Chapter 11

Delayed-Reinforcement Learning

11.1 The General Problem

Imagine a robot that exists in an environment in which it can sense and act. Suppose (as an extreme case) that it has no idea about the effects of its actions. That is, it doesn't know how acting will change its sensory inputs. Along with its sensory inputs are "rewards," which it occasionally receives. How should it choose its actions so as to maximize its rewards over the long run? To maximize rewards, it will need to be able to predict how actions change inputs, and in particular, how actions lead to rewards.

We formalize the problem in the following way: The robot exists in an environment consisting of a set, $\mathcal{S}$, of states. We assume that the robot's sensory apparatus constructs an input vector, $\mathbf{X}$, from the environment, which informs the robot about which state the environment is in. For the moment, we will assume that the mapping from states to vectors is one-to-one, and, in fact, will use the notation $\mathbf{X}$ to refer to the state of the environment as well as to the input vector. When presented with an input vector, the robot decides which action from a set, $\mathcal{A}$, of actions to perform. Performing the action produces an effect on the environment—moving it to a new state. The new state results in the robot perceiving a new input vector, and the cycle repeats. We assume a discrete time model; the input vector at time $t = i$ is $\mathbf{X}_i$, the action taken at that time is a_i, and the expected reward, r_i, received at $t = i$ depends on the action taken and on the state, that is $r_i = r(\mathbf{X}_i, a_i)$. The learner's goal is to find a *policy*, $\pi(\mathbf{X})$, that maps input vectors to actions in such a way that maximizes rewards accumulated over time. This type of learning is called *reinforcement learning*. The learner must find the policy by trial and error; it has no initial knowledge of the effects of its actions. The situation is as shown in Fig. 11.1.

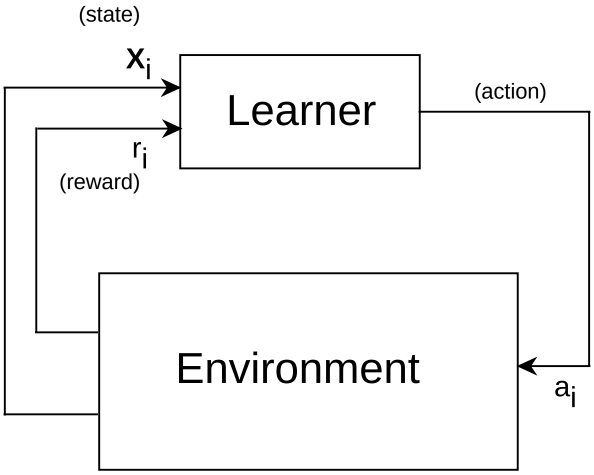

Figure 11.1: Reinforcement Learning

11.2 An Example

A "grid world," such as the one shown in Fig. 11.2 is ofEleven used to illustrate reinforcement learning. Imagine a robot initially in cell (2,3). The robot receives input vector (x_1, x_2) telling it what cell it is in; it is capable of four actions, n, e, s, w moving the robot one cell up, right, down, or left, respectively. It is rewarded one negative unit whenever it bumps into the wall or into the blocked cells. For example, if the input to the robot is (1,3), and the robot chooses action w, the next input to the robot is still (1,3) and it receives a reward of -1. If the robot lands in the cell marked G (for goal), it receives a reward of $+10$. Let's suppose that whenever the robot lands in the goal cell and gets its reward, it is immediately transported out to some random cell, and the quest for reward continues.

A *policy* for our robot is a specification of what action to take for every one of its inputs, that is, for every one of the cells in the grid. For example,

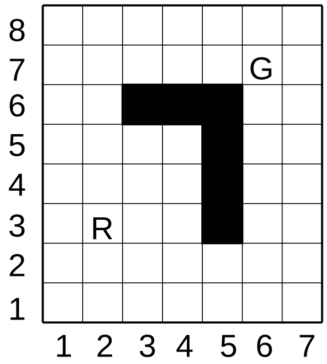

Figure 11.2: A Grid World

a component of such a policy would be "when in cell (3,1), move right." An *optimal policy* is a policy that maximizes long-term reward. One way of displaying a policy for our grid-world robot is by an arrow in each cell indicating the direction the robot should move when in that cell. In Fig. 11.3, we show an optimal policy displayed in this manner. In this chapter we will describe methods for learning optimal policies based on reward values received by the learner.

