CS780: Deep Reinforcement Learning

Assignment #1

Name: Divyaksh Shukla
Roll NO.: 231110603

Complete code with all tex files and sources images available at: https://github.com/divyaksh-shukla/cs780-homeworks

Solution to Problem 1: Multi-armed Bandits

Code available at: http://tinyurl.com/cs780-assignment-1-p1

1. Created the environment using Gymnasium library. The environment is a 2-armed bernoulli bandit with some stochasticity. The environment characteristics are taken from a config file which specify the α and β values. I tested out with different values of α and β $[(0,0),(0,1),(1,0),(1,1),(0.5,0.5)]$ and the environment is working as expected. The agent recieves a reward only if it takes an action which takes it correctly in the direction of movement. That is if the the agent moves 'left' and lands in state 1 or moves 'right' and lands in state 2, only then it gets a positive reward.

2. I created a similar environment with 10-arms just like the above. But in this case the environment is not stochastic. The action and state-transitions are deterministic. The expected reward for each action $q_*(s,a)$ is sampled from a standard normal distribution ($\mathcal{N}(0,1)$). The agent then receives a reward from a normal distribution with mean $q_*(s,a)$ and variance 1. The agent has to then learn the optimal action to take in each state, by taking actions and observing the rewards.

3. I created 6 types of bandit agents following different strategies to solve the bandit problem. The agents are:

    (a) Greedy Agent: This agent always takes the action with the highest estimated value. It does not explore the environment.

    (b) Epsilon-Greedy Agent: This agent takes the action with the highest estimated value with probability $1-\epsilon$ and takes a random action with probability ϵ.

    (c) Decaying Epsilon-Greedy Agent: This agent is similar to the epsilon-greedy agent, but the value of ϵ decays "linearly" $\epsilon = max(0, \epsilon_0 - decay\_rate * episode)$ or "exponentially" $\epsilon = \epsilon_0 e^{-decay\_rate*episode}$ with time.

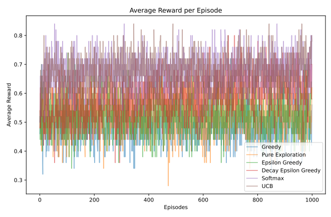
    (d) Softmax Agent: This agent takes actions with probability proportional to the exponential of the estimated value of the action. The agent explores the environment by taking actions by choosing from the below distribution

    $$\pi(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_b e^{Q(s,b)/\tau}} \tag{1}$$

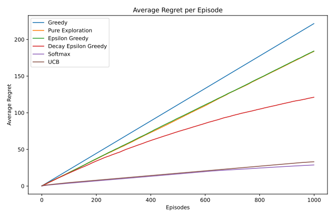
    (e) UCB Agent: This agent takes actions by choosing the action with the highest upper confidence bound. The upper confidence bound is calculated as $Q(s,a) + c\sqrt{\frac{\ln t}{N(s,a)}}$ where c is a constant and $N(s,a)$ is the number of times the action a has been taken in state s. The action is taken by taking the argmax of the upper confidence bound.

4. Created 50 different bandit problems for 2-armed Bernoulli Bandit with α and β values chosen from uniform distribution $\mathcal{U}(0,1)$. The agents were then tested on these bandit problems. The agents were tested for 1000 episodes and the average reward, average regret and optimal action percentage was calculated. The results are shown in the plots below (Figure 1a, 1b, 1c).

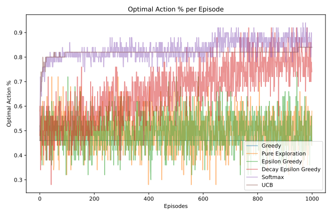
5. Created 50 different bandit problems for 10-armed Gaussain Bandit with $q_*(s,a)$ values chosen from standard normal distribution $\mathcal{N}(0,1)$ and then the agent receives a reward from $\mathcal{N}(q_*(s,a),1)\forall a \in A$. The agents were then tested on these bandit problems for 1000 episodes and the average reward, average regret and optimal action percentage was calculated. The results are shown in the plots below (Figure 2a, 2b, 2c).



