Monte Carlo Control for Modified Blackjack

Jacob Epifano, Sean McGuire

***Abstract*—This paper explorers the creation of a Monte Carlo (MC) agent used to play the game Blackjack. This agent was trained on a Blackjack simulator in python. This Blackjack simulator deals the cards and requests an action from the agent (hit or stay). The agent receives the sum of the players cards and the showing dealers card, this is the agent state. The MC agent returns an action selected from the policy. Following optimal blackjack strategy a win-rate of 42.42% is expected. Our agent was able to play blackjack with a win rate of 41.27%. This shows that Monte Carlo Control was able to generate a policy capable of formidable play in the game of Blackjack. Additionally, we believe our agent was able to learn higher level concepts such as card counting (estimating the changing probabilities of the next card given the showing cards) due to the fixed size of our deck.**

I. Introduction and Background

In this project a reinforcement learning agent was created to play the game Blackjack. Blackjack is a casino game where the players compete against the dealer to see who can get closer to a score of 21 without going over. Each card 2-10 is given their face value while cards Jack, Queen, King are all given a value of 10. The Ace card can be either 1 or 11. Each round 2 cards are dealt face up to each player. 2 cards are also dealt to the dealer, however only one of them is face up. In order the players can either choose to get another card (hit) or pass their turn (stay). In this implementation there is only a single player and the dealer at the table. The dealer must hit until their sum is 17 or higher. If the player has a higher value after the dealer is done drawing then the player wins. If the dealer or the player goes over 21, they lose. If the dealer and the player both settle on the same score the round is a "push" or draw and the bets remain on the table. In this implementation we will draw from a deck of 52 cards and reshuffle after each game. This is done to avoid the changing probabilities which occur when using a finite number of decks over many games. The concern is that the agent will learn to count cards, or learn to estimate the probability of different cards coming out over many games. [2]

This problem will be formulated as an MDP with the states covering the value of the players card and the visible dealer card. The rewards will be -1 for losing (bust or a value lower than the dealer), 0 for a draw, and 1 for winning. The actions will be to hit (draw another card) or stay (stop drawing cards). In this problem, returns will not be discounted as the rewards are only given at the end of an episode and the episode length is relatively short.

We will use the Monte Carlo (MC) Control algorithm to attempt to learn the optimal policy. Since each episode (game) only has 3 decisions on average, there is no point in using any truncated learning algorithm (like n-step TD). For the sub-type of algorithm we will use on-policy MC with an ϵ-greedy policy. ϵ-greedy algorithms have a non-zero chance to select a random action as opposed to the option that maximizes reward during training. We do this so that our algorithm explores non-optimal positions in hope that the agent can find the best way to play from every state it reaches. Some blackjack strategies suggest relying on a bad hand if the dealer is showing a low card, than to hit to obtain a higher score. This is because the dealer has a high chance of busting as it is assumed the next card will be a value of 10 (most common card) and the dealers sum will be under 17 so they will have to hit again and have a high chance of busting. The agent cannot reason and figure this out, it has to learn this through many iterations of game play.

II. Approach

A. Markov Decision Process Formulation

The first step in reinforcement learning is fully defining the problem in terms of a Markov Decision Process (MDP). Defining the problem as an MDP includes defining state space, state transition probabilities, action space, and rewards. The Blackjack state space in this approach was defined as a combination of the sum of the users cards and the value of the dealers showing card. This state formulation contains almost all of the information needed by the agent to make a decision. The one piece of information left out is if an Ace is in the players hand. In this case the Ace can be valued as a 1 or an 11. To simplify the problem the highest allowable value of an ace was used, for example if the player was dealt an Ace and an 8 the current state would be 19. The action space at a given state is either hitting (getting another card from the dealer) or sticking (ending the game with the value at hand). If the player decided to hit then the game will request another action. The transitions between states is the probability of drawing a given card to get to the given state. In this case these probabilities are unknown to the agent and it has to learn these probabilities by playing many games. The game of Blackjack can terminate in one of three ways, a tie, win, or loss. The rewards are defined as +1 for winning, -1 for losing, and 0 for ties. This reward structure emulates a player betting at a casino. It is also important to note that the agent is operating in a partially observable environment as the dealers second card is unknown. This means that it may require many iterations over a set of states to properly estimate the value of the given state.

