CS780: Deep Reinforcement Learning

Assignment #1

Name: TA
Roll NO.: 000

Solution to Problem 1: Multi-armed Bandits

1. In this two-armed Bernoulli Bandit environment, there are two arms, each associated with a Bernoulli distribution.

   - Arm 1: Reward $R_0 \sim \text{Bernoulli}(\alpha)$
   - Arm 2: Reward $R_1 \sim \text{Bernoulli}(\beta)$

   2023 is globally used as a seed for all random number generation procedure. α and β are the probabilities of going in the direction of action left and right respectively. If both α and β are 1 then we should always get a reward 1. If either of them is 1, then we should get a reward of 1 whenever we take that action. Here are the results for some test cases for 10 episodes respectively.

   - test case: [1,1](should always give a reward of 1)
   - output:

     ```
     Episode:1 End State: 2 Reward:1 Action:1
     Episode:2 End State: 2 Reward:1 Action:1
     Episode:3 End State: 2 Reward:1 Action:1
     Episode:4 End State: 2 Reward:1 Action:1
     Episode:5 End State: 2 Reward:1 Action:1
     Episode:6 End State: 0 Reward:1 Action:0
     Episode:7 End State: 0 Reward:1 Action:0
     Episode:8 End State: 2 Reward:1 Action:1
     Episode:9 End State: 2 Reward:1 Action:1
     Episode:10 End State: 0 Reward:1 Action:0
     ```

   - test case: [1,0](should give 0 reward for every right action and 1 reward for every left action)
   - output:

     ```
     Episode:1 End State: 0 Reward:0 Action:1
     Episode:2 End State: 0 Reward:0 Action:1
     Episode:3 End State: 0 Reward:0 Action:1
     Episode:4 End State: 0 Reward:0 Action:1
     Episode:5 End State: 0 Reward:0 Action:1
     Episode:6 End State: 0 Reward:1 Action:0
     Episode:7 End State: 0 Reward:1 Action:0
     Episode:8 End State: 0 Reward:0 Action:1
     Episode:9 End State: 0 Reward:0 Action:1
     Episode:10 End State: 0 Reward:1 Action:0
     ```

   - test case: [0.5,0.5](We can't tell much about which action will give what reward.)
   - output:

```
Episode:1 End State: 0 Reward:0 Action:1
Episode:2 End State: 2 Reward:1 Action:1
Episode:3 End State: 2 Reward:1 Action:1
Episode:4 End State: 0 Reward:0 Action:1
Episode:5 End State: 0 Reward:0 Action:1
Episode:6 End State: 0 Reward:1 Action:0
Episode:7 End State: 0 Reward:1 Action:0
Episode:8 End State: 2 Reward:1 Action:1
Episode:9 End State: 2 Reward:1 Action:1
Episode:10 End State: 2 Reward:0 Action:0
```

2. The 10-armed Gaussian Bandit environment is defined as follows:

   - Each arm k has an expected reward $q^*(k) \sim \mathcal{N}(0, 1)$.
   - The reward R_k for pulling arm k follows $R_k \sim \mathcal{N}(q^*(k), 1)$.

   σ^2 and μ is the variance and the mean of the Gaussian Distribution from which our environment samples the rewards. From the sample test cases, tt is clear that if $\sigma = 0$ then our reward of each action should be the respective μ.

   - When $\sigma = 0$ and $\mu = 1$

     ```
     Episode:1 End State: 7 Reward:1.0 Action:6
     Episode:2 End State: 8 Reward:1.0 Action:7
     Episode:3 End State: 7 Reward:1.0 Action:6
     Episode:4 End State: 6 Reward:1.0 Action:5
     Episode:5 End State: 10 Reward:1.0 Action:9
     Episode:6 End State: 10 Reward:1.0 Action:9
     Episode:7 End State: 6 Reward:1.0 Action:5
     Episode:8 End State: 2 Reward:1.0 Action:1
     Episode:9 End State: 2 Reward:1.0 Action:1
     Episode:10 End State: 5 Reward:1.0 Action:4
     ```

   - When $\sigma = 0$ and $\mu = 0.5$

     ```
     Episode:1 End State: 7 Reward:0.5 Action:6
     Episode:2 End State: 8 Reward:0.5 Action:7
     Episode:3 End State: 7 Reward:0.5 Action:6
     Episode:4 End State: 6 Reward:0.5 Action:5
     Episode:5 End State: 10 Reward:0.5 Action:9
     Episode:6 End State: 10 Reward:0.5 Action:9
     Episode:7 End State: 6 Reward:0.5 Action:5
     Episode:8 End State: 2 Reward:0.5 Action:1
     Episode:9 End State: 2 Reward:0.5 Action:1
     Episode:10 End State: 5 Reward:0.5 Action:4
     ```

   - When $\sigma = 1$ and $\mu = 0$

     ```
     Episode:1 End State: 7 Reward:1.5778856155498644 Action:6
     Episode:2 End State: 8 Reward:1.1149232358223107 Action:7
     Episode:3 End State: 7 Reward:1.6189093854468868 Action:6
     Episode:4 End State: 6 Reward:-1.5676130044460233 Action:5
     Episode:5 End State: 10 Reward:-4.150893548459952 Action:9
     Episode:6 End State: 10 Reward:-3.6433063574212365 Action:9
     Episode:7 End State: 6 Reward:-1.5640501930760686 Action:5
     Episode:8 End State: 2 Reward:-0.4237291307420886 Action:1
     ```

CS780 – Assignment #1 TA (000)

```
Episode:9 End State: 2 Reward:-1.7047429694305634 Action:1
Episode:10 End State: 5 Reward:0.19974297477995479 Action:4
```

3. (a) **Pure Exploitation (Greedy) Strategy**: In this strategy always, we select the action with the highest estimated value. It exploits the current knowledge acquired about the environment. One of the key advantages of this strategy is its simplicity and computational efficiency. However, a major drawback is that it can get stuck in suboptimal actions if exploration is not performed. Additionally, if the initial estimates are inaccurate, it may not discover better actions.

Episode no.	Q estimate for Action 0	Q estimate for Action 1	Rewards	Action Taken
1	0.000	0.000	0	0
2	0.000	0.000	0	1
3	0.000	0.500	1	1
4	0.000	0.667	1	1
5	0.000	0.500	0	1
6	0.000	0.400	0	1
7	0.000	0.333	0	1
8	0.000	0.286	0	1
9	0.000	0.375	1	1
10	0.000	0.444	1	1

(b) **Pure Exploration Strategy**: The Pure Exploration strategy randomly selects actions without considering their estimated values. It focuses solely on exploration, ensuring that all actions are explored. This approach guarantees a comprehensive understanding of the environment. However, it can be inefficient if the environment has many actions, and it may not exploit learned knowledge effectively.

Episode no.	Q estimate for Action 0	Q estimate for Action 1	Rewards	Action Taken
1	0.000	0.000	0	0
2	0.000	0.000	0	1
3	0.000	0.500	1	1
4	0.000	0.667	1	1
5	0.000	0.500	0	1
6	0.000	0.400	0	1
7	1.000	0.400	1	0
8	1.000	0.400	1	0
9	1.000	0.500	1	1
10	1.000	0.571	1	1

(c) **Epsilon-Greedy Strategy**: The ϵ-greedy strategy balances exploration and exploitation by selecting the action with the highest estimated value with probability $1-\epsilon$, and selecting a random action with probability ϵ. This strategy combines the benefits of exploration and exploitation, allowing for the discovery of better actions while still exploiting known good actions. However, tuning the ϵ parameter is necessary, and there is a risk of getting stuck in suboptimal actions if ϵ is too small.

CS780 – Assignment #1 TA (000)

Episode no.	Q estimate for Action 0	Q estimate for Action 1	Rewards	Action Taken
1	0.000	0.000	0	0
2	0.000	0.000	0	1
3	0.000	0.500	1	1
4	0.000	0.667	1	1
5	0.000	0.500	0	1
6	0.000	0.400	0	1
7	0.000	0.333	0	1
8	0.000	0.286	0	1
9	0.000	0.375	1	1
10	0.000	0.444	1	1

(d) **Decaying epsilon-Greedy Strategy**: Similar to the ϵ-Greedy strategy, the decaying strategy selects actions based on their estimated values. However, the ϵ parameter decays over time, reducing exploration as the agent learns more about the environment. This approach adapts exploration to the agent's learning progress and can maintain a good balance between exploration and exploitation. If the decay rate is too slow, the agent may converge prematurely to suboptimal actions, similar to epsilon-greedy with a small fixed value.

Episode no.	Q estimate for Action 0	Q estimate for Action 1	Rewards	Action Taken
1	0.000	0.000	0	0
2	0.000	1.000	1	1
3	0.000	0.500	0	1
4	0.000	0.500	0	0
5	0.000	0.333	0	1
6	0.000	0.500	1	1
7	0.000	0.400	0	1
8	0.000	0.333	0	1
9	0.500	0.333	1	0
10	0.500	0.286	0	1

(e) **Softmax Exploration**: Softmax exploration selects actions based on their estimated rewards, assigning higher probabilities to actions with higher estimated rewards. The selection probability of an action a is given by:

$$P(a) = \frac{e^{\frac{Q(a)}{\tau}}}{\sum_b e^{\frac{Q(b)}{\tau}}} \tag{1}$$

where $Q(a)$ is the estimated reward for action a, τ is the temperature parameter controlling exploration, and the denominator is the sum over all actions, ensuring probabilities sum to 1. The first table is for linear temperature decay and the second table is for exponential temperature decay.

Episode no.	Q estimate for Action 0	Q estimate for Action 1	Rewards	Action Taken
1	0.000	0.000	0	0
2	0.000	0.000	0	1
3	1.000	0.000	1	0
4	1.000	0.000	1	0
5	1.000	0.000	0	1
6	1.000	0.000	0	1
7	1.000	0.250	1	1
8	1.000	0.200	0	1
9	1.000	0.200	1	0
10	0.750	0.200	0	0

CS780 – Assignment #1 TA (000)

Episode no.	Q estimate for Action 0	Q estimate for Action 1	Rewards	Action Taken
1	0.000	0.000	0	0
2	1.000	0.000	1	0
3	1.000	0.000	0	1
4	1.000	0.500	1	1
5	1.000	0.500	1	0
6	1.000	0.667	1	1
7	1.000	0.667	1	0
8	0.750	0.667	0	0
9	0.600	0.667	0	0
10	0.500	0.667	0	0

(f) **Upper Confidence Bound (UCB) Strategy**: The UCB strategy selects actions by considering both their estimated rewards and the uncertainty in these estimates. It aims to pick the action with the highest upper confidence bound, calculated as:

$$UCB(a) = Q(a) + c\sqrt{\frac{\ln t}{N(a)}} \tag{2}$$

Here, $Q(a)$ is the estimated reward for action a, c is a constant that influences the level of exploration, t is the total number of actions taken, and $N(a)$ is the number of times action a has been selected. This approach encourages the exploration of less frequently chosen actions.

