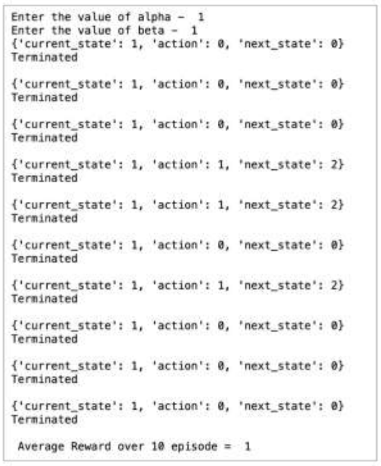
CS780: Deep Reinforcement Learning

Assignment #1

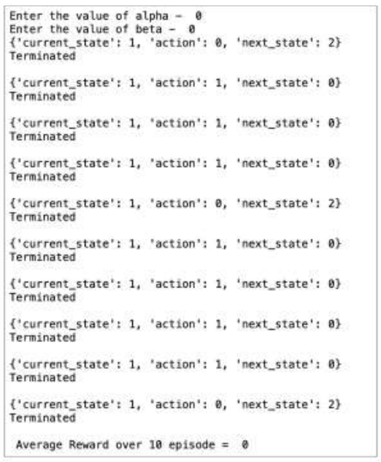
Name: Kislay Aditya Oj
Roll NO.: 210524

Solution to Problem 1: Multi-armed Bandits

1. For the given Bernoulli Bandit we can intuitively see that we will get a reward if and only if the agent take the exact step given by the action i.e. the agent won't get any reward if it slips.
   From this we can interpret that, for alpha = 1.0 and beta = 1.0 , there is 100% probability of getting a reward of +1. This is confirmed in the following snippet where we took alpha and beta to be 1 .



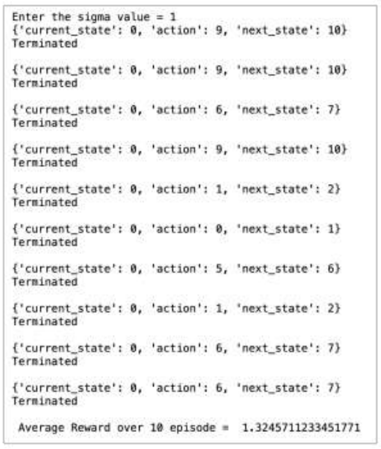
```
Enter the value of alpha -  1
Enter the value of beta -  1
{'current_state': 1, 'action': 0, 'next_state': 0}
Terminated

{'current_state': 1, 'action': 0, 'next_state': 0}
Terminated

{'current_state': 1, 'action': 0, 'next_state': 0}
Terminated

{'current_state': 1, 'action': 1, 'next_state': 2}
Terminated

{'current_state': 1, 'action': 1, 'next_state': 2}
Terminated

{'current_state': 1, 'action': 0, 'next_state': 0}
Terminated

{'current_state': 1, 'action': 1, 'next_state': 2}
Terminated

{'current_state': 1, 'action': 0, 'next_state': 0}
Terminated

{'current_state': 1, 'action': 0, 'next_state': 0}
Terminated

{'current_state': 1, 'action': 0, 'next_state': 0}
Terminated

 Average Reward over 10 episode =  1
```



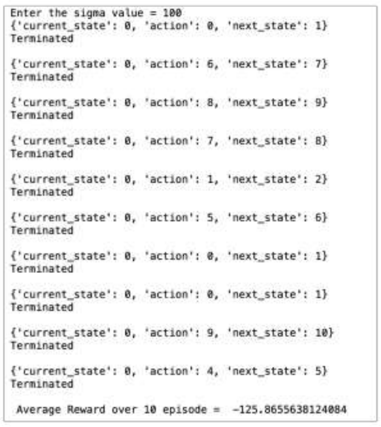
```
Enter the value of alpha -  0
Enter the value of beta -  0
{'current_state': 1, 'action': 0, 'next_state': 2}
Terminated

{'current_state': 1, 'action': 1, 'next_state': 0}
Terminated

{'current_state': 1, 'action': 1, 'next_state': 0}
Terminated

{'current_state': 1, 'action': 1, 'next_state': 0}
Terminated

{'current_state': 1, 'action': 0, 'next_state': 2}
Terminated

{'current_state': 1, 'action': 1, 'next_state': 0}
Terminated

{'current_state': 1, 'action': 1, 'next_state': 0}
Terminated

{'current_state': 1, 'action': 1, 'next_state': 0}
Terminated

{'current_state': 1, 'action': 1, 'next_state': 0}
Terminated

{'current_state': 1, 'action': 0, 'next_state': 2}
Terminated

 Average Reward over 10 episode =  0
```

   Also , for alpha and beta = 0 , we can clearly see that the average reward over 10 episodes is 0 , which further confirms the postulate. (Note that the policy used here is completely random i.e. the action will be selected from the action space randomly)

2. In the given Bernoulli Bandit , this time the rewards are stochastic i.e. the reward value are completely random and depends only upon the standard deviation chosen. For a sigma value of 1 the average reward often comes between -1 and 1 , but for larger values such as sigma = 100 , the average rewards varies a lot. Here are two examples from the code having different sigma values.



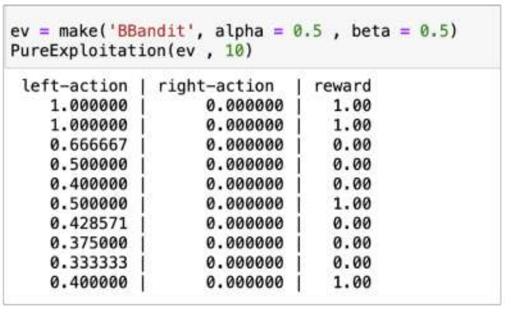
```
Enter the sigma value = 1
{'current_state': 0, 'action': 9, 'next_state': 10}
Terminated

{'current_state': 0, 'action': 9, 'next_state': 10}
Terminated

{'current_state': 0, 'action': 6, 'next_state': 7}
Terminated

{'current_state': 0, 'action': 9, 'next_state': 10}
Terminated

{'current_state': 0, 'action': 1, 'next_state': 2}
Terminated

{'current_state': 0, 'action': 0, 'next_state': 1}
Terminated

{'current_state': 0, 'action': 5, 'next_state': 6}
Terminated

{'current_state': 0, 'action': 1, 'next_state': 2}
Terminated

{'current_state': 0, 'action': 6, 'next_state': 7}
Terminated

{'current_state': 0, 'action': 6, 'next_state': 7}
Terminated

 Average Reward over 10 episode =  1.3245711233451771
```



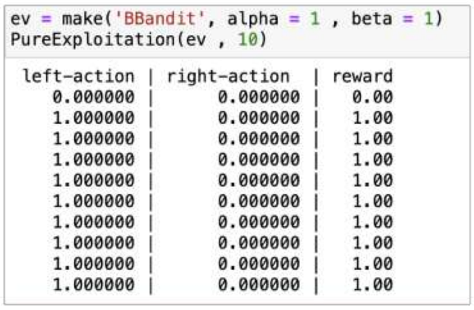
```
Enter the sigma value = 100
{'current_state': 0, 'action': 0, 'next_state': 1}
Terminated

{'current_state': 0, 'action': 6, 'next_state': 7}
Terminated

{'current_state': 0, 'action': 8, 'next_state': 9}
Terminated

{'current_state': 0, 'action': 7, 'next_state': 8}
Terminated

{'current_state': 0, 'action': 1, 'next_state': 2}
Terminated

{'current_state': 0, 'action': 5, 'next_state': 6}
Terminated

{'current_state': 0, 'action': 0, 'next_state': 1}
Terminated

{'current_state': 0, 'action': 0, 'next_state': 1}
Terminated

{'current_state': 0, 'action': 9, 'next_state': 10}
Terminated

{'current_state': 0, 'action': 4, 'next_state': 5}
Terminated

 Average Reward over 10 episode =  -125.8655638124084
```

Additionally , we can see that this environment is quite random and generates different rewards based on different values of sigma.

