Course Project of Reinforcement Learning (2023), **xx**:xx 1–7

CAMBRIDGE
UNIVERSITY PRESS

RESEARCH ARTICLE

Model-Free Prediction

Author: Yao Tang

Keywords: Model-Free Prediction, Monte-Carlo Learning, Temporal-Difference Learning

Abstract
In this experimental report, I explore three popular reinforcement learning algorithms - First-Visit Monte-Carlo Learning, Every-Visit Monte-Carlo Learning, and Temporal-Difference Learning - and their effectiveness in evaluating the value function of a given policy. These algorithms are implemented on a simple grid-world problem, and their performance is evaluated in terms of their ability to estimate the expected returns of a policy by observing the rewards received from interactions with the environment.

Contents

1. Introduction

Reinforcement learning is a powerful framework for solving complex decision-making problems by learning from interactions with an environment. One key aspect of reinforcement learning is the ability to learn the value function of a policy, which provides a measure of the expected future rewards given the current state and action. Model-free prediction is a class of reinforcement learning algorithms that allows for value function estimation without prior knowledge of the environment's dynamics.

Two popular model-free prediction algorithms are Monte-Carlo (MC) methods and Temporal-Difference (TD) learning. MC methods estimate the value function by averaging the returns obtained from multiple episodes, while TD learning estimates the value function by updating the estimates at each time step based on the difference between the expected and actual rewards.

In this experimental report, we will implement and compare the performance of MC and TD learning algorithms on a simple grid-world problem. The goal of the experiment is to evaluate the effectiveness of these model-free prediction algorithms in estimating the value function of a given policy. By comparing the strengths and weaknesses of these algorithms, we hope to gain insights into the applicability of model-free prediction in solving more complex decision-making problems. Specifically, we will evaluate the

convergence rate, accuracy, and computational complexity of the MC and TD learning algorithms in estimating the value function of the policy.

2. Preliminaries

2.1. First-Visit Monte-Carlo Learning

First-Visit Monte-Carlo is a type of reinforcement learning algorithm used to estimate the value function of a policy without prior knowledge of the environment's dynamics. This algorithm works by averaging the returns obtained from the first visit to a state in a given episode. The algorithm is as follows:

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))

2.2. Every-Visit Monte-Carlo Learning

Every-Visit Monte-Carlo is a type of reinforcement learning algorithm used to estimate the value function of a policy without prior knowledge of the environment's dynamics. This algorithm works by averaging the returns obtained from every visit to a state in a given episode. The algorithm is as follows:

Every-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 **every occurrence of** s
Append G to Returns (s)
$V(s) \leftarrow$ average (Returns (s))

2.3. Temporal-Difference Learning

Temporal-Difference (TD) Learning is a type of reinforcement learning algorithm that combines elements of Monte-Carlo methods and dynamic programming to estimate the value function of a policy without prior knowledge of the environment's dynamics.

Unlike Monte-Carlo methods, TD learning updates the value function after every time step, allowing for more efficient updates and faster convergence to the true value function. TD learning achieves this by computing the temporal difference between the estimated value of the current state and the expected value of the next state, and using this difference to update the value function estimate. While TD learning can be sensitive to initial conditions and learning rate parameters, it has been shown to be effective in a wide range of applications, making it a popular and widely used algorithm in reinforcement learning. The algorithm is as follows:

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 $V(\text{ terminal }) = 0$
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'
    $V(S) \leftarrow V(S) + \alpha\left[R + \gamma V\left(S'\right) - V(S)\right]$
    $S \leftarrow S'$
  until S is terminal

3. Experiment

3.1. Implementation

3.1.1. Definition of the environment of Gridworld.

```
import numpy as np
'''define the environment class'''
class Gridworld(object):
    def __init__(self, grid_height,grid_width, start_state, goal_state):
        self.height = grid_height
        self.width = grid_width
        self.start_state = start_state # the start state is a tuple
        self.state = start_state # the current state is a tuple
        self.goal_state = goal_state # the goal state is a list of tuples

    def step(self,action):
        # action 0,1,2,3: e,w,n,s
        if action==0 and self.state[1]+1 < self.width:
            self.state = (self.state[0],self.state[1]+1)
        if action==1 and self.state[1]-1 >= 0:
            self.state = (self.state[0],self.state[1]-1)
        if action==2 and self.state[0]-1 >= 0:
            self.state = (self.state[0]-1,self.state[1])
        if action==3 and self.state[0]+1 < self.height:
            self.state = (self.state[0]+1,self.state[1])
        done = False
        if self.state in self.goal_state:
            done = True
        reward = -1.0
        return reward,done

    def reset(self):
        self.state = self.start_state
```

