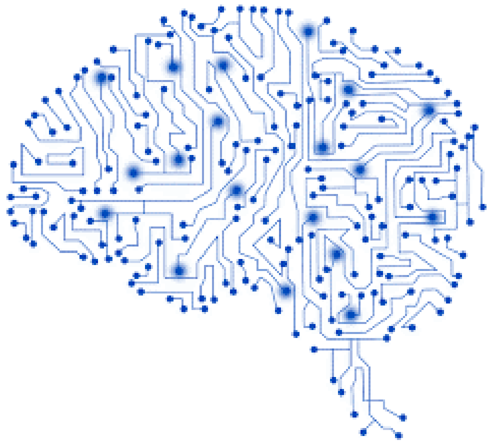
University of Minho
School of Engineering

Intelligent Systems Associate Laboratory



Dados e Aprendizagem Automática

Reinforcement Learning:

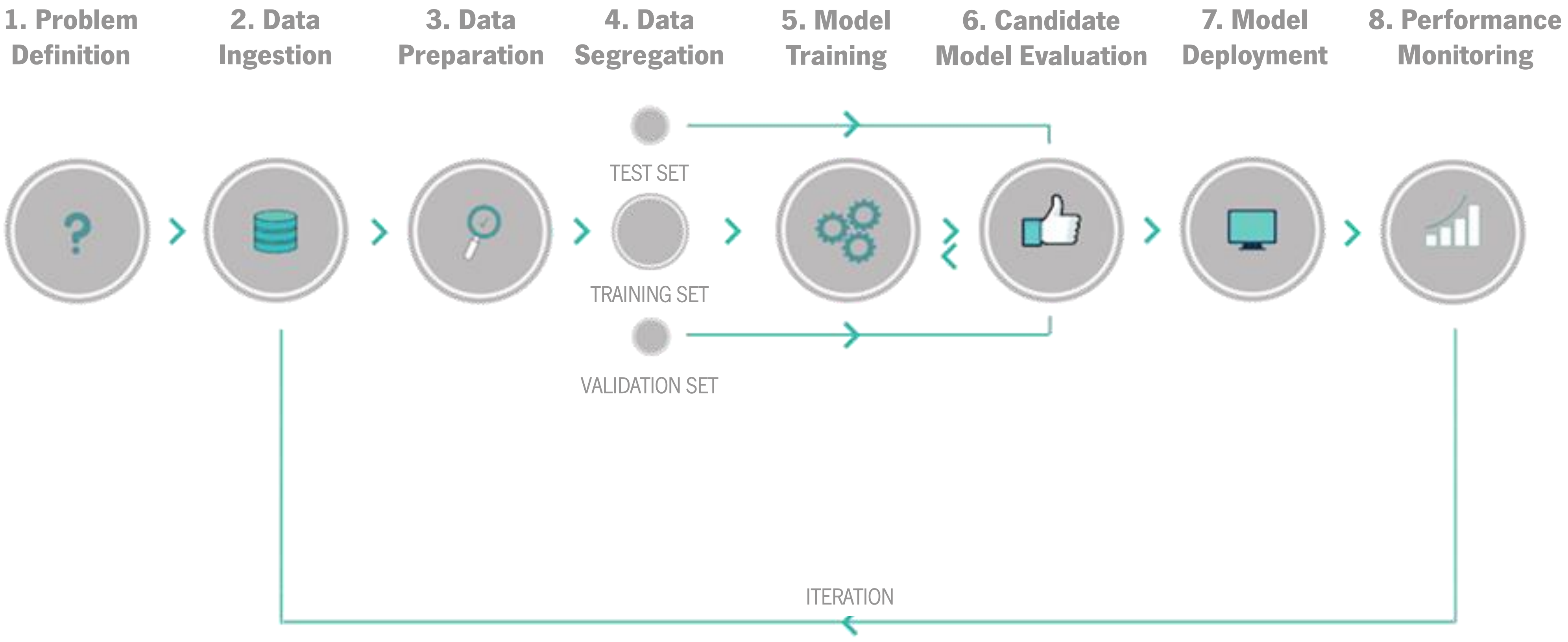
Q-Learning and SARSA

DAA @ MEI-1º/MiEI-4º – 1º Semestre

Bruno Fernandes, Dalila Alves, Filipa Ferraz, Victor Alves

Part XI

Contents


1. Problem Definition
2. Data Ingestion
3. Data Preparation
4. Data Segregation
5. Model Training
6. Candidate Model Evaluation
7. Model Deployment
8. Performance Monitoring
TEST SET
TRAINING SET
VALIDATION SET
ITERATION

Reinforcement Learning

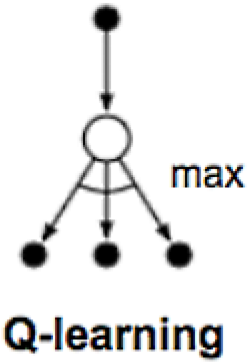
Reinforcement Learning

Let's suppose that there is the need to develop an intelligent bot to make decisions in order to solve a specific problem. One of the possibilities would be to train a **Reinforcement Learning (RL)** algorithm.

