COMP 138 RL: Racetrack Problem

Tung Pham

October 18, 2024

1 Introduction

For the gamers that love speed, everyone would have heard of the name Need for Speed, a game developed by EA. It used to be my childhood favorite back in the 2000s. On the old machine with Window XP, I used to enjoy this game to the fullest. I've played the Need for Speed: Pursuit, which is the more recent version of the game.

However, I've always wondering how to efficiently taking a turn as I was always get passed by or get throw off track when it comes to these portion of the track is extremely difficult and you can easily get slowed down and lose the match. This prone me to thinking, with the materials I learned in CS138, can I essentially applies any of the techniques I learn so far into figure out the most efficient way to take a turn?

In this experiment, I'll use 2 approaches to figure this out, an Off-policy Monte Carlo Control and Double Q-Learning Temporal Difference method. Both methods allow the driver to learn from the experience of the environment by having a learning strategy that will be improve over the training courses. Eventually, the learning approaches, or policy denoted π, can leads to the agent figure out the optimal way to turn and able to achieve the highest score (R_t). After each attempt, episode, the driver will have a better understanding of the environment using the learning approach and therefore getting better score result.

2 Problem

To make the problem a little less complex, I'll use a 2D matrix as my representation of the racetrack. We'll also ignore most of the physics mechanism as listed here:

- Centripetal force
- Inertia
- We won't be using any metrics like kilometer but instead will be relying on the matrix unit.

- We'll also not using seconds as the time step but instead a time step is relying on the clock speed of the computer that used to simulate.
- No air friction.

We'll also assume some idealistic scenario:

- The racetrack doesn't have any bumps and is a smooth curve
- Acceleration happens in integer values
- The maximum velocity the driver can get is 5 matrix unit.
- When the car move to the next location on the grid, it will have a teleportation behavior and jumping straight to the location, but if the location is out of bound, it will fall off.
- When the car falls off, it will be teleported back to a random location on the starting line and retrying the race.
- If the driver go too fast and the next location he arrive is not the finish line, he'll be out of bound and fall off the racing track.

At each time step, the driver can decided to hit the brake or hit the gas which will decelerate or accelerate the car by 1 matrix unit respectively. To make things a little challenging to the driver, at 10% of the time the brake and gas pedal won't work because of how smooth the racetrack it and cause the car to slip and no friction was applies which cause no acceleration or deceleration can occurs.

The score of the action that the driver takes will be accumulated through out each attempt and the average over all the attempts will be measured.

In this experiment, we'll be doing 2000 attempts on 2 different tracks using each method. We're also conducting some initial tests that was used on 2 other smaller size grid which can be seen in Figure 2 and Figure 3.

3 Methodology

3.1 Environment

To represent the racetrack, we'll be using a 2D grid. However, as you'll see later on, because the track is upside down in the code matrix, we'll have to flip the matrix so that the environment is correctly represent that the driver or the car is going from the bottom up.

To represent the state, since there are 4 values that is in effect in this problem which is the position on the grid and the velocity. Since this is a grid, we'll need the coordinate which is x value and the y value. Also for the velocity, since we'll be moving in both x and y direction, we'll need to track the 2 velocity as well. Our state eventually becomes a tuple of (x coordinate, y coordinate, vx, vy).

