*Course Project of Reinforcement Learning* (2023), **xx**:xx 1–8

RESEARCH ARTICLE

# Model-Free Control

Author: Yao Tang

**Keywords:** Model-Free Control, SARSA,Q-Learning

**Abstract**
In this project, we implemented two classic Model-Free Control algorithms in RL: SARSA and Q-Learning. We trained both algorithms online in a cliff-walking environment that we designed ourselves and evaluated their performance in the same environment using the OpenAI Gym toolkit. We also conducted experiments to examine the impact of reward shaping and the epsilon-greedy exploration strategy on algorithm performance. Our results showed that both SARSA and Q-Learning were able to learn effective policies in the cliff-walking environment, while the former policy tends to be conservative and the latter one tends to be aggresive. We also found that reward shaping can significantly improve the speed and stability of learning, and that the optimal value of epsilon in the epsilon-greedy algorithm depends on the specific environment and task.

## Contents

## 1. Introduction

Model-free control is a subfield of RL that focuses on learning optimal policies without requiring an explicit model of the environment. In this project, we compare two classic model-free control algorithms, SARSA and Q-Learning, in a cliff-walking environment, which is a popular benchmark for evaluating RL algorithms, as it presents a challenging navigation task that requires agents to learn to avoid falling off cliffs while navigating to a goal location.

In our experiments, we train both SARSA and Q-Learning online in the cliff-walking environment and evaluate their performance using the OpenAI Gym toolkit. We also conduct experiments to explore the impact of reward shaping and the epsilon-greedy exploration strategy on algorithm performance. SARSA and Q-Learning are both widely used algorithms in RL, but they differ in their approach to updating the Q-values and the exploration-exploitation tradeoff. By comparing these algorithms, we aim to gain insights into their relative strengths and weaknesses in the context of model-free control.

We find that Q-Learning generally outperforms SARSA in convergence rate in the cliff-walking environment. Also, reward shaping significantly improves the speed and stability of learning for both algorithms, and that the optimal value of epsilon in the epsilon-greedy algorithm depends on the specific environment and task. These findings demonstrate the importance of careful algorithm selection and exploration-exploitation strategy for achieving optimal performance in RL environments.

## 2. Preliminaries

### *2.1. SARSA*

SARSA (State-Action-Reward-State-Action) is an on-policy algorithm that learns the optimal policy by estimating the action-value function Q(s,a) for each state-action pair (s,a) by first choosing the actions for next timestep using epsilon-greedy policy and then updating the Q(s,a) at the current timestep.

Here is the SARSA algorithm:

```
Input: a state-action value function $Q$, a step size $\alpha$, a small positive number $\epsilon$
Initialize $Q(s, a)$ for all $s \in S$, $a \in A(s)$, arbitrarily, and $Q(\text{terminal state}, ) = 0$
Repeat (for each episode):
   Initialize $S$
   Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
   Repeat (for each step of episode):
      a. Take action $A$, observe $R$, $S'$
      b. Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
      c. $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
      d. $S \leftarrow S'$, $A \leftarrow A'$
until convergence of $Q$
```

### *2.2. Q-Learning*

In contrast to SARSA, Q-Learning is an off-policy algorithm. The agent acts by the epsilon-greedy policy while the Q function updates its value using the action maximizing its Q value.

Here is the Q-Learning algorithm:

```
Input: a state-action value function $Q$, a step size $\alpha$, a small positive number $\epsilon$
Initialize $Q(s, a)$ for all $s \in S$, $a \in A(s)$, arbitrarily, and $Q(\{terminalstate\}, ) = 0$
Repeat (for each episode):
   Initialize $S$
   Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
   Repeat (for each step of episode):
      b. Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
      a. Take action $A$, observe $R$, $S'$
      c. $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_{A'} Q(S', A') - Q(S, A)]$
      d. $S \leftarrow S'$
until convergence of $Q$
```

## 3. Experiment

### *3.1. Implementation*

#### 3.1.1. Environment

The size of GridWorld used in our experiment is 4*12, where the agent start from (3,0). If the agent gets to any state betwwen (3,1) and (3,10), it will falls down the cliff and get a reward of -100. Any movement except leading to the cliff get the agent a reward of -1. The goal state is (3,11).

