COL870: Reinforcement Learning
Assignment 1

Arpan Mangal — 2016CS10321

October 2019

1 State Space

There are two sets of states, one that the agent needs for it's decision making process, and one that the environment needs and maintains to track the state of the game.

1.1 Environment State

Environment contains a state representation for the dealer as well as the player.

Firstly we note that we can reduce the actual number of states (where a state is a certain combination of cards in your hand), by representing it as a *minimum_sum* (i.e. the sum obtained by adding the values of the card in our hand, without using any card as a special card), and three booleans representing whether we have a special card of each type or not.

Thus the environment maintains two such vectors (for the player and the dealer), each of the form: $[minimum\_sum, S1, S2, S3]$.

1.2 Player State

The agent state can be reduced further from the environment state, by considering only the information that the player needs to make decisions.

For ex. For the purposes of the agent, once it knows the *minimum_sum*, there is no difference whether it uses 1 as a special card or 2 or 3 as a special card (assuming it has it and it is using a single special card). The agent is only concerned with what is the *maximum_sum* it can get with the cards in hand, and in obtaining that *maximum_sum* how many special cards have it used (the latter is helpful to hedge against *Hit* losses in cases the *maximum_sum* is large.

Thus the agent only concerns itself with the *maximum_sum* it can achieve, the number of special cards it has used, and the value of the dealer card (1-10). (Since the dealer can't have a negative card in beginning, dealer card has value from 1-10 only).

Thus for the purposes of the agent we have $4 \times 32 \times 10$ states. 4 for the number of special cards (0 to 3), 32 for *maximum_sum* (0 to 31), and 10 for the dealer card value.

1.3 Actionable states

For a rational agent, it has a single choice of action in the state where it's *maximum_sum* is 31. Clearly it can't do better, and choosing an action *Hit*, will only lead to a score of less or equal with a fair possibility of getting *bust*.

2 Simulator

Simulator provides the API for the *reset* and the *step* functions, and has private functions for drawing a card randomly, and simulating the dealer's play at the end of the game.

3 Policy Evaluation

3.1 Monte Carlo

Below are the State-value function graphs for each state. As described above there are a total of $4 \times 32 \times 10$ states, which are then divided and plotted in 4 parts of 32×10 states each.

In the graphs, *green* denotes a state value > 0, while *red* denotes a state value < 0.

3.1.1 Every-Visit MC

State Value functions shown in Figure 1.

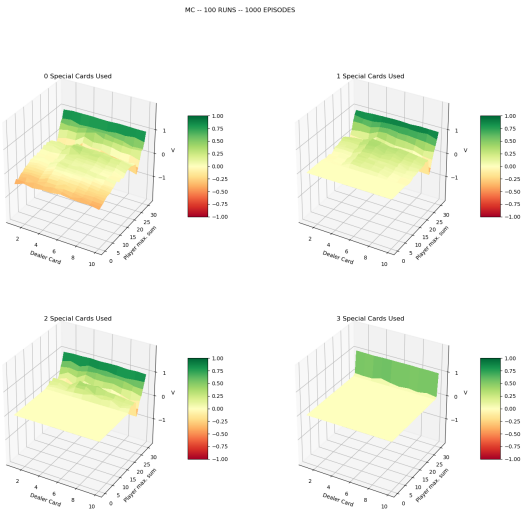
There is little difference by increasing the number of runs, since number of runs just reduce the variance which is already low after 100 runs.

Increasing the number of episodes to 10,000, sees a better estimate of the State Value function.

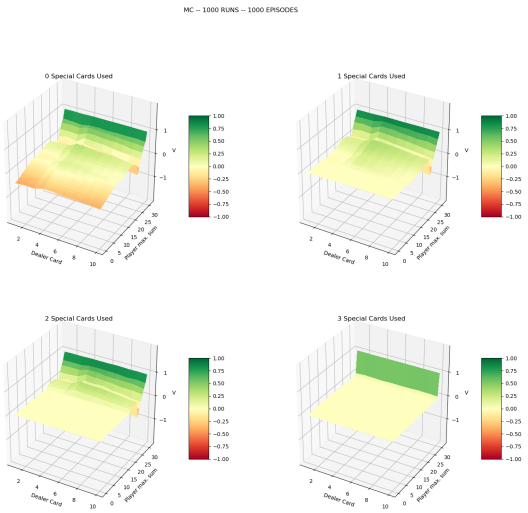
3.1.2 First-Visit MC

State Value functions shown in Figure 2.

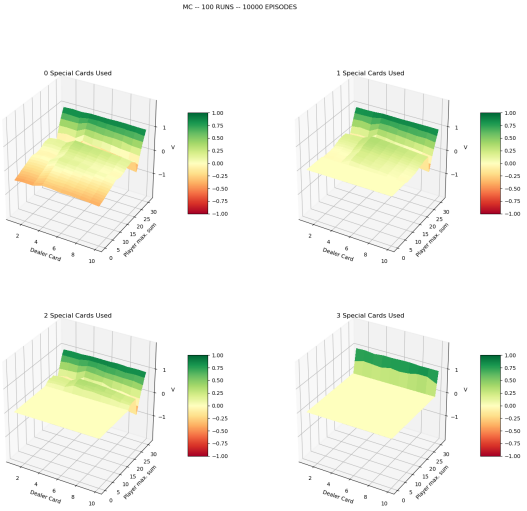
There is hardly any difference between *Every Visit MC* and *First Visit MC* because it's very unlikely that the agent will get the same state in a single episode more than once.



