Lab Assignment: Reinforcement learning

Lab team: J04

Name (member 1): Adam Maltoni

Name (member 2): Ibón de Mingo Arroyo

Training a reinforcemnt learning agent at the gymnasium

Exercise 1: The gymnasium

- Basic usage: https://gymnasium.farama.org/introduction/basic_usage/
- Training an RL-agent: https://gymnasium.farama.org/introduction/train_agent/

```
%load_ext autoreload
%autoreload 2

import numpy as np
import matplotlib.pyplot as plt
import gymnasium as gym
import imageio
import os

from tqdm.notebook import tqdm
import rl_utils
import time
from IPython.display import display, clear_output


from reinforcement_learning import (
    sarsa_learning,
    q_learning,
    greedy_policy,
    epsilon_greedy_policy,
)
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

```
# Install packages if needed
# !pip install pygame
# !pip install gymnasium

import gymnasium as gym
env = gym.make('CartPole-v1')
```

```
env = gym.make("LunarLander-v3", render_mode="human")
observation, info = env.reset()

episode_over = False
while not episode_over:
    action = env.action_space.sample()  # agent policy that uses the
observation and info
    observation, reward, terminated, truncated, info =
env.step(action)

    episode_over = terminated or truncated

env.close()
```

Exercise 2: Q-learning

We're now ready to code our Q-Learning algorithm

Step 0: Set up and understand Frozen Lake environment

Let's begin with a simple 4x4 map and non-slippery, meaning the agent always moves in the desired direction.

We add a parameter called `render_mode` that specifies how the environment should be visualised. In our case because we **want to record a video of the environment at the end, we need to set render_mode to rgb_array**.

As explained in the documentation "rgb_array": Return a single frame representing the current state of the environment. A frame is a np.ndarray with shape (H, W, 3) representing RGB values for an H (height) times W (width) pixel image.

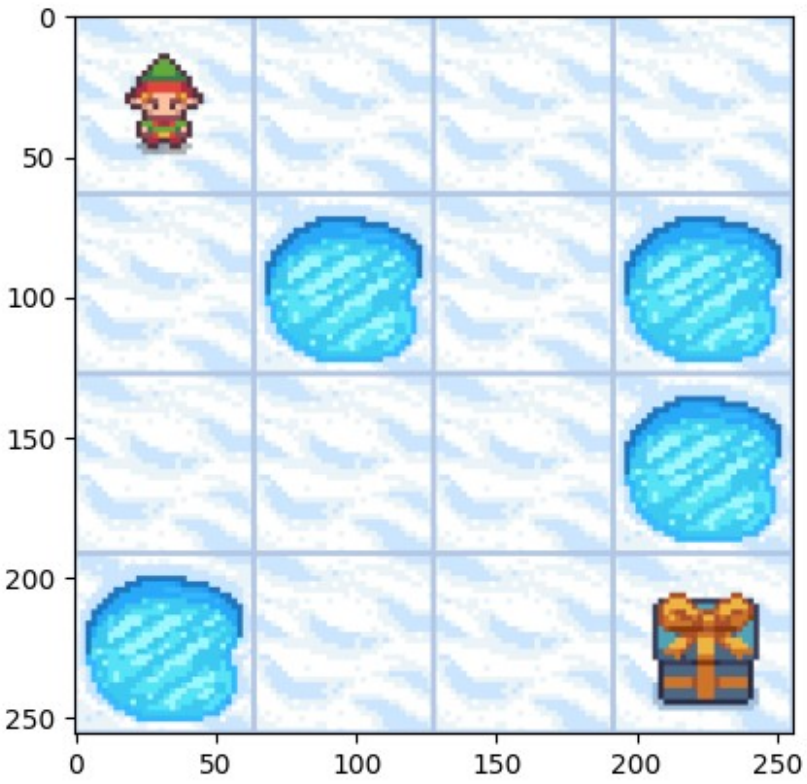
```
small_environment = gym.make(
    'FrozenLake-v1',
    map_name='4x4',
    is_slippery=False,
    render_mode='rgb_array',
)
```

Let's see what the environment looks like:

```
state, info = small_environment.reset()  # observation state
action_names = {0: 'Left', 1: 'Down', 2: 'Right', 3: 'Up'}

print(state)
print(info, '(Probability that the action has led to the current
state)')
n_states = small_environment.observation_space.n
print("There are ", n_states, " possible states")
```

```
n_actions = small_environment.action_space.n
print("There are ", n_actions, " possible actions")
print(action_names)

fig, ax = plt.subplots()
game_image = ax.imshow(small_environment.render())
```

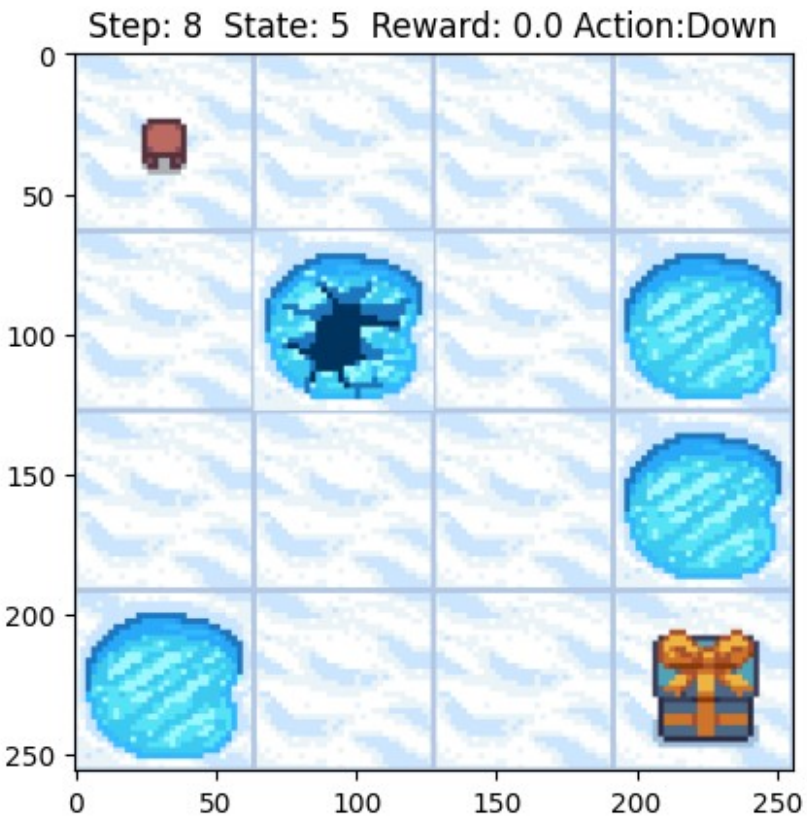
```
0
{'prob': 1} (Probability that the action has led to the current state)
There are  16  possible states
There are  4  possible actions
{0: 'Left', 1: 'Down', 2: 'Right', 3: 'Up'}
```




