18 *Reinforcement Learning*

In reinforcement learning, the learner is a decision-making agent that takes actions in an environment and receives reward (or penalty) for its actions in trying to solve a problem. After a set of trial-and-error runs, it should learn the best policy, which is the sequence of actions that maximize the total reward.

18.1 Introduction

LET US SAY we want to build a machine that learns to play chess. In this case we cannot use a supervised learner for two reasons. First, it is very costly to have a teacher that will take us through many games and indicate us the best move for each position. Second, in many cases, there is no such thing as the best move; the goodness of a move depends on the moves that follow. A single move does not count; a sequence of moves is good if after playing them we win the game. The only feedback is at the end of the game when we win or lose the game.

Another example is a robot that is placed in a maze. The robot can move in one of the four compass directions and should make a sequence of movements to reach the exit. As long as the robot is in the maze, there is no feedback and the robot tries many moves until it reaches the exit and only then does it get a reward. In this case there is no opponent, but we can have a preference for shorter trajectories, implying that in this case we play against time.

These two applications have a number of points in common: There is a decision maker, called the *agent*, that is placed in an *environment* (see figure 18.1). In chess, the game-player is the decision maker and the environment is the board; in the second case, the maze is the environment

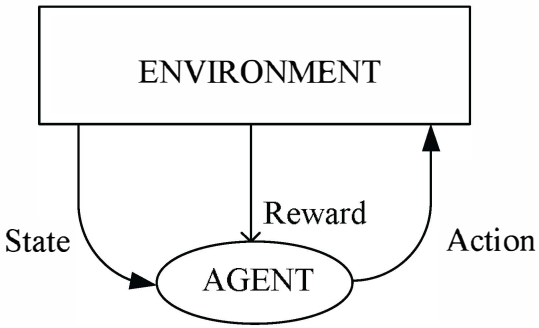

Figure 18.1 The agent interacts with an environment. At any state of the environment, the agent takes an action that changes the state and returns a reward.

of the robot. At any time, the environment is in a certain *state* that is one of a set of possible states—for example, the state of the board, the position of the robot in the maze. The decision maker has a set of *actions* possible: legal movement of pieces on the chess board, movement of the robot in possible directions without hitting the walls, and so forth. Once an action is chosen and taken, the state changes. The solution to the task requires a sequence of actions, and we get feedback, in the form of a *reward* rarely, generally only when the complete sequence is carried out. The reward defines the problem and is necessary if we want a *learning* agent. The learning agent learns the best sequence of actions to solve a problem where "best" is quantified as the sequence of actions that has the maximum cumulative reward. Such is the setting of *reinforcement learning*.

Reinforcement learning is different from the learning methods we discussed before in a number of respects. It is called "learning with a critic," as opposed to learning with a teacher which we have in supervised learning. A *critic* differs from a teacher in that it does not tell us what to do but only how well we have been doing in the past; the critic never informs in advance. The feedback from the critic is scarce and when it comes, it comes late. This leads to the *credit assignment* problem. After taking several actions and getting the reward, we would like to assess the individual actions we did in the past and find the moves that led us to win the reward so that we can record and recall them later on. As we see shortly, what a reinforcement learning program does is that it learns to generate

CRITIC

CREDIT ASSIGNMENT

an *internal value* for the intermediate states or actions in terms of how good they are in leading us to the goal and getting us to the real reward. Once such an internal reward mechanism is learned, the agent can just take the local actions to maximize it.

The solution to the task requires a *sequence* of actions, and from this perspective, we remember the Markov models we discussed in chapter 15. Indeed, we use a Markov decision process to model the agent. The difference is that in the case of Markov models, there is an external process that generates a sequence of signals, for example, speech, which we observe and model. In the current case, however, it is the agent that generates the sequence of actions. Previously, we also made a distinction between observable and hidden Markov models where the states are observed or hidden (and should be inferred) respectively. Similarly here, sometimes we have a partially observable Markov decision process in cases where the agent does not know its state exactly but should infer it with some uncertainty through observations using sensors. For example, in the case of a robot moving in a room, the robot may not know its exact position in the room, nor the exact location of obstacles nor the goal, and should make decisions through a limited image provided by a camera.

18.2 Single State Case: *K*-Armed Bandit

K-ARMED BANDIT

We start with a simple example. The K-*armed bandit* is a hypothetical slot machine with K levers. The action is to choose and pull one of the levers, and we win a certain amount of money that is the reward associated with the lever (action). The task is to decide which lever to pull to maximize the reward. This is a classification problem where we choose one of K. If this were supervised learning, then the teacher would tell us the correct class, namely, the lever leading to maximum earning. In this case of reinforcement learning, we can only try different levers and keep track of the best. This is a simplified reinforcement learning problem because there is only one state, or one slot machine, and we need only decide on the action. Another reason why this is simplified is that we immediately get a reward after a single action; the reward is not delayed, so we immediately see the value of our action.

Let us say $Q(a)$ is the value of action a. Initially, $Q(a) = 0$ for all a. When we try action a, we get reward $r_a \geq 0$. If rewards are deterministic, we always get the same r_a for any pull of a and in such a case, we can

just set $Q(a) = r_a$. If we want to exploit, once we find an action a such that $Q(a) > 0$, we can keep choosing it and get r_a at each pull. However, it is quite possible that there is another lever with a higher reward, so we need to explore.

We can choose different actions and store $Q(a)$ for all a. Whenever we want to exploit, we can choose the action with the maximum value, that is,

$$\text{choose } a^* \text{ if } Q(a^*) = \max_a Q(a) \tag{18.1}$$

If rewards are not deterministic but stochastic, we get a different reward each time we choose the same action. The amount of the reward is defined by the probability distribution $p(r|a)$. In such a case, we define $Q_t(a)$ as the estimate of the value of action a at time t. It is an average of all rewards received when action a was chosen before time t. An online update can be defined as

$$Q_{t+1}(a) \leftarrow Q_t(a) + \eta[r_{t+1}(a) - Q_t(a)] \tag{18.2}$$

where $r_{t+1}(a)$ is the reward received after taking action a at time $(t+1)$st time.

Note that equation 18.2 is the *delta rule* that we have used on many occasions in the previous chapters: η is the learning factor (gradually decreased in time for convergence), r_{t+1} is the desired output, and $Q_t(a)$ is the current prediction. $Q_{t+1}(a)$ is the *expected* value of action a at time $t+1$ and converges to the mean of $p(r|a)$ as t increases.

The full reinforcement learning problem generalizes this simple case in a number of ways. First, we have several states. This corresponds to having several slot machines with different reward probabilities, $p(r|s_i, a_j)$, and we need to learn $Q(s_i, a_j)$, which is the value of taking action a_j when in state s_i. Second, the actions affect not only the reward but also the next state, and we move from one state to another. Third, the rewards are delayed and we need to be able to estimate immediate values from delayed rewards.