(a) 100 RUNS 1000 EPISODES

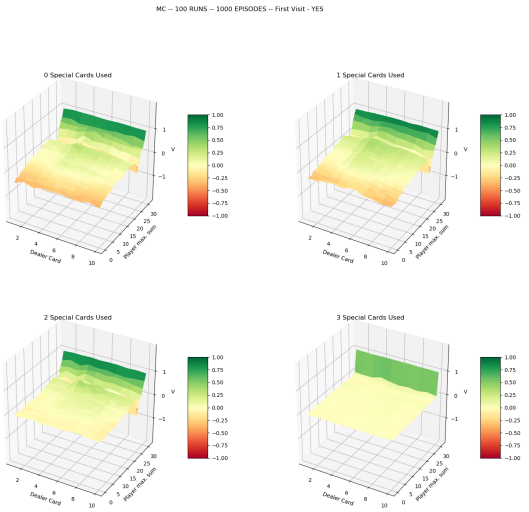


(b) 1000 RUNS 1000 EPISODES

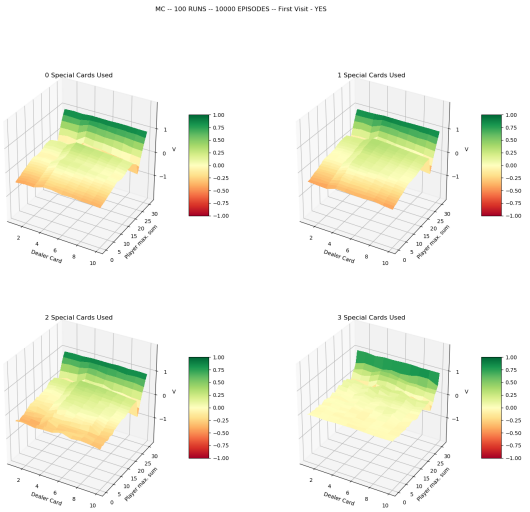


(c) 100 RUNS 10000 EPISODES

Figure 1: Every Visit MC



(a) 100 RUNS 1000 EPISODES



(b) 100 RUNS 10000 EPISODES

Figure 2: First Visit MC

3.2 k-step TD

Figure 3 shows the State Value function graphs for k-step TD with k = 3.

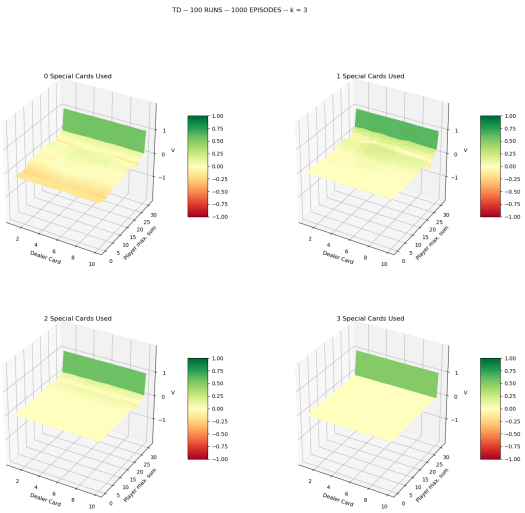
Again, increasing the number of runs doesn't make much difference.

Increasing the number of episodes drastically improves State Value estimate as 1000 episodes is far from sufficient for convergence, in the case of 3-step TD.

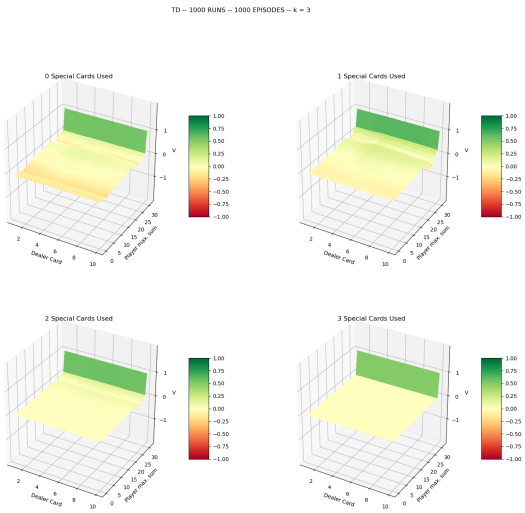
3.2.1 Varying k

Figure 4 shows the State Value functions as we change the value of k. At large values of k, TD essentially becomes similar to MC.

