CS698R: Deep Reinforcement Learning

Mid-Semester Exam

Name: Samrudh Govindaraj
Roll NO.: 20128409

Solution to Problem 1: Random-Maze Environment Implementation

The default seed value(31) is used in the project wherever the seed value has not been specified.The Random Maze environment was created with 8 with a discrete observation space of size 12 and a discrete of action space of size 4, for the different actions (up,down,left,right). Two terminal states were included, a goal state at state 3 and a hole state at state 7. A wall was also created at state 5. The agent starts exploring from state 8 and the episode is terminated when the agent reaches either the goal or hole states and receives a reward of 1 and -1 respectively. A living penalty of 0.04 is added for each time step in the episode. The agent moves in the intended directed with $P = 0.8$ and each of the orthogonal directions with $P = 0.1$. The gamma value used here is 0.99 and the seed value is 31. The plots show that the

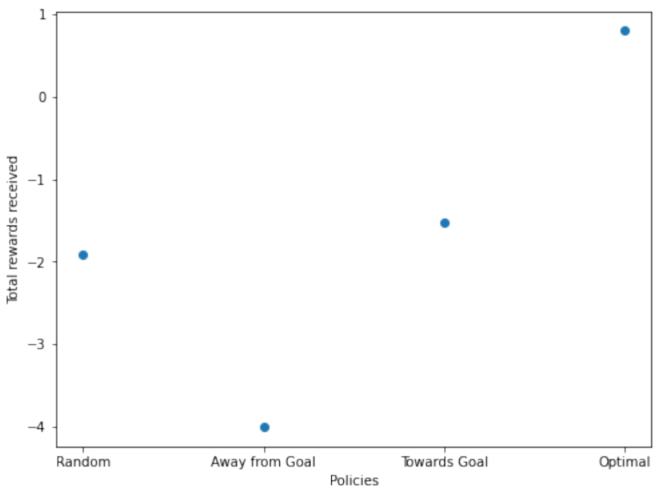


Figure 1: Comparison of different policies

environment works in the expected way as 'better policies' have greater rewards compared to detrimental and random policies. In the away from goal policy, the agent moved only down and left and in the towards goal policy the agent moved only up and right. The optimal policy used was [3,3,3,0,0,0,0,0,0,3,0,2]. The maximum number of steps used was 100.

Solution to Problem 2: RME Optimal Policy via Dynamic Programming

1. To use the dynamic programming algorithms, the reward vector and transition probabilities were created first using the information mentioned above (s(3)=+1, s(7)=-1, others=-0.04). The policy evaluation algorithm was then implemented which takes in a policy and calculates a value function for the different states, using the specified policy v_π. Since the underlying dynamics are known, we can use the transition probabilities and the rewards for each state to calculate how 'valuable' the policy is and compare it to other policies. The policy improvement algorithm was then implemented which takes in a value vector and calculates the q-values for each state-action pair. The ideal policy is then calculated as the actions which have the highest q-values in each state.
A combination of the two algorithms is then used in the policy iteration algorithm where we start off with a random policy. The policy is then fed to the policy evaluation algorithm which gives us the

value approximations using the particular policy. The v function is then fed into the policy improvement algorithm which gives us an improved policy. The cycle is repeated until the difference of the value functions between two iterations, is minimal $\theta = 10^{-10}$. The initial policy used here was, [2 3 0 2 2 3 2 0 0 1 2 0]. The optimal policy obtained from the algorithm is [3 3 3 0 0 0 0 0 0 3 0 2]. The algorithm took 4 iterations to converge to the final policy (seed=31).

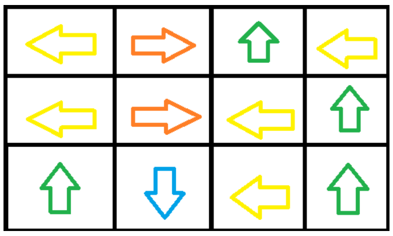


Figure 2: Initialised Random Policy

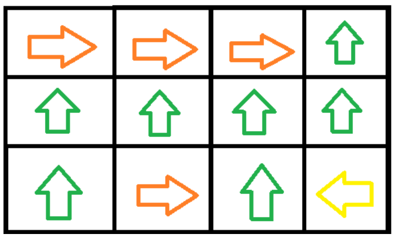


Figure 3: Final Optimal Policy

The Optimal policy is logical, such that in the first row the agent moves towards the goal state. In the second row, the agent moves up instead of right to avoid the hole in state 7. In the third row the agent moves up for the same reason. In state 9 the agent moves right instead of up in order to avoid hitting the wall. In state 11 the agent moves left in order to avoid the hole state above.

