Solving Blackjack with Monte Carlo Methods

Botond A. Nás
Department of Computer Science
Eötvös Lóránd University
Budapest, HU
khupib@inf.elte.hu

Abstract

Reinforcement learning is concerned with how agents should act in an environment to maximize reward. In most real life situations, the model for the environment, including the state transition probabilities, are not known. In such situations, the agent can instead learn from experience. One such class of methods are the Monte Carlo methods, which are offline methods that rely on sampling episodes to evaluate or estimate a policy. Using OpenAI Gym's Blackjack environment, this report aims to evaluate provided strategies and approximate optimal strategies for winning blackjack using Monte Carlo methods.

1 Introduction

The backbone of reinforcement learning is the estimation of value functions and discovering optimal policies. In order to understand the state-of-the-art learning algorithms and methods, it is important to understand the first and very basic methods, such as the Monte Carlo methods. Monte Carlo methods do not assume complete knowledge of the environment, instead they solve reinforcement learning problems based on averaging sample returns. The prime demonstration of Monte Carlo methods for value function approximation, and the topic of this report, is the popular card game blackjack. The goal of blackjack is to obtain cards of whose sum is as great as possible without exceeding 21. The policy of the player determines when they 'hit', or ask for another card, and when they 'stick', or stop receiving cards. Monte Carlo policy evaluation can be used to find the state-value function for the given policy. In such cases, the input is the policy to be evaluated, and the output is the state-value function determined using Monte Carlo policy evaluation. It is useful to evaluate a given policy, but far more interesting to find the best, or optimal, strategy instead. This report discusses the estimation of optimal strategies using Monte Carlo control methods. The goal of this report is to understand how various Monte Carlo methods work through their application to blackjack using the blackjack environment of OpenAI Gym.

2 Related Work

This problem is most famously presented by Sutton and Barto in their book *Reinforcement Learning: An Introduction* [Sutton and Barto, 1998]. They use the problem as a demonstration of Monte Carlo methods and their advantages over dynamic programming in certain situations. Following this description, most implementations focus on using the described Monte Carlo methods to solve the problem, as this report does. There are, however, examples of implementations that use temporal difference (TD) learning methods. In his report, de Granville explores reinforcement learning as a means of approximating an optimal blackjack strategy using the Q-learning algorithm. The Q-learning algorithm allows learning to take place during play, making it a good choice for the blackjack domain [de Granville, 2018]. His results are promising, showing that the agent converged to a near optimal policy. This result is supported by Handa's report which compares Q-learning and SARSA to the strategy proposed by Edward Thorpe in his book "Beat the Dealer". The report concludes that

Q-learning approximates a better strategy then SARSA, while both outperform Edward Thrope's strategy.

3 Problem Description

The goal of blackjack is to obtain cards of whose sum is as great as possible without exceeding 21. The environment I used corresponds to the version of the blackjack problem described by Sutton and Barto. In this environment, the player competes against a fixed dealer with an infinite deck. Face cards (Jack, Queen, King) have value 10 and each numbered cards' value is its corresponding number. Aces can either count as 11 or 1. The game starts with each competitor (player and dealer) having one face up and one face down card. The player can request additional cards (hits) until they decide to stop or exceed 21. After the player stops requesting cards (sticks), the dealer reveals their face down card, and draws until their sum is 17 or greater. If the dealer goes bust, the player wins. If neither player nor dealer busts, the outcome (win, lose, draw) is decided by whose sum is closer to 21.

Playing blackjack can be formulated as an episodic finite MDP, where each game of blackjack is an episode. The reward for winning is +1, drawing is 0, and losing is -1. Rewards are not discounted ($\gamma = 1$). The player chooses from two actions: to hit or to stick. The states are determined by the player's cards and the dealer's showing card. According to the description of the game in Sutton and Barto [1998], if the player holds an ace that he could count as 11 without going bust, then the ace is said to be *usable*. In this case it is always counted as 11 because counting it as 1 would make the sum 11 or less, in which case there is no decision to be made because, obviously, the player should always hit. With these rules, there are exactly 200 states. The player makes decisions based on three variables: their current sum, the dealer's one showing card, and whether or not the player holds a usable ace. The player's policy determines when the player sticks and when he hits, depending on his current sum. To evaluate a player's policy by a Monte Carlo approach, one simulates many blackjack games using the policy and averages the returns following each state. Approximating optimal policies, on the other hand, involves Monte Carlo control methods which utilize the idea of policy iteration.

