Solving the Race-Track Problem Using On-Policy Monte Carlo Control

Zichen Yang

October 2023

1 Introduction

In the realm of Reinforcement Learning (RL), Monte Carlo (MC) methods have proven to be valuable tools for solving complex problems. MC methods leverage the power of random sampling to estimate values and policies, making them particularly effective for situations where the environment is not fully known. In this report, I delve into the MC Control, a reinforcement learning technique that aims to discover the optimal policy for an agent to maximize cumulative rewards.

The MC Control approach is grounded in the idea of learning from experience. Instead of having prior knowledge about the dynamics of the environment, MC Control allows an agent to learn from interactions by collecting valuable episodes of experience. These episodes, which involve sequences of states, actions, and rewards, serve as the agent's basis for updating its policy.

The specific problem I tackle in this report is the "Race-Track Problem," a simplified racing scenario where a car navigates a discrete racetrack grid. The track is actually a right-turn, and the car's objective is to complete the right-turn in the shortest time possible without leaving the track boundaries.

My approach to solving the Race-Track Problem involves implementing a Monte Carlo Control algorithm. Through multiple episodes of interaction with the racetrack environment, the agent will progressively refine its policy. By the end of this report, I aim to showcase the effectiveness of the Monte Carlo Control method in enabling the car to navigate the racetrack efficiently and, ultimately, to provide insights into the power of RL techniques in solving complex, real-world problems.

2 Programming Exercise 5.12: *RaceTrack*

2.1 Exercise Description

Imagine navigating a race car around a turn, aiming to maximize speed without veering off the track's edges. In this simplified scenario, the car occupies discrete

grid positions, each corresponding to a cell on the race track. The car's velocity is also discrete, indicating the number of grid cells it can move horizontally and vertically in each time step. The car's movements are controlled by actions that adjust its velocity components—these actions can increment by +1, -1, or remain unchanged, allowing for nine distinct action possibilities (3x3). Both velocity components are constrained to be non-negative and less than 5, and they can't be both zero except when at the starting line.

Each episode of this challenge begins from randomly selected initial states with both velocity components set to zero. Episodes conclude when the car successfully crosses the finish line. During the race, the car earns a reward of -1 for each step taken until the car successfully crosses the finish line as the episode concludes. However, if the car collides with the track boundary, it's promptly relocated to a random position on the starting line with zero velocity, and the episode continues.

Before determining the car's new position at each time step, we scrutinize whether the projected path of the car intersects with the track boundary. If the intersection occurs at the finish line, the episode concludes. However, if the intersection happens at any other point along the track boundary, the car is deemed to have struck the track's edge and will be sent back to the starting line. The problem also introduces a layer of complexity: there is a 10% chance at each time step that both velocity increments will be zero, regardless of the intended increments.

2.2 Methodology

To solve this problem, I employ an on-policy every-visit MC control algorithm. The solution is composed by four components: track construction, environment, agent and the driver program.

2.2.1 Track Construction (track.py)

The race track in this experiment should contain a right turn. To simplify the construction process, I hard-coded a race track as shown in the Figure 5.5 in the textbook. It's a 32x32 grid where the start line is at the bottom and the finish line is at the up-right corner. All cells that are on the track, which means the positions that are valid for the car to be on, are marked as 0; all cells that are on the starting line are marked as 1; all cells that are on the finished line are marked as 2; all cells that are out of track (gravels) are marked as -1. To build the track, import this *track* module and call *build_track*(), and it will return a 2-D array with all cells marked appropriately. Figure 1 shows the track built for this experiment.