We will approach two RL methods:

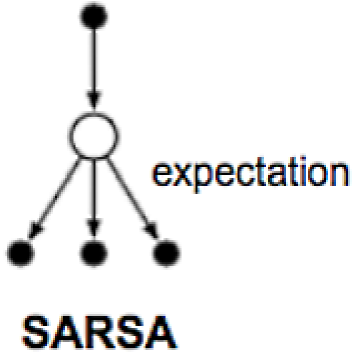
- **Q-Learning**, an off-policy and greed learner

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



- **SARSA**, a on-policy learner

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



Reinforcement Learning

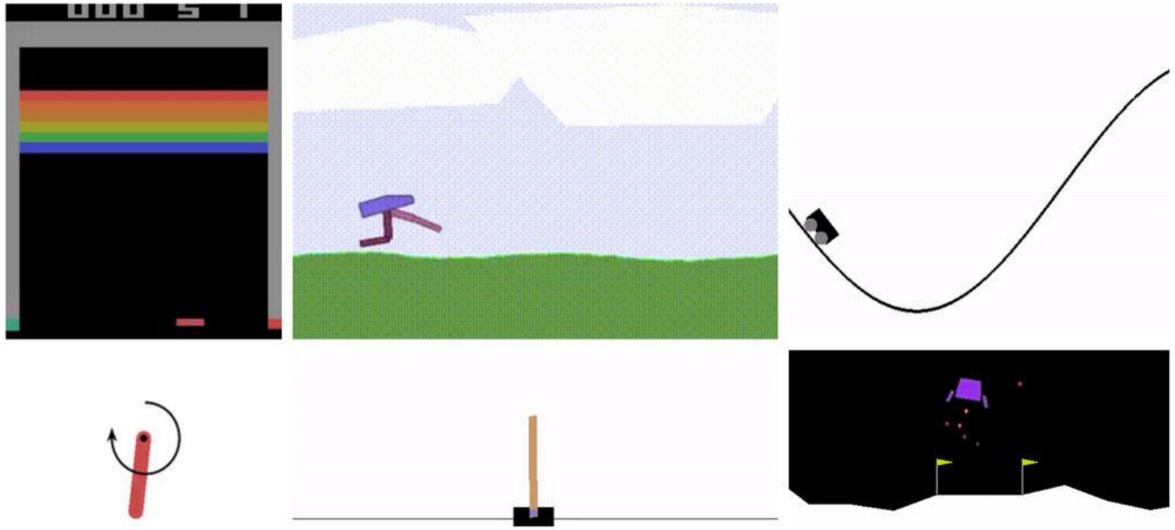
To implement our models, it is need to install some packages:

- Gymnasium - an open-source Python library for developing and comparing RL algorithms by providing a standard API to communicate between learning algorithms and environments, as well as a standard set of environments compliant with that API
- Pyglet - Python library for developing games and other visually rich applications
- Pygame - set of Python modules designed for writing games.

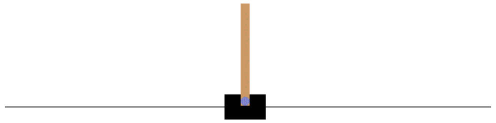
```
pip install gymnasium
pip install pyglet
pip install pygame
```

Gymnasium for Reinforcement Learning

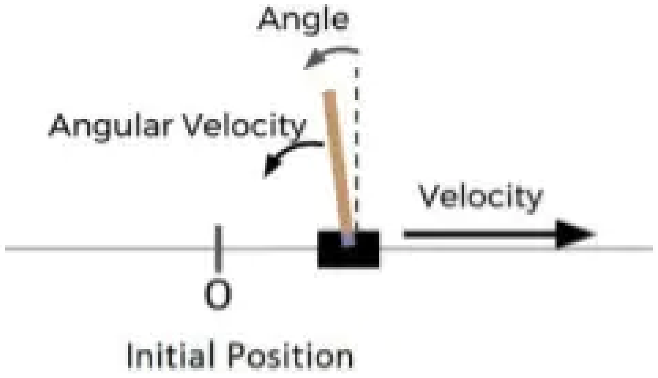
- OpenAI's Gymnasium is An API standard for reinforcement learning with a diverse collection of reference environments
- The Gymnasium interface is simple, pythonic, and capable of representing general RL problems
- It has seen tremendous growth and popularity in the RL community