18.3 Elements of Reinforcement Learning

The learning decision maker is called the *agent*. The agent interacts with the *environment* that includes everything outside the agent. The agent has sensors to decide on its *state* in the environment and takes an *action*

that modifies its state. When the agent takes an action, the environment provides a *reward*. Time is discrete as $t = 0, 1, 2, \ldots$, and $s_t \in S$ denotes the state of the agent at time t where S is the set of all possible states. $a_t \in \mathcal{A}(s_t)$ denotes the action that the agent takes at time t where $\mathcal{A}(s_t)$ is the set of possible actions in state s_t. When the agent in state s_t takes the action a_t, the clock ticks, reward $r_{t+1} \in \Re$ is received, and the agent moves to the next state, s_{t+1}. The problem is modeled using a *Markov decision process* (MDP). The reward and next state are sampled from their respective probability distributions, $p(r_{t+1}|s_t, a_t)$ and $P(s_{t+1}|s_t, a_t)$. Note that what we have is a *Markov* system where the state and reward in the next time step depend only on the current state and action. In some applications, reward and next state are deterministic, and for a certain state and action taken, there is one possible reward value and next state.

MARKOV DECISION PROCESS

Depending on the application, a certain state may be designated as the initial state and in some applications, there is also an absorbing terminal (goal) state where the search ends; all actions in this terminal state transition to itself with probability 1 and without any reward. The sequence of actions from the start to the terminal state is an *episode*, or a *trial*.

EPISODE

The *policy*, π, defines the agent's behavior and is a mapping from the states of the environment to actions: $\pi : S \rightarrow \mathcal{A}$. The policy defines the action to be taken in any state s_t: $a_t = \pi(s_t)$. The *value* of a policy π, $V^\pi(s_t)$, is the expected cumulative reward that will be received while the agent follows the policy, starting from state s_t.

POLICY

In the *finite-horizon* or *episodic* model, the agent tries to maximize the expected reward for the next T steps:

FINITE-HORIZON

$$V^\pi(s_t) = E[r_{t+1} + r_{t+2} + \cdots + r_{t+T}] = E\left[\sum_{i=1}^{T} r_{t+i}\right] \tag{18.3}$$

Certain tasks are continuing, and there is no prior fixed limit to the episode. In the *infinite-horizon* model, there is no sequence limit, but future rewards are discounted:

INFINITE-HORIZON

$$V^\pi(s_t) = E[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots] = E\left[\sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i}\right] \tag{18.4}$$

where $0 \leq \gamma < 1$ is the *discount rate* to keep the return finite. If $\gamma = 0$, then only the immediate reward counts. As γ approaches 1, rewards further in the future count more, and we say that the agent becomes more farsighted. γ is less than 1 because there generally is a time limit

DISCOUNT RATE

to the sequence of actions needed to solve the task. The agent may be a robot that runs on a battery. We prefer rewards sooner rather than later because we are not certain how long we will survive.

OPTIMAL POLICY

For each policy π, there is a $V^{\pi}(s_t)$, and we want to find the *optimal policy* π^* such that

$$V^*(s_t) = \max_{\pi} V^{\pi}(s_t), \forall s_t \tag{18.5}$$

In some applications, for example, in control, instead of working with the values of states, $V(s_t)$, we prefer to work with the values of state-action pairs, $Q(s_t, a_t)$. $V(s_t)$ denotes how good it is for the agent to be in state s_t, whereas $Q(s_t, a_t)$ denotes how good it is to perform action a_t when in state s_t. We define $Q^*(s_t, a_t)$ as the value, that is, the expected cumulative reward, of action a_t taken in state s_t and then obeying the optimal policy afterward. The value of a state is equal to the value of the best possible action:

$$\begin{aligned} V^*(s_t) &= \max_{a_t} Q^*(s_t, a_t) \\ &= \max_{a_t} E\left[\sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i}\right] \\ &= \max_{a_t} E\left[r_{t+1} + \gamma \sum_{i=1}^{\infty} \gamma^{i-1} r_{t+i+1}\right] \\ &= \max_{a_t} E\left[r_{t+1} + \gamma V^*(s_{t+1})\right] \end{aligned}$$

$$V^*(s_t) = \max_{a_t} \left(E[r_{t+1}] + \gamma \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t) V^*(s_{t+1})\right) \tag{18.6}$$

To each possible next state s_{t+1}, we move with probability $P(s_{t+1}|s_t, a_t)$, and continuing from there using the optimal policy, the expected cumulative reward is $V^*(s_{t+1})$. We sum over all such possible next states, and we discount it because it is one time step later. Adding our immediate expected reward, we get the total expected cumulative reward for action a_t. We then choose the best of possible actions. Equation 18.6 is known as *Bellman's equation* (Bellman 1957). Similarly, we can also write

BELLMAN'S EQUATION

$$Q^*(s_t, a_t) = E[r_{t+1}] + \gamma \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t) \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) \tag{18.7}$$

```
Initialize V(s) to arbitrary values
Repeat
    For all s ∈ S
        For all a ∈ A
            Q(s,a) ← E[r|s,a] + γ Σ_{s'∈S} P(s'|s,a)V(s')
        V(s) ← max_a Q(s,a)
Until V(s) converge
```

Figure 18.2 Value iteration algorithm for model-based learning.

Once we have $Q^*(s_t, a_t)$ values, we can then define our policy π as taking the action a_t^*, which has the highest value among all $Q^*(s_t, a_t)$:

$$\pi^*(s_t) : \text{Choose } a_t^* \text{ where } Q^*(s_t, a_t^*) = \max_{a_t} Q^*(s_t, a_t) \tag{18.8}$$

This means that if we have the $Q^*(s_t, a_t)$ values, then by using a greedy search at each *local* step we get the optimal sequence of steps that maximizes the *cumulative* reward.

18.4 Model-Based Learning

We start with model-based learning where we completely know the environment model parameters, $p(r_{t+1}|s_t, a_t)$ and $P(s_{t+1}|s_t, a_t)$. In such a case, we do not need any exploration and can directly solve for the optimal value function and policy using dynamic programming. The optimal value function is unique and is the solution to the simultaneous equations given in equation 18.6. Once we have the optimal value function, the optimal policy is to choose the action that maximizes the value in the next state:

$$\pi^*(s_t) = \arg\max_{a_t} \left(E[r_{t+1}|s_t, a_t] + \gamma \sum_{s_{t+1} \in S} P(s_{t+1}|s_t, a_t) V^*(s_t + 1) \right) \tag{18.9}$$

18.4.1 Value Iteration

VALUE ITERATION

To find the optimal policy, we can use the optimal value function, and there is an iterative algorithm called *value iteration* that has been shown to converge to the correct V^* values. Its pseudocode is given in figure 18.2.

Initialize a policy π' arbitrarily
Repeat
    $\pi \leftarrow \pi'$
    Compute the values using π by
        solving the linear equations
            $V^{\pi}(s) = E[r|s, \pi(s)] + \gamma \sum_{s' \in S} P(s'|s, \pi(s)) V^{\pi}(s')$
    Improve the policy at each state
        $\pi'(s) \leftarrow \arg\max_a (E[r|s, a] + \gamma \sum_{s' \in S} P(s'|s, a) V^{\pi}(s'))$
Until $\pi = \pi'$

Figure 18.3 Policy iteration algorithm for model-based learning.

We say that the values converged if the maximum value difference between two iterations is less than a certain threshold δ:

$$\max_{s \in S} |V^{(l+1)}(s) - V^{(l)}(s)| < \delta$$

where l is the iteration counter. Because we care only about the actions with the maximum value, it is possible that the policy converges to the optimal one even before the values converge to their optimal values. Each iteration is $\mathcal{O}(|S|^2|\mathcal{A}|)$, but frequently there is only a small number $k < |S|$ of next possible states, so complexity decreases to $\mathcal{O}(k|S||\mathcal{A}|)$.

