CS489 Assignment 2 Report

517030910214 Hongzhou Liu

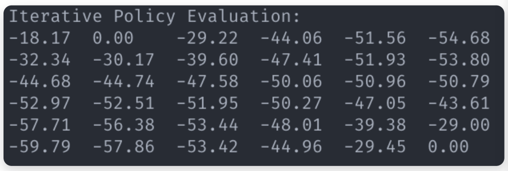
0. Introduction

In this assignment, we are required to implement Monte-Carlo (MC) Learning and Temporal-Difference (TD) Learning. These are two methods that can evaluate a given policy directly by learning from episodes of experience without knowledge of MDP model. We will still implement these methods based on GridWorld.

Environment:

- Ubuntu 18.04 LTS
- Python 3.7.4

We've already calculated true value of the given policy in last assignment. We consider it as a baseline.



```
Iterative Policy Evaluation:
-18.17  0.00    -29.22  -44.06  -51.56  -54.68
-32.34  -30.17  -39.60  -47.41  -51.93  -53.80
-44.68  -44.74  -47.58  -50.06  -50.96  -50.79
-52.97  -52.51  -51.95  -50.27  -47.05  -43.61
-57.71  -56.38  -53.44  -48.01  -39.38  -29.00
-59.79  -57.86  -53.42  -44.96  -29.45  0.00
```

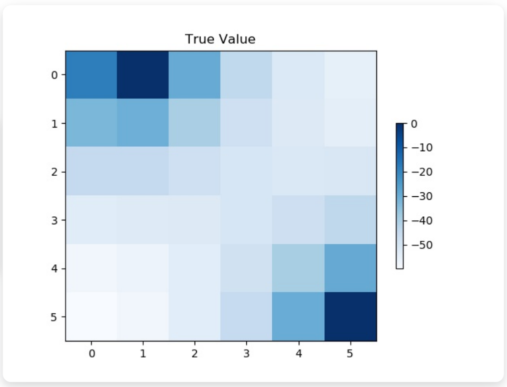


Fig.1 Baseline

1. Monte-Carlo Learning

1.1 Implementation

1.1.1 First Visit MC

MC can learn directly from episodes of experience. The idea behind it is quite simple, which regards average sample returns as value. According to law of large numbers, if we sample infinite episodes, the sample average returns is the true value.

In MC algorithm, we should first generate an episode under policy π. The policy here is still the random uniform policy. The code to generate an episode is shown as follows:

```python
def __episodeGenerator(self):
    episode = []
    episode.append(random.randint(0, 35))
    curState = episode[0]
    while curState not in self.terminalState:
        curAction = random.randint(0, 3)
        curReward = self.reward[curState][curAction]
        curState = self.trans[curState][curAction]
        episode.extend([curAction, curReward, curState])
    return episode
```

It's easy to implement as we've perfectly defined all the data structure needed.

In first visit MC, for a given episode and a state s in the episode, we find the first time-step t that it is visited in the episode. Then, we increment the counter $N(s)$ and update estimate value $V(s)$ incrementally by

$$V(s) \leftarrow V(s) + \frac{1}{N(s)}(G_t - V(s))$$

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

Notice that MC can only apply to episodic MDPs ($\gamma = 1$). To solve the "first visit" problem, we use `visited`, a boolean array, to track if a state is visited in an episode. The code:

```python
def firstVisitMC(self, iterTime, alpha=1.0, useAlpha=False):
    #Initialize
    self.value = [0.0 for i in range(self.state)]
    counter = [0 for i in range(self.state)]
    visited = [False for i in range(self.state)]

    #Sample & Evaluate
    for i in range(iterTime):
        episode = self.__episodeGenerator()
        for j in range(0, len(episode), 3):
            curState = episode[j]
            if not visited[curState]:
                visited[curState] = True
                counter[curState] += 1
                G = 0.0
                decay = 1.0
                for k in range(j + 2, len(episode), 3):
                    G += decay * episode[k]
                    decay *= self.gamma
                if useAlpha:
                    self.value[curState] = self.value[curState] + alpha * (G - self.value[curState])
                else:
                    self.value[curState] = self.value[curState] + (G - self.value[curState]) / counter[curState]
        visited = [False for i in range(self.state)]
```

As shown in the code, we use outer i-loop for different episode samples, middle j-loop for going through every states in the episode and inner k-loop for calculating return of a given first-visit state.