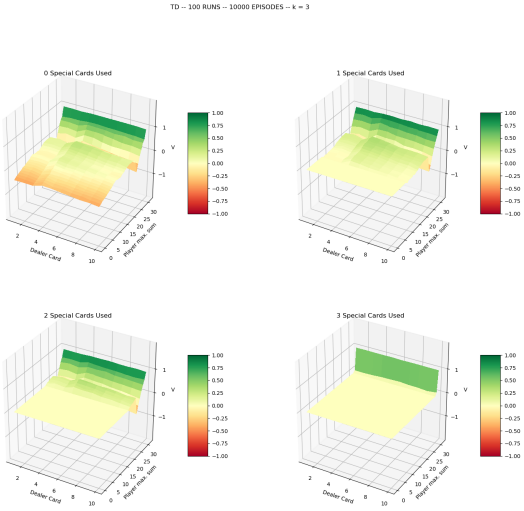
There is small difference in the values as k increases. This difference further becomes less perceivable as the number of episodes increase.



(a) 100 RUNS 1000 EPISODES

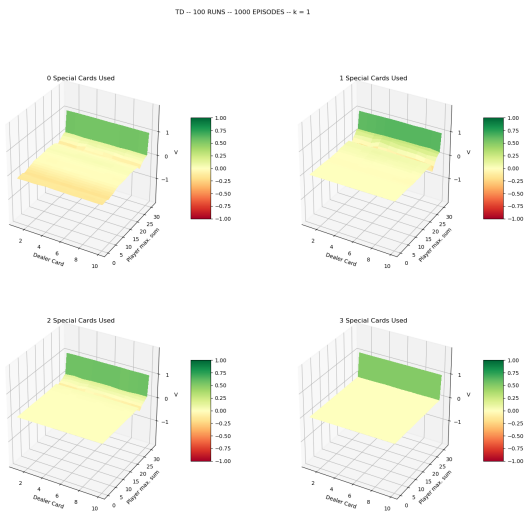


(b) 1000 RUNS 1000 EPISODES

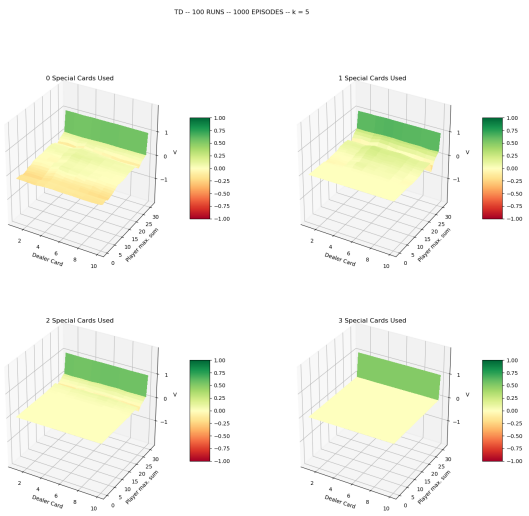


(c) 100 RUNS 10000 EPISODES

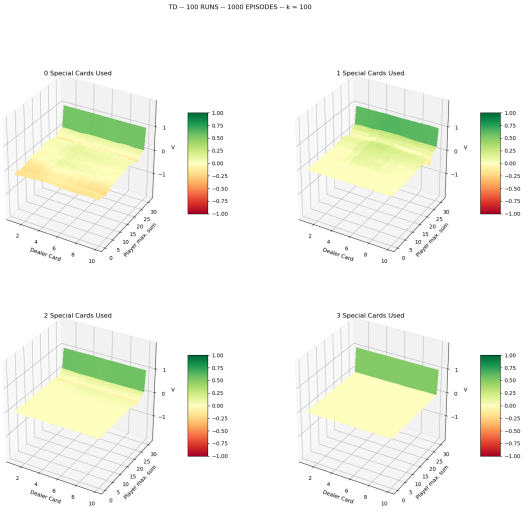
Figure 3: 3-step TD



(a) 100 RUNS 1000 EPISODES — k = 1



(b) 100 RUNS 1000 EPISODES — k = 5



(c) 100 RUNS 1000 EPISODES — k = 100

Figure 4: k-step TD

4 Policy Control

For this part, we maintain a $q(s, a)$ table for maintaing the Q values for each possible state and action combination.

The values in the table are initialised to 0 and then improved using one of the following algorithms:

1. k-step lookahead SARSA
2. k-step lookahead SARSA with decaying epsilon
3. Q-Learning with 1-step greed look-ahead policy
4. Forward view of eligibility traces for TD(0.5)
5. TD(0.5) with decaying epsilon

We train each algorithm for 100,000 episodes, and for various checkpoint number of episodes, we stop and test the model on a separate test case of 1000 episodes to get the average reward.

Next we observe the changes in the average reward as alpha changes.

All these are presented in the following sections.

4.1 Algorithm Learning

Figures 5 and 6 depicts the average rewards (on separate test case) for different learning algorithms as the number of episodes increase.

