Reinforcement Learning

Project – Blackjack

Vincent Gariepy

December 7th 2021

Introduction

The problem we have decided to tackle is the game of Blackjack, a classic gambling game where the odds of you winning should be near 49%, the best odds through all games of a casino. Seeing as how this problem was already lightly tackled in assignment 2, we decided to use this as our base to then expand and delve deeper into the intricacies of the game. We started off by using an every-visit Monte Carlo policy improvement method to try and find an optimal policy in every state. We then used the REINFORCE method to find a percentage-based policy and compare its performance with also using a baseline. Finally, we tested out whether a binary policy would be better than a percentage-based policy.

Blackjack

In the assignment, all that was asked was to simulate many games of Blackjack and estimate its state value function. However, these games were all independent and identically distributed, as every card drawn had a 1 in 52 chance of being picked, independent of what was previously picked. This is not how it is in real life though, there is a certain number of decks in play and once a card is drawn it can not be drawn again. This means that if you count cards, you should not play the same if all the big cards have already been drawn, compared to if none of them have been drawn yet. We decided that it would be interesting to tackle this card counting example, as it should change the way the game is being played. So, this project is split up into two different sections: one where we consider a finite number of decks with card counting and one with independent and identically distributed cards drawn. The Monte Carlo policy improvement method was used on the former while the REINFORCE and policy test was used on the latter.

Finite number of decks

Let us first consider the game of Blackjack with finite decks. In casino's they use between 4 and 8 decks, in our simulation we used 6 decks. The dealer (usually with the help of a machine) will shuffle all the decks into a pile which is where the cards will be drawn from, until there are about 20 cards left in the pile. Then the dealer will reshuffle all the cards and start over. The game is

played the same way as was described in the assignment; the player goes first after receiving their first 2 cards, they can see only the second card dealt to the dealer. They can either ask for a card or stop, and if they get a total of over 21 then they have busted and have lost. If they do not bust, then the dealer plays and will ask for a card if his total is below 17 or stop if it is 17 or above. If the dealer busts, then the player has won; if not they compare totals, whoever has the highest wins, and it is a draw if both are equal. That is one game of Blackjack, they will keep playing with the same pile until there are 20 cards or less left.

There are many ways to count cards in Blackjack, however one of them is more commonly used, the Hi-Lo method. This is the one that will be studied in our project. The basic principle of this method is to keep track of the ratio of high cards and low cards. Here is an explanation of how its done. A value is assigned to some cards: the cards 2, 3, 4, 5 and 6 will have value +1 and the cards 10, Jack, Queen, King and Ace will have value -1, while the cards 7, 8 and 9 have a value of 0. The method uses a running count, meaning you start at the beginning of a new pile at 0 and add the value of every card seen to the total. For example, if you play a game and get a Jack and a Queen, while the dealer gets a 10 a 3 and a 7, then the running count is -2. It is not as simple as that though; this count is not a true count since a -2 when there are still 200 cards in the pile is not worth as much as when there are only 50. So, you must divide your count by the number of decks left, let's say this was our first hand we would divide by 6 and get a true count of -1/3. How would we go about interpreting this number? A negative number means that more big cards (10+) have been delt, while a positive number means that more small cards (6-) have been delt. A positive number is advantageous to the player while a neutral or negative number is advantageous to the dealer. Players will bet more when the true count is high and positive and bet progressively less as the count decreases.

The states of this simulation will be of 4 dimensions

X: The total of the player, we will take every instance when it is higher than 12. Values range from 12 to 21.

Y: The visible card of the dealer. Values range from 2 to 11

Z: The number of usable aces for the player. Values are 0 or 1

K: The true count of cards, this number will be rounded and limited to

{-3, -2.5, -2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2, 2.5, 3}

While the possible actions are

L: The action taken. 1 is asking for a card and 2 is stopping

The rewards for the simulation are -1 for a loss, 0 for a draw and 1 for a victory. The rewards are given after every game, not only at the end of the pile.

The objective for this scenario will be to us Monte Carlo every-visit policy improvement to try and find an optimal policy for each state. The optimal policy will be telling the player whether to ask for a card or not (1 or 2) in every state. We will also look at the convergence of the state value function for different states and see if we can find a correlation between the card count and your chances of winning.

Independent and identically distributed cards

When dealing with infinite cards without replacement the game becomes much simpler. We will not go over the rules again as they are the same as explained previously, except there is no more card counting now.

The states of this simulation will be of 3 dimensions

X: The total of the player, we will take every instance when it is higher than 12. Values range from 12 to 21.

