We could optimize this by optimizing each variable separately, except for the fact that the parent relation is constrained by the acyclic condition of the belief network. However, given a total ordering of the variables, we have a classification problem in which we want to predict the probability of each variable given the predecessors in the total ordering. To represent $P(X_i|par(X_i, model))$ we could use, for example, a decision tree with probabilities of the leaves [as described in Section 7.5.1 (page 322)] or learn a squashed linear function. Given the preceding score, we can search over total orderings of the variables to maximize this score.

11.2.5 General Case of Belief Network Learning

The general case is with unknown structure, hidden variables, and missing data; we do not even know what variables exist. Two main problems exist. The first is the problem of missing data discussed earlier. The second problem is computational; although there is a well-defined search space, it is prohibitively large to try all combinations of variable ordering and hidden variables. If one only considers hidden variables that simplify the model (as seems reasonable), the search space is finite, but enormous.

One can either select the best model (e.g, the model with the highest a posteriori probability) or average over all models. Averaging over all models gives better predictions, but it is difficult to explain to a person who may have to understand or justify the model.

The problem with combining this approach with missing data seems to be much more difficult and requires more knowledge of the domain.

11.3 Reinforcement Learning

Imagine a robot that can act in a world, receiving rewards and punishments and determining from these what it should do. This is the problem of **reinforcement learning**. This chapter only considers fully observable, single-agent reinforcement learning [although Section 10.4.2 (page 441) considered a simple form of multiagent reinforcement learning].

We can formalize reinforcement learning in terms of Markov decision processes (page 400), but in which the agent, initially, only knows the set of possible states and the set of possible actions. Thus, the dynamics, $P(s'|a, s)$, and the reward function, $R(s, a, s')$, are initially unknown. An agent can act in a world and, after each step, it can observe the state of the world and observe what reward it obtained. Assume the agent acts to achieve the optimal discounted reward (page 403) with a discount factor γ.

Example 11.7 Consider the tiny reinforcement learning problem shown in Figure 11.8 (on the next page). There are six states the agent could be in, labeled as $s_0, \ldots, s_5$. The agent has four actions: *UpC*, *Up*, *Left*, *Right*. That is all the

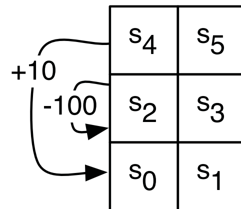

Figure 11.8: The environment of a tiny reinforcement learning problem

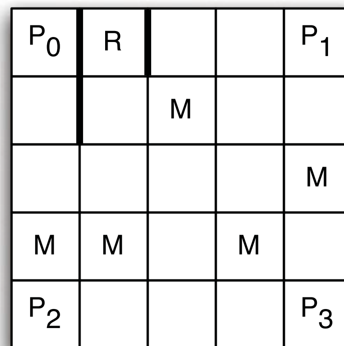

Figure 11.9: The environment of a grid game

agent knows before it starts. It does not know how the states are configured, what the actions do, or how rewards are earned.

Figure 11.8 shows the configuration of the six states. Suppose the actions work as follows:

upC (for "up carefully") The agent goes up, except in states s_4 and s_5, where the agent stays still, and has a reward of -1.

right The agent moves to the right in states s_0, s_2, s_4 with a reward of 0 and stays still in the other states, with a reward of -1.

left The agent moves one state to the left in states s_1, s_3, s_5. In state s_0, it stays in state s_0 and has a reward of -1. In state s_2, it has a reward of -100 and stays in state s_2. In state s_4, it gets a reward of 10 and moves to state s_0.

up With a probability of 0.8 it acts like *upC*, except the reward is 0. With probability 0.1 it acts as a *left*, and with probability 0.1 it acts as *right*.

Suppose there is a discounted reward (page 403) with a discount of 0.9. This can be translated as having a 0.1 chance of the agent leaving the game at any step, or as a way to encode that the agent prefers immediate rewards over future rewards.

Example 11.8 Figure 11.9 shows the domain of a more complex game. There are 25 grid locations the agent could be in. A prize could be on one of the cor-

ners, or there could be no prize. When the agent lands on a prize, it receives a reward of 10 and the prize disappears. When there is no prize, for each time step there is a probability that a prize appears on one of the corners. Monsters can appear at any time on one of the locations marked M. The agent gets damaged if a monster appears on the square the agent is on. If the agent is already damaged, it receives a reward of -10. The agent can get repaired (i.e., so it is no longer damaged) by visiting the repair station marked R.

In this example, the state consists of four components: $\langle X, Y, P, D \rangle$, where X is the X-coordinate of the agent, Y is the Y-coordinate of the agent, P is the position of the prize ($P = 0$ if there is a prize on P_0, $P = 1$ if there is a prize on P_1, similarly for 2 and 3, and $P = 4$ if there is no prize), and D is Boolean and is true when the agent is damaged. Because the monsters are transient, it is not necessary to include them as part of the state. There are thus $5 \times 5 \times 5 \times 2 = 250$ states. The environment is fully observable, so the agent knows what state it is in. But the agent does not know the meaning of the states; it has no idea initially about being damaged or what a prize is.

The agent has four actions: *up*, *down*, *left*, and *right*. These move the agent one step – usually one step in the direction indicated by the name, but sometimes in one of the other directions. If the agent crashes into an outside wall or one of the interior walls (the thick lines near the location R), it remains where is was and receives a reward of -1.

The agent does not know any of the story given here. It just knows there are 250 states and 4 actions, which state it is in at every time, and what reward was received each time.

This game is simple, but it is surprisingly difficult to write a good controller for it. There is a Java applet available on the book web site that you can play with and modify. Try to write a controller by hand for it; it is possible to write a controller that averages about 500 rewards for each 1,000 steps. This game is also difficult to learn, because visiting R is seemingly innocuous until the agent has determined that being damaged is bad, and that visiting R makes it not damaged. It must stumble on this while trying to collect the prizes. The states where there is no prize available do not last very long. Moreover, it has to learn this without being given the concept of *damaged*; all it knows, initially, is that there are 250 states and 4 actions.

Reinforcement learning is difficult for a number of reasons:

- The **blame attribution problem** is the problem of determining which action was responsible for a reward or punishment. The responsible action may have occurred a long time before the reward was received. Moreover, not a single action but rather a combination of actions carried out in the appropriate circumstances may be responsible for the reward. For example, you could teach an agent to play a game by rewarding it when it wins or loses; it must determine the brilliant moves that were needed to win. You may try to train a dog by saying "bad dog" when you come home and find a mess. The dog has to determine, out of all of the actions it did, which of them were the actions that were responsible for the reprimand.

- Even if the dynamics of the world does not change, the effect of an action of the agent depends on what the agent will do in the future. What may initially seem like a bad thing for the agent to do may end up being an optimal action because of what the agent does in the future. This is common among planning problems, but it is complicated in the reinforcement learning context because the agent does not know, a priori, the effects of its actions.
- The explore–exploit dilemma: if the agent has worked out a good course of actions, should it continue to follow these actions (exploiting what it has determined) or should it explore to find better actions? An agent that never explores may act forever in a way that could have been much better if it had explored earlier. An agent that always explores will never use what it has learned. This dilemma is discussed further in Section 11.3.4 (page 471).