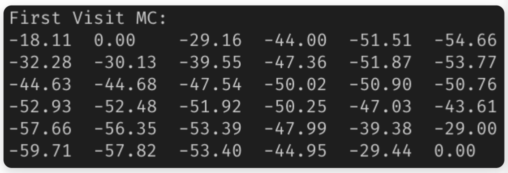
1.1.2 Every Visit MC

The every visit MC algorithm can be implemented by deleting `visited` array and the if-statement in first visit MC. So the middle j-loop is now:

```python
        for j in range(0, len(episode), 3):
            curState = episode[j]
            counter[curState] += 1
            G = 0.0
            decay = 1.0
            for k in range(j + 2, len(episode), 3):
                G += decay * episode[k]
                decay *= self.gamma
            if useAlpha:
                self.value[curState] = self.value[curState] + alpha * (G - self.value[curState])
            else:
                self.value[curState] = self.value[curState] + (G - self.value[curState]) / counter[curState]
```

1.2 Result

We set `iterTime` for both MC algorithms as 10000000. And finally we got



```
First Visit MC:
-18.11  0.00    -29.16  -44.00  -51.51  -54.66
-32.28  -30.13  -39.55  -47.36  -51.87  -53.77
-44.63  -44.68  -47.54  -50.02  -50.90  -50.76
-52.93  -52.48  -51.92  -50.25  -47.03  -43.61
-57.66  -56.35  -53.39  -47.99  -39.38  -29.00
-59.71  -57.82  -53.40  -44.95  -29.44  0.00
```

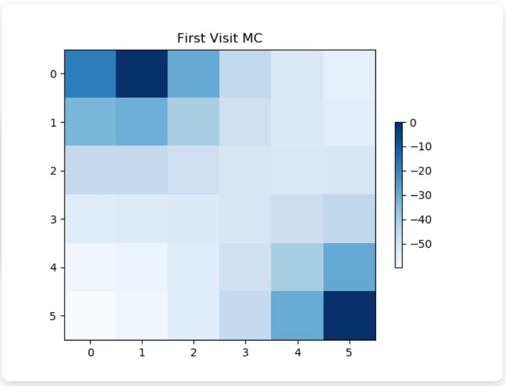
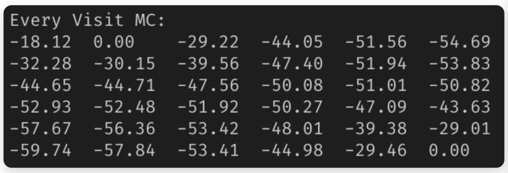


Fig.2 First Visit MC



```
Every Visit MC:
-18.12  0.00    -29.22  -44.05  -51.56  -54.69
-32.28  -30.15  -39.56  -47.40  -51.94  -53.83
-44.65  -44.71  -47.56  -50.08  -51.01  -50.82
-52.93  -52.48  -51.92  -50.27  -47.09  -43.63
-57.67  -56.36  -53.42  -48.01  -39.38  -29.01
-59.74  -57.84  -53.41  -44.98  -29.46  0.00
```

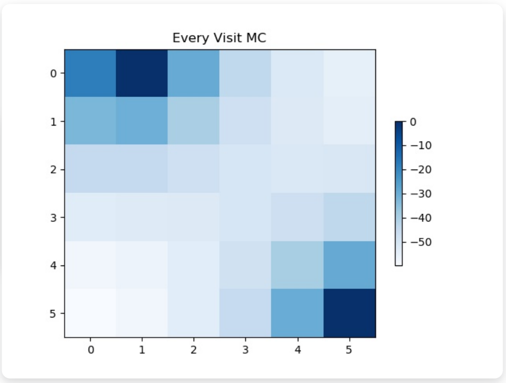


Fig.3 Every Visit MC

As we can see, the results of both MC after quantities of iterations are almost the true value. Thus, we can use MC to estimate a policy, despite that it is time consuming.

