DL in Applied Mathematics

Lecture 12: **Reinforcement Learning**

Marat Nurtas
Associate professor of IITU, PhD in Mathematical and Computer Modeling
Department of Mathematical and Computer Modeling
International Information Technology University, Almaty, Kazakhstan

***Reinforcement Learning* (RL)** is one of the most exciting fields of Machine Learning today, and also one of the oldest. It has been around since the 1950s, producing many interesting applications over the years, in particular in games (e.g., *TD-Gammon*, a *Backgammon* playing program) and in machine control, but seldom making the headline news. But a revolution took place in 2013 when researchers from an English startup called DeepMind demonstrated a system that could learn to play just about any Atari game from scratch, eventually outperforming humans in most of them, using only raw pixels as inputs and without any prior knowledge of the rules of the games.

Learning to Optimize Rewards

In Reinforcement Learning, a software *agent* makes *observations* and takes *actions* within an *environment*, and in return it receives *rewards*. Its objective is to learn to act in a way that will maximize its expected long-term rewards.

This is quite a broad setting, which can apply to a wide variety of tasks. Here are a few examples (see Figure 16-1):

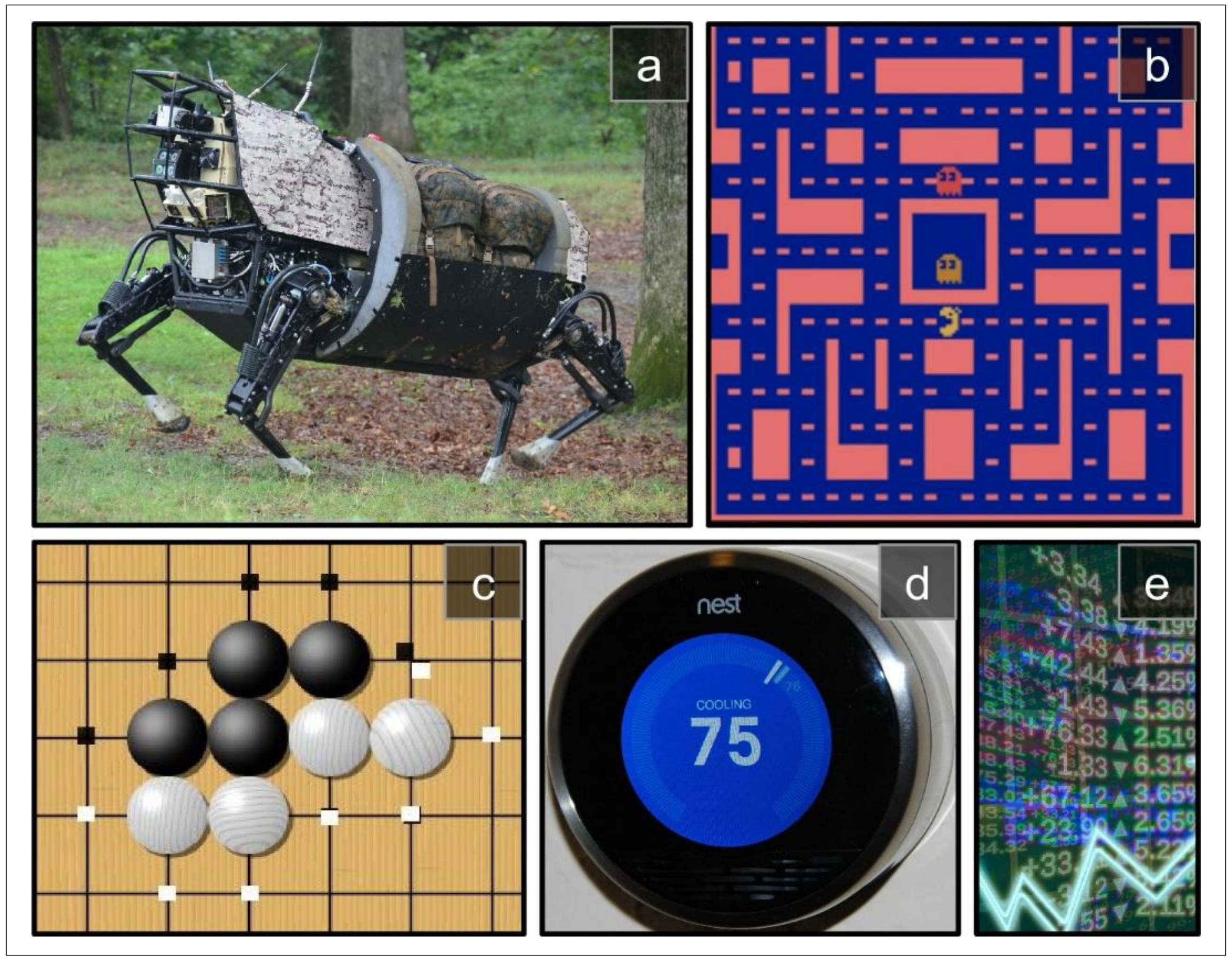


Figure 16-1. Reinforcement Learning examples: (a) walking robot, (b) Ms. Pac-Man, (c) Go player, (d) thermostat, (e) automatic trader

Policy Search

How would you train such a robot? There are just two *policy parameters* you can tweak: the probability *p* and the angle range *r*. One possible learning algorithm could be to try out many different values for these parameters, and pick the combination that performs best (see Figure 16-3).

