stochastic learning algorithm that can be used by a unit learning to produce real-valued outputs by estimating the mean μ, and standard deviation σ, of the normal distribution $\Psi(\mu, \sigma)$ it uses to generate its activation. We assume that the output of the unit is a continuous, monotonic function of the activation. The reinforcement received by the unit from the environment is also assumed to be in [0, 1] with 1.0 denoting the maximum attainable 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:

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

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 close the current expected output (i.e., the current value of the mean) is to the optimal output for that input. Since the reinforcement signal returned by 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.[3]

To accomplish this, we first compute the *expected reinforcement* $\hat{r}(t)$ as a weighted sum of the inputs, using a different set of weights v_i, as follows:

$$\hat{r}(t) = \sum_{i=1}^{n} v_i(t)x_i(t) + v_{\text{thres}}(t). \tag{3}$$

This expected reinforcement is used to compute the standard deviation as

$$\sigma(t) = s(\hat{r}(t)), \tag{4}$$

where $s(.)$ is a monotonically decreasing, nonnegative function of $\hat{r}(t)$. Moreover, $s(1.0) = 0.0$, so that when the maximum reinforcement is expected, the standard deviation is zero. In the simulations reported in this paper, $s(.)$ was defined as

$$s(\hat{r}(t)) = \max\left(\left(\frac{1.0 - \hat{r}(t)}{5.0}\right), 0.0\right). \tag{5}$$

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

$$a(t) \sim \Psi(\mu(t), \sigma(t)). \tag{6}$$

Finally, the activation $a(t)$ is transformed into the output of the unit $y(t)$ using the output function $f(.)$, so that

$$y(t) = f(a(t)). \tag{7}$$

For the purposes of this paper, we will use the logistic function of eqn (1) as the output function of the unit. This choice is in keeping with the requirements that the output function should be continuous and monotonic.

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:

$$w_i(t+1) = w_i(t) + \alpha\Delta_w(t)x_i(t), \tag{8}$$

$$w_{\text{thres}}(t+1) = w_{\text{thres}}(t) + \alpha\Delta_w(t), \tag{9}$$

where α is the learning rate parameter and

$$\Delta_w(t) = (r(t) - \hat{r}(t))\left(\frac{a(t) - \mu(t)}{\sigma(t)}\right). \tag{10}$$

We can view the fraction in eqn (10) 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

[3] If, unlike our assumption above, the range of reinforcement and the maximum reinforcement attainable are not known a priori, there is no absolute standard of performance against which the performance of the unit can be compared, and no values of reinforcement are particularly high or low. One alternative in such cases would be to have the units keep track of the maximum and minimum values of reinforcement attained over time and use this range to normalize the actual reinforcement obtained. However, this increases the likelihood of the unit being trapped at a local maximum of the reinforcement.

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* than 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 (1960), as shown in the following equations:

$$v_i(t+1) = v_i(t) + \beta\Delta_v(t)x_i(t), \tag{11}$$

where β is the learning rate parameter and

$$\Delta_v(t) = r(t) - \hat{r}(t). \tag{12}$$

Figure 2 shows how the above algorithm can be implemented as a stochastic learning unit. We call units implementing the above algorithm SRV (stochastic real-valued) units.

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 eqn (2);

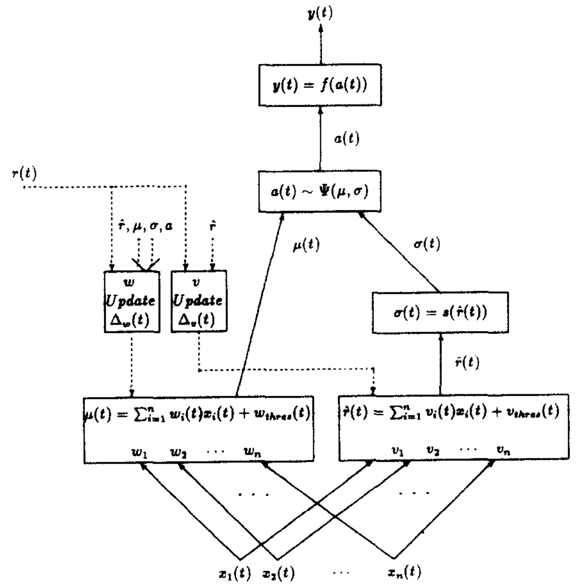
$y(t)$
$y(t) = f(a(t))$
$a(t)$
$a(t) \sim \Psi(\mu, \sigma)$
$r(t)$
$\hat{r}, \mu, \sigma, a$
$\hat{r}$
$\mu(t)$
$\sigma(t)$
w Update $\Delta_w(t)$
v Update $\Delta_v(t)$
$\sigma(t) = s(\hat{r}(t))$
$\hat{r}(t)$
$\mu(t) = \sum_{i=1}^n w_i(t)x_i(t) + w_{thres}(t)$
$\hat{r}(t) = \sum_{i=1}^n v_i(t)x_i(t) + v_{thres}(t)$
$w_1 \; w_2 \; \cdots \; w_n$
$v_1 \; v_2 \; \cdots \; v_n$
$x_1(t) \; x_2(t) \; \cdots \; x_n(t)$

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 dashed lines.

3. the unit computes $\hat{r}(t)$ using the weights $v_i(t)$ as in eqn (3) and uses it to compute $\sigma(t)$ using eqn (4);
4. the unit then computes its activation and its output using eqns (6) and (7);
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 eqns (8)–(12).

In the case of a network of SRV 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 reinforcement signal can then be used by *all* the units to update their weights. Alternatively, we can use various credit assignment schemes to determine the reinforcement for each individual unit from the global reinforcement signal. The units then use their individual reinforcement signals to update their weights. (For a review of credit assignment schemes, see Anderson, 1986).

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. Although faster learning can be achieved in such situations by having the environment provide the desired response for each input, that is, by using supervised learning, our objective here is to test the ability of reinforcement learning method to discover the correct outputs. In other words, we are using these tasks as a simple means of simulating cases in which the environment evaluates the output of a 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.

Single Unit Tasks

In each of the three tasks in this group, a single SRV unit with two input lines for context inputs and an additional input line for reinforcement was used (Figure 3). Each task was attempted over several *training runs*, where a training run consisted of a single attempt at training the unit to perform the task. For each of these tasks and each of the training runs, the unit was started with all its weights set to zero. The learning rates used were $\alpha = 0.25$ and $\beta = 0.5$. In each training run, 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, for each presentation, the learning cycle described in Section 3 was executed. At the end of step 4 of each cycle, an error value was computed as the difference between the desired and actual output. Since both these values lie in (0, 1), $|\text{error}| \leq 1$. This error was used to determine the reinforcement signal broadcast to the unit. The unit then used this signal to update its weights. Each training run was continued until the average error fell below a criterion or until a fixed number of epochs had elapsed. Two different reinforcement signals were used: $r_1(t)$ and $r_2(t)$. The first of these, $r_1(t)$, was defined as

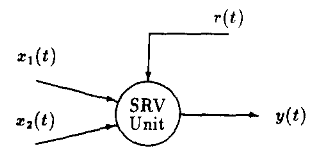

FIGURE 3. A single stochastic real-valued (SRV) unit with two context inputs and one output. The unit also has a reinforcement input.

$$r_1(t) = 1 - |\text{error}|. \tag{13}$$

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.[4] Thus

$$r_2(t) \sim \Psi(r_1(t), 0.005). \tag{14}$$

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 constructed as in Figure 4 and consists of nine A_{R-P} units connected to a single linear output unit. Each binary A_{R-P} unit contributes a value of 0.0 or 0.1 to the

[4]Note that $r_2(t)$ might exceed 1.0 when $r_1(t)$ is close to 1.0.

TABLE 1
The Five Tasks Used to Test the Performance of the SRV Units

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	Figure 5	Figure 6
	(0.9, 0.2) → 0.4		
2	(0.3, 0.1) → 0.8	Figure 7	Figure 8
	(0.2, 0.7) → 0.5		
	(0.8, 0.5) → 0.2		
3	(0.1, 0.1) → 0.1	Figure 9	Figure 10
(AND)	(0.1, 0.9) → 0.1		
	(0.9, 0.1) → 0.1		
	(0.9, 0.9) → 0.9		
4	(0.1, 0.1) → 0.1	Figure 12	Figure 13
(AND)	(0.1, 0.9) → 0.1		
(Network)	(0.9, 0.1) → 0.1		
	(0.9, 0.9) → 0.9		
5	(0.1, 0.1) → 0.1	Figure 14	Figure 16
(XOR)	(0.1, 0.9) → 0.9	(Normal reinf.)	
	(0.9, 0.1) → 0.9	Figure 15	
	(0.9, 0.9) → 0.1	(Modified reinf.)	

