5

Model Uncertainty

Mykel J. Kochenderfer

The previous chapter discussed sequential decision problems with a known transition and reward model. In many problems, the dynamics and rewards are not known exactly, and the agent must learn to act through experience. By observing the outcomes of its actions in the form of state transitions and rewards, the agent is to choose actions that maximize its long-term accumulation of rewards. Solving such problems in which there is model uncertainty is the subject of the field of *reinforcement learning* and the focus of this chapter. Several challenges in addressing model uncertainty will be discussed. First, the agent must carefully balance exploration of the environment with the exploitation of that knowledge gained through experience. Second, rewards may be received long after the important decisions have been made, so credit for later rewards must be assigned to earlier decisions. Third, the agent must generalize from limited experience. This chapter will review the theory and some of the key algorithms for addressing these challenges.

5.1 Exploration and Exploitation

Reinforcement learning problems require us to carefully balance *exploration* of the environment with *exploitation* of knowledge obtained through evaluative feedback. If we continuously explore our environment, then we may be able to build a comprehensive model, but we will not be able to accumulate much reward. If we continuously make the decision we believe is best without ever trying a new strategy, then we may miss out on improving our strategy and accumulating more reward. This section introduces the challenges of balancing exploration with exploitation on problems with a single state.

5.1.1 Multi-Armed Bandit Problems

Some of the earliest studies of balancing exploration with exploitation were focused on slot machines—sometimes referred to as *one-armed bandits* because they are often controlled by a single lever and, on average, they take away gamblers' money. Bandit problems appear in a wide variety of applications, such as the allocation of clinical trials and adaptive network routing. They were originally formulated during World War II and proved exceptionally challenging to solve. According to Peter Whittle, "efforts to solve [bandit problems] so sapped the energies and minds of Allied analysts that the suggestion was made that the problem be dropped over Germany as the ultimate instrument of intellectual sabotage" (see comment following article [1]).

Many different formulations of bandit problems are presented in the literature, but we will focus on a simple one involving a slot machine with n arms. Arm i pays off 1 with probability θ_i and 0 with probability $1-\theta_i$. There is no deposit to play, but we are limited to h pulls. We can view this problem as an h-step finite horizon Markov decision process (Chapter 4) with a single state, n actions, and an unknown reward function $R(s,a)$.

5.1.2 Bayesian Model Estimation

We can use the beta distribution introduced in Section 2.3.2 to represent our posterior over the win probability θ_i for arm i, and we will use the uniform prior distribution, which corresponds to Beta$(1,1)$. We just have to keep track of the number of wins w_i and the number of losses ℓ_i for each arm i. The posterior for θ_i is given by Beta(w_i+1,ℓ_i+1). We can then compute the posterior probability of winning:

$$\rho_i = P(\text{win}_i \mid w_i, \ell_i) = \int_0^1 \theta \times \text{Beta}(\theta \mid w_i+1, \ell_i+1)\, d\theta = \frac{w_i+1}{w_i+\ell_i+2}. \tag{5.1}$$

For example, suppose we have a two-armed bandit that we have pulled six times. The first arm had 1 win and 0 losses, and the second arm has 4 wins and 1 loss. Assuming a uniform prior, the posterior distribution for θ_1 is given by Beta$(2,1)$, and the posterior distribution for θ_2 is given by Beta$(5,2)$. The posteriors are plotted in Figure 5.1.

The maximum likelihood estimate for θ_1 is 1 and the maximum likelihood estimate for θ_2 is 4/5. If we were to choose the next pull solely on the basis of the maximum-likelihood estimate, then we would want to go with the first arm—with a guaranteed win! Of course, just because we have not yet observed a loss with the first arm does not mean that a loss is impossible.

In contrast with the maximum likelihood estimate of the payoff probabilities, the Bayesian posterior shown in Figure 5.1 assigns non-zero probability to probabilities between 0 and 1. The density at 0 for both arms is 0 because at least one win was observed from both arms. There is also zero density at $\theta_2 = 1$ because a loss was observed. Using Equation (5.1), we can compute the payoff probabilities:

$$\rho_1 = 2/3 = 0.67 \tag{5.2}$$

$$\rho_2 = 5/7 = 0.71. \tag{5.3}$$

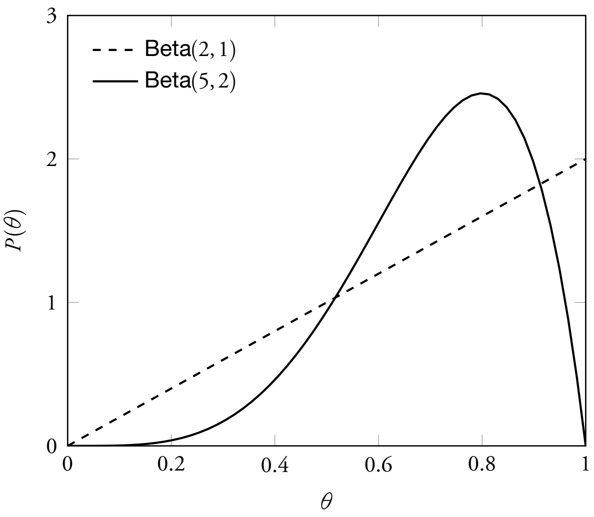

Figure 5.1 Posterior distribution of payoff probabilities.