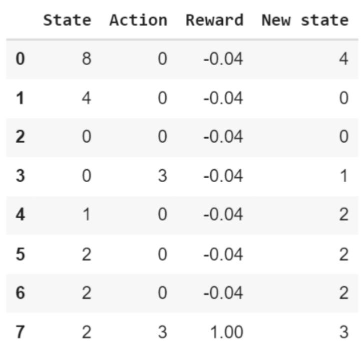
2. The Value iteration algorithm, we start off with a random policy and null values for the V and Q functions. We then iterate over all possible states and actions and if they are possible in the environment, we calculate the q-values based on the known transition probabilities, reward values and online approximated values for the states. The Q value with the maximum value is chosen as the value for the states on every iteration. The policy is chosen in the same manner as PI (argmax of Q-values). If the difference between the V values on 2 iterations is minimal, $\theta = 10^{-10}$, the function is terminated and existing values are returned. Using the same random policy generated for the PI algorithm, [2 3 0 2 2 3 2 0 0 1 2 0], the optimal policy obtained from the VI algorithm is [3 3 3 0 0 0 0 0 0 3 0 2]. The optimal policy was obtained in 28 iterations. (Please note, the policies are not shown visually as they are identical to the ones displayed above.)

3. As we can see, the two optimal policies obtained from the algorithms are identical. However, the PI algorithm converged in 4 iterations while the VI algorithm required 28 iterations. The PI algorithm however requires the usage of two other helper functions and the policy evaluation algorithm runs for a maximum of 100 iterations. However, considering this the PI algorithm is still much faster than VI. If the number of actions and states are very large, running through all possible combinations, as required in VI, can be computationally expensive. Since both the algorithms are guaranteed to converge to the true value function, given sufficient time, the algorithms will end up with the same optimal policy. However things can get interesting if the number of iterations are limited. The algorithms need not converge to the same policy in limited time as the PI algorithm is faster than the VI algorithm. The values are also calculated slight differently in the two algorithms and therefore the policy need not be the same.

Solution to Problem 3: RME Prediction with MDP Unknown

1. A function was created to generate a trajectory in the RME environment. The default values for the initial state was set to 8 and for the maximum number of steps before termination was set to 100. The

function also provides the option of selecting actions randomly from a given set of choices or selecting them deterministically from a given policy.



	State	Action	Reward	New state
0	8	0	-0.04	4
1	4	0	-0.04	0
2	0	0	-0.04	0
3	0	3	-0.04	1
4	1	0	-0.04	2
5	2	0	-0.04	2
6	2	0	-0.04	2
7	2	3	1.00	3

Figure 4: Generated Trajectory

As we can see the agent starts off at state 8 and goes up (action 0) till reaching state 0. The agent then moves to state 1 by moving right. The agent then tries moving up multiple times but is unsuccessful. The agent then moves to state 3 and receives reward 1 and the episode is terminated (seed=31).

2. A function was created to decay the α parameter with both linear and exponential decay functionality. The alpha value was initialised to 1 and decayed to 0.01 in 100 steps. The linspace and geomspace functionality in the numpy library was used to implement the linear and exponential decay functions.

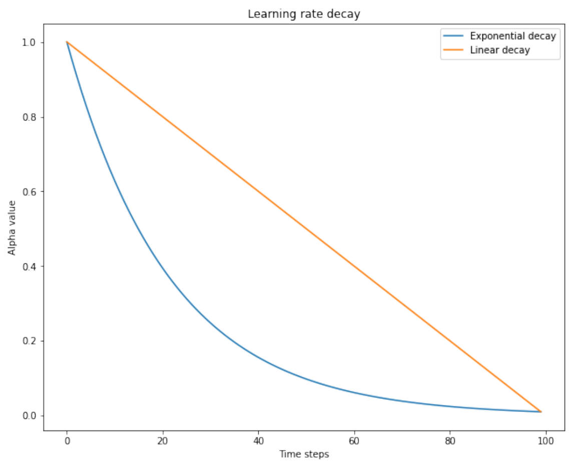


Figure 5: Decay of alpha values

3. The Monte Carlo algorithm is used in situations where we do not have prior knowledge of the underlying MDP dynamics. The value of a state, V(s) is the expected return from a state (s), where the return is the sum of reward received. The algorithm approximates the value of a state (s) by drawing a large number of sample trajectories from the state (s) and taking the average of all the returns.

$$V_{e+1}(s) = \frac{G_1 + G_2 + G_3.... + G_n}{N_s(e)}$$

The implemented Monte Carlo algorithm incrementally changes the value for a state (s) per episode. It does this by calculating the MC error, or difference between the received return in the episode and current value of the state. The MC error is then multiplied by the learning rate and added to the current value for the state.

$$V_{e+1}(s) = V_e(s) + \alpha(e)[G_e - V_e(s)]$$

There are two variants of the Monte Carlo algorithm, First Visit Monte Carlo (FVMC), where only the first visit to a state (s) in an episode is considered and Every Visit Monte Carlo (EVMC), where every visit to a state (s) in an episode is considered.

