Monte-Carlo Methods for Race Track Reinforcement Learning Problem

Cooper Golemme

March 10, 2025

1 Motivation

The ultimate goal of reinforcement learning is to gather knowledge from the environment to generate an optimal way an agent should act in its environment. In RL, agents follow policies. A *policy* informs the agent on how to act when it finds itself in a given state in the environment. To learn, we want to iterate the agent's policy to eventually converge to some optimal policy. It would be easy to find an optimal policy if all of the environment's dynamics are known. If we know for each state in the environment, which actions yield the best results, we know an optimal policy. A common problem with reinforcement learning is the uncertainty of the environment dynamics. Instead of solving the policy iteration problem by examining the environment dynamics, Monte Carlo methods of reinforcement learning iterates agent policies through learning episodically. Each episode is a sample of a simulated experience for the agent in that environment. Learning from simulated *episodes* is powerful because we do not need to know the complete dynamics of the environment, only simulations of episodes within the environment. Perhaps surprisingly, Monte Carlo methods can be shown to achieve optimal policies, given sufficient episode generations, even without ever knowing the entirety of the environments dynamics. In this paper, we will apply a Monte Carlo Exploring Starts method to a common reinforcement learning problem, test its performance after a training period, and offer improvements to the training process to achieve better results. **Extra Task HERE**

2 Introduction

To begin, we will outline the problem. This problem is *Exercise 5.12: Racetrack* from Chapter 5 of Sutton and Barto's Reinforcement learning textbook [1].

2.1 Problem

The Racetrack problem is as it sounds. The environment is a simplified racetrack like the ones pictured below in Figure 1. The agent is a car that starts at one of the positions on the start line, with the goal of reaching the finish line as fast as possible. At each time step, the car is at one of the descrete set of grid positions. The velocity of the car too is discrete and can be changed once at each time step. The agent change its x and y velocity by 1 or choose to not change it. The agent thus has a (3×3) action space. Below are sample tracks that the book gives.

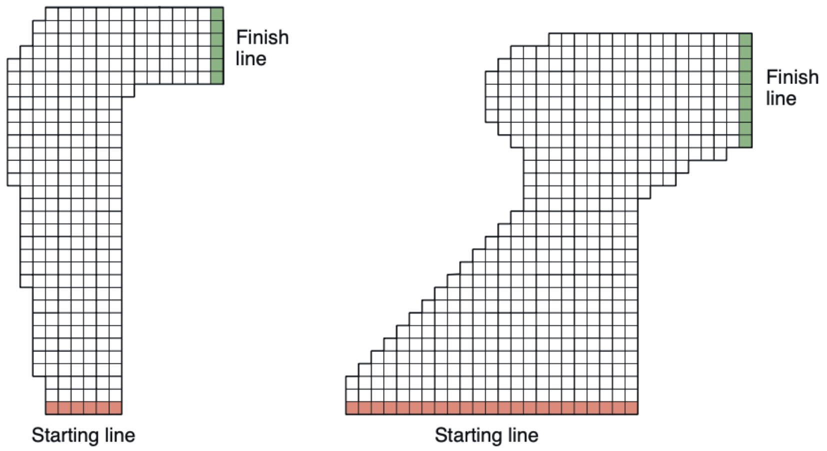


Figure 1: Enter Caption

2.2 Objectives

The goal for this paper is to create and compare Monte Carlo methods to allow the car to learn an optimal path through the racetrack.

3 Experimental Design

In this section, we will detail more precisely the environment, reward structure and methods used in the Racetrack problem.

3.1 Environment

By this point, we have laid out the problem, which is visualized in Figure 1. In order to apply Monte Carlo methods to this problem, we will need to detail more about the environment dynamics as they pretain to Monte Carlo methods.

As beifly mentioned in the motivation section of this paper, in order to learn optimal policies, Monte Carlo methods rely on the concept of episodes. An episode is a simulated sequence of states, actions, and rewards generated from some environment. An episode looks like

$$S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$$

where T is the time step indicating the termination of the episode. The termination of an episode usually happens at the end of the game or when the agent has achieved some desireable task. For this problem, we will consider the time step T to be the time when the agent crosses the finish line. Thus, all episodes we generate from the environment end when the car crosses the finish line.

At every other time step, as mentioned in the previous section, the agent is located at one of the discrete locations on the grid. This corresponds to only part of the state we encode for the agent. Aside from the car's position on the grid, the state also holds the velocity of the car at its position on the grid. We do this to capture more of the environment's dynamics; we might want the agent to act differently in a given state depending on how fast it is moving as well as its position.

