COL870 - Reinforcement Learning

Assignment 1

*By Ankur Sharma (**2015CS50278**)*

1. In my implementation, a valid (or actionable) state is represented as

state ≅**(A, B, [p, q, r])**

- **A => Player's Sum** (Dimension = 32)
  - It goes bust if it exceeds 31 or if it is negative (less than 0)
  - Sum can take any value from [0, 1, 2, ..., 30, 31]
  - Total number of possible values for the sum = 31 + 1 = 32

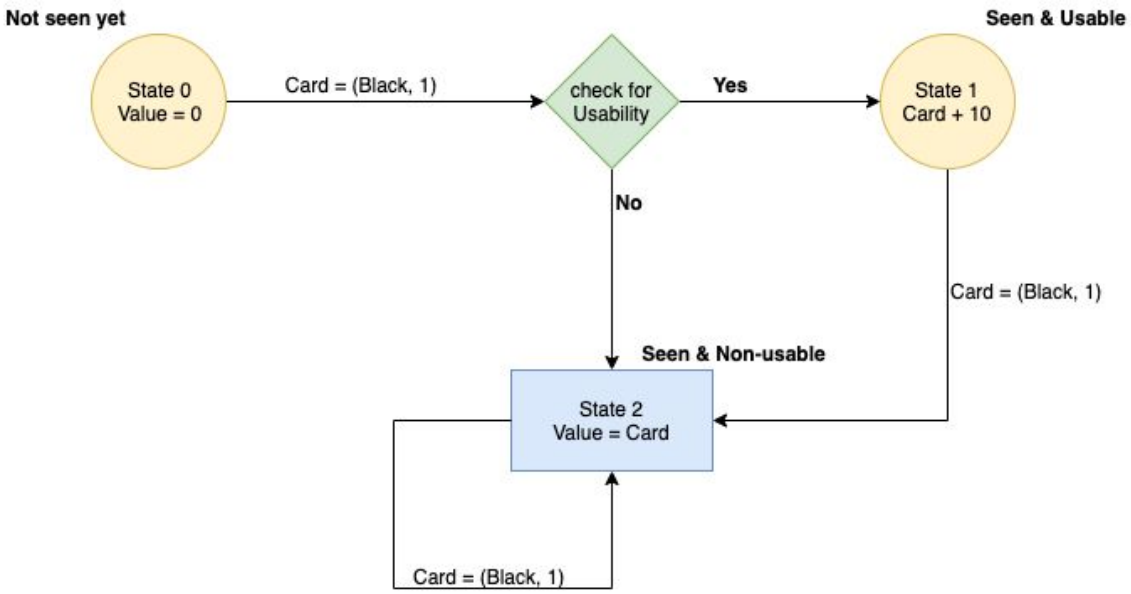
- **B => Dealer's Face Card** (Dimension = 10)
  - Initial show card of the dealer comprise as a valid factor in determining the agent's state. (*This value can range from 4 to 13 (both inclusive))*

- **[p, q, r] => Usable Set** (Dimension = 27)
  - There are 3 valid states (0, 1 or 2) for each of the black cards with value 1, 2 or 3
    - State 0 → Card not present.
    - State 1 → Card is present and it is usable (i.e. card's value + 10)
    - State 2 → Card is present but it is not usable (same value is used)
  - For each black card 1, 2, and 3 we can have one of these states which gives rises to a tuple (p, q, r) where each one of them can take 0, 1, or 2
  - This gives 3 x 3 x 3 = 27 possible combinations for usables

    Total number of actionable states = 32 x 10 x 27 = 8640 states

- Total number of actions = 2
  - Stick or Hit

Total number of valid state-action pairs = 32 x 10 x 27 x 2 = 8640 x 2 = 17280



Finite State Machine for the State Transitions in Usable Set

Non-actionable states refer to the set of all terminal states when the episode ends. These states can be reached directly (by going bust without taking any action in the start) or can be reached after our agent wins or loses. Once we are in such a terminal state, we are guaranteed that the agent will only have a single choice of action, and any other action will lead to a lower total reward. For example, if my agent loses, I land up in a non-actionable state, hence only possible action for my rational agent would be to stick. Agent trying to hit at this point (after the end of the game) will only gather negative rewards since the dealer has already won. Hence, Terminal States (or Non-Actionable States) can be any state on which the game ends.

(D, x, [p, q, r]) or **(-A, x, [p, q, r])** or **(x, -B, [p, q, r])** are all such non-actionable states in my current state space. (where A & B > 0 and D > 31, and x, p, q and r are any possible valid values)

2. Simulator

After implementing the classes for *Player*, *Dealer*, *Card*, *Action*, etc., the main task was to implement an *Environment* class in which my player can play and earn rewards. Player will start from some state, and take a particular action according to some policy which can be pre-defined or learnt. As the player executes the action, it jumps to the next state and collects an immediate reward. Determining the next state and reward essentially means that our environment should encapsulate all the rules of the game and inherently

implement a joint distribution i.e. $\mathbb{P}(s^{(t+1)}, r^{(t+1)} | s^{(t)}, a^{(t)})$. *Step* function of my simulator enables the agent to execute a particular action on a given state to collect reward and jump itself to a new state. *Reset* function is needed to get an initial state. Not that in special cases, our initial state can be terminal as well and we need to make sure that we choose an appropriate non-terminal starting state in generating our episode since we cannot execute an action on a terminal state. In my case, a state is unique determined by a tuple of *(x, y, p, q, r)* and this tuple representation has been used throughout the code for indexing and updating the value functions and action-value functions.

Implementing the state FSM was necessary to make sure that our usables are updated properly as planned.

3. Policy Evaluation (Model Free)

1.

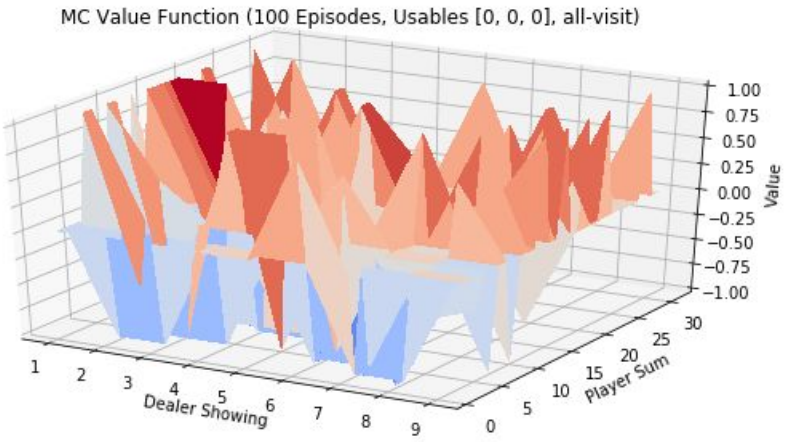
Create a Generic *Agent* class with some fixed values like *action_value_function*, *value_function*, *etc.* and some fixed functions like *take_random_action()*, *get_value_function()*, *etc.* Construct two Agents by extending the *Agent* class namely *MCAgentEvaluation* & *kTDEvaluation* for separating these two methods. Since here we need to evaluate a fixed policy, we define the *policy(state)* method inside these agents and execute our ***Monte-Carlo*** & ***k-Step TD Algorithm.***

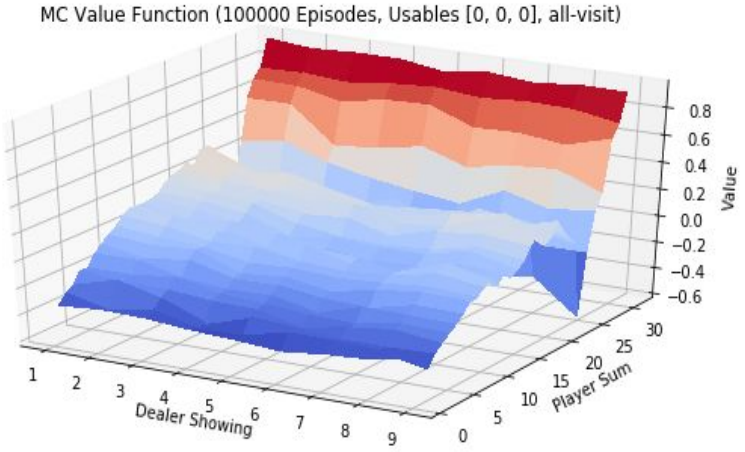
2.

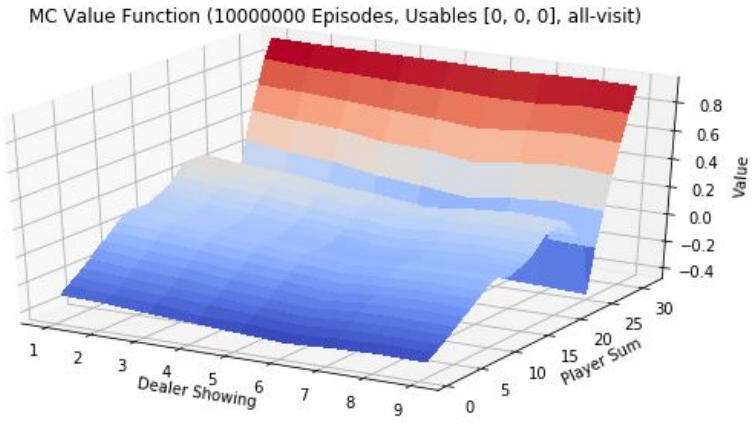
We train both the algorithms MC (*first-visit* & *every-visit*) & TD (for several *k*-values) as per the assignment objectives. To show the value function, we need to fix the value of the usable tuple *(p, q, r)*. For every such tuple value, we achieve a plot of a value function. Occurence of cases with *(0,0,0)* is much more likely than the cases which have usable tuple ([*2, 2,2*] being the worst of all the cases). Here, we consider only 3 cases for representation purposes.

