CS698R: Deep Reinforcement Learning

Mid-Semester Exam

Name: Kislay Aditya Oj
Roll NO.: 210524

Solution to Problem 1: Random-Maze Environment Implementation

1. The Random Maze Environment was implemented successfully with few test cases. One of the test cases is attached below -

```
{'current_state': 8, 'action': 1, 'next_state': 9}
{'current_state': 9, 'action': 0, 'next_state': 9}
{'current_state': 9, 'action': 3, 'next_state': 8}
{'current_state': 8, 'action': 2, 'next_state': 8}
{'current_state': 8, 'action': 1, 'next_state': 9}
{'current_state': 9, 'action': 2, 'next_state': 9}
{'current_state': 9, 'action': 1, 'next_state': 10}
{'current_state': 10, 'action': 3, 'next_state': 9}
{'current_state': 9, 'action': 2, 'next_state': 9}
{'current_state': 9, 'action': 3, 'next_state': 8}
 Average Reward over 10 episode =  -0.04
```

As we can see , the test case was ran for 10 iterations and the average reward over that are calculated. As the agent did not reach a terminal state the average reward is -0.04 which is same as the living penalty thus confirming the correctness of the code logic.

Solution to Problem 2: RME Optimal Policy via Dynamic Programming

1. For the choice of Random policy , I chose always go Up policy which means the agent will always take the action 0 no matter what the current state is with 100 percent probability. For the Policy Iteration algorithm we got the optimal policy as shown in Figure 1 .In this case the PI algorithm required only 4 iterations to converge.
2. In the case of Value Iteration algorithm the policy taken was same as above and it took 5 iterations to converge to an optimal policy which is shown in Figure 1.
3. In the above two cases where the initial policy is taken as always go up , the Policy Iteration converged one step earlier but to different conclusions. Both of them did not converge to same optimal policy because the initial position and environment dynamics affects the evaluation of both the algorithms. Policy Iteration algorithm compares policies starting with certain random policy.It stops when two consecutive policy repeats. While value iteration compares values of the states and gives the best possible course of action based on maximum of action value pairs.

Solution to Problem 3: RME Prediction with MDP Unknown

1. The function Generate Trajectory was created and tested for various cases. It was working as expected.
2. The plot for Decay Function (both linear and exponential) is given in Figure 2.
3. The test cases and observations are as shown in Figure 3. Both algorithm seems to be with somewhat high variance the values quickly converges corresponding to it's proximity to goal or hole state.

