

Report for Assignment 3

519021910913 黃喆敏

Directory Structure

```
lab3
├── code
│   ├── CliffWalking.py
│   ├── QLearning.py
│   ├── Sarsa.py
│   └── main.py
└── docs
    ├── assets
    │   ├── eps_0.png          # comparison between SARSA and Q-Learning
    │   ├── eps_1.png
    │   ├── eps_2.png
    │   ├── heatmap             # heatmap of Q matrix
    │   │   ├── q_learning_0.png
    │   │   ├── q_learning_1.png
    │   │   ├── q_learning_2.png
    │   │   ├── sarsa_0.png
    │   │   ├── sarsa_1.png
    │   │   └── sarsa_2.png
    │   ├── policy               # policy generated by SARSA and Q-Learning
    │   │   ├── q_learning_0.png
    │   │   ├── q_learning_1.png
    │   │   ├── q_learning_2.png
    │   │   ├── sarsa_0.png
    │   │   ├── sarsa_1.png
    │   │   └── sarsa_2.png
    │   └── qlearning.png
    └── report.md
```

All codes are placed in `./code` directory. You can directly run `main.py` to observe the results of SARSA and Q-Learning methods.

Cliff Walking

In `cliffWalking.py`, I have built the Cliff Walking environment, which is quite similar to the previous GridWorld environment.

For visualization, I use `seaborn` library to generate heatmaps, to observe Q matrix. I also visualized paths generated by SARSA and Q-Learning methods. You can see the detail in `draw_path` and `draw_heatmap` function.

$\epsilon - greedy$ policy is a simple way to balance exploration and exploitation by choosing exploration and exploitation randomly. For $\epsilon - greedy$ policy,

$$\pi(a|s) = \begin{cases} \frac{\epsilon}{m} + 1 - \epsilon, & \text{if } a^* = \operatorname{argmax}_{a \in A} \\ \frac{\epsilon}{m}, & \text{otherwise} \end{cases}$$

We use $\epsilon - greedy$ policy to choose action in both SARSA and Q-Learning methods, and the implementation is mentioned below.

```
def eps_greedy(self, pos):
    p = np.random.random()
    if p < self.eps:
        return np.random.choice(4)
    else:
        idx = []
        mx = max(self.Q[pos[0], pos[1], :])
        for i in range(4):
            if self.Q[pos[0], pos[1], i] == mx:
                idx.append(i)
        return np.random.choice(idx)
```

Besides, we need to notice that stepping into 'The Cliff' region incurs a reward of -100 and sends the agent back to the start; otherwise, the reward is -1.

```
def move(self, state, idx):
    next_state = np.array(state) + np.array(self.action[idx])
    if not (0 <= next_state[0] <= self.n - 1 and 0 <= next_state[1] <= self.m - 1):
        next_state = state
    reward = self.usual_reward
    if next_state[0] == self.n - 1 and 0 < next_state[1] < self.m - 1:
        next_state = self.start
        reward = self.cliff_reward
    return next_state, reward
```

Algorithm Details

SARSA Method

The SARSA method is an on-policy algorithm for TD-Learning. New action a' is chosen **using the same $\epsilon - greedy$ policy** as the action a , the one that generated s' . For SARSA method, the algorithm can be given as follows.

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\text{terminal-state}, \cdot) = 0$
 Repeat (for each episode):

 Initialize S

 Choose A from S using policy derived from Q (e.g., ε -greedy)

 Repeat (for each step of episode):

 Take action A , observe R, S'

 Choose A' from S' using policy derived from Q (e.g., ε -greedy)

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$$

$$S \leftarrow S'; A \leftarrow A';$$

 until S is terminal

In `sarsa.py`, I have implemented the Sarsa method. The code is mentioned below.

```
def work(self):
    episode = 1
    while episode <= self.num + 10:
        state = self.env.start
        action = self.env.eps_greedy(state)
        num = 0
        total_reward = 0
        while not self.env.is_terminal(state):
            num += 1
            next_state, reward = self.env.move(state, action)
            total_reward += reward

            next_action = self.env.eps_greedy(next_state)
            self.env.Q[state[0], state[1], action] += self.alpha * (
                reward +
                self.gamma * self.env.Q[next_state[0], next_state[1], next_action]
                - self.env.Q[state[0], state[1], action])
            state = next_state
            action = next_action
        self.reward.append(total_reward)
        episode += 1

    for i in range(self.num):
        self.avg_reward.append(np.mean(self.reward[i:i + 10]))
```

