Date / /

Ex - 5

1) Off - Policy Method.

a) $$V_n = \frac{\sum_{k=1}^{n-1} W_k G_k}{\sum_{k=1}^{n-1} W_k} \quad n \geq 2 \qquad \text{—(5.7)}$$

$$V_{n+1} = V_n + \frac{W_n}{C_n}\left[G_n - V_n\right] \quad n \geq 1 \qquad \text{—(5.8)}$$

The weighted update rule eq^n (5.8) from (5.7)

$$V_{n+1} = \frac{\sum_{k=1}^{n} W_k G_k}{\sum_{k=1}^{n} W_k} \quad n \geq 1$$

$$= \frac{\sum_{k=1}^{n-1} W_k G_k}{\sum_{k=1}^{n} W_k} + \frac{W_n G_n}{\sum_{k=1}^{n} W_k}$$

$$= \frac{\sum_{k=1}^{n-1} W_k}{\sum_{k=1}^{n-1} W_k} \times \frac{\sum_{k=1}^{n-1} W_k G_k}{\sum_{k=1}^{n} W_k} + \frac{W_n G_n}{\sum_{k=1}^{n} W_k}$$

$$= V_n - (G_n - V_n) \times \frac{W_n}{\sum_{k=1}^{n} W_n = C_n}$$

$$\boxed{V_{n+1} = V_n + \frac{W_n}{C_n} \times \left[G_n - V_n\right]}$$ //.

b) As it only allowed to change W if $A_t = \pi(S_t)$.
And due to π is deterministic, we are safe to say $\pi(A_t \mid S_t) = 1$ during the update of W.

DOMS Page No.
Date / /

2) Temporal difference Vs Monte carlo :

a) In the given example, if the starting point changes, but the route to the highway remains unchanged, one must relearn only the initial segment of the journey to the highway.

[When adapting to a new starting point in a familiar driving route, TD learning efficiently updates the initial part of the route without altering the known segments. Conversely, MC learning requires revisiting the entire route before updating. Hence, TD learing is more efficient for such incremental adjustments].

b) Some of the example such as maze or puzzle, in the environment the agent rewards only when they reach the goal with 1 point and every other step with 0 rewards. In this case Monte-Carlo will perform better than the TD. Since ~~TD~~ MC will update every state along the trajectory but TD needs to run for a long time and episodes to have good results.

DOMS Page No.
Date / /

3) Random walk task

a) The first episode terminated on state A. All state value is initialized to be 0.5. And the reward is all 0 except for state E for 1, $\gamma = 1$

Taking $\alpha = 0.1$, we get.

$$RMS(A) = \alpha [R + \gamma V_T - V(A)]$$
$$= 0.1 [0 + 0 - 0.5]$$
$$= -0.05$$

$$RMS(K) = \alpha [R + \gamma V_{K+1} - V(K)]$$
$$= 0.1 [0 + 1(0.5) - 0.5]$$
$$= 0.$$

So the start changed by -0.05, and other remain unchanged

b) The results shown in the graph of random walk example are affected by α, and the results finally converge to a relative stable value that won't change a lot anymore. (TD performs best when $\alpha = 0.05$ & MC performs best when $\alpha = 0.01$) even with different vale of α.

Thus which algorithm is better won't change the range of value α.

c) The different value of α may be one of the reasons that cause this. And is also could be of fun^n of how appx. value fun can initialize.