Hence, if we assume that we only have one pull remaining, it is best to pull the second arm.

5.1.3 Ad Hoc Exploration Strategies

Several different ad hoc exploration strategies have been suggested in the literature. In one of the most common strategies, ϵ-greedy, we choose a random arm with probability ϵ; otherwise, we choose $\arg\max_i \rho_i$. Larger values of ϵ allow us to more quickly identify the best arm, but more pulls are wasted on suboptimal arms.

Directed exploration strategies involve using information gathered from previous pulls. For example, the *softmax* strategy is to pull arms according to the logit model (introduced in Section 3.3.3), where arm i is selected with probability proportional to $\exp(\lambda\rho_i)$. The precision parameter $\lambda \geq 0$ controls the amount of exploration, with uniform random selection as $\lambda \to 0$ and greedy selection as $\lambda \to \infty$. Another approach is to use *interval exploration*, by which we compute the α% confidence interval for θ_i and choose the arm with the highest upper bound. Larger values for α result in more exploration.

5.1.4 Optimal Exploration Strategies

The counts $w_1, \ell_1, \ldots, w_n, \ell_n$ represent a *belief state*, which summarizes our belief about payoffs. As discussed in Section 5.1.2, these $2n$ numbers can be used to represent n

continuous probability distributions over possible values for $\rho_{1:n}$. These belief states can be used as states in an MDP that represents the n-armed bandit problem. We can use dynamic programming to determine an optimal policy π^*, which specifies which arm to pull given the counts.

We use $Q^*(w_{1:n}, \ell_{1:n}, i)$ to represent the expected payoff after pulling arm i and then acting optimally. The optimal utility function and policy are given in terms of Q^*:

$$U^*(w_1, \ell_1, \ldots, w_n, \ell_n) = \max_i Q^*(w_1, \ell_1, \ldots, w_n, \ell_n, i) \tag{5.4}$$

$$\pi^*(w_1, \ell_1, \ldots, w_n, \ell_n) = \arg\max_i Q^*(w_1, \ell_1, \ldots, w_n, \ell_n, i). \tag{5.5}$$

We can decompose Q^* into two terms:

$$\begin{aligned} Q^*(w_1, \ell_1, \ldots, w_n, \ell_n, i) = & \frac{w_i + 1}{w_i + \ell_i + 2}\left(1 + U^*(\ldots, w_i + 1, \ell_i, \ldots)\right) \\ & + \left(1 - \frac{w_i + 1}{w_i + \ell_i + 2}\right) U^*(\ldots, w_i, \ell_i + 1, \ldots). \end{aligned} \tag{5.6}$$

The first term is associated with a win for arm i, and the second term is associated with a loss. The value $(w_i + 1)/(w_i + \ell_i + 2)$ is the posterior probability of a win, which comes from Equation (5.1). The first U^* in the equation above assumes that pulling arm i brought a win, and the second U^* assumes a loss.

Assuming a horizon h, we can compute Q^* for the entire belief space. We start with the belief states with $\sum_i (w_i + \ell_i) = h$. With no pulls left, $U^*(w_1, \ell_1, \ldots, w_n, \ell_n) = 0$. We can then work backward to the states where $\sum_i (w_i + \ell_i) = h - 1$ and apply Equation (5.6).

Although this dynamic programming solution is optimal, the number of belief states—and consequently the amount of computation and memory required—is exponential in h. We can formulate an infinite horizon, discounted version of the problem that can be solved efficiently using the *Gittins allocation index*. The allocation index can be stored as a lookup table that specifies a scalar allocation index value given the number of pulls and the number of wins associated with an arm. The arm that has the highest allocation index is the one that should be pulled next.

5.2 Maximum Likelihood Model-Based Methods

A variety of reinforcement learning methods have been proposed for addressing problems with multiple states. Solving problems with multiple states is more challenging than bandit problems because we need to plan to visit states to determine their value. One approach to reinforcement learning involves estimating the transition and reward models directly from experience. We keep track of the counts of transitions $N(s, a, s')$ and the sum of rewards $\rho(s,a)$. The maximum likelihood estimates of the transition and reward models are as follows:

$$N(s,a) = \sum_{s'} N(s,a,s') \tag{5.7}$$

$$T(s' \mid s,a) = N(s,a,s')/N(s,a) \tag{5.8}$$

$$R(s,a) = \rho(s,a)/N(s,a). \tag{5.9}$$

If we have prior knowledge about the transition probabilities or the rewards, then we can initialize $N(s,a,s')$ and $\rho(s,a)$ to values other than 0.

We can solve the MDP assuming the estimated models are correct. Of course, we have to incorporate some exploration strategy, such as those mentioned in Section 5.1.3, in order to ensure that we converge to an optimal strategy. The basic structure of maximum likelihood model-based reinforcement learning is outlined in Algorithm 5.1.