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. Note that the AND task appears twice in this table (tasks 3 and 4). This is because we wanted to compare the performance of a single SRV unit with that of a network of SRV and back-propagation units for a given task.

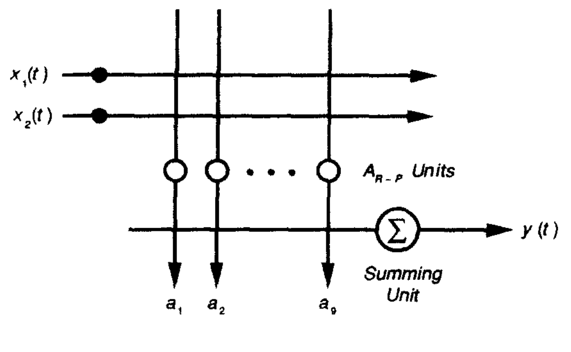

FIGURE 4. An ensemble of binary A_{R-P} units used to simulate a real-valued unit. Each A_{R-P} unit outputs a 0 or a 1, and the output of the ensemble is the weighted sum of the outputs of the A_{R-P} units, with each unit being assigned a weight of 0.1. Thus the ensemble as a whole can produce outputs ranging from 0.0 to 0.9 in steps of 0.1.

output depending on whether its output is 0 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 $r_2(t)$). 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

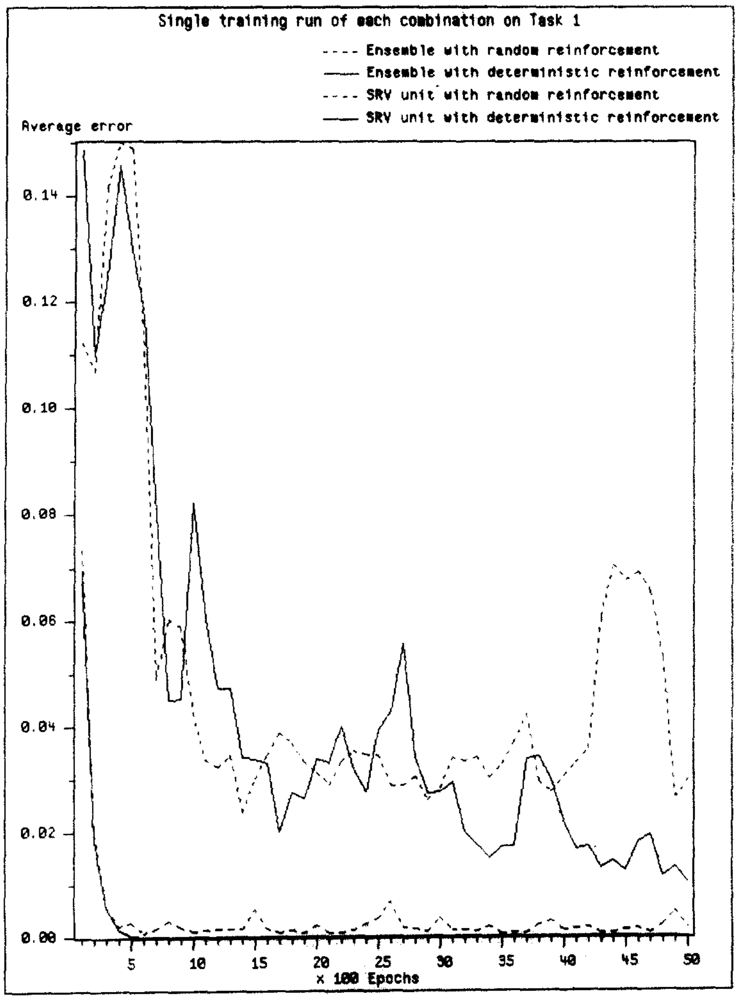

FIGURE 5. Plot of the average error (over the last 100 epochs) at each epoch for each of the four combinations [real-valued and ensemble with r_1 and r_2] trained on Task 1. This plot is for a single training run. The top two curves are for the ensemble, while the bottom two are for the SRV unit.

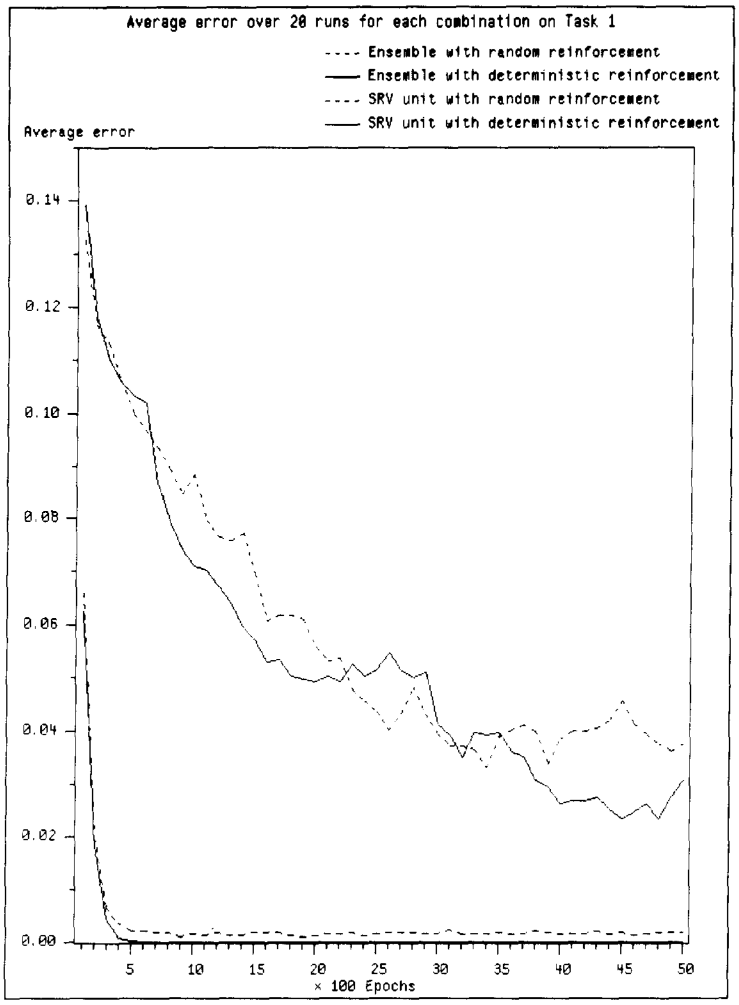

FIGURE 6. Plot of the average error (over the last 100 epochs) at each epoch for each of the four combinatons [real-valued and ensemble with r_1 and r_2] trained on Task 1. The error at each epoch is further averaged over twenty training runs. The top two curves are for the ensemble, while the bottom two are for the SRV unit.

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 for a single unit, there is no set of weights that can compute the given mapping. However, the unit very quickly learns a close approximation of the task (average error is below 0.075 after 2000 epochs), even though the output values do not exactly match the desired values.

From these simulations we can conclude that the SRV algorithm works reasonably well for simple tasks and shows good convergence properties. The time to convergence of the algorithm on these tasks is comparable to that of other reinforcement learning schemes. It is also apparent that even for the simple tasks considered here, an ensemble of binary units is much slower than the SRV algorithm.

Tasks using Networks of Units

As the next step in testing our algorithm, we attempted to train networks consisting of only SRV units to learn various tasks. However, our preliminary experiments indicated that such networks are not very stable due to the nonspecificity of the training signal. We therefore switched to networks in which all the output units were SRV units and the rest of the units were back-propagation units. Such hybrid networks of stochastic reinforcement learning units and back-propagation units have previously been studied by Anderson (1986). Note that such a hybrid structure combines the best of both supervised and reinforcement learning. The network as a whole still has all the benefits of reinforcement learning due to its stochastic search behavior. On the other hand, the use of back-propagation nodes in the hidden layer enables the layer to develop the right features needed for solving the task, as seen in supervised learning networks.

