20 REINFORCEMENT LEARNING

In which we examine how an agent can learn from success and failure, reward and punishment.

20.1 INTRODUCTION

In the previous two chapters, we have studied learning methods that learn from examples. That is, the environment provides input/output pairs, and the task is to learn a function that could have generated those pairs. These supervised learning methods are appropriate when a teacher is providing correct values or when the function's output represents a prediction about the future that can be checked by looking at the percepts in the next time step. In this chapter, we will study how agents can learn in much less generous environments, where the agent receives no examples, and starts with no model of the environment and no utility function.

For example, we know an agent can learn to play chess by supervised learning—by being given examples of game situations along with the best move for that situation. But if there is no friendly teacher providing examples, what can the agent do? By trying random moves, the agent can eventually build a predictive model of its environment: what the board will be like after it makes a given move, and even how the opponent is likely to reply in a given situation. But without some feedback as to what is good and what is bad, the agent will have no grounds for deciding which move to make. Fortunately, the chess-playing agent does receive some feedback, even without a friendly teacher—at the end of the game, the agent perceives whether it has won or lost. This kind of feedback is called a **reward, or reinforcement.** In games like chess, the reinforcement is received only at the end of the game. We call this a **terminal state** in the state history sequence. In other environments, the rewards come more frequently—in ping-pong, each point scored can be considered a reward. Sometimes rewards are given by a teacher who says "nice move" or "uh-oh" (but does not say what the best move is).

REWARD

TERMINAL STATE

The task of **reinforcement learning** is to use rewards to learn a successful agent function. This is difficult because the agent is never told what the right actions are, nor which rewards are

due to which actions. A game-playing agent may play flawlessly except for one blunder, and at the end of the game get a single reinforcement that says "you lose." The agent must somehow determine which move was the blunder.

Within our framework of agents as functions from percepts to actions, a reward can be provided by a percept, but the agent must be "hardwired" to recognize that percept as a reward rather than as just another sensory input. Thus, animals seem to be hardwired to recognize pain and hunger as negative rewards, and pleasure and food as positive rewards. Training a dog is made easier by the fact that humans and dogs happen to agree that a low-pitched sound (either a growl or a "bad dog!") is a negative reinforcement. Reinforcement has been carefully studied by animal psychologists for over 60 years.

In many complex domains, reinforcement learning is the only feasible way to train a program to perform at high levels. For example, in game playing, it is very hard for a human to provide accurate and consistent evaluations of large numbers of positions, which would be needed to train an evaluation function directly from examples. Instead, the program can be told when it has won or lost, and can use this information to learn an evaluation function that gives reasonably accurate estimates of the probability of winning from any given position. Similarly, it is extremely difficult to program a robot to juggle; yet given appropriate rewards every time a ball is dropped or caught, the robot can learn to juggle by itself.

In a way, reinforcement learning is a restatement of the entire AI problem. An agent in an environment gets percepts, maps some of them to positive or negative utilities, and then has to decide what action to take. To avoid reconsidering all of AI and to get at the principles of reinforcement learning, we need to consider how the learning task can vary:

- The environment can be accessible or inaccessible. In an accessible environment, states can be identified with percepts, whereas in an inaccessible environment, the agent must maintain some internal state to try to keep track of the environment.
- The agent can begin with knowledge of the environment and the effects of its actions; or it will have to learn this model as well as utility information.
- Rewards can be received only in terminal states, or in any state.
- Rewards can be components of the actual utility (points for a ping-pong agent or dollars for a betting agent) that the agent is trying to maximize, or they can be hints as to the actual utility ("nice move" or "bad dog").

PASSIVE LEARNER
ACTIVE LEARNER

- The agent can be a **passive learner** or an **active learner.** A passive learner simply watches the world going by, and tries to learn the utility of being in various states; an active learner must also act using the learned information, and can use its problem generator to suggest explorations of unknown portions of the environment.

Furthermore, as we saw in Chapter 2, there are several different basic designs for agents. Because the agent will be receiving rewards that relate to utilities, there are two basic designs to consider:

- The agent learns a utility function on states (or state histories) and uses it to select actions that maximize the expected utility of their outcomes.

ACTION-VALUE
Q-LEARNING

- The agent learns an **action-value** function giving the expected utility of taking a given action in a given state. This is called **Q-learning**.

An agent that learns utility functions must also have a model of the environment in order to make decisions, because it must know the states to which its actions will lead. For example, in order to make use of a backgammon evaluation function, a backgammon program must know what its legal moves are *and how they affect the board position.* Only in this way can it apply the utility function to the outcome states. An agent that learns an action-value function, on the other hand, need not have such a model. As long as it knows its legal moves, it can compare their values directly without having to consider their outcomes. Action-value learners therefore can be slightly simpler in design than utility learners. On the other hand, because they do not know where their actions lead, they cannot look ahead; this can seriously restrict their ability to learn, as we shall see.

We first address the problem of learning utility functions, which has been studied in AI since the earliest days of the field. (See the discussion of Samuel's checker player in Chapter 5.) We examine increasingly complex versions of the problem, while keeping initially to simple state-based representations. Section 20.6 discusses the learning of action-value functions, and Section 20.7 discusses how the learner can generalize across states. Throughout this chapter, we will assume that the environment is nondeterministic. At this point the reader may wish to review the basics of decision making in complex, nondeterministic environments, as covered in Chapter 17.

20.2 PASSIVE LEARNING IN A KNOWN ENVIRONMENT

To keep things simple, we start with the case of a passive learning agent using a state-based representation in a known, accessible environment. In passive learning, the environment generates state transitions and the agent perceives them.[1] Consider an agent trying to learn the utilities of the states shown in Figure 20.1 (a). We assume, for now, that it is provided with a model My giving the probability of a transition from state i to state$j$, as in Figure 20.1(b). In each **training sequence,** the agent starts in state (1,1) and experiences a sequence of state transitions until it reaches one of the terminal states (3,2) or (3,3), where it receives a reward.[2] A typical set of training sequences might look like this:

TRAINING SEQUENCE

$$(1,1)\rightarrow(1,2)\rightarrow(1,3)\rightarrow(1,2)\rightarrow(1,3)\rightarrow(1,2)\rightarrow(1,1)\rightarrow(1,2)\rightarrow(2,2)\rightarrow(3,2)\ \underline{-1}$$
$$(1,1)\rightarrow(1,2)\rightarrow(1,3)\rightarrow(2,3)\rightarrow(2,2)\rightarrow(2,3)\rightarrow(3,3)\ \underline{+1}$$
$$(1,1)\rightarrow(1,2)\rightarrow(1,1)\rightarrow(1,2)\rightarrow(1,1)\rightarrow(2,1)\rightarrow(2,2)\rightarrow(2,3)\rightarrow(3,3)\ \underline{+1}$$
$$(1,1)\rightarrow(1,2)\rightarrow(2,2)\rightarrow(1,2)\rightarrow(1,3)\rightarrow(2,3)\rightarrow(1,3)\rightarrow(2,3)\rightarrow(3,3)\ \underline{+1}$$
$$(1,1)\rightarrow(2,1)\rightarrow(2,2)\rightarrow(2,1)\rightarrow(1,1)\rightarrow(1,2)\rightarrow(1,3)\rightarrow(2,3)\rightarrow(2,2)\rightarrow(3,2)\ \underline{-1}$$
$$(1,1)\rightarrow(2,1)\rightarrow(1,1)\rightarrow(1,2)\rightarrow(2,2)\rightarrow(3,2)\ \underline{-1}$$

The object is to use the information about rewards to learn the expected utility $U(i)$ associated with each nonterminal state i. We will make one big simplifying assumption: the utility of a sequence is the sum of the rewards accumulated in the states of the sequence. That is, the utility

1 Another way to think of a passive learner is as an agent with a fixed policy trying to determine its benefits.

2 The period from initial state to terminal state is often called an **epoch.**

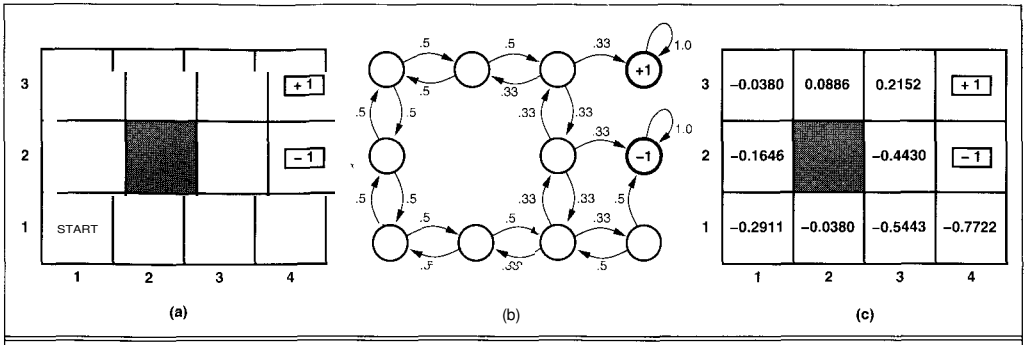


Figure 20.1 (a) A simple stochastic environment. State (1,1) is the start state. (b) Each state transitions to a neighboring state with equal probability among all neighboring states. State (4,2) is terminal with reward −1, and state (4,3) is terminal with reward +1. (c) The exact utility values.

