```
        [[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

each transition at least once to know the rewards, and it must experience them multiple times if it is to have a reasonable estimate of the transition probabilities.

The *Temporal Difference Learning* (TD Learning) algorithm is very similar to the Value Iteration algorithm, but tweaked to take into account the fact that the agent has only partial knowledge of the MDP. In general we assume that the agent initially knows only the possible states and actions, and nothing more. The agent uses an *exploration policy*—for example, a purely random policy—to explore the MDP, and as it progresses the TD Learning algorithm updates the estimates of the state values based on the transitions and rewards that are actually observed (see Equation 16-4).

*Equation 16-4. TD Learning algorithm*

$$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha\left(r + \gamma \cdot V_k(s')\right)$$

- $\alpha$ is the learning rate (e.g., 0.01).

TD Learning has many similarities with Stochastic Gradient Descent, in particular the fact that it handles one sample at a time. Just like SGD, it can only truly converge if you gradually reduce the learning rate (otherwise it will keep bouncing around the optimum).

For each state $s$, this algorithm simply keeps track of a running average of the immediate rewards the agent gets upon leaving that state, plus the rewards it expects to get later (assuming it acts optimally).

Similarly, the Q-Learning algorithm is an adaptation of the Q-Value Iteration algorithm to the situation where the transition probabilities and the rewards are initially unknown (see Equation 16-5).

*Equation 16-5. Q-Learning algorithm*

$$Q_{k+1}(s,a) \leftarrow (1-\alpha)Q_k(s,a) + \alpha\left(r + \gamma \cdot \max_{a'} Q_k(s',a')\right)$$

For each state-action pair $(s, a)$, this algorithm keeps track of a running average of the rewards $r$ the agent gets upon leaving the state $s$ with action $a$, plus the rewards it expects to get later. Since the target policy would act optimally, we take the maximum of the Q-Value estimates for the next state.

Here is how Q-Learning can be implemented:

```
import numpy.random as rnd

learning_rate0 = 0.05
learning_rate_decay = 0.1
n_iterations = 20000

s = 0 # start in state 0

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

for iteration in range(n_iterations):
    a = rnd.choice(possible_actions[s])  # choose an action (randomly)
    sp = rnd.choice(range(3), p=T[s, a]) # pick next state using T[s, a]
    reward = R[s, a, sp]
    learning_rate = learning_rate0 / (1 + iteration * learning_rate_decay)
    Q[s, a] = learning_rate * Q[s, a] + (1 - learning_rate) * (
            reward + discount_rate * np.max(Q[sp])
        )
    s = sp # move to next state
```

Given enough iterations, this algorithm will converge to the optimal Q-Values. This is called an *off-policy* algorithm because the policy being trained is not the one being executed. It is somewhat surprising that this algorithm is capable of learning the optimal policy by just watching an agent act randomly (imagine learning to play golf when your teacher is a drunken monkey). Can we do better?

## Exploration Policies

Of course Q-Learning can work only if the exploration policy explores the MDP thoroughly enough. Although a purely random policy is guaranteed to eventually visit every state and every transition many times, it may take an extremely long time to do so. Therefore, a better option is to use the *ε-greedy policy*: at each step it acts randomly with probability ε, or greedily (choosing the action with the highest Q-Value) with probability 1-ε. The advantage of the ε-greedy policy (compared to a completely random policy) is that it will spend more and more time exploring the interesting parts of the environment, as the Q-Value estimates get better and better, while still spending some time visiting unknown regions of the MDP. It is quite common to start with a high value for ε (e.g., 1.0) and then gradually reduce it (e.g., down to 0.05).

Alternatively, rather than relying on chance for exploration, another approach is to encourage the exploration policy to try actions that it has not tried much before. This can be implemented as a bonus added to the Q-Value estimates, as shown in Equation 16-6.