11.3 Temporal Discounting and Optimal Policies

In delayed reinforcement learning, one often assumes that rewards in the distant future are not as valuable as are more immediate rewards. This preference can be accomodated by a *temporal discount factor*, $0 \leq \gamma < 1$. The present value of a reward, r_i, occuring i time units in the future, is taken to be $\gamma^i r_i$. Suppose we have a policy $\pi(\mathbf{X})$ that maps input vectors into actions, and let $r_i^{\pi(\mathbf{X})}$ be the reward that will be received on the i-th time step after one begins executing policy π starting in state $\mathbf{X}$. Then the total reward accumulated over all time steps by policy π beginning in state $\mathbf{X}$ is:

$$V^\pi(\mathbf{X}) = \sum_{i=0}^{\infty} \gamma^i r_i^{\pi(\mathbf{X})}$$

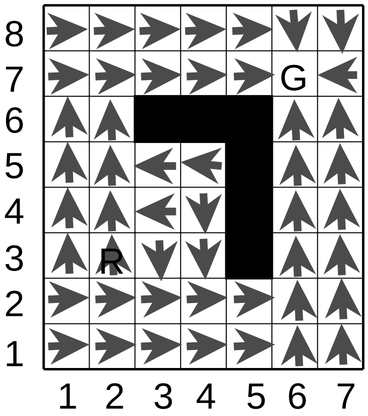

Figure 11.3: An Optimal Policy in the Grid World

One reason for using a temporal discount factor is so that the above sum will be finite. An optimal policy is one that maximizes $V^{\pi}(\mathbf{X})$ for all inputs, $\mathbf{X}$.

In general, we want to consider the case in which the rewards, r_i, are random variables and in which the effects of actions on environmental states are random. In Markovian environments, for example, the probability that action a in state $\mathbf{X}_i$ will lead to state $\mathbf{X}_j$ is given by a transition probability $p[\mathbf{X}_j|\mathbf{X}_i, a]$. Then, we will want to maximize *expected* future reward and would define $V^{\pi}(\mathbf{X})$ as:

$$V^{\pi}(\mathbf{X}) = E\left[\sum_{i=0}^{\infty} \gamma^i r_i^{\pi(\mathbf{X})}\right]$$

In either case, we call $V^{\pi}(\mathbf{X})$ the *value* of policy π for input $\mathbf{X}$.

If the action prescribed by π taken in state $\mathbf{X}$ leads to state $\mathbf{X}'$ (randomly according to the transition probabilities), then we can write $V^{\pi}(\mathbf{X})$ in terms of $V^{\pi}(\mathbf{X}')$ as follows:

$$V^{\pi}(\mathbf{X}) = r[\mathbf{X}, \pi(\mathbf{X})] + \gamma \sum_{\mathbf{X}'} p[\mathbf{X}'|\mathbf{X}, \pi(\mathbf{X})] V^{\pi}(\mathbf{X}')$$

where (in summary):

γ = the discount factor,

$V^\pi(\mathbf{X})$ = the value of state $\mathbf{X}$ under policy π,

$r[\mathbf{X}, \pi(\mathbf{X})]$ = the expected immediate reward received when we execute the action prescribed by π in state $\mathbf{X}$, and

$p[\mathbf{X}'|\mathbf{X}, \pi(\mathbf{X})]$ = the probability that the environment transitions to state $\mathbf{X}'$ when we execute the action prescribed by π in state $\mathbf{X}$.

In other words, the value of state $\mathbf{X}$ under policy π is the expected value of the immediate reward received when executing the action recommended by π plus the average value (under π) of all of the states accessible from $\mathbf{X}$.