MC Value-Function Plots

(All Visit Update, [0, 0, 0] usables)

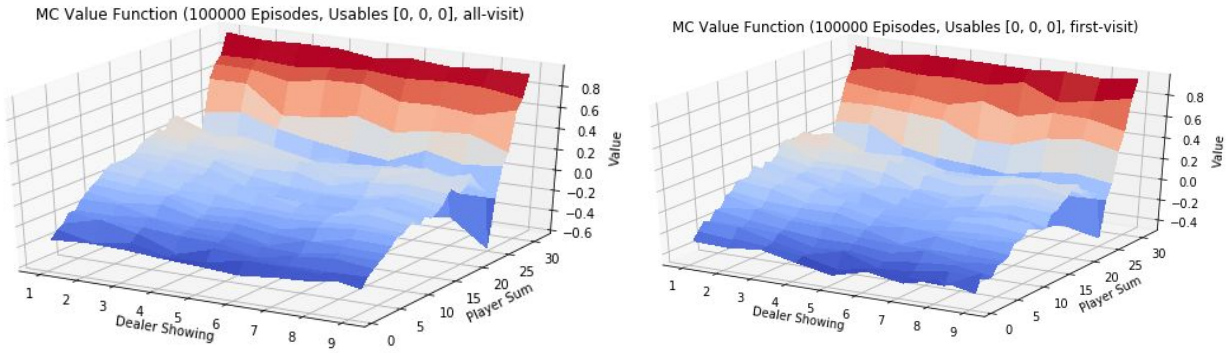






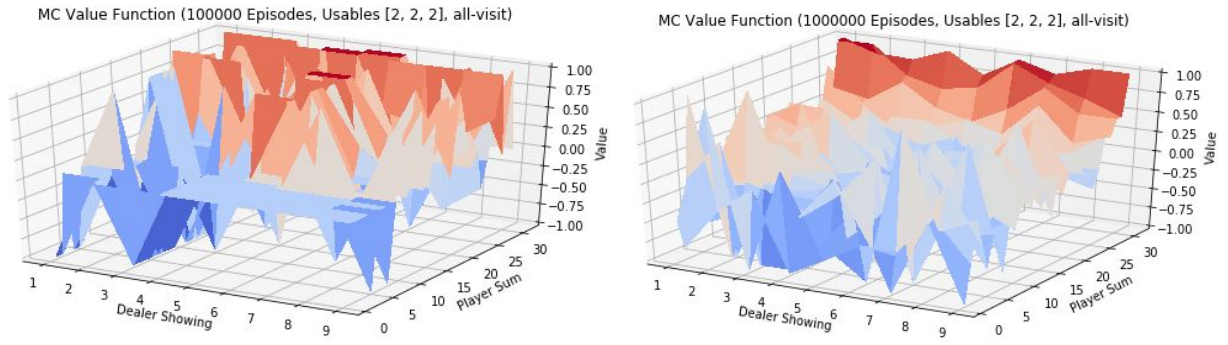
Red correspond to good states, while blue corresponds to bad states. From the graphs, it is clear that it is good to have a good sum. Jump in the values in the later rows signify that these hands are more likely to land into states that will yield a reward of +1. It also shows that irrespective of dealer's card, very low player sum or sum around 24 will more likely lead to poor states. This intuitively makes sense also since low sums are more likely to get bust (become negative), and sums around 24 (given that you do not stick) are more likely to cross the sum 31 and get busted. Plot also shows that the player lands up in a decently good state if its sum is around 16, which is like a safe state. When the player hits around 16, it is not going to bust directly.

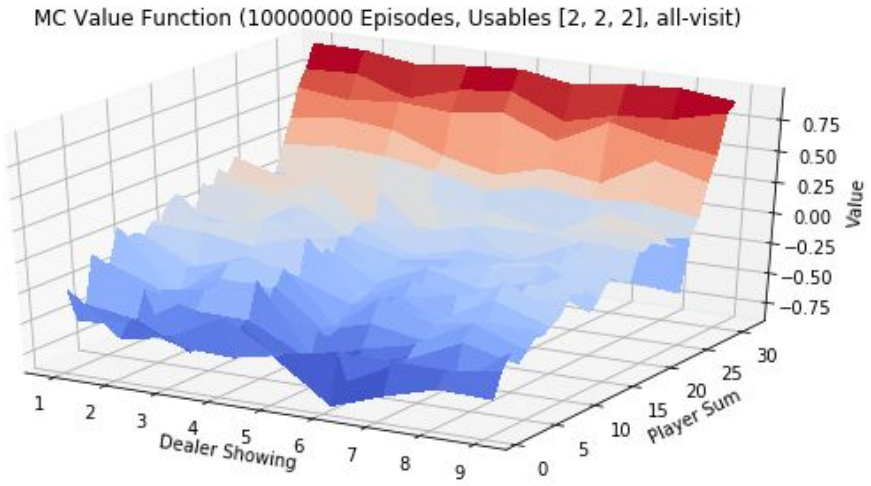
First Visit vs Every Update



All-visit update curve turns out to be a little smoother than first-visit curves since all-visit makes sure to capture more occurences (hence more returns) of that state and all such fluctuations average out this way. However, when the number of episodes are extremely large both of them converge to a uniform smooth distribution.

(All Visit Update, [2, 2, 2] usables)



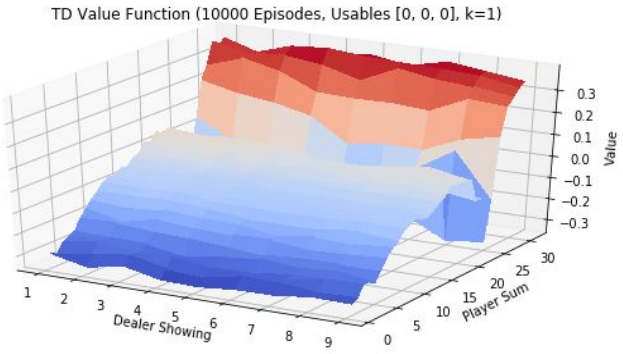


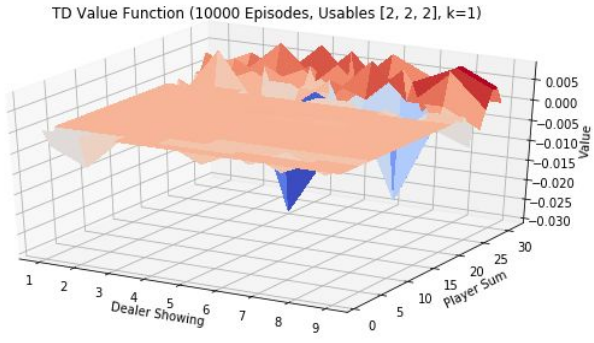
Since the number of states with (2,2,2) usables are very low (they are encountered the least likely), hence they take quite a large number of episodes (larger than (0,0,0) usables) to converge to a meaningful value function. Here, we can see how the value function for (2,2,2) changes over different set of episodes, and eventually attains a shape after 10 million episodes.

We also see that MC is not sensitive to initialisation of the action-value function, and hence is bias-free. Although we will see that MC takes more number of iterations to converge than TD, it is important to note that bias-free MC usually has a high variance and is very well shown in all the plots. And we can see as experience increases, MC (first-visit or every-visit) eventually converges to the value function easily since MC has good convergence properties.

