Assignment 1

Reinforcement Learning Programming - CSCN 8020

> July 05, 2024
>
> Student Name: Chen, Kun
>
> Student ID: 8977010

Problem 1: Off-policy Monte Carlo [30]

Problem Statement

Off-policy Monte Carlo with Importance Sampling. The environment can be described as follows:

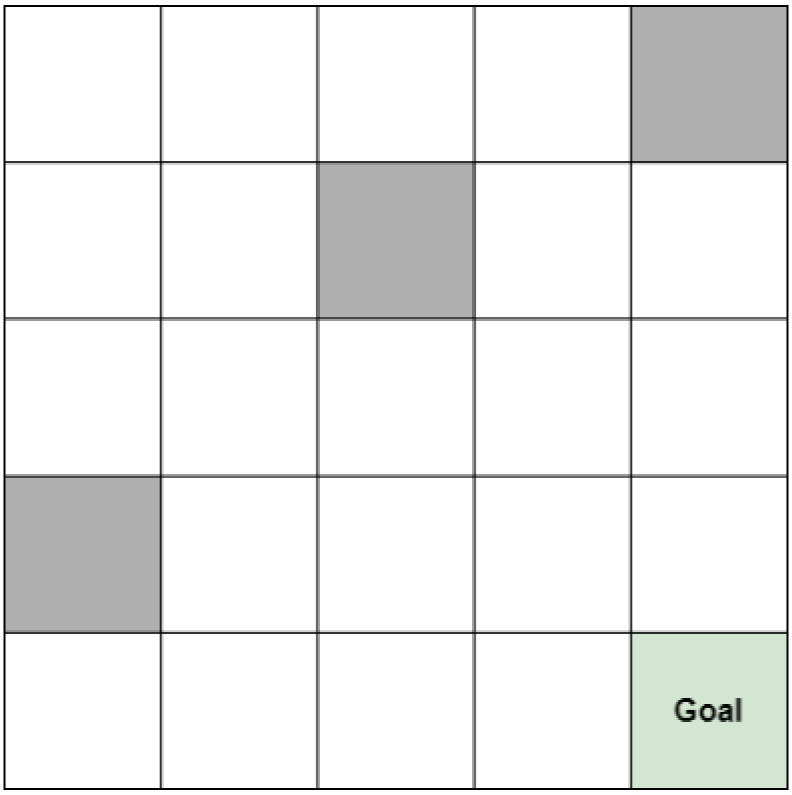


Figure 3: 5x5 Gridworld

- States: states are identified by their row and column, the same as a regular matrix. Ex: the state in row 0 and column 3 is s0,3 (Figure: 3)

  > Terminal/Goal state: The episode ends if the agent reached this state. sGoal = s4,4
  >
  > Grey states: {s1,2, s3,0, s0,4}, these are valid but non-favourable states, as will be seen in the reward function.

- Actions: a1 = right, a2 = down, a3 = left, a4 = up for all states.
- Transitions: If an action is valid, the transition is deterministic, otherwise s' = s
- Rewards R(s):

> R(s) = +10 if s = s4,4
>   = −5 if s ∈ Sgrey = {s0,4, s1,2, s3,0}
>   = −1 otherwise

Task

Implement the off-policy Monte Carlo with Importance sampling algorithm to estimate the value function for the given gridworld. Use a fixed behavior policy b(a|s) (e.g., a random policy) to generate episodes and a greedy target policy.

Suggested steps

1. You can assume a specific discount factor (e.g., γ = 0.9) for this problem.
2. Use the same main algorithm implemented in lecture 4 in class.
3. Generate multiple episodes using the behavior policy b(a|s).
4. For each episode, calculate the returns (sum of discounted rewards) for each state.
5. Use importance sampling to estimate the value function and update the target policy π(a|s)

Deliverables

• Full code with comments to explain key steps and calculations.
• Provide the estimated value function for each state.
• Important Compare the estimated value function obtained from Monte Carlo with the one obtained from Value Iteration in terms of optimization time, number of episodes, computational complexity, and any other aspects you notice.

Answer:

1. **Define Gridworld Environment**

   States: Identify each state by its row and column.

   Actions: Define actions (right, down, left, up).

   Rewards: Define the rewards based on the given conditions.

2. **Define Off-Policy algorithm**

   Input: Arbitrary target policy π

   Initialization: For all s∈S, a∈A(s):

   Q(s,a)⟵arbitrary value

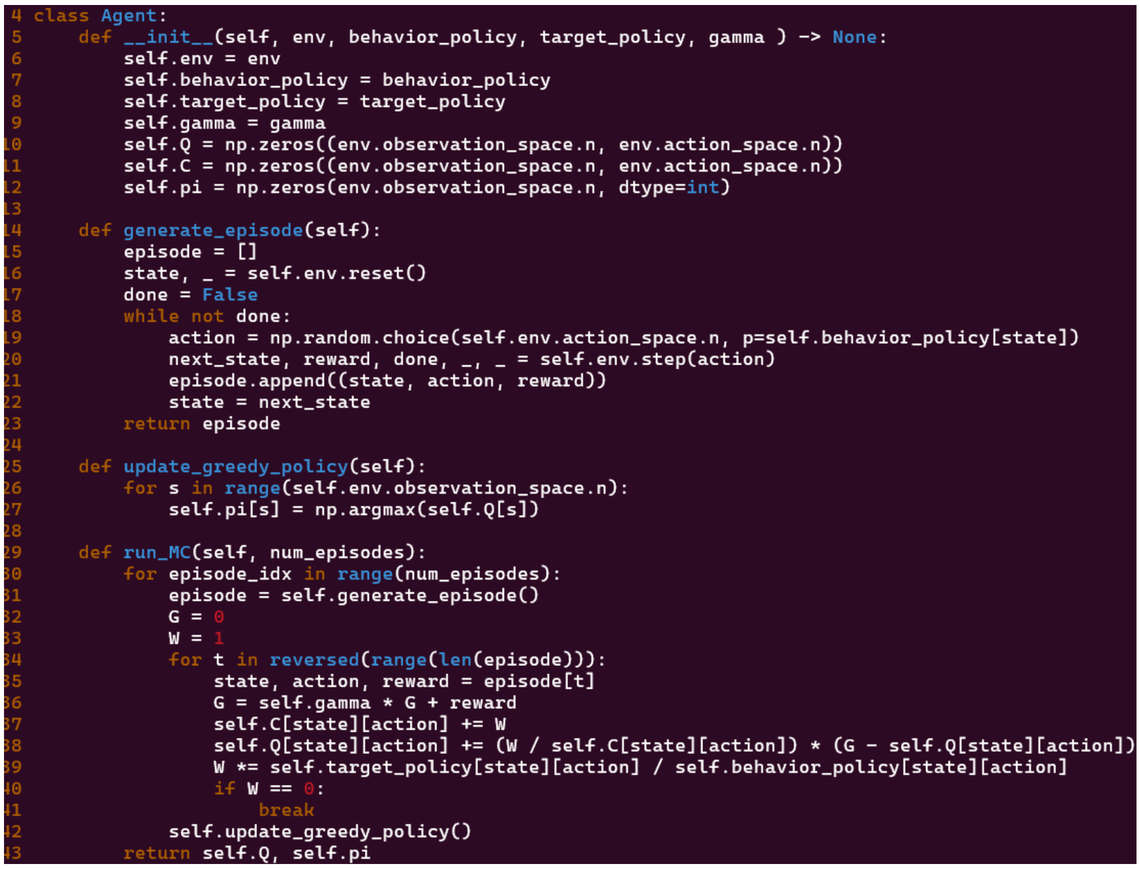
   C(s,a)⟵0

   Repeat forever:

   1. b⟵behavior policy used to generate the episodes
   2. Generate an episode using b: S_0, A_0, R_0,..., S_{T-1}, A_{T-1}, R_T, S_T
   3. G ⟵ 0

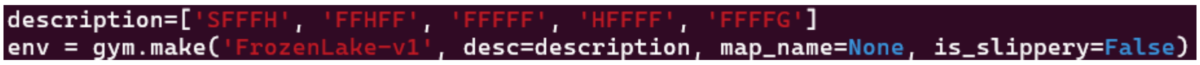
```
4. W ← 1
5. for t = T−1, T−2, …, 0:
    ▪ $G \leftarrow \gamma G + R_{t+1}$
    ▪ $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
    ▪ $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
    ▪ $W \leftarrow W \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$
    ▪ if W=0W = 0W=0 then exit for loop
```

3. Full code in mc_frozen_lake.py & mc_maze_agent.py Below are the snippets of Monte Carlo off-policy training agent



```python
class Agent:
    def __init__(self, env, behavior_policy, target_policy, gamma ) -> None:
        self.env = env
        self.behavior_policy = behavior_policy
        self.target_policy = target_policy
        self.gamma = gamma
        self.Q = np.zeros((env.observation_space.n, env.action_space.n))
        self.C = np.zeros((env.observation_space.n, env.action_space.n))
        self.pi = np.zeros(env.observation_space.n, dtype=int)

    def generate_episode(self):
        episode = []
        state, _ = self.env.reset()
        done = False
        while not done:
            action = np.random.choice(self.env.action_space.n, p=self.behavior_policy[state])
            next_state, reward, done, _, _ = self.env.step(action)
            episode.append((state, action, reward))
            state = next_state
        return episode

    def update_greedy_policy(self):
        for s in range(self.env.observation_space.n):
            self.pi[s] = np.argmax(self.Q[s])

    def run_MC(self, num_episodes):
        for episode_idx in range(num_episodes):
            episode = self.generate_episode()
            G = 0
            W = 1
            for t in reversed(range(len(episode))):
                state, action, reward = episode[t]
                G = self.gamma * G + reward
                self.C[state][action] += W
                self.Q[state][action] += (W / self.C[state][action]) * (G - self.Q[state][action])
                W *= self.target_policy[state][action] / self.behavior_policy[state][action]
                if W == 0:
                    break
            self.update_greedy_policy()
        return self.Q, self.pi
```

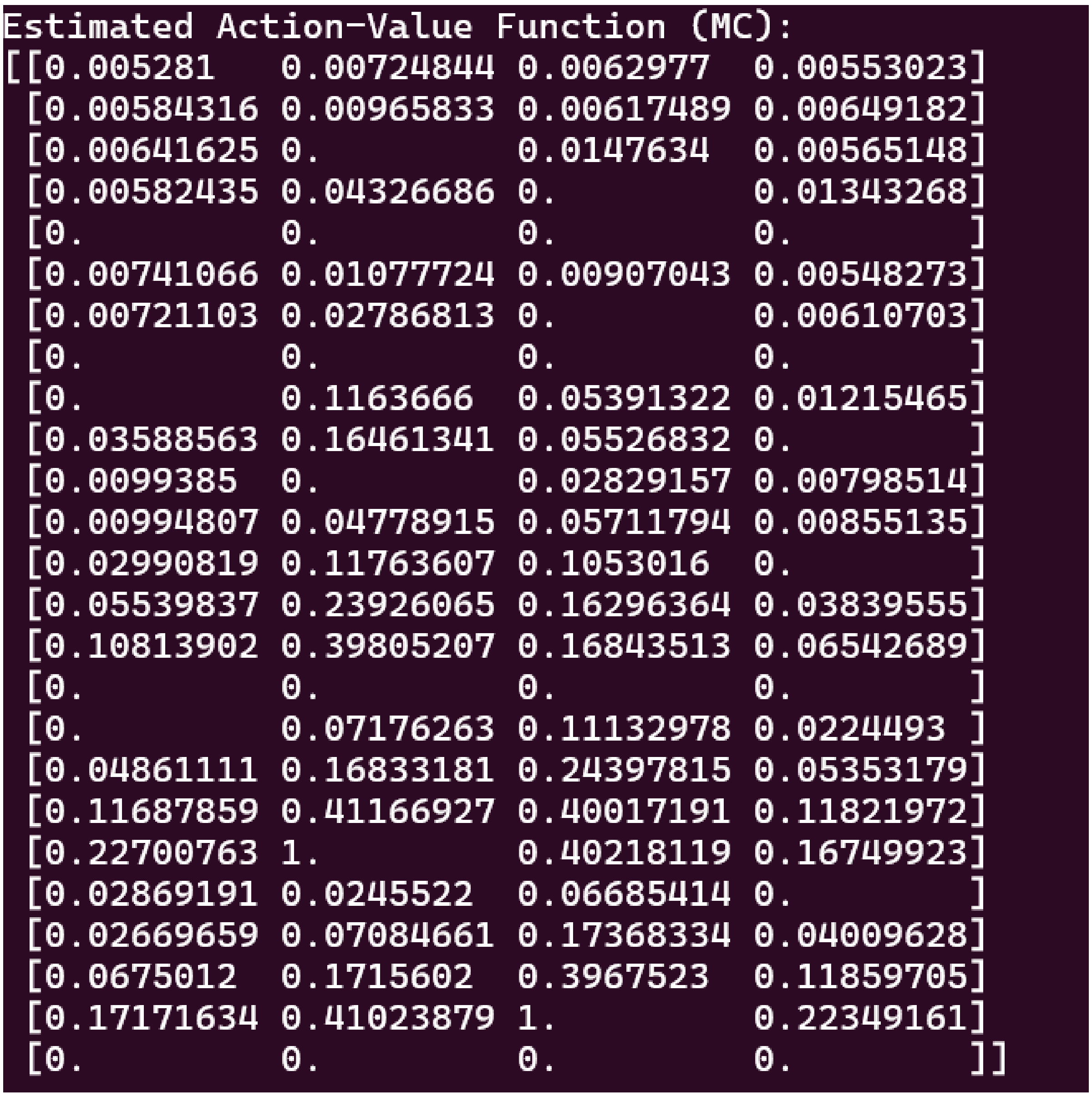
Define a 5x5 Gridworld



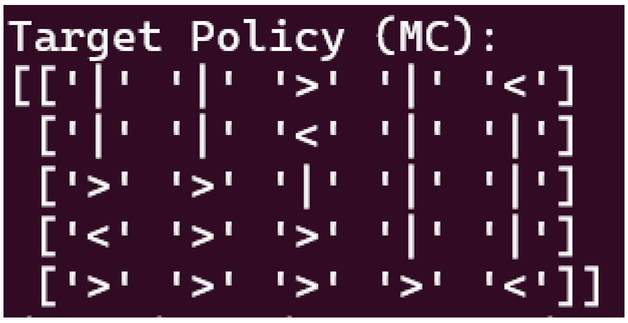
```python
description=['SFFFH', 'FFHFF', 'FFFFF', 'HFFFF', 'FFFFG']
env = gym.make('FrozenLake-v1', desc=description, map_name=None, is_slippery=False)
```