Episode no.	Q estimate for Action 0	Q estimate for Action 1	Rewards	Action Taken
1	0.000	0.000	0	0
2	1.000	0.000	1	0
3	1.000	1.000	1	1
4	1.000	0.500	0	1
5	0.500	0.500	0	0
6	0.500	0.667	1	1
7	0.500	0.500	0	1
8	0.667	0.500	1	0
9	0.750	0.500	1	0
10	0.800	0.500	1	0

4. **Comparative Analysis of Bandit Strategies for two-armed Bernoulli Bandit environment**: In our comprehensive study, we evaluated six strategies on the 2-armed Bernoulli Bandit problem over 1000 timesteps and across 50 different environments. The strategies included Pure Exploration, UCB Exploration, Pure Exploitation, Epsilon-Greedy, Softmax Exploration, and Decaying Epsilon-Greedy. We plotted the average rewards per timestep for each strategy. The rewards at each timestep were averaged over all environments.

   From the individual plots (Figures 8 to 6), we can make the following observations:

   - The UCB Exploration strategy shows a higher average reward compared to Pure Exploration and Pure Exploitation.
   - The Softmax Exploration strategy's performance gradually increases, reflecting its learning capability over time. We can see that it performs even better on longer run.

   Figure 7 displays the comparative analysis, where we can visually observe the aforementioned trends. The UCB Exploration and Decaying Epsilon-Greedy strategies are particularly notable for their high and stable average reward profiles.

   In summary, the data suggests that strategies which adapt their exploration-exploitation balance according to the environment, such as UCB Exploration and Decaying Epsilon-Greedy, offer a performance advantage in the 2-armed Bernoulli Bandit problem.

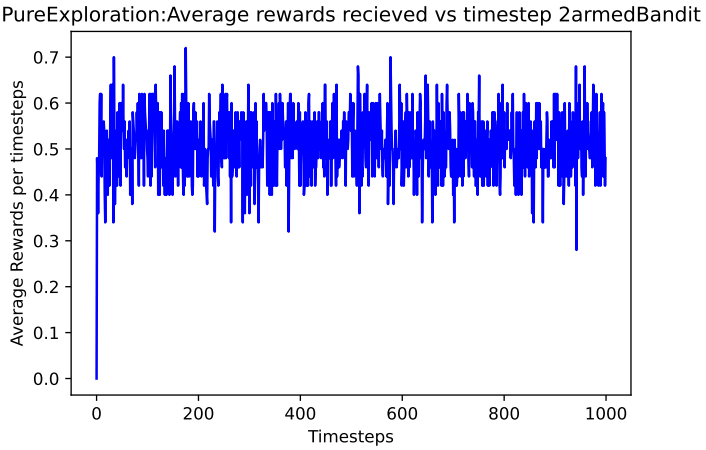


Figure 1: Pure Exploration strategy performance.

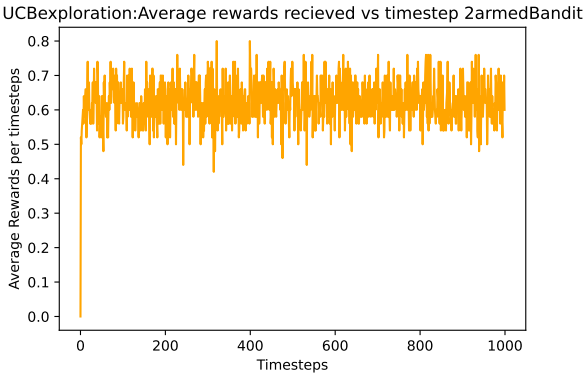


Figure 2: UCB Exploration strategy performance.

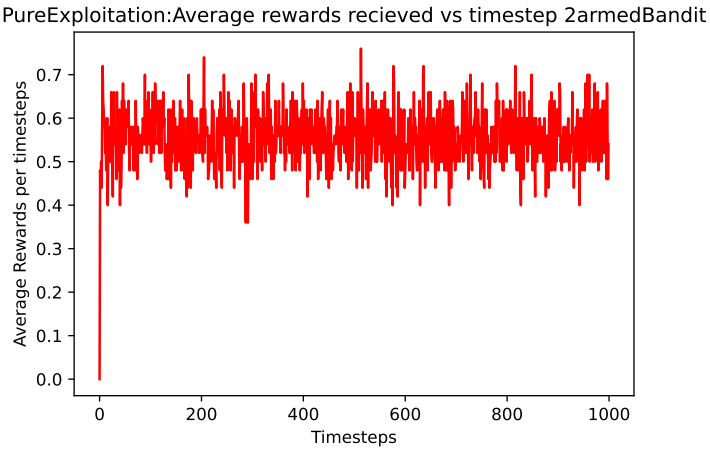


Figure 3: Pure Exploitation strategy performance.

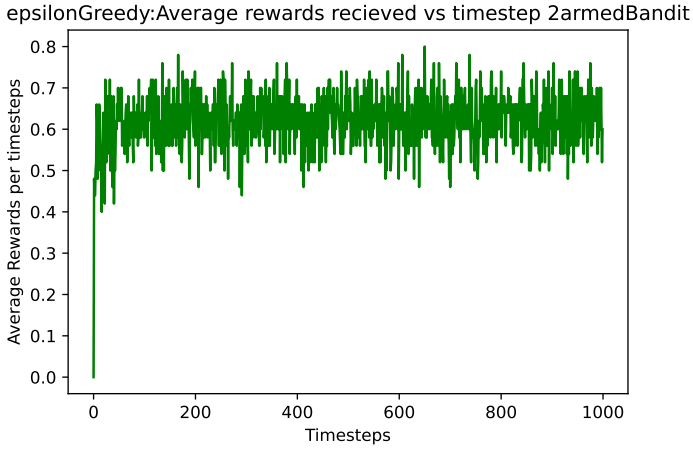


Figure 4: Epsilon-Greedy strategy performance.

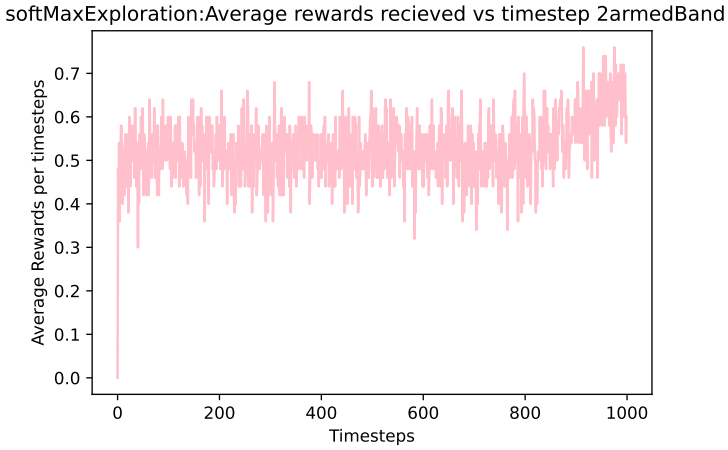


Figure 5: Softmax Exploration strategy performance.

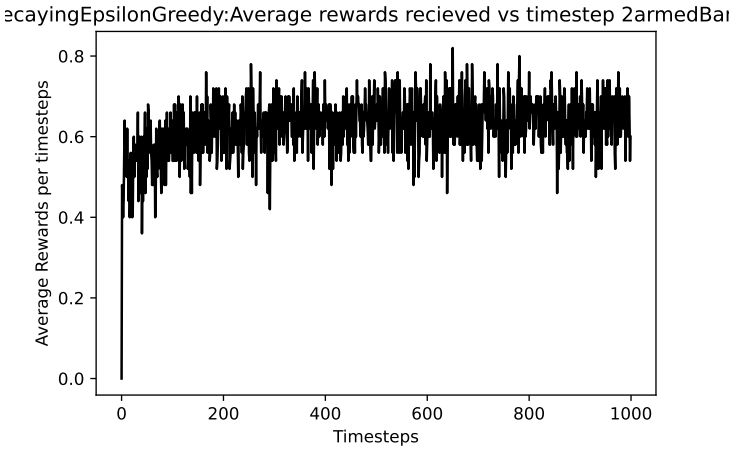


Figure 6: Decaying Epsilon-Greedy strategy performance.

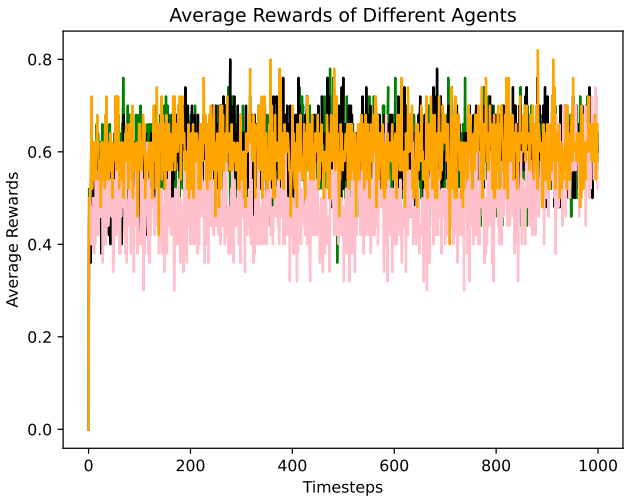


Figure 7: Composite plot illustrating the average rewards of all agents. Strategies like UCB Exploration (orange) and Decaying Epsilon-Greedy (black) achieve higher average rewards.

5. **Comparative Analysis of Bandit Strategies for ten-armed Gaussian environment**: We analyzed the performance of six strategies on a 10-armed Gaussian Bandit problem. Each strategy was executed over 1000 timesteps, with rewards recorded at each step. The average rewards were then computed across multiple runs to ascertain the efficiency and learning progression of each strategy.