Q-Learning Method

The Q-Learning method is called off-policy learning, because the new action is given as greedy, not using the current policy. Q-Learning learns the optimal policy even when actions are selected according to a more exploratory or even random policy. For Q-Learning method, the algorithm can be given as follows.

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\text{terminal-state}, \cdot) = 0$

Repeat (for each episode):

- Initialize S
- Repeat (for each step of episode):

 - Choose A from S using policy derived from Q (e.g., ε -greedy)
 - Take action A , observe R, S'
 - $$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$$
 - $S \leftarrow S'$;

- until S is terminal

In `QLearning.py`, I have implemented the Q-Learning method, which is similar to the code above.

```
def work(self):
    episode = 1
    while episode <= self.num + 20:
        state = self.env.start
        num = 0
        total_reward = 0
        while not self.env.is_terminal(state):
            num += 1
            action = self.env.eps_greedy(state)
            next_state, reward = self.env.move(state, action)
            total_reward += reward

            self.env.Q[state[0], state[1], action] += self.alpha *
                (reward + self.gamma * np.max(self.env.Q[next_state[0], next_state[1], :]) -
                 self.env.Q[state[0], state[1], action])
            state = next_state
        self.reward.append(total_reward)
        episode += 1

    for i in range(self.num):
        self.avg_reward.append(np.mean(self.reward[i:i + 10]))
```

It is crucial to point out that for both methods, we **smooth the result** by averaging the reward from 10 successive episodes'.

Hyperparameters

The hyperparameters we used in the Q-value update process are:

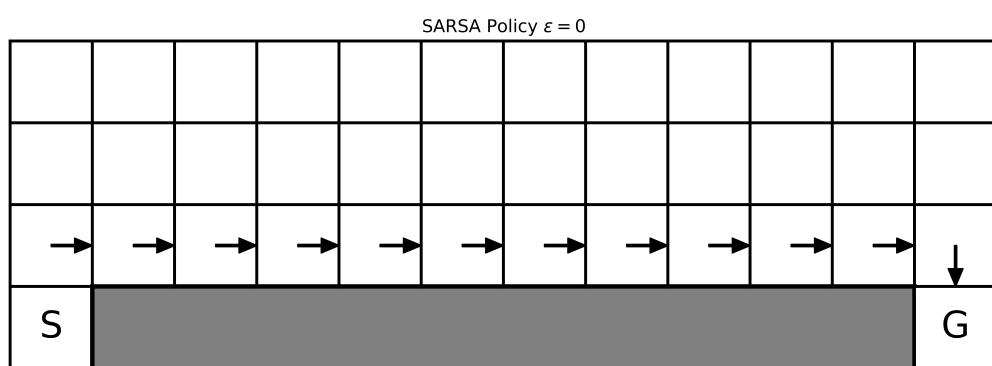
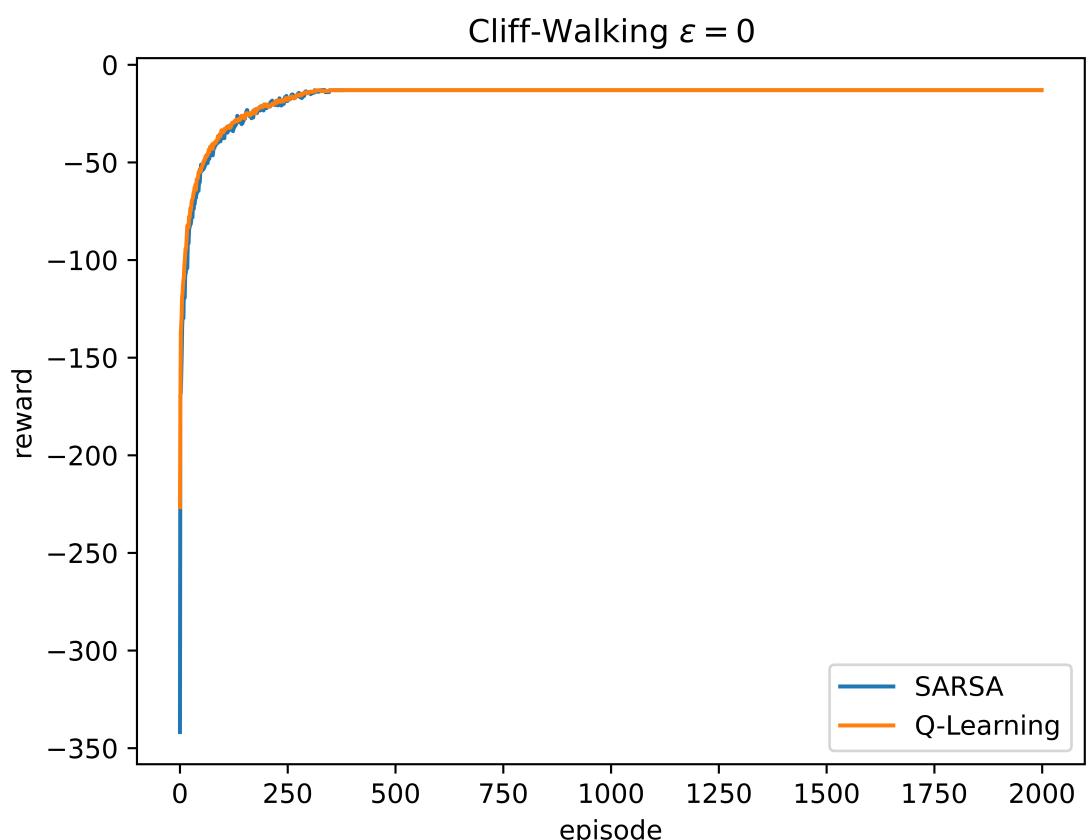
- α : the learning rate, set between 0 and 1. Setting it to 0 means that the Q-values are never updated. Since the number of episodes can be high, we set $\alpha = 0.1$.

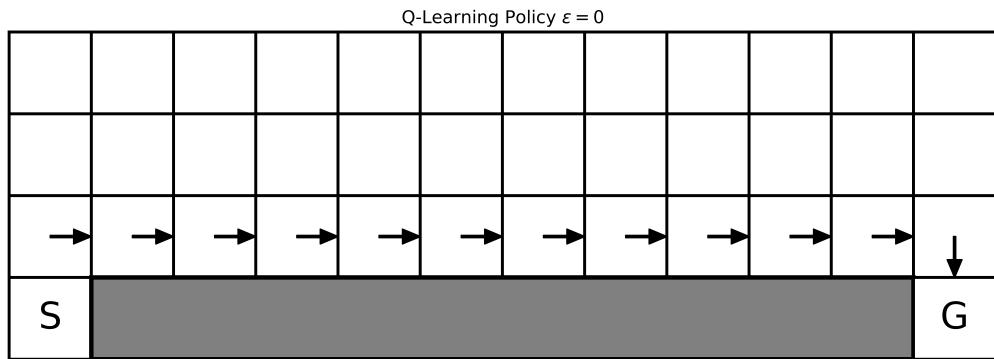
- γ : the discount factor, also set between 0 and 1. This models the fact that future rewards are worth less than immediate rewards. In this assignment, we set **gamma = 0.95**, which has a good performance.
- ϵ : the probability used in $\epsilon - greedy$ policy. Different settings for ϵ can bring different exploration on policy update. Therefore, I have tried **$\epsilon = 0, \epsilon = 0.1, \epsilon = 0.3$** .

Experiment Process

$\epsilon = 0$

For $\epsilon = 0$, **both methods have chosen the optimal path**. Besides, both methods are converged quickly.