```
# Generate a random observed state
print("Observation state (randomly selected)",
small_environment.observation_space.sample())

# Generate a random action from the current state
print("Action (randomly selected):",
small_environment.action_space.sample())
```

```
Observation state (randomly selected) 7
Action (randomly selected): 3
```

```
#  Generate an episode

obsevation, info = small_environment.reset()  # state state_state
fig, ax = plt.subplots()
game_image = ax.imshow(small_environment.render())


MAX_STEPS = 100
refresh_rate = 1 # in (1 / seconds)

episode_over = False
n_steps = 0

while not episode_over and n_steps < MAX_STEPS:
    n_steps += 1
    action = small_environment.action_space.sample()

    state, reward, terminated, truncated, info =
small_environment.step(action)
    episode_over = terminated or truncated

    ax.set_title(
        'Step: {}  State: {}  Reward: {} Action:{}'.format(
            n_steps,
            state,
            reward,
            action_names[action],
        )
    )

    display(fig)
    time.sleep(1.0 / refresh_rate)
    clear_output(wait=True)  # Clear previous output
    game_image.set_data(small_environment.render())


small_environment.close()
```



Step 1: Greedy and Epsilon greedy policies

Since Q-Learning is an **off-policy** algorithm, we have two policies. This means we're using a **different policy for acting and updating the value function**.

- Epsilon-greedy policy (acting policy)
- Greedy-policy (updating policy)

The greedy policy will also be the final policy we'll have when the Q-learning agent completes training. The greedy policy is used to select an action using the Q-table.

Epsilon-greedy is the training policy that handles the exploration/exploitation trade-off.

- With *probability* $1-\varepsilon$: **we do exploitation** (i.e. our agent selects the action with the highest state-action pair value).
- With *probability* ε: we do **exploration** (trying a random action).

As the training continues, we progressively **reduce the epsilon value since we will need less and less exploration and more exploitation.**

Step 2: Train the RL agent

TO DO: Implement the Q-learning algorithm in `reinforcement_learning.py`

Define the hyperparameters for the learning process ⚙

The exploration related hyperparamters are some of the most important ones.

- We need to make sure that our agent **explores enough of the state space** to learn a good value approximation. To do that, we need to have progressive decay of the epsilon.
- If you decrease epsilon too fast (too high decay_rate), **you take the risk that your agent will be stuck** in a local optimum, since your agent didn't explore enough of the state space and hence can't solve the problem.

```
# Training hyperparameters

n_training_episodes = 1000
max_steps = 100 # Maximum number of steps per episode
learning_rate = 0.7
gamma = 0.95 # Discount factor

# Exploration parameters
max_epsilon = 1.0  # Initial exploration probability
min_epsilon = 0.05  # Minimum exploration probability
decay_rate = 0.0005 # Exponential decay rate for the exploration
probability

# Initialize Q-table
Qtable_small = np.zeros(
    (
        small_environment.observation_space.n,
        small_environment.action_space.n
    )
)


# Learn Q-table
Qtable_small = q_learning(
    small_environment,
    n_training_episodes,
    max_steps,
    learning_rate,
    gamma,
    min_epsilon,
    max_epsilon,
    decay_rate,
    Qtable_small,
)
```

{"model_id":"c6144c1982024ab6a9a661bad911102e","version_major":2,"version_minor":0}

Let's see what our Q-Learning table looks like now

```
Qtable_small
```

```
array([[0.73509189, 0.77378094, 0.77378094, 0.73509189],
       [0.73509189, 0.        , 0.81450625, 0.77378094],
       [0.77378094, 0.857375  , 0.77378094, 0.81450625],
       [0.81450625, 0.        , 0.77378073, 0.77378084],
       [0.77378094, 0.81450625, 0.        , 0.73509189],
       [0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.9025    , 0.        , 0.81450625],
       [0.        , 0.        , 0.        , 0.        ],
       [0.81450625, 0.        , 0.857375  , 0.77378094],
       [0.81450625, 0.9025    , 0.9025    , 0.        ],
       [0.857375  , 0.95      , 0.        , 0.857375  ],
       [0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.9025    , 0.95      , 0.857375  ],
       [0.9025    , 0.95      , 1.        , 0.9025    ],
       [0.        , 0.        , 0.        , 0.        ]])
```

...and what our agent is doing!

```
frames = rl_utils.generate_greedy_episode(small_environment,
Qtable_small)
rl_utils.show_episode(frames, interval=250)
```

```
<IPython.core.display.HTML object>
```

A more challenging problem

We're ready now to find our way in more challenging environments

```
large_environment = gym.make(
    'FrozenLake-v1',
    map_name='8x8',
    is_slippery=False,
    render_mode='rgb_array'
)

n_states = large_environment.observation_space.n
print("There are ", n_states, " possible states")

n_actions = large_environment.action_space.n
print("There are ", n_actions, " possible actions")
```

```
There are  64  possible states
There are  4  possible actions
```

```
n_training_episodes = 30000
max_steps = 500
learning_rate = 0.5
gamma = 0.99

max_epsilon = 1.0
min_epsilon = 0.2
decay_rate = 0.0001

# Initialize Q-table
Qtable_large = np.zeros(
    (
        large_environment.observation_space.n,
        large_environment.action_space.n
    )
)

# Learn Q-table
Qtable_large = q_learning(
    large_environment,
    n_training_episodes,
    max_steps,
    learning_rate,
    gamma,
    min_epsilon,
    max_epsilon,
    decay_rate,
    Qtable_large,
)

print(Qtable_large)
```

{"model_id":"e5aed05465c34b99897d51bb0f7ac1a0","version_major":2,"version_minor":0}

```
[[0.86874581 0.87752102 0.87752102 0.86874581]
 [0.86874581 0.88638487 0.88638487 0.87752102]
 [0.87752102 0.89533825 0.89533825 0.88638487]
 [0.88638487 0.90438208 0.90438208 0.89533825]
 [0.89533825 0.91351725 0.91351725 0.90438208]
 [0.90438208 0.92274469 0.92274469 0.91351725]
 [0.91351725 0.93206535 0.93206535 0.92274469]
 [0.92274469 0.94148015 0.93206535 0.93206535]
 [0.87752102 0.88638487 0.88638487 0.86874581]
 [0.87752102 0.89533825 0.89533825 0.87752102]
 [0.88638487 0.90438208 0.90438208 0.88638487]
 [0.89533825 0.         0.91351725 0.89533825]
```

