CS698R: Deep Reinforcement Learning

Mid-Semester Exam

Name: Naman Gupta
Roll NO.: 190527

Solution to Problem 1: Random-Maze Environment Implementation

Pseudo random number generators and Seeding: Throughout this report, manual seeds in numpy for random number generators are set by passing rng objects (np.random.default_rng(seed)). This is the recommended way of setting manual seeds (NEP 19) since it is easier to track which seed is used where.

I have implemented the class `RandomMazeEnvironment` using OpenAI Gym. The environment has four helper functions – `up`, `right`, `down`, `left` which correspond to the respective actions. These also handle the maze boundaries and the wall where the agent jumps back to the original state it was in.

For testing the environment implementation it is sufficient to test the transitions, that is, verify the new state and the received reward based on the original state and action. For this, I have created a function `test_rme_transitions(state, action)` which takes the current state and action as input and prints after calculating, all possible next states with their corresponding transition probabilites. Similarly, it also prints all possible rewards along with their associated probabilites. Below are some of the test cases which I used to test the implementation. Same random seed 1 is used for initalizing the environment in all the test cases.
Notation – s is the original state, s' is the new state after transition, a is the action taken and r is the reward obtained. Actions $(0, 1, 2, 3)$ correspond to (up, right, down, left) respectively.

1. $(s = 0, a = 0)$
   On moving up the agent would bounce back to state 0 since it encounters the maze boundary. This happens with probability 0.8. The agent moves left with probability 0.1 but again encounters the maze boundary and jumps back to state 0. The agent moves to state 1 with probability 0.1. Thus, $p(s' = 0) = 0.9$ and $p(s' = 1) = 0.1$. In any above transition, the reward obtained is -0.04. Hence $p(r = -0.04) = 1$.
   The results obtained by testing the environment are –

   $$\begin{aligned} s' = 0 \quad & p = 0.892 \\ s' = 1 \quad & p = 0.108 \\ r = -0.04 \quad & p = 1.0 \end{aligned}$$

   The probabilites 0.892 and 0.108 are very close to the actual probability, it shows that the transitions for $(s = 0, a = 0)$ are correct.

2. $(s = 2, a = 0)$
   On moving up the agent would bounce back to state 2 since it encounters the maze boundary. This happens with probability 0.8. The agent moves left to state 1 with probability 0.1 and moves right to state 3 with probability 0.1. Thus, $p(s' = 2) = 0.8$, $p(s' = 1) = 0.1$ and $p(s' = 3) = 0.1$. For transitions leading to s' as 1 or 2, the reward obtained is -0.04. Hence $p(r = -0.04) = 0.9$. For transitions to $s' = 3$ which is the goal state, a $+1$ reward is obtained. Hence, $p(r = 1) = 0.1$
   The results obtained by testing the environment are –

   $$\begin{aligned} s' = 1 \quad & p = 0.108 \\ s' = 2 \quad & p = 0.784 \\ s' = 3 \quad & p = 0.108 \\ r = 1 \quad & p = 0.108 \\ r = -0.04 \quad & p = 0.892 \end{aligned}$$

   All the measured probabilites are very close to the actual probabilities. Further, $p(r = 1) = p(s' = 3)$ and $p(r = -0.04) = p(s' = 1) + p(s' = 2)$ which is expected. This shows that the transitions for $(s = 2, a = 0)$ are correct.

3. $(s = 11, a = 1)$
   On moving right the agent would bounce back to state 11 since it encounters the maze boundary. This happens with probability 0.8. The agent moves down with probability 0.1 but again encounters the maze boundary and jumps back to state 11. The agent moves up to state 7 with probability 0.1. Thus, $p(s' = 11) = 0.9$ and $p(s' = 7) = 0.1$. For transitions leading to $s' = 11$, the reward obtained is -0.04. Hence $p(r = -0.04) = 0.9$. For transitions to $s' = 7$ which is the hole state, a -1 reward is obtained. Hence, $p(r = -1) = 0.1$
   The results obtained by testing the environment are –

$$\begin{aligned} s' = 11 \quad p = 0.892 \\ s' = 7 \quad p = 0.108 \\ r = -1 \quad p = 0.108 \\ r = -0.04 \quad p = 0.892 \end{aligned}$$

   All the measured probabilites are very close to the actual probabilities. Further, $p(r = -1) = p(s' = 7)$ and $p(r = -0.04) = p(s' = 11)$ which is expected. This shows that the transitions for $(s = 11, a = 1)$ are correct.

4. $(s = 4, a = 2)$
   On moving left the agent would bounce back to state 4 since it encounters the maze boundary. This happens with probability 0.1. The agent moves right with probability 0.1 but again encounters the wall and jumps back to state 4. The agent moves down to state 8 with probability 0.1. Thus, $p(s' = 4) = 0.2$ and $p(s' = 8) = 0.8$. In any above transition, the reward obtained is -0.04. Hence $p(r = -0.04) = 1$.
   The results obtained by testing the environment are –

$$\begin{aligned} s' = 4 \quad p = 0.216 \\ s' = 8 \quad p = 0.784 \\ r = -0.04 \quad p = 1.0 \end{aligned}$$

   All the measured probabilites are very close to the actual probabilities. This shows that the transitions for $(s = 4, a = 2)$ are correct.

