Reinforcement Learning Assignment Report

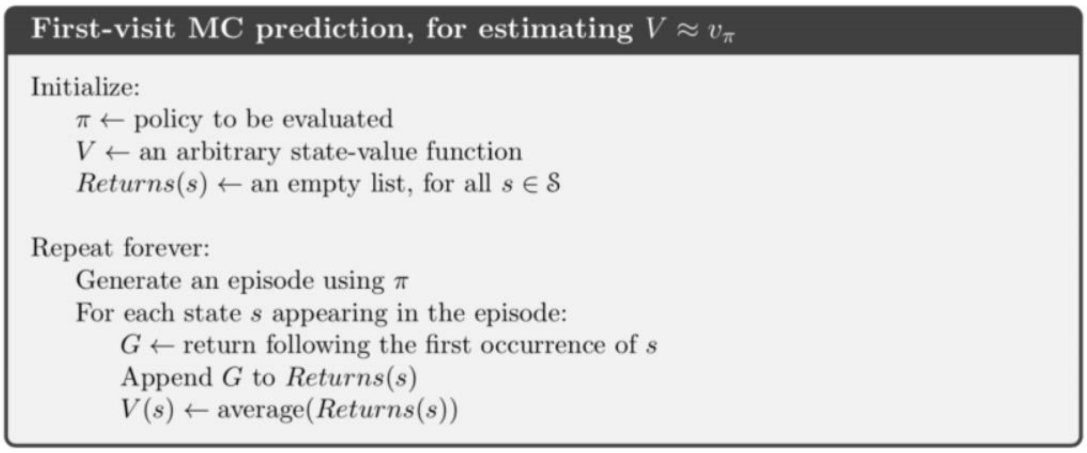
516030910373 XueyanLi

Part 1: First-visit Monte-Carlo Policy Evaluation

✧ Introduction

Since the environmental state and transition probability of many models are unknown in reinforcement learning, full probability expansion cannot be performed. Therefore, Monte-Carlo method (MC) is used to evaluate the policy. The Monte-Carlo method does not refer to a specific method, but simply refers to the calculation method based on random sampling.

The MC randomly samples from a certain state and records each complete route as one episode. For the first occurrence of each episode, which is called *first-visit*, the subsequent reward is taken as its value, of which the pseudo code is shown as follow:



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 $\pi$
    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($Returns(s)$)
```

✧ Experiment setup

As every grid can be taken as the start of an episode (except the two terminals), I take all the 14 states as starts and generate 10000 episodes for every start. Therefore, there are 14000 episodes in all. In every episode, I set an upper limit of 100 steps which means the episode can't be longer than 100 steps. Of course, if the state has reached the terminal, the episode will suspend at once. What's more, I set the γ as 1.

✧ Code

My code is shown as follow:

```python
def generateepisode(V, policy):
    for oristate in range(15):
        if oristate == 0:
            continue
        for times in range(10000):
            state = oristate
            S = []
            S.append((state, 0))
            for i in range(100):
                action = random.randint(0, 3)
                nextstate = A[action][statepos[state][0]][statepos[state][1]]
                S.append((nextstate, -1))
                #print(nextstate)
                if nextstate == 0:
                    break
                if nextstate == 15:
                    break
                state = nextstate
            #print(S)
            seperate_episode = set([eps[0] for eps in S])

            for s_eps in seperate_episode:
                for i, x in enumerate(S):
                    if x[0] == s_eps:
                        first_visit_pos = i
                G = sum([e[1] for e in S[first_visit_pos + 1:]])

                return_sum[s_eps] += G
                #print(s_eps, G)
                return_count[s_eps] += 1.0

    for state in range(16):
        if (return_count[state] != 0):
            V[statepos[state][0]][statepos[state][1]] = return_sum[state] / return_count[state]
```

✧ Result

The result of my code is shown as follow. Every number represents the value of the corresponding state. For example, the number -12 in line 1 column 3 means the value of this policy when reaching the grid of line 1 column 3.