18.4.2 Policy Iteration

In policy iteration, we store and update the policy rather than doing this indirectly over the values. The pseudocode is given in figure 18.3. The idea is to start with a policy and improve it repeatedly until there is no change. The value function can be calculated by solving for the linear equations. We then check whether we can improve the policy by taking these into account. This step is guaranteed to improve the policy, and when no improvement is possible, the policy is guaranteed to be optimal. Each iteration of this algorithm takes $\mathcal{O}(|\mathcal{A}||S|^2 + |S|^3)$ time that is more than that of value iteration, but policy iteration needs fewer iterations than value iteration.

18.5 Temporal Difference Learning

Model is defined by the reward and next state probability distributions, and as we saw in section 18.4, when we know these, we can solve for the optimal policy using dynamic programming. However, these methods are costly, and we seldom have such perfect knowledge of the environment. The more interesting and realistic application of reinforcement learning is when we do not have the model. This requires exploration of the environment to query the model. We first discuss how this exploration is done and later see model-free learning algorithms for deterministic and nondeterministic cases. Though we are not going to assume a full knowledge of the environment model, we will however require that it be stationary.

As we will see shortly, when we explore and get to see the value of the next state and reward, we use this information to update the value of the current state. These algorithms are called *temporal difference* algorithms because what we do is look at the difference between our current estimate of the value of a state (or a state-action pair) and the discounted value of the next state and the reward received.

TEMPORAL DIFFERENCE

18.5.1 Exploration Strategies

To explore, one possibility is to use *ϵ-greedy* search where with probability ϵ, we choose one action uniformly randomly among all possible actions, namely, explore, and with probability $1 - \epsilon$, we choose the best action, namely, exploit. We do not want to continue exploring indefinitely but start exploiting once we do enough exploration; for this, we start with a high ϵ value and gradually decrease it. We need to make sure that our policy is *soft*, that is, the probability of choosing any action $a \in \mathcal{A}$ in state $s \in S$ is greater than 0.

We can choose probabilistically, using the softmax function to convert values to probabilities

$$P(a|s) = \frac{\exp Q(s,a)}{\sum_{b \in \mathcal{A}} \exp Q(s,b)} \tag{18.10}$$

and then sample according to these probabilities. To gradually move from exploration to exploitation, we can use a "temperature" variable T and define the probability of choosing action a as

$$P(a|s) = \frac{\exp[Q(s,a)/T]}{\sum_{b \in \mathcal{A}} \exp[Q(s,b)/T]} \tag{18.11}$$

When T is large, all probabilities are equal and we have exploration. When T is small, better actions are favored. So the strategy is to start with a large T and decrease it gradually, a procedure named *annealing*, which in this case moves from exploration to exploitation smoothly in time.

18.5.2 Deterministic Rewards and Actions

In model-free learning, we first discuss the simpler deterministic case, where at any state-action pair, there is a single reward and next state possible. In this case, equation 18.7 reduces to

$$Q(s_t, a_t) = r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \tag{18.12}$$

and we simply use this as an assignment to update $Q(s_t, a_t)$. When in state s_t, we choose action a_t by one of the stochastic strategies we saw earlier, which returns a reward r_{t+1} and takes us to state s_{t+1}. We then update the value of *previous* action as

$$\hat{Q}(s_t, a_t) \leftarrow r_{t+1} + \gamma \max_{a_{t+1}} \hat{Q}(s_{t+1}, a_{t+1}) \tag{18.13}$$

where the hat denotes that the value is an estimate. $\hat{Q}(s_{t+1}, a_{t+1})$ is a later value and has a higher chance of being correct. We discount this by γ and add the immediate reward (if any) and take this as the new estimate for the previous $\hat{Q}(s_t, a_t)$. This is called a *backup* because it can be viewed as taking the estimated value of an action in the next time step and "backing it up" to revise the estimate for the value of a current action.

BACKUP

For now we assume that all $\hat{Q}(s, a)$ values are stored in a table; we will see later on how we can store this information more succinctly when $|S|$ and $|\mathcal{A}|$ are large.

Initially all $\hat{Q}(s_t, a_t)$ are 0, and they are updated in time as a result of trial episodes. Let us say we have a sequence of moves and at each move, we use equation 18.13 to update the estimate of the Q value of the previous state-action pair using the Q value of the current state-action pair. In the intermediate states, all rewards and therefore values are 0, so no update is done. When we get to the goal state, we get the reward r and then we can update the Q value of the previous state-action pair as γr. As for the preceding state-action pair, its immediate reward is 0 and the contribution from the next state-action pair is discounted by γ because it is one step later. Then in another episode, if we reach this

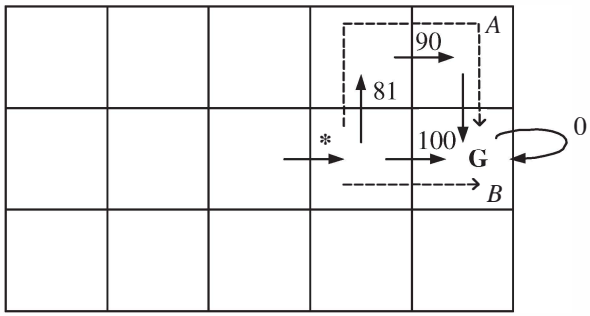

Figure 18.4 Example to show that Q values increase but never decrease. This is a deterministic grid-world where G is the goal state with reward 100, all other immediate rewards are 0, and $\gamma = 0.9$. Let us consider the Q value of the transition marked by asterisk, and let us just consider only the two paths A and B. Let us say that path A is seen before path B, then we have $\gamma \max(0, 81) = 72.9$; if afterward B is seen, a shorter path is found and the Q value becomes $\gamma \max(100, 81) = 90$. If B is seen before A, the Q value is $\gamma \max(100, 0) = 90$; then when A is seen, it does not change because $\gamma \max(100, 81) = 90$.

state, we can update the one preceding that as $\gamma^2 r$, and so on. This way, after many episodes, this information is backed up to earlier state-action pairs. Q values increase until they reach their optimal values as we find paths with higher cumulative reward, for example, shorter paths, but they never decrease (see figure 18.4).

Note that we do not know the reward or next state functions here. They are part of the environment, and it is as if we query them when we explore. We are not modeling them either, though that is another possibility. We just accept them as given and learn directly the optimal policy through the estimated value function.

18.5.3 Nondeterministic Rewards and Actions

If the rewards and the result of actions are not deterministic, then we have a probability distribution for the reward $p(r_{t+1}|s_t, a_t)$ from which rewards are sampled, and there is a probability distribution for the next state $P(s_{t+1}|s_t, a_t)$. These help us model the uncertainty in the system that may be due to forces we cannot control in the environment: for instance, our opponent in chess, the dice in backgammon, or our lack of

```
Initialize all Q(s, a) arbitrarily
For all episodes
    Initalize s
    Repeat
        Choose a using policy derived from Q, e.g., ε-greedy
        Take action a, observe r and s′
        Update Q(s, a):
            Q(s, a) ← Q(s, a) + η(r + γ max_a′ Q(s′, a′) − Q(s, a))
        s ← s′
    Until s is terminal state
```

Figure 18.5 Q learning, which is an off-policy temporal difference algorithm.

knowledge of the system. For example, we may have an imperfect robot which sometimes fails to go in the intended direction and deviates, or advances shorter or longer than expected.

