[S&B Book] Chapter 3: Finite Markov Decision Process

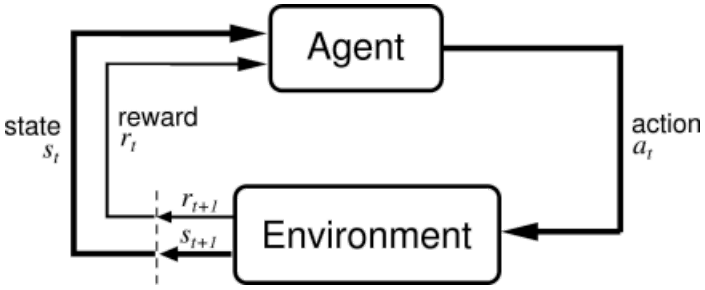
Tags

- **The definition of MDP:**

  MDP stands for Markov Decision Process, which is a mathematical framework used to model decision-making situations where outcomes are partly random and partly under the control of a decision-maker. In an MDP, an agent makes a series of decisions that affect a system with uncertain outcomes, and the goal is to maximize some notion of cumulative reward over time.

- MDPs involve delayed reward and the need to trade of immediate and delayed reward.

▼ 3.1 The Agent-Environment Interface



The agent-environment interaction in a Markov Decision Process

- A sequence of trajectory: $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \dots$

- ***finite* MDP:** In a MDP, the sets of state, action, and reward all have finite number of elements;
- ***dynamics* of the MDP:** (four-argument dynamics function)

  $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0,1], \quad \sum_{s' \in S} \sum_{r \in R} p(s', r|s, a) = 1$

  $$p(s', r|s, a) \doteq \Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}$$

  *** $\doteq$: equality relationship that is true by definition*

- From the four-argument dynamics function, one can compute anything else one might want to know about the environment:
  - *state-transition probability (three-argument function $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0,1]$):*

    $$p(s'|s, a) \doteq \Pr\{S_t = s'|S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in R} p(s', r|s, a)$$

  - *expected rewards for state-action pairs (two argument function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$):*

    $$r(s, a) \doteq \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a] = \sum_{r \in R} r \sum_{s' \in S} p(s', r|s, a)$$

  - *expected rewards for state-action-next-state triples (three argument function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$):*

    $$r(s, a, s') \doteq \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a, S_t = s'] = \sum_{r \in R} r \frac{r(s', r|s, a)}{p(s'|s, a)}$$

- **The MDP framework is a considerable abstraction of the problem of goal-directed learning from interaction. It can be applied to may different problems in many different ways.**

The limitation of the MDP framework:

1. *Non-stationary environments*: MDPs assume that the environment's dynamics and reward functions are stationary, meaning they do not change over time. If the environment is non-stationary, the MDP framework might not adequately represent the task. In these cases, adaptive or online learning techniques might be more appropriate.

▼ 3.2 Goals and Rewards

- **Reward:** The purpose or goal of the agent been formalized in terms of a special signal.
- Reward hypothesis:

> That all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of received scalar signal (call reward).

- Instance of using negative reward:

> In making a robot learn how to escape from a maze, the reward is often -1 for every time step that passes prior to escape; this encourages the agent to escape as quickly as possible.

- **The importance of defining a real goal:**

> A cheese-playing agent should be rewarded only for actually winning, not for achieving subgoals such as taking its opponents pieces or aiming control of the center of the board.

▼ 3.3 Returns and Episodes

- **The objective of learning:** **maximizing the *expected return***
- **Return**: G_t, is some function of the reward sequence. In the simplest case, it is the sum of the rewards:

$$G_t = R_{t+1} + R_{t+2} + \cdots + R_T \quad T : \text{final time step}$$

- **Episode:**

  The subsequences of the agent-environment interaction. Each episode ends in a special state called the *terminal state.*

- **Episodic Task:**

  The episode can all be considered to end in the same terminal state, with different rewards for the different outcomes. Tasks with episodes of this kind are called *episodic tasks.*

  $\mathcal{S}$: the set of all nonterminal states; $\mathcal{S}^+$: the set of all states plus the terminal state.

