Researchers try to find algorithms that work well even when the agent initially knows nothing about the environment. However, unless you are writing a paper, you should inject as much prior knowledge as possible into the agent, as it will speed up training dramatically. For example, you could add negative rewards proportional to the distance from the center of the screen, and to the pole's angle. Also, if you already have a reasonably good policy (e.g., hardcoded), you may want to train the neural network to imitate it before using policy gradients to improve it.

Despite its relative simplicity, this algorithm is quite powerful. You can use it to tackle much harder problems than balancing a pole on a cart. In fact, AlphaGo was based on a similar PG algorithm (plus *Monte Carlo Tree Search*, which is beyond the scope of this book).

We will now look at another popular family of algorithms. Whereas PG algorithms directly try to optimize the policy to increase rewards, the algorithms we will look at now are less direct: the agent learns to estimate the expected sum of discounted future rewards for each state, or the expected sum of discounted future rewards for each action in each state, then uses this knowledge to decide how to act. To understand these algorithms, we must first introduce *Markov decision processes* (MDP).

Markov Decision Processes

In the early 20th century, the mathematician Andrey Markov studied stochastic processes with no memory, called *Markov chains*. Such a process has a fixed number of states, and it randomly evolves from one state to another at each step. The probability for it to evolve from a state s to a state s' is fixed, and it depends only on the pair (s,s'), not on past states (the system has no memory).

Figure 16-7 shows an example of a Markov chain with four states. Suppose that the process starts in state s_0, and there is a 70% chance that it will remain in that state at the next step. Eventually it is bound to leave that state and never come back since no other state points back to s_0. If it goes to state s_1, it will then most likely go to state s_2 (90% probability), then immediately back to state s_1 (with 100% probability). It may alternate a number of times between these two states, but eventually it will fall into state s_3 and remain there forever (this is a *terminal state*). Markov chains can have very different dynamics, and they are heavily used in thermodynamics, chemistry, statistics, and much more.

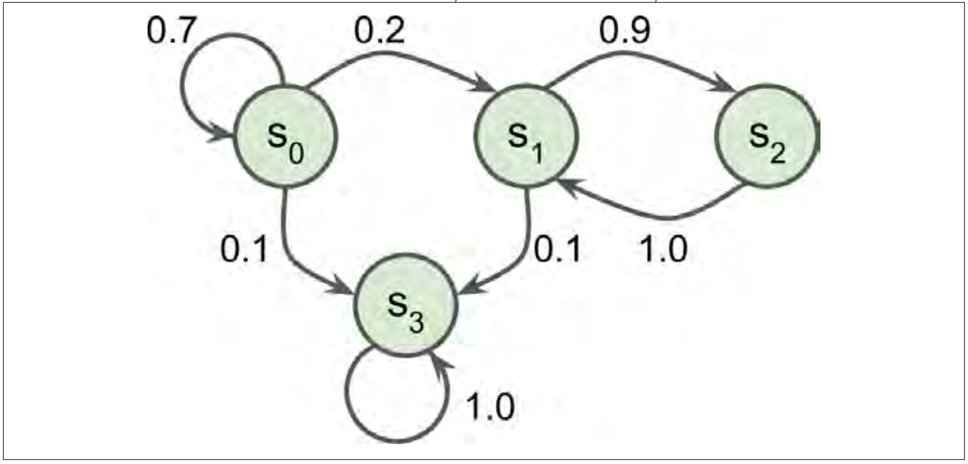

Figure 16-7. Example of a Markov chain

Markov decision processes were first described in the 1950s by Richard Bellman.[11] They resemble Markov chains but with a twist: at each step, an agent can choose one of several possible actions, and the transition probabilities depend on the chosen action. Moreover, some state transitions return some reward (positive or negative), and the agent's goal is to find a policy that will maximize rewards over time.