2.2.2 Environment (env.py)

The *Environment* class will be responsible to update the car's position and check all the requirements such as the car's speed limit or if the car goes

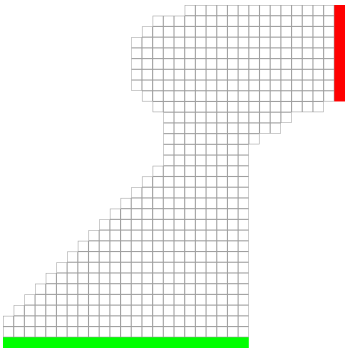

Figure 1: The race track

across the boundaries. Another functionality of environment is to visualize the training process so that I can check if the car is being training properly. The *Environment* class has the following attributes:

- *track*: the track in which the car will be navigated. It is a 2-D array with cells marked by -1, 0, 1, 2, depending on their location.
- *start_set*: the set that contains the coordinates of all cells that are on the starting line.
- *gravel_set*: the set that contains the coordinates of all cells that are out of track.
- *screen*: the screen parameter for the pygame to render. It is only relevant to the visualization of the training but not relevant to the solution.
- *clock*: the clock parameter for the pygame to maintain a constant frame rate. It is only relevant to the visualization of the training but not relevant to the solution.

The *Environment* class has the following methods:

- *reset*(): this function will reset the car by returning a randomly-chosen coordinate from the *start_set* and the speed with both components being 0.
- *check_finish*(*position*): given a position (i', j'), this function will check if the given position has passed the finish line. Suppose the finish line consists of cells (i, j), where $r_{FinishStart} \leq i \leq r_{FinishEnd}$ and $j = c_{FinishStart}$, if $i' \in [r_{FinishStart}, r_{FinishEnd}]$ and $j' \geq c_{FinishStart}$, then the given position is considered to be crossed the finish line and thus return Ture; else, return False.
- *check_crash*(*old_position*, *new_position*): given the old and new position, this function will check if the car intersects the track's boundaries. Firstly,

this function checks if the new positions is out of the 32x32 grid or is in the gravel set. If so, return True promptly; else, check if the projected path from *old_position* to *new_position* intersects with the boundaries, and return True if it does, else return False.

- *take_action*(*position*, *speed*, *action*): given the state and action of the car, the *Environment* instance will give the car a new state and reward, and also return a flag indicating whether the episode is end. The pseudo code is shown below:

Algorithm 1: Pseudocode for *take_action* method

$reward \leftarrow -1$;
$\epsilon \leftarrow p$ where p is randomly chosen from $[0, 1]$;
if $\epsilon <= 0.1$ **then**
    speed remains unchanged;
else
    $new\_speed \leftarrow speed + action$, where both components $\in [0, 5)$ and cannot be 0 at the same time.
$new\_position \leftarrow position + new\_speed$;
$terminated \leftarrow False$;
if $check\_finish(new\_position) == True$ **then**
    $terminated \leftarrow True$
else if $check\_crash(new\_position) == True$ **then**
    $new\_position \leftarrow s$, where $s \in start\_set$;
    $new\_speed \leftarrow (0, 0)$
return *reward*, *terminated*, *new_position*, *new_speed*

There are two more methods, *draw_grid* and *display*, that are used to visualize the training process. Since they are not relevant to the scope of this experiment, they won't be expanded here.

2.2.3 Agent (agent.py)

The *Agent* class is responsible for choosing the action given the current state and policy. It also needs to learn the optimal policy based on the action and the reward associated with it. The *Agent* class has the following attributes:

- *epsilon*: the *epsilon* for *epsilon*-soft policy, with default value of 0.1.
- *gamma*: the discount rate for future rewards, with default value of 0.9.
- *speed*: the speed of the car, represented by $[v_{horizontal}, v_{vertical}]$.
- *position*: the position of the car, represented by (r, c).

- *actions*: the dictionary of actions the car can choose from. Each action is represented by $[v_{horizontal\_increment}, v_{vertical\_increment}]$.
- *num_actions*: the number of actions the car can choose from.
- Q: the action value for each action in each position. It's initialized as a 3-D array with all entries being 0.
- N: the number of how many times an action is chosen at each position. It's initialized as a 3-D array with all entries being 0.