REWARD-TO-GO

function is **additive** in the sense defined on page 502. We define the **reward-to-go** of a state as the sum of the rewards from that state until a terminal state is reached. Given this definition, it is easy to see that *the expected utility of a state is the expected reward-to-go of that state.*

The generic agent design for passive reinforcement learning of utilities is shown in Figure 20.2. The agent keeps an estimate U of the utilities of the states, a table N of counts of how many times each state was seen, and a table M of transition probabilities from state to state. We assume that each percept e is enough to determine the STATE (i.e., the state is accessible), the agent can determine the REWARD component of a percept, and the agent can tell if a percept indicates a TERMINAL? state. In general, an agent can update its current estimated utilities after each observed transition. The key to reinforcement learning lies in the algorithm for updating the utility values given the training sequences. The following subsections discuss three possible approaches to UPDATE.

Naive updating

ADAPTIVE CONTROL THEORY

A simple method for updating utility estimates was invented in the late 1950s in the area of **adaptive control theory** by Widrow and Hoff (1960). We will call it the LMS (least mean squares) approach. In essence, it assumes that for each state in a training sequence, the *observed* reward-to-go on that sequence provides direct evidence of the actual expected reward-to-go. Thus, at the end of each sequence, the algorithm calculates the observed reward-to-go for each state and updates the estimated utility for that state accordingly. It can easily be shown (Exercise 20.1) that the LMS approach generates utility estimates that minimize the mean square error with respect to the observed data. When the utility function is represented by a table of values for each state, the update is simply done by maintaining a running average, as shown in Figure 20.3.

If we think of the utility function as a function, rather than just a table, then it is clear that the LMS approach is simply learning the utility function directly from examples. Each example has the state as input and the observed reward-to-go as output. This means that we have reduced reinforcement learning to a standard inductive learning problem, as discussed in Chapter 18. As

```
function PASSIVE-RL-AGENT(e) returns an action
  static: U, a table of utility estimates
          N, a table of frequencies for states
          M, a table of transition probabilities from state to state
          percepts, a percept sequence (initially empty)

  add e to percepts
  increment N[STATE[e]]
  U ← UPDATE(U, e, percepts, M, N)
  if TERMINAL?[e] then percepts ← the empty sequence
  return the action Observe
```

Figure 20.2 Skeleton for a passive reinforcement learning agent that just observes the world and tries to learn the utilities, U, of each state. The agent also keeps track of transition frequencies and probabilities. The rest of this section is largely devoted to defining the UPDATE function.

```
function LMS-UPDATE(U, e, percepts, M, N) returns an updated U

  if TERMINAL?[e] then reward-to-go ← 0
  for each e_i in percepts (starting at end) do
      reward-to-go ← reward-to-go + REWARD[e_i]
      U[STATE[e_i]] ← RUNNING-AVERAGE(U[STATE[e_i]], reward-to-go, N[STATE[e_i]])
  end
```

Figure 20.3 The update function for least mean square (LMS) updating of utilities.

we will show, it is an easy matter to use more powerful kinds of representations for the utility function, such as neural networks. Learning techniques for those representations can be applied directly to the observed data.

One might think that the LMS approach more or less solves the reinforcement learning problem—or at least, reduces it to one we already know a lot about. In fact, the LMS approach misses a very important aspect of the reinforcement learning problem, namely, the fact that the utilities of states are not independent! The structure of the transitions among the states in fact imposes very strong additional constraints: *The actual utility of a state is constrained to be the probability-weighted average of its successors' utilities, plus its own reward.* By ignoring these constraints, LMS-UPDATE usually ends up converging very slowly on the correct utility values for the problem. Figure 20.4 shows a typical run on the 4 x 3 environment in Figure 20.1, illustrating both the convergence of the utility estimates and the gradual reduction in the root-mean-square error with respect to the *correct* utility values. It takes the agent well over a thousand training sequences to get close to the correct values.

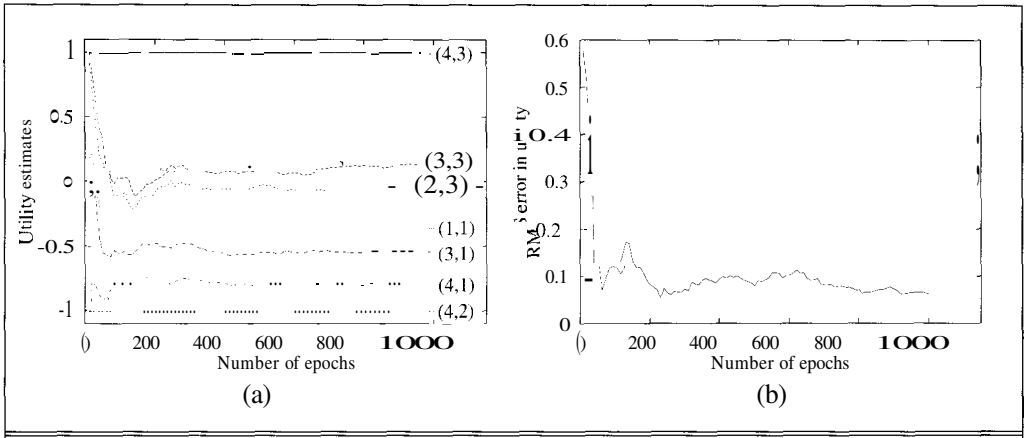


Figure 20.4 The LMS learning curves for the 4×3 world shown in Figure 20.1. (a) The utility estimates of the states over time. (b) The RMS error compared to the correct values.

Adaptive dynamic programming

Programs that use knowledge of the structure of the environment usually learn much faster. In the example in Figure 20.5 (from (Sutton, 1988)), the agent already has a fair amount of experience with the three states on the right, and has learned the values indicated. However, the path followed from the new state reaches an unusually good terminal state. The LMS algorithm will assign a utility estimate of +1 to the new state, whereas it is clear that the new state is much less promising, because it has a transition to a state known to have utility ≈ -0.8, and no other known transitions.

Fortunately, this drawback can be fixed. Consider again the point concerning the constraints on neighboring utilities. Because, for now, we are assuming that the transition probabilities are listed in the known table M_{ij}, the reinforcement learning problem becomes a well-defined sequential decision problem (see Chapter 17) as soon as the agent has observed the rewards for all the states. In our 4×3 environment, this usually happens after a handful of training sequences, at which point the agent can compute the exact utility values for all states. The utilities are computed by solving the set of equations

$$U(i) = R(i) + \sum_j M_{ij} U(j) \tag{20.1}$$

where $R(i)$ is the reward associated with being in state i, and M_{ij} is the probability that a transition will occur from state i to state j. This set of equations simply formalizes the basic point made in the previous subsection. Notice that because the agent is passive, no maximization over actions is involved (unlike Equation (17.3)). The process of solving the equations is therefore identical to a single **value determination** phase in the policy iteration algorithm. The exact utilities for the states in the 4×3 world are shown in Figure 20.1(c). Notice how the "safest" squares are those along the top row, away from the negative reward state.

ADAPTIVE DYNAMIC PROGRAMMING
ADP

We will use the term **adaptive dynamic programming (or ADP)** to denote any reinforcement learning method that works by solving the utility equations with a dynamic programming algorithm. In terms of its ability to make good use of experience, ADP provides a standard

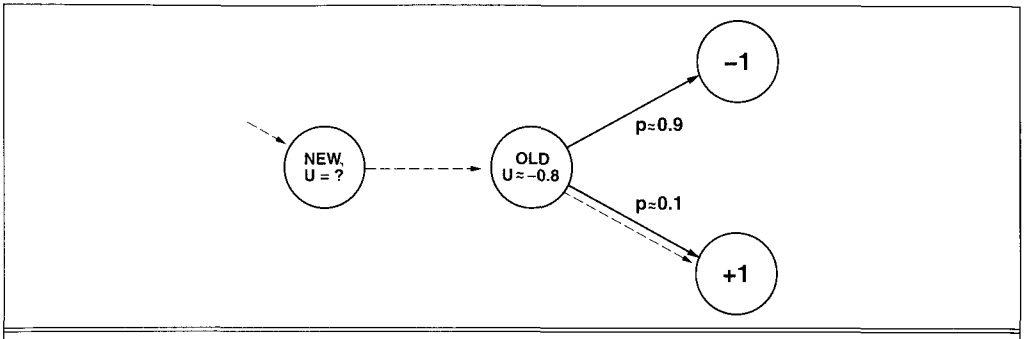


Figure 20.5 An example where LMS does poorly. A new state is reached for the first time, and then follows the path marked by the dashed lines, reaching a terminal state with reward +1.

against which to measure other reinforcement learning algorithms. It is, however, somewhat intractable for large state spaces. In backgammon, for example, it would involve solving roughly 10^{50} equations in 10^{50} unknowns.