Y: The visible card of the dealer. Values range from 2 to 11

Z: The number of usable aces for the player. Values are 0 or 1

While the possible actions are

L: The action taken. 1 is asking for a card and 2 is stopping

However, what would be interesting would be trying out an optimal policy that is not binary, meaning seeing if it could be better to ask for a card 60% percent of the time in a state rather than 100%. To try this will use the REINFORCE method since it allows us to compute a percentage-based policy while also improving to find the optimal policy. We will test out different learning rates (alpha) and see which one has the best convergence and then compare it with using REINFORCE with a baseline. After having found an optimal policy, we will create a copy of it that is rounded, so everything less than 0.5 goes to 0 and over 0.5 goes to 1. We will simulate more games and evaluate the state value function of each policy using first-visit Monte Carlo policy evaluation method.

Analysis

Finite number of decks

The code

First, we must create functions to be able to simulate a game of Blackjack with n decks. The first function *GetDeck* simply sets up a pile of cards from n decks in a random order which will be used to draw from every turn. Then we have the function *counting* which takes in the current count of cards and the card to add to the count, then returns the new count. Finally, we have the main simulation function *SimulateBlackJackEpisodes* which will generate one episode of Blackjack and return the results. For this exercise, we consider playing games until the pile of decks runs out (20 cards left or less) to be one episode. The function will call the *GetDeck* function to get its random pile, then will start dealing out the cards, two to the dealer and two to the player. The player will play depending on the policy that was passed as a parameter in the function, and then if he has not busted the dealer will play. To ensure correct exploration, the current optimal policy for the player will not always be used, we employed an epsilon-greedy approach to sometimes pick the non-optimal policy. After the game is played, the *counting* function is called to update the count of cards depending on the cards that the player has seen. For example, if the player has busted, then he does not see the hidden card from the dealer so he cannot add it to the count. For every step of the game, the player total, dealer visible card, number of usable aces for the player, true count, action taken, and reward are recorded in their

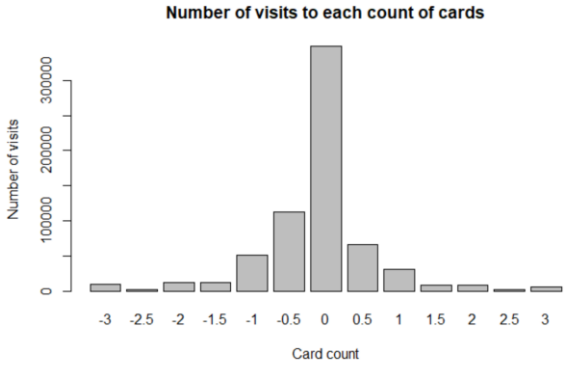
respective vector. After having simulated through the whole pile, these result vectors are returned.

Now that our simulation is setup, we must employ an every-visit Monte Carlo policy improvement method. We first decide on our initial values, we will use 6 decks, 10,000 episodes and start our epsilon at 0.5. We start our epsilon this high to make sure we explore enough through all the state-action pairs, we will gradually lower it by 0.1 starting after 5,000 episodes until it reaches 0. We have four arrays to keep track of while looping through the episodes. The optimal policy has the 4 state dimensions and a 1 or 2 in each position saying the optimal action, we initialize all states at 1. Then we have the action value function array which has the 4 state dimensions plus an extra dimension of size 2 for each action that can be taken, we initialize it at 0. We also have the number of sample arrays which will record the number of visits to every state-action pair, so it has the same dimensions as the action value function. Finally, we have the state value function which has the 4 state dimensions, it will be the maximum of both action for the action value function.

The first part of the loop simply calls the simulation function and records the result vectors. Then there is another loop which will loop though every step of the episode and will update the action value function for the appropriate state-action pair. After going through the whole episode, the new optimal policy and state value function is calculated, and we start over by calling the simulation function with our new policy.

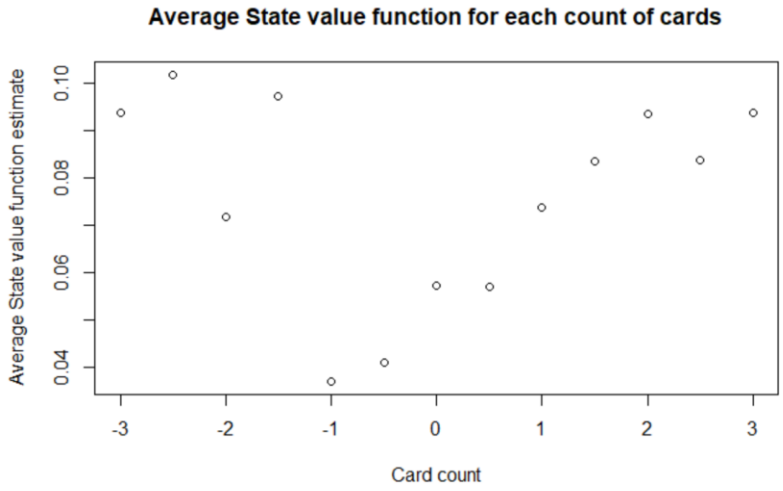
The results

Let us look at what results we get from this simulation. We first look at our sampling per true card count :