For example, the MDP represented in Figure 16-8 has three states and up to three possible discrete actions at each step. If it starts in state s_0, the agent can choose between actions a_0, a_1, or a_2. If it chooses action a_1, it just remains in state s_0 with certainty, and without any reward. It can thus decide to stay there forever if it wants. But if it chooses action a_0, it has a 70% probability of gaining a reward of +10, and remaining in state s_0. It can then try again and again to gain as much reward as possible. But at one point it is going to end up instead in state s_1. In state s_1 it has only two possible actions: a_0 or a_1. It can choose to stay put by repeatedly choosing action a_1, or it can choose to move on to state s_2 and get a negative reward of –50 (ouch). In state s_3 it has no other choice than to take action a_1, which will most likely lead it back to state s_0, gaining a reward of +40 on the way. You get the picture. By looking at this MDP, can you guess which strategy will gain the most reward over time? In state s_0 it is clear that action a_0 is the best option, and in state s_3 the agent has no choice but to take action a_1, but in state s_1 it is not obvious whether the agent should stay put (a_0) or go through the fire (a_2).

11 "A Markovian Decision Process," R. Bellman (1957).

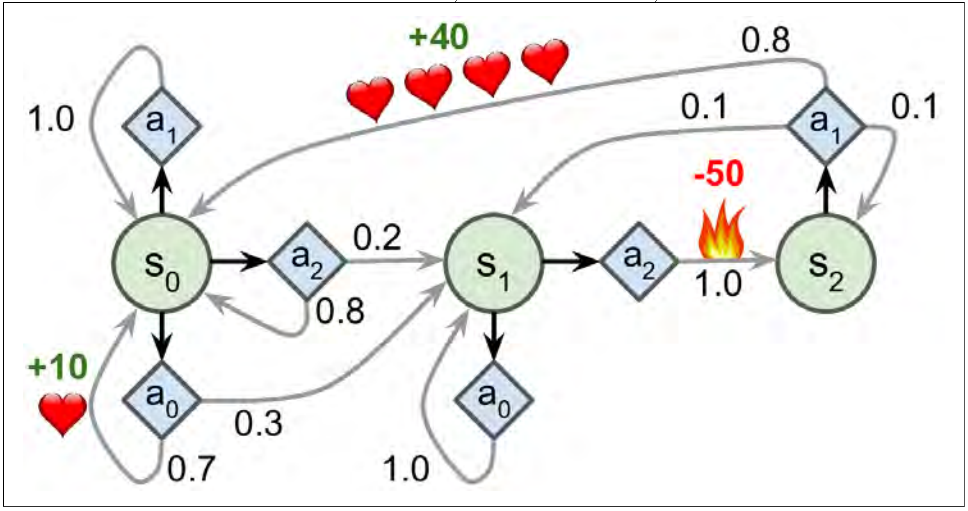

Figure 16-8. Example of a Markov decision process

Bellman found a way to estimate the *optimal state value* of any state *s*, noted $V^*(s)$, which is the sum of all discounted future rewards the agent can expect on average after it reaches a state *s*, assuming it acts optimally. He showed that if the agent acts optimally, then the *Bellman Optimality Equation* applies (see Equation 16-1). This recursive equation says that if the agent acts optimally, then the optimal value of the current state is equal to the reward it will get on average after taking one optimal action, plus the expected optimal value of all possible next states that this action can lead to.

Equation 16-1. Bellman Optimality Equation

$$V^*(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma . V^*(s')] \quad \text{for all } s$$

- $T(s, a, s')$ is the transition probability from state *s* to state *s*′, given that the agent chose action *a*.
- $R(s, a, s')$ is the reward that the agent gets when it goes from state *s* to state *s*′, given that the agent chose action *a*.
- γ is the discount rate.

This equation leads directly to an algorithm that can precisely estimate the optimal state value of every possible state: you first initialize all the state value estimates to zero, and then you iteratively update them using the *Value Iteration* algorithm (see Equation 16-2). A remarkable result is that, given enough time, these estimates are guaranteed to converge to the optimal state values, corresponding to the optimal policy.

Equation 16-2. Value Iteration algorithm

$$V_{k+1}(s) \leftarrow \max_{a} \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma \cdot V_k(s')\right] \quad \text{for all } s$$

- $V_k(s)$ is the estimated value of state s at the k^{th} iteration of the algorithm.