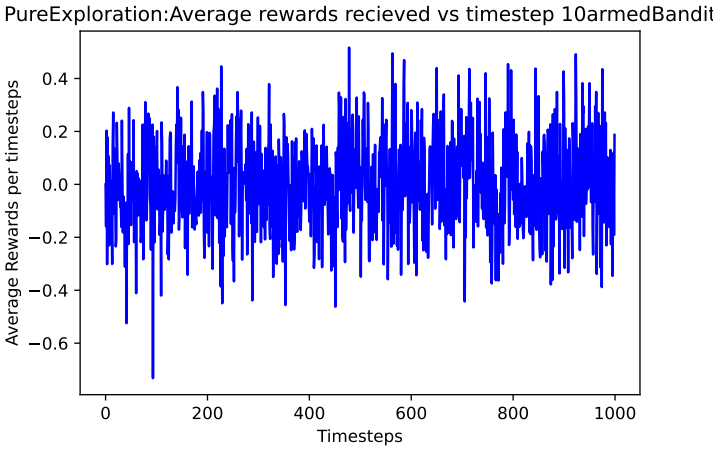


Figure 8: Pure Exploration strategy performance.

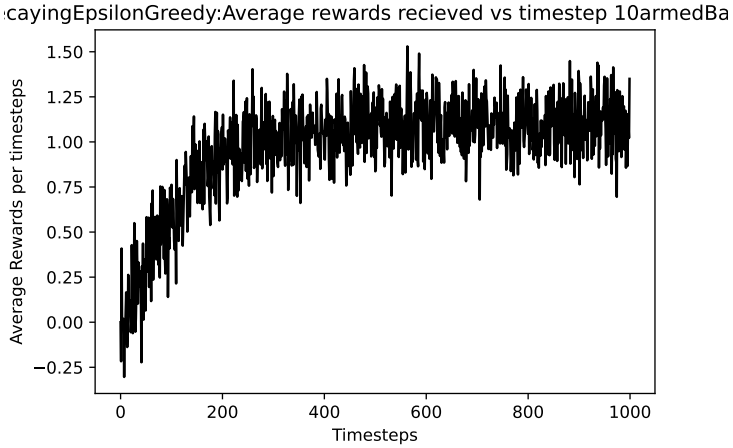


Figure 10: Decaying Epsilon-Greedy strategy performance.

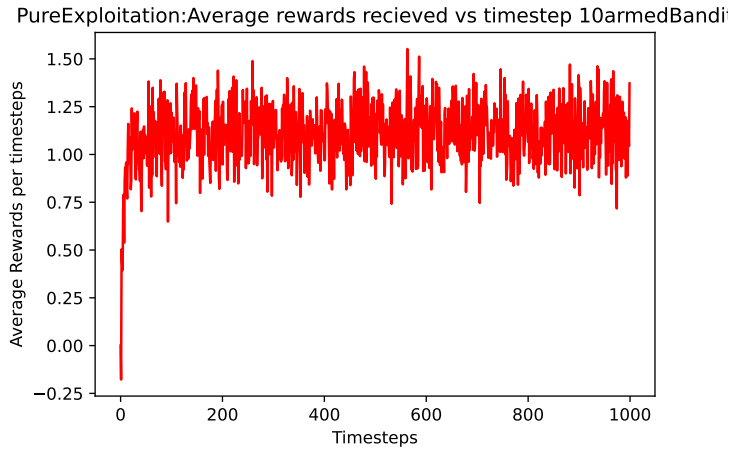


Figure 11: Pure Exploitation strategy performance.

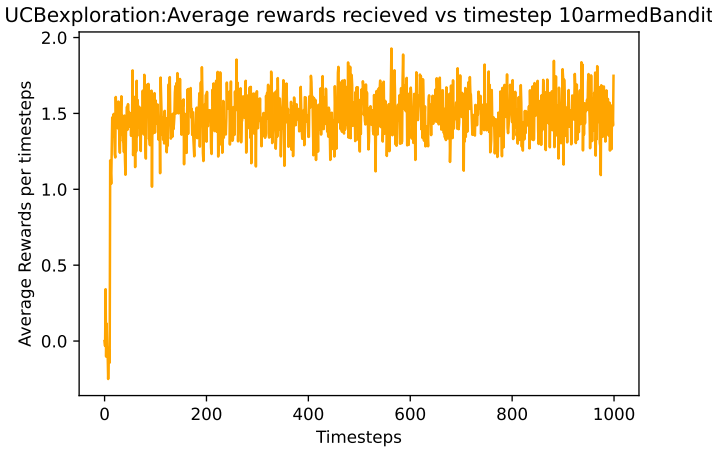


Figure 9: UCB Exploration strategy performance.

Analysis of the plots reveals the following:

- The UCB Exploration strategy displays a relatively stable average reward level, suggesting a consistent performance (Figure 9).

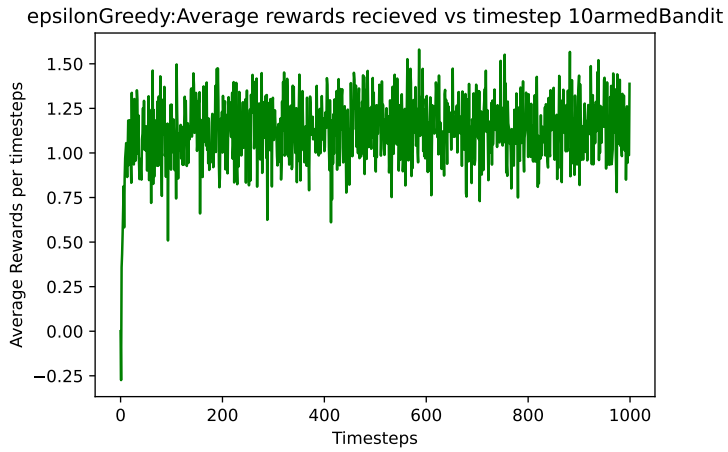


Figure 12: Pure Exploitation strategy performance.

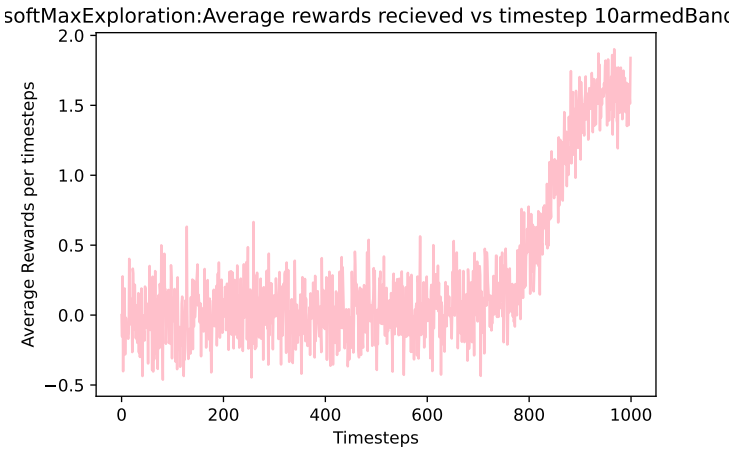


Figure 13: Softmax exploration strategy performance.

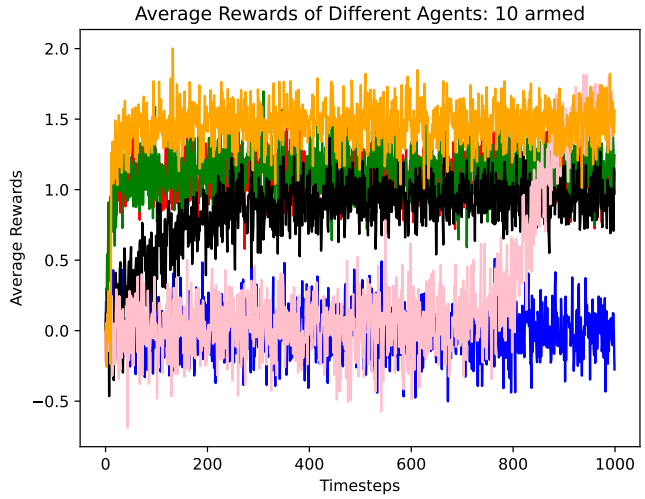


Figure 14: Composite plot comparing the average rewards of all six strategies.

- The Decaying Epsilon-Greedy strategy shows an increasing trend in average rewards, indicating effective learning and exploitation over time (Figure 10).
- The Softmax Exploration strategy's performance exhibits a gradual increase, which could be interpreted as a steady improvement in identifying and exploiting the more rewarding arms (Figure not shown).

6. **Regret Analysis for 2-Armed Bernoulli Bandit**:

   In Figure 15, the regret trajectories for a 2-armed bandit problem are shown. The UCB Exploration strategy (orange line) achieves the least regret, reinforcing its capability to efficiently manage the exploration-exploitation dilemma. The Pure Exploration strategy (blue line), conversely, shows the highest regret, indicating the pitfalls of a strategy that does not exploit the gathered knowledge to optimize its actions.

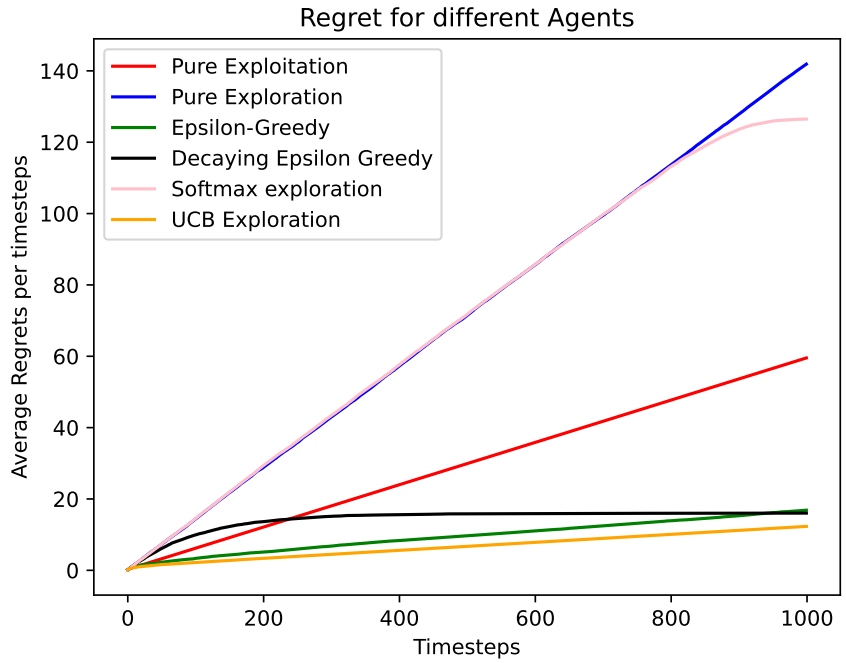


Figure 15: Regret for different agents in a 2-armed Bernoulli Bandit environment.