Algorithm 5.1 Maximum likelihood model-based reinforcement learning

```
 1: function MaximumLikelihoodModelBasedReinforcementLearning
 2:     $t \leftarrow 0$
 3:     $s_0 \leftarrow$ initial state
 4:     Initialize $N$, $\rho$, and $Q$
 5:     loop
 6:         Choose action $a_t$ based on some exploration strategy
 7:         Observe new state $s_{t+1}$ and reward $r_t$
 8:         $N(s_t, a_t, s_{t+1}) \leftarrow N(s_t, a_t, s_{t+1}) + 1$
 9:         $\rho(s_t, a_t) \leftarrow \rho(s_t, a_t) + r_t$
10:         Update $Q$ based on revised estimate of $T$ and $R$
11:         $t \leftarrow t + 1$
```

5.2.1 Randomized Updates

Although we can use any dynamic programming algorithm to update Q in Line 10 of Algorithm 5.1, the computational expense is often not necessary. One algorithm that avoids solving the entire MDP at each time step is *Dyna*. Dyna performs the following update at the current state:

$$Q(s,a) \leftarrow R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) \max_{a'} Q(s',a'). \tag{5.10}$$

Here, R and T are the estimated reward and transition functions. We then perform some number of additional updates of Q for random states and actions depending on how much time is available between decisions. Following the updates, we use Q to choose which action to execute—perhaps using softmax or one of the other exploration strategies.

5.2.2 Prioritized Updates

An approach known as *prioritized sweeping* uses a priority queue to help identify which states require updating Q the most (Algorithm 5.2). If we transition from s to s', then we update $U(s)$ based on our updated transition and reward models. We then iterate over the predecessor set, $\text{pred}(s) = \{(s', a') \mid T(s \mid s', a') > 0\}$, the set of all state-action pairs leading immediately to state s. The priority of s' is increased to $T(s \mid s', a') \times |U(s) - u|$, where u was the value of $U(s)$ prior to the update. Hence, the larger the change in $U(s)$, the higher the priority of the states leading to s. The process of updating the highest priority state in the queue continues for some fixed number of iterations or until the queue becomes empty.

Algorithm 5.2 Prioritized sweeping

```
 1: function PrioritizedSweeping($s$)
 2:     Increase the priority of $s$ to $\infty$
 3:     while priority queue is not empty
 4:         $s \leftarrow$ highest priority state
 5:         Update($s$)
 6: function Update($s$)
 7:     $u \leftarrow U(s)$
 8:     $U(s) \leftarrow \max_a [R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U(s')]$
 9:     for $(s', a') \in \text{pred}(s)$
10:         $p \leftarrow T(s \mid s', a') \times |U(s) - u|$
11:         Increase priority of $s'$ to $p$
```

5.3 Bayesian Model-Based Methods

The previous section used maximum likelihood estimates of the transition probabilities and rewards and then relied on a heuristic exploration strategy to converge on an optimal strategy in the limit. Bayesian methods, in contrast, allow us to optimally balance exploration with exploitation without having to rely on heuristics. This section describes a generalization of the formulation for multi-armed bandit problems (discussed in Section 5.1.4) applied to general MDPs.

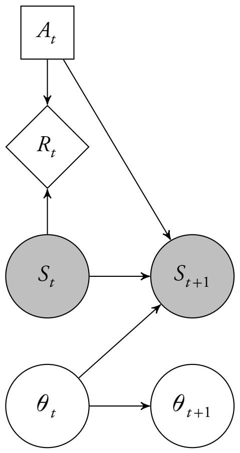

Figure 5.2 Markov decision process with model uncertainty.

5.3.1 Problem Structure

In Bayesian reinforcement learning, we specify a prior distribution over all the model parameters θ. These model parameters may include the parameters governing the distribution over immediate rewards, but we will focus on the parameters governing the state transition probabilities. If $\mathcal{S}$ represents the state space and $\mathcal{A}$ represents the action space, then the parameter vector θ consists of $|\mathcal{S}|^2|\mathcal{A}|$ components representing every possible transition probability. The component of θ that governs the transition probability $T(s' \mid s, a)$ is denoted $\theta_{(s,a,s')}$.

The structure of the problem can be represented using the dynamic decision network shown in Figure 5.2, which is an extension of the network shown in Figure 4.1b with the model parameters made explicit. As indicated by the shaded nodes, the states are observed, but the model parameters are not. We generally assume that the model parameters are time invariant, and so $\theta_{t+1} = \theta_t$. However, our belief about θ evolves with time as we transition to new states.

5.3.2 Beliefs over Model Parameters

We want to represent a prior belief over θ, and a natural way to do this for discrete state spaces is with a product of Dirichlet distributions. Each Dirichlet would represent a distribution over the next state given the current state s and action a. If $\theta_{(s,a)}$ is an $|\mathcal{S}|$-element vector representing the distribution over the next state, then the prior distribution is given by

$$\mathrm{Dir}(\boldsymbol{\theta}_{(s,a)} \mid \boldsymbol{\alpha}_{(s,a)}). \tag{5.11}$$

The Dirichlet distribution above is governed by the $|\mathcal{S}|$ parameters in $\boldsymbol{\alpha}_{(s,a)}$. It is common to use a uniform prior with all the components of $\boldsymbol{\alpha}_{(s,a)}$ set to 1, but if we have prior knowledge about the dynamics, then we can set these parameters differently, as discussed in Section 2.3.2.

The prior distribution over θ is given by the product

$$b_0(\theta) = \prod_s \prod_a \text{Dir}(\theta_{(s,a)} \mid \boldsymbol{\alpha}_{(s,a)}). \tag{5.12}$$

The factorization shown above is often used for small discrete state spaces, but other lower dimensional parametric representations may be desirable.

