Assignment 1
COL870 : Reinforcement Learning

Navreet Kaur
Indian Institute of Technology, Delhi
2015TT10917
tt1150917@iitd.ac.in

I. State Space

State Representation

The State Space has been represented using the following variables:

1. *agent_score*: The sum of cards that agent has. It ranges from 0 to 31 for non-terminal states.
2. *dealer_first_card*: The first card of the dealer that is visible to the agent. It goes from 4 to 13, since in the case of special cards, i.e. black 1/2/3, their value will always be considered higher for maximising the sum.
3. *agent_usables*: This is a list of flags corresponding to usable black cards with values 1, 2 and 3, i.e. the ones which can be used as 11, 12 or 13. Flags have values 0/1/2 which mean:
    a. 0: The card is has not occurred until now
    b. 1: The card is usable i.e. the first time occurence of the card and using it at higher value does not bust the player
    c. 2: If a special card occurs but is not usable, i.e. either it has appeared before or using special value of the card at first occurrence busts the player

In this implementation, the value function is represented by a numpy array with shape [31,10,3,3,3] and the state-value function is represented by a numpy array with shape[31,10,3,3,3,2], since there are two actions - hit and stick.
In the rest of the document, the term 'Usables' will be used to refer to *agent_usables* , with meanings of flags 0, 1 and 2 described as above. For example, Usuables[0,0,0] would mean agent has no usable special cards, Usuables[0,1,0] means agent has a card with value 2 that is usable. Here the index of the list attached to 'Usables' corresponds to predecessor of special card's value.

Valid States

A valid state can be completely represented given a tuple of 5 values in their respective ranges as:
(*agent_score*, *dealer_first_card*, *agent_usable*[1], *agent_usable*[2], *agent_usable*[3])
where 0<=*agent_score*<=31, 4<=*dealer_first_card*<=13, 0<=*agent_usuable*[i]<=2 for i={1,2,3}, and all variables are integers.

Non-actionable states

States where

- *agent_sum*=31,4<=*dealer_first_card*<=13, 0<=*agent_usuable*[i]<=2 for i={1,2,3}
    - The only rational action in this case would be to stick

- *agent_sum* < *dealer_first_card*, 4<=*dealer_first_card*<=13, 0<=*agent_usuable*[i]<=2 for i={1,2,3}
    - This implies that *agent_sum* is anywhere between 0 to 12, in which case agent should always hit since the probability of getting a positive card (2/3) is more than getting a negative card (1/3) and agent loses anyway even if the dealer sticks at first card.

All the terminal states are also non-actionable since at the end of the game, no action can be taken by the agent except stick (implicitly, since the game cannot continue). Consider the following scenarios:

- *agent_sum*<0 or *agent_sum*>31: The agent is busted and cannot take any action
- *agent_sum*>0 and *dealer_first_card*<0: The agent wins by default, the agent should always stick

II. Simulator

Simulator is the class *Environment* that models the Achieve 31 BlackJack environment. The *Deck* and *Dealer* are a part of the environment. *Dealer* has a fixed policy and the *Deck* is assumed to have infinite collection of cards as mentioned in the assignment statement.
At the start of the game, both the *Agent* and the *Dealer* draw a card from the *Deck*. The agent starts taking actions first, and once it sticks, the dealer's policy is executed. If at any point, the agent's score becomes less than 0 or greater than 31 then it goes *bust* and receives a reward of -1. A reward of -1 is also given when at a terminal state, none of them have gone bust and agent's score is less than dealer's score. The agent receives a reward of 1 when either the dealer goes bust or agent does not go bust and has more score than dealer at the end of a game.
The *reset()* function gives the initial state of the environment where both agent and dealer have a card each. The *step()* function gives the next state, reward and a boolean representing end of game.

III. Policy Evaluation (Model-Free)

Policy:

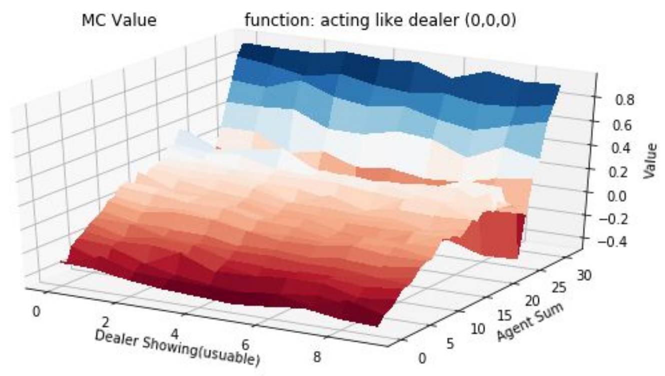
```
if score is greater than 25 then stick
else hit
```

(1) Monte Carlo Evaluation

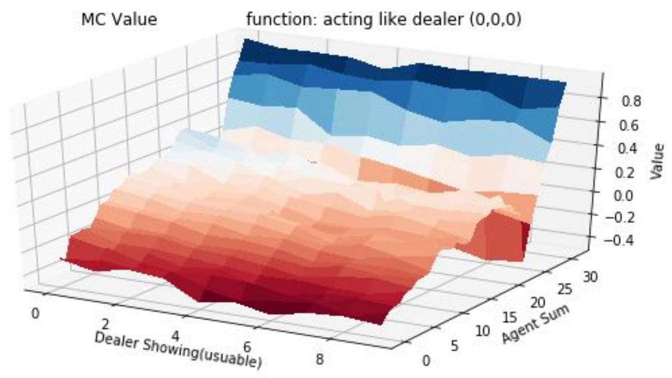
The following graphs of value functions for every valid state (with usables=[0,0,0], i.e. when no special card occurs), were obtained after evaluating the above mentioned policy on 100, 1000, and 100000 episodes. A value of *x* on the dealer showing axis represents that the dealer has card of value *x+4*. These show the evolution of value function with increase in number of episodes. This demonstrates the fact that MC requires more episodes to converge because the samples have high variance.

The values in blue color represent higher values of value function while red ones represent low ones. The graphs show that it is good to have a big sum and act according to this policy, but it is not good for small values of sum. The darker red positions for low agent sum and high dealer first card show that those are bad states.
However, when the agent sum is high(greater than ~28-29), no matter what the dealer showing card is, all the states have high value functions. This is because, in these states, the agent already has a high sum and always chooses to stick, hence it does not go bust and has a large probability of getting more sum than the dealer, hence winning the game and getting reward of 1.
Another notable fact is that the value function, increases till agent sum in around 17, after which it dips till 25 and rises again. This dip is sharp for values where dealer showing card is greater than 9, in which case the dealer may have a usable card and hence has higher chances of getting a larger sum and consequently winning the game. For the same value of agent sum, ~17, the value function has higher value if dealer showing card is smaller in value, which makes sense since if starting card of the dealer has a low value, then it has slightly lesser chance of having sum greater than the agent, given that agent already has a decent sum.

Every Visit v/s First Visit
Figure 3.1 shows comparison between value functions learnt by *every-visit* and *first-visit* MC evaluation. The graph for *every-visit* is smoother than the *first-visit* one which may be attributed to more updates for a state in case of *every-visit*. There is not much difference in the plots and algorithms otherwise, except that *every-visit* may converge faster.