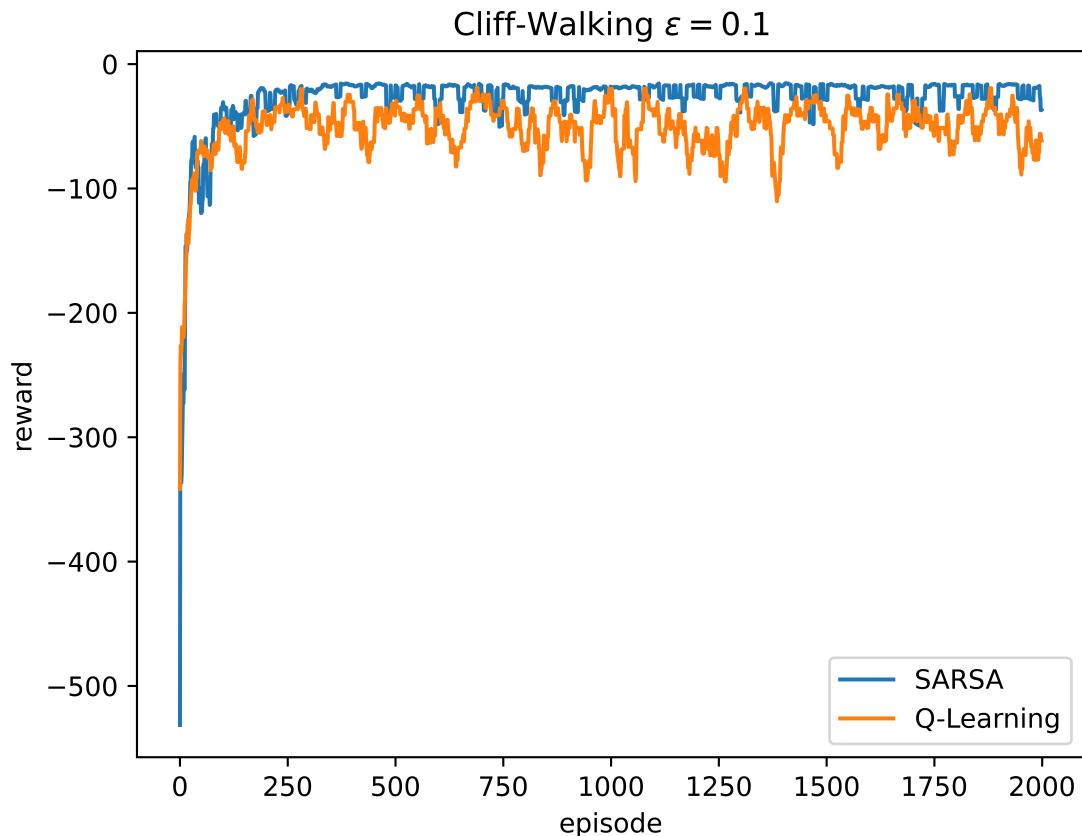


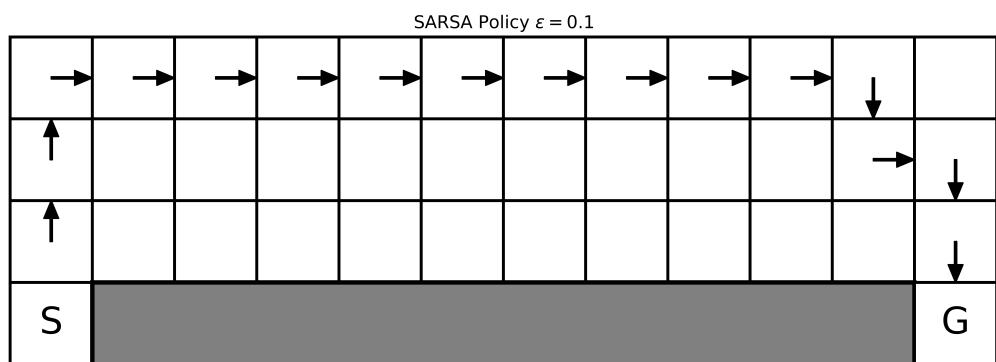
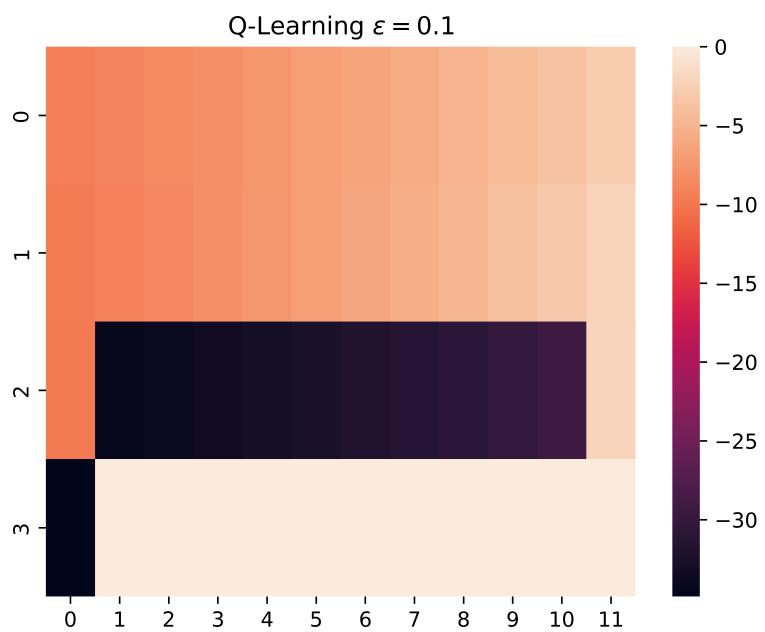
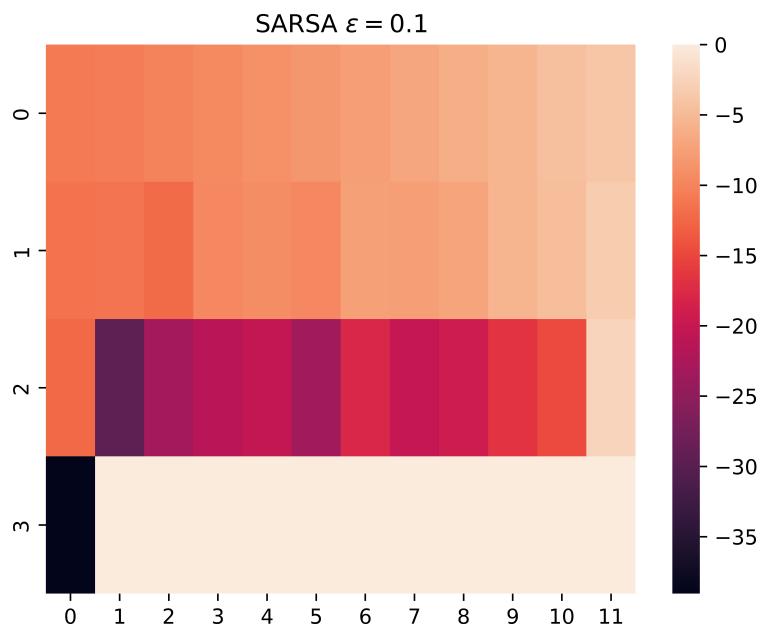
$$\epsilon = 0.1$$

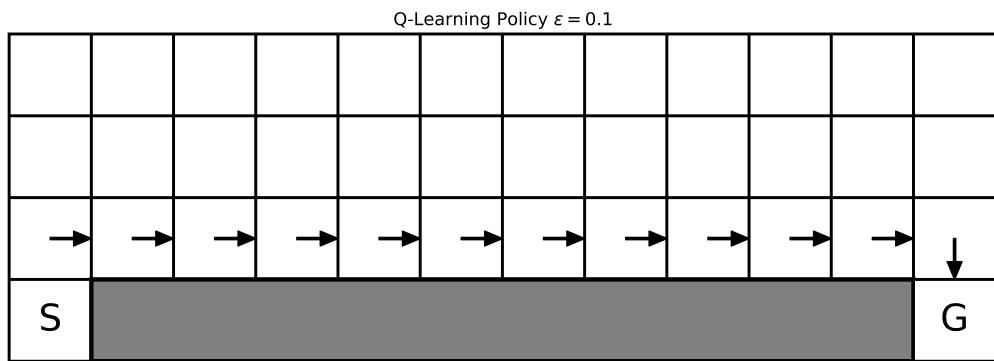
With respect to any $\epsilon - \text{greedy}$ policy with $\epsilon > 0$ **not so small**, the cells just above the cliff are dangerous and should have small value function for Sarsa method. Therefore, for $\epsilon = 0.1$, **Sarsa chooses a safer path, while Q-Learning chooses an optimal path.**

