Assignment 2

余北辰 519030910245

1 Experimental Purpose

- Implement first-visit MC method, every-visit MC method and TD(0) method separately
- Evaluate the uniform random policy through the three methods separately

2 Experimental Background

In this section, we present the methods of Monte-Carlo learning and Temporal-Difference learning separately.

2.1 Monte-Carlo Learning

Monte-Carlo learning is a model free method of reinforcement learning, which means learning without advance notice of MDP state transfer probability and immediate rewards. On the contrary, Monte-Carlo learning learns directly from episodes of experience. In this experiment, we use Monte-Carlo learning method to do policy evaluation.

First Visit Monte-Carlo Policy Evaluation

First visit Monte-Carlo learning use the episode under some policy π to evaluate the value function V:

$$S_1, A_1, R_1, S_2, \cdots, S_T, A_T, R_T \sim \pi$$

Monte-Carlo policy evaluation uses empirical average sample returns, instead of expected return. In first visit Monte-Carlo learning, for each episode, the value function under one state is modified only when the state is first time being visited.

When a state s is firstly visited, the counter increase: $N(s) = N(s) + 1$, and the total return is increased: $S(s) = S(s) + G_t$, in which G_t is the return of the episode in time-step t:

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-1} R_T$$

Then the value function is estimated by average return $V(s) = S(s)/N(s)$. When the number of episodes is big enough, the estimated value function will convergence to the true value.

The pseudo-code for first visit Monte-Carlo policy evaluation procedure is shown below:

First-visit MC prediction, for estimating $V \approx v_\pi$

Initialize:
$\pi \leftarrow$ policy to be evaluated
$V \leftarrow$ an arbitrary state-value function
$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Repeat forever:
Generate an episode using π
For each state s appearing in the episode:
$G \leftarrow$ return following the first occurrence of s
Append G to $Returns(s)$
$V(s) \leftarrow$ average$(Returns(s))$

Every Visit Monte-Carlo Policy Evaluation

Every visit Monte-Carlo policy evaluation has no difference with first visit Monte-Carlo in general, but the value function V is updated every time when a stated is visited.

2.2 Temporal-Difference Policy Evaluation

Temporal-Difference learning is another model free method of reinforcement learning. Different from Monte-Carlo learning, Temporal-Difference learning learns from incomplete episodes. In this experiment we implement TD(0), which learn from just the information of the next time-step. The progress of updating the value function is shown as below:

$$V(s) = V(s) + \alpha(R + \gamma V(s') - V(s))$$

The pseudo-code for first visit Temporal-Difference policy evaluation procedure is shown below:

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop for each episode:
Initialize S
Loop for each step of episode:
$A \leftarrow$ action given by π for S
Take action A, observe R, S'
$V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
$S \leftarrow S'$
until S is terminal

3 Experimental Procedure

The source code for this experiment includes `gridworld.py`, `FirstVisitMC.py`, `EveryVisitMC.py`, `TD.py` and `main.py`.

3.1 gridworld.py

`gridworld.py` is exactly the same as that in the previous experiment.

3.2 FirstVisitMC.py

`FirstVisitMC.py` is implemented based on the above pseudo-code.

```
def policy_evaluation(gridworld, policy, episode_num, gamma=0.99):
    V = np.zeros(gridworld.nS, dtype=np.float)
    N = np.zeros(gridworld.nS, dtype=np.float)
    for i in range(episode_num):
        is_done = False
        state = np.random.randint(gridworld.nS)
        episode_reward = []
        episode_state = []
        while not is_done:
            action = np.random.choice(4, 1, p=policy[state])[0]
            prob, state, reward, is_done = gridworld.P[state][action][0]
            episode_reward.append(reward)
            episode_state.append(state)
        T = len(episode_reward)
        state_visited = []
        for t in range(T):
            current_state = episode_state[t]
            if current_state not in state_visited:
                state_visited.append(current_state)
                N[current_state] += 1
                G = 0.0
                discount = 1.0
                for j in range(t, T):
                    G += discount*episode_reward[j]
                    discount *= gamma
                V[current_state] = V[current_state] + (G-
V[current_state])/N[current_state]
    return V
```

In each cycle, first of all, an initial state is randomly chosen. Then in each state, the action of the agent follows the policy to be evaluate. The state visited and the reward of each time-step is stored in two lists. After getting the whole episode, the value function V is updated following the equation above. It is worth noting that to guarantee "first visit", a judgment condition that whether the state is already visited is added in the algorithm.

