Rapid Reinforcement Learning for Reactive Control Policy Design in Autonomous Robots

Andrew H. Fagg* David Lotspeich# Joel Hoff* George A. Bekey*

af0a@robotics.usc.edu lotspeic@robotics.usc.edu hoffj@robotics.usc.edu bekey@robotics.usc.edu

Robotics Research Laboratory and Center for Neural Engineering
*Department of Computer Science and #Department of Industrial & Systems Engineering
Henry Salvatori Building #300
University of Southern California
Los Angeles, California 90089-0781

Abstract

A key property of Artificial Life systems is their ability to autonomously learn from experience while behaving within a dynamic environment. By ***autonomous learning****, we mean that a system is capable of acquiring control programs with little information provided by some external teacher. To this end, we have developed a neural-based reinforcement learning architecture for the design of reactive control policies for an autonomous robot. Reinforcement learning techniques allow a programmer to specify the control program at the level of the desired behavior of the robot, rather than at the level of the program that generates that behavior. In this chapter, we address the issue of state representation which can greatly affect the system's ability to learn quickly and to apply what has already been learned to novel situations. Finally, we demonstrate the architecture as applied towards a real robot that is learning to move safely about its environment.*

Introduction

Traditional methods of constructing intelligent robotic systems, employing artificial intelligence-based techniques, have met great difficulties in application to real-world problems. Such systems have often required a tremendous amount of computational power in order to make control decisions, and thus have sacrificed the ability to make decisions in real time. In addition, these systems must make many assumptions about the information that is supplied by the sensory processing subcomponents or about the results of actions that are taken. When these assumptions become invalid (which is typically the case when presented with a dynamic environment), these systems become brittle, and they are ultimately unable to reliably accomplish their mission.

Reactive or behavior-based control systems have been offered as alternative methods to these more traditional techniques of designing robotic control systems (Arbib, 1989; Arkin, 1990; Bekey & Tomovic, 1986; Brooks, 1986; Brooks, 1991). These approaches embody two key principles. First of all, the control problem is decomposed into a set of simple computing modules, each of which may be designed and implemented separately. Secondly, each module extracts only the information that it needs in order to make a reasonable decision. This approach contrasts significantly with traditional AI approaches, in which the sensing system is used to update a global world model, from which all decisions are then made.

However, despite some of the recently reported successes of reactive or behavior-based approaches, hiding behind most successes is a graduate student who spends many hours carefully designing, testing, and redesigning the set of control modules, until the desired behavior is achieved (some exceptions to this include Maes & Brooks (1990) and Mahadevan & Connell (1992)). One reason that this process is so laborious is that it is typically very difficult for a programmer to put herself *in the shoes of the robot*, and truly understand the information that is being provided by the (often imperfect) sensing subcomponents, as well as understand the range of possible outcomes of actions taken by the robot. In addition, when a robot is picked up and placed into a new environment, there is no guarantee that the control program will continue to function as desired (Verschure, Kröse, & Pfeifer, 1992).

One possible approach to these difficulties is the application of a reinforcement-based learning technique (Barto & Bradtke, 1991; Barto, Sutton, & Anderson, 1983; Samuel, 1967; Sutton, 1988; Watkins & Dayan, 1992; Williams, 1987). Here, the programmer (or teacher) provides the robot with only an evaluation of its behavior. In our case, this evaluation is a scalar score that is potentially (but not necessarily) given for each control decision that is made by the controller. We refer to the manner in which the reinforcement information is computed as the *reinforcement policy*. Based upon this abstract representation of the desired behavior, the task of the learning system is to infer a reactive control strategy that satisfies the reinforcement policy provided by the teacher[1].

---

1 By satisfies the reinforcement policy, we mean that positive reinforcement is maximized and negative reinforcement

From the teacher's point of view, this approach appears to be much simpler than actually hand-coding a control program. However, the problem of programming still exists - we have just deferred a large part of the problem to a learning algorithm. When positive (or negative) reinforcement information is made available by the teacher, the algorithm must decide what was appropriate (or inappropriate) that led to receiving the reinforcement. We refer to this as the *structural credit assignment problem*. In addition, it is possible that reinforcement information only becomes available after some unknown delay from the time that a critical action is taken. Computing when these critical decisions are made is known as the *temporal credit assignment problem*.

These two problems can further conspire to make the learning problem very difficult. For example, the reinforcement that is delivered to the learning system might be a function of an entire sequence of actions taken by the robot. In these situations, it is only after the robot has taken several different combinations of actions, that the learning algorithm can begin to infer the structure of the problem.

One key issue in this information gathering process is that of *generalization*: how does the learning algorithm apply what has already been learned to the current situation, even though it has never experienced exactly the same situation before? The way in which a neural learning algorithm[2] generalizes is determined to a large degree by the manner in which *state* (the current situation) is encoded as a pattern of neural activity. Purely localist representations (Barto & Bradtke, 1991; Barto, et al., 1983; Watkins & Dayan, 1992) utilize non-overlapping patterns of activity for each possible state that the system might encounter. Such an approach suffers because it scales poorly with an increasing number of state variables. In addition, neighboring states are not able to share information with one-another, requiring that the system visit all states during the learning process in order to ensure a reasonable control policy.

On the other end of the spectrum are the completely distributed representations, such as backpropagation-based techniques (Williams, 1987), where every hidden unit participates to some degree in each learned mapping. Although such approaches allow for generalization between states,

---

is minimized.

[2] This statement is actually applicable to other forms of learning.

the very same mechanism that gives us this feature also causes a significant amount of interference between dissimilar states. The result is a significant slow-down in learning as the state space becomes larger or more complicated. In this work, we seek a state representation that sits somewhere between these two extremes - that allows us to capture some degree of generalization, but is still localist enough such that learning in different regions of the state space will not interfere with one-another.

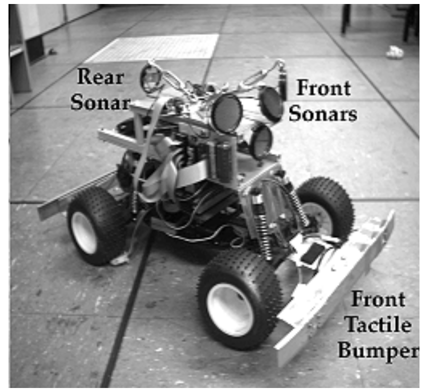

Figure 1: Mobile robot used in learning experiments.

In the remainder of this chapter, we first present a specific neural architecture for the representation of reactive control programs. We next show how this model can be updated in light of reinforcement information from a teacher, and how a module that predicts future reinforcement can be used to solve the temporal credit assignment problem. Finally, we illustrate the behavior of the architecture when it is presented with a couple of tasks to be learned.

Problem Description

The Robot

The experiments described below were performed in simulation, and in some cases with a robot (Figure 1). The robot is an adapted radio-controlled car, which is equipped with two tactile bumpers that are mounted on the front and the rear of the vehicle. In addition, there are five sonar sensors which are oriented in different directions: Left, Forward, Right, Up (forward), and Rear.

The Task

The world in which the robot is situated is a standard laboratory environment, with a large variety of obstacles, some of which (such as chair and table legs) are rather difficult to detect, especially with a moving sonar platform. In work to date, we have experimented with two different tasks that the robot is to learn:

1. Avoid collision with obstacles.

2. Environmental exploration.

For each different task, the teacher chooses an appropriate reinforcement policy. This reinforcement policy expresses a mapping from an observed behavior to a scalar evaluation of that behavior. For example, one possible reinforcement policy for task 1 is to punish the robot when it collides with an obstacle (reinforcement = -1), and otherwise no information is given (reinforcement = 0). From this sparse information, the learning control system must acquire a control policy that reliably keeps the robot from receiving the negative reinforcement.