The GUI of the environment is shown in Figure 1. And the setting is included in the following code.

```
class CliffWalking(object):
    def __init__(self, grid_height,grid_width):
        self.height = grid_height
        self.width = grid_width
        self.start_state = (self.height-1,0) # the start state is a tuple
        self.state = self.start_state # the current state is a tuple
        self.cliff = [(self.height-1,i) for i in range(1,self.width-1)]
        self.goal_state = (self.height-1,self.width-1)

    def step(self,action):
        # action 0,1,2,3: up,right,down,left
        if action==1 and self.state[1]+1 < self.width:
            self.state = (self.state[0],self.state[1]+1)
        if action==3 and self.state[1]-1 >= 0:
            self.state = (self.state[0],self.state[1]-1)
        if action==0 and self.state[0]-1 >= 0:
            self.state = (self.state[0]-1,self.state[1])
        if action==2 and self.state[0]+1 < self.height:
            self.state = (self.state[0]+1,self.state[1])
        if self.state in self.cliff:
            done = True
            reward = -100.0
        else:
            reward = -1.0
        if self.state == self.goal_state:
            done = True
        return self.state,reward,done
```

To prevent the agent from getting stuck in a local optimum of taking actions beyond the boundaries, we have also set up another Reward and punishment mechanism that gives the agent a slightly greater penalty for taking actions beyond the boundaries compared to exploration actions that do not.

```
def step(self,action):
    reward = -1.0
    x = -5.0
    if action==1 :
        if self.state[1]+1 < self.width:
            self.state = (self.state[0],self.state[1]+1)
        else:
            reward = x
    if action==3 :
        if self.state[1]-1 >= 0:
```

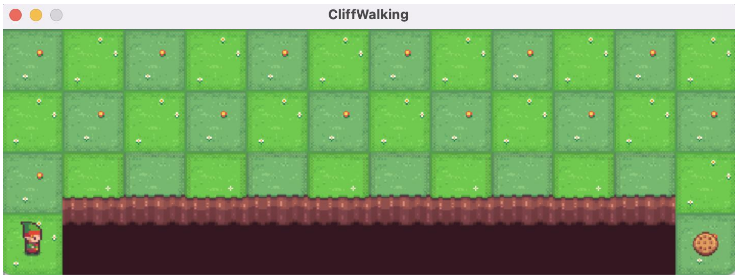

***Figure 1.*** *The environment of Gridworld used in our experiment..*

```
            self.state = (self.state[0],self.state[1]-1)
        else:
            reward = x
    if action==0 :
        if self.state[0]-1 >= 0:
            self.state = (self.state[0]-1,self.state[1])
        else:
            reward = x
    if action==2 :
        if self.state[0]+1 < self.height:
            self.state = (self.state[0]+1,self.state[1])
        else:
            reward = x
    done = False
    if self.state in self.cliff:
        done = True
        reward = -100.0
    if self.state == self.goal_state:
        done = True
    return self.state,reward,done
```

### 3.1.2. SARSA

The code for this function is consistent with the pseudocode we introduced earlier, so the code is provided without further elaboration.

```
Q = np.random.normal(0, 0.1, (height, width, 4))
    Q [0,1:width-1,:] = 0.0
    env = CliffWalking(height,width)# Repeat for each episode
    for i in range(num_episodes):
        done = False
        state = env.reset()
        action = epsilon_greedy(epsilon,Q,state)
        while not done:
            next_state,reward,done = env.step(action)
            next_action = epsilon_greedy(epsilon,Q,state)
            Q[state[0],state[1],action] = Q[state[0],state[1],action] + alpha*
            (reward+gamma*Q[next_state[0],next_state[1],next_action]
            - Q[state[0],state[1],action])
            state = next_state
            action = next_action
```

