TECHNICAL UNIVERSITY OF CRETE

Reinforcement Learning

and

Dynamic Optimization

Assignments - Project 1 - Card Counting Black Jack: Phase 2

Georgios Apollon Taouxis AM:2019030011

Antonis Katsoulis AM:2013030177

---

1. Environment construction

The environment was created by ourselves from scratch. We used random.choice and random.sample to draw uniformly from the deck for all playing entities. We did Task 1 with tabular q learning and we implemented Task 2 with both tabular q-learning and Deep Q Networks, in order to find which performs best.

2. Phase 1

We used Tabular q learning for both Tasks. The training parameters we used were $\alpha = 0.1$, $\gamma = 1$ since the algorithm doesn't have to make a lot of choices to reach the reward. For the exploration part during training we used a decaying $epsilon = 1.0 - episode/episodes$ and set the final $epsilon = 0.01$

3. Phase 1 performance

A) Win rate

The win rate we managed to achieve in Phase 1 Task 1 using a basic strategy blackjack agent was in tha range [-0.6, -0.5]

The policies are defined by the values of the Q-table. The Q-table is updated using bellman's equation every time the agent takes a decision based on the Q-table's values, as well as every time a reward is awarded from actions taken. At the beginning all cells are assigned the value of 0. Every time the agent has an action to take e-greedy is applied as follows: If epsilon is higher than random number in range [0,1] the agent chooses with equal probability between Stay or Draw.

B) Relevant Statistics

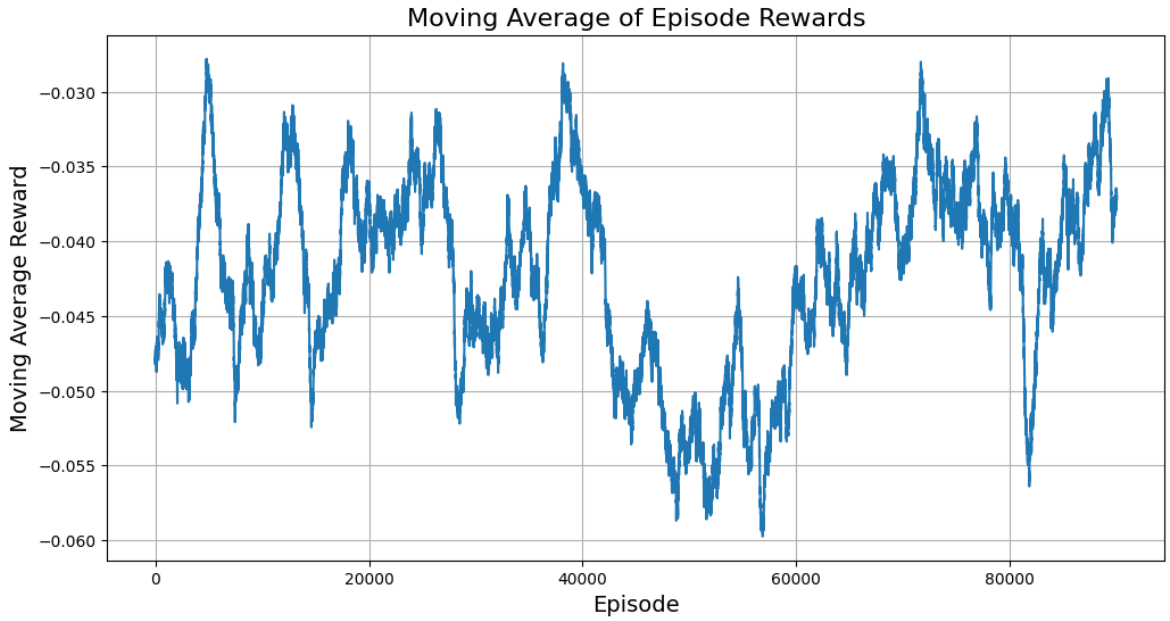
For all Tasks, we trained for 1.000.000 episodes and evaluated for 100.000 episodes.