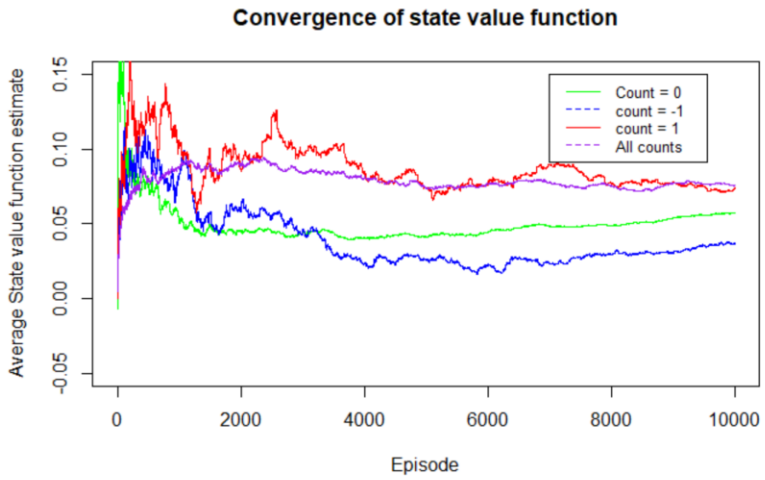
Although we use an epsilon-greedy approach to explore every action in every state, it does not seem to translate well to the card count. It seems that on average the true card count will rarely deviate from [-0.5,0.5] and will in most cases be 0. We will keep this in mind going forward with our analysis.

It is said that the higher the count the higher the chances of the player winning, so let us see if our simulation gives the same results by looking at the average state value function across all different card counts.



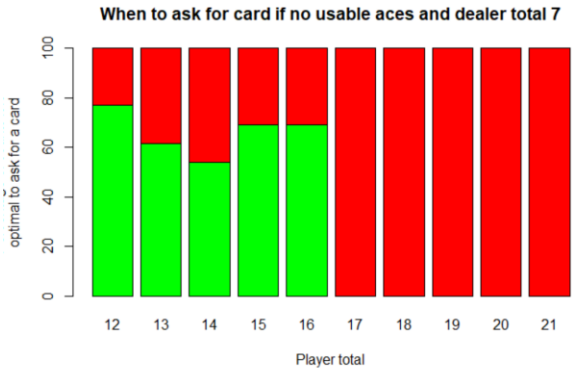
There does not seem to be a strong correlation, but if we only look at the card counts which are the most visited: [-1,1] we can see a clear upward trend in the state value function as the count increases. However, we do not believe this is enough to draw a conclusion.

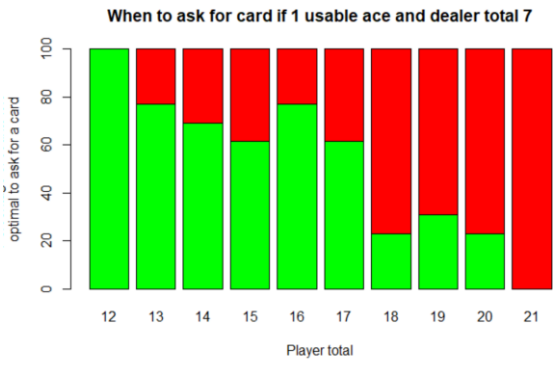
We can also look at the convergence of the state value function, we will observe it for the states 1,0,-1 and also the average across all states.

