Report Project Machine Learning

Mario Morra 2156770, Leonardo Sereni

Project Objectives

The objective of this project is to design, implement, and analyze Reinforcement Learning agents based on the Q-learning framework. Two complementary approaches are considered: a tabular representation of the action-value function and a function approximation approach based on Deep Q-Networks (DQN).

The main goals of the project are:

- To formally model the selected environment as a Markov Decision Process (MDP).
- To implement the Q-learning algorithm in its tabular form, explicitly representing the action-value function.
- To extend the same learning principle to a neural-network-based approximation through a Deep Q-Network.
- To design and analyze an exploration strategy based on the epsilon-greedy policy.
- To investigate the convergence behavior and learning dynamics of both approaches.
- To evaluate and compare the learned policies in terms of performance, stability, and generalization capability.

The tabular approach provides a clear and interpretable implementation of Q-learning in environments with finite and relatively small state spaces, such as Taxi-v3, where the action-value function can be stored explicitly. In contrast, the Deep Q-Network approach replaces the table with a parameterized function approximator, enabling scalability to larger or continuous state spaces.

Together, these two implementations allow a comprehensive analysis of value-based reinforcement learning, highlighting both the theoretical foundations and the practical limitations of tabular methods, as well as the advantages introduced by neural network approximation.

Environment Description

Environment: Taxi-v3

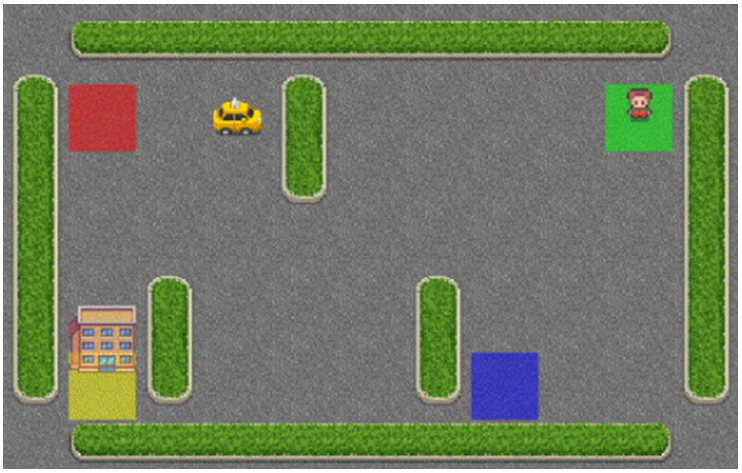


Figure 1: Taxi-v3 environment visualization

The environment selected for this project is Taxi-v3, a classic benchmark environment provided by the Gymnasium library. It represents a discrete, fully observable, episodic Markov Decision Process.

In this environment, a taxi agent operates in a 5×5 grid world. The task consists of:

1. Navigating to the passenger's location,
2. Executing a pickup action,
3. Navigating to the specified destination,
4. Executing a dropoff action.

The state space consists of 500 discrete states. Each state encodes:

- The taxi position (25 possible grid locations),
- The passenger location (4 fixed locations or inside the taxi),
- The destination location (4 fixed locations).

The action space is composed of 6 discrete actions:

- Move south,
- Move north,
- Move east,

- Move west,
- Pickup passenger,
- Dropoff passenger.

The reward structure is defined as follows:

- +20 for successfully delivering the passenger,
- -1 for each time step (to encourage efficiency),
- -10 for illegal pickup or dropoff actions.

Each episode terminates either when the passenger is successfully delivered or when a maximum number of steps (200) is reached. The latter condition prevents infinite trajectories and ensures bounded returns.

Taxi-v3 is deterministic, meaning that for each state-action pair, the next state is uniquely determined. This property simplifies the learning dynamics and makes the environment particularly suitable for analyzing tabular Q-learning behavior.

Introduction to reinforcement learning

Reinforcement Learning (RL) is a learning paradigm in which an agent interacts with an environment in order to maximize cumulative reward through trial-and-error experience. Unlike supervised learning, where labeled input-output pairs are provided, in RL the agent must discover which actions yield the highest long-term benefit by interacting with the environment and observing feedback in the form of rewards.