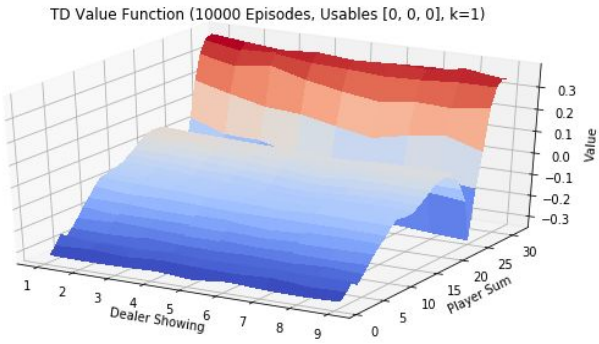
TD Value-Function Plots

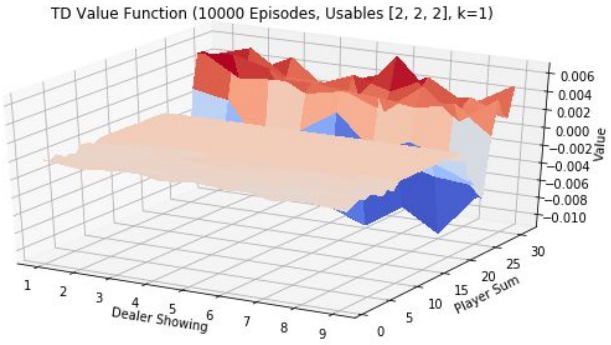
- **K = 1**
  - **100 Runs**





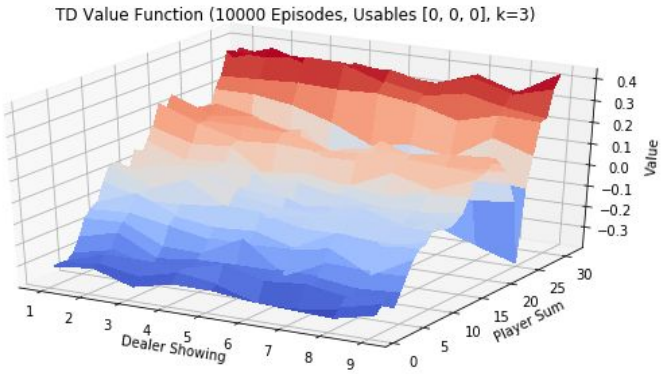
  - **1000 Runs**

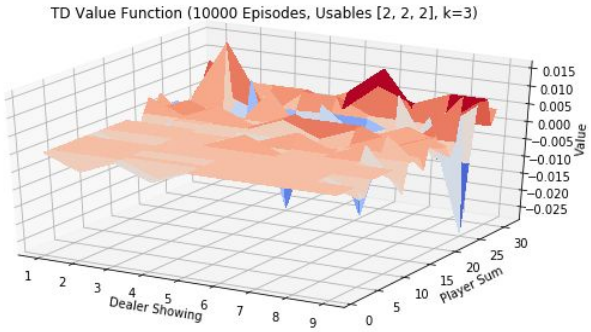




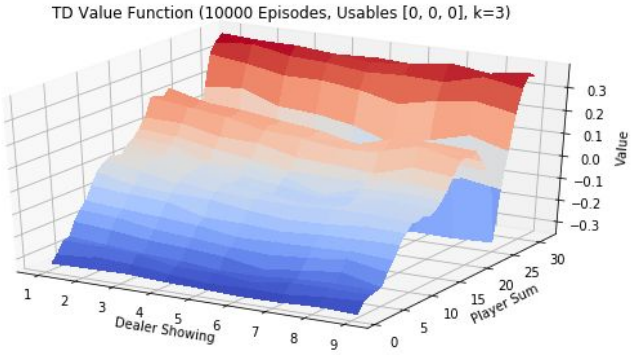
For 1-Step TD, we notice that the value function is quite fluctuating for 100 runs but for 1000 runs it is pretty much smooth. 1-step TD offers the largest number of updates in each run by learning online after every step. (2,2,2) still needs a lot of updates to converge but we see that the curve for (0,0,0) is much clearer for the fixed policy we're evaluating. Since, TD is more sensitive to initial value, we see that when the number of episodes are not sufficient (e.g. in case (2,2,2) 10000 episodes are not sufficient for convergence), it is sensitive to initial value (i.e. zero value function). Hence most of the values are zero for player sum less than 20. It is more efficient than MC in the sense that it converges faster (only 10k episodes) and almost to the same value, but then it has an inherent bias which MS doesn't. For the same number of episodes, we can also see that MC has high variance than TD. Comparing TD for other values in the following pages.

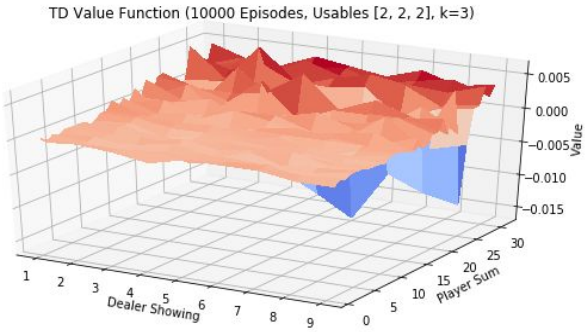
- **K = 3**
  - **100 Runs**





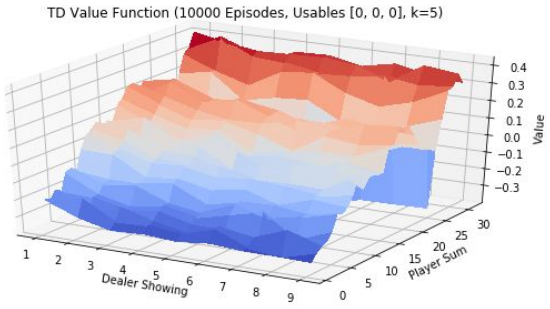
  - **1000 Runs**

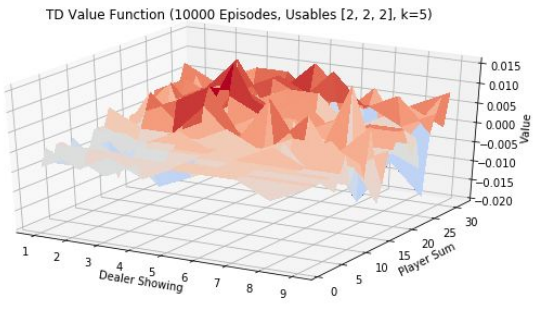




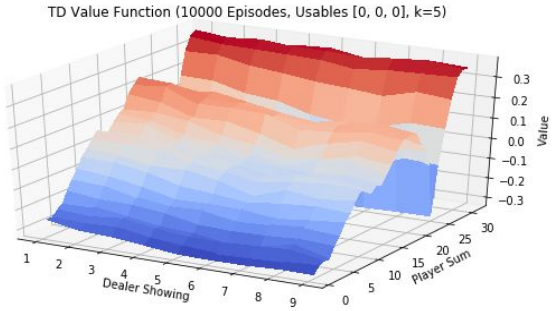
Increasing the value of K from 1 to 3 increase fluctuations in the curve even after averaging. We can observe in this graph and in a couple of graphs to follow that increasing the value of K increases the variance of the value functions. Moreover, we see that the peak in the middle region (around 20) also hiked up. (note color change from blue to red) This tells us that us updating value function after every 3 steps in an episode leads to better states when the sum is in the middle region (around 17). Interesting observation to note is that now we can see the updates even for regions with a lower player sum. Similar trends will be followed in cases where K = 5, K = 10 and so on.

