University of Tehran

School of Mechanical Engineering

Artificial Intelligence

Home Work 6

Professor:

Dr. Masoud Shariat Panahi

Author:

Masoud Pourghavam

July, 2023

Table of Contents

1. Questions 4
   A. Off-policy and on-policy 4
      i. Difference 4
      ii. Applications 5
   B. Model-free 6
      i. Challenges 6
      ii. Solutions 7
   C. Reward 8
      .i Issue 8
      ii. Suggestions 9
   D. Village 11
      i. MDP 11
      ii. Algorithms 13
2. Autonomous taxi 20
      i. Codes explanation 22
   A. Action and observation spaces 31
   B. Unavailable states 35
   C. Stability and convergence 38
      i. 1st case: Constant learning rate and discount factor = 0.999 38
      ii. 2nd case: Learning rate decay and discount factor = 0.999 43
      iii. 3rd case: Constant learning rate and discount factor = 0.9 47
      iv. 4th case: Learning rate decay and discount factor = 0.9 52
   D. Rendering 56
      i. 1st case: Constant learning rate and discount factor = 0.999 57
      ii. 2nd case: Learning rate decay and discount factor = 0.999 59
      iii. 3rd case: Constant learning rate and discount factor = 0.9 60
      iv. 4th case: Learning rate decay and discount factor = 0.9 61
   E. Deep Q-Learning 63

List of Figures

Figure 1. On-Policy vs. off-Policy for deep reinforcement learning 5
Figure 2. Action space 32
Figure 3. Action space in grid representation 33
Figure 4. Unavailable states analysis 35
Figure 5. Average rewards in terms of episodes for Monte Carlo - Constant learning rate and discount factor = 0.999 39
Figure 6. Average rewards in terms of episodes for Q-Learning - Constant learning rate and discount factor = 0.999 41
Figure 7. Average rewards in terms of episodes for Monte Carlo - Learning rate decay and discount factor = 0.999 44
Figure 8. Average rewards in terms of episodes for Q-Learning - Learning rate decay and discount factor = 0.999 45
Figure 9. Average rewards in terms of episodes for Monte Carlo - Constant learning rate and discount factor = 0.9 48
Figure 10. Average rewards in terms of episodes for Q-Learning - Constant learning rate and discount factor = 0.9 50
Figure 11. Average rewards in terms of episodes for Monte Carlo - Learning rate decay and discount factor = 0.9 53
Figure 12. Average rewards in terms of episodes for Q-Learning - Learning rate decay and discount factor = 0.9 54
Figure 13. Rendering of Monte Carlo - Constant learning rate and discount factor = 0.999 58
Figure 14. Rendering of Q-Learning - Constant learning rate and discount factor = 0.999 58
Figure 15. Rendering of Monte Carlo - Learning rate decay and discount factor = 0.999 59
Figure 16. Rendering of Q-Learning - Learning rate decay and discount factor = 0.999 60
Figure 17. Rendering of Monte Carlo - Constant learning rate and discount factor = 0.9 60
Figure 18. Rendering of Q-Learning - Constant learning rate and discount factor = 0.9 61
Figure 19. Rendering of Monte Carlo - Learning rate decay and discount factor = 0.9 62
Figure 20. Rendering of Q-Learning - Learning rate decay and discount factor = 0.9 62
Figure 21. DQN model visualization 66
Figure 22. DQN average rewards in terms of episodes 69

List of Tables

Table 1. Libraries used in section 2. 20
Table 2. Parameters of the 1st case for monte carlo 38
Table 3. Monte Carlo algorithm results for 1st case 39
Table 4. Parameters of the 1st case for q-learning 41
Table 5. Q-Learning algorithm results for 1st case 42
Table 6. Parameters of the 2nd case for monte carlo 43
Table 7. Monte Carlo algorithm results for 2nd case 44
Table 8. Parameters of the 2nd case for q-learning 45
Table 9. Q-Learning algorithm results for 2nd case 46
Table 10. Parameters of the third case for monte carlo 47
Table 11. Monte Carlo algorithm results for 3rd case 48
Table 12. Parameters of the third case for q-learning 49
Table 13. Q-Learning algorithm results for 3rd case 50
Table 14. Parameters of the 4th case for monte carlo 52
Table 15. Monte Carlo algorithm results for 4th case 53
Table 16. Parameters of the 4th case for q-learning 54
Table 17. Q-Learning algorithm results for 4th case 55
Table 18. Parameters of the deep q-learning algorithm 63

1. Questions

A. Off-policy and on-policy

i. Difference

When it comes to convergence and stability in reinforcement learning, on-policy and off-policy methods exhibit distinct characteristics. On-policy methods, such as SARSA and REINFORCE, update their value functions based on the data generated by the current policy. These methods learn and improve the policy simultaneously, converging to a policy that aligns closely with the optimal policy for the current value function. The key advantage of on-policy methods is their ability to refine the policy as the value function improves. However, they can be more susceptible to instability due to changes in the policy during learning. As the policy constantly evolves, the data distribution used for updates changes as well. This dynamic nature of the policy can introduce high variance in the updates and slow down the convergence process. In contrast, off-policy methods like Q-learning and DQN learn the value function using data collected by a different policy called the behavior policy, while simultaneously updating a target policy. These methods have the advantage of being able to converge to the optimal value function for the target policy, regardless of the behavior policy used for data collection. By decoupling the target policy from the behavior policy, off-policy methods can achieve greater stability during learning. The separation allows for consistent updates, as the target policy remains fixed throughout the value function updates. This stability helps in dealing with variations in the data and leads to more reliable learning.

So, on-policy methods excel in providing convergence guarantees but can be sensitive to changes in the policy, potentially resulting in instability. On the other hand, off-policy methods offer greater stability by separating the target policy from the behavior policy, enabling more robust learning from a diverse range of data. In Figure 1, comparison of these methods for deep reinforcement learning is given.

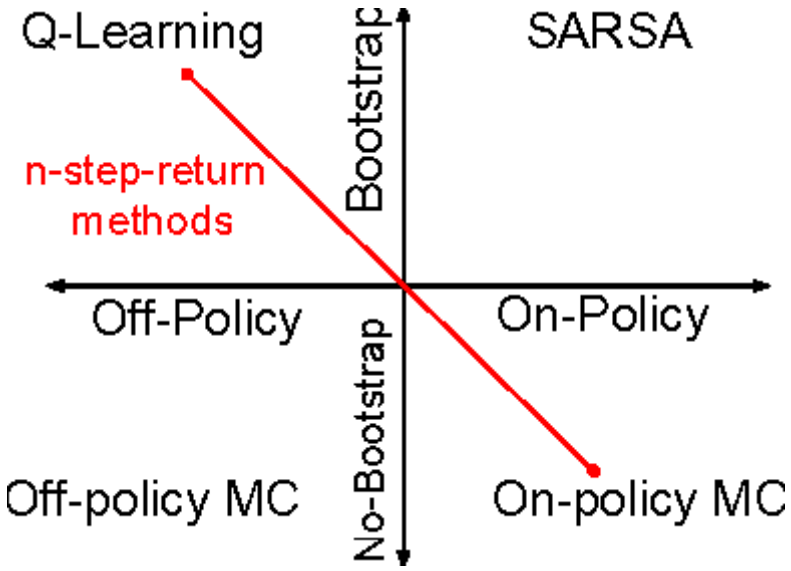

Figure 1. On-Policy vs. off-Policy for deep reinforcement learning

ii. Applications

On-policy methods:

On-policy methods, such as REINFORCE and Proximal Policy Optimization (PPO), find their suitability in different use cases. These methods are commonly employed when the primary objective is to optimize the policy itself. They excel in scenarios where the policy needs to be continuously refined based on the agent's own experiences. For instance, in robotics applications, on-policy methods are often used to train agents that interact directly with the environment, allowing them to learn and improve their policies in real-time. By optimizing an objective function that measures the expected return, these methods iteratively update the policy, making them well-suited for policy optimization tasks.

Off-policy methods:

Off-policy methods, such as Q-learning and DQN, serve specific use cases that leverage their distinctive characteristics. They are particularly suitable for value-based reinforcement learning tasks. These methods focus on learning the value function without explicitly optimizing the policy. Consequently, they excel in environments with large state and action spaces, as they efficiently learn the value of different actions in a given state. Additionally, off-policy methods often employ experience replay, which involves storing past experiences in a memory buffer. This enables more efficient utilization of data, allowing the agent to learn from a diverse range of experiences and enhance overall learning performance. Furthermore, off-

policy methods are advantageous in batch reinforcement learning scenarios where a fixed dataset of experiences is available. By decoupling the behavior policy used for data collection from the target policy being learned and improved, off-policy methods can effectively learn from the fixed dataset without requiring further interaction with the environment.

So, we can conclude that on-policy methods are well-suited for policy optimization tasks and interactive learning scenarios, where the agent directly interacts with the environment. Off-policy methods, on the other hand, are suitable for value-based reinforcement learning tasks, especially in environments with large state and action spaces. They are also advantageous in scenarios involving experience replay and batch reinforcement learning, where a fixed dataset is available. Consideration of these factors, along with the trade-offs in convergence, stability, sample efficiency, and computational requirements, aids in selecting the appropriate method for a given use case.

B. Model-free

i. Challenges

In model-free reinforcement learning, where the machine learns optimal strategies through iterative interactions with the environment, there are several challenges that arise due to the reliance on outcomes for obtaining the value function.

One problem is sample inefficiency. Model-free methods often require a large number of interactions with the environment to accurately estimate the value function. This can be time-consuming and computationally expensive, particularly in complex environments with a vast number of possible states. It also limits the scalability of model-free approaches in real-world applications. Convergence and stability are also concerns in model-free reinforcement learning. Depending on the algorithm and learning settings, convergence issues or instability may arise during the learning process. The learning process can be sensitive to hyperparameter settings, such as learning rates, which can impact the convergence and stability of the learned policy.

The credit assignment problem is another issue. It arises when delayed rewards occur, making it difficult to attribute the rewards to specific actions or states. This challenge hinders the accurate updating of the value function, which is crucial for learning optimal policies. Another challenge is the exploration-exploitation trade-off. To learn the optimal policy, the agent needs to balance between exploring new actions and exploiting known good actions. Inadequate

exploration may lead to suboptimal policies, while excessive exploration can slow down learning and delay convergence to an optimal solution. Furthermore, the non-stationarity of the environment poses a problem. In model-free learning, the learned policy may become ineffective or outdated when the environment dynamics change. Adapting to these changes and maintaining up-to-date policies becomes challenging in such cases.

Finally, the curse of dimensionality adds to the challenges. As the dimensionality of the state space increases, the number of possible states grows exponentially. This makes it difficult for the agent to explore and learn an optimal policy, as it needs to explore a vast number of state-action pairs.

ii. Solutions

In order to tackle the challenges in model-free reinforcement learning, researchers have devised several solutions and techniques. These approaches aim to enhance the efficiency, stability, and effectiveness of the learning process. One key technique is experience replay, where past experiences are stored and randomly sampled during training. This method helps break the temporal correlation between consecutive experiences, leading to improved sample efficiency and more effective learning.

Proper parameter initialization and optimization techniques are crucial for ensuring stability and convergence. Techniques such as learning rate scheduling, momentum, and adaptive learning rate methods like Adam aid in finding better solutions and improving convergence speed. Exploration strategies play a vital role in addressing the exploration-exploitation trade-off. Techniques such as ε-greedy exploration, softmax exploration, and Upper Confidence Bound (UCB) allow agents to strike a balance between exploring new actions and exploiting already known good actions. By dynamically adjusting the exploration level, agents can efficiently learn optimal policies while gathering valuable information from the environment. Function approximation methods, such as neural networks, have proven effective in handling large state spaces. By approximating the value function or policy, these methods allow agents to generalize their learning across similar states and actions, thus improving efficiency and scalability. To overcome the credit assignment problem, eligibility traces like TD(λ) can be employed. These traces help in attributing rewards to the relevant actions and states, even in scenarios where rewards are delayed. This enables agents to accurately update the value function and learn optimal policies. Transfer learning and knowledge transfer can be leveraged

to accelerate learning in new tasks by utilizing knowledge and pre-trained models from related tasks. This approach enhances sample efficiency and enables agents to leverage prior learning experiences.

Curriculum learning, involving a gradual increase in the complexity of learning tasks, can facilitate more efficient learning. By starting with simpler tasks and progressively introducing more challenging ones, agents can acquire foundational skills and converge to optimal solutions more effectively. Regularization techniques, such as L1 or L2 regularization, can prevent overfitting and encourage smoother policies, improving generalization and stability. Additionally, exploration bonuses can be added to the reward signal to explicitly promote exploration in unexplored regions of the state space.

C. Reward

i. Issue

When rewards in reinforcement learning are delayed, meaning they are not immediately obtained after an action, several challenges arise that can impact the learning process. One significant issue is the credit assignment problem, where it becomes difficult to determine which actions and states were responsible for earning the delayed rewards. This challenge hampers the agent's ability to accurately update the value function and learn optimal policies based on delayed feedback. Assigning credit to actions that had long-term consequences also becomes more complex as the time gap between the action and the reward increases, making the temporal credit assignment a challenging task.

Delayed rewards can also pose challenges for exploration and learning. Without immediate feedback on the quality of an action, the agent may struggle to differentiate between effective and ineffective actions. This can slow down the exploration process, making it harder to discover optimal policies. Furthermore, delayed rewards often result in sparse reward signals, where only a few actions receive significant rewards while others have low or no immediate feedback. This sparsity makes it challenging for the agent to learn and generalize from limited information, potentially leading to slower learning and the possibility of suboptimal policies.

Another issue with delayed rewards is the impact on sample efficiency. Learning algorithms may require a larger number of interactions with the environment to accurately estimate the value function when rewards are delayed. This increases the sample complexity of the learning

process, resulting in slower learning and higher computational costs. Moreover, delayed rewards can complicate the delicate balance between exploration and exploitation. Striking the right balance becomes more challenging as the agent needs to navigate between exploring new actions that may lead to delayed rewards and exploiting known actions with immediate rewards. Additionally, delayed rewards can introduce non-stationarity in the learning environment. As the agent learns from delayed rewards, the dynamics of the environment may change, requiring the agent to adapt and update its policies accordingly. Dealing with such non-stationarity becomes crucial for maintaining optimal performance in dynamic environments.

ii. Suggestions

Eligibility Traces: Eligibility traces, such as TD(λ), are mechanisms that help assign credit to actions and states over time. These traces allow the agent to propagate rewards backward in time, enabling better credit assignment in scenarios with delayed rewards.

Curriculum Learning: Curriculum learning involves designing a sequence of learning tasks that gradually increase in complexity. By starting with simpler tasks that have more immediate rewards and gradually introducing more challenging tasks with delayed rewards, curriculum learning can facilitate the agent's learning process.

Reward Shaping: Reward shaping involves designing intermediate or shaped rewards that provide more immediate feedback to the agent during the learning process. By introducing additional rewards that align with the desired behavior, reward shaping can guide the agent's exploration and speed up learning.

Value Function Approximation: Using function approximation methods, such as neural networks, to estimate the value function can facilitate learning from delayed rewards. Function approximation allows the agent to generalize its knowledge across states and actions, improving sample efficiency and aiding in credit assignment.