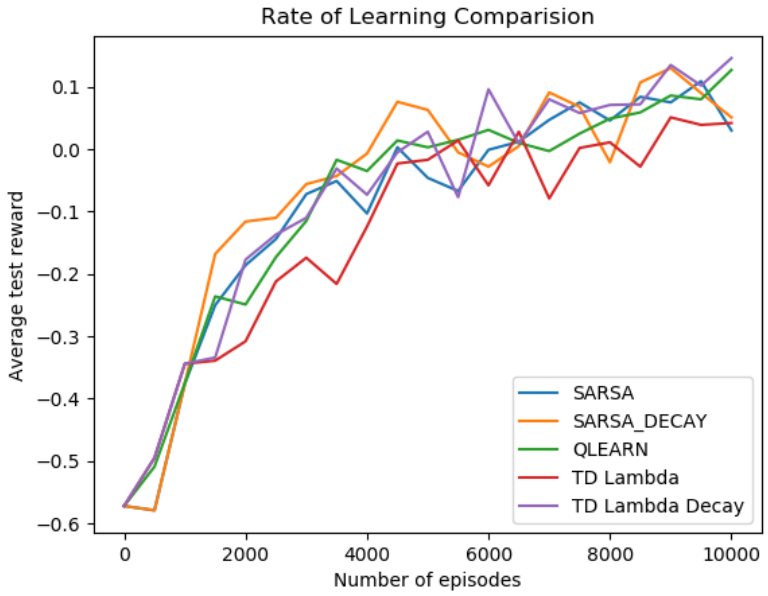


Figure 5: Rate of Learning Comparison

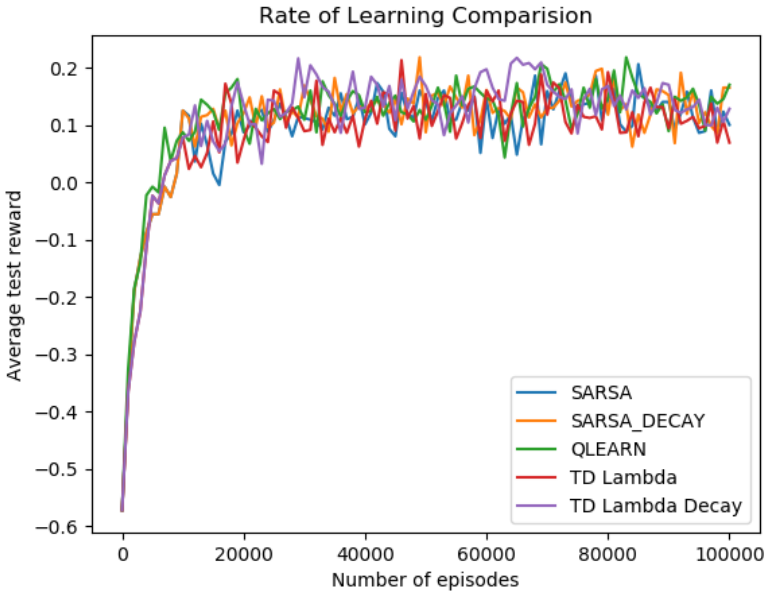


Figure 6: Rate of Learning Comparison

In the initial phases (till around 5000 episodes), SARSA with decaying ϵ, Q-learning and TD Lambda with decaying ϵ gives the highest average reward, while TD Lambda with fixed ϵ lags behind others.

As the number of episodes increases, other methods catch up, giving almost the same average reward with stochastic perturbations.

The reason of decaying ϵ and greedy algorithms performing better initially is that they exploit more than explore, thus getting accurate estimate of the $Q(S, A)$ values which are highly rewarding and are likely to occur in an actual game where a greedy policy is being followed. However subsequently, as the number of episodes increase, all the states are explored sufficiently, and there is not much different between $Q(s, a)$ values of the different algorithms.

4.2 SARSA – Varying K

We compared the performance of k-step SARSA as the value of k is changed. Figure 7 represents this relationship.

We observe that increasing k, leads to a fall in speed of learning and performance, possible reason being the algorithm starts learning more from just the final rewards, and do not care about the learnt state values of the future states. This could lead to high variance in the states which occur rarely in the training set, since they have only a certain number of final rewards to learn from.