7. **Regret Analysis for 10-Armed Gaussian Bandit**:

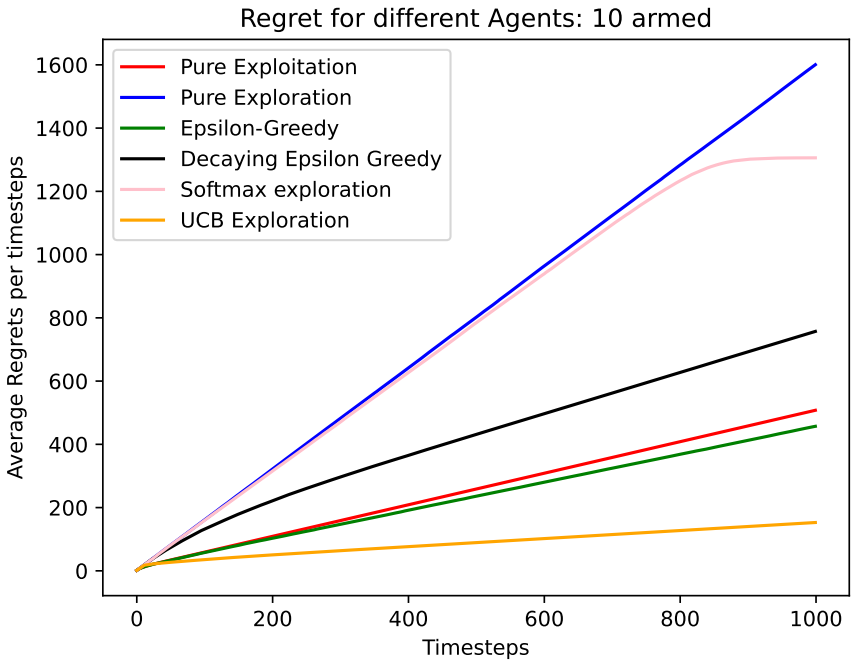


Figure 16: Regret for different agents in a 10-armed Gaussian Bandit environment.

Figure 16 presents the regret over time for several strategies in a 10-armed bandit environment. The UCB Exploration strategy, represented by the orange line, exhibits the lowest regret, making it the most effective in balancing the exploration-exploitation trade-off. In stark contrast, the Pure Exploration strategy, shown by the blue line, accumulates the highest regret due to its consistent exploration without leveraging what it has learned, resulting in suboptimal action choices.

8. **% of optimal Actions taken by different agents in 2 armed Bernoulli bandit environment**

   As we can see in 17, UCB Exploration (orange) achieves consistently optimal action selection, with a steady line indicating low variability. The Decaying Epsilon Greedy strategy (black) also shows high performance with minimal variability towards the end. Both strategies demonstrate robustness against different conditions within the environment.

   The Epsilon-Greedy approach (green) has a moderate level of variability in its performance, yet maintains a decent percentage of optimal actions.

   In contrast, Pure Exploration (blue) strategy exhibits the highest variability, with wide bands indicating inconsistent performance. Despite this, Pure Exploration does not improve significantly over time, while Pure Exploitation also remains at a low performance level throughout.

   Softmax Exploration (pink) displays an initial period of high variability, which gradually reduces, and the performance also seems to improve towards the end.

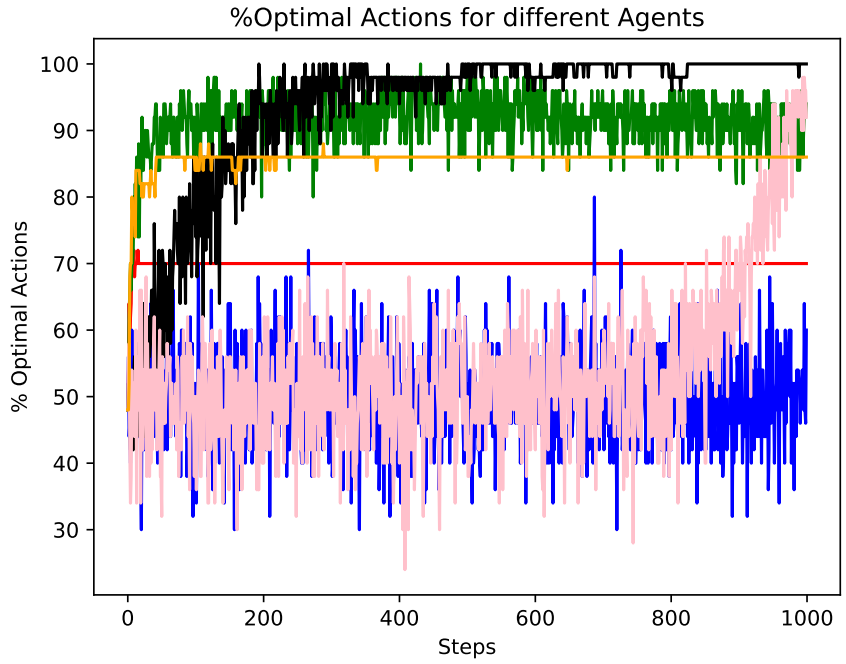


Figure 17

9. **% of optimal Actions taken by different agents in 10 armed Gaussian environment**

As we can see in 18, the UCB Exploration (orange) maintains the highest percentage of optimal actions consistently across episodes, with a very tight performance band indicating low variability. The Decaying Epsilon Greedy (black) also performs well but with more fluctuation, suggesting occasional explorative decisions.

Epsilon-Greedy (green) exhibits considerable variability (only lower than UCB for most of the time) and a moderate percentage of optimal actions, while Exploration (blue) show both higher variability and lower performance levels.

The Softmax Exploration strategy (pink) begins with a low percentage of optimal actions. As the episodes progress, the strategy shows a gradual improvement in the selection of optimal actions. However, this improvement is accompanied by a noticeable degree of variability, as evidenced by the thickness of the pink shaded area, which indicates the confidence interval or the range of performance across different instances.

Pure Exploitation (red), characterized by its narrow band, indicates a consistent selection of sub-optimal actions, reinforcing the idea that without exploration, the strategy cannot adapt to find better options.

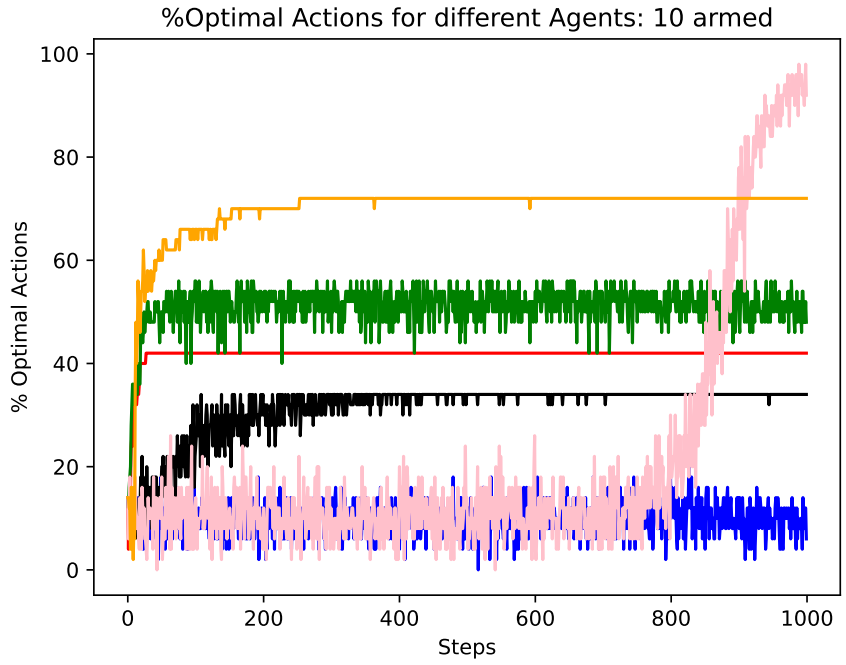


Figure 18

Solution to Problem 2: MC Estimates and TD Learning

Random Walk Environment Implementation

In this report, we present the implementation details of the Random Walk Environment.

Environment Class: `RandomWalkEnv`

The `RandomWalkEnv` class is designed to simulate a 1D grid world with an agent that can take actions to move left or right. Key attributes and methods include:

- `size`: The size of the 1D grid.
- `observation_space`: Discrete space representing the possible states.
- `action_space`: Discrete space representing possible actions (left or right).
- `slip_prob`: Probability of slipping when taking an action.
- `start_loc`: Initial location of the agent.
- `reset`: Reset the environment to its initial state.
- `step`: Take an action and observe the next state, reward, and termination status.
- `render`: Render the environment in either human or RGB array mode.

Rendering and Visualization

The environment supports rendering in two modes: "human" for interactive rendering and "rgb_array" for programmatic access to the rendered frames. The `_render_frame` method handles the visualization of the environment using Pygame.

Usage Example

Here's an example of using the `RandomWalkEnv` class:

```
import gym
env = gym.make('RandomWalkEnv')
observation = env.reset()
action = env.action_space.sample()
next_observation, reward, done, info = env.step(action)
```

Visualization

To visualize the environment, you can use the `render` method:

```
env.render()
```

This will display the environment in the chosen rendering mode.

1. ## Trajectory Generation Function

   The `generateTrajectory` function simulates and generates a trajectory for the Random Walk Environment based on a given policy and a maximum number of steps. The trajectory is represented as a list of experience tuples, each containing the state, action, reward, next state, and termination flag.

   - `generateTrajectory(env, π, maxSteps)`: Generates a trajectory using the provided policy in the environment `env` with a maximum number of steps `maxSteps`. The function returns a list of experience tuples.

Policy Function

The policy function `policy(state)` is defined to always return 0, representing the action to "go left."

Code Usage and Test

We tested the implemented functions using the Random Walk Environment and the defined policy. A trajectory was generated with a maximum of 100 steps.

Generated Trajectory

The generated trajectory is as follows:

```
Generated Trajectory:
(3, 0, 0, 2, False)
(2, 0, 0, 1, False)
(1, 0, 0, 0, True)
```

This indicates that the agent started at state 3, took a left action, moved to state 2, took another left action, and reached the goal state 0, terminating the episode.

2. Step Size Parameter Decay

In this problem, we implemented a function to decay the step size parameter (α) from an initial value to a final value over a specified number of steps. The decay can follow either a linear or an exponential pattern.

Decay Function

The `decayAlpha` function takes four parameters: `initialValue`, `finalValue`, `maxSteps`, and `decayType`. It returns a list of step size parameter values over time.

- `decayAlpha(initialValue, finalValue, maxSteps, decayType)`: Decays the step size parameter from `initialValue` to `finalValue` over `maxSteps` steps. `decayType` specifies the type of decay, either 'linear' or 'exponential'.

