Reinforcement Learning: Monte Carlo Methods

Mateo Guaman Castro

October 3, 2018

Abstract

In this homework assignment, a Monte Carlo on-policy control method was implemented to learn the optimal policy for a simulation of a racing car navigating a racetrack. In this problem, the objective is for the racing car to get to the finish line as fast as possible. Monte Carlo methods are widely used in simulations to estimate unknown values, and in this case they are used to find the optimal policy of a reinforcement learning agent. As shown in this homework assignment, Monte Carlo methods are an effective, although not the most efficient, way of estimating $\pi*$, the optimal policy for an agent.

1 Background

In reinforcement learning, two of the most important problems are estimating value functions for actions and values and discovering the optimal policy for an agent. Depending on the type of problem, there are multiple ways of estimating these values and discovering the optimal policy for an agent to follow. These methods have different properties and require that different conditions be met. For example, one theoretically important, but certainly not ideal, way of solving these problems is to use dynamic programming methods. However, dynamic programming methods require a model for the environment as a Markov Decision Process. This is usually not available to the agent, as it is very unlikely that the agent will know beforehand all of the state values and dynamics. And even if the agent had that information available, at that point the problem would be more related to planning than to learning.

Monte Carlo methods are another way of solving these two problems. These methods are model-free methods, meaning that they do not require prior knowledge of the dynamics of the environment. Monte Carlo methods rely solely on experience, that is, sample sequences of the states, actions, and returns of the agent. In order to obtain value functions, Monte Carlo methods average the returns obtained from selecting certain actions at certain states. It is important to note that in this case, returns are not the same as rewards. Returns are defined as:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

This means that the return at time t depends on all the future rewards until the end of an episode, scaled by a power of γ, which is called the discount factor.

One issue with Monte Carlo methods is that many of the state-action pairs may never be visited, since the values depend on the averages of the state functions that have been visited. This problem is called maintaining exploration, and there are multiple ways to fix it when using Monte Carlo methods. One such way is to ensure that at the start of every episode, the starting state-action pair is selected randomly so that in infinitely many episodes, all of the states are chosen. However, in most real world applications it is impossible to start at any of the possible states of the agent. There exist two other methods used to maintain exploration: on-policy and off-policy methods. In on-policy methods, the agent tries to improve the policy that is used to determine the state trajectories. These policies are generally soft policies, meaning that the probability of choosing any action at any particular state is greater than 0. In off-policy methods, the agent determines the optimal policy using a different policy that is used to generate the data. In this assignment, we are only concerned with on-policy methods. The main drawback of on-policy methods is that the policy converges to the optimal policy allowed by the soft policy instead of the absolute optimal policy, as happens with off-policy methods.

2 Motivation

Monte Carlo methods are the first realistic reinforcement methods learned in this course, since they do not require a model of the environment, like DP methods, and they can handle problems with multiple states, unlike K-Armed Bandit problems. These are the first experience based methods we learned, and many, more efficient, methods explored in later chapters of the book [1] are based on the positive qualities of Monte Carlo methods. For example, Temporal Difference (TD) learning methods use a combination of Monte Carlo and DP methods. In general, however, Monte Carlo methods are found in a wide spread of fields in science and engineering. It is interesting to see how they are used in the field of Reinforcement Learning to estimate value functions.