Temporal difference learning

It is possible to have (almost) the best of both worlds—that is, one can approximate the constraint equations shown earlier without solving them for all possible states. *The key is to use the observed transitions to adjust the values of the observed states so that they agree with the constraint equations.* Suppose that we observe a transition from state i to state j, where currently $U(i) = -0.5$ and $U(j) = +0.5$. This suggests that we should consider increasing $U(i)$ to make it agree better with its successor. This can be achieved using the following updating rule:

$$U(i) \leftarrow U(i) + \alpha(R(i) + U(j) - U(i)) \tag{20.2}$$

TEMPORAL-DIFFERENCE

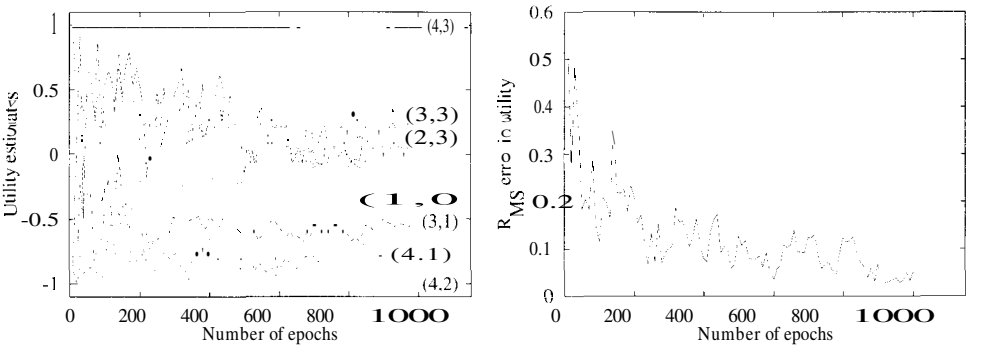
where α is the **learning rate** parameter. Because this update rule uses the difference in utilities between successive states, it is often called the **temporal-difference,** or **TD,** equation.

The basic idea of all temporal-difference methods is to first define the conditions that hold locally when the utility estimates are correct; and then to write an update equation that moves the estimates toward this ideal "equilibrium" equation. In the case of passive learning, the equilibrium is given by Equation (20.1). Now Equation (20.2) does in fact cause the agent to reach the equilibrium given by Equation (20.1), but there is some subtlety involved. First, notice that the update only involves the actual successor, whereas the actual equilibrium conditions involve all possible next states. One might think that this causes an improperly large change in $U(i)$ when a very rare transition occurs; but, in fact, because rare transitions occur only rarely, the *average value* of $U(i)$ will converge to the correct value. Furthermore, if we change α from a fixed parameter to a function that decreases as the number of times a state has been visited increases, then $U(i)$ itself will converge to the correct value (Dayan, 1992). This gives us the algorithm TD-UPDATE, shown in Figure 20.6. Figure 20.7 shows a typical run of the TD learning algorithm on the world in Figure 20.1. Although TD generates noisier values, the RMS error is actually significantly less than that for LMS after 1000 iterations.

```
function TD-UPDATE(U, e, percepts, M, N) returns the utility table U

  if TERMINAL?[e] then
      U[STATE[e]] ← RUNNING-AVERAGE(U[STATE[e]], REWARD[e], N[STATE[e]])
  else if percepts contains more than one element then
      e' ← the penultimate element of percepts
      i, j ← STATE[e'], STATE[e]
      U[i] ← U[i] + α(N[i])(REWARD[e'] + U[j] - U[i])
```

Figure 20.6 An algorithm for updating utility estimates using temporal differences.



(a) (b)

Figure 20.7 The TD learning curves for the 4 x 3 world. (a) The utility estimates of the states over time. (b) The RMS error compared to the correct values.

20.3 PASSIVE LEARNING IN AN UNKNOWN ENVIRONMENT

The previous section dealt with the case in which the environment model M is already known. Notice that of the three approaches, only the dynamic programming method used the model in full. TD uses information about connectedness of states, but only from the current training sequence. (As we mentioned before, all utility-learning methods will use the model during subsequent action selection.) Hence LMS and TD will operate unchanged in an initially unknown environment.

The adaptive dynamic programming approach simply adds a step to PASSIVE-RL-AGENT that updates an estimated model of the environment. Then the estimated model is used as the basis for a dynamic programming phase to calculate the corresponding utility estimates after each observation. As the environment model approaches the correct model, the utility estimates will,

of course, converge to the correct utilities. Because the environment model usually changes only slightly with each observation, the dynamic programming phase can use value iteration with the previous utility estimates as initial values and usually converges quite quickly.

The environment model is learned by direct observation of transitions. In an accessible environment, each percept identifies the state, and hence each transition provides a direct input/output example for the transition function represented by M. The transition function is usually stochastic—that is, it specifies a probability for each possible successor rather than a single state. A reinforcement learning agent can use any of the techniques for learning stochastic functions from examples discussed in Chapters 18 and 19. We discuss their application further in Section 20.7.

Continuing with our tabular representation for the environment, we can update the environment model M simply by keeping track of the percentage of times each state transitions to each of its neighbors. Using this simple technique in the 4 x 3 world from Figure 20.1, we obtain the learning performance shown in Figure 20.8. Notice that the ADP method converges far faster than either LMS or TD learning.

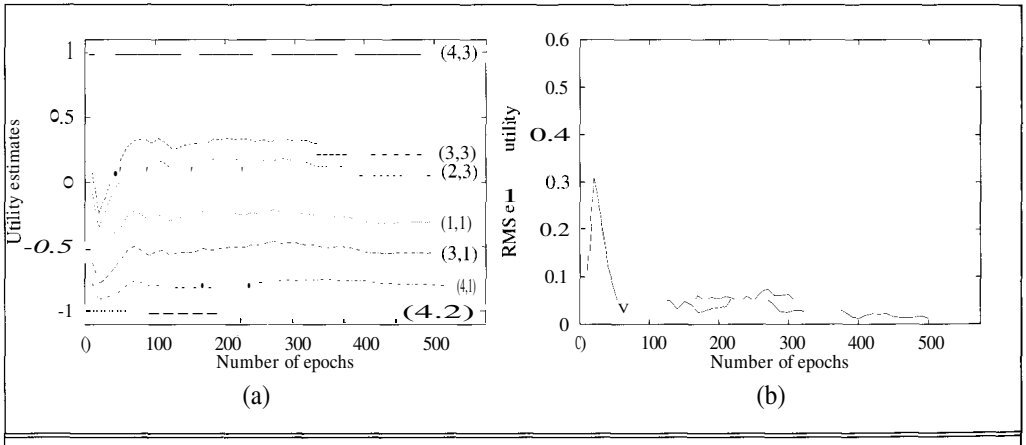


Figure 20.8 The ADP learning curves for the 4 × 3 world.

The ADP approach and the TD approach are actually closely related. Both try to make local adjustments to the utility estimates in order to make each state "agree" with its successors. One minor difference is that TD adjusts a state to agree with its *observed* successor (Equation (20.2)), whereas ADP adjusts the state to agree with *all* of the successors that might occur given an optimal action choice, weighted by their probabilities (Equation (20.1)). This difference disappears when the effects of TD adjustments are averaged over a large number of transitions, because the frequency of each successor in the set of transitions is approximately proportional to its probability. A more important difference is that whereas TD makes a single adjustment per observed transition, ADP makes as many as it needs to restore consistency between the utility estimates U and the environment model M. Although the observed transition only makes a local change in M, its effects may need to be propagated throughout U. Thus, TD can be viewed as a crude but efficient first approximation to ADP.

Each adjustment made by ADP could be viewed, from the TD point of view, as a result of a "pseudo-experience" generated by simulating the current environment model. It is possible to extend the TD approach to use an environment model to generate several pseudo-experiences—transitions that the TD agent can imagine *might* happen given its current model. For each observed transition, the TD agent can generate a large number of imaginary transitions. In this way, the resulting utility estimates will approximate more and more closely those of ADP—of course, at the expense of increased computation time.

In a similar vein, we can generate more efficient versions of ADP by directly approximating the algorithms for value iteration or policy iteration. Recall that full value iteration can be intractable when the number of states is large. Many of the adjustment steps, however, are extremely tiny. One possible approach to generating reasonably good answers quickly is to bound the number of adjustments made after each observed transition. One can also use a heuristic to rank the possible adjustments so as to carry out only the most significant ones. The

PRIORITIZED-SWEEPING

prioritized-sweeping heuristic prefers to make adjustments to states whose *likely* successors have just undergone a *large* adjustment in their own utility estimates. Using heuristics like this, approximate ADP algorithms usually can learn roughly as fast as full ADP, in terms of the number of training sequences, but can be several orders of magnitude more efficient in terms of computation (see Exercise 20.3). This enables them to handle state spaces that are far too large for full ADP. Approximate ADP algorithms have an additional advantage: in the early stages of learning a new environment, the environment model M often will be far from correct, so there is little point in calculating an exact utility function to match it. An approximation algorithm can use a minimum adjustment size that decreases as the environment model becomes more accurate. This eliminates the very long value iterations that can occur early in learning due to large changes in the model.

20.4 ACTIVE LEARNING IN AN UNKNOWN ENVIRONMENT

A passive learning agent can be viewed as having a fixed policy, and need not worry about which actions to take. An active agent must consider what actions to take, what their outcomes may be, and how they will affect the rewards received.