Gymnasium Example: Cart Pole



- We will use the CartPole-v1 example to create instances and environments.
- A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum is placed upright on the cart and the goal is to balance the pole by applying forces in the left and right direction on the cart.
- A **reward** of +1 is provided for every time step that the pole remains upright, with a default reward threshold of 500.
- The **episode ends** when the pole angle is more than ±12° from vertical, the cart moves more than 2.4 units from the center, or the episode length is greater than 500.
- The **action space** is a `numpy` array which values indicate the direction of the fixed force the cart is pushed with:
  - `0`: push the cart to the left
  - `1`: push the cart to the right
- The **observation space** is a `numpy` array with 4 floating point values:
  - Cart Position
  - Cart Velocity
  - Pole Angle
  - Pole Angular Velocity



Gymnasium's Functions

- `make()`: creates the environment
- `reset()`: sets the environment to the default starting state
- `render()`: creates a popup window to display simulation of bot interacting with the environment
- `step()`: action taken by the bot. It returns an observation in the `numpy` array format `<observations, reward, done, info>`
- `sample()`: random input samples for the bot
- `close()`: closes the environment after action performed

environment initialization

```
env = gym.make("CartPole-v1", render_mode = "human")
env.reset()
for _ in range(200):
    env.render()
    env.step(env.action_space.sample())
env.close()
```

a random action is executed by the bot, based on the possible actions in the environment

200 time-steps are executed and rendered

Gymnasium's Observations

Observations are environment specific information variables:

- `observation(object)`: an environment-specific object representing the observation of the environment, e.g., joint angles and joint velocities of a robot, or the board state in a board game
- `reward(float)`: amount of reward achieved by the previous action. The scale varies between environments, but the goal is always to increase your total reward
- `terminated(boolean)`: whether a terminal state is reached. Most tasks are divided into well-defined episodes and terminated being True indicates the episodes has terminated. For example, the pole tipped too far, or the bot lost its last life
- `truncated(bool)`: whether a truncation condition is satisfied. In this case, when the episode length is greater than 500. Can be used to end the episode prematurely before a terminal state is reached
- `info(dict)`: diagnostic information useful for debugging, e.g., by containing the raw probabilities behind the environment's last state change

Gymnasium's Observations

The process gets started by calling `reset()`, which returns an initial observation.

A more proper way of writing the previous code with respect to the episodes and `done` flag:

```
env = gym.make("CartPole-v1", render_mode = "human")
env.reset()
for _ in range(200):
    env.render()
    env.step(env.action_space.sample())
env.close()
```

```
env = gym.make("CartPole-v1", render_mode = "human")
env.reset()

for i_episode in range(20):
    observation = env.reset()
    for t in range(30):
        env.render()
        print(observation)
        action = env.action_space.sample()
        observation, reward, terminated, truncated, info = env.step(action)
        if terminated:
            print("Episode finished after {} time steps".format(t+1))
            break

env.close()
```

definition of number of episodes

definition of number of time steps per episode

bot perception for each step, based on action taken

verify if episode is over

Gymnasium's Observations

The process gets started by calling `reset()`, which returns an initial observation.

A more proper way of writing the previous code with respect to the episodes and `done` flag:

```
env = gym.make("CartPole-v1", render_mode = "human")
env.reset()
for _ in range(200):
    env.render()
    env.step(env.action_space.sample())
env.close()
```

```
env = gym.make("CartPole-v1", render_mode = "human")
env.reset()
```

```
for i_episode in range(20):
    observation = env.reset()
    for t in range(30):
        env.render()
        print(observation)
        action = env.action_space.sample()
        observation, reward, terminated, truncated, info = env.step(action)
        if terminated:
            print("Episode finished after ", t+1, "time steps")
            break
```

```
env.close()
```

definition of number of episodes

definition of number of time steps per episode

bot perception for each step, based on action taken

verify if episode is over

Hard-Coded Policy

Now that we understand the basic aim to balance the pole upright, how could we do that?

Well, we need to come up with a **policy** (or strategy) the agent may follow to achieve the balance at each step. It can use all the past actions and observations to decide what to do.