Network Model

The general network architecture is depicted in Figure 2, and has evolved from our work on modeling of primate visual/motor conditional learning (Fagg & Arbib, 1992). The network maps the current sensory inputs into appropriate actions. Inputs from five sonar units, and two bumpers (I) activate a particular pattern activity across the feature detector layer (F). The feature detector units then interact with one-another through a cooperation/competition mechanism to contrast-enhance this activity pattern (G). The result is that individual feature units (G_j) become active in response to specific conjunctions of inputs.

These units then vote for a favored set of actions at the *action-selection layer* (A). The votes from the set of active feature detector units are gathered together at each action through a summation operation. The one action with the highest activity is chosen to be executed for the current time-step. The six possible actions that may be taken are defined by: {Left, Straight, Right}x{Forward, Reverse}.

After action execution, the teacher responds by providing a reinforcement signal. This signal is used to update the connection matrices W and W'.

In the interest of computational efficiency of the simulation, the unit dynamics are implemented as a one-pass computation (as opposed to a set of differential equations that must be integrated). The sensory vector (I) consists of one neuron for each bumper sensor (active if touching something), and four neurons each for the sonar inputs; distances are coded in these sub-vectors using a gaussian coding scheme. The mapping from sensors (I) to feature detector units (F) is implemented as a simple matrix-

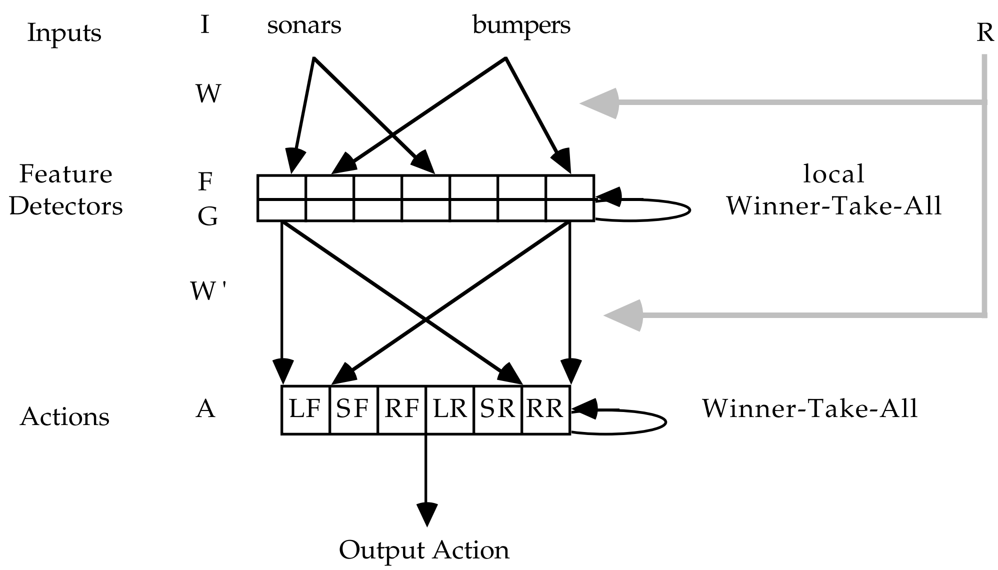

Figure 2: The general network architecture. The sensor inputs activate a set of feature detector units. Those units remaining after the local Winner-Take-All operation is performed instantiate votes for one of six actions (LF = left forward, SF = straight forward, ... , RR = right reverse). The one action unit with the highest number of votes is output for execution.

vector operation:

$$F_j = \mathrm{h_j}\left(\sum_i \left(W_{ij} * I_i\right) + Noise_j\right) \tag{1}$$

where

I and F are vectors representing the input unit activities and the feature detector inputs, respectively.

W_{ij} is the connection strength from input unit i to feature detector j.

$Noise$ is a vector of random signals that are injected into the feature detector units.

$\mathrm{h_j}(\ )$ is a non-linear function parameterized by j (See Appendix A)

The contrast enhancement function at the feature detector layer is implemented by a *local winner-take-all* operation. Unlike the standard *winner-take-all algorithm* (e.g. Didday, 1976), a particular unit only has an effect over a small neighborhood of the feature detector field, thereby segmenting it into regions of activity similar to that discussed by von der Malsburg (1973).

This interaction amongst the feature-detector units is implemented according to the rule:

$$Winner_j = \begin{cases} 1 & \text{if } F_j = \underset{j-N \le m \le j+N}{\text{Max}} \{F_m\} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

where:

N defines a local neighborhood of feature detectors.

The feature detector output activity vector (G) is then computed by:

$$G_j = \begin{cases} F_j & \textit{if } Winner_j = 1 \\ \gamma F_m & \textit{if } Winner_j = 0 \textit{ and } Winner_m = 1 \textit{ and } m \textit{ is close to } j \\ 0 & \textit{otherwise} \end{cases} \tag{3}$$

where:

the distance from m to j is small relative to N (we use the term *local neighborhood* to describe the set of such m's).

This contrast-enhancement operation serves a vital role in the assignment of credit during the learning process. Through its application, it is guaranteed that only a small number of units in the feature detector layer actually become active at any one time (and thus instantiate their votes at the action selection layer). When reinforcement information becomes available, identifying those feature detector units that are responsible for the selected action (so that learning can occur) will be a much more precise computation. This point will be elaborated further in the Discussion section.

The action votes are collected by the action selection units :

$$A_k = \sum_j \left(W'_{jk} * G_j \right) + Noise'_k \tag{4}$$

The action that is output is action **p** such that:

$$A_p = \underset{k}{Max} \{A_k\} \tag{5}$$

Once the selected action is executed, a new set of sensory inputs is presented to the network and the process of selecting an action is repeated.

Network Learning

In parallel to the execution process described above, the learning algorithm makes updates to the weight matrices W and W' (Figure 2) based on the reinforcement information (R) that is received from the teacher. The sign of the reinforcement signal indicates the appropriateness ($R > 0$) or inappropriateness ($R < 0$) of the *recent behavior* of the robot, and the magnitude of the signal represents the degree of (in)appropriateness. Inherent in this definition is the fact that an entire sequence of actions may contribute to the final reinforcement signal that is provided by the teacher. Thus, the learning algorithm is faced with propagating current reinforcement information backward through time in such a way that the recent actions are updated appropriately.

At the level of the individual feature detector, the intent of learning is to:

1. Discover behaviorally-relevant conjunctions of signals from the input layer, and
2. Based upon the recognition of such a feature, discover the action or set of actions that are most appropriate.

Weight Matrix Configuration and Initialization

The projections from sensory inputs to feature detectors and feature detectors to outputs are sparse projections in that only a subset of all possible connections are actually made. Before an experiment begins, this subset is selected at random. The initial connection strengths are set such that relative to a single feature detector unit j:

$$W_{ij} = \begin{cases} \dfrac{1}{b_j} & \text{if } W_{ij} \text{ is in the subset} \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

where b_j is the number of connections in the subset projecting to j; and

$$W'_{jk} = \begin{cases} \dfrac{1}{b'_j} & \text{if } W'_{jk} \text{ is in the subset} \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

where b'_j is the number of connections in the subset projecting from unit j. In addition, the following constraints hold before learning begins, and will be enforced at each learning step:

$$\sum_i W_{ij} = 1 \tag{8}$$

$$\sum_k W'_{jk} = 1 \tag{9}$$

Weight Update

We will first consider the *structural credit assignment problem*, which states: given some instantaneous state of the network and a reinforcement signal, R, how is blame assigned to the individual weights represented in W and W'?

Suppose that the execution of the most recent action yields a positive reinforcement signal from the teacher. In order to increase the probability of making the same decision the next time the same (or similar) situation arises, two things must be done. First of all, we must insure that the same set of features are recognized (i.e. the identical set of feature detector units are turned on). This is accomplished by increasing the strengths of the synaptic weights from the currently active input units to the active feature detector units. Second, given that the same set of features are recognized, the same action must be taken. This is captured by increasing the strength of the synaptic weights from the active feature detector units to the selected action.