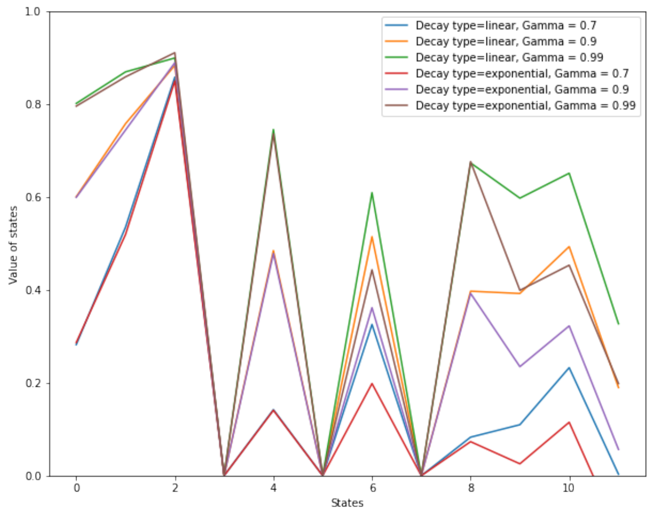


Figure 6: Test cases for FVMC algorithm

As we can see, the values for the terminal and wall states, 3 and 5 and 7 are 0. This is the expected return in these states is 0. The value for state 2 which is closest to the goal state is the highest. The values of other states are dependent on their distance from the goal state. The decay type and gamma values were manipulated here. As gamma increases, the valuation of the states also increase. We also notice that the values for exponential decay learning rates are slightly lower compared to linear decay. This is because, as seen above, the exponential decay rates drop much faster than linear decay rates and therefore the value updation becomes smaller much quicker. The seed used here is 31 and maximum number of episodes is 500.

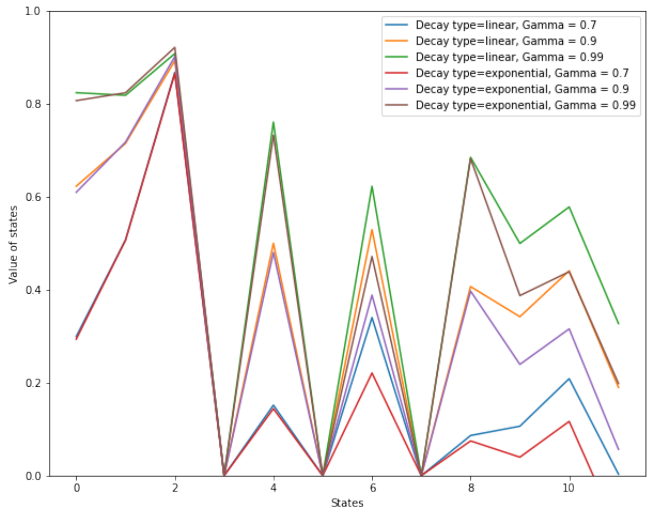


Figure 7: Test cases for EVMC algorithm

A similar plot is seen in the EVMC condition using the same hyper-parameter and seed values.

4. The temporal difference algorithm functions differently such that it approximates the value of a state, V(s) by using the reward received at the next state and the approximated value of the new state V(s') at every time step(t). The difference between the target value and current value of the state is the TD error. The TD error is multiplied by the learning rate and added to the current value of the state at every time step, to get the new value, V t+1(s).

$$V_{t+1}(s) = V_t(s) + \alpha(e)[R_{t+1} + \gamma V_t(s+1) - V_t(s)]$$

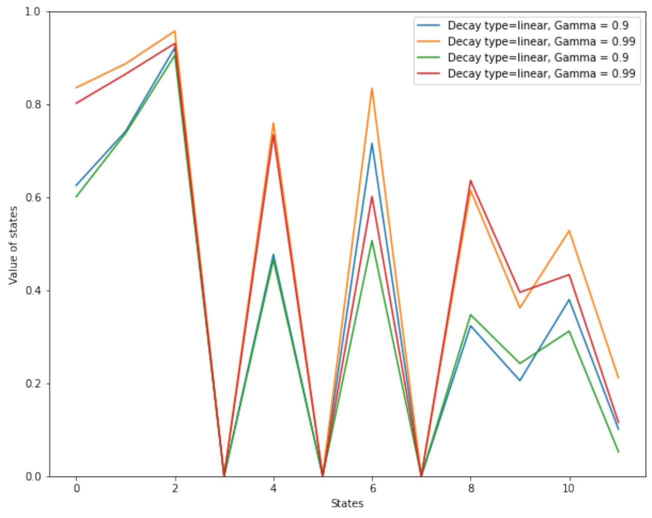


Figure 8: Test cases for TD algorithm

The TD algorithm shows similar findings when compared to the MC algorithm. However the valuation for the states is higher in the TD algorithm.

5. -

6. The TD algorithm uses an eligibility trace as a vector for short term memory. The trace is then used to update the values of all visited states at every time step.Therefore, states that have been visited are continuously updated depending on the value of the future states. The algorithm was implemented with default lambda value of 0.3 and run for 500 episodes.