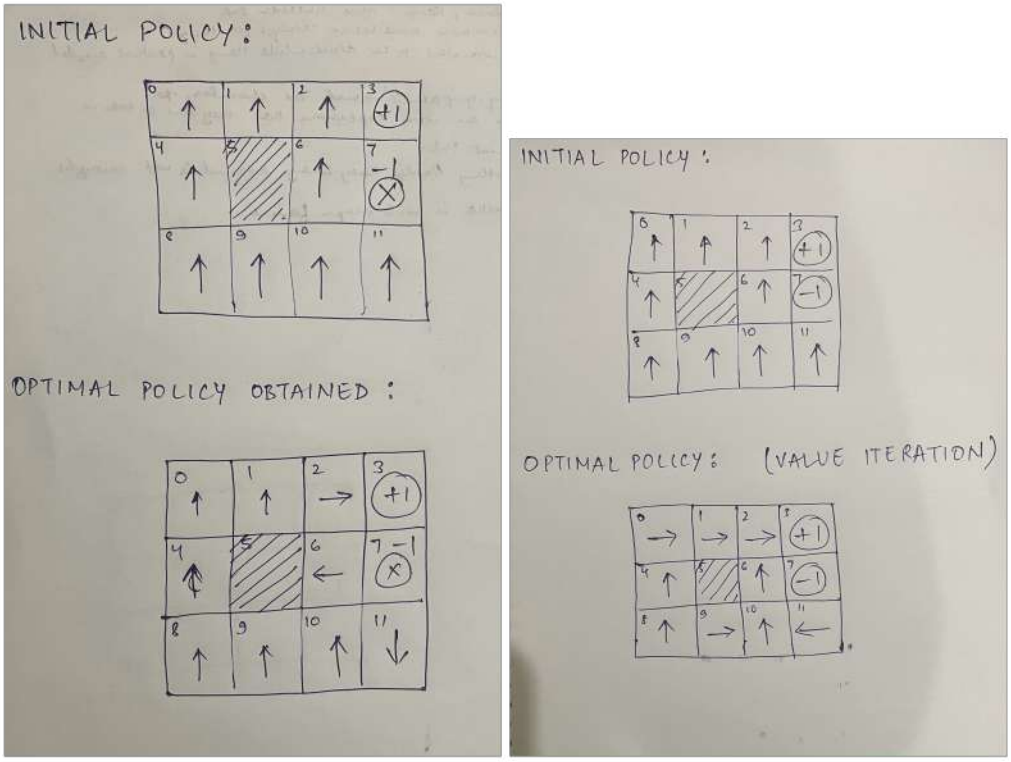

Figure 1: Policy Iteration(left) and Value Iteration(right)

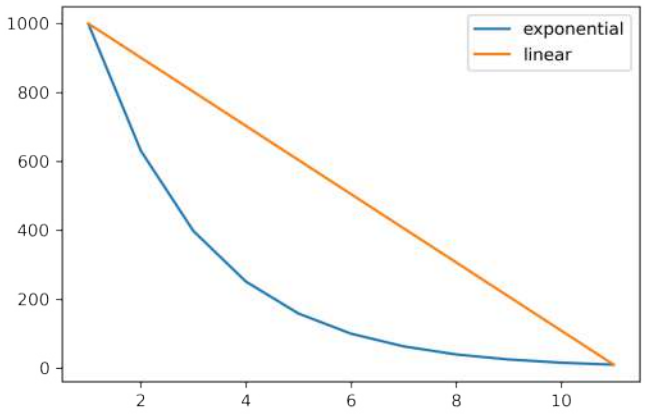

Figure 2: Exponential vs Linear decay

```
environment = make('RMaze' )
observation = environment.reset(seed=0)
MonteCarloPrediction( environment, 100,5,optimal_policy)
```

```
array([[ 0.00486393,  0.01725688,  0.1       ,  0.        , -0.00319286,
         0.        ,  0.        ,  0.        , -0.00716093,  0.        ,
         0.        ,  0.        ],
       [ 0.01861573,  0.05125961,  0.1738    ,  0.        ,  0.00779809,
         0.        ,  0.        ,  0.        ,  0.00076811,  0.        ,
         0.        ,  0.        ],
       [ 0.0005194 ,  0.043396  ,  0.2266768 ,  0.        , -0.01670728,
         0.        ,  0.        ,  0.        , -0.02560728,  0.        ,
         0.        ,  0.        ],
       [ 0.00651429,  0.07394022,  0.26224967,  0.        , -0.01182015,
         0.        ,  0.        ,  0.        , -0.02219193,  0.        ,
         0.        ,  0.        ],
       [-0.03498409,  0.03271757,  0.28150668,  0.        , -0.053512  ,
         0.        ,  0.        ,  0.        , -0.06429315,  0.        ,
         0.        ,  0.        ]])
```

```
environment = make('RMaze' )
observation = environment.reset(seed=0)
MonteCarloPrediction( environment, 100,5,optimal_policy)
```

```
array([[ 0.1374131 ,  0.43293395,  0.1       ,  0.        , -0.08190726,
         0.        ,  0.        ,  0.        , -0.08508818,  0.        ,
         0.        ,  0.        ],
       [-0.35948673,  0.36131983,  0.1738    ,  0.        , -0.2338815 ,
         0.        ,  0.        ,  0.        , -0.38086056,  0.        ,
         0.        ,  0.        ],
       [ 0.20783671, -0.04073683,  0.2266768 ,  0.        , -0.29446342,
         0.        ,  0.        ,  0.        , -0.43384032,  0.        ,
         0.        ,  0.        ],
       [ 0.27849721,  0.24513482,  0.26224967,  0.        , -0.26540068,
         0.        ,  0.        ,  0.        , -0.41535238,  0.        ,
         0.        ,  0.        ],
       [ 0.35147004,  0.30586489,  0.28290668,  0.        , -0.25430585,
         0.        ,  0.        ,  0.        , -0.40121555,  0.        ,
         0.        ,  0.        ]])
```