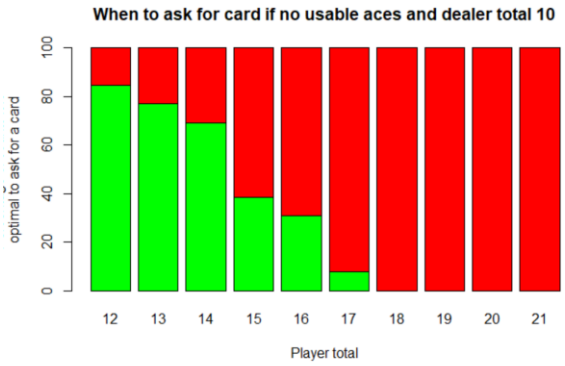


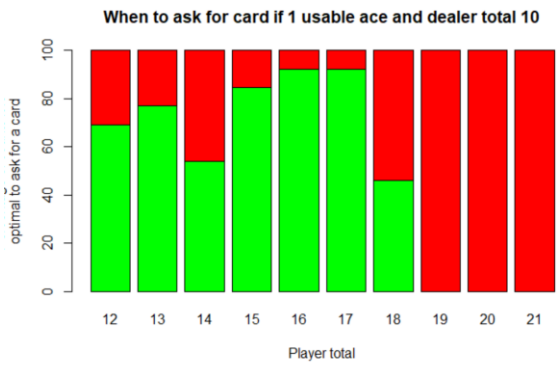
We can see that there seems to be convergence but still with some variance. For the count of 0, the most visited, we see some variation at first but after episode 2,000 it is much less and does seem to slightly increase as the optimal policy is refined. However, for the count of 1, there is a lot of variation even as we approach the 10,000 episodes. So it seems we do observe some convergence however we could probably get more precise estimates if we could visit more the other card counts.

Our initial goal was to find an optimal policy, so let us see if we have managed to do so. Without having a true theoretical value of the optimal policy for the states we have picked it is hard to know how precise our estimation is. Also, with over 2,600 states we cannot look at each of them individually. Here we have graphed out certain scenarios to look which differ on the visible card of the dealer and the number of usable aces. We look at the percentage of card counts where it is optimal to ask for a card or not through all player totals.









Just from these examples, there are many things to notice, except for a few more obvious player totals not all card counts have the same optimal policy. This could be because our sampling for some counts are not sufficient to get a real optimal policy. For example asking for a card when your total is 20 seems illogical. However, it could also be because in some card counts the chances of a big card coming out are too high and would lead to a bust so even if the player total is low he should not ask for a card. It seems that for the majority of card counts, we should ask for a card from 12 to 15 and not ask one for 19 and 20. Now were it gets a bit more split is from 16 to 18 and one of the big factors that seem to affect the policy is wether you have a usable ace or not. If you do not have one, the optimal policy is much more conservative and will not want you to ask for a card if your total is 17 or above. On the other hand, if you do have a usable ace you should ask for a card even if your total is higher as it is impossible to bust, so you should only stop if you think your total is high enough to win, in this case 19, 20 and 21.

Independent and identically distributed cards

The code

In this exercise, our functions to generate a game of Blackjack will be much simpler, since we will only generate one game every time and not keep track of the cards that have been dealt before. Though, one of the key differences is that is does not use a binary policy, it will take a percentage chance of asking for a card and will use a binomial random variable to decide the action. First, we have the function *GetCard* which simply returns a random card; but it returns

the value of the card so for a King it will return 10, same thing for a Jack. Then we have the main simulation function *SimulateBlackJackEpisode*, this will deal the cards to the player and the dealer. Then depending on the policy of the state *rbinom()* with the probability of the policy will be used to see if the player asks for a card or not. The game plays out the same way as before for the dealer. At the end, the function will return the player total the first time it passes 11, the dealer visible card, the number of usable aces at that time, the action taken at that time and the reward.

To get the optimal policy we will use the REINFORCE method mixed with the first-visit Monte Carlo simulation method. We want to first try and find an optimal learning rate to use, so we will test out (0.1,0.05,0.01). To do so we first set up our optimal policy matrix which has the 3 state dimensions, then another dimension of size n+1 to keep track of how it evolves over each episode and finally, a dimension of size 3 for each learning rate. We initialize the optimal policy at 50% for all states. We also setup our theta as an array of the 3 state dimension and another dimension of size 2 for each action. To update the theta we will use the formula:

$\theta_{t+1} = \theta_t + \text{alpha} * G * \nabla \ln(\pi(A_t|S_t, \theta))$

Where G is simply the reward of the episode and

$$\nabla \ln(\pi(a|s, \theta)) = \begin{bmatrix} 1_{\{At=1, St=s\}} - \pi(1|s, \theta) \\ 1_{\{At=2, St=s\}} - \pi(1|s, \theta) \end{bmatrix}$$

We also use

$$x(s, a) = \left[1_{\{At=1, St=s\}}, 1_{\{At=2, St=s\}}\right]$$

$$h(s, a, \theta) = \theta^T x(s, a)$$

$$\pi(a|s, \theta) = \frac{e^{h(s,a,\theta)}}{\sum_{b \in A(s)} e^{h(s,b,\theta)}}$$

