CS698R: Deep Reinforcement Learning

Assignment #1

Name: Abhinav Kumar
Roll NO.: 16907018

Solution to Problem 1: Multi-armed Bandits

1. The environment for 2-armed Bernoulli Bandit follows OpenAI function definitions. In its *__init__()* function the MDP is also defined using a dictionary data structure **P** as mentioned in the lectures. The step function also utilizes this MDP data structure **P** to calculate next state, reward and done values.

   For testing the environment, a function for Policy Evaluation is implemented which follows the algorithm mentioned in the lectures.

   The following policies are evaluated for this environment i) $\overleftarrow{\pi}$ Always Go Left Policy and ii) $\overrightarrow{\pi}$ Always Go Right Policy and the value of the only non-terminal state matches with theoretical values i.e.,

   $$v_{\overleftarrow{\pi}}(0) = 1 - \alpha \quad \text{for Always Go Left}$$
   $$v_{\overrightarrow{\pi}}(0) = \beta \quad \text{for Always Go Right}$$

2. The Ten-Armed Gaussian Bandit environment implementation follows OpenAI function definitions and generates a Ten-Armed Gaussian Bandit as defined in lectures. The MDP data structure **P** is implemented according to the Optimal Q values $q_*(k) \forall k$ and corresponding Rewards $R_t(k)$.

   A function for Policy Evaluation is implemented that follows the algorithm mentioned in lectures. We evaluate a Greedy Policy π_g which selects actions as $a_* = \arg\max_a q_*(a)$ and tries to estimate the value of the only non-terminal state.

   Upon running the python script '*ques1_part2.py*' we can see that action is selected greedily according to Optimal Q values and the Policy Evaluation function estimates the corresponding reward for ten different seed values from 0 to 9.

   Note: Actions range from 0 to 9, any action 'a' will lead to one of the terminal state '$s = a + 1$'.

```
************ SEED=6 ************
Greedy Policy Evaluation
Optimal Q values: [-0.31178367  0.72900392  0.21782079 -0.8990918  -2.48678065  0.91325152
  1.12706373 -1.51409323  1.63929108 -0.4298936 ]
Rewards corresponding to optimal Q values: [ 2.31949688  1.33082617 -0.11806083  0.33864605 -2.37565248  1.04240277
  1.20319134 -1.66922139  2.27351643  0.3807614 ]
Greedy Action: 8
Greedy Policy Distribution: [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
    Value of initial state 0 is 2.27
    Value of terminal state 1 is 0.0
    Value of terminal state 2 is 0.0
    Value of terminal state 3 is 0.0
    Value of terminal state 4 is 0.0
    Value of terminal state 5 is 0.0
    Value of terminal state 6 is 0.0
    Value of terminal state 7 is 0.0
    Value of terminal state 8 is 0.0
    Value of terminal state 9 is 0.0
    Value of terminal state 10 is 0.0
*****************************************
```

Figure 1: Output for Greedy Policy Evaluation on Ten-Armed Gaussian Bandit. We can see that Policy Evaluation function is trying to estimate the True Reward Value = 2.27351643 corresponding to action = 8.

3. (a) The Pure Exploitation algorithm is implemented according to the lectures. In Fig. 2 we can see how Q value estimates evolve with time steps for this strategy. The Going Left action is mapped to 0 and Going Right action is mapped to 1. We can see that the agent is always picking the left action greedily.

The Q function estimation and action selection is given by:

$$Q_t(a) = \frac{\sum_{i=1}^{t-1} R_i * \delta_{A_i==a}}{\sum_{i=1}^{t-1} \delta_{A_i==a}} \tag{1}$$

$$A_t = \underset{a}{\text{argmax}} Q_t(a) \tag{2}$$

For example, after time step $t = 3$ (t starts from 0) the Q value estimate according to the Eq. 1 will be $Q_{t=3}(a = 0) = \frac{0+1+0+1}{4} = 0.5$ as can be seen in the table. Whereas the agent never takes Going Right action and its Q value remains zero throughout.

Note: Seed value = 5, $\alpha = \beta = 0.8$ and file name is *ques1_part3a_pureExploitation.py*.

```
------------------SEED: 5------------------
Q_val|Q[a=0]|Q[a=1]| actionTaken | reward
Q[0] | 0.00 | 0.00 | action: 0.0 | reward: 0.0
Q[1] | 0.50 | 0.00 | action: 0.0 | reward: 1.0
Q[2] | 0.33 | 0.00 | action: 0.0 | reward: 0.0
Q[3] | 0.50 | 0.00 | action: 0.0 | reward: 1.0
Q[4] | 0.40 | 0.00 | action: 0.0 | reward: 0.0
Q[5] | 0.33 | 0.00 | action: 0.0 | reward: 0.0
Q[6] | 0.29 | 0.00 | action: 0.0 | reward: 0.0
Q[7] | 0.25 | 0.00 | action: 0.0 | reward: 0.0
Q[8] | 0.22 | 0.00 | action: 0.0 | reward: 0.0
Q[9] | 0.20 | 0.00 | action: 0.0 | reward: 0.0
```

Figure 2: Output from Pure Exploitation function on Two-Armed Bernoulli Bandit.

(b) For Pure Exploration the action is chosen randomly at every step. In Fig. 3 we can see how the Q value estimates evolve with time steps. Here, in contrast with Pure Exploitation, the agent is choosing values randomly and hence is able to estimate Q values for actions better.

For example, after time step $t = 4$ (t starts from 0), the Q value estimate by Eq. 1 for action $a = 0$ is $Q_{t=4}(a = 0) = \frac{1+0}{2} = 0.5$ and for action $a = 1$ is $Q_{t=4}(a = 1) = \frac{1+1+1}{3} = 1$ as can be seen in Fig. 3.

Note: Seed value = 5, $\alpha = \beta = 0.8$ and file name is *ques1_part3b_pureExploration.py*.

```
------------------SEED: 5------------------
Q_val|Q[a=0]|Q[a=1]| actionTaken | reward
Q[0] | 0.00 | 1.00 | action: 1.0 | reward: 1.0
Q[1] | 0.00 | 1.00 | action: 1.0 | reward: 1.0
Q[2] | 1.00 | 1.00 | action: 0.0 | reward: 1.0
Q[3] | 0.50 | 1.00 | action: 0.0 | reward: 0.0
Q[4] | 0.50 | 1.00 | action: 1.0 | reward: 1.0
Q[5] | 0.33 | 1.00 | action: 0.0 | reward: 0.0
Q[6] | 0.33 | 0.75 | action: 1.0 | reward: 0.0
Q[7] | 0.33 | 0.80 | action: 1.0 | reward: 1.0
Q[8] | 0.25 | 0.80 | action: 0.0 | reward: 0.0
Q[9] | 0.25 | 0.67 | action: 1.0 | reward: 0.0
```

Figure 3: Output from Pure Exploration function on Two-Armed Bernoulli Bandit.

(c) The epsilon greedy agent bridges the gap between pure exploitation and pure exploration and brings a balance between the two. The algorithm is run for three values of epsilon $\epsilon = 0.1, \epsilon = 0.01$, $\epsilon = 0$ which is pure exploitation and $\epsilon = 1$ which is pure exploration.

In Fig. 4 we can see that $\epsilon = 0.1, \epsilon = 0.01$ are performing better than Pure Exploration and Pure Exploitation.

Moreover, throughout the episodes there is exploration present with ϵ probability and hence we observe that the plots have a lot of noise in them even after smoothing.

We want to reduce this exploration probability with time and exploit our estimates, we do this in the next part.

Note: Seed value = 0, $\alpha = \beta = 0.8$ and file name is *ques1_part3b_epsilonGreedy.py*.

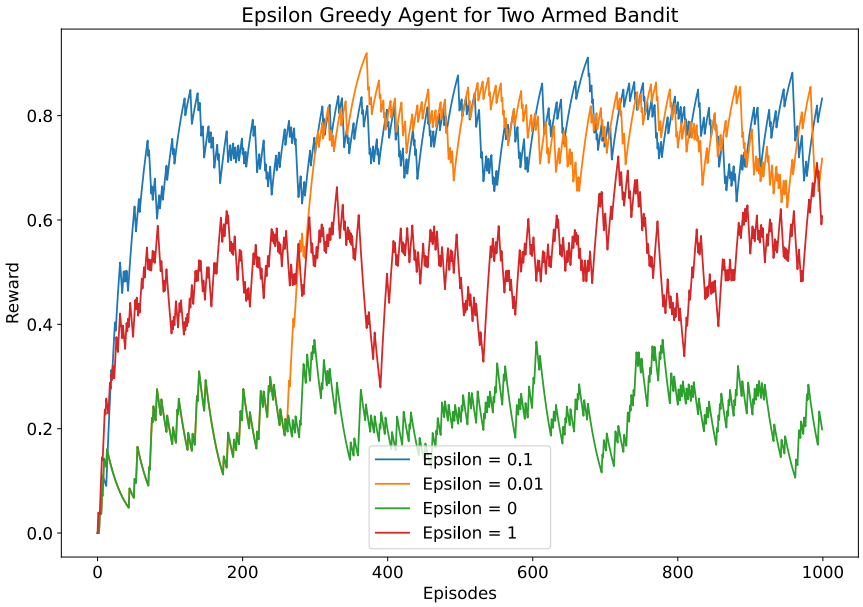


Figure 4: Output from Epsilon Greedy function on Two-Armed Bernoulli Bandit.