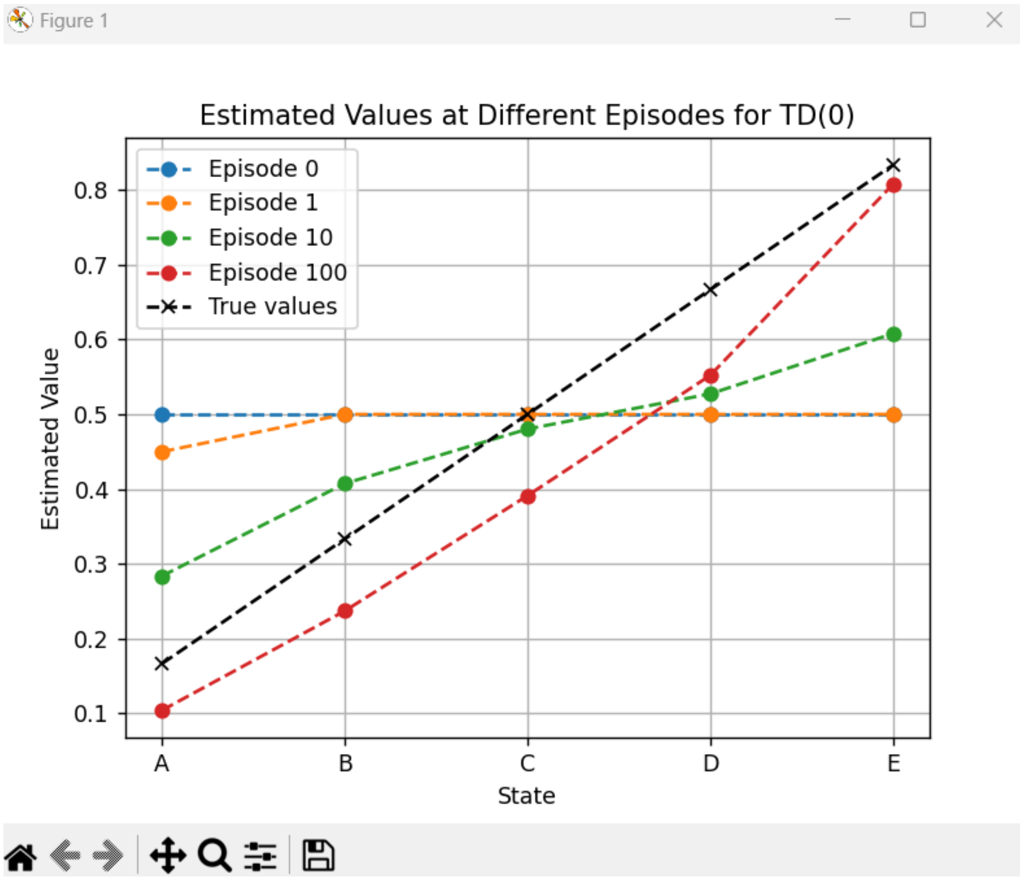


Figure 1
Estimated Values at Different Episodes for TD(0)
Episode 0
Episode 1
Episode 10
Episode 100
True values
Estimated Value
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
A
B
C
D
E
State