B. Possible Solutions

There are many different reinforcement learning methods which can be applied to solve problems such as Blackjack. Some of these methods include temporal difference (TD) learning, dynamic programming (DP), and functional approximation. Temporal difference learning was found to not provide a meaningful benefit over Monte Carlo learning as the episode lengths in blackjack are rather short and would not benefit much from bootstrapping. Dynamic programming is a possible solution to this problem but it is computationally expensive as it would require a very high number of iterations to converge. Dynamic Programming is also essentially a brute force approach to the problem as it manually explores all of the states and does not attempt to learn from its own play. Functional approximation methods are not needed as the state space and action space are manageable sizes and there is not much to be approximated.

C. Monte Carlo Method

The algorithm we settled on was an on-policy Monte Carlo method. The policy was initialized as an ϵ-soft policy. We used an ϵ-greedy approach to optimize our policy with and ϵ of 0.1. We ran the agent to train over one million iterations, while saving the policy every one hundred thousand iterations. The algorithm below describes our training process.

Algorithm 1 Decision Loop

```
for $A \in episode$ do
    if $\epsilon < Unif(0,1)$ then
        $A \leftarrow \pi(s)$
    else
        $A \leftarrow RandomAction$
    end if
end for
return $G_t \leftarrow (-1, 0, 1)$
```

For every decision in an episode we compare a random value generated by python with ϵ. If the value is greater than epsilon we look to the policy for the decision. If the value is less than ϵ we take a random action. This action loop will return -1 return if the actions resulted in a loss, 0 if the actions resulted in a tie and +1 if the actions resulted in a win.

Algorithm 2 Monte Carlo On-Policy [2]

```
$\pi(s) \leftarrow Unif(0,1), \forall \quad s \in S$
$Q(s,a) \leftarrow 0 \quad \forall \quad s \in S, a \in A$
$C(s,a) \leftarrow 1 \quad \forall \quad s \in S, a \in A$
for $1,000,000$ do
    for all $(S, A)$ pairs do
        $G \leftarrow$ Decision Loop$(\pi(s))$
        $Q(s,a) \leftarrow (Q(s,a) + (G - Q(s,a)/C(s,a)))$ or
        $V(s) \leftarrow (V(s) + (G - V(s))/C(s)))$
        $C(s,a) += 1$
        $\pi(s) \leftarrow argmax_a Q(s,a)$
    end for
end for
return $\pi_*(s)$
```

For each episode we get our actions and the reward. We use this reward to update the value function for all actions and states in the episode. We are using every-visit Monte Carlo and taking incremental averages of our rewards at each iteration. The counter is initialized at 1 for each state and then incremented after the update. Then, for each action taken we update the policy to take the action based on the value function. If the value function was positive it indicated hitting, if the value function was negative it indicated staying. This is repeated until the value function converges and then we have our optimal policy.

When evaluating the effectiveness of the learned policy the stochasticity was removed and the actions taken were dictated by the policy. This allows for the true evaluation of the policy and removes the exploration aspect of the agent.

III. Experiments

Code was written in python to simulate the game of Blackjack and create our agent. The Blackjack game simulator code was sourced from Github [3]. This code has the foundations of a Blackjack simulator, however, some win conditions were not accounted for, rewards were not defined and there was not way to have the agent manipulate the game. To fix this, the base code was modified to allow our agent to interact with the environment. The agent was also written in python. The agent is passed the sum of the two cards the player is dealt and the one card showing for the dealer. The agent would then return an action to the blackjack simulator so the game can continue. This process repeats until the game terminates and the reward is passed back to the agent and the policy, counts, and value function are updated.