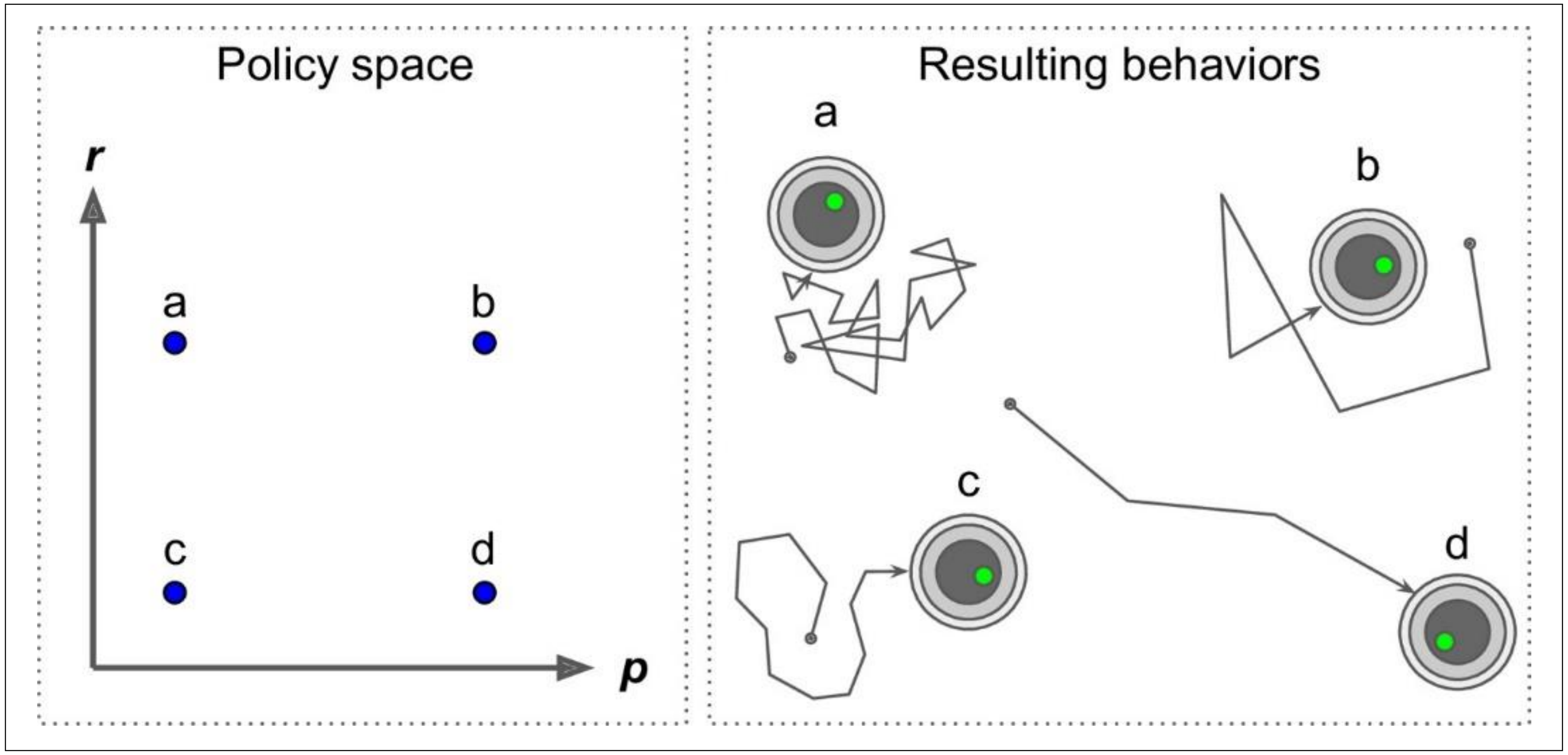


Figure 16-3. Four points in policy space and the agent's corresponding behavior

Introduction to OpenAI Gym

OpenAI gym8 is a toolkit that provides a wide variety of simulated environments (Atari games, board games, 2D and 3D physical simulations, and so on), so you can train agents, compare them, or develop new RL algorithms.

Let’s install OpenAI gym. For a minimal OpenAI gym installation, simply use pip:

```
$ pip3 install --upgrade gym
```

Next open up a Python shell or a Jupyter notebook and create your first environment:

```
>>> import gym
>>> env = gym.make("CartPole-v0")
[2016-10-14 16:03:23,199] Making new env: MsPacman-v0
>>> obs = env.reset()
>>> obs
array([-0.03799846, -0.03288115, 0.02337094, 0.00720711])
>>> env.render()
```

Introduction to OpenAI Gym

The `make()` function creates an environment, in this case a CartPole environment. This is a 2D simulation in which a cart can be accelerated left or right in order to balance a pole placed on top of it (see Figure 16-4).