(a) Every visit

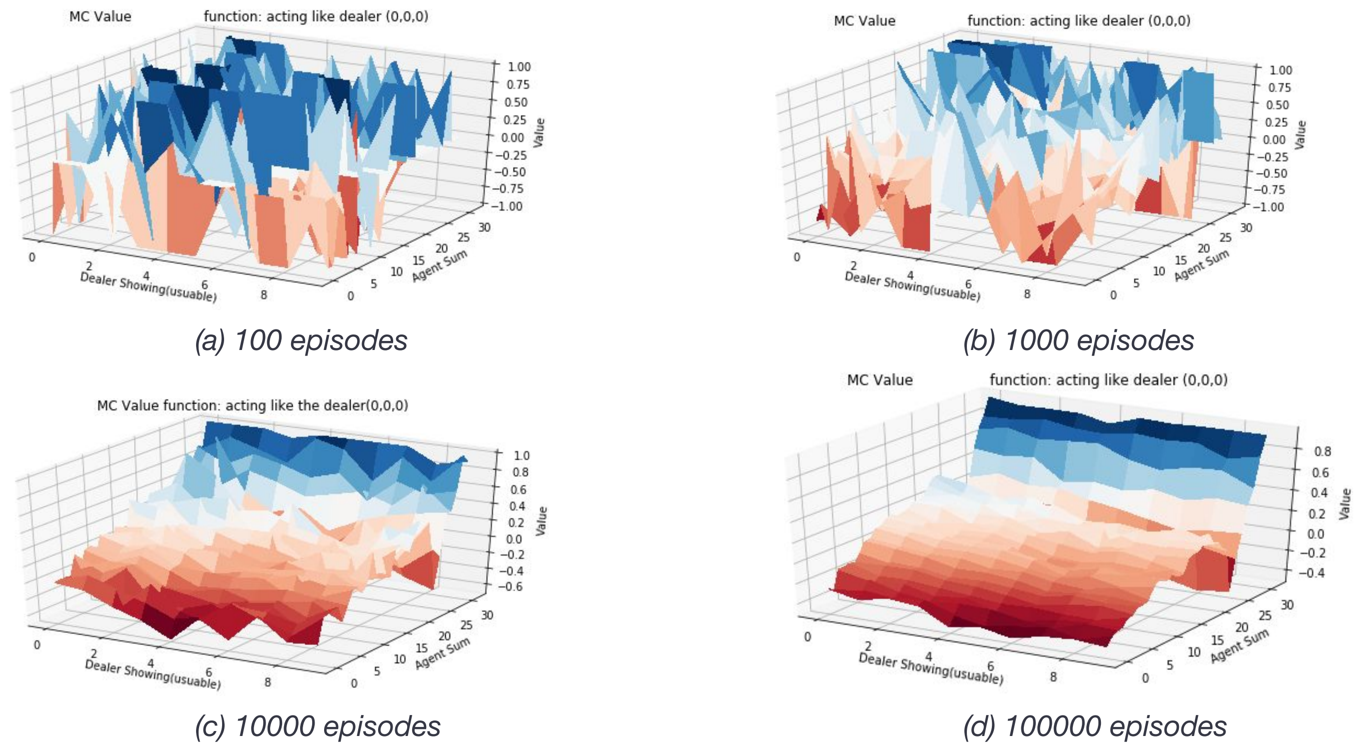


(b) First-visit

Figure 3.1

The Figure 3.2 and 3.3 show change in value function with increasing number of episodes for *every-visit* and *first-visit.*

<u>Every Visit:</u> *Change of Value Function with Number of Episodes*

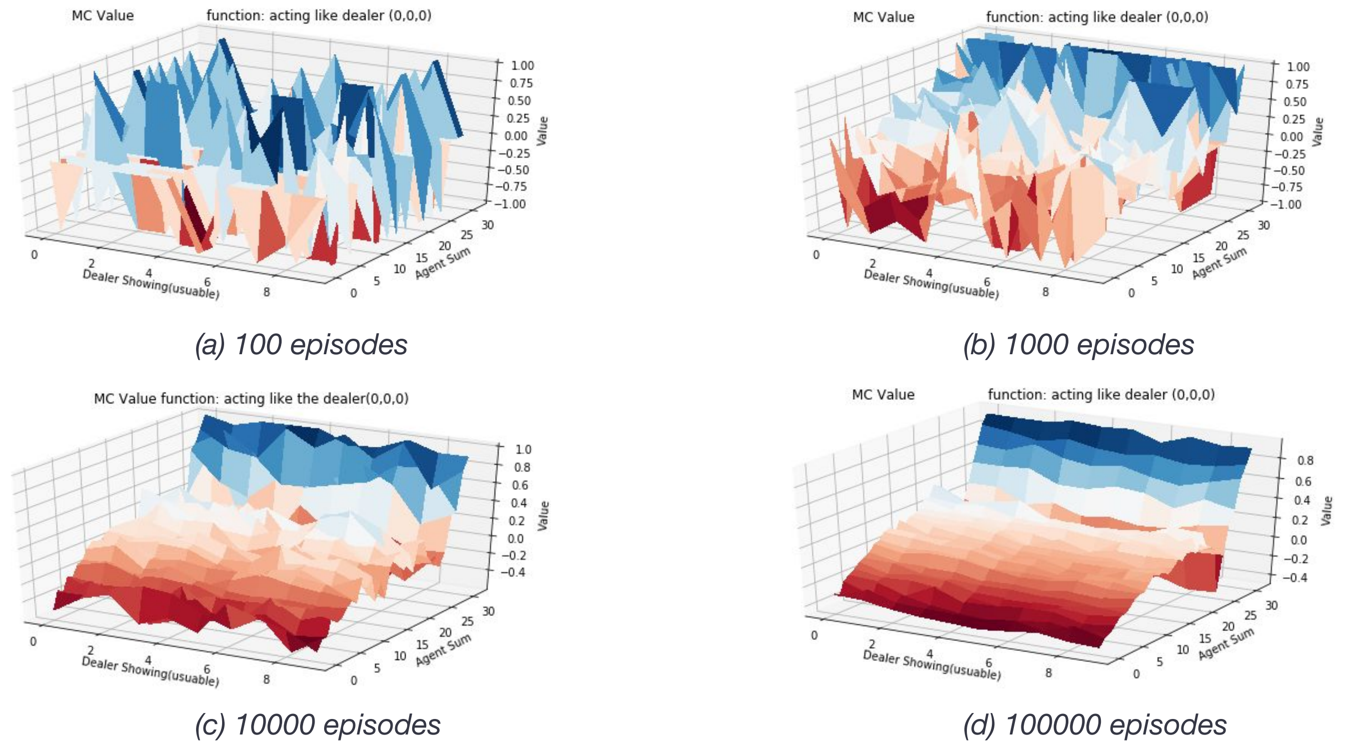


(a) 100 episodes *(b) 1000 episodes*

(c) 10000 episodes *(d) 100000 episodes*

<u>Figure 3.2</u>

<u>First Visit</u>: *Evolution of Value Function with Number of Episodes*



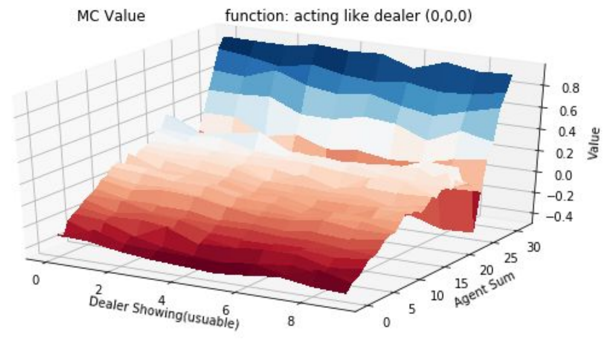
(a) 100 episodes *(b) 1000 episodes*

(c) 10000 episodes *(d) 100000 episodes*

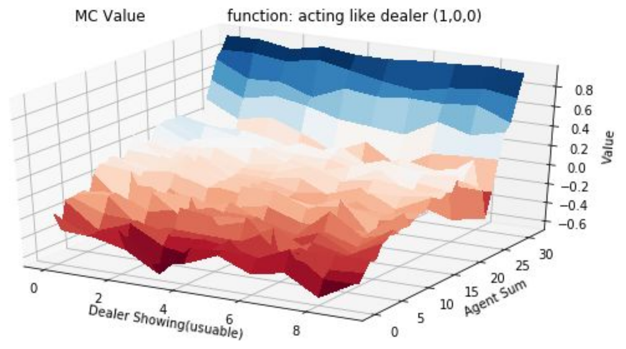
<u>Figure 3.3</u>

Figure 3.4 shows Value Functions for some of the Usuables. Since there are a total of 27 Usuables, only 4 of them are shown here for analysis.