```
[0.90438208 0.92274469 0.92274469 0.90438208]
[0.91351725 0.93206535 0.93206535 0.91351725]
[0.92274469 0.94148015 0.94148015 0.92274469]
[0.93206535 0.95099005 0.94148015 0.93206535]
[0.88638487 0.89533825 0.89533825 0.87752102]
[0.88638487 0.90438208 0.90438208 0.88638487]
[0.89533825 0.91351725 0.         0.89533825]
[0.         0.         0.         0.        ]
[0.         0.93206535 0.93206535 0.91351725]
[0.92274469 0.         0.94148015 0.92274469]
[0.93206535 0.95099005 0.95099005 0.93206535]
[0.94148015 0.96059601 0.95099005 0.94148015]
[0.89533825 0.88638487 0.90438208 0.88638487]
[0.89533825 0.89533825 0.91351725 0.89533825]
[0.90438208 0.90438208 0.92274469 0.90438208]
[0.91351725 0.         0.93206535 0.        ]
[0.92274469 0.94148015 0.         0.92274469]
[0.         0.         0.         0.        ]
[0.         0.96059601 0.96059601 0.94148015]
[0.95099005 0.970299   0.96059601 0.95099005]
[0.88638487 0.87752102 0.89533825 0.89533825]
[0.88638487 0.         0.90438208 0.90438208]
[0.89533825 0.         0.         0.91351725]
[0.         0.         0.         0.        ]
[0.         0.93206535 0.95099005 0.93206535]
[0.94148015 0.94148015 0.96059601 0.        ]
[0.95099005 0.         0.970299   0.95099005]
[0.96059601 0.9801     0.970299   0.96059601]
[0.87752102 0.86874581 0.         0.88638487]
[0.         0.         0.         0.        ]
[0.         0.         0.         0.        ]
[0.         0.         0.93206535 0.        ]
[0.92274469 0.         0.94148015 0.94148015]
[0.93206535 0.93206535 0.         0.95099005]
[0.         0.         0.         0.        ]
[0.         0.99       0.9801     0.970299  ]
[0.86873653 0.86005348 0.         0.87752102]
[0.         0.         0.         0.        ]
[0.         0.         0.         0.        ]
[0.         0.         0.         0.        ]
[0.         0.         0.         0.        ]
[0.         0.92274315 0.         0.94148015]
[0.         0.         0.         0.        ]
[0.         1.         0.99       0.9801    ]
[0.85926946 0.85813886 0.75974737 0.86874551]
[0.84209527 0.44299506 0.         0.        ]
[0.         0.         0.         0.        ]
[0.         0.         0.         0.        ]
[0.         0.         0.         0.        ]
```

```
[0.         0.85423861 0.2475     0.93206524]
[0.         0.         0.75       0.        ]
[0.         0.         0.         0.        ]]
```

```
frames = rl_utils.generate_greedy_episode(large_environment,
Qtable_large)
rl_utils.show_episode(frames, interval=250)
```

Slippery environment

Our environment is now slippery, meaning the agent sometimes slips and moves in an unintended direction

```
slippery_environment = gym.make(
    'FrozenLake-v1',
    map_name='8x8',
    is_slippery=True,
    render_mode='rgb_array',
)
```

To visualize the challenges in this environment, let's make our agent go right several times and see what happens:

```
go_right = []
action = 2  # go-right
n_steps = 20
state, info = slippery_environment.reset()
for i in range(n_steps):
    go_right.append(slippery_environment.render())  # Capture current
frame (RGB array)
    slippery_environment.step(action)

rl_utils.show_episode(go_right, interval=250)
```

Let's see what happens when we train our agent in the same way as before.

```
# TO DO: Training hyperparameters
n_training_episodes = 30000
max_steps = 700 # Maximum number of steps per episode
learning_rate =  0.7
gamma =  0.99 # Discount factor

# Exploration parameters
max_epsilon =  1 # Initial exploration probability
min_epsilon =  0.1 # Minimum exploration probability
decay_rate =  0.00001# Exponential decay rate for the exploration
probability
```

```
# Initialize Q-table
Qtable_slippery = np.zeros(
    (
        slippery_environment.observation_space.n,
        slippery_environment.action_space.n
    )
)

# Learn Q-table
Qtable_slippery = q_learning(
    slippery_environment,
    n_training_episodes,
    max_steps,
    learning_rate,
    gamma,
    min_epsilon,
    max_epsilon,
    decay_rate,
    Qtable_slippery,
)
```

{"model_id":"98919dbc9b3f4d539af4c25c71cef279","version_major":2,"version_minor":0}

```
frames = rl_utils.generate_greedy_episode(slippery_environment, Qtable_slippery)
rl_utils.show_episode(frames, interval=250)
```

Now we fix it

```
# TO DO: Training hyperparameters
n_training_episodes = 100000
max_steps = 800 # Maximum number of steps per episode
learning_rate =  0.1
gamma =  0.995 # Discount factor

# Exploration parameters
max_epsilon =  1 # Initial exploration probability
min_epsilon =  0.5 # Minimum exploration probability
decay_rate =  0.00001# Exponential decay rate for the exploration probability

# Initialize Q-table
Qtable_slippery = np.zeros(
    (
        slippery_environment.observation_space.n,
        slippery_environment.action_space.n
    )
)
```

```
# Learn Q-table
Qtable_slippery = q_learning(
    slippery_environment,
    n_training_episodes,
    max_steps,
    learning_rate,
    gamma,
    min_epsilon,
    max_epsilon,
    decay_rate,
    Qtable_slippery,
)
```

{"model_id":"b30b2d46c2e24f15b33155c7eb2fc2a8","version_major":2,"version_minor":0}

```
frames = rl_utils.generate_greedy_episode(slippery_environment, Qtable_slippery)
rl_utils.show_episode(frames, interval=250)
```

```
<IPython.core.display.HTML object>
```

Exercise 3: Train the RL agent using SARSA

TO DO: Implement the SARSA learning algorithm in `reinforcement_learning.py`

```
slippery_environment = gym.make(
    'FrozenLake-v1',
    map_name='8x8',
    is_slippery=True,
    render_mode='rgb_array',
)

n_training_episodes = 100000
max_steps = 800
learning_rate = 0.1
gamma = 0.995

max_epsilon = 1.0
min_epsilon = 0.5
decay_rate = 0.0001

# Initialize Q-table
Qtable_slippery = np.zeros(
    (
        slippery_environment.observation_space.n,
        slippery_environment.action_space.n
    )
```