As for the actions, at each time step, the agent then can choose to increment is velocity how it would like (which corresponds to an action A_0 for example. Afterwards, the agent receives some reward, which is a signal of the success or failure of the action taken. Before the agent receives its reward for preforming its action, we first check to see if the action taken results in the agent going off the track in the next time step. For example, say the agent was in the right most starting position in the left track of Figure1 and chose to increment is x velocity by 1. In the next time step, the agent would run of the track. If this is the case, we do not end the episode, but restart the agent randomly at one of the starting positions.

At each time step, as long as the agent remains on the track and has not yet crossed the finish line, we provide the agent with -1 reward for every action taken. This hopefully will encourage the agent to finish the race as soon as possible, which is what we hope to teach it.

To summarize, episodes will consist of states – a position on the grid and the car's velocity at that position – , actions – one of 9 possible choices – , and rewards – which are -1 for each time step the agent is on the track.

3.2 Monte Carlo Methods

Now that we have explained the environment and the reward structure, we can discuss how we propose to learn an optimal way of moving through our environment from the start line to the finish. But first, we discuss some important concepts.

3.2.1 Policies

A policy is a mapping from states to actions in the action space of that state. We denote it as $\pi(s) = a \in A(s)$. Basically, a policy tells the agent what action to perform when it finds itself in a given state. The goal is to come up with an optimal policy: one that correctly guides our agent through the environment.

3.2.2 State-Action Value Function

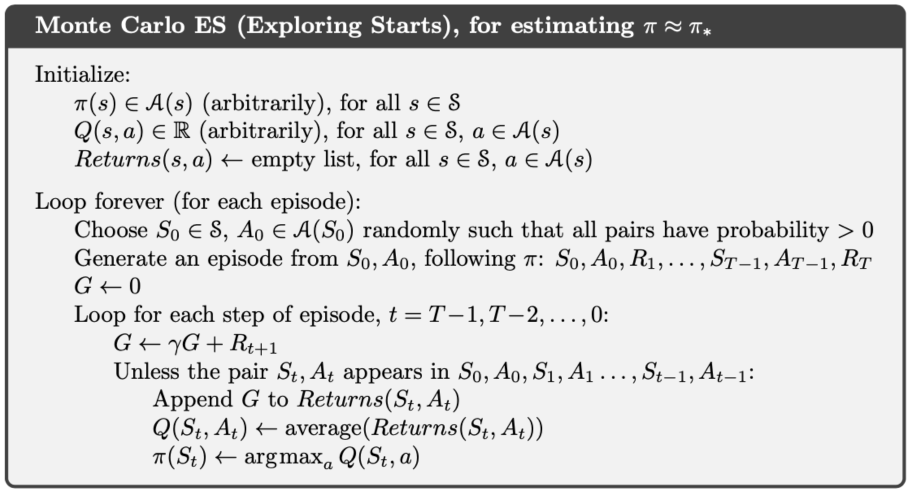
A state-action value function is a mapping from states and actions to a real number: $Q(s, a) \in \mathbb{R}$ for all states and actions. We use this to estimate the reward we expect from performing action, a, in state, s.

3.2.3 Returns

Instead of measuring rewards as we would in other forms of reinforcement learning, we measure returns, denoted G_t. A return is the the reward generated from a single episode.

3.3 Algorithms

For this problem, we used two Monte Carlo algorithms. The first is an exploring starts on-policy algorithm shown below. I choose this because it was easy to implement and demonstrates the capabilities of Monte Carlo methods for this problem.



```
Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:
    $\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
    $Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
    $Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Loop forever (for each episode):
    Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability $> 0$
    Generate an episode from $S_0, A_0$, 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($Returns(S_t, A_t)$)
            $\pi(S_t) \leftarrow \mathrm{argmax}_a\, Q(S_t, a)$
```

Figure 2: Algorithm for Monte Carlo ES. Instead of storing the returns for each state and action pair and averaging for each timestep and episode, in practice, I implemented this using an incremental average update. This has the benefit of not having to store all of the returns and is quicker.

The second algorithm I used was an off-policy Monte Carlo control algorithm. Off-policy algorithms generate episodes following a different policy than the policy they want to evaluate and improve. Off-policy algorithms allow for more continuous exploration while not sacrificing updating the target policy. In theory, this algorithm should perform better in this environment.

```
**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 inner Loop (proceed to next episode)
        $W \leftarrow W \frac{1}{b(A_t|S_t)}$
```

Figure 3: Off-policy MC control algorithm for estimating optimal policy.