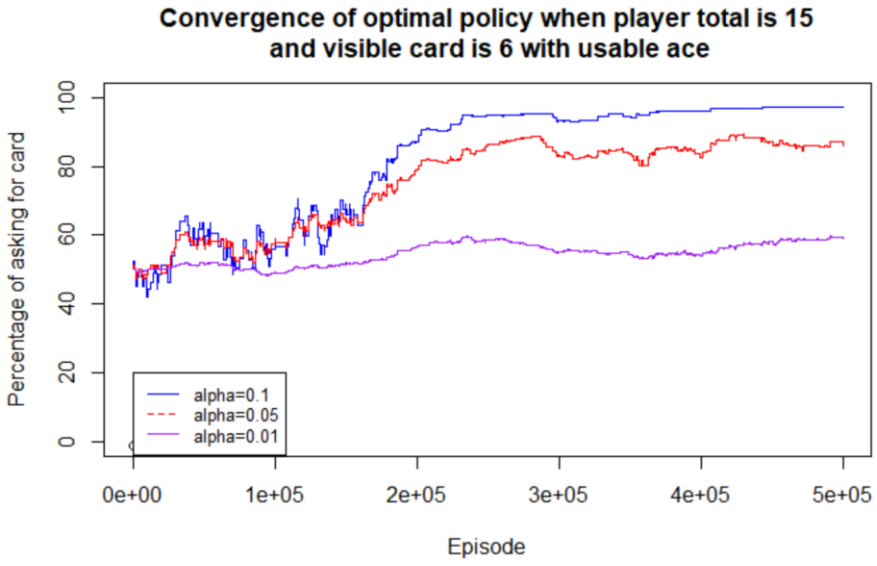
After having found an optimal learning rate, we use a very similar loop but this time using a baseline to see if we can get even better and faster convergence. The baseline

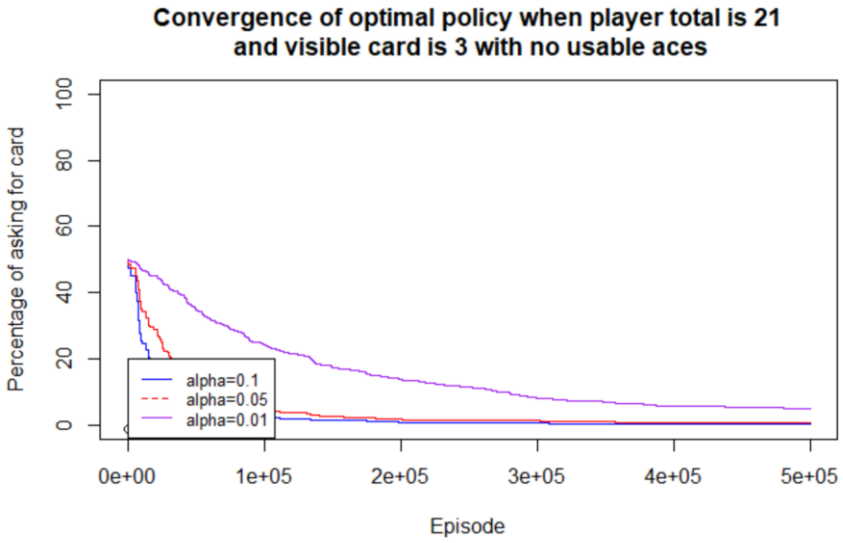
used will be the state value function, which means we will be updated every loop as new rewards are generated.

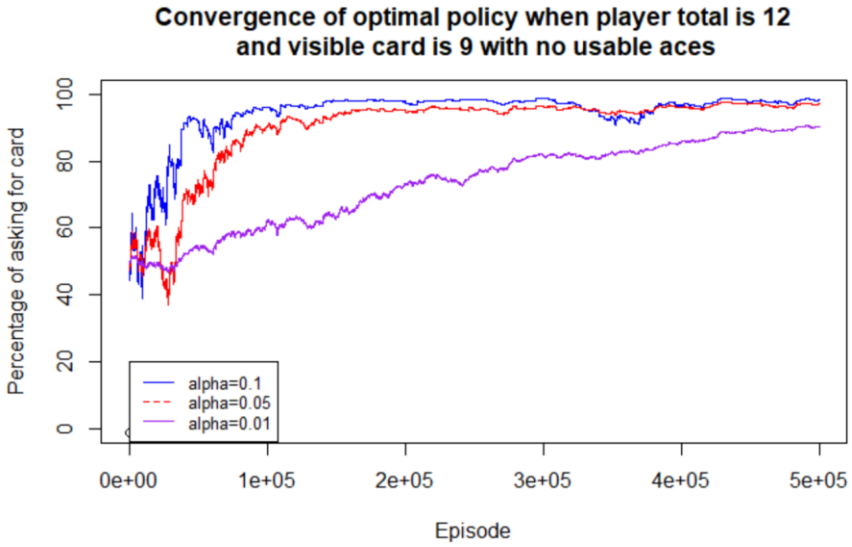
We then would like to see to if our percentage-based policy obtained would perform better than a binary policy so we will run a simulation using both policies and compare their respective state value functions. To get our binary policy we will simply round to the nearest digit our percentage-based policy, so if it is below 50% it will go to 0 and if it is 50% or above it will go to 100%.