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 (eqn (10)) of the output units as ap-

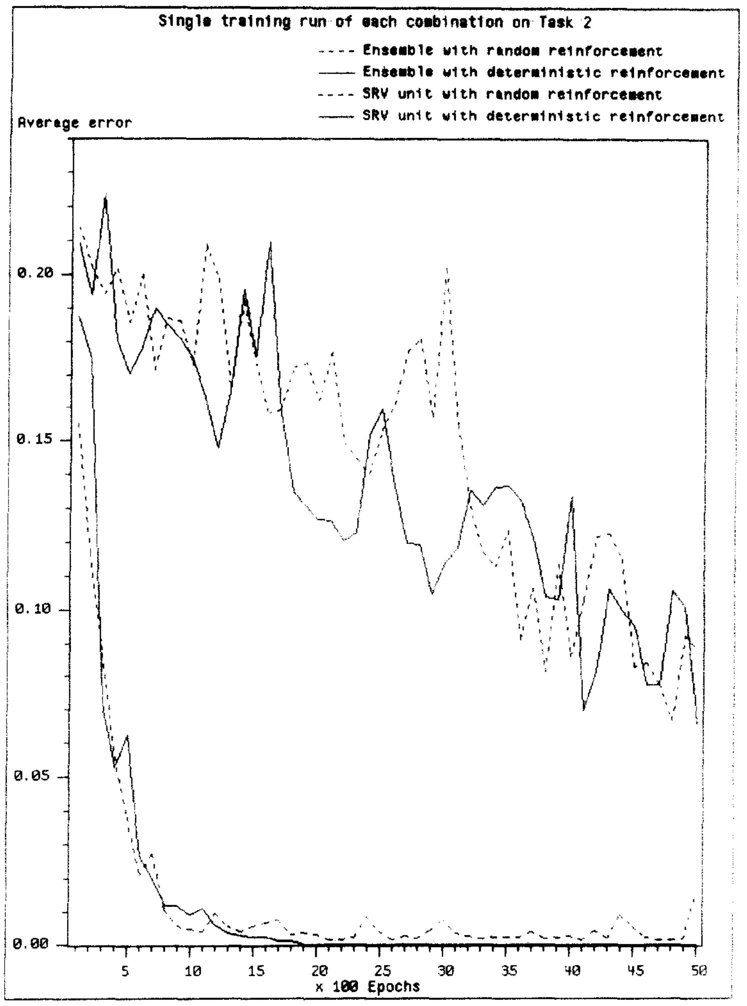

FIGURE 7. Plot of the average error (over the last 100 epochs) at each epoch for each of the four combinations [real-valued and ensemble with r_1 and r_2] trained on Task 2. This plot is for a single training run. The top two curves are for the ensemble, while the bottom two are for the SRV unit.

proximations 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. Furthermore, under certain conditions, we can show that $\Delta_w(t)$ provides an unbiased estimate of the gradient of the reinforcement signal with respect to the activation. Therefore there is strong justification for using $\Delta_w(t)$ as an approximation of the error term for each output unit. An additional step was added to the learning cycle described in Section 3, in which the $\Delta_w(t)$ terms of the output units were propagated backwards as errors to the hidden units, which were back-propagation units, and the weights of these hidden units were updated.

We present here results of simulations in which a simple hybrid network with one SRV unit and one back-propagation unit 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 compute the reinforcement using eqns (13) and (14). The error also served as a measure of the performance of the algorithm.

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.

The task definitions in Table 1 imply that Task 5 (the XOR task) should be harder to learn than the other tasks because of the following reason. We can regard the real-valued function learned when an SRV unit, or the network, is trained on a task to be a surface whose shape is constrained by the training stimuli for that task. Clearly, the constraints imposed by the definitions of the first four tasks can be met by learning simple surfaces of constant or monotonic slopes. On the other hand, for the XOR task, any surface that satisfies the constraints of the task definition has a complex shape that is harder to learn. Figure 14 shows the performance of the network for 10 training runs of this task using the reinforcement signal r_1. 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 twenty 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 50,000th epoch was greater than 0.05. In the fourteen 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.

One critical factor that affects the performance of any reinforcement learning system is the quality of the reinforcement signal supplied to it by the environment, since this signal provides the entire information available to the learning system about the task. We therefore decided to see if we could improve the performance of the network on the XOR task by incorporating more task-specific information into the reinforcement signal. A weaker definition of the XOR task, one that captures the "concept" of XOR, is the following: The output produced when the two inputs in the stimulus are the same should be *less*

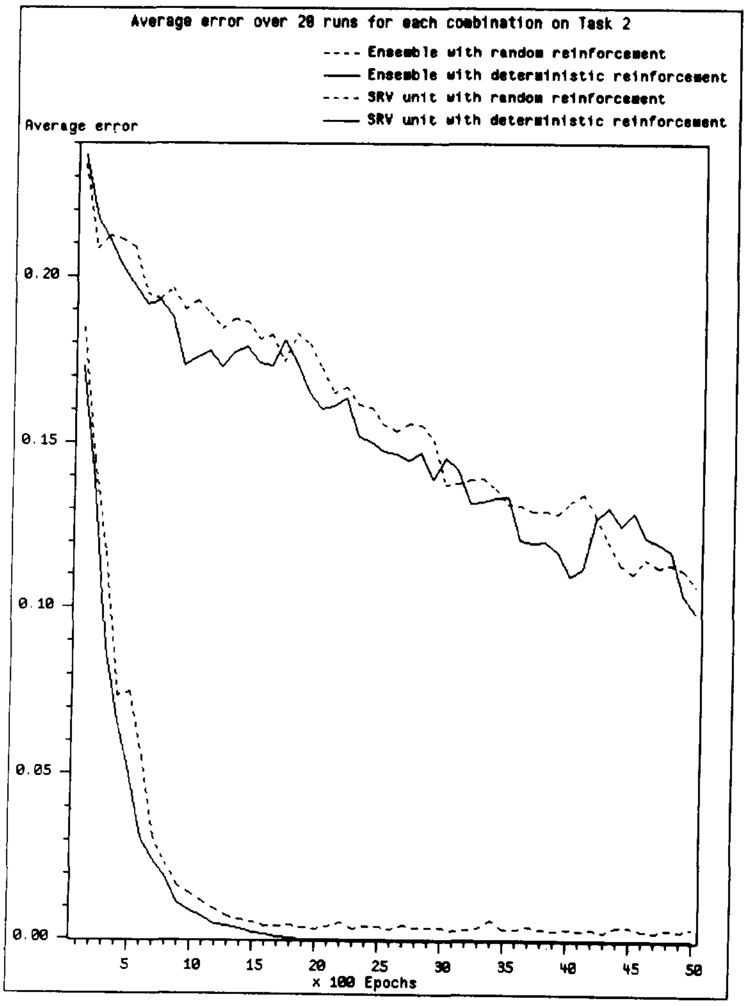

FIGURE 8. Plot of the average error (over the last 100 epochs) at each epoch for each of the four combinations [real-valued and ensemble with r_1 and r_2] trained on Task 2. The error at each epoch is further averaged over twenty training runs. The top two curves are for the ensemble, while the bottom two are for the SRV unit.

than the value produced when they are different. In order to incorporate an evaluation of how well the performance of the network satisfied this weaker definition, we added an additional factor r_{task} to the usual reinforcement. Here r_{task} was computed by storing the latest outputs produced by the network for each of the four inputs, and using them to check if the condition stated above was satisfied. After each presentation of a stimulus, if the latest outputs on similar inputs was less than each of the latest outputs on dissimilar inputs, r_{task} was set to 0.5 and if they were not, then r_{task} was set to -0.5. The reinforcement was now computed as:

$$r_s = \frac{r_1 + r_{task}}{2}, \qquad (15)$$

where r_1 is as defined in eqn (13). The inclusion of the reinforcement component r_{task} could be viewed as *shaping* the response of the network, since the network initially tries to obtain a partial reward by satisfying the weaker definition of the XOR task, which makes it easier for the network to satisfy the strong definition and obtain the maximum reward.