3. (a) **Pure - Exploitation :**
For alpha , beta pairs we got :



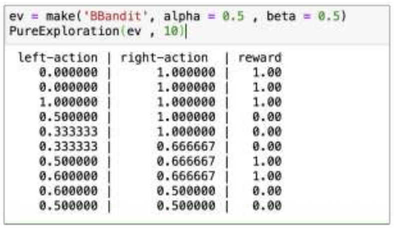
```
ev = make('BBandit', alpha = 0.5 , beta = 0.5)
PureExploitation(ev , 10)

 left-action | right-action  | reward
    1.000000 |      0.000000 |   1.00
    1.000000 |      0.000000 |   1.00
    0.666667 |      0.000000 |   0.00
    0.500000 |      0.000000 |   0.00
    0.400000 |      0.000000 |   0.00
    0.500000 |      0.000000 |   1.00
    0.428571 |      0.000000 |   0.00
    0.375000 |      0.000000 |   0.00
    0.333333 |      0.000000 |   0.00
    0.400000 |      0.000000 |   1.00
```



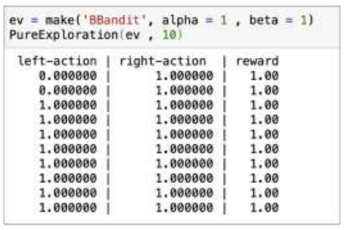
```
ev = make('BBandit', alpha = 1 , beta = 1)
PureExploitation(ev , 10)

 left-action | right-action  | reward
    0.000000 |      0.000000 |   0.00
    1.000000 |      0.000000 |   1.00
    1.000000 |      0.000000 |   1.00
    1.000000 |      0.000000 |   1.00
    1.000000 |      0.000000 |   1.00
    1.000000 |      0.000000 |   1.00
    1.000000 |      0.000000 |   1.00
    1.000000 |      0.000000 |   1.00
    1.000000 |      0.000000 |   1.00
    1.000000 |      0.000000 |   1.00
```

As expected , the agent goes for greedy strategy , once it got a reward it only stuck to that particular action .

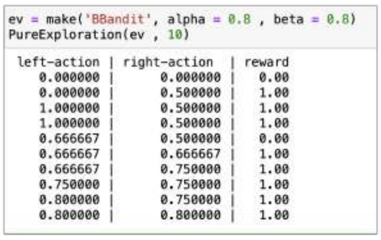
(b) **Pure - Exploration :**
For alpha , beta pairs we got :



```
ev = make('BBandit', alpha = 0.5 , beta = 0.5)
PureExploration(ev , 10)

 left-action | right-action  | reward
    0.000000 |      1.000000 |   1.00
    0.000000 |      1.000000 |   1.00
    1.000000 |      1.000000 |   1.00
    0.500000 |      1.000000 |   0.00
    0.333333 |      1.000000 |   0.00
    0.333333 |      0.666667 |   0.00
    0.500000 |      0.666667 |   1.00
    0.600000 |      0.666667 |   1.00
    0.600000 |      0.500000 |   0.00
    0.500000 |      0.500000 |   0.00
```



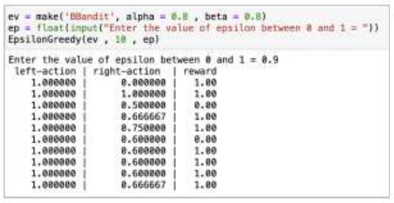
```
ev = make('BBandit', alpha = 1 , beta = 1)
PureExploration(ev , 10)

 left-action | right-action  | reward
    0.000000 |      1.000000 |   1.00
    0.000000 |      1.000000 |   1.00
    1.000000 |      1.000000 |   1.00
    1.000000 |      1.000000 |   1.00
    1.000000 |      1.000000 |   1.00
    1.000000 |      1.000000 |   1.00
    1.000000 |      1.000000 |   1.00
    1.000000 |      1.000000 |   1.00
    1.000000 |      1.000000 |   1.00
    1.000000 |      1.000000 |   1.00
```



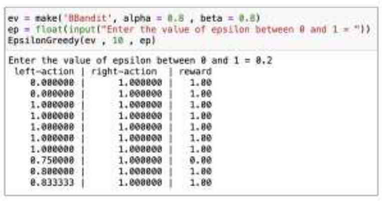
```
ev = make('BBandit', alpha = 0.8 , beta = 0.8)
PureExploration(ev , 10)

 left-action | right-action  | reward
    0.000000 |      0.000000 |   0.00
    0.000000 |      0.500000 |   1.00
    1.000000 |      0.500000 |   1.00
    1.000000 |      0.500000 |   1.00
    0.666667 |      0.500000 |   0.00
    0.666667 |      0.666667 |   1.00
    0.666667 |      0.750000 |   1.00
    0.750000 |      0.750000 |   1.00
    0.800000 |      0.750000 |   1.00
    0.800000 |      0.800000 |   1.00
```

As expected , the agent gave the preference to exploration and did both the actions to get a significantly better reward average than pure exploitation.

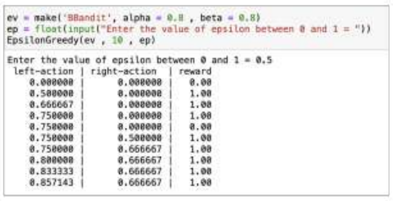
(c) **Epsilon-Greedy :**
For alpha , beta pairs we got :



```
ev = make('BBandit', alpha = 0.8 , beta = 0.8)
ep = float(input("Enter the value of epsilon between 0 and 1 = "))
EpsilonGreedy(ev , 10 , ep)

Enter the value of epsilon between 0 and 1 = 0.9
 left-action | right-action  | reward
    1.000000 |      0.000000 |   1.00
    1.000000 |      1.000000 |   1.00
    1.000000 |      0.500000 |   0.00
    1.000000 |      0.666667 |   1.00
    1.000000 |      0.750000 |   1.00
    1.000000 |      0.600000 |   0.00
    1.000000 |      0.600000 |   1.00
    1.000000 |      0.600000 |   1.00
    1.000000 |      0.600000 |   1.00
    1.000000 |      0.666667 |   1.00
```