(a) Average reward



(b) Average regret

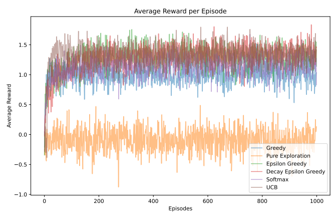


(c) Optimal Action %

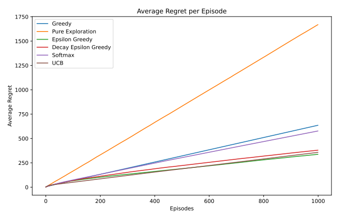
Figure 1: Results of 2-armed Bernoulli Bandit

6. Created a plot of the average regret of the agents for the 2-armed Bernoulli Bandit. Figure 1b shows the average regret of the agents for the 2-armed Bernoulli Bandit. From the plot we can see that Greedy agent has the highest regret and it is continuously increasing. The epsilon-greedy and pure exploration agents have similar regret values. The regret of epsilon-decay rises as it is exploring but flattens out once it is confident about its environment. While the UBC and Softmax agents do not have much regret.

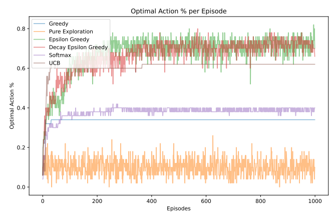
7. Created a plot of the average regret of the agents for the 10-armed Gaussian Bandit. Figure 2b shows the average regret of the agents for the 10-armed Gaussian Bandit. From the plot we can see that Pure exploration leads to random reward retrivals making the agent gather more regret. The epsilon-greedy, epsilon-decay and UCB agents have similar regret values. While the greedy and softmax agents are very similar in regret values. We also notice that the regret does not sigificantly flatten out for any agent. Thus, we may need to run more steps to see the regret flatten out.



(a) Average reward



(b) Average regret



(c) Optimal Action %

Figure 2: Results of 10-armed Gaussian Bandit

8. This is the same as question 4.

9. Plotting the optimal action percentage of each agent for the 2-armed Bernoulli Bandit. Running 50 experiments did not reduce the variance by a noticeable margin, making the plots look very noisy. The

results are shown in Figure 1c. From the plot we can see that the greedy agent has the lowest optimal action percentage. The epsilon-greedy and pure exploration agents have similar optimal action percentage, which means I have to change the eplison value even more. The UCB and Softmax agents have the highest optimal action percentage. While the epsilon-decay agent shows potential as its optimal action percentage increases with time. Running more expisodes could've proven the epsilon-decay agent to be the best.

10. Plotting the optimal action percentage of each agent for the 10-armed Gaussian Bandit. The plots here are not as noisy as the 2-armed Bernoulli Bandit. The results are shown in Figure 2c. From the plot we can see that the Pure exploration has the lowest optimal action percentage, which is inline with the regret plot of Pure exploration. This is followed by greedy and softmax agents. The optimal-action percentage plot shows that epsilon-greedy and epsilon-decay agents have the best action selection strategy and beat UCB agent by a noticeable margin, which is not evident from the regret plots.

Solution to Problem 2: MC Estimates and TD Learning

Code available at: `http://tinyurl.com/cs780-assignment-1-p2`

I started by implementing the `RandomWalk-v0` environment by subclassing the `gym.Env` class. The environment is a simple random walk with 7 states. The agent starts at the middle state and only moves left following the "go-left" policy. The agent terminates actions and receives a reward of +1 if it reaches the rightmost state and a reward of 0 if it reaches the leftmost state. The agent receives a reward of 0 on non-terminal states. The environment is completely stochastic and the agent has a probability of 0.5 of moving left and right. The environment is working as expected, based on pygame observations and trajectory anaylsis.