3.4 Track Generation

To test our methods, we wanted a systematic way to create tracks similar to the tracks present in the text book [1]. These two example tracks provided have a starting line at the bottom of the gird, and both have a single right turn. Our goal with our generated tracks was to create tracks where the optimal path was generally up and to the right, which simplifies episode generation, and tracks that are about as wide and long as the example tracks.

To do generate the tracks, I employed a random walk method starting from the bottom left corner of the grid and moving up to the top right corner. At each step, we have a choice to move up or to the right to generate the track. The movement up is biased by some factor. For the ones generated below, this factor was 0.45. With a small probability, 0.1 in this case, we add a random movement up and to the right at the same time, to give the track a move curved feel.

Once we have this rough track connecting the start to the finish, we then walk along it, widening it at each step by some fixed, tuneable amount. This allows for the agent to explore more space of the track, which makes the task more interesting.

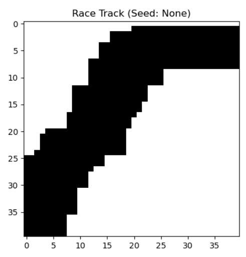


Figure 4: Example random track generated using the method I explained above.

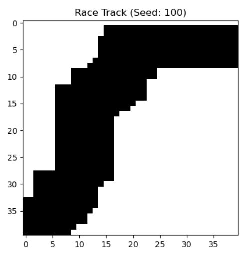


Figure 5: Another example track generated using a seed for reproducibility. This is the track that was used for the results shown below.

4 Results

In this section, I will detail the experiments that were performed, motivate why we choose them, and offer reasons for the results we observe.

4.1 Experiment 1: On-policy Parameters

For this experiment, I wanted to test how changing the various parameters affects the performance of the on-policy MC method. There are a few parameters we have a choice over. Below is a table of those paramters.

Parameter	**Meaning**
γ	Discount factor
α	Step size for average updates
ϵ	Exploration rate
finish reward	Reward for crossing finish line
finish crash	Reward for crashing
finish time-step	Reward at each time-step before finishing

Table 1: Parameters for on-policy Monte Carlo method

To test the training learning rate of the agents, we test the agent for each 10,000 episodes it trains. To test, we place the agent at a random position on the start line and record how long it takes to finish. We then average this result over 10 trials to compute the average steps to finish. We use the average steps to finish to compare model's learning rates to assess the best choice for parameters in this section.

4.1.1 α Selection

From Table 1, we know that α corresponds to the step size for moving average updates. Below we show how the learning rate is affected by choice of α.

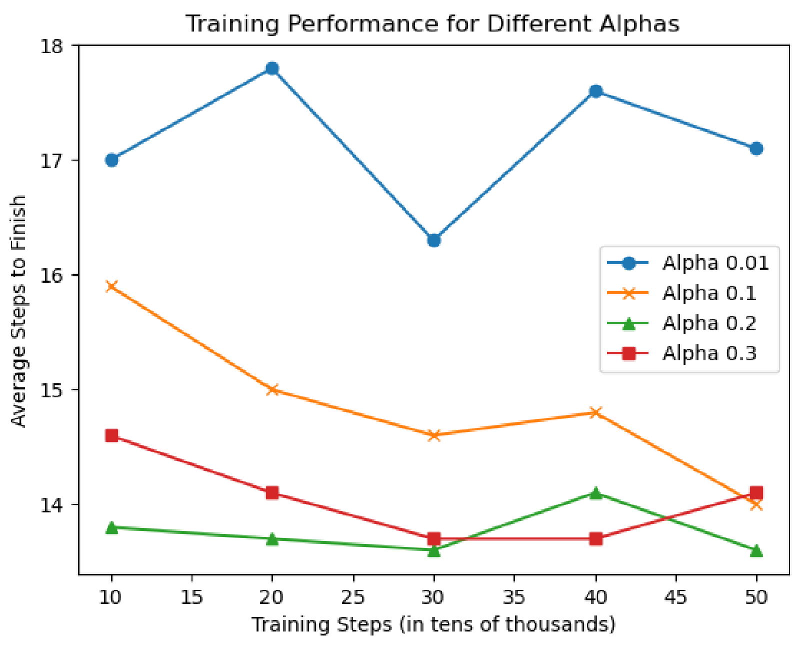


Figure 6: Average steps for the 10 agents randomly placed on the start line to finish the race after a given amount of training steps. We can notice the trend that smaller α's generally perform worse. An optimal value is somewhere around the order of 0.1

4.1.2 γ Selection

From Table 1, we know γ corresponds to the discount factor used when we update each agent's policy. Below we show how the learning rate is affected by choice of γ. A good choice for γ is where $0 \leq \gamma \leq 1$. If γ exceedes 1, then future rewards are weighted more heavily than immediate rewards which can cause several issues. Below is a graph showing the agent's learning rate is affected by choice of γ.