On the other hand, suppose that a negative reinforcement signal is received from the teacher. The selection of the incorrect action may be due to one of two cases. First of all, the incorrect set of feature detector units may have been selected. If this is the case, then the connection strengths from the currently active input units to the active feature detector units should be decreased. The next time that the same situation arises, the total input to these feature detector units will be weaker, giving them less of a chance to win the local winner-take-all competition. In the second case, the set of feature detector units is correct, but the selected action is incorrect. For this case, the connection strength from the active feature detector units to the selected action is decreased, thus reducing its future probability of being selected. One difficulty is that the system does not know which of the two cases are the correct assessment of blame. We therefore choose to update both sets of weights together.

The update rule for both the positive and negative reinforcement cases may be expressed by:

$$\Delta W_{ij} = \alpha R I_i G_j W_{ij} \tag{10}$$

$$\Delta W'_{jk} = \alpha' R G_j \hat{A}_k W'_{jk} \tag{11}$$

where:

α and α' are learning rate constants.

$\hat{A}$ is a vector in which all elements are 0, except for the $\mathbf{p}^{\mathbf{th}}$ element, where $\mathbf{p}$ is the winning action.

$I_i\, G_j\, W_{ij}$ and $G_j\, \hat{A}_k\, W'_{jk}$ are measures of the responsibility of specific synapses in the last decision (synapses between input unit i and feature detector j; and feature detector j and action unit k, respectively). We refer to these measures as the *instantaneous eligibility* of the synapse.

Finally, the new connection strengths are computed by adding in the delta values and then normalizing:

$$W_{ij} = \frac{\left(W_{ij} + \Delta W_{ij}\right)}{\sum_m \left(W_{mj} + \Delta W_{mj}\right)} \tag{12}$$

$$W'_{jk} = \frac{\left(W'_{jk} + \Delta W'_{jk}\right)}{\sum_p \left(W'_{jp} + \Delta W'_{jp}\right)} \tag{13}$$

As it is used here, normalization provides several important computational properties. First, it ensures that the connection strengths are both bounded and not all connection strengths will simultaneously approach zero. Second, normalization implements a form of competition between the different inputs into an individual feature detector (equation 12). This ultimately ensures that a feature detector is responsive to only a small set of situations. Third, because we are normalizing across the outputs of the feature detector, a competition is implemented between the different actions that the feature detector might support (equation 13). We simultaneously make sure that at least one action is supported, and encourage support for only a small number of the possible actions.

This formulation of credit assignment (equations 10-13) works as long as an appropriate reinforcement signal can be given immediately after every action is made. However, because the teacher is evaluating the overall behavior of the robot, such a reinforcement policy may not exist or be easily determined by the teacher. For this reason, the learning algorithm must be able to correctly determine what effect actions in the past may have had upon the current reinforcement signal. This is referred to as the *temporal credit assignment* problem. Our approach uses three learning mechanisms to approach this problem: eligibility, reinforcement prediction, and weight solidification. In the following sections, we show how each mechanism is implemented, and how the learning algorithm defined initially in equations 10 and 11 evolves as these mechanisms are included in the overall learning algorithm.

Eligibility

The concept of eligibility was first introduced by Klopf (1982) and later used by others, including Barto, et al. (1983). The eligibility of a weight is defined as a temporal memory of a synapse's participation in recent action decisions, and is computed as follows:

$$\tau \frac{d\, e_{ij}}{dt} = -e_{ij} + I_i\, G_j\, W_{ij} \tag{14}$$

$$\tau' \frac{d\, e'_{jk}}{dt} = -e'_{jk} + G_j\, \hat{A}_k\, W'_{jk} \tag{15}$$

where

e_{ij} and e'_{jk} are the eligibility measures.

τ and τ' determine the temporal width of the memories.

Finally, the weight updates become:

$$\Delta W_{ij} = \alpha\, R e_{ij} \tag{16}$$

$$\Delta W'_{jk} = \alpha'\, R e'_{jk} \tag{17}$$

with normalization being computed in the same manner as equations 12 and 13.

By this definition of eligibility, the assignment of credit or blame for a reinforcement signal is given with the highest weight to the most recent action decision that was made, and exponentially decreasing weight for decisions that were made further back in time. One key aspect of this definition of memory is that the number of memory elements does not depend upon the size of the time window over which the memories are stored, and only depends upon the number of synapses in the network.

Reinforcement Prediction

Eligibility provides a simple means by which credit can be assigned to a sequence of control decisions. However, the propagation of reinforcement information is limited to a fixed window of time defined by the decay of the eligibility memory. It is thus possible that a critical decision occurs outside of this window, and therefore would not receive any reinforcement information. As a result, the control network cannot learn to behave properly when an action and the corresponding reinforcement signal are separated by a length of time that is greater than that of the eligibility memory.

This problem is approached in this work through the method of reinforcement prediction (Sutton, 1988). Suppose that we have a prediction network $P(x(t))$ that maps the current state of the system ($x(t)$) into a measure of the expected future reinforcement (relative to a fixed control policy). More explicitly:

$$P(x(t)) = E\left\{\sum_{\tau=t}^{\infty} \lambda^{\tau-t} R(\tau)\right\} \tag{18}$$

where:

$R(t)$ is the reinforcement received at time t.

λ is the discount factor for future reinforcement.

The function $P(x(t))$ can be viewed as the *goodness* of being in state $x(t)$. Now observe that:

$$\begin{aligned} P(x(t)) &= E\left\{\sum_{\tau=t}^{\infty} \lambda^{\tau-t} R(\tau)\right\} \\ &= E\left\{R(t) + \lambda \sum_{\tau=t+1}^{\infty} \lambda^{\tau-t-1} R(\tau)\right\} \\ &= E\left\{R(t) + \lambda P(x(t+1))\right\} \end{aligned} \tag{19}$$

And define:

$$R'(t) = R(t) + \lambda P(x(t+1)) - P(x(t)) \tag{20}$$

where:

$P(x(t))$ and $P(x(t+1))$ are observations along a specific action path.

$R(t)$ is the reinforcement received after the action is executed at time t.

$R'(t)$ can be interpreted as a measure of the deviation of the actual reinforcement received from that which was expected by the prediction network (for an individual time-step). In other words, if $R'(t) > 0$, then the system performed better in the last time-step than it expected to perform; and when $R'(t) < 0$, the system performed worse than expected. This measure can be used as an internally-generated reinforcement signal, that has the potential for delivering more meaningful reinforcement information at *every instant* that a decision is made, even when the teacher is providing a very sparse reinforcement signal.

It is important to note that the function $P\big(x(t)\big)$ is also relative to the control policy implemented by the control network. As this control policy adapts through experience, so must this prediction function. In this work, we make use of the method of *Temporal Difference Learning* (Sutton, 1988) to acquire the prediction function. This learning process is performed in parallel with the adaptation of the control network. This algorithm is given in Appendix B.

The $R'(t)$ that we compute from the non-stationary control policy has several important properties:

- The measure rewards incremental improvements in performance. When the controller discovers an action that leads to a higher level of performance than what was expected, the control network adjusts itself such that the probability of executing the same action given a similar situation is increased. This results in an overall improvement in the system's performance. The prediction network quickly adapts itself such that it expects this new level of performance. At this point, execution of the same action yields a null internal reinforcement signal, and a positive signal is only received if a new action further improves the performance.

- The above implies that even when the teacher only provides negative reinforcement as a way of specifying the desired behavior, the internal reinforcement signal can provide positive reinforcement information for actions that cause the system to receive less negative reinforcement than it was receiving earlier.