The interaction between the agent and the environment is typically modeled as a Markov Decision Process (MDP). At each discrete time step t, the agent:

- observes the current state $s_t \in \mathcal{S}$,
- selects an action $a_t \in \mathcal{A}$,
- receives a reward r_{t+1},
- transitions to a new state s_{t+1}.

The objective of the agent is to maximize the expected cumulative discounted reward, also called the return:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

where $\gamma \in [0, 1)$ is the discount factor. The discount factor determines the relative importance of future rewards compared to immediate rewards. A value

of γ close to 1 encourages long-term planning, while a smaller value makes the agent more short-sighted.

A central concept in Reinforcement Learning is the policy, denoted by π. A policy defines the behavior of the agent and specifies how actions are chosen given states:

$$\pi(a \mid s) = P(a_t = a \mid s_t = s)$$

The goal of learning is to find an optimal policy π^* that maximizes the expected return.

To evaluate how good a state or action is under a given policy, value functions are introduced. The state-value function is defined as:

$$V^{\pi}(s) = \mathbb{E}_{\pi}[G_t \mid s_t = s]$$

Similarly, the action-value function (or Q-function) is defined as:

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}[G_t \mid s_t = s, a_t = a]$$

The optimal action-value function $Q^*(s, a)$ satisfies the Bellman optimality equation:

$$Q^*(s, a) = r(s, a) + \gamma \max_{a' \in \mathcal{A}} Q^*(s', a')$$

where s' denotes the next state resulting from taking action a in state s.

This recursive relationship expresses the principle of optimality: the value of a state-action pair equals the immediate reward plus the discounted value of the best possible action in the next state.

In practice, the optimal Q-function is not known in advance and must be approximated through interaction with the environment. Q-learning is one such method, using a temporal-difference update rule to iteratively approximate the Bellman optimality equation. In the tabular case, the action-value function is stored explicitly as a table and updated after each interaction step.

From Theory to Implementation

The Bellman optimality equation provides a theoretical characterization of the optimal action-value function:

$$Q^*(s, a) = r(s, a) + \gamma \max_{a' \in \mathcal{A}} Q^*(s', a')$$

However, this equation alone does not provide a direct computational method, since the optimal function Q^* is unknown. In practice, we approximate this function iteratively through interaction with the environment.

In the tabular setting, the action-value function is represented explicitly as a matrix of size $\mid \mathcal{S} \mid \times \mid \mathcal{A} \mid$. Each row corresponds to a state, and each column corresponds to a possible action. The entry $Q(s, a)$ stores the current estimate of the expected cumulative discounted reward obtained by taking action a in state $s$and subsequently acting optimally.

Initially, all Q-values are set to zero, representing a complete lack of prior knowledge. During training, the agent interacts with the environment over multiple episodes. At each step, after observing the reward and the next state, the Q-value corresponding to the executed state-action pair is updated according to:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

This update progressively pushes the current estimate toward the value suggested by the Bellman equation. Over many episodes, information about successful trajectories propagates backward through the table, allowing the agent to approximate the optimal policy.

Exploration strategy

Since the Q-values are initially inaccurate, the agent must explore the environment. For this reason, an ϵ-greedy strategy is adopted:

$$\pi(a \mid s) = \begin{cases} \text{random action} & \text{with probability } \epsilon \\ \arg\max_a Q(s, a) & \text{with probability } 1 - \epsilon \end{cases}$$

The exploration parameter ϵ is progressively reduced during training in order to shift from exploration to exploitation. This ensures that early learning phases sufficiently cover the state-action space, while later phases focus on refining the learned policy.

Training Procedure

Training is performed over 10,000 episodes. At the beginning of each episode, the environment is reset and a new initial state is sampled. The agent then repeatedly:

1. Selects an action according to the ϵ-greedy policy.
2. Executes the action in the environment.
3. Observes the reward and the next state.