We take the action space to be $\mathcal{A} = \{0, 1\}$ and the state space to be $\mathcal{S} = \{0, 1, 2, 3, 4, 5, 6\}$. Thus, to calculate the state-values, we can use the following equations:

$$\begin{aligned} V(s) &= \mathbb{E}[G_t | S_t = s] = \mathbb{E}[R_{t+1} + \gamma G_{t+1} | S_t = s] && (2) \\ &= \sum_a \pi(a|s) \sum_{s', r} p(s', r | s, a)[r + \gamma V(s')] && (3) \end{aligned}$$

We can now fill in values for each of the variables and calculate the state-values. As there is only 1 action we choose for each state (following a predetermined policy $\pi =$"go left"), we can ignore the $\pi(a|s)$ term.

$$\begin{aligned} v(0) &= 0 \\ v(1) &= p(0,0|1,0)[0 + \gamma v(0)] + p(2,0|1,0)[0 + \gamma v(2)] \\ v(2) &= p(1,0|2,0)[0 + \gamma v(1)] + p(3,0|2,0)[0 + \gamma v(3)] \\ v(3) &= p(2,0|3,0)[0 + \gamma v(2)] + p(4,0|3,0)[0 + \gamma v(4)] \\ v(4) &= p(3,0|4,0)[0 + \gamma v(3)] + p(5,0|4,0)[0 + \gamma v(5)] \\ v(5) &= p(4,0|5,0)[0 + \gamma v(4)] + p(6,1|5,0)[1 + \gamma v(6)] \\ v(6) &= 0 \end{aligned}$$

Upon rearranging the terms we get:

$$\begin{aligned} v(0) &= 0 \\ v(1) - \frac{\gamma}{2} v(2) &= 0 \\ v(2) - \frac{\gamma}{2} v(1) - \frac{\gamma}{2} v(3) &= 0 \\ v(3) - \frac{\gamma}{2} v(2) - \frac{\gamma}{2} v(4) &= 0 \\ v(4) - \frac{\gamma}{2} v(3) - \frac{\gamma}{2} v(5) &= 0 \\ v(5) - \frac{\gamma}{2} v(4) &= \frac{1}{2} \\ v(6) &= 0 \end{aligned}$$

Rewriting the above equations in matrix form we get:

$$\begin{bmatrix} 1 & -\frac{\gamma}{2} & 0 & 0 & 0 \\ -\frac{\gamma}{2} & 1 & -\frac{\gamma}{2} & 0 & 0 \\ 0 & -\frac{\gamma}{2} & 1 & -\frac{\gamma}{2} & 0 \\ 0 & 0 & -\frac{\gamma}{2} & 1 & -\frac{\gamma}{2} \\ 0 & 0 & 0 & -\frac{\gamma}{2} & 1 \end{bmatrix} \begin{bmatrix} v(1) \\ v(2) \\ v(3) \\ v(4) \\ v(5) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \frac{1}{2} \end{bmatrix}$$

Solving the above for $\gamma = 0.99$ we get:

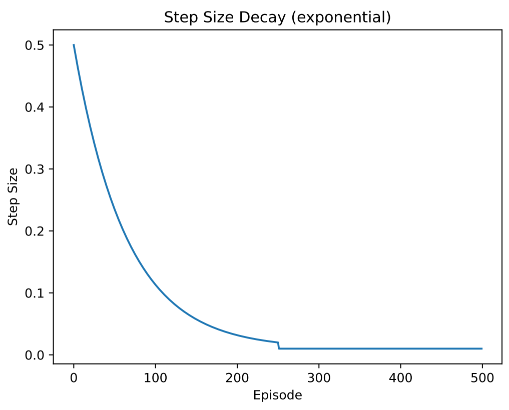
$$v(1) = 0.1501; v(2) = 0.3032; v(3) = 0.4624; v(4) = 0.6310; v(5) = 0.8124$$

In all the below test cases and scenarios a seed of 21 is used. I have also added `CONFIG` dictionary at any code block which is producing an output to be able to change values and observe outputs.