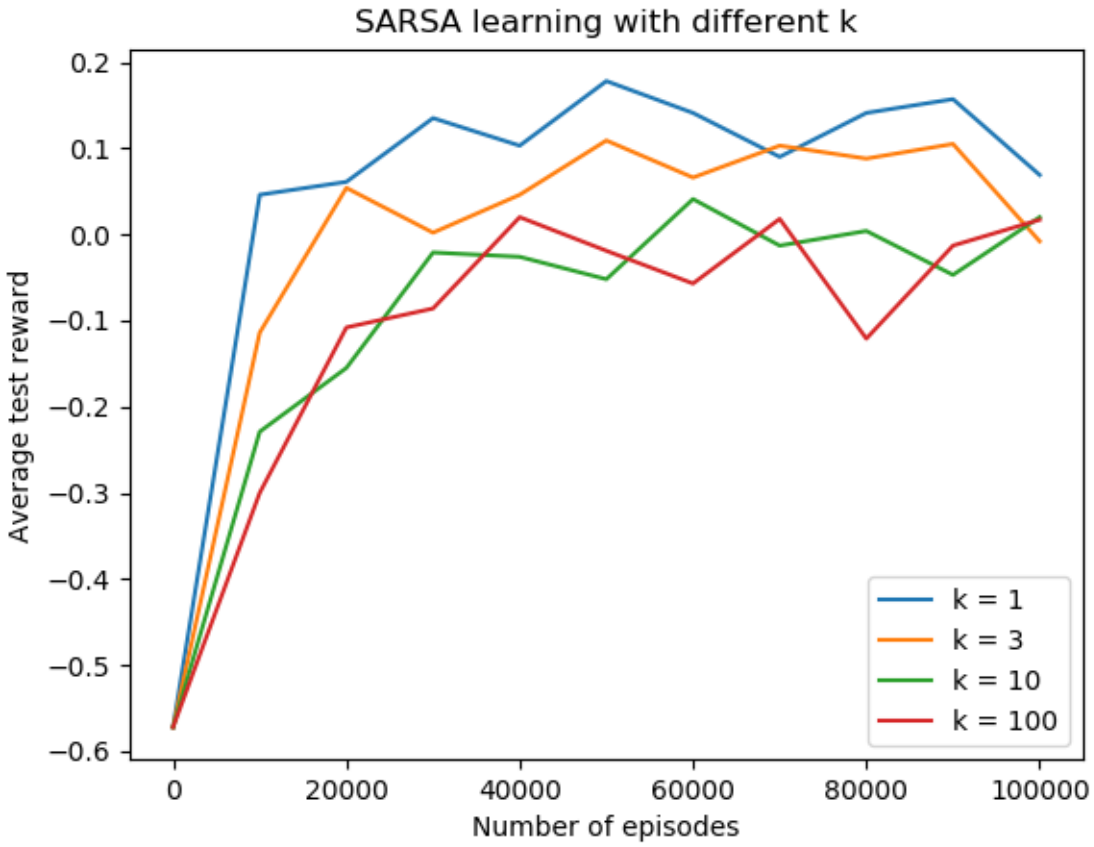


Figure 7: SARSA – Varying K

4.3 Varying α values

We try out the different algorithms with different α values. For this, we train each algorithm for 10,000 as well as 100,000 number of episodes, and after complete training test the learnt policy on a test set of 1000 episodes to generate average reward.

Figure 8 shows the variation in average reward with α when trained for 10,000 episodes, while Figure 9 shows the variation when trained with 100,000 episodes.

From Figure 8, we observe that increasing α value for on-policy algorithms with epsilon-greedy policy, leads to fall in the average reward. This is because high epsilon value causes the agent to explore more, and due to high alpha value, this exploration (which may lead to bad state) proves highly costly, as it may change the learnt policy highly towards the bad state.

Figure 9 depicts that with increased number of episodes, increasing α also affects the other algorithms. This is because with high α values, the optimal policy has high variance, and keeps changing towards more recent episodes, forgetting the experience from the older episodes, thus in effect reducing the number of episodes out of which the learning is happening.