2. Temporal-Difference Learning

2.1 Implementation

TD can also learn directly from episodes of experience. However, unlike MC, TD learns from incomplete episodes and updates a guess towards a guess. It learns v_π on-line from experience under policy π.

The simplest TD learning algorithm is TD(0). In every episodes, TD(0) randomly chooses a starting state S. Then, it samples an action under policy π, takes the action and observes next state S' and reward R. After that, it will update $V(S)$ by

$$V(S) \leftarrow V(S) + \alpha(R + \gamma V(S') - V(S))$$

The episode will be ended if some action leads TD(0) to a terminal state.

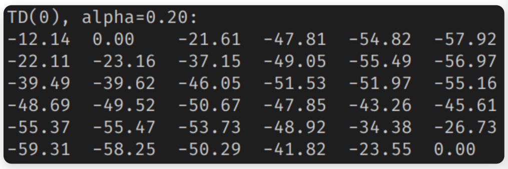
According to these ideas, we can implement TD(0) in Python:

```python
def TD0(self, iterTime, alpha):
    #Initialize
    self.value = [random.random() for i in range(self.state)]
    for i in self.terminalState:
        self.value[i] = 0.0

    #Sample & Evaluate
    for i in range(iterTime):
        curState = random.randint(0, 35)
        while curState not in self.terminalState:
            curAction = random.randint(0, 3)
            curReward = self.reward[curState][curAction]
            nextState = self.trans[curState][curAction]
            self.value[curState] = self.value[curState] + alpha * (curReward + self.gamma * self.value[nextState] - self.value[curState])
            curState = nextState
```

2.2 Result

We can run TD(0) with different α's. And for each α, we set `iterTime` as 10000000. Here's the results under different α's:



```
TD(0), alpha=0.20:
-12.14  0.00    -21.61  -47.81  -54.82  -57.92
-22.11  -23.16  -37.15  -49.05  -55.49  -56.97
-39.49  -39.62  -46.05  -51.53  -51.97  -55.16
-48.69  -49.52  -50.67  -47.85  -43.26  -45.61
-55.37  -55.47  -53.73  -48.92  -34.38  -26.73
-59.31  -58.25  -50.29  -41.82  -23.55  0.00
```

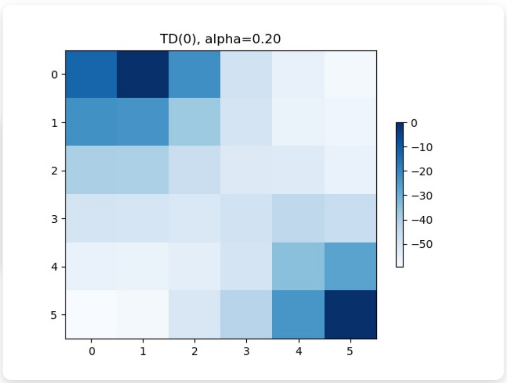
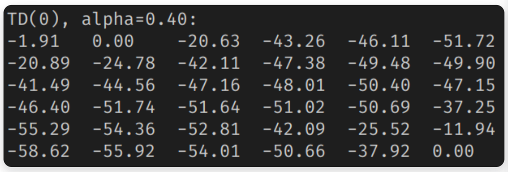


Fig.4 TD(0), alpha=0.2



```
TD(0), alpha=0.40:
-1.91   0.00    -20.63  -43.26  -46.11  -51.72
-20.89  -24.78  -42.11  -47.38  -49.48  -49.90
-41.49  -44.56  -47.16  -48.01  -50.40  -47.15
-46.40  -51.74  -51.64  -51.02  -50.69  -37.25
-55.29  -54.36  -52.81  -42.09  -25.52  -11.94
-58.62  -55.92  -54.01  -50.66  -37.92  0.00
```