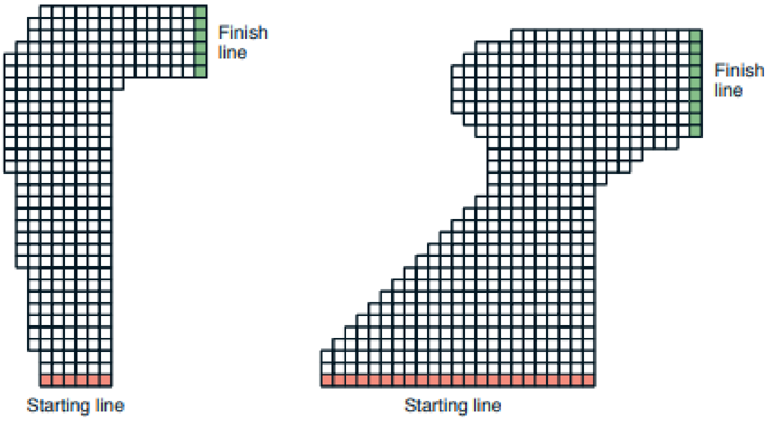


Figure 1: The large race track that was used for experiment

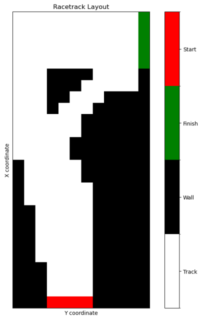


Figure 2: Smaller test racetrack 1

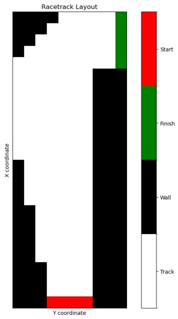


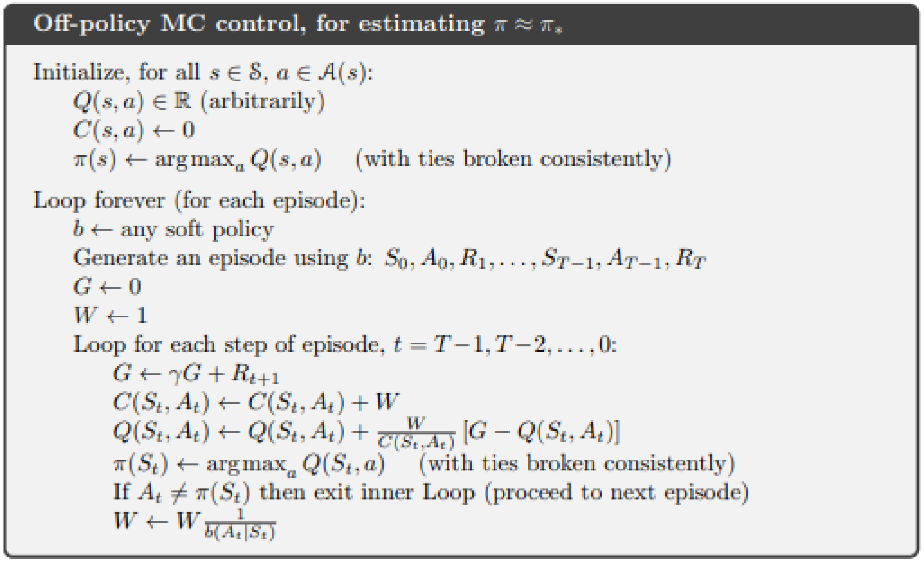
Figure 3: The second smaller test racetrack

Then at each step taken all the criterias mentioned in the problem statement will be in effects to determine the out-of-bound and the destination. It will also handles teleport the driver or the car back to the starting position.