As we observe the game, we may naively come to a thought that we need to move the cart to the right if the pole slants towards the right. As the pole tilts towards the left, we might want to push the cart to the left.

definition of hard-coded policies

move cart right if pole is falling to the right

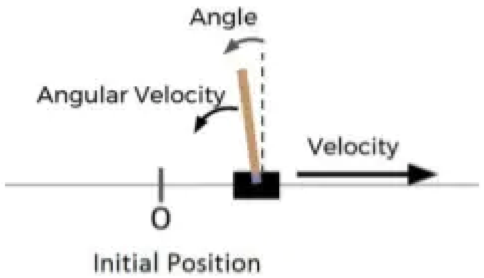
angle is measured off straight vertical line

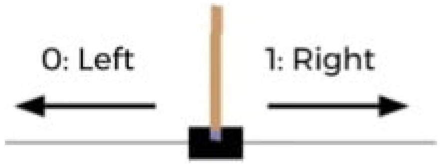
perform action

```
env = gym.make("CartPole-v1", render_mode = "human")
env.reset()

for t in range(20):
    env.render()
    print(observation)
    cart_pos, cart_vel, pole_ang, ang_vel = observation
    if pole_ang > 0:
        action = 1
    else:
        action = 0
    observation, reward, terminated, truncated, info=env.step(action)

env.close()
```

move right

move left





Reinforcement Learning - Environment

Global hyperparameters:

```
EPISODES = 5000
DISCOUNT = 0.95
EPISODE_DISPLAY = 500
LEARNING_RATE = 0.25
EPSILON = 0.2
```

Number of episodes: applied for training the RL model

Discount factor: used to measure how far ahead in time the algorithm must look, i.e., if `factor = 0` none of the future rewards are considered in Q-learning; if `factor = 1` future rewards are given a high weight

Episode display: defines the number of episodes necessary to run before rendering the episode, i.e., episodes 0, 500, 1000, 1500, .. are rendered. Positive to visually verify learning evolution of RL model

Learning rate: set between [0, 1], applied to facilitate the Q-value update at a desired rate, i.e., if `rate = 0` then Q-values are never updated, and nothing is learnt; if `rate = 1` then nothing is added to the current Q-value

Exploration constant: used to give the bot an element of exploration, i.e., if `epsilon = 0` then the algorithm only considers actions corresponding to the highest Q-value; if `epsilon = 1` then the algorithm only selects random action values

Reinforcement Learning - Environment

Let's prepare our environment and look at the observation and action spaces:

```
def prepare_env():
    env = gym.make("CartPole-v1")

    print('Env. Observation Space: ', env.observation_space)
    print('Env. Observation Space - High:', env.observation_space.high)
    print('Env. Observation Space - Low:', env.observation_space.low)

    print('Env. Action Space:', env.action_space)
    print('Env. Actions Space:', env.action_space.n)
    return env
```

Environment values

Observation Space:

`[0]` cart position along x-axis

`[1]` cart velocity

`[2]` pole angle (rad)

`[3]` pole angular velocity

continuous **min** and **max** values for each observation variable, i.e.,

`[position of cart, velocity of cart, angle of pole, rotation rate of pole]`

Action Space:

`[0]` push cart to the left

`[1]` push cart to the right

Reinforcement Learning – Discretize State's Results

When we execute `step()` it returns a continuous state. `discretised_state(state)` function converts these continuous states into discrete states. The `Pole Angle` and `Pole Angular Velocity` features will be used to train the RL model. In this case we will split the number of possibilities into 50 bins.

`state[2]→theta`

`state[3]→theta_dot`

```python
def discretised_state(state, theta_minmax, theta_dot_minmax, theta_state_size, theta_dot_state_size):
    discrete_state = np.array([0, 0])

    theta_window =  (theta_minmax - (-theta_minmax)) / theta_state_size
    discrete_state[0] = (state[2] - (-theta_minmax)) // theta_window
    discrete_state[0] = min(theta_state_size - 1, max(0, discrete_state[0]))

    theta_dot_window =  (theta_dot_minmax - (-theta_dot_minmax)) / theta_dot_state_size
    discrete_state[1] = (state[3] - (-theta_dot_minmax)) // theta_dot_window
    discrete_state[1] = min(theta_dot_state_size - 1, max(0, discrete_state[1]))

    return tuple(discrete_state.astype(np.int32))
```

discrete array initialized `[Pole Angle, Pole Angular Velocity]`

continuous state of `Pole Angle`

continuous state of `Pole Angular Velocity`

discrete state of `Pole Angle` and `Pole Angular Velocity`

Q-Learning

Q-table of size: `theta_state_size` x `theta_sot_state_size` x `env.action_space.n`

```
def train_cart_pole_qlearning(EPISODES, DISCOUNT, EPISODE_DISPLAY, LEARNING_RATE, EPSILON):
    env = prepare_env()

    theta_minmax = env.observation_space.high[2]
    theta_dot_minmax = math.radians(50)
    theta_state_size = 50
    theta_dot_state_size = 50

    Q_TABLE = np.random.randn(theta_state_size, theta_dot_state_size, env.action_space.n)

    ep_rewards = []
    ep_rewards_table = {'ep': [], 'avg': [], 'min': [], 'max': []}
```