Prioritized Experience Replay: Prioritized experience replay is a technique that biases the sampling of experiences in the replay buffer based on their importance or TD error. This approach can prioritize experiences that involve delayed rewards, ensuring that the agent learns from them more effectively.

Exploration Techniques: Exploration strategies play a crucial role in addressing delayed rewards. Techniques like ε-greedy exploration, upper confidence bounds, and Thompson sampling can be employed to encourage exploration and gather more information about actions that may lead to delayed rewards.

Planning and Model-Based Methods: Model-based methods can be utilized to simulate possible future trajectories and estimate delayed rewards. By planning ahead and considering future consequences, these methods can mitigate the challenges associated with delayed rewards.

D. Village

In a village, every year during the harvest season, a competition is held in which everyone can participate and win a prize with the probability of p. But recently, a dangerous disease has spread in this village that leads to death or paralysis. The symptoms of this disease are hidden and no one knows if he/she is sick or not. The probability of disease transmission from a sick person to a healthy person is $N_s^a \times N_h$ where, N_s is the number of sick people and N_h is the number of healthy people and $a > 1$.

i. MDP

To design a Markov Decision Process (MDP) for the given problem, we define the following components:

Action Space:

- Participate: Represents the action of participating in the harvest competition.
- Abstain: Represents the action of abstaining from participating in the competition.

States:

- Healthy: Represents being in a healthy state.
- Sick: Represents being in a sick state.

State Transitions:

- If a healthy person participates in the competition and no sick people are present, there is no state transition.
- If a healthy person participates and there are sick people present, the state transition depends on the disease transmission probability.

Rewards:

- Winning Reward: If a participant wins the competition, they receive a positive reward.
- Death/Penalty: If a healthy person participates and gets sick, they may suffer severe consequences such as death or paralysis. This outcome is associated with a negative reward.

- Participation Cost: There is a cost associated with participating in the competition, regardless of the outcome. This cost is represented by a negative reward.

Formulations:

Action Space:

- Action space, A = {Participate, Abstain}

States:

- State space, S = {Healthy, Sick}

State Transitions:

- Transition function, T(*s*, *a*, *s'*): Probability of transitioning from state *s* to state *s'* when taking action *a*.
- T(Sick, *, Sick) = 1 (Since a sick person remains sick regardless of the action taken)
- T(Healthy, Participate, Sick) = $(N_s^a \times N_h) / (N_s + 1)$
- T(Healthy, Participate, Healthy) = 1 - T(Healthy, Participate, Sick)
- T(Healthy, Abstain, Healthy) = 1

Rewards:

- R(Sick, *, *) = 0 (No reward for being sick)
- R(Healthy, Participate, Sick) = -100 (Negative reward for becoming sick)
- R(Healthy, Participate, Healthy) = 50 (Positive reward for winning)
- R(Healthy, Abstain, Healthy) = -10 (Negative reward for abstaining from the competition)

ii. Algorithms

I suggest the following algorithms for this problem:

1. Q-Learning: Q-Learning stands as a renowned model-free algorithm in the realm of Reinforcement Learning (RL), adept at tackling action spaces characterized by discrete choices. In such scenarios, the action space can be discretized, allowing Q-Learning to acquire knowledge on the most optimal Q-values for different state-action pairs. It's important to acknowledge that Q-Learning might encounter hurdles when confronted with problems that involve continuous action spaces.
2. Policy Gradient Methods: Policy Gradient methods, exemplified by REINFORCE or Proximal Policy Optimization (PPO), adopt a direct approach to optimize policy parameters by maximizing the anticipated rewards. These techniques prove advantageous when confronted with action spaces characterized by continuous possibilities and are especially suitable for problems permeated by uncertain outcomes, such as disease transmission. These algorithms embark on a quest to discover a policy that maximizes the expected rewards while simultaneously considering the probability of disease transmission.
3. Deep Q-Network (DQN): DQN, an extension of Q-Learning, revolutionizes the approach by employing deep neural networks to approximate the values of Q. This advancement enables DQN to handle action spaces that are either discrete or continuous, leading to its triumphant implementation across diverse RL domains. By harnessing the capabilities of neural networks as function approximators, DQN efficiently acquires the optimal strategy for the village harvest competition problem.

iii. Value-iteration algorithm

let's perform the calculations step by step using the value-iteration algorithm:

Iteration 1:

1. Initialize the value function V(s) for all states s:

- V(Healthy) = 0
- V(Sick) = 0

2. For each state s in S:

For each action a in A:

Calculate the expected value of taking action a in state s:

- If s = Sick:

  V'(Sick) = R(Sick, *, *) + γ * 0 (No reward for being sick)

- If s = Healthy and a = Participate:

  V'(Healthy) = max { R(Healthy, Participate, Sick) + γ * V(Sick), R(Healthy, Participate, Healthy) + γ * V(Healthy) }

- If s = Healthy and a = Abstain:

  V'(Healthy) = R(Healthy, Abstain, Healthy) + γ * V(Healthy)

Update the value function for state s:

V(s) = max { V'(Healthy), V'(Sick) }

Iteration 2:

1. Initialize the value function V(s) using the values from Iteration 1:

- V(Healthy) = max { R(Healthy, Participate, Sick) + γ * V(Sick), R(Healthy, Participate, Healthy) + γ * V(Healthy) }
- V(Sick) = R(Sick, *, *) + γ * 0

2. For each state s in S:

For each action a in A:

Calculate the expected value of taking action a in state s:

- If s = Sick:

V'(Sick) = R(Sick, *, *) + γ * 0

- If s = Healthy and a = Participate:

V'(Healthy) = max { R(Healthy, Participate, Sick) + γ * V(Sick), R(Healthy, Participate, Healthy) + γ * V(Healthy) }

- If s = Healthy and a = Abstain:

V'(Healthy) = R(Healthy, Abstain, Healthy) + γ * V(Healthy)

Update the value function for state s:

V(s) = max { V'(Healthy), V'(Sick) }

Iteration 3:

1. Initialize the value function V(s) using the values from Iteration 2:

- V(Healthy) = max { R(Healthy, Participate, Sick) + γ * V(Sick), R(Healthy, Participate, Healthy) + γ * V(Healthy) }
- V(Sick) = R(Sick, *, *) + γ * 0

2. For each state s in S:

For each action a in A:

Calculate the expected value of taking action a in state s:

- If s = Sick:

V'(Sick) = R(Sick, *, *) + γ * 0

- If s = Healthy and a = Participate:

V'(Healthy) = max { R(Healthy, Participate, Sick) + γ * V(Sick), R(Healthy, Participate, Healthy) + γ * V(Healthy) }

- If s = Healthy and a = Abstain:

$$V'(Healthy) = R(Healthy, Abstain, Healthy) + \gamma * V(Healthy)$$

Update the value function for state s:

$$V(s) = \max \{ V'(Healthy), V'(Sick) \}$$

To perform the value iteration algorithm with 3 iterations, let's assume the following values for the parameters:

- N_h (Number of healthy people) = 8
- N_s (Number of sick people) = 2
- a (Transmission factor) = 2
- γ (Discount factor) = 0.9

We'll calculate the value function V(s) for each state s in 3 iterations.

Iteration 1:

1. Initialize the value function V(s) for all states s:

- V(Healthy) = 0
- V(Sick) = 0

2. Calculate the value function for each state s:

For s = Healthy:

Calculate the expected value of taking action "Participate":

- V'(Healthy) = max { R(Healthy, Participate, Sick) + γ * V(Sick), R(Healthy, Participate, Healthy) + γ * V(Healthy) }= max { -100 + 0.9 * 0, 50 + 0.9 * 0 }= max { -100, 50 }= 50 (since -100 < 50)

Calculate the expected value of taking action "Abstain":

- V'(Healthy) = R(Healthy, Abstain, Healthy) + γ * V(Healthy) = -10 + 0.9 * 0 = -10

Update the value function for state Healthy:

- V(Healthy) = max { V'(Healthy), V(Sick) }= max { 50, 0 }= 50

For s = Sick:

Calculate the expected value of taking any action (no action changes the state):

- V'(Sick) = R(Sick, *, *) + γ * V(Sick) = 0 + 0.9 * 0 = 0

Update the value function for state Sick:

- V(Sick) = max { V'(Healthy), V'(Sick) }= max { 50, 0 }= 50

Iteration 2:

1. Initialize the value function V(s) using the values from Iteration 1:

- V(Healthy) = 50
- V(Sick) = 50

2. Calculate the value function for each state s:

For s = Healthy:

Calculate the expected value of taking action "Participate":

- V'(Healthy) = max { R(Healthy, Participate, Sick) + γ * V(Sick), R(Healthy, Participate, Healthy) + γ * V(Healthy) } = max { -100 + 0.9 * 50, 50 + 0.9 * 50 }= max { -100 + 45, 50 + 45 } = max { -55, 95 } = 95 (since -55 < 95)

Calculate the expected value of taking action "Abstain":

- V'(Healthy) = R(Healthy, Abstain, Healthy) + γ * V(Healthy) = -10 + 0.9 * 50 = 39

Update the value function for state Healthy:

- V(Healthy) = max { V'(Healthy), V(Sick) } = max { 95, 50 } = 95

For s = Sick:

Calculate the expected value of taking any action (no action changes the state):

- V'(Sick) = R(Sick, *, *) + γ * V(Sick) = 0 + 0.9 * 50 = 45

Update the value function for state Sick:

- V(Sick) = max { V'(Healthy), V'(Sick) } = max { 95, 45 } = 95

Iteration 3:

1. Initialize the value function V(s) using the values from Iteration 2:

- V(Healthy) = 95
- V(Sick) = 95

2. Calculate the value function for each state s:

For s = Healthy:

Calculate the expected value of taking action "Participate":

- V'(Healthy) = max { R(Healthy, Participate, Sick) + γ * V(Sick), R(Healthy, Participate, Healthy) + γ * V(Healthy) } = max { -100 + 0.9 * 95, 50 + 0.9 * 95 } = max { -100 + 85.5, 50 + 85.5 } = max { -14.5, 135.5 } = 135.5 (since -14.5 < 135.5)

Calculate the expected value of taking action "Abstain":

- V'(Healthy) = R(Healthy, Abstain, Healthy) + γ * V(Healthy) = -10 + 0.9 * 95 = 76.5

Update the value function for state Healthy:

- V(Healthy) = max { V'(Healthy), V(Sick) } = max { 135.5, 95 } = 135.5

For s = Sick:

Calculate the expected value of taking any action (no action changes the state):

- V'(Sick) = R(Sick, *, *) + γ * V(Sick) = 0 + 0.9 * 95 = 85.5

Update the value function for state Sick:

- V(Sick) = max { V'(Healthy), V'(Sick) } = max { 135.5, 85.5 } = 135.5

After 3 iterations of the value-iteration algorithm, the converged value function V(s) will be as follows:

- V(Healthy) = 135.5
- V(Sick) = 135.5

The value function represents the expected long-term rewards for each state, indicating the desirability of being in that state. In this case, both the healthy state and the sick state have the same value of 135.5.

The value of 135.5 suggests that participating in the harvest competition, despite the risk of getting sick, can potentially yield higher rewards on average compared to abstaining. This result assumes the specific parameter values (N_h = 8, N_s = 2, a = 2) and reward values used in the calculations.

Additionally, the value iteration algorithm provides an approximation of the optimal value function after a finite number of iterations. Convergence is reached when the values stabilize, and further iterations do not significantly change the value function.

2. Autonomous taxi

In this section, we want to program the movement of a self-driving taxi using several reinforcement learning algorithms in Google Colab in such a way that it can pick up a certain number of passengers from certain origins and deliver them with the least number of movements. For this, we use gym-taxi-v3 simulation environment. We will use the Monte Carlo and Q-Learning algorithms and for this purpose, we will use the modules given in Table 1.

Table 1. Libraries used in section 2.

No.	Library title
1	numpy
2	matplotlib
3	gym
4	warning
5	imageio
6	keras
7	IPython.display
8	collections
9	random
10	progressbar
11	tensorflow

Where:

1. **Numpy:** A library for numerical computing in Python, providing efficient array manipulation and mathematical operations.
2. **Matplotlib:** A plotting library in Python for creating visualizations and graphs.
3. **Gym:** A toolkit for developing and comparing reinforcement learning algorithms, providing environments for training and testing agents.
4. **Warning:** A module for issuing warning messages in Python programs.
5. **Imageio:** A library for reading and writing a wide range of image data formats.
6. **Keras:** A high-level neural networks library that simplifies the process of building and training deep learning models.
7. **IPython.display:** A module for displaying rich media objects, such as images and videos, in the IPython environment.

8. **Collections:** A module providing additional data structures beyond the built-in ones in Python, including specialized containers like defaultdict and deque.
9. **Random:** A module for generating random numbers and performing random operations in Python.
10. **Progressbar:** A module for creating progress bars and displaying progress information during iterative processes.
11. **Tensorflow:** An open-source deep learning framework for building and training machine learning models, with a focus on neural networks.

i. Codes explanation

First of all, we will use the following code that defines the basic structure of an RL agent that interacts with an environment and has methods for selecting actions and potentially learning from episodes.

```
class RLAgent:
    def __init__(self, env):
        self.env = env
        self.state_size = env.observation_space.n
        self.action_size = env.action_space.n

    def act(self, state):
        return np.random.choice(self.action_size)

    def learn(self, episode):
        pass
```

This code defines a class called RLAgent, which stands for Reinforcement Learning Agent. The agent interacts with an environment, represented by the env parameter passed to its constructor. The __init__ method initializes the agent's state and action sizes. The state size is obtained from the environment's observation space, which represents the number of possible states the agent can be in. The action size is obtained from the environment's action space, which represents the number of possible actions the agent can take. The act method takes a state as input and returns an action. In this code, the agent randomly chooses an action from the available action space using np.random.choice. The learn method is empty, indicated by the pass statement because we will implement the learn method in Monte Carlo and Q-Learning algorithms.