The `render()` method displays the environment as shown in Figure 16-4.

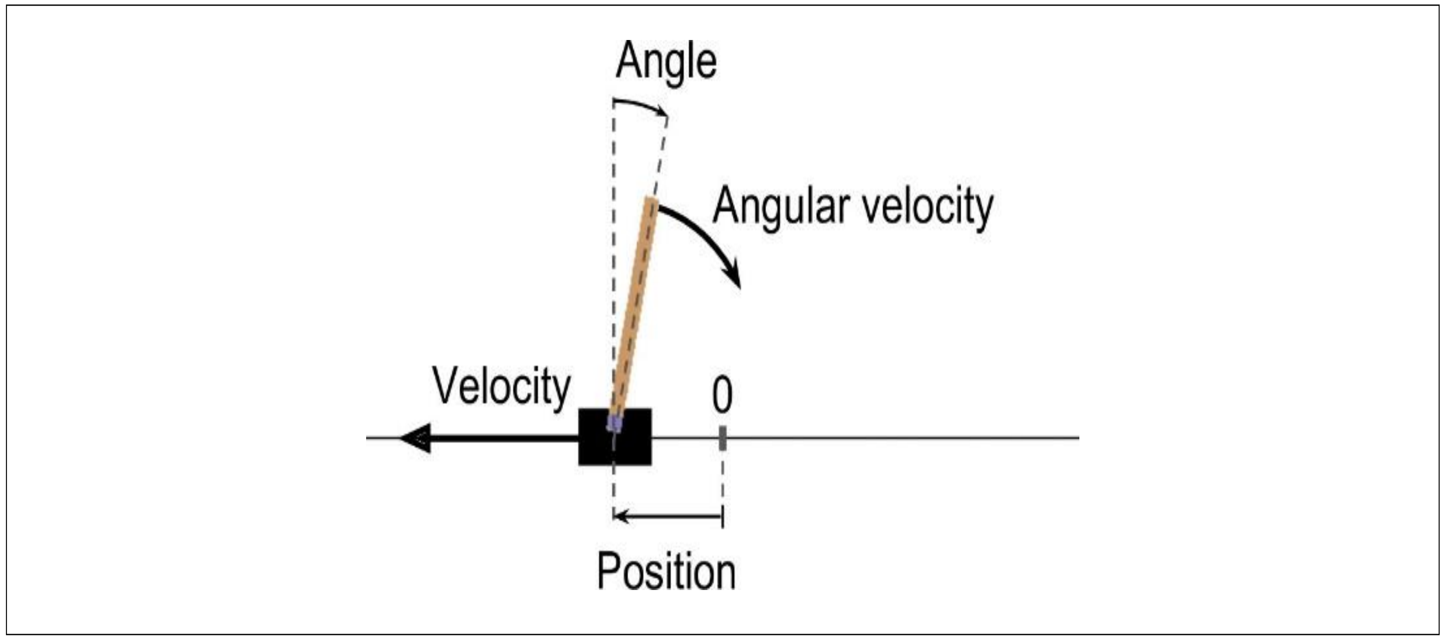


Figure 16-4. The CartPole environment

Neural Network Policies

Let's create a neural network policy. Just like the policy we hardcoded earlier, this neural network will take an observation as input, and it will output the action to be executed. More precisely, it will estimate a probability for each action, and then we will select an action randomly according to the estimated probabilities (see Figure 16-5).