Use **min** and **max** observation to convert continuous states into discrete states for features `Pole Angle` and `Pole Angular Velocity`

50 `Pole Angle` states
50 `Pole Angular Velocity` states

Q-table initiated with random values - used to calculate the maximum expected future rewards for action at each state. Q-table dimension varies depending on:

- **Environment possible actions** (2) - left & right
- **Environment number of states** (50 `pole angle` states, 50 `pole angular velocity` states) – *increased number of states provides a higher resolution of the state space*

`dict` model stats to verify model learning progression

Q-Learning


```
for episode in range(EPISODES):
    episode_reward = 0
    terminated = False
    i = 0
    if episode % EPISODE_DISPLAY == 0:
        render_state = True
    else:
        render_state = False
    curr_discrete_state = discretised_state(env.reset()[0],
                                            theta_minmax, theta_dot_minmax,
                                            theta_state_size, theta_dot_state_size)
```

Initialize variables at start of an episode

Q-Learning


```
while not terminated:
    if np.random.random() > EPSILON:
        action = np.argmax(Q_TABLE[curr_discrete_state])
    else:
        action = np.random.randint(0, env.action_space.n)

    new_state, reward, terminated, _, _ = env.step(action)
    new_discrete_state = discretised_state(new_state,
                                           theta_minmax, theta_dot_minmax,
                                           theta_state_size, theta_dot_state_size)
    if render_state:
        env.render()

    if not terminated:
        max_future_q = np.max(Q_TABLE[new_discrete_state[0], new_discrete_state[1]])
        current_q = Q_TABLE[curr_discrete_state[0], curr_discrete_state[1], action]
        new_q = current_q + LEARNING_RATE * (reward + DISCOUNT * max_future_q - current_q)
        Q_TABLE[curr_discrete_state[0], curr_discrete_state[1], action] = new_q
    i += 1
    curr_discrete_state = new_discrete_state
    episode_reward += reward
```

Based on **exploration constant**, select random action or action with highest Q-value

Bot executes selected action and acquires observation from new state

If episode not completed, update Q-table using Q-learning formula:

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

Update `current_state` & `episode_reward` until end of episode

Q-Learning


```python
ep_rewards.append(episode_reward)

if not episode % EPISODE_DISPLAY:
    avg_reward = sum(ep_rewards[-EPISODE_DISPLAY:]) / len(ep_rewards[-EPISODE_DISPLAY:])
    ep_rewards_table['ep'].append(episode)
    ep_rewards_table['avg'].append(avg_reward)
    ep_rewards_table['min'].append(min(ep_rewards[-EPISODE_DISPLAY:]))
    ep_rewards_table['max'].append(max(ep_rewards[-EPISODE_DISPLAY:]))
    print(f"Episode:{episode} avg:{avg_reward} min:{min(ep_rewards[-EPISODE_DISPLAY:])}
        max:{max(ep_rewards[-EPISODE_DISPLAY:])}")
```