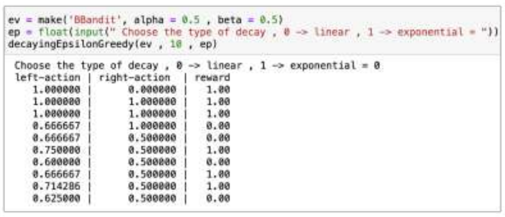
```
ev = make('BBandit', alpha = 0.8 , beta = 0.8)
ep = float(input("Enter the value of epsilon between 0 and 1 = "))
EpsilonGreedy(ev , 10 , ep)

Enter the value of epsilon between 0 and 1 = 0.2
 left-action | right-action  | reward
    0.000000 |      1.000000 |   1.00
    0.000000 |      1.000000 |   1.00
    1.000000 |      1.000000 |   1.00
    1.000000 |      1.000000 |   1.00
    1.000000 |      1.000000 |   1.00
    1.000000 |      1.000000 |   1.00
    1.000000 |      1.000000 |   1.00
    0.750000 |      1.000000 |   0.00
    0.800000 |      1.000000 |   1.00
    0.833333 |      1.000000 |   1.00
```



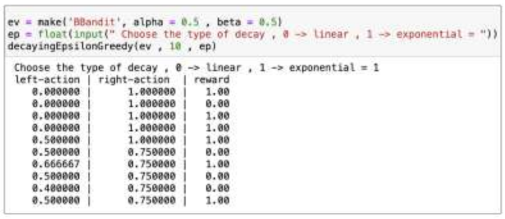
```
ev = make('BBandit', alpha = 0.8 , beta = 0.8)
ep = float(input("Enter the value of epsilon between 0 and 1 = "))
EpsilonGreedy(ev , 10 , ep)

Enter the value of epsilon between 0 and 1 = 0.5
 left-action | right-action  | reward
    0.000000 |      0.000000 |   0.00
    0.500000 |      0.000000 |   1.00
    0.666667 |      0.000000 |   1.00
    0.750000 |      0.000000 |   1.00
    0.750000 |      0.000000 |   0.00
    0.750000 |      0.500000 |   1.00
    0.750000 |      0.666667 |   1.00
    0.800000 |      0.666667 |   1.00
    0.833333 |      0.666667 |   1.00
    0.857143 |      0.666667 |   1.00
```

As we can see , the rewards depends upon the value of epsilon. More the value of epsilon more the agent will explore and less the value , the agent will exploit.

(d) **Decaying Epsilon-Greedy :**
For alpha , beta pairs we got :



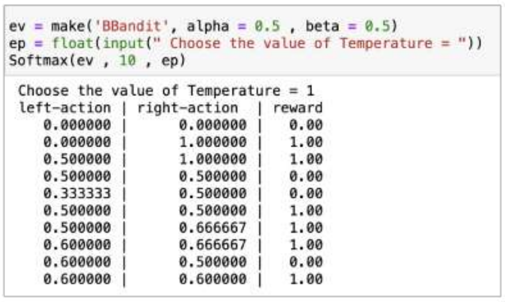
```
ev = make('BBandit', alpha = 0.5 , beta = 0.5)
ep = float(input(" Choose the type of decay , 0 -> linear , 1 -> exponential = "))
decayingEpsilonGreedy(ev , 10 , ep)

 Choose the type of decay , 0 -> linear , 1 -> exponential = 0
 left-action | right-action  | reward
    1.000000 |      0.000000 |   1.00
    1.000000 |      1.000000 |   1.00
    1.000000 |      1.000000 |   1.00
    0.666667 |      1.000000 |   0.00
    0.666667 |      0.500000 |   0.00
    0.750000 |      0.500000 |   1.00
    0.600000 |      0.500000 |   0.00
    0.666667 |      0.500000 |   1.00
    0.714286 |      0.500000 |   1.00
    0.625000 |      0.500000 |   0.00
```



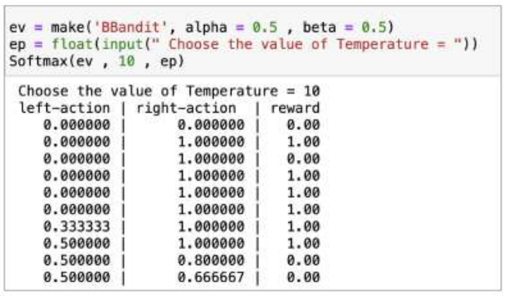
```
ev = make('BBandit', alpha = 0.5 , beta = 0.5)
ep = float(input(" Choose the type of decay , 0 -> linear , 1 -> exponential = "))
decayingEpsilonGreedy(ev , 10 , ep)

 Choose the type of decay , 0 -> linear , 1 -> exponential = 1
 left-action | right-action  | reward
    0.000000 |      1.000000 |   1.00
    0.000000 |      1.000000 |   0.00
    0.000000 |      1.000000 |   1.00
    0.000000 |      1.000000 |   1.00
    0.500000 |      1.000000 |   1.00
    0.500000 |      0.750000 |   0.00
    0.666667 |      0.750000 |   1.00
    0.500000 |      0.750000 |   0.00
    0.400000 |      0.750000 |   0.00
    0.500000 |      0.750000 |   1.00
```

This agent is kind of same as last , but instead of giving a single value we decrease the epsilon value to get a spectrum. We can either decrease linearly or exponentially as shown.

(e) **Soft-max :**
For alpha , beta pairs we got :



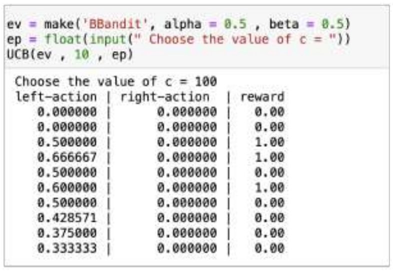
```
ev = make('BBandit', alpha = 0.5 , beta = 0.5)
ep = float(input(" Choose the value of Temperature = "))
Softmax(ev , 10 , ep)

 Choose the value of Temperature = 1
 left-action | right-action  | reward
    0.000000 |      0.000000 |   0.00
    0.000000 |      1.000000 |   1.00
    0.500000 |      1.000000 |   1.00
    0.500000 |      0.500000 |   0.00
    0.333333 |      0.500000 |   0.00
    0.500000 |      0.500000 |   1.00
    0.500000 |      0.666667 |   1.00
    0.600000 |      0.666667 |   1.00
    0.600000 |      0.500000 |   0.00
    0.600000 |      0.600000 |   1.00
```



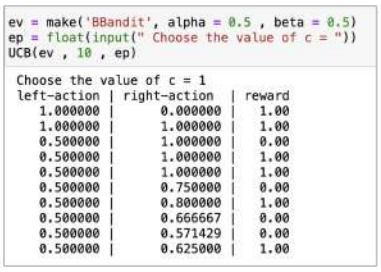
```
ev = make('BBandit', alpha = 0.5 , beta = 0.5)
ep = float(input(" Choose the value of Temperature = "))
Softmax(ev , 10 , ep)

 Choose the value of Temperature = 10
 left-action | right-action  | reward
    0.000000 |      0.000000 |   0.00
    0.000000 |      1.000000 |   1.00
    0.000000 |      1.000000 |   0.00
    0.000000 |      1.000000 |   1.00
    0.000000 |      1.000000 |   1.00
    0.000000 |      1.000000 |   1.00
    0.333333 |      1.000000 |   1.00
    0.500000 |      1.000000 |   1.00
    0.500000 |      0.800000 |   0.00
    0.500000 |      0.666667 |   0.00
```

Softmax is a strategic exploration policy where probability of selecting an option is proportional to current action-value estimate. The hyper parameter to be tuned here is temperature.