2. Provide the estimated value function for each state.

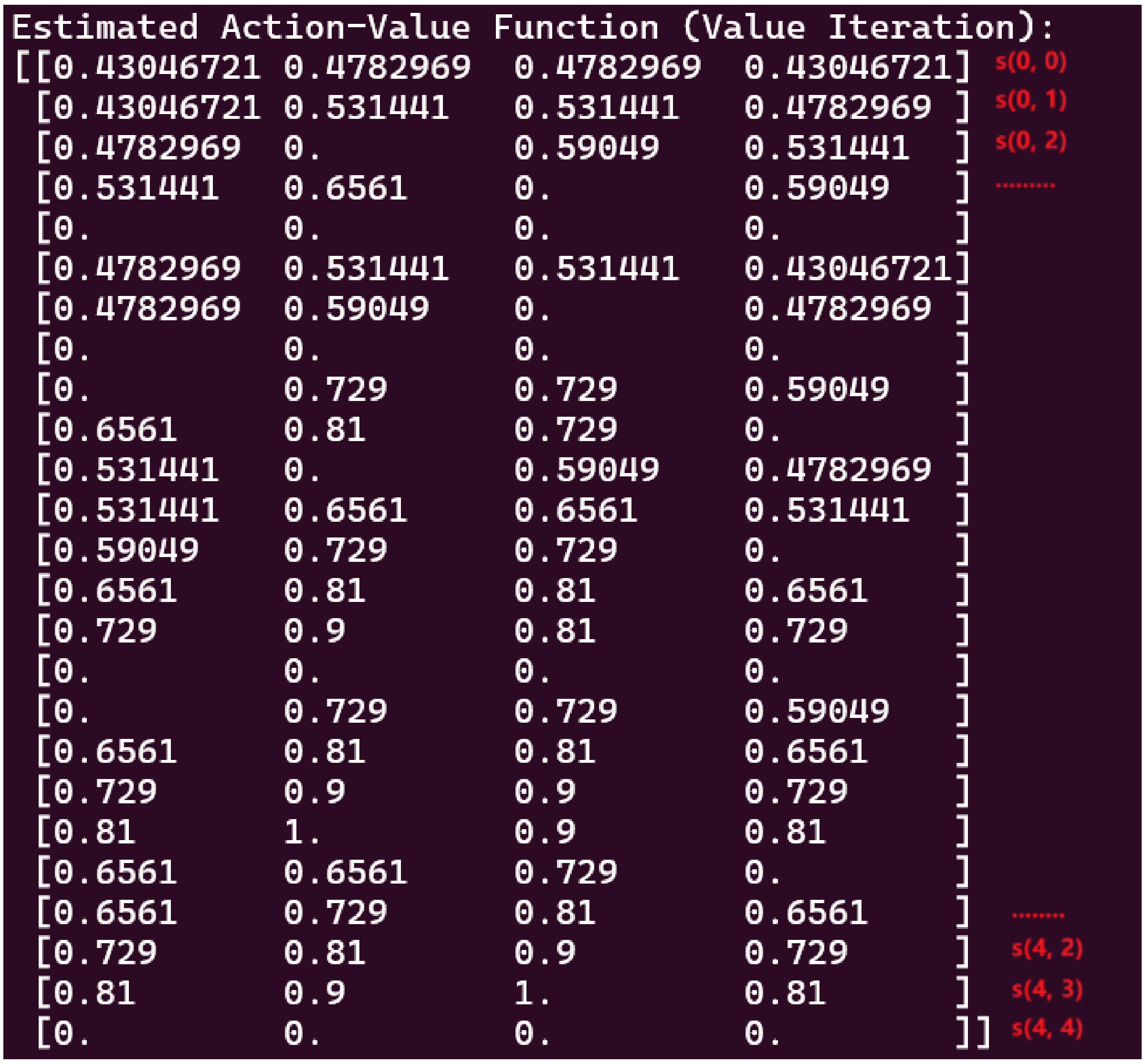
In the following figure is the Action value function, where each line represents a state from s(0, 0) to s(4, 4). Each column represents an action from the first to the last column which is Left Down, Right Up.



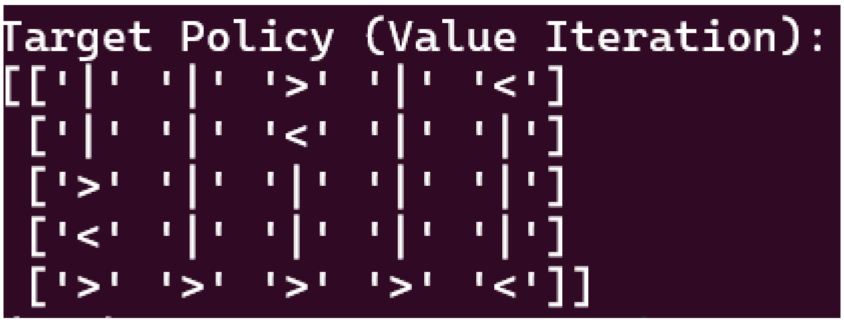
```
Estimated Action-Value Function (MC):
[[0.005281   0.00724844 0.0062977  0.00553023]
 [0.00584316 0.00965833 0.00617489 0.00649182]
 [0.00641625 0.         0.0147634  0.00565148]
 [0.00582435 0.04326686 0.         0.01343268]
 [0.         0.         0.         0.        ]
 [0.00741066 0.01077724 0.00907043 0.00548273]
 [0.00721103 0.02786813 0.         0.00610703]
 [0.         0.         0.         0.        ]
 [0.         0.1163666  0.05391322 0.01215465]
 [0.03588563 0.16461341 0.05526832 0.        ]
 [0.0099385  0.         0.02829157 0.00798514]
 [0.00994807 0.04778915 0.05711794 0.00855135]
 [0.02990819 0.11763607 0.1053016  0.        ]
 [0.05539837 0.23926065 0.16296364 0.03839555]
 [0.10813902 0.39805207 0.16843513 0.06542689]
 [0.         0.         0.         0.        ]
 [0.         0.07176263 0.11132978 0.0224493 ]
 [0.04861111 0.16833181 0.24397815 0.05353179]
 [0.11687859 0.41166927 0.40017191 0.11821972]
 [0.22700763 1.         0.40218119 0.16749923]
 [0.02869191 0.0245522  0.06685414 0.        ]
 [0.02669659 0.07084661 0.17368334 0.04009628]
 [0.0675012  0.1715602  0.3967523  0.11859705]
 [0.17171634 0.41023879 1.         0.22349161]
 [0.         0.         0.         0.        ]]
```



```
Target Policy (MC):
[['|' '|' '>' '|' '<']
 ['|' '|' '<' '|' '|']
 ['>' '>' '|' '|' '|']
 ['<' '>' '>' '|' '|']
 ['>' '>' '>' '>' '<']]
```



```
Estimated Action-Value Function (Value Iteration):
[[0.43046721 0.4782969  0.4782969  0.43046721]
 [0.43046721 0.531441   0.531441   0.4782969 ]
 [0.4782969  0.         0.59049    0.531441  ]
 [0.531441   0.6561     0.         0.59049   ]
 [0.         0.         0.         0.        ]
 [0.4782969  0.531441   0.531441   0.43046721]
 [0.4782969  0.59049    0.         0.4782969 ]
 [0.         0.         0.         0.        ]
 [0.         0.729      0.729      0.59049   ]
 [0.6561     0.81       0.729      0.        ]
 [0.531441   0.         0.59049    0.4782969 ]
 [0.531441   0.6561     0.6561     0.531441  ]
 [0.59049    0.729      0.729      0.        ]
 [0.6561     0.81       0.81       0.6561    ]
 [0.729      0.9        0.81       0.729     ]
 [0.         0.         0.         0.        ]
 [0.         0.729      0.729      0.59049   ]
 [0.6561     0.81       0.81       0.6561    ]
 [0.729      0.9        0.9        0.729     ]
 [0.81       1.         0.9        0.81      ]
 [0.6561     0.6561     0.729      0.        ]
 [0.6561     0.729      0.81       0.6561    ]
 [0.729      0.81       0.9        0.729     ]
 [0.81       0.9        1.         0.81      ]
 [0.         0.         0.         0.        ]]
```



```
Target Policy (Value Iteration):
[['|' '|' '>' '|' '<']
 ['|' '|' '<' '|' '|']
 ['>' '|' '|' '|' '|']
 ['<' '|' '|' '|' '|']
 ['>' '>' '>' '>' '<']]
```