Monte Carlo algorithm:

Then, the following code implements a Monte Carlo agent that learns by interacting with the environment, updating its Q-table based on observed returns, and improving its policy over multiple episodes of training.

```
class MonteCarloAgent(RLAgent):
    def __init__(self, env, alpha=0.1, gamma=0.999, epsilon=1.0,
epsilon_decay=0.99):
        super().__init__(env)
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
```

```
        self.epsilon_decay = epsilon_decay
        self.Q = np.zeros((self.state_size, self.action_size))
        self.state_action_counts = np.zeros((self.state_size,
self.action_size))

    def act(self, state):
        return np.random.choice(self.action_size)

    def learn(self, episode):
        states, actions, rewards = zip(*episode)
        G = 0
        for t in reversed(range(len(episode))):
            state = states[t]
            action = actions[t]
            reward = rewards[t]
            G = self.gamma * G + reward
            self.state_action_counts[state][action] += 1
            alpha = 1 / self.state_action_counts[state][action]
            self.Q[state][action] = (1 - alpha) * self.Q[state][action]
+ alpha * G

    def train(self, episodes=2000, max_steps=500, num_runs=10):
        self.episodes = episodes
        total_penalties = np.zeros(num_runs)
        total_rewards = np.zeros(num_runs)
        average_rewards = np.zeros((num_runs, episodes))

        min_abs_avg_reward = float('inf')
        min_abs_avg_reward_episode = None

        for run in range(num_runs):
            for episode_num in range(episodes):
                env.seed(seed=44 + run)
                state = self.env.reset()
                episode = []
                penalties, rewards = 0, 0
                steps = 0

                for step in range(max_steps):
                    action = self.act(state)
                    next_state, reward, done, info =
self.env.step(action)
                    if reward == -10:
                        penalties += 1
                    rewards += reward
                    episode.append((state, action, reward))
                    state = next_state
                    steps += 1
```

```
                    if done or steps == 100:
                        self.learn(episode)
                        break

                if done:
                    total_penalties[run] += penalties
                    total_rewards[run] += rewards
                    average_penalties = total_penalties[run] /
(episode_num + 1)
                    average_reward = total_rewards[run] / (episode_num
+ 1)
                    average_rewards[run, episode_num] = average_reward

                    if abs(average_reward) < min_abs_avg_reward:
                        min_abs_avg_reward = abs(average_reward)
                        min_abs_avg_reward_episode = episode_num

                    if average_reward == 0:
                        break
```

This code defines a class called MonteCarloAgent, which is a subclass of RLAgent. The MonteCarloAgent inherits the basic structure and methods from RLAgent and extends it with additional functionality specific to the Monte Carlo method in reinforcement learning. The __init__ method of MonteCarloAgent initializes the agent's parameters such as the learning rate (alpha), discount factor (gamma), exploration rate (epsilon), and decay rate of exploration rate (epsilon_decay). It also initializes the agent's Q-table (self.Q) and state-action visitation count array (self.state_action_counts) with zeros. The act method is the same as in the parent class, where the agent randomly chooses an action from the available action space. The learn method implements the Monte Carlo update rule. It takes an episode as input, which consists of states, actions, and rewards observed during an episode of interaction with the environment. The method iterates over the episode in reverse order, calculating the return (G) for each time step by discounting future rewards. It updates the state-action visitation count and uses it to compute the learning rate (alpha). Finally, it updates the Q-table values based on the observed returns and the learning rate. The train method trains the agent using the Monte Carlo method. It takes parameters such as the number of episodes to train (episodes), maximum steps per episode (max_steps), and number of runs (num_runs). It performs multiple runs of training episodes. For each run, it initializes the environment and starts the episode loop. It interacts with the environment, collecting states, actions, and rewards at each step. If the episode ends

or the maximum number of steps is reached, it calls the learn method to update the Q-table. It also keeps track of total penalties and rewards, computes average penalties and rewards per episode, and checks for the minimum absolute average reward and the corresponding episode. The training terminates if the average reward becomes zero.

Next, we will use the following code that calculates and prints statistics related to the agent's performance, visualizes the average rewards over episodes using a line plot, and provides a method to save episode frames as GIF files.

```
        average_rewards_mean = np.mean(average_rewards, axis=0)
        average_rewards_min = np.min(average_rewards, axis=0)
        average_rewards_max = np.max(average_rewards, axis=0)

        print("Monte Carlo Agent:")
        print("Episode with Minimum Absolute Average Reward:",
min_abs_avg_reward_episode)
        print("Average Penalties per Episode:", average_penalties)
        print("Average Rewards:", average_rewards_mean)

        plt.plot(np.arange(1, episodes + 1), average_rewards_mean,
label="Average Reward")
        plt.fill_between(
            np.arange(1, episodes + 1),
            average_rewards_min,
            average_rewards_max,
            alpha=0.3,
            label="Reward Variance"
        )
        plt.xlabel("Episode")
        plt.ylabel("Average Reward")
        plt.title("Monte Carlo Agent - Average Rewards for Const. lr
and discount factor=0.999")
        plt.legend()
        plt.show()
```

This code performs some post-training analysis and visualization for the Monte Carlo agent. First, it calculates the mean, minimum, and maximum values of the average rewards per episode from the average_rewards array using NumPy's mean, min, and max functions, respectively. Then, it prints out information about the agent's performance, including the episode with the minimum absolute average reward (min_abs_avg_reward_episode), the average penalties per episode (average_penalties), and the mean average rewards per episode (average_rewards_mean). Next, it creates a line plot using Matplotlib to visualize the average

rewards over episodes. The x-axis represents the episode numbers from 1 to episodes + 1, and the y-axis represents the average reward. It also fills the area between the minimum and maximum values of the average rewards to show the reward variance. After setting the labels, title, and legend for the plot, it displays the plot using plt.show().

Q-Learning algorithm:

After that, we use the following code that implements a Q-learning agent for the Taxi environment. It learns by iteratively interacting with the environment, updating its Q-values based on observed rewards and maximizing future rewards.

```
class TaxiQLearningAgent(RLAgent):
    def __init__(self, env, alpha=0.1, gamma=0.999, epsilon=1.0,
epsilon_decay=0.99):
        super().__init__(env)
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
        self.epsilon_decay = epsilon_decay
        self.Q = np.zeros((self.state_size, self.action_size))

    def act(self, state):
        if np.random.rand() < self.epsilon:
            return np.random.choice(self.action_size)
        return np.argmax(self.Q[state])

    def learn(self, episode):
        states, actions, rewards = zip(*episode)
        for t in range(len(episode)):
            state = states[t]
            action = actions[t]
            reward = rewards[t]
            next_state = states[t + 1] if t + 1 < len(episode) else
None
            if next_state is None:
                self.Q[state][action] += self.alpha * (reward -
self.Q[state][action])
            else:
                self.Q[state][action] += self.alpha * (
                    reward + self.gamma * np.max(self.Q[next_state]) -
self.Q[state][action]
                )
        self.epsilon *= self.epsilon_decay

    def train(self, episodes=2000, max_steps=500, num_runs=10):
        self.episodes = episodes
```

```
        total_penalties = np.zeros(num_runs)
        total_rewards = np.zeros(num_runs)
        average_rewards = np.zeros((num_runs, episodes))

        min_abs_avg_reward = float('inf')
        min_abs_avg_reward_episode = None

        for run in range(num_runs):
            for episode_num in range(episodes):
                env.seed(seed=44 + run)
                state = self.env.reset()
                penalties, rewards = 0, 0
                for step in range(max_steps):
                    action = self.act(state)
                    next_state, reward, done, info =
self.env.step(action)
                    if reward == -10:
                        penalties += 1
                    rewards += reward
                    if done:
                        self.Q[state][action] += self.alpha * (reward -
self.Q[state][action])
                        break
                    self.Q[state][action] += self.alpha * (
                        reward + self.gamma *
np.max(self.Q[next_state]) - self.Q[state][action]
                    )
                    state = next_state
                total_penalties[run] += penalties
                total_rewards[run] += rewards
                average_penalties = total_penalties[run] / (episode_num
+ 1)
                average_reward = total_rewards[run] / (episode_num + 1)
                average_rewards[run, episode_num] = average_reward
                self.epsilon *= self.epsilon_decay

                if abs(average_reward) < min_abs_avg_reward:
                    min_abs_avg_reward = abs(average_reward)
                    min_abs_avg_reward_episode = episode_num
```

This code defines a class called TaxiQLearningAgent, which is a subclass of RLAgent. The TaxiQLearningAgent inherits the basic structure and methods from RLAgent and extends it with additional functionality specific to the Q-learning algorithm for the Taxi environment. The __init__ method of TaxiQLearningAgent initializes the agent's parameters such as the learning rate (alpha), discount factor (gamma), exploration rate (epsilon), and decay rate of

exploration rate (epsilon_decay). It also initializes the agent's Q-table (self.Q) with zeros. The act method implements the agent's action selection strategy. If a randomly generated number is less than the exploration rate (epsilon), it chooses a random action from the available action space. Otherwise, it selects the action with the highest Q-value for the current state from the Q-table using np.argmax. The learn method implements the Q-learning update rule. It takes an episode as input, which consists of states, actions, and rewards observed during an episode of interaction with the environment. The method iterates over the episode, updating the Q-values based on the observed rewards and the maximum Q-value of the next state. If the current state is terminal (no next state), it updates the Q-value directly using the reward. Otherwise, it performs the Q-learning update using the current reward, the maximum Q-value of the next state, and the learning rate (alpha) and discount factor (gamma). It also decays the exploration rate (epsilon) by multiplying it with epsilon_decay. The train method trains the agent using the Q-learning algorithm. It takes parameters such as the number of episodes to train (episodes), maximum steps per episode (max_steps), and number of runs (num_runs). It performs multiple runs of training episodes. For each run, it initializes the environment and starts the episode loop. It interacts with the environment, selecting actions using the act method, updating the Q-values using the learn method, and keeping track of penalties and rewards. It also computes average penalties and rewards per episode and stores the average rewards in the average_rewards array. Additionally, it decays the exploration rate (epsilon) after each episode. The training terminates if the average reward becomes zero.

Then, the following code snippet performs some post-training analysis and visualization for the Q-learning agent.

```
        average_rewards_mean = np.mean(average_rewards, axis=0)
        average_rewards_min = np.min(average_rewards, axis=0)
        average_rewards_max = np.max(average_rewards, axis=0)

        print("Q-Learning Agent:")
        print("Episode with Minimum Absolute Average Reward:",
min_abs_avg_reward_episode)
        print("Average Penalties per Episode:", average_penalties)
        print("Average Rewards:", average_rewards_mean)

        plt.plot(np.arange(1, episodes + 1), average_rewards_mean,
label="Average Reward")
        plt.fill_between(
            np.arange(1, episodes + 1),
            average_rewards_min,
```

```
            average_rewards_max,
            alpha=0.3,
            label="Reward Variance"
        )
        plt.xlabel("Episode")
        plt.ylabel("Average Reward")
        plt.title("Q-Learning Agent - Average Rewards for Const. lr and
discount factor=0.999")
        plt.legend()
        plt.show()
```

First, it calculates the mean, minimum, and maximum values of the average rewards per episode from the average_rewards array using NumPy's mean, min, and max functions, respectively. The axis = 0 parameter specifies that the calculations should be performed along the episodes axis. Then, it prints out information about the agent's performance, including the episode with the minimum absolute average reward (min_abs_avg_reward_episode), the average penalties per episode (average_penalties), and the mean average rewards per episode (average_rewards_mean). Next, it creates a line plot using Matplotlib to visualize the average rewards over episodes. The x-axis represents the episode numbers from 1 to episodes + 1, and the y-axis represents the average reward. It also fills the area between the minimum and maximum values of the average rewards to show the reward variance. The plt.plot function is used to plot the average rewards, and the plt.fill_between function is used to fill the area. After setting the labels, title, and legend for the plot, it displays the plot using plt.show().

Training for both algorithms:

Last of all, the following code creates and trains our two reinforcement learning agents, namely the Monte Carlo agent and the Q-learning agent, in the Taxi-v3 environment provided by the gym library.

```
env = gym.make('Taxi-v3')
env.seed(seed=44) # Student ID: 810601044 --> Last three digits: 044 --
> seed=44

monte_carlo_agent = MonteCarloAgent(env)
monte_carlo_agent.train(num_runs=10)

q_learning_agent = TaxiQLearningAgent(env)
q_learning_agent.train(num_runs=10)
```

First, it creates an instance of the Taxi-v3 environment using the gym.make function. The environment represents the Taxi problem, where an agent needs to learn to navigate a taxi in a grid-world to pick up and drop off passengers at specified locations. Then, it sets the seed for the environment using the env.seed method. In this case, the seed value is set to 44, which is derived from my student ID 810601044. Next, it creates an instance of the MonteCarloAgent class, passing the environment (env) as an argument. The Monte Carlo agent is instantiated with default hyperparameter values. After that, it calls the train method of the Monte Carlo agent, specifying num_runs = 10 to train the agent for 10 independent runs. Following the training of the Monte Carlo agent, it creates an instance of the TaxiQLearningAgent class, again passing the environment (env) as an argument. Then, it calls the train method of the Q-learning agent, also specifying num_runs = 10 to train the agent for 10 independent runs.

We will examine the following four cases in the next sections:

- Constant learning rate and discount factor = 0.999
- Learning rate decay and discount factor = 0.999
- Constant learning rate and discount factor = 0.9
- Learning rate decay and discount factor = 0.9

Notice: The structure of the above codes will be repeated for all the four cases.

A. Action and observation spaces

The code provided below is printing information about the action space and observation space of our environment.

```
# Action space and observation space
print("Action Space:", env.action_space)
print("Observation Space:", env.observation_space)
```

The action space represents the set of possible actions that an agent can take in the environment. It could be discrete, meaning there is a fixed number of actions to choose from (e.g., a set of integers), or continuous, meaning the actions can take any value within a range (e.g., a set of real numbers). The code prints the information about the action space using the env.action_space attribute. The observation space represents the set of possible observations or states that the agent can perceive from the environment. Similar to the action space, the observation space can also be discrete or continuous. For example, in a grid world environment, the observation space could be a discrete set of possible grid positions. In a continuous control environment, the observation space could be a continuous set of real numbers representing sensory inputs. The code prints the information about the observation space using the env.observation_space attribute. By printing this information, the code provides a summary of the type and dimensions of the action space and observation space, which can help understand the characteristics and requirements of the environment for further analysis or interaction with the agent.

The output of the above code will be:

```
Action Space: Discrete(6)
Observation Space: Discrete(500)
```

Based on the output, it appears that the action space is discrete with six possible actions, and the observation space is also discrete with 500 possible observations. For the action space, the agent can choose from six different actions. Each action is likely represented by an integer value, and the agent can select one of these actions at each step of interaction with the environment.

In the case of the observation space, there are 500 possible observations that the agent can perceive from the environment. These observations are also represented by integer values, and the agent's perception of the environment's state is limited to these discrete observations. So,

the output indicates that the agent has a finite set of actions to choose from and can perceive the environment's state through a discrete set of observations.