The posterior distribution over θ after t steps is denoted b_t. Suppose that during the first t steps, we observe $m_{(s,a,s')}$ transitions from s to s' by action a. We compute the posterior using Bayes' rule. If $\mathbf{m}_{(s,a)}$ represents a vector of transition counts, then the posterior is given by

$$b_t(\theta) = \prod_s \prod_a \text{Dir}(\theta_{(s,a)} \mid \boldsymbol{\alpha}_{(s,a)} + \mathbf{m}_{(s,a)}). \tag{5.13}$$

5.3.3 Bayes-Adaptive Markov Decision Processes

We can formulate the problem of acting optimally in an MDP with an *unknown* model as a higher dimensional MDP with a *known* model. This higher dimensional MDP is known as a *Bayes-adaptive Markov decision process*, which is related to the partially observable Markov decision process discussed in the next chapter.

The state space in a Bayes-adaptive MDP is the Cartesian product $\mathcal{S} \times \mathcal{B}$, where $\mathcal{B}$ is the space of all possible beliefs over the model parameters θ. Although $\mathcal{S}$ is discrete, $\mathcal{B}$ is often a high-dimensional continuous space. A state in a Bayes-adaptive MDP is written as a pair (s, b) consisting of the state s of the base MDP and the belief state b. The action space and reward function are exactly the same as for the base MDP.

The transition function in a Bayes-adaptive MDP is $T(s', b' \mid s, b, a)$, which is the probability of transitioning to some new state s' with a new belief state b', given that you start in state s with belief b and execute action a. The new belief state b' is a deterministic function of s, b, a, and s' as computed by Bayes' rule in Section 5.3.2. Let us denote this deterministic function τ so that $b' = \tau(s, b, a, s')$. The Bayes-adaptive MDP transition function can be decomposed as follows:

$$T(s', b' \mid s, b, a) = \delta_{\tau(s,b,a,s')}(b') P(s' \mid s, b, a), \tag{5.14}$$

where $\delta_x(y)$ is the *Kronecker delta* function such that

$$\delta_x(y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}. \tag{5.15}$$

Computing $P(s' \mid s, b, a)$ requires integration:

$$P(s' \mid s, b, a) = \int_\theta b(\theta) P(s' \mid s, \theta, a)\, d\theta = \int_\theta b(\theta) \theta_{(s,a,s')}\, d\theta. \tag{5.16}$$

Similar to Equation (5.1), the integral above can be evaluated analytically.

5.3.4 Solution Methods

We can generalize the Bellman equation from Section 4.2.4 for MDPs with a known model to the case in which the model is unknown:

$$U^*(s, b) = \max_a \left(R(s, a) + \gamma \sum_{s'} P(s' \mid s, b, a) U^*(s', \tau(s, b, a, s')) \right). \tag{5.17}$$

Unfortunately, we cannot simply use the policy iteration and value iteration algorithms directly as presented in Chapter 4 because b is continuous. However, we can use the approximations in Section 4.5 as well as the online methods in Section 4.6. Methods that better leverage the structure of the Bayes-adaptive MDP will be presented in the next chapter.

An alternative to solving for the optimal value function over the belief space is to use a technique known as *Thompson sampling*. The idea here is to draw a sample θ from the current belief b_t and then assume θ is the true model. We use dynamic programming to solve for the best action. At the next time step, we update our belief, draw a new sample, and resolve the MDP. The advantage of this approach is that we do not have to decide on heuristic exploration parameters. However, Thompson sampling has been shown to over-explore, and resolving the MDP at every step can be expensive.

5.4 Model-Free Methods

In contrast to model-based methods, model-free reinforcement learning does not require building explicit representations of the transition and reward models. Avoiding explicit representations is attractive, especially when the problem is high dimensional.

5.4.1 Incremental Estimation

Many model-free methods involve incremental estimation of the expected discounted return from the various states in the problem. Suppose we have a random variable X and want to estimate the mean from a set of samples $x_{1:n}$. After n samples, we have the estimate:

$$\hat{x}_n = \frac{1}{n} \sum_{i=1}^{n} x_i. \tag{5.18}$$

We can show that

$$\hat{x}_n = \hat{x}_{n-1} + \frac{1}{n}(x_n - \hat{x}_{n-1}) \tag{5.19}$$
$$= \hat{x}_{n-1} + \alpha(n)(x_n - \hat{x}_{n-1}). \tag{5.20}$$

The function $\alpha(n)$ is referred to as the *learning rate*. The learning rate can be a function other than $1/n$; there are rather loose conditions on the learning rate to ensure convergence to the mean. If the learning rate is constant, which is common in reinforcement learning applications, then the weights of older samples decay exponentially at the rate $(1-\alpha)$. With a constant learning rate, we can update our estimate after observing x using the following rule:

$$\hat{x} \leftarrow \hat{x} + \alpha(x - \hat{x}). \tag{5.21}$$

The update rule above will appear again in later sections and is related to stochastic gradient descent. The magnitude of the update is proportional to the difference in the sample and the previous estimate. The difference between the sample and previous estimate is called the *temporal difference error*.