3.3 EveryVisitMC.py

`EveryVisitMC.py` is modified from `FirstVisitMC.py`, just delete the judgment condition:

```
def policy_evaluation(gridworld, policy, episode_num, gamma=0.99):
    V = np.zeros(gridworld.nS, dtype=np.float)
    N = np.zeros(gridworld.nS, dtype=np.float)
    for i in range(episode_num):
        is_done = False
        first_visit = np.zeros(gridworld.nS)
        state = np.random.randint(gridworld.nS)
        episode_reward = []
        episode_state = []
        while not is_done:
            action = np.random.choice(4, 1, p=policy[state])[0]
            prob, state, reward, is_done = gridworld.P[state][action][0]
            episode_reward.append(reward)
```

```
            episode_state.append(state)
        T = len(episode_reward)
        for t in range(T):
            current_state = episode_state[t]
            N[current_state] += 1
            G = 0.0
            discount = 1.0
            for j in range(t, T):
                G += discount*episode_reward[j]
                discount *= gamma
            V[current_state] = V[current_state] + (G-
V[current_state])/N[current_state]
    return V
```

3.4 TD.py

`TD.py` is implemented based on the above pseudo-code.

```
def policy_evaluation(gridworld, policy, episode_num, alpha=0.5,
gamma=0.99):
    V = np.zeros(gridworld.nS, dtype=np.float)
    for i in range(episode_num):
        is_done = False
        state = np.random.randint(gridworld.nS)
        episode_reward = []
        episode_state = []
        while not is_done:
            action = np.random.choice(4, 1, p=policy[state])[0]
            prob, state, reward, is_done = gridworld.P[state][action][0]
            episode_reward.append(reward)
            episode_state.append(state)
        T = len(episode_reward)
        for t in range(T-1):
            current_state = episode_state[t]
            next_state = episode_state[t+1]
            V[current_state] = V[current_state] + alpha * \
                (episode_reward[t+1]+gamma*V[next_state]-V[current_state])
    return V
```

The acquisition of episode is the same as MC. When updating the value function, it is just following the above equation.

3.5 main.py

`main.py` instantiates the environment, initialize the random policy and use MC or TD to evaluate that policy.

```
def test_first_visit_MC(gridworld, policy, episode_num):
    V = FirstVisitMC.policy_evaluation(gridworld, policy, episode_num)
    print('----------------First Visit MC---------------')
    print('Value Function:')
    print(np.around(np.reshape(V, gridworld.shape), decimals=4))
    print('episode Num:')
    print(episode_num)
    print('-------------------------------------------------')

```

```
10
11  def test_every_visit_MC(gridworld, policy, episode_num):
12      V = EveryVisitMC.policy_evaluation(gridworld, policy, episode_num)
13      print('----------------Every Visit MC----------------')
14      print('Value Function:')
15      print(np.around(np.reshape(V, gridworld.shape), decimals=4))
16      print('episode Num:')
17      print(episode_num)
18      print('------------------------------------------------')
19
20
21  def test_TD(gridworld, policy, episode_num):
22      V = TD.policy_evaluation(gridworld, policy, episode_num)
23      print('----------------TD----------------')
24      print('Value Function:')
25      print(np.around(np.reshape(V, gridworld.shape), decimals=4))
26      print('episode Num:')
27      print(episode_num)
28      print('----------------------------------')
```