4. Phase 2: Task 1

In the second phase we implemented double down, we did not implement split and changed reward from 1 to 1.5 for natural blackjacks. Double down is allowed only when the agent has two cards at hand. Moreover we added a separate bet_table with dimensions (1,10), one row and ten columns for 10 discrete betting choices with 10 being the max bet offered. The bet is decided prior to seeing the cards. We calculate average reward for 10.000 steps as the sum of rewards divided by the length of episodes per step. We calculate win rate in all Tasks as the number of episodes the agent received a positive reward, divided by the total number of episodes. Finally we fine tuned $alpha = 0.01$ as it gives better results compared to the previous.

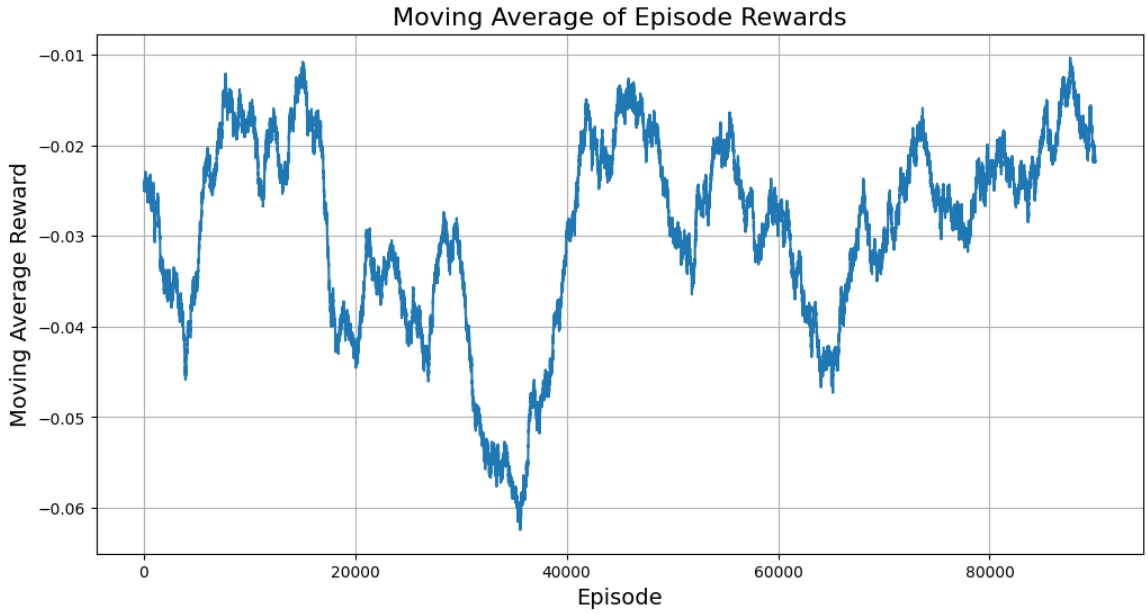
A) Win rate with basic strategy

The agent achieves an average reward of –0.042 per episode:

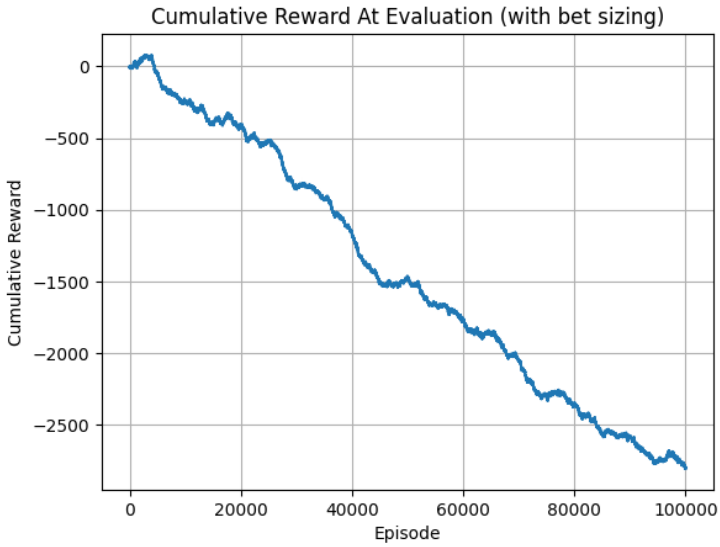


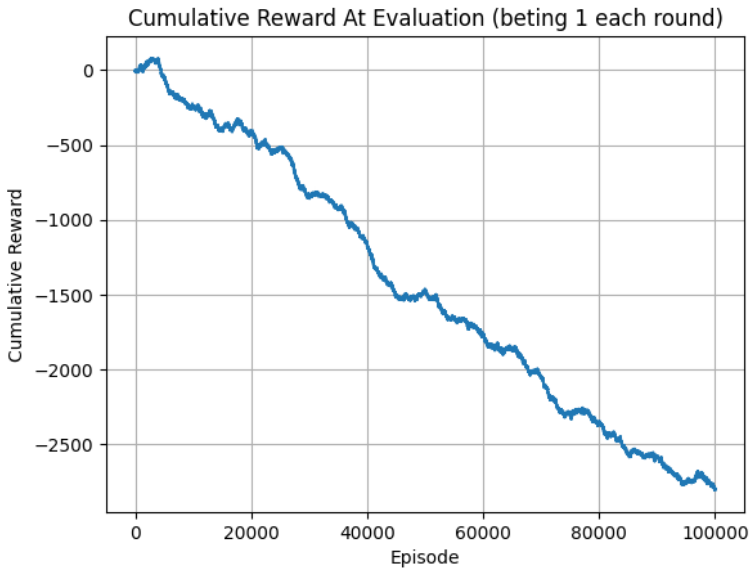
B) Win rate with counting

The agent achieves an average reward of –0.028 per episode:



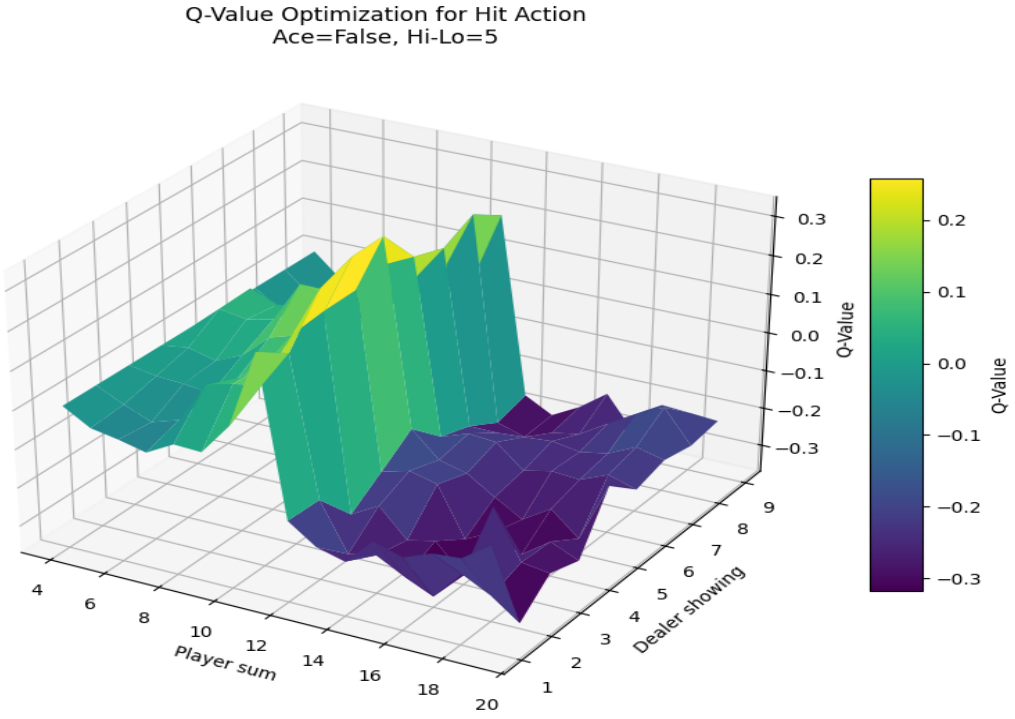
Followed by the cummulative reward with bet sizing, and the cumulative reward when the agent bets only 1 each round:





As these diagrams are the same, it is evident that as the agent achieves a losing average reward, it converges in betting the lowest possible amount in order to minimize loses, thus learning the optimal policy.

The following 3d metric features what the q_table looks like at the end of training, at Hi_Lo index equal to 5, which means that the running count is either equal to 3 or equal to 4, thus there are more high value cards left in the deck than low value cards.