5.4.2 Q-Learning

One of the most popular model-free reinforcement learning algorithms is *Q-learning*. The idea is to apply incremental estimation to the Bellman equation

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U(s') \tag{5.22}$$
$$= R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) \max_{a'} Q(s',a'). \tag{5.23}$$

Instead of using T and R, we use the observed next state s' and reward r to obtain the following incremental update rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a)). \tag{5.24}$$

Q-learning is outlined in Algorithm 5.3. As with the model-based methods, some exploration strategy is required to ensure that Q converges to the optimal state-action value function. We can initialize Q to values other than 0 to encode any prior knowledge we may have about the environment.

Algorithm 5.3 Q-learning

```
1: function QLearning
2:     $t \leftarrow 0$
3:     $s_0 \leftarrow$ initial state
4:     Initialize $Q$
5:     loop
6:         Choose action $a_t$ based on $Q$ and some exploration strategy
7:         Observe new state $s_{t+1}$ and reward $r_t$
8:         $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$
9:         $t \leftarrow t + 1$
```

5.4.3 Sarsa

An alternative to Q-learning is *Sarsa*, which derives its name from the fact that it uses $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$ to update the Q function at each step. It uses the actual action taken to update Q instead of maximizing over all possible actions as done in Q-learning. The Sarsa algorithm is identical to Algorithm 5.3 except that Line 8 is replaced with

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)). \tag{5.25}$$

With a suitable exploration strategy, a_{t+1} will converge to the $\arg\max_a Q(s_{t+1}, a)$ that is used to update the Q function in the Q-learning algorithm. Although Q-learning and Sarsa both converge to the optimal strategy, the speed of the convergence depends on the application.

5.4.4 Eligibility Traces

One of the disadvantages of Q-learning and Sarsa is that learning can be very slow. For example, suppose the environment has a single goal state that provides a large reward. The reward is zero at all other states. After some amount of random exploration in the environment, we reach the goal state. Regardless of whether we use Q-learning or Sarsa, we only update the state-action value of the state immediately preceding the goal state. The values at all other states leading up to the goal remain at zero. Much more exploration is required to slowly propagate non-zero values to the remainder of the state space.

Q-learning and Sarsa can be modified to assign credit to achieving the goal to past states and actions using *eligibility traces*. The reward associated with reaching the goal is propagated backward to the states and actions leading up to the goal. The credit is decayed exponentially, so states closer to the goal are assigned larger state-action values. It is common to use λ as the exponential decay parameter, and so the versions of Q-learning and Sarsa with eligibility traces are often called Q(λ) and Sarsa(λ).

Algorithm 5.4 shows a version of Sarsa(λ). We keep track of an exponentially decaying visit count $N(s, a)$ for all the state-action pairs. When we take action a_t in state s_t, $N(s_t, a_t)$ is incremented by 1. We then update $Q(s, a)$ by adding $\alpha\delta N(s, a)$ at every state s and for every action a, where

$$\delta = r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t). \tag{5.26}$$

After performing the updates, we decay $N(s, a) \leftarrow \gamma\lambda N(s, a)$. Although the impact of eligibility traces is especially pronounced in environments with sparse reward, the algorithm can speed learning in general environments where reward is more evenly distributed.

Algorithm 5.4 Sarsa(λ)-learning

```
1:  function SarsaLambdaLearning($\lambda$)
2:      Initialize $Q$ and $N$
3:      $t \leftarrow 0$
4:      $s_0, a_0 \leftarrow$ initial state and action
5:      loop
6:          Observe reward $r_t$ and new state $s_{t+1}$
7:          Choose action $a_{t+1}$ based on some exploration strategy
8:          $N(s_t, a_t) \leftarrow N(s_t, a_t) + 1$
9:          $\delta \leftarrow r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$
10:         for $s \in S$
11:             for $a \in A$
12:                 $Q(s, a) \leftarrow Q(s, a) + \alpha\delta N(s, a)$
13:                 $N(s, a) \leftarrow \gamma\lambda N(s, a)$
14:         $t \leftarrow t + 1$
```

5.5 Generalization

Up to this point in this chapter, we have assumed that the state-action value function can be represented as a table, which is only useful for small discrete problems. The problem with larger state spaces is not just the size of the state-action table but also the amount of experience required to accurately estimate the values. The agent must *generalize* from limited experience to states that have not yet been visited. Many different approaches have been explored, many related to techniques for approximate dynamic programming Section 4.5.

5.5.1 Local Approximation

The assumption in local approximation methods is that states that are close together are likely to have similar state-action values. A common technique is to store estimates of $Q(s,a)$ at a limited number of states in set S and actions in set A. We denote the vector containing these estimates as θ, which has $|S| \times |A|$ elements. To denote the component associated with state s and action a, we use $\theta_{s,a}$. If we use a weighting function such that $\sum_{s'} \beta(s,s') = 1$ for all s, then we can approximate the state action value at arbitrary states as

$$Q(s,a) = \sum_{s'} \theta_{s',a} \beta(s,s'). \tag{5.27}$$

We can define a vectorized version of the weighting function as follows:

$$\beta(s) = (\beta(s,s_1), \ldots, \beta(s,s_{|S|})), \tag{5.28}$$

where $s_1, \ldots, s_{|S|}$ are the states in S. We can also define a two-argument version of the function β that takes as input both a state and an action and returns a vector with $|S| \times |A|$ elements. The vector $\beta(s,a)$ is identical to $\beta(s)$, except that the elements associated with actions other than a are set to 0. This notation allows us to rewrite Equation (5.27) as follows:

$$Q(s,a) = \theta^\top \beta(s,a). \tag{5.29}$$

The linear approximation of Equation (5.29) can be easily integrated into Q-learning. The state-action value estimates represented by θ are updated as follows based on observing a state transition from s_t to s_{t+1} by action a_t with reward r_t:

$$\theta \leftarrow \theta + \alpha(r_t + \gamma \max_a \theta^\top \beta(s_{t+1},a) - \theta^\top \beta(s_t,a_t))\beta(s_t,a_t). \tag{5.30}$$

The update rule above comes from substituting Equation (5.29) directly into the standard Q-learning update rule and multiplying the last term by $\beta(s_t,a_t)$ to provide greater updates at states that are closer to s_t.