In such a case, we have

$$Q(s_t, a_t) = E[r_{t+1}] + \gamma \sum_{s_{t+1}} P(s_{t+1}|s_t, a_t) \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \tag{18.14}$$

We cannot do a direct assignment in this case because for the same state and action, we may receive different rewards or move to different next states. What we do is keep a running average. This is known as the Q *learning* algorithm:

Q LEARNING

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \eta(r_{t+1} + \gamma \max_{a_{t+1}} \hat{Q}(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \tag{18.15}$$

We think of $r_{t+1} + \gamma \max_{a_{t+1}} \hat{Q}(s_{t+1}, a_{t+1})$ values as a sample of instances for each (s_t, a_t) pair and we would like $\hat{Q}(s_t, a_t)$ to converge to its mean. As usual η is gradually decreased in time for convergence, and it has been shown that this algorithm converges to the optimal Q^* values (Watkins and Dayan 1992). The pseudocode of the Q learning algorithm is given in figure 18.5.

We can also think of equation 18.15 as reducing the difference between the current Q value and the backed-up estimate, from one time step later. Such algorithms are called *temporal difference* (TD) algorithms (Sutton 1988).

TEMPORAL DIFFERENCE

This is an *off-policy* method as the value of the best next action is used without using the policy. In an *on-policy* method, the policy is used to

OFF-POLICY

ON-POLICY

```
Initialize all Q(s, a) arbitrarily
For all episodes
    Initalize s
    Choose a using policy derived from Q, e.g., ϵ-greedy
    Repeat
        Take action a, observe r and s′
        Choose a′ using policy derived from Q, e.g., ϵ-greedy
        Update Q(s, a):
            Q(s, a) ← Q(s, a) + η(r + γQ(s′, a′) − Q(s, a))
        s ← s′, a ← a′
    Until s is terminal state
```

Figure 18.6 Sarsa algorithm, which is an on-policy version of Q learning.

determine also the next action. The on-policy version of Q learning is the SARSA *Sarsa* algorithm whose pseudocode is given in figure 18.6. We see that instead of looking for all possible next actions a' and choosing the best, the on-policy Sarsa uses the policy derived from Q values to choose one next action a' and uses its Q value to calculate the temporal difference. On-policy methods estimate the value of a policy while using it to take actions. In off-policy methods, these are separated, and the policy used to generate behavior, called the *behavior* policy, may in fact be different from the policy that is evaluated and improved, called the *estimation* policy.

Sarsa converges with probability 1 to the optimal policy and state-action values if a *GLIE policy* is employed to choose actions. A GLIE (greedy in the limit with infinite exploration) policy is where (1) all state-action pairs are visited an infinite number of times, and (2) the policy converges in the limit to the greedy policy (which can be arranged, e.g., with ϵ-greedy policies by setting $\epsilon = 1/t$).

The same idea of temporal difference can also be used to learn $V(s)$ TD LEARNING values, instead of $Q(s, a)$. *TD learning* (Sutton 1988) uses the following update rule to update a state value:

$$V(s_t) \leftarrow V(s_t) + \eta[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{18.16}$$

This again is the delta rule where $r_{t+1} + \gamma V(s_{t+1})$ is the better, later prediction and $V(s_t)$ is the current estimate. Their difference is the temporal difference, and the update is done to decrease this difference. The

update factor η is gradually decreased, and TD is guaranteed to converge to the optimal value function $V^*(s)$.

18.5.4 Eligibility Traces

ELIGIBILITY TRACE

The previous algorithms are one-step—that is, the temporal difference is used to update only the previous value (of the state or state-action pair). An *eligibility trace* is a record of the occurrence of past visits that enables us to implement temporal credit assignment, allowing us to update the values of previously occurring visits as well. We discuss how this is done with Sarsa to learn Q values; adapting this to learn V values is straightforward.

To store the eligibility trace, we require an additional memory variable associated with each state-action pair, $e(s,a)$, initialized to 0. When the state-action pair (s,a) is visited, namely, when we take action a in state s, its eligibility is set to 1; the eligibilities of all other state-action pairs are multiplied by $\gamma\lambda$. $0 \le \lambda \le 1$ is the trace decay parameter.

$$e_t(s,a) = \begin{cases} 1 & \text{if } s = s_t \text{ and } a = a_t, \\ \gamma\lambda e_{t-1}(s,a) & \text{otherwise} \end{cases} \tag{18.17}$$

If a state-action pair has never been visited, its eligibility remains 0; if it has been, as time passes and other state-actions are visited, its eligibility decays depending on the value of γ and λ (see figure 18.7).

We remember that in Sarsa, the temporal error at time t is

$$\delta_t = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \tag{18.18}$$

In Sarsa with an eligibility trace, named Sarsa(λ), *all* state-action pairs are updated as

$$Q(s,a) \leftarrow Q(s,a) + \eta\delta_t e_t(s,a), \ \forall s,a \tag{18.19}$$

This updates all eligible state-action pairs, where the update depends on how far they have occurred in the past. The value of λ defines the temporal credit: If $\lambda = 0$, only a one-step update is done. The algorithms we discussed in section 18.5.3 are such, and for this reason they are named $Q(0)$, Sarsa(0), or TD(0). As λ gets closer to 1, more of the previous steps are considered. When $\lambda = 1$, all previous steps are updated and the credit given to them falls only by γ per step. In online updating, all eligible values are updated immediately after each step; in offline updating, the updates are accumulated and a single update is done at

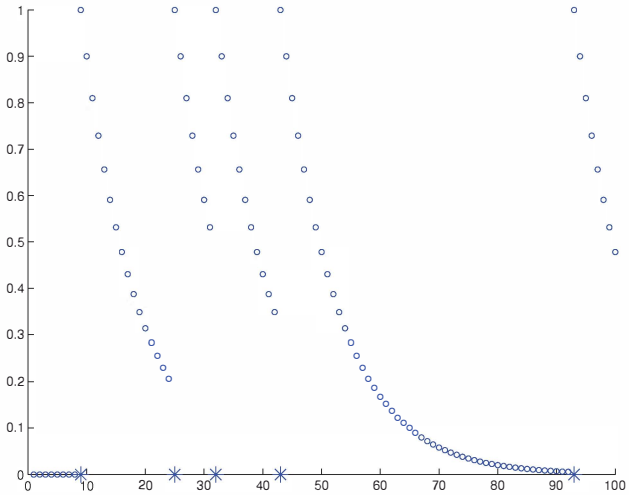

Figure 18.7 Example of an eligibility trace for a value. Visits are marked by an asterisk.

the end of the episode. Online updating takes more time but converges faster. The pseudocode for *Sarsa*(λ) is given in figure 18.8. $Q(\lambda)$ and TD(λ) algorithms can similarly be derived (Sutton and Barto 1998).