```
)

# Learn Q-table
Qtable_slippery = sarsa_learning(
    slippery_environment,
    n_training_episodes,
    learning_rate,
    gamma,
    min_epsilon,
    max_epsilon,
    decay_rate,
    max_steps,
    Qtable_slippery,
)
```

{"model_id":"b75998960efa4e3895b7f121a6e30f81","version_major":2,"version_minor":0}

```
frames = rl_utils.generate_greedy_episode(slippery_environment, Qtable_slippery)
rl_utils.show_episode(frames, interval=250)
```

```
<IPython.core.display.HTML object>
```

Exercise 4: Compare SARSA and Q-learning

Answer these questions:

- What is "episode reward" in RL and how is it related to the performance of an RL agent?
- What is "episode length" in RL and how is it related to the performance of an RL agent?
- What is "training error" in RL and how is it related to the performance of an RL agent?
  - Provide an explicit expression of the training error in the cases explored

What is *episode reward* in RL and how is it related to the performance of an RL agent?

Answer: In Reinforcement Learning (RL), the **episode reward** is the total cumulative reward that an agent receives during a single episode — from the initial state to a terminal state.

$$R^{(i)} = \sum_{t=0}^{T_i} r_t$$

Where:

- T_i is the total number of time steps in episode i

- r_t is the reward received at time step t

A higher episode reward generally indicates better performance. For example, in environments like FrozenLake, an agent receives a reward of 1 for reaching the goal. So, an episode reward of 1 means success, while 0 means failure. Tracking the average episode reward over time helps us understand how well the agent is learning.

What is *episode length* in RL and how is it related to the performance of an RL agent?

Answer: The **episode length** is the number of steps taken before the episode ends.

$$\text{Episode Length} = T$$

Where T is the total number of steps in an episode.

Episode length must be interpreted alongside the episode reward:

- **Short + reward = 1** → efficient path to goal
- **Long + reward = 1** → success but with a suboptimal path
- **Short + reward = 0** → early failure (e.g., fell into a hole)
- **Long + reward = 0** → wandering without reaching the goal

In general, **shorter episodes with higher rewards** indicate that the agent is learning an effective policy.

What is *training error* in RL and how is it related to the performance of an RL agent?

Answer: The **training error** in RL typically refers to the **Temporal Difference (TD) error**, which quantifies how much the Q-value estimate changes during learning.

In **Q-learning**, the TD error is:

$$\delta_t = r_t + \gamma \cdot \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)$$

In **SARSA**, the TD error is:

$$\delta_t = r_t + \gamma \cdot Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

Where:

- r_t is the reward at time step t
- γ is the discount factor
- $Q(s, a)$ is the Q-value of state s and action a

The **average training error per episode** can be calculated as:

$$\text{Training Error} = \frac{1}{T}\sum_{t=0}^{T}\left|\delta_t\right)$$

- A **high training error** indicates that the agent is still learning (i.e., Q-values are changing significantly).
- A **low training error** suggests that the agent's policy is stabilizing, which often correlates with improved and consistent performance.

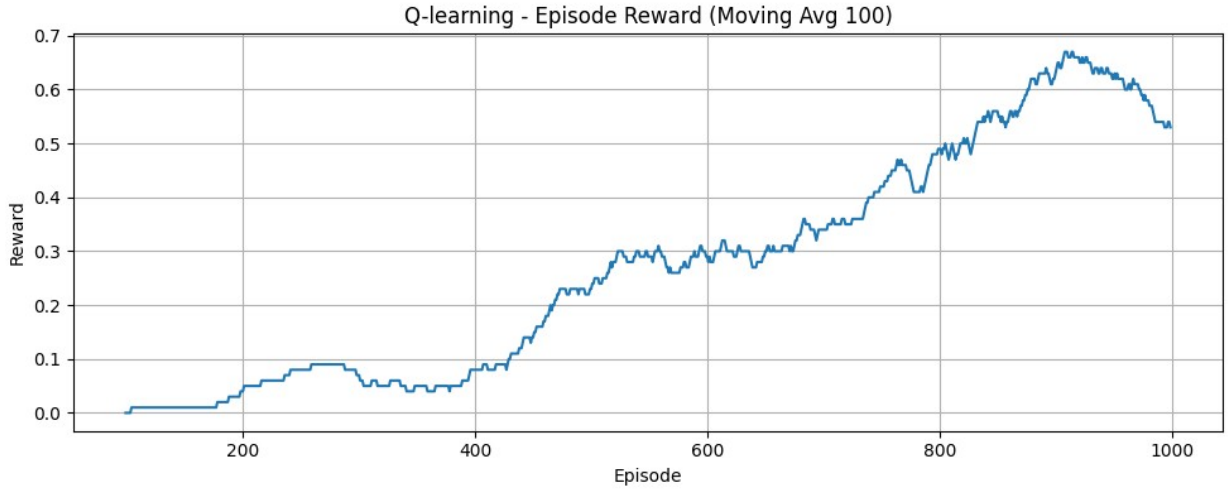
Exercise 5: Compare SARSA and Q-learning

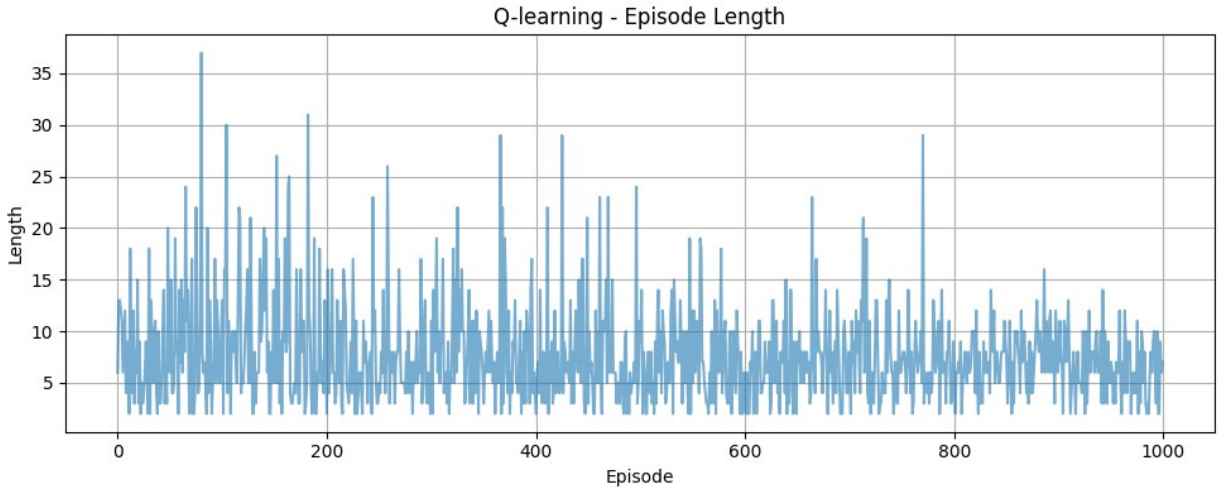
- Use matplotlib to compare the learning curves of SARSA and Q-learning, in terms of
  - episode reward.
  - episode length.
  - training error
- Discuss the results obtained.

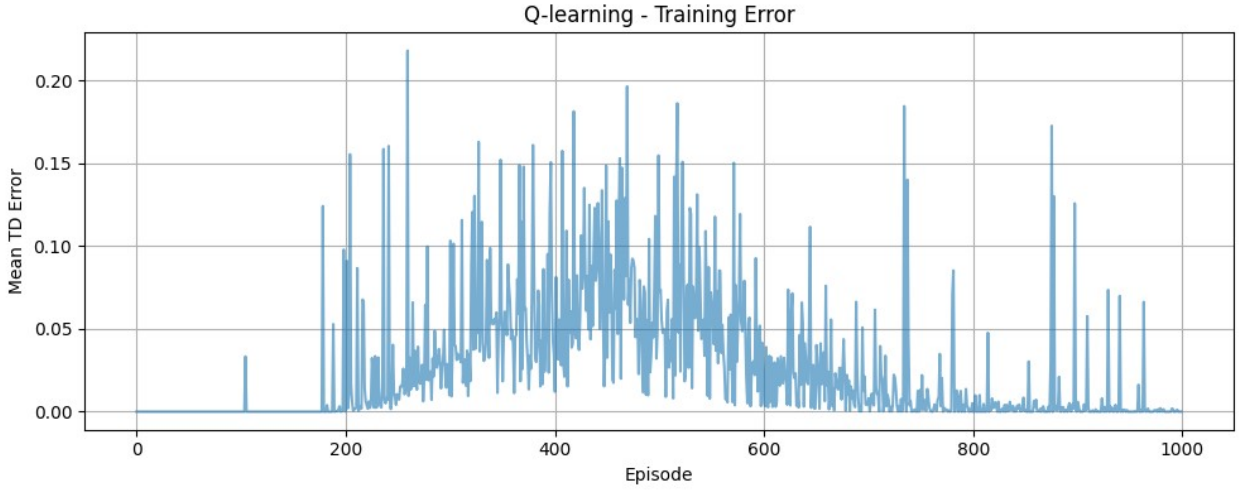
```
from rl_comparatives import *