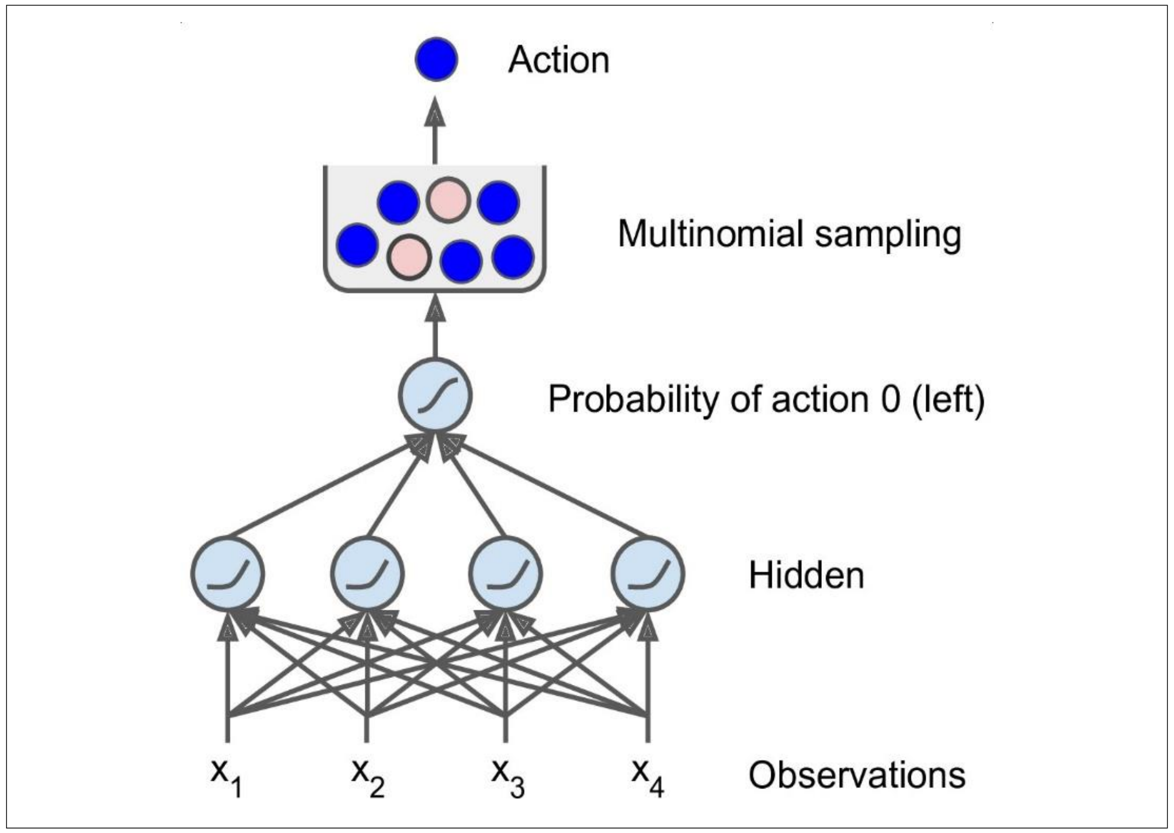


Figure 16-5. Neural network policy

Neural Network Policies

Here is the code to build this neural network policy using TensorFlow:

```
import tensorflow as tf
from tensorflow.contrib.layers import fully_connected

# 1. Specify the neural network architecture
n_inputs = 4 #  == env.observation_space.shape[0]
n_hidden = 4 # it's a simple task, we don't need more hidden neurons n_outputs = 1 # only outputs the prol
of accelerating left initializer = tf.contrib.layers.variance_scaling_initializer()

# 2. Build the neural network
X = tf.placeholder(tf.float32, shape=[None, n_inputs])
hidden = fully_connected(X, n_hidden, activation_fn=tf.nn.elu,
weights_initializer=initializer) logits = fully_connected(hidden, n_outputs, activation_fn=None,
weights_initializer=initializer) outputs = tf.nn.sigmoid(logits)

# 3. Select a random action based on the estimated probabilities
p_left_and_right = tf.concat(axis=1, values=[outputs, 1 - outputs]) action =
tf.multinomial(tf.log(p_left_and_right), num_samples=1)

init = tf.global_variables_initializer()
```

Evaluating Actions: The Credit Assignment Problem

This is called the *credit assignment problem*: when the agent gets a reward, it is hard for it to know which actions should get credited (or blamed) for it. Think of a dog that gets rewarded hours after it behaved well; will it understand what it is rewarded for?

To tackle this problem, a common strategy is to evaluate an action based on the sum of all the rewards that come after it, usually applying a *discount rate r* at each step. For example (see Figure 16-6)