For an optimal policy, π^* (and no others!), we have the famous "optimality equation:"

$$V^{\pi^*}(\mathbf{X}) = \max_a \left[r(\mathbf{X}, a) + \gamma \sum_{\mathbf{X}'} p[\mathbf{X}'|\mathbf{X}, a] V^{\pi^*}(\mathbf{X}') \right]$$

The theory of dynamic programming (DP) [Bellman, 1957, Ross, 1983] assures us that there is at least one optimal policy, π^*, that satisfies this equation. DP also provides methods for calculating $V^{\pi^*}(\mathbf{X})$ and at least one π^*, assuming that we know the average rewards and the transition probabilities. If we knew the transition probabilities, the average rewards, and $V^{\pi^*}(\mathbf{X})$ for all $\mathbf{X}$ and a, then it would be easy to implement an optimal policy. We would simply select that a that maximizes $r(\mathbf{X}, a) + \gamma \sum_{\mathbf{X}'} p[\mathbf{X}'|\mathbf{X}, a] V^{\pi^*}(\mathbf{X}')$. That is,

$$\pi^*(\mathbf{X}) = \arg\max_a \left[r(\mathbf{X}, a) + \gamma \sum_{\mathbf{X}'} p[\mathbf{X}'|\mathbf{X}, a] V^{\pi^*}(\mathbf{X}') \right]$$

But, of course, we are assuming that we do not know these average rewards nor the transition probabilities, so we have to find a method that effectively learns them.

If we had a model of actions, that is, if we knew for every state, $\mathbf{X}$, and action a, which state, $\mathbf{X}'$ resulted, then we could use a method called *value iteration* to find an optimal policy. Value iteration works as follows: We begin by assigning, randomly, an *estimated value* $\hat{V}(\mathbf{X})$ to every state, $\mathbf{X}$. On the i-th step of the process, suppose we are at state $\mathbf{X}_i$ (that is, our input on the i-th step is $\mathbf{X}_i$), and that the estimated value of state $\mathbf{X}_i$ on the i-th step is $\hat{V}_i(\mathbf{X}_i)$. We then select that action a that maximizes the estimated value of the predicted subsequent state. Suppose this subsequent state having the highest estimated value is $\mathbf{X}_i'$. Then we update the estimated value, $\hat{V}_i(\mathbf{X}_i)$, of state $\mathbf{X}_i$ as follows:

$$\hat{V}_i(\mathbf{X}) = (1 - c_i)\hat{V}_{i-1}(\mathbf{X}) + c_i\left[r_i + \gamma\hat{V}_{i-1}(\mathbf{X}_i')\right]$$

if $\mathbf{X} = \mathbf{X}_i$,

$$= \hat{V}_{i-1}(\mathbf{X})$$

otherwise.

We see that this adjustment moves the value of $\hat{V}_i(\mathbf{X}_i)$ an increment (depending on c_i) closer to $\left[r_i + \gamma\hat{V}_i(\mathbf{X}_i')\right]$. Assuming that $\hat{V}_i(\mathbf{X}_i')$ is a good estimate for $V_i(\mathbf{X}_i')$, then this adjustment helps to make the two estimates more consistent. Providing that $0 < c_i < 1$ and that we visit each state infinitely often, this process of value iteration will converge to the optimal values.

Discuss synchronous dynamic programming, asynchronous dynamic programming, and policy iteration.

11.4 Q-Learning

Watkins [Watkins, 1989] has proposed a technique that he calls *incremental dynamic programming.* Let $a; \pi$ stand for the policy that chooses action a once, and thereafter chooses actions according to policy π. We define:

$$Q^{\pi}(\mathbf{X}, a) = V^{a;\pi}(\mathbf{X})$$

Then the optimal value from state $\mathbf{X}$ is given by:

$$V^{\pi^*}(\mathbf{X}) = \max_a Q^{\pi^*}(\mathbf{X}, a)$$

This equation holds only for an optimal policy, π^*. The optimal policy is given by:

$$\pi^*(\mathbf{X}) = \arg\max_a Q^{\pi^*}(\mathbf{X}, a)$$

Note that if an action a makes $Q^{\pi}(\mathbf{X}, a)$ larger than $V^{\pi}(\mathbf{X})$, then we can improve π by changing it so that $\pi(\mathbf{X}) = a$. Making such a change is the basis for a powerful learning rule that we shall describe shortly.