Test and Visualization

We tested the `decayAlpha` function with different parameter settings and visualized the decay of the step size parameter over time for both linear and exponential decays.

```
Initial Value: 0.1
Final Value: 0.01
Max Steps: 100
```

Plots

The following plots illustrate the decay of the step size parameter (α) over time for both linear and exponential decay types.

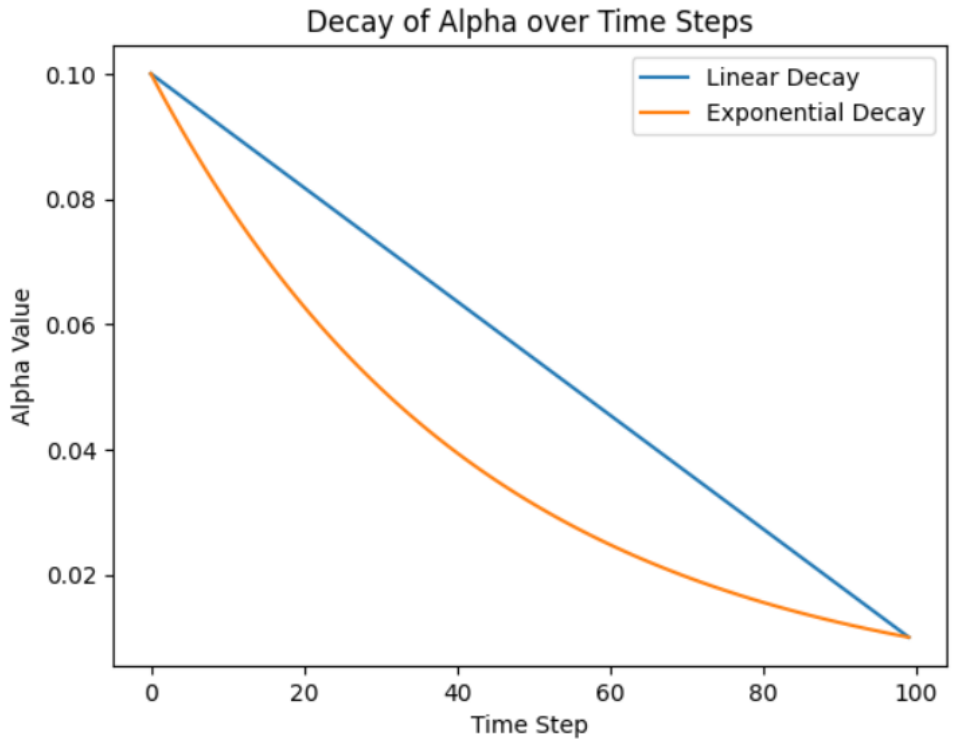


Figure 19: Decay of Step Size Parameter (α)

Observations

The plots clearly show the decay of the step size parameter over time. The choice between linear and exponential decay types can impact the rate at which α decreases. Linear decay results in a constant decrement in each step, while exponential decay leads to a more rapid initial decrease.

3. Monte Carlo Prediction

In this problem, we implemented the Monte Carlo Prediction algorithm for estimating state values in a given environment. The algorithm uses a specified policy, step size parameters, and decay type to update state values based on generated trajectories.

Monte Carlo Prediction Function

The `MonteCarloPrediction` function takes several parameters, including the environment, policy, maximum number of steps, initial and final values of the step size parameter, decay type, and visit type. It returns an array of state values estimated by the Monte Carlo Prediction algorithm.

- `MonteCarloPrediction(env, π, maxSteps, _initial, _final, decayType, visitType='first')`: Implements the Monte Carlo Prediction algorithm for estimating state values. `visitType` can be either 'first' or 'every' for first-visit or every-visit Monte Carlo prediction.

Test and Visualization

We tested the algorithm on the Random Walk Environment with a specified policy (always go left) and different visit types. The step size parameter follows a linear decay from an initial value of 0.1 to a final value of 0.01 over 1000 steps.

```
Environment: Random Walk
Initial Value: 0.1
Final Value: 0.01
Max Steps: 1000
Decay Type: Linear
```

Results

The estimated state values using both first-visit (FVMC) and every-visit (EVMC) Monte Carlo Prediction are as follows:

```
Estimated State Values using First Visit Monte Carlo Prediction:
[0.         0.19668167 0.32141383 0.47656478 0.66716926 0.82938219
 0.        ]

Estimated State Values using Every Visit Monte Carlo Prediction:
[0.         0.11224478 0.26519076 0.4176359  0.5793621  0.82152329
 0.        ]
```

4. Temporal Difference Prediction

In this problem, we implemented the Temporal Difference Prediction algorithm for estimating state values in a given environment. The algorithm uses a specified policy, step size parameters, decay type, and discount factor to update state values based on observed transitions.

Temporal Difference Prediction Function

The `TemporalDifferencePrediction` function takes several parameters, including the environment, policy, maximum number of steps, initial and final values of the step size parameter, decay type, and discount factor (γ). It returns an array of state values estimated by the Temporal Difference Prediction algorithm.

- `TemporalDifferencePrediction(env, π, maxSteps, _initial, _final, decayType, )`: Implements the Temporal Difference Prediction algorithm for estimating state values.

Test and Visualization

We tested the algorithm on the Random Walk Environment with a specified policy (always go left) and a discount factor of $\gamma = 0.99$. The step size parameter follows a linear decay from an initial value of 0.1 to a final value of 0.01 over 1000 steps.

```
Environment: Random Walk
Initial Value: 0.1
Final Value: 0.01
Max Steps: 1000
Decay Type: Linear
Discount Factor: 0.99
```

Results

The estimated state values using Temporal Difference Prediction are as follows:

```
Estimated State Values using Temporal Difference Prediction:
[0.         0.12357635 0.31215954 0.41810243 0.57597497 0.69986257
 0.        ]
```

5. MC-FVMC Estimate Progression

In this problem, we implemented the MC-FVMC estimate algorithm for the Random Walk Environment. The algorithm estimates state values using Monte Carlo methods with a first-visit approach and varying step size parameters.

MC-FVMC Estimate Function

The `plot_MC_FVMC_estimate` function takes several parameters, including the environment, policy, maximum number of episodes, initial and final values of the step size parameter, decay type, and discount factor. It plots the MC-FVMC estimate of each non-terminal state as the algorithm progresses through different episodes.

- `plot_MC_FVMC_estimate(env, π, maxEpisodes, _initial, _final, decayType, )`

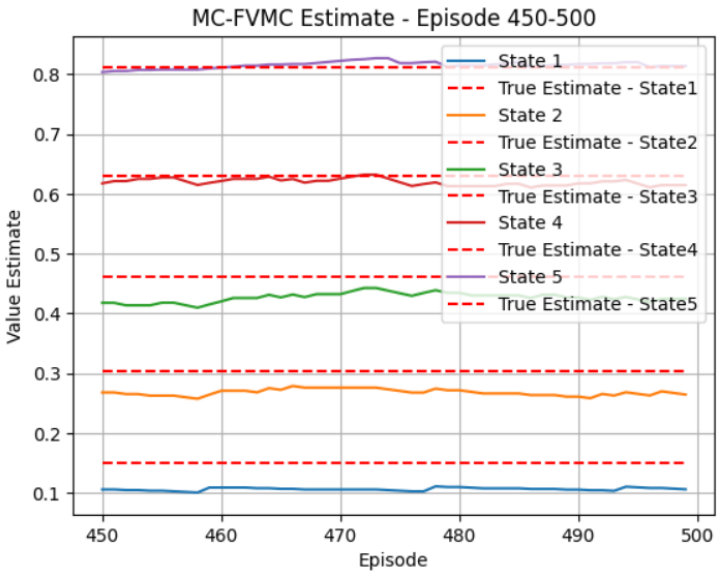
Test and Visualization

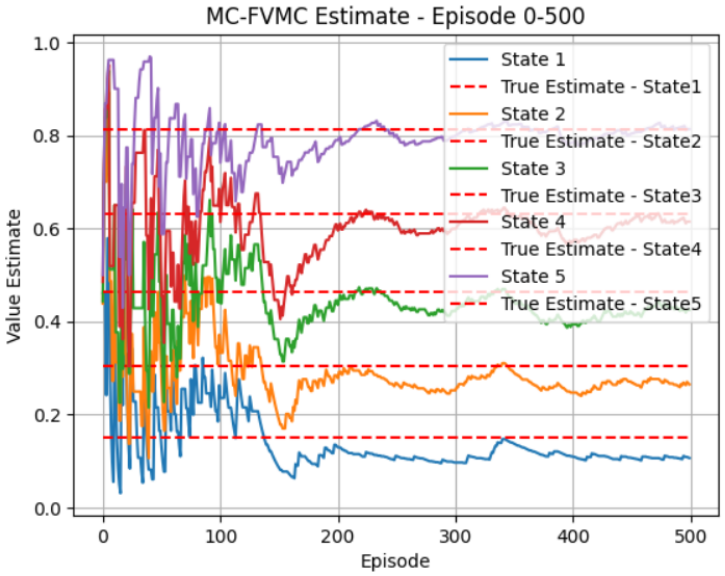
We tested the algorithm on the Random Walk Environment with a specified policy (always go left) and parameters for step size decay, discount factor, and maximum episodes.

```
Environment: Random Walk
Policy: Always go left
Max Episodes: 500
alpha_initial: 0.5
alpha_final: 0.01
Decay Type: Exponential
Discount Factor: 0.99
```

