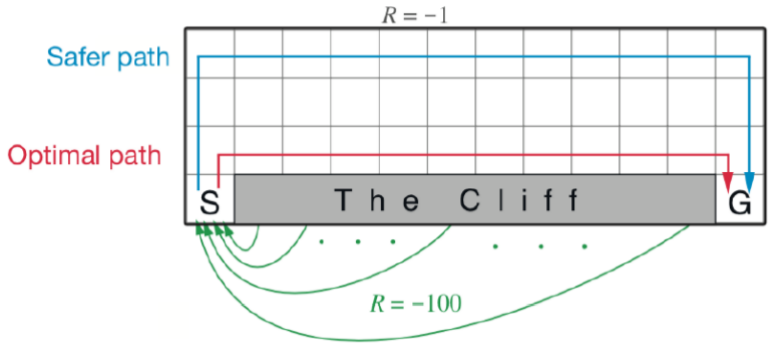
强化学习报告

基于 Cliff Walk 例子实现 Sarsa、Q-learning 算法

学院	计算机学院	姓名	张涛麟
班别	计科 1 班	学号	18351099

Cliff Walk



As shown in the figure above, the size of the entire map in the game is 4×12 grids. Grids with rows from 1 to 3 are flat, and the first gridin row 4 is the starting point, while the last square in row 4 is the end point. , The rest of grids in the 4th row is the cliff.
The character will start from the starting point and can move up, down, left, and right, and cannot go out of the map. The character will lose 1 point for each step, and will lose 100 points when it falls and return to the starting point until the character reaches the end and returns to the starting point.
The goal of this experiment is to use Q-learning and Sarsa to train the character's decision in the grids and finally visualize the results.

Analysis

We first need to construct a map. Here we directly use a two-dimensional array to construct, and use -1 to represent the cliff, 0 to represent the flat land, and 1 to represent the end point. In addition, it is also necessary to define a set of action strategies, respectively representing moving down, right, up and left.

```
graph = np.zeros((4, 12))  # graph
graph[3][1:11] = -1  # cliff
graph[3][11] = 1  # goal
# move down, right, up, left
actions = np.array([[1, 0], [0, 1], [-1, 0], [0, -1]])
```

For the strategy based on learning q function value, we can write a parent class to reduce repetitive code. The properties of the parent class are shown below.

```
class Strategy():
    def __init__(self, graph, actions, eposilon, alpha, gamma):
        self.graph = graph # graph
        self.actions = actions # actions set
        self.done = False # finished or not
        self.state = np.array([3, 0]) # current state
        self.action = 0 # current action
        self.eposilon = eposilon # greedy strategy parameter
        self.row, self.col = self.graph.shape # size of the graph
        self.rewards = 0 # the reward of current episode
```

```
        self.reward_list = np.zeros(0) # restore the reward of all the episodes
        self.q = np.zeros((4, 12, 4))  # q function
        self.alpha = alpha # learning rate
        self.gamma = gamma # discount factor
```

Next, we will design the functions that will be used in each episode as api to be called.
Every time an episode starts, the init function will initialize the relevant parameters.

```
def init(self): # init every episode
    self.done = False # restart
    self.state = np.array([3, 0]) # init state
    self.rewards = 0 # init reward
```

Each move returns the reward of the next state and action, which is implemented with the move function.

```
def move(self, state, action): # every step
    nstate = state + self.actions[action] # change state to next state(nstate)
    # ensure nstate is still in the graph
    if nstate[0] < 0:
        nstate[0] = 0
    if nstate[0] >= self.row:
       nstate[0] = self.row-1
    if nstate[1] < 0:
        nstate[1] = 0
    if nstate[1] >= self.col:
        nstate[1] = self.col-1
    # gain reward
    if self.graph[nstate[0]][nstate[1]] == 0: # normal grid
        reward = -1
    elif self.graph[nstate[0]][nstate[1]] == 1: # goal
        self.done = True # finish
        reward = -1
    elif self.graph[nstate[0]][nstate[1]] == -1: # cliff
        self.state = [3, 0] # go back to starting point
        reward = -100
    return nstate, reward
```

In addition, Q-learning and Sarsa methods both have a greedy strategy of $\epsilon - Greedy$, so we implement it directly in the parent class.

```
def epsilon_greedy(self, state): # epsilon-greedy to choose next action(naction)
    if np.random.random() < self.eposilon: # random choose
        naction = np.random.randint(4)
    else:
        naction = np.argmax(self.q[state[0]][state[1]]) # greedy
    return naction
```

The entire training process can be achieved through the following process.

```
stategy = stategy(graph, actions, eposilon, alpha, gamma)# define stategy
for episode in range(1, episodes+1):# each episode
    stategy.init()# init parameter
    stategy.action = sarsa.epsilon_greedy(sarsa.state) # choose first action based on
epsilon_greedy
    while not stategy.done: # keep moving until reach goal
        stategy.forward()
```

The only difference between Q-learning and Sarsa's algorithms is the strategy adopted when updating the q function value.