(d) Decaying Epsilon Greedy agent balances exploration vs exploitation and helps estimate the Q values.

In Fig. 5 we can see two types of decay, linear and exponential. Starting value of epsilon is $\epsilon_{max} = 1$ and it is decayed till $\epsilon_{min} = 1e^{-6}$ over 500 episodes. Total number of episodes is 1000.

Note: Seed value = 0, $\alpha = \beta = 0.8$ and file name is *ques1_part3b_DecayingEpsilonGreedy.py*.

(e) Softmax utilizes a probability distribution over action-value function controlled by the temperature parameter.

In Fig. 6 we can observe the behaviour of softmax strategy, it starts from optimistic outlook towards all actions and is it explores the temperature parameter is reduced to make the distribution over action-value function more peaky and then the agent chooses optimal actions.

Starting value of tau is $\tau_{max} = 1e5$, final value of tau is $\tau_{min} = 0.005$ and it is decayed till 500 episodes.

Note: Seed value = 0, $\alpha = \beta = 0.8$ and file name is *ques1_part3b_softmax.py*.

(f) UCB is an another optimistic strategy agent which also takes into account statistical knowledge of an action and reduces corresponding exploration accordingly.

In Fig. 7 we compare UCB agent for 4 values $c = 0.1, 1, 5, 10$. We can observe that it converges very fast for all values and to a better value for low values of c for this environment.

Note: Seed value = 0, $\alpha = \beta = 0.8$ and file name is *ques1_part3b_ucb.py*.

4. For this part we do the following to generate 50 environments and run 6 agents on each of the environment for 1000 time steps (episodes):

- SEED: vary seed values from 0 to 50 skipping 42 *(more references in python script)*.
- α, β: We generate both these using **numpy.random.uniform(0,1)** at every iteration.
- Env: We create a new instance of the environment using the above selected SEED and α, β.
- Agent: We call all 6 agents to run over this environment.
- Plot: We average reward values over 50 environments and smooth out the resultant average with a smoothing window of 50.

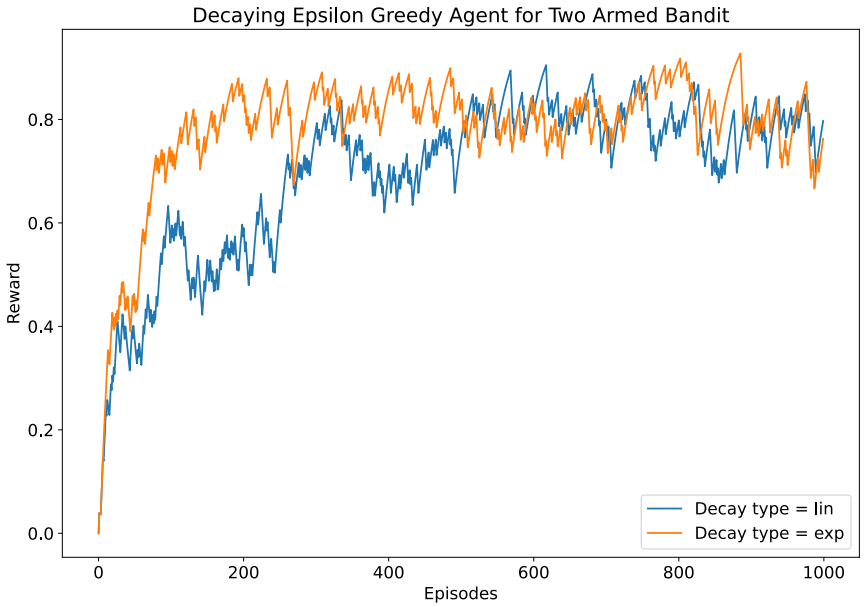


Figure 5: Output from Decaying Epsilon Greedy function on Two-Armed Bernoulli Bandit. $\epsilon_{max} = 1, \epsilon_{min} = 1e^{-6}$ and we decay it till 500 episodes. We can observe that Exponential decay converges faster than linear decay but in the long run both converge to similar values.

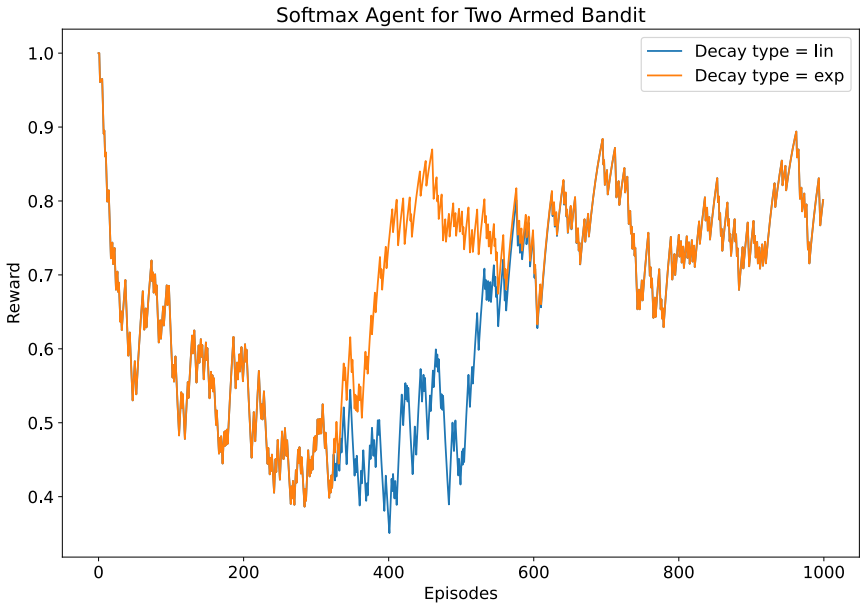


Figure 6: Output from Softmax function on Two-Armed Bernoulli Bandit. $\tau_{max} = 1e5, \tau_{min} = 0.005$ and we decay it till 500 episodes. We can observe that Exponential decay converges faster than linear decay but in the long run both converge to same values.

Following are the hyper-parameters used:

- Epsilon Greedy: $\epsilon = 0.1$
- Decaying Epsilon: $\epsilon_{max} = 1, \epsilon_{min} = 1e^{-6}$ and decayed over 500 time steps, decay type = exponential.
- Softmax: $\tau_{max} = 100, \tau_{min} = 0.005$ and decayed over 500 time steps, decay type = linear

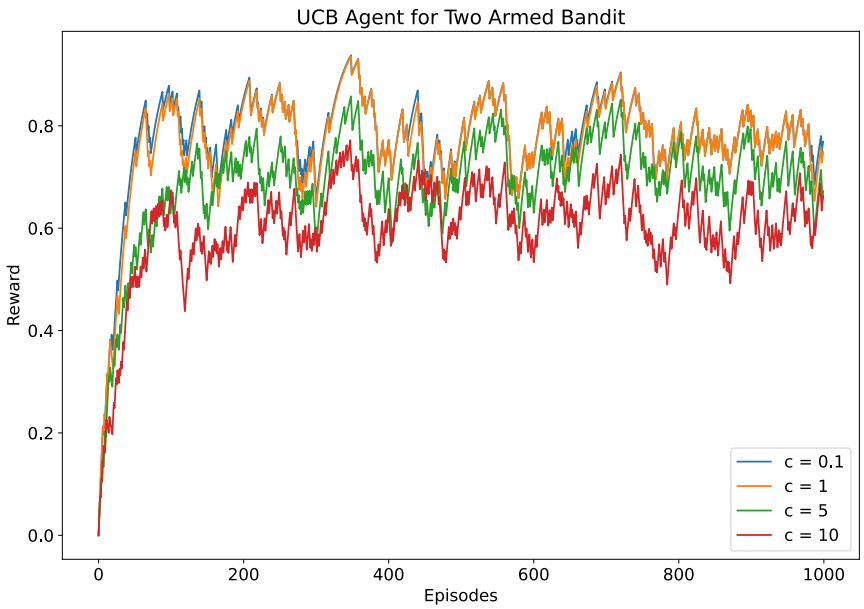


Figure 7: Output from UCB function on Two-Armed Bernoulli Bandit. We compare 4 values of the hyper-paramter 'c'. We can see that the agent performs better with low values of 'c' like 0.1 and 1 for this environment.

- UCB: Value of $c = 0.1$

In Fig. 8 we can see that Decaying Epsilon Greedy agent, Softmax agent, Epsilon Greedy agent and UCB agent converge to the optimal action-value estimates and give higher rewards. Whereas Pure Exploitation and Pure Exploration agents give minimal reward as they can't balance exploration vs exploitation.

Moreover, Decaying Epsilon Greedy agent, Epsilon Greedy agent and UCB agent converge faster than the rest of the agents.