The PASSIVE-RL-AGENT model of page 602 needs only minor changes to accommodate actions by the agent:

- The environment model must now incorporate the probabilities of transitions to other states *given a particular action.* We will use M_{ij}^{a} to denote the probability of reaching state j if the action a is taken in state i.
- The constraints on the utility of each state must now take into account the fact that the agent has a choice of actions. A rational agent will maximize its expected utility, and instead of Equation (20.1) we use Equation (17.3), which we repeat here:

$$U(i) = R(i) + \max_{a} \sum_{j} M_{ij}^{a} U(j) \tag{20.3}$$

- The agent must now choose an action at each step, and will need a performance element to do so. In the algorithm, this means calling PERFORMANCE-ELEMENT(e) and returning the resulting action. We assume that the model M and the utilities U are shared by the performance element; that is the whole point of learning them.

We now reexamine the dynamic programming and temporal-difference approaches in the light of the first two changes. The question of how the agent should act is covered in Section 20.5.

Because the ADP approach uses the environment model, we will need to change the algorithm for learning the model. Instead of learning the probability M_{ij} of a transition, we will need to learn the probability M_{ij}^{a} of a transition conditioned on taking an action a. In the explicit tabular representation for M, this simply means accumulating statistics in a three-dimensional table. With an implicit functional representation, as we will soon see, the input to the function will include the action taken. We will assume that a procedure UPDATE-ACTIVE-MODEL takes care of this. Once the model has been updated, then the utility function can be recalculated using a dynamic programming algorithm and then the performance element chooses what to do next. We show the overall design for ACTIVE-ADP-AGENT in Figure 20.9.

An active temporal-difference learning agent that learns utility functions also will need to learn a model in order to use its utility function to make decisions. The model acquisition problem for the TD agent is identical to that for the ADP agent. What of the TD update rule itself? Perhaps surprisingly, the update rule (20.2) remains unchanged. This might seem odd, for the following reason. Suppose the agent takes a step that normally leads to a good destination, but because of nondeterminism in the environment the agent ends up in a catastrophic state. The TD update rule will take this as seriously as if the outcome had been the normal result of the

```
function ACTIVE-ADP-AGENT(e) returns an action
  static: U, a table of utility estimates
          M, a table of transition probabilities from state to state for each action
          R, a table of rewards for states
          percepts, a percept sequence (initially empty)
          last-action, the action just executed

  add e to percepts
  R[STATE[e]] — REWARD[e]
  M — UPDATE-ACTIVE-MODEL(M, percepts, last-action)
  U — VALUE-ITERATION(U, M, R)
  if TERMINAL?[e] then
     percepts — the empty sequence
  last-action ← PERFORMANCE-ELEMENT(e)
  return last-action
```

Figure 20.9 Design for an active ADP agent. The agent learns an environment model M by observing the results of its actions, and uses the model to calculate the utility function U using a dynamic programming algorithm (here POLICY-ITERATION could be substituted for VALUE-ITERATION).

action, whereas one might suppose that because the outcome was a fluke, the agent should not worry about it too much. In fact, of course, the unlikely outcome will only occur infrequently in a large set of training sequences; hence in the long run its effects will be weighted proportionally to its probability, as we would hope. Once again, it can be shown that the TD algorithm will converge to the same values as ADP as the number of training sequences tends to infinity.

20.5 EXPLORATION

The only remaining issue to address for active reinforcement learning is the question of what actions the agent should take—that is, what PERFORMANCE-ELEMENT should return. This turns out to be harder than one might imagine.

One might suppose that the correct way for the agent to behave is to choose whichever action has the highest expected utility given the current utility estimates—after all, that is all the agent has to go on. But this overlooks the contribution of action to learning. In essence, an action has two kinds of outcome:[3]

- It gains rewards on the current sequence.
- It affects the percepts received, and hence the ability of the agent to learn—and receive rewards in future sequences.

An agent therefore must make a trade-off between its immediate good—as reflected in its current utility estimates—and its long-term well-being. An agent that simply chooses to maximize its rewards on the current sequence can easily get stuck in a rut. At the other extreme, continually acting to improve one's knowledge is of no use if one never puts that knowledge into practice. In the real world, one constantly has to decide between continuing in a comfortable existence and striking out into the unknown in the hopes of discovering a new and better life.

In order to illustrate the dangers of the two extremes, we will need a suitable environment. We will use the stochastic version of the 4×3 world shown in Figure 17.1. In this world, the agent can attempt to move *North, South, East,* or *West*; each action achieves the intended effect with probability 0.8, but the rest of the time, the action moves the agent at right angles to the intended direction. As before, we assume a reward of -0.04 (i.e., a cost of 0.04) for each action that doesn't reach a terminal state. The optimal policy and utility values for this world are shown in Figure 17.2, and the object of the learning agent is to converge towards these.

Let us consider two possible approaches that the learning agent might use for choosing what to do. The "wacky" approach acts randomly, in the hope that it will eventually explore the entire environment; and the "greedy" approach acts to maximize its utility using current estimates. As we see from Figure 20.10, the wacky approach succeeds in learning good utility estimates for all the states (top left). Unfortunately, its wacky policy means that it never actually gets better at reaching the positive reward (top right). The greedy agent, on the other hand, often finds a path to the +1 reward along the lower route via (2,1), (3,1), (3,2), and (3,3). Unfortunately, it then sticks to that path, never learning the utilities of the other states (bottom left). This means

[3] Notice the direct analogy to the theory of information value in Chapter 16.

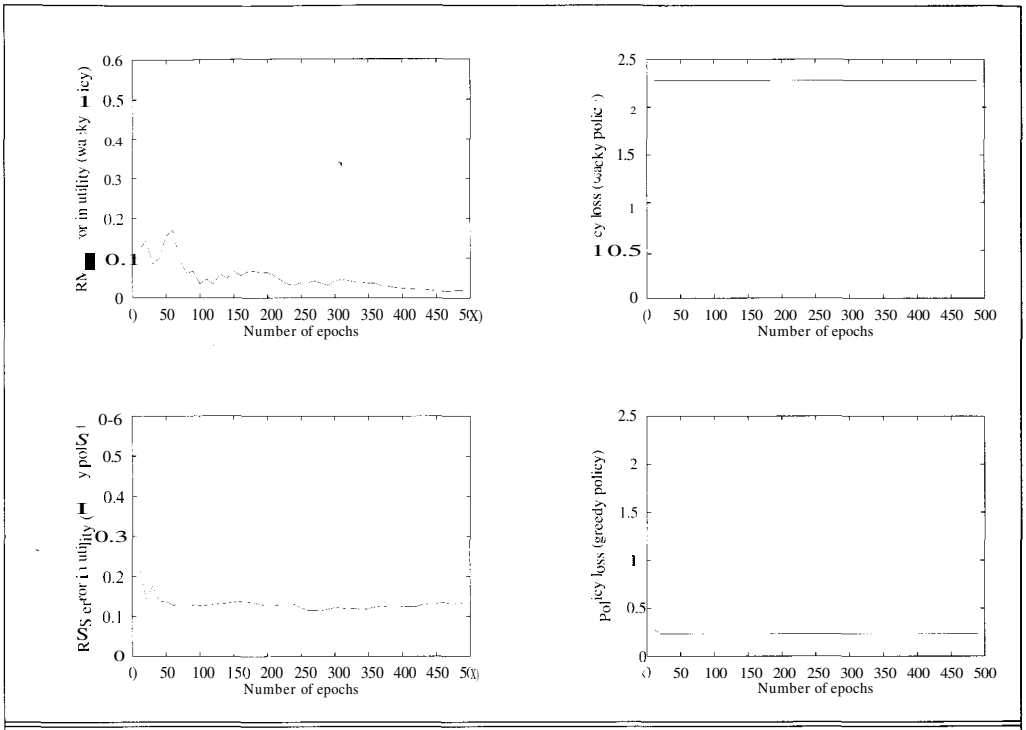


Figure 20.10 Curves showing the RMS error in utility estimates (left) and the total loss associated with the corresponding policy (right), for the wacky (top) and greedy (bottom) approaches to exploration.

that it too fails to achieve perfection (bottom right) because it does not find the optimal route via (1,2), (1,3), and (2,3).

Obviously, we need an approach somewhere between wackiness and greediness. The agent should be more wacky when it has little idea of the environment, and more greedy when it has a model that is close to being correct. Can we be a little more precise than this? Is there an *optimal* exploration policy? It turns out that this question has been studied in depth in the subfield of statistical decision theory that deals with so-called **bandit problems** (see sidebar).

BANDIT PROBLEMS

Although bandit problems are extremely difficult to solve exactly to obtain an *optimal* exploration policy, it is nonetheless possible to come up with a *reasonable* policy that seems to have the desired properties. In a given state, the agent should give some weight to actions that it has not tried very often, while being inclined to avoid actions that are believed to be of low utility. This can be implemented by altering the constraint equation (20.3) so that it assigns a higher utility estimate to relatively unexplored action-state pairs. Essentially, this amounts to an optimistic prior over the possible environments, and causes the agent to behave initially as if there were wonderful rewards scattered all over the place. Let us use $U^+(i)$ to denote the optimistic estimate of the utility (i.e., the expected reward-to-go) of the state i, and let $N(a,i)$ be the number of times action a has been tried in state i. Suppose we are using value iteration

EXPLORATION AND BANDITS

In Las Vegas, a *one-armed bandit* is a slot machine. A gambler can insert a coin, pull the lever, and collect the winnings (if any). An ***n*-armed bandit** has n levers. The gambler must choose which lever to play on each successive coin—the one that has paid off best, or maybe one that has not been tried?

