TÉCNICO LISBOA

Markov decision process (MDP)

AASMA 2024

Alberto Sardinha jose.alberto.sardinha@tecnico.ulisboa.pt
Rui Prada rui.prada@tecnico.ulisboa.pt
Manuel Lopes manuel.lopes@tecnico.ulisboa.pt

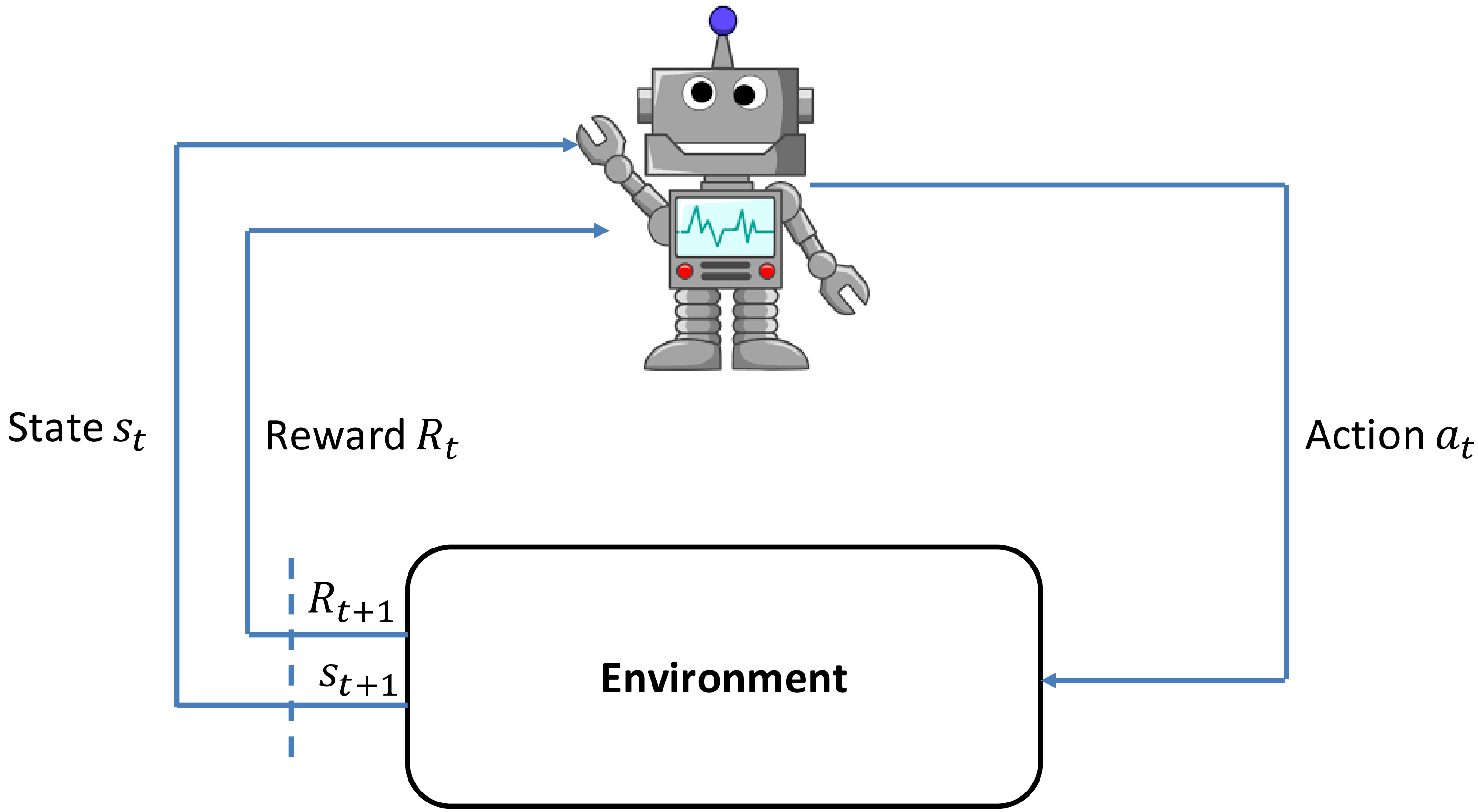
Outline

- **Single-agent learning**

Markov decision process (MDP)