Figure 3: MCEV and MCFV test cases

4. The test cases and observations are as shown in Figure 4. We can see that converging rate of Temporal difference algorithm is faster than the above Monte Carlo estimates. The number of episodes taken is only 5 .

```
environment = make('RMaze' )
observation = environment.reset(seed=0)
TemporalDifference( environment , 5 , 0.1 , optimal_policy)
```

```
array([[ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ,  0.        ],
       [-0.03662349, -0.03028887,  0.06309573,  0.        , -0.00252383,
         0.        ,  0.        ,  0.        , -0.00252383,  0.        ,
         0.        ,  0.        ],
       [-0.03662349, -0.03028887,  0.10039457,  0.        , -0.00252383,
         0.        , -0.00827073,  0.        , -0.00401578, -0.00464989,
        -0.00159243,  0.        ],
       [-0.08322723, -0.05772986,  0.12299163,  0.        , -0.00535483,
         0.        , -0.00827073,  0.        , -0.00498243, -0.00464989,
        -0.00159243,  0.        ],
       [-0.11135164, -0.05870404,  0.13689128,  0.        , -0.00720979,
         0.        , -0.00827073,  0.        , -0.00623944, -0.00716743,
        -0.00159243,  0.        ]])
```

Figure 4: TD test cases

5. The test cases and observations are as shown in Figure 5. n-step TD learning comes somewhere between MC estimates and TD learning . The values of the estimates also depends on the value of n taken .

```
ev = make('RMaze' )
observation = ev.reset(seed=0)
n_stepTemporalDifference( ev , optimal_policy , 5 , 5 )
```

```
array([[-0.14702985,  0.        ,  0.5       ,  0.        ,  0.        ,
         0.        ,  0.        ,  0.        , -0.0980199 ,  0.        ,
         0.        ,  0.        ],
       [-0.14702985,  0.12593265,  0.5       ,  0.        ,  0.        ,
         0.        ,  0.        ,  0.        , -0.1204324 ,  0.        ,
         0.        ,  0.        ],
       [-0.21566603, -0.11550383,  0.5       ,  0.        ,  0.        ,
         0.        ,  0.        ,  0.        , -0.14295875,  0.        ,
         0.        ,  0.        ],
       [-0.33863416, -0.18301618,  0.42683903,  0.        ,  0.        ,
         0.        ,  0.        ,  0.        , -0.15530418,  0.        ,
         0.        ,  0.        ],
       [-0.33863416, -0.17442463,  0.43937248,  0.        ,  0.        ,
         0.        ,  0.        ,  0.        , -0.16000087,  0.        ,
         0.        ,  0.        ]])
```