Algorithm 5.5 shows this linear approximation Q-learning method. If we have prior knowledge about the state-action values, then we can initialize θ appropriately. This linear approximation method can easily be extended to other reinforcement learning methods, such as Sarsa.

The algorithm presented above assumes that the points in S remain fixed. However, for some problems, it may be beneficial to adjust the locations of the points in S to produce a better approximation. The locations can be adjusted based on the temporal difference error using a representation such as a *self-organizing map* (see references in Section 5.7). Various criteria have been explored for identifying when it is appropriate to add new points to S, such as when a new state has been observed that is outside some

Algorithm 5.5 Linear approximation Q-learning

```
1: function LinearApproximationQLearning
2:     $t \leftarrow 0$
3:     $s_0 \leftarrow$ initial state
4:     Initialize $\theta$
5:     loop
6:         Choose action $a_t$ based on $\theta_a^\top \beta(s_t)$ and some exploration strategy
7:         Observe new state $s_{t+1}$ and reward $r_t$
8:         $\theta \leftarrow \theta + \alpha(r_t + \gamma \max_a \theta^\top \beta(s_{t+1}, a) - \theta^\top \beta(s_t, a_t))\beta(s_t, a_t)$
9:         $t \leftarrow t + 1$
```

threshold distance of the states in S. Although memory can become an issue, some methods simply store all observed states.

5.5.2 Global Approximation

Global approximation methods do not rely on a notion of distance. One such approximation method is a *perceptron*. Perceptrons have been widely used since at least the 1950s to mimic individual neurons for various learning tasks. A perceptron has a set of input nodes $x_{1:n}$, a set of weights $\theta_{1:n}$, and an output node q. The value of the output node is determined as follows:

$$q = \sum_{i=1}^{m} \theta_i x_i = \theta^\top \mathbf{x}. \tag{5.31}$$

The structure of a perceptron is shown in Figure 5.3a.

In *perceptron Q-learning*, we have a set of n perceptrons, one for each available action. The input is based on the state, and the output is the state-action value. We define a set of basis functions $\beta_1, \ldots, \beta_m$ over the state space, similar to the weighting functions in Section 5.5.1. The inputs to the perceptrons are $\beta_1(s), \ldots, \beta_m(s)$. If θ_a are the m weights associated with the perceptron for action a, then we have

$$Q(s, a) = \theta_a^\top \beta(s). \tag{5.32}$$

We can define a two-argument version of β as done in Section 5.5.1 and define θ to contain all the weights of all the perceptrons so that we can write

$$Q(s, a) = \theta^\top \beta(s, a). \tag{5.33}$$

Q-learning with a perceptron-based approximation follows Algorithm 5.5 exactly, but θ represents perceptron weights instead of value estimates and β represents basis functions instead of distance measures.

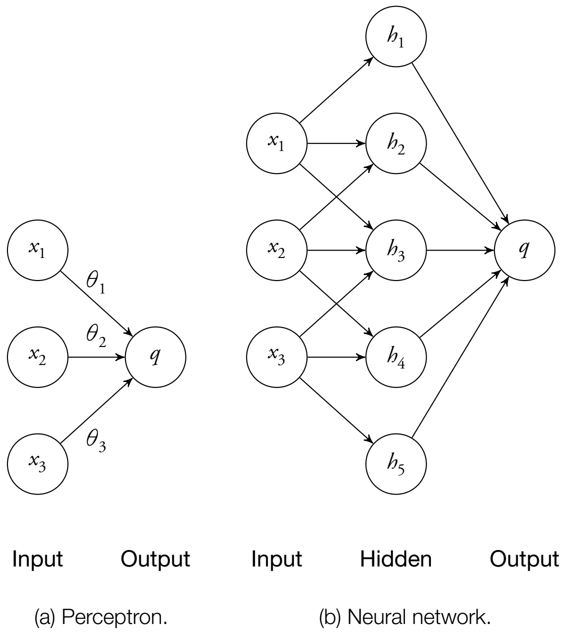

Figure 5.3 Approximation structures.

Perceptrons only represent linear functions, but *neural networks* can represent nonlinear functions. A neural network is a network of perceptrons. They are organized into an input layer, a hidden layer, and an output layer as shown in Figure 5.3b; the weights are omitted from the diagram. Adding hidden nodes generally increases the complexity of the state-action function the network can represent.

We can use an algorithm known as *backpropagation* to adjust the weights in the neural network to reduce the temporal difference error. The idea is to first adjust the weights associated with the edges leading from the hidden nodes to the output node in much the same way it is done for perceptron learning. We then compute the error associated with the hidden nodes and adjust the weights from the input nodes to the hidden nodes appropriately. Although convergence when using this form of function approximation is not guaranteed, it can result in satisfactory performance in a variety of domains.