3. Compare the estimated value functions obtained from Monte Carlo and Value Iteration

   3.1. **Optimization time:**

   - Monte Carlo: Generally slower, as it requires sampling many episodes to converge.
   - Value Iteration: Typically faster, as it directly computes the value function without sampling.

   3.2. **Number of episodes:**

   - Monte Carlo: Requires many episodes to achieve a good estimate due to its reliance on sampling.
   - Value Iteration: Typically requires fewer iterations as it updates all states simultaneously.

   3.3. **Computational complexity:**

   - Monte Carlo: O(number of episodes * average episode length). Has higher variance and computational overhead due to the generation and processing of episodes.

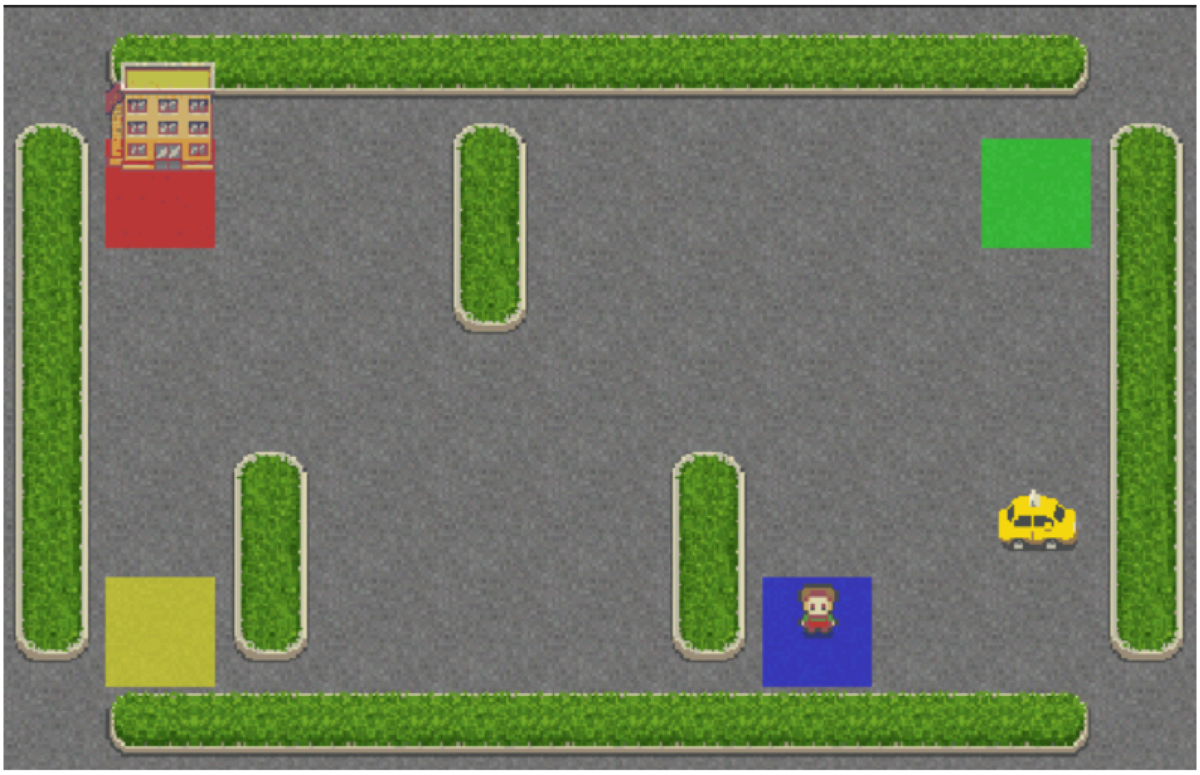
- Value Iteration: O(number of iterations * number of states * number of actions) More efficient computationally as it performs updates based on the entire state space and action space.

In this particular problem, the two algorithms yield essentially similar policy.

Problem 2: Q-Learning [70]

Introduction

You will work with the Taxi environment and implement QLearning. The way you interact with the environment will be very similar to the gym environments described in class. Therefore, most of the code we discussed is directly applicable.



The environment has 6 discrete actions, 500 discrete states, and the following rewards:
• -1 per step unless other reward is triggered.
• +20 delivering passenger.
• -10 executing “pickup” and “drop-off” actions illegally.

Action space
• 0: Move south (down)
• 1: Move north (up)
• 2: Move east (right)
• 3: Move west (left)
• 4: Pickup passenger
• 5: Drop off passenger

Destinations:
• 0: Red
• 1: Green
• 2: Yellow
• 3: Blue

An observation is returned as an int() that encodes the corresponding state, calculated by ((taxi row ∗5 + taxi col) ∗ 5 + passenger location) ∗ 4 + destination

Helper Utility
To assist you in understanding the environment further, you were provided with a file (assignment2 utils.py) that has a few methods to allow loading the environment and printing some basic information about it. It also allows you to switch to the detailed observation description using the state scalar value.