- More effective propagation of reinforcement information through time. Consider a controller that has learned for all points in the set A (Figure 3) the correct sequence of actions to drive the system to point p, where it receives positive reinforcement from the teacher. Outside of set A (e.g. point z), however, the system does not know how to get reliably to point p, and in these cases, generates random actions. Once the system has visited the points in set A a number of times, the expected future reinforcement will become equal to what is received at point p (with some discount in the number of steps required to reach p). Thus, the internal reinforcement for moving from any of these points towards p will effectively be zero.

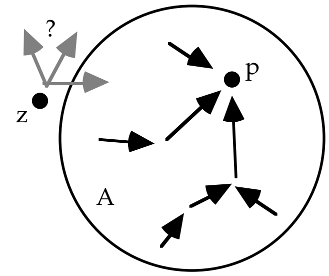

Figure 3: State space representation of a controller's actions. For those points within set A, the controller knows the correct sequence of actions to drive the system to point p, where positive reinforcement is received. However, outside of set A, the system has not yet learned the correct actions. By using R′ as the reinforcement signal, the system is able to deliver a large amount of positive reinforcement to network when the boundary is crossed from point z to a point in set A. This is the case even though reinforcement from the teacher is still several time-steps away.

For the case of point z, the expected future reinforcement will be small, if not zero, and expected internal reinforcement will also be zero (because from point z, the system will tend to wander aimlessly outside of the set A). However, there is some probability that an action chosen at state z will take the system into a point within A - i.e. from a state with low expected future reinforcement to a state with high expected future reinforcement. As a result, the internal reinforcement signal will be high, thus rewarding the action that brought the system into A. This is the case even though the actual reinforcement from the teacher is still several time-steps away.

- Prevention of over-learning. If the external reinforcement signal is used to update the control network, weight updates will continue to be made when positive reinforcement is received, even if the network is already completely committed to the correct action. This can cause problems in terms of unnecessary interference with learning in other nearby regions of the state space. By using R' as the reinforcement signal, the system only adjusts its weights as long it is necessary to ensure commitment to the correct action(s), and then learns no further. Thus, other regions of the state space can be learned more easily, and more neural hardware can be reserved for learning during later experiences.

The updated network architecture is shown in Figure 4. The prediction network is implemented as a

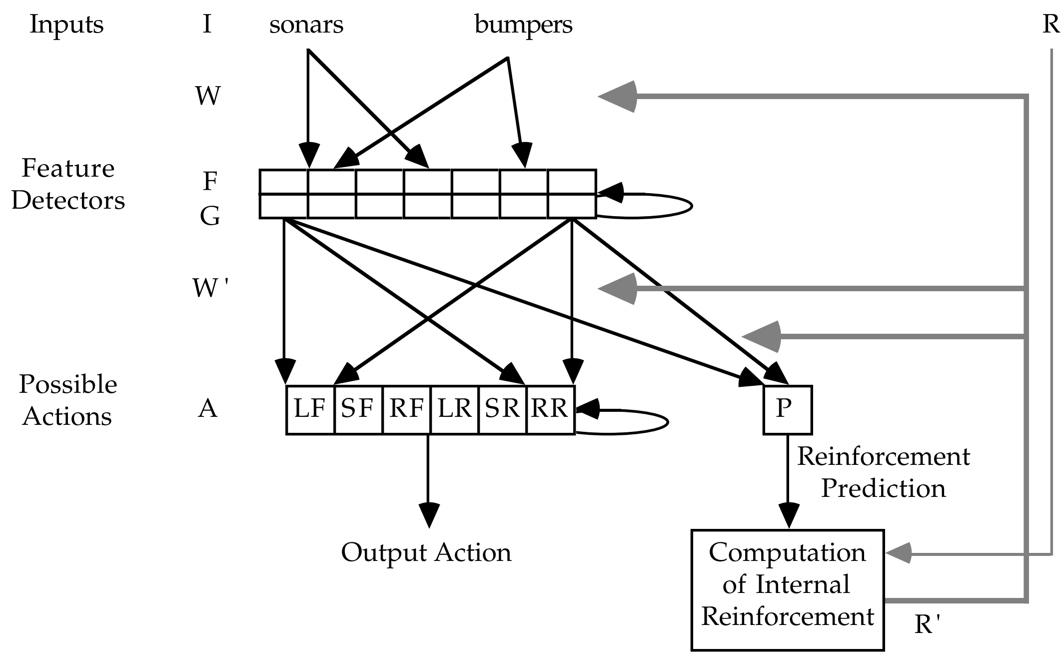

Figure 4: Modified network architecture. The reinforcement information from the teacher is first combined with the output of the Predictor Network to produce an internal reinforcement signal, R′. It is this signal that is used to update the weights in the control network. The Predictor Network (P) is implemented as a linear function of the feature detector outputs (G).

linear neural network, with the feature detector activity vector (G) serving as the input. The environmental reinforcement signal (R) is now combined with system's prediction of reinforcement to generate an internal reinforcement signal (R'). This signal is not only used to update the connections involved in the action selection process (W and W') (in equations 16 and 17, R is replaced with R'), it also serves as an update signal for the prediction network (P).

Although we have argued here that reinforcement prediction can produce more useful reinforcement information than eligibility, in practice both mechanisms are useful. This comes from the fact that our earlier assumption that $P(x(t))$ accurately predicts the expected future reward does not always hold. This is especially the case early in the learning process or after a major change is made to the control network. During these times, the learning algorithm relies upon the eligibility measure of credit assignment in order to quickly learn simple associations, essentially bootstrapping the control network to a partial level of performance.

Weight Solidification

One problem with an incremental learning algorithm such as the one presented here is its tendency to forget past experiences unless they are re-lived constantly. In a robot learning task, such as learning to avoid obstacles, this problem manifests itself in the following manner. First, the robot learns what it must do after it has collided with an obstacle. From there, it learns to anticipate the coming collision and takes the appropriate action to avoid the obstacle. After this point in time, however, the robot never (or rarely) visits the part of the state space in which it is in contact with the obstacle. Because it does not experience this region of the state space very often, the program that it has already learned for this region is slowly erased due to interference effects with other states. The result is that when the robot accidentally comes in contact with an obstacle, it will no longer know how to deal appropriately with the situation.

One way in which this problem can be solved in the proposed architecture is to halt the learning process for elements of the control network that reliably produce positive internal reinforcement. This is implemented by introducing a measure of a feature detector's participation in programs that tend to receive positive internal reinforcement:

$$\overline{R_j}(t) = \sum_{\tau=0}^{t} \mathrm{thresh}\big(R'(\tau)\big) * G_j(\tau) \tag{21}$$

where:

$\overline{R_j}(t)$ is the measure of feature detector j's tendency to participate in programs that perform well (up to time t).

$$\mathrm{thresh}(z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases} \tag{22}$$

The plasticity of the connections related to feature detector j is modulated by:

$$M_j(t) = e^{-\overline{R_j}(t) * accum\_rate} \tag{23}$$

where:

$accum\_rate$ is a parameter that determines how quickly $M_j(t)$ goes to zero as $\overline{R_j}(t)$ increases.

$M_j(t)$ always falls within the range [0,1]

the weight update equations then become:

$$\Delta W_{ij} = \alpha R' e_{ij} M_j \tag{24}$$

$$\Delta W'_{jk} = \alpha' R' e'_{jk} M_j \tag{25}$$

Experimental Results

In this section, we illustrate the behavior of the system through a series of learning experiments. At this time, we are especially interested in understanding how the reinforcement policy affects the learned behavior.

Experiment 1: Collision Avoidance

The collision avoidance problem has been a major focus of our experimentation, in which several different reinforcement policies have been explored:

1. Punish running into an obstacle. When this policy is used, the network has a difficult time determining which action is appropriate, because it is only being told when an action was a bad choice. As a result, the system must spend a significant amount of time searching for the action that will produced the desired behavior.

2. Reward not running into an obstacle. The network often quickly discovers a one-move local minimum that satisfies the basic requirements of the behavior. An example of this is always making a forward right turn, causing the robot to move in a circle. The network very quickly and completely commits itself to this simple strategy, and when later faced with a new situation (such as an obstacle in the path), it is very difficult for the robot to explore alternative actions.