The n-armed bandit problem is a formal model for real problems in many vitally important areas, such as deciding on the annual budget for AI research and development. Each arm corresponds to an action (such as allocating $20 million for development of new AI textbooks) and the payoff from pulling the arm corresponds to the benefits obtained from taking the action (immense). Exploration, whether it is exploration of a new research field or exploration of a new shopping mall, is risky, expensive, and has uncertain payoffs; on the other hand, failure to explore at all means that one never discovers *any* actions that are worthwhile.

To formulate a bandit problem properly, one must define exactly what is meant by optima] behavior. Most definitions in the literature assume that the aim is to maximize the expected total reward obtained over the agent's lifetime. These definitions require that the expectation be taken over the possible worlds that the agent could be in, as well as over the possible results of each action sequence in any given world. Here, a "world" is defined by the transition model M_{ij}^a. Thus, in order to act optimally, the agent needs a prior distribution over the possible models. The resulting optimization problems are usually wildly intractable. In some cases, however, appropriate independence assumptions enable the problem to be solved in closed form. With a row of real slot machines, for example, the rewards in successive time steps and on different machines can be assumed to be independent. It turns out that the fraction of one's coins invested in a given machine should drop off proportionally to the probability that the machine is in fact the best, given the observed distributions of rewards.

The formal results that have been obtained for optimal exploration policies apply only to the case in which the agent represents the transition model as an explicit table and is not able to generalize across states and actions. For more realistic problems, it is possible to prove only convergence to a correct model and optimal behavior in the limit of infinite experience. This is easily obtained by acting randomly on some fraction of steps, where that fraction decreases appropriately over time.

One can use the theory of n-armed bandits to argue for the reasonableness of the selection strategy in genetic algorithms (see Section 20.8). If you consider each arm in an n-armed bandit problem to be a possible string of genes, and the investment of a coin in one arm to be the reproduction of those genes, then genetic algorithms allocate coins optimally, given an appropriate set of independence assumptions.

in an ADP learning agent; then we need to rewrite the update equation (i.e., Equation (17.4)) to incorporate the optimistic estimate. The following equation does this:

$$U^+(i) \leftarrow R(i) + \max_a f\left(\sum_j M_{ij}^a U^+ j),\ N(a,i)\right) \tag{20.4}$$

EXPLORATION FUNCTION

where $f(u,n)$ is called the **exploration function.** It determines how greed (preference for high values of u) is traded off against curiosity (preference for low values of n, i.e., actions that have not been tried often). The function $f(u,n)$ should be increasing in u, and decreasing in n. Obviously, there are many possible functions that fit these conditions. One particularly simple definition is the following:

$$f(u,n) = \begin{cases} R^+ & \text{if } n < N_e \\ u & \text{otherwise} \end{cases}$$

where R^+ is an optimistic estimate of the best possible reward obtainable in any state, and N_e is a fixed parameter. This will have the effect of making the agent try each action-state pair at least N_e times.

The fact that U^+ rather than U appears on the right-hand side of Equation (20.4) is very important. As exploration proceeds, the states and actions near the start state may well be tried a large number of times. If we used U, the nonoptimistic utility estimate, then the agent would soon become disinclined to explore further afield. The use of U^+ means that the benefits of exploration are propagated back from the edges of unexplored regions, so that actions that lead *toward* unexplored regions are weighted more highly, rather than just actions that are themselves unfamiliar. The effect of this exploration policy can be seen clearly in Figure 20.11, which shows a rapid convergence toward optimal performance, unlike that of the wacky or the greedy approaches. A very nearly optimal policy is found after just 18 trials. Notice that the utility estimates themselves do not converge as quickly. This is because the agent stops exploring the unrewarding parts of the state space fairly soon, visiting them only "by accident" thereafter. However, it makes perfect sense for the agent not to care about the exact utilities of states that it knows are undesirable and can be avoided.

20.6 LEARNING AN ACTION-VALUE FUNCTION

An action-value function assigns an expected utility to taking a given action in a given state; as mentioned earlier, such values are also called **Q-values**. We will use the notation $Q(a,i)$ to denote the value of doing action a in state i. Q-values are directly related to utility values by the following equation:

$$U(i) = \max_a Q(a,i) \tag{20.5}$$

Q-values play an important role in reinforcement learning for two reasons: first, like condition-action rules, they suffice for decision making without the use of a model; second, unlike condition-action rules, they can be learned directly from reward feedback.

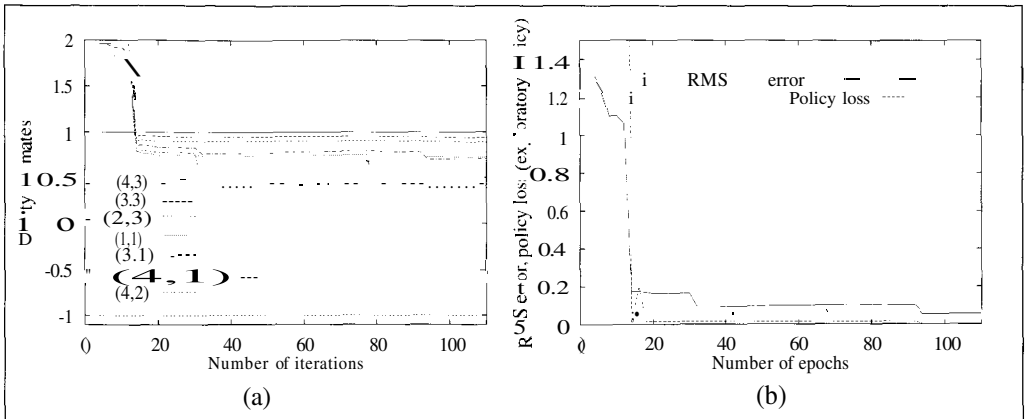


Figure 20.11 Performance of the exploratory ADP agent. using $R^+ = 2$ and $N_e = 5$. (a) Utility estimates for selected states over time. After the initial exploratory phase in which the states get an exploration bonus, the high-valued states quickly reach their correct values. The low-valued states converge slowly because they are seldom visited. (b) The RMS error in utility values and the associated policy loss.

As with utilities, we can write a constraint equation that must hold at equilibrium when the Q-values are correct:

$$Q(a,i) = R(i) + \sum_j M_{ij}^a \max_{a'} Q(a',j) \tag{20.6}$$

As in the ADP learning agent, we can use this equation directly as an update equation for an iteration process that calculates exact Q-values given an estimated model. This does, however, require that a model be learned as well because the equation uses M_{ij}^a. The temporal-difference approach, on the other hand, requires no model. The update equation for TD Q-learning is

$$Q(a,i) \leftarrow Q(a,i) + \alpha(R(i) + \max_{a'} Q(a',j) - Q(a,i)) \tag{20.7}$$

which is calculated after each transition from state i to state j.

The complete agent design for an exploratory Q-learning agent using TD is shown in Figure 20.12. Notice that it uses exactly the same exploration function f as used by the exploratory ADP agent, hence the need to keep statistics on actions taken (the table N). If a simpler exploration policy is used—say, acting randomly on some fraction of steps, where the fraction decreases over time—then we can dispense with the statistics.

Figure 20.13 shows the performance of the Q-learning agent in our 4×3 world. Notice that the utility estimates (derived from the Q-values using Equation (20.5)) take much longer to settle down than they did with the ADP agent. This is because TD does not enforce consistency among values via the model. Although a good policy is found after only 26 trials, it is considerably further from optimality than that found by the ADP agent (Figure 20.11).

Although these experimental results are for just one set of trials on one specific environment, they do raise a general question: is it better to learn a model and a utility function or to learn an action-value function with no model? In other words, what is the best way to represent the

```
function Q-LEARNING-AGENT(e) returns an action
  static: Q, a table of action values
          N, a table of state-action frequencies
          a, the last action taken
          (', the previous state visited
          r, the reward received in state i

  j ← STATE[e]
  if (' is non-null then
      N[a, i] — N[a, i] + 1
      Q[a, i] — Q[a, i] + α(r + max_a'  Q[a', j]  -  Q[a, i])
  if TERMINAL?[e] then
      i — null
  else
      i — j
      r — REWARD[e]
  a — arg max_a' f(Q[a', j], N[a', j])
  return a
```

Figure 20.12 An exploratory Q-learning agent. It is an active learner that learns the value $Q(a, i)$ of each action in each situation. It uses the same exploration function f as the exploratory ADP agent, but avoids having to learn the transition model M_{ij}^{a} because the Q-value of s state can be related directly to those of its neighbors.

