Training of Agent to Play Black Jack

Ivan Golovine
500813431

Aidan Uchimaru
500883656

Ogan Batu Aktolun
500861276

Mohammed Qasim Khan
50085658

CPS 824 Group 3

***Abstract*—We trained an agent to play blackjack against a dealer. The goal was to get as close to a 42% win rate which is considered to be the probability of a net win in blackjack. Using the OpenAI Gym blackjack environment, we train the agent using a greedy epsilon, first-visit Monte Carlo algorithm with varying epsilons, gammas and number of episodes the agent is trained for. We use both the win rate and the state value graph to determine the best variation of the algorithm for training the agent.**

I. Introduction

For our final project, we looked to deepen our understanding of on-policy model-free learning. To do so, we trained an agent to play blackjack using Monte Carlo methods. Most of us enjoy the card game and saw it as a project of appropriate difficulty to learn from. Our main goal was to have our agent attain as high of a win rate as possible using a greedy epsilon policy. In this report, we will be discussing how we set up the environment and the code for the Monte Carlo methods, performance metrics that we evaluated, and discuss our results.

II. Problem Statement and Environment

A. Problem Statement

Our agent plays a game of blackjack against the dealer. There are no other players involved in the game. We opted to not include any sort of betting into the game since betting is usually done before the cards are dealt meaning rewards from betting would not be indicative of the agent learning anything.

B. Environment

We used the blackjack-v0 environment from OpenAI Gym as the environment for our project. The game in this environment plays out by dealing 2 cards to both the player and the dealer from a deck. The cards are randomly chosen from an infinite deck. The player shows both cards while the dealer only shows 1 card similar to how the game is played in a casino. There are 2 actions that can be taken:

- Hit (1): additional card is dealt to the player
- Stick (0): player keeps current hand as is

Once the player sticks, the dealer will repeatedly hit until their hand totals to 17 or more. Rewards are given out based on the following scenarios:

- Reward of -1 if the agent hits and busts or if the agent ends with a lower score than the dealer
- Reward of 0 if the agent hits and doesn't bust or if the agent and dealer end with the same score
- Reward of 1 if the agent ends with a higher score than the dealer
- Reward of 1.5 if the agent has a natural blackjack (face card/10 and an ace as initial two cards)

C. Initial Code

Dictionaries were used to keep track of the Q value, number of times each state was visited and the policy for each state-action pair. States were determined by the following values:

1) Agent hand total: Integer representing the total value of all cards in agent's hand

2) Dealer card value: Integer representing the value of the dealer's face-up card

3) Usable ace: Boolean representin whether or not the agent has a usable ace in hand. A usable ace means an ace that can be assigned a value of 11 without the hand busting

These 3 values make-up a tuple that is used as the key in the dictionaries to store and retrieve values for each action. Values for each action are stored in a list where the index in the list corresponds to the number representing each action.

III. Methods and Models

The algorithm that we settled on implementing to teach our agent to play blackjack was an epsilon greedy policy improvement algorithm using Monte Carlo methods. We did not want to use a regular greedy policy improvement algorithm as we wanted to promote a controlled level of exploration by the agent within the environment. The policy always starts with each action in each state being equally likely. We looked at both first visit and every visit Monte Carlo policy evaluation during this experiment. The control variables we considered are outlines below:

1) Epsilon: We looked to determine an optimal balance between exploration and greedy decision making by the agent. We started with a decay rate of 1/iteration and fine tuned it from there. In some cases, we did not decay epsilon and kept it at a constant low value

2) Gamma: We tested with a couple of different gammas between the range of 0.3 and 1 to determine the effect of reduced delayed rewards on the agent's learning

3) Number of games to train agent: We trained the agent for up to a million games and at different intervals, had the agent play 1000 games and looked at the win rate at these different intervals.

We looked mainly at both the win-rate and the policy when determining the best agent at different steps. Win-rate while not completely indicative of how good the agent has become at playing black jack, can help in determining which agent is the best. The policy is a good reference for determining what the best agent is. There are specific actions that should be taken in some steps such as sticking when you have 20 and the dealer shows a 4. The optimal choice by the agent in given states was compared to the action that would typically be made by a human to determine how effective the training was.