SARSA(λ)

18.6 Generalization

Until now, we assumed that the $Q(s,a)$ values (or $V(s)$, if we are estimating values of states) are stored in a lookup table, and the algorithms we considered earlier are called *tabular* algorithms. There are a number of problems with this approach: (1) when the number of states and the number of actions is large, the size of the table may become quite large; (2) states and actions may be continuous, for example, turning the steering wheel by a certain angle, and to use a table, they should be discretized which may cause error; and (3) when the search space is large, too many episodes may be needed to fill in all the entries of the table with acceptable accuracy.

Instead of storing the Q values as they are, we can consider this a regression problem. This is a supervised learning problem where we define a regressor $Q(s,a|\boldsymbol{\theta})$, taking s and a as inputs and parameterized by a

Initialize all $Q(s,a)$ arbitrarily, $e(s,a) \leftarrow 0, \forall s,a$
For all episodes
  Initalize s
  Choose a using policy derived from Q, e.g., ϵ-greedy
  Repeat
    Take action a, observe r and s'
    Choose a' using policy derived from Q, e.g., ϵ-greedy
    $\delta \leftarrow r + \gamma Q(s',a') - Q(s,a)$
    $e(s,a) \leftarrow 1$
    For all s,a:
      $Q(s,a) \leftarrow Q(s,a) + \eta \delta e(s,a)$
      $e(s,a) \leftarrow \gamma \lambda e(s,a)$
    $s \leftarrow s',\ a \leftarrow a'$
  Until s is terminal state

Figure 18.8 Sarsa(λ) algorithm.

vector of parameters, $\boldsymbol{\theta}$, to learn Q values. For example, this can be an artificial neural network with s and a as its inputs, one output, and $\boldsymbol{\theta}$ its connection weights.

A good function approximator has the usual advantages and solves the problems discussed previously. A good approximation may be achieved with a simple model without explicitly storing the training instances; it can use continuous inputs; and it allows generalization. If we know that similar (s,a) pairs have similar Q values, we can generalize from past cases and come up with good $Q(s,a)$ values even if that state-action pair has never been encountered before.

To be able to train the regressor, we need a training set. In the case of Sarsa(0), we saw before that we would like $Q(s_t,a_t)$ to get close to $r_{t+1} + \gamma Q(s_{t+1},a_{t+1})$. So, we can form a set of training samples where the input is the state-action pair (s_t,a_t) and the required output is $r_{t+1} + \gamma Q(s_{t+1},a_{t+1})$. We can write the squared error as

$$E^t(\boldsymbol{\theta}) = [r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)]^2 \tag{18.20}$$

Training sets can similarly be defined for $Q(0)$ and TD(0), where in the latter case we learn $V(s)$, and the required output is $r_{t+1} + \gamma V(s_{t+1})$.

Once such a set is ready, we can use any supervised learning algorithm for learning the training set.

If we are using a gradient descent method, as in training neural networks, the parameter vector is updated as

$$\Delta\boldsymbol{\theta} = \eta[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]\nabla_{\boldsymbol{\theta}_t} Q(s_t, a_t) \tag{18.21}$$

This is a one-step update. In the case of Sarsa(λ), the eligibility trace is also taken into account:

$$\Delta\boldsymbol{\theta} = \eta\delta_t \boldsymbol{e}_t \tag{18.22}$$

where the temporal difference error is

$$\delta_t = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

and the vector of eligibilities of parameters are updated as

$$\boldsymbol{e}_t = \gamma\lambda \boldsymbol{e}_{t-1} + \nabla_{\boldsymbol{\theta}_t} Q(s_t, a_t) \tag{18.23}$$

with $\boldsymbol{e}_0$ all zeros. In the case of a tabular algorithm, the eligibilities are stored for the state-action pairs because they are the parameters (stored as a table). In the case of an estimator, eligibility is associated with the parameters of the estimator. We also note that this is very similar to the momentum method for stabilizing backpropagation (section 11.8.1). The difference is that in the case of momentum previous weight changes are remembered, whereas here previous gradient vectors are remembered. Depending on the model used for $Q(s_t, a_t)$, for example, a neural network, we plug its gradient vector in equation 18.23.

In theory, any regression method can be used to train the Q function, but the particular task has a number of requirements. First, it should allow generalization; that is, we really need to guarantee that similar states and actions have similar Q values. This also requires a good coding of s and a, as in any application, to make the similarities apparent. Second, reinforcement learning updates provide instances one by one and not as a whole training set, and the learning algorithm should be able to do individual updates to learn the new instance without forgetting what has been learned before. For example, a multilayer perceptron using backpropagation can be trained with a single instance only if a small learning rate is used. Or, such instances may be collected to form a training set and learned altogether but this slows down learning as no learning happens while a sufficiently large sample is being collected.

Because of these reasons, it seems a good idea to use local learners to learn the Q values. In such methods, for example, radial basis functions, information is localized and when a new instance is learned, only a local part of the learner is updated without possibly corrupting the information in another part. The same requirements apply if we are estimating the state values as $V(s_t|\boldsymbol{\theta})$.

18.7 Partially Observable States

18.7.1 The Setting

In certain applications, the agent does not know the state exactly. It is equipped with sensors that return an *observation*, which the agent then uses to estimate the state. Let us say we have a robot that navigates in a room. The robot may not know its exact location in the room, or what else is there in the room. The robot may have a camera with which sensory observations are recorded. This does not tell the robot its state exactly but gives some indication as to its likely state. For example, the robot may only know that there is an obstacle to its right.

The setting is like a Markov decision process, except that after taking an action a_t, the new state s_{t+1} is not known, but we have an observation o_{t+1} that is a stochastic function of s_t and a_t: $p(o_{t+1}|s_t, a_t)$. This is called a *partially observable MDP* (POMDP). If $o_{t+1} = s_{t+1}$, then POMDP reduces to the MDP. This is just like the distinction between observable and hidden Markov models and the solution is similar; that is, from the observation, we need to infer the state (or rather a probability distribution for the states) and then act based on this. If the agent believes that it is in state s_1 with probability 0.4 and in state s_2 with probability 0.6, then the value of any action is 0.4 times the value of the action in s_1 plus 0.6 times the value of the action in s_2.

PARTIALLY OBSERVABLE MDP

The Markov property does not hold for observations. The next state observation does not only depend on the current action and observation. When there is limited observation, two states may appear the same but are different and if these two states require different actions, this can lead to a loss of performance, as measured by the cumulative reward. The agent should somehow compress the past trajectory into a current unique state estimate. These past observations can also be taken into account by taking a past window of observations as input to the policy,

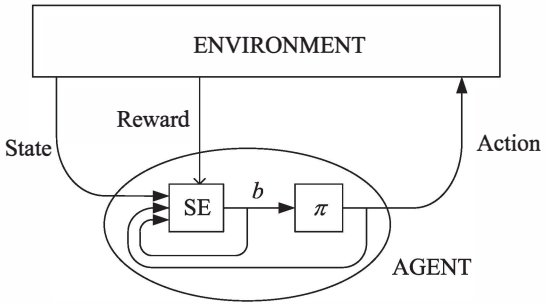

Figure 18.9 In the case of a partially observable environment, the agent has a state estimator (SE) that keeps an internal belief state b and the policy π generates actions based on the belief states.

or one can use a recurrent neural network (section 11.12.2) to maintain the state without forgetting past observations.