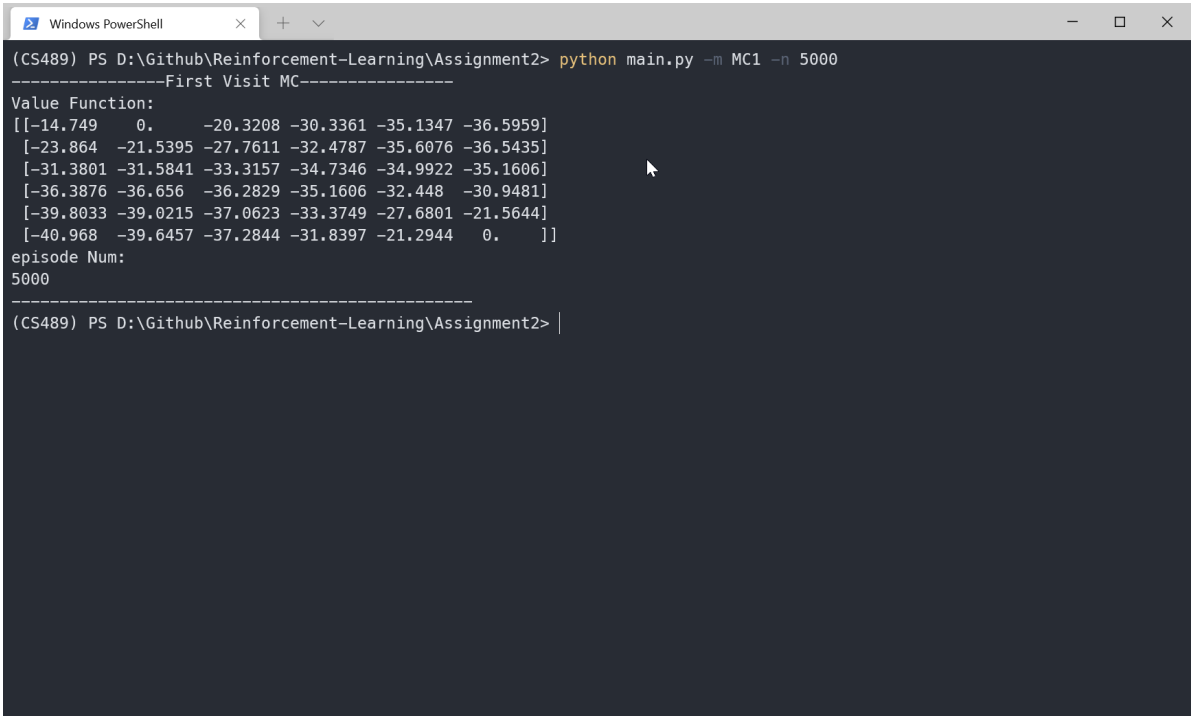
`test_first_visit_MC()`, `test_every_visit_MC()` and `test_TD()` respectively output the results of evaluation using first visit MC, every visit MC and TD(0). The number of episodes used is also shown.

```
1   def main():
2       parser = argparse.ArgumentParser()
3       parser.add_argument('-m', '--method', type=str, default='MC1')
4       parser.add_argument('-n', '--episode_num', type=int, default=1000)
5       args = parser.parse_args()
6       method = args.method
7       episode_num = args.episode_num
8
9       gridworld = GridworldEnv([6, 6])
10      policy = np.ones([gridworld.nS, gridworld.nA]) / gridworld.nA
11
12      if method == 'MC1':
13          test_first_visit_MC(gridworld, policy, episode_num)
14
15      if method == 'MC2':
16          test_every_visit_MC(gridworld, policy, episode_num)
17
18      if method == 'TD':
19          test_TD(gridworld, policy, episode_num)
```

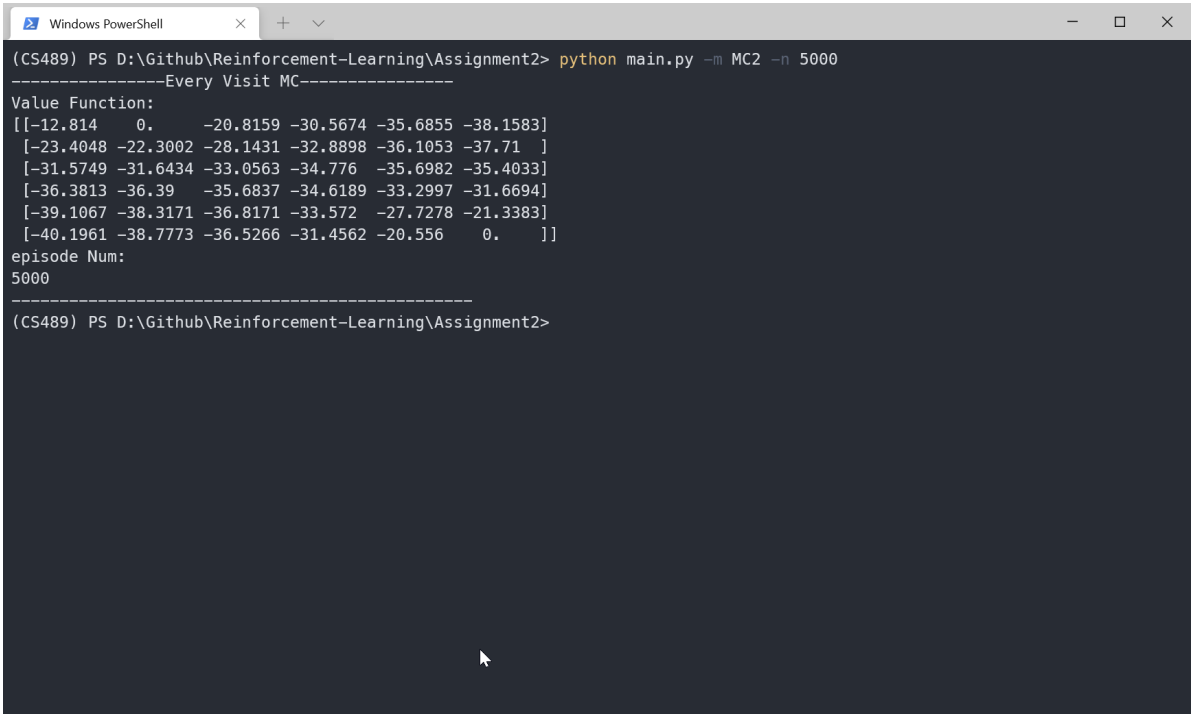
`main()` use parser to get the parameters entered on the command line, and depending on these parameters, the method of evaluation and the number of episodes used are chosen.