4. Updates the corresponding Q-table entry.

Each episode terminates either when the passenger is successfully delivered or when the maximum number of steps imposed by the environment is reached.

Implementation of the Tabular Q-Learning Agent

The tabular agent is implemented as a Python class that explicitly stores the action-value function $Q(s, a)$as a matrix with shape $\mid \mathcal{S} \mid \times \mid \mathcal{A} \mid$. In the Taxi-v3 environment, the state space is discrete with $\mid \mathcal{S} \mid = 500$and the action space has $\mid \mathcal{A} \mid = 6$actions. The Q-table is initialized to zero, meaning that initially the agent has no preference among actions.

Agent Initialization

The constructor receives the environment dimensions and learning hyperparameters. In particular:

- γ(discount factor) controls the importance of future rewards.
- ϵ controls exploration through an ϵ-greedy policy.
- ϵ_{decay} and ϵ_{min} reduce exploration over time while keeping a minimum probability of random actions.
- α(learning rate) controls how strongly new information updates the current estimate.

```
6   class Agent:
7
8       def __init__(self, n_states, n_actions, gamma=0.99,
9                    epsilon=1.0, epsilon_decay=0.999, epsilon_min=0.01):
10
11
12          self.n_states = n_states # 500
13          self.n_actions = n_actions # 6 = south, north, west, east, pickup, dropoff
14          self.gamma = gamma
15          self.epsilon = epsilon
16          self.epsilon_decay = epsilon_decay
17          self.epsilon_min = epsilon_min
18
19          # Initialize Q-table
20          self.Q = np.zeros((n_states, n_actions))
```

Figure 2: Agent initialization code

Action Selection: ϵ-Greedy Policy

The method select_action(state, training=True) implements the exploration strategy:

- during training, with probability ϵ, a random action is sampled uniformly from the discrete action set;

- otherwise, the greedy action is selected as the action with maximum estimated value in the current state:

$$a = \arg\max_{a \in \mathcal{A}} Q(s, a).$$

When training=False, exploration is disabled and the policy becomes purely greedy, which is appropriate for evaluation.

```
25      def select_action(self, state, training=True):
26
27          if training and np.random.random() < self.epsilon:
28              return np.random.randint(self.n_actions)
29          else:
30              return np.argmax(self.Q[state])
```

Figure 3: Action selection implementation

Q-table Update

After executing an action and observing $\left(s, a, r, s^{'}\right)$, the Q-table is updated according to the tabular Q-learning rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a^{'}} Q(s^{'}, a^{'}) - Q(s, a)].$$

This update moves the current estimate toward the Bellman optimality target $r + \gamma \max_{a^{'}} Q(s^{'}, a^{'})$, progressively improving the quality of the learned action-value function.

```
43      def update(self, state, action, reward, next_state):
44
45          td_target = reward + self.gamma * np.max(self.Q[next_state])
46          self.Q[state, action] = (1 - self.alpha) * self.Q[state, action] + self.alpha * td_target
```

Figure 4: Q-table update implementation

Exploration decay

The method decay_epsilon() updates the exploration rate after each episode:

$$\epsilon \leftarrow \max(\epsilon_{\min}, \epsilon \cdot \epsilon_{\text{decay}}),$$

allowing the agent to explore extensively in early episodes and gradually exploit the learned policy.