Suppose action a in state $\mathbf{X}$ leads to state $\mathbf{X}'$. Then using the definitions of Q and V, it is easy to show that:

$$Q^{\pi}(\mathbf{X}, a) = r(\mathbf{X}, a) + \gamma E[V^{\pi}(\mathbf{X}')]$$

where $r(\mathbf{X}, a)$ is the average value of the immediate reward received when we execute action a in state $\mathbf{X}$. For an optimal policy (and no others), we have another version of the optimality equation in terms of Q values:

$$Q^{\pi^*}(\mathbf{X}, a) = \max_a \left[r(\mathbf{X}, a) + \gamma E\left[Q^{\pi^*}(\mathbf{X}', a) \right] \right]$$

for all actions, a, and states, $\mathbf{X}$. Now, if we had the optimal Q values (for all a and $\mathbf{X}$), then we could implement an optimal policy simply by selecting that action that maximized $r(\mathbf{X}, a) + \gamma E\big[Q^{\pi^*}(\mathbf{X}', a)\big]$.
That is,

$$\pi^*(\mathbf{X}) = \arg\max_a \left[r(\mathbf{X}, a) + \gamma E\left[Q^{\pi^*}(\mathbf{X}', a) \right] \right]$$

Watkins' proposal amounts to a TD(0) method of learning the Q values. We quote (with minor notational changes) from [Watkins & Dayan, 1992, page 281]:

> "In Q-Learning, the agent's experience consists of a sequence of distinct stages or *episodes*. In the i-th episode, the agent:
>
> - observes its current state $\mathbf{X}_i$,
> - selects [using the method described below] and performs an action a_i,
> - observes the subsequent state $\mathbf{X}'_i$,
> - receives an immediate reward r_i, and
> - adjusts its Q_{i-1} values using a learning factor c_i, according to:
>
> $$Q_i(\mathbf{X}, a) = (1 - c_i)Q_{i-1}(\mathbf{X}, a) + c_i[r_i + \gamma V_{i-1}(\mathbf{X}'_i)]$$
>
> if $\mathbf{X} = \mathbf{X}_i$ and $a = a_i$,
>
> $$= Q_{i-1}(\mathbf{X}, a)$$
>
> otherwise,

> where
>
> $$V_{i-1}(\mathbf{X}') = \max_b [Q_{i-1}(\mathbf{X}', b)]$$
>
> is the best the agent thinks it can do from state $\mathbf{X}'$. ... The initial Q values, $Q_0(\mathbf{X}, a)$, for all states and actions are assumed given."

Using the current Q values, $Q_i(\mathbf{X}, a)$, the agent always selects that action that maximizes $Q_i(\mathbf{X}, a)$. Note that only the Q value corresponding to the state just exited and the action just taken is adjusted. And that Q value is adjusted so that it is closer (by an amount determined by c_i) to the sum of the immediate reward plus the discounted maximum (over all actions) of the Q values of the state just entered. If we imagine the Q values to be predictions of ultimate (infinite horizon) total reward, then the learning procedure described above is exactly a TD(0) method of learning how to predict these Q values. Q learning strengthens the usual TD methods, however, because TD (applied to reinforcement problems using value iteration) requires a one-step lookahead, using a model of the effects of actions, whereas Q learning does not.

A convenient notation (proposed by [Schwartz, 1993]) for representing the change in Q value is:

$$Q(\mathbf{X}, a) \xleftarrow{\beta} r + \gamma V(\mathbf{X}')$$

where $Q(\mathbf{X}, a)$ is the new Q value for input $\mathbf{X}$ and action a, r is the immediate reward when action a is taken in response to input $\mathbf{X}$, $V(\mathbf{X}')$ is the maximum (over all actions) of the Q value of the state next reached when action a is taken from state $\mathbf{X}$, and β is the fraction of the way toward which the new Q value, $Q(\mathbf{X}, a)$, is adjusted to equal $r + \gamma V(\mathbf{X}')$.