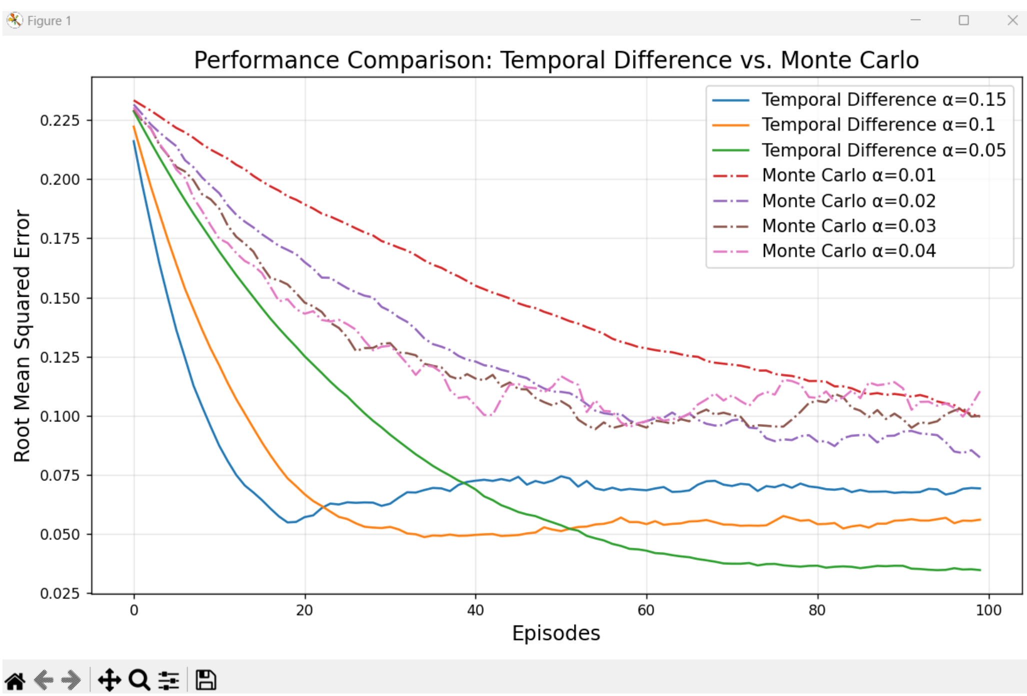


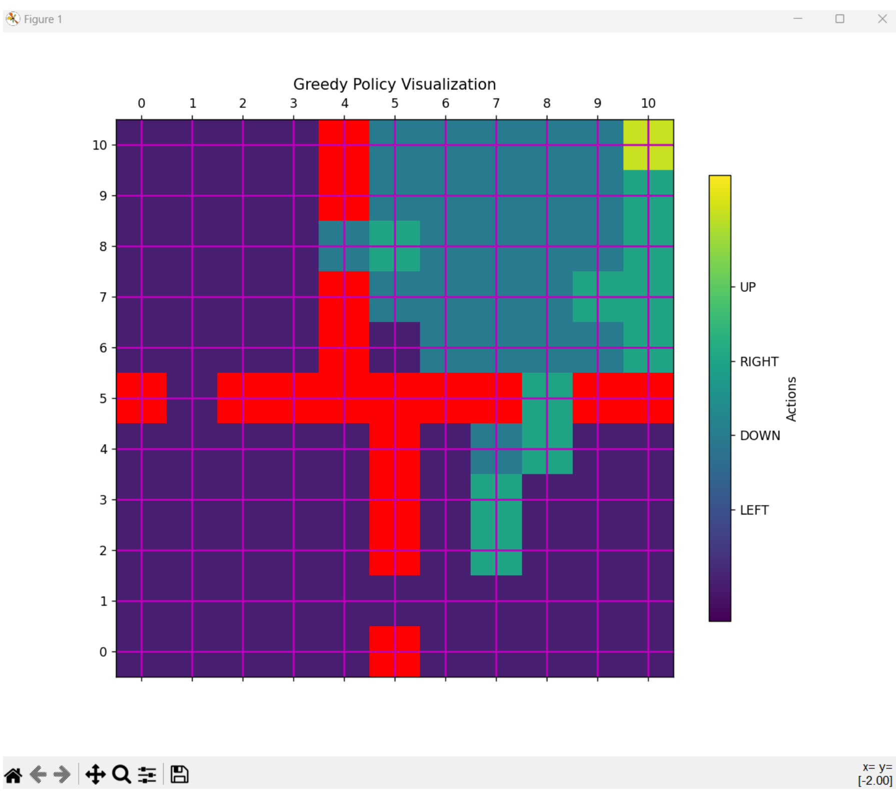
Figure 1
Performance Comparison: Temporal Difference vs. Monte Carlo
Temporal Difference α=0.15
Temporal Difference α=0.1
Temporal Difference α=0.05
Monte Carlo α=0.01
Monte Carlo α=0.02
Monte Carlo α=0.03
Monte Carlo α=0.04
Root Mean Squared Error
0.025
0.050
0.075
0.100
0.125
0.150
0.175
0.200
0.225
0
20
40
60
80
100
Episodes

4)