The results

The first test we did was the regular REINFORCE with three different learning rates, after many tries we were able to narrow it down to 3 leaning rates which seemed to perform the best : {0.1, 0.05, 0.01}. We will look at three different states and compare the convergence of the optimal policy over 500,000 episodes to see which one performs best

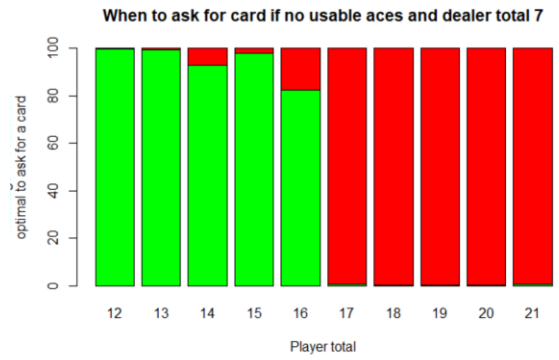


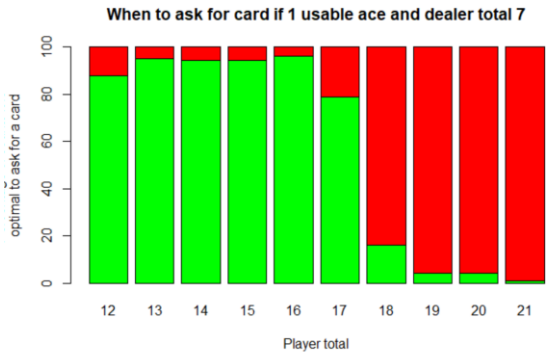


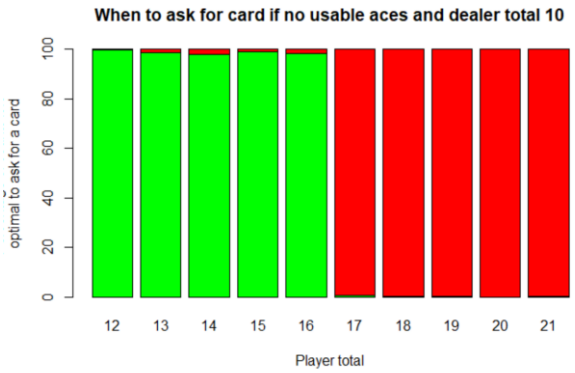


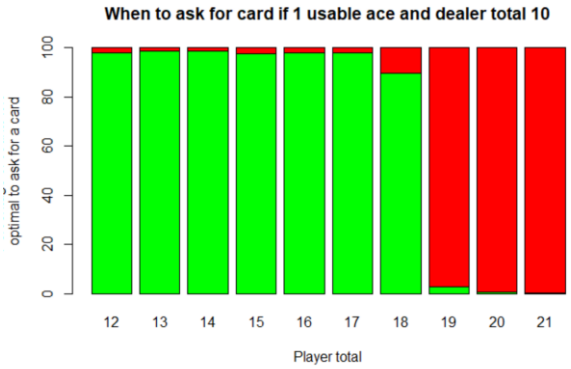
In all three states, we can observe that the alpha of 0.01 converges much slower and does not seem to converge to the same value as the two other learning rates. On the other hand, both the alpha of 0.1 and 0.05 seem to converge to a very similar value, the difference being the speed at which they do. While the alpha of 0.1 is faster to get to the value, it does have more volatility. Overall, the learning rate of 0.05 seems like it is optimal for this exercise since it converges much faster than the smaller learning rate while also being much less volatile than the bigger one.

Using the learning rate of 0.05 we can observe the optimal policy for each state. We will look at the same states as we did in the finite deck example, except this time the bars do not mean the same thing. The green part is the percentage of time it is optimal to ask for a card for each player total. Although this example does not consider the card count, it can be insightful to compare our results and see if the line up for the most part.