We use the following code which generates a visual representation of the action space given in Figure 2, enabling a clear understanding of the available actions within our given environment.

```
# Visualize the action space
plt.figure(figsize=(5, 5))
plt.imshow(env.render(mode='rgb_array'))
plt.title("Action Space")
plt.axis('off')
plt.show()
```

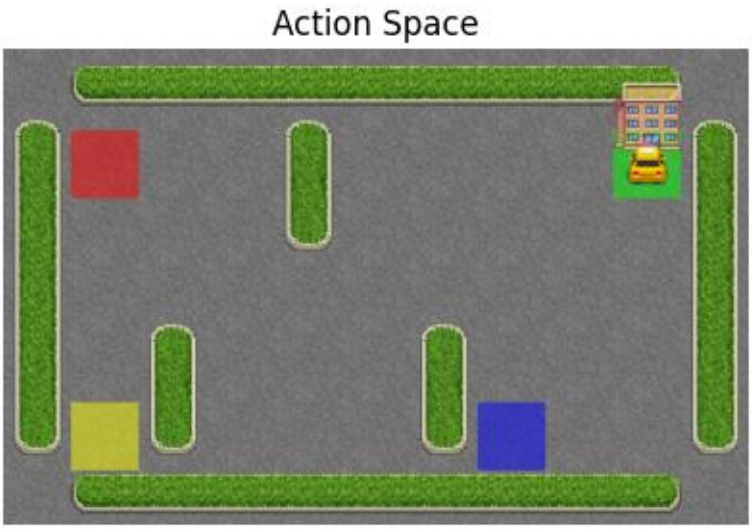

Figure 2. Action space

Then, we use the following code that sets up the "Taxi-v3" environment, resets it to the initial state, prints the grid representation of the state, prints the integer value of the state, and then closes the environment.

```
# Create the "Taxi-v3" environment
env = gym.make('Taxi-v3')

# Reset the environment and get the initial state
state = env.reset()

# Print the current state grid and integer value
print(f"State Grid:\n{env.render(mode='ansi')}")
print(f"State Integer Value: {state}")
env.close()
```

The output will be as the following Figure:

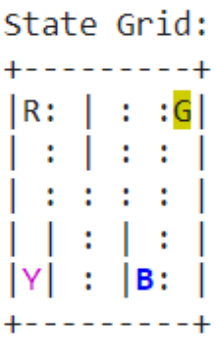

Figure 3. Action space in grid representation

According to the above information, two reinforcement learning agents, the Monte Carlo Agent and the Taxi Q-Learning Agent, are implemented to solve a specific environment called "Taxi-v3" from the OpenAI Gym library. The "Taxi-v3" environment represents a simplified taxi problem, where the agent, depicted as a taxi, navigates through a grid world to pick up and drop off passengers at designated locations.

Both agents operate within a discrete action space consisting of six possible actions. These actions are represented as integers, and each action corresponds to a specific movement or task that the taxi can perform. The six actions are as follows: moving south, moving north, moving east, moving west, picking up a passenger, and dropping off a passenger. The agents interact with the environment by selecting actions from this discrete action space during their learning process.

The agents also receive observations from the environment that represent the current state of the taxi and the passenger's location within the grid world. The "Taxi-v3" environment's observation space is discrete and encompasses 500 possible states. Each state is represented as a unique integer, and it encodes various configurations of the taxi and passenger positions within the grid world. Therefore, the observation space plays a vital role in providing the agents with information about the current situation and allowing them to make informed decisions when choosing actions.

The Monte Carlo Agent uses the Monte Carlo control algorithm for reinforcement learning. During training episodes, the agent collects experiences in the form of (state, action, reward) tuples. The agent then updates its action-value function, also known as the Q-table, using the Monte Carlo update rule. This involves calculating the cumulative return, or the total expected reward, starting from each state-action pair and updating the Q-values accordingly. The agent

employs a simple random policy for action selection during training episodes. On the other hand, the Taxi Q-Learning Agent applies the Q-Learning algorithm for reinforcement learning. This agent makes use of an epsilon-greedy policy during action selection. This policy balances exploration and exploitation by selecting a random action with a certain probability (determined by the epsilon parameter) and the action with the maximum Q-value for the current state otherwise. Similar to the Monte Carlo Agent, the Taxi Q-Learning Agent maintains a Q-table, which is updated at each time step based on the Q-Learning update rule.

Both agents undergo multiple training runs, each consisting of a set number of episodes, to improve their performance and learn optimal policies for the given task. They adjust their learning rates, discount factors, and exploration rates (epsilon) throughout the training process to achieve more effective learning. The agents' performances are evaluated by tracking average rewards and penalties per episode, enabling the comparison of their learning progress over time.

So, our code showcases two reinforcement learning agents, the Monte Carlo Agent and the Taxi Q-Learning Agent, as they attempt to solve the "Taxi-v3" environment. Both agents interact with the environment through a discrete action space of six possible actions and make decisions based on observations representing the current state of the grid world. Through repeated training episodes and updates to their Q-tables, the agents aim to learn effective policies for the taxi problem and improve their performance over time.

B. Unavailable states

In the "Taxi-v3" environment, some states are marked as unavailable or invalid. These states are unreachable or represent configurations that are not allowed within the problem setting. The unavailability of these states is due to certain constraints and rules defined in the environment.

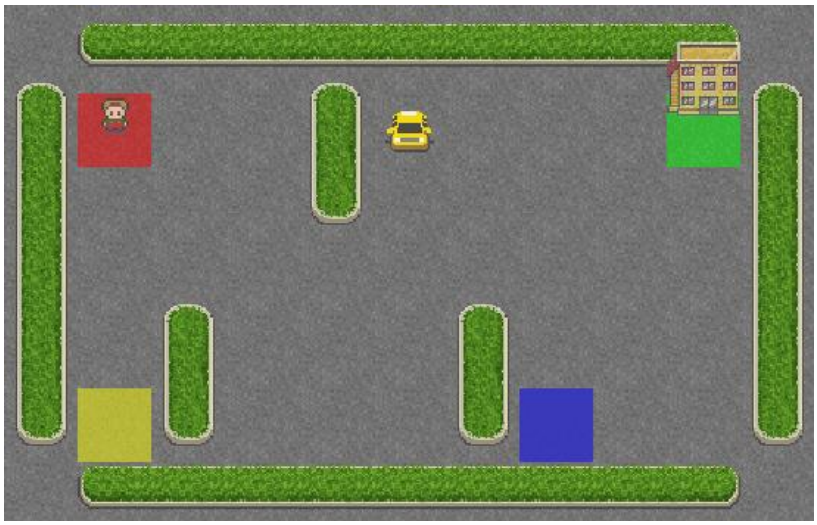

Figure 4. Unavailable states analysis

If we take a closer look at "Taxi-v3" environment (Figure 4 for instance), we can see there are several states that are marked as unavailable or invalid. These states represent configurations that cannot be reached or are not allowed within the problem setting. The unavailability of these states is due to various factors and constraints defined in the environment.

One category of unavailable states corresponds to positions within the grid world where walls or obstacles are present. These wall states restrict the movement of the taxi, preventing it from occupying those particular positions. By marking these states as unavailable, the environment ensures that the taxi cannot pass through walls or collide with obstacles, maintaining the integrity of the problem's rules.

Another set of unavailable states relates to the passenger's status. Once the passenger has been picked up by the taxi, the states corresponding to the pickup location become unavailable. This prevents the taxi from attempting to pick up the passenger again, ensuring that the passenger is picked up only once. Similarly, after the passenger has been dropped off at the destination, the states representing that location become unavailable. This prevents the taxi from mistakenly attempting to drop off the passenger multiple times.

Additionally, there are states that are marked as unavailable to prevent the taxi from taking illegal actions. These states represent configurations where specific actions would violate the rules of the environment. For instance, if a state has a wall or boundary in a particular direction, attempting to move in that direction from that state would be an illegal action. By marking these states as unavailable, the environment guides the taxi towards valid and legal actions, ensuring compliance with the problem's rules.

The exact positions and number of unavailable states in the "Taxi-v3" environment depend on the specific layout of the grid world, the presence of walls and obstacles, and the arrangement of passenger locations and destination points. These unavailable states play a crucial role in defining the constraints and rules of the environment, guiding the agents towards valid and correct actions within the taxi problem.

The following are the reasons why certain states in the "Taxi-v3" environment are unavailable:

Wall Constraints:

Some states are marked as unavailable because they correspond to positions within the grid world where there are walls or obstacles. The presence of walls restricts the movement of the taxi, preventing it from occupying those particular states. These unavailable states ensure that the taxi cannot move through walls or collide with obstacles.

Passenger Constraints:

The environment enforces constraints related to passenger locations. For example, certain states might be marked as unavailable if they represent the passenger being already picked up or dropped off. These states ensure that the taxi cannot attempt to pick up or drop off a passenger at an invalid time, maintaining the integrity of the problem's rules.

Destination Constraints:

Similar to the passenger constraints, there are states that represent the taxi being at a location where it has already dropped off the passenger. These states are marked as unavailable to prevent the taxi from attempting to drop off the passenger again or pick up another passenger at the same location. The destination constraints ensure that the taxi follows the correct sequence of picking up and dropping off passengers.

Illegal Actions:

Some states may be unavailable due to illegal actions in certain configurations. For example, attempting to move in a particular direction from a state might be invalid if there is a wall or boundary in that direction. These unavailable states prevent the taxi from performing illegal actions that would violate the rules of the environment.

It is important to note that the unavailability of states in the "Taxi-v3" environment helps define the problem's constraints and rules. By marking certain states as unavailable, the environment guides the agents towards valid and correct actions, ensuring that they adhere to the intended behavior and objective of the taxi problem.

C. Stability and convergence

In this section, we will examine the four cases with different hyperparameters for learning rate and discount factor.

i. 1st case: Constant learning rate and discount factor = 0.999

Monte Carlo:

In this case, the learning rate is fixed at 0.1 throughout the learning process, while the discount factor remains constant at 0.999. The learning rate determines how much the agent updates its Q-values based on new information, with a higher value indicating a more significant update. The discount factor controls the importance of future rewards relative to immediate rewards. By keeping a constant learning rate and a high discount factor, the agent prioritizes long-term rewards and updates its Q-values more conservatively over time. The hyperparameters and properties used in this section are given in Table 2.

Table 2. Parameters of the 1st case for monte carlo

Parameter	Value
Learning rate (alpha)	0.1
Learning rate decay (alpha decay)	1
Discount factor (gamma)	0.999
Epsilon	1.0
Epsilon decay	0.99
Episodes	2000
Max Steps	500
Number of runs	10

In the first case, when running the code 10 times with each run consisting of 2000 episodes using the Monte Carlo algorithm, we observe from Figure 5 that the average reward converges to 0 as the number of episodes increases. Initially, in the early episodes, there is more variability and undershoots in the average rewards. This is expected as the agent is exploring the environment and trying out different actions. However, as the number of episodes progresses, the agent learns from its experiences and improves its decision-making, resulting in a reduction in undershoots and variance.

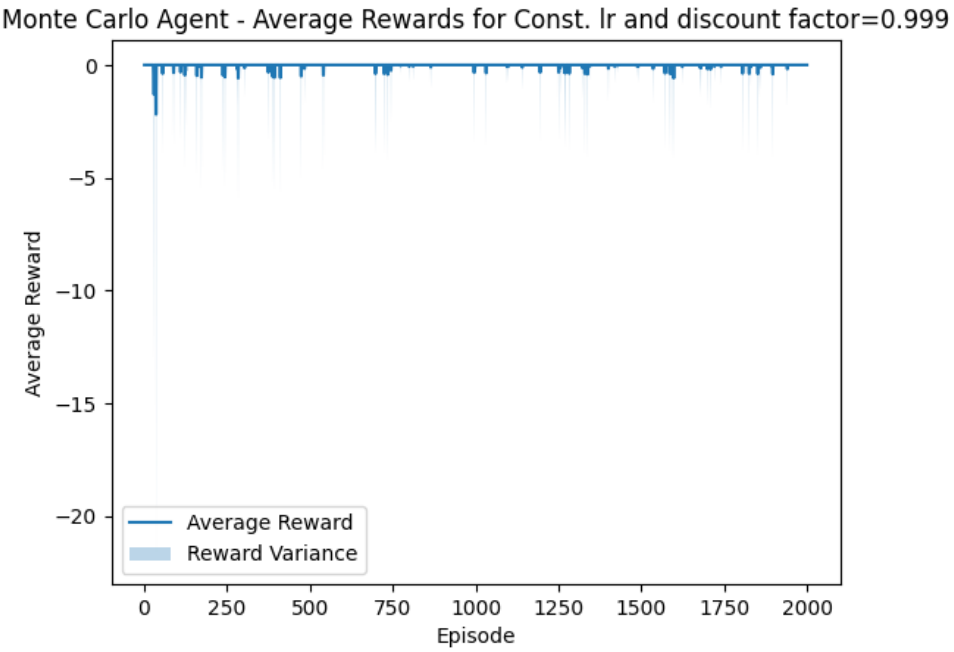

Figure 5. Average rewards in terms of episodes for Monte Carlo - Constant learning rate and discount factor = 0.999

Table 3. Monte Carlo algorithm results for 1st case

Parameter	Value
Episode with min. abs. avg. reward	1002
Average penalties per episode	0.0735

According to Figure 5, the convergence of the average reward to 0 indicates that the agent is effectively learning and optimizing its actions over time. It suggests that the agent becomes proficient at avoiding penalties and making progress towards the goal destination. However, it is important to note that this convergence to 0 does not necessarily imply that the agent achieves the maximum possible reward in every episode, but rather that it learns to minimize penalties and maximize rewards overall.

In the Monte Carlo algorithm, the episode is reset after every 100 movements of the agent. This reset prevents the agent from getting stuck or stopping prematurely. By resetting the episode, the agent has the opportunity to explore and learn optimal strategies for reaching the goal destination, even if it takes more than 100 movements. This approach enhances the agent's ability to fully explore the environment and discover effective policies.

Examining the specific results of the first case given in Table 3, we find that the episode with the minimum absolute average reward occurs at episode 1002. This indicates that the agent has

improved its performance significantly by this point. Additionally, the average penalties per episode are calculated to be 0.0735, indicating that the agent is successfully minimizing penalties and navigating the environment efficiently.

Overall, the convergence and stability of the Monte Carlo algorithm, considering the specified hyperparameters and the episode reset strategy, demonstrate the agent's ability to learn and optimize its actions over time. The agent progressively reduces undershoots and variance, leading to improved convergence to a reward close to 0 and effective navigation towards the goal destination.

In the first case, the episode with the minimum absolute average reward is found to be at episode 1002. This indicates that by the 1002nd episode, the agent has reached a point where the average reward has converged to its minimum absolute value.

Convergence in reinforcement learning refers to the point at which an agent's performance stabilizes and further training or episodes do not significantly improve its average reward or behavior. In this case, since the average reward converges to its minimum absolute value at episode 1002, we can say that the agent has achieved convergence by this point.

In this case, with the given hyperparameters and the Monte Carlo algorithm, the agent appears to achieve convergence within the 1002 episodes considered and we can say that required episodes for convergence could be almost 1002 episodes.

Note that in this part, we tuned the hyperparameters and selected the parameters in Table 2 by trial and error. It was observed that by choosing these parameters, we obtained the best average reward and in the least possible episodes.

Q-Learning:

In the first case, when running the code 10 times with each run consisting of 2000 episodes using the Q-Learning algorithm, we observe that the average reward gradually converges to 0 as the number of episodes increases. Unlike the Monte Carlo algorithm, where the convergence to 0 was achieved more quickly, the Q-Learning algorithm takes more episodes to reach convergence. The hyperparameters and properties used in this section are given in Table 4.