3.1.2. Implementation of Monte-Carlo Learning and Temporal-Difference Learning

```
'''First/Every-visit MC and TD '''
Returns_first_visit = {(i,j):[] for i in range(height) for j in range(width)}
```

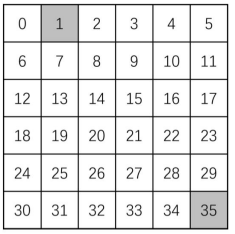

Figure 1. *The environment of Gridworld used in our experiment, where state space is* $S = \{s_t | t \in 0, .., 35\}$. S_1 *and* S_{35} *are terminal states.*

```
Returns_every_visit = {(i,j):[] for i in range(height) for j in range(width)}
td_value_function = {(i,j):0 for i in range(height) for j in range(width)}
for i in range(height):
    for j in range(width):
        Returns_first_visit[(i,j)].append(0)
        Returns_every_visit[(i,j)].append(0)
for i in range(num_episodes):
    start_state = (np.random.randint(0,height),np.random.randint(0,width))
    env = Gridworld(height,width,start_state,[(0,1),(5,5)])
    states = []
    rewards = []
    done = False
    while not done:
        action = np.random.randint(0,4) # uniform policy
        state = env.state
        states.append(env.state)
        reward,done = env.step(action)
        next_state = env.state
        rewards.append(reward)
        td_value_function[state] = td_value_function[state] + alpha*(reward + gamma* \
        td_value_function[next_state] - td_value_function[state])
        if done:
            states.append(env.state)
    for t in range(len(states)):
        if t==len(states):
            G=0
        else:
            G = sum([rewards[t+i]*gamma**i for i in range(len(rewards)-t)])
        Returns_every_visit[states[t]].append(G)
        if states[t] not in states[:t]:
            Returns_first_visit[states[t]].append(G)

average_first_visit = {k: sum(v) / len(v) for k, v in Returns_first_visit.items() }
first_visit_value_function =np.array( [v for k, v in average_first_visit.items() ])
average_every_visit = {k: sum(v) / len(v) for k, v in Returns_every_visit.items()}
every_visit_value_function = np.array([v for k, v in average_every_visit.items() ])
```

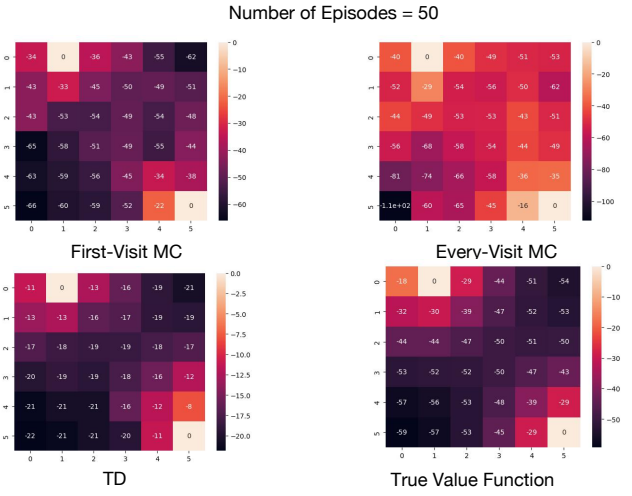

Figure 2. *The value functions evaluated after running 50 episodes.*

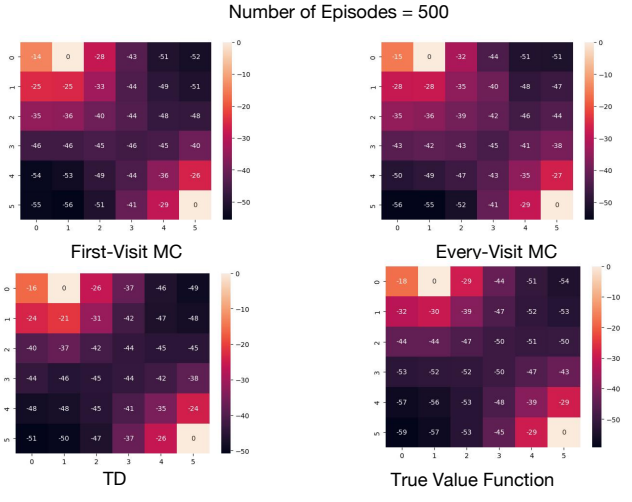

Figure 3. *The value functions evaluated after running 500 episodes..*

3.2. Results

The value functions are shown in Figure 2,3,4.