4 Experimental Result

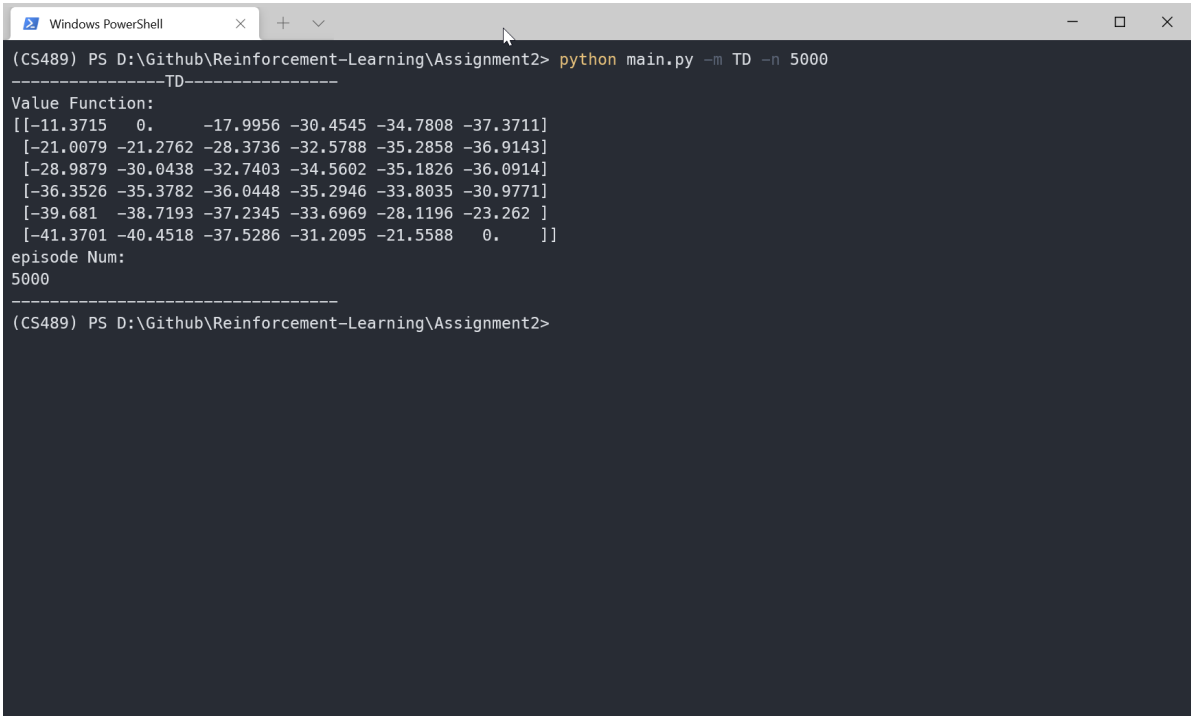
Run first visit MC:



Run every visit MC:



Run TD(0):



All the three results meet expectations, and the final evaluated value functions are similar. TD(0) runs quicker than MC methods, because that TD doesn't use the whole episode to learn, which make it quicker.

5 Experimental Summary

- Systematically learn and understand Monte-Carlo learning and Temporal-Difference learning for perdiction
- Implement and successfully run the algorithm of MC and TD policy evaluation based on pseudo-code