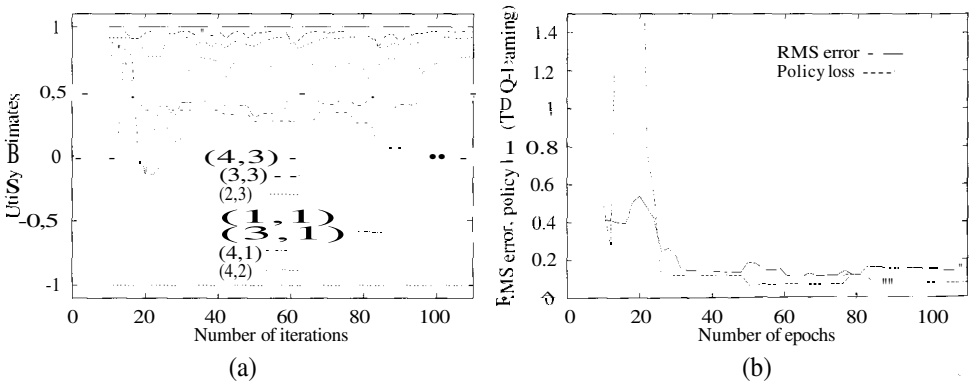


Figure 20.13 Performance of the exploratory TD Q-learning agent. using $R^{+} = 2$ and $N_e = 5$. (a) Utility estimates for selected states over time. (b) The RMS error in utility values and the associated policy loss.

agent function? This is an issue at the foundations of artificial intelligence. As we stated in Chapter 1, one of the key historical characteristics of much of AI research is its (often unstated) adherence to the **knowledge-based** approach. This amounts to an assumption that the best way to represent the agent function is to construct an explicit representation of at least some aspects of the environment in which the agent is situated.

Some researchers, both inside and outside AI, have claimed that the availability of model-free methods such as Q-learning means that the knowledge-based approach is unnecessary. There is, however, little to go on but intuition. Our intuition, for what it's worth, is that as the environment becomes more complex, the advantages of a knowledge-based approach become more apparent. This is borne out even in games such as chess, checkers (draughts), and backgammon (see next section), where efforts to learn an evaluation function using a model have met with more success than Q-learning methods. Perhaps one day there will be a deeper theoretical understanding of the advantages of explicit knowledge; but as yet we do not even have a formal definition of the difference between model-based and model-free systems. All we have are some purported examples of each.

20.7 GENERALIZATION IN REINFORCEMENT LEARNING

EXPLICIT REPRESENTATION

So far we have assumed that all the functions learned by the agents (U, M, R, Q) are represented in tabular form —that is, an **explicit representation** of one output value for each input tuple. Such an approach works reasonably well for small state spaces, but the time to convergence and (for ADP) the time per iteration increase rapidly as the space gets larger. With carefully controlled, approximate ADP methods, it may be possible to handle 10,000 states or more. This suffices for two-dimensional, maze-like environments, but more realistic worlds are out of the question. Chess and backgammon are tiny subsets of the real world, yet their state spaces contain on the order of 10^{50} to 10^{120} states. It would be absurd to suppose that one must visit all these states in order to learn how to play the game!

IMPLICIT REPRESENTATION

The only way to handle such problems is to use an **implicit representation** of the function—a form that allows one to calculate the output for any input, but that is usually much more compact than the tabular form. For example, an estimated utility function for game playing can be represented as a weighted linear function of a set of board features $f_1, \ldots, f_n$:

$$U(i) = w_1 f_1(i) + w_2 f_2(i) + \cdots + w_n f_n(i)$$

Thus, instead of, say, 10^{120} values, the utility function is characterized by the n weights. A typical chess evaluation function might only have about 10 weights, so this is an *enormous* compression. *The compression achieved by an implicit representation allows the learning agent to generalize from states it has visited to states it has not visited.* That is, the most important aspect of an implicit representation is not that it takes up less space, but that it allows for inductive generalization over input states. For this reason, methods that learn such representations are said to perform **input generalization.** To give you some idea of the power of input generalization: by examining only one in 10^{44} of the possible backgammon states, it is possible to learn a utility function that allows a program to play as well as any human (Tesauro, 1992).

INPUT GENERALIZATION

On the flip side, of course, there is the problem that there may be no function in the chosen space of implicit representations that faithfully approximates the true utility function. As in all inductive learning, there is a trade-off between the size of the hypothesis space and the time it takes to learn the function. A larger hypothesis space increases the likelihood that a good approximation can be found, but also means that convergence is likely to be delayed.

Let us now consider exactly how the inductive learning problem should be formulated. We begin by considering how to learn utility and action-value functions, and then move on to learning the transition function for the environment.

In the LMS (least mean squares) approach, the formulation is straightforward. At the end of each training sequence, the LMS algorithm associates a reward-to-go with each state visited along the way. The $\langle state, reward \rangle$ pair can be used directly as a labelled example for any desired inductive learning algorithm. This yields a utility function $U(i)$.

It is also possible for a TD (temporal-difference) approach to apply inductive learning directly, once the U and/or Q tables have been replaced by implicit representations. The values that would be inserted into the tables by the update rules (20.2 and 20.7) can be used instead as labelled examples for a learning algorithm. The agent has to use the learned function on the next update, so the learning algorithm must be incremental.

One can also take advantage of the fact that the TD update rules provide small changes in the value of a given state. This is especially true if the function to be learned is characterized by a vector of weights w (as in linear weighted functions and neural networks). Rather than update a single tabulated value of U, as in Equation (20.2), we simply adjust the weights to try to reduce the temporal difference in U between successive states. Suppose that the parameterized utility function is $U_{\mathbf{w}}(i)$. Then after a transition $i \rightarrow j$, we apply the following update rule:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[r + U_{\mathbf{w}}(j) - U_{\mathbf{w}}(i)] \nabla_{\mathbf{w}} U_{\mathbf{w}}(i) \tag{20.8}$$

This form of updating performs gradient descent in weight space, trying to minimize the observed local error in the utility estimates. A similar update rule can be used for Q-learning (Exercise 20.9). Because the utility and action-value functions have real-valued outputs, neural networks and other continuous function representations are obvious candidates for the performance element. Decision-tree learning algorithms that provide real-valued output can also be used (see for example Quinlan's (1993) **model trees),** but cannot use the gradient descent method.

MODEL TREES

The formulation of the inductive learning problem for constructing a model of the environment is also very straightforward. Each transition provides the agent with the next state (at least in an accessible environment), so that labelled examples consist of a state-action pair as input and a state as output. It is not so easy, however, to find a suitable implicit representation for the model. In order to be useful for value and policy iteration and for the generation of pseudo-experiences in TD learning, the output state description must be sufficiently detailed to allow prediction of outcomes several steps ahead. Simple parametric forms cannot usually sustain this kind of reasoning. Instead, it may be necessary to learn general action models in the logical form used in Chapters 7 and 11. In a nondeterministic environment, one can use the conditional-probability-table representation of state evolution typical of dynamic belief networks (Section 17.5), in which generalization is achieved by describing the state in terms of a large set of features and using only sparse connections. Although model-based approaches have advantages in terms of their ability to learn value functions quickly, they are currently hampered by a lack

of suitable inductive generalization methods for learning the model. It is also not obvious how methods such as value and policy iteration can be applied with a generalized model.

We now turn to examples of large-scale applications of reinforcement learning. We will see that in cases where a utility function (and hence a model) is used, the model is usually taken as given. For example, in learning an< evaluation function for backgammon, it is normally assumed that the legal moves, and their effects, are known in advance.

Applications to game-playing

The first significant application of reinforcement learning was also the first significant learning program of any kind—the checker-playing program written by Arthur Samuel (1959; 1967). Samuel first used a weighted linear function for the evaluation of positions, using up to 16 terms at any one time. He applied a version of Equation (20.8) to update the weights. There were some significant differences, however, between his program and current methods. First, he updated the weights using the difference between the current state and the backed-up value generated by full lookahead in the search tree. This works fine, because it amounts to viewing the state space at a different granularity. A second difference was that the program did *not* use any observed rewards! That is, the values of terminal states were ignored. This means that it is quite possible for Samuel's program not to converge, or to converge on a strategy designed to lose rather than win. He managed to avoid this fate by insisting that the weight for material advantage should always be positive. Remarkably, this was sufficient to direct the program into areas of weight space corresponding to good checker play (see Chapter 5).

The TD-gammon system (Tesauro, 1992) forcefully illustrates the potential of reinforcement learning techniques. In earlier work (Tesauro and Sejnowski, 1989), Tesauro tried learning a neural network representation of $Q(a, i)$ directly from examples of moves labelled with relative values by a human expert. This approach proved extremely tedious for the expert. It resulted in a program, called Neurogammon, that was strong by computer standards but not competitive with human grandmasters. The TD-gammon project was an attempt to learn from self-play alone. The only reward signal was given at the end of each game. The evaluation function was represented by a fully connected neural network with a single hidden layer containing 40 nodes. Simply by repeated application of Equation (20.8), TD-gammon learned to play considerably better than Neurogammon, even though the input representation contained just the raw board position with no computed features. This took about 200,000 training games and two weeks of computer time. Although this may seem like a lot of games, it is only a vanishingly small fraction of the state space. When precomputed features were added to the input representation, a network with 80 hidden units was able, after 300,000 training games, to reach a standard of play comparable with the top three human players worldwide.