11.3.1 Evolutionary Algorithms

One way to solve reinforcement algorithms is to treat this as an optimization problem (page 145), with the aim of selecting a policy that maximizes the expected reward collected. One way to do this via **policy search**. The aim is to search through the space of all policies to find the best policy. A policy is a controller (page 48) that can be evaluated by running it in the agent acting in the environment.

Policy search is often solved as a stochastic local search algorithm (page 134) by searching in the space of policies. A policy can be evaluated by running it in the environment a number of times.

One of the difficulties is in choosing a representation of the policy. Starting from an initial policy, the policy can be repeatedly evaluated in the environment and iteratively improved. This process is called an **evolutionary algorithm** because the agent, as a whole, is evaluated on how well it survives. This is often combined with genetic algorithms (page 142), which take us one step closer to the biological analogy of competing agents mutating genes. The idea is that crossover provides a way to combine the best features of policies.

Evolutionary algorithms have a number of issues. The first is the size of the state space. If there are n states and m actions, there are m^n policies. For example, for the game described in Example 11.7 (page 463), there are $4^6 = 4,096$ different policies. For the game of Example 11.8 (page 464), there are 250 states, and so $4^{250} \approx 10^{150}$ policies. This is a very small game, but it has more policies than there are particles in the universe.

Second, evolutionary algorithms use experiences very wastefully. If an agent was in state s_2 of Example 11.7 (page 463) and it moved left, you would like it to learn that it is bad to go left from state s_2. But evolutionary algorithms wait until the agent has finished and judge the policy as a whole. Stochastic local search will randomly try doing something else in state s_2 and so may eventually determine that that action was not good, but it is very indirect. Genetic algorithms are slightly better in that the policies that have the agent going left in state s_2 will die off, but again this is very indirect.

©Poole and Mackworth, 2009

Third, the performance of evolutionary algorithms can be very sensitive to the representation of the policy. The representation for a genetic algorithm should be such that crossover preserves the good parts of the policy. The representations are often tuned for the particular domain.

An alternative pursued in the rest of this chapter is to learn after every action. The components of the policy are learned, rather than the policy as a whole. By learning what do in each state, we can make the problem linear in the number of states rather than exponential in the number of states.

11.3.2 Temporal Differences

To understand how reinforcement learning works, first consider how to average experiences that arrive to an agent sequentially.

Suppose there is a sequence of numerical values, $v_1, v_2, v_3, \ldots$, and the goal is to predict the next value, given all of the previous values. One way to do this is to have a running approximation of the expected value of the v's. For example, given a sequence of students' grades and the aim of predicting the next grade, a reasonable prediction is to predict the average grade.

Let A_k be an estimate of the expected value based on the first k data points $v_1, \ldots, v_k$. A reasonable estimate is the sample average:

$$A_k = \frac{v_1 + \cdots + v_k}{k} .$$

Thus,

$$\begin{aligned} kA_k &= v_1 + \cdots + v_{k-1} + v_k \\ &= (k-1)A_{k-1} + v_k. \end{aligned}$$

Dividing by k gives

$$A_k = \left(1 - \frac{1}{k}\right) A_{k-1} + \frac{v_k}{k} .$$

Let $\alpha_k = \frac{1}{k}$; then

$$\begin{aligned} A_k &= (1-\alpha_k)A_{k-1} + \alpha_k v_k \\ &= A_{k-1} + \alpha_k(v_k - A_{k-1}). \end{aligned} \tag{11.1}$$

The difference, $v_k - A_{k-1}$, is called the **temporal difference error** or **TD error**; it specifies how different the new value, v_k, is from the old prediction, A_{k-1}. The old estimate, A_{k-1}, is updated by α_k times the TD error to get the new estimate, A_k. The qualitative interpretation of the temporal difference formula is that if the new value is higher than the old prediction, increase the predicted value; if the new value is less than the old prediction, decrease the predicted value. The change is proportional to the difference between the new value and the old prediction. Note that this equation is still valid for the first value, $k = 1$.

This analysis assumes that all of the values have an equal weight. However, suppose you are keeping an estimate of the expected value of students' grades. If schools start giving higher grades, the newer values are more useful for the estimate of the current situation than older grades, and so they should be weighted more in predicting new grades.

In reinforcement learning, the latter values of v_i (i.e., the more recent values) are more accurate than the earlier values and should be weighted more. One way to weight later examples more is to use Equation (11.1), but with α as a constant ($0 < \alpha \leq 1$) that does not depend on k. Unfortunately, this does not converge to the average value when variability exists in the values in the sequence, but it can track changes when the underlying process generating the values changes.

You could reduce α more slowly and potentially have the benefits of both approaches: weighting recent observations more and still converging to the average. You can guarantee convergence if

$$\sum_{k=1}^{\infty} \alpha_k = \infty \text{ and } \sum_{k=1}^{\infty} \alpha_k^2 < \infty.$$

The first condition is to ensure that random fluctuations and initial conditions get averaged out, and the second condition guarantees convergence.

Note that guaranteeing convergence to the average is not compatible with being able to adapt to make better predictions when the underlying process generating the values keeps changing.

For the rest of this chapter, α without a subscript is assumed to be a constant. With a subscript it is a function of the number of cases that have been combined for the particular estimate.

11.3.3 Q-learning

In Q-learning and related algorithms, an agent tries to learn the optimal policy from its history of interaction with the environment. A **history** of an agent is a sequence of state-action-rewards:

$$\langle s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, a_3, r_4, s_4 \ldots \rangle ,$$

which means that the agent was in state s_0 and did action a_0, which resulted in it receiving reward r_1 and being in state s_1; then it did action a_1, received reward r_2, and ended up in state s_2; then it did action a_2, received reward r_3, and ended up in state s_3; and so on.

We treat this history of interaction as a sequence of experiences, where an **experience** is a tuple

$$\langle s, a, r, s' \rangle ,$$

which means that the agent was in state s, it did action a, it received reward r, and it went into state s'. These experiences will be the data from which the

©Poole and Mackworth, 2009

agent can learn what to do. As in decision-theoretic planning, the aim is for the agent to maximize its value, which is usually the discounted reward (page 403).

Recall (page 404) that $Q^*(s,a)$, where a is an action and s is a state, is the expected value (cumulative discounted reward) of doing a in state s and then following the optimal policy.

Q-learning uses temporal differences to estimate the value of $Q^*(s,a)$. In Q-learning, the agent maintains a table of $Q[S,A]$, where S is the set of states and A is the set of actions. $Q[s,a]$ represents its current estimate of $Q^*(s,a)$.