When I tested the agent with $\gamma < 0.85$, I got inconsistent results. Often it would take too long for the agent to generate episodes, indicating that it got stuck in some position on the grid. Other times the testing episodes would fail, indicating again, the agent got stuck in a loop of crashing. I excluded testing these values in the figure below.

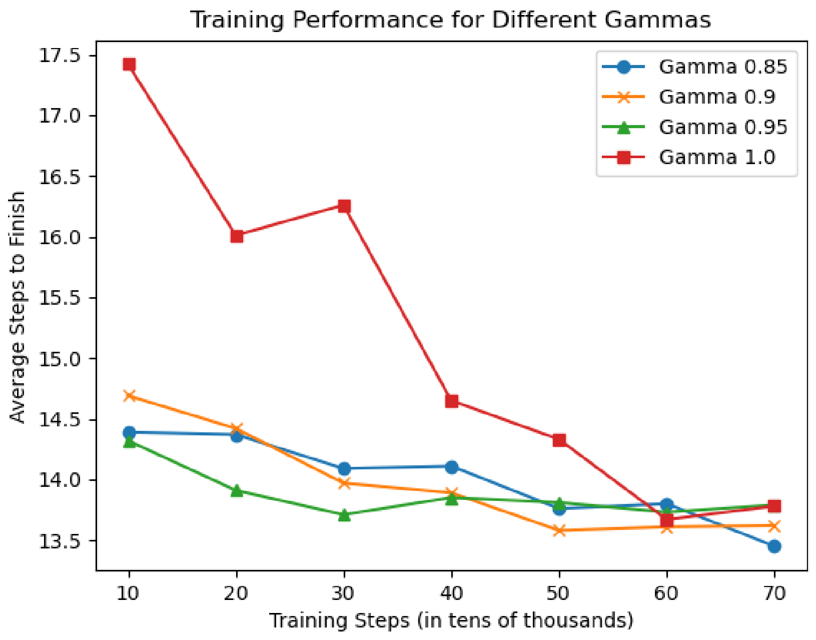


Figure 7: Average steps for 10 agents randomly placed on the start line to finish the race after a given amount of training steps. From the plots, an optimal value for γ is around 0.9.

4.1.3 Finish Reward

The problem in the book does not reward the agent at all for crossing the finish line. For me, this lead to really poor performance and long training times. I noticed that if I rewarded the agent after crossing the finish, these problems went away. Below are the results from experimenting with different finish rewards.

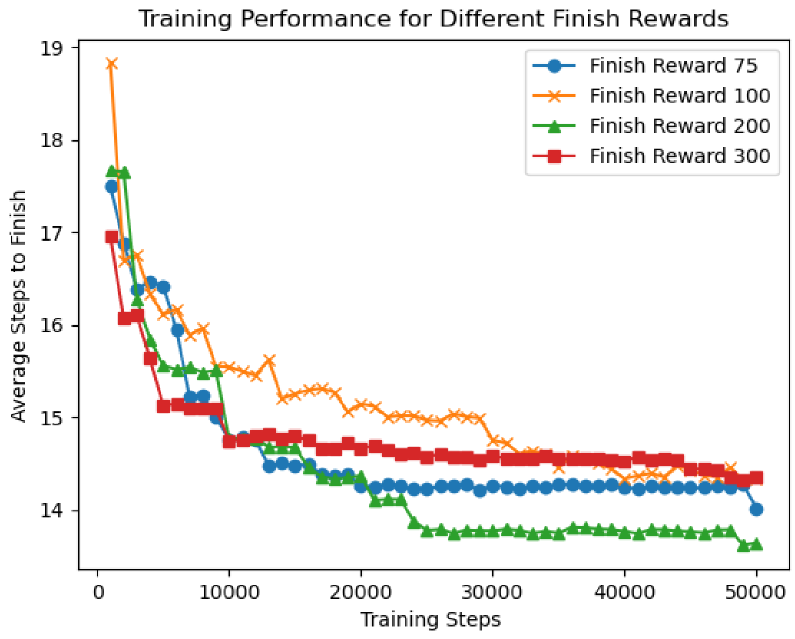


Figure 8: Average steps to finish for 10 agents randomly placed on the start line after a number of training steps. From this figure, we notice that 200 for a finish reward performs the best.