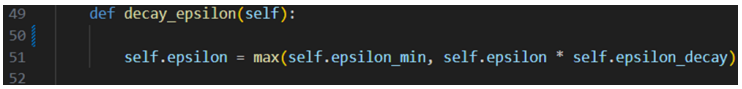


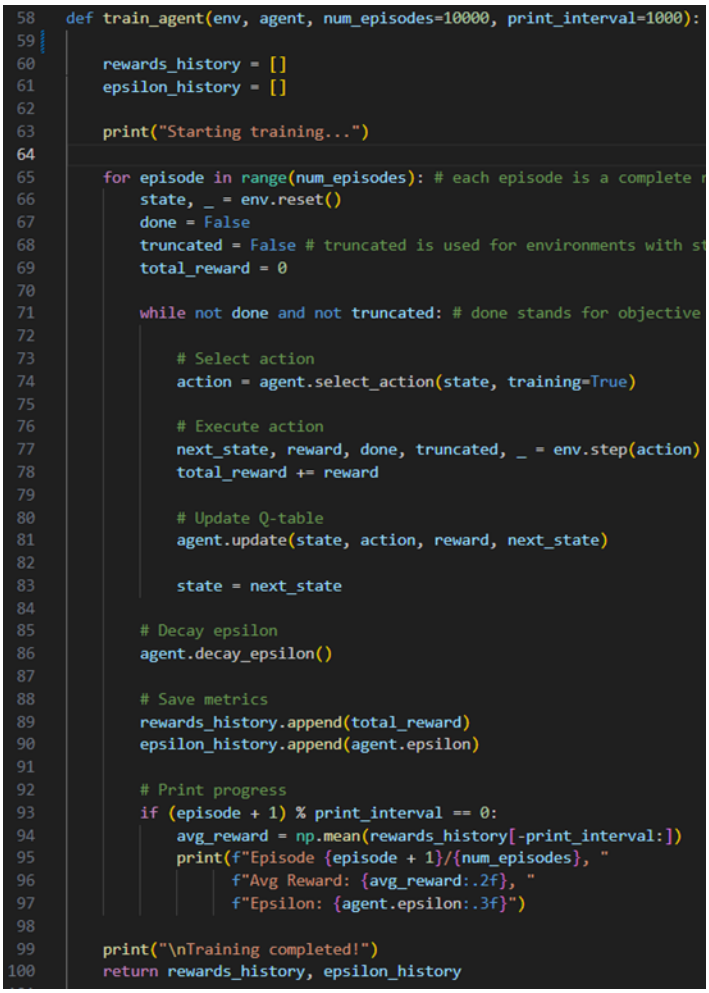
Figure 5: Epsilon decay implementation

Training Loop

The function train_agent(env, agent, num_episodes, print_interval) trains the agent for a fixed number of episodes. Each episode starts with env.reset() and proceeds until either the task is completed (terminated=True) or a step limit is reached (truncated=True). At each step:

1. an action is selected via select_action;
2. the environment transition is performed using env.step(action);
3. the Q-table is updated via update;
4. the next state becomes the current state.

The total reward per episode is recorded in rewards_history, while the evolution of ϵ is stored in epsilon_history. Periodically, an average reward over the last episodes is printed to monitor convergence.

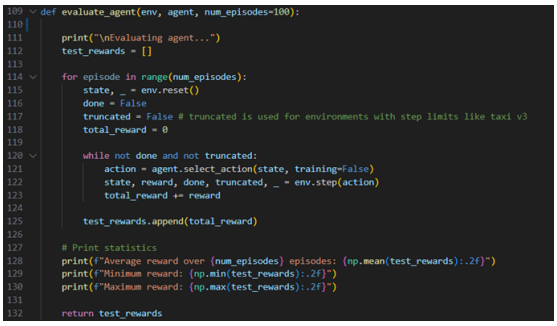


```python
def train_agent(env, agent, num_episodes=10000, print_interval=1000):

    rewards_history = []
    epsilon_history = []

    print("Starting training...")

    for episode in range(num_episodes): # each episode is a complete r
        state, _ = env.reset()
        done = False
        truncated = False # truncated is used for environments with st
        total_reward = 0

        while not done and not truncated: # done stands for objective

            # Select action
            action = agent.select_action(state, training=True)

            # Execute action
            next_state, reward, done, truncated, _ = env.step(action)
            total_reward += reward

            # Update Q-table
            agent.update(state, action, reward, next_state)

            state = next_state

        # Decay epsilon
        agent.decay_epsilon()

        # Save metrics
        rewards_history.append(total_reward)
        epsilon_history.append(agent.epsilon)

        # Print progress
        if (episode + 1) % print_interval == 0:
            avg_reward = np.mean(rewards_history[-print_interval:])
            print(f"Episode {episode + 1}/{num_episodes}, "
                  f"Avg Reward: {avg_reward:.2f}, "
                  f"Epsilon: {agent.epsilon:.3f}")

    print("\nTraining completed!")
    return rewards_history, epsilon_history
```