4 Methods

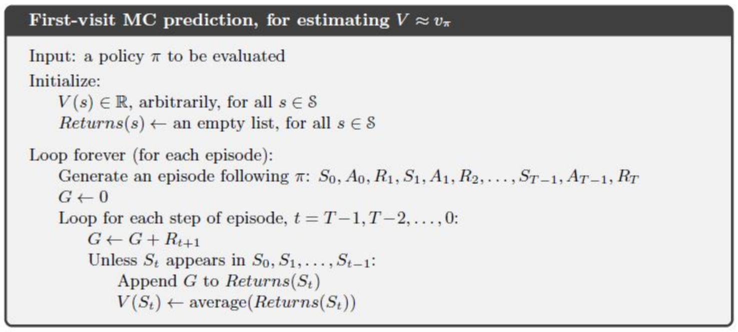
The solutions written in Python are inspired by Britz's solution. In his GitHub repository *reinforcement-learning/MC*, Britz presents various implementations of Monte Carlo methods applied to the Blackjack-v0 environment. These include Monte Carlo policy evaluation for policy evaluation and On-Policy and Off-Policy Monte Carlo control for finding the optimal policy. The solutions are written in Python and follow the algorithms presented in Sutton and Barto [1998].

4.1 Monte Carlo Policy Evaluation

The objective is to estimate the state-value function, $V(s)$, given a policy. The value of a state is the expected return – expected cumulative future discounted reward – starting from the state. A way to estimate it from experience, then, is to average the returns observed after first visits to that state. This method is called *first-visit MC*. As more returns are observed, the average converges quadratically to the expected value. The algorithm I used to implement this method is given in Figure 1. Given a policy and the number of episodes, my implementation determines the value function for the given policy.

4.2 Monte Carlo Control

Monte Carlo evaluation can be used in control as well. Monte Carlo control methods follow the pattern of policy iteration. In policy iteration both an approximate policy and an approximate value function are maintained. The value function is repeatedly altered to more closely approximate the value function for the current policy, and the policy is repeatedly improved with respect to the current value function [Sutton and Barto, 1998]. The Monte Carlo version of classical policy iteration performs alternating complete steps of policy evaluation and policy improvement, beginning with an arbitrary policy and ending with the optimal policy and optimal action-value function. Policy improvement is done by making the policy greedy with respect to the current value function [Sutton and Barto, 1998]. With this method, however, it can occur that many state-action pairs may never be



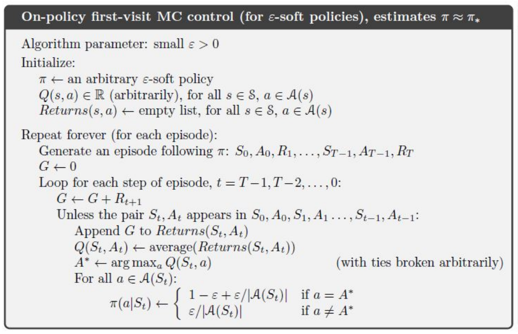
```
First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy $\pi$ to be evaluated
Initialize:
    $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
    $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
    Generate an episode following $\pi$: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow G + R_{t+1}$
        Unless $S_t$ appears in $S_0, S_1, \ldots, S_{t-1}$:
            Append $G$ to $Returns(S_t)$
            $V(S_t) \leftarrow$ average($Returns(S_t)$)
```

Figure 1: Algorithm for Monte Carlo Prediction from Sutton and Barto [1998]