5.5.3 Abstraction Methods

Abstraction methods involve partitioning the state space into discrete regions and estimating the state-action values for each of these regions. Abstraction methods tend to use model-based learning, which often results in faster convergence than model-free learning. Abstraction methods often use decision trees to partition the state space. Associated with the internal nodes of the tree are various tests along the various dimensions of the state space. The leaf nodes correspond to regions.

There are a variety of different abstraction methods, but one method is to start with a single region represented by a decision tree with a single node and then apply a series of acting, modeling, and planning phases. In the acting phase, we select an action based on the state-action value of the region associated with the current state s. After observing the transition from state s to s' by action a with reward r, the experience tuple (s, a, s', r) is stored in the leaf node associated with s.

In the modeling phase, we decide whether to split nodes. For each of the experience tuples at all of the leaf nodes, we compute

$$q(s,a) = r + \gamma U(s'), \tag{5.34}$$

where r is the observed reward and $U(s')$ is the value of the leaf node associated with the next state s'. We want to split a leaf node when the values of the experience tuples come from different distributions. One approach is to choose the split that minimizes the variance of the experience tuples at the resulting leaves. We stop splitting when some stopping criterion is met, such as when the variance at the leaf nodes is below some threshold.

In the planning phase, we use the experience tuples at the leaf nodes to estimate the state transition model and reward model. We then solve the resulting MDP using dynamic programming. The modeling and planning procedures require much more computation than what is required in the other generalization methods, but this approach can find better policies without as much interaction with the environment.

5.6 Summary

- Reinforcement learning is a computational approach to learning intelligent behavior from experience.
- Exploration must be carefully balanced with exploitation.
- In general, solving the exploration problem optimally is not feasible, but there are several Bayesian and heuristic approximation methods that can work well.
- It is important to determine how much actions in the past are responsible for later rewards.
- Model-based reinforcement learning involves building a model from experience and using this model to generate a plan.
- Model-free reinforcement learning involves directly estimating the values of states and actions without the use of transition and reward models.
- We must generalize from observations of rewards and state transitions because our interaction with the world is limited.
- Generalization can be done in a variety of ways, such as local and global approximations of the value function and state abstraction.

5.7 Further Reading

The classic book by Sutton and Barto, titled *Reinforcement Learning: An Introduction*, is the standard introductory text on classical reinforcement learning and provides a historical overview of the emergence of the field [2]. The volume edited by Wiering and Otterlo provides an up-to-date survey of much of the research that occurred since the publication of the Sutton and Barto book [3]. Kovacs and Egginton survey software for reinforcement learning [4].

Multi-armed bandit problems and their many variations have received considerable attention over the years [5]. Gittins developed the concept of an allocation index for solving multi-armed bandit problems [1]. Recent work has focused on improving the efficiency of computing allocation indices [6], [7].

Model-based reinforcement learning can be grouped into non-Bayesian and Bayesian methods [8]. The non-Bayesian methods generally rely on maximum likelihood estimation as discussed in Section 5.2. The Dyna approach was introduced by Sutton [9]. Prioritized sweeping was introduced by Moore and Atkeson [10].

Bayesian model-based methods have started to receive attention more recently [11]. Duff discusses the formulation of model-based reinforcement learning as a Bayes-adaptive Markov decision process [12]. In general, solving such a belief-state formulation exactly is intractable. Strens applies the concept of Thompson sampling [14] to model-based reinforcement learning [13]. Variants of the online planning algorithms presented in the previous chapter have been extended to Bayesian model-based reinforcement learning, including sparse sampling [15] and Monte Carlo tree search [16], [17].

Model-free reinforcement learning algorithms are often used for situations in which it is not feasible to build an explicit representation of the transition and reward models. Q-learning and Sarsa are two commonly used model-free techniques. Eligibility traces were proposed in the context of temporal difference learning by Sutton [18], and they were extended to Sarsa(λ) [19] and $Q(\lambda)$ [20], [21].

Much of the ongoing work in the field of reinforcement learning is concerned with generalizing from limited experience. The recent book *Reinforcement Learning and Dynamic Programming Using Function Approximators* by Busoniu et al. surveys a variety of different local and global function approximation methods [22]. Several different abstraction methods have been proposed over the years [23]–[26].

Although not discussed in this chapter, there has been some work on Bayesian approaches to model-free reinforcement learning. One approach is to maintain a distribution over state-action values [27], [28]. There are also Bayesian policy gradient methods that have been used with some success [29]. Multiagent reinforcement learning was also not discussed in this chapter, but Busoniu, Babuska, and De Schutter survey recent research in the area [30].

References

1. J.C. Gittins, "Bandit Processes and Dynamic Allocation Indices," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 41, no. 2, pp. 148–177, 1979.
2. R.S. Sutton and A.G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press, 1998.
3. M. Wiering and M. van Otterlo, eds., *Reinforcement Learning: State of the Art*. New York: Springer, 2012.
4. T. Kovacs and R. Egginton, "On the Analysis and Design of Software for Reinforcement Learning, with a Survey of Existing Systems," *Machine Learning*, vol. 84, no. 1-2, pp. 7–49, 2011. DOI: 10.1007/s10994-011-5237-8.
5. J. Gittins, K. Glazebrook, and R. Weber, *Multi-Armed Bandit Allocation Indices*, 2nd ed. Hoboken, NJ: Wiley, 2011.