At any time, the agent may calculate the most likely state and take an action accordingly. Or it may take an action to gather information and reduce uncertainty, for example, search for a landmark, or stop to ask for direction. This implies the importance of the *value of information*, and indeed POMDPs can be modeled as *dynamic* influence diagrams (section 14.8). The agent chooses between actions based on the amount of information they provide, the amount of reward they produce, and how they change the state of the environment.

VALUE OF INFORMATION

BELIEF STATE

To keep the process Markov, the agent keeps an internal *belief state* b_t that summarizes its experience (see figure 18.9). The agent has a *state estimator* that updates the belief state b_{t+1} based on the last action a_t, current observation o_{t+1}, and its previous belief state b_t. There is a policy π that generates the next action a_{t+1} based on this belief state, as opposed to the actual state that we had in a completely observable environment. The belief state is a probability distribution over states of the environment given the initial belief state (before we did any actions) and the past observation-action history of the agent (without leaving out any information that could improve agent's performance). Q learning in such a case involves the belief state-action pair values, instead of the actual state-action pairs:

$$Q(b_t, a_t) = E[r_{t+1}] + \gamma \sum_{b_{t+1}} P(b_{t+1}|b_t, a_t) V(b_{t+1}) \tag{18.24}$$

18.7.2 Example: The Tiger Problem

We now discuss an example that is a slightly different version of the *Tiger problem* discussed in Kaelbling, Littman, and Cassandra 1998, modified as in the example in Thrun, Burgard, and Fox 2005. Let us say we are standing in front of two doors, one to our left and the other to other right, leading to two rooms. Behind one of the two doors, we do not know which, there is a crouching tiger, and behind the other, there is a treasure. If we open the door of the room where the tiger is, we get a large negative reward, and if we open the door of the treasure room, we get some positive reward. The hidden state, z_L, is the location of the tiger. Let us say p denotes the probability that tiger is in the room to the left and therefore, the tiger is in the room to the right with probability $1 - p$:

$$p \equiv P(z_L = 1)$$

The two actions are a_L and a_R, which respectively correspond to opening the left or the right door. The rewards are

$r(A, Z)$	Tiger left	Tiger right
Open left	−100	+80
Open right	+90	−100

We can calculate the expected reward for the two actions. There are no future rewards because the episode ends once we open one of the doors.

$$\begin{aligned} R(a_L) &= r(a_L, z_L)P(z_L) + r(a_L, z_R)P(z_R) = -100p + 80(1 - p) \\ R(a_R) &= r(a_R, z_L)P(z_L) + r(a_R, z_R)P(z_R) = 90p - 100(1 - p) \end{aligned}$$

Given these rewards, if p is close to 1, if we believe that there is a high chance that the tiger is on the left, the right action will be to choose the right door, and, similarly, for p close to 0, it is better to choose the left door.

The two intersect for p around 0.5, and there the expected reward is approximately −10. The fact that the expected reward is negative when p is around 0.5 (when we have uncertainty) indicates the importance of collecting information. If we can add sensors to to decrease uncertainty—that is, move p away from 0.5 to either close to 0 or close to 1—we can take actions with high positive rewards. That sensing action, a_S, may

have a small negative reward: $R(a_S) = -1$; this may be considered as the cost of sensing or equivalent to discounting future reward by $\gamma < 1$ because we are postponing taking the real action (of opening one of the doors).

In such a case, the expected rewards and value of the best action are shown in figure 18.10a:

$$V = \max(a_L, a_R, a_S)$$

Let us say as sensory input, we use microphones to check whether the tiger is behind the left or the right door. But we have unreliable sensors (so that we still stay in the realm of partial observability). Let us say we can only detect tiger's presence with 0.7 probability:

$$P(o_L|z_L) = 0.7 \qquad P(o_L|z_R) = 0.3$$
$$P(o_R|z_L) = 0.3 \qquad P(o_R|z_R) = 0.7$$

If we sense o_L, our belief in the tiger's position changes:

$$p' = P(z_L|o_L) = \frac{P(o_L|z_L)P(z_L)}{p(o_L)} = \frac{0.7p}{0.7p + 0.3(1-p)}$$

The effect of this is shown in figure 18.10b where we plot $R(a_L|o_L)$. Sensing o_L turns opening the right door into a better action for a wider range. The better sensors we have (if the probability of correct sensing moves from 0.7 closer to 1), the larger this range gets (exercise 9). Similarly, as we see in figure 18.10c, if we sense o_R, this increases the chances of opening the left door. Note that sensing also decreases the range where there is a need to sense (once more).

The expected rewards for the actions in this case are

$$\begin{aligned}
R(a_L|o_L) &= r(a_L, z_L)P(z_L|o_L) + r(a_L, z_R)P(z_R|o_L) \\
&= -100p' + 80(1-p') \\
&= -100 \cdot \frac{0.7 \cdot p}{p(o_L)} + 80 \cdot \frac{0.3 \cdot (1-p)}{p(o_L)} \\
R(a_R|o_L) &= r(a_R, z_L)P(z_L|o_L) + r(a_R, z_R)P(z_R|o_L) \\
&= 90p' - 100(1-p') \\
&= 90 \cdot \frac{0.7 \cdot p}{p(o_L)} - 100 \cdot \frac{0.3 \cdot (1-p)}{p(o_L)} \\
R(a_S|o_L) &= -1
\end{aligned}$$

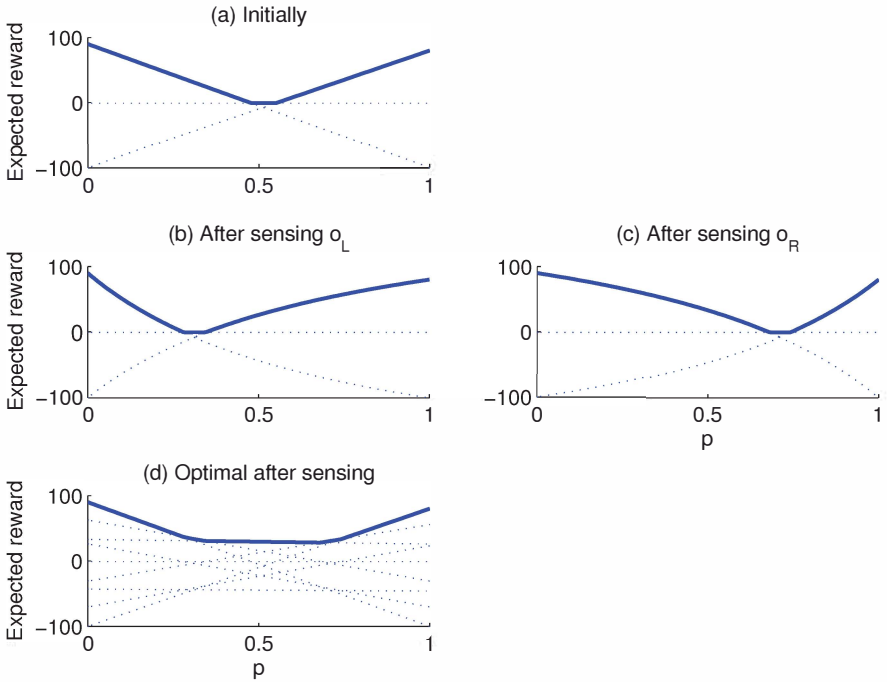

Figure 18.10 Expected rewards and the effect of sensing in the Tiger problem.