Table 4. Parameters of the 1st case for q-learning

Parameter	Value
Learning rate (alpha)	0.1
Learning rate decay (alpha decay)	1
Discount factor (gamma)	0.999
Epsilon	1.0
Epsilon decay	0.99
Episodes	2000
Max Steps	500
Number of runs	10

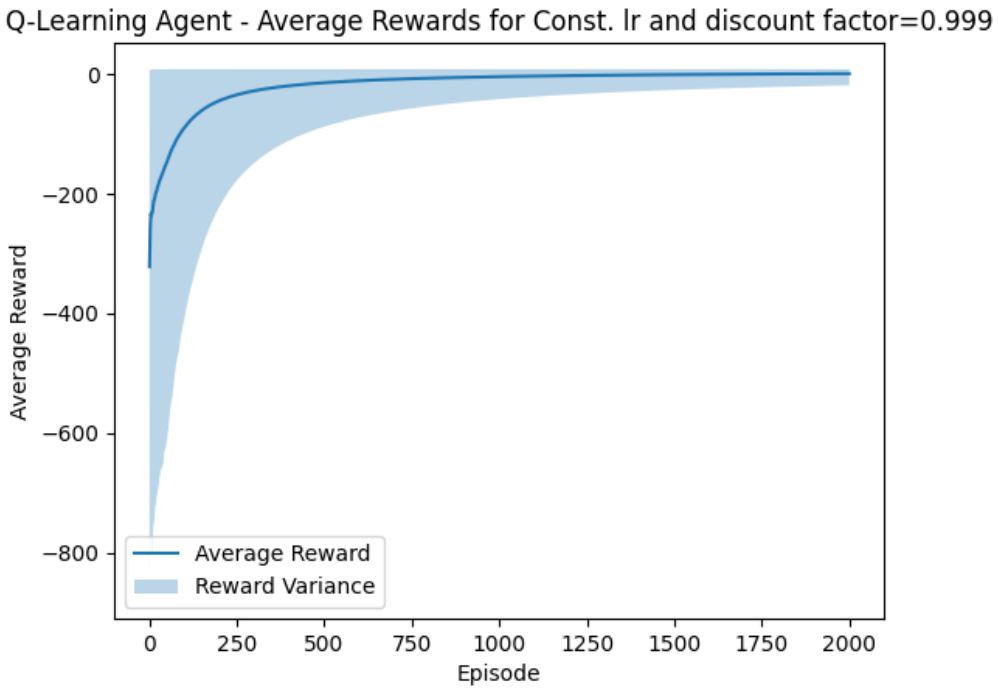

Figure 6. Average rewards in terms of episodes for Q-Learning - Constant learning rate and discount factor = 0.999

Table 5. Q-Learning algorithm results for 1st case

Parameter	Value
Episode with min. abs. avg. reward	271
Average penalties per episode	0.016

According to Figure 6, the convergence of the average reward to 0 indicates that the agent is gradually learning and improving its actions over time. As the agent explores the environment and interacts with it, it updates its Q-values based on the observed rewards and chooses actions that maximize its expected future rewards. With each episode, the agent refines its Q-values and policy, leading to a gradual reduction in penalties and an increase in rewards.

Examining the specific results of the first case for the Q-Learning algorithm which is shown in Table 5, we find that the episode with the minimum absolute average reward occurs at episode 271. This indicates that by the 271st episode, the agent has significantly improved its performance and we can say that almost 271 episodes are required for convergence in this case. Additionally, the average penalties per episode are calculated to be 0.016, indicating that the agent is successfully minimizing penalties and making progress in the environment. The convergence and stability of the Q-Learning algorithm, considering the specified hyperparameters, can be evaluated by analyzing the trend of the average rewards over episodes. The gradual convergence to 0 suggests that the agent is continuously learning and refining its policy. However, it's important to note that the convergence may not be as rapid or as smooth as in other algorithms, and the rate of convergence can be influenced by factors such as the learning rate, discount factor, and exploration-exploitation trade-off. In the first case with the Q-Learning algorithm, the episode with the minimum absolute average reward is found to be at episode 271.

Convergence in reinforcement learning refers to the point at which an agent's performance stabilizes and further training or episodes do not significantly improve its average reward or behavior. In this case, since the average reward converges to its minimum absolute value at episode 271, we can say that the agent has achieved convergence by this point.

Note that in this part, we tuned the hyperparameters and selected the parameters in Table 4 by trial and error. It was observed that by choosing these parameters, we obtained the best average reward and in the least possible episodes.

ii. 2nd case: Learning rate decay and discount factor = 0.999

Here, the learning rate starts at a higher value of 0.1 and gradually decays over time with an alpha decay of 0.99, while the discount factor remains constant at 0.999. The learning rate decay allows the agent to explore the environment more broadly in the early stages and then refine its learning as it progresses. By gradually reducing the learning rate, the agent places more emphasis on earlier experiences and stabilizes its learning process. The high discount factor still prioritizes long-term rewards.

Monte Carlo:

In the second case with the Monte Carlo algorithm and the given hyperparameters, we observe that the average reward converges to 0 as the number of episodes increases. Initially, there may be more undershoots and variance in the average reward, but as the episodes progress, the undershoots and variance decrease, leading to a better convergence and stability towards an average reward of 0. The hyperparameters and properties used in this section are given in Table 6.

Table 6. Parameters of the 2nd case for monte carlo

Parameter	Value
Learning rate (alpha)	0.1
Learning rate decay (alpha decay)	0.99
Discount factor (gamma)	0.999
Epsilon	1.0
Epsilon decay	0.99
Episodes	2000
Max Steps	500
Number of runs	10

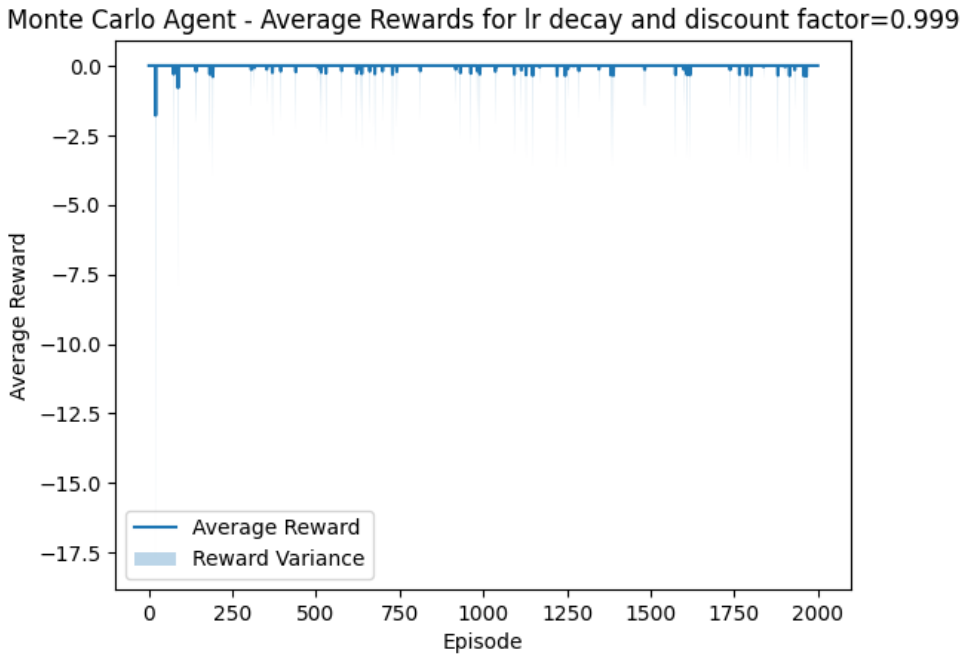

Figure 7. Average rewards in terms of episodes for Monte Carlo - Learning rate decay and discount factor = 0.999

Table 7. Monte Carlo algorithm results for 2nd case

Parameter	Value
Episode with min. abs. avg. reward	1269
Average penalties per episode	0.0418

In this case, from Figure 7 we understand that convergence to 0 is performed. According to Table 7, the episode with the minimum absolute average reward is found to be at episode 1269. This indicates that by the 1269th episode, the agent's average reward has converged to its minimum absolute value and this could be the number of episodes required for convergence in this case.

Convergence in reinforcement learning refers to the point at which the agent's performance stabilizes and further training or episodes do not significantly improve the average reward or behavior. In this case, the agent appears to achieve convergence within the considered 2000 episodes, with the minimum absolute average reward occurring at episode 1269.

Note that in this part, we tuned the hyperparameters and selected the parameters in Table 6 by trial and error. It was observed that by choosing these parameters, we obtained the best average reward and in the least possible episodes.

Q-Learning:

In the second case with the Q-Learning algorithm and the given hyperparameters, we observe that the average reward gradually converges to 0 as the number of episodes increases. The convergence indicates that the agent's learning process allows it to make better decisions over time, resulting in improved performance and a reduction in the average reward. The hyperparameters and properties used in this section are given in Table 8.

Table 8. Parameters of the 2nd case for q-learning

Parameter	Value
Learning rate (alpha)	0.1
Learning rate decay (alpha decay)	0.99
Discount factor (gamma)	0.999
Epsilon	1.0
Epsilon decay	0.99
Episodes	2000
Max Steps	500
Number of runs	10

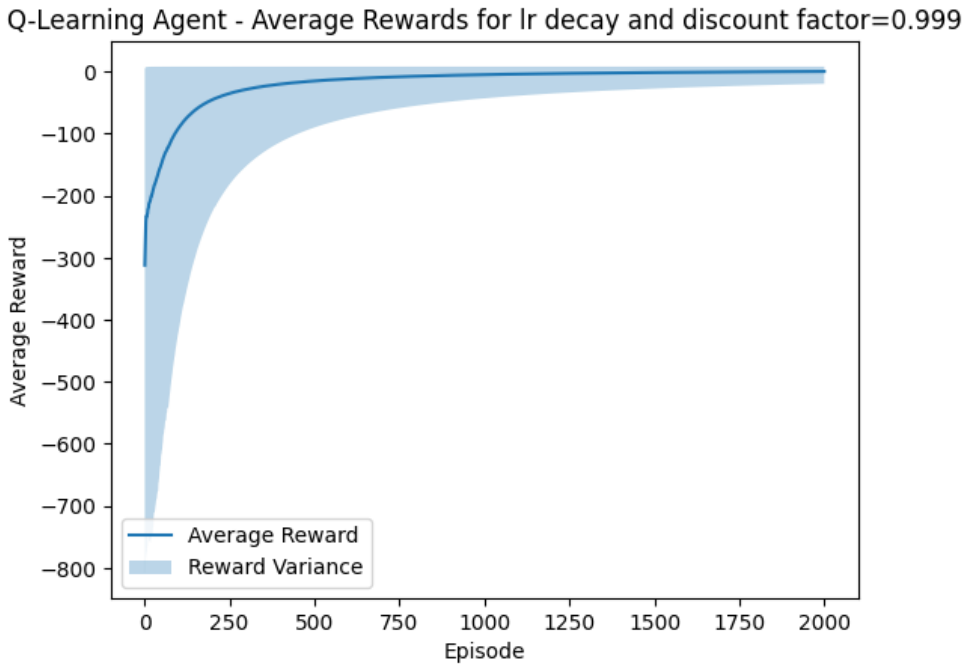

Figure 8. Average rewards in terms of episodes for Q-Learning - Learning rate decay and discount factor = 0.999

Table 9. Q-Learning algorithm results for 2nd case

Parameter	Value
Episode with min. abs. avg. reward	271
Average penalties per episode	0.1176

In this case, as shown in Table 9, the episode with the minimum absolute average reward is found to be at episode 271. This suggests that by the 271st episode, the agent's average reward has converged to its minimum absolute value. Also, as represented in Figure 8, the convergence to 0 is done perfectly and shows an acceptable learning process.

Convergence in reinforcement learning refers to the point at which the agent's performance stabilizes, and further training or episodes do not significantly improve the average reward or behavior. In this case, the agent appears to achieve convergence within the considered 2000 episodes, with the minimum absolute average reward occurring at episode 271.

Note that in this part, we tuned the hyperparameters and selected the parameters in Table 8 by trial and error. It was observed that by choosing these parameters, we obtained the best average reward and in the least possible episodes.

iii. 3rd case: Constant learning rate and discount factor = 0.9

In this case, the learning rate remains constant at 0.1, while the discount factor is set to 0.9. A constant learning rate of 0.1 indicates that the agent updates its Q-values more conservatively based on new information. The lower discount factor places less importance on future rewards compared to immediate rewards. This combination leads to a balance between considering immediate rewards and exploring future rewards, but with a focus on shorter-term optimization.

Monte Carlo:

In the third case with the Monte Carlo algorithm and the specified hyperparameters, we observe that the average reward converges to 0 as the number of episodes increases. Initially, there may be more undershoots and higher variance in the average reward, but as the episodes progress, these diminish, resulting in a better convergence towards 0. The hyperparameters and properties used in this section are given in Table 10.

Table 10. Parameters of the third case for monte carlo

Parameter	Value
Learning rate (alpha)	0.1
Learning rate decay (alpha decay)	1
Discount factor (gamma)	0.9
Epsilon	1.0
Epsilon decay	0.99
Episodes	2000
Max Steps	500
Number of runs	10

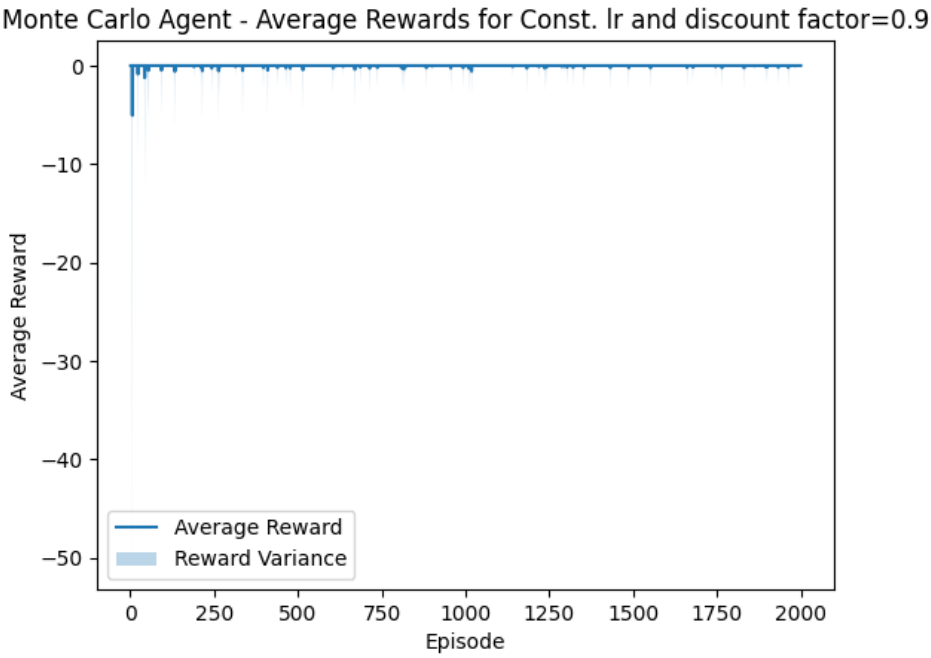

Figure 9. Average rewards in terms of episodes for Monte Carlo - Constant learning rate and discount factor = 0.9

Table 11. Monte Carlo algorithm results for 3rd case

Parameter	Value
Episode with min. abs. avg. reward	1893
Average penalties per episode	0.0707

Based on the provided results in Table 11, the episode with the minimum absolute average reward is found to be at episode 1893. This indicates that by the 1893rd episode, the agent's average reward has converged to its minimum absolute value and it could be the required number of episodes for convergence.

Based on the Figure 9, convergence to 0 refers to the point where the agent's performance stabilizes, and further training or episodes do not significantly improve the average reward or behavior.