Watkins and Dayan [Watkins & Dayan, 1992] prove that, under certain conditions, the Q values computed by this learning procedure converge to optimal ones (that is, to ones on which an optimal policy can be based).

We define $n^i(\mathbf{X}, a)$ as the index (episode number) of the i-th time that action a is tried in state $\mathbf{X}$. Then, we have:

Theorem 11.1 (Watkins and Dayan) *For Markov problems with states* $\{\mathbf{X}\}$ *and actions* $\{a\}$, *and given bounded rewards* $|r_n| \leq R$, *learning rates* $0 \leq c_n < 1$, *and*

$$\sum_{i=0}^{\infty} c_{n^i(\mathbf{X},a)} = \infty, \quad \sum_{i=0}^{\infty} \left[c_{n^i(\mathbf{X},a)}\right]^2 < \infty$$

for all $\mathbf{X}$ *and* a*, then*
$Q_n(\mathbf{X}, a) \rightarrow Q_n^*(\mathbf{X}, a)$ *as* $n \rightarrow \infty$*, for all* $\mathbf{X}$ *and* a*, with probability 1, where* $Q_n^*(\mathbf{X}, a)$ *corresponds to the* Q *values of an optimal policy.*

Again, we quote from [Watkins & Dayan, 1992, page 281]:

> "The most important condition implicit in the convergence theorem ... is that the sequence of episodes that forms the basis of learning must include an infinite number of episodes for each starting state and action. This may be considered a strong condition on the way states and actions are selected—however, under the stochastic conditions of the theorem, no method could be guaranteed to find an optimal policy under weaker conditions. Note, however, that the episodes need not form a continuous sequence—that is the $\mathbf{X}'$ of one episode need not be the $\mathbf{X}$ of the next episode."

The relationships among Q learning, dynamic programming, and control are very well described in [Barto, Bradtke, & Singh, 1994]. Q learning is best thought of as a stochastic approximation method for calculating the Q values. Although the definition of the optimal Q values for any state depends recursively on expected values of the Q values for subsequent states (and on the expected values of rewards), no expected values are explicitly computed by the procedure. Instead, these values are approximated by iterative sampling using the actual stochastic mechanism that produces successor states.

11.5 Discussion, Limitations, and Extensions of Q-Learning

11.5.1 An Illustrative Example

The Q-learning procedure requires that we maintain a table of $Q(\mathbf{X}, a)$ values for all state-action pairs. In the grid world that we described earlier, such a table would not be excessively large. We might start with random entries in the table; a portion of such an intial table might be as follows:

$\mathbf{X}$	a	$Q(\mathbf{X}, a)$	$r(\mathbf{X}, a)$
(2,3)	w	7	0
(2,3)	n	4	0
(2,3)	e	3	0
(2,3)	s	6	0
(1,3)	w	4	-1
(1,3)	n	5	0
(1,3)	e	2	0
(1,3)	s	4	0

Suppose the robot is in cell (2,3). The maximum Q value occurs for $a = w$, so the robot moves west to cell (1,3)—receiving no immediate reward. The maximum Q value in cell (1,3) is 5, and the learning mechanism attempts to make the value of $Q((2, 3), w)$ closer to the discounted value of 5 plus the immediate reward (which was 0 in this case). With a learning rate parameter $c = 0.5$ and $\gamma = 0.9$, the Q value of $Q((2, 3), w)$ is adjusted from 7 to 5.75. No other changes are made to the table at this episode. The reader might try this learning procedure on the grid world with a simple computer program. Notice that an optimal policy might not be discovered if some cells are not visited nor some actions not tried frequently enough.