An experience $\langle s,a,r,s'\rangle$ provides one data point for the value of $Q(s,a)$. The data point is that the agent received the future value of $r+\gamma V(s')$, where $V(s') = \max_{a'} Q(s',a')$; this is the actual current reward plus the discounted estimated future value. This new data point is called a **return**. The agent can use the temporal difference equation (11.1) to update its estimate for $Q(s,a)$:

$$Q[s,a] \leftarrow Q[s,a] + \alpha \left(r + \gamma \max_{a'} Q[s',a'] - Q[s,a] \right)$$

or, equivalently,

$$Q[s,a] \leftarrow (1-\alpha)Q[s,a] + \alpha \left(r + \gamma \max_{a'} Q[s',a'] \right) .$$

Figure 11.10 shows the Q-learning controller. This assumes that α is fixed; if α is varying, there will be a different count for each state–action pair and the algorithm would also have to keep track of this count.

Q-learning learns an optimal policy no matter which policy the agent is actually following (i.e., which action a it selects for any state s) as long as there is no bound on the number of times it tries an action in any state (i.e., it does not always do the same subset of actions in a state). Because it learns an optimal policy no matter which policy it is carrying out, it is called an **off-policy** method.

Example 11.9 Consider the domain Example 11.7 (page 463), shown in Figure 11.8 (page 464). Here is a sequence of $\langle s,a,r,s'\rangle$ experiences, and the update, where $\gamma = 0.9$ and $\alpha = 0.2$, and all of the Q-values are initialized to 0 (to two decimal points):

s	a	r	s'	Update
s_0	*upC*	-1	s_2	$Q[s_0, upC] = -0.2$
s_2	*up*	0	s_4	$Q[s_2, up] = 0$
s_4	*left*	10	s_0	$Q[s_4, left] = 2.0$
s_0	*upC*	-1	s_2	$Q[s_0, upC] = -0.36$
s_2	*up*	0	s_4	$Q[s_2, up] = 0.36$
s_4	*left*	10	s_0	$Q[s_4, left] = 3.6$
s_0	*up*	0	s_2	$Q[s_0, upC] = 0.06$
s_2	*up*	-100	s_2	$Q[s_2, up] = -19.65$
s_2	*up*	0	s_4	$Q[s_2, up] = -15.07$
s_4	*left*	10	s_0	$Q[s_4, left] = 4.89$

```
1: controller Q-learning(S, A, γ, α)
2:     Inputs
3:         S is a set of states
4:         A is a set of actions
5:         γ the discount
6:         α is the step size
7:     Local
8:         real array Q[S, A]
9:         previous state s
10:        previous action a
11:    initialize Q[S, A] arbitrarily
12:    observe current state s
13:    repeat
14:        select and carry out an action a
15:        observe reward r and state s'
16:        Q[s, a] ← Q[s, a] + α (r + γ max_a' Q[s', a'] − Q[s, a])
17:        s ← s'
18:    until termination
```

Figure 11.10: Q-learning controller

Notice how the reward of -100 is averaged in with the other rewards. After the experience of receiving the -100 reward, $Q[s_2, up]$ gets the value

$$0.8 \times 0.36 + 0.2 \times (-100 + 0.9 \times 0.36) = -19.65$$

At the next step, the same action is carried out with a different outcome, and $Q[s_2, up]$ gets the value

$$0.8 \times -19.65 + 0.2 \times (0 + 0.9 \times 3.6) = -15.07$$

After more experiences going *up* from s_2 and not receiving the reward of -100, the large negative reward will eventually be averaged in with the positive rewards and eventually have less influence on the value of $Q[s_2, up]$, until going *up* in state s_2 once again receives a reward of -100.

It is instructive to consider how using α_k to average the rewards works when the earlier estimates are much worse than more recent estimates. The following example shows the effect of a sequence of deterministic actions. Note that when an action is deterministic we can use $\alpha = 1$.

Example 11.10 Consider the domain Example 11.7 (page 463), shown in Figure 11.8 (page 464). Suppose that the agent has the experience

$$\langle s_0, right, 0, s_1, upC, -1, s_3, upC, -1, s_5, left, 0, s_4, left, 10, s_0 \rangle$$

©Poole and Mackworth, 2009

and repeats this sequence of actions a number of times. (Note that a real Q-learning agent would not keep repeating the same actions, particularly when some of them look bad, but we will assume this to let us understand how Q-learning works.)

Figure 11.11 (on the next page) shows how the Q-values are updated though a repeated execution of this action sequence. In both of these tables, the Q-values are initialized to 0.

(a) In the top run there is a separate α_k-value for each state–action pair. Notice how, in iteration 1, only the immediate rewards are updated. In iteration 2, there is a one-step backup from the positive rewards. Note that the -1 is not backed up because another action is available that has a Q-value of 0. In the third iteration, there is a two-step backup. $Q[s_3, upC]$ is updated because of the reward of 10, two steps ahead; its value is the average of its experiences: $(-1 + -1 + (-1 + 0.9 \times 6))/3$.

(b) The second run is where $\alpha = 1$; thus, it only takes into account the current estimate. Again, the reward is backed up one step in each iteration. In the third iteration, $Q[s_3, upC]$ is updated because of the reward of 10 two steps ahead, but with $\alpha = 1$, the algorithm ignores its previous estimates and uses its new experience, $-1 + 0.9 \times 0.9$. Having $\alpha = 1$ converges much faster than when $\alpha_k = 1/k$, but $\alpha = 1$ only converges when the actions are deterministic because $\alpha = 1$ implicitly assumes that the last reward and resulting state are representative of future ones.

(c) If the algorithm is run allowing the agent to explore, as is normal, some of the Q-values after convergence are shown in part (c). Note that, because there are stochastic actions, α cannot be 1 for the algorithm to converge. Note that the Q-values are larger than for the deterministic sequence of actions because these actions do not form an optimal policy.

The final policy after convergence is to do *up* in state s_0, *upC* in state s_2, *up* in states s_1 and s_3, and *left* in states s_4 and s_5.

You can run the applet for this example that is available on the book web site. Try different initializations, and try varying α.

11.3.4 Exploration and Exploitation

The Q-learning algorithm does not specify what the agent should actually do. The agent learns a Q-function that can be used to determine an optimal action. There are two things that are useful for the agent to do:

- **exploit** the knowledge that it has found for the current state s by doing one of the actions a that maximizes $Q[s, a]$.
- **explore** in order to build a better estimate of the optimal Q-function. That is, it should select a different action from the one that it currently thinks is best.

There have been a number of suggested ways to trade off exploration and exploitation:

This is a trace of Q-learning described in Example 11.10 (page 470).
(a) Q-learning for a deterministic sequence of actions with a separate α_k-value for each state–action pair, $\alpha_k = 1/k$.