- **Discounting:** Avoid infinite reward or infinite time steps

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$
$$\gamma : \text{discount rate} \quad 0 \le \gamma \le 1$$

  Note that because the returns at successive time steps are related to each other in a way that is important for the theory and algorithms of reinforcement learning:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots \\ &= R_{t+1} + \gamma (R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \dots) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned}$$

  This often makes it easy to compute returns from reward sequences.

Exercise:

Suppose $\gamma = 0.5$ and the following sequences of rewards is received $R_1 = -1, R_2 = 2, R_3 = 6, R_4 = 3, R_5 = 2$, with $T = 5$. What are $G_0, G_1, \ldots, G_5$?

Solution:

```
T = 5
R = [-1, 2, 6, 3, 2] # Rewards starts from t=1
G = [0] * (T + 1) # Returns from t=0 to t=T

gamma = 0.5

for i in range(len(R) - 1, -1, -1):
  G[i] = R[i] + gamma * G[i + 1]

print(G)
# Output: [2.0, 6.0, 8.0, 4.0, 2.0, 0]
```

- Although the return is a sum of an infinite number of terms, it is still finite if the reward is nonzero and constant, when $\gamma < 1$. For instance, if the reward is 1, then:

$$G_t = \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma}$$

Proof of the second equality of this function:

$$\sum_{k=0}^{\infty} \gamma^k (1-\gamma) = 1$$

$$\sum_{k=0}^{\infty} \gamma^k - \gamma^{k+1} = \gamma^0 - \lim_{k \to \infty} \gamma^{k+1} = 1 - 0 = 1$$

Exercise:

Suppose $\gamma = 0.9$, and the reward sequence is $R_1 = 2$ followed by an infinite sequence of 7s. What are G_0, G_1?

Solution:

$$G_1 = \text{inf\_reward} \times \frac{1}{1-\gamma} = 7 \times \frac{1}{1-0.9} = 70$$

$$G_0 = R_1 + \gamma G_1 = 2 + 0.9 \times 70 = 65$$

▼ 3.5 Policies and Value Functions

- ***value functions:***

  Functions of states (or state-action pairs) that estimate *how good (expected return)* it is for the agent to be in a given state.

- ***policy:***

  A mapping from states to probabilities of selecting each possible action;

  $\pi(a|s)$ → a probability of $A_t = a$ if $S_t = s$ at time t.

- **A value function of a state s under a policy π**

$$v_\pi \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right], \text{for all } s \in S$$

  $\mathbb{E}_\pi[.]$ denotes the expected value of a random variable given that the agent follows policy π, and t is any time step.

- **action-value function for policy π**

$$q_\pi(s,a) \doteq \mathbb{E}[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a\right]$$

Exercise:

1. Given an equation for v_π in terms of q_π and π;
2. Given an equation for q_π in terms of v_π and the four-argument p;

Solution:

$$1. \quad v_\pi = \sum_a \pi(a|s) q_\pi(s,a)$$

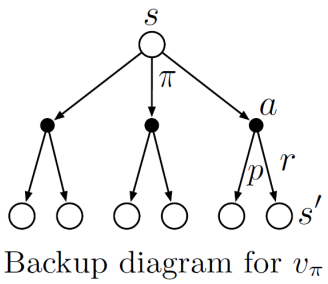
$$2. \quad q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')]$$

- **Bellman equation:**

  The Bellman equation expresses a relationship between the value of a state and the values of its successor states.