Initially, we ran our model for 100,000 episodes but discovered that our count matrix was low for certain states. Especially in states with very low probability of occurring (such as dealer: 2, and player sum: 4). This is also an artifact of playing each game with only one deck. This may mean that our action-value function at the time, had not converged to the true value. We increased our ϵ and increased the number of episodes. This gave us much better coverage over all state-action pairs and increased our confidence in the resultant action-value matrix. It was also found that using a action value function proved very expensive to learn as we were never able to achieve convergence. To fix this problem a state value function was utilized. This state value function was increased when hitting resulted in a win, and decreased when hitting resulted in a loss. The state value function was decreased when staying resulted in a win and the value was increased when staying resulted in a loss. This effectively reduced the state space by a factor of two as instead of a matrix for each action, there is only a single matrix. The optimal policy was then selected by checking if the value function for that state was above or below zero.

The Blackjack agent was trained for 1,000,000 episodes and it was found that the value function and policy have converged. At this point the agent was evaluated over 100,000 episodes.

IV. Results and Discussion

The win-rate of the agent with the action-value formulation was found to be 41.7% and 41.18% with the state value formulation. Perfect strategy blackjack has an average return of 42.42%. [1] These rates change when ignoring ties as the tie should not count against the win rate. Our win-rate when ignoring ties was found to be 44.03% with action-value and 43.89% with state-value, while perfect strategy blackjack results in a win-rate of 46.36%. We attribute the difference in score to the design of our game. The state-value formulation's policy and value function are much closer to the optimal from [2], but may not have performed as well due to the way our game is run.

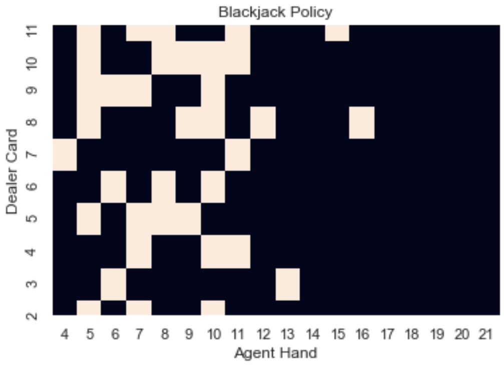

Figure 1. Sub-Optimal Policy

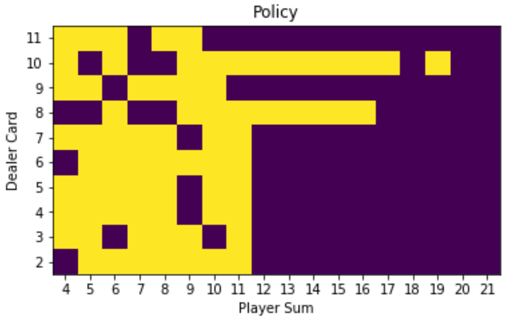

Figure 2. Optimal Policy

Figure 1 shows the sub-optimal policy of our agent using the action-value formulation. The black squares represent a "stay" action and the white squares represent a "hit" action. Note the patchiness of the policy. In [2] the policy is much more smooth across the hit regions. We believe this is due to how we set up the games. In each game we are using only one deck and then every game the deck is reshuffled. When a high card is drawn the probability of another high card is lower in this case than in reality with a multi-deck or a theoretical infinite deck game. Figure 2 shows the optimal policy when we switched to a state value formulation. This policy (yellow = hit, blue = stay) is much closer to the optimal policy in [2].

