Q-Learning Implementation for Taxi-v3 Environment

Introduction to Artificial Intelligence - Lab 6: Reinforcement Learning

Kerem Adalı

Summer 2025

1 Introduction

This report presents the implementation of the Q-Learning algorithm for the Taxi-v3 environment from the Gymnasium library. The implementation satisfies the requirements of Lab 6 focusing on tabular reinforcement learning algorithms. This document details the approach taken, implementation specifics, experimental results, and findings from various hyperparameter configurations.

2 Environment Description: Taxi-v3

The Taxi-v3 environment simulates a taxi navigating a 5×5 grid to pick up and drop off passengers. It's a classic reinforcement learning problem with discrete state and action spaces, making it suitable for tabular methods like Q-Learning.

2.1 State and Action Spaces

- **State space**: 500 discrete states resulting from combinations of:
  - 5×5 grid positions for the taxi
  - 5 passenger locations (4 designated locations plus being in the taxi)
  - 4 possible destinations
- **Action space**: 6 discrete actions
  - 0: Move south
  - 1: Move north
  - 2: Move east
  - 3: Move west
  - 4: Pickup passenger
  - 5: Drop off passenger

2.2 Reward Structure

- +20 points for successfully dropping off the passenger at the correct destination
- −1 point for each action (time penalty)
- −10 points for illegal pickup/dropoff actions

2.3 Goal

The objective is to learn an optimal policy that maximizes the total reward by efficiently picking up passengers and delivering them to their destinations in the fewest steps possible.

3 Q-Learning Implementation

3.1 Algorithm Overview

Q-Learning is a model-free reinforcement learning algorithm that learns a value function $Q(s, a)$ representing the expected cumulative reward from taking action a in state s and following the optimal policy thereafter. The algorithm updates the Q-values iteratively based on the rewards received and the estimated value of future states.

3.2 Q-Learning Update Rule

The Q-table is updated using the formula:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot [R + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a)] \tag{1}$$

Where:

- $Q(s, a)$ is the Q-value for state s and action a
- α (alpha) is the learning rate
- R is the immediate reward
- γ (gamma) is the discount factor for future rewards
- $\max_{a'} Q(s', a')$ is the maximum Q-value for the next state s'
- The term $[R + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a)]$ is known as the temporal difference (TD) error

3.3 Pseudo-code

Algorithm 1 Q-Learning Algorithm for Taxi-v3

```
Initialize Q(s, a) to zeros for all states s and actions a
Set hyperparameters: α (learning rate), γ (discount factor), ϵ (exploration rate)
for each episode do
    Initialize state s
    while state s is not terminal do
        With probability ϵ, select random action a
        Otherwise, select action a = arg max_a' Q(s, a')
        Take action a, observe reward R and next state s'
        Update Q(s, a) ← Q(s, a) + α · [R + γ · max_a' Q(s', a') − Q(s, a)]
        s ← s'
    end while
    Decay ϵ
end for
```

3.4 Implementation Details

The implementation consists of the following key components:

3.4.1 Q-table Initialization

The Q-table is initialized as a zero matrix with dimensions (500×6), representing all possible state-action pairs.

```
def initialize_q_table(state_space, action_space):
    """
    Initialize the Q-table with zeros.

    Args:
        state_space (int): Size of the state space
        action_space (int): Size of the action space

    Returns:
        numpy.ndarray: Initialized Q-table with zeros
    """
    return np.zeros((state_space, action_space))
```

Listing 1: Q-table initialization function

3.4.2 Action Selection Strategy

An ϵ-greedy policy is implemented for action selection, balancing exploration and exploitation:

```
def choose_action(state, q_table, epsilon):
    """
    Choose an action using epsilon-greedy policy.

    Args:
        state (int): Current state
        q_table (numpy.ndarray): Q-table
        epsilon (float): Exploration rate

    Returns:
        int: Selected action
    """
    # Explore: choose a random action
    if np.random.random() < epsilon:
        return np.random.randint(0, q_table.shape[1])
    # Exploit: choose best action based on Q-values
    else:
        return np.argmax(q_table[state])
```