The learning problem faced by the agent is to associate specific actions with specific input patterns. Q learning gradually *reinforces* those actions that contribute to positive rewards by increasing the associated Q values. Typically, as in this example, rewards occur somewhat after the actions that lead to them—hence the phrase *delayed-reinforcement* learning. One can imagine that better and better approximations to the optimal Q values gradually propagate back from states producing rewards toward all of the other states that the agent frequently visits. With random Q values to begin, the agent's actions amount to a random walk through its space of states. Only when this random walk happens to stumble into rewarding states does Q learning begin to produce Q values that are useful, and, even then, the Q values have to work their way outward from these rewarding states. The general problem of associating rewards with state-action pairs is called the *temporal credit assignment problem*—how should credit for a reward be apportioned to the actions leading up to it? Q learning is, to date, the most successful technique for temporal credit assignment, although a related method, called the *bucket brigade algorithm*, has been proposed by [Holland, 1986].

Learning problems similar to that faced by the agent in our grid world have been thoroughly studied by Sutton who has proposed an architecture, called DYNA, for solving them [Sutton, 1990]. DYNA combines reinforcement learning with planning. Sutton characterizes planning as learning in a simulated world that models the world that the agent inhabits. The agent's model of the world is obtained by Q learning in its actual world, and planning is accomplished by Q learning in its model of the world.

We should note that the learning problem faced by our grid-world robot could be modified to have several places in the grid that give positive rewards. This possibility presents an interesting way to generalize the classical notion of a "goal" in AI planning systems—even in those that do no learning. Instead of representing a goal as a condition to be achieved, we represent a "goal structure" as a set of rewards to be given for achieving various conditions. Then, the generalized "goal" becomes maximizing discounted future reward instead of simply achieving some particular condition. This generalization can be made to encompass so-called goals of *maintenance* and goals of *avoidance*. The example presented above included avoiding bumping into the grid-world boundary. A goal of maintenance, of a particular state, could be expressed in terms of a reward that was earned whenever the agent was in that state and performed an action that transitioned back to that state in one step.

11.5.2 Using Random Actions

When the next pattern presentation in a sequence of patterns is the one caused by the agent's own action in response to the last pattern, we have what is called an *on-line* learning method. In Watkins and Dayan's terminology, in on-line learning the episodes form a continous sequence. As already mentioned, the convergence theorem for Q learning does not require on-line learning; indeed, special precautions must be taken to ensure that on-line learning meets the conditions of the theorem. If on-line learning discovers some good paths to rewards, the agent may fixate on these and never discover a policy that leads to a possibly greater long-term reward. In reinforcement learning phraseology, this problem is referred to as the problem of *exploitation* (of already learned behavior) versus *exploration* (of possibly better behavior).

One way to force exploration is to perform occasional random actions (instead of that single action prescribed by the current Q values). For example, in the grid-world problem, one could imagine selecting an action randomly according to a probability distribution over the actions (n, e, s, and w). This distribution, in turn, could depend on the Q values. For example, we might first find that action prescribed by the Q values and then choose that action with probability 1/2, choose the two orthogonal actions with probability 3/16 each, and choose the opposite action with probability 1/8. This policy might be modified by "simulated annealing" which would gradually increase the probability of the action prescribed by the Q values more and more as time goes on. This strategy would favor exploration at the beginning of learning and exploitation later.

Other methods, also, have been proposed for dealing with exploration, including making unvisited states intrinsically rewarding and using an "interval estimate," which is related to the uncertainty in the estimate of a state's value [Kaelbling, 1993].

11.5.3 Generalizing Over Inputs

For large problems it would be impractical to maintain a table like that used in our grid-world example. Various researchers have suggested mechanisms for computing Q values, given pattern inputs and actions. One method that suggests itself is to use a neural network. For example, consider the simple linear machine shown in Fig. 11.4.

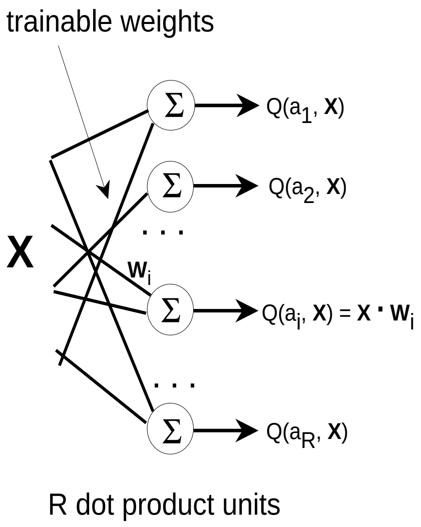

Figure 11.4: A Net that Computes Q Values

Such a neural net could be used by an agent that has R actions to select from. The Q values (as a function of the input pattern $\mathbf{X}$ and the action a_i) are computed as dot products of weight vectors (one for each action) and the input vector. Weight adjustments are made according to a TD(0) procedure to bring the Q value for the action last selected closer to the sum of the immediate reward (if any) and the (discounted) maximum Q value for the next input pattern.