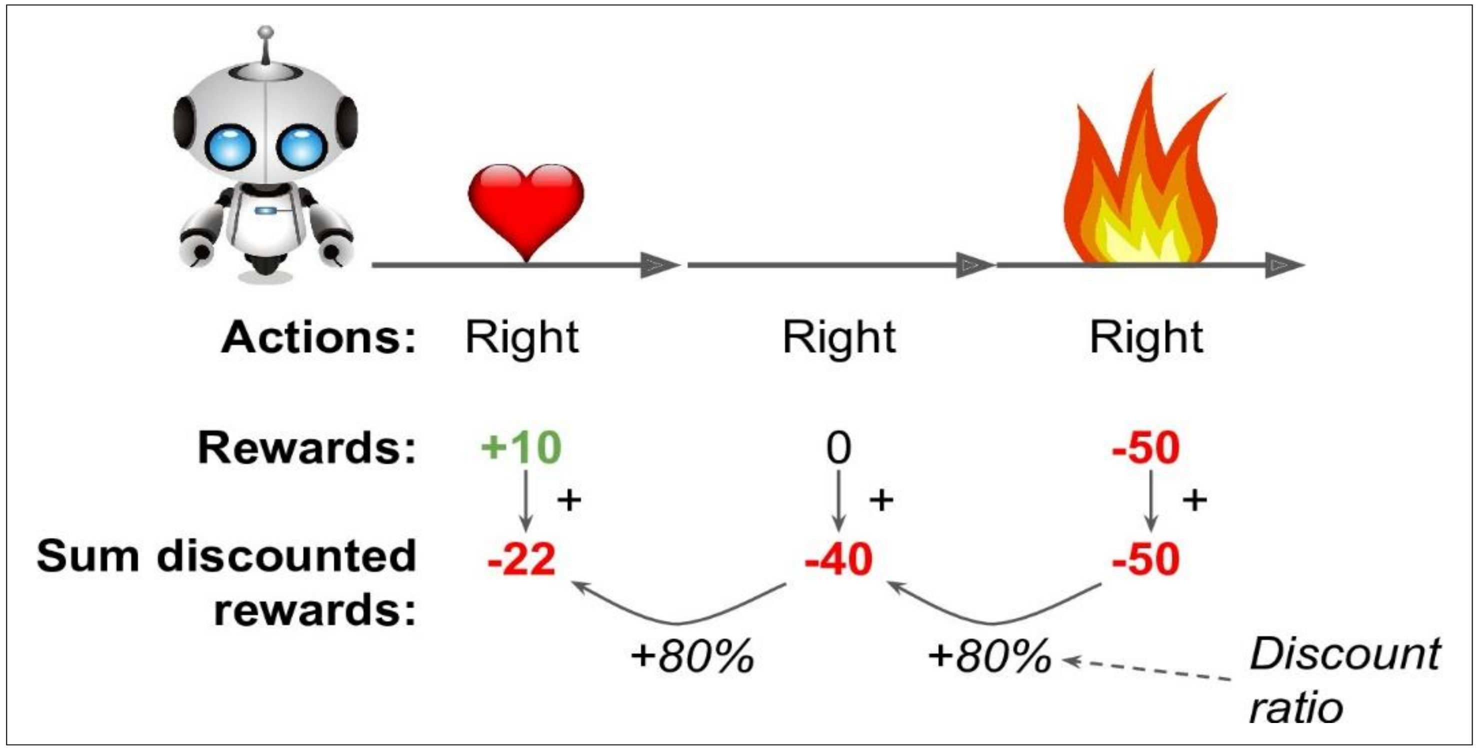


Figure 16-6. Discounted rewards

Evaluating Actions: The Credit Assignment Problem

This is called the *credit assignment problem*: when the agent gets a reward, it is hard for it to know which actions should get credited (or blamed) for it. Think of a dog that gets rewarded hours after it behaved well; will it understand what it is rewarded for?

To tackle this problem, a common strategy is to evaluate an action based on the sum of all the rewards that come after it, usually applying a *discount rate r* at each step. For example (see Figure 16-6)