Application to robot control

CART-POLE
INVERTED PENDULUM

The setup for the famous **cart-pole** balancing problem, also known as the **inverted pendulum,** is shown in Figure 20.14. The problem is to control the position x of the cart so that the pole stays roughly upright ($\theta \approx \pi/2$), while staying within the limits of the cart track as shown. This problem has been used as a test bed for research in control theory as well as reinforcement

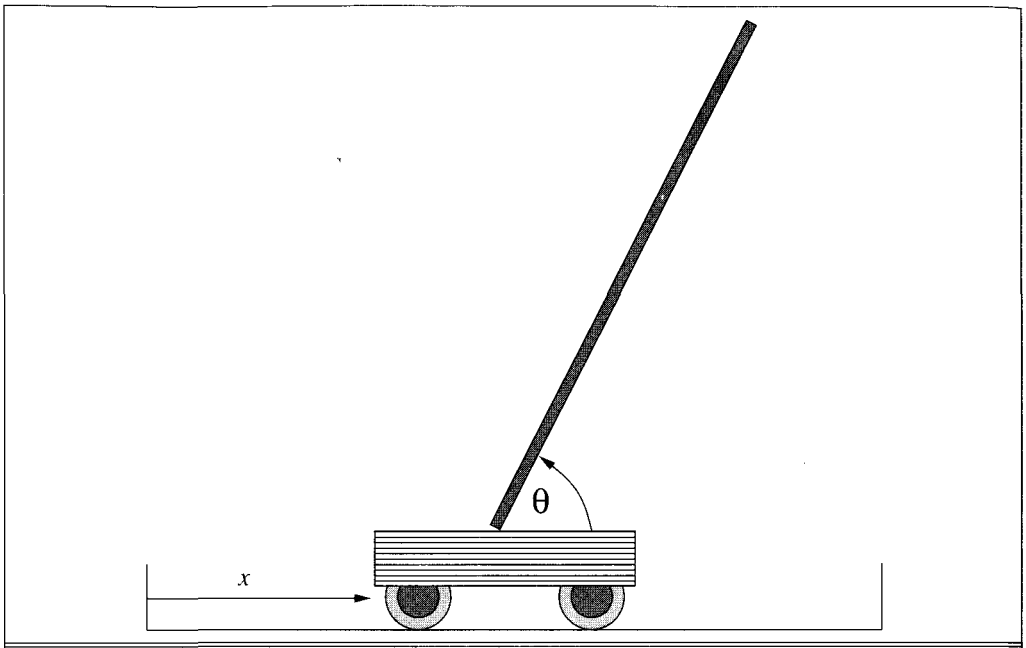


Figure 20.14 Setup for the problem of balancing a long pole on top of a moving cart. The cart can be jerked left or right by a controller that observes x, θ, $\dot{x}$, and $\dot{\theta}$.

BANG-BANG CONTROL

learning, and over 200 papers have been published on it. The cart-pole problem differs from the problems described earlier in that the state variables x, $\dot{x}$, θ, and $\dot{\theta}$ are continuous. The actions are usually discrete—jerk left or jerk right, the so-called **bang-bang control** regime.

The earliest work on learning for this problem was carried out by Michie and Chambers (1968). Their BOXES algorithm was able to balance the pole for over an hour after only about 30 trials. Moreover, unlike many subsequent systems, BOXES was implemented using a real cart and pole, not a simulation. The algorithm first discretized the four-dimensional state space into boxes, hence the name. It then ran trials until the pole fell over or the cart hit the end of the track. Negative reinforcement was associated with the final action in the final box, and then propagated back through the sequence. It was found that the discretization causes some problems when the apparatus was initialized in a different position from those used in training, suggesting that generalization was not perfect. Improved generalization and faster learning can be obtained using an algorithm that *adaptively* partitions the state space according to the observed variation in the reward.

More recently, neural networks have been used to provide a continuous mapping from the state space to the actions, with slightly better results. The most impressive performance, however, belongs to the control algorithm derived using classical control theory for the *triple* inverted pendulum, in which three poles are balanced one on top of another with torque controls at the joints (Furuta *et al.*, 1984). (One is disappointed, but not surprised, that this algorithm was implemented only in simulation.)

20.8 GENETIC ALGORITHMS AND EVOLUTIONARY PROGRAMMING

Nature has a robust way of evolving successful organisms. The organisms that are ill-suited for an environment die off, whereas the ones that are fit live to reproduce. Offspring are similar to their parents, so each new generation has organisms that are similar to the fit members of the previous generation. If the environment changes slowly, the species can gradually evolve along with it, but a sudden change in the environment is likely to wipe out a species. Occasionally, random mutations occur, and although most of these mean a quick death for the mutated individual, some mutations lead to new successful species. The publication of Darwin's *The Origin of Species on the Basis of Natural Selection* was a major turning point in the history of science.

It turns out that what's good for nature is also good for artificial systems. Figure 20.15 shows the GENETIC-ALGORITHM, which starts with a set of one or more individuals and applies selection and reproduction operators to "evolve" an individual that is successful, as measured by a **fitness function.** There are several choices for what the individuals are. They can be entire agent functions, in which case the fitness function is a performance measure or reward function, and the analogy to natural selection is greatest. They can be component functions of an agent, in which case the fitness function is the critic. Or they can be anything at all that can be framed as an optimization problem.

FITNESS FUNCTION

Since the evolutionary process learns an agent function based on occasional rewards (offspring) as supplied by the selection function, it can be seen as a form of reinforcement learning. Unlike the algorithms described in the previous sections, however, no attempt is made to learn the relationship between the rewards and the actions taken by the agent or the states of the environment. GENETIC-ALGORITHM simply searches directly in the space of individuals, with the goal of finding one that maximizes the fitness function. The search is parallel because each individual in the population can be seen as a separate search. It is hill climbing because we are making small genetic changes to the individuals and using the best resulting offspring. The key question is how to allocate the searching resources: clearly, we should spend most of our time on the most promising individuals, but if we ignore the low-scoring ones, we risk getting stuck on a local maximum. It can be shown that, under certain assumptions, the genetic algorithm allocates resources in an optimal way (see the discussion of n-armed bandits in, e.g., Goldberg (1989)).

Before we can apply GENETIC-ALGORITHM to a problem, we need to answer the following four questions:

- What is the fitness function?
- How is an individual represented?
- How are individuals selected?
- How do individuals reproduce?

The fitness function depends on the problem, but in any case, it is a function that takes an individual as input and returns a real number as output.

In the "classic" genetic algorithm approach, an individual is represented as a string over a finite alphabet. Each element of the string is called a **gene.** In real DNA, the alphabet is AGTC (adenine, guanine, thymine, cytosine), but in genetic algorithms, we usually use the binary alphabet (0,1). Some authors reserve the term "genetic algorithm" for cases where the representation