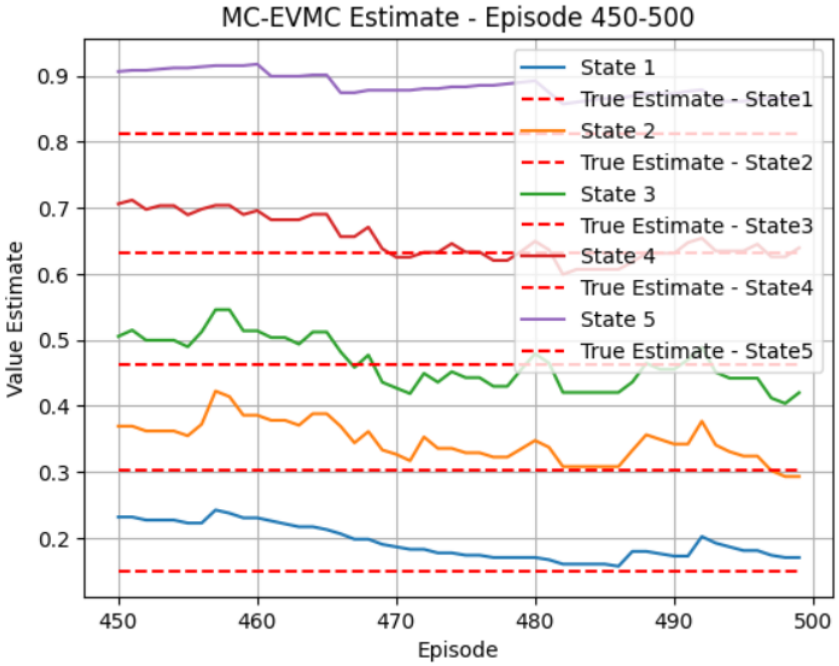
Results

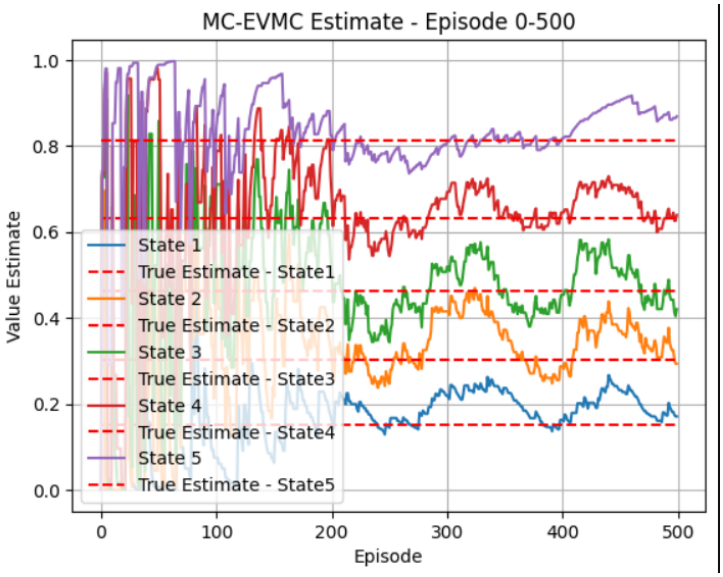
The MC-FVMC estimate progression for each non-terminal state is visualized in figure 2. The dashed line represents the true estimate.





6. MC-EVMC Estimate Function





7. TD Estimate Progression

In this problem, we implemented the Temporal Difference (TD) estimate algorithm for the Random Walk Environment. The algorithm estimates state values using the TD update rule and varying step size parameters.

TD Estimate Function

The `plot_TD_estimate` function takes several parameters, including the environment, policy, maximum number of episodes, initial and final values of the step size parameter, decay type, and discount factor. It plots the TD estimate of each non-terminal state as the algorithm progresses through different episodes.

- `plot_TD_estimate(env, π, maxEpisodes, _initial, _final, decayType, )`

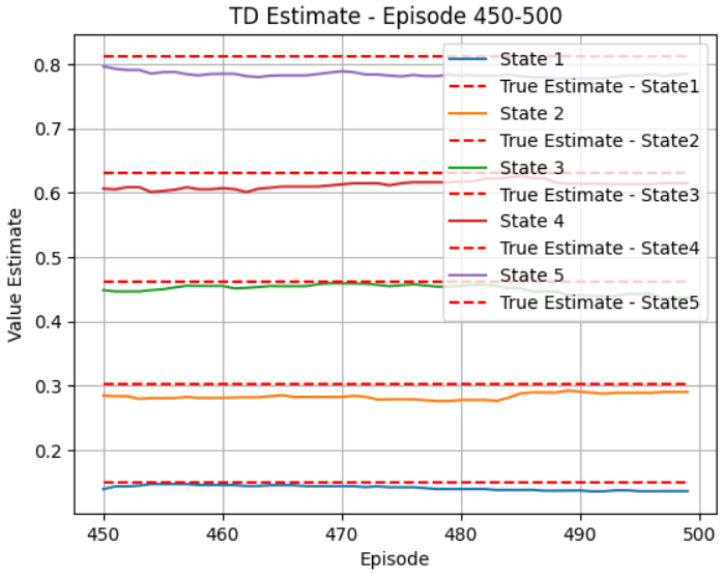
Test and Visualization

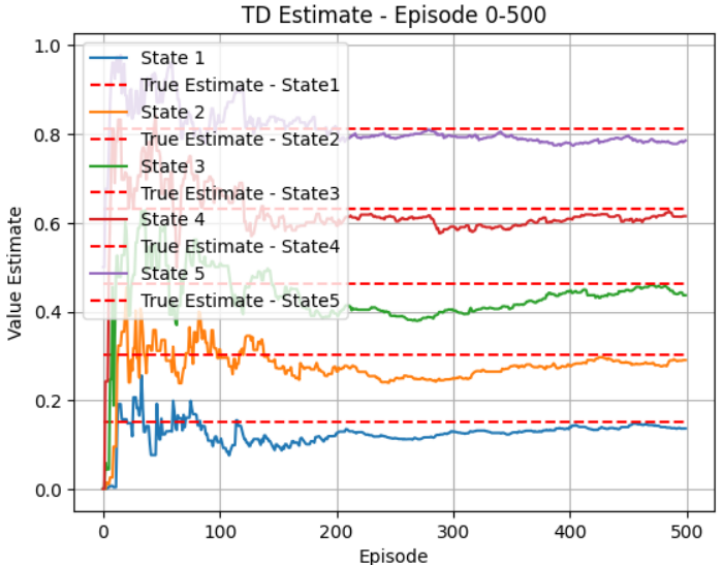
We tested the algorithm on the Random Walk Environment with a specified policy (always go left) and parameters for step size decay, discount factor, and maximum episodes.

```
Environment: Random Walk
Policy: Always go left
Max Episodes: 500
_initial: 0.5
_final: 0.01
Decay Type: Exponential
Discount Factor: 0.99
```

Results

The TD estimate progression for each non-terminal state is visualized in the Figure 4. The dashed line represents the true estimate.


TD Estimate - Episode 450-500
Value Estimate
Episode
State 1
True Estimate - State1
State 2
True Estimate - State2
State 3
True Estimate - State3
State 4
True Estimate - State4
State 5
True Estimate - State5


TD Estimate - Episode 0-500
Value Estimate
Episode
State 1
True Estimate - State1
State 2
True Estimate - State2
State 3
True Estimate - State3
State 4
True Estimate - State4
State 5
True Estimate - State5

8. Smoothed MC-FVMC, MC-EVMC, and TD Estimates

In this problem, we implemented and analyzed the smoothed Monte Carlo (MC) estimates for first-visit MC (FVMC), every-visit MC (EVMC), and Temporal Difference (TD) for each non-terminal state in the Random Walk Environment.

Smoothed Estimates Function

The `plot_smoothed_MC_TD_estimate` function takes several parameters, including the environment class, policy, maximum number of episodes, initial and final values of the step size parameter, decay type, discount factor, and the number of environment instances. It computes and plots the smoothed MC-FVMC, MC-EVMC, and TD estimates for each non-terminal state.

- `plot_smoothed_MC_TD_estimate(env_cls, π, maxEpisodes, α_initial, α_final, decayType, , num_instances)`

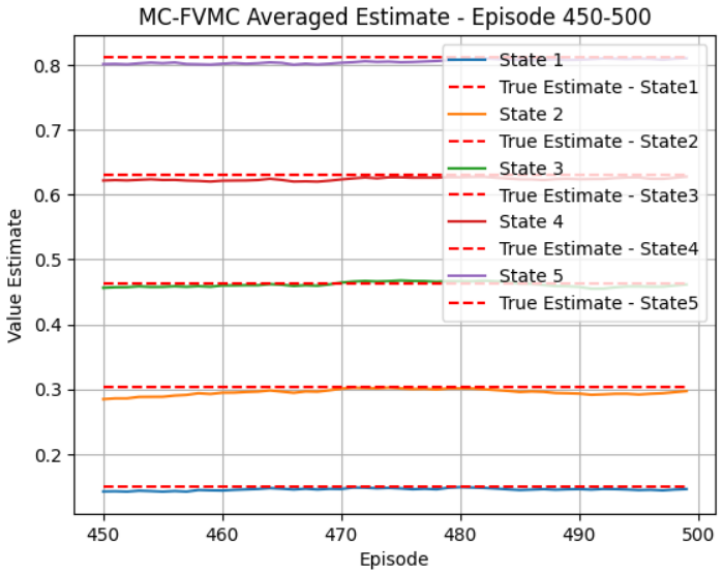
Test and Visualization

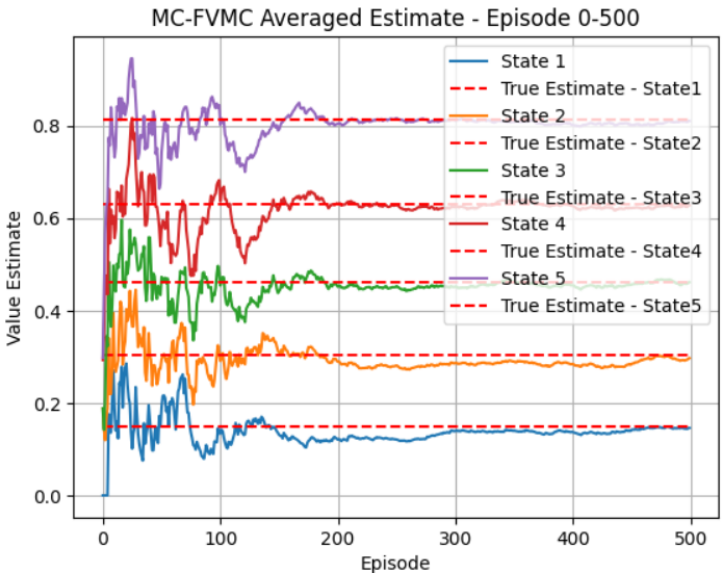
We tested the algorithm on the Random Walk Environment with a specified policy (always go left) and parameters for step size decay, discount factor, and the number of environment instances.