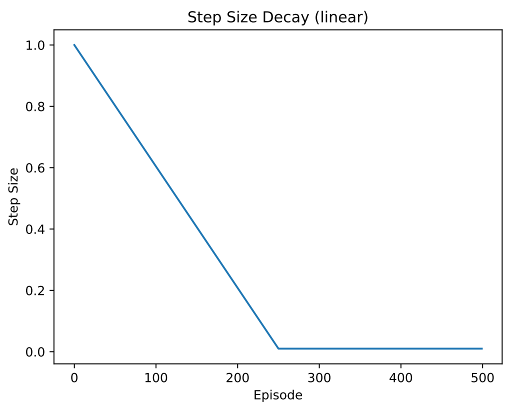
1. I created a function `generate_episode_trajectory(environment, config=None)` which takes in the `environment` and a `config` dictionary and returns a list of tuples of the form (s, a, r, s'). I tested the function with the `RandomWalk-v0` environment and the trajectory is as expected. The trajectory is shown in the code block below.

```
Episode Trajectory: [(3, 0, 0.0, 2), (2, 0, 0.0, 3), (3, 0, 0.0, 4),
(4, 0, 0.0, 3), (3, 0, 0.0, 2), (2, 0, 0.0, 3), (3, 0, 0.0, 4),
(4, 0, 0.0, 5), (5, 0, 1.0, 6)]
```

2. I wrote a routine to decay the step size at every step/episode: `decay_step_size(initial_value, final_value, episode, max_episode, decay_type, decay_stop=None)`. This routine returns $\alpha(e)$ which starts at `initial_value` and ends at `final_value` based on the `decay_type`. I tested the function and the results are present in Figure 3a and Figure 3b. The step size is decaying as expected.



(a) Exponential



(b) Linear

Figure 3: Exponential and Linear decay of step-size for 500 steps, initial value=0.5, final value=0.01 and decrease only till step 250 after which it remains constant

3. I implemented a routine to run Monte Carlo prediction for first-visit and every-visit scenarios. This is again controlled by `CONFIG` parameter. Below is an output from the routine for first-visit Monte Carlo prediction. The output consists of the state values, state-values for all episodes and the trajectory of the agent.

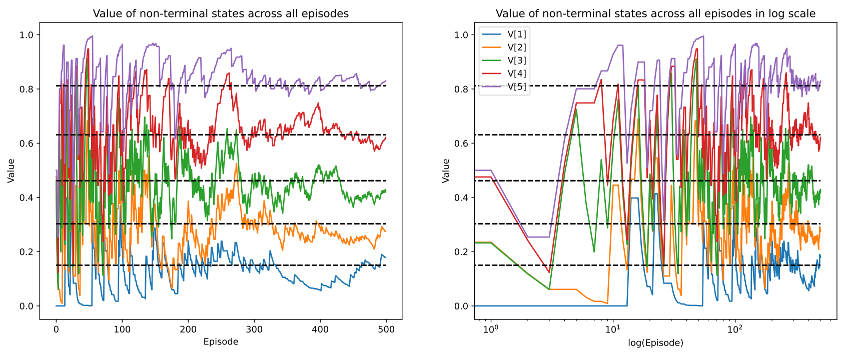
```
(array([0., 0., 0., 0., 0., 0., 0.]),
array([[0., 0., 0., 0., 0., 0., 0.]]),
[(3, 0, 0.0, 2), (2, 0, 0.0, 1), (1, 0, 0.0, 0)])
```

4. I implemented a routine to run Temporal difference prediction. This routine like the above not only does prediction but also computes the state-values. This is again controlled by `CONFIG` parameter. Below is an output from the routine for TD prediction routine. The output consists of the state values, state-values for all episodes and the trajectory of the agent.