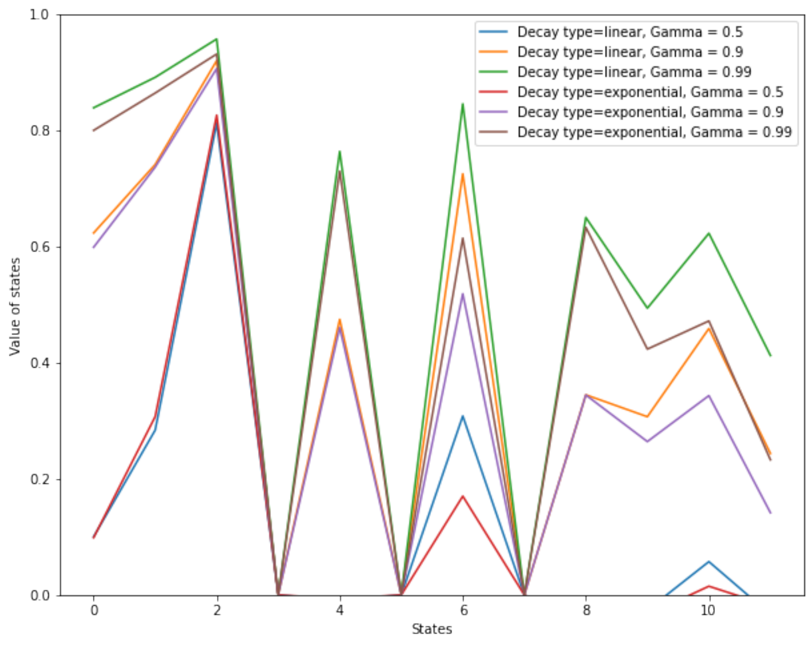


Figure 9: Test cases for TD lambda algorithm

The results seem to mimic the values found using the other algorithms but seems to show slightly higher estimates than the TD algorithm along with a greater difference for lower values of gamma.

7. The value iteration algorithm mentioned above was used to find the optimal V values for the 12 states. The obtained values are

[0.50822459,0.64083803, 0.84954296, 0., 0.37051388, 0., 0.54083803, 0., 0.28888143, 0.31751957, 0.44131259, 0.21751957]. The V values for the terminal and wall states are 0 and for the states closer to the goal state are higher than those further away.

8. Fifty seed values were generated to create fifty instances of the environment. The averaged reward over the fifty environments for 500 episodes was plotted for the different algorithms. The following seed values were used: [17 33 76 3 56 0 37 8 60 67 36 7 59 73 34 74 43 47 80 2 14 15 20 39 22 69 78 38 19 63 21 79 30 45 97 91 4 9 85 53 5 29 50 64 68 66 26 83 88 35].

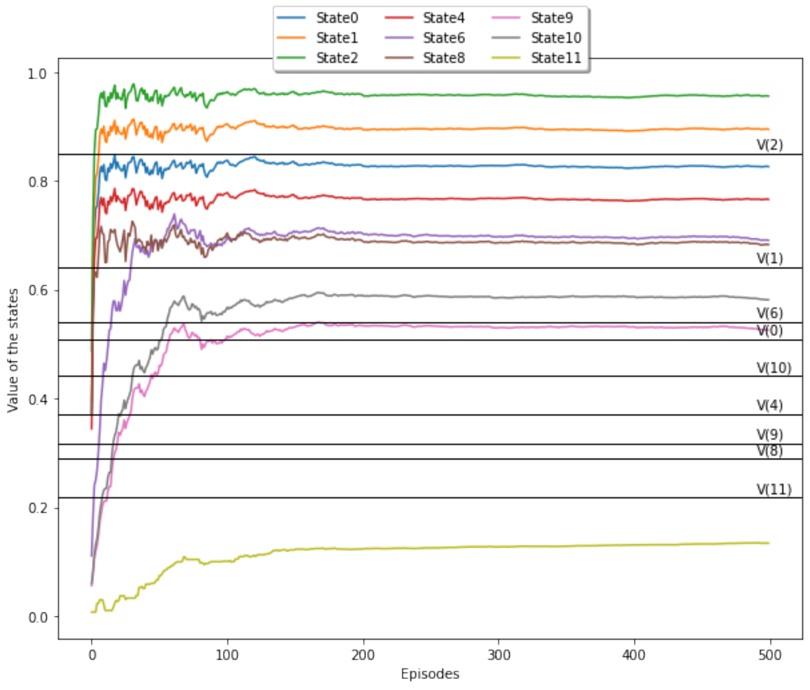


Figure 10: Averaged Rewards over the episodes (FVMC)

The algorithm seems to be overestimating the true values of the states. Values are also being under-approximated for state 11.

9. A similar trend is seen in the EVMC algorithm.
10. The TD algorithm seems to be doing slightly better but most states are still being over-approximated.
11. The TD λ algorithm seems to be performing nearly identical to the TD algorithm with slightly lower approximations of states.