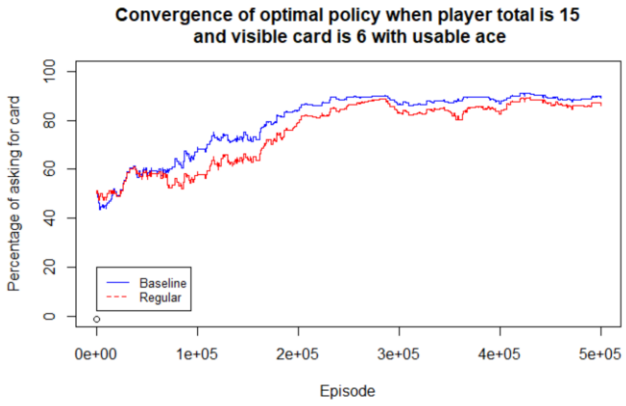


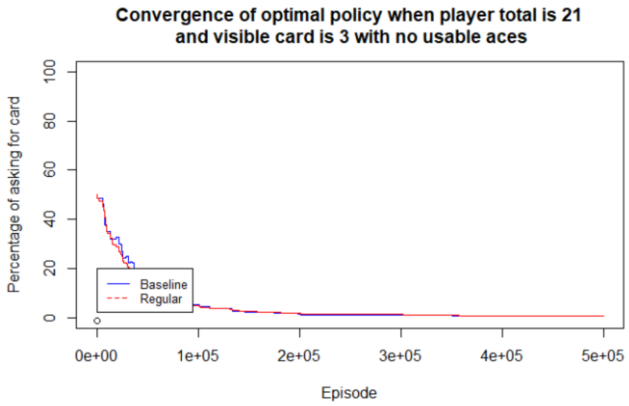


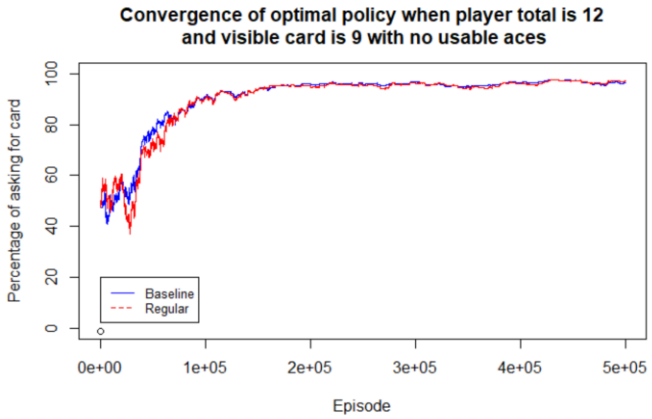
It seems that the results we get are different in some way to the ones using the card count, but they do share many similarities. The main visible difference Is that for every state the optimal policy is very definitive, it very rarely splits 50-50 on what is the optimal policy, either you should ask for a card most of the time or almost never. However, we notice the same pattern that the player can ask for a card when his total is higher when he has a usable ace. The cuttoff of when it is optimal to ask for a card when you have no usable ace is between 16 and 17, just like in our other exercise. It is more between 18 and 19 when you have an usable ace, also like in the finite deck example. Overall, it seems that this optimal policy is more precise then when considering the card count.

While the learning rate of 0.05 seemed to be the best comprimise of low volatility and fast convergence, it could be interesting to compare it with the REINFORCE with baseline method to see if we could get

even better convergence. We can compare the convergence of the optimal policy over 500,000 episodes for the baseline method against the regular REINFORCE both with a learning rate of 0.05.







From these graphs alone, it is hard to conclude whether the baseline is much better or not from the regular REINFORCE algorithm. Although, we can see slightly less variance in once state and

slightly faster convergence in another, we can say the performance benefits of the baseline method in this exercise are marginal.

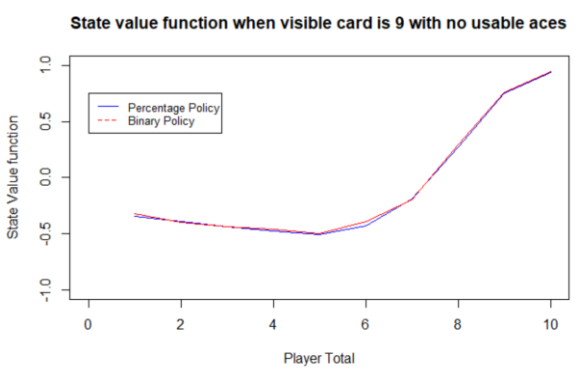
The final test we will conduct is to verify the relevance of our percentage-based policy. We will use our final optimal policy obtained through the REINFORCE with baseline method and create a copy of it but rounded to 0 or 1. We will then run 500,000 episodes of a first-visit Monte Carlo policy evaluation to obtain the state value functions for each policy. We will also keep track of the rewards for both the binary policy and the percentage-based policy. The first result we will look at is the number of wins, draws and losses.