5. $(s = 6, a = 3)$
   On moving left the agent would bounce back to state 6 since it encounters the wall. This happens with probability 0.8. The agent moves up to state 2 with probability 0.1 and moves down to state 10 with probability 0.1. Thus, $p(s' = 6) = 0.8$, $p(s' = 2) = 0.1$ and $p(s' = 10) = 0.1$. n any above transition, the reward obtained is -0.04. Hence $p(r = -0.04) = 1$.
   The results obtained by testing the environment are –

$$\begin{aligned} s' = 6 \quad p = 0.784 \\ s' = 2 \quad p = 0.108 \\ s' = 10 \quad p = 0.108 \\ r = -0.04 \quad p = 1.0 \end{aligned}$$

   All the measured probabilites are very close to the actual probabilities. This shows that the transitions for $(s = 6, a = 3)$ are correct.

Solution to Problem 2: RME Optimal Policy via Dynamic Programming

1. I have randomly chosen an adversarial initial policy such as to make the agent move away from the goal state and towards the hole state. The initial policy is shown in Figure 1 below. The first row consists of all left actions such as to move away from the goal state. Similarly, the states 8, 9, 10 in the last row are assigned action left by the policy. The state 11 is assigned action up to get more negative reward.

←	←	←	G
←	X	→	H
←	←	←	↑

Figure 1: Randomly chosen initial policy

The final obtained policy after PolicyIteration is shown in Figure 2 below. This policy seems intuitive in the sense that the agent tries to go towards the goal state, simultaneously avoid the hole state. It took 4 iterations of the PolicyIteration algorithm for it to converge to this policy. These four iterations in turn took 1972, 175, 38 and 38 iterations of the PolicyEvaluation algorithm to estimate the value function.

→	→	→	G
↑	X	↑	H
↑	←	↑	←

Figure 2: Optimal policy after PolicyIteration

2. I have randomly chosen an adversarial initial policy (same as part 1) such as to make the agent move away from the goal state and towards the hole state. The initial policy is shown in Figure 3 below. The first row consists of all left actions such as to move away from the goal state. Similarly, the states 8, 9, 10 in the last row are assigned action left by the policy. The state 11 is assigned action up to get more negative reward.
ValueIteration does not use an initial policy, so it does not matter what policy we start with.

←	←	←	G
←	X	→	H
←	←	←	↑

Figure 3: Randomly chosen initial policy

The final obtained policy after ValueIteration is shown in Figure 4 below. This policy seems intuitive in the sense that the agent tries to go towards the goal state, simultaneously avoid the hole state. It took 38 iterations of the ValueIteration algorithm for it to converge to this policy.

→	→	→	G
↑	X	↑	H
↑	←	↑	←

Figure 4: Optimal policy after ValueIteration

3. Each iteration of PolicyIteration consists of multiple iterations of PolicyEvaluation. Each iteration of PolicyEvaluation is similar in time complexity as compared to an iteration of ValueIteration. Hence, clearly ValueIteration converges way faster than PolicyIteration. ValueIteration takes a total of 38 iterations whereas PolicyIteration takes a total of (1972+175+38+38) = 2223 iterations (includes PolicyEvaluation).
   Further, the optimal policies obtained by PolicyIteration and ValueIteration are same.

   In general, both PolicyIteration and ValueIteration are guaranteed to converge at the same optimal policy. Consider PolicyIteration, in each iteration the policy either improves or remains same (in which case the algorithm ends). In other words it can be said that the same policy is not encountered by the algorithm twice. Since there are finite number of deterministic policies, the algorithm is guaranteed to converge to an optimal policy in finite number of iterations.
   Consider ValueIteration, in each iteration the value function gets closer to the true value function since the updated value function is obtained by selecting the best action from the action value function. Hence, ValueIteration is guaranteed to converge to the true value function (and the true action value function). Once it converges to the true action value function, the policy obtained after greedy selection is guaranteed to be optimal.

Solution to Problem 3: RME Prediction with MDP Unknown

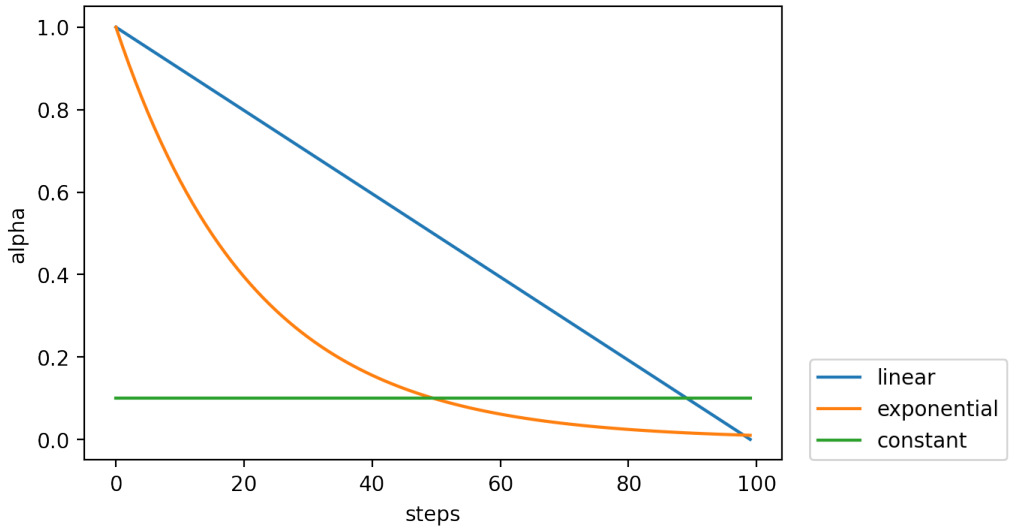
The optimal policy obtained from the last problem is used in this problem.