Figure 5: n-step TD test cases

6. The test cases and observations are as shown in Figure 6. TD lambda algorithm helps us to reduce the variance via maintaining a Eligibility matrix. The matrix keeps a memory of states and give higher values to previously visited states.

```
ev = make('RMaze' )
observation = ev.reset(seed=0)
TDLambdaPrediction( ev , optimal_policy , 5 , 0.1 )
```

```
array([[-0.00217534, -0.03376696,  1.10296   ,  0.        , -0.04439025,
         0.        ,  0.        ,  0.        , -0.04439463,  0.        ,
         0.        ,  0.        ],
       [-0.03228364,  0.3837275 ,  1.06087685,  0.        , -0.04648106,
         0.        ,  0.        ,  0.        , -0.07078446,  0.        ,
         0.        ,  0.        ],
       [-0.21614011,  0.46814936,  1.0509884 ,  0.        , -0.05121016,
         0.        ,  0.        ,  0.        , -0.07366671,  0.        ,
         0.        ,  0.        ],
       [-0.15293457,  0.45001003,  1.04759015,  0.        , -0.06007835,
         0.        ,  0.        ,  0.        , -0.07561927,  0.        ,
         0.        ,  0.        ],
       [-0.14265699,  0.43861354,  1.04632424,  0.        , -0.06478274,
         0.        ,  0.        ,  0.        , -0.07634967,  0.        ,
         0.        ,  0.        ]])
```

Figure 6: TD lambda test cases

7. The true value of states are calculates as in Figure 7. We used Bellman's equation assuming MDP process to get a solution via system of linear equations.

8. The MC-FV plot is given in Figure 8. The legend is ordered 0 to 11 from top to bottom. We can see that some of the values quickly come close to it's true values.One of reason that this plot has very less noise is that the alpha value taken here is quite low (starts from 0.1 and go upto 0.01) . More the value of alpha more the noise in graph.

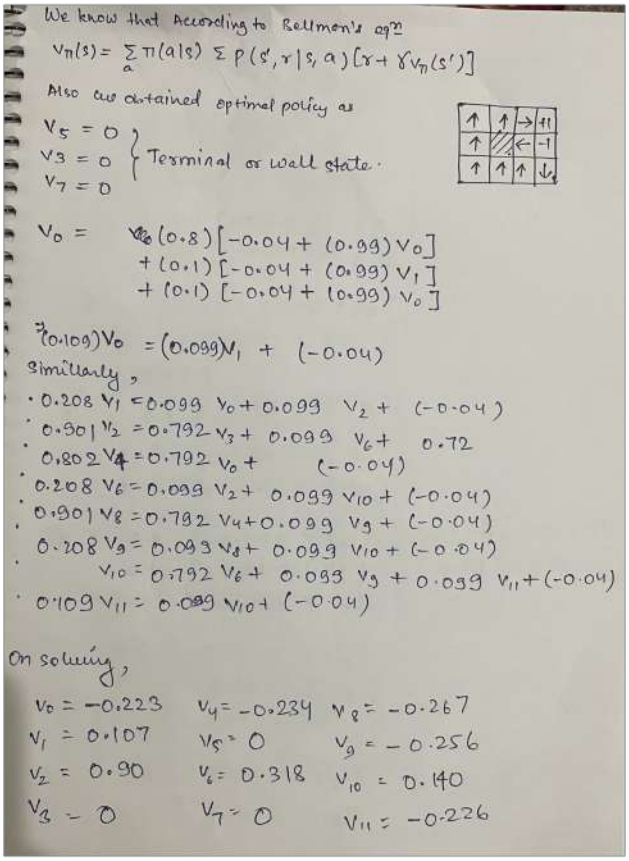

Figure 7: Solution

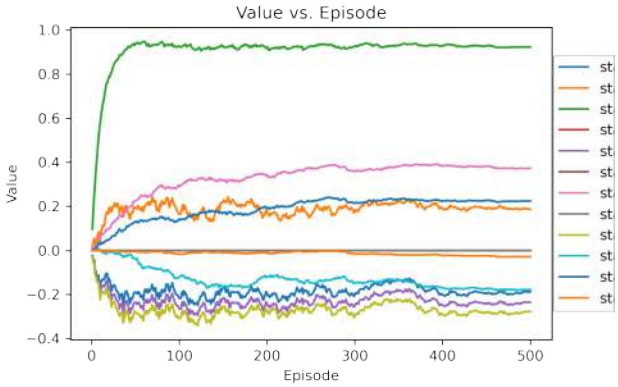

Figure 8: MC-FV estimates vs episodes

9. The every visit plot is given by Figure 9. One of the things we can notice is the amount of noise in the initial portion of the graph. The every visit graph has more noise than the first visit despite having same alpha is because it takes in more values of the state unlike in FV where per episode only one value is taken.