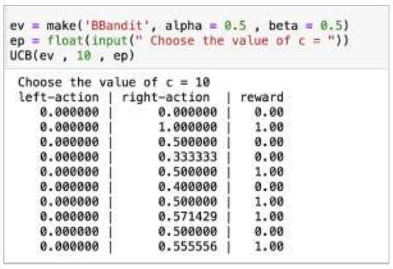
(f) **UCB :**
For alpha , beta pairs we got :



```
ev = make('BBandit', alpha = 0.5 , beta = 0.5)
ep = float(input(" Choose the value of c = "))
UCB(ev , 10 , ep)

 Choose the value of c = 100
 left-action | right-action  | reward
    0.000000 |      0.000000 |   0.00
    0.000000 |      0.000000 |   0.00
    0.500000 |      0.000000 |   1.00
    0.666667 |      0.000000 |   1.00
    0.500000 |      0.000000 |   0.00
    0.600000 |      0.000000 |   1.00
    0.500000 |      0.000000 |   0.00
    0.428571 |      0.000000 |   0.00
    0.375000 |      0.000000 |   0.00
    0.333333 |      0.000000 |   0.00
```



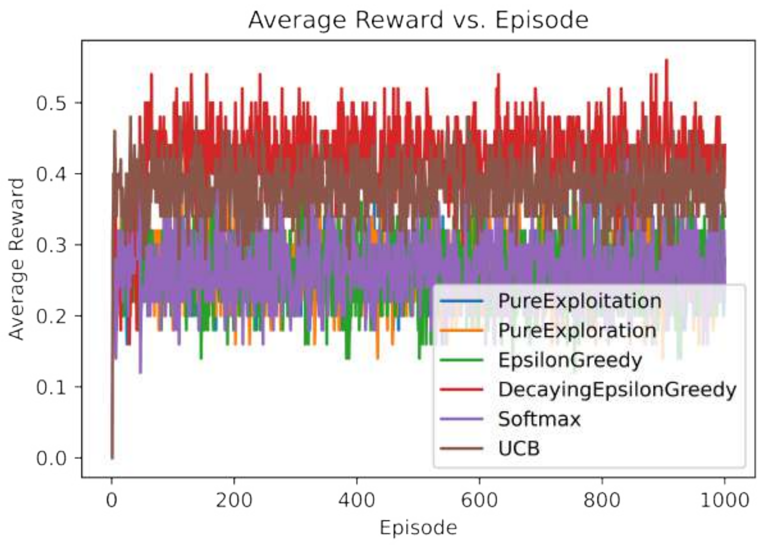
```
ev = make('BBandit', alpha = 0.5 , beta = 0.5)
ep = float(input(" Choose the value of c = "))
UCB(ev , 10 , ep)

 Choose the value of c = 1
 left-action | right-action  | reward
    1.000000 |      0.000000 |   1.00
    1.000000 |      1.000000 |   1.00
    0.500000 |      1.000000 |   0.00
    0.500000 |      1.000000 |   1.00
    0.500000 |      1.000000 |   1.00
    0.500000 |      0.750000 |   0.00
    0.500000 |      0.800000 |   1.00
    0.500000 |      0.666667 |   0.00
    0.500000 |      0.571429 |   0.00
    0.500000 |      0.625000 |   1.00
```



```
ev = make('BBandit', alpha = 0.5 , beta = 0.5)
ep = float(input(" Choose the value of c = "))
UCB(ev , 10 , ep)

 Choose the value of c = 10
 left-action | right-action  | reward
    0.000000 |      0.000000 |   0.00
    0.000000 |      1.000000 |   1.00
    0.000000 |      0.500000 |   0.00
    0.000000 |      0.333333 |   0.00
    0.000000 |      0.500000 |   1.00
    0.000000 |      0.400000 |   0.00
    0.000000 |      0.500000 |   1.00
    0.000000 |      0.571429 |   1.00
    0.000000 |      0.500000 |   0.00
    0.000000 |      0.555556 |   1.00
```

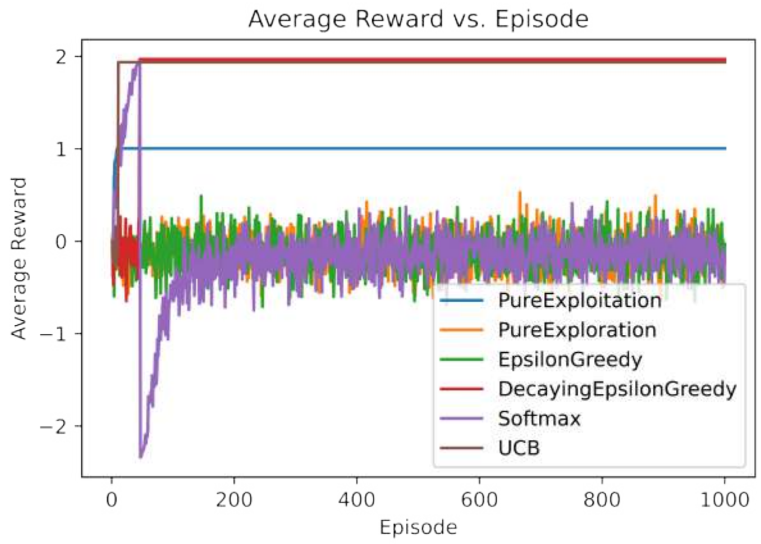
UCB or upper confidence bound is a technique which uses uncertainty as a bonus for exploration . We can tune the parameter c to get desired rewards.

4. From the plot we can clearly see that UCB and decaying epsilon is two of the best policy among the six . However , many other policies can also be tuned, but these two still remains on top. We can also observe here that Pure Exploitation works quite better than other strategies.

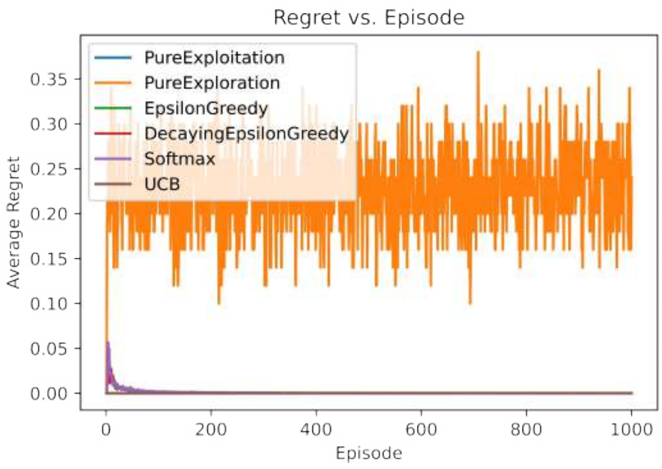


5. However in the 10 arm bandit problem , when the environment is completely random we see the clear difference between strategies. UCB and Decaying epsilon greedy is clearly the best with Pure exploitation following it. The Pure Exploitation policy works quite good in this environment because there are only

two actions to be taken with stocastic rewards on both of the actions. The softmax strategy has very high variance in initial episodes and took almost 200 episodes to converge. Other strategies are kind of comparable.