With this simple modification of the reinforcement, the performance of the network improved tremendously. Figure 15 shows a plot of ten training runs of the XOR task using the modified reinforcement. There were two improvements in the performance of the network. Out of twenty training runs, the network now converged to the correct weights in seventeen 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 twenty 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

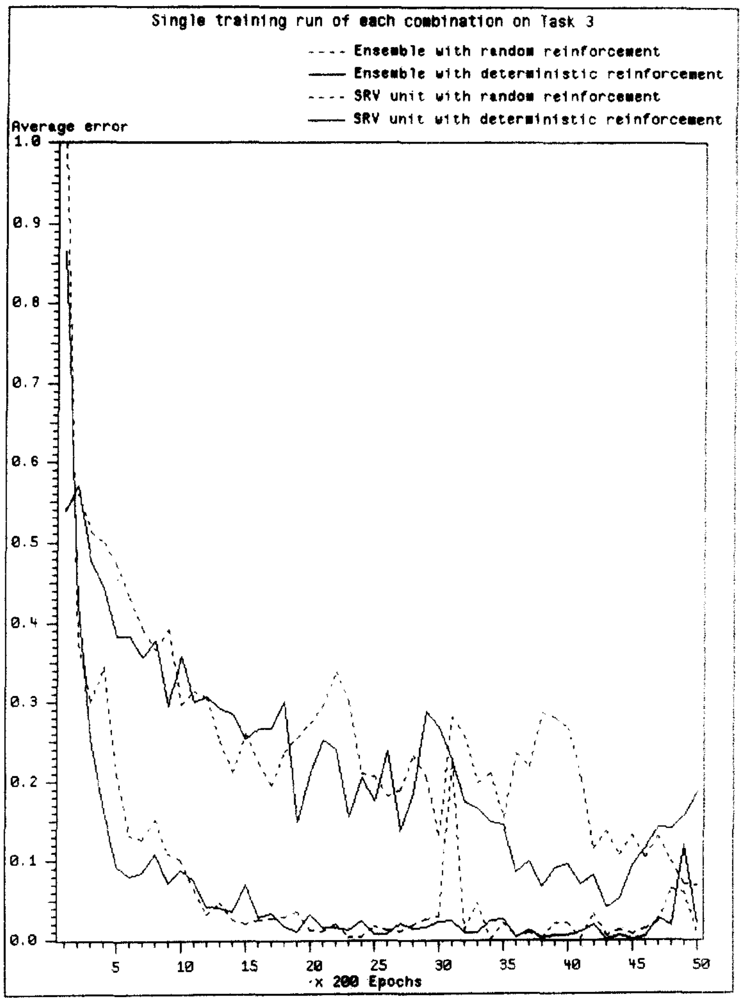

FIGURE 9. Plot of the average error (over the last 200 epochs) at each epoch for each of the four combinations [real-valued and ensemble with r_1 and r_2] trained on Task 3. This plot is for a single training run. The top two curves are for the ensemble, while the bottom two are for the SRV unit.

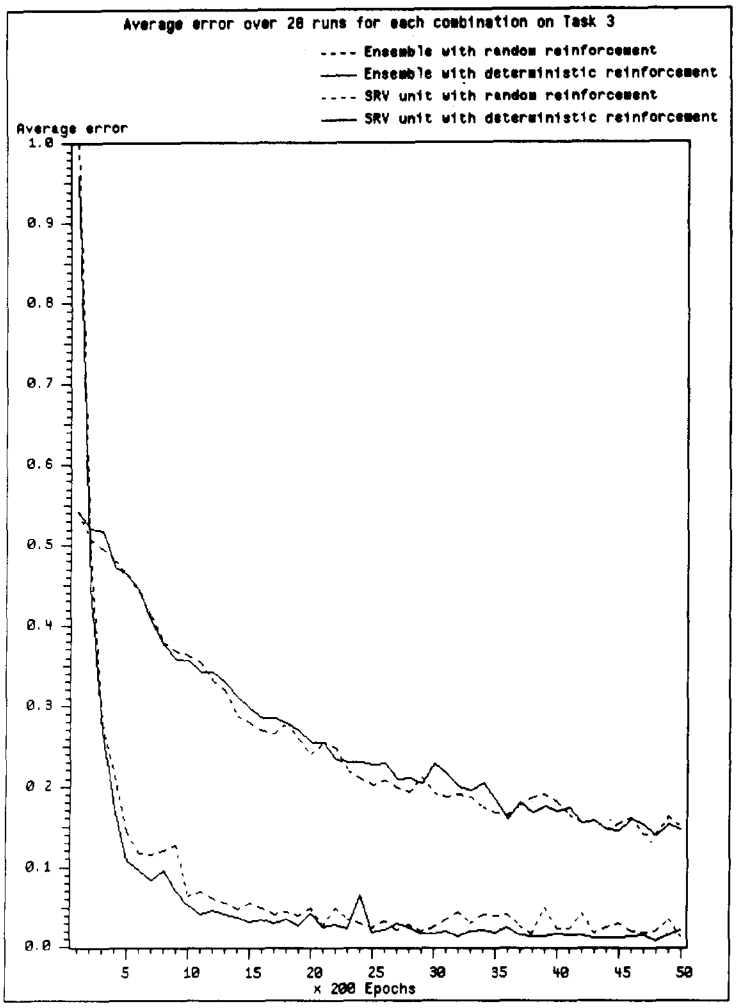

FIGURE 10. Plot of the average error (over the last 200 epochs) at each epoch for each of the four combinations [real-valued and ensemble with r_1 and r_2] trained on Task 3. The error at each epoch is further averaged over twenty training runs. The top two curves are for the ensemble, while the bottom two are for the SRV unit.

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 can indeed be trained to learn to perform tasks of varying degrees of difficulty. 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

Most tasks in robotics involve the manipulation of objects specified in terms of world coordinates. In order to accomplish these tasks, it is necessary to position a part of the robot (usually the end effector) at a specific location in 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 world coordinates, a set of joint angles that can achieve that position. This is known in robotics as the problem of computing the inverse kinematics. We present below an experiment in reinforcement learning in

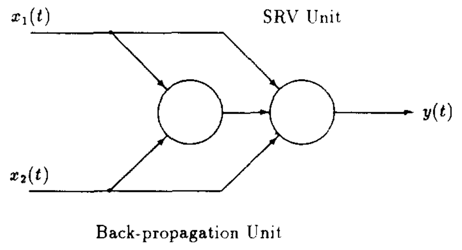

FIGURE 11. The network used for the 4 and 5 described in Table 1. Note that the output unit is a SRV unit, while the hidden unit is a back-propagation unit.

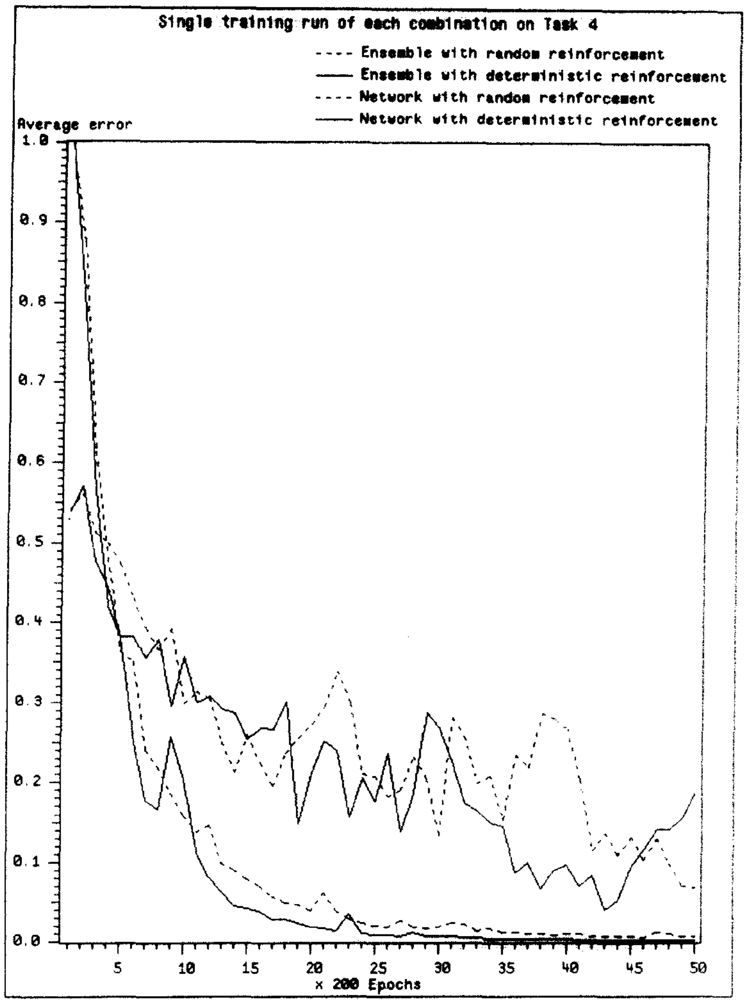

FIGURE 12. Plot of the average error (over the last 200 epochs) at each epoch for the network of SRV and back-prop units trained on Task 4 using reinforcement signals r_1 and r_2. This plot is for a single training run. The top two curves are the corresponding plots for the ensemble of A_{R-P} units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

which a network of units was trained to learn to perform a positioning task using a simulated 3-DOF robot 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. This work is an extension of our preliminary investigations reported in Gullapalli (1988).