GENE

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual
  repeat
    parents ← SELECTION(population, FITNESS-FN)
    population ← REPRODUCTION(parents)
  until some individual is fit enough
  return the best individual in population, according to FITNESS-FN
```

Figure 20.15 The genetic algorithm finds a fit individual using simulated evolution.

EVOLUTIONARY PROGRAMMING

is a bit string, and use the term **evolutionary programming** when the representation is more complicated. Other authors make no distinction, or make a slightly different one.

The selection strategy is usually randomized, with the probability of selection proportional to fitness. That is, if individual X scores twice as high as Y on the fitness function, then X is twice as likely to be selected for reproduction than is Y. Usually, selection is done with replacement, so that a very fit individual will get to reproduce several times.

Reproduction is accomplished by cross-over and mutation. First, all the individuals that have been selected for reproduction are randomly paired. Then for each pair, a cross-over point is randomly chosen. Think of the genes of each parent as being numbered from 1 to N. The

CROSS-OVER

cross-over point is a number in that range; let us say it is 10. That means that one offspring will get genes 1 through 10 from the first parent, and the rest from the second parent. The second offspring will get genes 1 through 10 from the second parent, and the rest from the

MUTATION

first. However, each gene can be altered by random **mutation** to a different value, with small independent probability. Figure 20.16 diagrams the process.

For example, suppose we are trying to learn a decision list representation for the restaurant waiting problem (see page 556). The fitness function in this case is simply the number of examples that an individual is consistent with. The representation is the tricky part. There are ten attributes in the problem, but not all of them are binary. It turns out that we need 5 bits to represent each distinct attribute/value pair:

$$
\begin{aligned}
&00000 : \mathit{Alternate}(x)\\
&00001 : \neg\mathit{Alternate}(x)\\
&\vdots\\
&10111 : \mathit{WaitEstimate}(x, 0{-}10)\\
&11000 : \mathit{WaitEstimate}(x, 10{-}30)\\
&11001 : \mathit{WaitEstimate}(x, \mathit{30{-}60})\\
&11010 : \mathit{WaitEstimate}(x, {>}60)
\end{aligned}
$$

We also need one bit for each test to say if the outcome is *Yes* or *No*. Thus, if we want to represent a k-DL with a length of up to t tests, we need a representation with $t(5k + 1)$ bits. We can use the standard selection and reproduction approaches. Mutation can flip an outcome or change an attribute. Cross-over combines the head of one decision list with the tail of another.

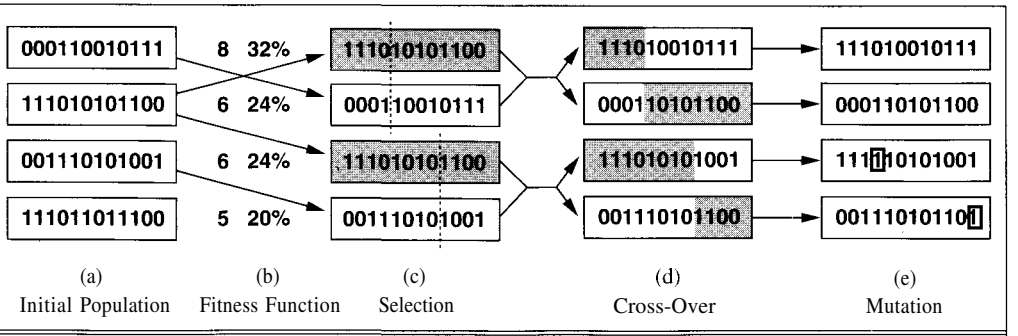


Figure 20.16 The genetic algorithm. In (a), we have an initial population of 4 individuals. They are scored by the fitness function in (b); the top individual scores an 8 and the bottom scores a 5. It works out that the top individual has a 32% chance of being chosen on each selection. In (c), selection has given us two pairs of mates, and the cross-over points (dotted lines) have been chosen. Notice that one individual mates twice; one not at all. In (d), we see the new offspring, generated by cross-over of their parents' genes. Finally, in (e), mutation has changed the two bits surrounded by boxes. This gives us the population for the next generation.

Like neural networks, genetic algorithms are easy to apply to a wide range of problems. The results can be very good on some problems, and rather poor on others. In fact, Denker's remark that "neural networks are the second best way of doing just about anything" has been extended with "and genetic algorithms are the third." But don't be afraid to try a quick implementation of a genetic algorithm on a new problem—just to see if it does work—before investing more time thinking about another approach.

20.9 SUMMARY

This chapter has examined the reinforcement learning problem—how an agent can become proficient in an unknown environment given only its percepts and occasional rewards. Reinforcement learning can be viewed as a microcosm for the entire AI problem, but is studied in a number of simplified settings to facilitate progress. The following major points were made:

- The overall agent design dictates the kind of information that must be learned. The two main designs studied are the model-based design, using a model M and a utility function U, and the model-free approach, using an action-value function Q.
- The utility of a state is the expected sum of rewards received between now and termination of the sequence.
- Utilities can be learned using three approaches.
  1. The LMS (least-mean-square) approach uses the total observed reward-to-go for a given state as direct evidence for learning its utility. LMS uses the model only for the purposes of selecting actions.

2. The ADP (adaptive dynamic programming) approach uses the value or policy iteration algorithm to calculate exact utilities of states given an estimated model. ADP makes optimal use of the local constraints on utilities of states imposed by the neighborhood structure of the environment.
3. The TD (temporal-difference) approach updates utility estimates to match those of successor states, and can be viewed as a simple approximation to the ADP approach that requires no model for the learning process. Using the model to generate pseudo-experiences can, however, result in faster learning.

- Action-value functions, or Q-functions, can be learned by an ADP approach or a TD approach. With TD, Q-learning requires no model in either the learning or action-selection phases. This simplifies the learning problem but potentially restricts the ability to learn in complex environments.
- When the learning agent is responsible for selecting actions while it learns, it must trade off the estimated value of those actions against the potential for learning useful new information. Exact solution of the exploration problem is infeasible, but some simple heuristics do a reasonable job.
- In large state spaces, reinforcement learning algorithms must use an implicit functional representation in order to perform input generalization over states. The temporal-difference signal can be used directly to direct weight changes in parametric representations such as neural networks.
- Combining input generalization with an explicit model has resulted in excellent performance in complex domains.
- Genetic algorithms achieve reinforcement learning by using the reinforcement to increase the proportion of successful functions in a population of programs. They achieve the effect of generalization by mutating and cross-breeding programs with each other.

Because of its potential for eliminating hand coding of control strategies, reinforcement learning continues to be one of the most active areas of machine learning research. Applications in robotics promise to be particularly valuable. As yet, however, there is little understanding of how to extend these methods to the more powerful performance elements described in earlier chapters. Reinforcement learning in *inaccessible* environments is also a topic of current research.

BIBLIOGRAPHICAL AND HISTORICAL NOTLS

Arthur Samuel's work (1959) was probably the earliest successful machine learning research. Although this work was informal and had a number of flaws, it contained most of the modern ideas in reinforcement learning, including temporal differencing and input generalization. Around the same time, researchers in adaptive control theory (Widrow and Hoff, 1960), building on work by Hebb (1949), were training simple networks using the LMS rule. (This early connection between neural networks and reinforcement learning may have led to the persistent misperception that the latter is a subfield of the former.) The cart-pole work of Michie and Chambers (1968) can also be seen as a reinforcement learning method with input generalization.

A more recent tradition springs from work at the University of Massachusetts in the early 1980s (Barto *et al.*, 1981). The paper by Sutton (1988) reinvigorated reinforcement learning research in AI, and provides a good historical overview. The Ph.D. theses by Watkins (1989) and Kaelbling (1990) and the survey by Barto *et al.* (1991) also contain good reviews of the field. Watkin's thesis originated Q-learning, and proved its convergence in the limit. Some recent work appears in a special issue of *Machine Learning* (Vol. 8, Nos. 3/4, 1992), with an excellent introduction by Sutton. The presentation in this chapter is heavily influenced by Moore and Atkeson (1993), who make a clear connection between temporal differencing and classical dynamic programming techniques. The latter paper also introduced the idea of prioritized sweeping. An almost identical method was developed independently by Peng and Williams (1993). Bandit problems, which model the problem of exploration, are analyzed in depth by Berry and Fristedt (1985).

Reinforcement learning in games has also undergone a renaissance in recent years. In addition to Tesauro's work, a world-class Othello system was developed by Lee and Mahajan (1988). Reinforcement learning papers are published frequently in the journal *Machine Learning,* and in the International Conferences on Machine Learning.

Genetic algorithms originated in the work of Friedberg (1958), who attempted to produce learning by mutating small FORTRAN programs. Since most mutations to the programs produced inoperative code, little progress was made. John Holland (1975) reinvigorated the field by using bit-string representations of agents such that any possible string represented a functioning agent. John Koza (1992) has championed more complex representations of agents coupled with mutation and mating techniques that pay careful attention to the syntax of the representation language. Current research appears in the annual Conference on Evolutionary Programming.

EXERCISES

20.1 Show that the estimates developed by the LMS-UPDATE algorithm do indeed minimize the mean square error on the training data.

20.2 Implement a passive learning agent in a simple environment, such as that shown in Figure 20.1. For the case of an initially unknown environment model, compare the learning performance of the LMS, TD, and ADP algorithms.

20.3 Starting with the passive ADP agent, modify it to use an approximate ADP algorithm as discussed in the text. Do this in two steps:

a. Implement a priority queue for adjustments to the utility estimates. Whenever a state is adjusted, all of its predecessors also become candidates for adjustment, and should be added to the queue. The queue is initialized using the state from which the most recent transition took place. Change ADP-UPDATE to allow only a fixed number of adjustments.

b. Experiment with various heuristics for ordering the priority queue, examining their effect on learning rates and computation time.

20.4 The environments used in the chapter all assume that training sequences are finite. In environments with no clear termination point, the unlimited accumulation of rewards can lead to problems with infinite utilities. To avoid this, a discount factor 7 is often used, where $7 < 1$. A reward k steps in the future is discounted by a factor of γ^k. For each constraint and update equation in the chapter, explain how to incorporate the discount factor.

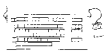
20.5 The description of reinforcement learning agents in Section 20.1 uses distinguished terminal states to indicate the end of a training sequence. Explain how this additional complication could be eliminated by modelling the "reset" as a transition like any other. How will this affect the definition of utility?

20.6 Prove formally that Equations (20.1) and (20.3) are consistent with the definition of utility as the expected reward-to-go of a state.

20.7 How can the value determination algorithm be used to calculate the expected loss experienced by an agent using a given set of utility estimates U and an estimated model M, compared to an agent using correct values?

20.8 Adapt the vacuum world (Chapter 2) for reinforcement learning by including rewards for picking up each piece of dirt and for getting home and switching off. Make the world accessible by providing suitable percepts. Now experiment with different reinforcement learning agents. Is input generalization necessary for success?

20.9 Write down the update equation for Q-learning with a parameterized implicit representation. That is, write the counterpart to Equation (20.8).



20.10 Extend the standard game-playing environment (Chapter 5) to incorporate a reward signal. Put two reinforcement learning agents into the environment (they may of course share the agent program) and have them play against each other. Apply the generalized TD update rule (Equation (20.8)) to update the evaluation function. You may wish to start with a simple linear weighted evaluation function, and a simple game such as tic-tac-toe.

20.11 (Discussion topic.) Is reinforcement learning an appropriate abstract model for human learning? For evolution?