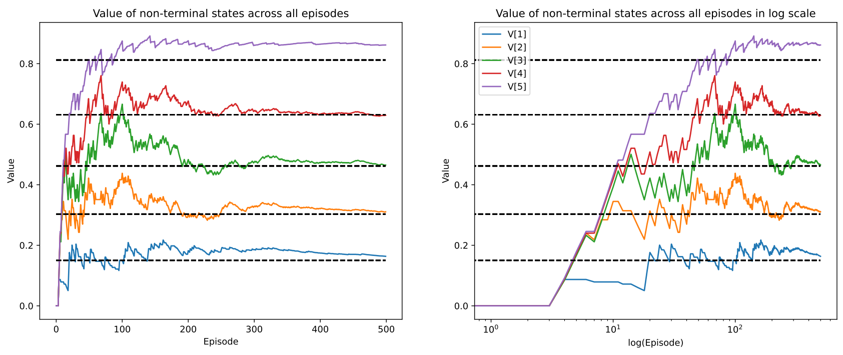
```
(array([0.  , 0.  , 0.  , 0.  , 0.  , 0.1, 0. ]),
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.1, 0. ]]),
[(3, 0, 0.0, 2), (2, 0, 0.0, 1), (1, 0, 0.0, 0)])
```

5. I wrote a routine to plot the state-values across episodes and log(episodes) to answer multiple questions below. `plot_episode_values(V_r, image_name, config=None)` takes the state-values across episodes along with an `image_name` to save the image and a `config` for additional information.

   I plotted the state-values for each state for First-Visit MC in Figure 4a and 4b. We can see from the from the figures that there is initially very high variance in the state-values but as the number of episodes increase the variance decreases and the state-values converge to the true values. The variance is higher in the default case as the step size is too high and the decay is too slow. The variance is lower in the tuned case as the step size is lower and the final value is also very low. On the log-scale plot, we can see that the state-values slowly rise towards the true values.



(a) Exponential decay of step size



(b) Exponential decay and tuned hyperparameters

Figure 4: First-Visit MC

6. I plotted the state-values for each state for Every-Visit MC in Figure 5 and 6. We can see from the from the figures that there is initially very high variance in the state-values but as the number of episodes increase the variance decreases and the state-values converge to the true values. The variance is higher in the default case as the step size is too high and the decay is too slow. I then tuned the hyperparameters and the variance is lower in the tuned case as the step size is lower and the final value is also very low, while achieving very low bias on the true values. On the log-scale plot, we can make a better judgement on the high variance phenomenon of Monte Carlo methods.