Save `episode_reward` for model learning analysis

Append episode's information on episode rewards table dict

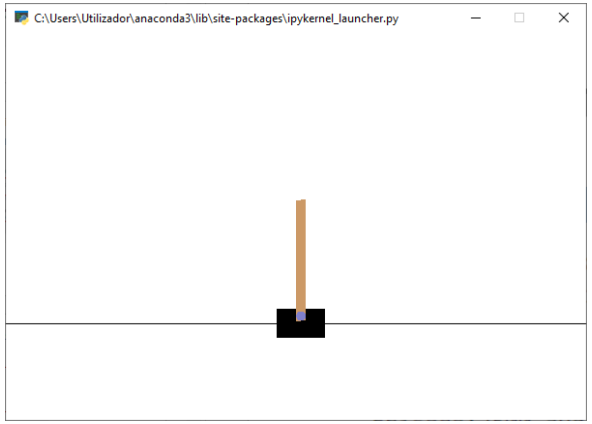
Q-Learning

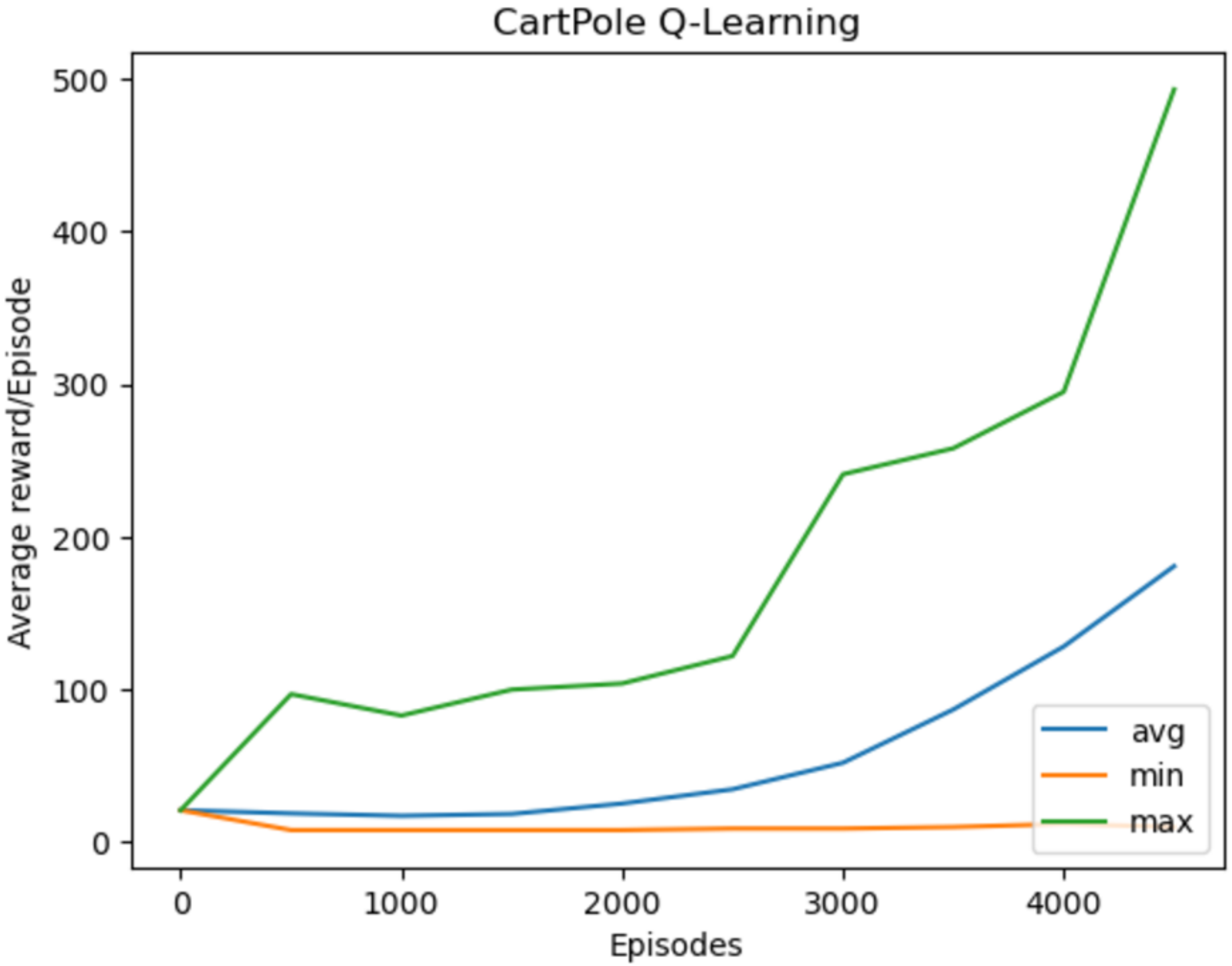

```
env.close()

plt.plot(ep_rewards_table['ep'], ep_rewards_table['avg'], label = "avg")
plt.plot(ep_rewards_table['ep'], ep_rewards_table['min'], label = "min")
plt.plot(ep_rewards_table['ep'], ep_rewards_table['max'], label = "max")
plt.legend(loc = 4)
plt.title('CartPole Q-Learning')
plt.ylabel('Average reward/Episode')
plt.xlabel('Episodes')
plt.show()

return ep_rewards_table
```

Plot model evolution performance: based on episode rewards table, generate a plot to verify episode rewards evolution for each episode