    Out of the above algorithms, the TD lambda seems to be performing the best.
12. Looking at the episodes over log scale shows us that the values quickly jump above the true values in the first 100 steps.
13. The EVMC seems to be performing identical to the FVMC algorithm and the variance is reduced by averaging over the 50 states.
14. There is much lesser variance in the TD algorithm and a smoother ascent towards higher value approximations.
15. The TD lambda algorithm is performing nearly identically to the TD algorithm.
16. The Eligibility traces show how the values for the previous states change along with the values of the current states and a similar trends is seen for the different state values over time.
17. The target values for the FVMC are mostly positive due to the optimal nature of the policy. Since the values are discounted over time and there is a living cost, the targets are much below 1. Negative rewards are rarely obtained.

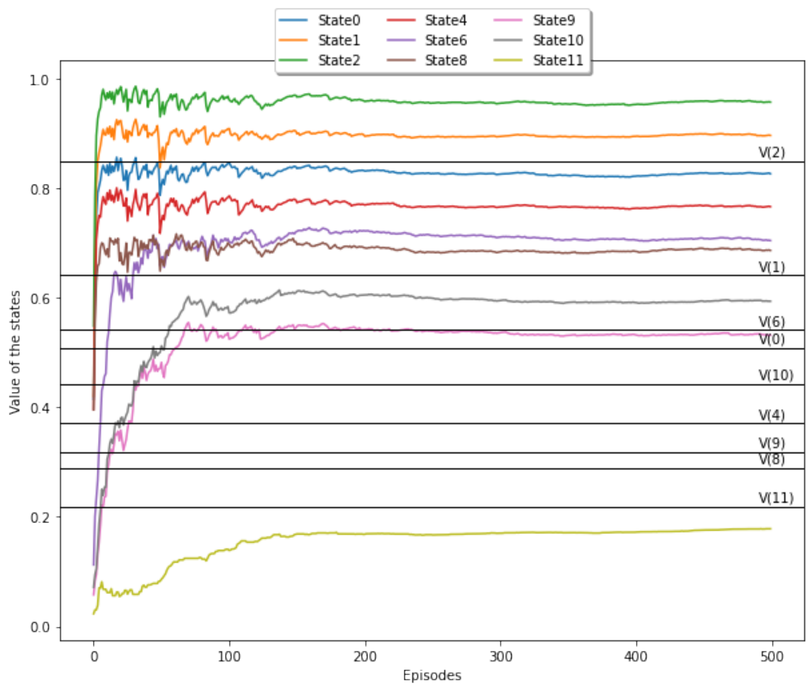


Figure 11: Averaged Rewards over the episodes (EVMC)

18. The EVMC is nearly identical to the FVMC.
19. The target rewards are vary much lesser and there seems to be a convergence but the values are over-approximated. However, the TD values seem to be hovering around the true values.
20. The TD lambda target values are also seeming to converge to an over approximation.

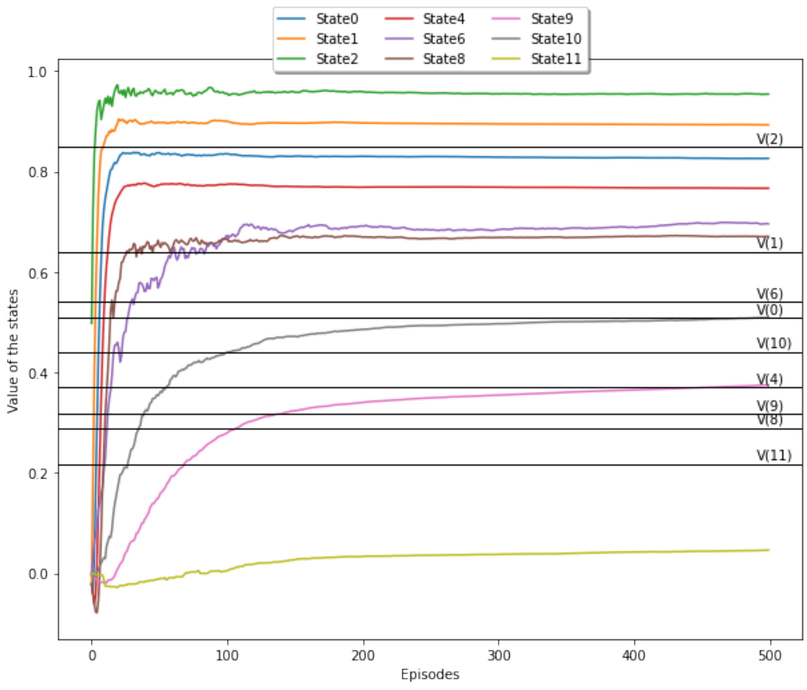


Figure 12: Averaged Rewards over the episodes (TD)