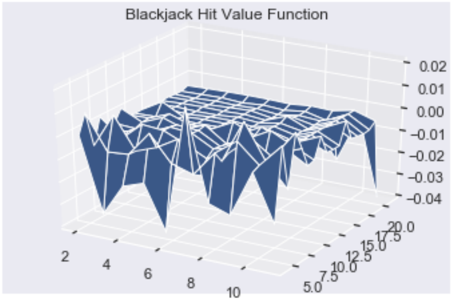

Figure 3. Value Surface for hit action

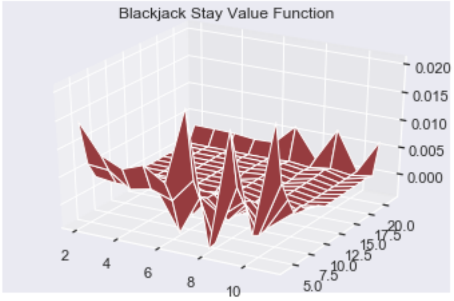

Figure 4. Value Surface for stay action

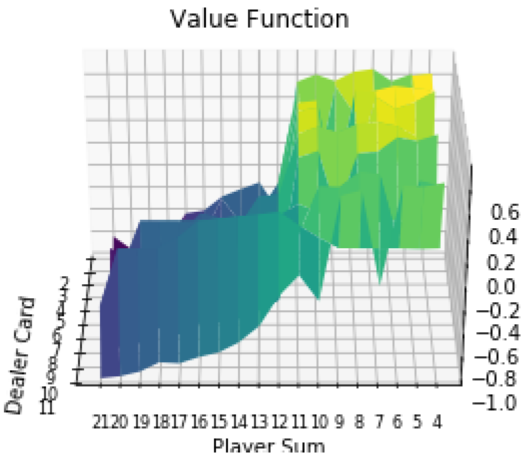

Figure 5. Optimal Value Function

Figures 3 and 4 show the action-value functions for each action plotted as surfaces. The first axis on the left is the dealer's face up card, and the second axis is the sum of the player's cards. The third axis represents the value of taking each action in that state. Note the jagged form of the value functions. Figure 5 shows the state value function. This surface is much more smooth and is comparable to [2].

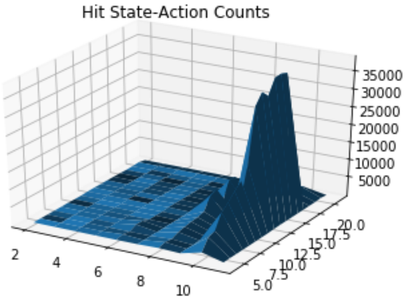

Figure 6. Count Surface for hit action

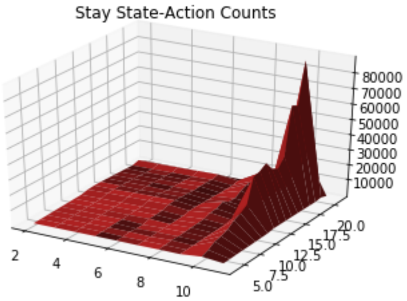

Figure 7. Count Surface for stay action

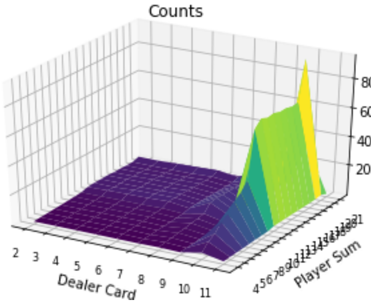

Figure 8. Count Surface

Figures 6 and 7 show the counts when using the action-value formulation. Note how much higher the stay counts are than the hit counts. This is due to the fact the algorithm must choose to "stay" at least once per game. When using the action-value formulation this may have given a large bias to the "stay" action since it has been explored many more times than the "hit" action.

V. Conclusion

It was found that the Monte Carlo agent was able to learn the game of Blackjack to a high level of play. The agent came within 2% of optimal play when ignoring ties. This 2% difference can be attributed to the state formulation. In formulating the state space aces were considered to be the highest acceptable value. This would lead the agent to learn a conservative, and sub optimal approach to blackjack as often it would make sense to hit in a situation in which the player has an Ace and a 3, but the agent would see a 14 and be fearful of busting when in reality if the player hit they cannot bust. This paper has demonstrated that our Monte Carlo method was able to come close to approximating optimal play in the game of Blackjack. We would expect our optimal policy to perform even closer to optimal play with a more accurate state representation.

References

[1] "Blackjack - Probability," The Wizard of Odds. [Online]. Available: https://wizardofodds.com/ask-the-wizard/blackjack/probability/.

[2] R. S. Sutton and A. G. Barto, Reinforcement learning: an introduction. Cambridge: The MIT Press, 2018.

[3] "Python Blackjack," Gist. [Online]. Available: https://gist.github.com/mjhea0/5680216.