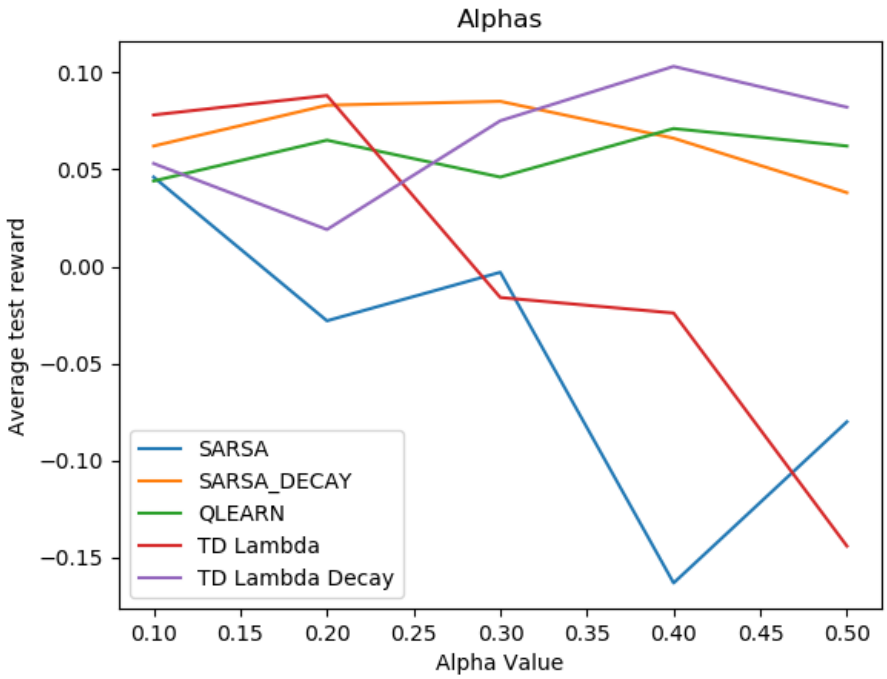


Figure 8: 10,000 Episodes

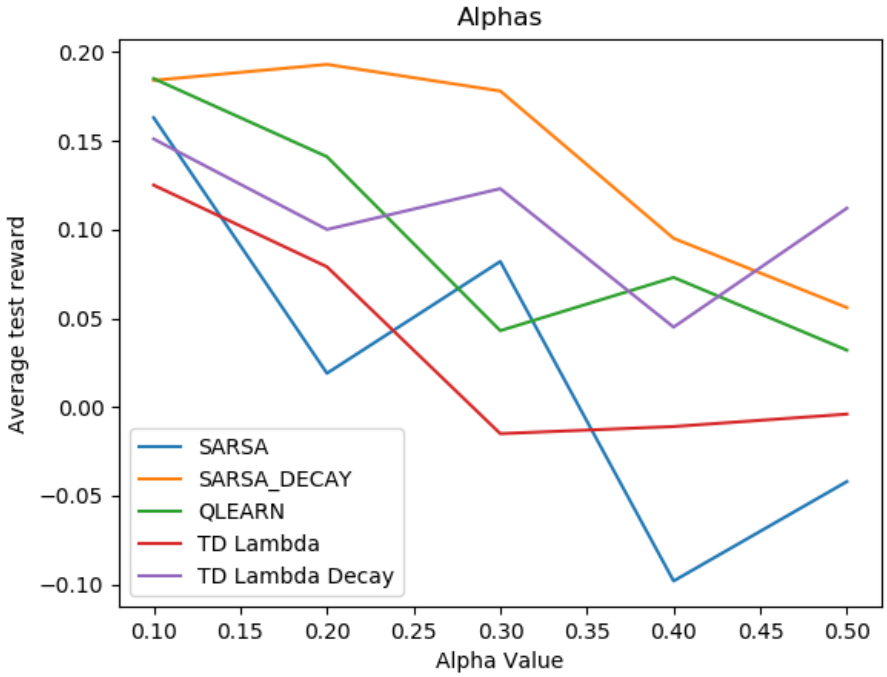


Figure 9: 100,000 Episodes

4.4 Visualising the State Value function

Below we plot the State Value function, Q-function and the optimal policy using the TD Lambda(0.5) with decaying epsilon algorithm, after training for 100,000 episodes as well as 1,000,000 episodes. Other algorithms produce similar results.

The green regions in the value function denotes high value, while the red ones denote negative values. For the policy plot, green cell denotes *Stick* in that state, while red cell denotes the action *Hit*.

The Value function $V(s)$, is very different from that observed in fixed policy used earlier.

We also observe that the policy still has not converged, clearly getting better on training for 1,000,000 episodes.