6. The Regret-Episode plot for the 2-armed bandit problem we got is

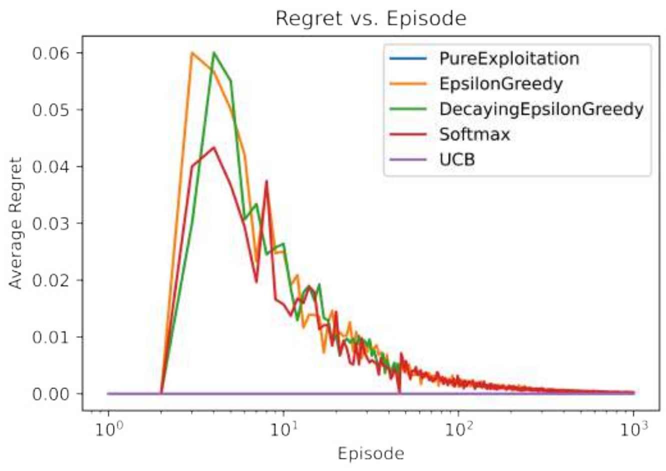


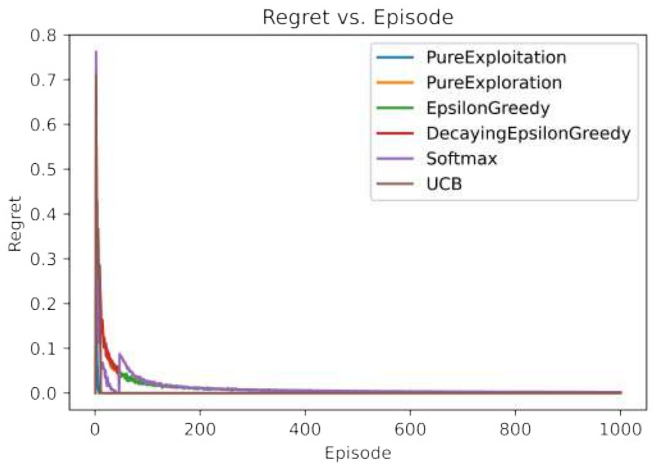
Looking at it once , it is clear that in most of the strategies , the regret converges to one as number of episodes increase , but in case of Pure Exploration the regret is quite high even after 1000 episodes. This shows that this strategy is not quite compatible with the 2 arm bandit environment.

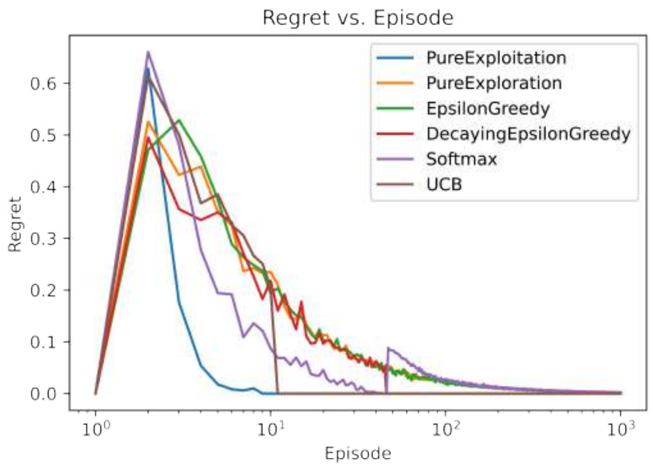
Now if we remove the Pure Exploration strategy and look at the plot at the logarithmic scale (top plot in next page) We can now compare different agents. At the first look we can see that as always UCB strategy works the best. The regret quickly converges to zero showing that it chooses the most optimal policy as the number of episodes increases. Other policies such as Epsilon Greedy (value of epsilon is equal to 0.5) , Decaying Epsilon greedy and Softmax function shows variable relationship. They increase at first and later decrease to zero. This also shows us that we need to run the environment for multiple episodes before arriving to any conclusion.

7. In the Above question we saw the plot for a 'predictable environment'. But in the case of 10-armed bandit , the environment is random. The plot is given by the middle figure in next page .

   For clarity the log axis is also plotted (bottom plot in next page)







In this case we see that almost all the agents follow the same trend , the regret in all case increases at first and over the episodes converges to zero . Also given that the reward is sampled from a gaussian distribution , the kind of trend is expected.

8. 9 and 10 .
   Here is the required plot for Average rewards vs episodes and Optimal Action percent vs episodes for both the environments.

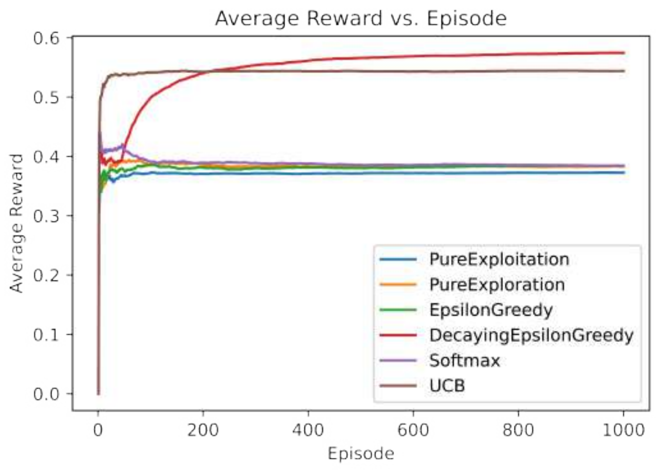


Figure 1: 2-Armed bandit environment

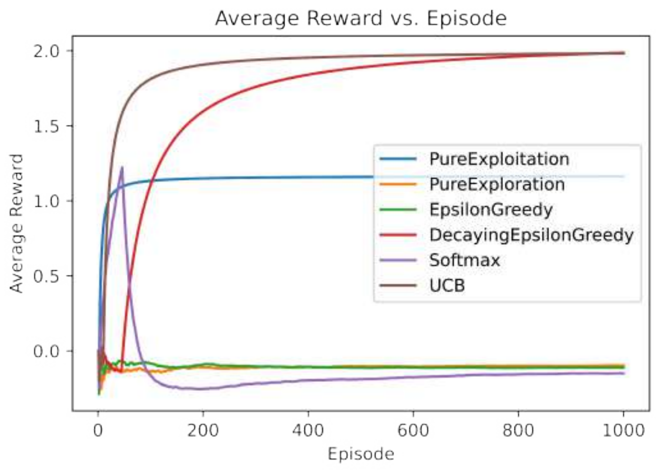


Figure 2: 10-Armed bandit environment

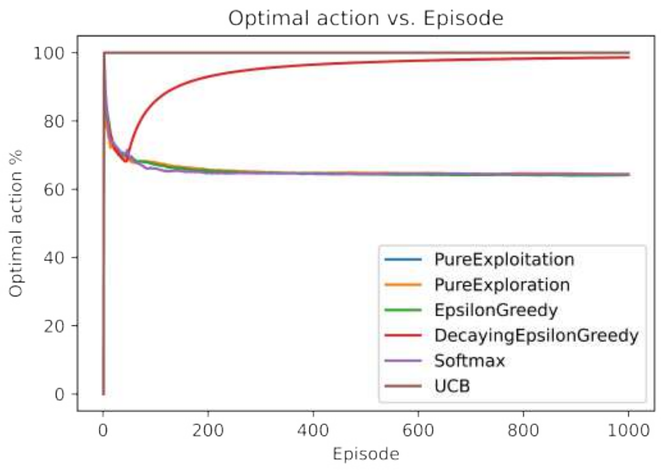


Figure 3: 2-Armed bandit environment (Normal scale)