1. I have tested the `generateTrajectory` function using two initializations of the environment using different random seeds (1 and 2) with `maxSteps=20`. It is expected that the trajectory will be along the path given from the policy with few orthogonal deviations in between due to the environment being stochastic. The trajectories are given below –

`seed=1`	`seed=2`
(8, 0, -0.04, 4)	(8, 0, -0.04, 4)
(4, 0, -0.04, 4)	(4, 0, -0.04, 0)
(4, 0, -0.04, 0)	(0, 1, -0.04, 4)
(0, 1, -0.04, 0)	(4, 0, -0.04, 0)
(0, 1, -0.04, 1)	(0, 1, -0.04, 1)
(1, 1, -0.04, 2)	(1, 1, -0.04, 2)
(2, 1, -0.04, 6)	(2, 1, 1, 3)
(6, 0, -0.04, 2)	
(2, 1, 1, 3)	

   To test the case where partial trajectory is discarded I tested with `maxSteps=5`, on which the function correctly returned an empty list.

2. The `decayAlpha` function decays α linearly/exponentially. I tested the function by plotting the list of α's returned by the function. The plot is shown below. It can be clearly seen that `decayType=exponential` decays α more than `decayType=linear`. By observing the plots, it can also be verified that the implementation is correct. A constant value $\alpha = 0.1$ is also shown in the same plot for reference.



3. I tested the `MonteCarloPrediction` algorithm by running it for 10 episodes (`noEpisodes=10` and `maxSteps=10` for trajectory) and observing the value function as the episodes grow. I used a constant $\alpha = 0.1$ and $\gamma = 0.99$. The observed value function for 10 episodes is given below for FVMC (for terminal states 3,5 and 7 the value is always zero).

(0.075, 0.085, 0.090, 0, 0.066,0, 0.095, 0, 0.061, 0, 0, 0)
(0.158, 0.172, 0.181, 0, 0.145, 0, 0.095, 0, 0.136, 0, 0, 0)
(0.232, 0.249, 0.263, 0, 0.215, 0, 0.095 0, 0.202, 0, 0, 0)
(0.289, 0.310, 0.327, 0, 0.269, 0, 0.095, 0, 0.253, 0, 0, 0)
(0.345,0.368, 0.394, 0, 0.322, 0. 0.095 0, 0.303,0, 0, 0)
(0.396, 0.421, 0.454, 0, 0.370, 0. 0.095 0, 0.348, 0, 0, 0)
(0.441,0.469, 0.509, 0, 0.413, 0. 0.095 0, 0.3888, 0, 0, 0)

(0.482, 0.512, 0.558, 0, 0.452, 0. 0.095 0, 0.416, 0.070, 0, 0)
(0.509,0.541, 0.602, 0, 0.473, 0. 0.095 0, 0.435, 0.070, 0, 0)
(0.544, 0.577,0.642, 0, 0.506, 0. 0.095 0, 0.467, 0.070, 0, 0)

We notice that initially the values for states are very low, as the episodes increase the values of these states also increase. The state values are propagated from state 2 to rest of the states, since it has the maximum value among all states. We also observe that the value of states 10 and 11 remains 0 throughout even though they are not terminal states. This is because there is no way to reach states 10 and 11 starting from state 8 owing to the policy. This further verifies that the implementation is correct, also highlights this shortcoming of `MonteCarloPrediction`.
EVMC setting can be analyzed in a similar way, there a faster propagation is expected.

4. I tested `TemporalDifferencePrediction` by running it for 10 episodes and observing the value function as the episodes increase. It can be noticed that in TD it takes more number of episodes for the value of state 2 to propagate to other states. This is because, in TD the value function is updated after each transition and the complete return is not calculated. On the other hand, in MC the complete return is calculated, the value of a state is updated after that. The obtained value function vs episodes is given in the notebook. (It was too timetaking to format into Latex).

5. I tested nStepTD for (`n=1`, `noEpisodes=10`) and (`n=20`, `noEpisodes=10`). For `n=1` it should give exactly the same output as TD, this is what is obtained. For `n=20`, we expect that the value for state 8 would only be updated while others would be very close to 0. This is because the agent takes 20 experience steps to update the state. Indeed, this is what we observe. The obtained value function vs episodes is given in the notebook.

6. I tested TDLambda by taking $\lambda = 1$ and $\lambda = 0$ with `nEpisodes=10`. TD(0) should be exactly same as TD, whereas TD(1) should be similar to MC (it would not be exactly same for the current implementation is of the backward view of TDLambda, the forward view would exactly be the same). The outputs obtained for both settings are given in the notebook. We observe that TD(0) is exactly same as TD. We also observe that the propagation of values of states takes place in a way similar to MC in TD(1). On testing for $\lambda = 0.5$ it can be observed that the value function in this case is sort of in between that of TD and MC, though highly skewed towards TD due to exponential factors in lambda.