The best action is this case is the maximum of these three. Similarly, if we sense o_R, the expected rewards become

$$\begin{aligned} R(a_L|o_R) &= r(a_L,z_L)P(z_L|o_R) + r(a_L,z_R)P(z_R|o_R) \\ &= -100 \cdot \frac{0.3 \cdot p}{p(o_R)} + 80 \cdot \frac{0.7 \cdot (1-p)}{p(o_R)} \\ R(a_R|o_R) &= r(a_R,z_L)P(z_L|o_R) + r(a_R,z_R)P(z_R|o_R) \\ &= 90 \cdot \frac{0.3 \cdot p}{p(o_R)} - 100 \cdot \frac{0.7 \cdot (1-p)}{p(o_R)} \\ R(a_S|o_R) &= -1 \end{aligned}$$

To calculate the expected reward, we need to take average over both sensor readings weighted by their probabilities:

$$V' = \sum_j \left[\max_i R(a_i|o_j) \right] P(O_j)$$

$$
\begin{aligned}
&= \max(R(a_L|o_L), R(a_R|o_L), R(a_S|o_L))P(o_L) + \\
&\quad \max(R(a_L|o_R), R(a_R|o_R), R(a_S|o_R))P(o_R) \\
&= \max(-70p + 24(1-p), 63p - 30(1-p), -0.7p - 0.3(1-p)) + \\
&\quad \max(-30p + 56(1-p), 27p - 70(1-p), -0.3p - 0.7(1-p)) \\
&= \max \begin{pmatrix} -100p & +80(1-p) \\ -43p & -46(1-p) \\ 33p & +26(1-p) \\ 90p & -100(1-p) \end{pmatrix} \qquad (18.25)
\end{aligned}
$$

Note that when we multiply by $P(o_L)$, it cancels out and we get functions linear in p. These five lines and the piecewise function that corresponds to their maximum are shown in figure 18.10d. Note that the line, $-40p - 5(1-p)$, as well as the ones involving a_S, are beneath others for all values of p and can safely be pruned. The fact that figure 18.10d is better than figure 18.10a indicates the *value of information*.

VALUE OF INFORMATION

What we calculate here is the value of the best action had we chosen a_S. For example, the first line corresponds to choosing a_L after a_S. So to find the best decision with an episode of length two, we need to back this up by subtracting -1, which is the reward of a_S, and get the expected reward for the action of sense. Equivalently, we can consider this as waiting that has an immediate reward of 0 but discounts the future reward by some $\gamma < 1$. We also have the two usual actions of a_L and a_R and we choose the best of three; the two immediate actions and the one discounted future action.

Let us now make the problem more interesting, as in the example of Thrun, Burgard, and Fox 2005. Let us assume that there is a door between the two rooms and without us seeing, the tiger can move from one room to the other. Let us say that this is a restless tiger and it stays in the same room with probability 0.2 and moves to the other room with probability 0.8. This means that p should also be updated as

$$p' = 0.2p + 0.8(1-p)$$

and this updated p should be used in equation 18.25 while choosing the best action after having chosen a_S:

$$V' = \max \begin{pmatrix} -100p' & +80(1-p') \\ 33p' & +26(1-p') \\ 90p' & -100(1-p') \end{pmatrix}$$

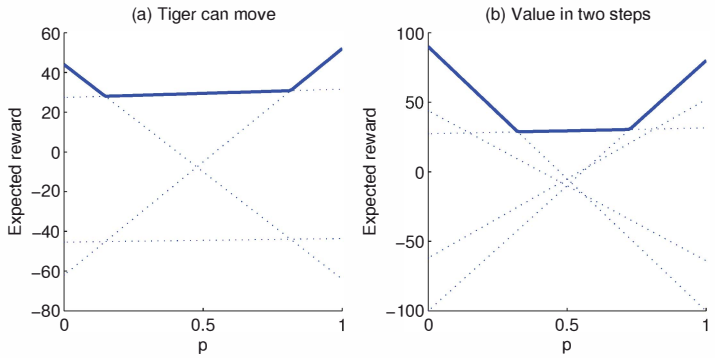

Figure 18.11 Expected rewards change (a) if the hidden state can change, and (b) when we consider episodes of length two.

Figure 18.11b corresponds to figure 18.10d with the updated p'. Now, when planning for episodes of length two, we have the two immediate actions of a_L and a_R, or we wait and sense when p changes and then we take the action and get its discounted reward (figure 18.11b):

$$V_2 = \max \begin{pmatrix} -100p & +80(1-p) \\ 90p & -100(1-p) \\ \max V' - 1 & \end{pmatrix}$$

We see that figure 18.11b is better than figure 18.10a; when wrong actions may lead to large penalty, it is better to defer judgment, look for extra information, and plan ahead. We can consider longer episodes by continuing the iterative updating of p and discounting by subtracting 1 and including the two immediate actions to calculate $V_t, t > 2$.

The algorithm we have just discussed where the value is represented by piecewise linear functions works only when the number of states, actions, observations, and the episode length are all finite. Even in applications where any of these is not small, or when any is continuous-valued, the complexity becomes high and we need to resort to approximate algorithms having reasonable complexity. Reviews of such algorithms are given in Hauskrecht 2000 and Thrun, Burgard, and Fox 2005.

18.8 Notes

More information on reinforcement learning can be found in the textbook by Sutton and Barto (1998) that discusses all the aspects, learning algorithms, and several applications. A comprehensive tutorial is Kaelbling, Littman, and Moore 1996. Recent work on reinforcement learning applied to robotics with some impressive applications is given in Thrun, Burgard, and Fox 2005.

Dynamic programming methods are discussed in Bertsekas 1987 and in Bertsekas and Tsitsiklis 1996, and TD(λ) and Q-learning can be seen as stochastic approximations to dynamic programming (Jaakkola, Jordan, and Singh 1994). Reinforcement learning has two advantages over classical dynamic programming: First, as they learn, they can focus on the parts of the space that are important and ignore the rest; and second, they can employ function approximation methods to represent knowledge that allows them to generalize and learn faster.

LEARNING AUTOMATA

A related field is that of *learning automata* (Narendra and Thathachar 1974), which are finite state machines that learn by trial and error for solving problems like the K-armed bandit. The setting we have here is also the topic of optimal control where there is a controller (agent) taking actions in a plant (environment) that minimize cost (maximize reward).

The earliest use of temporal difference method was in Samuel's checkers program written in 1959 (Sutton and Barto 1998). For every two successive positions in a game, the two board states are evaluated by the board evaluation function that then causes an update to decrease the difference. There has been much work on games because games are both easily defined and challenging. A game like chess can easily be simulated: The allowed moves are formal, and the goal is well defined. Despite the simplicity of defining the game, expert play is quite difficult.

TD-GAMMON

One of the most impressive application of reinforcement learning is the *TD-Gammon* program that learns to play backgammon by playing against itself (Tesauro 1995). This program is superior to the previous neurogammon program also developed by Tesauro, which was trained in a supervised manner based on plays by experts. Backgammon is a complex task with approximately 10^{20} states, and there is randomness due to the roll of dice. Using the TD(λ) algorithm, the program achieves master level play after playing 1,500,000 games against a copy of itself.