The average penalties per episode in the third case with the Monte Carlo algorithm and the specified hyperparameters are found to be approximately 0.0707. This indicates that, on average, the agent incurs 0.0707 penalties per episode during the training process.

Penalties are typically associated with undesirable actions or incorrect decisions made by the agent in the environment. In this case, the average penalties per episode represent the frequency at which the agent makes such mistakes or takes actions that lead to negative consequences. A

lower value for the average penalties per episode is desirable, as it indicates that the agent is making fewer errors or learning to navigate the environment more effectively. However, the significance of this value depends on the specific problem and the definition of penalties within the environment. By monitoring and analyzing the average penalties per episode, we can gain insights into the agent's performance and its ability to learn from past experiences. This information can be useful for evaluating the convergence and stability of the algorithm and making improvements if necessary.

Note that in this part, we tuned the hyperparameters and selected the parameters in Table 10 by trial and error. It was observed that by choosing these parameters, we obtained the best average reward and in the least possible episodes.

Q-Learning:

Based on the provided information for the third case with the Q-Learning algorithm and the specified hyperparameters, we observe the results in Table 13. The hyperparameters and properties used in this section are given in Table 12.

Table 12. Parameters of the third case for q-learning

Parameter	Value
Learning rate (alpha)	0.1
Learning rate decay (alpha decay)	1
Discount factor (gamma)	0.9
Epsilon	1.0
Epsilon decay	0.99
Episodes	2000
Max Steps	500
Number of runs	10

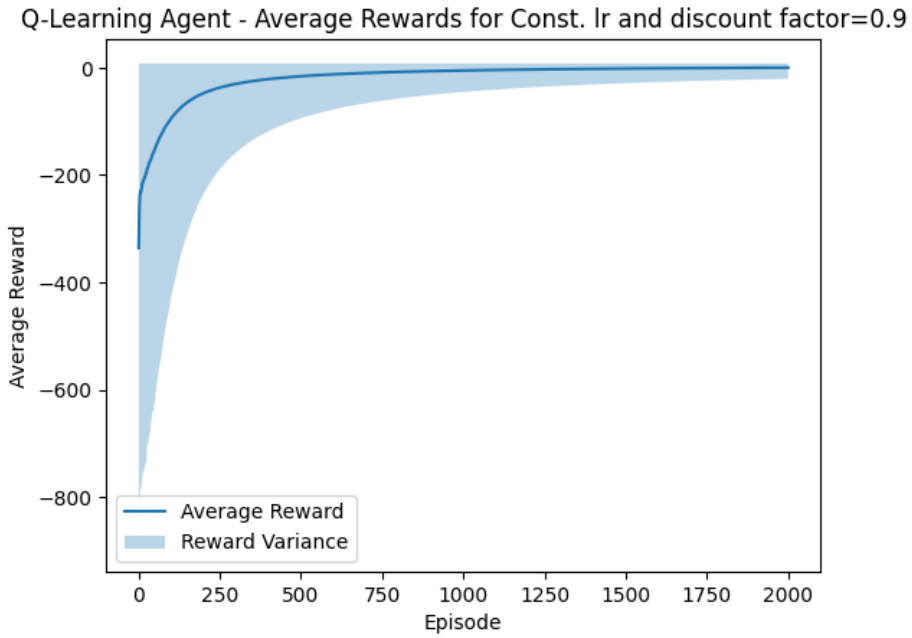

Figure 10. Average rewards in terms of episodes for Q-Learning - Constant learning rate and discount factor = 0.9

Table 13. Q-Learning algorithm results for 3rd case

Parameter	Value
Episode with min. abs. avg. reward	352
Average penalties per episode	0.014

To determine the number of episodes needed for convergence, we analyze the episode with the minimum absolute average reward. In this case, as shown in Table 13, the episode with a minimum absolute average reward of 352 indicates the point at which the algorithm achieved a relatively stable and low average reward. As shown in Figure 10, the average reward has converged to 0 that shows a good learning process.

Convergence in the context of reinforcement learning refers to the point at which the agent's policy and value estimates have sufficiently converged to the optimal or near-optimal values. It indicates that the agent has learned the optimal behavior in the given environment.

In this case, as shown in Table 13 based on the episode with the minimum absolute average reward of 352, we can conclude that the Q-Learning algorithm with the specified hyperparameters started to converge around that episode and it is the least required number of episodes.

In this case, with an average of 0.014 penalties per episode, the algorithm demonstrates good performance and shows a relatively low rate of making mistakes or incurring penalties. This suggests that the Q-Learning algorithm with the given hyperparameters is effectively learning the optimal or near-optimal policy in the environment.

Note that in this part, we tuned the hyperparameters and selected the parameters in Table 12 by trial and error. It was observed that by choosing these parameters, we obtained the best average reward and in the least possible episodes.

iv. 4th case: Learning rate decay and discount factor = 0.9

Similar to case 2, the learning rate starts at 0.1 and decays over time with an alpha decay of 0.99, while the discount factor remains constant at 0.9. This combination allows the agent to explore the environment more broadly in the early stages and then fine-tune its learning with a lower learning rate. The lower discount factor places less emphasis on long-term rewards compared to immediate rewards, resulting in a greater focus on shorter-term optimization.

Monte Carlo:

Based on the provided information for the fourth case with the Monte Carlo algorithm and the specified hyperparameters, we observe the following result given in Table 15. The hyperparameters and properties used in this section are given in Table 14.

Table 14. Parameters of the 4th case for monte carlo

Parameter	Value
Learning rate (alpha)	0.1
Learning rate decay (alpha decay)	0.99
Discount factor (gamma)	0.9
Epsilon	1.0
Epsilon decay	0.99
Episodes	2000
Max Steps	500
Number of runs	10

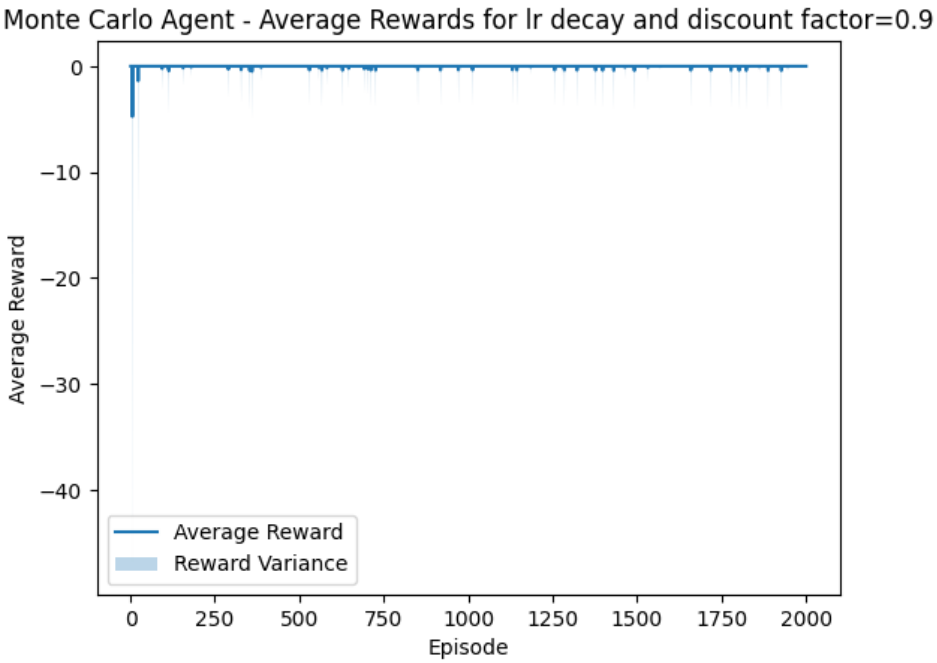

Figure 11. Average rewards in terms of episodes for Monte Carlo - Learning rate decay and discount factor = 0.9

Table 15. Monte Carlo algorithm results for 4th case

Parameter	Value
Episode with min. abs. avg. reward	1383
Average penalties per episode	0.0322

The episode with the minimum absolute average reward indicates the episode number at which the algorithm achieved the best performance in terms of average rewards. We can see form Table 15 that in this case, the episode with the min. absolute average reward is 1383 and we can conclude that it is the required number of episodes for convergence.

The average penalties per episode indicate the average number of penalties (undesirable actions or states) encountered by the agent during each episode. In this case, the average penalties per episode are 0.0322 and is relatively low.

To investigate the convergence and stability of the algorithm, we can analyze the trend of average rewards and penalties over the episodes as shown in Figure 11. In our case, the average rewards converge to a low value or approach zero, and the average penalties decrease or remain low as the episodes increase, it indicates that the algorithm is learning and improving its performance in the environment.

Note that in this part, we tuned the hyperparameters and selected the parameters in Table 14 by trial and error. It was observed that by choosing these parameters, we obtained the best average reward and in the least possible episodes.

Q-Learning:

In the fourth case with the Q-Learning algorithm and the specified hyperparameters, we observe the following results in Table 17. The hyperparameters and properties used in this section are given in Table 16.

Table 16. Parameters of the 4th case for q-learning

Parameter	Value
Learning rate (alpha)	0.1
Learning rate decay (alpha decay)	0.99
Discount factor (gamma)	0.9
Epsilon	1.0
Epsilon decay	0.99
Episodes	2000
Max Steps	500
Number of runs	10

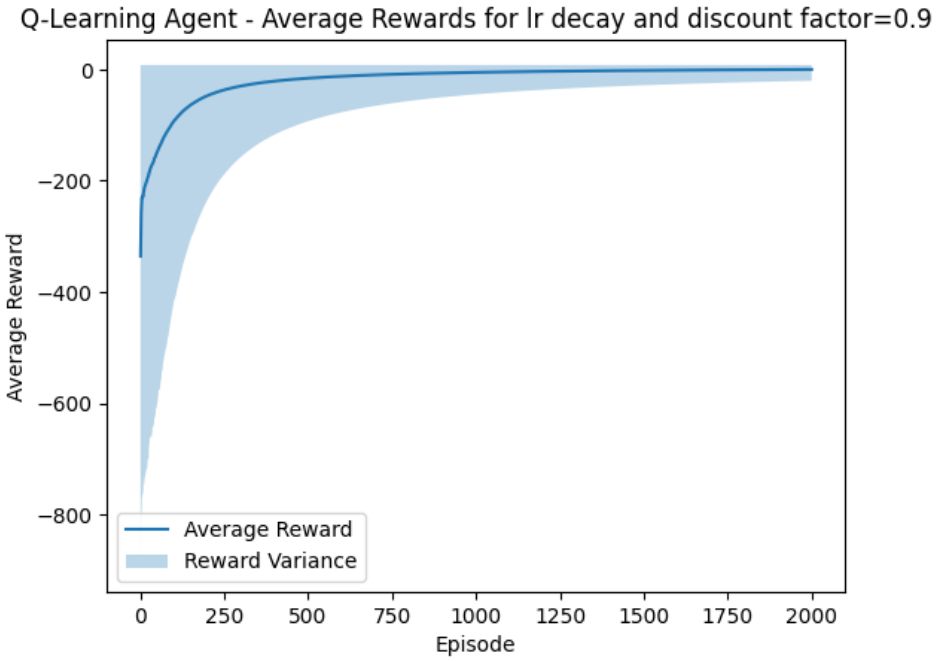

Figure 12. Average rewards in terms of episodes for Q-Learning - Learning rate decay and discount factor = 0.9

Table 17. Q-Learning algorithm results for 4th case

Parameter	Value
Episode with min. abs. avg. reward	352
Average penalties per episode	0.0793

To investigate the convergence and stability of the Q-Learning algorithm, we can analyze the trend of average rewards and penalties over the episodes. As the average rewards gradually converge to 0 or approach a low value as the episodes increase, it indicates that the algorithm is learning and improving its performance in the environment as shown in Figure 12. Additionally, a low average penalties per episode suggests that the agent is making fewer mistakes or encountering fewer undesirable states/actions.

Based on the given results in Table 17, we can see that the episode with the minimum absolute average reward is 352 and this is the least required number of episodes for convergence. This suggests that the algorithm has made progress and achieved a better reward compared to earlier episodes. The average penalties per episode are 0.0793, indicating that the agent encounters some penalties during its exploration, but the average number of penalties per episode is relatively low.

To further assess the convergence and stability of the algorithm, it would be beneficial to examine the trend of average rewards and penalties over multiple runs, as well as the variance or consistency in performance. Additionally, conducting additional experiments and analyzing the learning curves could provide more insights into the convergence behavior of the algorithm.

Therefore, based on the given results, we can conclude that the Q-Learning algorithm with the specified hyperparameters in the fourth case demonstrates convergence toward a lower average reward and relatively low average penalties per episode.

Note that in this part, we tuned the hyperparameters and selected the parameters in Table 16 by trial and error. It was observed that by choosing these parameters, we obtained the best average reward and in the least possible episodes.

D. Rendering

In this section, the following code defines a function called record_env that records the frames of an environment while an agent interacts with it. The function takes the environment, agent, and a maximum number of frames as input. It initializes an empty list to store the frames and starts a loop. In each iteration, the agent selects an action, interacts with the environment, and records the rendered frame. The loop continues until the episode is done or the maximum number of frames is reached. Finally, the function returns the list of recorded frames.

```
# Render
# Constant learning rate and discount factor = 0.999

def record_env(env, agent, max_frames):
    frames = []
    state = env.reset()
    frames.append(env.render(mode='rgb_array'))
    for _ in range(max_frames):
        action = agent.act(state)
        state, _, done, _ = env.step(action)
        frames.append(env.render(mode='rgb_array'))
        if done:
            break
    return frames
```

Then, we use the following code which generates and displays GIF animations for the Monte Carlo agent and the Q-learning agent interacting with the environment. It records the frames, saves them as GIF files, prints the file names, and displays the GIF animations using the display function.

```
# Render
# Generate Monte Carlo GIF
monte_carlo_frames = record_env(env, monte_carlo_agent, max_frames=500)
imageio.mimsave('monte_carlo for Const. lr and discount
factor=0.999.gif', monte_carlo_frames, duration=0.001)

print("monte_carlo for Const. lr and discount factor=0.999.gif:")
# Display Monte Carlo GIF
display(Image(filename='monte_carlo for Const. lr and discount
factor=0.999.gif'))
```

```
# Render
# Generate Q-Learning GIF
q_learning_frames = record_env(env, q_learning_agent, max_frames=500)
```

```
imageio.mimsave('q_learning for Const. lr and discount
factor=0.999.gif', q_learning_frames, duration=0.001)

print("q_learning for Const. lr and discount factor=0.999.gif:")
# Display Q-Learning GIF
display(Image(filename='q_learning for Const. lr and discount
factor=0.999.gif'))
```

This code generates and displays GIF animations for the Monte Carlo agent and the Q-learning agent interacting with the environment. First, it calls the record_env function to record the frames of the environment while the Monte Carlo agent interacts with it. The recorded frames are stored in the monte_carlo_frames variable. Then, it saves the monte_carlo_frames as a GIF file using the imageio.mimsave function, specifying the file name and the duration of each frame. After saving the GIF, it prints a message indicating the name of the Monte Carlo GIF.