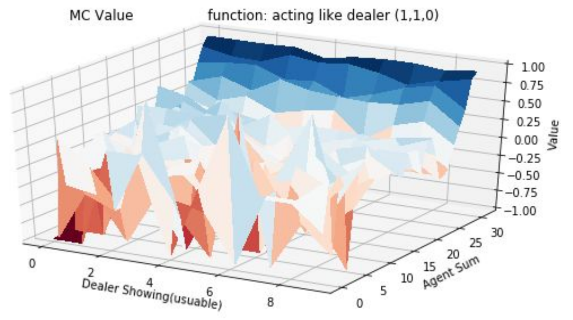
Value Function for different Usables



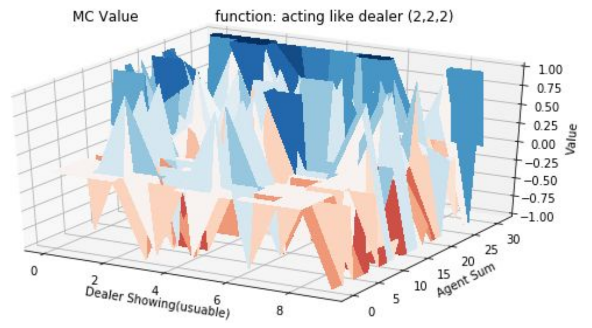
(a) Usuables[0,0,0]



(b) Usuables[1,0,0]



(c) Usuables[1,1,0]



(d) Usuables[2,2,2]

Figure 3.4

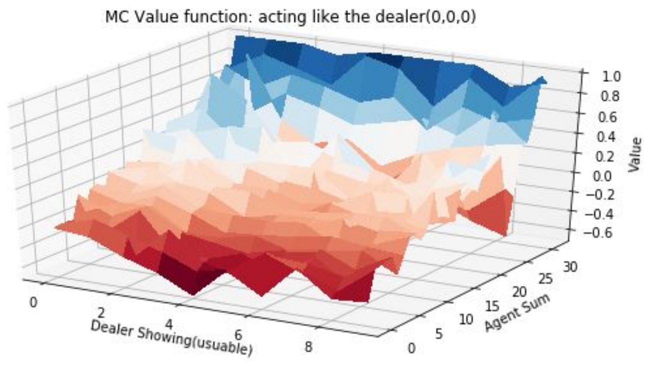
When there are no usable special cards, as mentioned earlier, the value is lower for low agent sum , that is less than ~17 and high for greater than ~25. For higher values of dealer showing card, the value becomes low even at higher agent sum indicating that dealer may win if it has a special usable card at the starting.
For the case when the agent has one usable card(in this case, 1, used as 11), the value becomes higher at all points as compared to no usables at all. This makes sense since having a usable card means player has a higher positive value card and hence more probability of scoring greater than the dealer and hence winning the game. Note that the value function still dips at for higher values of dealer sum and for agent sum ~25 since , however, its value if higher than before.
When the agent has two usable cards, it has a very large probability of winning since it has a large sum, i.e. 23(11+12) in this case and whenever it will hit, it will most likely get a black card with value more than 2, as black is sampled with higher probability and 1-10 values are sampled uniformly, which will lead to making its sum greater than 25. Hence, all the states in this case, for agent sum greater than ~11 have high value function.
Coming to the last case, where all the special cards are non-usable, i.e. using either of them will make us bust or the occurrence is not the first occurence. We see that the graph is not much interpretable except that it has high values for low dealer states and high agent sum, which are the most easy to learn. It is also noticed that the graph is not smooth in the last two cases. This is because there are not enough updates for these cases as getting two usables or all non-usuables is not very likely.

(2) k-step TD

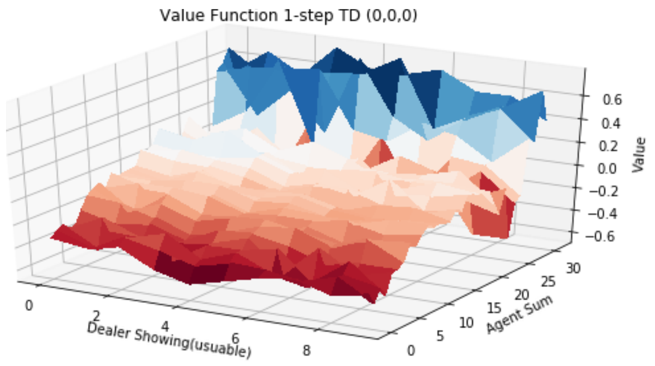
Figure 3.5 shows the value function from MC evaluation and 1-step TD evaluation after running on 10000 episodes. It can be noticed that 1-step TD has already learned the dip part of the value function that occurs around 20-25 agent sum.

Because of the presence of red cards, the whole sum can move backwards, so the episodes in this case, Achieve 31, can last longer than the ones in regular Blackjack, where we would expect shorter episodes. Hence, in this case, MC can take longer to converge, so we should prefer bootstrapping for learning the value function. As was noticed during the experiments, TD evaluation converged around 5000 episodes, while for MC, it took 100000 episodes.

MC v/s one step TD



(a) MC



(b) 1-step TD

Figure 3.5

Figure 3.6 and 3.7 show the change of value function with respect to number of averaged runs for 1-step and 3-step TD. Graphs corresponding to different values of usables have been shown.

It is noticed that all the graphs with 1000 averaged runs are smoother than the 100 averaged runs for the same value of k. This is because of the variance in value functions obtained from each run. Since the TD estimate is biased, and depends on our initialisation of V, using more runs to average out brings it closer to the actual value function.

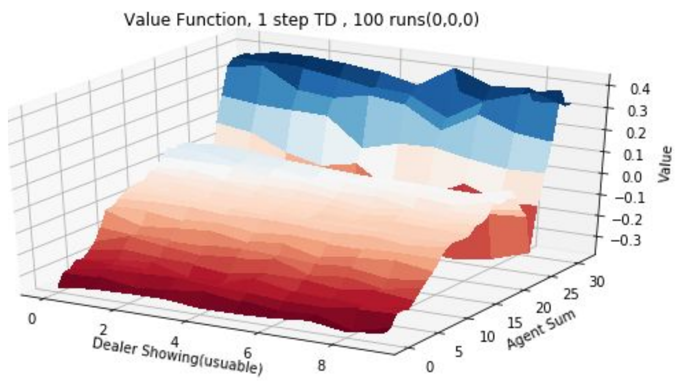
It is interesting to note that 1 and 3 step TD are able to learn the value function for usuables[2,2,2] which were very difficult to learn due to as the probability of visiting this case is quite low, and the function is quite smooth after averaging over 1000 runs.

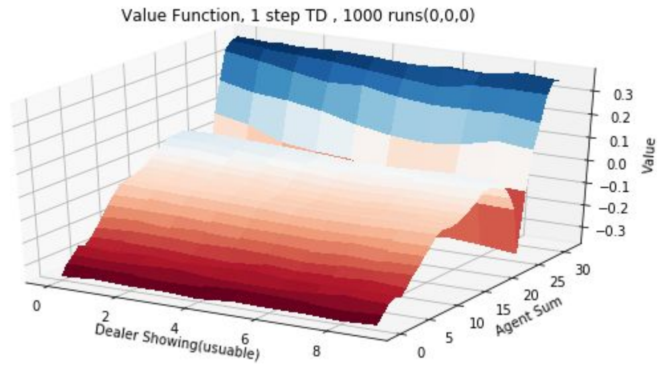
The value function in case of usuables[0,0,0] is as expected. In case of having usuables, it is observed that the value function is higher as compared to no-usuables case and constant for all states before ~23. The higher value is intuitive since we have higher sum and hence higher probability of winning. For the case of all non-usables, we see that the value function is nearly zero for smaller values of agent sum, lower for a narrow range around 25 agent sum and then positive again.