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 on 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. Although these three degrees of freedom imply a bigger potential workspace for the arm, for the purposes of this experiment we restricted the workspace to the rectangle $[-1.0, 1.0] \times [0.0, 1.0]$. The position of the arm at any instant can be specified by giving the location of the base x_b; the angular displacement of the first link θ; and the angular displacement of the second link ϕ. 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. In order to keep the computations simple, we did not use a dynamic model of the arm for the simulation.

The task to be learned is: *Given the current position of the arm in joint-space coordinates, the change* *in the position of the arm over the last time step, and a desired x-coordinate location for the endpoint 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.* In control engineering terms, we want to build a second-order controller for controlling the x-coordinate location of the endpoint of the arm. 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), which permit an infinite number of solutions for the task for any given x_d.

When dealing with excess degrees of freedom, the standard procedure in control engineering is to define additional performance criteria that can be used to select a unique solution that effectively utilizes the excess degrees of freedom. This selection process involves optimization of the performance criteria subject to several nonlinear constraints, a computationally intensive process. The goal of our experiment in learning control was to demonstrate how connectionist networks can be trained to learn such solutins through the incorportion of suitable performance measures into the reinforcement. To do this, we added a condition that the arm positioning task defined above should be performed 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 final arm configurations for a given target location. Moreover, the final arm configurations produced should be stable fixed-points of the network. In other words, if the network is given a desired endpoint location and an arm configuration close to a final arm configuration for that location as input, it should produce the final arm configuration as output.

Based on the above specifications of the physical

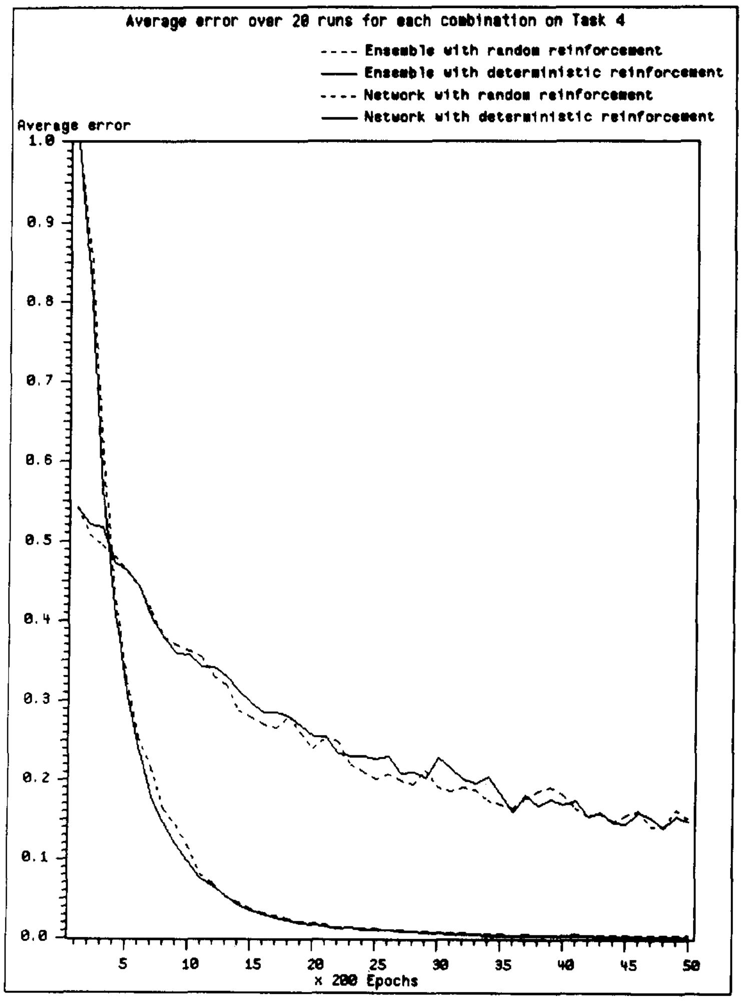

FIGURE 13. Plot of the average error (over the last 200 epochs) at each epoch for the network of SRV and back-prop units trained on Task 4 using reinforcement signals r_1 and r_2. The error at each epoch is further averaged over twenty training runs. The top two curves are the corresponding plots for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

system, the task to be learned, and the desired behavior of the trained network, we arrived at the following network structure and training procedure. The network used for this task is shown in Figure 18. There are seven inputs to the network and three outputs. The seven inputs are divided into two groups: the command input, which is the desired endpoint location x_d, and the context inputs, which define the context in which the network is to compute the solution to the task. The context inputs encode the state of the arm and consist of the arm configuration from the last time step $(x_b(t-1), \theta(t-1), \phi(t-1))$, and the joint velocities (changes in joint angles in the last time step) $(\dot{x}_b(t-1), \dot{\theta}(t-1), \dot{\phi}(t-1))$. The outputs of the network at each time step specify the joint angle coordinates of the arm for that time step, viz., $(x_b(t), \theta(t), \phi(t))$. In order to keep things simple, we assume that the arm is driven by accurate position control. Therefore the new physical configuration of the arm at each time step t is exactly that specified by the output of the network. We also assume that successive time steps occur after intervals of unit time, so that the joint velocities can be computed as the changes in the joint angles in successive time steps (for example, $\dot{\theta}(t) = \theta(t) - \theta(t-1)$). 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, each of which consisted of a sequence of five time steps. For each epoch, a random initial configuration for the arm and a random value for x_d were chosen. The values for x_d were restricted to the interval $(-0.9, 0.9)$ and the initial joint velocities were set to zero. These joint coordinates and velocities and x_d were supplied as the initial input to the network and the training began. At each time step dur-

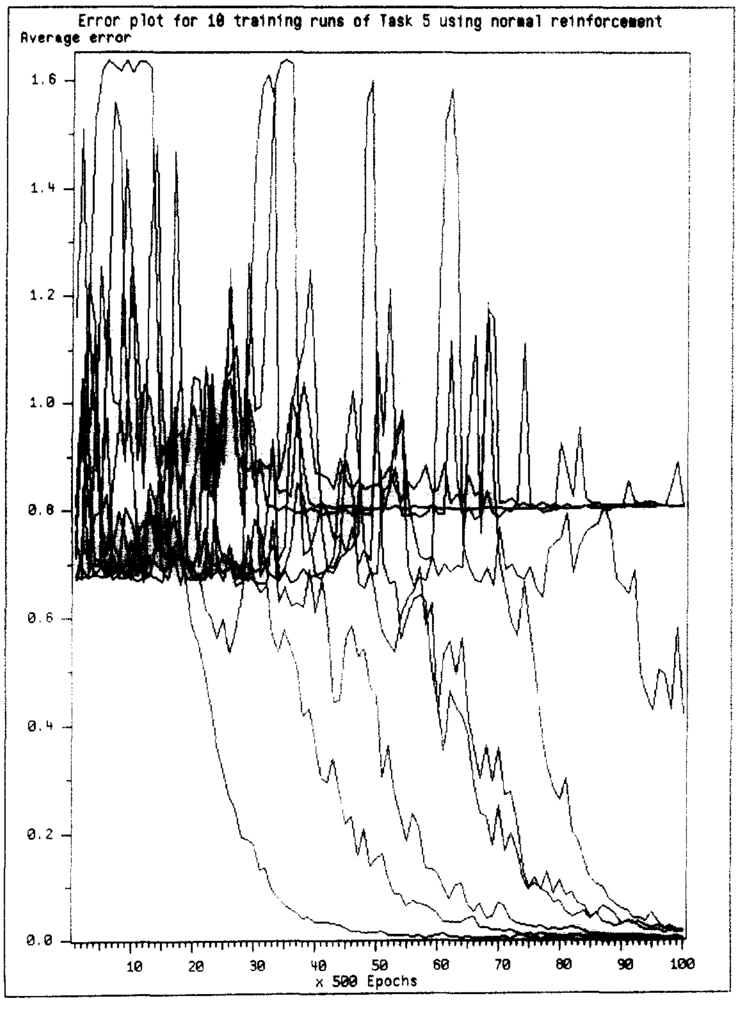

FIGURE 14. Plot of the average error (over the last 500 epochs) at each epoch for ten training runs on Task 5. The reinforcement used was r_1.