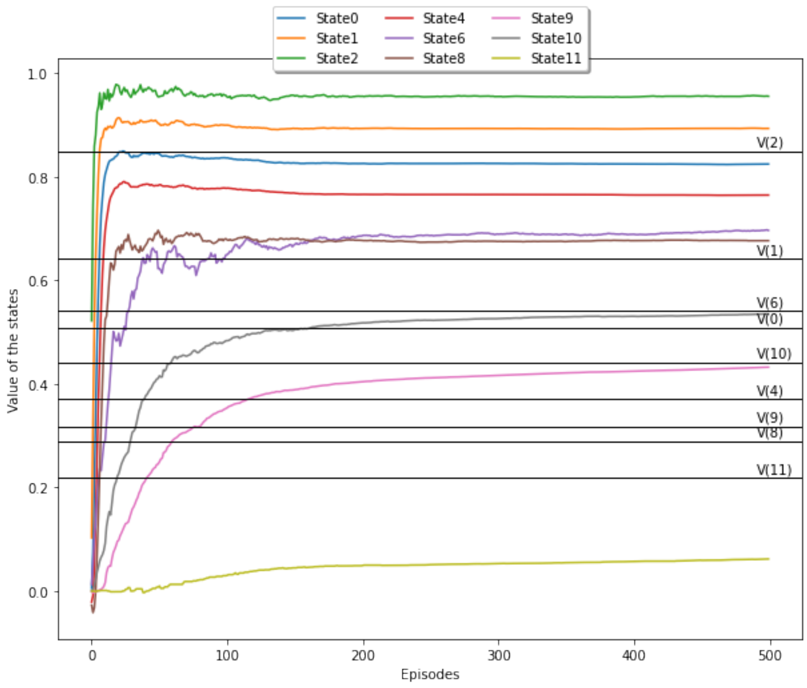


Figure 13: Averaged Rewards over the episodes (TD lambda)

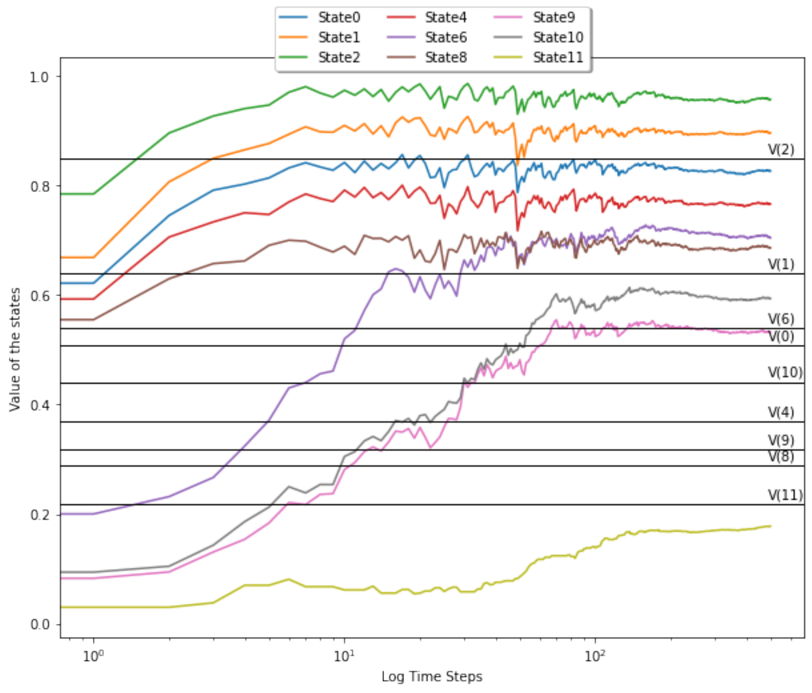


Figure 14: Averaged Rewards over log episodes (FVMC)

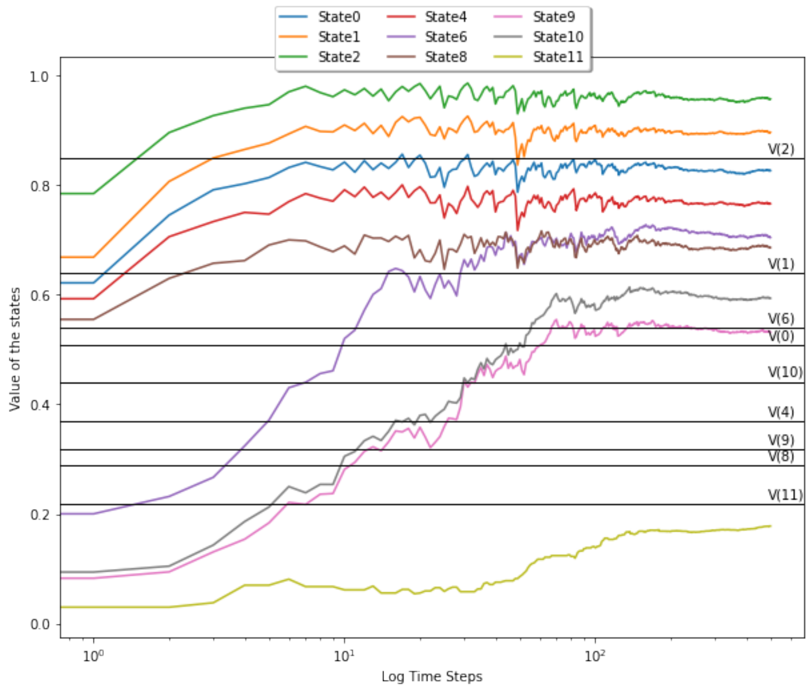


Figure 15: Averaged Rewards over log episodes (EVMC)