- A framework for **sequential decision making** of **a single agent**
- **Markovian transition model** and **fully observable**
  - i.e., we assume it verifies the Markov property
- **Planning horizon** can be infinite

Markov decision process (MDP)



Markov decision process (MDP)

- We can formally define an MDP with following elements:
  - **Discrete time** $t = 0, 1, 2, \dots$
  - A **discrete set of states** $s \in S$
  - A **discrete set of actions** $a \in A$
  - A **stochastic transition model** $P(s'|s, a)$
    - the world transitions stochastically to state s' when the agent takes action a at state s
  - A **reward function** $R: S \times A \to \mathbb{R}$
    - An agent receives a reward $R(s, a)$ when it takes action a at state s

Markov decision process (MDP)

- **Definition**: the **state-value function** of a state s under a policy π is the expected return the agent can receive when starting in state s and then following policy π :

$$V^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t\,, a_t) | s_0 = s, a_t = \pi(s_t)\right]$$

Definition: the **action-value function (Q-values)** of taking an action a in state s under a policy π is the expected return the agent can receive when starting in state s, taking action a, and then following policy π :

$$Q^{\pi}(s,a) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t\,, a_t) | s_0 = s, a_0 = a, a_{t>0} = \pi(s_t)\right]$$

Markov decision process (MDP)

- A **policy π is defined to be better than or equal to a policy π'** if the expected return of π is greater or equal to expected return of π':

$$\pi \geq \pi' \text{ if and only if } V^{\pi}(s) \geq V^{\pi'}(s), \text{ for all } s \in S$$

- **There is always at least one policy** that is better than or equal to all other policies, which is the **optimal policy**
  - Note that an MDP might have more than one optimal policy
  - We denote all the optimal polices by π^*

Markov decision process (MDP)

- These policies π^* share the same state-value function, called **optimal state-value function**, with the following definition:

$$V^*(s) = \max_{\pi} V^{\pi}(s), \text{ for all } s \in S$$

- These policies π^* also share the same action-value function, called **optimal action-value function**, with the following definition:

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a), \text{ for all } s \in S \text{ and } a \in A$$

Value Iteration

- We initialize arbitrarily a state-value function (e.g., with zeros, ones, etc.)
- Then we iteratively apply the Bellman equation turned into an assignment operation:

$$Q(s,a) \coloneqq R(s,a) + \gamma \sum_{s' \epsilon S} P(s'|s,a)V(s'),\ \forall s, \forall a$$

$$V(s) \coloneqq \max_{a \in A} Q(s,a), \forall s$$

- Repeat the above two equations until V does not change significantly between two consecutive steps

Value Iteration

- Value iteration converges to the optimal Q^* for any initialization

- After computing the optimal Q^*, we can extract the policy as follows:

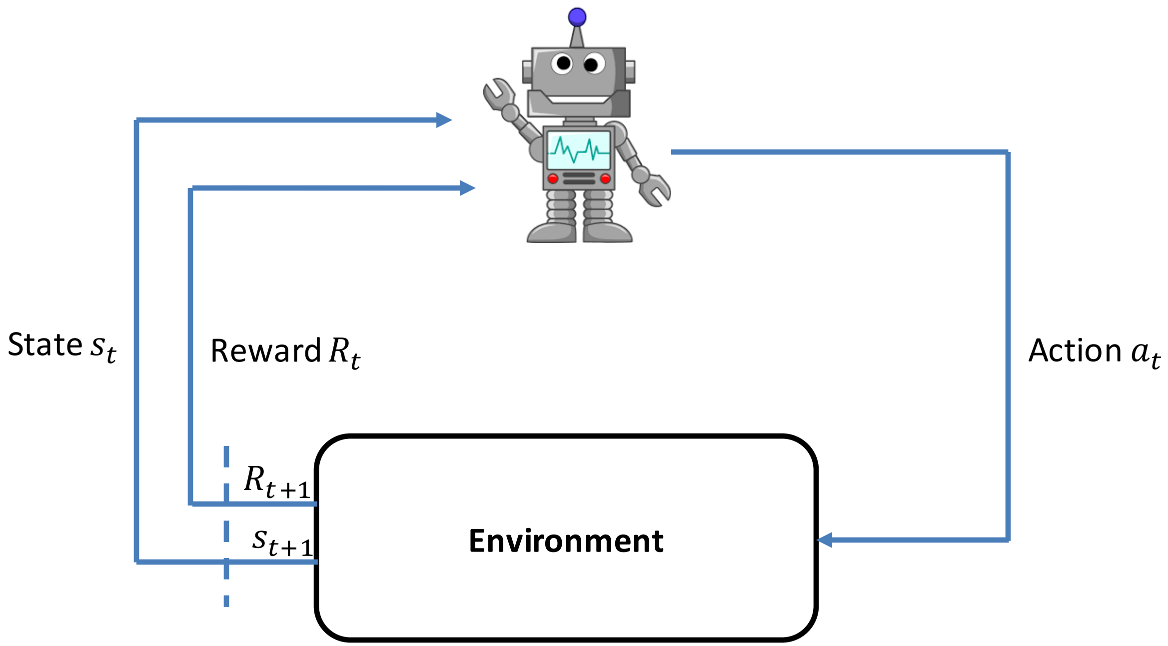
$$\pi^*(s) \in \operatorname*{argmax}_{a \in A} Q^*(s, a)$$