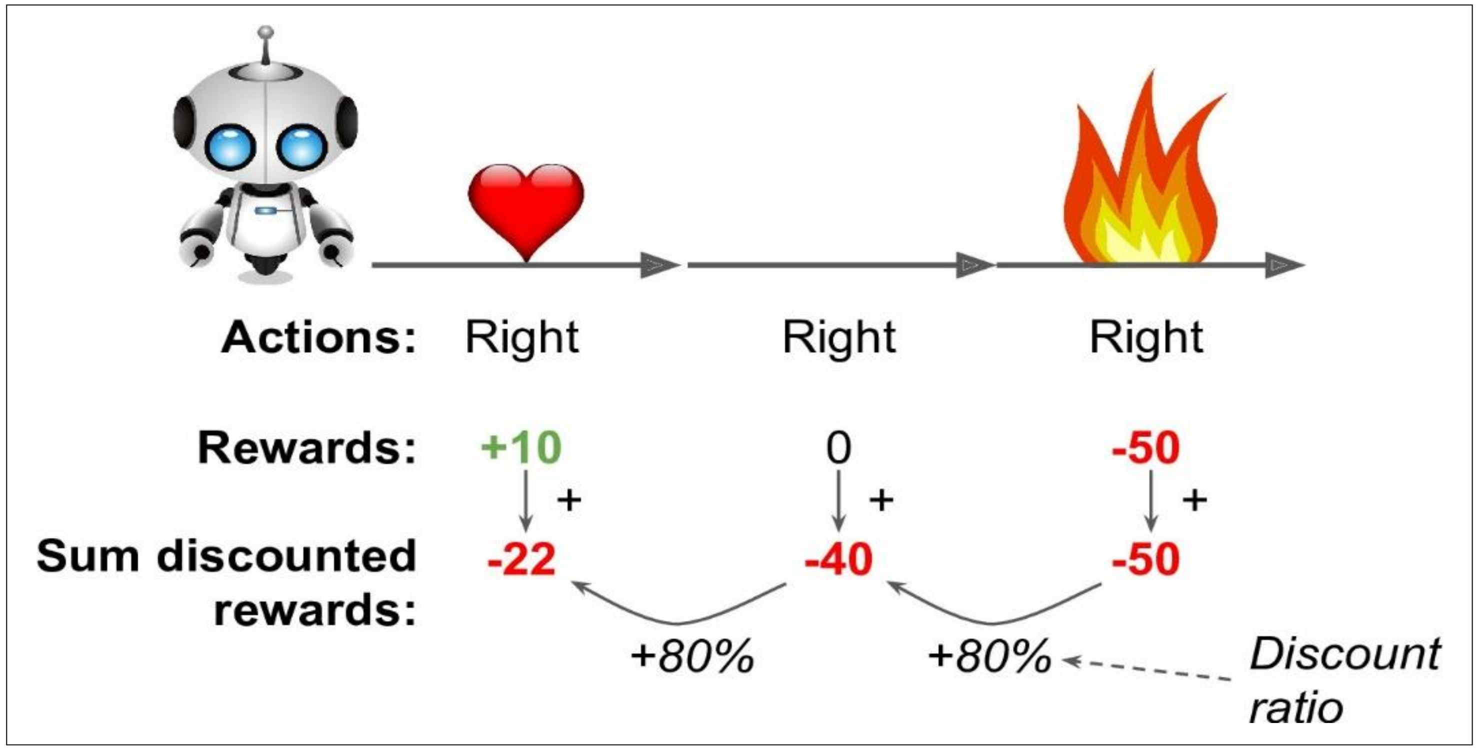


Figure 16-6. Discounted rewards

Policy Gradients

As discussed earlier, PG algorithms optimize the parameters of a policy by following the gradients toward higher rewards. One popular class of PG algorithms, called *REINFORCE algorithms*, was introduced back in 1992 by Ronald Williams. Here is one common variant:

1. First, let the neural network policy play the game several times and at each step compute the gradients that would make the chosen action even more likely, but don't apply these gradients yet.
2. Once you have run several episodes, compute each action's score (using the method described in the previous paragraph).
3. If an action's score is positive, it means that the action was good and you want to apply the gradients computed earlier to make the action even more likely to be chosen in the future. However, if the score is negative, it means the action was bad and you want to apply the opposite gradients to make this action slightly *less* likely in the future. The solution is simply to multiply each gradient vector by the corresponding action's score.
4. Finally, compute the mean of all the resulting gradient vectors, and use it to perform a Gradient Descent step.

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).

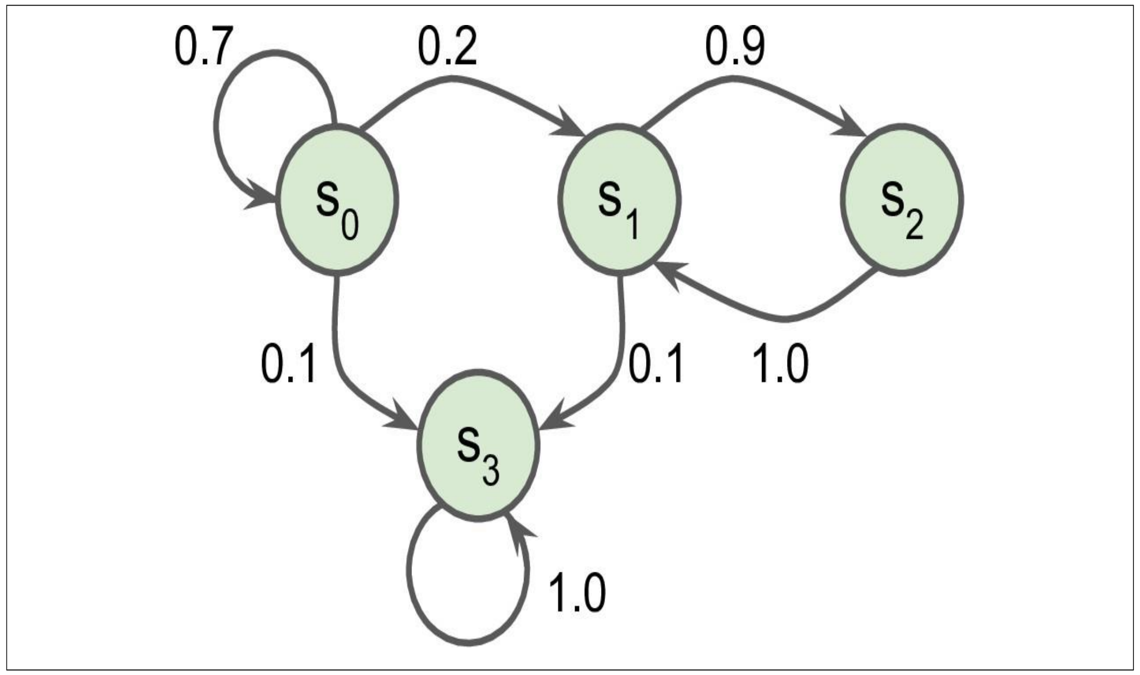


Figure 16-7. Example of a Markov chain

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).