Figure 6: Training loop implementation

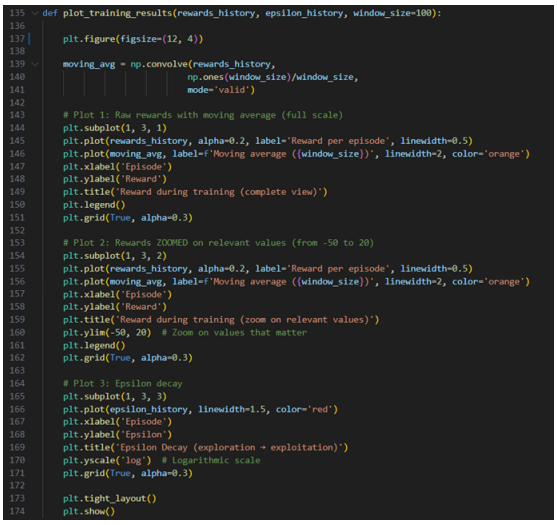
Evaluation and Visualization

The function evaluate_agent runs a set of evaluation episodes using the greedy policy (training=False) to estimate the final performance, reporting mean/min/max total rewards. The function plot_training_results visualizes learning dynamics by plotting episode rewards (with moving average smoothing) and the exploration decay curve. Finally, run_demo renders a single greedy episode to qualitatively inspect the learned behavior.



```python
def evaluate_agent(env, agent, num_episodes=100):

    print("\nEvaluating agent...")
    test_rewards = []

    for episode in range(num_episodes):
        state, _ = env.reset()
        done = False
        truncated = False # truncated is used for environments with step limits like taxi v3
        total_reward = 0

        while not done and not truncated:
            action = agent.select_action(state, training=False)
            state, reward, done, truncated, _ = env.step(action)
            total_reward += reward

        test_rewards.append(total_reward)

    # Print statistics
    print(f"Average reward over {num_episodes} episodes: {np.mean(test_rewards):.2f}")
    print(f"Minimum reward: {np.min(test_rewards):.2f}")
    print(f"Maximum reward: {np.max(test_rewards):.2f}")

    return test_rewards
```

Figure 7: Evaluation function implementation



```python
def plot_training_results(rewards_history, epsilon_history, window_size=100):

    plt.figure(figsize=(12, 4))

    moving_avg = np.convolve(rewards_history,
                             np.ones(window_size)/window_size,
                             mode='valid')

    # Plot 1: Raw rewards with moving average (full scale)
    plt.subplot(1, 3, 1)
    plt.plot(rewards_history, alpha=0.2, label='Reward per episode', linewidth=0.5)
    plt.plot(moving_avg, label=f'Moving average ({window_size})', linewidth=2, color='orange')
    plt.xlabel('Episode')
    plt.ylabel('Reward')
    plt.title('Reward during training (complete view)')
    plt.legend()
    plt.grid(True, alpha=0.3)

    # Plot 2: Rewards ZOOMED on relevant values (from -50 to 20)
    plt.subplot(1, 3, 2)
    plt.plot(rewards_history, alpha=0.2, label='Reward per episode', linewidth=0.5)
    plt.plot(moving_avg, label=f'Moving average ({window_size})', linewidth=2, color='orange')
    plt.xlabel('Episode')
    plt.ylabel('Reward')
    plt.title('Reward during training (zoom on relevant values)')
    plt.ylim(-50, 20)  # Zoom on values that matter
    plt.legend()
    plt.grid(True, alpha=0.3)

    # Plot 3: Epsilon decay
    plt.subplot(1, 3, 3)
    plt.plot(epsilon_history, linewidth=1.5, color='red')
    plt.xlabel('Episode')
    plt.ylabel('Epsilon')
    plt.title('Epsilon Decay (exploration → exploitation)')
    plt.yscale('log')  # Logarithmic scale
    plt.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.show()
```