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')], \quad \text{for all } s \in \mathcal{S} \end{aligned}$$

- **Backup diagram**

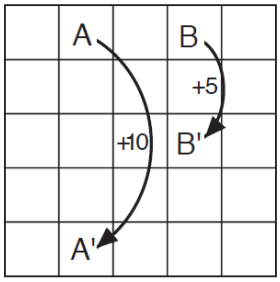


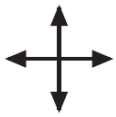
Backup diagram for v_π

Each open circle represents a state and the values of its successor states. Starting from state s, the root node at the top, the agent could take any of some set of actions based on its policy π. From each of these, the environment could respond with one of several next states s', along with a reward r, depending on its dynamics given by the function p.

Example: Gridworld

- Actions: `north, south, east, west`
- Rewards:
  - Actions would take the agent off the grid leave its location unchanged, but also results in a reward of -1;
  - Other actions result in a reward of 0, except those that in states `A` and `B`;
  - From state `A`, all four actions yield a reward of +10 and take the agent to `A'`;
  - From state `B`, all four actions yield a reward of +5 and take the agent to `B'`;
  - The learning rate (gamma) for this example is 0.9




Actions

3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

Implementation:

```
import numpy as np
import seaborn as sns
from matplotlib import pyplot as plt

# get a reward based on the state and action
def get_reward(state, act):
    '''
    state: current state, tuple (row, col)
    act: action, tuple (d_row, d_col)
```

```
    return: reward and next state
    '''
    if state == (0, 1):
        return 10, (4, 1)
    if state == (0, 3):
        return 5, (2, 3)

    next_row = state[0] + act[0]
    next_col = state[1] + act[1]

    if not (0 <= next_row < 5 and 0 <= next_col < 5):
        return -1, state

    return 0, (next_row, next_col)

# value function
def value_update(grid_world):
    '''
    grid_world: the 5x5 grid world map, numpy array
    '''
    for row in range(5):
        for col in range(5):
            curr_value = 0
            for act in actions:
                reward, next_state = get_reward((row, col), act)
                next_value = grid_world[next_state]
                curr_value += 0.25 * (reward + gamma * next_value)
            grid_world[row, col] = curr_value

# plot the value table
def plot_grid(grid_world, iteration=0):
    annot_kwargs = {
        'fontsize': '18'
    }
    plt.figure(figsize=(6, 6), dpi=100)
    sns.heatmap(grid_world,
                annot=True,
                annot_kws=annot_kwargs,
                linewidths=1.,
                fmt='.1f',
                cmap='crest',
                cbar=False)
    plt.tick_params(
                bottom=False,
                labelbottom=False,
                left=False,
                labelleft=False)
    plt.title(f'Iteration {iteration}', fontsize=20, fontweight='bold', pad=10)
    plt.savefig(f'./plots/example_3_5/{iteration}.png')
    # plt.show()


if __name__ == "__main__":
    # Initialize the grid world
    grid_world = np.zeros(shape=(5, 5))
    actions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    gamma = 0.9

    iter_to_save = {0, 1, 4, 10, 32}

    for i in range(40):
        prev_grid = np.copy(grid_world)
        value_update(grid_world)
        if i in iter_to_save:
            plot_grid(grid_world, i)
        if np.allclose(prev_grid, grid_world, rtol=0.01):
            print('done', i)
```

```
            plot_grid(grid_world, i)
            break
```

result:

▼ 3.6 Optimal Policies and Optimal Value Functions

- ***Optimal policy:***

  There is always at least one policy that is better than or equal to all other policies by π_*.