To summarize, in this section, we experimented with an on-policy Monte Carlo control method on the racetrack problem. We showed how to choose parameters to optimize learning rates as measured by the average steps to finish the race. In the next section, we will take this optimized model and compare it to an off-policy Monte Carlo agent.

4.2 Experiment 2: Off-Policy vs. On-Policy

In this section, we will compare the on-policy agent we outlined in the previous section to an off-policy Monte Carlo agent. I described the implementation for this agent in Figure 3. The agent follows an ϵ-greedy policy to move around the environment as it learns an optimal policy.

Below, we show the results from training the off and on policy methods. We can see that the off-policy method converges much quicker to an optimal policy.

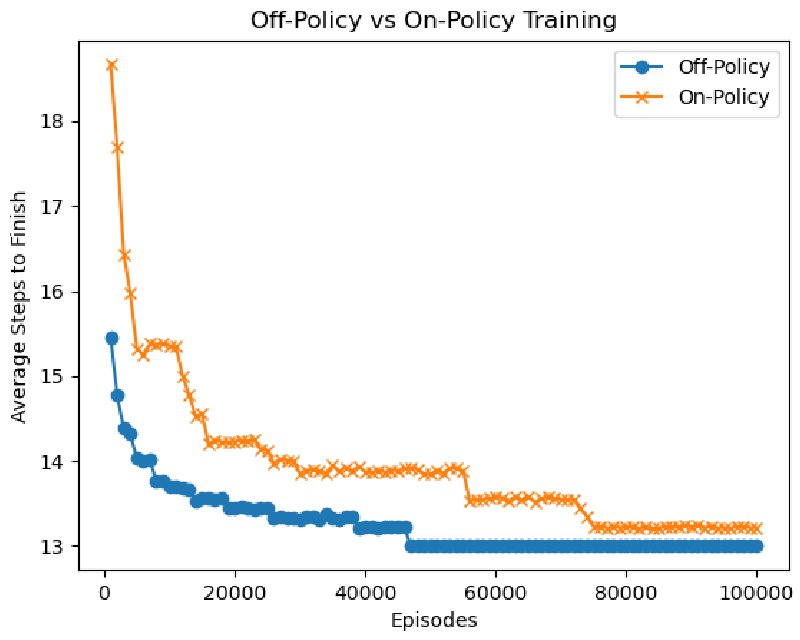


Figure 9: Average steps to finish after training 1000 steps at a time. From this we can see that the off-policy method does much better initially after only a few thousand iterations of training episodes. After a while, the two policies become closer as the converge to the optimal policy for the given racetrack. Overall, the off-policy method out preforms the on-policy method.

A better way to understand the difference between the two models is to see what races look like after a few thousands steps of training and then after they are all trained. Below are some animations that show off-policy and on-policy methods after 1000 training steps, and then again after 100,000 training steps.

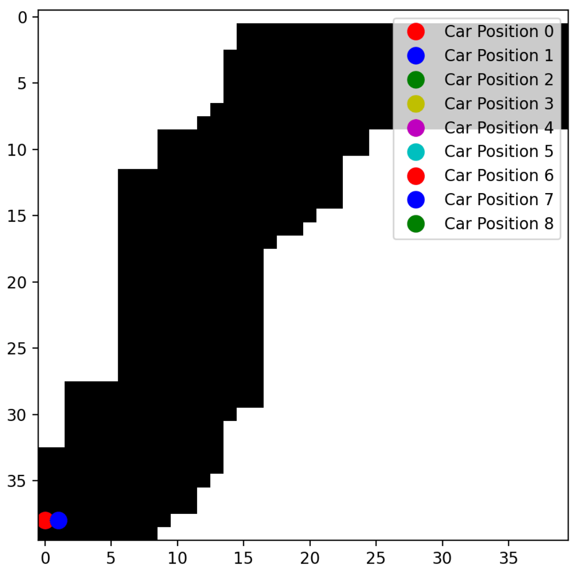


Figure 10: Local gif

References

[1] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, 1995.