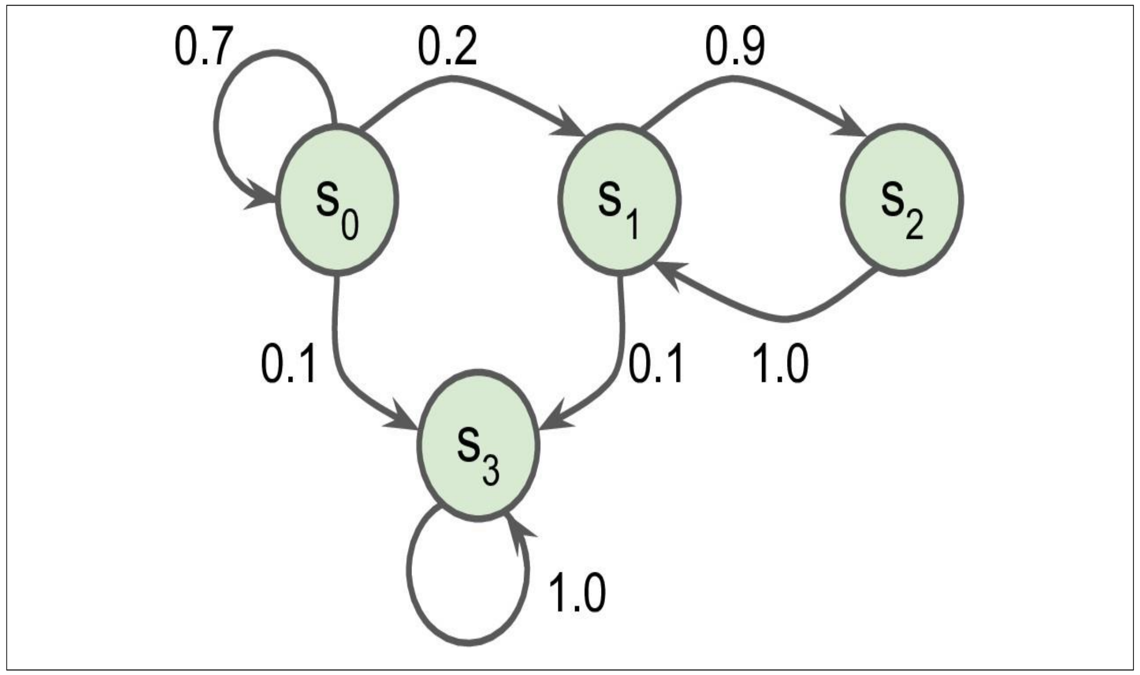


Figure 16-7. Example of a Markov chain

Markov Decision Processes

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.