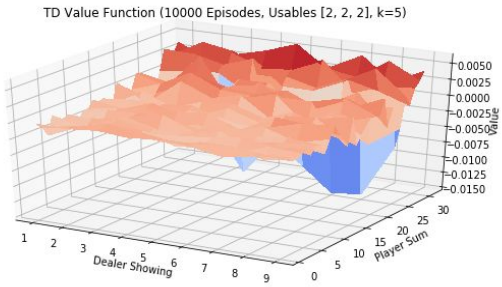
- **K = 5** **100 Runs**



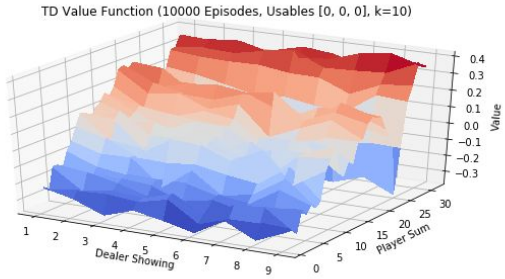


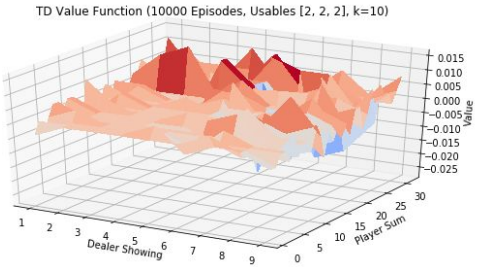
1000 Runs



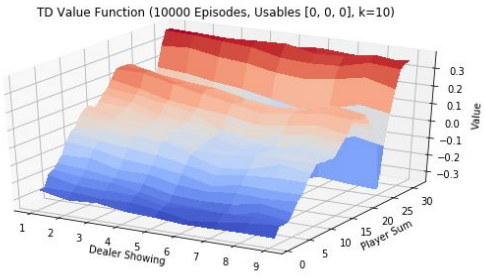


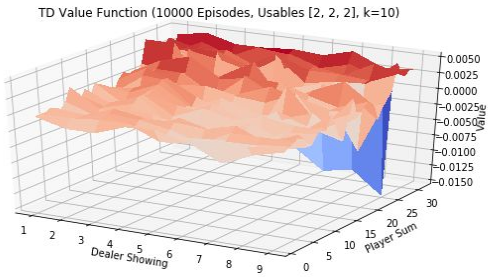
- **K = 10** **100 Runs**





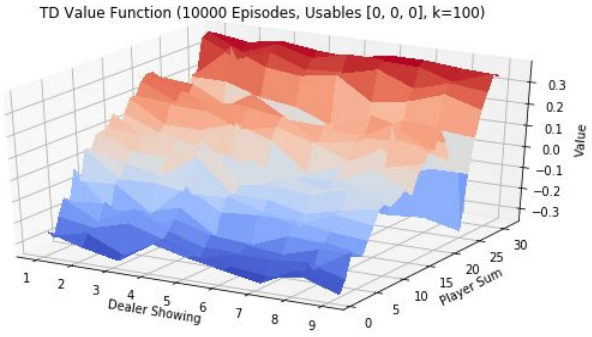
1000 Runs

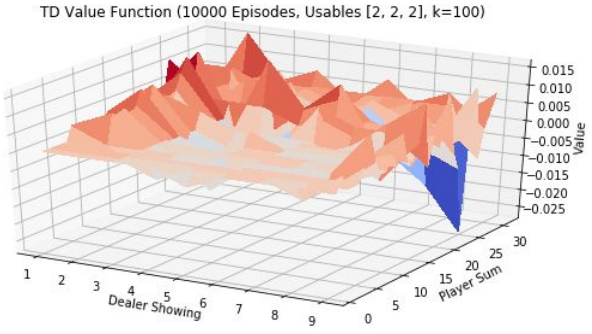




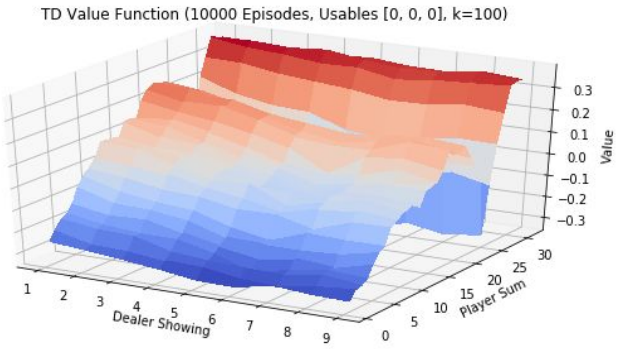
We see how averaging over 1000 runs helps in removing the variance in the value functions over 100 runs. Hence, while the plot for 100 runs is still shaky, we note that the plot for 1000 runs is pretty smoothed out and balanced. Again, we can verify that increasing K increases the height of the peak around sum's middle region (more red region noticed), and updates are now more visible on the (2,2,2) spectrum as well now. It is trying to de-flatten itself but the number of episodes are too small for (2,2,2) convergence.

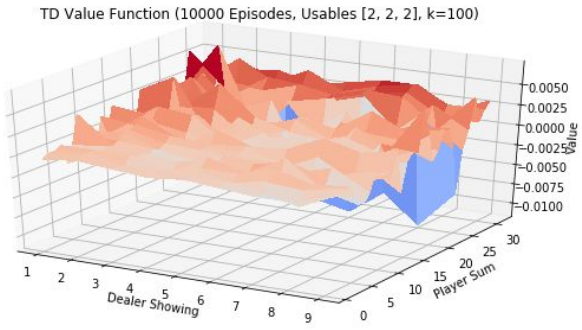
- **K = 100**
  - **100 Runs**



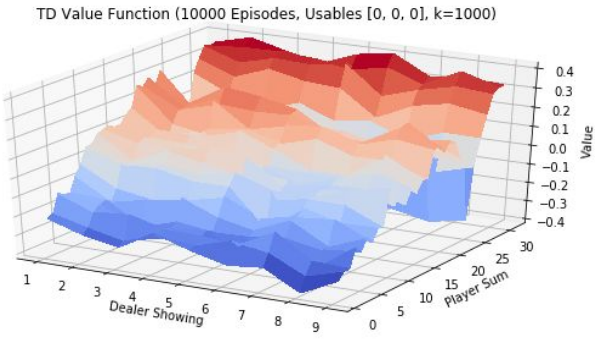


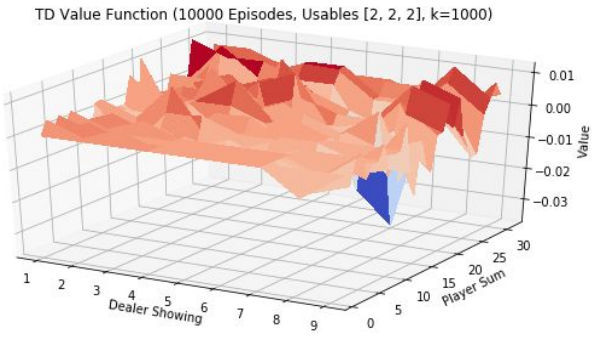
  - **1000 Runs**



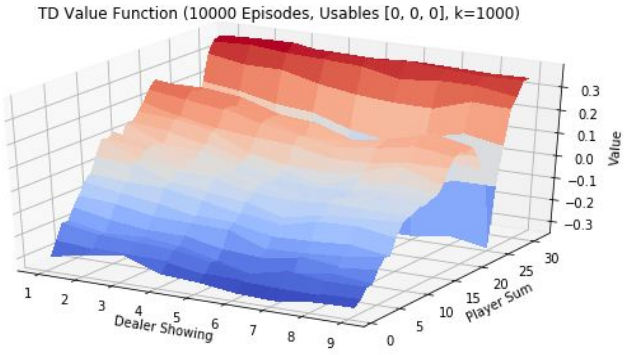


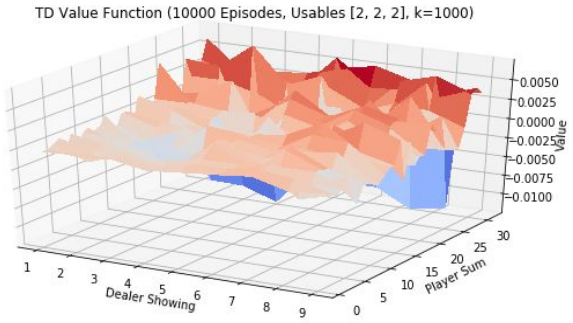
- **K = 1000**
    - **100 Runs**





    - **1000 Runs**

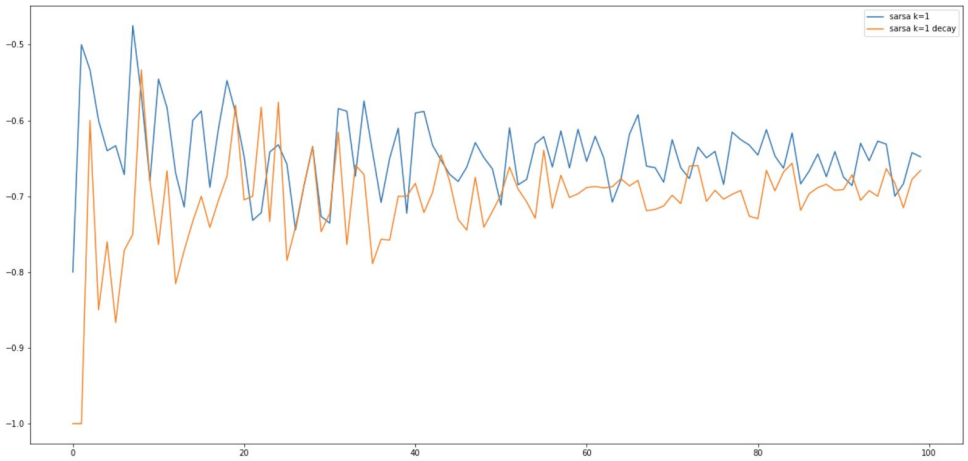




1. Compiling all the results above, we see that increasing K increases variance in the graphs and eventually in MC (when K → ∞) we can compare from the previous curves how MC offers high variance which we can observe here also as K increases.
2. Averaging helps in reducing the variance and giving out smooth plots.
3. Increasing K increases the value of the peak observed and hence leading to better states.
4. Compared TD v/s MC
    a. TD
        i. Lower Variance but Biased
        ii. more sensitive to initial value, Faster convergence
    b. MC
        i. High variance but no bias
        ii. Less sensitive to initial value, Slow convergence

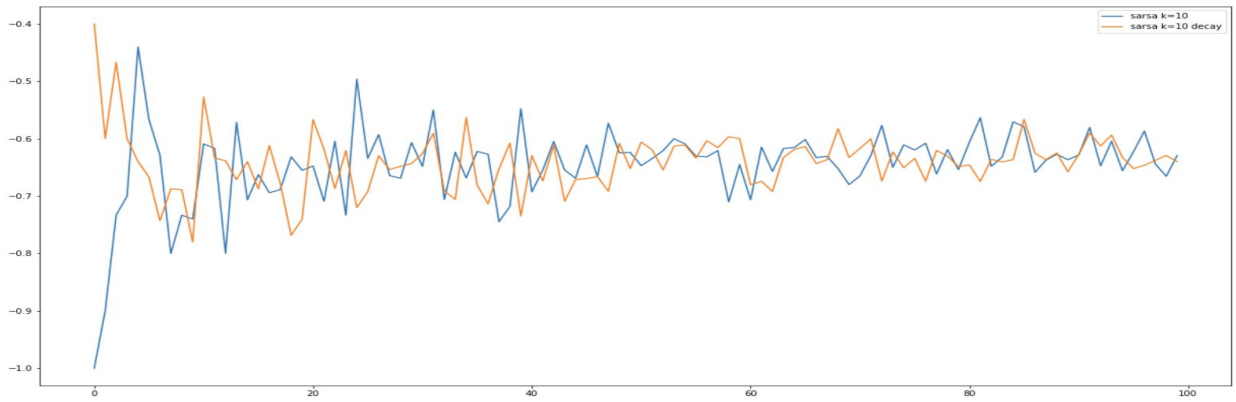
4. (2) Performance variation with number of episodes

1-step SARSA Comparison (fixed v/s decaying epsilon)