Value function after averages over 100 and 1000 runs, trained over 5000 episodes

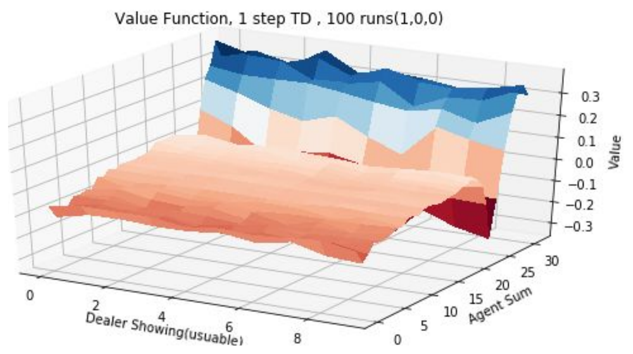
<u>k=1</u> **100 runs** **1000 runs**

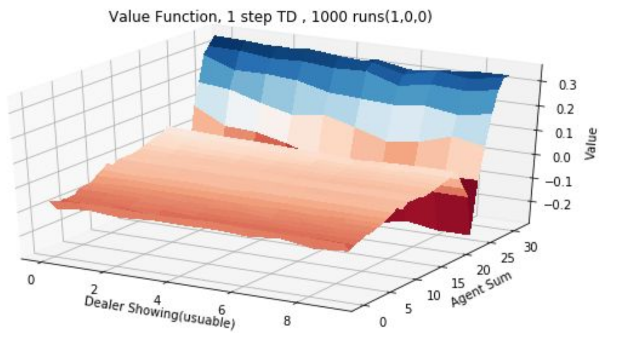
Usables=[0,0,0]



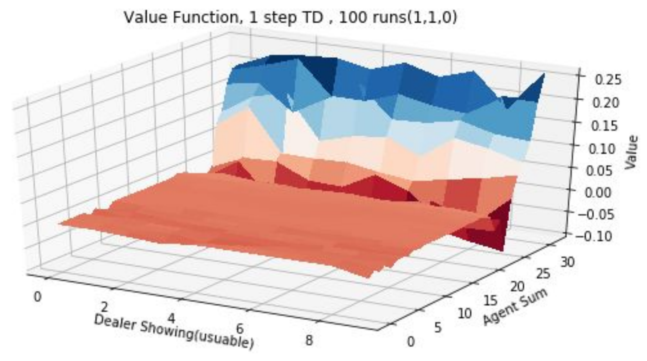


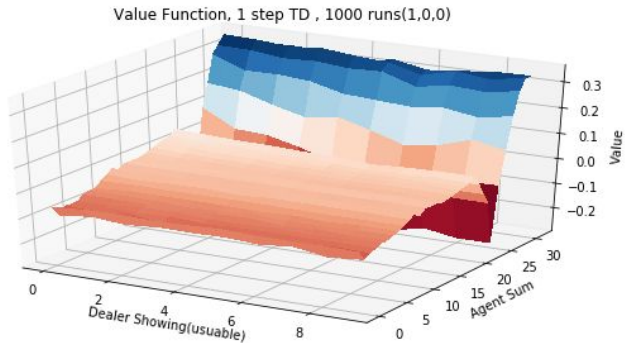
Usables=[1,0,0]



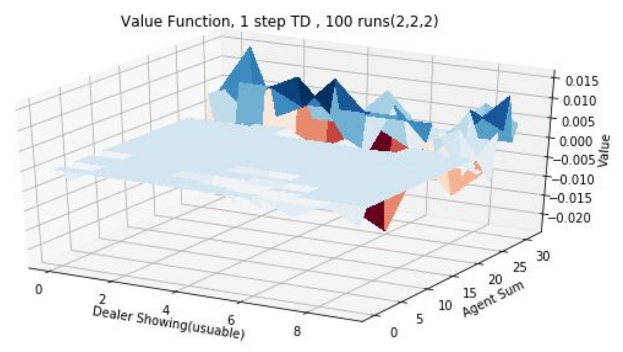


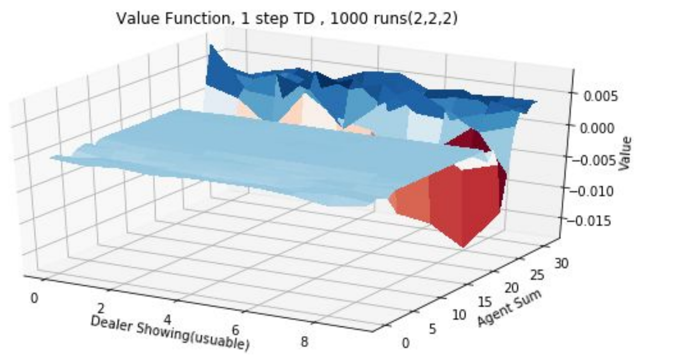
Usables=[1,1,0]





Usables=[2,2,2]



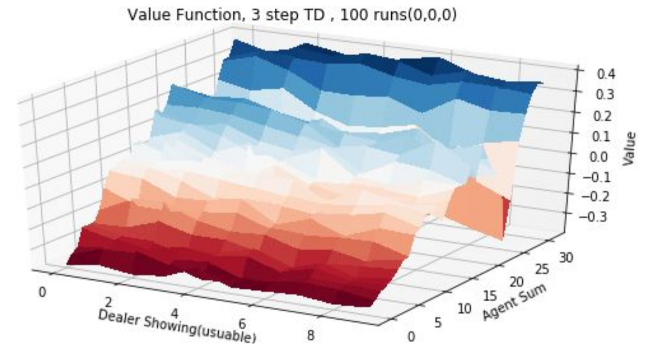


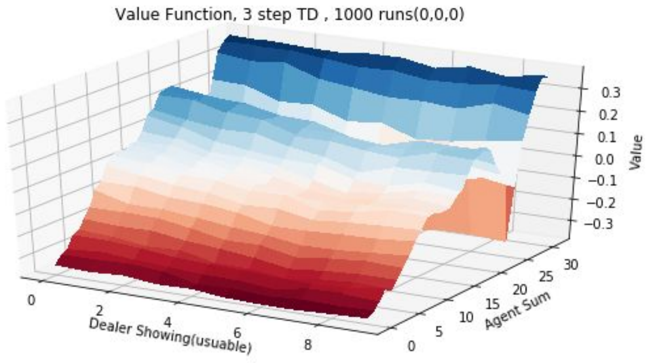
<u>Figure 3.6</u>

k=3

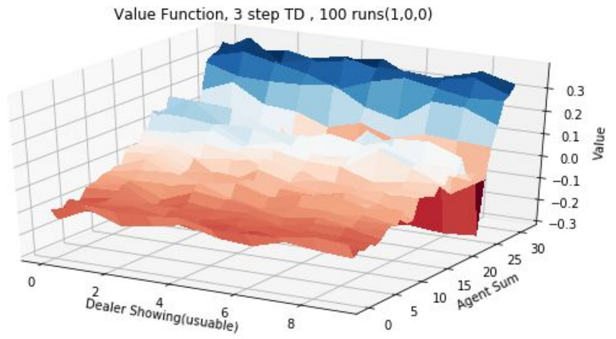
100 runs **1000 runs**

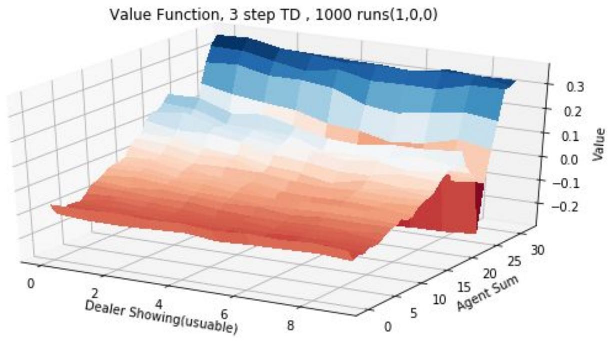
Usables=[0,0,0]



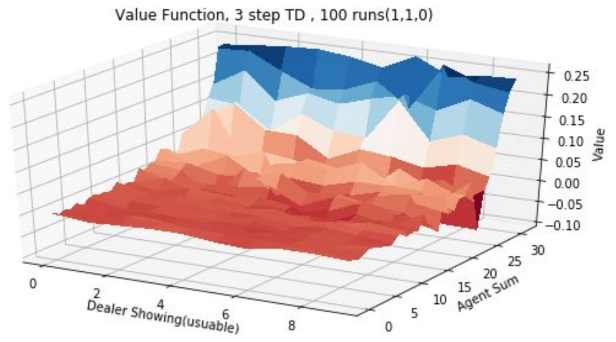


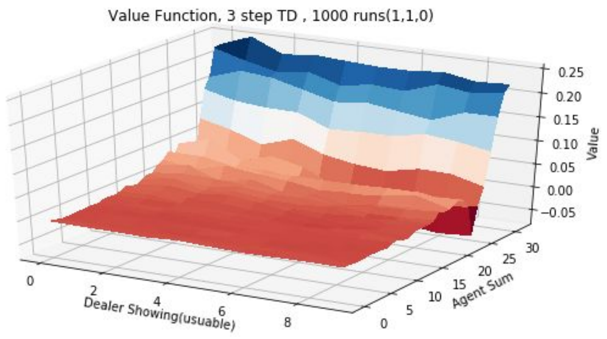
Usables=[1,0,0]