Another interesting application is in *job shop scheduling*, or finding a schedule of tasks satisfying temporal and resource constraints (Zhang

and Dietterich 1996). Some tasks have to be finished before others can be started, and two tasks requiring the same resource cannot be done simultaneously. Zhang and Dietterich used reinforcement learning to quickly find schedules that satisfy the constraints and are short. Each state is one schedule, actions are schedule modifications, and the program finds not only one good schedule but a schedule for a class of related scheduling problems.

Recently hierarchical methods have also been proposed where the problem is decomposed into a set of subproblems. This has the advantage that policies learned for the subproblems can be shared for multiple problems, which accelerates learning a new problem (Dietterich 2000). Each subproblem is simpler and learning them separately is faster. The disadvantage is that when they are combined, the policy may be suboptimal.

Though reinforcement learning algorithms are slower than supervised learning algorithms, it is clear that they have a wider variety of application and have the potential to construct better learning machines (Ballard 1997). They do not need any supervision, and this may actually be better since then they are not biased by the teacher. For example, Tesauro's TD-Gammon program in certain circumstances came up with moves that turned out to be superior to those made by the best players. The field of reinforcement learning is developing rapidly, and we may expect to see other impressive results in the near future.

18.9 Exercises

1. Given the grid world in figure 18.12, if the reward on reaching on the goal is 100 and $\gamma = 0.9$, calculate manually $Q^*(s,a)$, $V^*(S)$, and the actions of optimal policy.
2. With the same configuration given in exercise 1, use Q learning to learn the optimal policy.
3. In exercise 1, how does the optimal policy change if another goal state is added to the lower-right corner? What happens if a state of reward -100 (a very bad state) is defined in the lower-right corner?
4. Instead of having $\gamma < 1$, we can have $\gamma = 1$ but with a negative reward of $-c$ for all intermediate (nongoal) states. What is the difference?
5. In exercise 1, assume that the reward on arrival to the goal state is normal distributed with mean 100 and variance 40. Assume also that the actions are

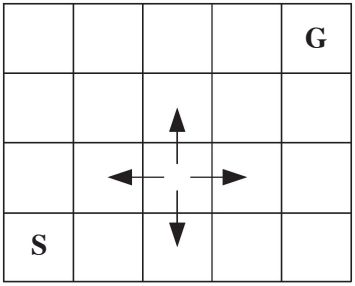

Figure 18.12 The grid world. The agent can move in the four compass directions starting from *S*. The goal state is *G*.

also stochastic in that when the robot advances in a direction, it moves in the intended direction with probability 0.5 and there is a 0.25 probability that it moves in one of the lateral directions. Learn $Q(s,a)$ in this case.

6. Assume we are estimating the value function for states $V(s)$ and that we want to use TD(λ) algorithm. Derive the tabular value iteration update.

   SOLUTION: The temporal error at time t is

   $$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

   All state values are updated as

   $$V(s) \leftarrow V(s) + \eta \delta_t e_t(s), \ \forall s$$

   where the eligibility of states decay in time:

   $$e_t(s) = \begin{cases} 1 & \text{if } s = s_t \\ \gamma \lambda e_{t-1}(s) & \text{otherwise} \end{cases}$$

7. Using equation 18.22, derive the weight update equations when a multilayer perceptron is used to estimate Q.

   SOLUTION: Let us say for simplicity we have one-dimensional state value s_t and one-dimensional action value a_t, and let us assume a linear model:

   $$Q(s,a) = w_1 s + w_2 a + w_3$$

   We can update the three parameters w_1, w_2, w_3 using gradient descent (equation 16.21):

   $$\begin{aligned} \Delta w_1 &= \eta \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right] s_t \\ \Delta w_2 &= \eta \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right] a_t \\ \Delta w_3 &= \eta \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right] \end{aligned}$$

In the case of a multilayer perceptron, only the last term will differ to update the weights on all layers.

In the case of Sarsa(λ), $\boldsymbol{e}$ is three-dimensional: e^1 for w_1, e^2 for w_2, and e^3 for w_0. We update the eligibilities (equation 18.23):

$$\begin{aligned} e_t^1 &= \gamma\lambda e_{t-1}^1 + s_t \\ e_t^2 &= \gamma\lambda e_{t-1}^2 + a_t \\ e_t^3 &= \gamma\lambda e_{t-1}^3 \end{aligned}$$

and we update the weights using the eligibilities (equation 18.22):

$$\begin{aligned} \Delta w_1 &= \eta\left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right] e_t^1 \\ \Delta w_2 &= \eta\left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right] e_t^2 \\ \Delta w_3 &= \eta\left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right] e_t^3 \end{aligned}$$

8. Give an example of a reinforcement learning application that can be modeled by a POMDP. Define the states, actions, observations, and reward.
9. In the tiger example, show that as we get a more reliable sensor, the range where we need to sense once again decreases.
10. Rework the tiger example using the following reward matrix

$r(A, Z)$	Tiger left	Tiger right
Open left	−100	+10
Open right	20	−100

18.10 References

Ballard, D. H. 1997. *An Introduction to Natural Computation*. Cambridge, MA: MIT Press.

Bellman, R. E. 1957. *Dynamic Programming*. Princeton: Princeton University Press.

Bertsekas, D. P. 1987. *Dynamic Programming: Deterministic and Stochastic Models*. New York: Prentice Hall.

Bertsekas, D. P., and J. N. Tsitsiklis. 1996. *Neuro-Dynamic Programming*. Belmont, MA: Athena Scientific.

Dietterich, T. G. 2000. "Hierarchical Reinforcement Learning with the MAXQ Value Decomposition." *Journal of Artificial Intelligence Research* 13:227–303.

Hauskrecht, M. 2000. "Value-Function Approximations for Partially Observable Markov Decision Processes." *Journal of Artificial Intelligence Research* 13:33–94.

Jaakkola, T., M. I. Jordan, and S. P. Singh. 1994. "On the Convergence of Stochastic Iterative Dynamic Programming Algorithms." *Neural Computation* 6:1185–1201.

Kaelbling, L. P., M. L. Littman, and A. R. Cassandra. 1998. "Planning and Acting in Partially Observable Stochastic Domains." *Artificial Intelligence* 101:99–134.

Kaelbling, L. P., M. L. Littman, and A. W. Moore. 1996. "Reinforcement Learning: A Survey." *Journal of Artificial Intelligence Research* 4:237–285.

Narendra, K. S., and M. A. L. Thathachar. 1974. "Learning Automata—A Survey." *IEEE Transactions on Systems, Man, and Cybernetics* 4:323–334.

Sutton, R. S. 1988. "Learning to Predict by the Method of Temporal Differences." *Machine Learning* 3:9–44.

Sutton, R. S., and A. G. Barto. 1998. *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press.

Tesauro, G. 1995. "Temporal Difference Learning and TD-Gammon." *Communications of the ACM* 38 (3): 58–68.

Thrun, S., W. Burgard, and D. Fox. 2005. *Probabilistic Robotics*. Cambridge, MA: MIT Press.

Watkins, C. J. C. H., and P. Dayan. 1992. "Q-learning." *Machine Learning* 8:279–292.

Zhang, W., and T. G. Dietterich. 1996. "High-Performance Job-Shop Scheduling with a Time-Delay TD(λ) Network." In *Advances in Neural Information Processing Systems 8*, ed. D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, 1024–1030. Cambridge, MA: The MIT Press.