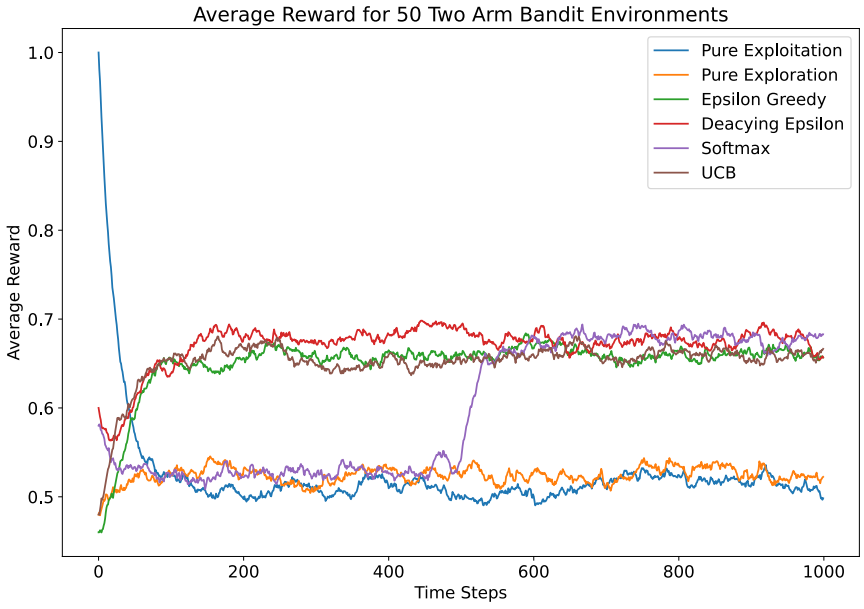


Figure 8: Average Reward Value for different agents averaged over 50 randomly generated environments.

5. (a) Pure Exploitation strategy for Ten-Armed Gaussian Bandit acts greedily and converges to the action giving high rewards as opposed to the action with the highest action-value due to Gaussian noise. In the long run it performs sub-optimally as it does not explores all the actions and acts greedily.
It is clear from Fig. 9 that this agent converges to a sub-optimal action $a = 2$ whereas the best action according to optimal Q values is $a = 3$.
Note: Seed value = 0, $\sigma = 1$ and file name is *ques1_part5_3a_pureExploitation.py*.

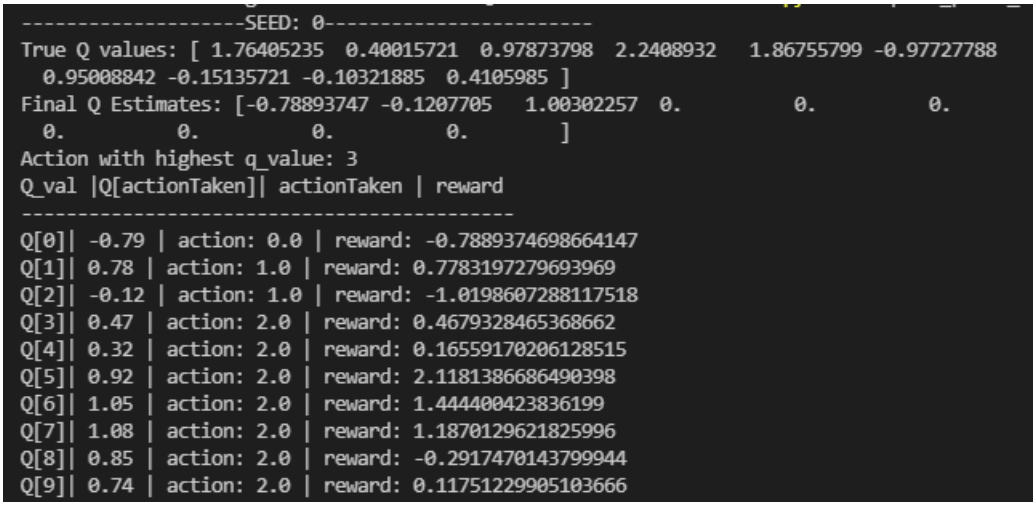


Figure 9: Output from Pure Exploitation function on Ten-Armed Gaussian Bandit.

(b) Pure Exploration strategy takes random actions at every time step and doesn't converges to any optimal value. If run for a long time, it can estimate Q values of the actions.
In Fig. 10 we can see that the best action is $a = 3$ but the agent is exploring randomly and trying to estimate all Q values but it never exploits them.
Note: Seed value = 0, $\sigma = 1$ and file name is *ques1_part5_3b_pureExploration.py*.

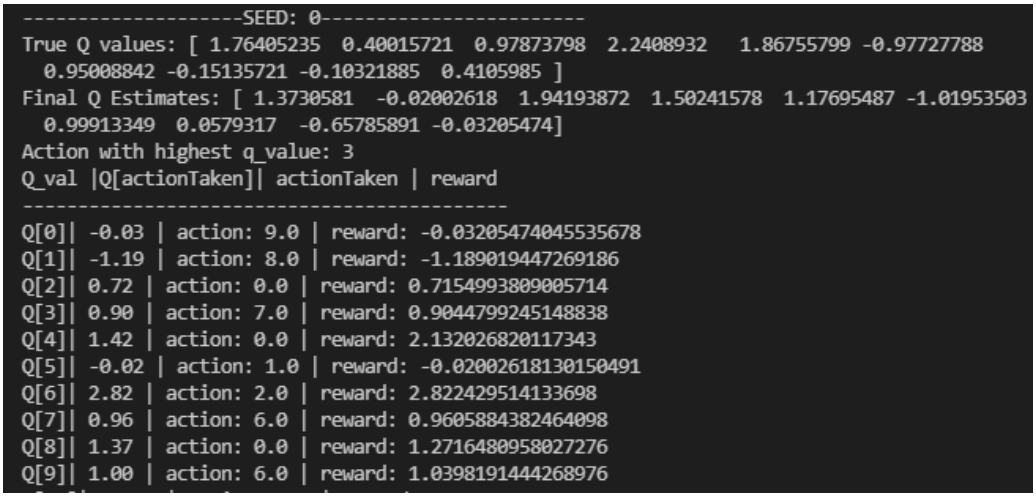


Figure 10: Output from Pure Exploration function on Ten-Armed Gaussian Bandit.

(c) Epsilon Greedy agent converges to the optimal action by estimating its Q value and keeps exploiting it without trying to estimate other Q values depending upon the value of epsilon.
In Fig. 11 half of the action value estimates along with their true action values has been plotted. We can see that the action-value for the optimal action has been estimated and the agent has not explored rest of the actions much.
Note: Seed value = 0, $\sigma = 1$, $\epsilon = 0.1$ and file name is *ques1_part5_3c_epsilonGreedy.py*.

(d) Decaying epsilon greedy agent balances exploration vs exploitation much better and converges to the optimal values.

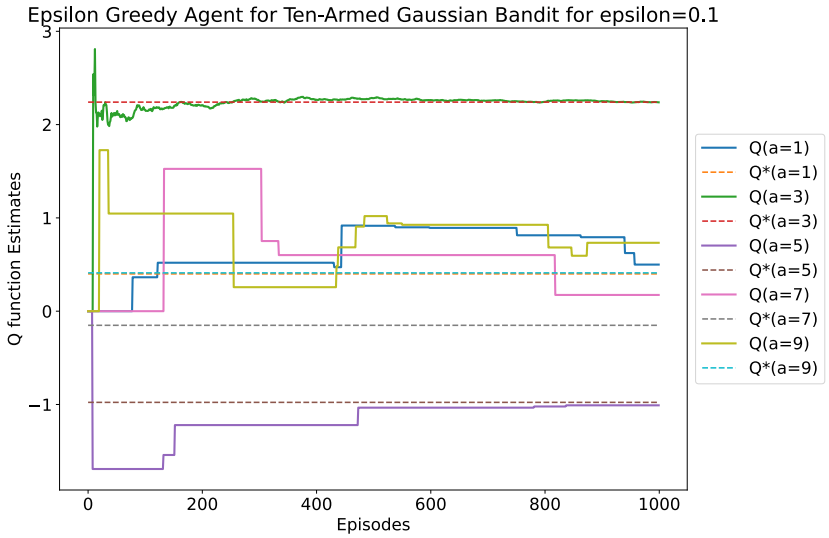


Figure 11: Q value estimated by epsilon greedy agent. Q*(a=k) represents the true action value.

In Fig. 12 we can see the agent's performance for linear and exponential decay. Initial value of epsilon is $\epsilon_{max} = 1$ and we decay it till $\epsilon_{min} = 1e^{-6}$ over 500 episodes.
We can observe that exponential decay converges faster, but after 500 episodes (when the decay is over) both the agents perform similar.
Note: Seed value = 0, $\sigma = 1$, $\epsilon_{max} = 1$, $\epsilon_{min} = 1e^{-6}$, decayed till 500 episodes and file name is *ques1_part5_3d_DecayingEpsilonGreedy.py*.