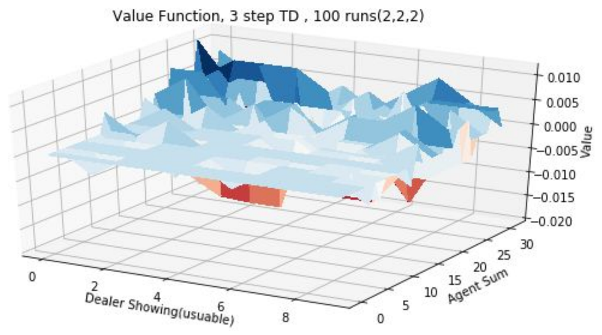


Usables=[1,1,0]





Usables=[2,2,2]



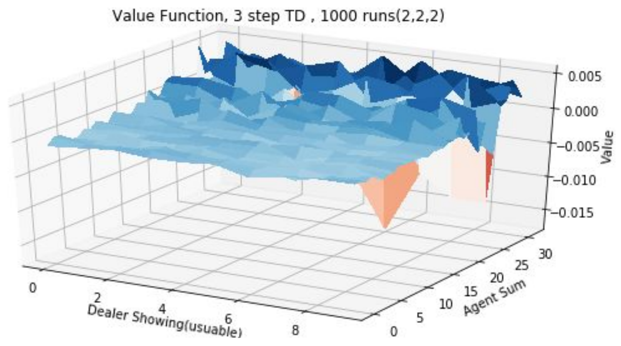
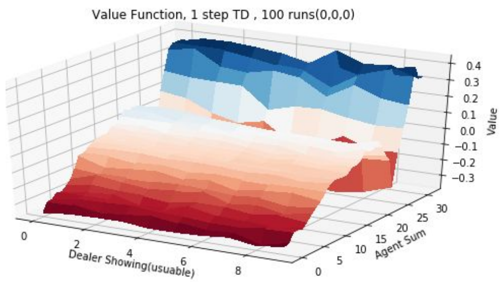
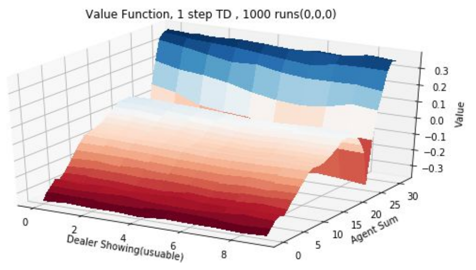


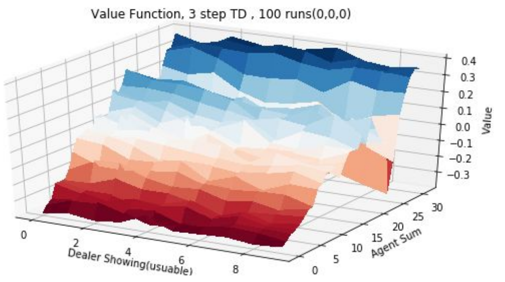
Figure 3.7

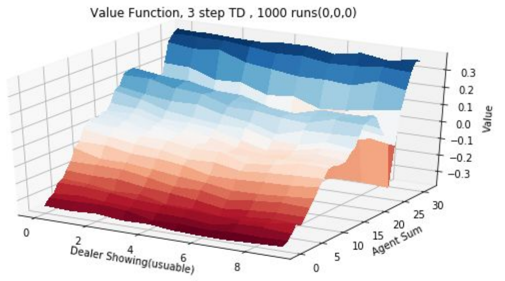
100 runs **1000 runs**



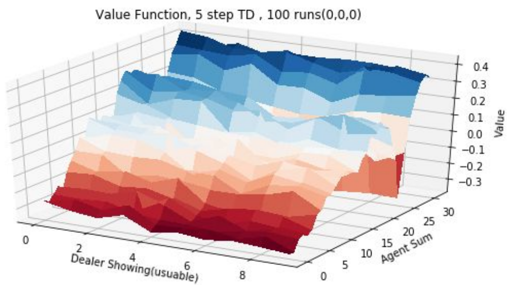


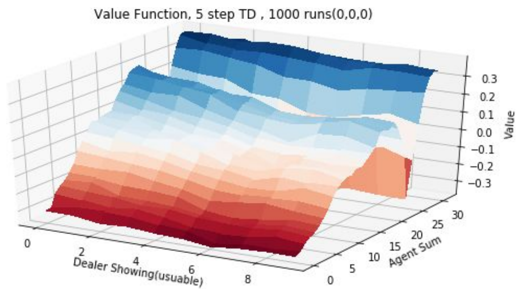
k=1



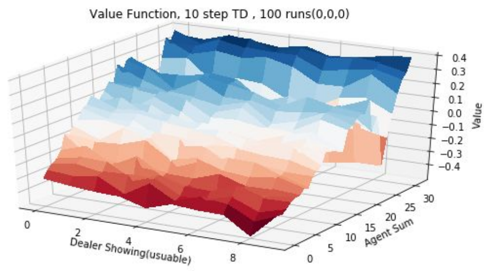


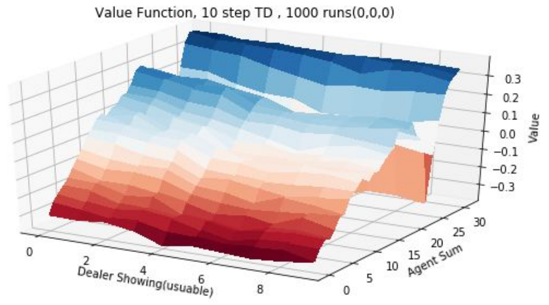
k=3



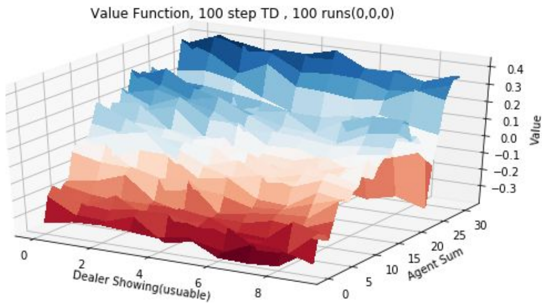


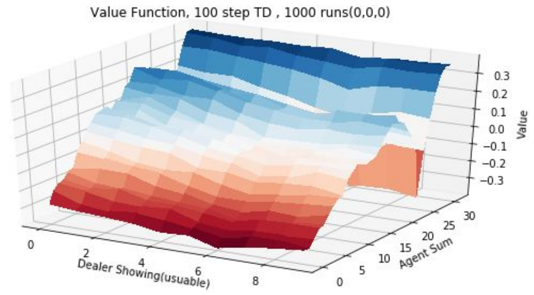
k=5



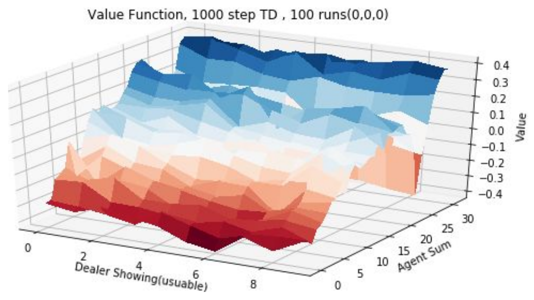


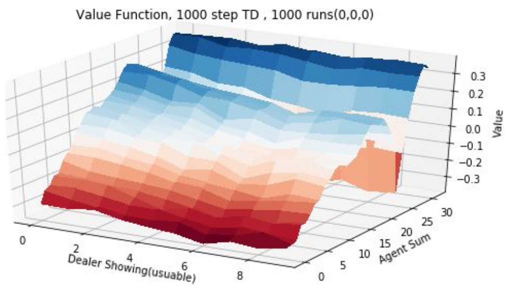
k=10





k=100





k=1000

The above figure shows the variation of value function with increasing step size of TD for 100 and 1000 averaged runs.
The plot is smoother for same k and more number of runs.
With increasing k, the plots become more bumpy. Thi is because as we take more samples from the environment with more k, it induces variance due to the samples. With increasing k, the updates become more like that in MC. Hence, we get the smoothest curve for 1 step TD with 1000 runs. We also notice that the graphs are very similar for k=100 and k=1000. This is because our episode length being smaller than 100, k=100 and k=1000 are same as Monte Carlo evaluation.