3. Punish running into a wall and reward for not. Two sub-cases are possible:

3A. |Punishment| < |Reward|. This type of policy enables the robot to develop short cycles of movements, which result with overall positive reinforcement. When faced with an obstacle, a very common behavior that is learned is one in which the robot moves forward, collides with the obstacle, backs up one step, and then repeats the process. Even though this strategy receives negative reinforcement for the collision, it makes up for it by ensuring that it receives a higher degree of positive reinforcement at the next time-step.

3B. |**Punishment**| > |**Reward**|. This policy produces control programs that perform the best for the collision avoidance problem. This is due to a balance between policy 1 (from above), where learning requires a long time, and policy 2, where learning happens so quickly that the system does not have adequate opportunity to explore the control space.

Figures 5 and 6 show the learning results of using policy 3B to solve the collision avoidance problem (see Appendix E for the reinforcement policy parameters). Figure 5 tabulates a number of the feature detectors that were commonly learned over several experiments, and the actions that these feature detectors supported. In many cases, the input units that activate the feature detectors logically match the preferred actions.

Sensory Inputs	**Supported Actions**
Mid Range Left Sonar Far Range Left Sonar Far Range Up Sonar	Straight Forward
Front Bumper Near Range Forward Sonar Mid Range Up Sonar Far Range Up Sonar	Straight Reverse
Far Range Up Sonar	Straight Forward
Rear Bumper Far Range Up	Right Forward Left Forward
Near Range Forward Sonar Mid Range Forward Sonar Near Range Right Sonar	Left Reverse

Figure 5: Five common feature detectors and the actions for which they vote. Over a set of several learning experiments (policy 3B), these rules (with some variation) were consistently learned.

Figure 6 shows the actual and internal reinforcement signals as learning takes place. We see a significant improvement in performance up through about 300 time-steps (as measured by the actual reinforcement curve). Also note that the internal reinforcement begins to deviate from the actual reinforcement curve at about 100 time-steps, demonstrating that the reinforcement predictor has begun to learn something useful. By the 600th time-step, this curve has leveled off, indicating that the predictor has learned to anticipate reinforcement to the best of its ability, and that the controller learning has stopped.

Figure 6: Actual (upper curve) and internal reinforcement (lower) over the course of a single learning collision avoidance experiment.

Experiment 2: Environmental Exploration

In this experiment, the goal is to develop a behavior that causes the robot to cover a large area of its environment. The policy reinforces this behavior by rewarding the robot for moving in a straight forward direction, and punishing reverse movements. As it is still important to avoid obstacles, this rule is combined with policy 3B from experiment 1. It is possible to assign different weights to each of these two sub-policies to reflect learning priorities. It was observed that the sub-policy given higher priority was learned faster and typically dominated the other. The parameters reported in Appendix E are selected such that both aspects of the desired behavior are present after learning has taken place.

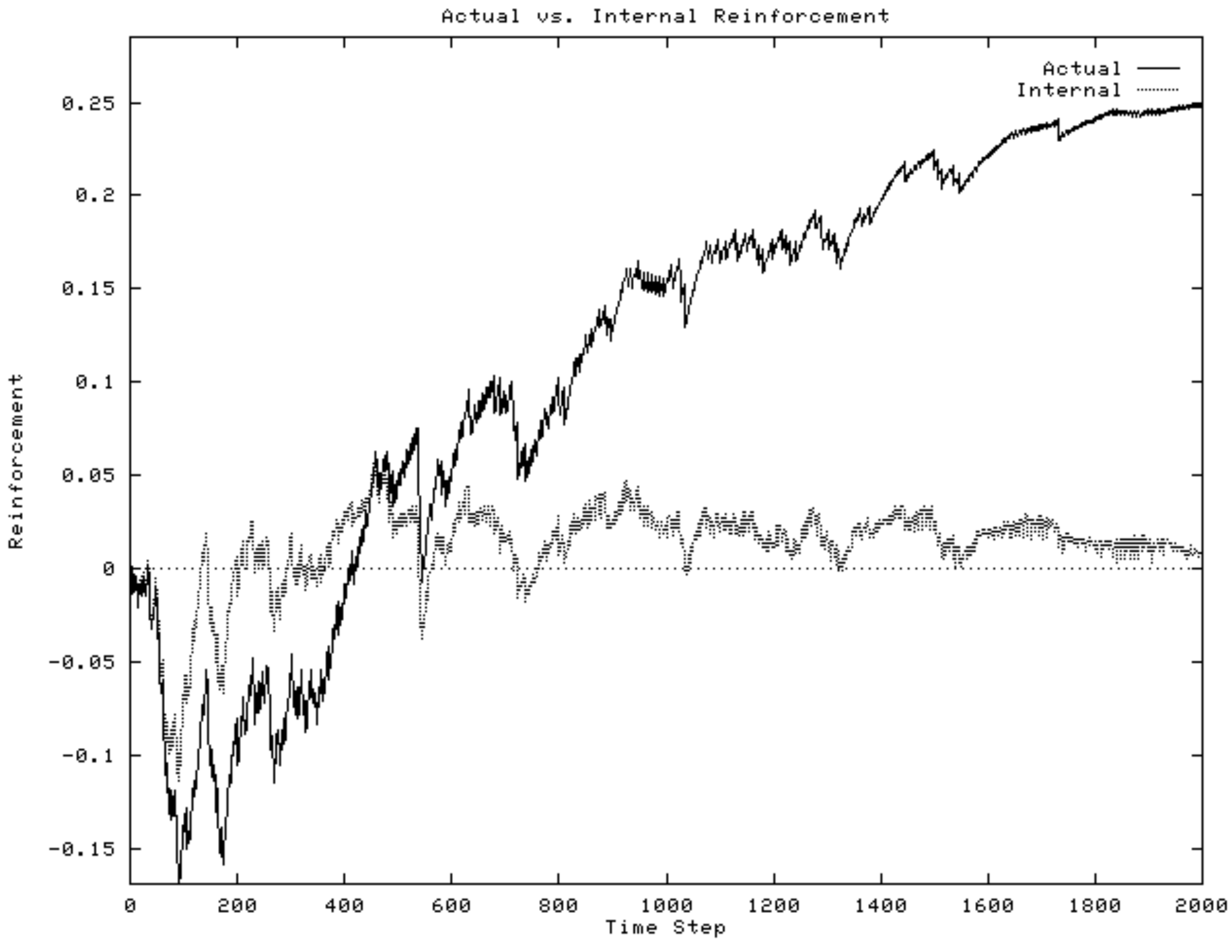

Figure 7: Actual (dark curve) and Internal (gray) reinforcement as the robot learns the environmental exploration task. Note that the actual reinforcement converges to just below 0.25 (as compared with 0.8 for experiment 1). This is due to the fact that the maximum reinforcement that the system can receive at a single step for this task is 0.3.