6. J. Niño-Mora, "A $(2/3)^n$ Fast-Pivoting Algorithm for the Gittins Index and Optimal Stopping of a Markov Chain," *INFORMS Journal on Computing*, vol. 19, no. 4, pp. 596–606, 2007. DOI: 10.1287/ijoc.1060.0206.
7. I.M. Sonin, "A Generalized Gittins Index for a Markov Chain and Its Recursive Calculation," *Statistics and Probability Letters*, vol. 78, no. 12, pp. 1526–1533, 2008. DOI: 10.1016/j.spl.2008.01.049.
8. P.R. Kumar, "A Survey of Some Results in Stochastic Adaptive Control," *SIAM Journal on Control and Optimization*, vol. 23, no. 3, pp. 329–380, 1985. DOI: 10.1137/0323023.
9. R.S. Sutton, "Dyna, an Integrated Architecture for Learning, Planning, and Reacting," *SIGART Bulletin*, vol. 2, no. 4, pp. 160–163, 1991. DOI: 10.1145/122344.122377.
10. A.W. Moore and C.G. Atkeson, "Prioritized Sweeping: Reinforcement Learning With Less Data and Less Time," *Machine Learning*, vol. 13, no. 1, pp. 103–130, 1993. DOI: 10.1007/BF00993104.
11. P. Poupart, N.A. Vlassis, J. Hoey, and K. Regan, "An Analytic Solution to Discrete Bayesian Reinforcement Learning," in *International Conference on Machine Learning (ICML)*, 2006.
12. M.O. Duff, "Optimal Learning: Computational Procedures for Bayes-Adaptive Markov Decision Processes," PhD thesis, University of Massachusetts at Amherst, 2002.
13. M.J.A. Strens, "A Bayesian Framework for Reinforcement Learning," in *International Conference on Machine Learning (ICML)*, 2000.
14. W.R. Thompson, "On the Likelihood That One Unknown Probability Exceeds Another in View of the Evidence of Two Samples," *Biometrika*, vol. 25, no. 3/4, pp. 285–294, 1933. DOI: 10.2307/2332286.
15. T. Wang, D.J. Lizotte, M.H. Bowling, and D. Schuurmans, "Bayesian Sparse Sampling for On-Line Reward Optimization," in *International Conference on Machine Learning (ICML)*, 2005.
16. J. Asmuth and M.L. Littman, "Learning Is Planning: Near Bayes-Optimal Reinforcement Learning via Monte-Carlo Tree Search," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2011.
17. A. Guez, D. Silver, and P. Dayan, "Scalable and Efficient Bayes-Adaptive Reinforcement Learning Based on Monte-Carlo Tree Search," *Journal of Artificial Intelligence Research*, vol. 48, pp. 841–883, 2013. DOI: 10.1613/jair.4117.
18. R. Sutton, "Learning to Predict by the Methods of Temporal Differences," *Machine Learning*, vol. 3, no. 1, pp. 9–44, 1988. DOI: 10.1007/BF00115009.

19. G.A. Rummery, "Problem Solving with Reinforcement Learning," PhD thesis, University of Cambridge, 1995.
20. C.J. C.H. Watkins, "Learning from Delayed Rewards," PhD thesis, University of Cambridge, 1989.
21. J. Peng and R.J. Williams, "Incremental Multi-Step Q-Learning," *Machine Learning*, vol. 22, no. 1-3, pp. 283–290, 1996. DOI: 10.1023/A:1018076709321.
22. L. Busoniu, R. Babuska, B. De Schutter, and D. Ernst, *Reinforcement Learning and Dynamic Programming Using Function Approximators*. Boca Raton, FL: CRC Press, 2010.
23. D. Chapman and L.P. Kaelbling, "Input Generalization in Delayed Reinforcement Learning: An Algorithm and Performance Comparisons," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 1991.
24. A.K. McCallum, "Reinforcement Learning with Selective Perception and Hidden State," PhD thesis, University of Rochester, 1995.
25. W.T.B. Uther and M.M. Veloso, "Tree Based Discretization for Continuous State Space Reinforcement Learning," in *AAAI Conference on Artificial Intelligence (AAAI)*, 1998.
26. M.J. Kochenderfer, "Adaptive Modelling and Planning for Learning Intelligent Behaviour," PhD thesis, University of Edinburgh, 2006.
27. R. Dearden, N. Friedman, and S.J. Russell, "Bayesian Q-Learning," in *AAAI Conference on Artificial Intelligence (AAAI)*, 1998.
28. Y. Engel, S. Mannor, and R. Meir, "Reinforcement Learning with Gaussian Processes," in *International Conference on Machine Learning (ICML)*, 2005.
29. M. Ghavamzadeh and Y. Engel, "Bayesian Policy Gradient Algorithms," in *Advances in Neural Information Processing Systems (NIPS)*, 2006.
30. L. Busoniu, R. Babuska, and B. De Schutter, "A Comprehensive Survey of Multiagent Reinforcement Learning," *IEEE Transactions on Systems Science and Cybernetics Part*, vol. 38, no. 2, pp. 156 –172, 2008. DOI: 10.1109/TSMCC.2007.913919.