Problem Statement
Implement the Q-Learning algorithm on the Taxi environment from OpenAI Gym. Train an agent to efficiently navigate and pick up passengers. Use the following hyperparameters:
• Learning Rate α: 0.1
• Exploration Factor ϵ: 0.1
• Discount Factor γ: 0.9

Tasks
• Implement the Q-Learning algorithm and train an agent on the Taxi environment.
• Report the following metrics after training:

1. Total episodes
2. Total steps taken per episode
3. Return per episode
4. Average cumulative reward per episode (you can smooth your plot by using average cumulative reward every **n** episodes, keeping *n* constant across your runs).

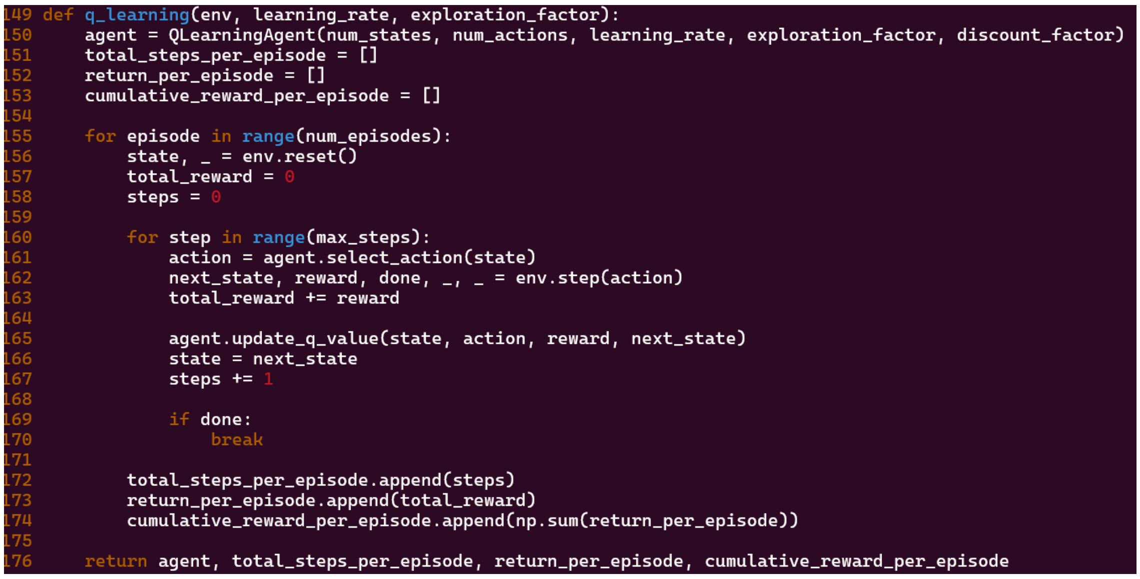
Note: Average cumulative reward = $\sum_{i}^{inf} R_i$

• Make a deliberate change to the following parameters (separately) and use each value once.
– Learning Rate α = [0.01, 0.001, 0.2]
– Exploration Factor ϵ = [0.2, 0.3]

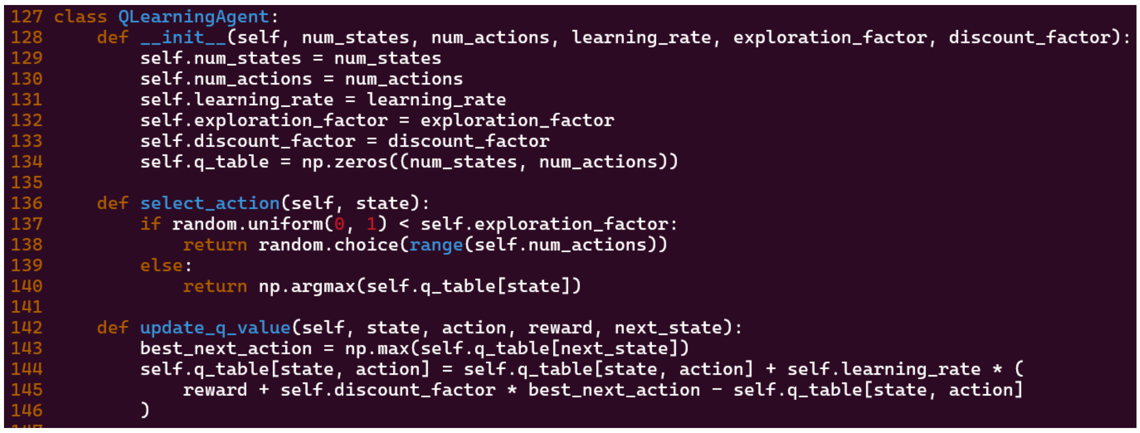
Deliverables
• Python code implementing Q-Learning and running it for the different hyperparameters. [20]
• A report (in PDF) containing the metrics, observations, and comments on the parameter change. Make sure you include appropriate plots for the previously stated metrics to support your arguments. [30]
• Based on your findings, choose what you think would be the best combination of learning rate and exploration factor and re-run the training. Report and comment on the differences observed. [20]

Answer:

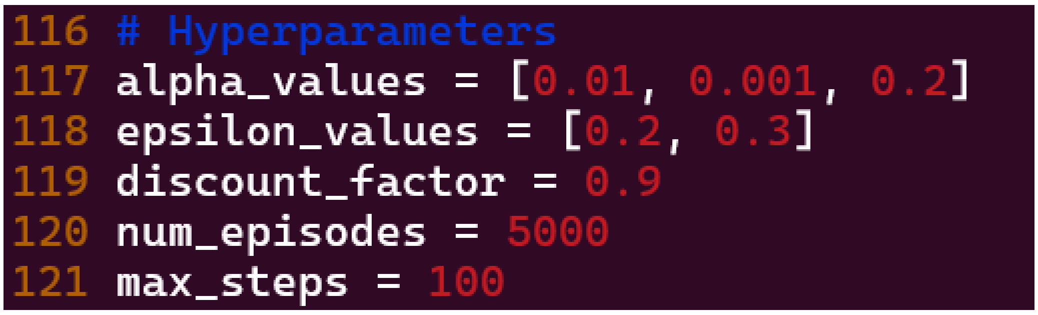
1. Full Python code: in assignment2_utils.py. Below are the snippets of Q-Learning and Q-Learning Agent



```python
def q_learning(env, learning_rate, exploration_factor):
    agent = QLearningAgent(num_states, num_actions, learning_rate, exploration_factor, discount_factor)
    total_steps_per_episode = []
    return_per_episode = []
    cumulative_reward_per_episode = []

    for episode in range(num_episodes):
        state, _ = env.reset()
        total_reward = 0
        steps = 0

        for step in range(max_steps):
            action = agent.select_action(state)
            next_state, reward, done, _, _ = env.step(action)
            total_reward += reward

            agent.update_q_value(state, action, reward, next_state)
            state = next_state
            steps += 1

            if done:
                break

        total_steps_per_episode.append(steps)
        return_per_episode.append(total_reward)
        cumulative_reward_per_episode.append(np.sum(return_per_episode))

    return agent, total_steps_per_episode, return_per_episode, cumulative_reward_per_episode
```