Figure 7 shows the results of one such experiment that was done in simulation. The added difficulty of this task (over task #1) is illustrated by the fact that the system required nearly 2000 time steps to converge to a reasonable solution. In addition, the system received (on average) negative reinforcement for the first 400 time steps (as opposed to only a few in the first task). This added difficulty comes from the direct conflict between the two halves of the behavior specification. Without barriers, the single optimal action choice is to move forward (for which R=0.3). However, when a barrier is encountered, the system must learn to suppress the forward movement and generate a forward turn early enough such that a collision is avoided (R=0.1). In the simple obstacle avoidance task, the forward movement does not have a special status; when there are no barriers to immediately contend with, the control network can generate a movement in any direction, for which it will receive the same reinforcement. When a

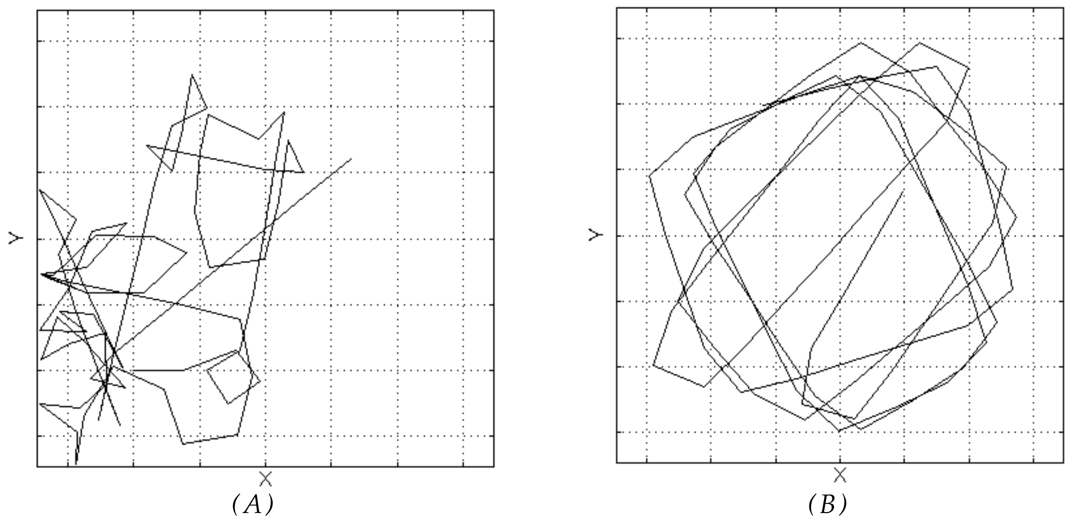

Figure 8: Path of the simulated robot during the first 100 time steps (A) and the last 100 time steps (B). Walls are located along the edges of the environment.

barrier is then encountered, very little learning is required to suppress movements that would lead to a collision.

Figure 8 illustrates the behavior of the robot before and after learning. Prior to learning, the robot makes many unnecessary turns, and collides with the wall on many occasions (8A). Once the learning algorithm has converged, the system has settled on a strategy that makes use of the entire length of the room (8B). In addition, at no point does the robot collide with a wall.

Discussion

This work has drawn on the results of TD (Temporal Difference) Learning of Sutton (Sutton, 1988), as well as those on Q-Learning (Barto & Bradtke, 1991; Watkins & Dayan, 1992). The primary difference with our algorithm is in its more neural orientation, and in the type of predictive information that is stored at each state. In our case, we only store the expected future discounted reinforcement (Watkin's V() function), as opposed to the expected future reinforcement relative to the next action to be taken (Q-values). The information provided by the Q-values is given to some degree by the activity levels of the output units, in that those actions that have the higher Q-values will tend to acquire higher and higher activity levels as learning progresses.

Coding State

Our primary contribution has been an attempt to identify more efficient coding schemes for state space representation. In this case, efficiency constitutes the amount of experience necessary to yield a competent reactive policy (how many times does one need to visit different regions of the state space), and the amount of time required to actually learn the policy. For the robot experiments described above, learning was performed in real time (as the actions were being executed by the robot), and the robot was allowed at most 60 minutes to acquire its program (about 2000 time steps). These results are possible through a combination of the reinforcement predictor (as discussed earlier), and the input state coding scheme used at the feature detector layer.

The feature detector learning algorithm is similar to the Kohonen self-organizing feature map (Kohonen, 1989) in two key ways:

1. Feature units are allocated to the input space in a distribution that reflects the set of inputs that is actually presented to the system. Thus, computing hardware tends to be devoted only to areas of the input space where it will be needed.

2. At the local level (looking at a neighborhood around a feature detector), the mapping from input space to feature space is somewhat topology preserving. What this means is that two similar input patterns will map to close neighbors in the feature detector space. As the two input patterns move away from each other, the likelihood that their feature detector representations share overlapping units drops quickly.

However, there are several important differences:

1. The allocation of feature detectors across the input space is not only determined by the distribution of the inputs, but this distribution is biased by the behavioral relevance of finely partitioning specific regions of the input space. In other words, regions of the input space that share a common set of appropriate actions will tend to be allocated a smaller number of feature detectors. However, if the set of appropriate actions changes quickly within a small region of the input space, then a larger number of feature detectors will be concentrated in this region.

2. The connection matrix from input units to feature detectors is sparse. Hence, an individual

feature unit will have only a limited perspective on the actual input state. Computationally, this can be important: there are many cases when we would like to make a decision based upon only some subset of the input state, and not be distracted by incidental associations with other irrelevant elements. As an example, consider the situation when the robot moves forward and collides with an obstacle. One of the most important associations that must be made immediately is that given a tactile input to the front, the robot should choose to move backwards. This choice should be made in general and should not depend upon other contextual information.

3. Unlike the Kohonen algorithm, more than one feature detector unit is allowed to become active at any one time (as determined by the *local winner-take-all* computation). This property ensures that despite (2), the entire relevant input space can be represented in the population of active feature detector units.

Similarities can also be drawn to the work of von der Malsburg (1987). In this style of self-organizing feature map, the dynamics of the local winner-take-all mechanism allow for the formation of a standing wave (multiple peaks) in the feature layer. Learning the map from input space to feature units is implemented using a normalized Hebbian algorithm. The result is a topology preserving map: as the input slowly changes its position in the input space, the standing wave shifts its position.

In our work, rather than a perfect topological representation, what tend to form are *fractionated topologies*. In other words, islands in the feature detector space will form, within which a topological relationship can be identified. However at some boundary, the topological relationship ceases to hold. Often in the experiments described above, islands would form such that one end of the island was sensitive to obstacles far in front of the robot. As the robot moved closer to an obstacle, the pattern of activity within the island would slowly shift to the other end (the physical feature detector space is only one-dimensional, and hence an island has only has two ends). At either end of the islands, feature detectors would be involved in other aspects of the task (not related to detecting the position of the robot relative to an obstacle in front of it), or would not be involved at all in the decision process.

Efficient state space representation for reinforcement learning as applied to robot learning has also been addressed in Mahadevan & Connell (1992) using statistical clustering techniques.

Learning the Appropriate Action to Generate

The local *winner-take-all* operation at the feature detector layer ensures that only a small subset of feature units become active at any one time. Therefore, moderately different input activity patterns activate different sets of feature detector units. Because of this property, and by the nature of the Hebbian/Anti-Hebbian algorithm, learning that is performed in one region of the feature space tends not to interfere with learning in other regions of the space, as is the case with backpropagation-based reinforcement learning algorithms (Williams, 1987).

This style of mapping an intermediate representation of the input state onto some output value is similar to what is done in the CMAC model (Albus, 1975). Here, a hand-selected hashing function activates a small subset of intermediate units. The hashing function is such that the closer two distinct input patterns are in the input space, the more the corresponding sets of intermediate units will overlap. The key difference in our architecture is that the 'hashing function' is not selected by hand, but is rather developed based upon the representational requirements of the task.

Conclusions

Reactive or behavior-based systems have been offered as alternative approaches to the more traditional AI-based techniques for the design of intelligent control systems for autonomous robots. Their ability to deal with uncertainties in sensing and in action generation, as well as their limited computational requirements have allowed them to demonstrate some level of success beyond that of AI systems. However, these systems are constructed in a trial-and-error fashion, and can often require long periods of programming time to reliably achieve the desired behavior. Moreover, the determination of an appropriate control program is something of a black art where the criteria is “Does it work?”.

This paper has presented an approach to constructing reactive control systems that allows the programmer (or teacher) to specify only the desired behavior of the program. This specification comes in the form of a reinforcement policy (rewards and punishments for certain behaviors that the robot exhibits), from which the learning algorithm must infer a control policy that attempts to maximize the rewards and minimize the punishments that are received. Specifying programs in this way is useful because the specification happens at a level that is easy for the programmer to express and understand, and yet the programs are evaluated within the environment in which they must ultimately perform.