from ex5 import plot1

plot1()
```

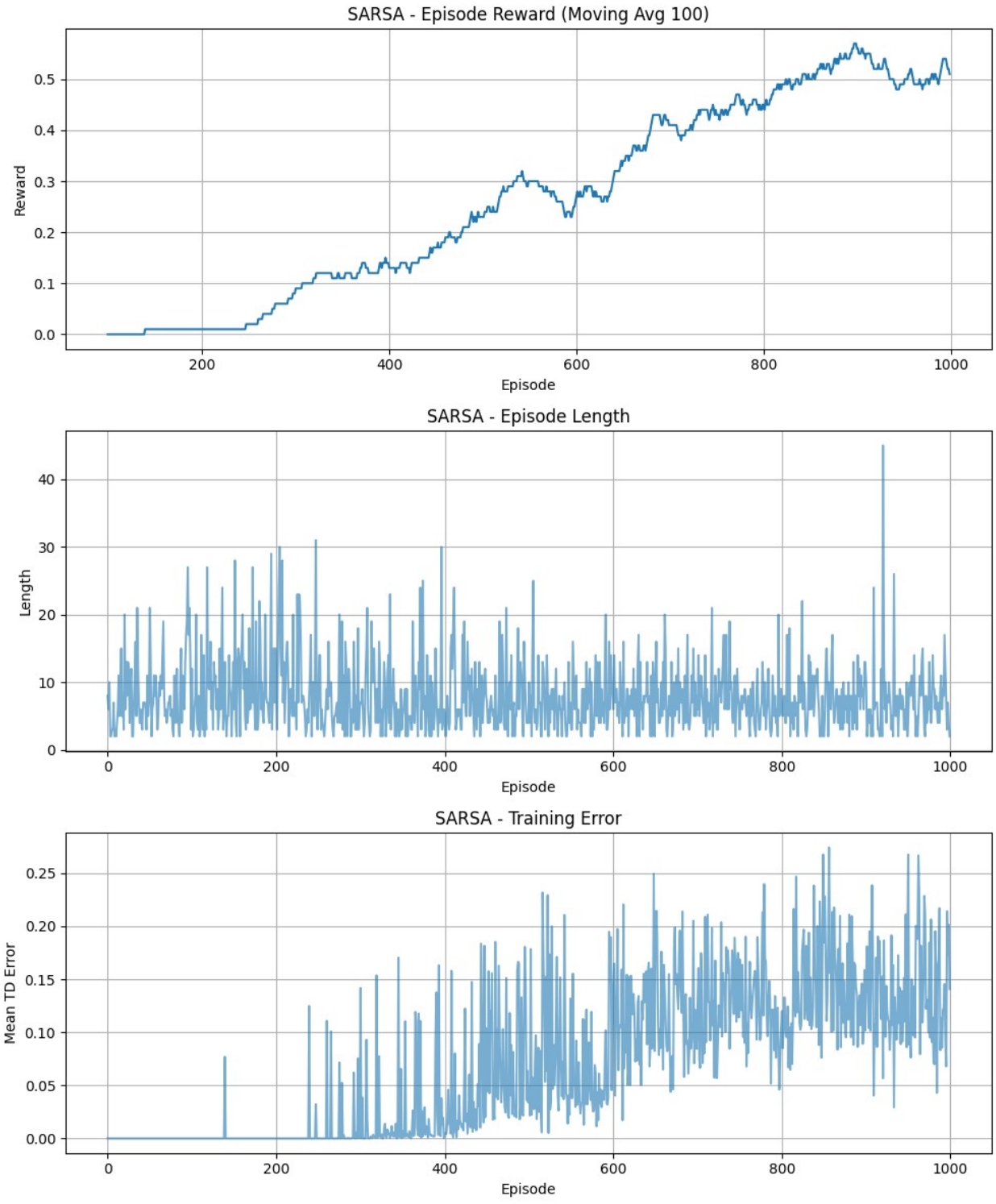

```
{"model_id":"4445a2e9eac74ccc80749a4681754449","version_major":2,"version_minor":0}
```