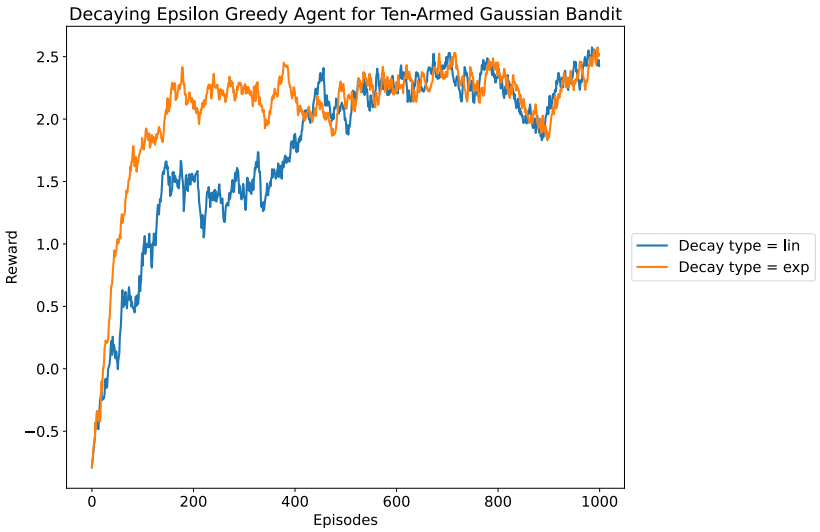


Figure 12: Reward achieved by Decaying Epsilon Greedy agent for two types of decay.

(e) Softmax is an optimistic strategy and assumes all actions are good until it has better knowledge about the distribution over Q values.
In Fig. 13 the reward achieved by Softmax agent for Ten-Armed Gaussian Bandit with two types of decay is plotted. Temperature parameter τ is decayed till 500 episodes after which both agents perform same.
Note: Seed value = 0, $\sigma = 1$, $\tau_{max} = 1e5$, $\tau_{min} = 0.005$, decayed till 500 episodes and file name is *ques1_part5_3e_softmax.py*.

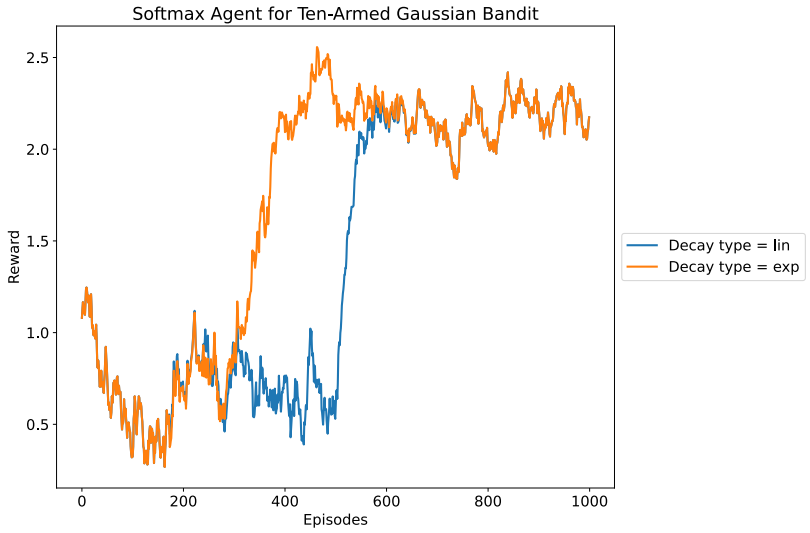


Figure 13: Reward achieved by Softmax agent for two types of decay. ($\tau_{max} = 1e5, \tau_{min} = 0.005$)

(f) UCB is one of the best agents for Bandit problems. It is also an optimistic agent and it converges really fast to the optimal values. It requires tuning of a hyper-parameter c which controls how much statistical knowledge dependent noise to add.
In Fig. 14 we can see UCB agent for four different values of c, and it performs best with $c = 1$.
Note: Seed value = 0, $\sigma = 1$, $c = \{0.1, 1, 5, 10\}$ and file name is *ques1_part5_3f_ucb.py*.

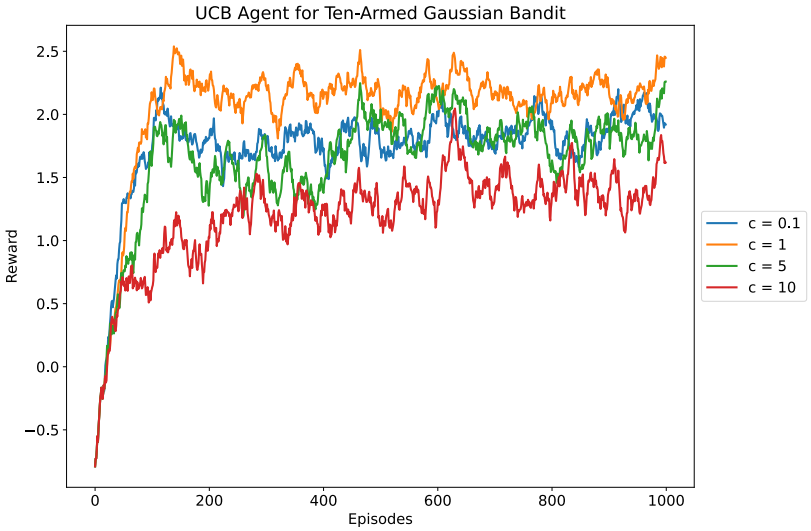


Figure 14: Reward achieved by UCB agent for different values of hyper-parameter c.

5 - (4) For this part we do the following to generate 50 environments and run 6 agents on each of the environment for 1000 time steps (episodes):

- SEED: vary seed values from 0 to 50 skipping 42 *(more references in python script)*.
- σ: We generate this using **numpy.random.uniform(0,2.5)** at every iteration.
- Env: We create a new instance of the environment using the above selected SEED and σ.
- Agent: We call all 6 agents to run over this environment.
- Plot: We average reward values over 50 environments and smooth out the resultant average with a smoothing window of 50.

Following are the hyper-parameters used:

- Epsilon Greedy: $\epsilon = 0.1$
- Decaying Epsilon: $\epsilon_{max} = 1, \epsilon_{min} = 1e^{-6}$ and decayed over 500 time steps, decay type = exponential.
- Softmax: $\tau_{max} = 100, \tau_{min} = 0.005$ and decayed over 500 time steps, decay type = exponential
- UCB: Value of $c = 1$

In Fig. 15 we can see that Softmax agent and UCB agent give the best rewards and converge to a higher value than others. Decaying Epsilon Greedy agent performs almost as well as the above two agents. Whereas Pure Exploitation gives sub-optimal rewards and Pure Exploration agent give minimal reward.

Note: File name is *ques1_part5_4.py*.

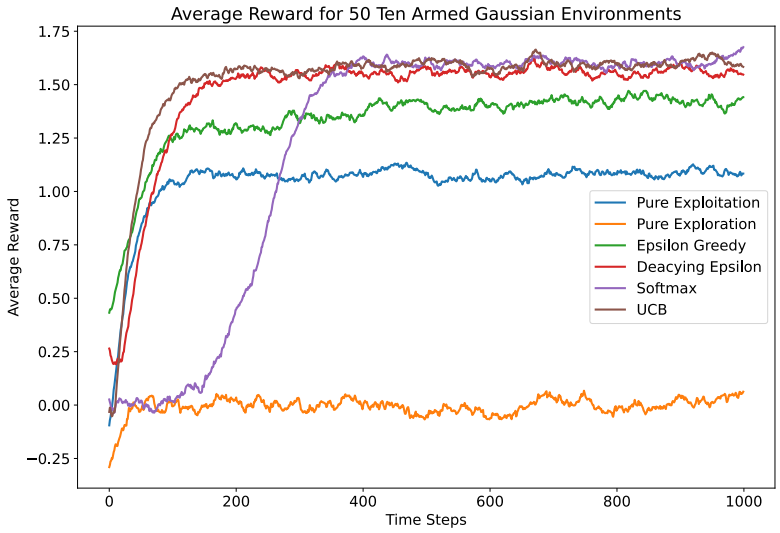


Figure 15: Performance of 6 different agents on Ten-Armed Gaussian Bandit.

6. For this part we do the following to generate 50 environments and run 6 agents on each of the environment for 1000 time steps (episodes):

   - SEED: vary seed values from 0 to 50 skipping 42 *(more references in python script)*.
   - α, β: We generate both these using **numpy.random.uniform(0,1)** at every iteration.
   - Env: We create a new instance of the environment using the above selected SEED and α, β.
   - Agent: We call all 6 agents to run over this environment.
   - Plot: We average regret values over 50 environments and smooth out the resultant average with a smoothing window of 50.

   Following are the hyper-parameters used:

   - Epsilon Greedy: $\epsilon = 0.1$
   - Decaying Epsilon: $\epsilon_{max} = 1, \epsilon_{min} = 1e^{-6}$ and decayed over 500 time steps, decay type = exponential.
   - Softmax: $\tau_{max} = 1e5, \tau_{min} = 0.005$ and decayed over 500 time steps, decay type = exponential
   - UCB: Value of $c = 0.1$

   Observations:

   - In Fig. 16 we can see that Pure Exploitation and Pure Exploration accumulate the most regret as they do not converge to the optimal values.

- Softmax agent decays its temperature parameter τ exponentially till 500 episodes after which it converges to the optimal and exploits the optimal action, hence its regret gets constant i.e., it stops accumulating more regret.
- For Decaying Epsilon greedy agent the ϵ is decayed exponentially till 500 episodes to a very low value $1e^{-6}$ and hence it also exploits the optimal action and it stops accumulating more regret.
- For Epsilon Greedy agent and UCB agent there is some amount of exploration always present due to constant ϵ in case of Epsilon greedy and the noise term in UCB goes to zero in the limit of episodes tending to infinity i.e., its exploration reduces as episodes increase. Hence, if run for more episodes, UCB will stop accumulating regret and epsilon greedy will keep accumulating some regret.