The functions used for the code in the monte carlo methods were based off those used in assignment 2.

IV. Results and Discussions

We had the agent play 1000 games after, training it for 1000, 5000 and 10000 games for most of the training. While we looked at 50000 and 100000 games of training for some cases, we noticed no real improvement past 10000 iterations of training for most as the policy will have already converged by then. Results for the 50000 and 100000 iterations of training are ommited from some tests for these reasons.

We were hoping to attain a consistent 42% win rate by optimizing our code. 42% is the probability of a net win in blackjack [1].

A. *First-Visit Monte Carlo vs Every Visit Monte Carlo*

In upper half of figure I, the average data of 3 run throughs of the code using first-visit Monte Carlo are shown. In the lower half of figure I, the average data from 3 run throughs of the code using every-visit Monte Carlo are shown. For these, epsilon was equal to 1/iterations and gamma was 1.

		1000 games	**5000 games**	**10000 games**
First Visit MC	**Win Rate**	0.374	0.385	0.409
	Loss Rate	0.537	0.544	0.524
	Draw Rate	0.089	0.071	0.067
Every Visit MC	**Win Rate**	0.371	0.372	0.37
	Loss Rate	0.561	0.569	0.555
	Draw Rate	0.068	0.059	0.075

Fig 1: Average agent stats when trained with first-visit and every-visit Monte Carlo Methods

At first glance, we can see that the first-visit Monte Carlo generates better performance. We found this interesting as we did not expect this based on the theory. These two methods only differ in that the Q-value of each state is only updated the first time a state is visited in an episode for first-visit Monte Carlo. Since you cannot visit the same state more than once in an episode in black jack, the method should not matter. Given how much randomness there is to black jack, the difference in win rates could not mean too much as to the performance of the algorithm. To confirm this we did some more run-throughs of both algorithms and confirmed that the win rate, regardless of the algorithm used, seems to always fall between 36% and 41%.

Looking at the state values, the first-visit and every-visit Monte Carlo gave similar graphs. In figure 2, the state value for each state is shown. These graphs looked as we expected where the highest state values occur when the player has a high hand value.

Since we did notice slight better results from the first-visit Monte Carlo algorithm, we continued with this method for the rest of the experiments.

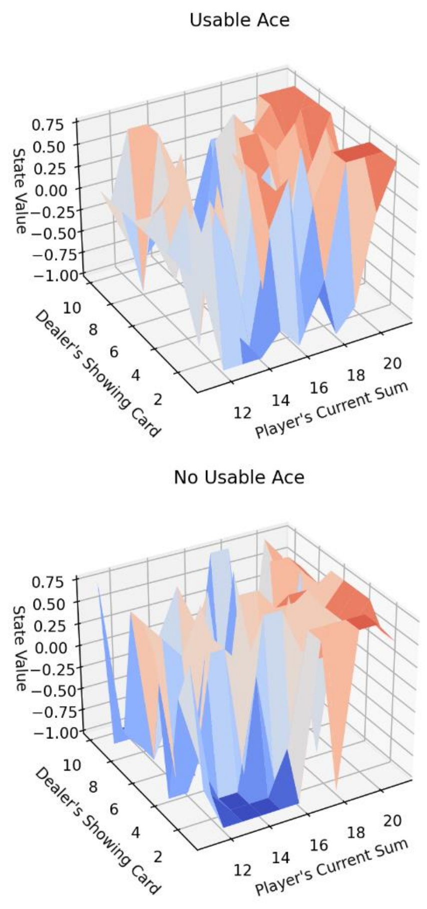


Fig 2: State Values for First-Visit Monte Carlo

One last thing that can be noticed is the gradual increase in win rate as the agent is trained more. As stated before, we looked at 50000 and 100000 games and saw no real improvement.

B. *Epsilon Effect on Training*

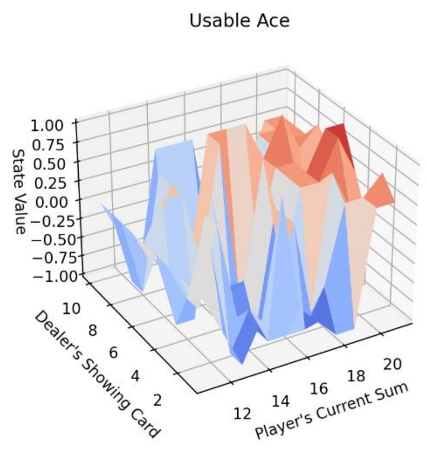
For this part, we compared 3 epsilon values:
Epsilon = 1 / iterations: This was the base case the results are previously depicted in Figures 1 and 2.