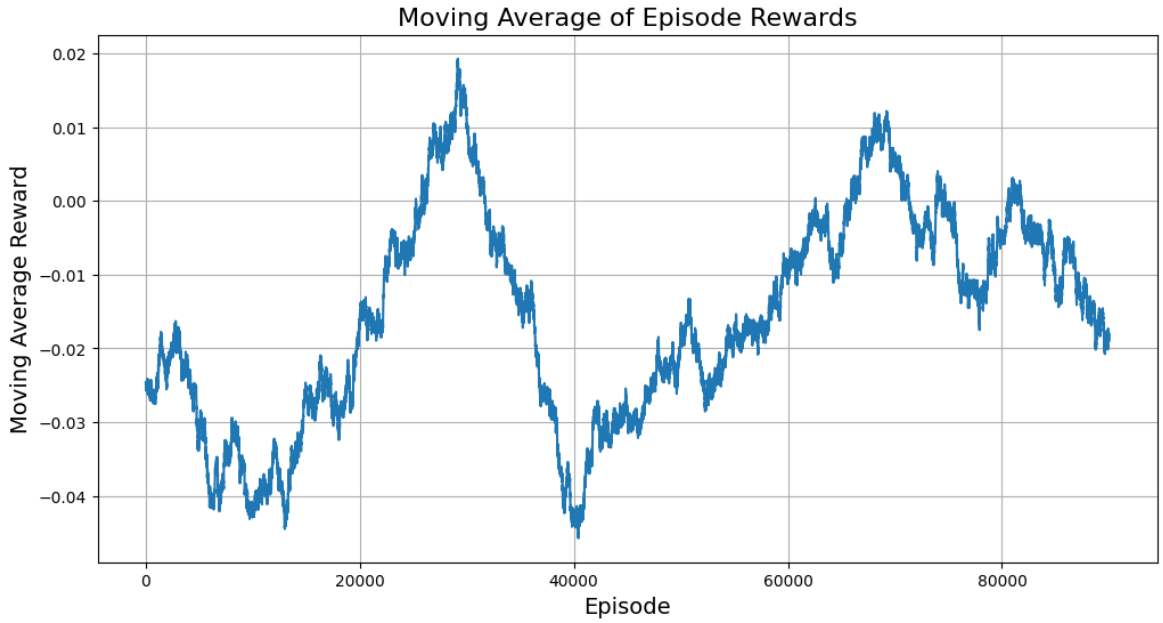
This diagram showcases ascending q values as player sum ascends with the cutoff player sum value at 11. This explains that the agent gets the most positive rewards when its cards value is 11 and it makes perfect sense, as with a cards value of 11 and a new card drawn with a value of ten, which is very possible since the running count is high, 21 is reached and a reward is received. Moreover, the q values drop suddenly after 11 which explains that with a sum of 12 and a 10 drawn 21 is exceeded and a negative rewards is received. All these help us understand that the counting is working correctly and that the agent also learns the optimal policy correctly.

5. Phase 2: Task 2 with Tabular q-learning

For task 2 we changed the way the agents decides the bet for the following round. We added the logic to base its decision on the current Hi_Lo, and on how many aces are left on the deck.