Figure 8: Plot training results implementation

```python
def run_demo(env_name, agent):

    print("\nDemo of one episode (rendered):")

    env_render = gym.make(env_name, render_mode="human")

    state, _ = env_render.reset()
    done = False
    truncated = False
    total_reward = 0
    steps = 0

    while not done and not truncated:
        action = agent.get_greedy_action(state)
        state, reward, done, truncated, _ = env_render.step(action)
        total_reward += reward
        steps += 1

    print(f"\nEpisode completed in {steps} steps with total reward: {total_reward}")
    env_render.close()

```

Figure 9: Demo execution implementation

Experimental Results and Analysis

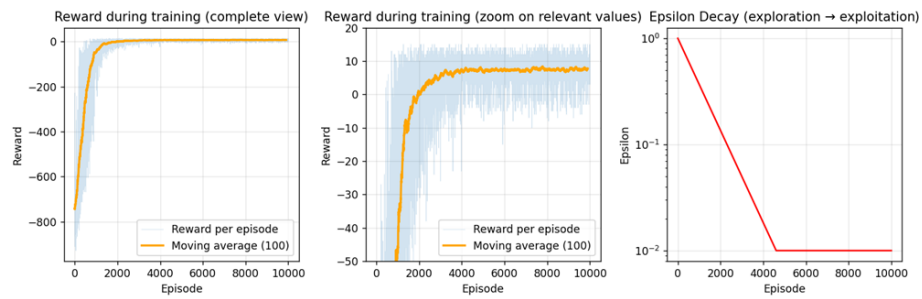


Figure 10: Training progress: reward evolution over 10,000 episodes

The training process was conducted for 10,000 episodes. The evolution of the average reward reveals a clear and structured learning progression.

During the initial phase (episodes 1-1000), the agent obtains extremely negative rewards, reaching values as low as -742 on average. This behavior is expected, as the exploration rate is still high ($\epsilon \approx 0.9$) and actions are mostly random. In this phase, the agent frequently performs illegal pickup and dropoff actions (-10 penalty) and often fails to complete the task within the step limit, accumulating large negative returns.

Episode 100/10000, Avg Reward: -742.15, Epsilon: 0.905

Episode 200/10000, Avg Reward: -668.14, Epsilon: 0.819

Episode 300/10000, Avg Reward: -550.35, Epsilon: 0.741

Episode 400/10000, Avg Reward: -453.34, Epsilon: 0.670

Episode 500/10000, Avg Reward: -369.38, Epsilon: 0.606

Episode 600/10000, Avg Reward: -268.34, Epsilon: 0.549

Episode 700/10000, Avg Reward: -190.51, Epsilon: 0.496

Episode 800/10000, Avg Reward: -137.31, Epsilon: 0.449

Episode 900/10000, Avg Reward: -99.17, Epsilon: 0.406

Episode 1000/10000, Avg Reward: -54.96, Epsilon: 0.368

Between episodes 1000 and 2500, the learning curve shows a steady improvement. The average reward gradually approaches zero and then becomes positive. This transition marks the point at which the agent begins to consistently complete the task. The decreasing exploration rate (ϵ decreasing from 0.36 to approximately

0.08) indicates a gradual shift from exploration toward exploitation of learned knowledge.