```
{"model_id":"6997227f57294db1945a111f719d7c39","version_major":2,"version_minor":0}
```


Q-learning - Episode Reward (Moving Avg 100)
Reward
Episode
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
200
400
600
800
1000


Q-learning - Episode Length
Length
Episode
5
10
15
20
25
30
35
0
200
400
600
800
1000


Q-learning - Training Error
Mean TD Error
Episode
0.00
0.05
0.10
0.15
0.20
0
200
400
600
800
1000



Q-learning learns faster: its average reward climbs by episode 200 and, despite a dip around 600, settles near 0.50 by episode 1000.

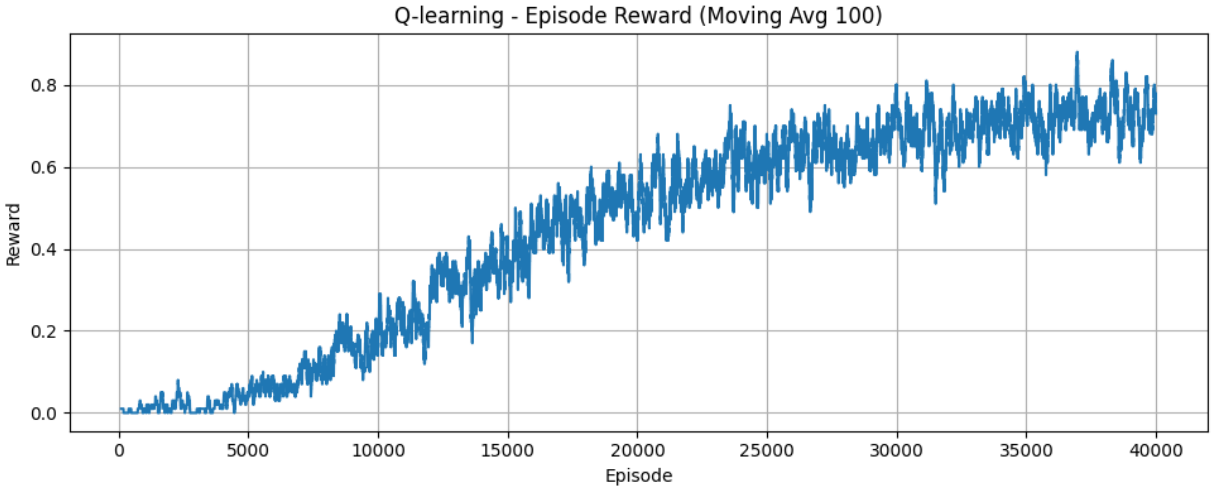
SARSA takes longer to improve but rises more smoothly, eventually peaking just above 0.55. Both methods cut their episode lengths from dozens of random steps down to about 4–10 moves, though Q-learning stops producing very long runs a bit sooner.

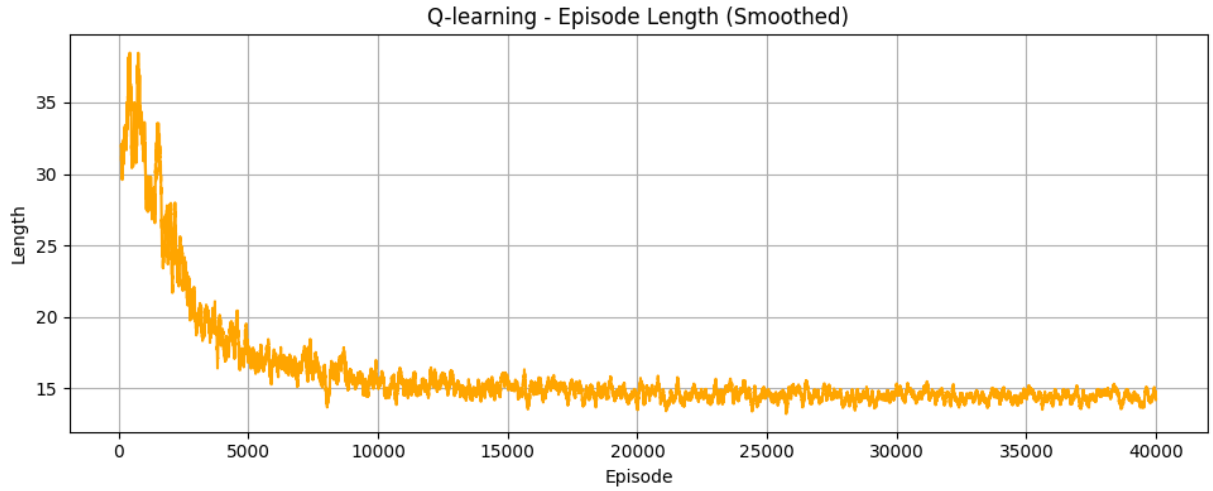
In terms of TD error, Q-learning's estimates converge to near zero around episode 700, while SARSA's error stays around 0.1–0.2 as it keeps updating on-policy. In short, Q-learning is quicker but a bit wobbly; SARSA is steadier and edges out slightly higher final rewards.

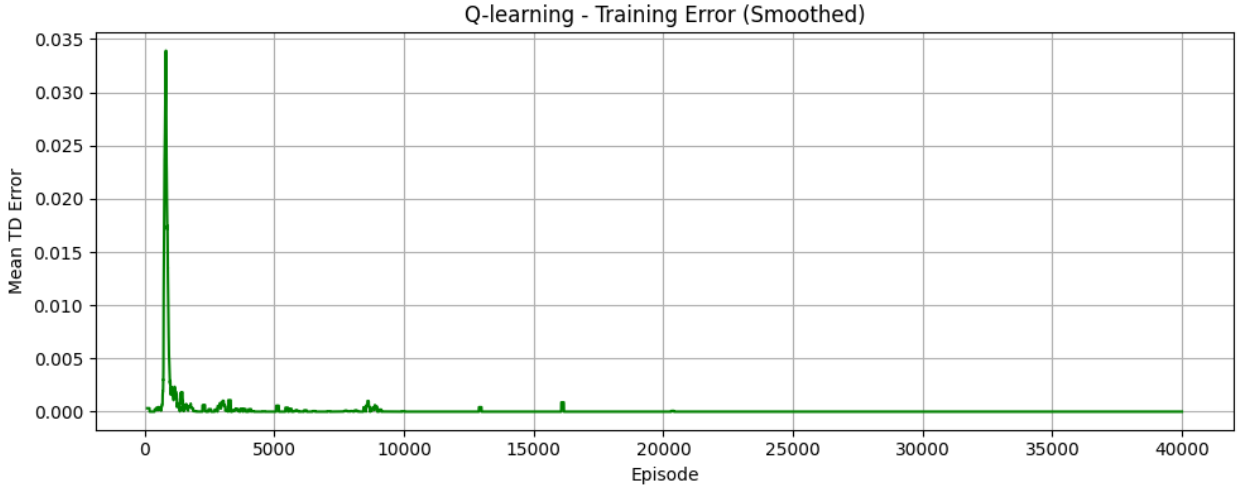
```
from ex5 import plot2