Iteration	$Q[s_0, right]$	$Q[s_1, upC]$	$Q[s_3, upC]$	$Q[s_5, left]$	$Q[s_4, left]$
1	0	-1	-1	0	10
2	0	-1	-1	4.5	10
3	0	-1	0.35	6.0	10
4	0	-0.92	1.36	6.75	10
10	0.03	0.51	4	8.1	10
100	2.54	4.12	6.82	9.5	11.34
1000	4.63	5.93	8.46	11.3	13.4
10,000	6.08	7.39	9.97	12.83	14.9
100,000	7.27	8.58	11.16	14.02	16.08
1,000,000	8.21	9.52	12.1	14.96	17.02
10,000,000	8.96	10.27	12.85	15.71	17.77
∞	11.85	13.16	15.74	18.6	20.66

(b) Q-learning for a deterministic sequence of actions with $\alpha = 1$:

Iteration	$Q[s_0, right]$	$Q[s_1, upC]$	$Q[s_3, upC]$	$Q[s_5, left]$	$Q[s_4, left]$
1	0	-1	-1	0	10
2	0	-1	-1	9	10
3	0	-1	7.1	9	10
4	0	5.39	7.1	9	10
5	4.85	5.39	7.1	9	14.37
6	4.85	5.39	7.1	12.93	14.37
10	7.72	8.57	10.64	15.25	16.94
20	10.41	12.22	14.69	17.43	19.37
30	11.55	12.83	15.37	18.35	20.39
40	11.74	13.09	15.66	18.51	20.57
∞	11.85	13.16	15.74	18.6	20.66

(c) Q-values after full exploration and convergence:

Iteration	$Q[s_0, right]$	$Q[s_1, upC]$	$Q[s_3, upC]$	$Q[s_5, left]$	$Q[s_4, left]$
∞	19.5	21.14	24.08	27.87	30.97

Figure 11.11: Updates for a particular run of Q-learning

©Poole and Mackworth, 2009

- The ϵ-greedy strategy is to select the greedy action (one that maximizes $Q[s,a]$) all but ϵ of the time and to select a random action ϵ of the time, where $0 \le \epsilon \le 1$. It is possible to change ϵ through time. Intuitively, early in the life of the agent it should select a more random strategy to encourage initial exploration and, as time progresses, it should act more greedily.
- One problem with an ϵ-greedy strategy is that it treats all of the actions, apart from the best action, equivalently. If there are two seemingly good actions and more actions that look less promising, it may be more sensible to select among the good actions: putting more effort toward determining which of these promising actions is best, rather than putting in effort to explore the actions that look bad. One way to do that is to select action a with a probability depending on the value of $Q[s,a]$. This is known as a **soft-max** action selection. A common method is to use a **Gibbs** or **Boltzmann distribution**, where the probability of selecting action a in state s is proportional to $e^{Q[s,a]/\tau}$. That is, in state s, the agent selects action a with probability

  $$\frac{e^{Q[s,a]/\tau}}{\sum_a e^{Q[s,a]/\tau}}$$

  where $\tau > 0$ is the **temperature** specifying how randomly values should be chosen. When τ is high, the actions are chosen in almost equal amounts. As the temperature is reduced, the highest-valued actions are more likely to be chosen and, in the limit as $\tau \to 0$, the best action is always chosen.
- An alternative is "optimism in the face of uncertainty": initialize the Q-function to values that encourage exploration. If the Q-values are initialized to high values, the unexplored areas will look good, so that a greedy search will tend to explore. This does encourage exploration; however, the agent can hallucinate that some state–action pairs are good for a long time, even though there is no real evidence for it. A state only gets to look bad when all its actions look bad; but when all of these actions lead to states that look good, it takes a long time to get a realistic view of the actual values. This is a case where old estimates of the Q-values can be quite bad estimates of the actual Q-value, and these can remain bad estimates for a long time. To get fast convergence, the initial values should be as close as possible to the final values; trying to make them an overestimate will make convergence slower. Relying only on optimism in the face if uncertainty is not useful if the dynamics can change, because it is treating the initial time period as the time to explore and, after this initial exploration, there is no more exploration.

It is interesting to compare the interaction of the exploration strategies with different choices for how α is updated. See Exercise 11.8 (page 488).

11.3.5 Evaluating Reinforcement Learning Algorithms

We can judge a reinforcement learning algorithm by how good a policy it finds and how much reward it receives while acting in the world. Which is more important depends on how the agent will be deployed. If there is sufficient

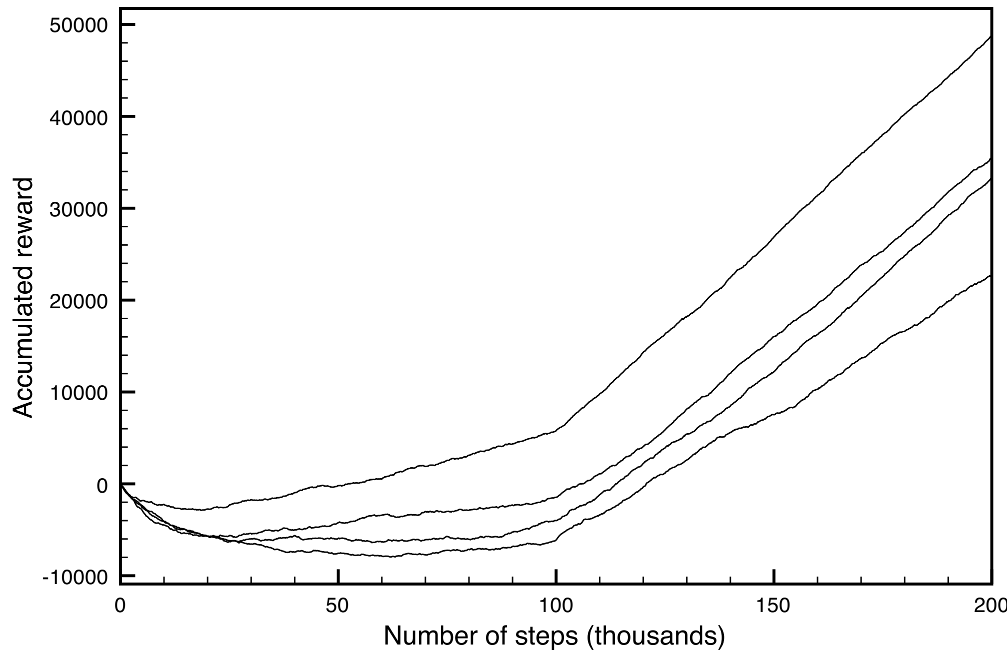

Figure 11.12: Cumulative reward as a function of the number of steps

time for the agent to learn safely before it is deployed, the final policy may be the most important. If the agent has to learn while being deployed, it may never get to the stage where it has learned the optimal policy, and the reward it receives while learning may be what the agent wants to maximize.

One way to show the performance of a reinforcement learning algorithm is to plot the cumulative reward (the sum of all rewards received so far) as a function of the number of steps. One algorithm dominates another if its plot is consistently above the other.

Example 11.11 Figure 11.12 compares four runs of Q-learning on the game of Example 11.8 (page 464). These plots were generated using the "trace on console" of the applet available on the course web site and then plotting the resulting data.

The plots are for different runs that varied according to whether α was fixed, according to the initial values of the Q-function, and according to the randomness in the action selection. They all used greedy exploit of 80% (i.e., $\epsilon = 0.2$) for the first 100,000 steps, and 100% (i.e., $\epsilon = 0.0$) for the next 100,000 steps. The top plot dominated the others.