a), b), c), d)

```
PS C:\Users\rajas\Desktop\ex-5> & C:/Users/rajas/AppData/Local/Microsoft/WindowsApps/python3.11.exe c:/Users/rajas/Desktop/ex-5/4a.py
Generating episodes: 100%|████████████████████████████████████████| 10000/10000 [02:20<00:00, 70.99it/s]
State: (10, 9), Q-values: [0. 1. 1. 1.]
State: (10, 8), Q-values: [0. 1. 1. 0.]
State: (9, 8), Q-values: [0. 1. 1. 0.]
State: (9, 9), Q-values: [1. 1. 1. 1.]
State: (9, 7), Q-values: [0. 1. 0. 0.]
State: (10, 7), Q-values: [1. 1. 0. 1.]
State: (10, 6), Q-values: [0. 1. 0. 0.]
State: (9, 10), Q-values: [0. 1. 1. 0.]
State: (8, 9), Q-values: [0. 0. 1. 0.]
State: (9, 6), Q-values: [0. 1. 0. 0.]
State: (9, 5), Q-values: [0. 1. 0. 1.]
PS C:\Users\rajas\Desktop\ex-5>
```

```
PS C:\Users\rajas\Desktop\ex-5> & C:/Users/rajas/AppData/Local/Microsoft/WindowsApps/python3.11.exe c:/Users/rajas/Desktop/ex-5/4b.py
Generating episodes: 100%|████████████████████████████████████████| 10000/10000 [02:18<00:00, 72.10it/s]
State: (10, 9), Q-values: [0.8 1.  0.9 0.8]
State: (9, 9), Q-values: [0.7 0.9 0.9 0.7]
State: (8, 9), Q-values: [0.6 0.8 0.8 0.6]
State: (7, 9), Q-values: [0.5        0.68171429 0.7        0.5       ]
State: (7, 10), Q-values: [0.6 0.7 0.8 0.6]
State: (9, 10), Q-values: [0.8 0.9 1.  0.8]
State: (8, 10), Q-values: [0.7 0.8 0.9 0.7]
State: (10, 8), Q-values: [0.7 0.9 0.8 0.7]
State: (9, 8), Q-values: [0.6 0.8 0.8 0.6]
State: (8, 8), Q-values: [0.5 0.7 0.7 0.5]
State: (8, 7), Q-values: [0.4        0.6        0.54181818 0.24      ]
State: (9, 7), Q-values: [0.5 0.7 0.7 0.5]
State: (7, 8), Q-values: [0.4 0.6 0.6 0.4]
State: (7, 7), Q-values: [0.3 0.5 0.5 0. ]
State: (10, 7), Q-values: [0.6 0.8 0.7 0.6]
State: (10, 6), Q-values: [0.5 0.7 0.6 0.5]
State: (10, 5), Q-values: [0.4 0.6 0.5 0.5]
State: (9, 5), Q-values: [0.3 0.5 0.5 0.4]
State: (9, 6), Q-values: [0.4 0.6 0.6 0.4]
State: (8, 6), Q-values: [0.  0.5 0.4 0.3]
State: (6, 9), Q-values: [0.5 0.6 0.6 0.4]
State: (6, 10), Q-values: [0.6 0.6 0.7 0.5]
State: (6, 8), Q-values: [0.3 0.5 0.5 0.3]
State: (8, 5), Q-values: [0.  0.4 0.4 0. ]
State: (6, 7), Q-values: [0.3 0.4 0.4 0. ]
State: (5, 8), Q-values: [0.2 0.3 0.4 0.3]
State: (7, 6), Q-values: [0.  0.4 0.4 0.2]
State: (7, 5), Q-values: [0.  0.3 0.  0. ]
State: (6, 5), Q-values: [0.1 0.2 0.2 0.1]
State: (4, 8), Q-values: [0.  0.  0.3 0. ]
State: (4, 9), Q-values: [0.  0.  0.1 0.2]
State: (6, 6), Q-values: [0.  0.  0.3 0. ]
```



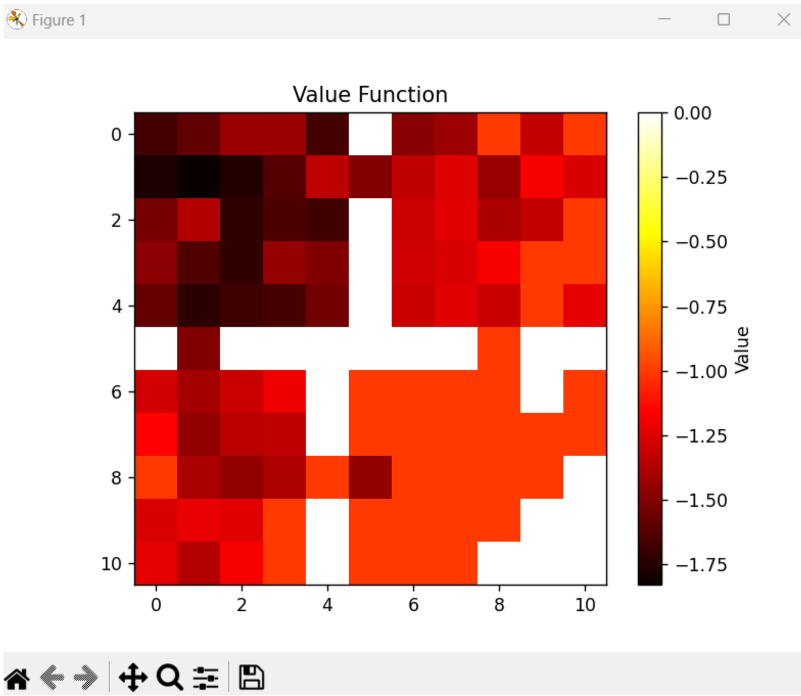


Figure 1
Value Function
0
2
4
6
8
10
0.00
−0.25
−0.50
−0.75
−1.00
−1.25
−1.50
−1.75
Value