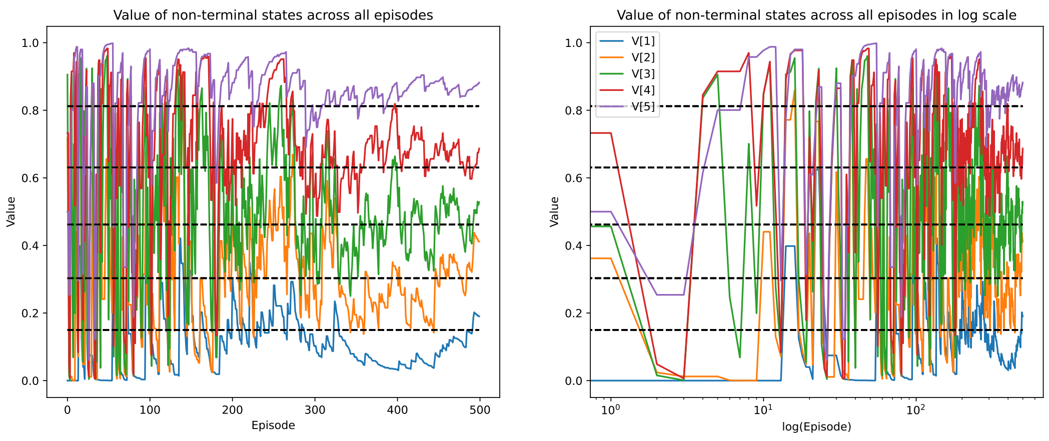


Figure 5: EVMC with exponential decay of step size

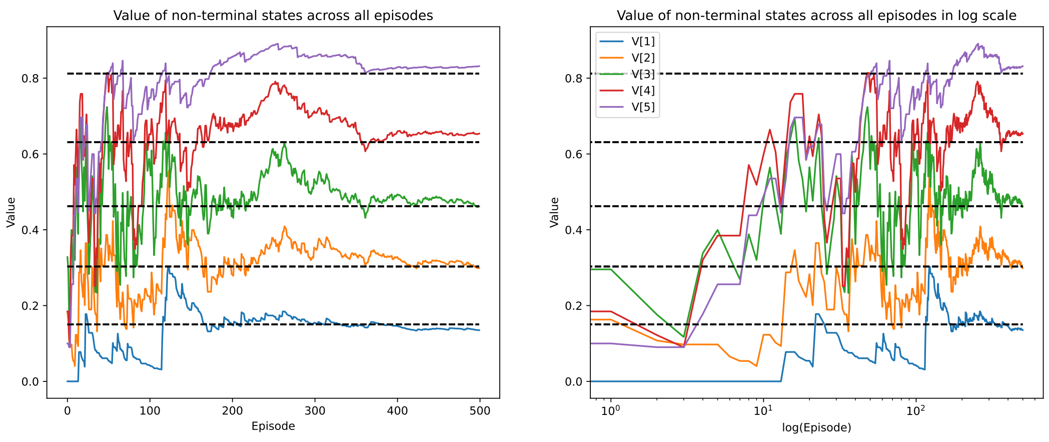


Figure 6: EVMC with exponential decay and tuned hyperparameters

7. I plotted the state-values for each state for TD prediction in Figure 7 and 8. Before and after tuning we can observe that the variance in all the state-values of TD are low but the state-values never seem to converge at the true values. This shows the high bias TD learning has, as discussed in class.

8. In this question I wrote a routine to run experiments for `max_runs` number of times. The routine runs First-Visit MC, Every-Visit MC and TD learning for `max_episodes` number of episodes using the tuned hyperparameters for each of the algorithms. The routine then plots the average state-values across all runs for each algorithm. The results are shown in Figure 9, 10 and 11. We can see that the average state-values for First-Visit MC and Every-Visit MC converge to the true values but the average state-values for TD learning do not converge to the true values. This is consistent with the general behaviours of the algorithms.

9. **Comparision of FVMC, EVMC and TD based on state-value functions across episodes and log-scale of episodes:** We can see that the state-values for FVMC and EVMC converge to the true values but the state-values for TD learning do not converge to the true values. This is consistent with the general behaviours of the algorithms. The variance in the state-values for FVMC and EVMC is higher than TD learning but the bias in the state-values for TD learning is higher than FVMC and EVMC. The plot on log-scale helps us to zoom into the first few episodes and make a better judgement on the high variance phenomenon of Monte Carlo methods.

10. **Comparision of FVMC, EVMC and TD based on average target value for state 3:** We can see that the average across episodes for FVMC and EVMC osciilates between 0 and 1 but the average for TD learning has fewer oscillations but does not land at the true values. Thus, this is another view to note