3.2 Off-policy Monte Carlo Control

3.2.1 Setup

In Chapter 5.8 of Sutton and Barton, we were introduced with an off-policy Monte Carlo Control algorithm which is described in Figure 4.



```
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 4: The Off-policy Monte Carlo Control algorithm in Sutton and Barton

The algorithm starts out with initializing Q with arbitrarily non negative floating valule. In our case, we're initializing Q with a uniform distribution between 0 and 1 for all actions at state 0. This encourages the driver to explore more possbilities in the unknown environment and fail alot. However, with these failures, the drivers will learn and do a better job next time.

Then our target policy, $\pi(s)$, is a greedy algorithm that takes the action that yields the most value or reward. Which can be describe as:

$$\pi(s) = argmax_a Q(S_t, a)$$

This algorithm separates the behavior policy from the policy that we want to maximize which is the target policy and behavior policy can be any soft algorithm. In our case, we're using ϵ-soft algorithm for this policy which is described as below:

$$b \leftarrow argmax_a b(a|S_t)$$

where

$$b(a|S_t) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|A(S_t)| & \text{if } a = A^* \\ \varepsilon/|A(S_t)| & \text{if } a \neq A^* \end{cases}$$

To choose the value of ϵ, since we want the driver to exploit the current route more but also have a little of exploration, we don't want ϵ to be too high and therefore decided to go with 0.1. Also, since we're adding a little bit of a challenge where at 10% chance the car slipped and all acceleration or decelleration will not work, according to the rule of $\epsilon - soft$, we can update the behavior policy further:

$$b(a|S_t) \leftarrow \begin{cases} 0.1 + b(a|S_t) & \text{if } a \neq A_0 \\ 0.9 * b(a|S_t) & \text{if } a = A_0 \end{cases}$$

where A_0 indicates the action of not accelerating or decellerating in any direction.