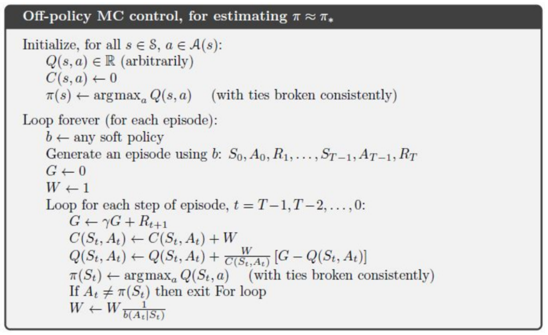
visited. The only general way to ensure that all actions are selected infinitely many times is for the agent to continue to select them. The two approaches I used to ensure this are *on-policy* control with ϵ-greedy policies and *off-policy* control with weighted importance sampling. A comparison of the two algorithms is given in Figure 2.

4.3 On-Policy Monte Carlo Control with ϵ-Greedy Policies

On-policy methods attempt to evaluate or improve the policy that is used to make decisions [Sutton and Barto, 1998]. I implemented a solution to use an ϵ-greedy policy instead of a full-greedy policy. Most of the time, an ϵ-greedy policy chooses an action that has maximal estimated action value (greedy), but with probability ϵ it instead selects and action from the action space at random. Given an ϵ value, a discount factor γ, and the number of episodes, my implementation returns the optimal policy and an action-value function, Q, as a dictionary mapping states to action values. From this, a state value function is created by picking the best action at each state.



```
On-policy first-visit MC control (for ε-soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\varepsilon > 0$
Initialize:
    $\pi \leftarrow$ an arbitrary $\varepsilon$-soft policy
    $Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
    $Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):
    Generate an episode following $\pi$: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow G + R_{t+1}$
        Unless the pair $S_t, A_t$ appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
            Append $G$ to $Returns(S_t, A_t)$
            $Q(S_t, A_t) \leftarrow$ average($Returns(S_t, A_t)$)
            $A^* \leftarrow \arg\max_a Q(S_t, a)$          (with ties broken arbitrarily)
            For all $a \in \mathcal{A}(S_t)$:
                $\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$
```



```
Off-policy MC control, for estimating $\pi \approx \pi_*$

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s,a) \in \mathbb{R}$ (arbitrarily)
    $C(s,a) \leftarrow 0$
    $\pi(s) \leftarrow \arg\max_a Q(s,a)$    (with ties broken consistently)

Loop forever (for each episode):
    $b \leftarrow$ any soft policy
    Generate an episode using $b$: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    $W \leftarrow 1$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
        $\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$    (with ties broken consistently)
        If $A_t \neq \pi(S_t)$ then exit For loop
        $W \leftarrow W \frac{1}{b(A_t|S_t)}$
```

Figure 2: Algorithm for On-Policy and Off-Policy MC from Sutton and Barto [1998]

4.4 Off-Policy Monte Carlo Evaluation with Weighted Importance Sampling

The distinguishing feature of on-policy methods is that they estimate the value of a policy while using it for control. In off-policy methods, these functions are separated. The policy used to generate behavior, called the behavior policy, may in fact be unrelated to the policy that is evaluated and improved, called the estimation policy. An advantage of this separation is that the estimation policy may be deterministic (e.g. greedy), while the behavior policy can continue to sample all possible actions. My implementation uses a random policy as the behavior policy and a greedy policy as the estimation policy. With the input of the behavior policy, a discount rate γ, and the number of episodes, it returns the action-value function and optimal estimation policy. For visualization and comparison purposes, I create a state-value function from the action-value function by picking the best action at each state.

5 Results

Using Monte Carlo policy evaluation, I estimated the state-value function of the policy described in the problem description from Sutton and Barto where the player sticks only when their sum is 20 or 21. Comparing evaluations with a varying number of episodes demonstrate the converge of the prediction. As mentioned before, the discount rate γ is 1. The approximated state-value functions are visualized in Figure 3.