Markov decision process (MDP)

- What happens if we **do not know the stochastic transition model** and **reward function**?
  - We can use **Reinforcement Learning**

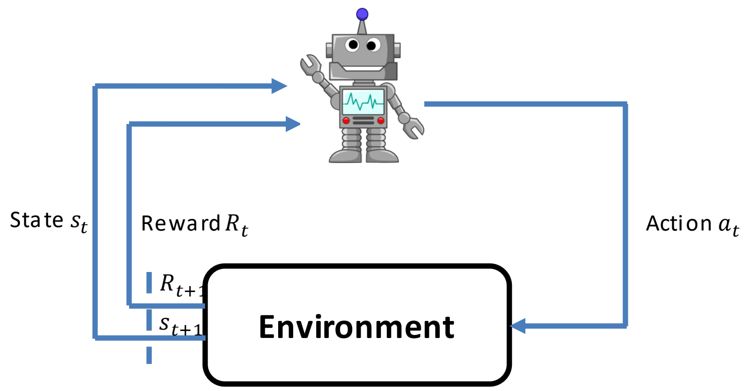
Reinforcement Learning

- Why not **interact with the environment**?



Reinforcement Learning

- Why not **interact with the environment**?
  - At each time step t
    - The agent observes the state s_t
    - The agent takes action a_t
    - The agent observes a reward R_t and the new state s_{t+1}



Reinforcement Learning

- Thus, my data point at each iteration is:

$$(s_t, a_t, R_t, s_{t+1})$$

- And what is the agent's goal?
  - Compute the **optimal policy of the MDP** with the data points

- Today, we focus on the most famous RL algorithm: **Q-learning**

Reinforcement Learning

Q-learning

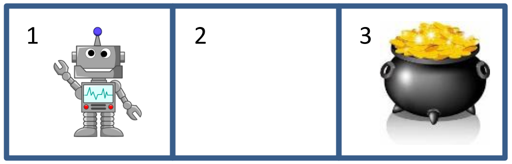
- We start with some estimate Q
- Initialize current state s
- Loop for each step:
  - choose some action a (e.g., using ϵ-greedy)
  - Take action a and observe next state s' and reward r
  - Update Q estimate according to
    $$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$
  - $s \leftarrow s'$

Reinforcement Learning

- Important observations about Q-Learning:
  - Update depends on previous estimate (*bootstrap*)
  - Update rule **propagates** reward information
  - To compute Q, algorithm must visit **every** state-action

Example

- We have the following MDP:
  - $S = \{1, 2, 3\}$
  - $A = \{left, right\}$
  - $\boldsymbol{P(s'|s, a = left) = ?}$
  - $\boldsymbol{P(s'|s, a = right) = ?}$
  - $\boldsymbol{R(s, a) = ?}$
  - $\gamma = 0.9, \alpha = 0.3$