4.4.1 100,000 episode training

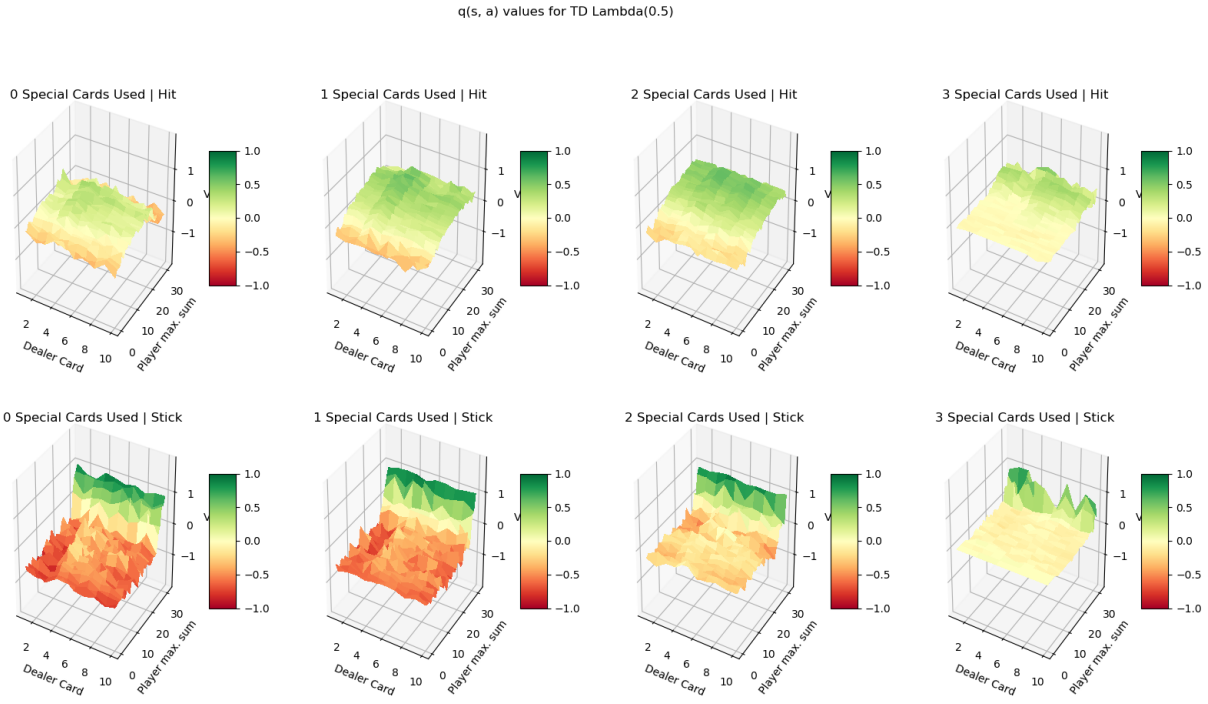


Figure 10: $q(s, a)$ after training for 100,000 episodes

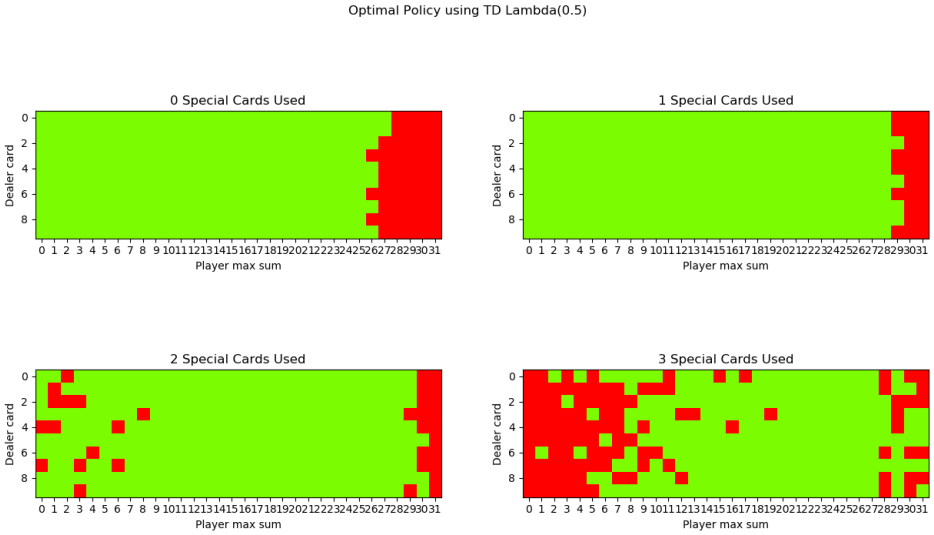


Figure 11: Policy $\pi(s)$ after training for 100,000 episodes

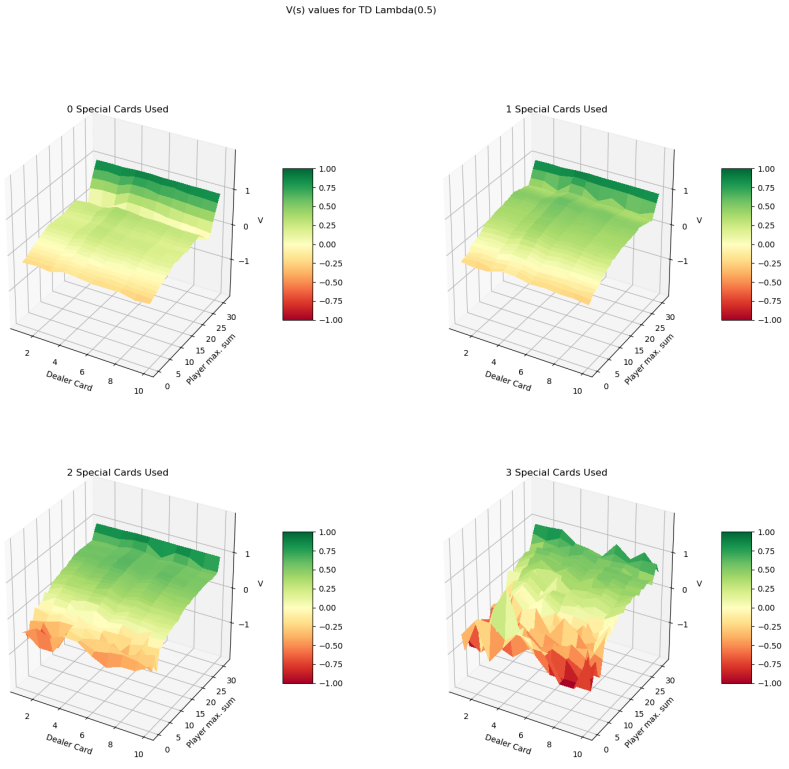


Figure 12: $V(s)$ after training for 100,000 episodes