Episode 1000/10000, Avg Reward: -54.96, Epsilon: 0.368

Episode 1100/10000, Avg Reward: -46.07, Epsilon: 0.333

Episode 1200/10000, Avg Reward: -33.63, Epsilon: 0.301

Episode 1300/10000, Avg Reward: -24.68, Epsilon: 0.272

Episode 1400/10000, Avg Reward: -12.66, Epsilon: 0.246

Episode 1500/10000, Avg Reward: -8.42, Epsilon: 0.223

Episode 1600/10000, Avg Reward: -9.99, Epsilon: 0.202

Episode 1700/10000, Avg Reward: -3.92, Epsilon: 0.183

Episode 1800/10000, Avg Reward: -3.77, Epsilon: 0.165

Episode 1900/10000, Avg Reward: -1.41, Epsilon: 0.149

Episode 2000/10000, Avg Reward: -1.43, Epsilon: 0.135

Episode 2100/10000, Avg Reward: 0.60, Epsilon: 0.122

Episode 2200/10000, Avg Reward: 0.97, Epsilon: 0.111

Episode 2300/10000, Avg Reward: 1.56, Epsilon: 0.100

Episode 2400/10000, Avg Reward: 2.38, Epsilon: 0.091

Episode 2500/10000, Avg Reward: 3.78, Epsilon: 0.082

From approximately episode 3000 onward, the reward stabilizes in the positive region. After episode 4000, the agent consistently achieves average rewards above 6.5. Once ϵ reaches its minimum value (0.01), the learning curve becomes stable, oscillating around a plateau between 7.0 and 8.2.

Episode 3500/10000, Avg Reward: 6.20, Epsilon: 0.030

Episode 3600/10000, Avg Reward: 6.45, Epsilon: 0.027

Episode 3700/10000, Avg Reward: 5.89, Epsilon: 0.025

Episode 3800/10000, Avg Reward: 6.65, Epsilon: 0.022

Episode 3900/10000, Avg Reward: 7.06, Epsilon: 0.020

Episode 4000/10000, Avg Reward: 6.57, Epsilon: 0.018

Episode 5000/10000, Avg Reward: 7.77, Epsilon: 0.010

Episode 6000/10000, Avg Reward: 6.98, Epsilon: 0.010

Episode 7000/10000, Avg Reward: 6.97, Epsilon: 0.010

Episode 8000/10000, Avg Reward: 7.33, Epsilon: 0.010

Episode 9000/10000, Avg Reward: 6.99, Epsilon: 0.010

Episode 10000/10000, Avg Reward: 7.75, Epsilon: 0.010

At the end of training (episode 10,000), the average reward over the final interval is:

$$\text{Average reward} \approx 7.75$$

This indicates convergence toward a near-optimal policy.

Interpretation of the Learned Policy

In Taxi-v3, the reward structure is defined as:

- $+20$ for successful dropoff,
- -1 per time step,
- -10 for illegal pickup/dropoff.

An optimal trajectory typically requires between 12 and 14 steps, depending on the relative positions of the taxi, passenger, and destination. Therefore, the expected reward under an optimal policy can be approximated as:

$$20 - (\text{number of optimal steps}) \approx 6\text{–}8.$$

The observed training plateau ($\approx 7.7 - 8.2$) is fully consistent with this theoretical estimate, indicating that the agent has learned near-optimal trajectories and avoids unnecessary actions.

Evaluation Phase

After training, the agent was evaluated over 100 episodes using a purely greedy policy ($\epsilon = 0$).

The evaluation results were:

- Average reward: 7.93
- Minimum reward: 3.00
- Maximum reward: 13.00

The evaluation average (7.93) is slightly higher than the training plateau. This is expected because exploration is completely disabled during evaluation, eliminating occasional suboptimal exploratory actions.

The variability between minimum and maximum reward is due to random initial configurations. Since the passenger and destination positions are randomly sampled, the optimal path length varies. Short optimal paths yield higher

rewards (up to 13), while longer optimal paths yield lower but still positive rewards.

Importantly, no large negative rewards are observed during evaluation. This confirms that the agent has learned to avoid illegal actions and inefficient trajectories.

Convergence properties

The convergence behavior suggests that:

1. The Q-table has sufficiently explored and updated the relevant state-action pairs.
2. The learning rate α allows stable incremental updates.
3. The ϵ-decay schedule effectively balances exploration and exploitation.

The stabilization of performance after approximately 4000–5000 episodes indicates that the selected hyperparameters are appropriate for the Taxi-v3 environment.

Overall, the results demonstrate that tabular Q-learning successfully converges to a near-optimal policy in finite, deterministic environments with moderate state spaces.