Note: File name is *ques1_part6.py*.

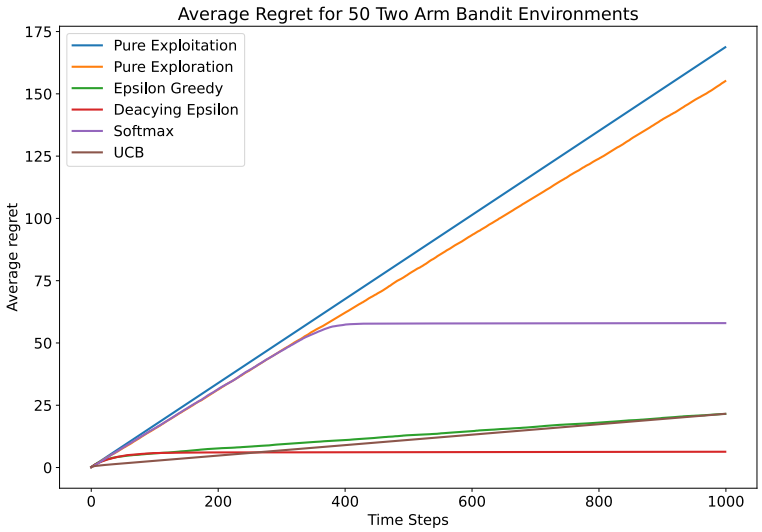


Figure 16: Regret of 6 different agents on Ten-Armed Gaussian Bandit.

7. For this part we do the following to generate 50 environments and run 6 agents on each of the environment for 1000 time steps (episodes):

   - SEED: vary seed values from 0 to 50 skipping 42 *(more references in python script)*.
   - σ: We generate this using **numpy.random.uniform(0,2.5)** at every iteration.
   - Env: We create a new instance of the environment using the above selected SEED and σ.
   - Agent: We call all 6 agents to run over this environment.
   - Plot: We average regret values over 50 environments and smooth out the resultant average with a smoothing window of 50.

   Following are the hyper-parameters used:

   - Epsilon Greedy: $\epsilon = 0.1$
   - Decaying Epsilon: $\epsilon_{max} = 1, \epsilon_{min} = 1e^{-6}$ and decayed over 500 time steps, decay type = exponential.
   - Softmax: $\tau_{max} = 100, \tau_{min} = 0.005$ and decayed over 500 time steps, decay type = exponential
   - UCB: Value of $c = 1$

   Observations:

   - In Fig. 16 we can see that Pure Exploration accumulates the most regret as it does not converge to the optimal values, whereas Pure Exploitation converges to a sub-optimal and hence keeps accumulating regret but it is lower than Pure Exploration.

- Softmax agent decays its temperature parameter τ exponentially till 500 episodes after which it converges to the optimal and exploits the optimal action, hence its regret gets constant i.e., it stops accumulating more regret.
- UCB agent accumulates the least regret and stops accumulating anymore the earliest.
- For Decaying Epsilon greedy agent the ϵ is decayed exponentially till 500 episodes to a very low value $1e^{-6}$ and hence it also exploits the optimal action and it stops accumulating more regret.
- For Epsilon Greedy agent there is some amount of exploration always present due to constant ϵ. Hence, epsilon greedy will keep accumulating some regret.

UCB is the best agent for this environment. Note: File name is *ques1_part7.py*.

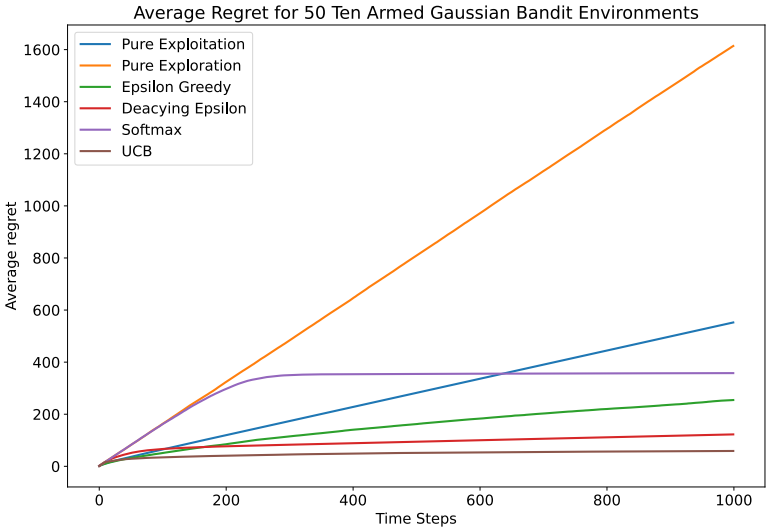


Figure 17: Regret of 6 different agents on Ten-Armed Gaussian Bandit.

8. For this part we do the following to generate 50 environments and run 6 agents on each of the environment for 2000 time steps (episodes):

   - SEED: vary seed values from 0 to 50 skipping 42 *(more references in python script)*.
   - α, β: We generate both these using **numpy.random.uniform(0,1)** at every iteration.
   - Env: We create a new instance of the environment using the above selected SEED and α, β.
   - Agent: We call all 6 agents to run over this environment.
   - Plot: We average percentage of optimal actions taken over 50 environments and smooth out the resultant average with a smoothing window of 50.

   Following are the hyper-parameters used:

   - Epsilon Greedy: $\epsilon = 0.1$
   - Decaying Epsilon: $\epsilon_{max} = 1, \epsilon_{min} = 1e^{-6}$ and decayed over 500 time steps, decay type = exponential.
   - Softmax: $\tau_{max} = 100, \tau_{min} = 0.005$ and decayed over 333 time steps, decay type = exponential
   - UCB: Value of $c = 0.1$

   Observations:

   - In Fig. 18 we can see that Decaying Epsilon (decayed over 500 episodes) gives most percentage of optimal actions for Two-Armed Bandit. It even outperforms UCB.
   - UCB agent converges the fastest and constantly gives high percentage of optimal actions.

- Softmax agents starts giving more optimal actions after its exponential temperature decay finishes at 500 episodes and tries to reach at par with Decayin Epsilon agent.
- Epsilon greedy agent has constant exploration present and slowly reaches at par with UCB agent.
- Pure Exploitation converges to a sub-optimal and constantly gives the same percentage of optimal action.
- Pure Exploration performs the worst as it doesn't converges.

Hence, we can say that for this environment and agent configuration Decaying Epsilon Greedy gives most percentage of optimal actions.

Note: File name is *ques1_part8.py*.

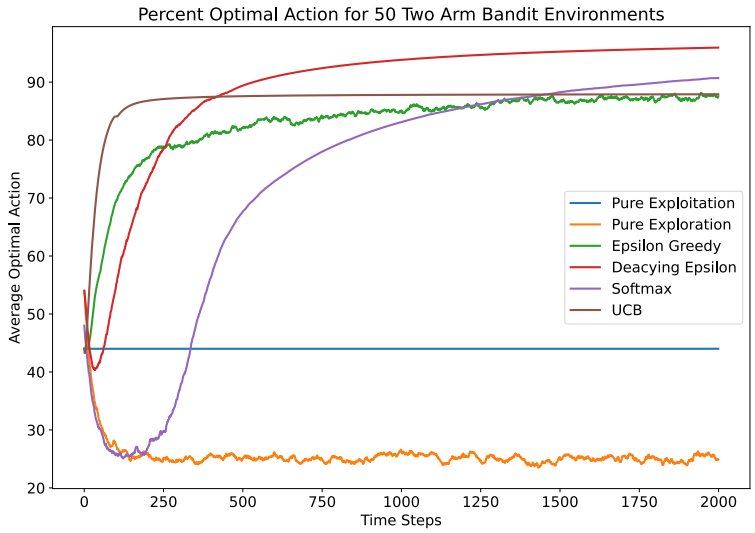


Figure 18: Percent Optimal Actions taken by 6 different agents on Ten-Armed Gaussian Bandit.

9. For this part we do the following to generate 50 environments and run 6 agents on each of the environment for 2000 time steps (episodes):

   - SEED: vary seed values from 0 to 50 skipping 42 *(more references in python script)*.
   - σ: We generate this using **numpy.random.uniform(0,2.5)** at every iteration.
   - Env: We create a new instance of the environment using the above selected SEED and σ.
   - Agent: We call all 6 agents to run over this environment.
   - Plot: We average percentage of optimal actions taken over 50 environments and smooth out the resultant average with a smoothing window of 50.

   Following are the hyper-parameters used:

   - Epsilon Greedy: $\epsilon = 0.1$
   - Decaying Epsilon: $\epsilon_{max} = 1, \epsilon_{min} = 1e^{-6}$ and decayed over 1000 time steps, decay type = exponential.
   - Softmax: $\tau_{max} = 100, \tau_{min} = 0.005$ and decayed over 500 time steps, decay type = exponential
   - UCB: Value of $c = 1$

   Observations:

   - In Fig. 19 we observe that Decaying Epsilon, UCB and Softmax agents reach similar percentage optimal action values.