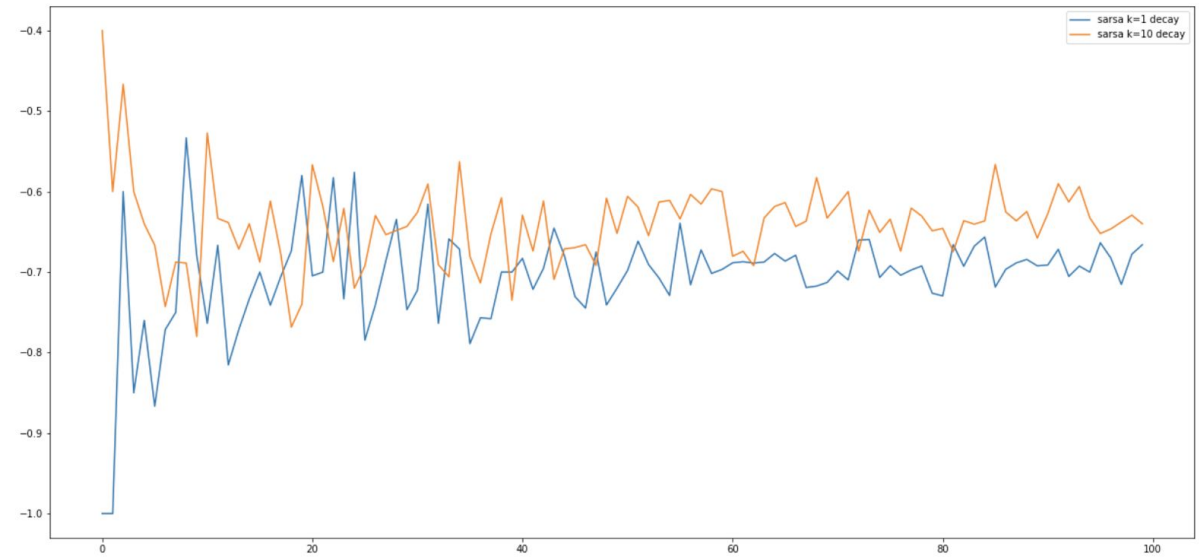
X-axis denotes the episode count, and Y-axis denotes the average cumulative reward for 10 runs. Here, we compare the effect of ε -decay on 1-Step SARSA. As per the curve obtained, we see how 1-step SARSA gives rise to better rewards than the decaying it with the number of iterations. ε -greedy policy is useful in taking into account exploration along with exploitation. As per the policy, we take a random action with ε probability and choose the best known action with (1- ε) probability. Choosing ε = 0.1 forces the agent to choose the best known action with high probability (0.9), and explore with only 0.1 probability. Decaying ε hampers exploitation which might not help the agent achieve its optimal value function more so when the update is occuring at each step (k = 1). Reducing exploration over time in SARSA by decaying epsilon enables it to learn an optimal policy directly *among explored states*. Hence, in this case, decaying doesn't help in improving rewards by actually trimming down a set of unexplored states which might have some optimal states.

Performance for SARSA (k = 10) with fixed/decaying epsilon



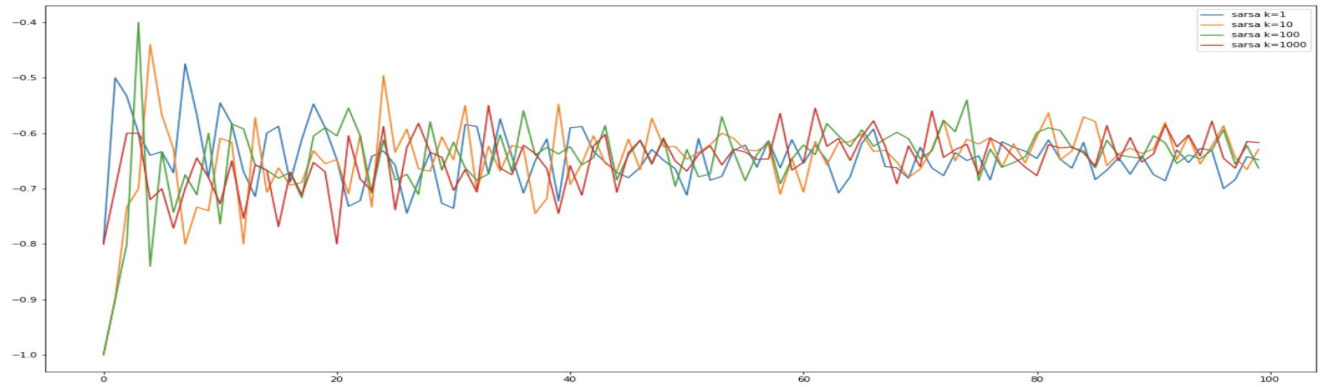
Note that previous trend is not universal and may not hold for other K for whom exploration doesn't really help and decaying epsilon actually helps as can be seen in the figure on the next page. Here, the result is independent of the decay since the updates occur at a larger step size (i.e. 10).

1-Step v/s 10-step SARSA Comparison (decaying epsilon)



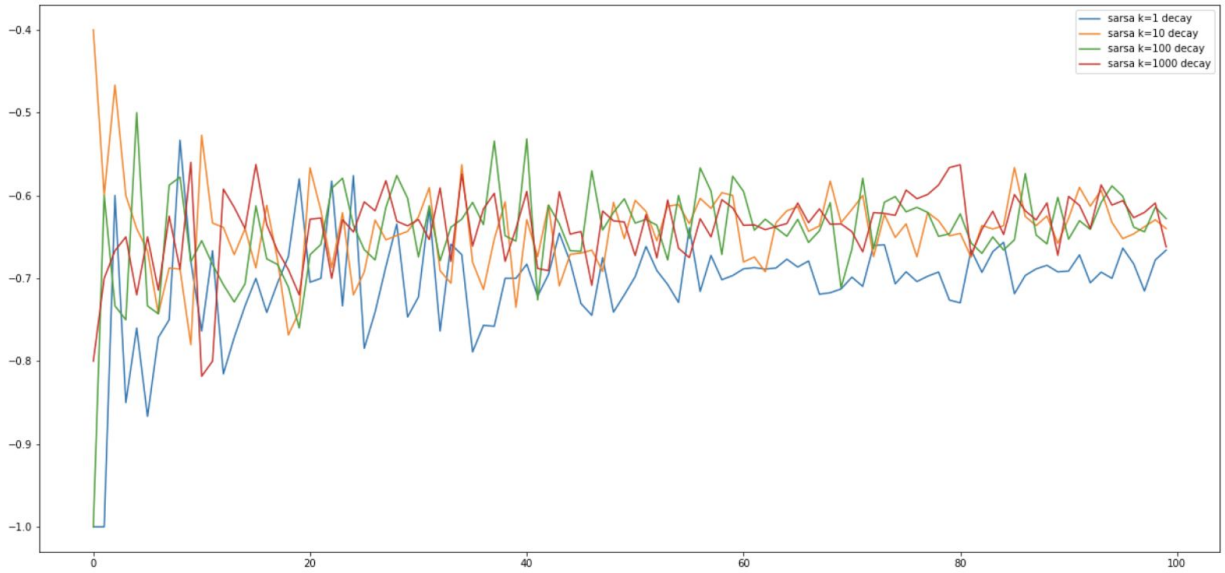
10-step SARSA performs better than 1-step SARSA with decaying epsilon since we need to keep a balance between TD(0) & MC. This makes sense also because TD(0) suffers from initial estimates of the action-value function which can be highly inaccurate. Thus, updating after 10 steps this way actually helps in incorporating the robustness of MC methods in TD while making sure that we do not have to wait till the end of the episode before updating.

k-Step SARSA Comparison (fixed epsilon)



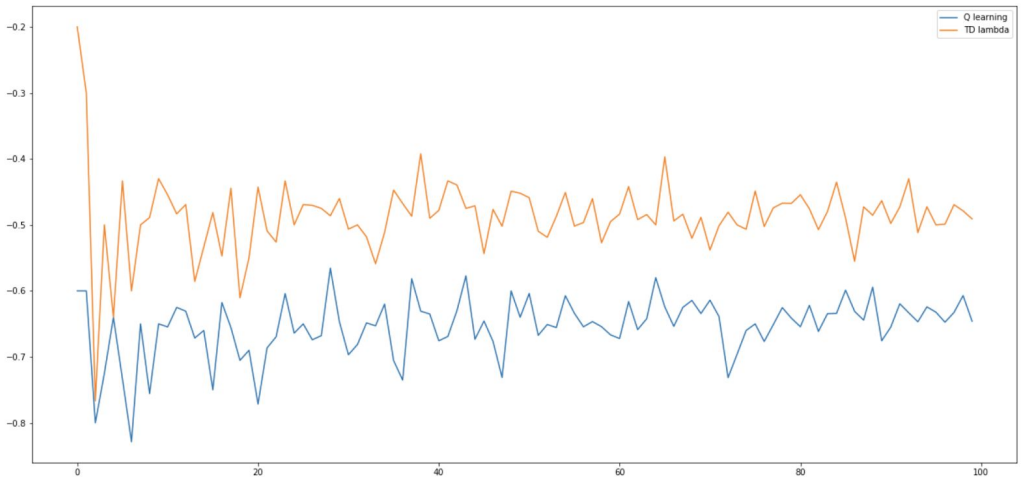
We observe that keeping the epsilon fixed doesn't help improve the policy by always acting non-optimally with a certain probability (ε). From a holistic view, we cannot see a clear distinction on their relative performance after averaging the rewards by keeping the epsilon fixed.