10. The Temporal Difference plot is given by Figure 10. As seen in the figure TD algorithms are quickest to converge to the true values. This is because unlike MC it does not wait for entire episode or trajectory but instead keeps updating the value as the program runs.

11. The n-step Temporal difference plot is given by Figure 11. Compared to TD one step it has more noise , which is confirmation of the case that it lies between two extremes of MC estimates and TD one step .

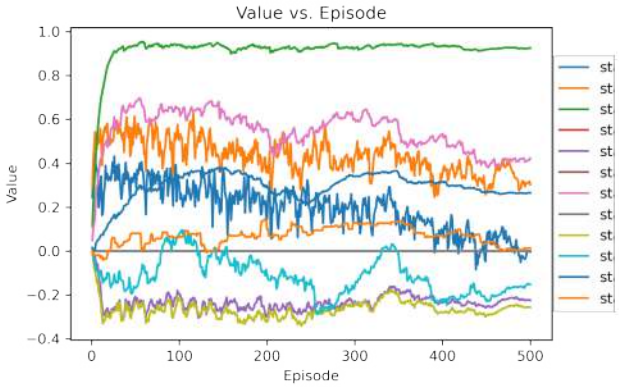

Figure 9: MC-EV estimates vs episodes

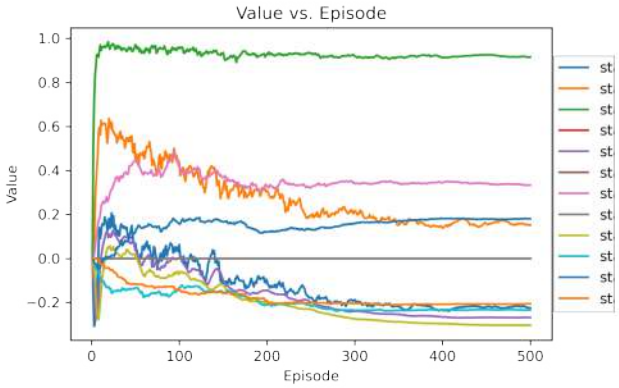

Figure 10: TD estimates vs episodes

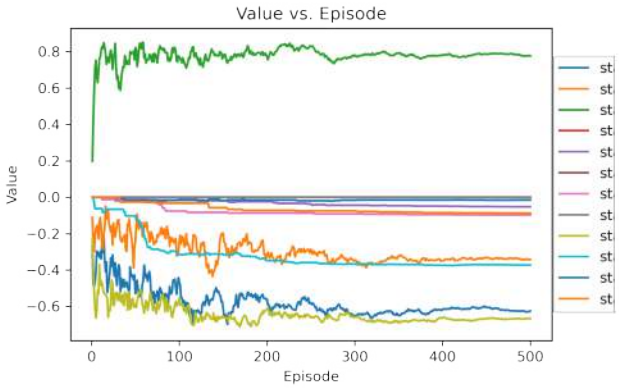

Figure 11: n step TD estimates vs episodes

12. The TD lambda plot plot is given by Figure 12. It starts with high variations but eventually dries down to the original values.

13. The plot of Eligibility trace of TD lambda algorithm at episode 100 is given by Figure 13.

14. The MC-FV log plot is given in Figure 14. In the initial stages we can see that the state which is closer to goal i.e state 2 attains a very high value very early. This is because the policy chosen is very optimal. Also the alpha taken here is quite low.