- Decaying Epsilon decays to $1e^{-6}$ exponentially till 1000 episodes and starts giving highest percentage of optimal actions.
- UCB agent starts out better than Decaying Epsilon but ultimately settles down just below it.
- Softmax decays temperature τ from 100 to 0.005 till 500 episodes exponentially after which it shoots up and reaches at par with Decaying Epsilon agent.
- Pure Exploitation is sub-optimal and Pure Exploration doesn't converge to anything.

For this environment UCB and Decaying Epsilon seems to perform better than the rest. Note: File name is *ques1_part9.py*.

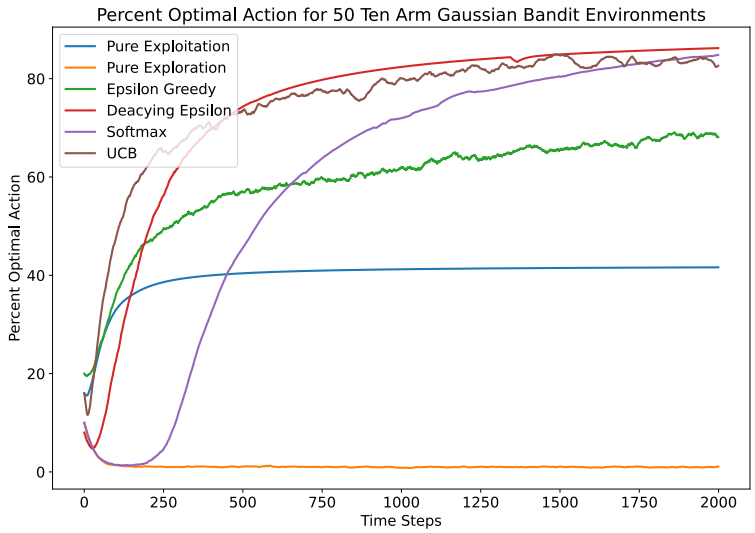


Figure 19: Regret of 6 different agents on Ten-Armed Gaussian Bandit.

Solution to Problem 2: MC Estimates and TD Learning

1. Generate Trajectory function is implemented as said and a random policy is utilized for generating trajectories for different seed values as can be seen in Fig 20.

   Trajectory contains experience tuples of the form (state, action, next state, reward, done).

   We also observe that for seed=3 the function returns nothing which means that the episode did not terminate for seed=3. Note: File name is *ques2_part1.py*.

```
----------------Generate Trajectory--------------------
Seed Value = 0
(2, 0, 3, 0, False)(3, 1, 4, 0, False)(4, 1, 5, 0, False)(5, 1, 6, 0, False)(6, 1, 6, 0, True)
Seed Value = 1
(1, 0, 0, 0, True)
Seed Value = 2
(1, 1, 2, 0, False)(2, 0, 1, 0, False)(1, 1, 0, 0, False)(0, 1, 0, 0, True)
Seed Value = 3

Seed Value = 4
(3, 1, 2, 0, False)(2, 1, 3, 0, False)(3, 1, 4, 0, False)(4, 0, 5, 0, False)(5, 1, 4, 0, False)(4
```

Figure 20: Output of Generate Trajectory function for Random Walk.

2. A function to decay step size α is implemented as said and we can see the decay behaviour in Fig. 21

   Note: File name is *ques2_part2.py*.

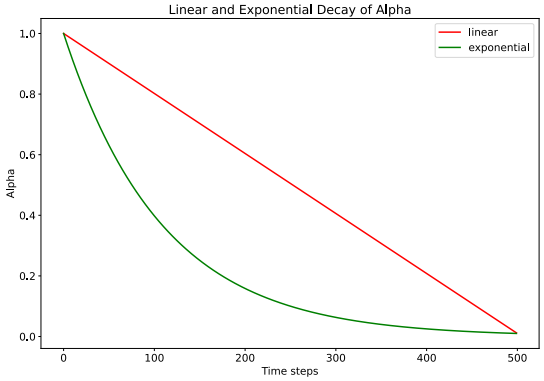


Figure 21: Step size α decay.

3. A function to implement Monte Carlo Prediction Algorithm is implemented as said. We test it for single instance of the environment with SEED = 0 and Always Go Left policy and the results are smoothed with a window size of 10.

   To calculate true value we implement Policy Evaluation function according to the slides and utilize the MDP dynamics data structure for Random Walk environment.

   In Fig. 22 we can see the behaviour of First Visit Monte Carlo and Every Visit Monte Carlo, we observe that Monte Carlo Prediction stabilize and try to converge to the true Value function as marked by black dashed lines.

   Note: File name is *ques2_part3.py*.

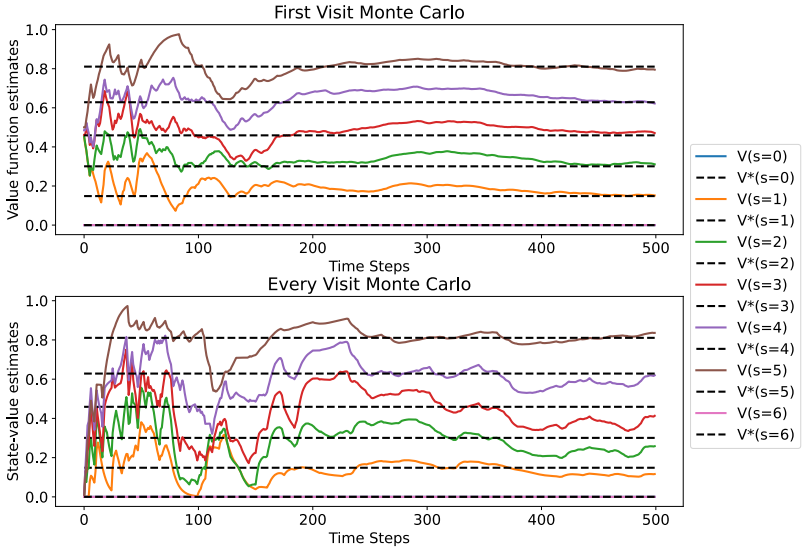


Figure 22: Monte Carlo Prediction Algorithm for FVMC and EVMC

4. A function to implement Temporal Difference Learning Algorithm is implemented as said. We test it for single instance of the environment with SEED = 0 and Always Go Left policy and the results are smoothed with a window size of 10.

   To calculate true value we implement Policy Evaluation function according to the slides and utilize the MDP dynamics data structure for Random Walk environment.

   In Fig. 23 we observe that TD estimates try to follow true value but not quite because TD estimates are biased.

   Note: File name is *ques2_part4.py*.

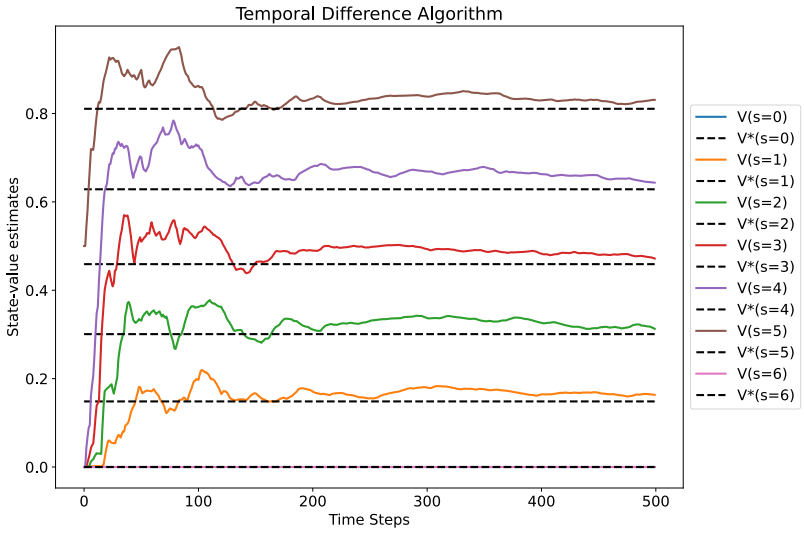


Figure 23: Value function estimates from TD Learning Algorithm

5. We do the following for the generation of the required plots for **Part 5, Part 6, Part 7**:

   - Total no of episodes = 500, max steps up to which to decay $\alpha = 250$, value of gamma $\gamma = 0.99$ and decay type for α is exponential.
   - SEED: We iterate for 50 times and vary seed values from 0 to 50 skipping 42 *(more references in python script)*.
   - Env: We generate a Random Walk environment at each iteration with above seed value.
   - True Value: We use a function for Policy Evaluation to evaluate true state values for an Always Go Left policy.
   - Alpha α: We fix final value of alpha as 0.01 and generate starting value of alpha as **numpy.random.uniform(final_alpha,1)**.
   - Average all state value estimates over 50 environments and plot.

   In Fig. 24 we can see the behaviour of First Visit Monte Carlo, it varies around the true value in the start but then converges to true value.

   Note: File name is *ques2_part5_part6.py*.

6. In Fig. 25 we can see the behaviour of Every Visit Monte Carlo, it varies a lot more than FVMC around the true value in the start but then converges to true value later.