Epsilon = 1 - 1e^{-6} * iteration: We ran this one over the course of 100000 games to allow epsilon to finish at 0. The goal of this was to see if higher epsilons at later training iterations would lead to a different action being the best action in some states. Having a higher chance to explore could lead to some changes in the optimal policy.

All that we saw was that the learning curve was less steep. As seen in Figure 3, the win rate started lower and steadily increased over time as the agent was trained. By the end of the training, the agent was performing to a similar level as the base case. The only difference being the rate at which the agent learns.

	1000 games	5000 games	10000 games	50000 games	100000 games
Win Rate	0.27	0.27	0.279	0.331	0.389
Loss Rate	0.668	0.668	0.673	0.603	0.532
Draw Rate	0.04	0.04	0.048	0.066	0.079

*Fig 3: Average agent stats trained with epsilon = 1 - 1e^{-6} * iteration*



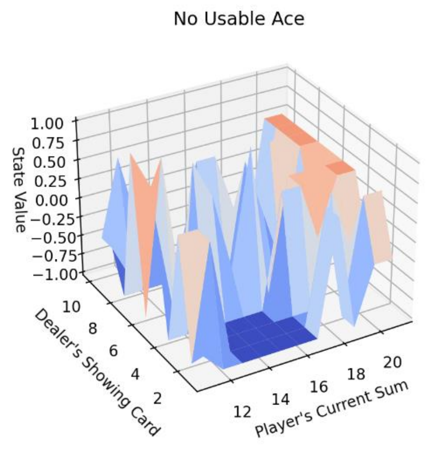


Fig 4: State values with slow decaying epsilon

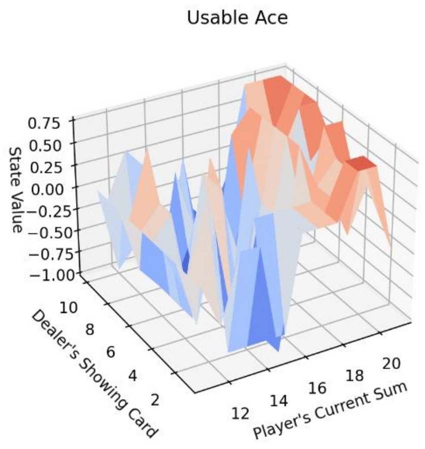
Looking at the state values in figure 4, we notice that there once again isn't any significant difference from those of the base case. Based on these results, we believe it to be more effective to train the user as we did with the base case as it takes less time to train the agent and with similar results.

Epsilon = 0.05: Using a constant epsilon generated similar results when looking at win rate. There was steady learning before 10000 iterations and then the win rates level out after that. This makes sense as the greedy action is almost always taken so the policy should converge in a similar manner to the other tests.

	1000 games	5000 games	10000 games	50000 games	100000 games
Win Rate	0.347	0.344	0.38	0.357	0.38
Loss Rate	0.589	0.585	0.551	0.567	0.567
Draw Rate	0.064	0.071	0.069	0.071	0.076

Fig 4: Average stats trained with constant 0.05 epsilon

Looking at the state values, we notice that the graph is the most consistent with what we expect where the state values are high as the player's hand value is higher and flat on leading up to that section of the graph. These peaked the most of any graph at the right end and had the lowest overall values as you move left along the graph.