There is a great deal variability of each algorithm on different runs, so to actually compare these algorithms one must run the same algorithm multiple times. For this domain, the cumulative rewards depend on whether the agent learns to visit the repair station, which it does not always learn. The cumula-

©Poole and Mackworth, 2009

tive reward therefore tends to be bimodal for this example. See Exercise 11.8 (page 488).

There are three statistics of this plot that are important:

- The asymptotic slope shows how good the policy is after the algorithm has stabilized.
- The minimum of the curve shows how much reward must be sacrificed before it starts to improve.
- The zero crossing shows how long it takes until the algorithm has recouped its cost of learning.

The last two statistics are applicable when both positive and negative rewards are available and having these balanced is reasonable behavior. For other cases, the cumulative reward should be compared with reasonable behavior that is appropriate for the domain; see Exercise 11.7 (page 488).

One thing that should be noted about the cumulative reward plot is that it measures total reward, yet the algorithms optimize discounted reward at each step. In general, you should optimize for, and evaluate your algorithm using, the optimality criterion that is most appropriate for the domain.

11.3.6 On-Policy Learning

Q-learning learns an optimal policy no matter what the agent does, as long as it explores enough. There may be cases where ignoring what the agent actually does is dangerous (there will be large negative rewards). An alternative is to learn the value of the policy the agent is actually carrying out so that it can be iteratively improved. As a result, the learner can take into account the costs associated with exploration.

An **off-policy learner** learns the value of the optimal policy independently of the agent's actions. Q-learning is an off-policy learner. An **on-policy learner** learns the value of the policy being carried out by the agent, including the exploration steps.

SARSA (so called because it uses state-action-reward-state-action experiences to update the Q-values) is an *on-policy* reinforcement learning algorithm that estimates the value of the policy being followed. An experience in SARSA is of the form $\langle s, a, r, s', a' \rangle$, which means that the agent was in state s, did action a, received reward r, and ended up in state s', from which it decided to do action a'. This provides a new experience to update $Q(s, a)$. The new value that this experience provides is $r + \gamma Q(s', a')$.

Figure 11.13 gives the SARSA algorithm.

SARSA takes into account the current exploration policy which, for example, may be greedy with random steps. It can find a different policy than Q-learning in situations when exploring may incur large penalties. For example, when a robot goes near the top of stairs, even if this is an optimal policy, it may be dangerous for exploration steps. SARSA will discover this and adopt

controller SARSA(S, A, γ, α)
inputs:
  S is a set of states
  A is a set of actions
  γ the discount
  α is the step size
internal state:
  real array $Q[S, A]$
  previous state s
  previous action a
begin
  initialize $Q[S, A]$ arbitrarily
  observe current state s
  select action a using a policy based on Q
  repeat forever:
    carry out an action a
    observe reward r and state s'
    select action a' using a policy based on Q
    $Q[s, a] \leftarrow Q[s, a] + \alpha\,(r + \gamma Q[s', a'] - Q[s, a])$
    $s \leftarrow s'$
    $a \leftarrow a'$
  end-repeat
end

Figure 11.13: SARSA: on-policy reinforcement learning

a policy that keeps the robot away from the stairs. It will find a policy that is optimal, taking into account the exploration inherent in the policy.

Example 11.12 In Example 11.10 (page 470), the optimal policy is to go up from state s_0 in Figure 11.8 (page 464). However, if the agent is exploring, this may not be a good thing to do because exploring from state s_2 is very dangerous.

If the agent is carrying out the policy that includes exploration, "when in state s, 80% of the time select the action a that maximizes $Q[s, a]$, and 20% of the time select an action at random," going up from s_0 is not optimal. An on-policy learner will try to optimize the policy the agent is following, not the optimal policy that does not include exploration.

If you were to repeat the experiment of Figure 11.11 (page 472), SARSA would back up the -1 values, whereas Q-learning did not because actions with an estimated value of 0 were available. The Q-values in parts (a) and (b) of that figure would converge to the same values, because they both converge to the value of that policy.

The Q-values of the optimal policy are less in SARSA than in Q-learning. The values for SARSA corresponding to part (c) of Figure 11.11 (page 472), are

©Poole and Mackworth, 2009

as follows:

Iteration	$Q[s_0, right]$	$Q[s_1, upC]$	$Q[s_3, upC]$	$Q[s_5, left]$	$Q[s_4, left]$
∞	9.2	10.1	12.7	15.7	18.0

The optimal policy using SARSA is to go right at state s_0. This is the optimal policy for an agent that does 20% exploration, because exploration is dangerous. If the rate of exploration were reduced, the optimal policy found would change. However, with less exploration, it would take longer to find an optimal policy.

SARSA is useful when you want to optimize the value of an agent that is exploring. If you want to do offline learning, and then use that policy in an agent that does not explore, Q-learning may be more appropriate.

11.3.7 Assigning Credit and Blame to Paths

In Q-learning and SARSA, only the previous state–action pair has its value revised when a reward is received. Intuitively, when an agent takes a number of steps that lead to a reward, all of the steps along the way could be held responsible and so receive some of the credit or the blame for a reward. This section gives an algorithm that assigns the credit and blame for all of the steps that lead to a reward.

Example 11.13 Suppose there is an action *right* that visits the states s_1, s_2, s_3, and s_4 in this order and a reward is only given when the agent enters s_4 from s_3, and any action from s_4 returns to state s_1. There is also an action *left* that moves to the left except in state s_4. In Q-learning and SARSA, after traversing right through the states s_1, s_2, s_3, and s_4 and receiving the reward, only the value of $Q[s_3, right]$ is updated. If the same sequence of states is visited again, the value of $Q[s_2, right]$ will be updated when it transitions into s_3. The value of $Q[s_1, right]$ is only updated after the next transition from state s_1 to s_2. In this sense, we say that Q-learning does a one-step backup.

Consider updating the value of $Q[s_3, right]$ based on the reward for entering state s_4. From the perspective of state s_4, the algorithm is doing a one-step backup. From the perspective of state s_3, it is doing a one-step look-ahead. To make the algorithm allow the blame to be associated with more than the previous step, the reward from entering step s_4 could do a two-step backup to update s_2 or, equivalently, a two-step look-ahead from s_2 and update s_2's value when the reward from entering s_4 is received. We will describe the algorithm in terms of a look-ahead but implement it using a backup.

With a two-step look-ahead, suppose the agent is in state s_t, does action a_t, ends up in state s_{t+1}, and receives reward r_{t+1}, then does action a_{t+1}, resulting in state s_{t+2} and receiving reward r_{t+2}. A two-step look-ahead at time t gives the return $R_t^{(2)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 V(s_{t+2})$, thus giving the TD error

$$\delta_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 V(s_{t+2}) - Q[s_t, a_t],$$

where $V(s_{t+2})$ is an estimate of the value of s_{t+2}. The two-step update is

$$Q[s_t, a_t] \leftarrow Q[s_t, a_t] + \alpha\delta_t.$$

Unfortunately, this is not a good estimate of the optimal Q-value, Q^*, because action a_{t+1} may not be an optimal action. For example, if action a_{t+1} was the action that takes the agent into a position with a reward of -10, and better actions were available, the agent should not update $Q[s_0, a_0]$. However, this multiple-step backup provides an improved estimate of the policy that the agent is actually following. If the agent is following policy π, this backup gives an improved estimate of Q^π. Thus multiple-step backup can be used in an on-policy method such as SARSA.