The *Agent* class has the following methods that help it learn the optimal policy:

- $soft\_policy(Qs)$: this function will return a policy which gives the probability of choosing each action based on the action values, Qs, provided. The probabilities of actions will be updated by the following rules: Let A^* be the optimal action at state S_t, for all $a \in A(S_t)$,

$$\pi(a|S_t) \leftarrow 1 - \epsilon - \epsilon/|A(S_t)| \qquad \text{if } a = A^* \tag{1}$$
$$\pi(a|S_t) \leftarrow \epsilon/|A(S_t)| \qquad \text{if } a \neq A^* \tag{2}$$

- $on\_policy\_mc\_control(env,\ num\_episodes)$: this function is where the on-policy every-visit MC control algorithm is implemented, which helps the agent to learn the optimal policy. The every-visit method means that the action value of a state take the average of The algorithm is described as Algorithm 2 on the next page. The update of action value is implemented incrementally as shown on line 17 so that there's no need to maintain an array to record all rewards received at each state.

2.2.4 Driver Program (racetrack.ipynb)

The driver program is fairly simple: all it needs to do is to build the track, create an *Environment* and an *Agent* instance, and then train the *Agent* by calling its method. The pseudocode is shown as Algorithm 3 on the next page:

2.2.5 Results

Here, I train agents with different epsilons. After training for 1000 episodes, the agents are able to finish the track really smoothly. Figure 2 shows the length of each episode during training of each agent, the shorter the length, the quicker an episode concludes. We can see that the policy becomes stable after 100 episodes and barely has any spike after 1000 episodes. All agents converge approximately at the same time, and the agent with epsilon = 0.1 seems to have less fluctuation on the learning curve so that it's chosen as the best one among the four agents. Figure 3 shows the rewards of 1000 randomly generated episode by this agent, and the average rewards over 1000 episodes is -14.7.

Algorithm 2: Pseudocode for *on_policy_mc_control* method

```
position, speed ← env.reset() ;
episode_state ← [ ] ;
while True do
    policy ← soft_policy(Q[position]);
    action ← chosen based on policy;
    if position is in env.start_set and action results in [0,0] speed then
        action ← [0, 1]
    reward, terminated, new_position, new_speed ←
      env.take_action(position, speed, action);
    append (state, action, reward) onto episode_states;
    state ← new_state;
    speed ← new_speed;
    if terminated == True then
        break out of the loop;
G ← 0 ;
for t = T − 1 to 0 do
    pos_t, a_t, r_t ← episode_states[t];
    G ← gamma ∗ G + r;
    N[pos_t][a_t] ← N[pos_t][a_t] + 1;
    Q[pos_t][a_t] ← Q[pos_t][a_t] + 1/N[pos_t][a_t] ∗ (G − Q[pos_t][a_t])
```

Algorithm 3: Pseudocode for the driver-program

```
track ← build_track();
env ← Environment(track);
agent ← Agent();
num_episode ← somebiginteger;
agent.on_policy_mc_control(env, num_episodes)
```