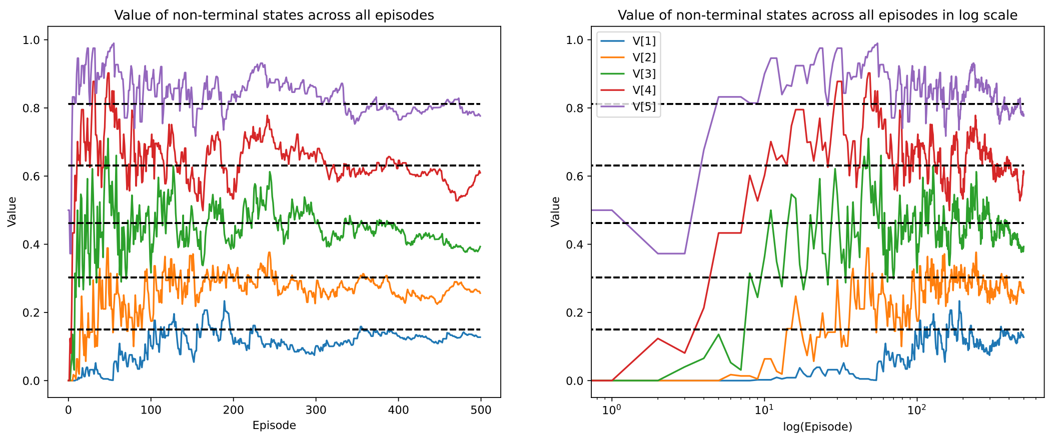


Figure 7: TD with exponential decay of step size

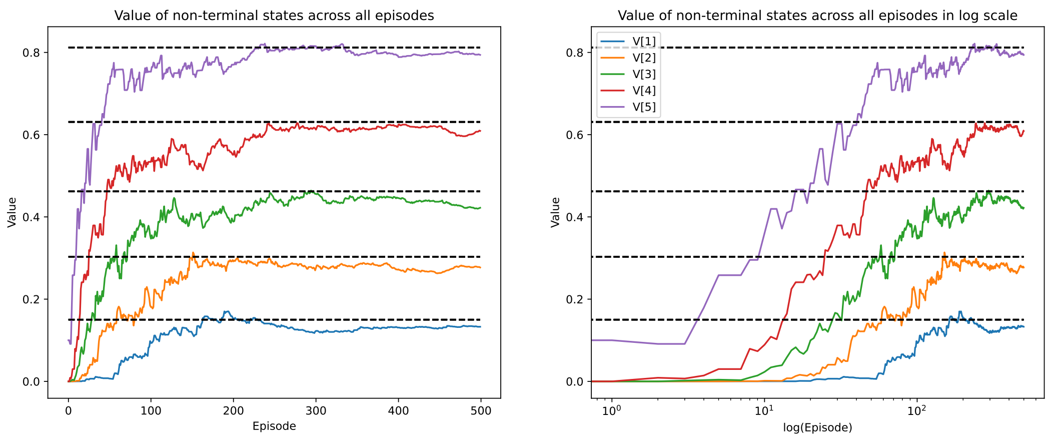


Figure 8: TD with exponential decay and tuned hyperparameters

the high variance in Monte Carlo methods and the high bias in TD learning.