k-Step SARSA Comparison (decaying epsilon)



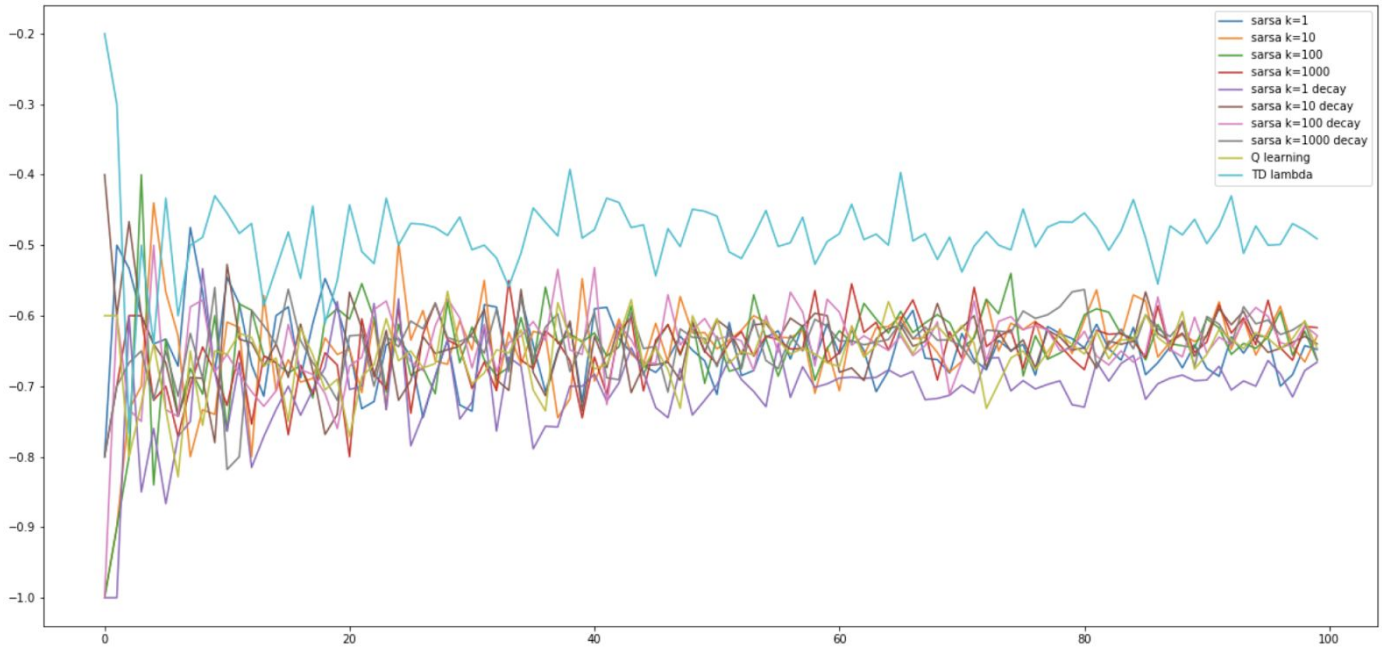
On the other hand, when we decay epsilon we can make the following conclusions. We see how higher K values like 10 or 100 perform better than K = 1 showing how n-step TD is actually successful in exploiting the robustness of MC within it. Taking K as 1000 is too large for the typical size of an episode due to which for those episodes, it will essentially become equivalent to MC.

Q-Learning v/s TD(λ) Comparison



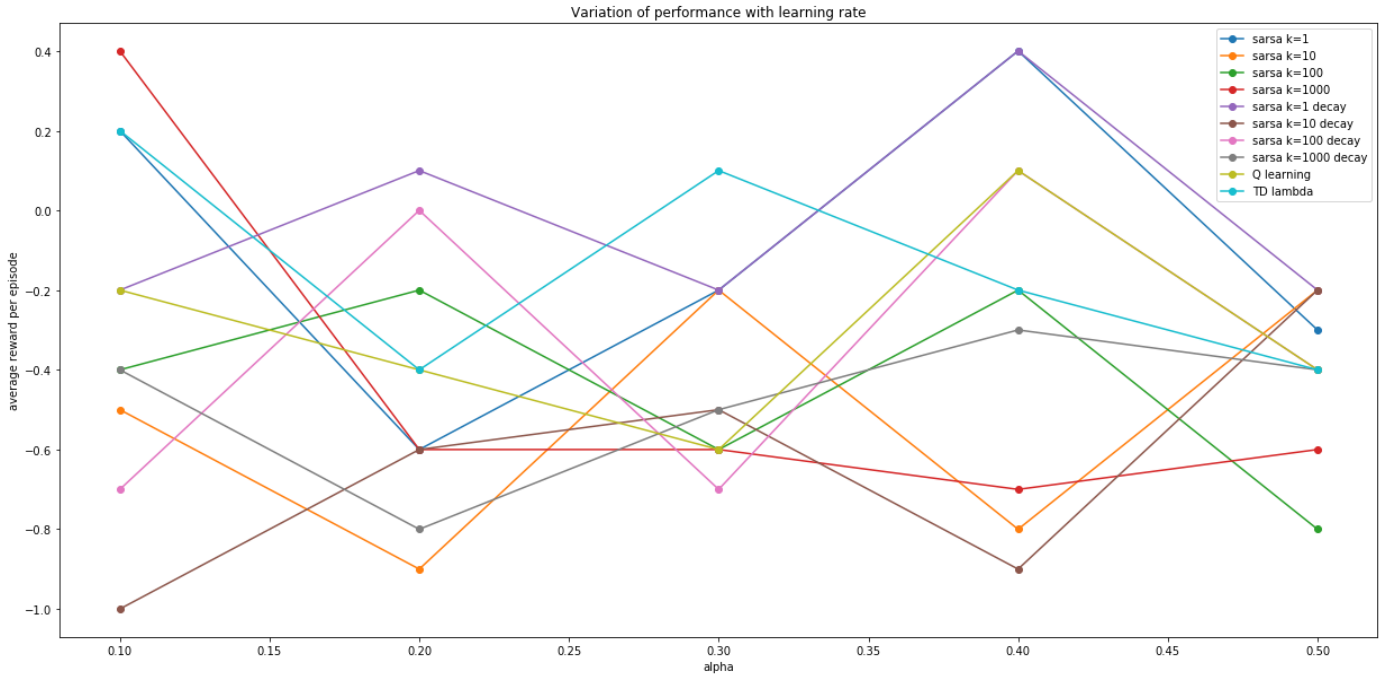
Clearly, TD(λ) performs better than Q-Learning. In TD-learning, instead of just updating the Q-value of your last step, you also update the previous steps. TD(λ) uses a discounting factor λ to make changes to predictions made further in the past smaller relative to changes in the immediate past. This accounts for any uncertainty about the dynamics of the world and the possibility that the policy being followed is not optimal, for example: the decision we made 10 steps ago might have been right, but afterwards we made a mistake, which we are now correcting, so we should not change our prediction from 10 steps ago too much. This way TD(λ) corrects for the updates dynamically and eventually converge to optimal value function.

Comparison of all Algorithms



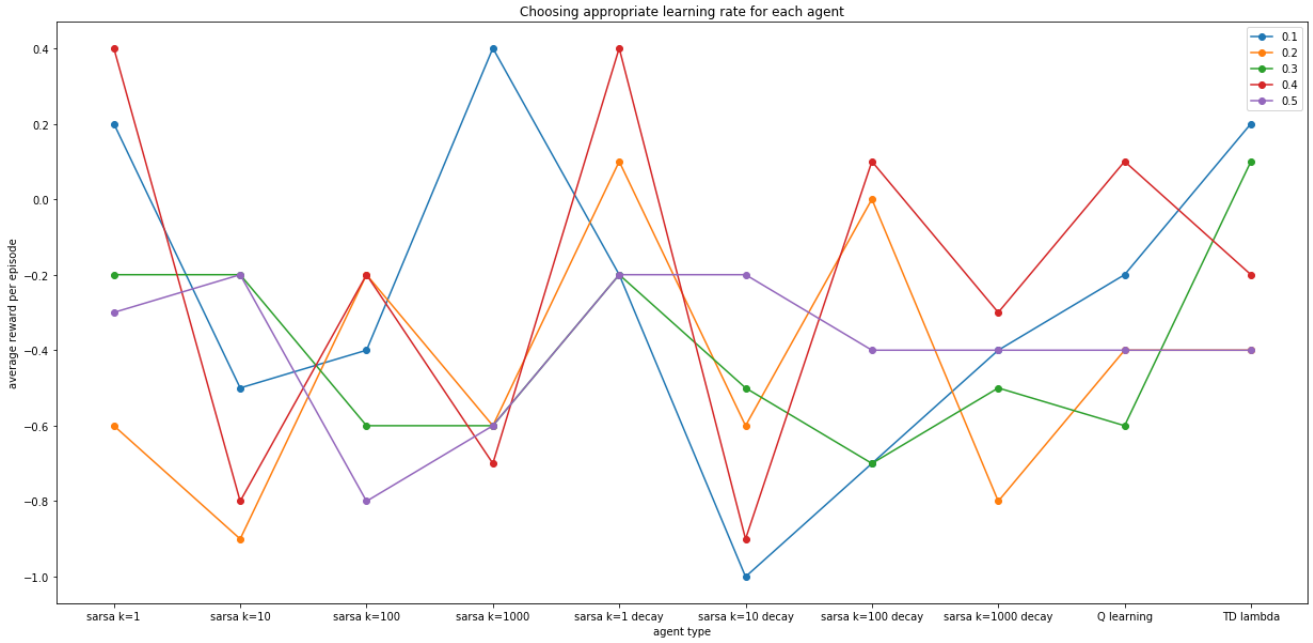
Comparing all the algorithms together, we see that TD(λ) is able to learn the fastest with the best cumulative average reward. By similar reasoning as above, TD(λ) can account for uncertainty in the environment and can correct itself in a dynamic manner by making changes to predictions made further in the past smaller than the immediate past. k-Step SARSA & Q-Learning can only make update the last step and cannot account for such uncertainty.