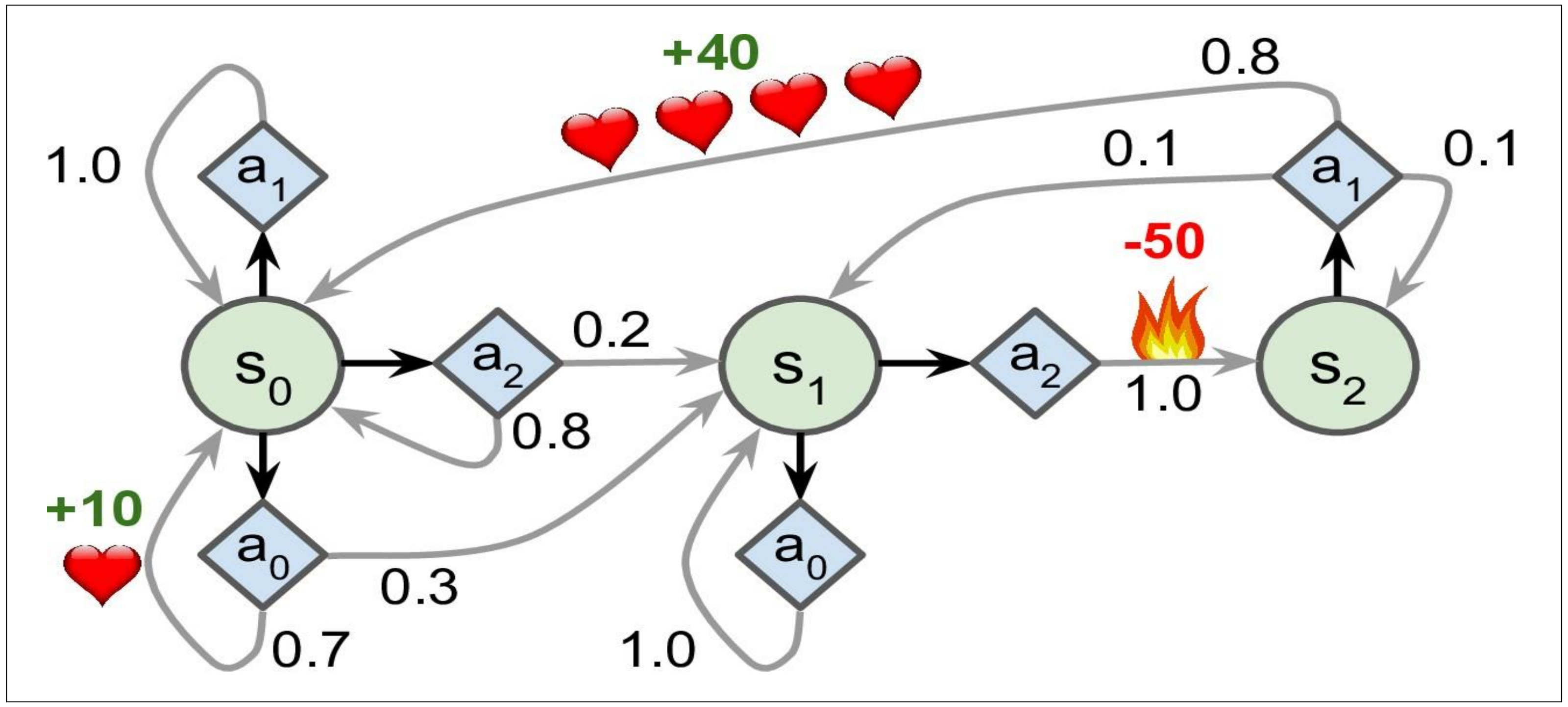
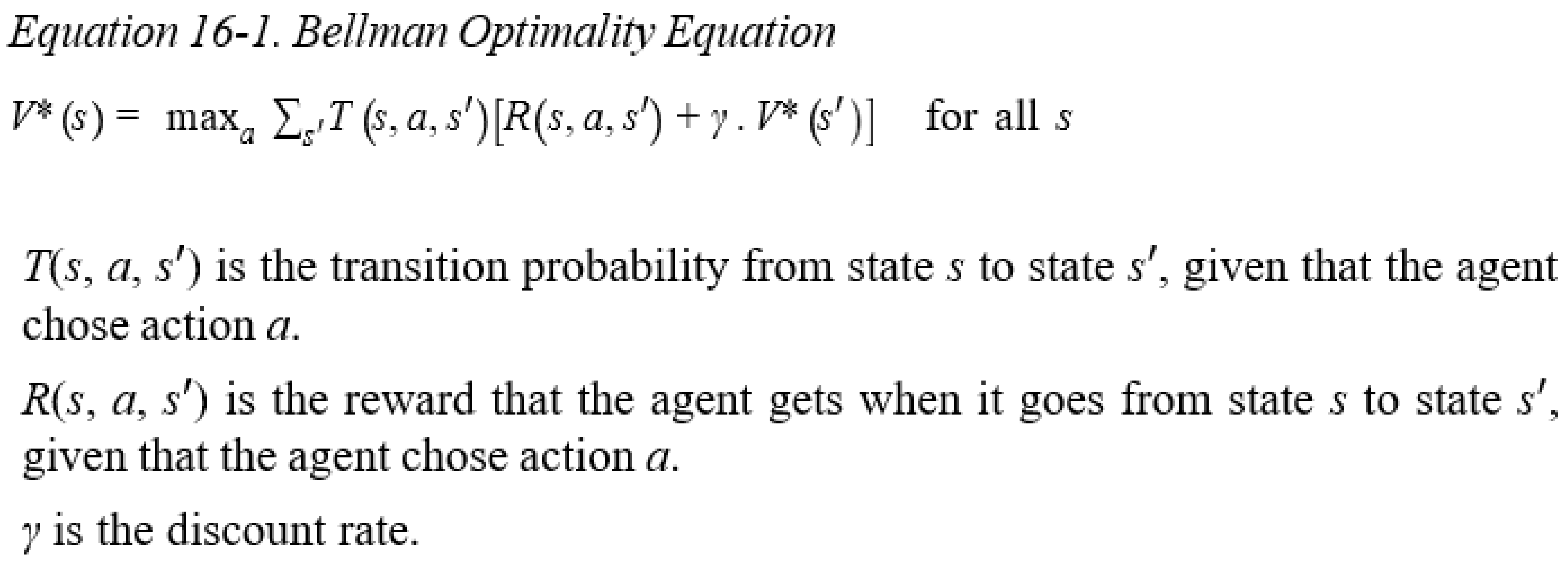


Figure 16-8. Example of a Markov decision process

Markov Decision Processes

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).



Equation 16-1. Bellman Optimality Equation

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

- $T(s, a, s')$ 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.

Markov Decision Processes

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 pol icy.

Equation 16-2. Value Iteration algorithm

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

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

This algorithm is an example of *Dynamic Programming*

Markov Decision Processes

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

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

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: $\pi^*(s) = \underset{a}{\operatorname{argmax}}\ Q^*(s,a)$.

Markov Decision Processes

Let’s apply this algorithm to the MDP represented in Figure 16-8. First, we need t
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]]
```

Markov Decision Processes

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])
```

Markov Decision Processes

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])
```

Markov Decision Processes

Result

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.