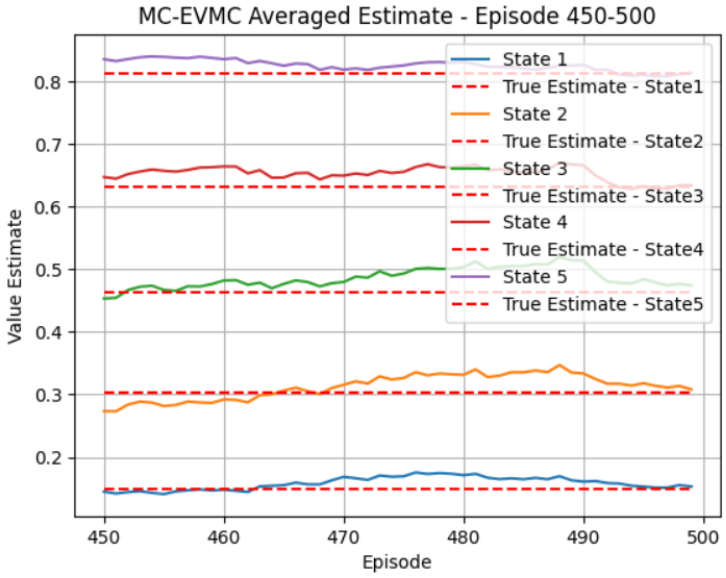
```
Environment Class: Random Walk
Policy: Always go left
Max Episodes: 500
alpha_initial: 0.5
alpha_final: 0.01
Decay Type: Exponential
Discount Factor: 0.99
Number of Instances: 5
```

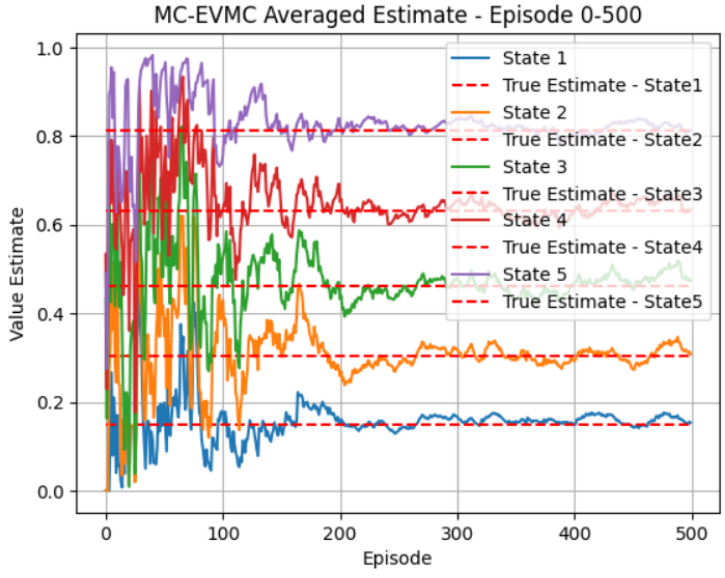
Results

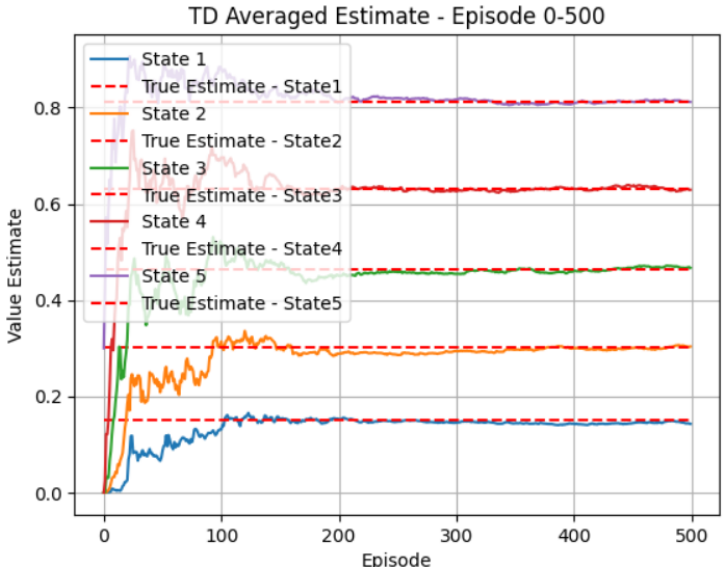
The smoothed MC-FVMC, MC-EVMC, and TD estimates for each non-terminal state are visualized in figure 5. The dashed line represents the true estimate.

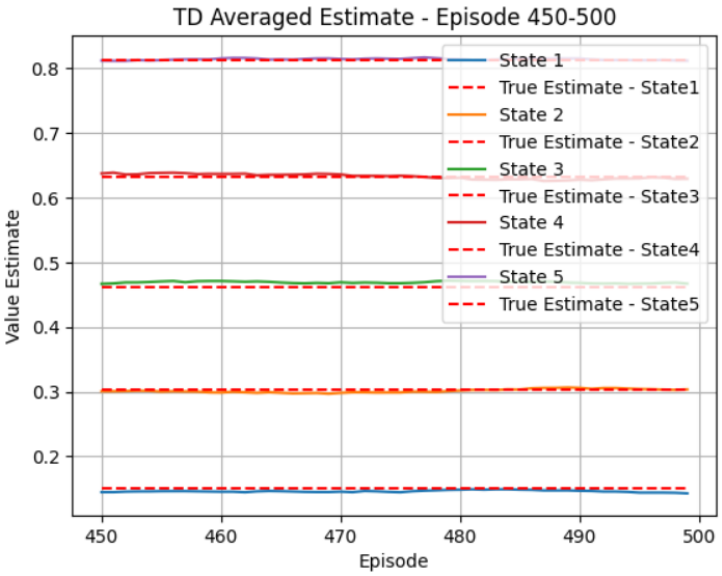



MC-FVMC Averaged Estimate - Episode 0-500
State 1
True Estimate - State1
State 2
True Estimate - State2
State 3
True Estimate - State3
State 4
True Estimate - State4
State 5
True Estimate - State5
Value Estimate
Episode
0.0
0.2
0.4
0.6
0.8
0
100
200
300
400
500


MC-EVMC Averaged Estimate - Episode 450-500
State 1
True Estimate - State1
State 2
True Estimate - State2
State 3
True Estimate - State3
State 4
True Estimate - State4
State 5
True Estimate - State5
Value Estimate
Episode
0.2
0.3
0.4
0.5
0.6
0.7
0.8
450
460
470
480
490
500

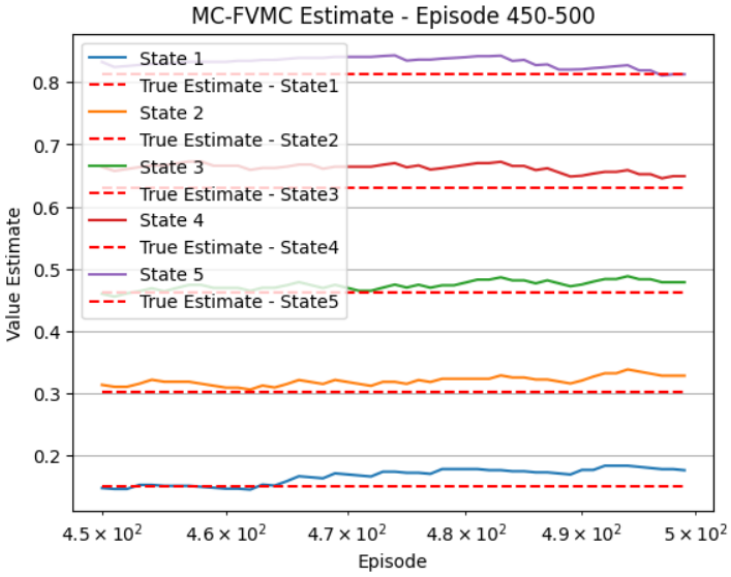

MC-EVMC Averaged Estimate - Episode 0-500
State 1
True Estimate - State1
State 2
True Estimate - State2
State 3
True Estimate - State3
State 4
True Estimate - State4
State 5
True Estimate - State5
Value Estimate
Episode
0.0
0.2
0.4
0.6
0.8
1.0
0
100
200
300
400
500

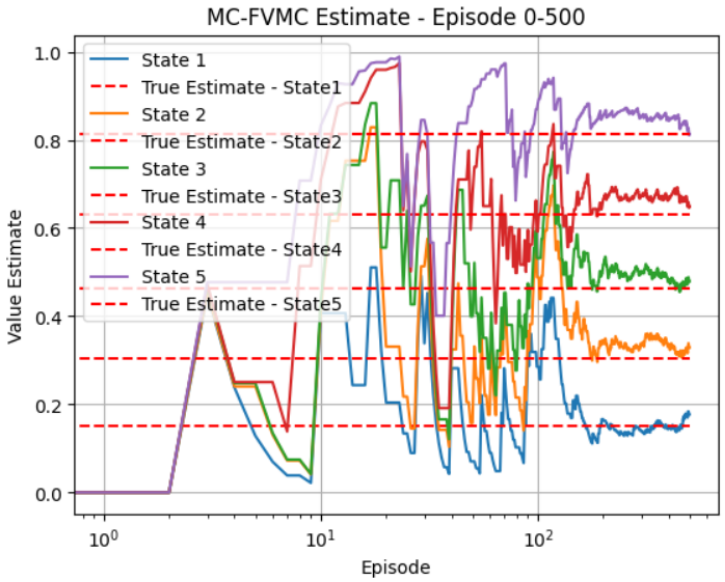

TD Averaged Estimate - Episode 0-500
State 1
True Estimate - State1
State 2
True Estimate - State2
State 3
True Estimate - State3
State 4
True Estimate - State4
State 5
True Estimate - State5
Value Estimate
0.0
0.2
0.4
0.6
0.8
0
100
200
300
400
500
Episode


TD Averaged Estimate - Episode 450-500
State 1
True Estimate - State1
State 2
True Estimate - State2
State 3
True Estimate - State3
State 4
True Estimate - State4
State 5
True Estimate - State5
Value Estimate
0.2
0.3
0.4
0.5
0.6
0.7
0.8
450
460
470
480
490
500
Episode

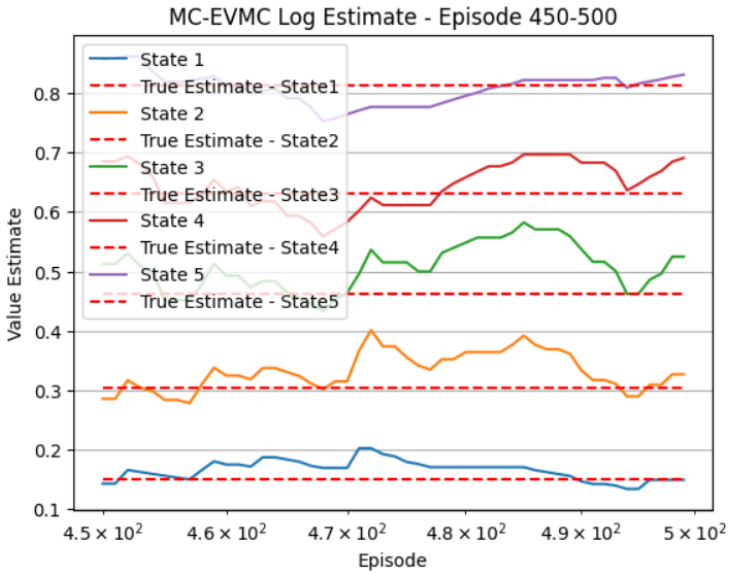
9. MC-FVMC estimate of each non-terminal state of RWE