plot2()
```

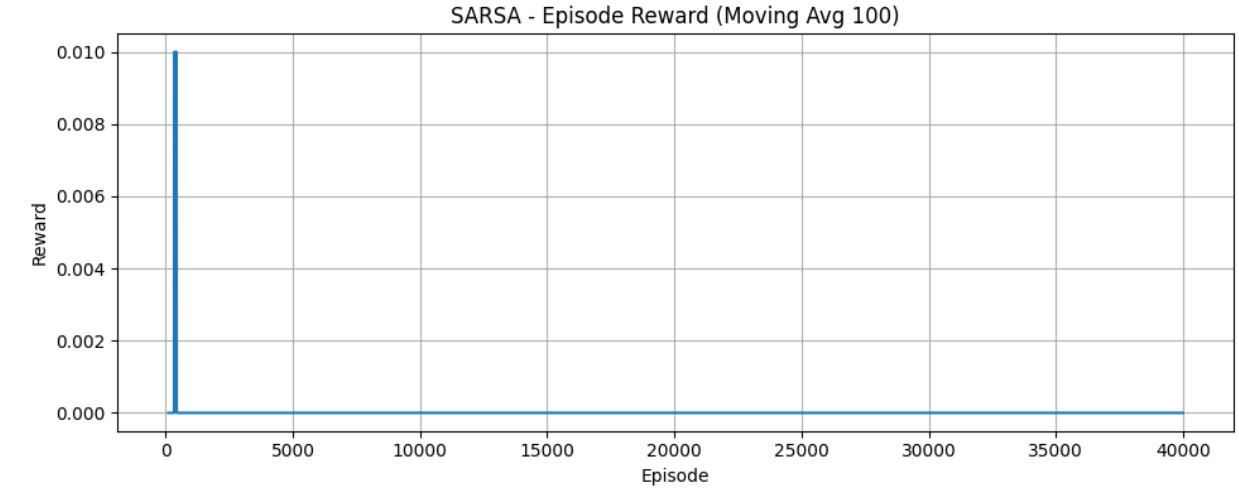

```
{"model_id":"ea4f45c07843495ab9fc1a04a8a90642","version_major":2,"version_minor":0}
```

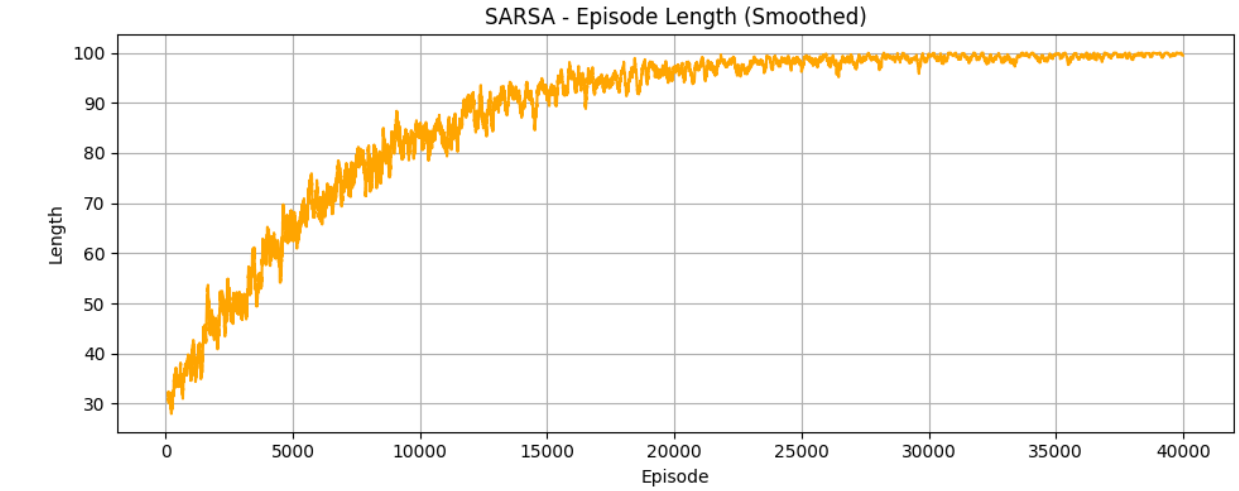
```
{"model_id":"88f728c773904940b7596ec48688d0da","version_major":2,"version_minor":0}
```

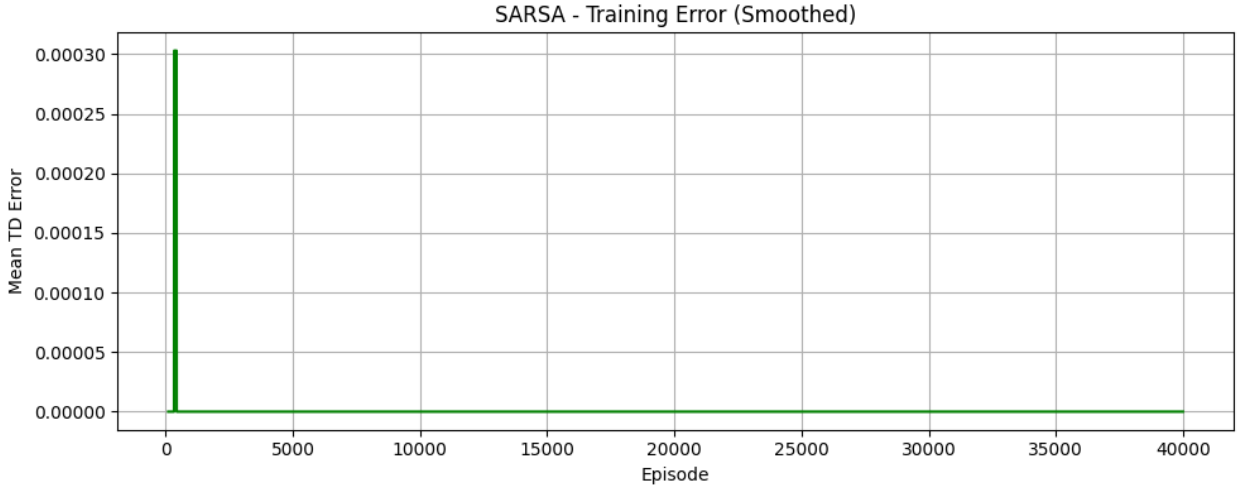

Q-learning - Episode Reward (Moving Avg 100)
Reward
0.0
0.2
0.4
0.6
0.8
0
5000
10000
15000
20000
25000
30000
35000
40000
Episode


Q-learning - Episode Length (Smoothed)
Length
15
20
25
30
35
0
5000
10000
15000
20000
25000
30000
35000
40000
Episode


Q-learning - Training Error (Smoothed)
Mean TD Error
0.000
0.005
0.010
0.015
0.020
0.025
0.030
0.035
0
5000
10000
15000
20000
25000
30000
35000
40000
Episode





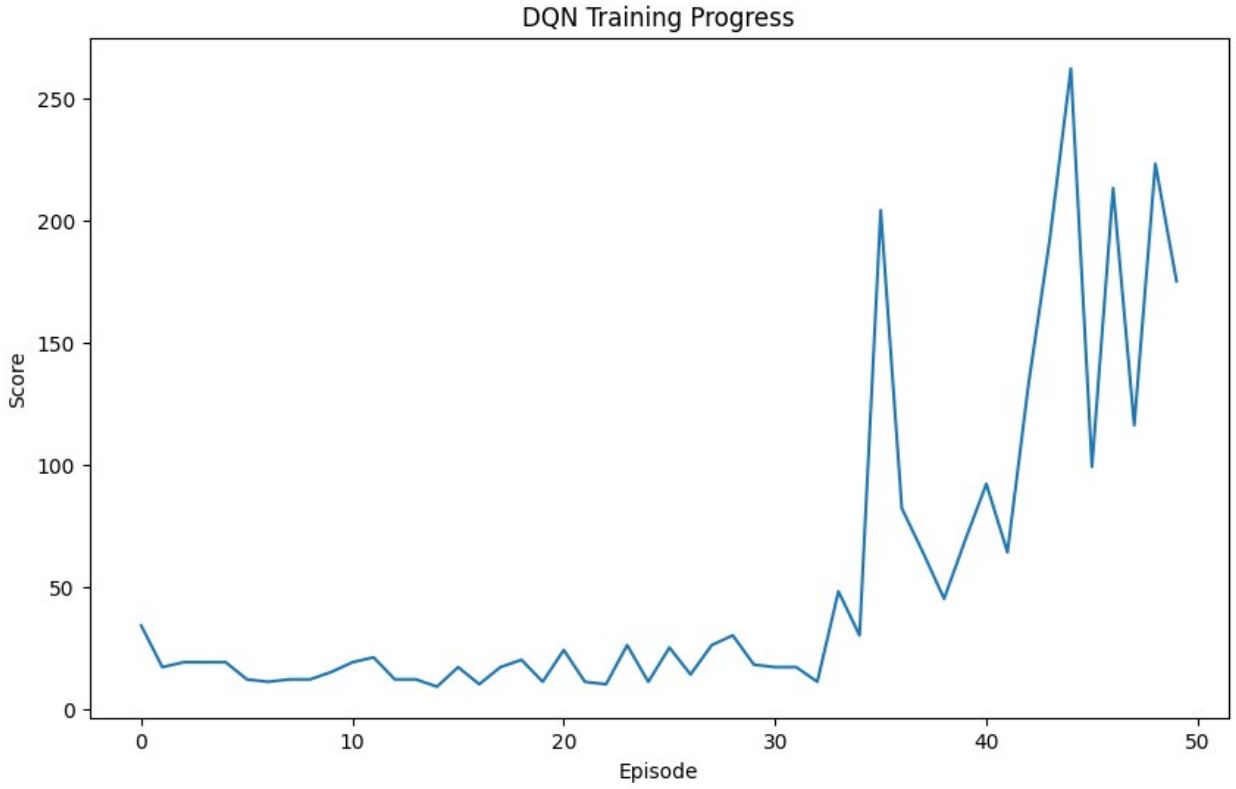


In the FrozenLake environment, Q-learning learned successfully: rewards increased, episode lengths dropped, and training error decreased over time. SARSA, however, failed to learn—the reward stayed at zero, episodes got longer, and the agent didn't improve. This likely happened because SARSA was too cautious and didn't explore enough.

Exercise 6: Train a deep Q-learning agent (optional: extra point)

```
from ex6 import *

agent = run_training()
```

```
Episode: 0/50, Score: 34.0, Epsilon: 1.00
Episode: 10/50, Score: 19.0, Epsilon: 0.53
Episode: 20/50, Score: 24.0, Epsilon: 0.25
Episode: 30/50, Score: 17.0, Epsilon: 0.10
Episode: 40/50, Score: 92.0, Epsilon: 0.01