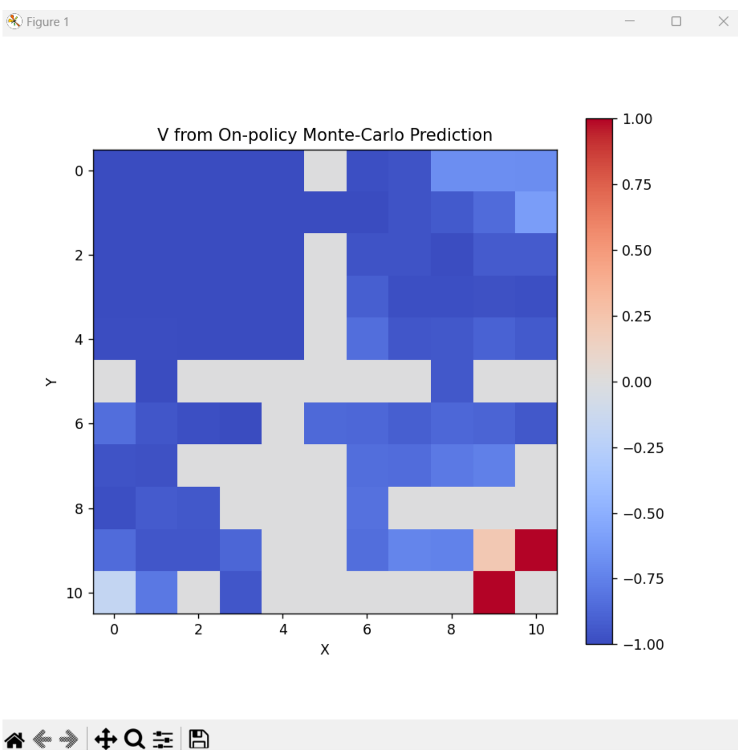


Figure 1
V from On-policy Monte-Carlo Prediction
0
2
4
6
8
10
Y
X
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

e) Comparison between the estimates obtained from random policy Monte Carlo prediction and epsilon-greedy Monte Carlo prediction would likely show:

Random Policy: Lower quality estimates due to lack of goal-directed exploration.

Epsilon-Greedy Policy: Better estimates as the policy partly exploits the knowledge of the environment, leading to more efficient learning.

f) - Dynamic Programming and Comparison: Dynamic Programming: Calculate the true values of $q\pi$ and $q*$ using policy and value iteration, serving as a benchmark for accuracy.

Comparison: Monte Carlo estimates would be measured against the dynamic programming results to determine their accuracy.

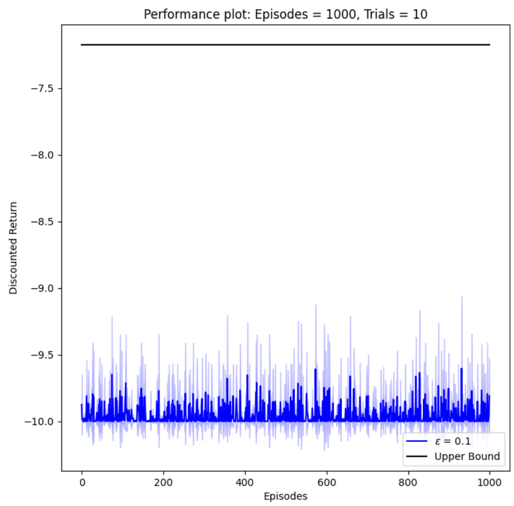
The greedy policy derived from Monte Carlo estimates would be compared to the optimal policy from dynamic programming to evaluate performance.