IV. Policy Control

A. Algorithms

For the purpose of all the experiments, ∈ = 0.1 and α = 0.1 were used.
The following algorithms were used for policy control:

a. 1 step SARSA
b. 1 step SARSA with decaying epsilon
c. 10 step SARSA
d. 10 step SARSA with decaying epsilon
e. 100 step SARSA
f. 100 step SARSA with decaying epsilon
g. 1000 step SARSA
h. 1000 step SARSA with decaying epsilon
i. Q Learning
j. TD(λ)

B. Performance Analysis

Average Rewards v/s Number of episodes

a. All algorithms

TD(λ) performs significantly better than all other algorithms.

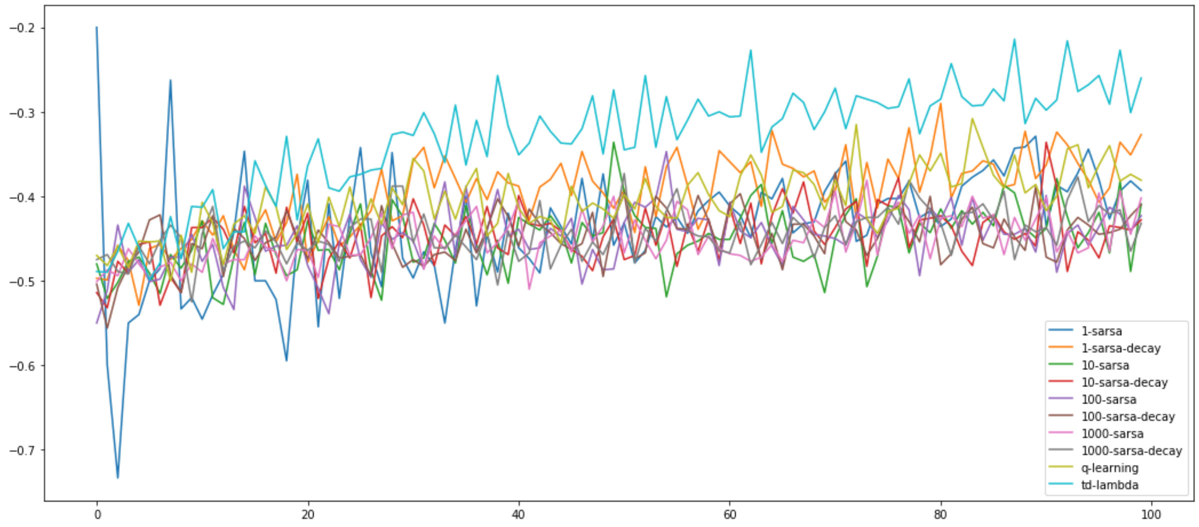


Figure 4.1

b. k-step SARSA

Asymptotic performance is similar. It can be seen that there is high variance in the rewards for small step Sarsa. Variance decreases as k increases and the performance becomes equivalent for k=100 and k=1000 since all the episodes are less than 100 length.

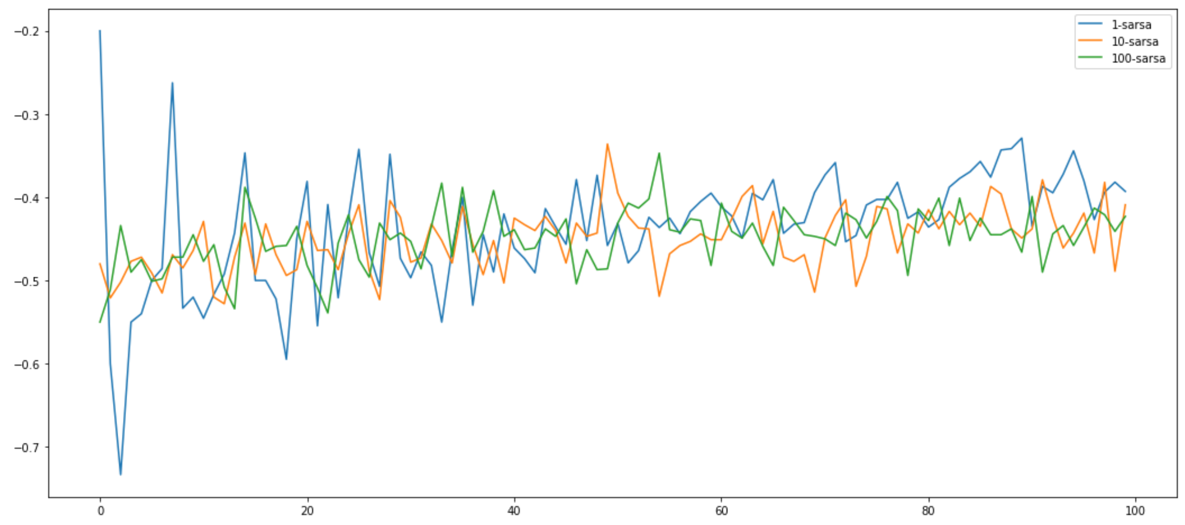


Figure 4.2

c. k-step SARSA with decay

For k=1, Sarsa with decay performs better. But as k increases, performance of Sarsa with and without decay start to converge. This may be because reducing epsilon with time helps in reducing the bias in the updates. But as k increases, we are including more steps from actually interacting with the environment before bootstrapping, hence decaying epsilon does not have much effect.
Hence, k-step Sarsa with and without decay for large k perform similar.

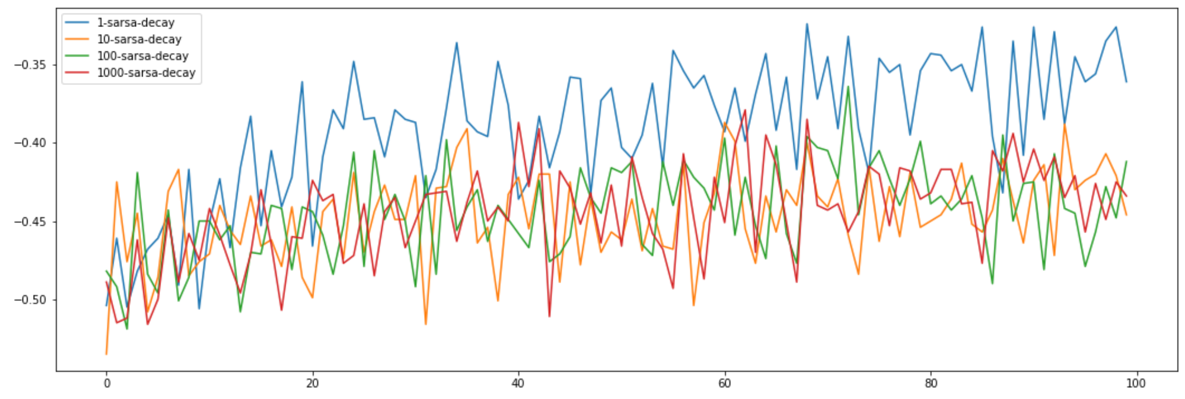
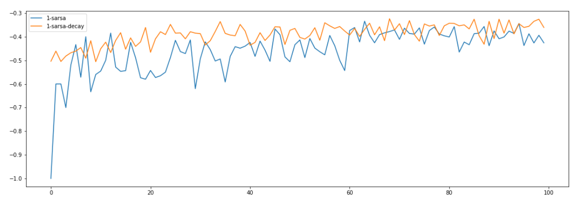
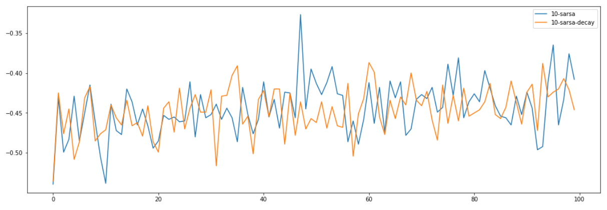


Figure 4.3

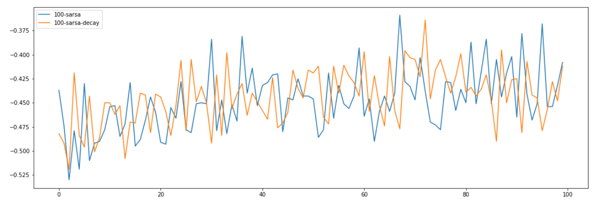
k-step Sarsa decay v/s no decay



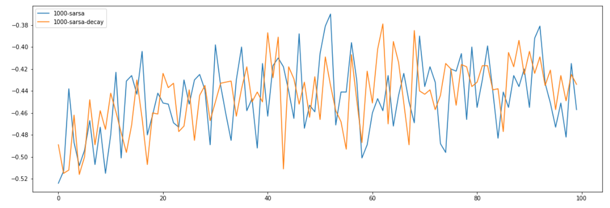
(a) 1 step Sarsa



(b) 10 step sarsa



(c) 100 step Sarsa



(d) 1000 step sarsa

Figure 4.3

d. Q-Learning v/s TD(λ)

TD(λ) performs significantly better than Q-learning. This is because it updates the values of all the steps rather than updating the Q value of just the last step unlike Q learning. It corrects for the updated dynamically and converges to the optimal policy. However, in Q learning we cannot change the Q value of a bad decision we might have made earlier.