7. Using the Bellman Equation for value function

$$v_\pi(s) = \sum_a \pi(a \mid s) \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma v_\pi\left(s'\right)\right]$$

$$\begin{aligned}
v(0) &= 0.1(-0.04 + \gamma v(0)) + 0.1(-0.04 + \gamma v(4)) + 0.8(-0.04 + \gamma v(1)) \\
v(1) &= 0.2(-0.04 + \gamma v(1)) + 0.8(-0.04 + \gamma v(2)) \\
v(2) &= 0.1(-0.04 + \gamma v(2)) + 0.1(-0.04 + \gamma v(6)) + 0.8(1 + \gamma v(3)) \\
v(3) &= 0 \\
v(4) &= 0.8(-0.04 + \gamma v(0)) + 0.2(-0.04 + \gamma v(4)) \\
v(5) &= 0 \\
v(6) &= 0.1(-0.04 + \gamma v(6)) + 0.1(-1 + \gamma v(7)) + 0.8(-0.04 + \gamma v(2)) \\
v(7) &= 0 \\
v(8) &= 0.8(-0.04 + \gamma v(4)) + 0.1(-0.04 + \gamma v(8)) + 0.1(-0.04 + \gamma v(9)) \\
v(9) &= 0.2(-0.04 + \gamma v(9)) + 0.8(-0.04 + \gamma v(8)) \\
v(10) &= 0.1(-0.04 + \gamma v(9)) + 0.1(-0.04 + \gamma v(11)) + 0.8(-0.04 + \gamma v(6)) \\
v(11) &= 0.1(-1 + \gamma v(7)) + 0.1(-0.04 + \gamma v(11)) + 0.8(-0.04 + \gamma v(10))
\end{aligned}$$

8. We observe that exponentially decaying α 0.5 to 0.01 converges faster than constant α. As seen earlier value of state 10 and 11 remains 0. During the initial episodes there is a high variance in the value of the states even after averaging out 100 environments. The variance is lower for constant α initially but higher in the later episodes. This is due to alpha being small comparatively initially but larger later.

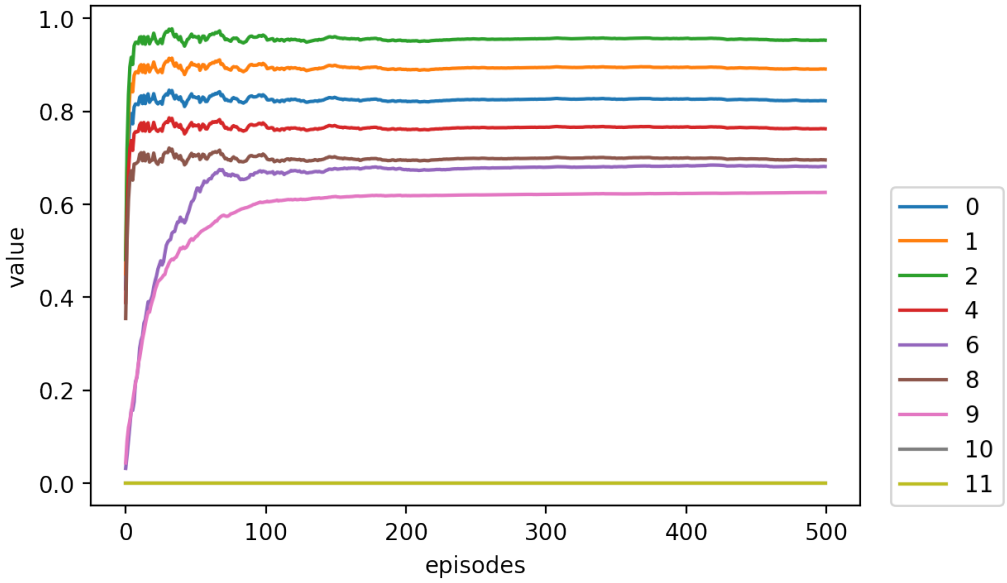


Figure 5: FVMC exponentially decaying α 0.5 to 0.01

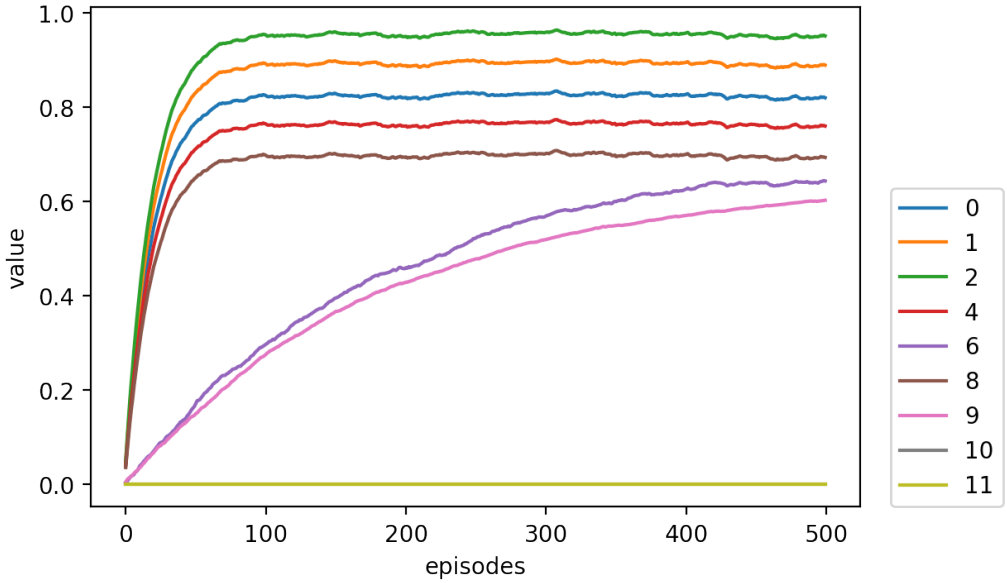


Figure 6: FVMC constant α 0.05

9. We observe that exponentially decaying α 0.5 to 0.01 converges faster than constant α. As seen earlier value of state 10 and 11 remains 0. During the initial episodes there is a high variance in the value of the states even after averaging out 100 environments. The variance is lower for constant α initially but higher in the later episodes. This is due to alpha being small comparatively initially but larger later. EVMC is similar to FVMC.