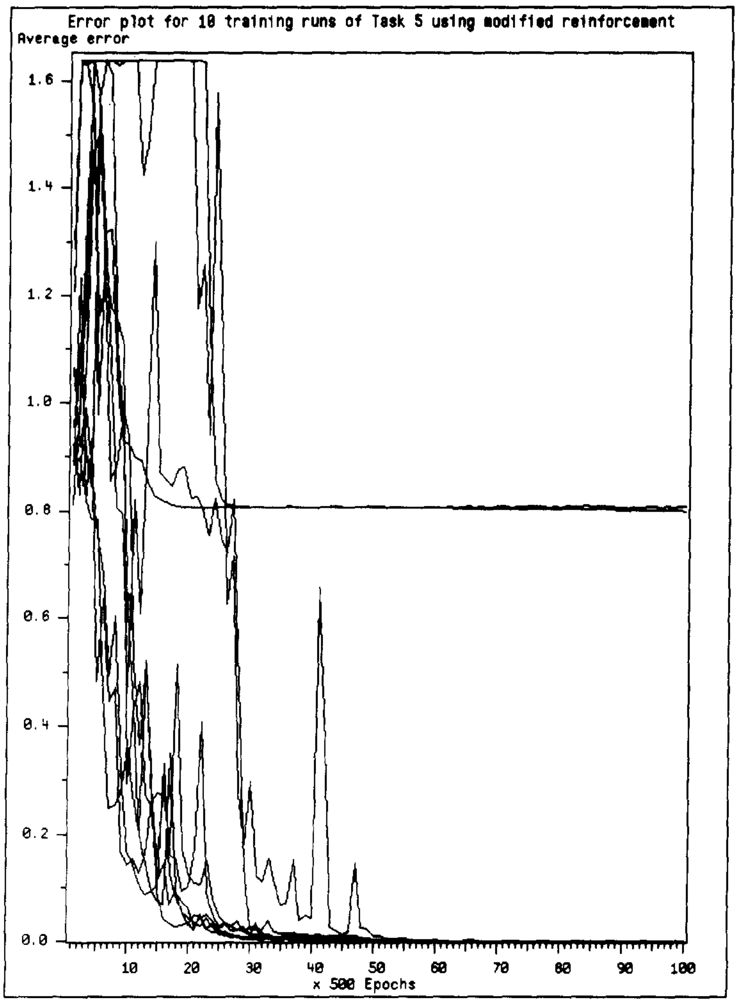

FIGURE 15. Plot of the average error (over the last 500 epochs) at each epoch for ten training runs on Task 5. The reinforcement used was r_3.

ing training, the following computations were performed. First, a new set of joint coordinates was 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 new x–y coordinates of its endpoint. Based on this new endpoint location, the environment returned a reinforcement signal to the network. The evaluation function used to compute the reinforcement is described below. The computations for each time step concluded with the updating of the network's weights using the reinforcement received. The computations for the next time step were then begun with the same x_d and the latest set of context inputs. At the end of every five time steps, a new epoch was started with the choice of a new set of random initial inputs.

The evaluation function used to evaluate the performance of the network at each time step during training was computed as

$$r(t) = \frac{e_1(t) + e_2(t)}{2.0}, \tag{16}$$

where the two components e_1 and e_2 were measures of two different aspects of the performance. The first component e_1 measured how well the network achieved its primary goal of moving the endpoint to the desired x-location. e_1 was computed as a function of the distance of the endpoint of the arm from the desired x-coordinate x_d. This distance was computed as

$$d(t) = |x(t) - x_d|, \tag{17}$$

where $(x(t), y(t))$ denotes the location of the endpoint in the workspace at the end of time step t. e_1 was computed at each time step as

$$e_1(t) = \begin{cases} 1 & \text{if } d(t) = 0 \\ 0.5 + \dfrac{d(t-1) - d(t)}{2 \times \max(d(t-1), d(t))} & \text{otherwise.} \end{cases} \tag{18}$$

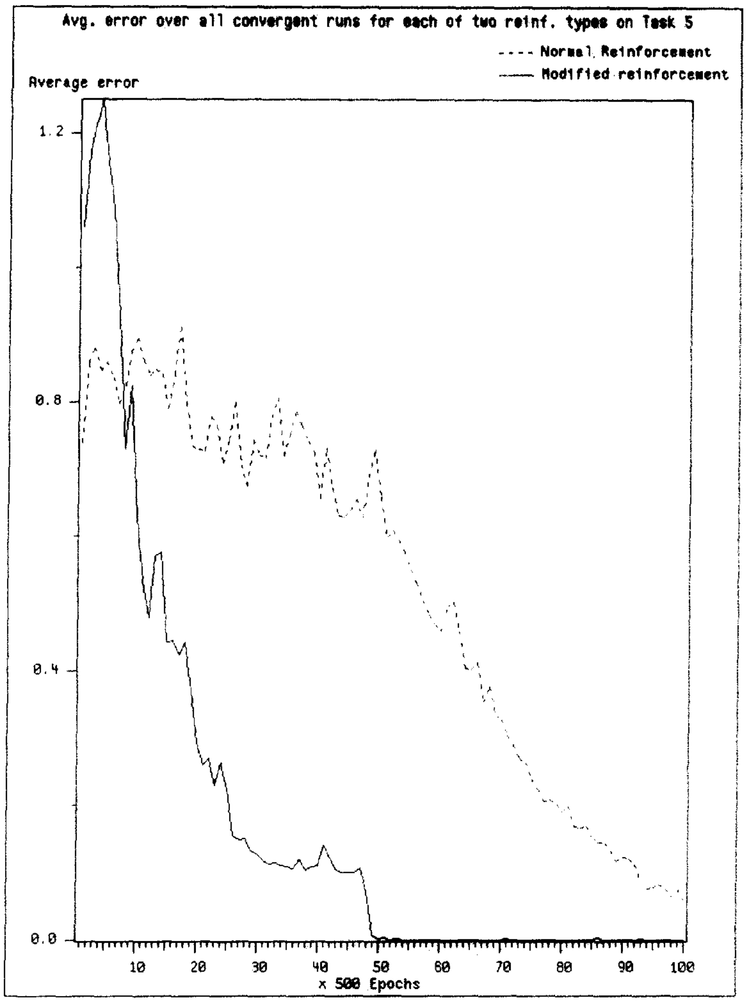

FIGURE 16. Plot of the average error (over the last 500 epochs) at each epoch for twenty training runs on Task 5. The plot shows the relative performances of the network when using the normal reinforcement r_1 and when using the modified reinforcement r_3.

$e_1(t)$ determined the size of the evaluation based on (a) the fraction of the distance towards (away from) the desired location traveled by the endpoint in one time step, and (b) the actual distance of the endpoint from the desired location. Thus the maximum reward ($e_1(t) = 1$) was given when the network moved the endpoint onto the desired location ($d(t) = 0$), while the maximum punishment ($e_1(t) = 0$) occurred when the endpoint, which was at the desired location at the previous time step ($d(t - 1) = 0$), was moved *away* from the desired location ($d(t) > 0$).