Since this is a Monte Carlo variant algorithm, all the updates to the policy must occurs at the end of each episode. Therefore, we'll have to initially, generate all the possible steps that the drive will be taking. This means that the driver has to blindly do a first attempt and will taking T steps to complete an episode.

At each step, the driver will take an action based on the bahavior policy and yields a reward R for that action and we define R as follows:

- The car reached the goal: 100
- The car hit the wall or fly out of bound of the race track: -5
- The car take a step: -1

3.2.2 Testing

To test, we're running the algorithm on a smaller track and print out the velocity and the route that the driver took which is illustrated in Figure 5 and Figure 6 below.

Looking at the 2 figures, we can see that the step function behave correctly as whenenever the next velocity of the action resulted in the car out of the racetrack, we can see a straight line back to the starting point. And when the stopping point is at the finish line, we can see that the program is successfully end the episode.

3.2.3 Experiment

To do the experiment, we're then use a much larger racetrack and view the reward change over the number of episodes. Figure 7 illustrate the average rewards over the number of episodes as the number of episode increases on the first track and Figure 8 is for the second track.

We can see that the reward is displaying a log-like curve which gradually increase as the number of episodes increase meaning that given enough time, the algorith can converges to the optimal value.

The driver at first perform extremely bad as we can see that the reward is going all the way down to -14000 for track 1 and more than -500 for track 2. However, since this is the first stage, the driver is exploring the unknown environment and make a lot of mistakes. As the number of episodes increase, the reward obtained by the driver is also increases and that cause our graph to curve up but in the long run, we're exploiting more and less exploring, the reward began to converges to a single value which is the optimal reward that we can get.

3.3 Off-policy TD method with Double Q-Learning

3.3.1 Setup