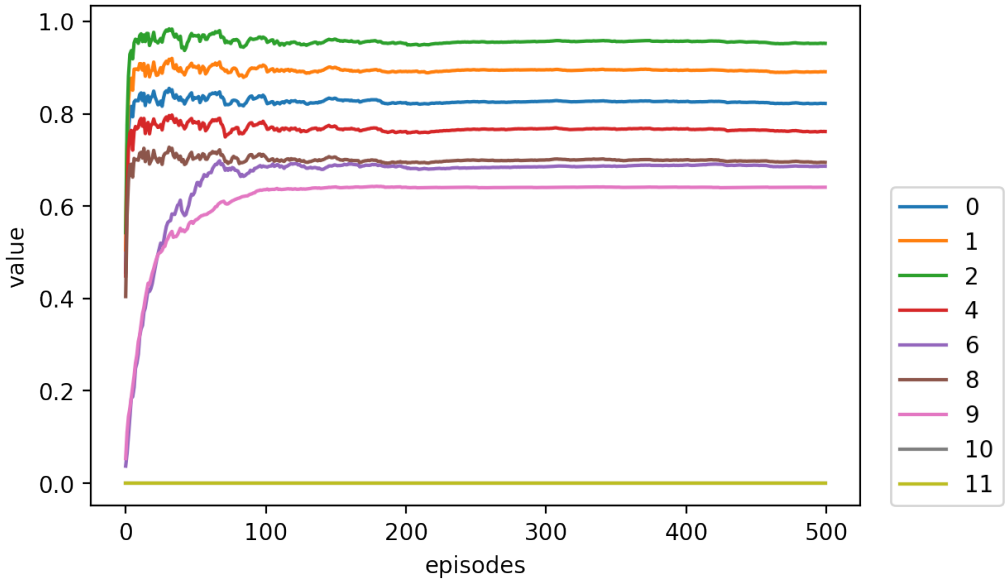


Figure 7: EVMC exponentially decaying α 0.5 to 0.01

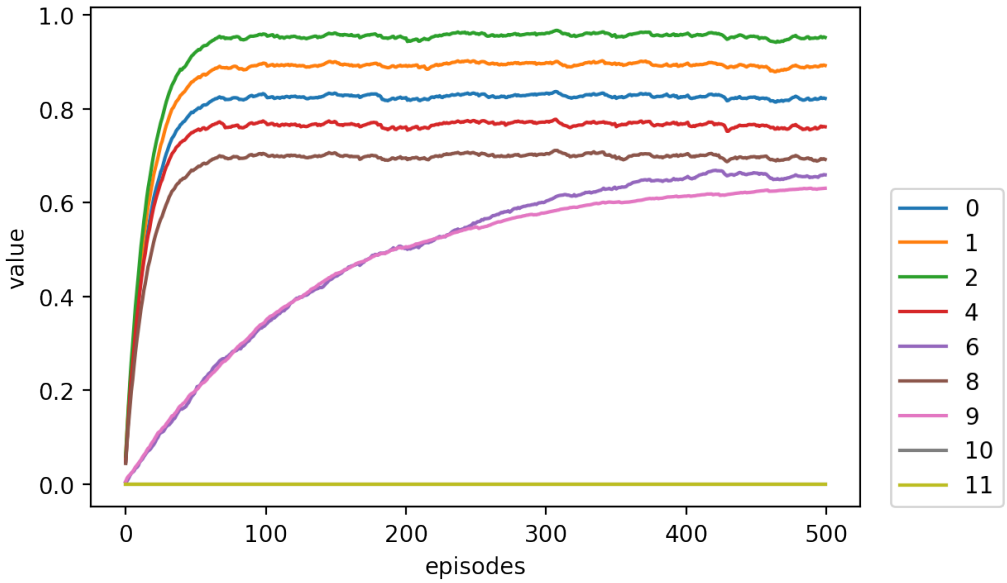


Figure 8: EVMC constant α 0.05

10. TD has lower variance compared to MC, almost similar convergence rate.

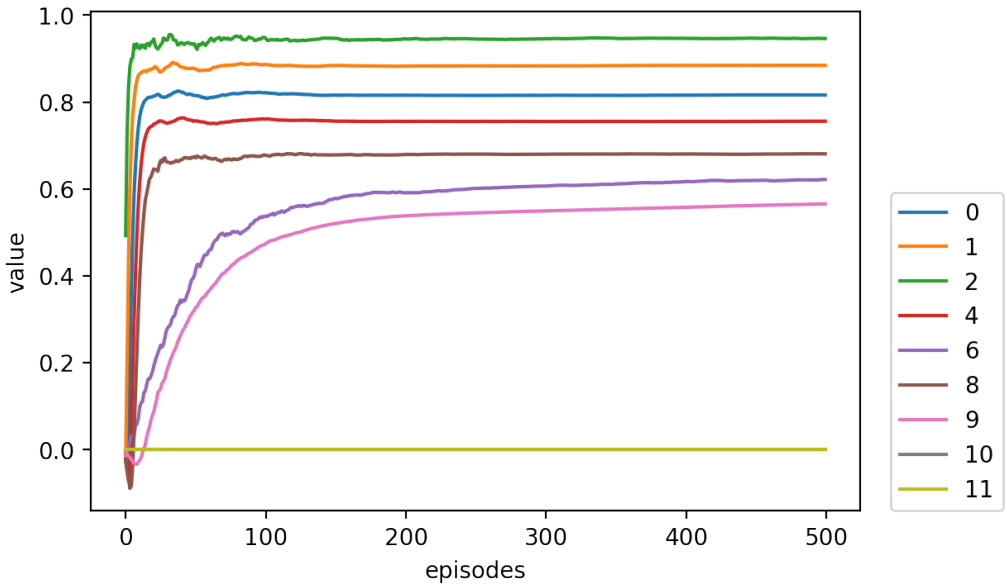


Figure 9: TD exponentially decaying α 0.5 to 0.01

11. nStepTD converges slower than TD, states 8, 9 do not achieve true value. This is because they are skipped when $n = 3$ is taken.