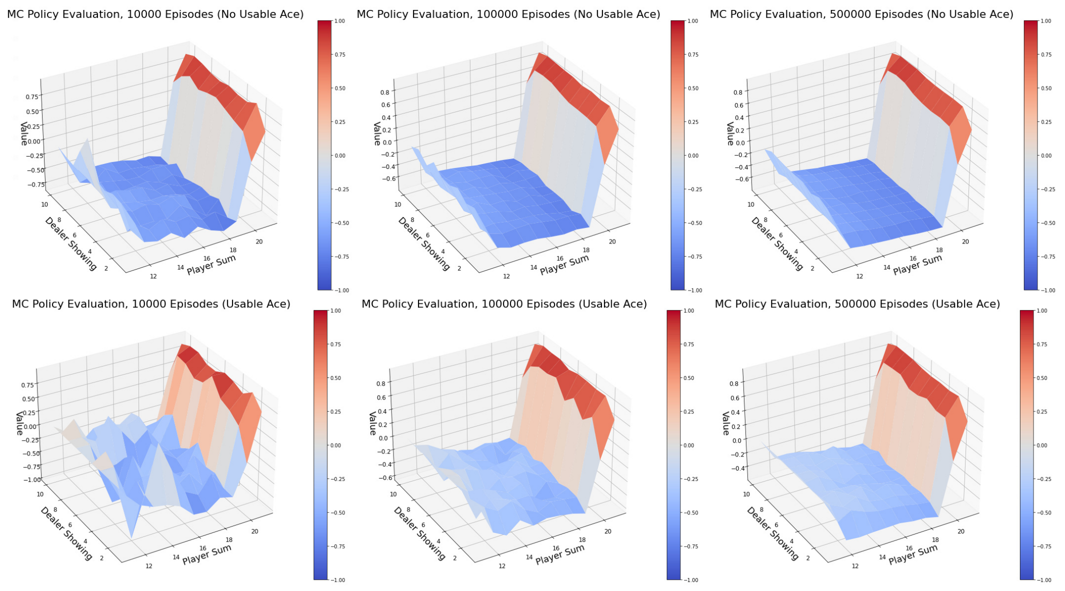


Figure 3: Comparison of MC policy prediction evaluations

For both cases, usable and no usable ace, evaluating more episodes resulted in a smoother function as the approximation converges with more samples. The last two rows have a high value because the player hits until reaching 20 or 21 according to the policy. That means that the player would risk going over 21 by hitting when their sum is close to 20, resulting in low value right before 20. When the player's sum is 20 or 21, they stick and have a very high probability of winning because the dealer draws until it reaches 17. In the case when the player does not have a usable ace, when his sum is less than 12, the value is higher than if he does have a usable ace. This is because the player is less likely to go over 21 but still might reach 20 or 21. Overall, however, the winning rate is small no matter what the dealer's showing card and the player's sum is, proving that this policy is not the optimal policy.

Using on-policy and off-policy methods, I approximated the optimal policy and plotted their respective state-value functions (see Figure 4). Each simulation sampled for 500,000 episodes. Both control methods approximated a policy more optimal than the original policy I tested. The state-value functions of the both approximated optimal policies exhibit higher overall winner rates, not deviating from each other much.

Evaluating the trained policies provides a clearer comparison between the methods. Running each policy approximation method for one million episodes, I evaluated the resulting policies by playing 100,000 games against the dealer with each policy five times and averaging the results (see Table 1).

Method (policy)	Wins	Draws	Losses
Original Policy	29,433.6	5,880.4	64,686.0
Policy approximated by On-Policy MC	43,032.8	8,914.4	48,052.8
Policy approximated by Off-Policy MC	41,836.6	7684.6	50,478.8

Table 1: Average game results

The original policy from Sutton and Barto [1998] acted as the control. The two approximated policies performed better (had a better winning rate) then the control policy, having nearly 50% more wins.