Example

- We start with some estimate Q

$$Q = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}$$

- Initialize current state s

$$s \rightarrow 1$$

Example

- First iteration (current state $s \rightarrow 1$)
  - choose some action a
    - $a \rightarrow left$
  - Take action a and observe next state s' and reward r
    - $s' \rightarrow 1$
    - $r \rightarrow 0$
  - Update Q estimate according to
    - $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
    - $Q(1, left) \leftarrow 1 + 0.3[0 + 0.9 \times 1 - 1]$
    - $Q(1, left) \leftarrow 0.97$

Example

- Updated Q

$$Q = \begin{bmatrix} 0.97 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}$$

Example

- Second iteration (current state $s \rightarrow 1$)
  - choose some action a
    - $a \rightarrow right$
  - Take action a and observe next state s' and reward r
    - $s' \rightarrow 2$
    - $r \rightarrow 0$
  - Update Q estimate according to
    - $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
    - $Q(1, right) \leftarrow 1 + 0.3[0 + 0.9 \times 1 - 1]$
    - $Q(1, right) \leftarrow 0.97$

Example

- Updated Q

$$Q = \begin{bmatrix} 0.97 & 0.97 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}$$

Example

- Third iteration (current state $s \rightarrow 2$)
  - choose some action a
    - $a \rightarrow left$
  - Take action a and observe next state s' and reward r
    - $s' \rightarrow 1$
    - $r \rightarrow 0$
  - Update Q estimate according to
    - $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
    - $Q(2,left) \leftarrow 1 + 0.3[0 + 0.9 \times 0.97 - 1]$
    - $Q(2,left) \leftarrow 0.9619$

Example

- Updated Q

$$Q = \begin{bmatrix} 0.97 & 0.97 \\ 0.9619 & 1 \\ 1 & 1 \end{bmatrix}$$

Example

- Updated Q (after many iterations)

$$Q = \begin{bmatrix} 6.86 & 7.99 \\ 7.22 & 8.91 \\ 9.2 & 10 \end{bmatrix}$$

- This is Q^* with Value Iteration

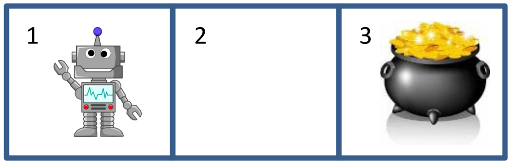
$$Q^* = \begin{bmatrix} 6.89 & 7.66 \\ 7.08 & 8.73 \\ 9.07 & 9.95 \end{bmatrix}$$

Example

- After computing Q, we can extract the policy as follows:

$$\pi^*(s) \in \operatorname*{argmax}_{a \in A} Q(s, a)$$

$$\pi^* = \begin{bmatrix} right \\ right \\ right \end{bmatrix}$$



$$\pi^* = \begin{bmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}$$

Example

```
import numpy as np
np.set_printoptions(precision=2, suppress=True)

# States
S = ['1', '2', '3']

# Actions
A = ['L', 'R']

# Transition probabilities

L = np.array([[1.0, 0.0, 0.0],
              [0.8, 0.2, 0.0],
              [0.0, 0.8, 0.2]])

R = np.array([[0.2, 0.8, 0.0],
              [0.0, 0.2, 0.8],
              [0.0, 0.0, 1.0]])

P = [L, R]

# Reward function

R = np.array([[0.0, 0.0],
              [0.0, 0.0],
              [1.0, 1.0]])

gamma = 0.9
```

Example

```python
def egreedy(Q,state,eps):

    p = np.random.random()

    if p < eps:
        action = np.random.choice(num_actions)
    else:
        action = np.argmax(Q[state,:])

    return action
```

Example

```
STEPS = 1000000
num_actions = len(A)
num_states = len(S)
ALPHA = 0.3

# Initialize Q-values
Q = np.ones((num_states, num_actions))

# Initialize current state
state = 0

for t in range(STEPS):

    # choose action
    action = egreedy(Q,state,0.05)

    # choose next state
    next_state = np.random.choice(num_states, p=P[action][state, :])

    # obtain reward
    reward = R[state,action]

    # Update Q
    Q[state, action] = Q[state, action] + ALPHA*(reward + gamma*max(Q[next_state, :]) - Q[state, action])

    state = next_state

print(Q)
```