```
[0.0, -7.533657797618089, -10.173400023498212, -12.17993491050194]
[-7.627617631851086, -8.470220916198771, -9.161528022798208, -10.193701263780586]
[-10.148148148148149, -9.115978470655394, -8.470102141680396, -7.517001822204868]
[-12.144155299055614, -10.109875691769416, -7.523206136518109, 0.0]
```

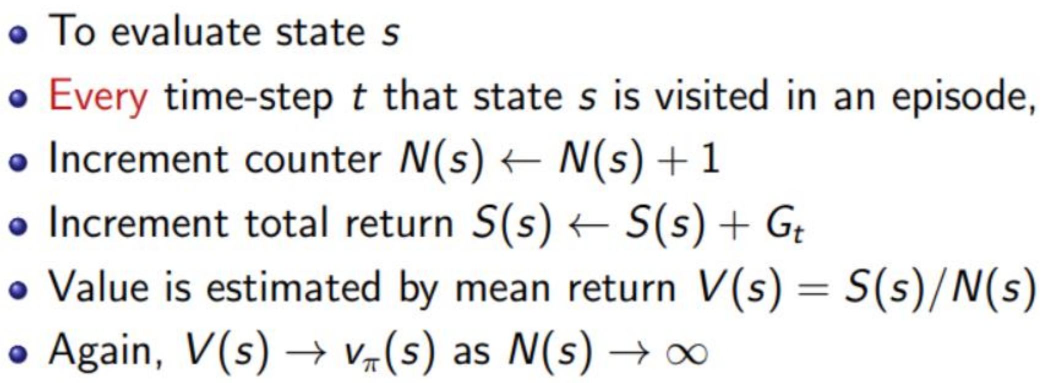
✧ Discussion

It can be seen from the result that the nearer the grid is to end, the higher its value is. Therefore, MC is a method to evaluate the policy.

Part 2: Every-visit Monte-Carlo Policy Evaluation

✧ Introduction

The main difference between first-visit MC and every-visit MC is that first-visit MC only consider the first times of a state in the episode while every-visit MC consider every times of a state which will have more samples of value, of which the pseudo code is shown as follow:

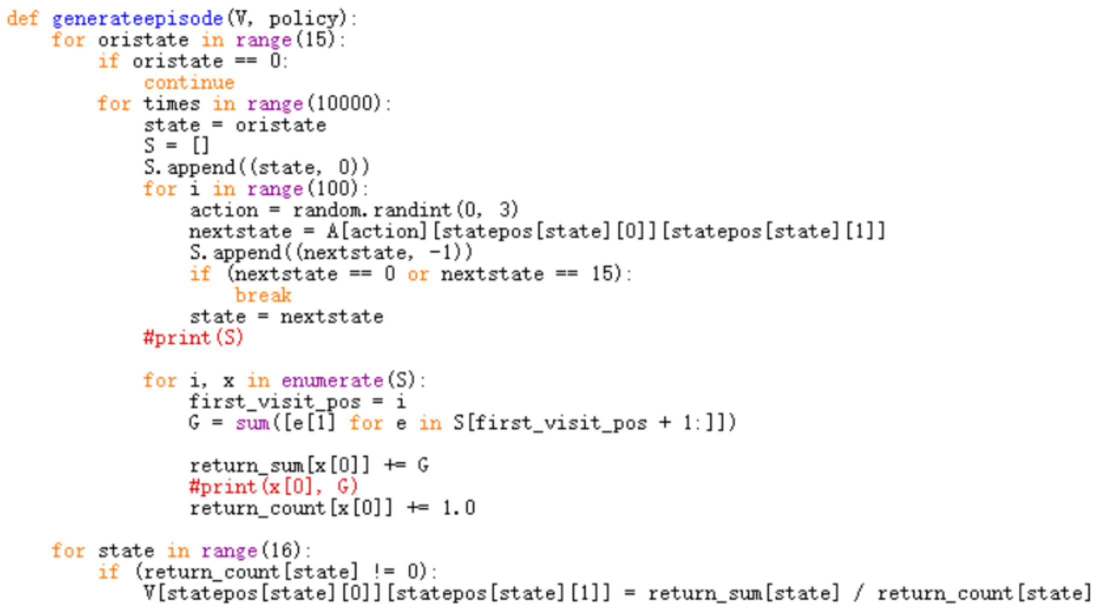


- To evaluate state s
- Every time-step t that state s is visited in an episode,
- Increment counter $N(s) \leftarrow N(s) + 1$
- Increment total return $S(s) \leftarrow S(s) + G_t$
- Value is estimated by mean return $V(s) = S(s)/N(s)$
- Again, $V(s) \rightarrow v_\pi(s)$ as $N(s) \rightarrow \infty$

✧ Experiment setup

Just like the setup in first-visit MC, I take all the 14 states as starts and generate 10000 episodes for every start. Therefore, there are 14000 episodes in all. In every episode, I set an upper limit of 100 steps. The γ is set to 1.

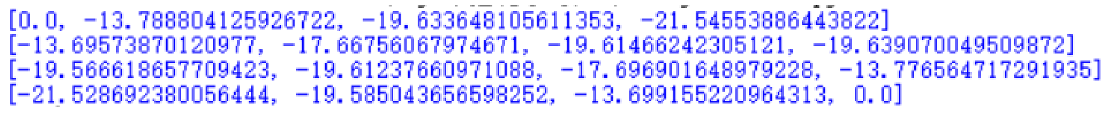
✧ Code



```
def generateepisode(V, policy):
    for oristate in range(15):
        if oristate == 0:
            continue
        for times in range(10000):
            state = oristate
            S = []
            S.append((state, 0))
            for i in range(100):
                action = random.randint(0, 3)
                nextstate = A[action][statepos[state][0]][statepos[state][1]]
                S.append((nextstate, -1))
                if (nextstate == 0 or nextstate == 15):
                    break
                state = nextstate
            #print(S)

            for i, x in enumerate(S):
                first_visit_pos = i
                G = sum([e[1] for e in S[first_visit_pos + 1:]])

                return_sum[x[0]] += G
                #print(x[0], G)
                return_count[x[0]] += 1.0

    for state in range(16):
        if (return_count[state] != 0):
            V[statepos[state][0]][statepos[state][1]] = return_sum[state] / return_count[state]
```

✧ Result

The result of my code is shown as follow. Every number represents the value of the corresponding state. For example, the number -19 in line 1 column 3 means the value of this policy when reaching the grid of line 1 column 3.