Suppose the agent is in state s_t, and it performs action a_t resulting in reward r_{t+1} and state s_{t+1}. It then does action a_{t+1}, resulting in reward r_{t+2} and state s_{t+2}, and so forth. An n-step return at time t, where $n \geq 1$, written $R_r^{(n)}$, is a data point for the estimated future value of the action at time t, given by looking n steps ahead, is

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V(s_{t+n}).$$

This could be used to update $Q[s_t, a_t]$ using the TD error $R_t^{(n)} - Q[s_t, a_t]$. However, it is difficult to know which n to use. Instead of selecting one particular n and looking forward n steps, it is possible to have an average of a number of n-step returns. One way to do this is to have a weighted average of all n-step returns, in which the returns in the future are exponentially decayed, with a decay of λ. This is the intuition behind the method called SARSA(λ); when a reward is received, the values of all of the visited states are updated. Those states farther in the past receive less of the credit or blame for the reward.

Let

$$R_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)},$$

where $(1-\lambda)$ is a normalizing constant to ensure we are getting an average. The following table gives the details of the sum:

look-ahead	Weight	Return
1 step	$1-\lambda$	$r_{t+1} + \gamma V(s_{t+1})$
2 step	$(1-\lambda)\lambda$	$r_{t+1} + \gamma r_{t+2} + \gamma^2 V(s_{t+2})$
3 step	$(1-\lambda)\lambda^2$	$r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 V(s_{t+3})$
4 step	$(1-\lambda)\lambda^3$	$r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \gamma^4 V(s_{t+3})$
$\cdots$	$\cdots$	$\cdots$
n step	$(1-\lambda)\lambda^{n-1}$	$r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^n V(s_{t+n})$
$\cdots$	$\cdots$	$\cdots$
total	1	

©Poole and Mackworth, 2009

Collecting together the common r_{t+i} terms gives

$$\begin{aligned} R_t^\lambda = \; & r_{t+1} + \gamma V(s_{t+1}) - \lambda\gamma V(s_{t+1}) \\ & + \lambda\gamma r_{t+2} + \lambda\gamma^2 V(s_{t+2}) - \lambda^2\gamma^2 V(s_{t+2}) \\ & + \lambda^2\gamma^2 r_{t+3} + \lambda^2\gamma^3 V(s_{t+3}) - \lambda^3\gamma^3 V(s_{t+3}) \\ & + \lambda^3\gamma^3 r_{t+4} + \lambda^3\gamma^4 V(s_{t+4}) - \lambda^4\gamma^4 V(s_{t+4}) \\ & + \dots . \end{aligned}$$

This will be used in a version of SARSA in which the future estimate of $V(s_{t+i})$ is the value of $Q[s_{t+i}, a_{t+i}]$. The TD error – the return minus the state estimate – is

$$\begin{aligned} R_t^\lambda - Q[s_t, a_t] = \; & r_{t+1} + \gamma Q[s_{t+1}, a_{t+1}] - Q[s_t, a_t] \\ & + \lambda\gamma(r_{t+2} + \gamma Q[s_{t+2}, a_{t+2}] - Q[s_{t+1}, a_{t+1}]) \\ & + \lambda^2\gamma^2(r_{t+3} + \gamma Q[s_{t+3}, a_{t+3}] - Q[s_{t+2}, a_{t+2}]) \\ & + \lambda^3\gamma^3(r_{t+4} + \gamma Q[s_{t+4}, a_{t+4}] - Q[s_{t+3}, a_{t+3}]) \\ & + \dots . \end{aligned}$$

Instead of waiting until the end, which may never occur, SARSA(λ) updates the value of $Q[s_t, a_t]$ at every time in the future. When the agent receives reward r_{t+i}, it can use the appropriate sum in the preceding equation to update $Q[s_t, a_t]$. The preceding description refers to all times; therefore, the update $r_{t+3} + \gamma Q[s_{t+3}, a_{t+3}] - Q[s_{t+2}, a_{t+2}]$ can be used to update all previous states. An agent can do this by keeping an **eligibility trace** that specifies how much a state–action pair should be updated at each time step. When a state–action pair is first visited, its eligibility is set to 1. At each subsequent time step its eligibility is reduced by a factor of $\lambda\gamma$. When the state–action pair is subsequently visited, 1 is added to its eligibility.

The eligibility trace is implemented by an array $e[S, A]$, where S is the set of all states and A is the set of all actions. After every action is carried out, the Q-value for every state–action pair is updated.

The algorithm, known as **SARSA(λ)**, is given in Figure 11.14 (on the next page).

Although this algorithm specifies that $Q[s, a]$ is updated for every state s and action a whenever a new reward is received, it may be much more efficient and only slightly less accurate to only update those values with an eligibility over some threshold.

11.3.8 Model-Based Methods

In many applications of reinforcement learning, plenty of time is available for computation between each action. For example, a physical robot may have many seconds between each action. Q-learning, which only does one backup per action, will not make full use of the available computation time.

An alternative to just learning the Q-values is to use the data to learn the model. That is, an agent uses its experience to explicitly learn $P(s'|s, a)$ and

controller SARSA($\lambda, S, A, \gamma, \alpha$)
inputs:
  S is a set of states
  A is a set of actions
  γ the discount
  α is the step size
  λ is the decay rate
internal state:
  real array $Q[S, A]$
  real array $e[S, A]$
  previous state s
  previous action a
begin
  initialize $Q[S, A]$ arbitrarily
  initialize $e[s, a] = 0$ for all s, a
  observe current state s
  select action a using a policy based on Q
  repeat forever:
    carry out an action a
    observe reward r and state s'
    select action a' using a policy based on Q
    $\delta \leftarrow r + \gamma Q[s', a'] - Q[s, a]$
    $e[s, a] \leftarrow e[s, a] + 1$
    for all s'', a'' :
      $Q[s'', a''] \leftarrow Q[s'', a''] + \alpha \delta e[s'', a'']$
      $e[s'', a''] \leftarrow \gamma \lambda e[s'', a'']$
    $s \leftarrow s'$
    $a \leftarrow a'$
  end-repeat
end

Figure 11.14: SARSA(λ)

©Poole and Mackworth, 2009

$R(s,a,s')$. For each action that the agent carries out in the environment, the agent can then do a number of steps of asynchronous value iteration (page 407) to give a better estimate of the Q-function.

Figure 11.15 (on the next page) shows a generic model-based reinforcement learner. As with other reinforcement learning programs, it keeps track of $Q[S,A]$, but it also maintains a model of the dynamics, represented here as T, where $T[s,a,s']$ is the count of the number of times that the agent has done a in state s and ended up in state s'. The counts are added to prior counts, as in a Dirichlet distribution (page 338), to compute probabilities. The algorithm assumes a common prior count. The $R[s,a,s']$ array maintains the average reward for transitioning from state s, doing action a, and ending up in state s'.