15. The every visit log plot is given by Figure 14. As usual the plot has higher noise than FV for the same

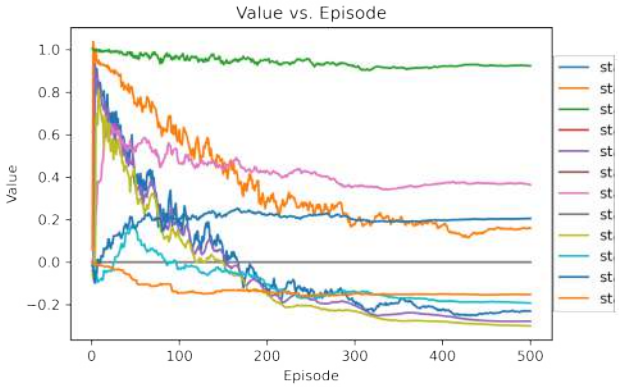

Figure 12: TD lambda vs episodes

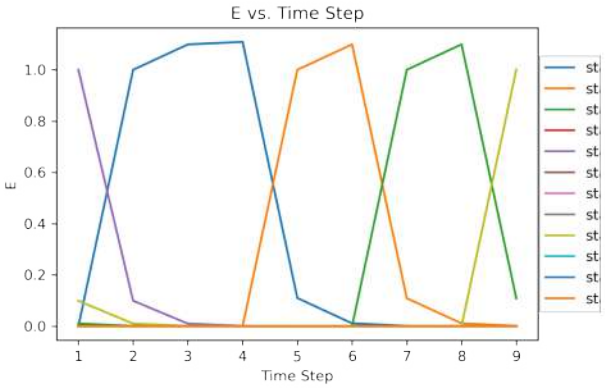

Figure 13: E vs Time step

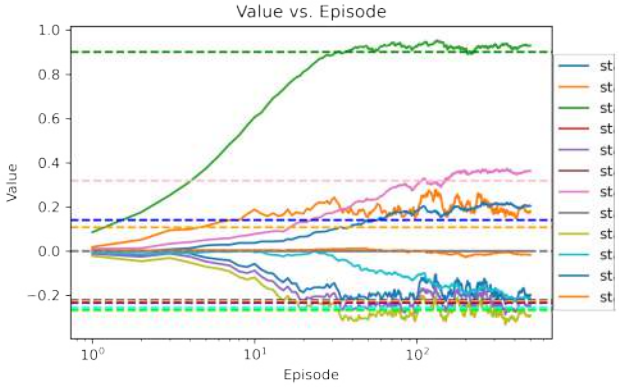

Figure 14: MC-FV estimates vs episodes (log)

reasons mentioned above. The value of the states oscillates around the mean value.

16. The Temporal Difference plot is given by Figure 15. Compared to other two plots the TD plots are faster but quite biased.

17. The n-step Temporal difference plot is given by Figure 16. To reduce this biasing n number of steps is used instead of one. This also comes at a cost of high varience.

18. The TD lambda plot plot is given by Figure 17. On eof thing to observe here is initially all values are higher than their true estimates but then they reduces to it's own value. It's kind of similar to DP

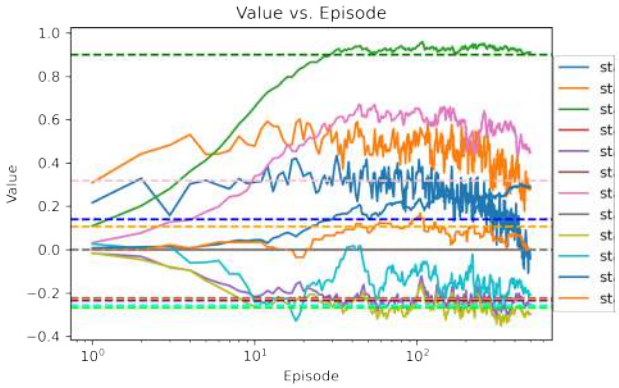

Figure 15: MC-EV estimates vs episodes(log)