```
[0.0, -13.788804125926722, -19.633648105611353, -21.54553886443822]
[-13.69573870120977, -17.66756067974671, -19.61466242305121, -19.639070049509872]
[-19.566618657709423, -19.61237660971088, -17.696901648979228, -13.776564717291935]
[-21.528692380056444, -19.585043656598252, -13.699155220964313, 0.0]
```

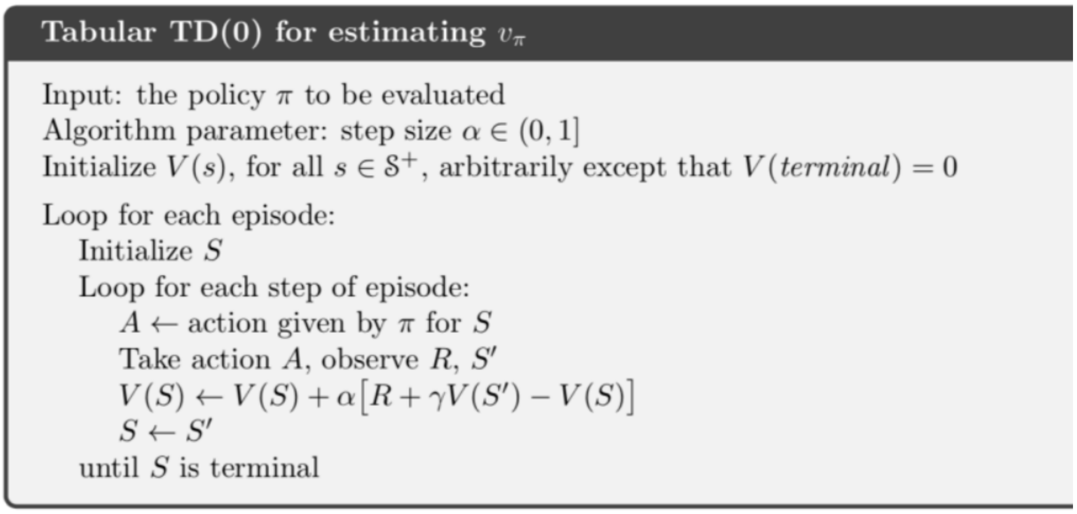
✧ Discussion

It can be seen from the result that the difference between the value of different state is much larger than in first-visit MC.

Part 3: Temporal Different Policy Evaluation

✧ Introduction

Different from MC which uses the exact return to update the value, Temporal Different (TD) use the Bellman equation to estimate the value, and then updates the estimate as the target value. The advantage of TD is that it can update after every step which can be seen as online learning. The pseudo code of TD is shown as follow:



```
Tabular TD(0) for estimating $v_\pi$

Input: the policy $\pi$ 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 $\pi$ for $S$
        Take action $A$, observe $R$, $S'$
        $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
        $S \leftarrow S'$
    until $S$ is terminal
```

✧ Experiment setup

In TD, I take all the 14 states as starts and generate 10000 episodes for every start, of which is 14000 episodes in all. And in every episode, unlike MC, I didn’t set any upper limit but wait until the episode reaches the terminal. The γ is set 1 and the α is set 0.5.

✧ Code

```python
def generateepisode(V, policy):
    for oristate in range(15):
        if oristate == 0:
            continue
        for times in range(10000):
            state = oristate
            S = []
            S.append((state, 0))
            nextstate = 1
            while (nextstate != 0 and nextstate != 15):
                action = random.randint(0, 3)
                nextstate = A[action][statepos[state][0]][statepos[state][1]]
                V[statepos[state][0]][statepos[state][1]] = \
                    V[statepos[state][0]][statepos[state][1]] + \
                    alpha * (-1 + V[statepos[nextstate][0]][statepos[nextstate][1]] - \
                    V[statepos[state][0]][statepos[state][1]])

                state = nextstate
```

✧ Result

The result of my code is shown as follow of which the meaning is the same with that of MC.

```
[0, -9.91371584391418, -20.209525266380833, -23.09346866721426]
[-15.54606037235609, -22.39550773798839, -23.529016063469115, -24.443313666533868]
[-23.211457959039937, -19.516685773898995, -22.44676195834196, -13.842524469164733]
[-28.627741060454003, -28.64574946959503, -12.898457119035022, 0]
```