4.4.2 1,000,000 episode training

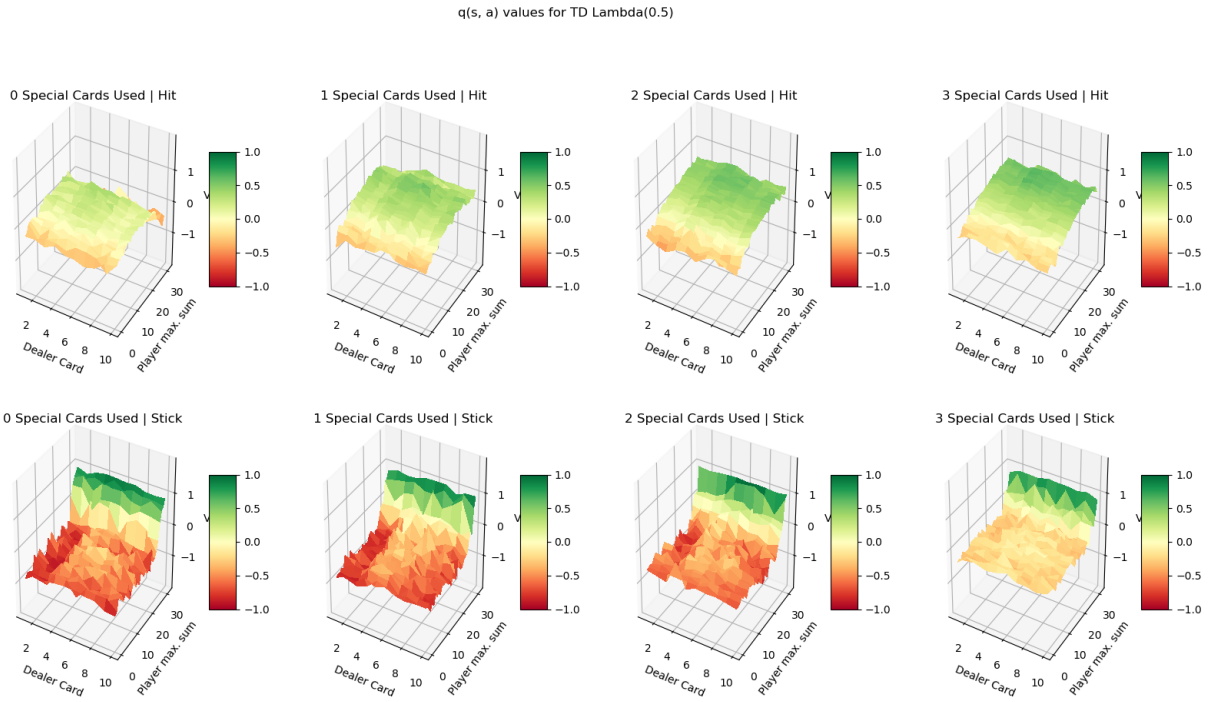


Figure 13: $q(s, a)$ after training for 1,000,000 episodes

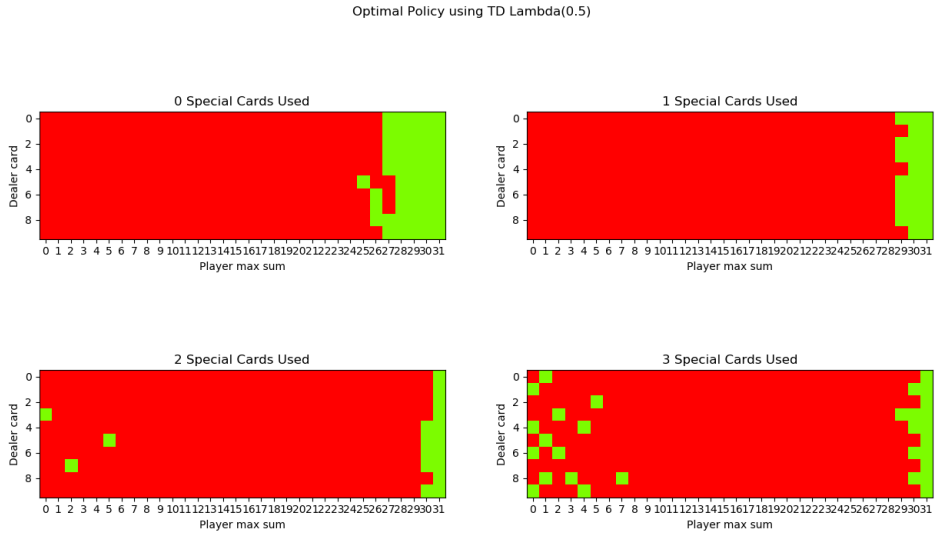


Figure 14: Policy $\pi(s)$ after training for 1,000,000 episodes

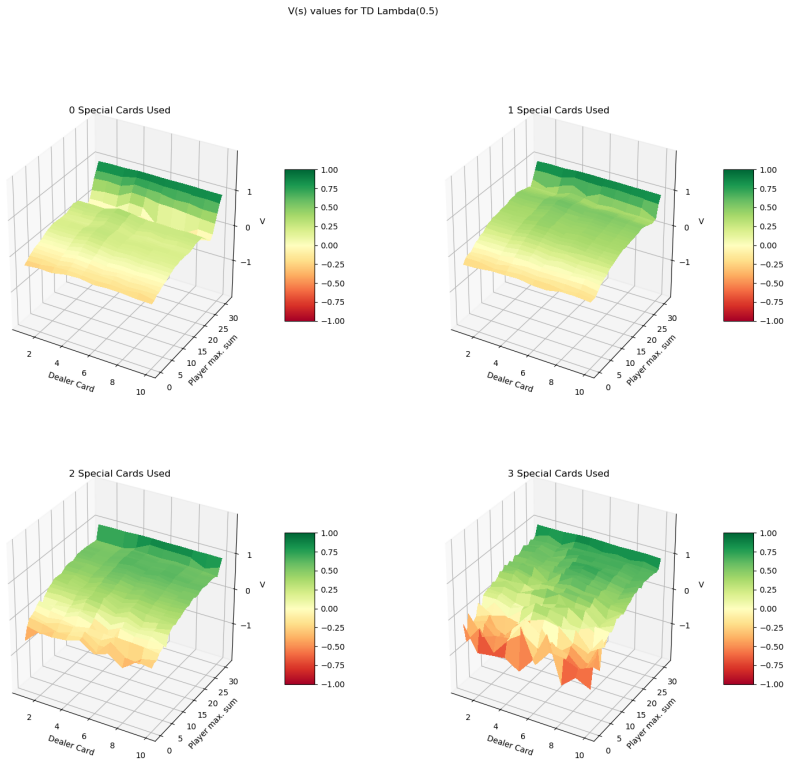


Figure 15: $V(s)$ after training for 1,000,000 episodes