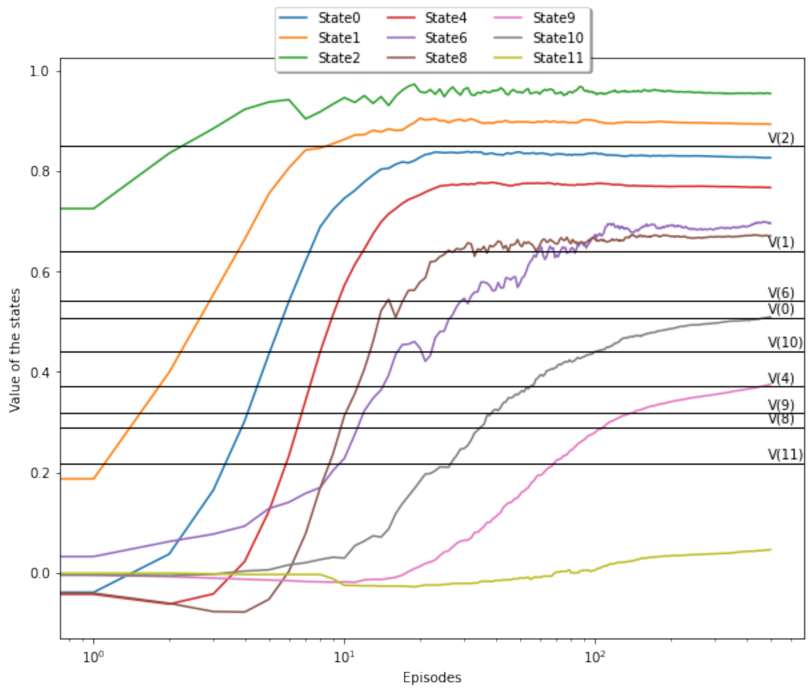


Figure 16: Averaged Rewards over log episodes (TD)

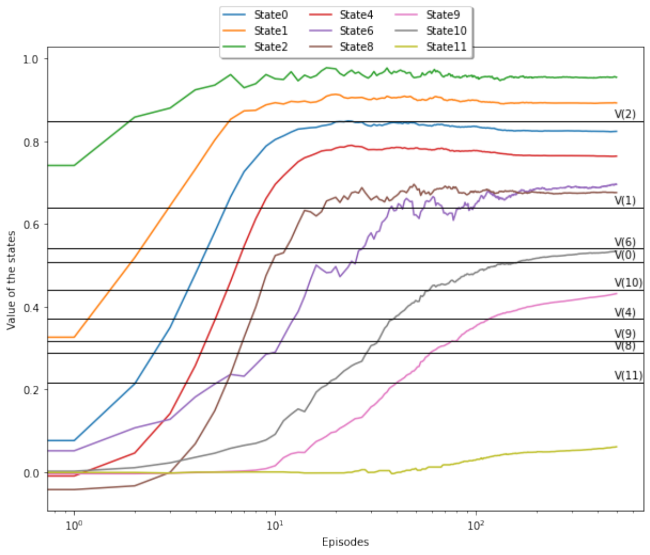


Figure 17: Averaged Rewards over log episodes (TD lambda)