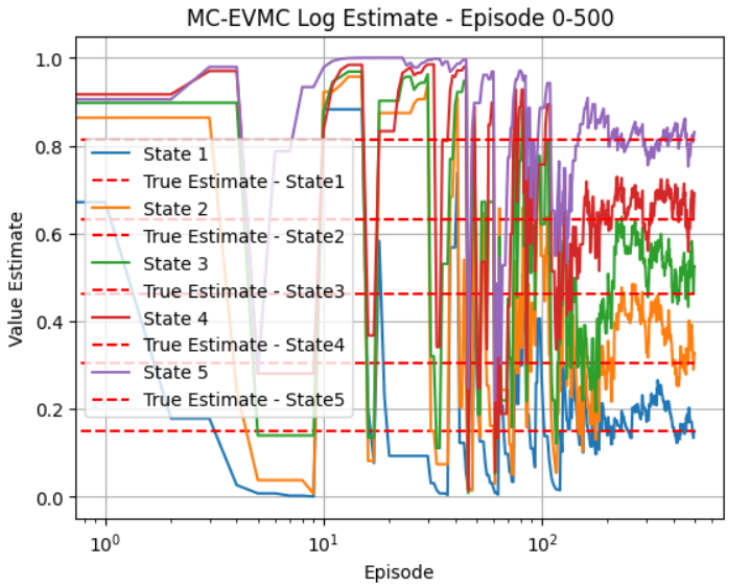
following figure shows the plot for MC-FVMC with logscale.





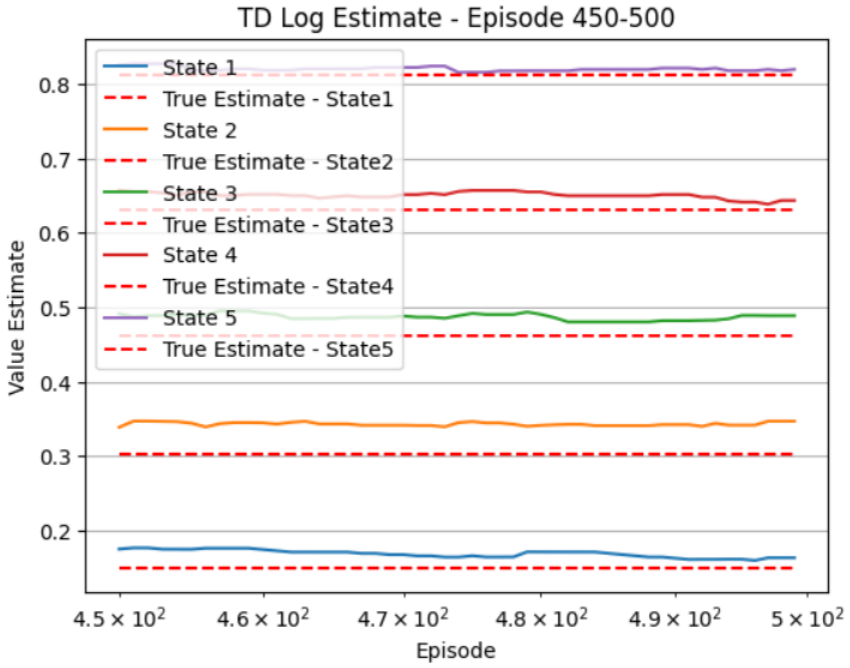
10. MC-EVMC Estimate with log scale

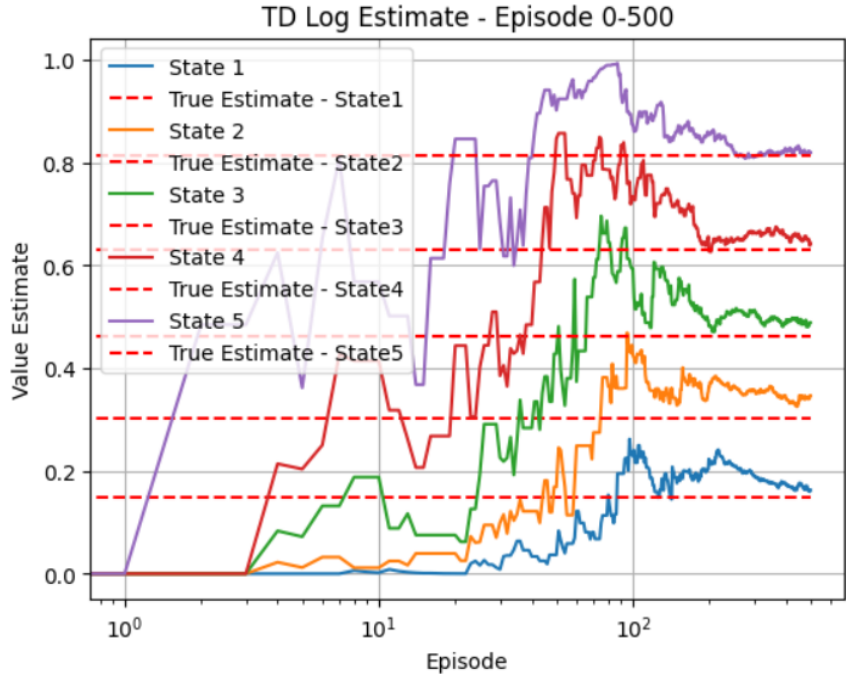




11. TD estimate of each non-terminal state of RWE

following fig. shows the TD with logscale.





12. Compare MC-FVMC, MC-EVMC, and TD

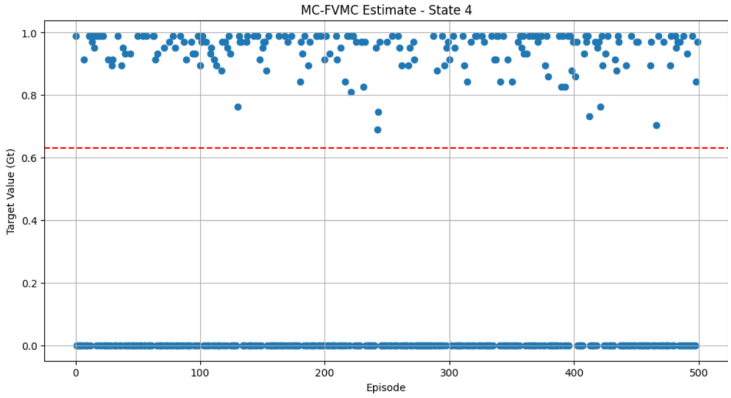
To compare MC-FVMC, MC-EVMC, and TD, we analyze their performance based on state value estimates:

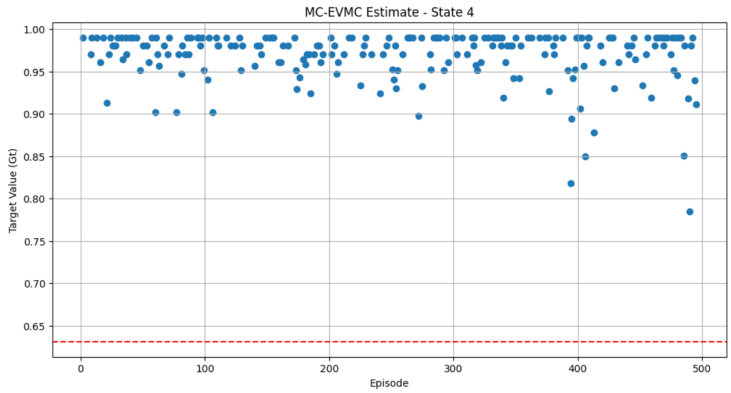
MC-FVMC updates values only for the first visit to a state, showing gradual improvement over episodes but slower convergence. MC-EVMC updates values every visit, converging more rapidly but may exhibit higher variance due to frequent updates. TD updates values after each state

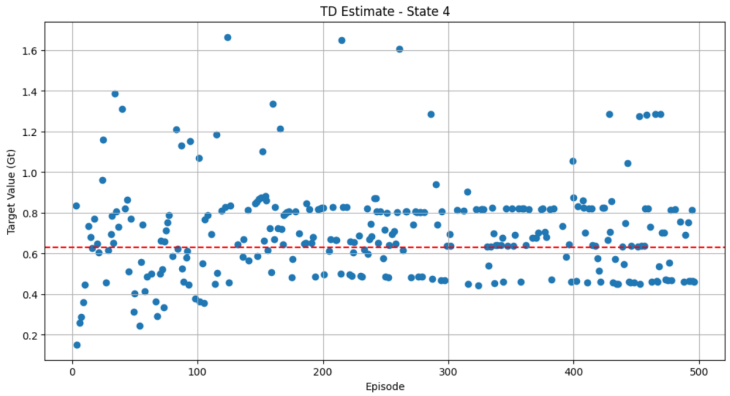
transition, leading to faster convergence and less variance compared to MC methods. Choice depends on problem characteristics and computational resources: MC suited for episodic tasks, TD for online learning with faster convergence.

13. Target Value G_t for State 4 using MC-FVMC, MC-EVMC and TD

The following plots shows the Target Value for the state 4 over the various episodes using the methods of MC-FVMC, MC-EVMC and TD for question 13,14 and 15 refer to fig.9 , fig. 10 and fig 11 respectively.







14. Last question 16:Compare MC-FVMC, MC-EVMC and TD

MC-FVMC: - Smoother convergence compared to MC-EVMC due to updates only at first visits. - Slower convergence compared to TD due to episodic nature and delayed updates. - Lower variance in targets compared to MC-EVMC due to less frequent updates.

MC-EVMC: - Rapid convergence compared to MC-FVMC due to updates at every visit. - Higher variance in targets compared to MC-FVMC due to more frequent updates. - Improvement over time but may exhibit fluctuations in target values.

TD: - Faster convergence compared to both MC-FVMC and MC-EVMC due to incremental updates. - Less variance in targets compared to MC methods, especially in early stages. - More stable and consistent improvement over episodes.

Overall, TD shows faster convergence and lower variance compared to MC methods, with MC-EVMC balancing between rapid convergence and increased variance.