The second component e_2 was a rough indicator of the "efficiency" of the latest change of arm configuration computed by the network. It was based on the joint velocities at each time step and was computed as

$$e_2(t) = \begin{cases} 1 & \text{if } |\dot{x}_b(t)| + |\dot{\theta}(t)| + |\dot{\phi}(t)| \le d(t-1) \\ 0 & \text{otherwise.} \end{cases} \tag{19}$$

This second component punished the network whenever the sum of the magnitudes of the joint velocities of the arm exceeded a limiting value that decreased linearly as the endpoint moved closer to the desired location.

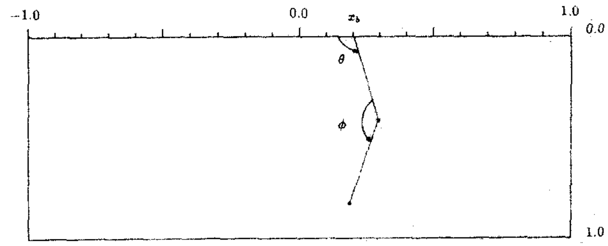

FIGURE 17. The simulated robot arm in its workspace. The three degrees of freedom of the arm are the translation along the x-axis x_b, the rotation of the first link θ, and the rotation of the second link ϕ.

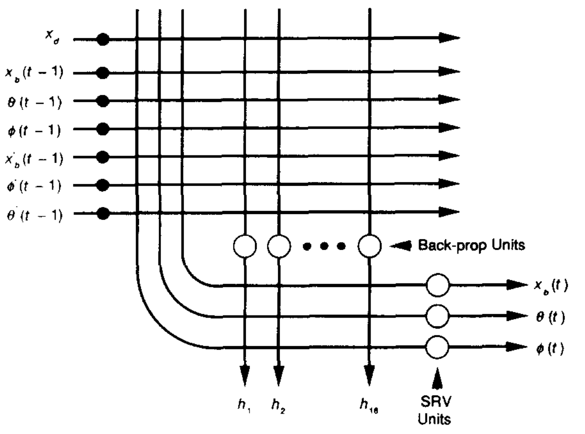

FIGURE 18. The network used to learn the task of controlling the simulated robot arm. The output layer of the network is composed of SRV units and the units in the hidden layer are back-propagation units. Each of the sixteen back-propagation units receives inputs from all seven input units, while each of the three SRV output units is connected to all the seven input units as well as all the sixteen hidden units.

Performance

There are two aspects to the performance of the trained network which need to be evaluated in order to determine if the network has learned the specified positioning task. The first of these is the accuracy of the positioning of the endpoint. The second is the effectiveness of utilization of the excess degrees of freedom. But before we discuss these aspects, we would like to describe the general behavior of the trained network. After training the network in the manner described above for 30,000 epochs, the variances of the SRV units were set to zero so that their outputs were computed using their mean activation values. Thus the stochasticity in the output of the network, which is only useful for learning, was eliminated.[5] The network was then used to control the position of the arm as follows. An initial set of joint coordinates and velocities and the desired endpoint location were given as input to the network. Then, for one or more time steps, the network was used to compute a new joint configuration, with state feedback at the end of each time step being used to update the context inputs to the network. Each new joint configuration was fed to a simulator which computed the new location of the endpoint of the arm and produced a graphic display. Figure 19 shows a sample display in which the network was run for 20 time steps, starting with an initial joint configuration of (-0.167, 60°, 135°), zero initial joint velocities, and with the target location $x_d = 0.415$.

One of the things that is immediately obvious from the figure is that the network output converges to a stable configuration with the endpoint at (or near) the target location. This is true for all target locations and initial configurations. This means that the network has at least one *limit point* for each target location, and furthermore, the limit configuration is a solution for the positioning task. We find that the network takes about 15 time steps on the average to produce the final, stable solution. A second important observation is that the movement of the arm appears to be *ballistic* initially, with the arm making a big jump into the immediate neighborhood of the target location, following which the movement appears to become *sensory-guided* with the state feedback being used to fine-tune the exact location of the endpoint. Again, this behavior has been observed for all target locations and initial configurations. Based on observations of human arm and eye movements, Arbib (1981) suggested that biological motor control schemes use feedforward control to quickly reduce large discrepancies in the desired output, while slower, more accurate feedback control is used to bring the output to the exact value desired. It is interesting to observe that the behavior of our network described above is very similar to the behavior of a coactivation feedforward controller described by Arbib. That such a behavior can emerge directly from the reinforcement learning paradigm and the training procedure used is certainly noteworthy and deserves further study.

We evaluated the accuracy of the positioning of the endpoint by running 10,000 test runs on the trained network and collecting data on the desired and computed endpoint locations. Each test run started with a random target location and initial joint configuration (with zero initial joint velocities) being input to the network. We then cycled through the network over successive time steps until the network produced a stable set of outputs (i.e., until the joint velocities became zero). At this point we recorded the x-location of the endpoint as well as the target location x_d, and started the next test run. The data collected through these test runs is presented in the

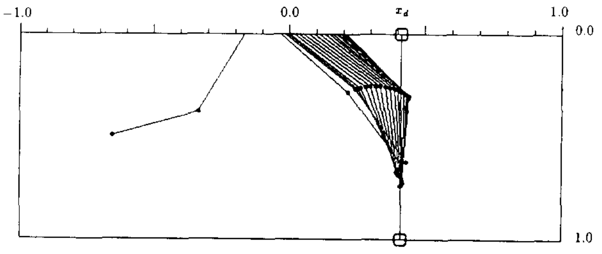

FIGURE 19. An example of the computation of successive joint angle configurations by the network for a given target location x_d = 0.415 and an initial joint configuration of (-0.167, 60°, 135°).

[5]At the end of 30,000 epochs, the weights in the network had not converged completely, and hence the variances of the SRV units, though small, had not yet become zero.

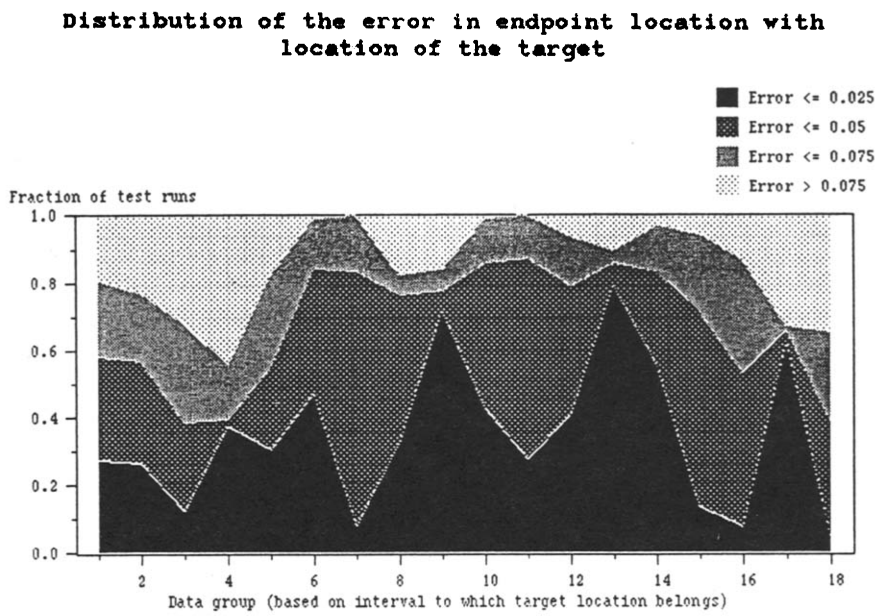

FIGURE 20. Performance of the arm over 10,000 test runs. The test runs were split into 18 groups by dividing the target x-coordinate range into equal intervals and grouping the test runs based on the interval to which the target location used in each run belonged. The figure shows the fraction of test runs in each group for which the magnitude of the error in positioning the arm by the network was in each of the intervals [0.0, 0.025], (0.025, 0.5], (0.05, 0.075], and greater than 0.075.

following fashion. The x-coordinate range of the target (0.9, 0.9), was divided into 18 intervals $(-0.9, -0.8)$, $[-0.8, -0.7)$, $\cdots$ $[0.7, 0.8)$, $[0.8, 0.9)$, and each test run was put in one of 18 groups based on the interval to which the target location used in the run belonged. We then determined the fraction of test runs in a group for which the error in positioning $|x - x_d|$ was in each of the intervals $[0.0, 0.025]$, $(0.025, 0.05]$, $(0.05, 0.075]$, and greater than 0.075. These fractions were computed for all the groups and are displayed in Figure 20. From the figure, it is clear that over most of the range of the target location, a majority of the test runs resulted in positioning errors of less than 0.05, with a significant fraction of errors being less than 0.025 in magnitude. This shows that the network has been able to successfully learn the positioning task. However, over the entire target location range, about 20% of the test runs resulted in positioning errors larger than 0.05 in magnitude. We will discuss the reason for this shortly.