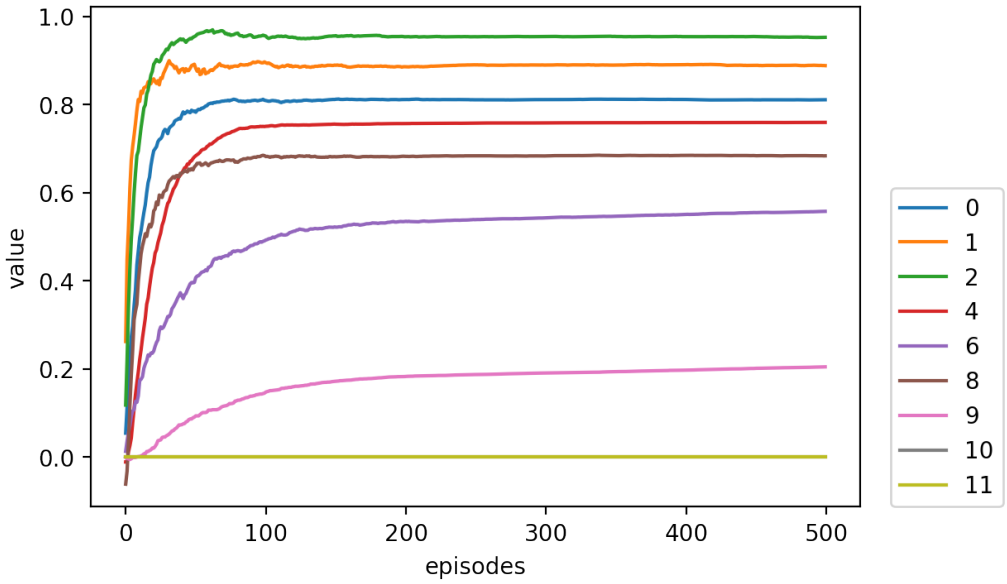


Figure 10: nStepTD exponentially decaying α 0.5 to 0.01

12. TDLambda converges faster than nStepTD and similar to TD. All the states are developed here.

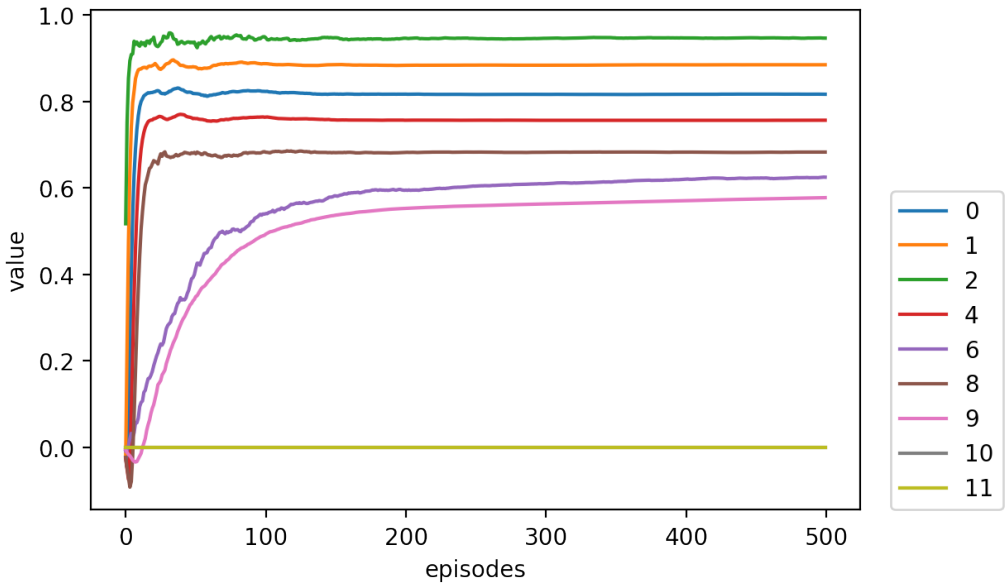


Figure 11: TDLambda exponentially decaying α 0.5 to 0.01

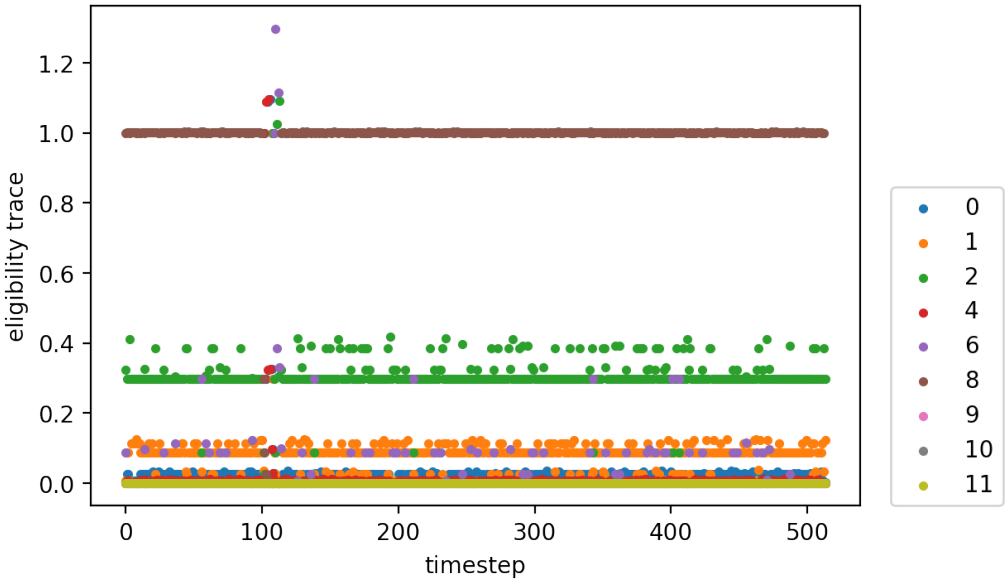


Figure 12: TDLambda Eligibility Trace

13.