4. (3) Change in performance with α



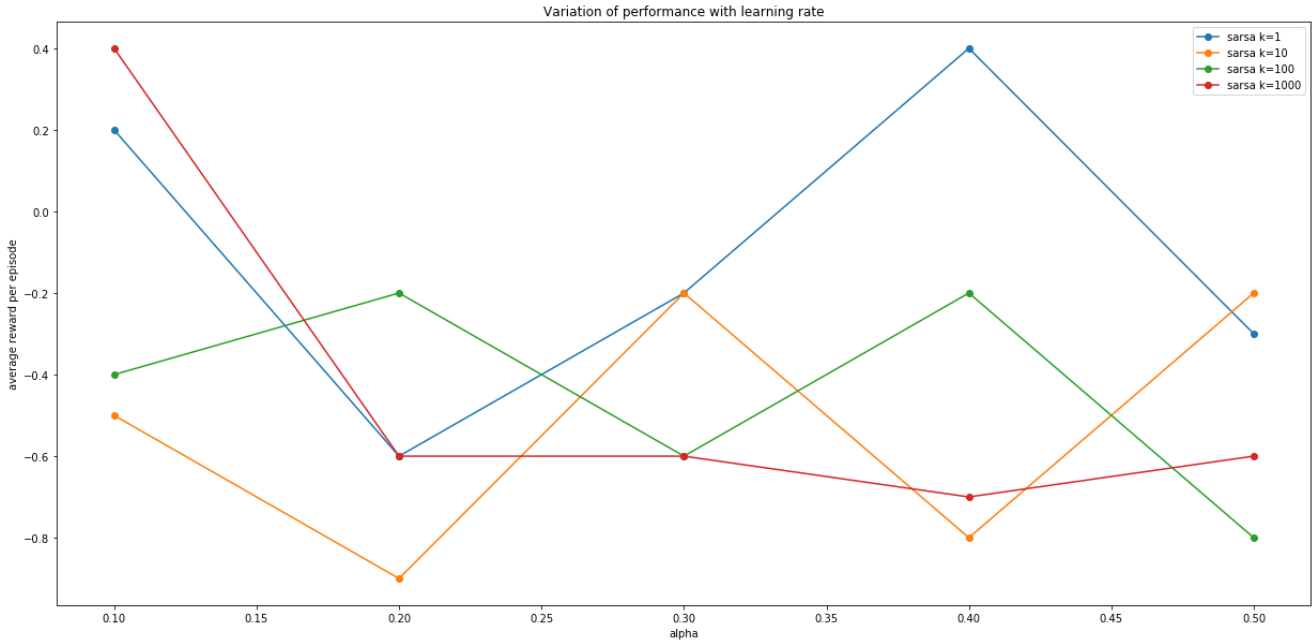
Alpha (α)	Best performing algorithm
0.1	1000-step SARSA (fixed ε)
0.2	1-step SARSA (decaying ε)
0.3	TD(λ)
0.4	1-step SARSA (fixed or decaying ε)
0.5	1-step or 10-step SARSA (decaying ε)/10-step SARSA (fixed ε)

Comparing the variation of average reward with **α**, we see that every agent has its own optimal value of alpha at which it leads to maximum reward for a given number of episodes. Nonetheless, all of them will converge to their optimal value regardless of the learning rate. Best & Worst performing algorithms typically use **α** to be 0.1 or 0.4 as can be seen in the graph.



Algorithm	Optimal Alpha (α)
1-step SARSA (decay)	0.4
10-step SARSA (decay)	0.5
100-step (decay)	0.4
1000-step (decay)	0.4
1-step SARSA (fixed)	0.4
10-step SARSA (fixed)	0.3/0.5
100-step SARSA (fixed)	0.2/0.4
1000-step SARSA (fixed)	0.1
Q-Learning	0.4
TD(λ)	0.1

Optimal value of **α** also varies for each agent and it is mostly centered around 0.4 for most of these agents. Peak performance is attained at 1-step SARSA (fixed/decaying epsilon) at **α**=0.4. Worst performance occurs at 10-step decay with reward of -1.0 at **α**=0.1.

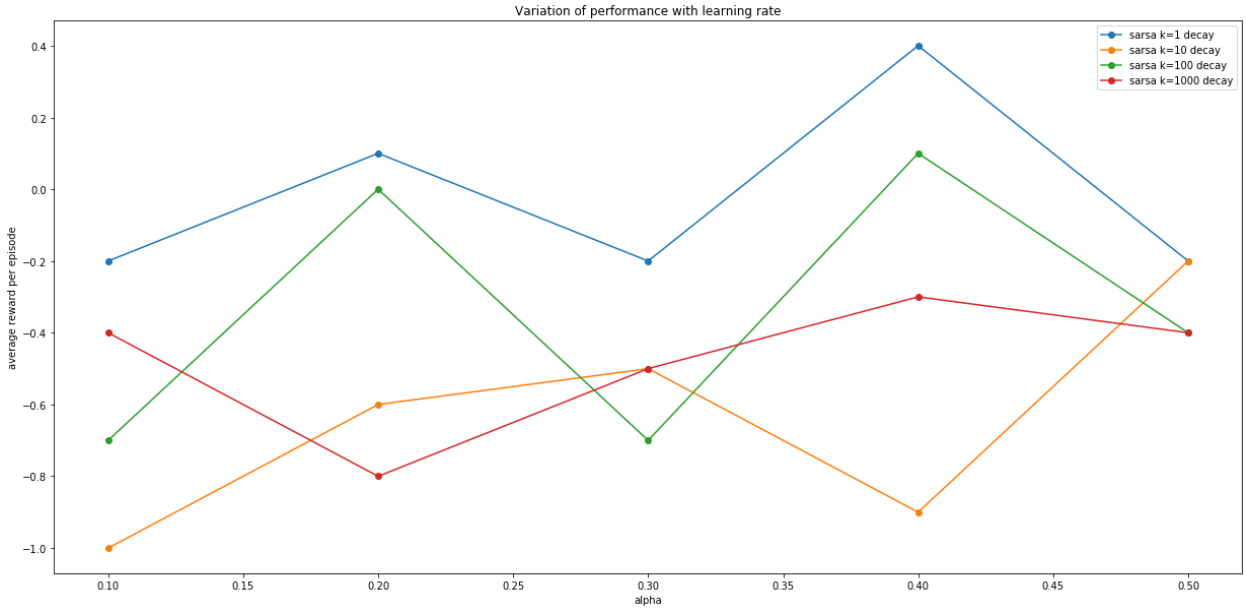


Further, we can see that for SARSA (with decay), optimal value of **α** decreases with increase in the value of K. This can be explained from the fact that as the step size increases (and epsilon stays fixed), collective updates become larger and even small **α** can be used to update. As per the table below, it can be concluded that smaller alpha can give optimal updates for agents that have a big step size. This is true as long as the epsilon is fixed.

Value of k (SARSA without Decay)	Optimal Alpha (α)
1	0.4
10	0.3/0.5
100	0.2/0.4
1000	0.1

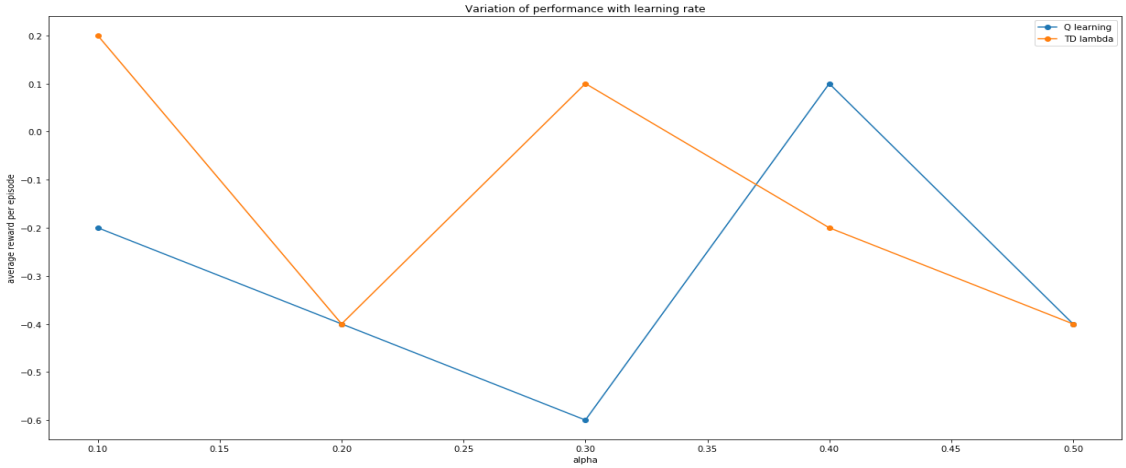
Value of k (SARSA with Decay)	Optimal Alpha (α)
1	0.4
10	0.5
100	0.4
1000	0.4

When the epsilon varies, then we observe that almost all the k-step SARSA agents have an optimal learning rate of 0.4 irrespective of the step size.