   EVMC has higher variance than FVMC as in EVMC for consider every occurrence of some state s_k during the trajectory. Which means we update the estimates of s_k as many times as it appears in the trajectory. Each time we update it we are adding more and more variance. Hence, as opposed to only updating a state once in FVMC, in EVMC we update it every time it appears in a trajectory and due to this we end up adding more variance than FVMC.

   Note: File name is *ques2_part5_part6.py*.

7. In Fig. 26 we can see the behaviour of TD algorithm, its estimates are biased and hence don't vary about the true estimates, they either stay above or below the true estimates and reach towards the true value with more and more episodes.

   Moreover, as TD estimate of the Return has a lot less variance than that of Monte Carlo Prediction, the estimates as we can see don't vary as much as FVMC or EVMC.

   Note: File name is *ques2_part7.py*.

8. We follow part 5 and change the scale of x-axis to log as plotted in Fig. 27.

   Note: File name is *ques2_part8_part9.py*.

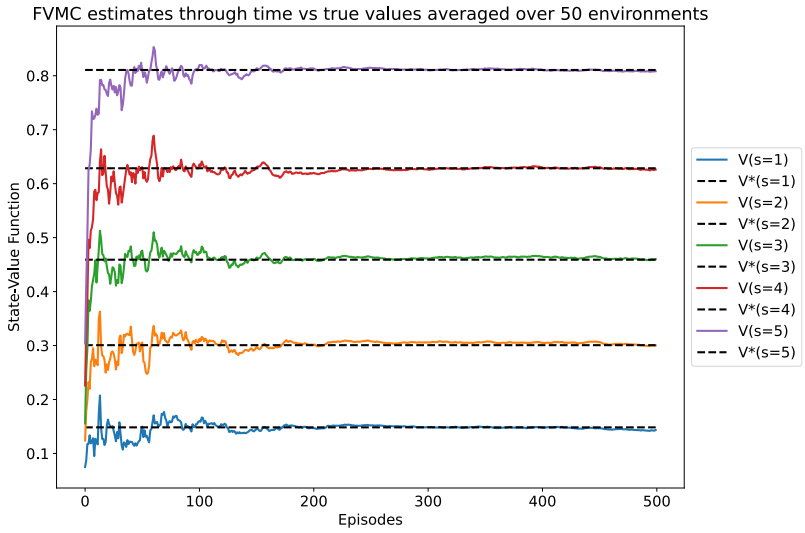


Figure 24: Monte Carlo Prediction Algorithm for FVMC

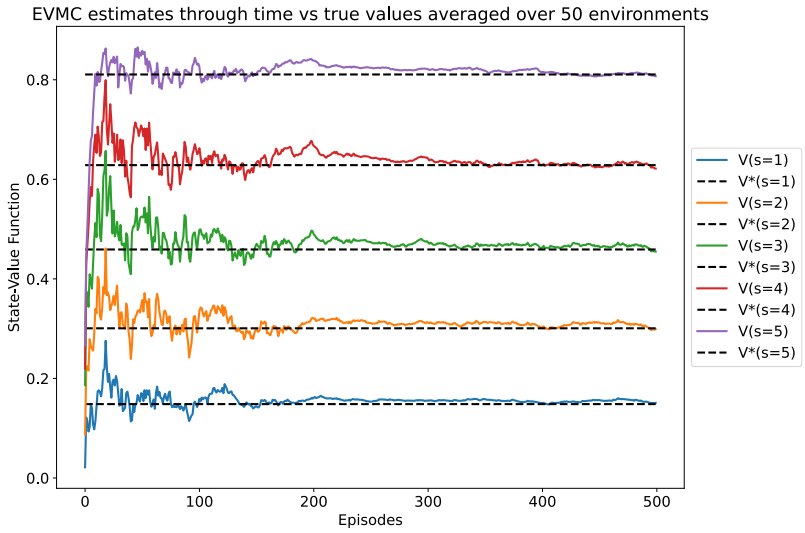


Figure 25: Monte Carlo Prediction Algorithm for EVMC

9. We follow part 6 and change the scale of x-axis to log as plotted in Fig. 28.

   Note: File name is *ques2_part8_part9.py*.

10. We follow part 7 and change the scale of x-axis to log as plotted in Fig. 29.

    Note: File name is *ques2_part10.py*.

11. From the above 6 parts and their plots we observe and conclude the following:

    - FVMC and EVMC are unbiased estimators of State value function as is clear from their values varying about the true state value and then settling down to the true state value towards the end in Figures 24, 25, 27, 28.
    - TD is a biased estimator of State value function as is clear from Figure 26 and 29. We can see that TD estimate values approach true state value either from above or below the true state value line and does not vary about the true state value as was happening for FVMC and EVMC.

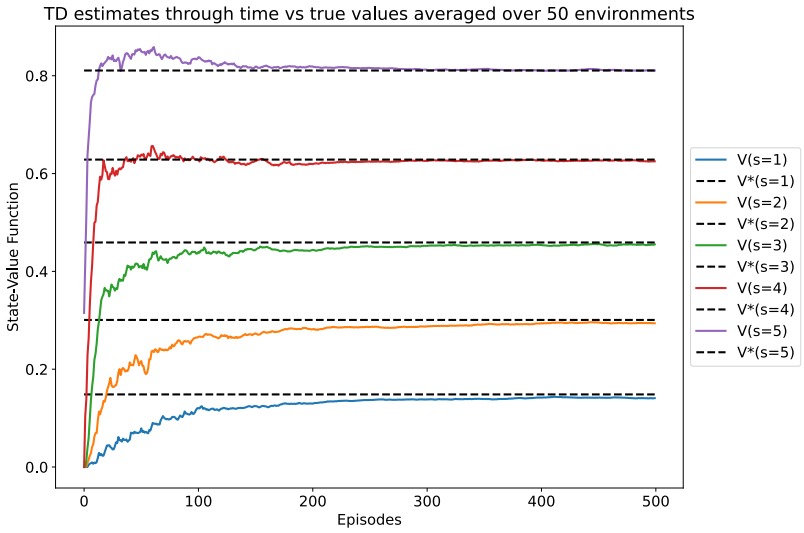


Figure 26: Temporal Difference Learning Algorithm

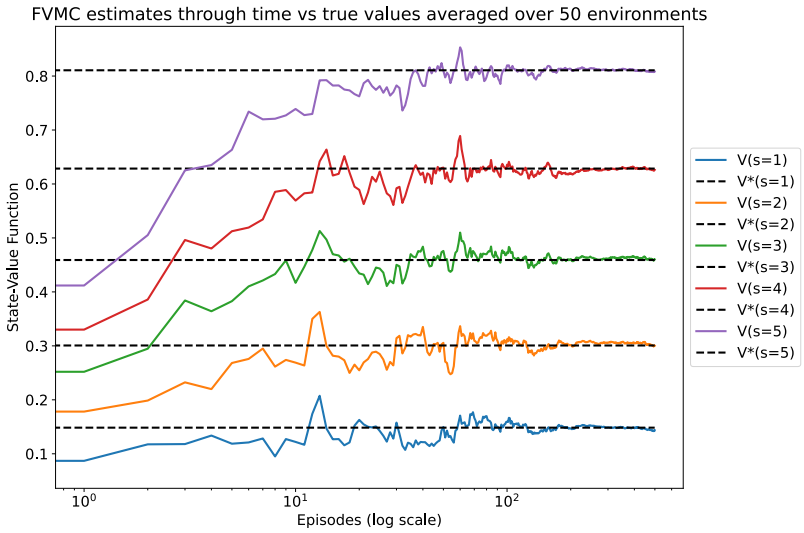


Figure 27: Monte Carlo Prediction Algorithm for FVMC

- TD uses an estimate of target return and hence incorporates a lot less variance as opposed to FVMC and EVMC which calculate the full return thus incorporating a lot more variance as the rewards themselves are random (Gaussian Distribution). This is also clear from the plots as TD estimates are varying less.
- EVMC has the highest variance. In EVMC we calculate the return for a state s_k every time it appears in the trajectory as opposed to only once as in FVMC. In doing so we incorporate the randomness of rewards (Gaussian Distribution) multiple times as opposed to doing so only once in FVMC and hence EVMC estimates vary a lot more than FVMC.

12. We follow part 5 and track the target value of state $s = 3$ for FVMC whenever it is encountered and the plot it along with the true value of state $s = 3$ in Fig. 30.

    We observe that the return G fluctuates between 0 and 1 for gamma $\gamma = 1$ as these are the only possibilities for Random Walk environment.