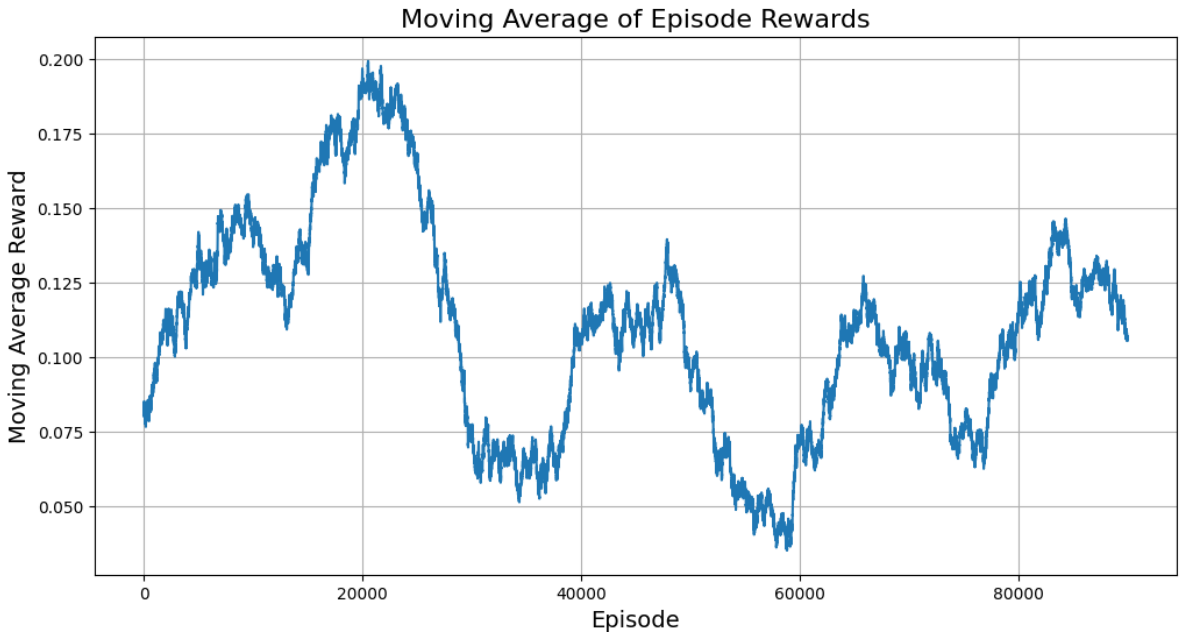
A) Performance with 1 deck

With one deck we were not able to achieve a positive average reward, however we improved a little bit from the previous one as we got –0.016