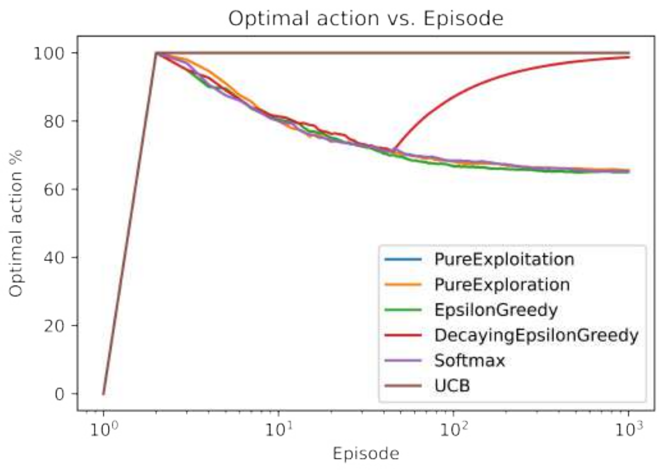


Figure 4: 2-Armed bandit environment(Log-scale)

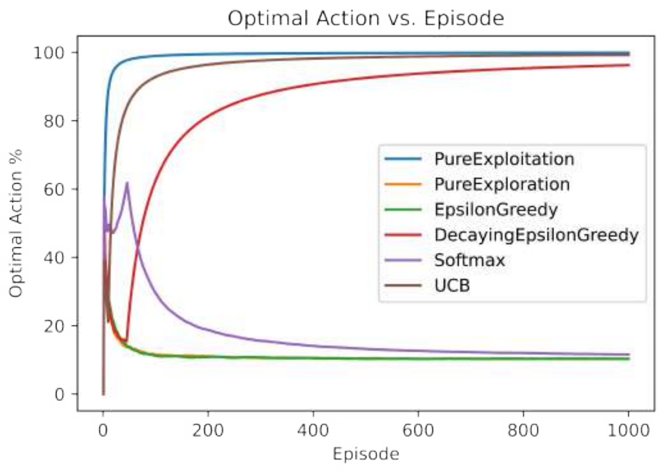


Figure 5: 10-Armed bandit environment (Normal scale)

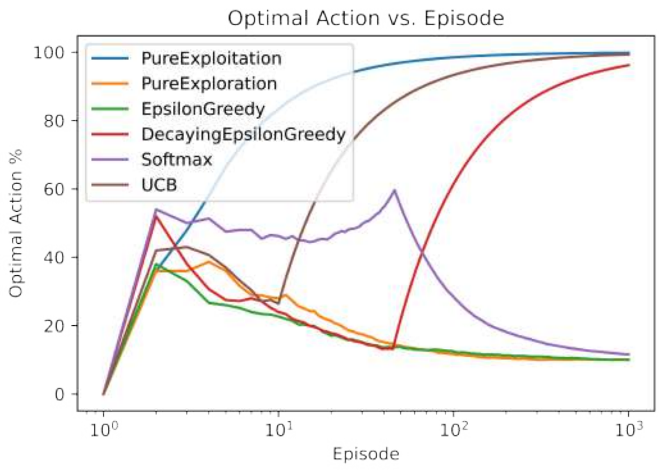


Figure 6: 10-Armed bandit environment(Log-scale)

Solution to Problem 2: MC Estimates and TD Learning

1. Implemented and tested successfully
2. Here's the plot for decay function :

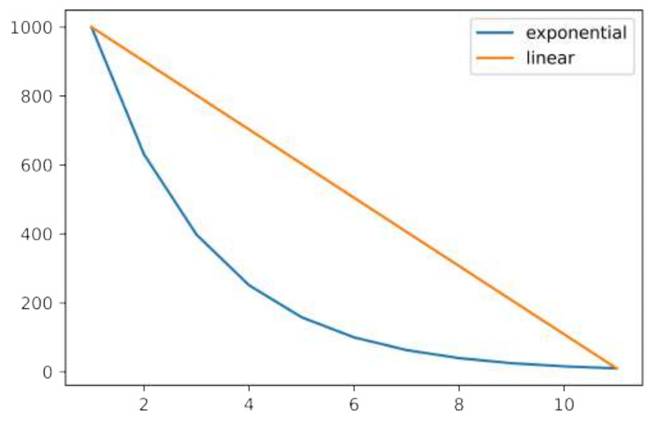


Figure 7: Exponential vs Linear decay

3. The Monte-Carlo estimate function was implemented and checked . It was working as expected. The state near the goal has higher value than others.Also , due to randomness of the environment all other state also had quite similar values . One thing which I found interesting was that the values somewhat depends upon where we started. For example , if we start near state 5 the value of state 5 will be a little bit more than if we started from state let's say 3. Also the policy was always go left which somewhat affected the values of a state.
4. The Temporal difference method gave better estimates of the states but the problem is , it depends on the reward of very next state, therefore in an environment like Random Walk where the reward is at the very end this takes a lot of episode to converge and kind of biased. Also the values of state highly depends upon where it started as closer the goal is better is the reward.
5. The plot is given by :

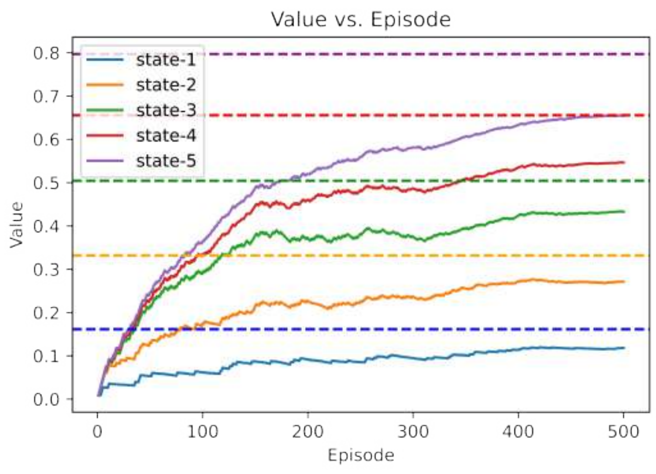


Figure 8: MC-FV estimates vs episodes

One thing to notice there is there is very less noise int the graph and the agents are taking long to converge. This is primarily because of the low alpha value taken . In this particular graph the alpha starts from 0.1 and decays to 0.001 which is pretty low considering normally we start from 1 or 0.5. Higher the value of alpha higher the noise we get in the graph.