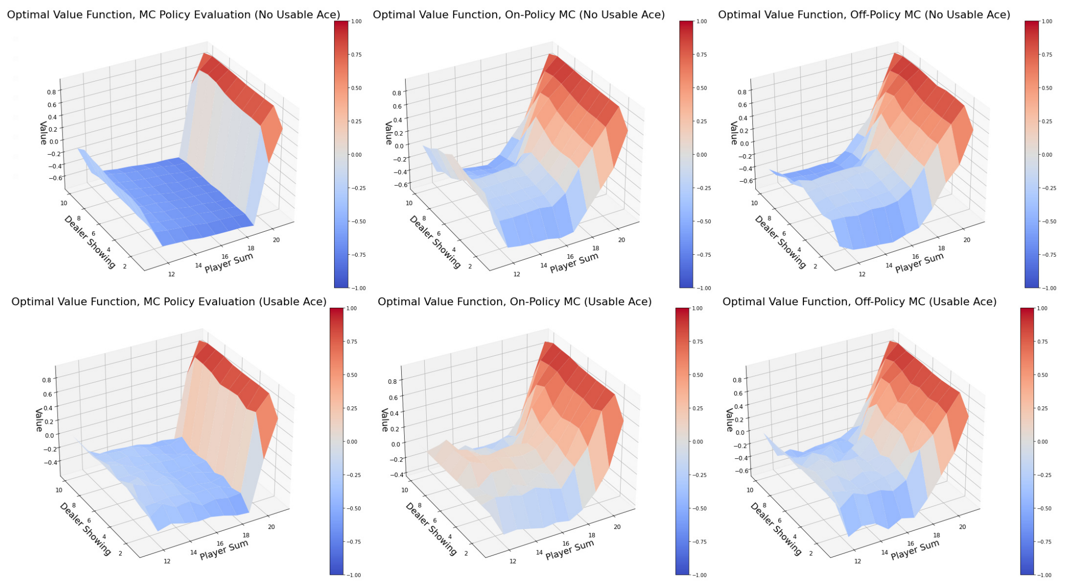


Figure 4: Comparison of approaches (500000 episodes)

Between the two control methods, the policy approximated by the on-policy MC with ϵ-greedy policies outperformed the policy approximated by off-policy MC with weighted importance sampling. The former had about 2% more wins and about 4% less losses than the latter. These results are congruent with the value functions visualized in Figures 3 and 4, where the policy approximated by on-policy MC had the highest overall values over all states.

6 Conclusion/Future Work

Monte Carlo methods learn state-value functions and optimal policies by sampling episodes, allowing them to learn optimal behavior directly from interaction with the environment, with no model of the environment dynamics. Understanding Monte Carlo methods is trivial to grasp more advanced reinforcement learning methods, such as temporal difference learning. Playing blackjack, the popular casino card game, is a model reinforcement problem to demonstrate how Monte Carlo methods find state-value functions and optimal policies. Monte Carlo policy evaluation estimates the state-value function of a predefined policy. If more episodes are sampled, the state-value function converges, giving us an accurate approximation.

To find the optimal policy, control methods are used which follow the overall schema of policy iteration. On-policy and off-policy Monte Carlo control methods approximate action-value functions which are then used to improve the policy without requiring a model of the environment. Maintaining sufficient exploration is important for control methods, as it is not enough to select the actions currently estimated to be the best because there might be alternative, better, actions. On-policy methods try to find the best policy that still explores, while off-policy methods learn a deterministic optimal policy that may be different from the policy followed. When applied to blackjack, the two approaches yield similar optimal policies and similar state-value functions. Given more time, I would compare the convergence speed of on-policy and off-policy Monte Carlo methods in more detail. I would also compare the performance of off-policy methods with varying behavior policies, observing their convergence speed.

References

D. Britz. reinforcement-learning/mc. `https://github.com/dennybritz/reinforcement-learning/tree/master/MC`, 2019.

C. de Granville. Applying reinforcement learning to blackjack using q-learning, 2018. URL `https://cs.ou.edu/~granville/paper.pdf`.

I. Handa. Learning to play blackjack with deep learning and reinforcement learning, 2019. URL `https://i.cs.hku.hk/fyp/2018/fyp18013/reports/InterimReport_3035238565.pdf`.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998. URL `http://www.cs.ualberta.ca/~sutton/book/the-book.html`.