In Chapter 6 of Sutton and Barton, we were introduced with an off-policy TD method algorithm with Double Q-Learning where we have 2 different Q tables that we're keeping track with 50-50 chance to update one than the other Q table. Figure 9 illustrate this algorithm.

Since the algorithm also starts out with initializing both Qs to an arbitrary number, we're again use uniform distribution.

To more weight on the future value, we're using a larger gamma value than what we used in Monte Carlo which is 0.9 but because we also want the rewards distribution to be smoother, we're using a small value for α.

For our greedy action selection, we're using 0.1 for our ϵ value as we want the driver to explore options once in a while but not too often which could steer the driver away from the optimal route.

3.3.2 Test

We're then testing our implementation on the smaller test set as we can see in Figure 10 and 11.

We can see that the driver was able to reach the destination despite the overwhelming amount of steps taken to reach it. This is because we're running on the initial episode and without any training, the driver doesn't learn and can't figure out what's the best solution. Then we'll see what happens if we train the driver with this algorithm.

3.3.3 Experiment

Figure 12 and Figure 13 is the plot of cummulative average reward

We can observe that both graphs demonstrated a bad average reward at first but the performance increase in log-like behavior. However, the convergences point is rather low at -500 which indicates that either the algorithm needs more time to observe the convergence.

4 Discussion

When comparing the figures of both Monte Carlo and TD with Q-learning algorithm, we can see that both approaches yield a desireable results. However, the convergence point of Q-Learning is much lower than Monte Carlo indicates that the algorithm needs more fine tuning of its parameters or more episodes in order to achieve the same or better result. A lower convergent point also suggest that TD approach is yielding overall less rewards than Monte-Carlo and is not reaching the optimality as good as Monte-Carlo.