Besides, we can find that **the average reward of Q-Learning method is smaller than Sarsa's**. Because Sarsa learns the safer path, it actually receives a higher average reward per episode than Q-Learning even though it does not walk the optimal path.

We also drawn the heatmap of Q matrix for both methods. It is clear that for both methods, the average Q values of cells just beyond the cliff are much smaller, due to the danger.



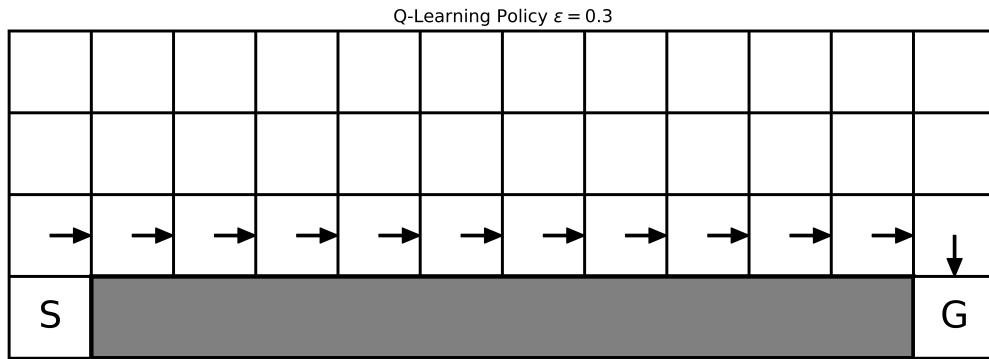
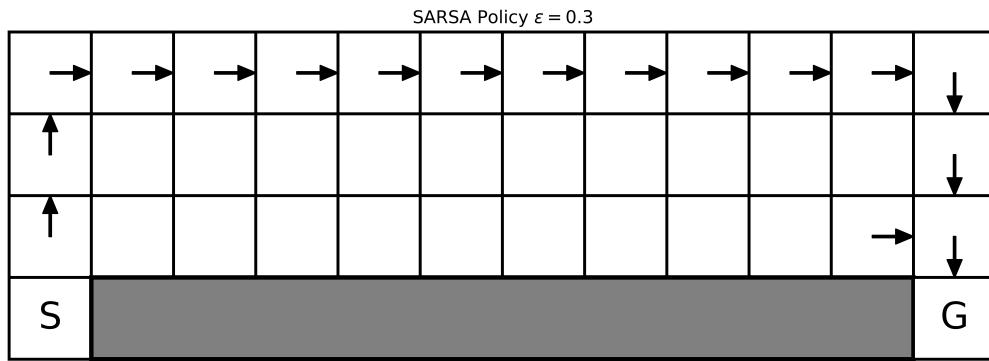




$\epsilon = 0.3$

For $\epsilon = 0.3$, both methods behave like $\epsilon = 0.1$. But rewards of both methods are much harder to converge. The reason is that when ϵ is larger, ϵ – *greedy* method is more likely to choose a random action, instead of the optimal action.





Comparison

Q-Learning learns the optimal policy, which moves along the cliff, but random exploration leads to higher chance of falling off. Thus, Q-Learning is less stable and has higher penalties.

Meanwhile, Sarsa is more conservative. It finds a safer, but not the optimal path, in which the agent prefers to go further away from the cliff.

In practical application, we tend to use Q-Learning method rather than Sarsa method, in order to find the optimal path. Besides, ϵ should not be too large, otherwise it would be more difficult to converge.