After each action, the agent observes the reward r and the resulting state s'. It then updates the transition-count matrix T and the average reward R. It then does a number of steps of asynchronous value iteration, using the updated probability model derived from T and the updated reward model. There are three main undefined parts to this algorithm:

- Which Q-values should be updated? It seems reasonable that the algorithm should at least update $Q[s,a]$, because more data have been received on the transition probability and reward. From there it can either do random updates or determine which Q-values would change the most. The elements that potentially have their values changed the most are the $Q[s_1,a_1]$ with the highest probability of ending up at a Q-value that has changed the most (i.e., where $Q[s_1,a_2]$ has changed the most). This can be implemented by keeping a priority queue of Q-values to consider.
- How many steps of asynchronous value iteration should be done between actions? An agent should continue doing steps of value iteration until it has to act or until it gets new information. Figure 11.15 (on the next page) assumes that the agent acts and then waits for an observation to arrive. When an observation arrives, the agent acts as soon as possible. There are may variants, including a more relaxed agent that runs the repeat loop in parallel with observing and acting. Such an agent acts when it must, and it updates the transition and reward model when it observes.
- What should be the initial values for $R[S,A,S]$ and $Q[S,A]$? Once the agent has observed a reward for a particular $\langle s,a,s'\rangle$ transition, it will use the average of all of the rewards received for that transition. However, it requires some value for the transitions it has never experienced when updating Q. If it is using the exploration strategy of optimism in the face of uncertainty, it can use *Rmax*, the maximum reward possible, as the initial value for R, to encourage exploration. As in value iteration (page 405), it is best to initialize Q to be as close as possible to the final Q-value.

The algorithm in Figure 11.15 (on the next page) assumes that the prior count is the same for all $\langle s,a,s'\rangle$ transitions. If some prior knowledge exists that some transitions are impossible or some are more likely, the prior count should not be uniform.

controller ModelBasedReinforementLearner(S, A, γ, c)
inputs:
  S is a set of states
  A is a set of actions
  γ the discount
  c is prior count
internal state:
  real array $Q[S, A]$
  real array $R[S, A, S]$
  integer array $T[S, A, S]$
  state s, s'
  action a
initialize $Q[S, A]$ arbitrarily
initialize $R[S, A, S]$ arbitrarily
initialize $T[S, A, S]$ to zero
observe current state s
select and carry out action a
repeat forever:
  observe reward r and state s'
  select and carry out action a
  $T[s, a, s'] \leftarrow T[s, a, s'] + 1$
  $R[s, a, s'] \leftarrow R[s, a, s'] + \frac{r - R[s, a, s']}{T[s, a, s']}$
  $s \leftarrow s'$
  repeat
    select state s_1, action a_1
    let $P = \sum_{s_2} (T[s_1, a_1, s_2] + c)$
    $Q[s_1, a_1] \leftarrow \sum_{s_2} \frac{T[s_1, a_1, s_2] + c}{P} \left(R[s_1, a_1, s_2] + \gamma \max_{a_2} Q[s_2, a_2] \right)$
  until an observation arrives

Figure 11.15: Model-based reinforcement learner

©Poole and Mackworth, 2009

This algorithm assumes that the rewards depend on the initial state, the action, and the final state. Moreover, it assumes that the reward for a $\langle s, a, s' \rangle$ transition is unknown until that exact transition has been observed. If the reward only depends on the initial state and the action, it is more efficient to have an $R[S, A]$. If there are separate action costs and rewards for entering a state, and the agent can separately observe the costs and rewards, the reward function can be decomposed into $C[A]$ and $R[S]$, leading to more efficient learning.

It is difficult to directly compare the model-based and model-free reinforcement learners. Typically, model-based learners are much more efficient in terms of experience; many fewer experiences are needed to learn well. However, the model-free methods often use less computation time. If experience was cheap, a different comparison would be needed than if experience was expensive.

11.3.9 Reinforcement Learning with Features

Usually, there are too many states to reason about explicitly. The alternative to reasoning explicitly in terms of states is to reason in terms of features. In this section, we consider reinforcement learning that uses an approximation of the Q-function using a linear combination of features of the state and the action. This is the simplest case and often works well. However, this approach requires careful selection of features; the designer should find features adequate to represent the Q-function. This is often a difficult engineering problem.

SARSA with Linear Function Approximation

You can use a linear function of features to approximate the Q-function in SARSA. This algorithm uses the on-policy method SARSA, because the agent's experiences sample the reward from the policy the agent is actually following, rather than sampling an optimum policy.

A number of ways are available to get a feature-based representation of the Q-function. In this section, we use features of both the state and the action to provide features for the linear function.

Suppose $F_1, \ldots, F_n$ are numerical features of the state and the action. Thus, $F_i(s, a)$ provides the value for the ith feature for state s and action a. These features are typically binary, with domain $\{0, 1\}$, but they can also be other numerical features. These features will be used to represent the Q-function.

$$Q_{\overline{w}}(s, a) = w_0 + w_1 F_1(s, a) + \cdots + w_n F_n(s, a)$$

for some tuple of weights, $\overline{w} = \langle w_0, w_1, \ldots, w_n \rangle$. Assume that there is an extra feature F_0 whose value is always 1, so that w_0 does not have to be a special case.

Example 11.14 In the grid game of Example 11.8 (page 464), some possible features are the following:

- $F_1(s,a)$ has value 1 if action a would most likely take the agent from state s into a location where a monster could appear and has value 0 otherwise.
- $F_2(s,a)$ has value 1 if action a would most likely take the agent into a wall and has value 0 otherwise.
- $F_3(s,a)$ has value 1 if step a would most likely take the agent toward a prize.
- $F_4(s,a)$ has value 1 if the agent is damaged in state s and action a takes it toward the repair station.
- $F_5(s,a)$ has value 1 if the agent is damaged and action a would most likely take the agent into a location where a monster could appear and has value 0 otherwise. That is, it is the same as $F_1(s,a)$ but is only applicable when the agent is damaged.
- $F_6(s,a)$ has value 1 if the agent is damaged in state s and has value 0 otherwise.
- $F_7(s,a)$ has value 1 if the agent is not damaged in state s and has value 0 otherwise.
- $F_8(s,a)$ has value 1 if the agent is damaged and there is a prize ahead in direction a.
- $F_9(s,a)$ has value 1 if the agent is not damaged and there is a prize ahead in direction a.
- $F_{10}(s,a)$ has the value of the x-value in state s if there is a prize at location P_0 in state s. That is, it is the distance from the left wall if there is a prize at location P_0.
- $F_{11}(s,a)$ has the value $4-x$, where x is the horizontal position in state s if there is a prize at location P_0 in state s. That is, it is the distance from the right wall if there is a prize at location P_0.
- $F_{12}(s,a)$ to $F_{29}(s,a)$ are like F_{10} and F_{11} for different combinations of the prize location and the distance from each of the four walls. For the case where the prize is at location P_0, the y-distance could take into account the wall.