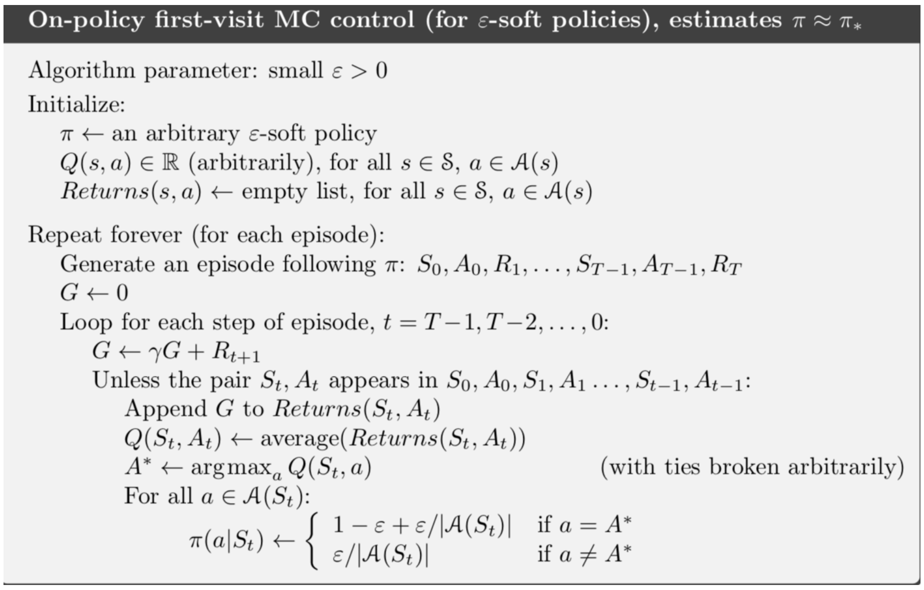
3 Experiment

3.1 Hypothesis

On-policy Monte Carlo methods are mathematically guaranteed to converge to the true value functions, according to Sutton and Barto [1]. Therefore, it is expected that the action value functions will converge to the expected value and that the agent will learn the optimal policy given enough episodes. However, it is also expected that these Monte Carlo methods will be very computationally expensive, since they update their value functions in an episodic way.

3.2 Methodology

There are two parts to the experiment. The first part is the implementation of an on-policy first-visit Monte Carlo control method that estimates the optimal policy. The ϵ-soft policy used in this implementation is ϵ-greedy. The objective of the agent is to navigate the racetrack given the limitations of the state in the fastest possible time. The environment is defined as a grid that models a race-track with walls at the side of the track, based on the Racetrack mathematical problem published on the the January 1973 edition of Scientific American by Gardner [2], and then discussed by Barto et al. in [3]. This experiment follows the algorithm presented in [1], which can be seen in Figure 1



```
On-policy first-visit MC control (for $\varepsilon$-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, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        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 $a \in \mathcal{A}(S_t)$:
                $\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$
```

Figure 1: Algorithm for On-Policy first-visit Monte Carlo control (optimal policy estimation) for ϵ-soft policies

Since the main objective of the experiment was to experimentally demonstrate the convergence properties of a functioning implementation of on-policy Monte Carlo control, the second part of the experiment explored variations of the original problem and the implications of changing the hyper parameters and the environment of the problem.

3.3 Implementation

As mentioned before, the implementation of the on-policy follows the algorithm in Figure 1. The original domain in which the agent was tested was on a bidirectional racetrack, where the agent was only able to go up or right at speeds from 0 to 4. Whenever the agent hit the wall, it would spawn again at a random position on the starting line. The episode only ended when the agent got to the finish line. After testing that it worked in these type of domains, the same method was tested in different environments to test how it would work out. A second version of the code allowed the agent to move in all four directions: up, down, left, right, and it allowed velocity increments of -4 to 4. This second version was used to test Monte Carlo methods under a larger state space.

The implementation of the state of the agent consisted of a 4-element tuple:

$$S(t) = (row, column, v_r, v_c)$$

. The action had 9 actions available at each state: modify either of its velocity components by +1, 0, or -1. Actions were also stored as 2-element tuples:

$$A(t) = (\Delta v_r, \Delta v_c)$$

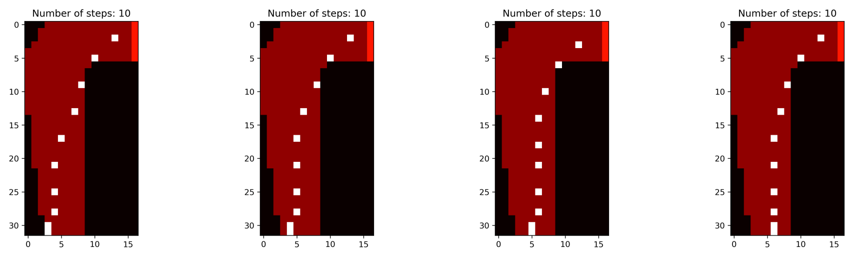
The Q and Return values were stored in dictionaries that had a 2-element tuple in order to be able to access them in a modular way:

$$Q(s, a) = ((row, column, v_r, v_c), (\Delta v_r, \Delta v_c))$$

For all runs of the algorithm, the Monte Carlo control method was run for 10000 runs, with $\gamma = 0.9$, and with $noise = 0.1$. The value used for epsilon depended on the problem at hand. At first, epsilon was set to $\epsilon = 0.1$. However, this value was too low, so it was later changed to a constant value of $\epsilon = 0.3$ that proved to be better in practice than all the other ones tested for the main problem. Finally, for very computationally expensive environments, epsilon was set to vary depending on the current time step, with a value of $\epsilon = \frac{1}{t+1}$. One modification of this value was used in order to make exploration last longer while still reducing its value with time: $\epsilon = min(\frac{10}{t+1}, 1)$