6. The every visit plot is given by :

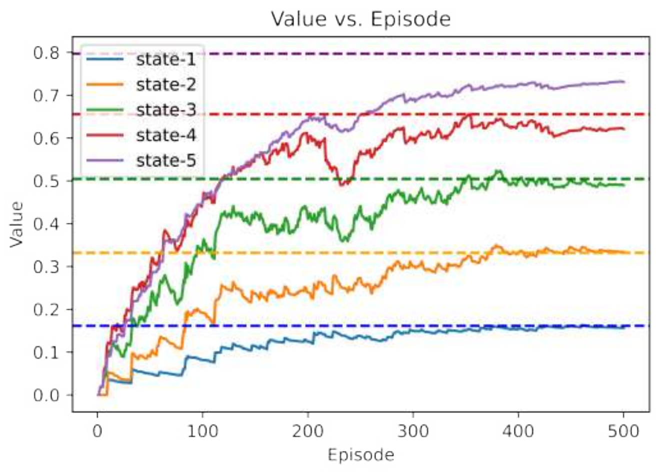


Figure 9: MC-EV estimates vs episodes

First thing to note here is that the noise in initial stages of plot is comparatively higher than the noise in first visit MC estimates. The alpha takes here is same as the above but due to increase in information about the visits the values deflects a lot . However it converges to it's true value faster. Also , higher the value of alpha taken higher is the variations. (Note that the dotted line in graph represent the true value of the states)

7. The Temporal Difference plot is given by :

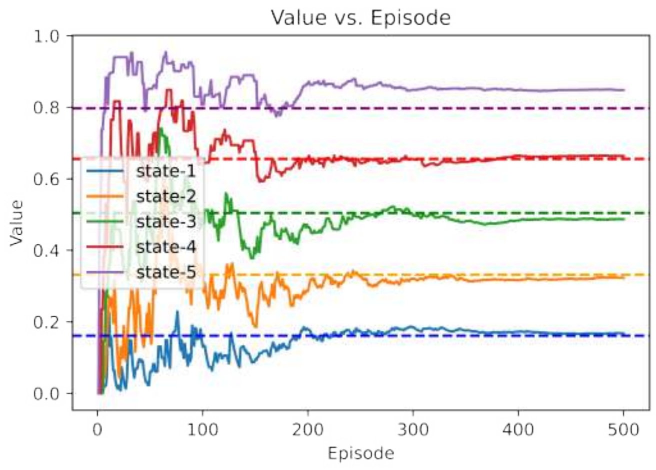


Figure 10: TD estimates vs episodes

While it's true that there's a lot of variance in the plot initially (alpha used is 0.5) , but the values of a state converges to it's true value faster than any of the above MC Estimates. Also , not to be confused why is it giving higher variance than Monte - Carlo as the alpha used in above plots are 0.1 , but the alpha used here is 0.5. If we compare it to 0.5 alpha - MC estimates , the variance is much lower than them. This is particularly because , TD- learning is a form of online learning , which updates it's value along the natural direction of the code.

8. The Plots are :

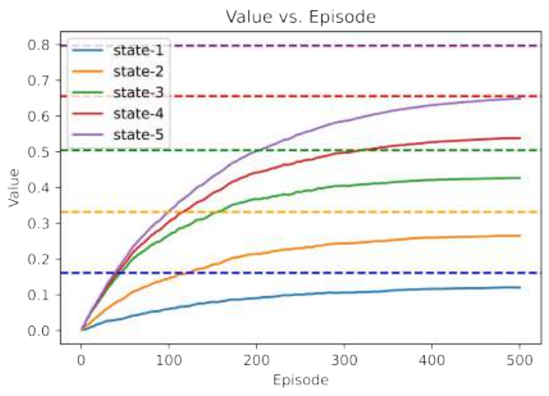


Figure 11: Averaged out MC-FV estimates vs episodes

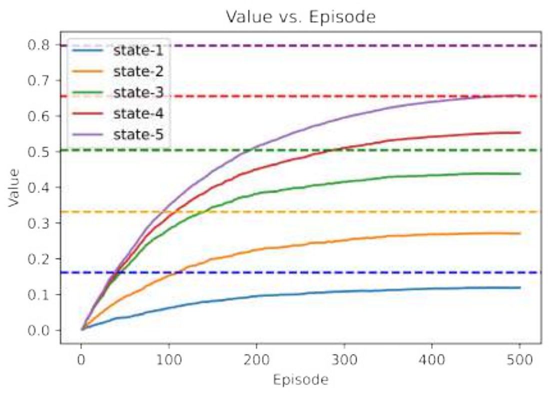


Figure 12: Averaged out MC-EV estimates vs episodes

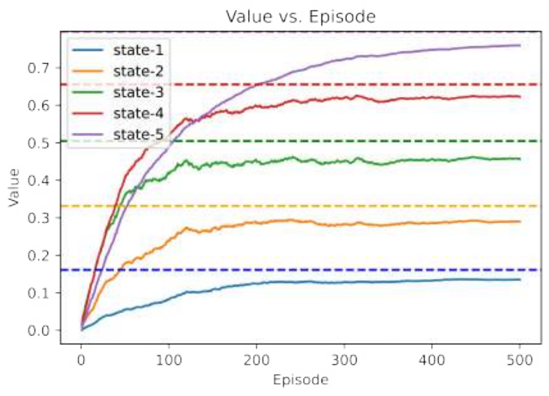


Figure 13: Averaged out TD estimates vs episodes

Now that we have Averaged out version of all three plots we can now compare it clearly. Also , we can see that the noise decreased and the curves are now flattened. This is due to the fact we are sampling from 20 different environments (The seeds taken are 0 to 20). The plots shows us faster convergence of TD approach and less biasing of MC values.

9. The log scaled MCFV plot is given by (Figure 14)

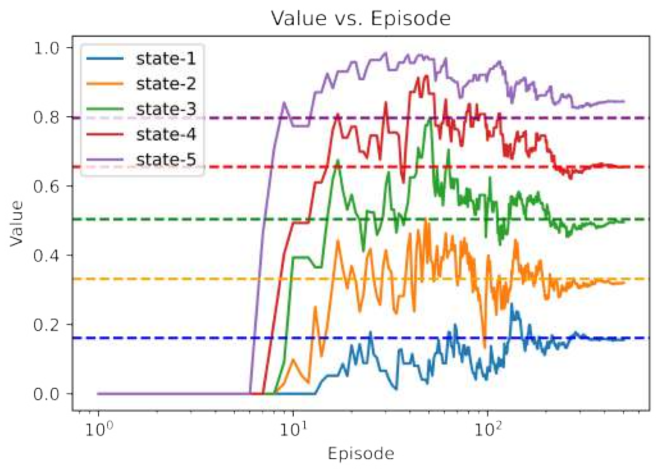


Figure 14: MCFV estimates vs episodes (log)

Here we can clearly see the variance which was not intuitively available to us in normal scale plots. Every state reaches it's true value but it oscillates around it a lot.

10. The log scaled MCEV plot is given by (Figure 15)

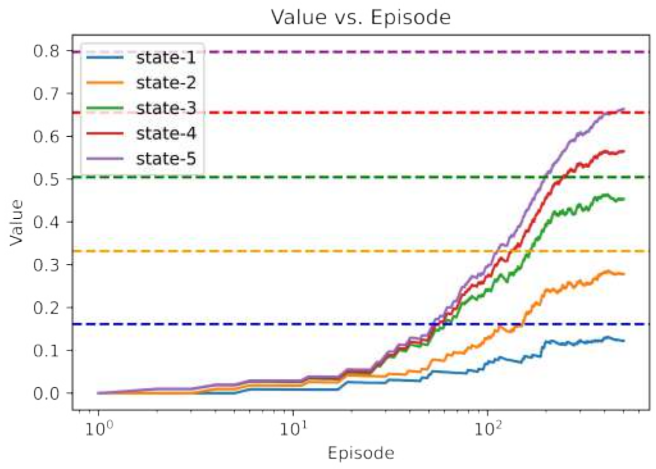


Figure 15: MCEV estimates vs episodes(log)

The log scale show the slow process of Monte carlo estimate reaching it's true value. The MCEV plot is particularly low because of extreme alpha of 0.01 taken but still fall behind TD methods which reaches to it's value faster.