### 3.1.3. Q-Learning

The code for this function is consistent with the pseudocode we introduced earlier, so the code is provided without further elaboration.

```
Q_2 = np.random.normal(0, 0.1, (height, width, 4))
    Q_2 [0,1:width-1,:] = 0.0
    for i in range(num_episodes):
        done = False
        state = env.reset()
        action = epsilon_greedy(epsilon,Q_2,state)
        while not done:
            next_state,reward,done = env.step(action)
            next_action = epsilon_greedy(epsilon,Q_2,state)
            #the difference between Q-learning and Sarsa
            next_action_for_update = np.argmax(Q_2[next_state[0],next_state[1],:])
            Q_2[state[0],state[1],action] = Q_2[state[0],state[1],action] + alpha*
            (reward + gamma*Q_2[next_state[0],next_state[1],next_action_for_update]
            - Q_2[state[0],state[1],action])
            state = next_state
            action = next_action
```

### 3.1.4. Evaluation

To analyze the experimental results more intuitively, we used the CliffWalking-v0 environment from the gymnasium package, which has the same state space and action space as our defined CliffWalking. The implementation is shown in the following code in detail.

```
# Evaluation
    gym_env = gym.make('CliffWalking-v0',render_mode="human")
    # Evaluate the SARSA result
    obs, info = gym_env.reset()
    state = (int(obs/12),obs%12)
    done = False
    truncated = False
    while not done and not truncated:
        action = np.argmax(Q[state[0],state[1],:])
        print(action)
        obs, reward, done, truncated, info = gym_env.step(action)
        state=(int(obs/12),obs%12)
    gym_env.close()

    # Evaluate the Q-Learning result
```

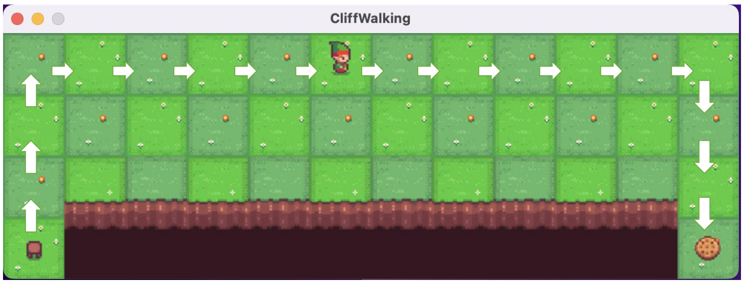

***Figure 2.*** *The evaluation result of SARSA..*

```
obs, info = gym_env.reset()
state = (int(obs/12),obs%12)
done = False
truncated = False
while not done and not truncated:
    action = np.argmax(Q_2[state[0],state[1],:])
    obs, reward, done, truncated, info = gym_env.step(action)
    state=(int(obs/12),obs%12)
gym_env.close()
```

### *3.2. Results*

#### 3.2.1. Evaluation of Q-Learning

- Firstly, we analyzed the performance of the policy corresponding to the Q function obtained by the SARSA algorithm in the Gym environment. The result is shown in Figure 2.
- Then we evaluated the performance of Q-Learning. The result is shown in Figure 3.
- We also explore different environment setting by adjusting the reward of each movement and epsilon. The best config we have tested is that epsilon = 0.3, alpha = 0.1, gamma = 1.0, $num_episodes$ = 10000.

## 4. Analysis

- Comparing Figure 2 and Figure 3, we can see that the agent corresponding to Q-Learning walks close to the cliff edge, while the agent corresponding to SARSA algorithm moves away from the cliff to avoid danger and maximize safety. Therefore, Q-Learning corresponds to a more bold strategy, while SARSA is a relatively conservative strategy. This interesting phenomenon may be caused by the different update methods of SARSA and Q-learning algorithms.
    - SARSA selects the action according to the current policy in the current state, then updates the Q value based on the new state and new action. Therefore, SARSA is an on-policy algorithm. This is equivalent to looking ahead for danger and making plans in advance when updating the Q value. SARSA(0) plans one step ahead, while SARSA($\lambda$) plans $\lambda$ steps ahead and feeds