This algorithm is an example of *Dynamic Programming*, which breaks down a complex problem (in this case estimating a potentially infinite sum of discounted future rewards) into tractable subproblems that can be tackled iteratively (in this case finding the action that maximizes the average reward plus the discounted next state value).

Knowing the optimal state values can be useful, in particular to evaluate a policy, but it does not tell the agent explicitly what to do. Luckily, Bellman found a very similar algorithm to estimate the optimal *state-action values*, generally called *Q-Values*. The optimal Q-Value of the state-action pair (s,a), noted $Q^*(s,a)$, is the sum of discounted future rewards the agent can expect on average after it reaches the state s and chooses action a, but before it sees the outcome of this action, assuming it acts optimally after that action.

Here is how it works: once again, you start by initializing all the Q-Value estimates to zero, then you update them using the *Q-Value Iteration* algorithm (see Equation 16-3).

Equation 16-3. Q-Value Iteration algorithm

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma \cdot \max_{a'} Q_k(s',a')\right] \quad \text{for all } (s,a)$$

Once you have the optimal Q-Values, defining the optimal policy, noted $\pi^*(s)$, is trivial: when the agent is in state s, it should choose the action with the highest Q-Value for that state: $\pi^*(s) = \underset{a}{\operatorname{argmax}}\ Q^*(s,a)$.

Let's apply this algorithm to the MDP represented in Figure 16-8. First, we need to define the MDP:

```
nan=np.nan  # represents impossible actions
T = np.array([  # shape=[s, a, s']
        [[0.7, 0.3, 0.0], [1.0, 0.0, 0.0], [0.8, 0.2, 0.0]],
        [[0.0, 1.0, 0.0], [nan, nan, nan], [0.0, 0.0, 1.0]],
        [[nan, nan, nan], [0.8, 0.1, 0.1], [nan, nan, nan]],
    ])
R = np.array([  # shape=[s, a, s']
        [[10., 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]],
        [[10., 0.0, 0.0], [nan, nan, nan], [0.0, 0.0, -50.]],
        [[nan, nan, nan], [40., 0.0, 0.0], [nan, nan, nan]],
    ])
possible_actions = [[0, 1, 2], [0, 2], [1]]
```

Now let's run the Q-Value Iteration algorithm:

```
Q = np.full((3, 3), -np.inf)  # -inf for impossible actions
for state, actions in enumerate(possible_actions):
    Q[state, actions] = 0.0  # Initial value = 0.0, for all possible actions

learning_rate = 0.01
discount_rate = 0.95
n_iterations = 100

for iteration in range(n_iterations):
    Q_prev = Q.copy()
    for s in range(3):
        for a in possible_actions[s]:
            Q[s, a] = np.sum([
                T[s, a, sp] * (R[s, a, sp] + discount_rate * np.max(Q_prev[sp]))
                for sp in range(3)
            ])
```

The resulting Q-Values look like this:

```
>>> Q
array([[ 21.89498982,  20.80024033,  16.86353093],
       [  1.11669335,         -inf,   1.17573546],
       [        -inf,  53.86946068,         -inf]])
>>> np.argmax(Q, axis=1)  # optimal action for each state
array([0, 2, 1])
```

This gives us the optimal policy for this MDP, when using a discount rate of 0.95: in state s_0 choose action a_0, in state s_1 choose action a_2 (go through the fire!), and in state s_2 choose action a_1 (the only possible action). Interestingly, if you reduce the discount rate to 0.9, the optimal policy changes: in state s_1 the best action becomes a_0 (stay put; don't go through the fire). It makes sense because if you value the present much more than the future, then the prospect of future rewards is not worth immediate pain.

Temporal Difference Learning and Q-Learning

Reinforcement Learning problems with discrete actions can often be modeled as Markov decision processes, but the agent initially has no idea what the transition probabilities are (it does not know $T(s, a, s')$), and it does not know what the rewards are going to be either (it does not know $R(s, a, s')$). It must experience each state and