    For gamma $\gamma = 0.99$ we see that it discounts the future returns and depending upon how far we are from

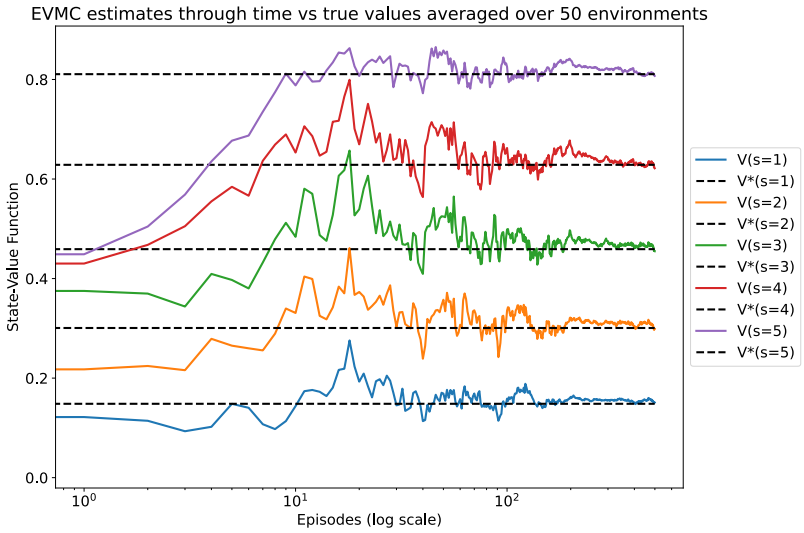


Figure 28: Monte Carlo Prediction Algorithm for EVMC

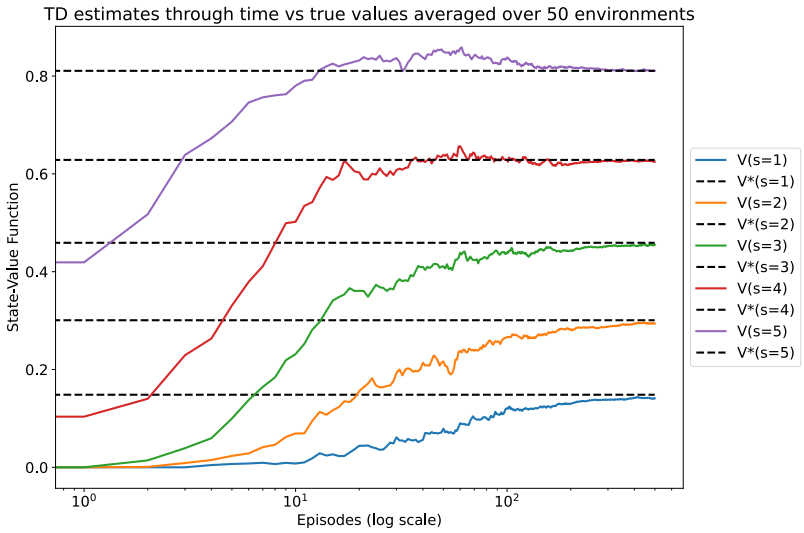


Figure 29: Temporal Difference Learning Algorithm

terminal state, the return $G = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$ varies. It is less if we are away and it is more if we are close to the terminal due to the discount γ^k.

Note: File name is *ques2_part12_part13.py*.

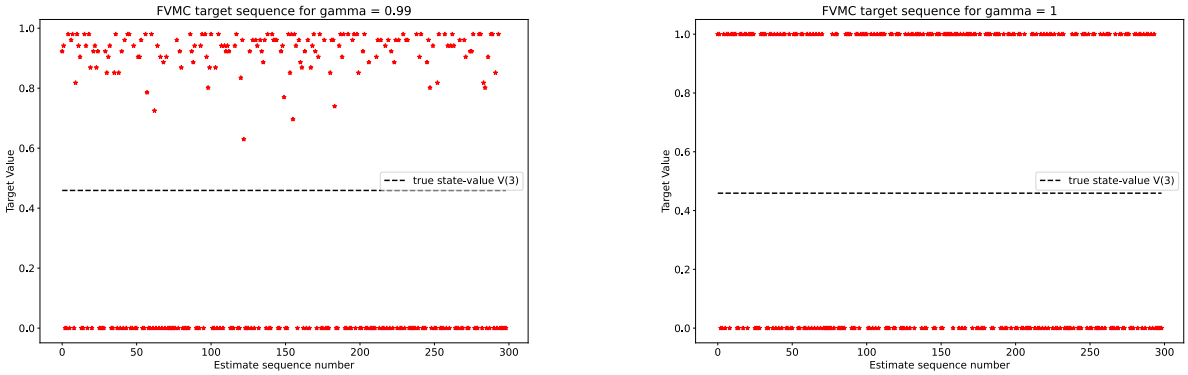
13. We follow part 6 and track the target value of state $s = 3$ for EVMC whenever it is encountered and the plot it along with the true value of state $s = 3$ in Fig. 31.

    For EVMC, a state s is considered as many times it appears in the trajectory, we track its target value as many times as it appears hence the sequence of target value for EVMC is longer than that of FVMC.

    And the rest of the behaviour of the return G is similar to FVMC.

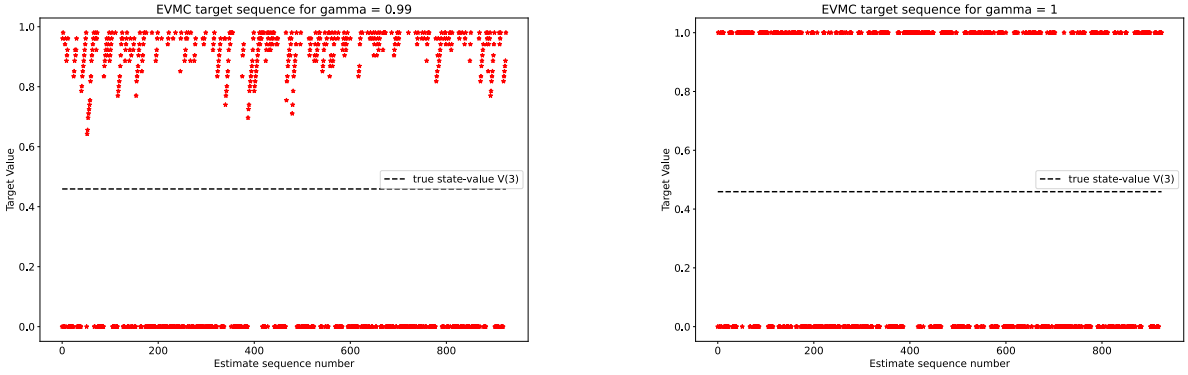
    Note: File name is *ques2_part12_part13.py*.

14. We follow part 7 and track the target value of state $s = 3$ for TD whenever it is encountered and the plot it along with the true value of state $s = 3$ in Fig. 32.



(a) $\gamma = 0.99$ (b) $\gamma = 1$

Figure 30: Target value sequence for FVMC for state $s = 3$

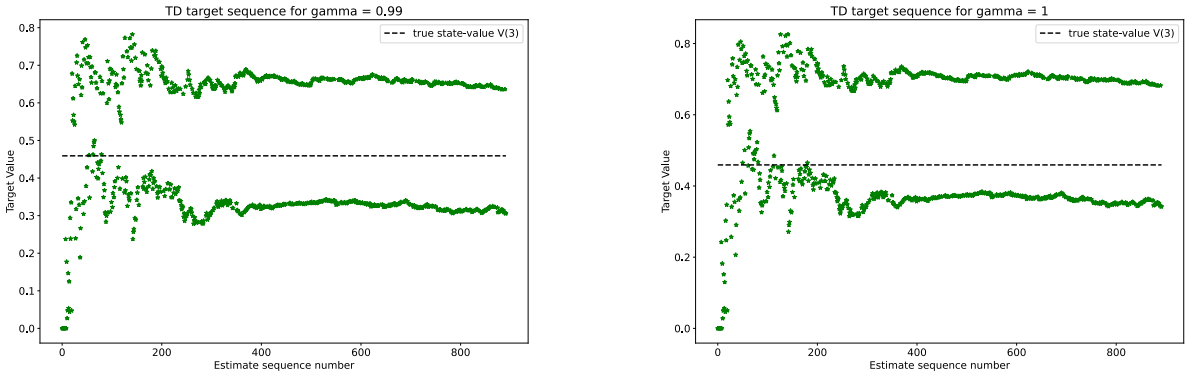


(a) $\gamma = 0.99$ (b) $\gamma = 1$

Figure 31: Target value sequence for EVMC for state $s = 3$

In TD we use estimate of the State value function to calculate the return G and hence the estimate of G does not fluctuates between 0 or 1. It varies between 0 and 1 in start and then converges to some value towards the end.

Note: File name is *ques2_part14.py*.



(a) $\gamma = 0.99$ (b) $\gamma = 1$

Figure 32: Target value sequence for TD for state $s = 3$

15. From above 3 parts and their plots we observe and conclude the following:

    - For Monte Carlo Prediction methods, the target return G fluctuates between 0 and 1 for gamma $\gamma = 1$. It sees some trajectory directly as giving some reward or no reward. For gamma $\gamma = 0.99$ we observe the discounting behaviour of the return for state $s = 3$ as is visible from the plots for $\gamma = 0.99$. Which means that if we are at state $s = 3$ and the rest of the trajectory took a long time to reach terminal state then there will be more discount due to the decay of gamma and vice versa.
    - For EVMC the target sequence is longer than that of FVMC, this is because in EVMC for every occurrence of state s in a trajectory its target is calculated and the estimates are updated as opposed to doing so only once in FVMC.
    - In TD we use an estimate of return which depends upon the estimate of state value itself and hence here the target varies between 0 and 1 and then as state value estimate stabilizes the return also stabilizes.