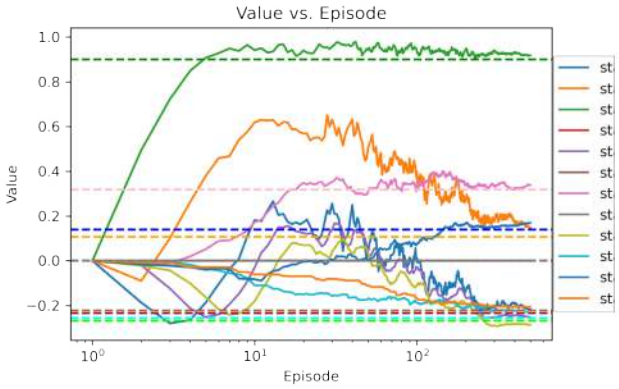

Figure 16: TD estimates vs episodes(log)

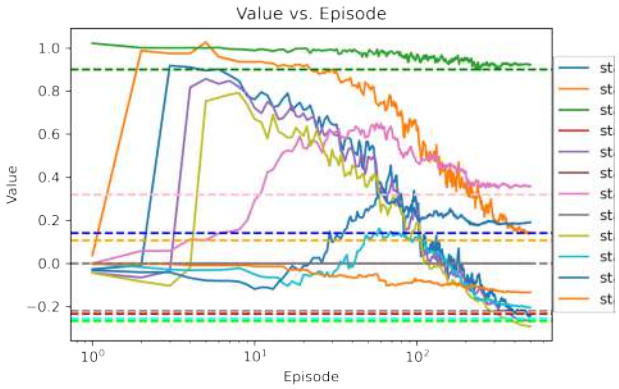

Figure 17: n step TD estimates vs episodes (log)

approaches.

19. Based on the above plots we can conclude that
    a. MC FV which averages returns from only the first visit has unbiased estimates, high variance and slow convergence.
    b. MC FV which averages returns from all the visits has less biased than FVMC, higher variance and slower convergence.
    c. TD learning which uses one step updates is faster than MC , has lower variance and can be biased.
    d. n step TD which uses n steps updates trades off bias and variance. Higher n reduces bias and increase

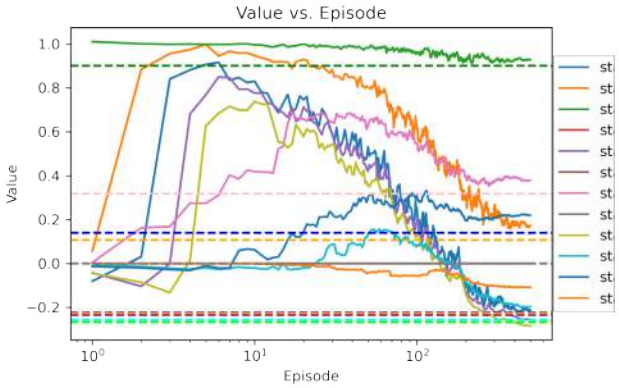

Figure 18: TD lambda vs episodes (log)

variance.
e. TD lambda combines MC and TD ideas which can be adjusted using lambda.

The plot concerning state 6 of every algorithm is given in Figure 19.

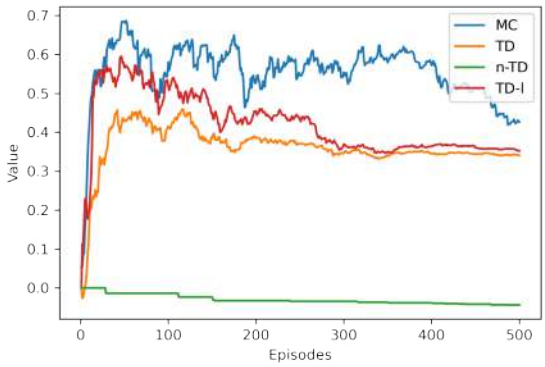

Figure 19: Different approaches for same state 6

20. The plot of MCFV Target value of state 6 is given by (Figure 20). We can see that most of the points are above 0 and closer to one. This is because of it's closeness to the reward state. Also some values are below 0 because the hole state is also below the goal state.

21. The plot of MCEV Target value of state 6 is given by Figure 21. In every state visit the target are either 0 or one because it consider every values of that the agent visited in the episodes. With every visit the values becomes closer to what it originally intended in that state.