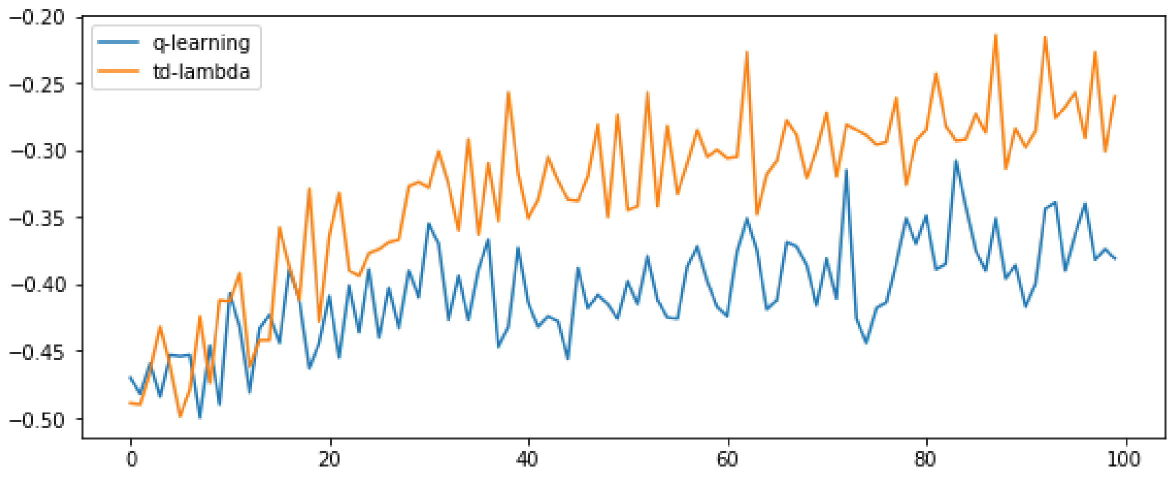


Figure 4.4

C. Change in performance with learning rate (*α*)

Figure 4.5 shows the variation in performance for all the policy control algorithms described in IV(A) with increasing learning rate.

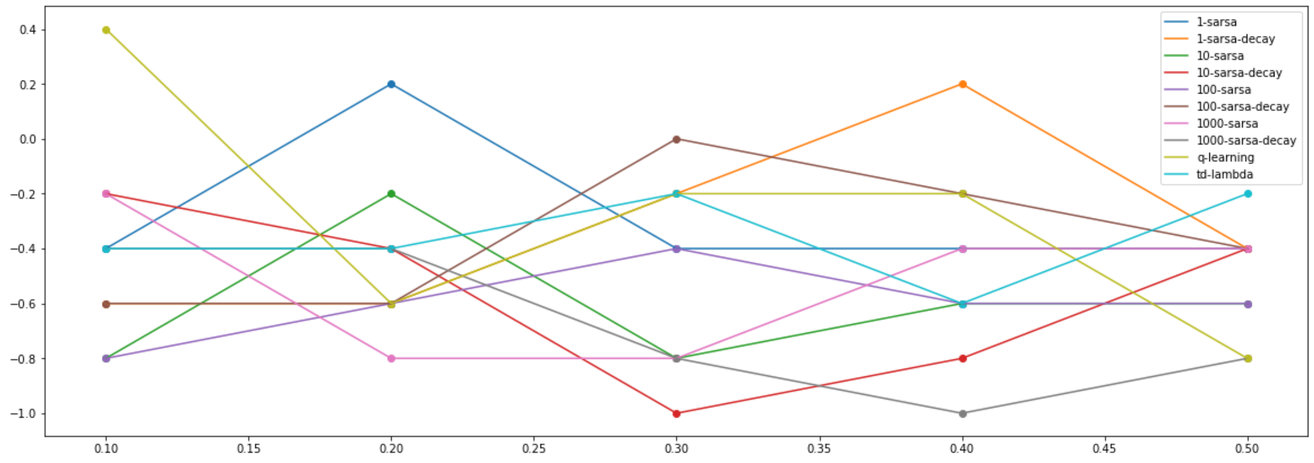
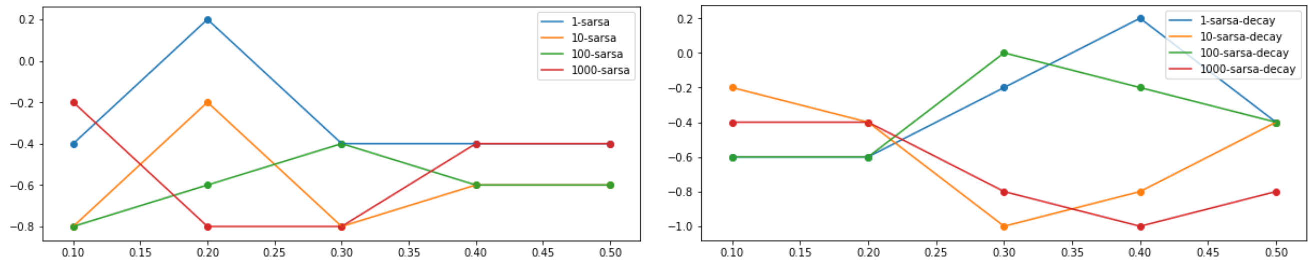


Figure 4.5



(a) Sarsa *(b) Sarsa with epsilon decay*

Figure 4.6

Sarsa:

On increasing the step size of Sarsa(without decaying epsilon), the optimal value of α increases; while on increasing the step size of Sarsa with decaying epsilon, optimal value of α decreases.

When we are decaying epsilon, we are tending more towards the greedy policy, which is less stochastic in nature, and tends to increase the variance of the updates. The rewards we get now can be seen to be similar as obtained in expected sarsa. Thus, performance gets better with smaller values of alpha.

Sarsa has high variance for more step size, due to which, at larger learning rates, larger learning rates help in evening out the variance. If smaller learning rates are used, the algorithm may take a lot of time to reach the optimal policy due to higher variance in updates (while epsilon is constant).

Q-Learning v/s TD(λ):