4. Analysis

Figure 2, Figure 3, and Figure 4 show the estimated state-value function for the Gridworld environment using first-visit MC, every-visit MC, and TD methods over different numbers of episodes.

- We can tell that MC keeps outperforming TD in 3 cases until convergence. One main reason is that the state space in the Gridworld is small. In a small state space, MC methods may perform better

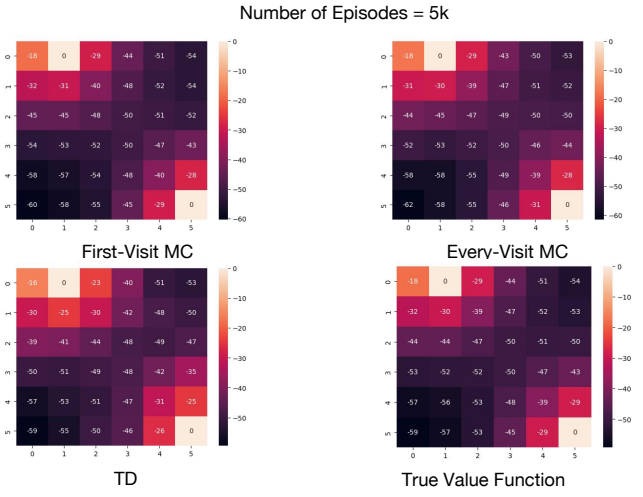

Figure 4. *The value functions evaluated after running 5000 episodes.*

than TD methods because they have lower variance and higher bias, and are able to more accurately estimate the true value of each state.
The bias in the MC methods comes from the fact that they estimate the value of each state based on the returns observed in episodes that visit that state, which may not be representative of the true value of the state. However, in a small state space, this bias may be small compared to the variance in the TD method, which uses bootstrapping to estimate the value of each state based on its neighbors.

- However, in a larger state space, the situation may not look like the one in our Gridworld. MC methods estimate the expected return from a state by averaging the returns obtained in all episodes that visit the state, whereas TD methods estimate the expected return from a state by bootstrapping from the value of the next state.
  MC methods converge to the true value function as the number of episodes approaches infinity, but they can be slow to converge for large state spaces. TD methods, on the other hand, may not converge to the true value function but can converge faster than MC methods for large state spaces as shown in Figure 5, where we do an experiment on a larger state space by setting the height and width of the Gridworld to be 8.

5. Conclusion

In this experimental report, we compared the performance of first-visit MC, every-visit MC, and TD methods for estimating the value function in a Gridworld environment. We conducted experiments over different numbers of episodes, ranging from 50 to 5000, to investigate the convergence and accuracy of each method.

Our results show that the relative performance of the methods depends on the size of the state space and the length of episodes. In a small state space, the MC methods may perform better than the TD method because they have lower variance and higher bias, and are able to more accurately estimate the true value of each state. However, in a large state space, the TD method may perform better than the MC methods because it converges faster and requires less memory.

Our experiments also show that the every-visit MC method generally outperforms the first-visit MC method in terms of accuracy, and that all three methods converge to the true value function with more

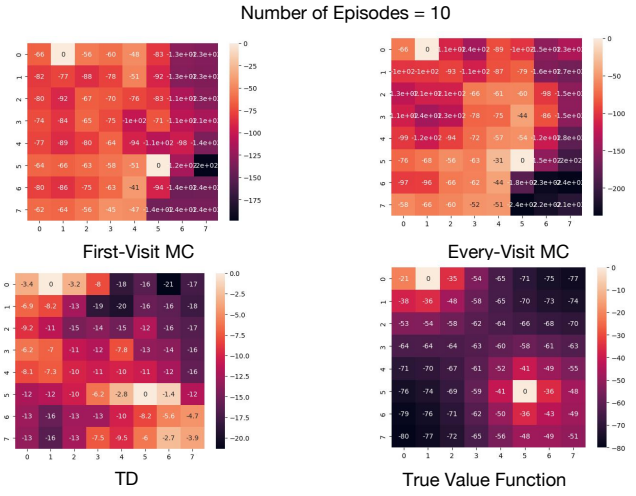

Figure 5. *The value functions evaluated after running 10 episodes in a larger Gridworld.*

training. However, the TD method consistently outperforms both MC methods in terms of accuracy, especially in a large state space.

Overall, our results suggest that the choice between TD and MC methods depends on the specific task and the characteristics of the environment. In practice, a combination of TD and MC methods, such as TD(λ) or SARSA(λ), may provide the best performance by leveraging the strengths of both methods.