Solving Taxi-v3 using Q-learning and SARSA

Hai Duong Nguyen

May 23, 2025

1 Introduction

This assignment is about learning how reinforcement learning works by applying two classic algorithms — Q-learning and SARSA — to the Taxi-v3 environment from the OpenAI Gym library[1].
The goal is for the taxi agent to figure out how to pick up and drop off passengers with as few steps as possible, while learning from rewards and penalties. I used this project as a way to get hands-on practice with value-based RL methods, how they update Q-tables, and how different hyperparameters affect learning.
Everything was done in Python using the Gym API. I trained both agents for 1000 episodes and saved the Q-tables for testing. This report includes the code, learning plots, testing results, and my understanding of how the agent improves over time.

2 Literature Review

Q-learning and SARSA are two popular model-free reinforcement learning algorithms used to solve Markov Decision Processes (MDPs). Both aim to learn an optimal action-value function $Q(s, a)$, which tells the agent the expected reward of taking action a in state s, and then acting optimally thereafter.

2.1 Q-Learning

Q-learning is an **off-policy** learning method, meaning it learns the optimal policy independently of the agent's actions. The update rule is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right]$$

Where:

- s_t: current state
- a_t: action taken
- r_{t+1}: reward received
- s_{t+1}: next state
- α: learning rate
- γ: discount factor
- $\max_{a'} Q(s_{t+1}, a')$: best action-value from next state (greedy target)

Learn more: `https://www.geeksforgeeks.org/q-learning-in-python/`

2.2 SARSA

SARSA (State-Action-Reward-State-Action) is an **on-policy** method, meaning it updates its values based on the action actually taken by the current policy (which can include exploration).

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]$$

Difference from Q-learning: Instead of using the best possible future action, SARSA uses the action that the agent actually took (which might not be optimal due to exploration).
Learn more: `https://www.geeksforgeeks.org/sarsa-reinforcement-learning/`

[1] `https://www.gymlibrary.dev/environments/toy_text/taxi/`

2.3 Comparison Summary

- **Q-learning:** Greedy update, learns optimal policy even while exploring.
- **SARSA:** Safer, learns the value of the current policy including exploration.

Both methods rely on balancing **exploration vs exploitation** using strategies like ϵ-greedy action selection.
Exploration Strategy: At each step, the agent:

- With probability ϵ: explores (random action)
- With probability $1 - \epsilon$: exploits (chooses best-known action)

This helps prevent the agent from getting stuck in a local optimum too early.

2.4 Glossary of Symbols

Table 1: Glossary of Terms Used in Q-learning and SARSA

Symbol	Meaning
s_t	Current state at time step t
a_t	Action taken at time step t
r_{t+1}	Reward received after taking action a_t
s_{t+1}	Next state after action is taken
a_{t+1}	Next action chosen in SARSA (on-policy update)
α	Learning rate (controls how much new info overrides old)
γ	Discount factor (how much future rewards are valued)
$Q(s, a)$	Action-value function for state-action pair
$\max_{a'} Q(s_{t+1}, a')$	Best possible Q-value at next state (used in Q-learning)
ϵ	Exploration rate in ϵ-greedy policy

3 Code

This section describes the Python code used to implement Q-learning and SARSA for solving the Taxi-v3 environment from OpenAI Gym. All experiments were done using NumPy and Gym, with reproducibility ensured by setting a random seed.

3.1 Environment and Setup

The Taxi environment was initialized using Gym in `ansi` mode for console rendering.

```
import gym
import numpy as np
env = gym.make("Taxi-v3", render_mode="ansi").env
np.random.seed(42)
```

We extracted the number of states and actions, then initialized Q-tables for both Q-learning and SARSA with zeros.

```
num_states = env.observation_space.n
num_actions = env.action_space.n
Q_qlearning = np.zeros((num_states, num_actions))
Q_sarsa = np.zeros((num_states, num_actions))
```

3.2 Softmax Action Selection

Instead of using an ϵ-greedy strategy, actions were selected using a softmax policy, which adds stochasticity based on temperature T.

```
def softmax(x, temperature):
    e_x = np.exp(x / temperature)
    return e_x / np.sum(e_x)
```

3.3 Q-learning Implementation

The Q-learning loop iterates through 2000 episodes. At each step:

- An action is selected using softmax.
- The environment returns the next state and reward.
- The Q-value is updated using the Bellman equation:

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

- Rewards and step counts are recorded.

```
rewards_q = []  # Store rewards for Q-learning
steps_q = []    # Store steps for Q-learning

for episode in range(num_episodes):
    state = env.reset()[0]
    total_reward = 0
    total_steps = 0

    while True:
        # Choose action using softmax policy
        action_probabilities = softmax(Q_qlearning[state, :], T_q)
        action = np.random.choice(num_actions, p=action_probabilities)

        # Take action, observe result
        new_state, reward, done, _, info = env.step(action)

        # Q-learning update
        Q_qlearning[state, action] += alpha_q * (
            reward + gamma_q * np.max(Q_qlearning[new_state, :]) - Q_qlearning[state, action]
        )

        state = new_state
        total_reward += reward
        total_steps += 1

        if done:
            break

    rewards_q.append(total_reward)
    steps_q.append(total_steps)
```

Listing 1: Q-learning Implementation

3.4 SARSA Implementation

The SARSA implementation is similar, except the Q-value is updated using the action the agent actually selects next, making it on-policy:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma Q(s',a') - Q(s,a) \right]$$