11. The log scaled TD plot is given by (Figure 16)

    The TD plot show us the true nature of the algorithm which is basically learning s fast as it can. Within 100 episodes , the TD can already guess much better than MC values.

12. Now that we have all the required plots we can conclude some differences between both these methods.
    (a). MC methods generally shows slower convergence compared to TD methods. This is because MC methods require the entire episode to complete before updating state values, while TD methods update state values at every time step. Within MC methods, FVMC might converge slower than EVMC because FVMC updates state values only the first time a state is visited in an episode, while EVMC updates state values every time a state is visited.
    (b).MC methods typically have higher variance than TD methods. This is because MC methods rely on sampling full episodes, which can lead to high variability in returns.Within MC methods, FVMC may

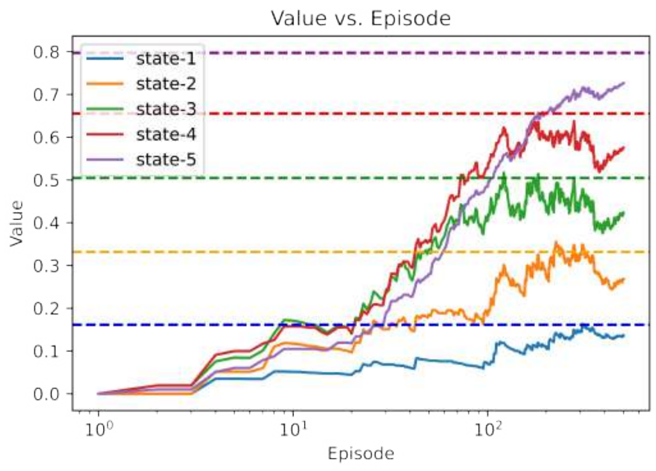


Figure 16: TD estimates vs episodes(log)

have higher variance than EVMC due to the episodic nature of updates in FVMC.
(c).TD methods are generally more sample-efficient than MC methods, especially in environments with long episodes or large state spaces. TD methods update state values more frequently and can learn from incomplete episodes.

13. The plot of MCFV Target value of state 5 is given by (Figure 17)

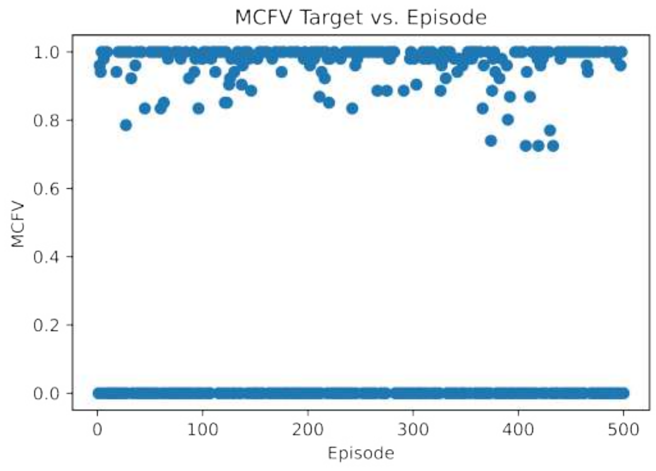


Figure 17: MCFV Target value vs episodes

14. The plot of MCEV Target value of state 5 is given by (Figure 18) One thing we can notice in both plots is that the value is either 0 or close to 1 (some of the values are discounted based on how close the state 5 is compared to current state) . Some value is particularly zero because of non appearance or zero reward dynamics of state 5.

15. The plot of TD Target value of state 5 is given by (Figure 19) Also the TD-error vs episodes is also plotted in (Figure 20)

16. Based on the given TD target and error plots one thing we can say for sure is the heavy influence of Environment dynamics on these scatters. The environment dynamics is completely random and gives rewards only on one state , this is one of the main reason of concentration of values near zeros and ones.

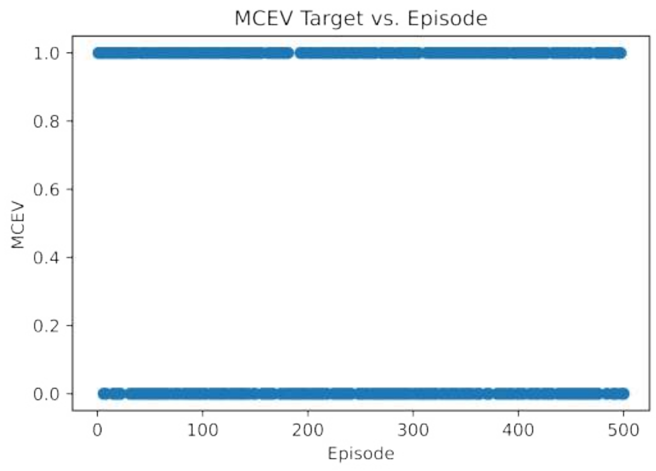


Figure 18: MCEV Target value vs episodes

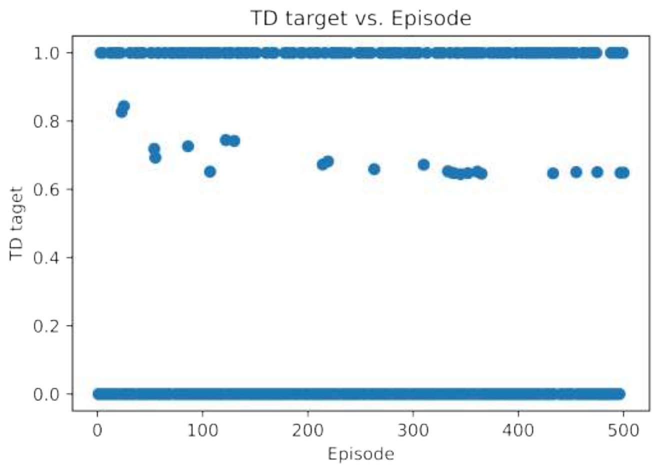


Figure 19: TD Target value vs episodes

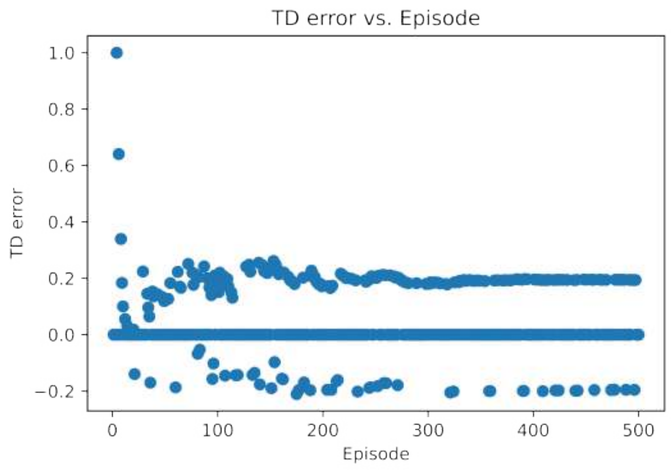


Figure 20: TD Error vs episodes

References

1. CS780 Lecture Slides.
2. OpenAI Gym Documentation