In order to evaluate the second aspect of the performance, namely the effectiveness with which the excess degrees of freedom are utilized, we tried to determine how the starting arm configuration influenced the final configuration computed by the network. With the given degrees of freedom, there are four typical initial configurations for the arm that arise from the location of the base of the arm (to the left or right of x_d), and the orientation of the elbow (pointing towards or away from x_d). Figure 21 shows a set of four test runs in which we observed how the same target location was reached when the network was started in each of these four typical initial configurations. Each initial configuration and the corresponding limit configuration computed by the network are labeled with the same number (1, 2, 3, or 4). It is obvious from the figure that the network does utilize the excess degrees of freedom to find solutions to the positioning task that are qualitatively efficient. We cannot, however, readily quantify the degree of efficiency because any such quantitative comparison would necessitate the computation of the minimal-joint-angle-change soloution, which is a very hard task. Observations of the behavior of the trained network indicate that the network had up to four limit configurations for most target locations, although for extreme values of x_d (close to ±0.9), the number of limit configurations decreased to two. This is because at the extremities of the workspace the

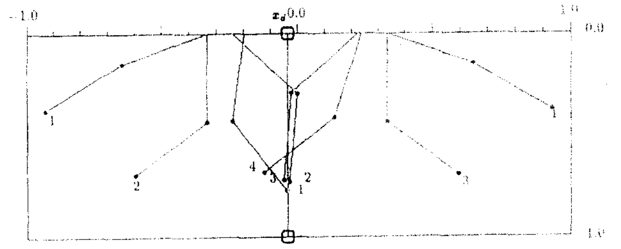

FIGURE 21. An example of how the same target location is reached when the arm is started at each of four typical initial joint configurations. The final stable configurations corresponding to each of the initial configurations are identified by the labels 1, 2, 3, and 4.

number of degrees of freedom available for the task drops from three to two. Another important observation was that the positioning error was not the same for all the final configurations that could be reached for a given target. Usually, one of the final configurations had a large positioning error, while the rest were very accurate. This discrepancy can be seen for the example in Figure 21 and accounts for the rather large fraction of trials with error greater than 0.05 in Figure 20. But, regardless of the error in positioning, we have observed that the solutions to the positioning task computed by the network have always met our requirement of "efficiency" as defined in the task statement.

6. CONCLUSIONS AND COMMENTS

In this paper, we have presented a new stochastic reinforcement learning algorithm for learning real-valued functions. The algorithm is designed to use the two parameters of the normal distribution to explore the space of possible output values, and to produce, for a given input vector, the output value that elicits the highest expected reinforcement from the environment. We have implemented this algorithm as a connectionist unit, which we call an SRV unit, and have used these units in networks that are trained to perform tasks of varying levels of difficulty. The simulations described in Section 4 indicate that the SRV units perform well in simple learning tasks. The SRV algorithm also does not suffer from the problem of output saturation described by Albus (see the Introduction). 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. The modified reinforcement signal delivered a partial reward when the network satisfied a weaker definition of the XOR task, with the full reward being given when the actual definition of the XOR task was satisfied. This process could be regarded as using the reinforcement signal to shape the response of the network. Thus our experiment also serves to illustrate the utility of shaping as a means of improving learning performance.

Our experiment in learning position control of a robot arm in the presence of excess degrees of freedom serves to illustrate the utility of SRV units. The results presented in Section 5 indicate that a network of SRV and back-propagation units can be trained, using simple, easy-to-evaluate reinforcement signals, to successfully compute position commands for the arm. The trained network also has very interesting and useful computational properties that arise from the existence of limit points that correspond to each target location. As discussed above, the limit points mean that the network can produce stable solutions for the positioning task. These limit points are points of attraction in the state space of the arm, much like the limit cycles of the sequential networks described by Jordan (1986) and others. The limit points render the computation of the position commands relatively insensitive to perturbations. This means that even if the controller for the arm was inaccurate, or there were inaccuracies in the state feedback, the network would still be able to compute a stable solution for the positioning task. Moreover, multiple limit points for the same target location result in the network being able to stably compute several solutions to the same problem, with the choice of the solution being based on prespecified criteria. It is not at all obvious that the rather simple training procedure adopted here would produce networks that can exhibit the complex behavior described above, and more work is needed to analyze both, the cause for the behavior, and the behavior itself. However, the fact that the trained network does behave in this fashion enhances the utility of the learning methods presented in this paper. Furthermore, although we can qualitatively describe the performance of our network as "efficient", it is difficult to come up with a quantitative measure of how efficient the performance really is. This is because it is very hard to analytically determine a minimal joint-angle-change solution to the positioning task, against which the solution computed by the network can be compared. This observation underlines the importance of training procedures such as the one described above, since they can be used to train a network to find solutions with specific desirable properties for problems that are hard to solve using other methods.

REFERENCES

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

Albus, J. S. (1975). *Brains, behavior, and robotics*. Peterborough, NH: BYTE Books.

Alspector, J., Allen, R. B., Hu, V., & Satyanarayana, S. (1987). Stochastic learning networks and their electronic implementation. *Proceedings of the Conference on Neural Information Processing Systems*, Denver, Colorado.

Anderson, C. W. *Learning and problem solving with multilayer connectionist systems*. Ph.D. dissertation, University of Massachusetts, Amherst, 1986.

Arbib, M. A. (1981). Perceptual structures and distributed motor control. In V. B. Brooks (Ed.), *Handbook of physiology—The nervous system II, motor control*. Bethesda, MD: American Psychological Society.

Atkinson, R. C., Bower, G. H., & Crothers, E. J. (1965). *An introduction to mathematical learning theory*. New York: Wiley.

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

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

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.

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

Barto, A. G., Sutton, R. S., & Watkins, C. (to appear). Prediction, control, and learning. In M. Gabriel and J. W. Moore (Eds.), *Learning and computational neuroscience*. Cambridge, MA: The MIT Press.

Bush, R. R., & Estes, W. K. (Eds.). (1959). *Studies in mathematical learning theory*. Stanford, California: Stanford University Press.

Bush, R. R., & Mosteller, F. (1955). *Stochastic models for learning*. New York: Wiley.

Farley, B. G., & Clark, W. A. (1954). Simulation of self-organizing systems by digital computer. *I.R.E. Transactions on Information Theory*, **PGIT-4**, 76–84.

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

Gullapalli, V. (1988). *A stochastic algorithm for learning real-valued functions via reinforcement feedback* (COINS Technical Report 88-91). Amherst: Dept. of Computer and Info. Sciences, University of Massachusetts.

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

Hinton, G. E. (1987). *Connectionist learning procedures*. (Technical Report CMU-CS-87-115). Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

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

Jordan, M. I. (1986). Attractor dynamics and parallelism in a connectionist sequential machine. *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*. Hillsdale, NJ: Erlbaum.

McMurtry, G. J. (1970). Adaptive optimization procedures. In J. M. Mendel and K. S. Fu (Eds.), *Adaptive, learning and pattern recognition systems: Theory and applications*. New York and London: Academic Press.

Mendel, J. M., & McLaren, R. W. (1970). Reinforcement-learning control and pattern recognition systems. In J. M. Mendel and K. S. Fu (Eds.), *Adaptive, learning and pattern recognition systems: Theory and applications*. New York and London: Academic Press.

Narendra, K. S., & Lakshmivarahan, S. (1977). Learning automata—A critique. *Journal of Cybernetics and Information Science*, **1**, 53–65.

Narendra, K. S., & Thathachar, M. A. L. (1989). *Learning automata: An introduction*. Englewood Cliffs, NJ: Prentice Hall.

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

Skinner, B. F. (1983). *The behavior of organisms: An experimental analysis*. New York: D. Appleton Century.

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

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

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

Widrow, B., & Hoff, M. E. (1960). Adaptive switching circuits. *1960 WESCON Convention Record Part IV*, pp. 96–104.

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