Next, it displays the Monte Carlo GIF using the display function from the IPython.display module. It reads and displays the GIF file specified by the file name. The same process is repeated for the Q-learning agent. The frames of the environment while the Q-learning agent interacts with it are recorded and stored in the q_learning_frames variable. The frames are saved as a GIF file, and the file name is printed. Finally, the Q-learning GIF is displayed.

i. 1st case: Constant learning rate and discount factor = 0.999

Monte Carlo algorithm:

In the first case, where we use a constant learning rate of 0.1 and a discount factor of 0.999 with the Monte Carlo algorithm, when we render the environment, we observe from Figure 13 that the taxi moves correctly and successfully completes its task of picking up and delivering passengers to specific states in some frames.

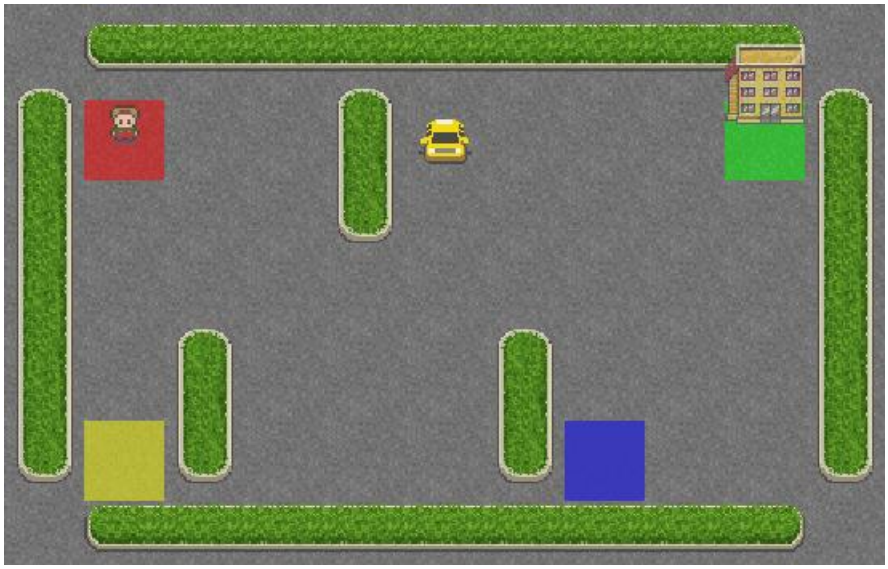

Figure 13. Rendering of Monte Carlo - Constant learning rate and discount factor = 0.999

Q-Learning algorithm:

In the first case, where we use a constant learning rate of 0.1 and a discount factor of 0.999 with the Q-learning algorithm, when we render the Taxi-v3 environment, as shown in Figure 14, we observe that the taxi moves correctly and successfully completes its task of picking up the passenger and delivering them to the office destination.

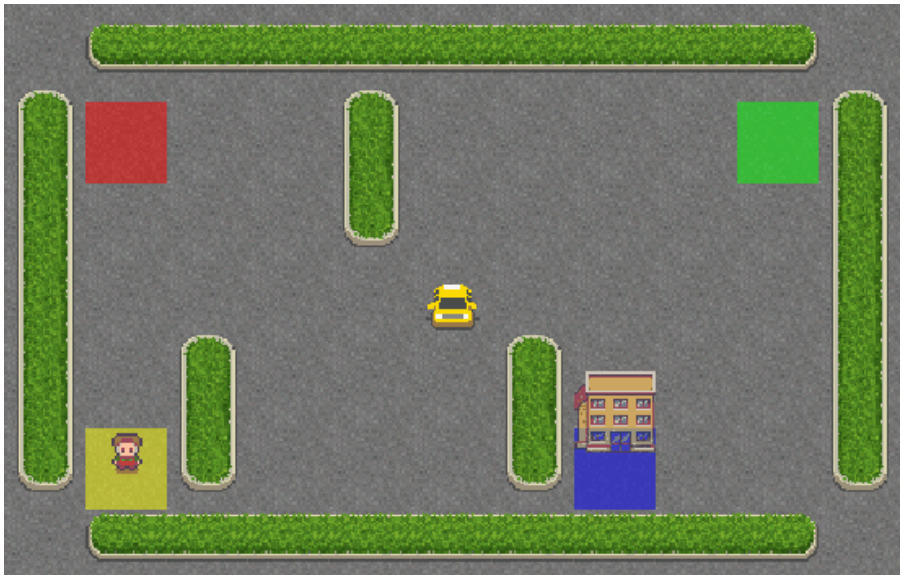

Figure 14. Rendering of Q-Learning - Constant learning rate and discount factor = 0.999

ii. 2nd case: Learning rate decay and discount factor = 0.999

Monte Carlo algorithm:

In the second case, we use a learning rate decay with an initial learning rate of 0.1, an alpha decay of 0.99, and a discount factor of 0.999 with the Monte Carlo algorithm. When we render the Taxi-v3 environment as represented in Figure 15, we see that the taxi moves correctly and, in some frames, successfully picks up the passenger and delivers them to specific states.

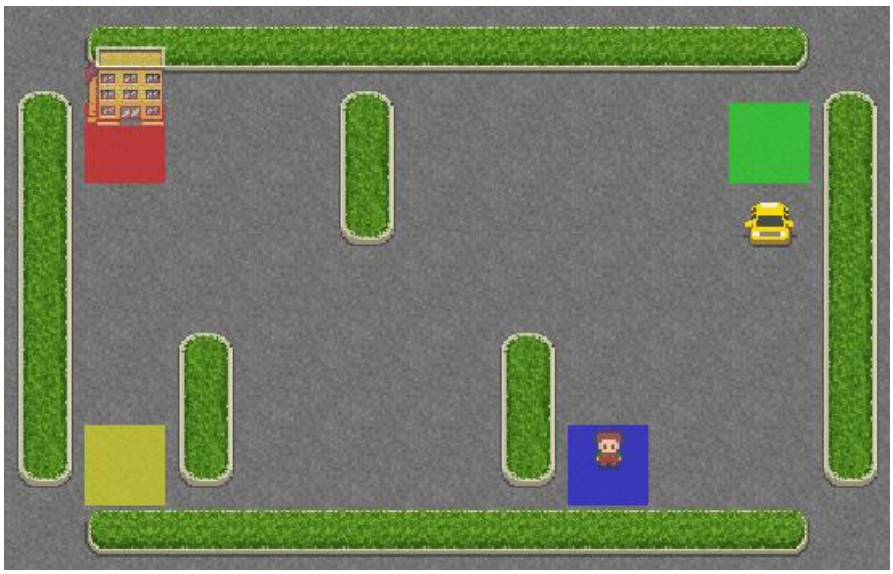

Figure 15. Rendering of Monte Carlo - Learning rate decay and discount factor = 0.999

Q-Learning algorithm:

In the second case, we utilize a learning rate decay with an initial learning rate of 0.1, an alpha decay of 0.99, and a discount factor of 0.999 with the Q-learning algorithm. After rendering the Taxi-v3 environment as shown in Figure 16, we observe that the taxi moves correctly, successfully picks up passengers and delivers them to the destination goal, which is the office.

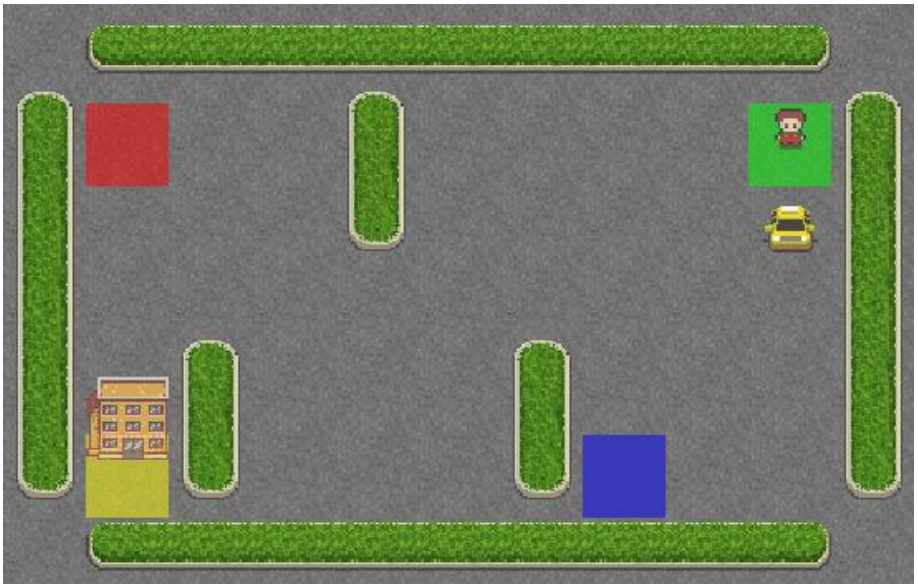

Figure 16. Rendering of Q-Learning - Learning rate decay and discount factor = 0.999

iii. 3rd case: Constant learning rate and discount factor = 0.9

Monte Carlo algorithm:

In the third case, we employ a constant learning rate of 0.1 and a discount factor of 0.9 with the Monte Carlo algorithm. After rendering the Taxi-v3 environment, we observe that the taxi moves correctly and, in certain frames, successfully picks up passengers and delivers them to specific states. The rendering in this case, is given in a GIF file as shown in Figure 17.

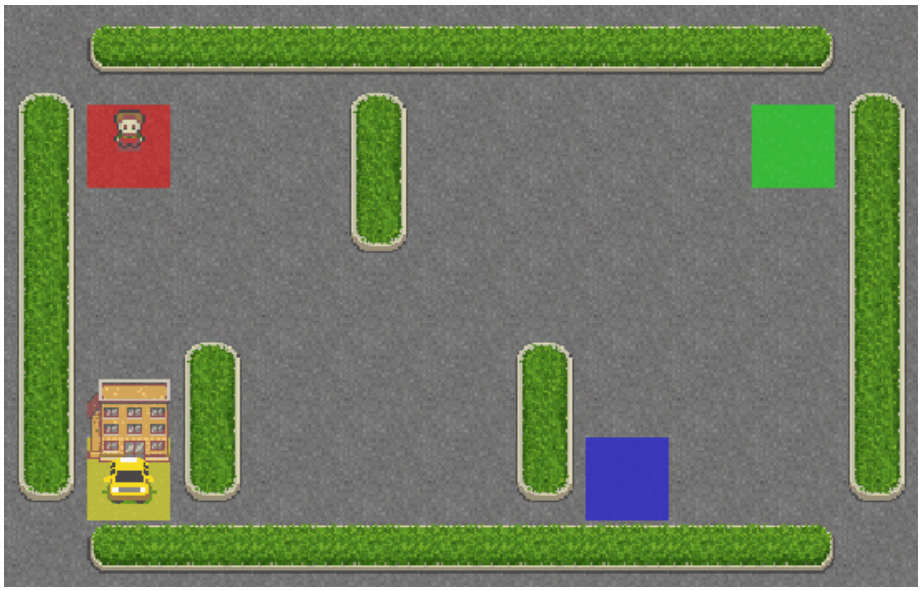

Figure 17. Rendering of Monte Carlo - Constant learning rate and discount factor = 0.9

Q-Learning algorithm:

In the third case, we utilize a constant learning rate of 0.1 and a discount factor of 0.9 with the Q-learning algorithm. After rendering the Taxi-v3 environment, we observe that the taxi moves correctly, picks up passengers, and successfully delivers them to the destination goal, which is the office. The rendering with the format of GIF file is given in Figure 18.

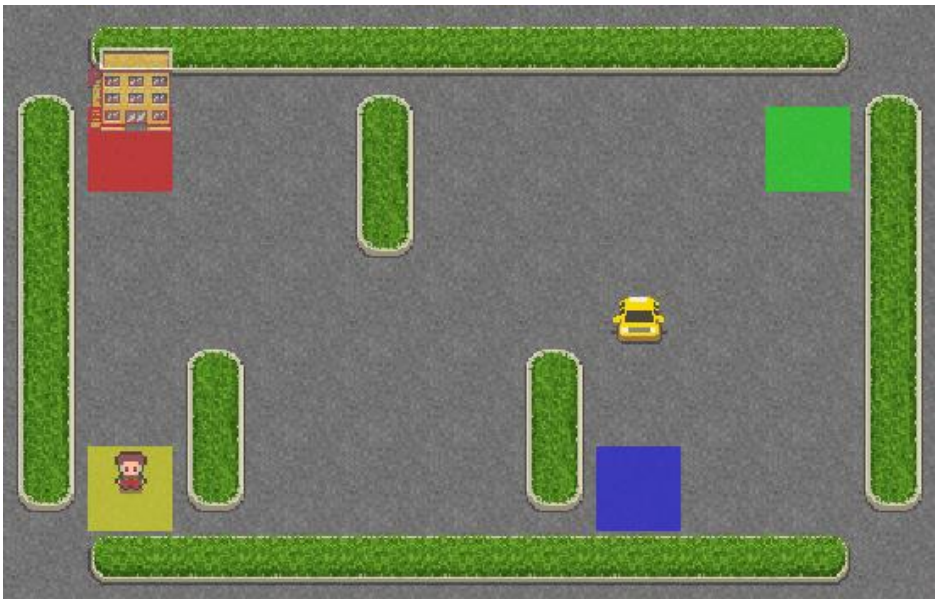

Figure 18. Rendering of Q-Learning - Constant learning rate and discount factor = 0.9

iv. 4th case: Learning rate decay and discount factor = 0.9

Monte Carlo algorithm:

In the fourth case, we utilize a learning rate decay strategy and a discount factor of 0.9 with the Monte Carlo algorithm. After rendering the Taxi-v3 environment as represented in Figure 19, we observe that the taxi moves correctly and, in some frames, picks up the passenger and delivers them to certain states.

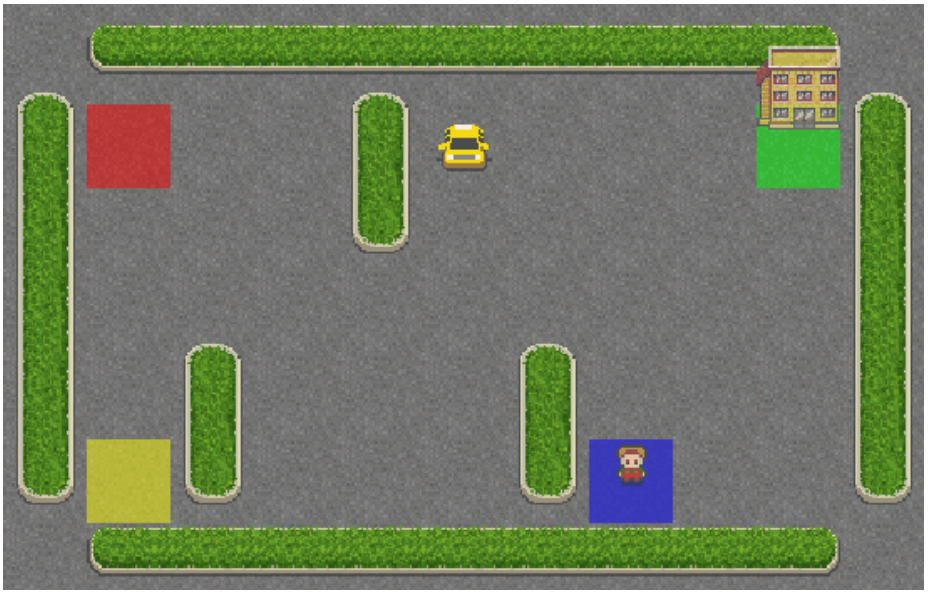

Figure 19. Rendering of Monte Carlo - Learning rate decay and discount factor = 0.9

Q-Learning algorithm:

In the fourth case, we employ a learning rate decay strategy and a discount factor of 0.9 with the Q-learning algorithm. After rendering the Taxi-v3 environment as shown in Figure 20, we observe that the taxi moves correctly, picks up passengers, and successfully delivers them to the destination goal, which is the office.