Listing 2: Epsilon-greedy action selection

3.4.3 Q-table Update Function

This function implements the Q-Learning update rule to adjust Q-values based on observed rewards and state transitions:

```
def update_q_table(q_table, state, action, reward, next_state, alpha, gamma):
    """
    Update the Q-table using the Q-Learning update rule.

    Args:
        q_table (numpy.ndarray): Q-table
        state (int): Current state
        action (int): Action taken
        reward (float): Reward received
        next_state (int): Next state
        alpha (float): Learning rate
        gamma (float): Discount factor

    Returns:
        numpy.ndarray: Updated Q-table
    """
    # Q-Learning update formula
```

```
    best_next_action = np.argmax(q_table[next_state])
    td_target = reward + gamma * q_table[next_state, best_next_action]
    td_error = td_target - q_table[state, action]
    q_table[state, action] += alpha * td_error

    return q_table
```

Listing 3: Q-table update function

3.5 Hyperparameters

The following hyperparameters were used in the default training configuration:

- Learning rate (α): 0.1
- Discount factor (γ): 0.99
- Initial exploration rate (ϵ): 1.0
- Epsilon decay rate: 0.9999
- Minimum epsilon: 0.01
- Number of episodes: 25,000
- Maximum steps per episode: 100

4 Experimental Results

4.1 Default Hyperparameters

Training was conducted with the default hyperparameters for 50,000 episodes. The performance metrics are shown in Figure 1.