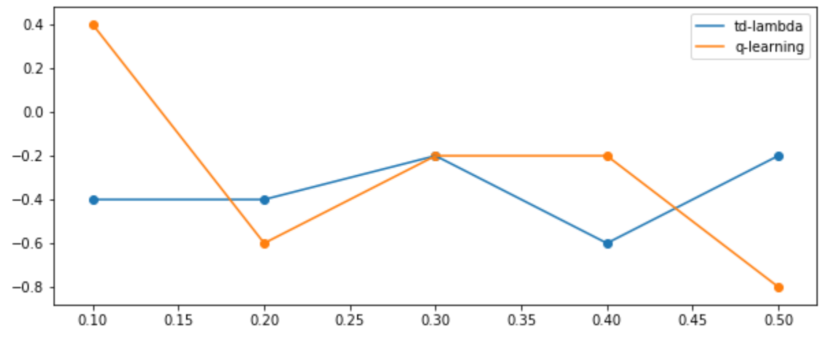


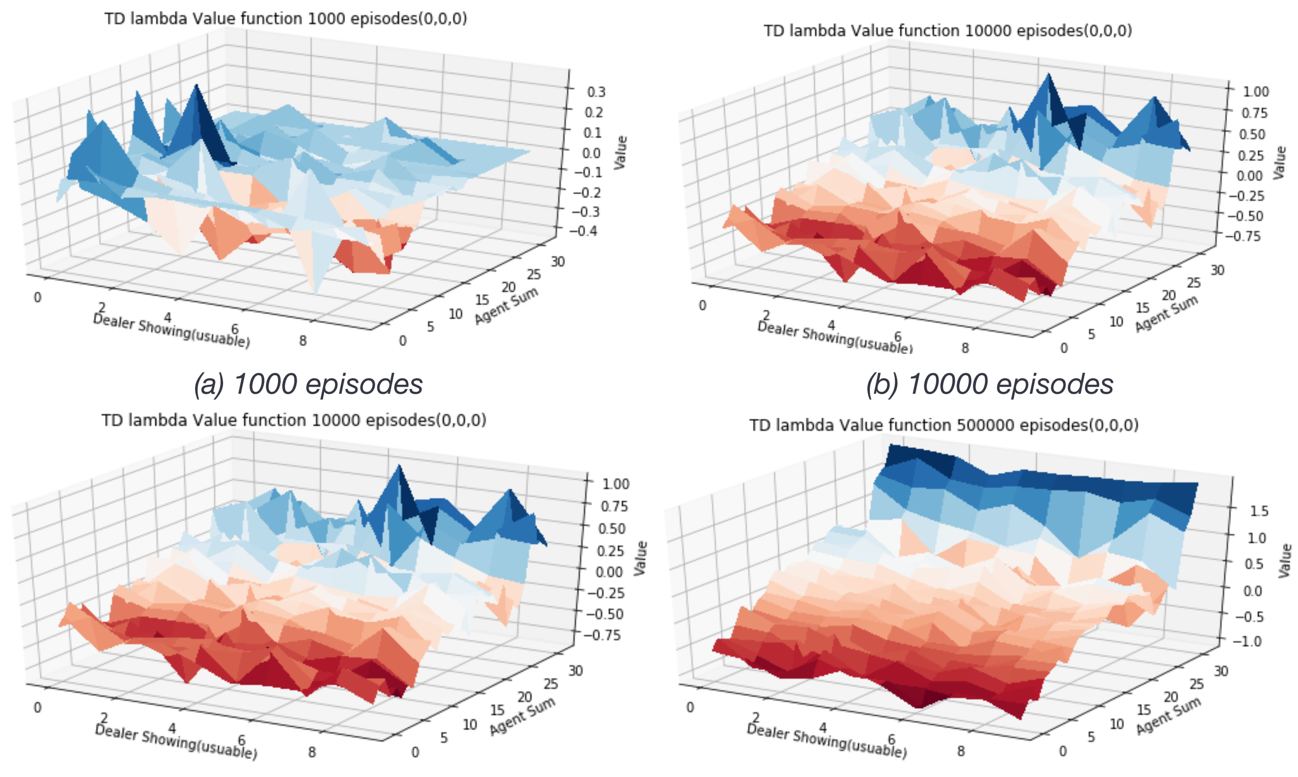
Figure 4.7

In case of Q-learning, the smallest value of α=0.1 is the optimal.
For TD(λ), values of α do not significantly affect the performance which remains nearly the same for all α.
It can be inferred that Sarsa and Q-Learning are quite sensitive to values of learning rate whereas TD(λ) is not.

D. Value function for TD(λ)

Figure 4.8 shows the change in value function while running TD(λ) policy control with increasing number of episodes, where convergence is around 1M episodes.

Change in Value function with number of episodes

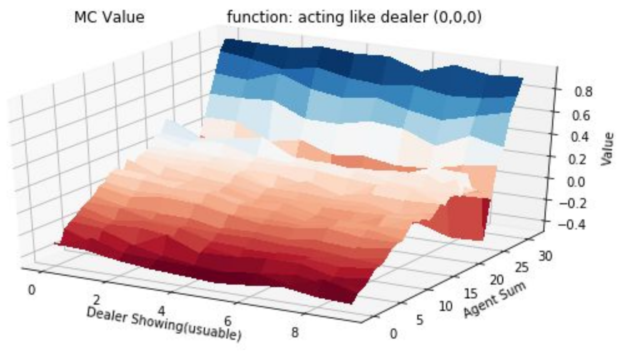


(a) 1000 episodes *(b) 10000 episodes*

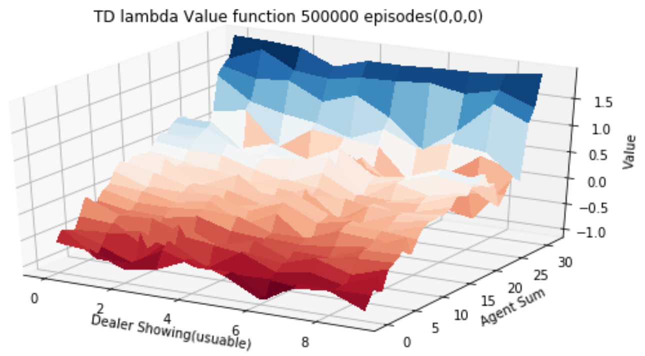
(c) 100000 episodes *(d) 1000000 episodes*

Figure 4.8

Difference between Dealer Policy and Policy learnt from TD(0.5)



(a) Value function of Dealer Policy

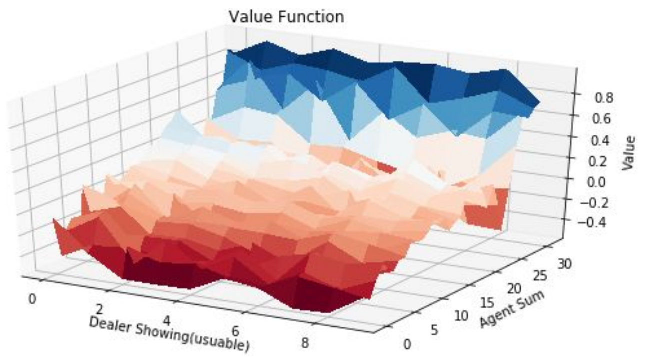


(b) Optimal value function

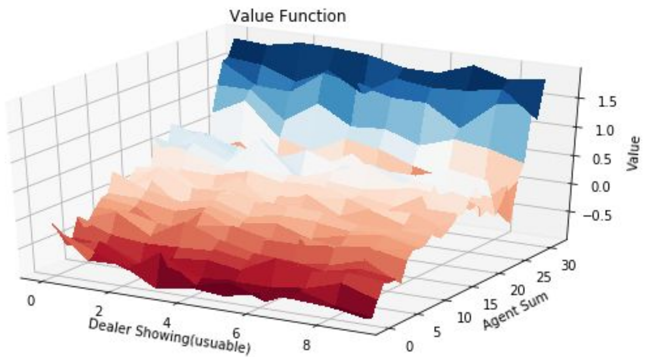
Figure 4.9

Figure 4.9 shows the difference between the value functions (usables[0,0,0]) of policy of the dealer (say π), that was evaluated in Section III, and the optimal policy (say π*) as obtained from TD(0.5) with 0.1 learning rate and decaying epsilon, starting from 0.1.
It can be noticed that Vπ has a sharp dip when agent sum is in between 20 to 25 whereas there is no such dip in the optimal value function (V*).
While Vπ linearly increases for agent sum from 0 to 15 for all dealer showing cards, V* increases in a curved manner. It only has high negative values when agent sum is very small unlike Vπ where high negative values occur near 23.
We can also see that there is a gradient from small dealer showing card to bigger values in V*, which is similar to that in Vπ, which makes sense as dealer will have a higher probability of winning is it has a special card. V* is higher for larger agent sum values as compared to Vπ.

Q Learning v/s TD(λ)



(a) Optimal Value function from Q-learning



(b) Optimal Value function from TD(0.5)

Figure 4.10

Figure 4.10 shows the difference between optimal value functions obtained from Q-learning and TD(0.5). Q-learning seems to learn a policy that is better than the dealer policy but worse than one learned by TD(0.5), as is clear from the more negative values for higher agent sum.

Note:
The function for plotting value functions has been adapted from *https://github.com/mari-linhares*
All the plots can be found here:
https://owncloud.iitd.ac.in/nextcloud/index.php/s/oRN4czAbnKePrNR