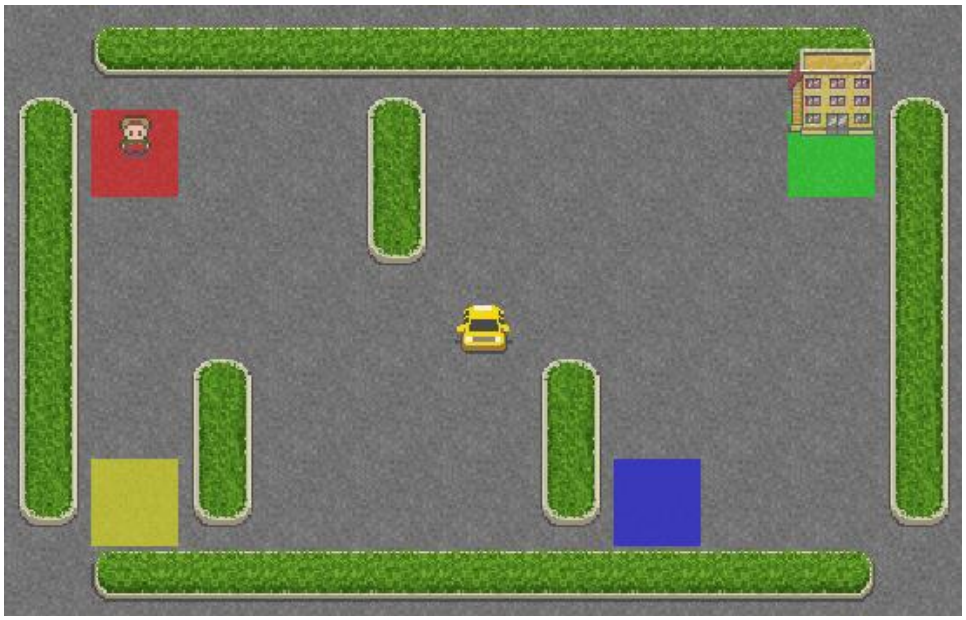

Figure 20. Rendering of Q-Learning - Learning rate decay and discount factor = 0.9

E. Deep Q-Learning

Table 18 provides the parameter values used in the Deep Q-Learning algorithm implemented in the code presented below. These parameter values define the settings and behavior of the Deep Q-Learning algorithm, including the exploration-exploitation trade-off, training iterations, network architecture, and optimization algorithm.

Table 18. Parameters of the deep q-learning algorithm

Parameter	Value
Learning rate (alpha)	0.1
Maxlen	1.0
Discount factor (gamma)	0.6
Epsilon	1.0
Epsilon decay	0.99
Episodes	1000
Epsilon min	0.01
Batch size	32
Time steps per episode	2
Loss function	MSE
Hidden layers activation	Relu
Last layer activation	Linear
Optimizer	Adam

The following code implements a Deep Q-Learning (DQL) algorithm using the Gym library in Python.

```
##### Deep Q-Learning (DQL) #####

# Create the Taxi environment
env = gym.make("Taxi-v3").env
env.seed(seed=44) # Student ID: 810601044 --> Last three digits: 044 --
> seed=44
```

The above code creates an instance of the Taxi environment. In Gym, environments are encapsulated as objects, and each environment represents a specific problem or scenario for reinforcement learning. Here, the gym.make() function is used to create an instance of the Taxi environment. The string "Taxi-v3" specifies the environment and indicates that the code is using the third version of the Taxi problem in the Gym library. The .env attribute is accessed to obtain the underlying environment object.

After creating the environment, the next line sets the random seed. The env.seed() function is called to specify a random seed value, which ensures reproducibility of results. In this case, the seed value is based on the my student ID number (810601044), and the last three digits (044) are used as the seed value (i.e., seed=44).

The following code defines a class called Agent that represents an agent in a reinforcement learning scenario.

```
class Agent:
    def __init__(self, env, optimizer):
        self._state_size = env.observation_space.n
        self._action_size = env.action_space.n
        self._optimizer = optimizer

        self.experience_replay = deque(maxlen=1) # Experience replay
buffer

        self.gamma = 0.6 # Discount factor
        self.epsilon = 1.0 # Exploration rate
        self.epsilon_decay = 0.99 # Decay rate for exploration rate
        self.epsilon_min = 0.01 # Minimum exploration rate

        self.q_network = self._build_compile_model() # Q-Network
        self.target_network = self._build_compile_model() # Target Q-
Network
        self.align_target_model() # Align target Q-Network with Q-
Network

    def store(self, state, action, reward, next_state,
terminated):     # Store the experience (state, action, reward,
next_state, terminated) in the experience replay buffer
        self.experience_replay.append((state, action, reward,
next_state, terminated))
```

The above code defines an Agent class for reinforcement learning. The class constructor initializes the agent with environment details, hyperparameters, and neural network models. It sets up an experience replay buffer to store the agent's experiences. The store() method adds experiences to the buffer. Overall, the code prepares the agent for training and learning in a reinforcement learning scenario.

The following code defines a private method called _build_compile_model(), which is responsible for constructing and compiling a Q-Network model for reinforcement learning.

```
    def _build_compile_model(self):     # Build and compile the Q-
Network
        model = Sequential()
        model.add(Embedding(self._state_size, 10, input_length=1))
        model.add(Reshape((10,)))
        model.add(Dense(65, activation='relu'))
        model.add(Dense(65, activation='relu'))
        model.add(Dense(self._action_size, activation='linear'))

        model.compile(loss='mse', optimizer=self._optimizer)
        return model
```

The method begins by creating a Sequential model using the Keras library. Sequential models allow for a linear stack of layers.

The constructed model consists of several layers:

- An Embedding layer is added, which maps discrete state values to dense vectors of size 10. It takes the _state_size as input and has an input_length of 1 since the input is a single state.
- A Reshape layer follows, reshaping the output of the embedding layer into a 1-dimensional vector of size 10 to prepare it for the subsequent layers.
- Two Dense layers with 65 units each are added. These fully connected layers use the rectified linear unit (ReLU) activation function, which introduces non-linearity to the network and helps capture complex patterns in the data.
- The final Dense layer has _action_size units, representing the output size of the Q-Network. It uses the linear activation function, producing unbounded continuous values.

Once the model is constructed, it is compiled by specifying the loss function and optimizer. The loss function is set to 'mse' (mean squared error), commonly used in Q-Learning to measure the discrepancy between predicted and target Q-values. The _optimizer parameter, which is provided during the agent's initialization, is used as the optimizer for training the model.

Finally, the compiled model is returned, providing the Q-Network that can be used for action selection and value estimation in reinforcement learning algorithms. Then, we visualize the model as given in Figure 21.

```
# The model visualization
keras.utils.plot_model(model)
```

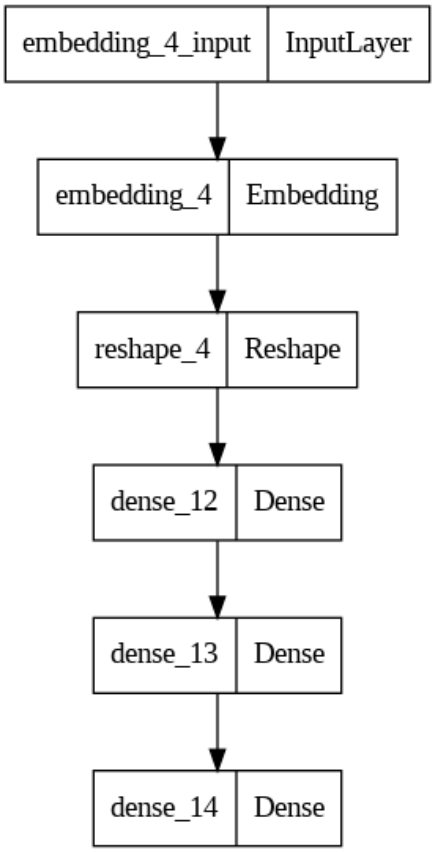

Figure 21. DQN model visualization

Then, we use the following code that implements a reinforcement learning agent using the Deep Q-Learning (DQL) algorithm.

```
    def align_target_model(self):
    # Update the target Q-Network with the weights from the Q-Network
        self.target_network.set_weights(self.q_network.get_weights())

    def act(self, state):
        if np.random.rand() <= self.epsilon:
        # Exploration: choose a random action
            return env.action_space.sample()

        # Exploitation: choose the action with the highest Q-value
        q_values = self.q_network.predict(state)
        return np.argmax(q_values[0])

    def retrain(self, batch_size):
        minibatch = random.sample(self.experience_replay, batch_size)

        for state, action, reward, next_state, terminated in minibatch:
            target = self.q_network.predict(state)
```

```
            if terminated:
            # If the episode is terminated, the target Q-value is just
the reward
                target[0][action] = reward
            else:
            # If the episode is not terminated, update the target Q-
value using the Bellman equation
                t = self.target_network.predict(next_state)
                target[0][action] = reward + self.gamma * np.amax(t)

        # Train the Q-Network using the target Q-value
            self.q_network.fit(state, target, epochs=1, verbose=0)

    def update_epsilon(self):
          # Decay the exploration rate (epsilon) over time
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay


optimizer = Adam(learning_rate=0.1)
agent = Agent(env, optimizer)

num_of_episodes = 1000
timesteps_per_episode = 2
batch_size = 32

episode_rewards = []
average_rewards = []
min_rewards = []
max_rewards = []

for episode in range(num_of_episodes):
    state = env.reset()
    state = np.reshape(state, [1, 1])

    total_reward = 0
    terminated = False

    for timestep in range(timesteps_per_episode):
        action = agent.act(state)
        next_state, reward, terminated, _ = env.step(action)
        next_state = np.reshape(next_state, [1, 1])
        agent.store(state, action, reward, next_state, terminated)
        state = next_state
        total_reward += reward

        if terminated:
            agent.align_target_model()
            break
```

```
        if len(agent.experience_replay) > batch_size:
            agent.retrain(batch_size)

    episode_rewards.append(total_reward)
    average_reward = np.mean(episode_rewards[max(0, episode-
99):episode+1])
    average_rewards.append(average_reward)

    agent.update_epsilon()

    print(f"Episode: {episode+1}/{num_of_episodes}, Reward:
{total_reward}, Average Reward (Last 100 Episodes): {average_reward}")

# Plotting the average rewards per episode
plt.plot(range(num_of_episodes), average_rewards, label='Average
Reward')
plt.xlabel('Episodes')
plt.ylabel('Reward')
plt.title('Reward per Episode')
plt.legend()
plt.show()
```

In this code, the agent interacts with an environment for a specified number of episodes, taking actions based on a combination of exploration and exploitation. During each episode, the agent stores experiences in an experience replay buffer. After a certain number of experiences have been accumulated, the agent retrains its Q-Network by sampling a batch of experiences and updating the Q-values using the Bellman equation. The target Q-Network is periodically updated with the weights of the Q-Network. The agent's exploration rate (epsilon) is decayed over time to gradually shift from exploration to exploitation. The rewards obtained during each episode are recorded, and the average reward over the last 100 episodes is calculated and plotted.

After running the code, we will have the average reward in terms of episodes as given in Figure 22.

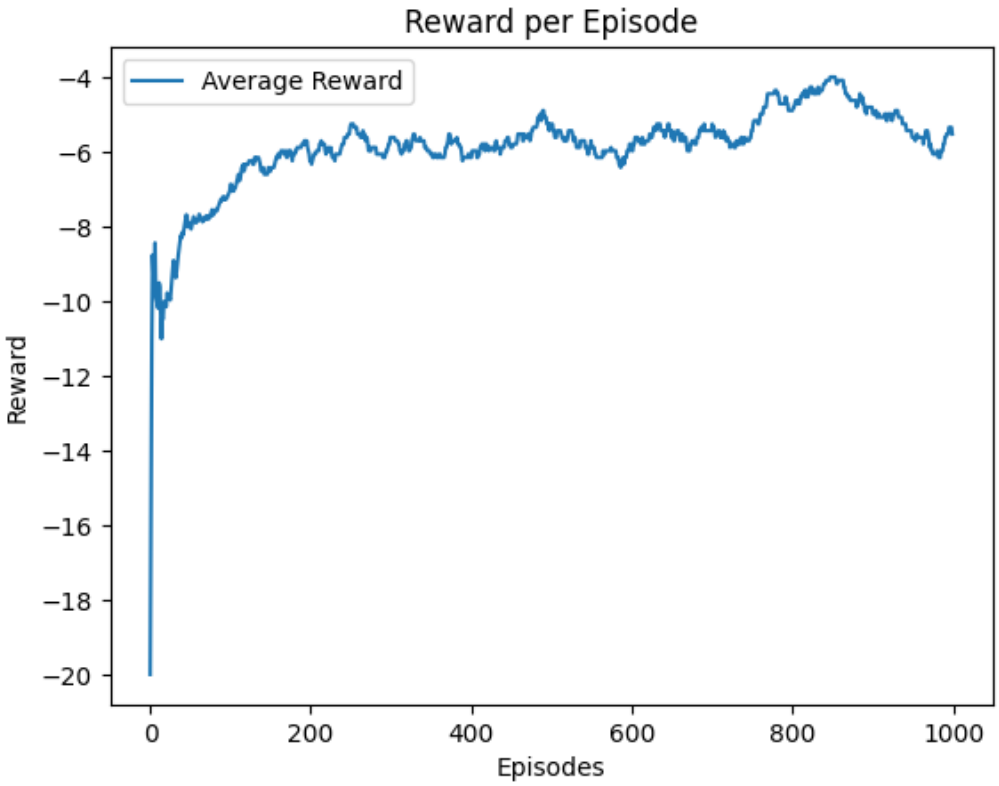

Figure 22. DQN average rewards in terms of episodes

According to Figure 22, the average reward obtained by the agent starts from -20 at episode 0 and gradually increases over the episodes, converging towards -6 at around -200 episodes. This indicates that the agent is initially performing poorly but gradually improves its policy through the learning process. The increase in average reward demonstrates that the agent is learning to make better decisions and achieve higher rewards. However, since the convergence point is still negative, it suggests that there is still room for further improvement.

If the number of episodes is increased and the code is run again, it is likely that the average reward will continue to improve and eventually reach or approach 0. A longer training duration allows the agent to explore and exploit the environment more extensively, leading to better performance.

Notice: Running the code with an increased number of episodes, steps, and maxlen will lead to better results. However, due to the significant amount of time required and the need for powerful hardware, we cannot further increase the number of episodes, steps, and maxlen.

Thanks for your Time