4 Results and interpretation

4.1 Base Problem



(a) 10 Steps (b) 10 Steps (c) 10 Steps (d) 10 Steps

Figure 2: Base simulation of racetrack using on-policy MC methods

After setting up the environment and the algorithm for the agent, a few interesting results were found. The main point of the experiment was to demonstrate the convergence of the policy to the optimal policy. As it can be seen in Figure 2, the policy function in four different runs of the Monte Carlo on-policy control algorithm, using ϵ-greedy as its soft policy, converges to the optimal policy. This can be seen from each of the runs taking 10 steps to reach the finish line, even though the agent starts at a different locations. Thus, this is experimental validation of the property mentioned in the book that Monte Carlo methods will converge to the expected value functions given infinite time. The initial trajectory of these problems usually took just under 1000 time steps, meaning that after running the MC control method, the agent went from having no prior knowledge of the problem to being able to get to the finish line optimally, in only 10 steps. This is a speed improvement of about 100 times.

4.2 Different values of ϵ

In order to gain a better understanding about how the chosen policy determines the performance of the algorithm, the value of ϵ used in the ϵ-greedy policy was varied. At first, the value selected for the simple bidirectional problem shown in the previous subsection was $\epsilon = 0.1$. However, it was soon noted that, with this value, the control method took a very long time to finish the first episode. This was fixed by increasing the value to $\epsilon = 0.3$. This value made the algorithm run much faster at the beginning, while providing enough greedy transitions as to not slow down the learning process too much. Out of curiosity, a higher value of epsilon was chosen, $\epsilon = 0.5$, but this as well as the time-varying epsilon values proved to converge slower to the optimal policy. For the problems in the next subsection, however, these time-varying values for ϵ worked better, as will be discussed.

4.3 Different Environments

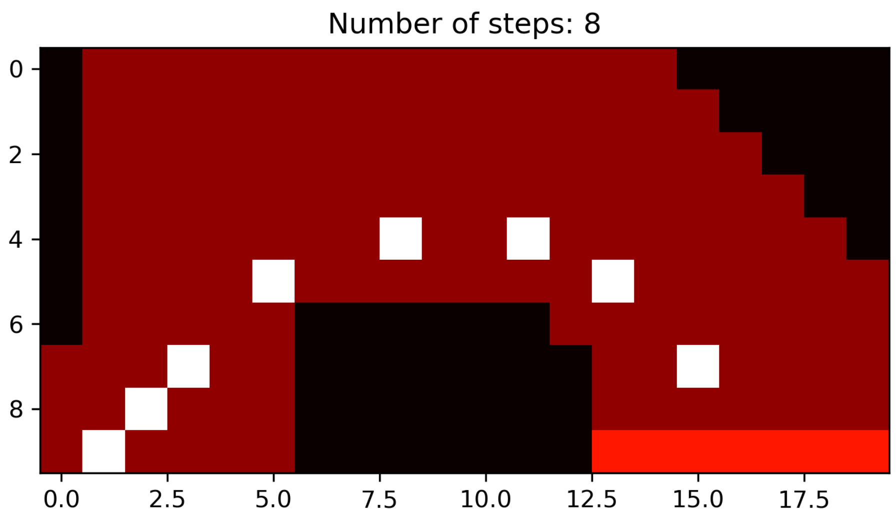


Figure 3: n-shaped, four-directional track

Two alternative cases were considered for Monte Carlo on-policy control methods. In particular, environments where the state space became larger by a significant amount were considered. These new environments were four-directional, and they required the agent to turn in, at least, three directions. The first environment was a simple n-shaped track, as shown in Figure 3. In this problem, the value for epsilon needed to be set as a time-varying parameter since a lot of exploration was needed at the beginning of the algorithm for the agent to roughly determine the shape of the track. However, keeping the epsilon value as high as was needed in the beginning would have become detrimental as time went on since it was too computationally expensive to keep exploring. The first episode of the algorithm took 76256 steps to arrive at the finish line. The optimal solution took 8 steps, which means that the Monte Carlo control method caused the agent to learn to get to the finish line almost

10000 times faster than the first time it got there. This example shows both the benefits and the drawbacks of Monte Carlo methods. The benefit is that the algorithm is guaranteed to converge to the optimal policy function eventually. However, when the state space becomes larger, Monte Carlo methods become prohibitively expensive.