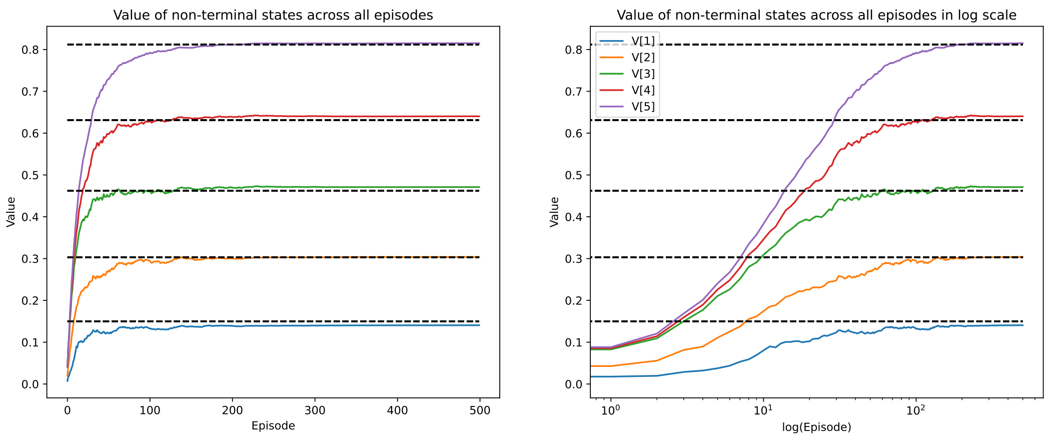


Figure 9: Average state-values of 50 runs of FVMC with exponential decay and tuned hyperparameters

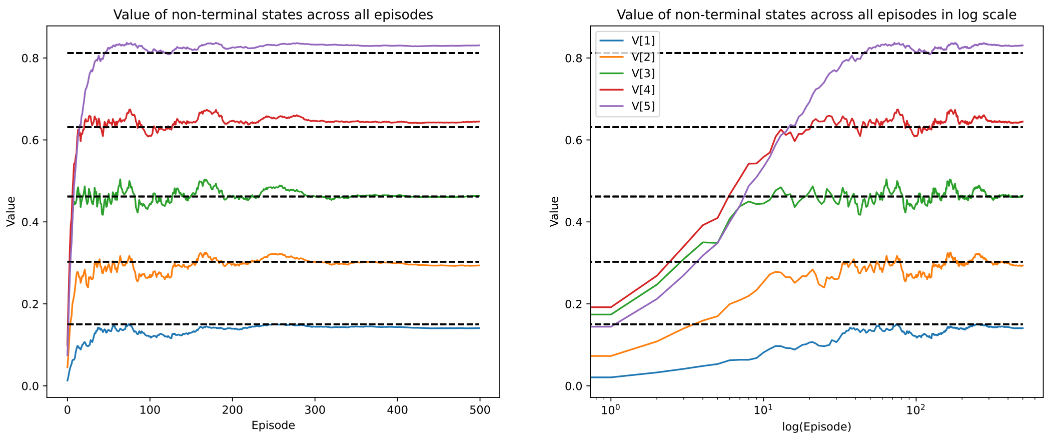


Figure 10: Average state-values of 50 runs of EVMC with exponential decay and tuned hyperparameters

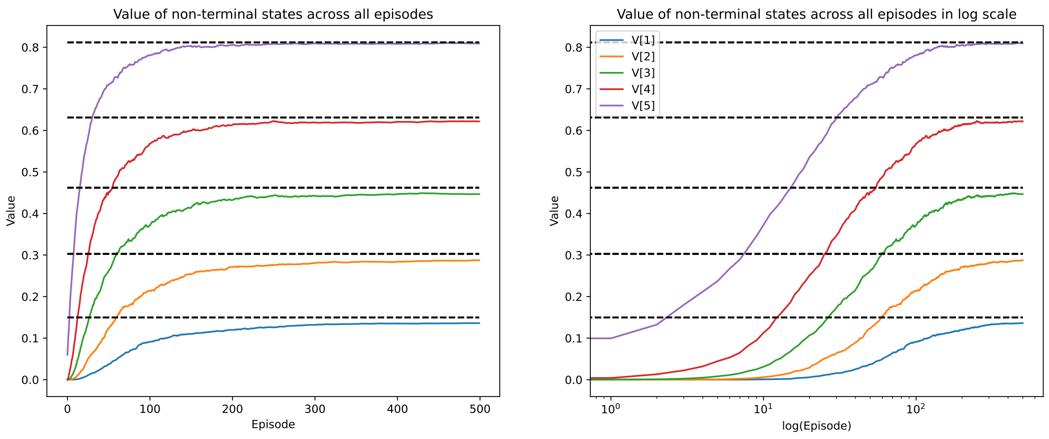


Figure 11: Average state-values of 50 runs of TD with exponential decay and tuned hyperparameters