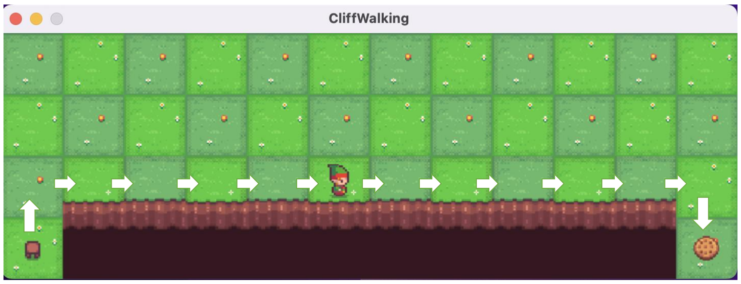

***Figure 3.*** *The evaluation result of Q-Learning.*

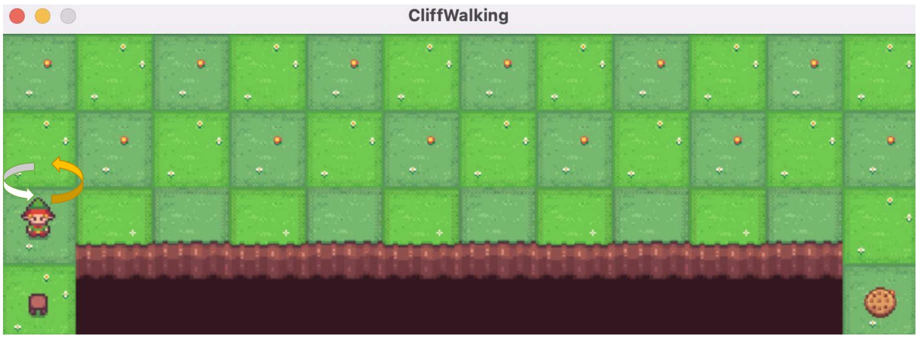

***Figure 4.*** *The evaluation result of local optimum..*

this consideration of danger into the current Q value for the current step. Therefore, the Q function obtained by SARSA is sensitive to danger (very small reward).

– On the other hand, Q-learning updates the Q value for the current step based on the new state and the maximum Q value in the new state. This indicates that the agent it trains is very confident in its Q values, and it believes that it will also choose the action with the maximum Q value in the future and obtain a large reward. Since it updates the current Q value based on the maximum Q value in the next step, it is not sensitive to potential danger in the future, resulting in a more adventurous policy.

• If we decrease the reward for each movement of the agent or the value of epsilon, we would expect that the policy learned by SARSA algorithm would become closer to the policy learned by

Q-learning, as SARSA would try to minimize the number of steps taken to get to the goal. However, we found that this would lead to the emergence of a lazy agent, as the high cost of exploration might prevent the agent from exploring the environment sufficiently and getting stuck in a local optimum at a particular step. Therefore, balancing exploration and exploitation requires adjusting the environment settings according to the specific situation as shown in Figure 4.
- Increasing the reward for each movement can encourage exploration, similar to the effect of increasing epsilon. However, if the reward is increased too much, it may lead to the occurrence of local optimal situations. This is because when the reward for each movement is too high, the agent does not expect to reach the goal state and end the game round, but rather to stay in the game for as long as possible. To solve this problem, one solution is to appropriately reduce the reward for each movement, or give a larger reward when the agent reaches the goal state and set a discount factor less than 1 to encourage the agent to approach the goal state as quickly as possible.

## 5. Conclusion

In conclusion, the comparison between SARSA and Q-learning algorithms showed that SARSA is a more conservative strategy, while Q-learning is a more adventurous strategy. This difference in behavior can be attributed to the update methods used by the two algorithms. SARSA is an on-policy algorithm that selects actions based on the current policy and plans ahead to avoid potential danger, while Q-learning updates the Q value based on the maximum Q value in the next step and is more confident in its Q values, resulting in a more adventurous policy. Balancing exploration and exploitation is crucial, and adjusting the environment settings according to the specific situation is necessary. It is also important to note that increasing the reward for each movement may encourage exploration but may lead to local optimal situations if the reward is too high. Therefore, a suitable reward setting and discount factor are needed to encourage the agent to approach the goal state as quickly as possible.