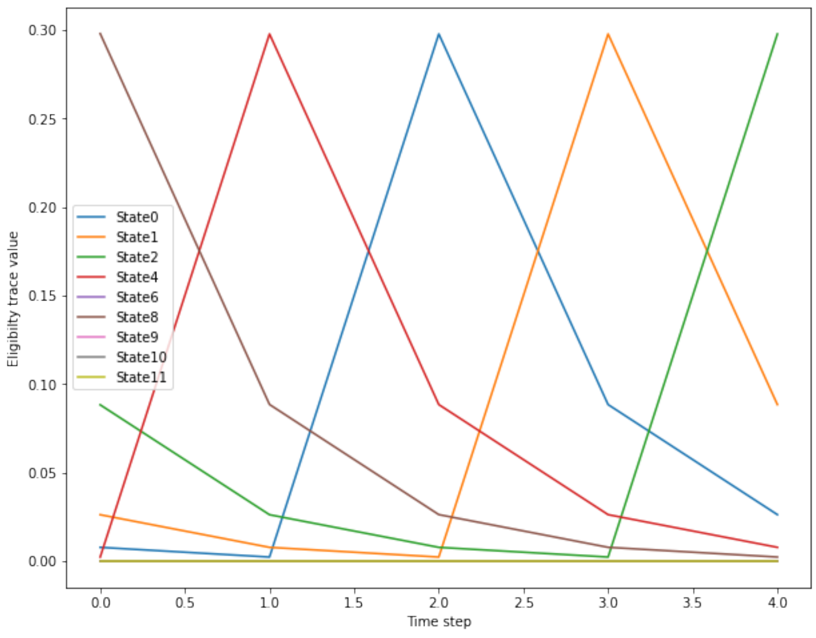


Figure 18: Eligibility trace for episode 100

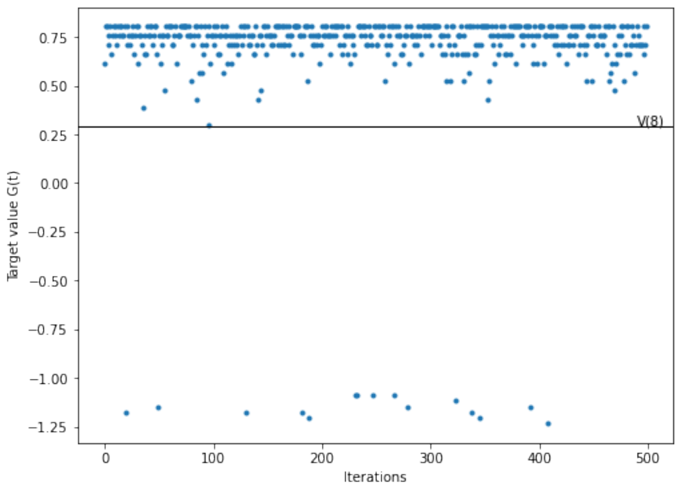


Figure 19: Target rewards obtained over time (FVMC)

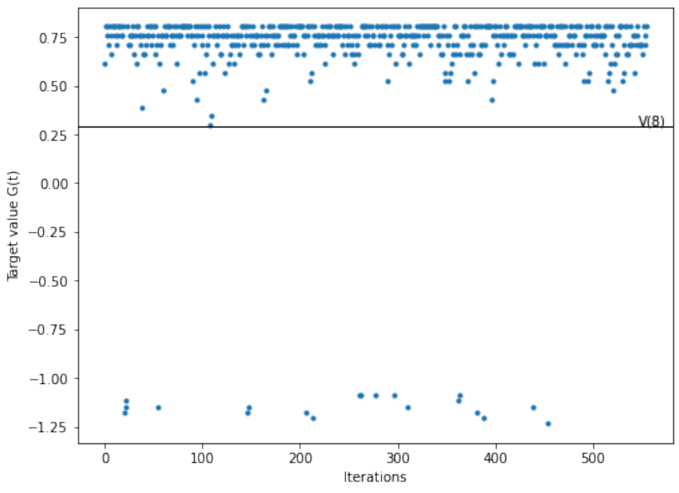


Figure 20: Target rewards obtained over time(EVMC)

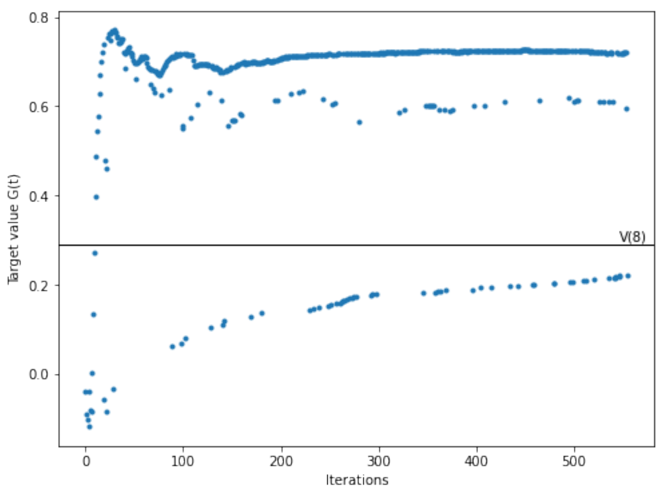


Figure 21: Target rewards obtained over time(TD)

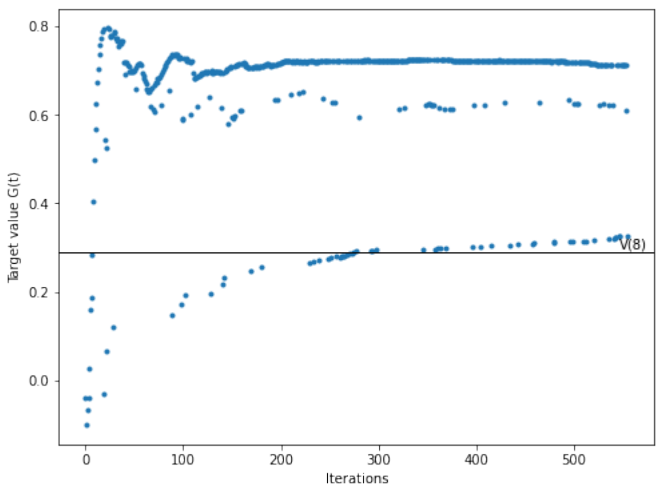


Figure 22: Target rewards obtained over time(TD lambda)