- The Q-learning algorithm adopts an absolute greedy strategy to update the q function value
- Sarsa algorithm continues to adopt the $\epsilon - Greedy$ strategy to update the q function value

Next, we will implement the details of Q-learning and Sarsa.

Q-learning

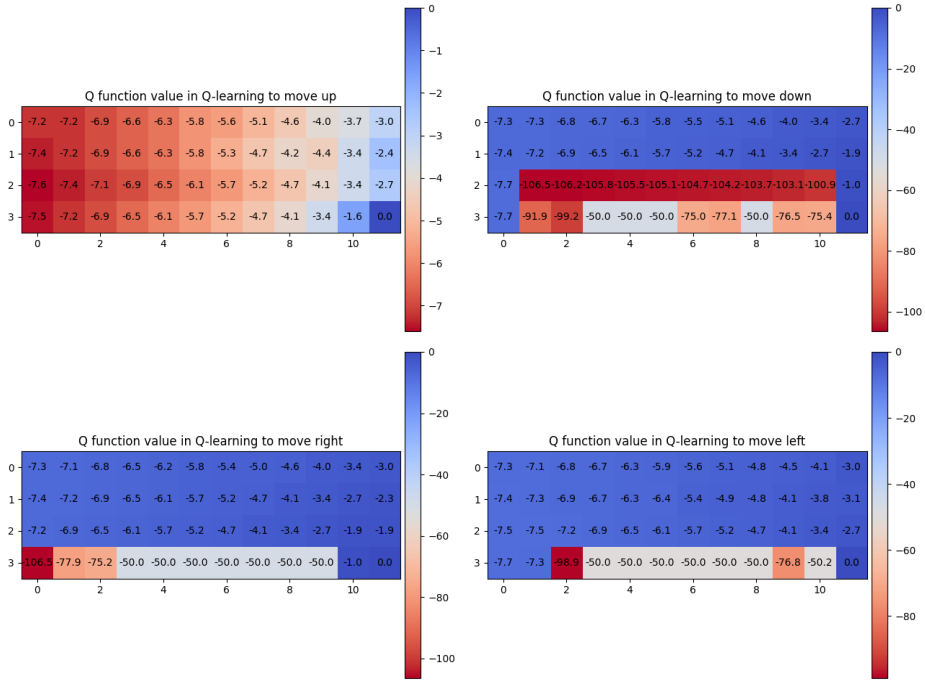
> The main algorithms of Q-learning are as follows.
>
> Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state,\cdot) = 0$
> Repeat (for each episode):
> Initialize S
> Repeat (for each step of episode):
> Choose A from S using policy derived from Q (e.g., ε-greedy)
> Take action A, observe R, S'
> $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
> $S \leftarrow S'$;
> until S is terminal
>
> After selecting action A based on $\epsilon - Greedy$ strategy , determine the next state S' and obtain the reward R.
> Then use an absolute greedy strategy to update the q function value of the current state.
> Lastly update state $S \leftarrow S'$

Code:

```
def forward(self): # Qlearning step
    state = self.state # current state
    action = self.epsilon_greedy(state) # action
    nstate, reward = self.move(state, action) # next action and reward
    self.q[state[0]][state[1]][action] += self.alpha * \
        (reward + self.gamma * np.max(self.q[nstate[0]][nstate[1]])-self.q[state[0]][state[1]]
[action]) # update q function value
    self.state = nstate # update state
    self.rewards += reward # gain reward
    if self.done: # finish
        self.reward_list = np.append(self.reward_list, self.rewards)
```

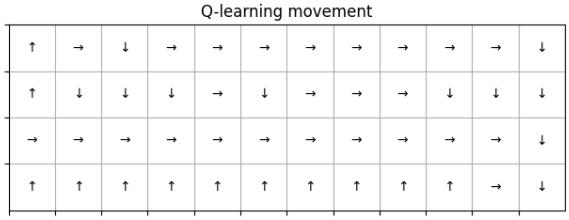
Result Analysis:

After training 500 episodes, we get the q function value of each state with different actions, as shown in the figure below.