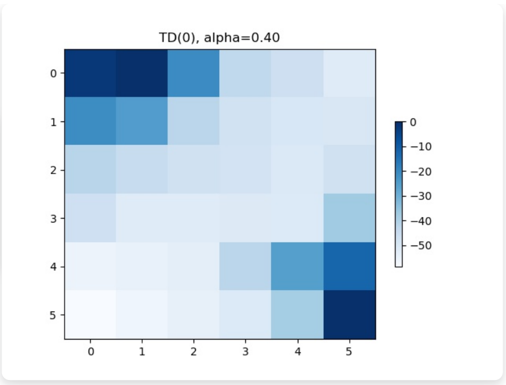
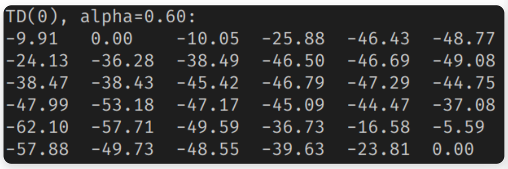


Fig.5 TD(0), alpha=0.4



```
TD(0), alpha=0.60:
-9.91   0.00    -10.05  -25.88  -46.43  -48.77
-24.13  -36.28  -38.49  -46.50  -46.69  -49.08
-38.47  -38.43  -45.42  -46.79  -47.29  -44.75
-47.99  -53.18  -47.17  -45.09  -44.47  -37.08
-62.10  -57.71  -49.59  -36.73  -16.58  -5.59
-57.88  -49.73  -48.55  -39.63  -23.81  0.00
```

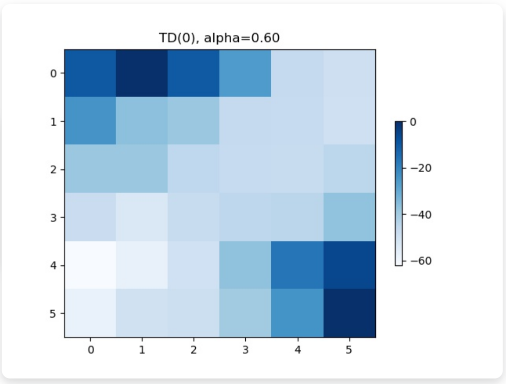
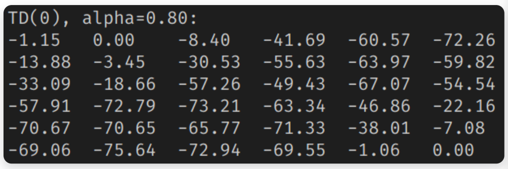


Fig.6 TD(0), alpha=0.6



```
TD(0), alpha=0.80:
-1.15   0.00    -8.40   -41.69  -60.57  -72.26
-13.88  -3.45   -30.53  -55.63  -63.97  -59.82
-33.09  -18.66  -57.26  -49.43  -67.07  -54.54
-57.91  -72.79  -73.21  -63.34  -46.86  -22.16
-70.67  -70.65  -65.77  -71.33  -38.01  -7.08
-69.06  -75.64  -72.94  -69.55  -1.06   0.00
```

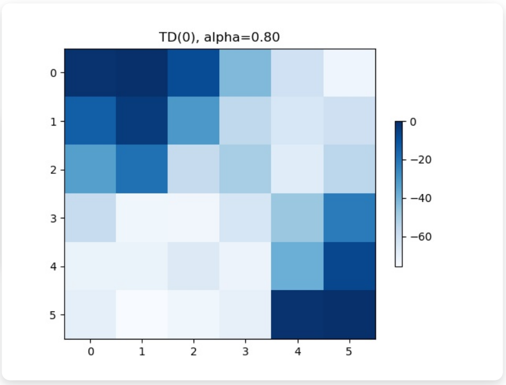


Fig.7 TD(0), alpha=0.8

As we can see, the smaller the α is, the closer the result is to the real value. However, we can still use TD(0) to estimate a policy, because the "trend" of the value is the same as the "trend" of true value.

3. Summary

In this assignment, I implemented MC and TD(0) and had a better understand of model-free algorithms.