The problem of programming, however, has not gone away. It has only shifted from specifying sets of reactive rules to specifying reinforcement policies that lead the learning system to discover sets of rules that accomplish the desired task. As was demonstrated in our experiments described above, this second process is also an iterative one, where the final behavior is observed, the reinforcement policy is adjusted, and the learning process is then repeated.

The current challenge is to construct principles from which reinforcement policies can be designed, such that the time required to learn a desired behavior is minimized. With this goal in mind, two directions are being pursued:

- **Staged Learning** is a technique in which partial solutions to a problem are first taught to the robot before it is expected to solve the entire problem (Lewis, Fagg, & Solidum, 1992; Lin, 1993). This is accomplished by first presenting the robot with a reinforcement policy that encourages the development of the partial solutions. Once the robot has obtained this intermediate level of capability, the reinforcement policy is changed such that the robot is then required to solve the entire problem in order to receive positive reinforcement. This technique can be used to lead the robot along a specific path of learning, significantly reducing the amount of search that is necessary to discover a program that produces the desired behavior. The behavior-based decompositions for the learning system described in (Mahadevan & Connell, 1992) also has some interesting parallels to staged learning.

- **Learning by Demonstration** is an approach in which motor programs are demonstrated to the robot by the human (Fagg, 1993; Handleman & Lane, 1993; Lin, 1993; Liu & Asada, 1992). The robot first learns to mimic the behavior of the human through a more supervised learning approach. Reinforcement learning is then applied to further refine the learned motor programs such that they are better matched with the robot's own sensory and actuation abilities.

Acknowledgments

The authors would like to thank Dr. Michael Arbib, Gaurav Sukhatme, Mike McHenry, Monica Jan, Nicolas Schweighofer, Tony Lewis, Man-Wai Jason Chu, Parag Batavia, Anand Ramakrishna, Dan Spitzley, and Mathew Lamb for their comments, suggestions, and help during the course of this work. The neural model has been implemented in NSL (Neural Simulation Language (Weitzenfeld, 1991)). Some experiments have been performed using the Erratic Robot Simulator, written by Kurt Konolige at SRI International. This work has been supported in part by grants from the University of Southern California Graduate School, School of Engineering, and Computer Science Department, the Human Frontiers Science Program, and the National Science Foundation.

More information may be obtained at the following locations:

Robotics Research Laboratory:	http://www.usc.edu/dept/robotics/
NSL:	http://www-hbp.usc.edu:8376/HBP/tools/NSL/home.html

References

Albus, J. S. (1975). A New Approach to Manipulator Control: The Cerebellar Model Articulation Controller (CMAC). Transactions of the ASME Journal of Dynamic Systems, Measurement, and Control, **97**, 220-227.

Arbib, M. A. (1989). Schemas and Neural Networks for Sixth Generation Computing. Journal of Parallel and Distributed Computing, **6**, 185-216.

Arkin, R. C. (1990). Integrating Behavioral, Perceptual, and World Knowledge in Reactive Navigation. Robotics and Autonomous Systems, **6**, 105-122.

Barto, A. G., & Bradtke, S. H. (1991). Real-Time Learning and Control using Asynchronous Dynamic Programming (TR No. 91-57). Department of Computer Science, University of Massachusetts, Amherst.

Barto, A. G., Sutton, R. S., & Anderson, C. W. (1983). Neuron-like Adaptive Elements That Can Solve Difficult Learning Control Problems. IEEE Transactions on Systems, Man, and Cybernetics, **SMC-5**, 834-46.

Bekey, G. A., & Tomovic, R. (1986). Reflex Control of Robot Actions. In IEEE International Conference on Robotics and Automation, (pp. 240-47). San Francisco: IEEE Press.

Brooks, R. A. (1986). A Robust Layered Control System for a Mobile Robot. IEEE J. Robotics and Automation, **RA-2**(RA-2), 14-23.

Brooks, R. A. (1991). Intelligence Without Reason (A.I. Memo No. 1293). MIT.

Didday, R. L. (1976). A Model of Visuomotor Mechanisms in the Frog Optic Tectum. Mathematical Biosciences, **30**, 169-180.

Fagg, A. H. (1993). Developmental Robotics: A New Approach to the Specification of Robot Programs. In G. A. Bekey & K. Y. Goldberg (Eds.), Neural Networks in Robotics (pp. 459-86). Norwell, Massachusetts: Kluwer Academic Publishers.

Fagg, A. H., & Arbib, M. A. (1992). A Model of Primate Visual-Motor Conditional Learning. Journal of Adaptive Behavior, **1**(1), 3-37.

Handleman, D. A., & Lane, S. H. (1993). Fast Sensorimotor Skill Acquisition Based on Rule-Based Training of Neural Nets. In G. A. Bekey & K. Y. Goldberg (Eds.), Neural Networks in Robotics (pp. 255-70). Norwell, Massachusetts: Kluwer Academic Publishers.

Klopf, A. H. (1982). The Hedonistic Neuron. Washington, D. C.: Hemisphere.

Kohonen (1989). Self-Organization and Associative Memory. New York: Springer-Verlag.

Lewis, M. A., Fagg, A. H., & Solidum, A. (1992). Genetic Programming Approach to the Construction of a Neural Network for Control of a Walking Robot. In Proceedings of the 1992 IEEE Conference on Robotics and Automation, (pp. 2618-23). Nice, France:

Lin, J. J. (1993). Hierarchical Learning of Robot Skills by Reinforcement. In Proceedings of the 1993 IEEE Conference on Neural Networks, (pp. 181-6). San Francisco, California: IEEE.

Liu, S., & Asada, H. (1992). Transferring Manipulative Skills to Robots - Representation and Acquisition of Tool Manipulative Skills Using a Process Dynamic Model. Journal of Dynamic

Systems Measurement and Control-Transactions of the ASME, **114**(2), 220-8.

Maes, P., & Brooks, R. A. (1990). Learning to Coordinate Behaviors. In AAAI, (pp. 796-802). Boston, MA.

Mahadevan, S., & Connell, J. (1992). Automatic Programming of Behavior-Based Robots Using Reinforcement Learning. Artificial Intelligence, **55**, 311-365.

Samuel, A. L. (1967). Some Studies in Machine Learning Using the Game of Checkers. IBM Journal of Research and Development, **11**, 601-17.

Sutton, R. S. (1984) Temporal Credit Assignment in Reinforcement Learning. University of Massachusetts, Amherst.

Sutton, R. S. (1988). Learning to Predict by the Methods of Temporal Differences. In Machine Learning 3 (pp. 9-44). Boston: Kluwer Academic Publishers.

Verschure, P. F. M. J., Kröse, B. J. A., & Pfeifer, R. (1992). Distributed Adaptive Control: The Self-Organization of Structured Behavior. Robotics and Autonomous Systems, **9**, 181-196.

von der Malsburg, C. (1973). Self-organizing of Orientation Sensitive Cells in the Striate Cortex. Kybernetik, **14**, 85-100.

von der Malsburg, C. (1987). Ordered Retinotectal Projections and Brain Organization. In F. E. Yates, A. Garfinkel, D. O. Walter, & G. Yates (Eds.), Self-Organizing Systems - The Emergence of Order (pp. 265-78). New York: Plenum Press.

Watkins, C. J. C. H., & Dayan, P. (1992). Q-Learning. Machine Learning, **8**(3), 279-92.

Weitzenfeld, A. (1991). NSL - Neural Simulation Language Version 2.1 (TR No. CNE-91-05). Center for Neural Engineering, University of Southern California, Los Angeles, CA.

Williams, R. J. (1987). Reinforcement Learning Connectionist Systems (TR No. NU-CCS-87-3). Northeastern University, College of Computer Science.