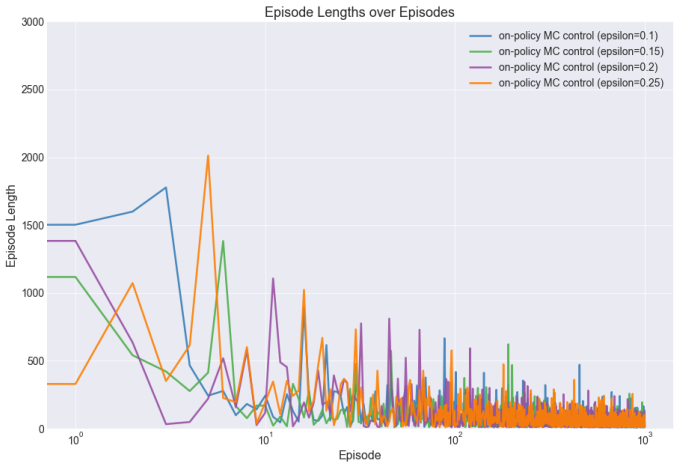

Figure 2: Episode lengths throughout on-policy MC control

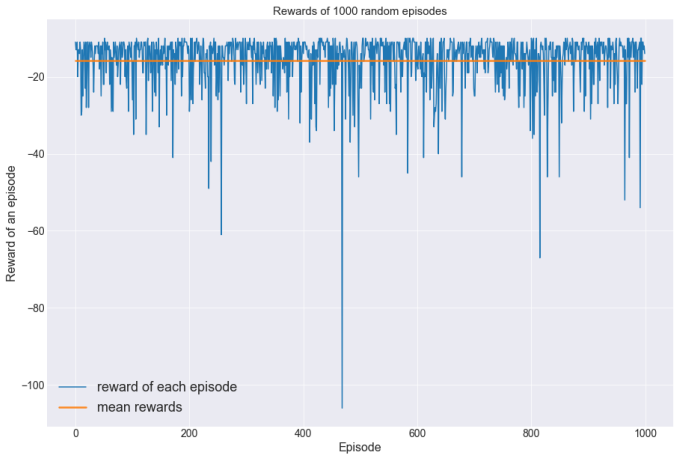

Figure 3: Rewards of 1000 random episodes generated using the final policy produced by on-policy MC control (epsilon=0.1)

3 Addition Question: on-policy vs off-policy

Other than using on-policy MC control, an off-policy MC control can also be used to solve this problem. However, which method is better? To implement the off-policy every-visit MC control, some new attributes should be added to

the *Agent* class:

- C: is a 3-D array that stores the cumulative weights for each action at each position. All entries are initialized to 0.
- *target_policy*: the greedy policy learnt from the off-policy MC control. It's initialize arbitrarily.

In order to reuse as much code from on-policy method as possible, I directly modify the *on_policy_mc_control* method to be:

- $mc\_control(env,\ num\_episodes,\ on\_policy = True)$: this modified function takes in one more argument *on_policy* which is the flag indicating whether it's performing on-policy or off-policy MC control. For off-policy MC control, the behavior policy is also a ϵ-soft policy so that either on/off-policy MC control can share the same code of generating episodes. The new algorithm is described in Algorithm 4.

Algorithm 4: Pseudocode for *mc_control* method

```
1  position, speed ← env.reset() ;
2  episode_state ← [ ] ;
3  Algorithm 2 line 3 to 13;
4  if on_policy == True then
5  |  Update N and Q according to Algorithm 2 line 15 to 19;
6  else
7  |  W ← 1;
8  |  for t = T − 1 to 0 do
9  |  |  pos_t, a_t, r_t ← episode_states[t];
10 |  |  G ← gamma ∗ G + r_t;
11 |  |  C[pos_t][a_t] ← C[pos_t][a_t] + W;
12 |  |  Q[pos_t][a_t] ← Q[pos_t][a_t] + 1/N[pos_t][a_t] ∗ (G − Q[pos_t][a_t]);
13 |  |  π(pos_t) ← argmax(Q[pos_t]);
14 |  |  if π(pos_t) ≠ a_t then
15 |  |  |  break;
16 |  |  W ← W ∗ 1/policy[a_t];
```

To conduct a controlled experiment, I also train the agent with off-policy every-visit MC control 1000 episodes with various epsilon value. The learning curve of off-policy every-visit MC control with epsilon = 0.1 is plotted in Figure 4 along with the on-policy every-visit MC control with epsilon = 0.1. It can be seen that the episode lengths for off-policy method at the end of the training are higher than on-policy method. Also, the mean reward of 1000 randomly generated episodes using the target policy of off-policy method (shown in Figure 5 on the last page) is much larger than policy produced by on-policy method, which

means the target policy hasn't converged enough. One of the reason could be that the off-policy method tend to have higher variance in their updates due to the importance sampling correction, which s can lead to slower convergence because the updates have a wider range. Also, since off-policy method needs to learn from both the behavior policy and the target policy, while on-policy method only need to learn a single policy.