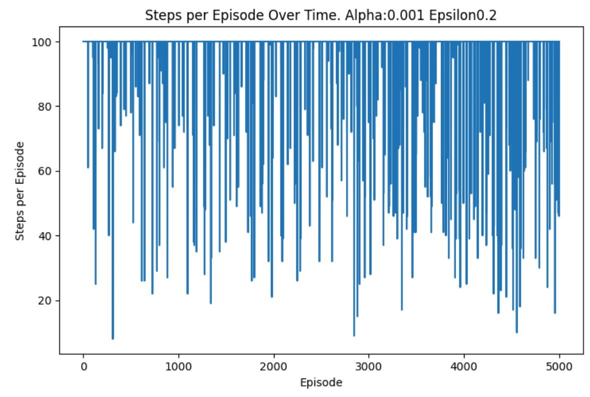
```python
class QLearningAgent:
    def __init__(self, num_states, num_actions, learning_rate, exploration_factor, discount_factor):
        self.num_states = num_states
        self.num_actions = num_actions
        self.learning_rate = learning_rate
        self.exploration_factor = exploration_factor
        self.discount_factor = discount_factor
        self.q_table = np.zeros((num_states, num_actions))

    def select_action(self, state):
        if random.uniform(0, 1) < self.exploration_factor:
            return random.choice(range(self.num_actions))
        else:
            return np.argmax(self.q_table[state])

    def update_q_value(self, state, action, reward, next_state):
        best_next_action = np.max(self.q_table[next_state])
        self.q_table[state, action] = self.q_table[state, action] + self.learning_rate * (
            reward + self.discount_factor * best_next_action - self.q_table[state, action]
        )
```

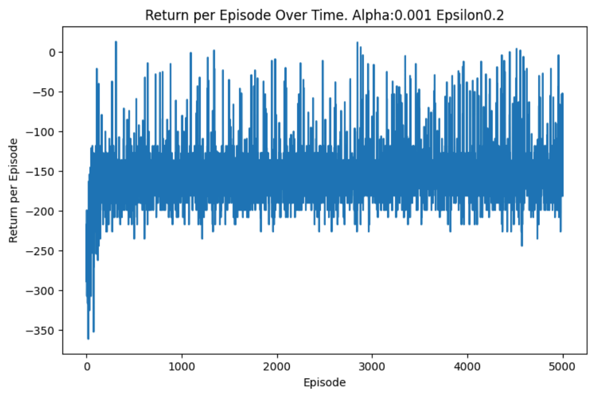


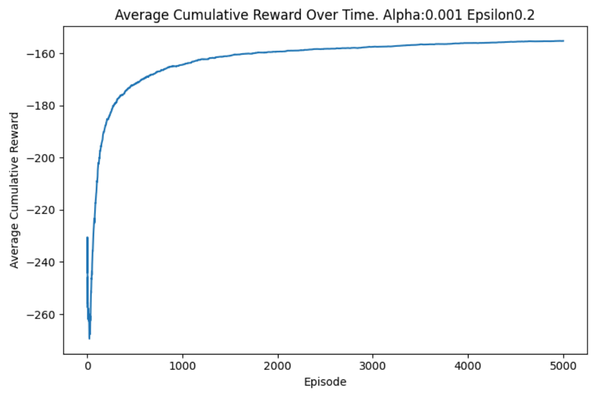
```python
# Hyperparameters
alpha_values = [0.01, 0.001, 0.2]
epsilon_values = [0.2, 0.3]
discount_factor = 0.9
num_episodes = 5000
max_steps = 100
```

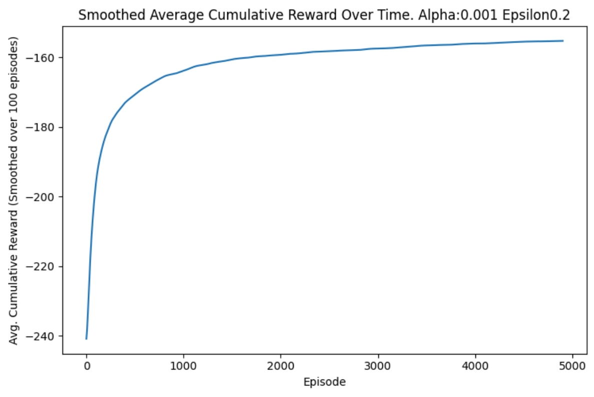
2. metrics:

Alpha 0.001 Epsilon: 0.2

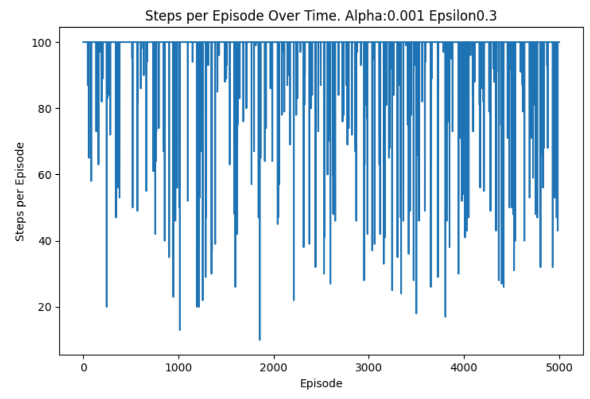


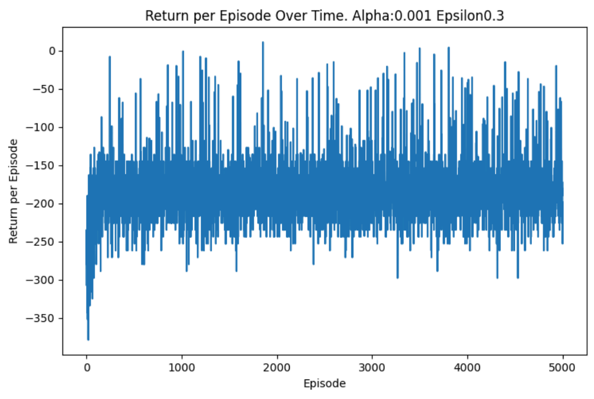


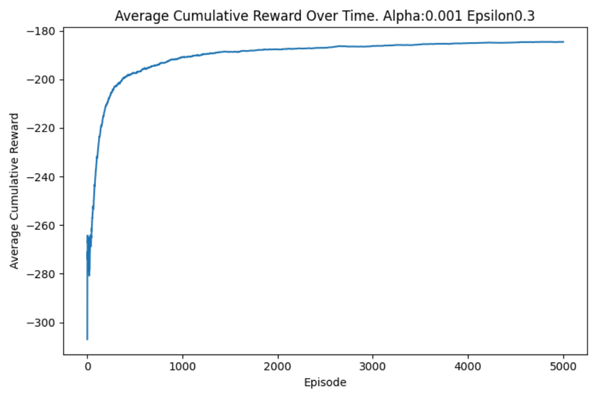


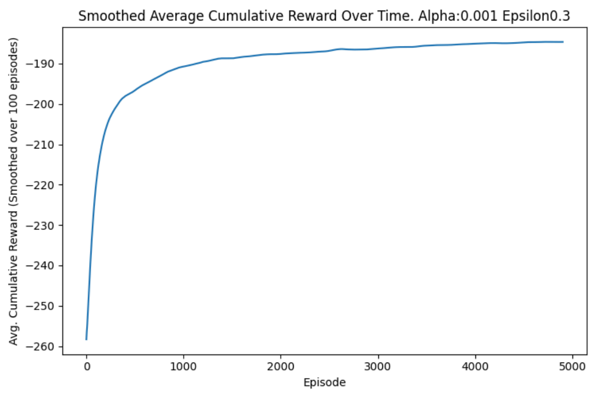


Alpha 0.001 Epsilon: 0.3

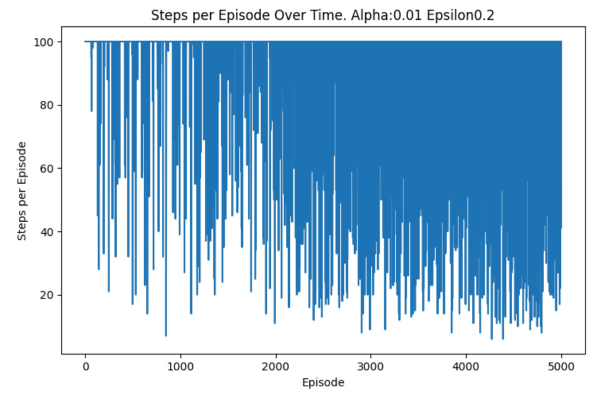


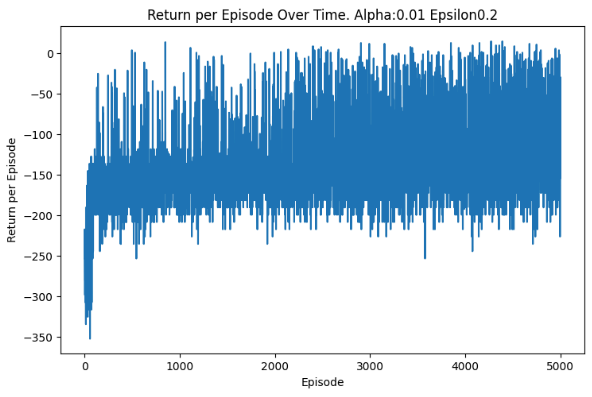


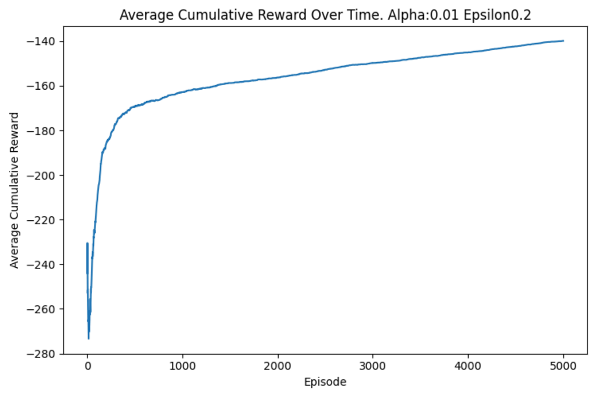


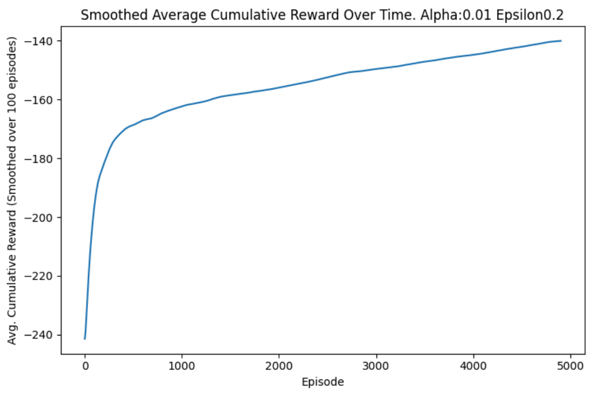


Alpha 0.01 Epsilon: 0.2

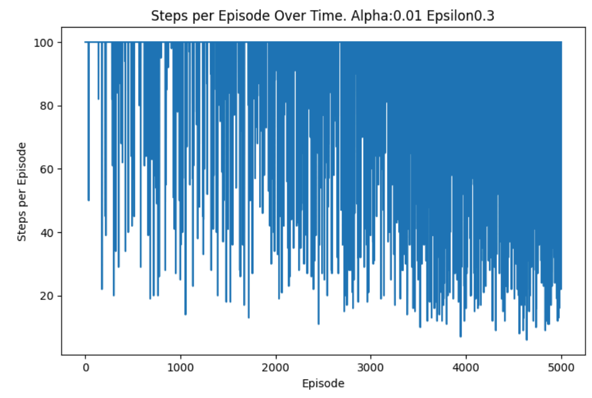


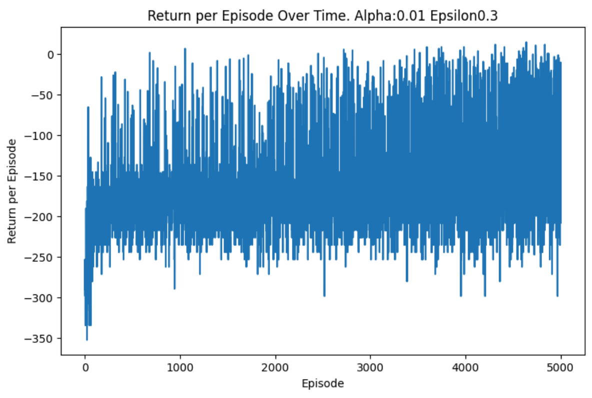


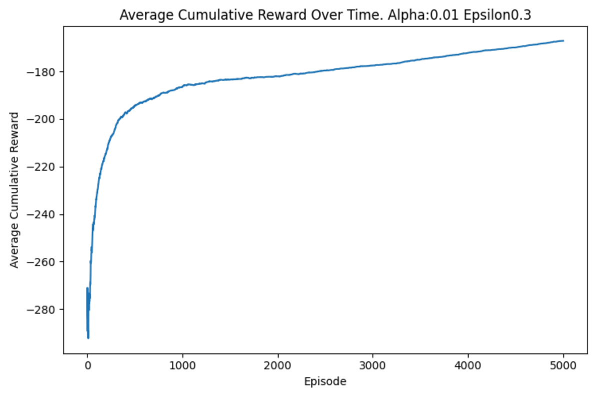


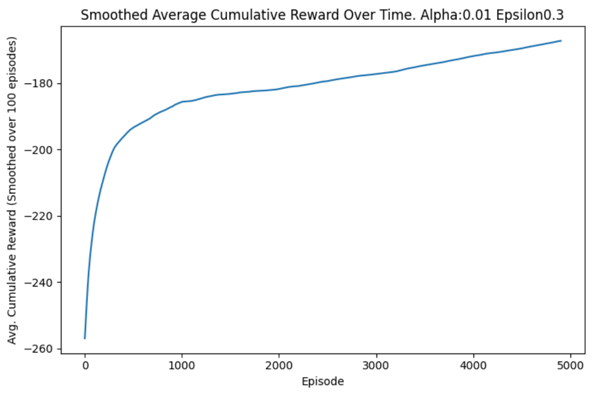


Alpha 0.01 Epsilon: 0.3