Appendix A - Non-linear Activation Function for Feature Detectors

A non-linear activation function is used for the feature detector units (Figure A.1, equation A.1). The non-linear function has a sliding threshold ($thresh_j(t)$) that depends upon the maximum net input that the unit has experienced during its lifetime (equation A.2):

$$h_j^t(x) = \begin{cases} x & x \geq thresh_j(t) \\ thresh_j(t) * \left(\dfrac{x}{thresh_j(t)} \right)^{scale} & 0 < x < thresh_j(t) \\ 0 & otherwise \end{cases} \tag{A.1}$$

$$thresh_j(t) = \left(\underset{0 \leq \tau \leq t}{Max} \{ Z_j(\tau) \} \right) - offset \tag{A.2}$$

where:

$h_j^t(\ )$ is the non-linear activation function for unit j at learning time step t.

$thresh_j(t)$ is the sliding threshold for unit j.

$Z_j(t)$ is the net input for unit j at time step t. In other words,

$Z_j(t) = \sum_i \left(W_{ij} * I_i(t) \right) + Noise_j(t)$ (see equation 1).

$scale$ is an exponent that determines the steepness the non-linear region.

Once the feature detector unit has achieved a high level of input (over the process of learning), in order for it to become active again, this same high level must be achieved. For lower levels of input, the unit will not activate to any significant degree, allowing other feature detector units to win the *local winner-take-all* competition.

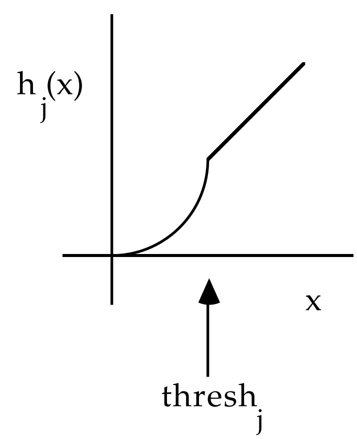

Figure A.1: non-linear activation function for feature detector unit j.

Appendix B - Temporal Difference Learning Algorithm

The earlier discussion on reinforcement prediction assumed a mechanism for learning a predictor module. Here, we briefly lay out the temporal difference learning algorithm due to Sutton (1984, 1988) and show how our architecture makes use of it.

The goal is that given some fixed control policy, we want to learn a function $P(\ )$ that maps some representation of the system state at time t ($x(t)$) into an expectation of future reward. More precisely, we want $P(\ )$ to satisfy the following:

$$P(x(t)) = E\left\{\sum_{\tau=t}^{\infty} \lambda^{\tau-t} R(\tau)\right\} \tag{B.1}$$

We represent $P(\ )$ as a function of some parameter vector $\overline{W}(t)$ and input state $x(t)$, the elements of which are adjusted through time using the Temporal Difference Learning Algorithm:

$$\overline{W}(t+1) = \overline{W}(t) + \mu * R' * \sum_{\tau=0}^{t} \upsilon^{t-\tau} \nabla_{\overline{W}} P\left(x(\tau), \overline{W}(\tau)\right) \tag{B.2}$$

where:

μ is the learning rate.

υ is a discount rate for past experiences

If we define $\overline{\overline{W}}(t)$ as:

$$\overline{\overline{W}}(t) = \sum_{\tau=0}^{t} \upsilon^{t-\tau} \nabla_{\overline{W}} P\left(x(\tau), \overline{W}(\tau)\right) \tag{B.3}$$

then note that:

$$\begin{aligned}\overline{\overline{W}}(t) &= \nabla_{\overline{W}} P\left(x(t), \overline{W}(t)\right) + \upsilon \sum_{\tau=0}^{t-1} \upsilon^{t-\tau-1} \nabla_{\overline{W}} P\left(x(\tau), \overline{W}(\tau)\right) \\ &= \nabla_{\overline{W}} P\left(x(t), \overline{W}(t)\right) + \upsilon \overline{\overline{W}}(t-1)\end{aligned} \tag{B.4}$$

and:

$$\overline{W}(t+1) = \overline{W}(t) + \mu * R' * \overline{\overline{W}}(t) \tag{B.5}$$

Hence, $P(\ )$ can be learned incrementally, only requiring us to store the vectors $\overline{W}(t)$ and $\overline{\overline{W}}(t)$ in memory.

In the architecture presented here, $P(\ )$ is a linear function of the feature detector states, i.e.:

$$x(t) = G(t) \tag{B.6}$$

and

$$P\left(x(t), \overline{W}(t)\right) = G(t) \cdot \overline{W}(t) \tag{B.7}$$

Equation B.4 simplifies to:

$$\overline{\overline{W}}(t) = G(t) + \upsilon \overline{\overline{W}}(t-1) \tag{B.8}$$

The *local winner-take-all* component of the algorithm has a tendency to encourage rather different regions of the input space to map to orthogonal representations in the G vector. This property is important in light of our linear representation for the predictor ($P(\ )$) in that it reduces the interference due to learning in disparate regions of the state space.

Appendix C - Primary Variables

Variable	Description
Network Layers	
I	Input vector
F	Feature Detector Activity Levels
$Winner$	Vector indicating local winners in the feature detector layer
G	Feature Detector Activity after local Winner-Take-All
A	Action Selection Activity Levels
W	The matrix representing the connections from I to F.
W'	The matrix representing the connections from G to A.
$Noise$	Noise injected into the Feature Detector Layer (F)
$Noise'$	Noise injected into the Action Selection Layer (A)
Feature Detector Non-Linearity	
$h_j^t(\ )$	Feature detector non-linear activation function for unit j at time t.
Z	Vector of net inputs into the feature detector layer.
$thresh_j$	Threshold of non-linearity
Reinforcement Signals	
$R(t)$	Reinforcement for time t
$R'(t)$	Internal Reinforcement for time t
Predictor	
$P(x(t))$	Predicted Expected Future Reinforcement for state $x(t)$
$\overline{W}$	Predictor parameters
$\overline{\overline{W}}$	Predictor eligibilities
Eligibility	
e_{ij}	Eligibility for connection corresponding to W_{ij}
e'_{jk}	Eligibility for connection corresponding to W'_{jk}
Weight Solidification	
$\overline{R}_j$	Measure of the j'th feature detector's positive participation in a program
M_j	Synaptic plasticity modulation for feature detector j

Appendix D - Key Parameters for Robot Learning Experiments

Parameter	Value	Description
Network Architecture		
Inputs	24	Number of input units.
Feature detectors	250	Number of feature detectors
Outputs	6	Number of outputs.
N	+/-5	Neighborhood size
N'	+/-1	Local neighborhood size
γ	0.35	Local neighborhood activation scale factor
Connection Matrix Initialization		
I_F_Weights_prob	0.35	Probability of connecting an arbitrary Input/Feature pair.
F_A_Weights_prob	1.0	Probability of connecting an arbitrary Feature/Action pair.
Noise		
noise_f_gain	0.05	Feature noise is uniformly distributed between +/- gain
noise_a_gain	0.15	Action noise is uniformly distributed between +/- gain
Feature detector non-linearity		
scale	4.0	Steepness of non-linearity.
offset	0.05	Sensitivity of feature detector.
Weight Update		
α	0.1	Learning rate for W
α'	0.15	Learning rate for W'
Eligibility		
τ	1.0	Temporal width of eligibility memory for W
τ'	1.0	Temporal width of eligibility memory for W'
Reinforcement Prediction		
λ	0.85	Discount factor for future reinforcement
μ	0.01	Predictor learning rate
υ	0.5	Discount factor for past experiences.
Weight Solidification		
accum_rate	0.05	Rate of solidification

Appendix E - Reinforcement Policies

The first condition to match determines the reinforcement given by the teacher.

Obstacle Avoidance Policy

Hitting obstacle	-1.0
Not hitting obstacle	0.8

Environmental Exploration Policy

Hitting obstacle	-1.0
Straight forward movement	0.3
Forward turn movement	0.1
Reverse movement	-0.8