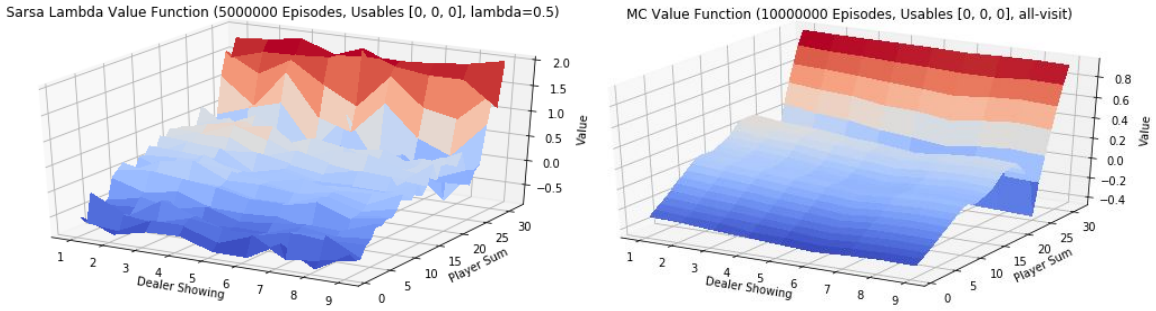
Comparing the effect of learning rate on Q-learning & TD(λ), we note that that a Q-Learner needs high learning rate of 0.4 for optimal updates, while a TD(λ) agent can achieve a better optimal update with a lesser value of learning rate. This difference can arise from the fact that TD(λ) not only updates the last time step, but also the previous time steps, and hence a large learning rate wouldn't help a TD(λ) agent to dynamically correct the uncertainty it comes across. Q-Learner only updates the action value of last encountered state-action pair, and a large learning rate would be sufficient for it.

Average Reward v/s Learning Rate for Q-Learner & TD(λ) Agent

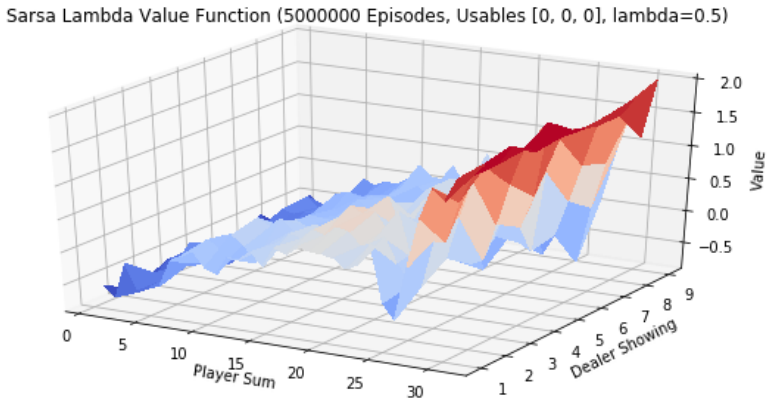


4. (4) Executing TD( ) with $\lambda = 0.5$ using the forward view requires us to compute the value of G_t^{λ} explicitly which involves the generation of an episode with ε -greedy policy. Making use of the fact that the internal rewards are 0, I've optimised the algorithm to easily calculate the value of $G_t^{(k)}$. Comparing this with the fixed policy, we see that the surface is much gradual and flat now till it reaches around 26 where it jumps. There is no such distinguishing maxima observed here as it was in the fixed policy case. Transitions are gradual, and we see the dependence of dealer's card on the value function as well. Low values for the dealer card give rise to better states for the agent when the sum is around 24. This dependence was not seen on the fixed policy case.

Very good states occur around the sum greater than 29 or so. Overall, the learning is faster in TD(λ) but it has a high variance (more fluctuations) making the entire curve less smooth even after training for 2 million episodes than the case with the fixed policy. Also, note the difference in the peak value functions which is much higher in this case than in the fixed policy case.

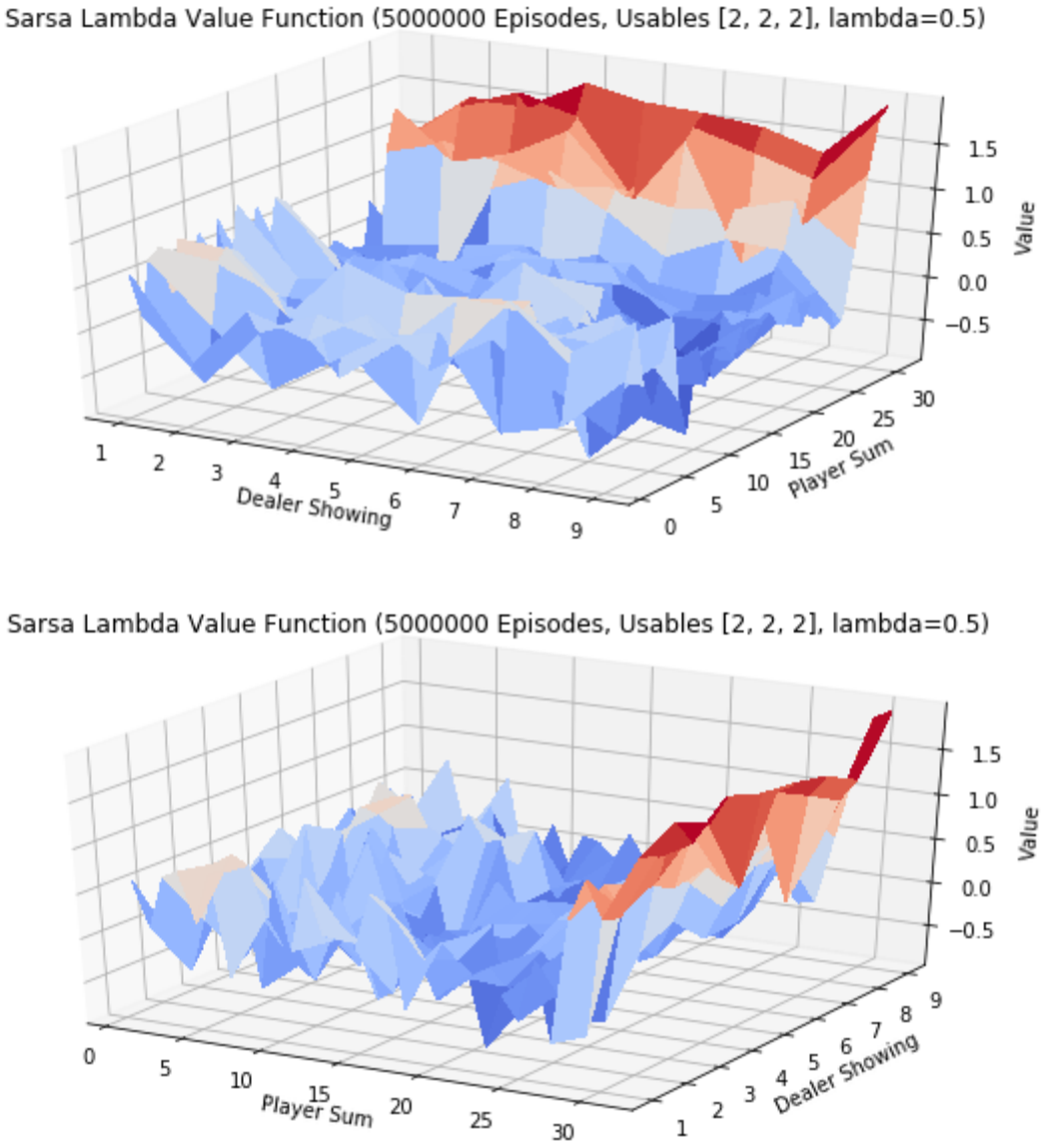


Policy learnt from TD( ) v/s Fixed Policy



Front & Side View for the optimal value function using TD( )

For case (2,2,2), we see a similar trend in the fixed policy case with the difference that it is able to learn faster even when the likelihood of the occurrence of this case is quite low. It also converges to the optimal value function.



Front & Side View for Optimal Value Function for (2,2,2) case using TD( )

References:

- David Silver Course on RL: Assignment Easy21
    - Plot function and basic class structure has been adapted for reference
- Libraries used (no in-built library used)
    - numpy
    - matplotlib
    - random
    - enum
    - copy
    - tqdm
- Jupyter Notebook (*RL-1.ipynb*) has been submitted. It contains all the methods in a step-by-step manner
- Remaining Results/Photos on Google Drive Link