Q-Learning - Results


```
ep_rewards_table_qlearning = train_cart_pole_qlearning(EPISODES, DISCOUNT, EPISODE_DISPLAY, LEARNING_RATE, EPSILON)
Env. Observation Space:  Box([-4.8               -inf -0.41887903        -inf],
[4.8               inf 0.41887903        inf], (4,), float32)
Env. Observation Space - High: [4.8               inf 0.41887903        inf]
Env. Observation Space - Low: [-4.8               -inf -0.41887903        -inf]
Env. Action Space: Discrete(2)
Env. Actions Space: 2
Episode:0 avg:21.0 min:21.0 max:21.0
Episode:500 avg:18.924 min:8.0 max:97.0
Episode:1000 avg:17.36 min:8.0 max:83.0
Episode:1500 avg:18.566 min:8.0 max:100.0
Episode:2000 avg:25.388 min:8.0 max:104.0
Episode:2500 avg:34.716 min:9.0 max:122.0
Episode:3000 avg:52.078 min:9.0 max:241.0
Episode:3500 avg:86.98 min:10.0 max:258.0
Episode:4000 avg:128.21 min:12.0 max:295.0
Episode:4500 avg:180.806 min:10.0 max:493.0
```





SARSA


```python
def train_cart_pole_sarsa(EPISODES, DISCOUNT, EPISODE_DISPLAY, LEARNING_RATE, EPSILON):
    env = prepare_env()

    theta_minmax = env.observation_space.high[2]
    theta_dot_minmax = math.radians(50)
    theta_state_size = 50
    theta_dot_state_size = 50

    Q_TABLE = np.random.randn(theta_state_size, theta_dot_state_size, env.action_space.n)

    ep_rewards = []
    ep_rewards_table = {'ep': [], 'avg': [], 'min': [], 'max': []}
for episode in range(EPISODES):
    episode_reward = 0
    terminated = False
    if episode % EPISODE_DISPLAY == 0:
        render_state = True
    else:
        render_state = False
    curr_discrete_state = discretised_state(env.reset()[0],
                                            theta_minmax, theta_dot_minmax,
                                            theta_state_size, theta_dot_state_size)
    if np.random.random() > EPSILON:
        action = np.argmax(Q_TABLE[curr_discrete_state])
    else:
        action = np.random.randint(0, env.action_space.n)
```

The preparation of environment to apply SARSA is the same

SARSA


```
while not terminated:
    new_state, reward, terminated, _, _ = env.step(action)
    new_discrete_state = discretised_state(new_state,
                                           theta_minmax, theta_dot_minmax,
                                           theta_state_size, theta_dot_state_size)
    if np.random.random() > EPSILON:
        new_action = np.argmax(Q_TABLE[new_discrete_state])
    else:
        new_action = np.random.randint(0, env.action_space.n)

    if render_state:
        env.render()

    if not terminated:
        current_q = Q_TABLE[curr_discrete_state + (action,)]
        max_future_q = Q_TABLE[new_discrete_state + (new_action,)]
        new_q = current_q + LEARNING_RATE * (reward + DISCOUNT * max_future_q-current_q)
        Q_TABLE[curr_discrete_state + (action,)] = new_q
    curr_discrete_state = new_discrete_state
    action = new_action
    episode_reward += reward
```

Based on **exploration constant**, select random action or action with highest Q-value **for next state**

If episode not completed, update Q-table using SARSA formula:

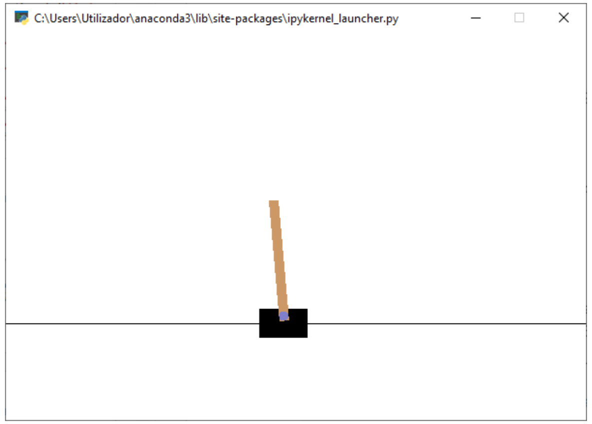
$$\underset{a_{t+1}}{Q(s_t, a_t)} \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \underset{s_{t+1}}{Q(s_{t+1}, a_{t+1})} - Q(s_t, a_t)]$$