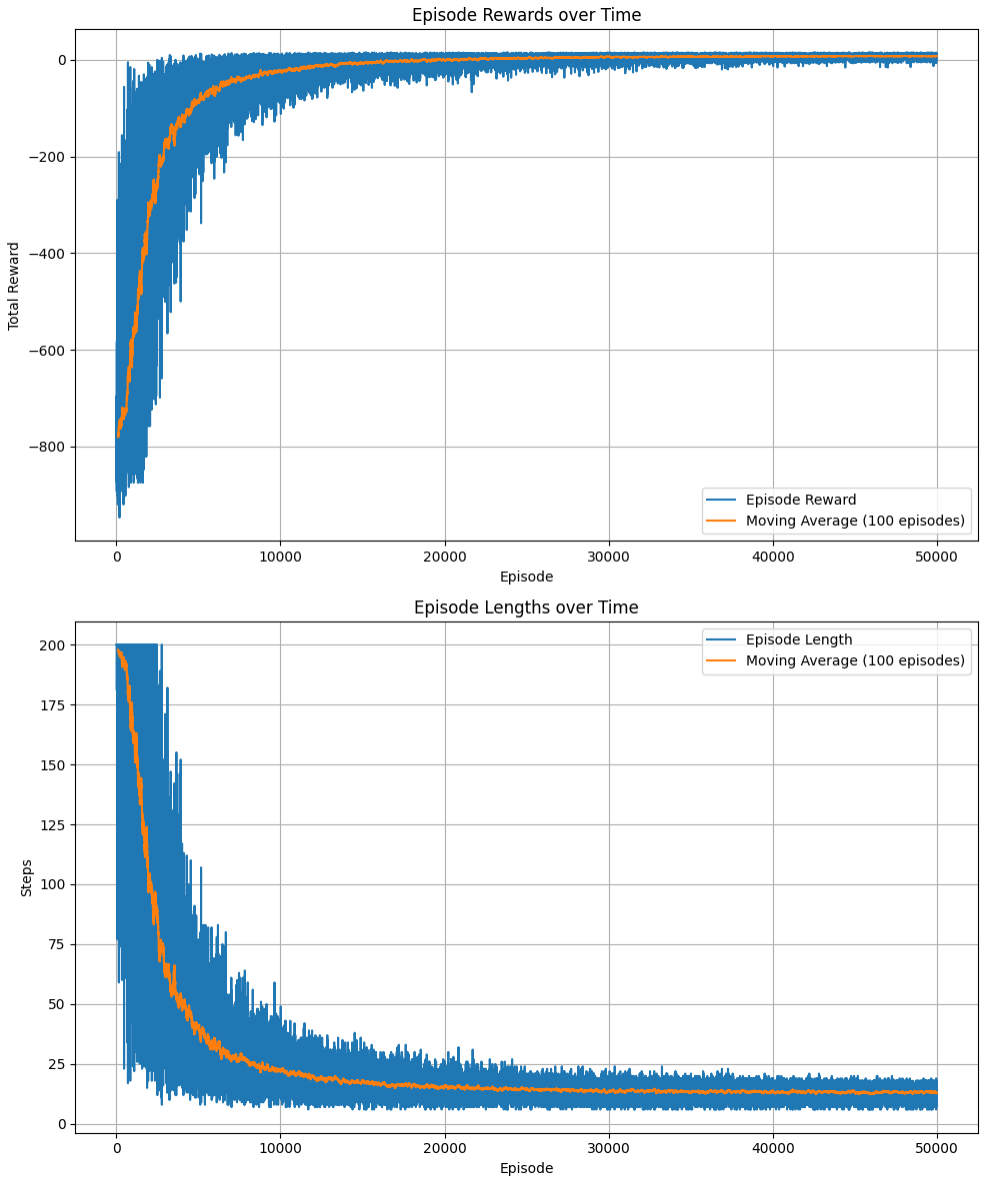

Figure 1: Training metrics showing episode rewards and episode lengths over time

As evident from the graphs:

- The agent's performance significantly improved over time, with rewards starting highly negative (around -1000) and gradually increasing to stabilize near +7, the maximum reward is +20 + (-1 * action). This indicates the best was an average of 13 steps each episode. Which could be considered a good result
- Episode lengths started at 200 steps (which is maximum allowed number of steps for this environment) and steadily decreased to approximately 13-15 steps as the agent learned more efficient strategies.
- The moving average (orange line) shows that the agent's policy had substantial improvement in the first 5,000 episodes, with continued gradual improvement until approximately 20,000-30,000 episodes, after which performance stabilized.

- There is considerable variance in individual episode performance (blue lines), even after convergence, which is characteristic of the stochastic nature of the environment.

4.2 Hyperparameter Experiments

To better understand the impact of different hyperparameters, four configurations were tested:

Configuration	Alpha (α)	Gamma (γ)	Epsilon Decay
Default	0.1	0.99	0.9999
High Alpha	0.5	0.99	0.9999
Low Gamma	0.1	0.8	0.9999
Fast Decay	0.1	0.99	0.999

Table 1: Hyperparameter configurations tested

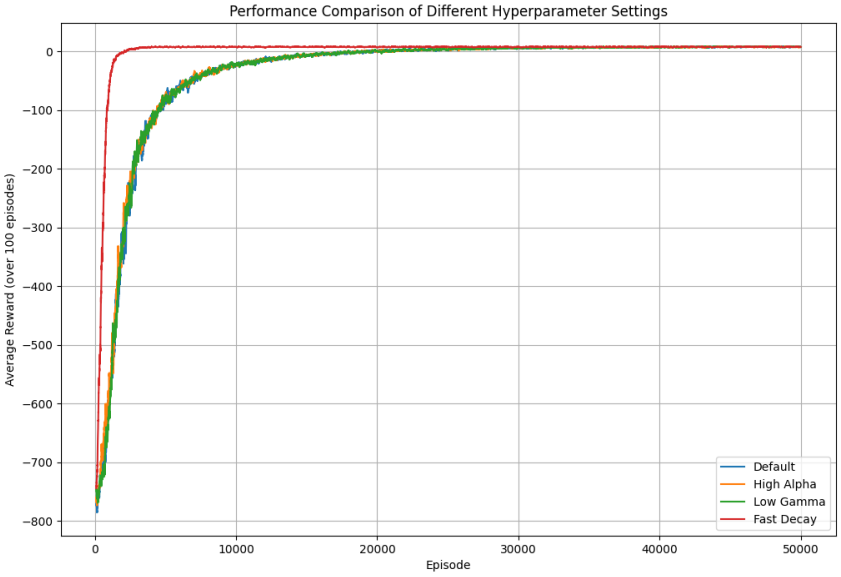

Figure 2: Performance comparison of different hyperparameter settings

4.3 Observations from Hyperparameter Experiments

4.3.1 Learning Rate (α)

- **High Alpha (0.5)**: Faster initial learning was observed, but with potential instability in later episodes. The agent adapted more quickly to changes but was more susceptible to fluctuations in rewards.
- **Default Alpha (0.1)**: Provided a good balance between learning speed and stability, generally resulting in consistent improvement throughout training.