If the optimum Q values for the problem (whatever they might be) are more complex than those that can be computed by a linear machine, a layered neural network might be used. Sigmoid units in the final layer would compute Q values in the range 0 to 1. The TD(0) method for updating Q values would then have to be combined with a multi-layer weight-changing rule, such as backpropagation.

Networks of this sort are able to aggregate different input vectors into regions for which the same action should be performed. This kind of aggregation is an example of what has been called *structural credit assignment.* Combining TD(λ) and backpropagation is a method for dealing with both the temporal and the structural credit assignment problems.

Interesting examples of delayed-reinforcement training of simulated and actual robots requiring structural credit assignment have been reported by [Lin, 1992, Mahadevan & Connell, 1992].

11.5.4 Partially Observable States

So far, we have identified the input vector, $\mathbf{X}$, with the actual state of the environment. When the input vector results from an agent's perceptual apparatus (as we assume it does), there is no reason to suppose that it uniquely identifies the environmental state. Because of inevitable perceptual limitations, several different environmental states might give rise to the same input vector. This phenomenon has been referred to as *perceptual aliasing.* With perceptual aliasing, we can no longer guarantee that Q learning will result in even useful action policies, let alone optimal ones. Several researchers have attempted to deal with this problem using a variety of methods including attempting to model "hidden" states by using internal memory [Lin, 1993]. That is, if some aspect of the environment cannot be sensed currently, perhaps it was sensed once and can be remembered by the agent. When such is the case, we no longer have a Markov problem; that is, the next $\mathbf{X}$ vector, given any action, may depend on a sequence of previous ones rather than just the immediately preceding one. It might be possible to reinstate a Markov framework (over the $\mathbf{X}$'s) if $\mathbf{X}$

includes not only current sensory precepts but information from the agent's memory.

11.5.5 Scaling Problems

Several difficulties have so far prohibited wide application of reinforcement learning to large problems. (The TD-gammon program, mentioned in the last chapter, is probably unique in terms of success on a high-dimensional problem.) We have already touched on some difficulties; these and others are summarized below with references to attempts to overcome them.

1. Exploration versus exploitation.
    - use random actions
    - favor states not visited recently
    - separate the learning phase from the use phase
    - employ a teacher to guide exploration
2. Slow time to convergence
    - combine learning with prior knowledge; use estimates of Q values (rather than random values) initially
    - use a hierarchy of actions; learn primitive actions first and freeze the useful sequences into macros and then learn how to use the macros
    - employ a teacher; use graded "lessons"—starting near the rewards and then backing away, and use examples of good behavior [Lin, 1992]
    - use more efficient computations; *e.g.* do several updates per episode [Moore & Atkeson, 1993]
3. Large state spaces
    - use hand-coded features
    - use neural networks
    - use nearest-neighbor methods [Moore, 1990]
4. Temporal discounting problems. Using small γ can make the learner too greedy for present rewards and indifferent to the future; but using large γ slows down learning.

- use a learning method based on average rewards [Schwartz, 1993]

5. No "transfer" of learning . What is learned depends on the reward structure; if the rewards change, learning has to start over.

    - Separate the learning into two parts: learn an "action model" which predicts how actions change states (and is constant over all problems), and then learn the "values" of states by reinforcement learning for each different set of rewards. Sometimes the reinforcement learning part can be replaced by a "planner" that uses the action model to produce plans to achieve goals.

Also see other articles in the special issue on reinforcement learning: *Machine Learning*, 8, May, 1992.

11.6 Bibliographical and Historical Remarks

To be added.