The second case that was considered was the original track published by Barto et al. in [3]. This was a larger track with more turns, which required the agent to move in all 4 directions to get to the finish line. This track can be seen in Figure 4

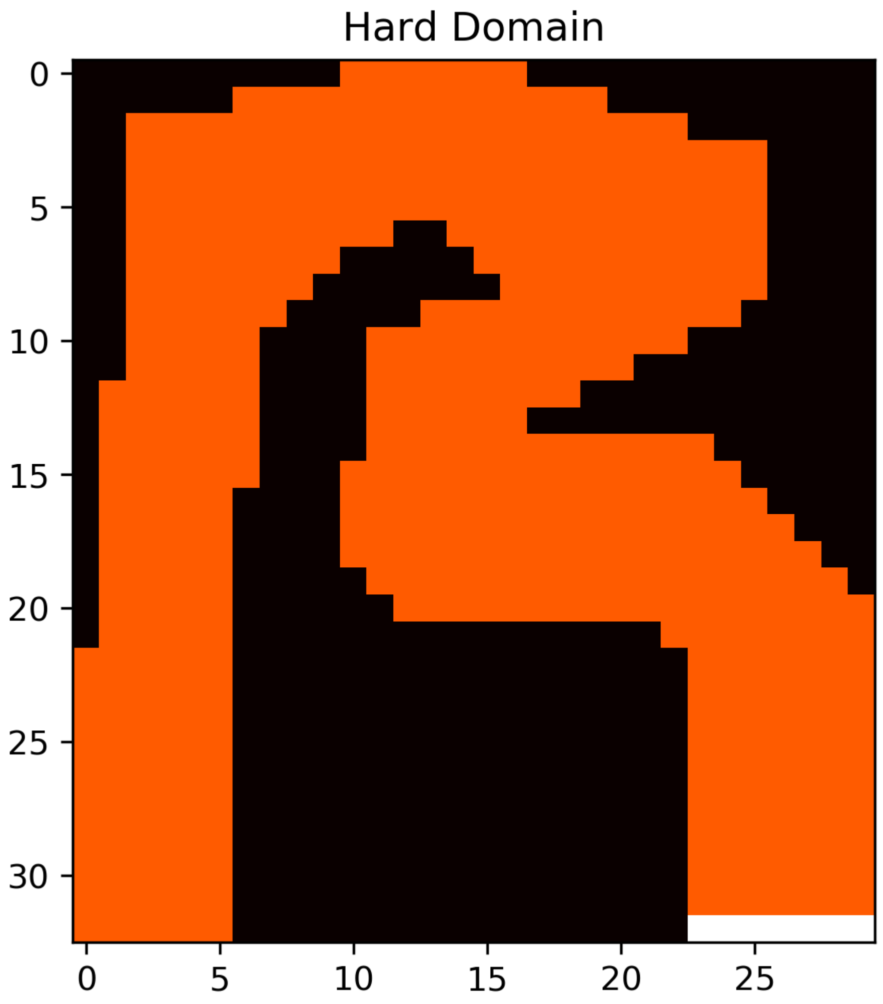


Figure 4: Hard track from original publication by Barto et al.

This track turned out to be too computationally expensive or Monte Carlo methods given the time frame of this experiment. The learning algorithm was left to run for more than two hours and it was not able to compute a first estimate from the first episode. This clearly shows just about how resource intensive Monte Carlo methods can be, which should be a hint towards the existence of better learning methods out there.

5 Conclusion

In this homework assignment, the convergence properties of Monte Carlo estimation methods were experimentally proven by using an on-policy Monte Carlo control method with ϵ-greedy as its soft policy. This was done by having an agent learn the fastest way to navigate a racetrack given some restrictions from the environment. Additionally, different configurations of the agent's parameters as well as the environment of the agent were explored in order to get a better intuition of the positives and negatives of Monte Carlo methods.

References

[1] Sutton, Richard S., and Andrew G. Barto. *Reinforcement Learning: an Introduction.* 2nd ed., The MIT Press, 2018.

[2] Gardner, Martin. *Mathematical games* Sci. Amer. 228 (1973) 108.

[3] Barto, Andrew G, et al. "Learning to Act Using Real-Time Dynamic Programming." NeuroImage, Academic Press, 7 Apr. 2000, www.sciencedirect.com/science/article/pii/000437029400011O.