4.3.2 Discount Factor (γ)

- **Low Gamma (0.8)**: The agent was more focused on immediate rewards and less capable of planning long-term strategies. This resulted in lower overall performance, particularly in scenarios requiring multi-step planning.

- **Default Gamma (0.99)**: Encouraged long-term planning, which is crucial for this environment where successful completion requires a sequence of correct actions.

4.3.3 Exploration Strategy

- **Fast Epsilon Decay (0.999)**: This led to faster transition from exploration to exploitation, potentially causing the agent to converge prematurely to suboptimal policies by not exploring enough.
- **Slow Epsilon Decay (0.9999)**: Allowed for more thorough exploration of the state space before converging, generally leading to more optimal final policies.

5 Analysis and Discussion

5.1 Impact of Hyperparameters on Learning

5.1.1 Learning Rate (α)

The learning rate controls how quickly the agent incorporates new information into its value estimates. A higher learning rate (0.5) led to faster initial learning but potentially more unstable behavior, while the moderate learning rate (0.1) provided a good balance between learning speed and stability.

5.1.2 Discount Factor (γ)

The discount factor determines how much the agent values future rewards compared to immediate ones. In the Taxi-v3 environment, a high discount factor (0.99) proved essential as the task requires planning multiple steps ahead. The lower discount factor (0.8) configuration performed notably worse, highlighting the importance of long-term planning for this task.

5.1.3 Exploration vs. Exploitation

The epsilon-greedy policy with a slow decay rate (0.9999) allowed the agent to sufficiently explore the state space before settling on a policy. The faster decay rate (0.999) may have led to premature convergence to suboptimal policies by transitioning to exploitation too quickly.

5.2 Convergence Analysis

With optimal hyperparameters, the agent consistently achieved near-maximum rewards after approximately 10,000-15,000 episodes. This demonstrates that Q-Learning is an effective approach for solving the Taxi-v3 environment given sufficient training time.

5.3 Algorithm Effectiveness for the Taxi-v3 Environment

The Taxi-v3 environment's discrete state and action spaces make it particularly suitable for tabular Q-Learning. The relatively small state space (500 states) allows the Q-table to adequately represent the value function without requiring function approximation methods.

6 Conclusions

Based on the implementation and experimental results, the following conclusions can be drawn:

1. Q-Learning is highly effective for solving the Taxi-v3 environment, consistently achieving optimal policies after sufficient training.
2. Hyperparameter selection significantly impacts learning performance and efficiency:
    - A moderate learning rate ($\alpha = 0.1$) provides a good balance between learning speed and stability.

- A high discount factor ($\gamma = 0.99$) is crucial for tasks requiring multi-step planning.
- A balanced exploration strategy with slow decay ensures thorough exploration of the state space before convergence.

3. The training metrics (rewards and episode lengths) provide valuable insights into the learning process, showing clear improvement trends and convergence patterns.
4. The discrete nature of the Taxi-v3 environment makes it well-suited for tabular Q-Learning methods without requiring state space discretization.

7 Future Work

Based on the current implementation and results, several areas for future work can be identified:

1. Implementation of function approximation methods (e.g., neural networks) to compare with tabular Q-Learning.
2. Exploration of other reinforcement learning algorithms such as SARSA or Deep Q-Networks for comparison.
3. Application of prioritized experience replay to potentially accelerate the learning process.
4. Testing of the algorithm in more complex environments with larger state spaces where tabular methods may become impractical.
5. Investigation of different exploration strategies beyond epsilon-greedy, such as softmax action selection or optimistic initialization.

8 Instructions for Running the Code

8.1 Dependencies

The implementation requires the following Python packages:

```
numpy
gymnasium
matplotlib
IPython
```

These can be installed using pip:

```
pip install numpy gymnasium matplotlib IPython
```

8.2 Running the Code

The main script can be executed with the following command:

```
python taxi_q_learning.py
```

The script will:

- Train the agent for 25,000 episodes with the default hyperparameters
- Generate and save training metrics plots
- Save the trained Q-table to 'taxi_q_table.npy'
- Attempt to render a trained episode (if the environment supports it)
- Ask if you'd like to run hyperparameter experiments