  The optimal policies share the same *optimal state-value function*, denote v_*, and define as:

$$v_* \doteq \max_\pi v_\pi(s)$$

  They also share the same *optimal action-value function*, denote q_*, and define as:

$$q_* \doteq \max_\pi q_\pi(s, a)$$

  writing q_* in terms of v_*:

$$q_*(s, a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a]$$

- Bellman optimality equation:

$$
\begin{aligned}
v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\
&= \max_a \mathbb{E}_{\pi_*}[G_t | S_t = s, A_t = a] \\
&= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\
&= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\
&= \max_a \sum_{s',r} p(s', r | s, a)[r + \gamma v_*(s')]
\end{aligned}
$$

$$
\begin{aligned}
q_*(s,a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a] \\
&= \sum_{s',r} p(s', r | s, a)[r + \gamma \max_{a'} q_*(s', a')]
\end{aligned}
$$

Example: Solving the Gridworld

Solving the Bellman equation for v_* for the simple Gridworld example;

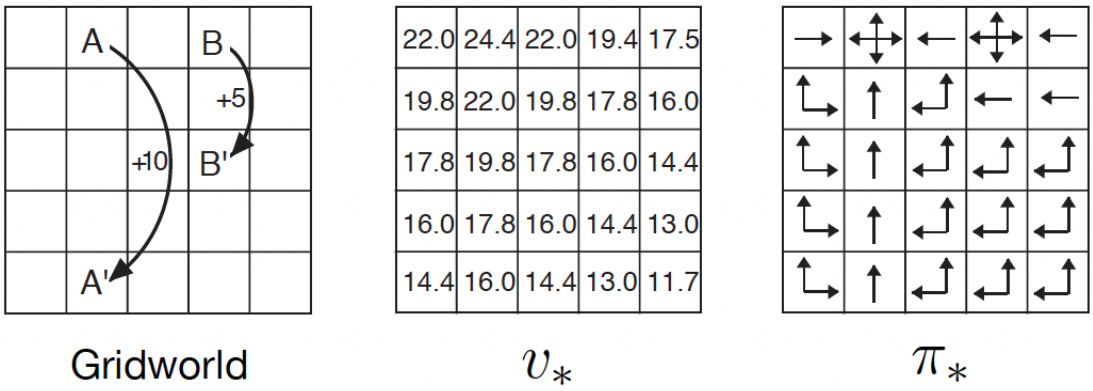


Figure 3.5: Optimal solutions to the gridworld example.

Implementation:

```
# value function
def value_update(grid_world, actions):
    '''
    grid_world: the 5x5 grid world map, numpy array
    '''
    # Hex code for (right, down, left, up)
    act_lists = np.array([0x2190, 0x2193, 0x2192, 0x2191])
    optim_acts = []

    for row in range(5):
        acts = []
        for col in range(5):
            value_candidates = np.zeros(shape=(len(actions)))
            # Iterate over all four actions
            for i , act in enumerate(actions):
                reward, next_state = get_reward((row, col), act)
```

```
                next_value = grid_world[next_state]
                # Get a discounted next value for current action
                value_candidates[i] = reward + gamma * next_value

            # Bellman optimal equation
            grid_world[row, col] = value_candidates.max()

            # Finding the optimal action(s)
            max_args = np.where(value_candidates == value_candidates.max())
            selected_acts = ''.join([chr(c) for
                                     c in act_lists[max_args].tolist()])
            acts.append(selected_acts)

        optim_acts.append(acts)

    return optim_acts
```

result:

- **Solving the Bellman optimality equation relies on at lease three assumptions that are rarely in practice:**
  1. The dynamics of the environment are actually known;
  2. The computational resource are sufficient to complete the calculation;
  3. The states have Markov property;

▼ 3.7 Optimality and Approximation

- It is usually **not possible** to simply compute an optimal policy by solving the Bellman optimal equation. A critical aspect of the problem facing the agent is always the computational power available to it, in

particular, the amount of computation it can perform in a single time step.

- *Tabular method*:

  In tasks with small, finite state sets, it is possible to form the optimal policy approximations using arrays or tables with one entry for each state (or state-action pair). This we call the *tabular* case, and corresponding methods we call tabular methods.

- **Approximation:**

  The online nature makes it possible to approximate optimal policies in ways that put more efforts into learning to make good decisions for frequently encountered states, at the expense of less effort for infrequently encountered states.

  This is one key property that distinguishes reinforcement learning from other approaches to approximated solving MDPS.