14. The high variance is more apparent here.

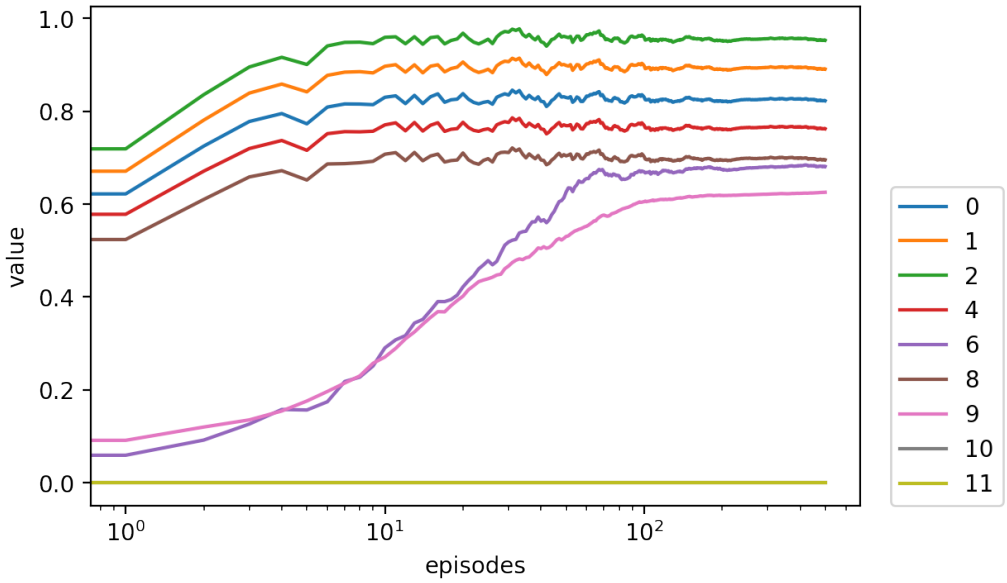


Figure 13: FVMC exponentially decaying α 0.5 to 0.01 logscale

15. The high variance is more apparent here.

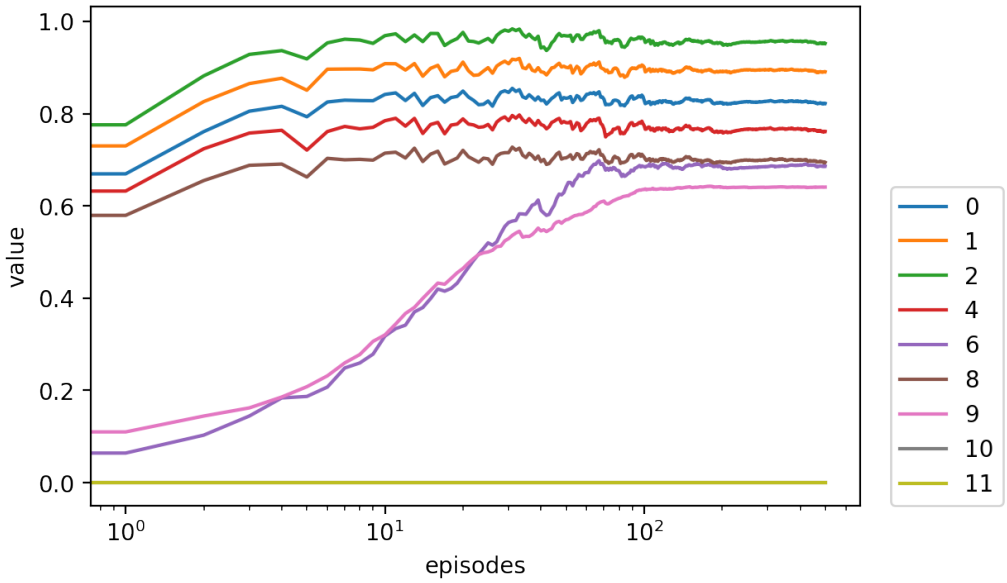


Figure 14: EVMC exponentially decaying α 0.5 to 0.01 logscale

16. Even in logscale the plot is smooth indicating low variance.

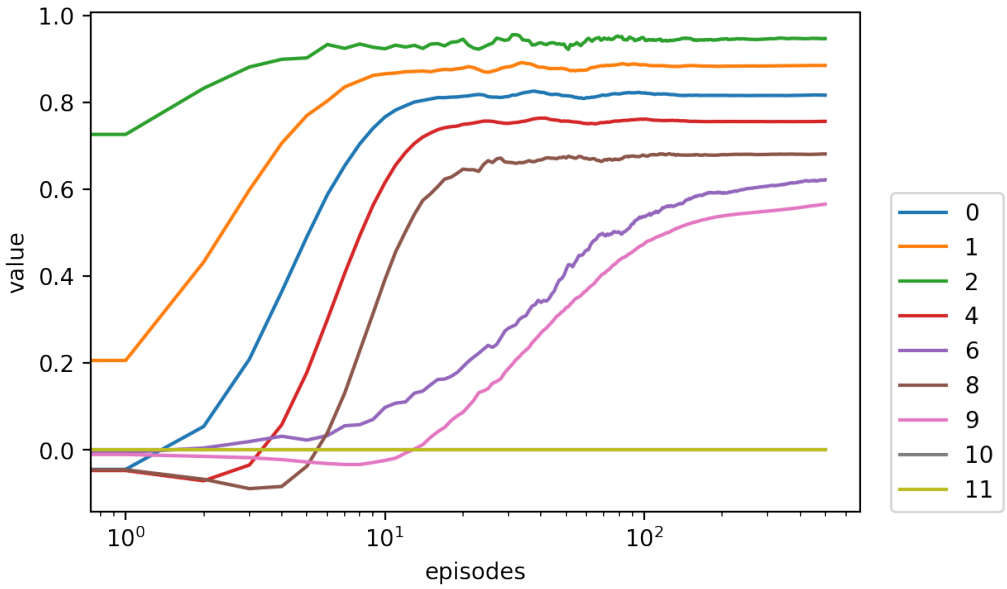