After training 500 episodes, the optimal decision under different grids is shown in the figure below:

Q-learning movement



Sarsa

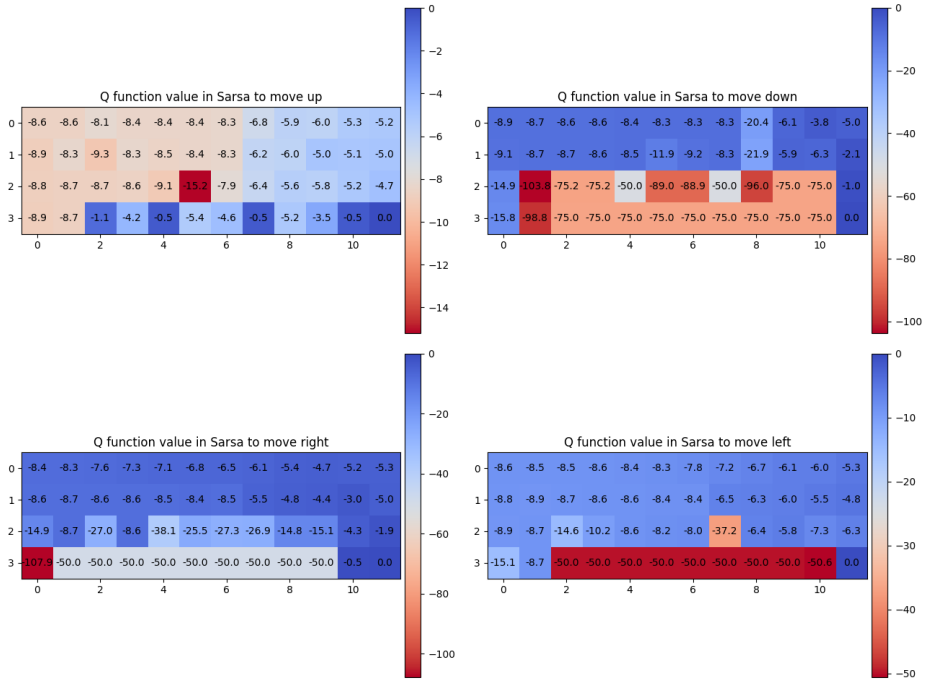
> The main algorithms of Sarsa are as follows.
>
> Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
> Repeat (for each episode):
> Initialize S
> Choose A from S using policy derived from Q (e.g., ε-greedy)
> Repeat (for each step of episode):
> Take action A, observe R, S'
> Choose A' from S' using policy derived from Q (e.g., ε-greedy)
> $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma Q(S',A') - Q(S,A)\right]$
> $S \leftarrow S'$; $A \leftarrow A'$;
> until S is terminal
>
> After executing the last round of greedy strategy based on $\epsilon - Greedy$ to select action A, determine the next state S' and obtain the reward R .
> Then continue to select the next round of action A' based on the $\epsilon - Greedy$ strategy , and update the q function value of the current state.
> Lastly update status $S \leftarrow S'$

Code:

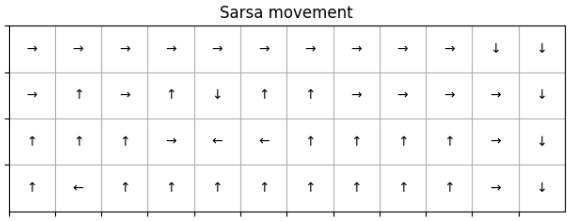
```
def forward(self): # Qlearning step
    state = self.state # current state
    action = self.epsilon_greedy(state) # action
    nstate, reward = self.move(state, action) # next action and reward
    self.q[state[0]][state[1]][action] += self.alpha * \
        (reward + self.gamma * np.max(self.q[nstate[0]][nstate[1]])-self.q[state[0]][state[1]]
[action]) # update q function value
    self.state = nstate # update state
    self.rewards += reward # gain reward
    if self.done: # finish
        self.reward_list = np.append(self.reward_list, self.rewards)
```

Result Analysis:

After training 500 episodes, we get the q function value of each state with different actions, as shown in the figure below.



After training 500 episodes, the optimal decision under different grids is shown in the figure below:



The rewards of training process are as follows.