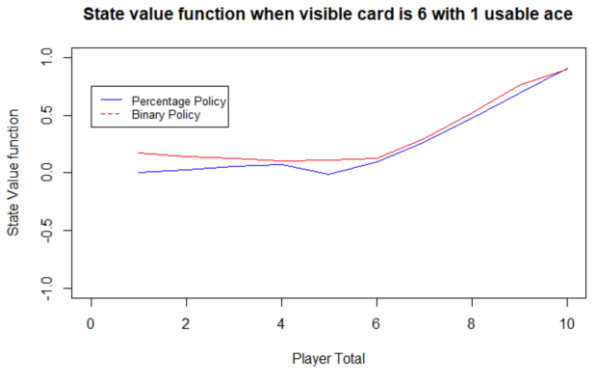
	Number of wins	**Number of draws**	**Number of losses**
Percentage Policy	214585	45342	240073
Binary Policy	215654	45979	238367

At first glance the binary policy seems to perform better since it has slightly more wins and draws, and less losses. Let us compute the average reward for each now, considering +1 for a win, 0 for a draw and -1 for a loss. We will also compute the 95% confidence interval.

	Average Reward	**95% Confidence Interval**
Percentage Policy	-0.05098	(-0.05362, -0.04834)
Binary Policy	-0.04543	(-0.04806, -0.04279)

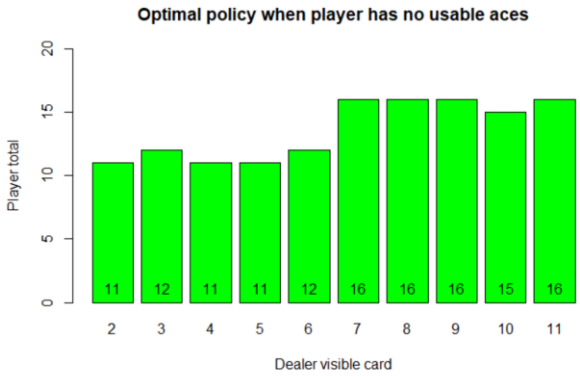
The difference in average reward between both is very small, but not negligible. Also, we can see that both confidence intervals do not overlap it seems safe to conclude that a binary policy will give a better result than a percentage-based policy. We can also look at the state value function to see if the results are the same, let us look at a few states.

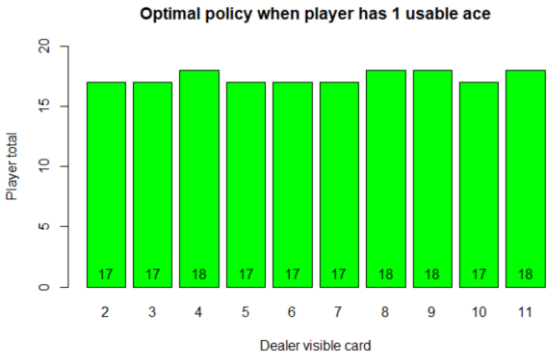




The state value function of the binary policy is always equal or higher than the state value function of the percentage policy so it seems clear that the player should use a binary policy.

However, this does not mean that the results we get from the REINFORCE method cannot be used, as we saw here, if we take the percentage results and round them to get a binary policy, we get a very strong optimal policy. Here is what the final optimal policy would look like.





The bars represent the last player total which he should ask for a card, depending on what the visible card is for the dealer. One interesting difference is that when the player has no usable aces, he should not even ask for a card above 11 or 12 if the dealer visible card is low. On the other hand, when he does have a usable ace, the policy seems very consistent across all the dealer visible cards and is obviously higher than when he has no usable aces, like we concluded before.

Conclusion

After many tests, using different methods and even different game structures, there are a few conclusions we can draw. The first being that if the player has a usable ace, he should ask for a card in more cases than if he does not have a usable ace. Also, that he should employ a binary policy of either always asking for one or never asking for one in every state. Regarding what is the optimal policy, we have failed to find a clear policy with regards to counting cards, mainly because the simulation did not visit the more extreme counts enough. However, when we did our other simulation with an infinite deck, we were able to find the best convergence to an optimal policy. After transforming this policy to binary, it seems like a very clear and simple optimal policy to follow. If you have no usable ace and the dealer card is below 7, ask for a card when your total is below 12. If you have no usable ace and the dealer card is 7 or above, ask for a card when your total is below or equal to 16. If you have a usable ace, no matter the dealer visible card you should ask for a card when your total is below or equal to 17 or 18.

Considering the first part of our project with the finite deck and counting of cards was not as successful, it would be interesting to try and get better results from another technique. Instead of generating a pile of cards and playing through the whole deck, we could generate a random count of cards and play one hand. When generating the random count, the probabilities of getting each card would be changed to what they should be in that count, this would make the exploration to every card count much better. With this new simulation method, we could most likely get better and more conclusive results when testing for the optimal policy.