When comparing the performance of both algorithm, TD took a lot faster to train as it only needs a single loop to both generate the episodes and update the policy while Monte Carlo is esssentially less performant where it took significant amount of time to calculate.

Another note to make is that for Q-learning with TD, we can see that the rewards trends converge later than Monte Carlo and has more upwards slope than Monte Carlo. This suggests that given enough time, Q-Learning can still yield the reward as competitive as Monte-Carlo produce.

The agent was also reach positive result meaning that it did not take more than 100 steps or fall out of the racetrack more than 20 times. Some instance even taking less than 20 steps to complete. This results was observed by printing out the results rewards as seen in the code.

5 Conclusion

Using both off-policy Monte Carlo control method and Double Q-Learning Temporal Difference method yield us the desireable result where both reward increasing as time goes by. However, performance wise, Monte Carlo even though achieves better results, is quite resource consuming while Q-Learning with TD method yield less desired result but is extremely more performant in both speed and resource consumption.

In the end, I was able to get the best optimal direction to drift and take the turn with the least amount of time. This way, I can practice (obviously I can't do it immediately if I know the trick) my craft so that I can improve my race winning rates for the coming racing games.

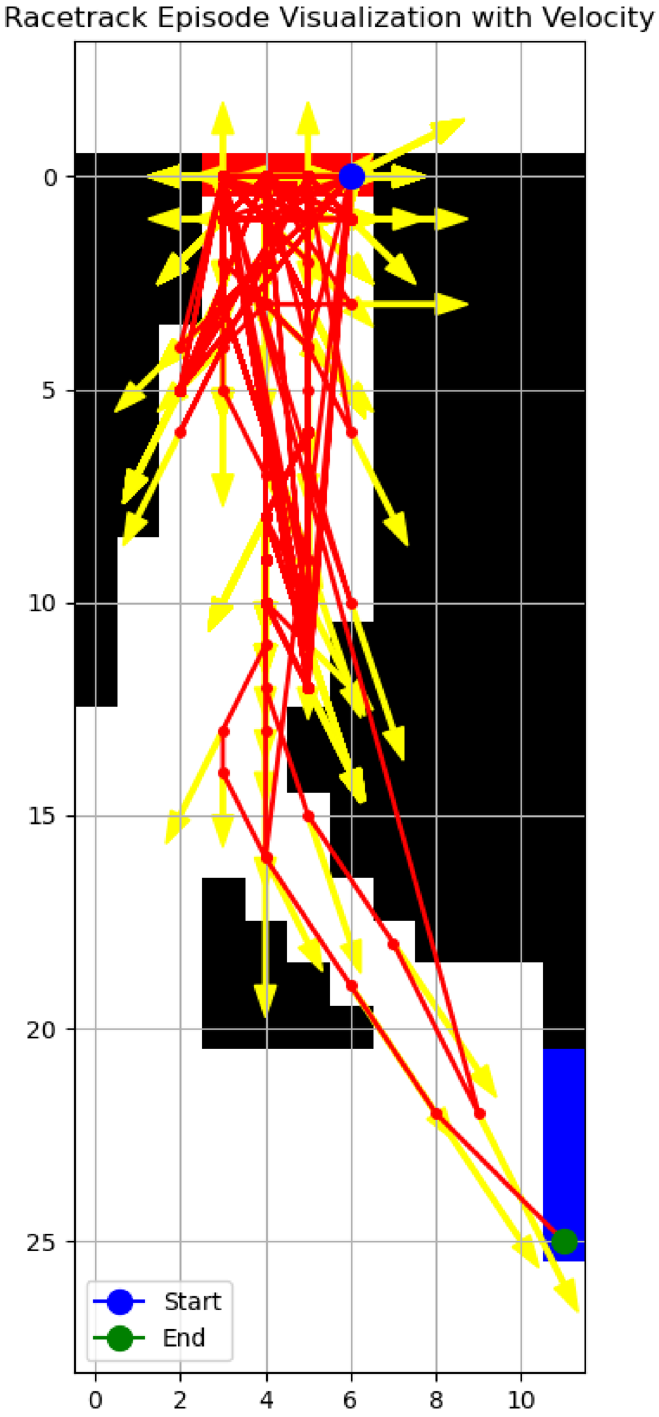


Figure 5: State transition of Off-policy Monte-Carlo control when run on track 1

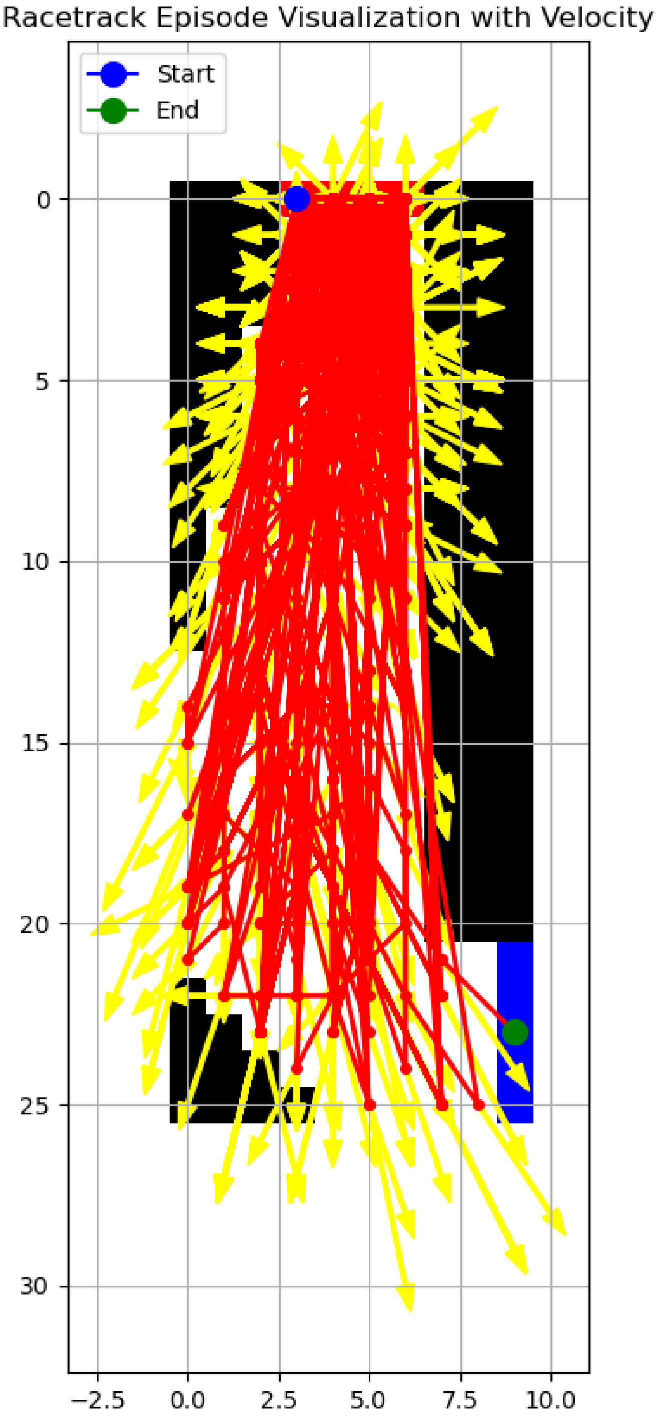


Figure 6: State transition of Off-policy Monte-Carlo control when run on track 2

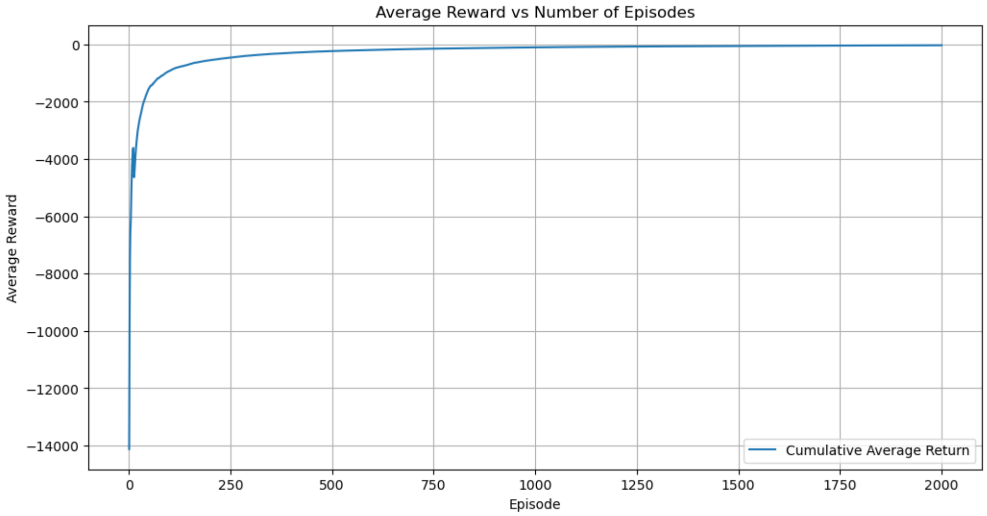


Figure 7: Average reward over the episodes vs the number of episodes used for track 1.

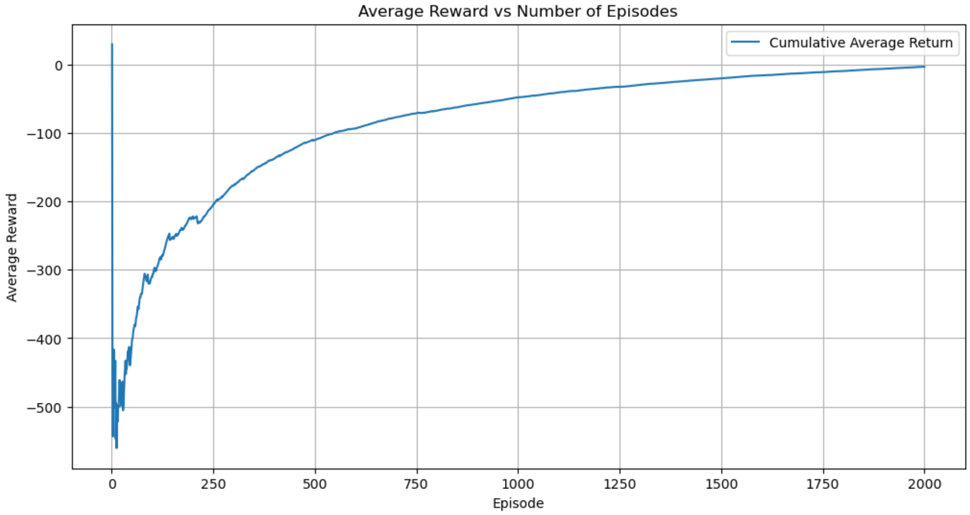
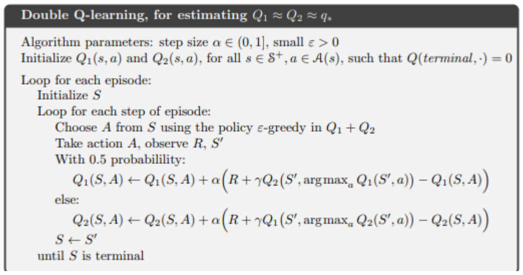


Figure 8: Average reward over the episodes vs the number of episodes used for track 2.