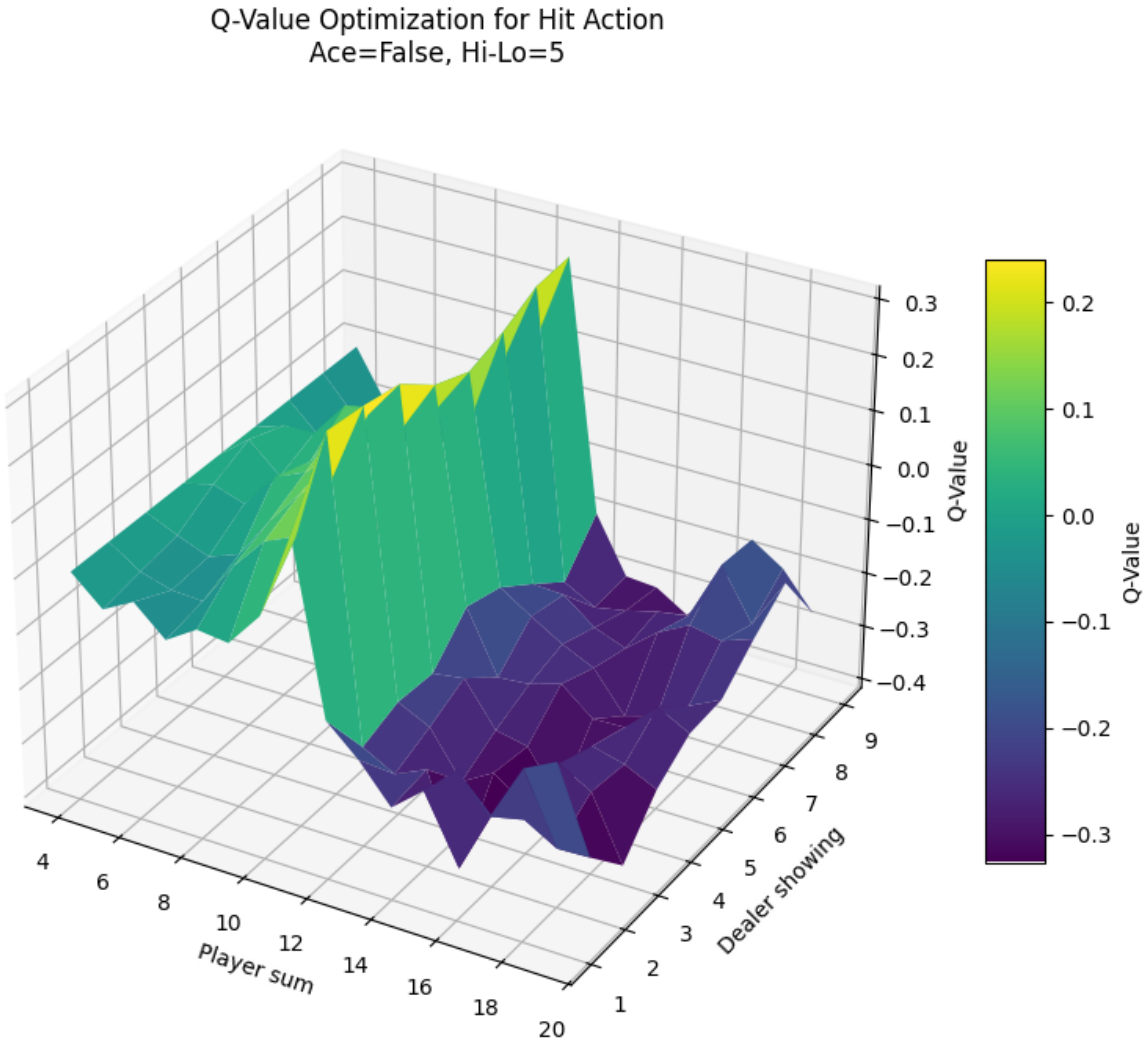


B) Performance with 4 decks

With 4 decks we managed to achieve a positive reward of +0.108

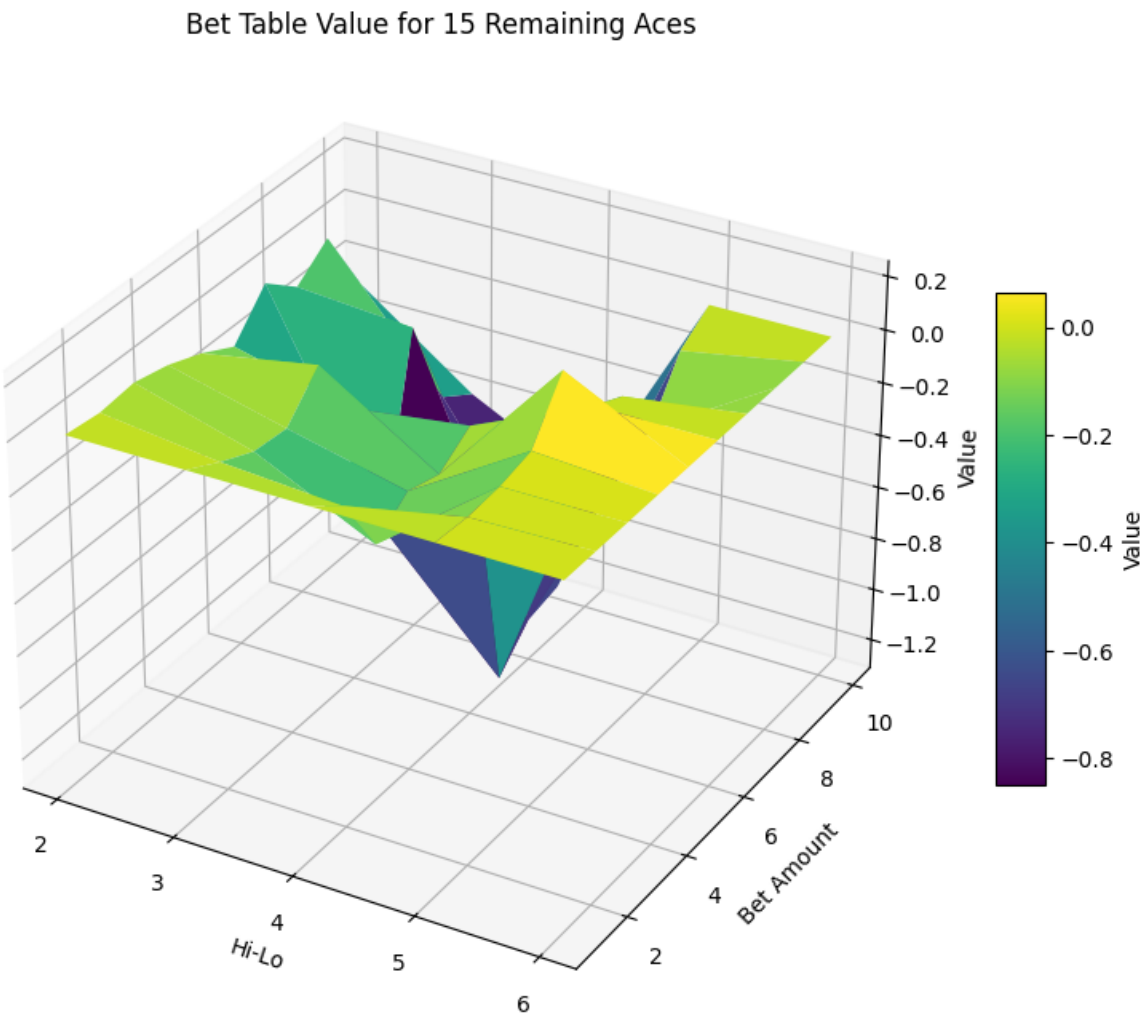


The following metric, as before explains that the agent is indeed learning.



C) Betting policy

Regarding the betting policy the next metric is an optic of the bet table and it explains that the agent learns to bet higher when Hi_Lo is higher as the Highest values are observed when Hi_Lo is is highest.



Phase 2: Task 2 with Deep Q Networks

Since we achieved a winning rate we decided to compare training and betting methods.

We tried trainning using a neural network. Procedure of coding the neural network was taken pytorch tutorial videos and troubleshooting was done using chatGPT.

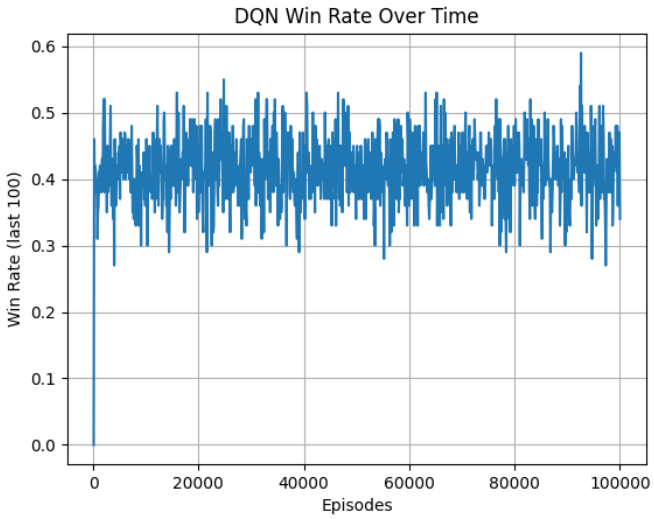
For specifics :

State dimension was 4 (hand, opponent hand, has ace, true count)

Action dimension was 4 (hit, stand, double down, split)

*no more than splitting once was allowed in order to avoid large loops.

Overall an average win to loss ratio of 85% over 100000 testing episodes was reached using this method.



*this graph measures wins/losses per 100 episodes

Now for betting, we tried 3 ways, one was using a neural network with a state dimension of 1 (true count) and an action dimension of 10 (different bet sizes),

number two was a simple q table using only a true count state and choosing the right bet and number three was a hard coded betting system depending on the true count.

The q table method was able to return a profit of roughly -7500 over 100000 episodes which roughly translates to a 7.5% loss of income ratio. If we were just betting 1 unit per game, we would have a loss ratio of 15% and now we managed to half that.

The idea of the hard coded systems stems from the fact that card counting is a strategy of its own and it would be much easier implementing the strategy rather than having to train an agent to try and learn it by actually veryfying it. We are already winning an 85% of the time so it becomes a matter of scalling our bet.

7. Comparison between Tabular q-learning and Deep Q Networks

One of the largest drawbacks of using neural networks was running time. The time it took to run each iteration of the algorithm reached 30 mins on some occasions. Training the betting agent with DQ learning also proved ineffective. This stems from the fact that neural networks struggle solving probabilistic problems. Neither option was able to provide an perfect playing and betting agent, however tabular q learning was able to make a profit.

Overall, tabular q-learning proved to be more efficient in tackling the matter at hand.