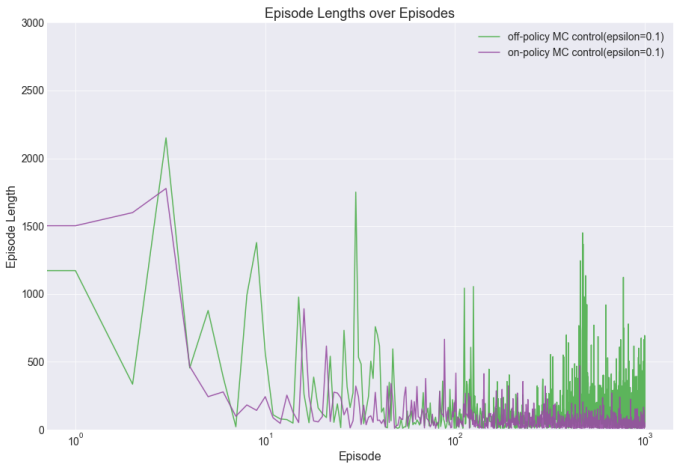

Figure 4: Comparison between episode lengths of on/off-policy method (epsilon=0.1)

4 Conclusion

The racetrack exercise is successfully solved by both on-policy and off-policy MC control, and on-policy method seems to have better performance. Some sample trajectories generated by polices produced by both methods are shown in Figure 6 on the next page.

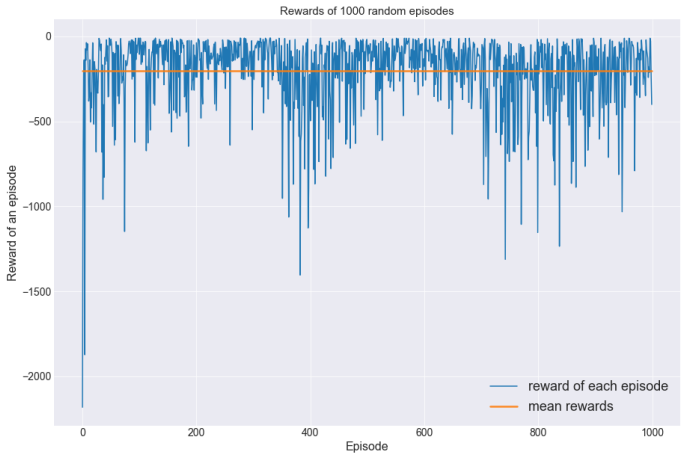

Figure 5: Rewards of 1000 random episodes generated using the target policy produced by off-policy MC control (epsilon=0.1)

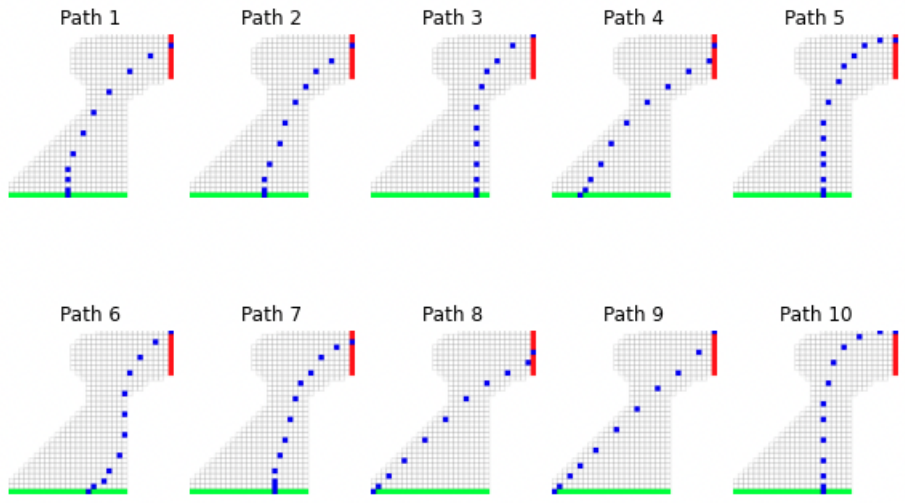

Figure 6: Sample paths made using policy produced by on/off-policy MC control