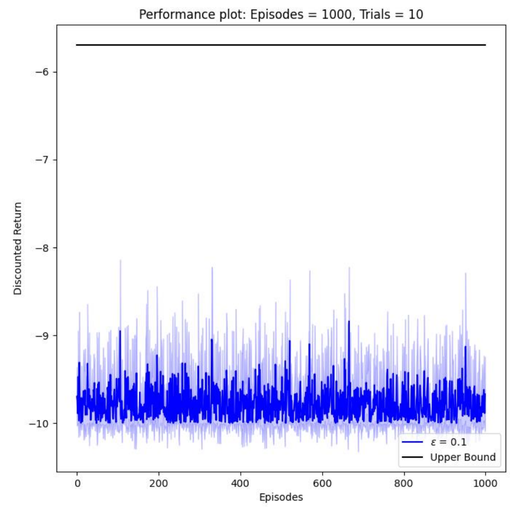
The consistency and reliability of Monte Carlo methods will be assessed by repeating the comparison multiple times.

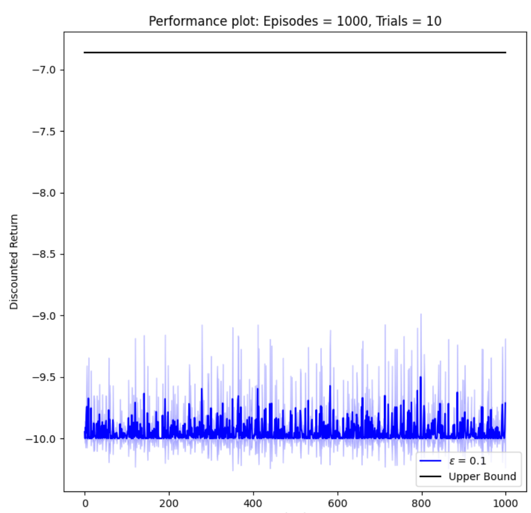
The expectation is that Monte Carlo methods, with sufficient episodes, should approximate the true values closely, with the epsilon-greedy method outperforming the random policy approach. The greedy policy from Monte Carlo is anticipated to be like the optimal policy, particularly with an adequate number of episodes and proper exploration settings.

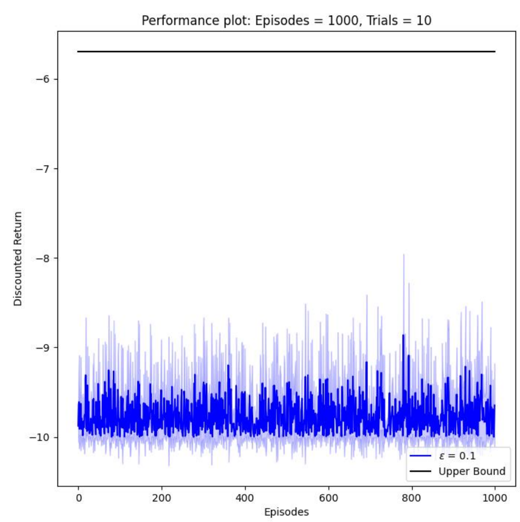
6. Extra Credit

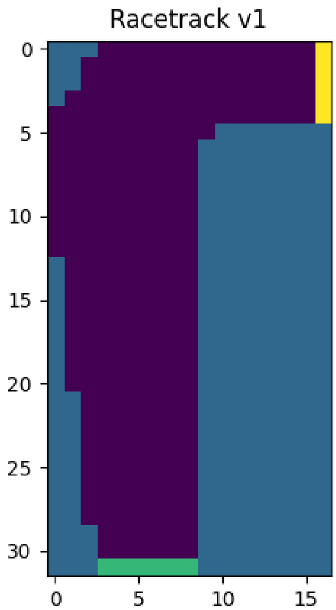
a),b)

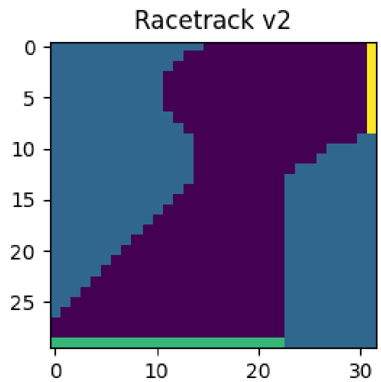












c)The on-policy method, using an ε-greedy strategy, directly improves the policy that governs decision-making and tends to be more stable but may lack exploration. In contrast, off-policy methods, which employ a separate exploratory behavior policy to improve a greedy target policy, have the potential for greater performance but can suffer from instability and convergence issues. Empirical results suggest that the off-policy approach outperforms on-policy due to more effective exploration, particularly in complex environments. The comparison of racetracks shows that the off-policy method yields more consistent performance across different tracks, indicating a stable learning policy even in complex track layouts.