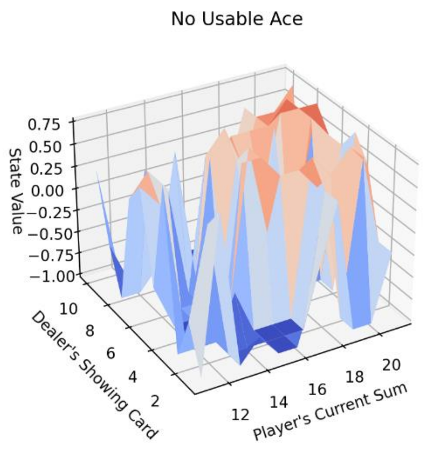


Fig 5: State values with constant epsilon after 100000 iterations

C. Effect of Gamma on Training

Changing the gamma yielded similar results to the initial tests we did. We looked at a gamma value of 0.8 and 0.9. Looking at figure 6 and 7, we see that there was no noticeable change in overall win rate. This is most likely due to the fact that a non-zero reward can only be received at the end of an episode when the player either busts or beats the dealer. Because of this, Q values in states where the agent has a lower total hand value will be lower overall but would still have the same optimal action.

	1000 games	5000 games	10000 games
Win Rate	0.366	0.404	0.406
Loss Rate	0.573	0.572	0.527
Draw Rate	0.061	0.059	0.067

Fig 6: Average stats trained with 0.9 gamma

	1000 games	5000 games	10000 games
Win Rate	0.381	0.379	0.38
Loss Rate	0.547	0.558	0.547
Draw Rate	0.072	0.063	0.073

Fig 7: Average stats trained with 0.8 gamma

What was noticed through some tests was that the policy converged to the optimal policy to hit on lower player hand values sooner. We assume this to be due to the agent staying would receive the entire -1 reward if it stayed and lost however if it hits and loses later, the delayed reward of -1 is smaller. This would immediately teach the agent to always hit as it has a better chance of winning and loses less reward for losing at a later step in the episode because of the reduction in delayed rewards.

D. Next Steps

If we were to continue this project, we would look to teach the agent to play with other players at the table. Having other players at the table influences decision making since there are more cards worth 10 than any other card in a deck. If the dealer needs to hit, then you want them to have the highest chance to bust which in many cases would be the result of a card worth 10 being drawn. In this case, we would no longer be drawing from a pool with replacement after each card is drawn. Instead, we would likely create an array of 4 decks worth of cards and use those cards to play through each round. Cards would be reshuffled into the deck following each episode. With this implementation, there would be more decision making involved in determining when to hit if the agent considers the option that it may be worth sticking and hoping the dealer busts as you might see in a casino at a table with various players.

Another future implementation that is possible is to add a third action which would be to split. This would involve the agent tracking additional hands when they split and playing accordingly. How the agent plays should not be affected by the addition of this action once they have their two split hands.

One last thing we wanted to do was do some more testing with higher games played to see how flattened the state value graph changed more as the ones we produced with the results from 10000 or 100000 games played by looking more at in the 500000 or even 1000000 games played range. This was becoming too time consuming to test on our machines whenever we exceeded 100000 games played.

V. Conclusion

Our findings showed that our first implementation of the algorithm was pretty efficient at teaching the agent. To train it quicker, a gamma slightly smaller than 1 can be used with an epsilon that decreases relatively quickly or by having a small constant epsilon. Taking into account the graphs, a constant epsilon seems to be the best way to train the agent however the agent loses out on exploration in earlier iterations doing this.

VI. Implementtaion and Code

We used the gym, numpy, matplotlib.pyplot and mpl_toolkits libraries. Code was designed based on the code provided in assignment 2 and learned in class as well as some code from philtabor on github [2]. Graphing code was pulled from jorditorresBCN on github [3].

VII. References

[1] M. Shackleford, “Variance in Blackjack,” 🧙 Wizard Of Odds &gt; Guide to Gambling Games &amp; Online Casinos, 21-Jan-2019. [Online]. Available: https://wizardofodds.com/games/blackjack/variance/. [Accessed: 19-Apr-2021].

[2] Philtabor, “philtabor/Youtube-Code-Repository,” *GitHub*, 23-Jun-2020. [Online]. Available: https://github.com/philtabor/Youtube-Code-Repository/blob/master/ReinforcementLearning/Fundamentals/blackJack-no-es.py. [Accessed: 19-Apr-2021].

[3] jorditorresBCN, “jorditorresBCN/Deep-Reinforcement-Learning-Explained,” GitHub, 21-Jul-2020. [Online]. Available: https://github.com/jorditorresBCN/Deep-Reinforcement-Learning-Explained/blob/master/DRL_13_Monte_Carlo.ipynb. [Accessed: 19-Apr-2021].