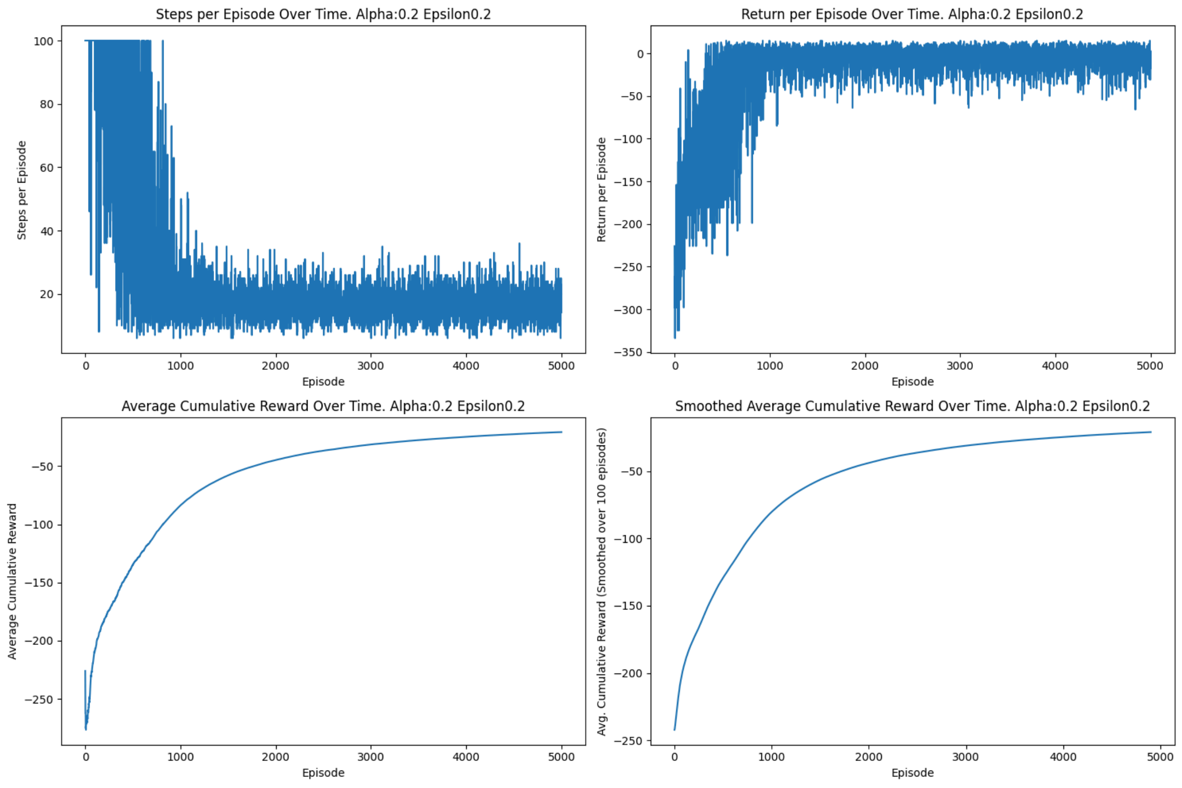




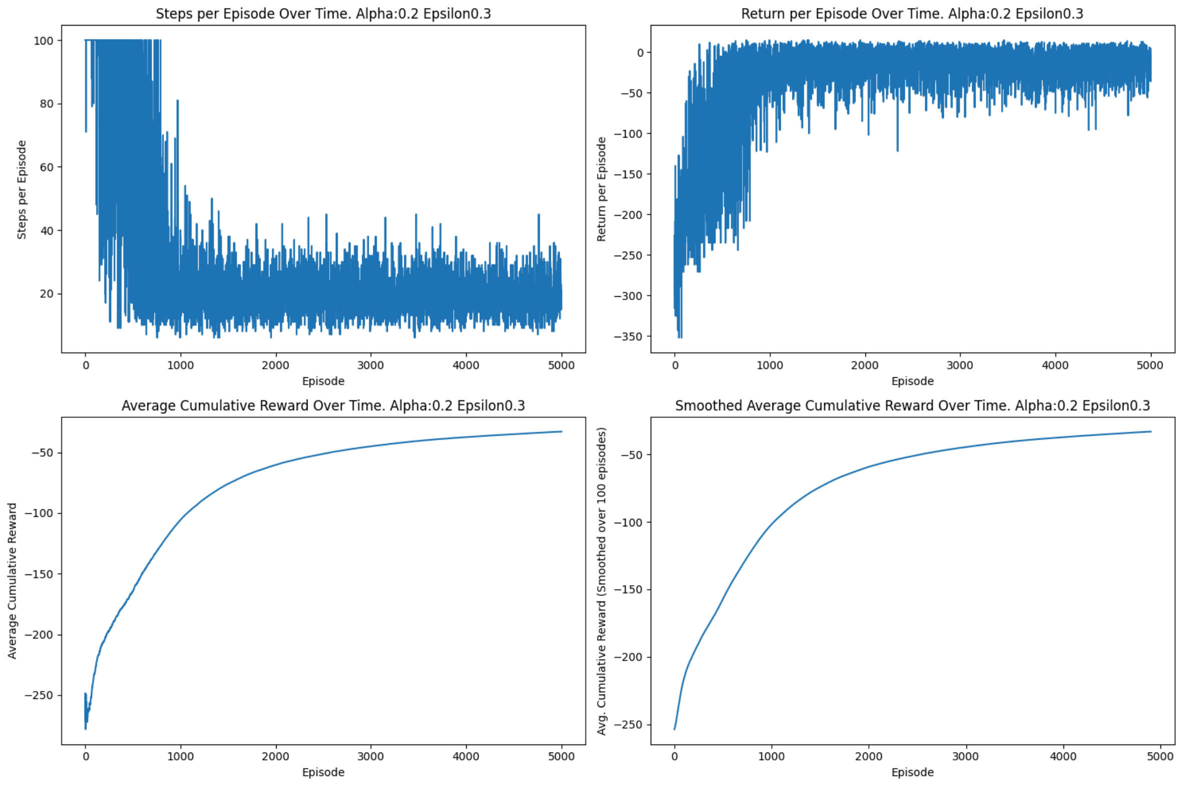




Alpha 0.2 Epsilon: 0.2



Alpha 0.2 Epsilon: 0.3



Looks like Alpha 0.2 Epsilon: 0.2 is the best parameter, fast convergence and more stability. Reducing the total steps per episode and increasing the return per episode. Higher learning rates, and possible over-exploration from higher exploration factors.

By choosing the best learning rate and exploration factor based on average cumulative reward, the agent's performance should improve, resulting in more efficient and effective navigation and passenger pick-up/drop-off.