Figure 15: TD exponentially decaying α 0.5 to 0.01 logscale

17. Variance slightly higher than TD. The states 9, 10 are not developed since they are skipped on taking $n = 3$.

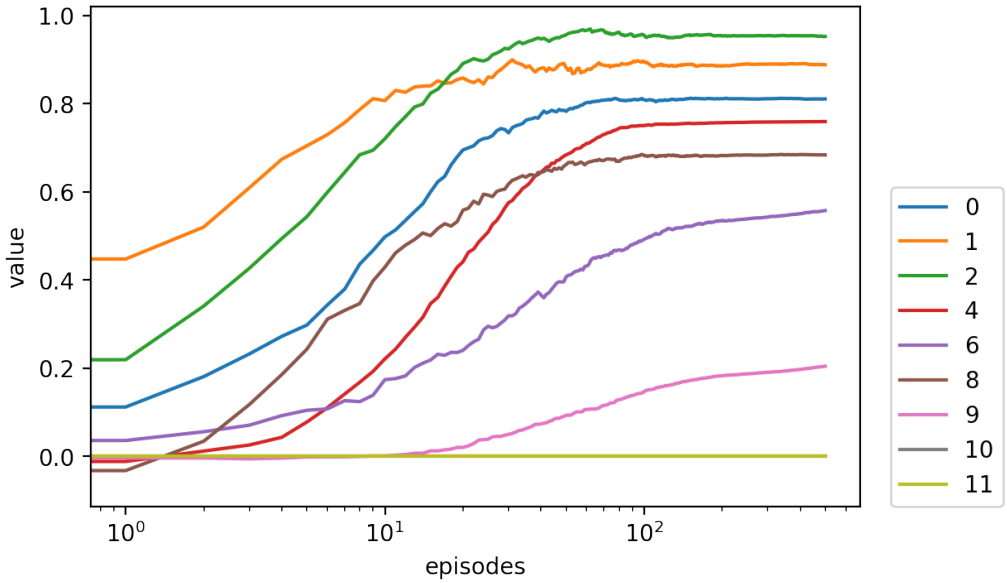


Figure 16: nStepTD exponentially decaying α 0.5 to 0.01 logscale

18. Variance lower than nStepTD but similar to TD.

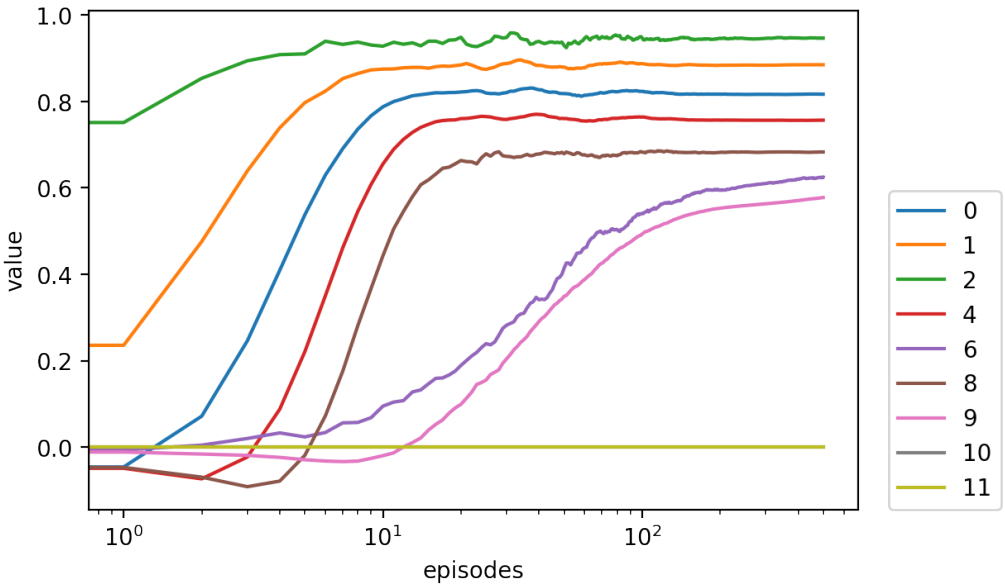


Figure 17: TDLambda exponentially decaying α 0.5 to 0.01 logscale