```
Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, such that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A$ from $S$ using the policy $\varepsilon$-greedy in $Q_1 + Q_2$
        Take action $A$, observe $R$, $S'$
        With 0.5 probabilility:
            $Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \arg\max_a Q_1(S', a)\big) - Q_1(S, A)\Big)$
        else:
            $Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \arg\max_a Q_2(S', a)\big) - Q_2(S, A)\Big)$
        $S \leftarrow S'$
    until $S$ is terminal
```

Figure 9: The Double Q-learning TD method

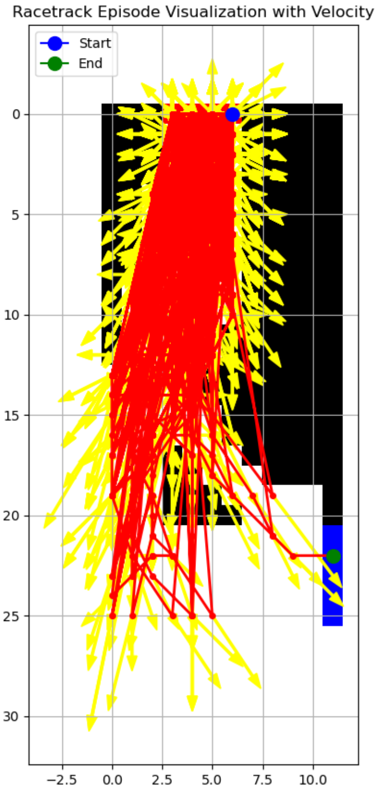


Figure 10: State transition of Double Q-learning TD method runs for a single episode on a small test track 1

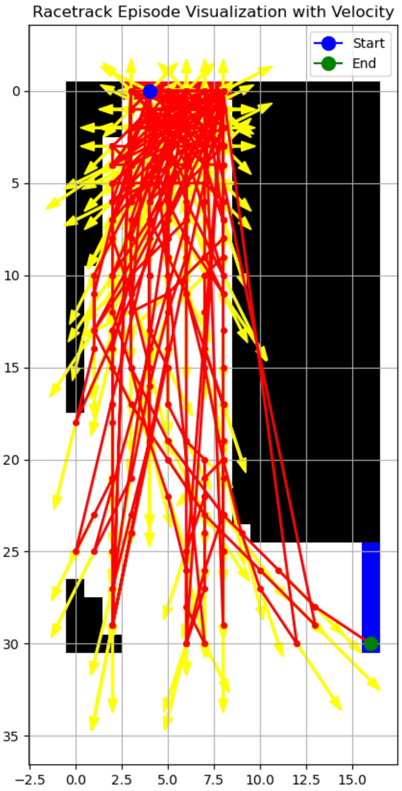


Figure 11: State transition at each time step of Double Q-learning TD method runs for a single episode on a small test track 2

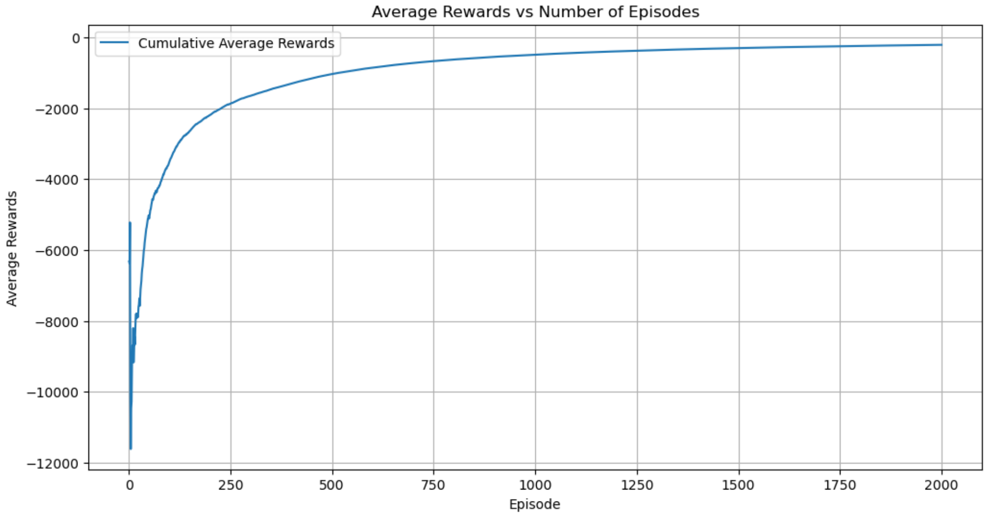


Figure 12: The Average rewards over the number of episode versus the number of episode for Double Q-Learning TD method on track 1.

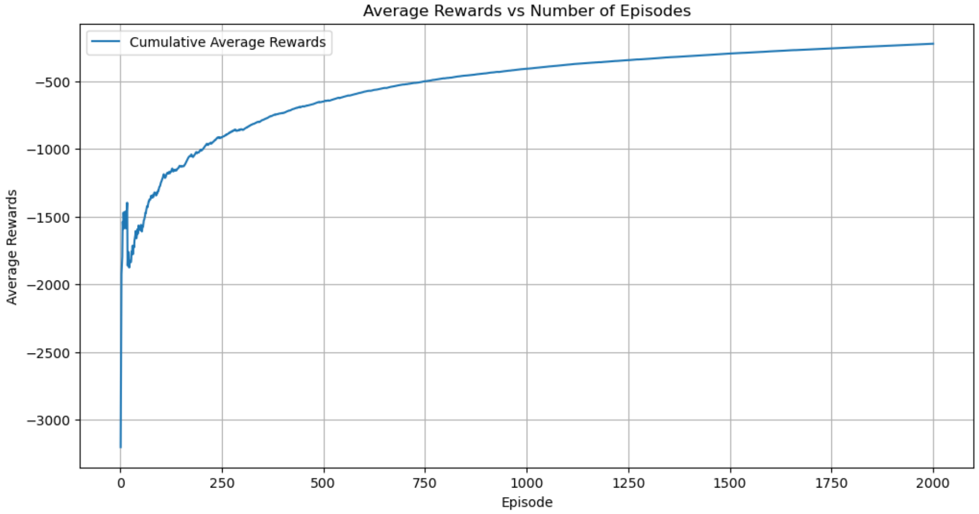


Figure 13: The Average rewards over the number of episode versus the number of episode for Double Q-Learning TD method on track 2.