22. The plot of TD Target value of state 6 is given by Figure 21. Compared to MCEV many values are closer to one but not one , this is because TD does not consider the full episode but only the next state. This reduces the overall value from one after discounting it by gamma.

23. Observing the three plots we can see that the value of state 6 clearly lies between 0 and 1 with 0 weighted more than one. We can confirm this by it's true value calculated in question 7 (i.e 0.318) . Also , for other state like state 2 , the target plots with be concentrated near one as the value of that state is 0.9. Similarly the average of each target value plots of each state must be equal or close to their corresponding true values .

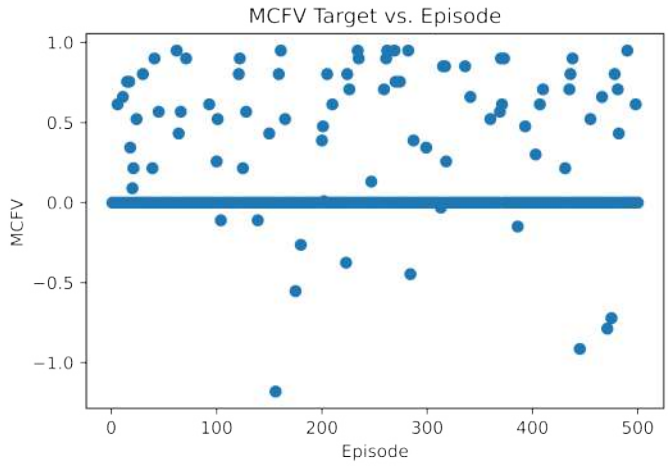

Figure 20: MCFV Target value vs episodes

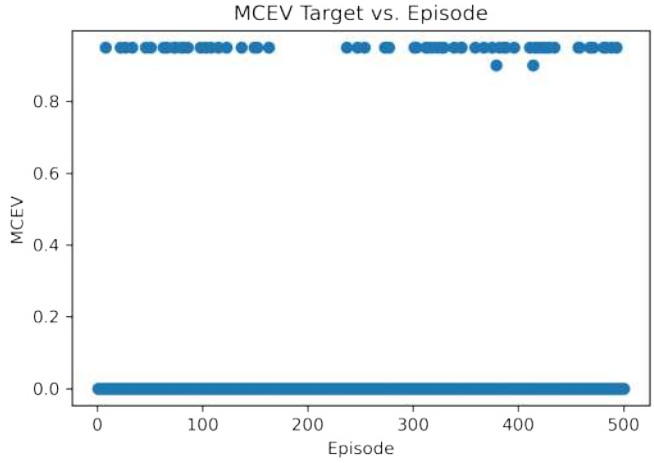

Figure 21: MCEV Target value vs episodes

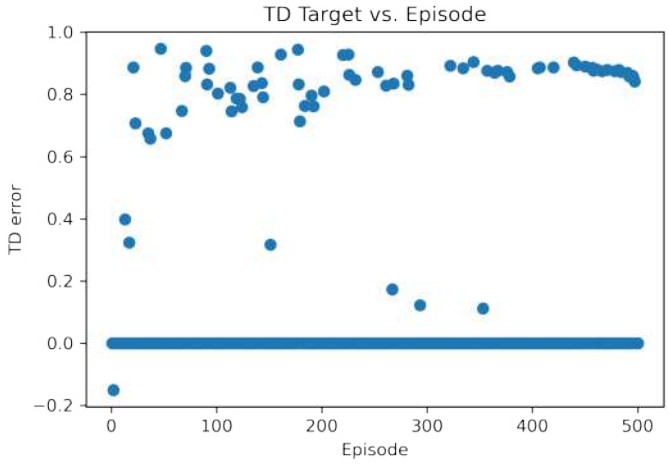

Figure 22: TD Target value vs episodes

References

1. CS780 Lecture Slides.