```




```
Test score: 122.0
```

```
rendering(agent) # a pygame window will open
```

```
Render test score: 366.0
```

Deep Q-Learning Agent

Deep Q-Learning (DQN) combines Q-learning with deep neural networks to handle complex state spaces. Instead of using a Q-table, it uses a neural network to approximate Q-values, enabling it to work with continuous or high-dimensional state spaces. In this implementation, we're training an agent to balance a pole on a moving cart by learning optimal actions through trial and error, using experiences stored in a replay buffer to improve stability and convergence.

References

- Suárez, A. (s.f.). *Reinforcement Learning* [Slides]. Universidad Autónoma de Madrid.
- Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd ed.). MIT Press. http://incompleteideas.net/book/RLbook2020.pdf
- OpenAI. (2023). *ChatGPT* [Large language model]. https://chat.openai.com/ (Used to assist in summarizing and learning)
- DeepLizard. (2019, Jan 21). *Q-learning vs SARSA | Reinforcement Learning* [Video]. YouTube. https://www.youtube.com/watch?v=0g4j2k_Ggc4
- Hugging Face. (2022). *Reinforcement Learning with Q-learning*. https://huggingface.co/learn/deep-rl-course/unit1/introduction
- Towards Data Science. (2020). *Learning with SARSA, a good alternative to Q learning algorithm* https://towardsdatascience.com/reinforcement-learning-with-sarsa-a-good-alternative-to-q-learning-algorithm-bf35b209e1c/