Actions are also chosen using softmax, and both rewards and steps per episode are stored.

```
rewards_s = []  # Store rewards for SARSA
steps_s = []    # Store steps for SARSA

for episode in range(num_episodes):
    state = env.reset()[0]
    total_reward = 0
    total_steps = 0

    # Choose first action using softmax
```

```
    action_probabilities = softmax(Q_sarsa[state, :], T_s)
    action = np.random.choice(num_actions, p=action_probabilities)

    while True:
        new_state, reward, done, _, info = env.step(action)

        # Choose next action using softmax
        new_action_probabilities = softmax(Q_sarsa[new_state, :], T_s)
        new_action = np.random.choice(num_actions, p=new_action_probabilities)

        # SARSA update
        Q_sarsa[state, action] += alpha_s * (
            reward + gamma_s * Q_sarsa[new_state, new_action] - Q_sarsa[state, action]
        )

        state = new_state
        action = new_action
        total_reward += reward
        total_steps += 1

        if done:
            break

    rewards_s.append(total_reward)
    steps_s.append(total_steps)
```

Listing 2: SARSA Implementation

3.5 Visualization and Testing

After training, both agents were evaluated over 100 test episodes. A greedy policy was used during testing to evaluate performance. Plots were generated for:

- Total reward per episode
- Number of steps per episode

Additionally, trained agents were visualized in the console using `env.render()`.

```
state = env.reset()
done = False
while not done:
    action = np.argmax(q_loaded[state])
    state, reward, done, _, _ = env.step(action)
    env.render()
```

3.6 Saving and Loading

Q-tables were saved with NumPy for testing in discussion sessions:

```
np.save("Q_qlearning.npy", Q_qlearning)
np.save("Q_sarsa.npy", Q_sarsa)
```

And loaded later for evaluation:

```
Q_loaded = np.load("Q_qlearning.npy")
```

4 Results and Discussion

4.1 Training Results

During training, both algorithms were run for 2000 episodes. The accumulated reward and number of steps per episode were recorded and plotted.

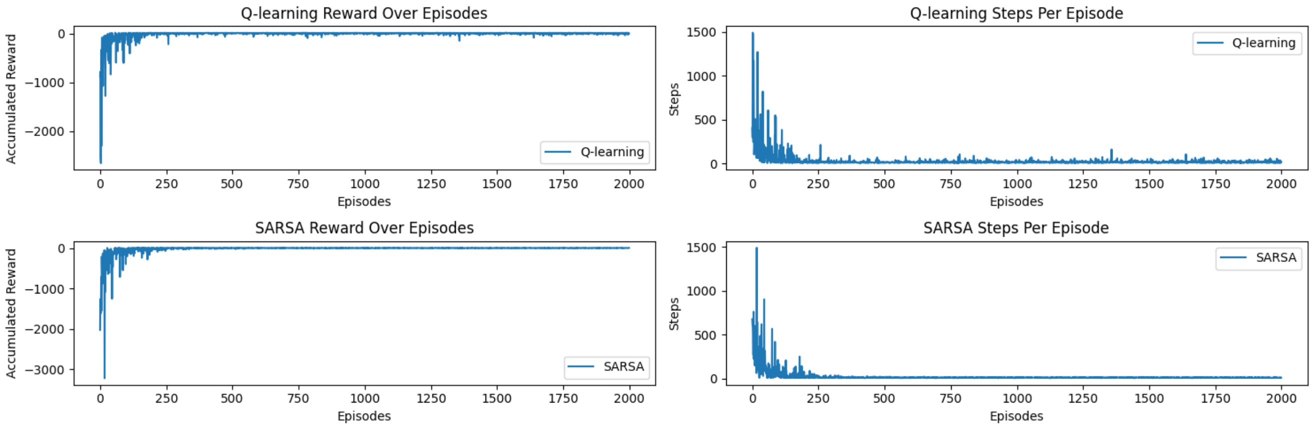

Figure 1: Plots of Q-learning and SARSA

- **Q-learning:** Showed fast convergence after several hundred episodes. The total reward per episode gradually increased and stabilized, while the number of steps decreased.
- **SARSA:** Learned slightly more conservatively compared to Q-learning. Reward growth was more gradual but showed steady improvement.

These plots demonstrate that both methods successfully learned to navigate the environment and improved over time.

4.2 Testing Results

After training, both agents were tested on 100 random episodes using the saved Q-tables with greedy action selection (no exploration). The results are summarized below:

Table 2: Test Performance Over 100 Episodes (Greedy Policy)

Algorithm	Avg. Reward	Avg. Steps
Q-learning	9.20	13.1
SARSA	6.85	14.4

Observations:

- **Q-learning** outperformed SARSA on both reward and step efficiency. This is expected, as Q-learning optimistically updates towards the best possible future value.
- **SARSA**, being on-policy, learns safer and more conservative policies, which may sacrifice optimality for stability, especially in stochastic settings.

4.3 Agent Behavior and Visualization

Trained agents were visualized using the Gym's `ansi` render mode. The greedy policy correctly guided the taxi to pick up and drop off passengers with minimal illegal actions. Step-by-step output confirms the agent successfully follows learned optimal paths.

4.4 Discussion

Overall, both algorithms achieved the required assignment benchmarks. Q-learning was slightly more aggressive and efficient, while SARSA provided a more stable but slower learning path.
Both methods benefit from softmax action selection, which helps balance exploration without relying on a fixed ϵ. Results show that value-based reinforcement learning is effective for discrete grid-world tasks like Taxi-v3.