An example linear function is

$$\begin{aligned}
&Q(s,a)\\
&= 2.0 - 1.0 * F_1(s,a) - 0.4 * F_2(s,a) - 1.3 * F_3(s,a)\\
&\quad - 0.5 * F_4(s,a) - 1.2 * F_5(s,a) - 1.6 * F_6(s,a) + 3.5 * F_7(s,a)\\
&\quad + 0.6 * F_8(s,a) + 0.6 * F_9(s,a) - 0.0 * F_{10}(s,a) + 1.0 * F_{11}(s,a) + \dots .
\end{aligned}$$

These are the learned values (to one decimal place) for one run of the algorithm that follows.

An experience in SARSA of the form $\langle s,a,r,s',a'\rangle$ (the agent was in state s, did action a, and received reward r and ended up in state s', in which it decided to do action a') provides the new estimate of $r+\gamma Q(s',a')$ to update $Q(s,a)$. This experience can be used as a data point for **linear regression** (page 304).

©Poole and Mackworth, 2009

```
1: controller SARSA-FA(F̄, γ, η)
2:     Inputs
3:         F̄ = ⟨F_1, ..., F_n⟩: a set of features
4:         γ ∈ [0,1]: discount factor
5:         η > 0: step size for gradient descent
6:     Local
7:         weights w̄ = ⟨w_0, ..., w_n⟩, initialized arbitrarily
8:     observe current state s
9:     select action a
10:    repeat
11:        carry out action a
12:        observe reward r and state s'
13:        select action a' (using a policy based on Q_w̄)
14:        let δ = r + γQ_w̄(s', a') − Q_w̄(s, a)
15:        for i = 0 to n do
16:            w_i ← w_i + ηδF_i(s, a)
17:        s ← s'
18:        a ← a'
19:    until termination
```

Figure 11.16: SARSA with linear function approximation

Let $\delta = r + \gamma Q(s', a') - Q(s, a)$. Using Equation (7.2) (page 305), weight w_i is updated by

$$w_i \leftarrow w_i + \eta \delta F_i(s, a).$$

This update can then be incorporated into SARSA, giving the algorithm shown in Figure 11.16.

Selecting an action a could be done using an ϵ-greedy function: with probability ϵ, an agent selects a random action and otherwise it selects an action that maximizes $Q_{\overline{w}}(s, a)$.

Although this program is simple to implement, **feature engineering** – choosing what features to include – is non-trivial. The linear function must not only convey the best action to carry out, it must also convey the information about what future states are useful.

Many variations of this algorithm exist. Different function approximations, such as a neural network or a decision tree with a linear function at the leaves, could be used. A common variant is to have a separate function for each action. This is equivalent to having the Q-function approximated by a decision tree that splits on actions and then has a linear function. It is also possible to split on other features.

A linear function approximation can also be combined with other methods such as SARSA(λ), Q-learning, or model-based methods. Note that some of

these methods have different convergence guarantees and different levels of performance.

Example 11.15 On the AIspace web site, there is an open-source implementation of this algorithm for the game of Example 11.8 (page 464) with the features of Example 11.14 (page 483). Try stepping through the algorithm for individual steps, trying to understand how each step updates each parameter. Now run it for a number of steps. Consider the performance using the evaluation measures of Section 11.3.5 (page 473). Try to make sense of the values of the parameters learned.

11.4 Review

The following are the main points you should have learned from this chapter:

- EM is an iterative method to learn the parameters of models with hidden variables (including the case in which the classification is hidden).
- The probabilities and the structure of belief networks can be learned from complete data. The probabilities can be derived from counts. The structure can be learned by searching for the best model given the data.
- Missing values in examples are often not missing at random. Why they are missing is often important to determine.
- A Markov decision process is an appropriate formalism for reinforcement learning. A common method is to learn an estimate of the value of doing each action in a state, as represented by the $Q(S, A)$ function.
- In reinforcement learning, an agent should trade off exploiting its knowledge and exploring to improve its knowledge.
- Off-policy learning, such as Q-learning, learns the value of the optimal policy. On-policy learning, such as SARSA, learns the value of the policy the agent is actually carrying out (which includes the exploration).
- Model-based reinforcement learning separates learning the dynamics and reward models from the decision-theoretic planning of what to do given the models.

11.5 References and Further Reading

Unsupervised learning is discussed by Fischer [1987] and Cheeseman, Kelly, Self, Stutz, Taylor, and Freeman [1988]. Bayesian classifiers are discussed by Duda et al. [2001] and Langley, Iba, and Thompson [1992]. Friedman and Goldszmidt [1996a] discuss how the naive Bayesian classifier can be generalized to allow for more appropriate independence assumptions.

For an overview of learning belief networks, see Heckerman [1999], Darwiche [2009], and [Koller and Friedman, 2009]. Structure learning using decision trees is based on Friedman and Goldszmidt [1996b].

©Poole and Mackworth, 2009

For an introduction to reinforcement learning, see Sutton and Barto [1998] or Kaelbling, Littman, and Moore [1996]. Bertsekas and Tsitsiklis [1996] investigate function approximation and its interaction with reinforcement learning.

11.6 Exercises

Exercise 11.1 Consider the unsupervised data of Figure 11.1 (page 454).

(a) How many different stable assignments of examples to classes does the k-means algorithm find when $k = 2$? [Hint: Try running the algorithm on the data with a number of different starting points, but also think about what assignments of examples to classes are stable.] Do not count permutations of the labels as different assignments.

(b) How many different stable assignments are there when $k = 3$?

(c) How many different stable assignments are there when $k = 4$?

(d) Why might someone suggest that three is the natural number of classes in this example? Give a definition for "natural" number of classes, and use this data to justify the definition.

Exercise 11.2 Suppose the k-means algorithm is run for an increasing sequence of values for k, and that it is run for a number of times for each k to find the assignment with a global minimum error. Is it possible that a number of values of k exist for which the error plateaus and then has a large improvement (e.g., when the error for $k = 3$, $k = 4$, and $k = 5$ are about the same, but the error for $k = 6$ is much lower)? If so, give an example. If not, explain why.

Exercise 11.3 Give an algorithm for EM for unsupervised learning [Figure 11.4 (page 457)] that does not store an A array, but rather recomputes the appropriate value for the M step. Each iteration should only involve one sweep through the data set. [Hint: For each tuple in the data set, update all of the relevant M_i-values.]

Exercise 11.4 Suppose a Q-learning agent, with fixed α and discount γ, was in state 34, did action 7, received reward 3, and ended up in state 65. What value(s) get updated? Give an expression for the new value. (Be as specific as possible.)

Exercise 11.5 Explain what happens in reinforcement learning if the agent always chooses the action that maximizes the Q-value. Suggest two ways to force the agent to explore.

Exercise 11.6 Explain how Q-learning fits in with the agent architecture of Section 2.2.1 (page 46). Suppose that the Q-learning agent has discount factor γ, a step size of α, and is carrying out an ϵ-greedy exploration strategy.

(a) What are the components of the belief state of the Q-learning agent?

(b) What are the percepts?

(c) What is the command function of the Q-learning agent?

(d) What is the belief-state transition function of the Q-learning agent?