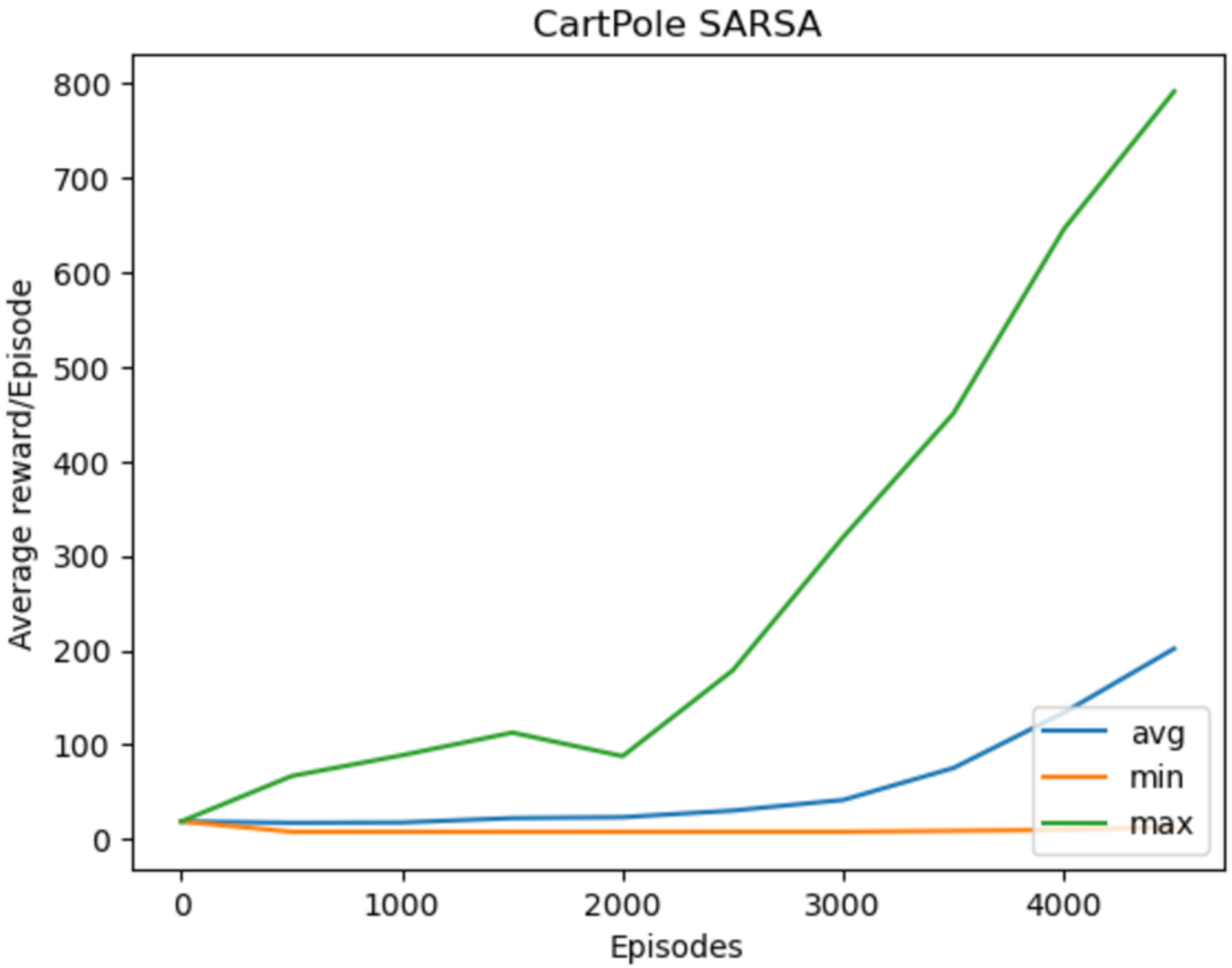
SARSA


```
    ep_rewards.append(episode_reward)

    if not episode % EPISODE_DISPLAY:
        avg_reward = sum(ep_rewards[-EPISODE_DISPLAY:]) / len(ep_rewards[-EPISODE_DISPLAY:])
        ep_rewards_table['ep'].append(episode)
        ep_rewards_table['avg'].append(avg_reward)
        ep_rewards_table['min'].append(min(ep_rewards[-EPISODE_DISPLAY:]))
        ep_rewards_table['max'].append(max(ep_rewards[-EPISODE_DISPLAY:]))
        print(f"Episode:{episode} avg:{avg_reward} min:{min(ep_rewards[-EPISODE_DISPLAY:])} max:{max(ep_rewards[-EPISODE_DISPLAY:])}")

env.close()

plt.plot(ep_rewards_table['ep'], ep_rewards_table['avg'], label = "avg")
plt.plot(ep_rewards_table['ep'], ep_rewards_table['min'], label = "min")
plt.plot(ep_rewards_table['ep'], ep_rewards_table['max'], label = "max")
plt.legend(loc = 4)
plt.title('CartPole SARSA')
plt.ylabel('Average reward/Episode')
plt.xlabel('Episodes')
plt.show()

return ep_rewards_table
```

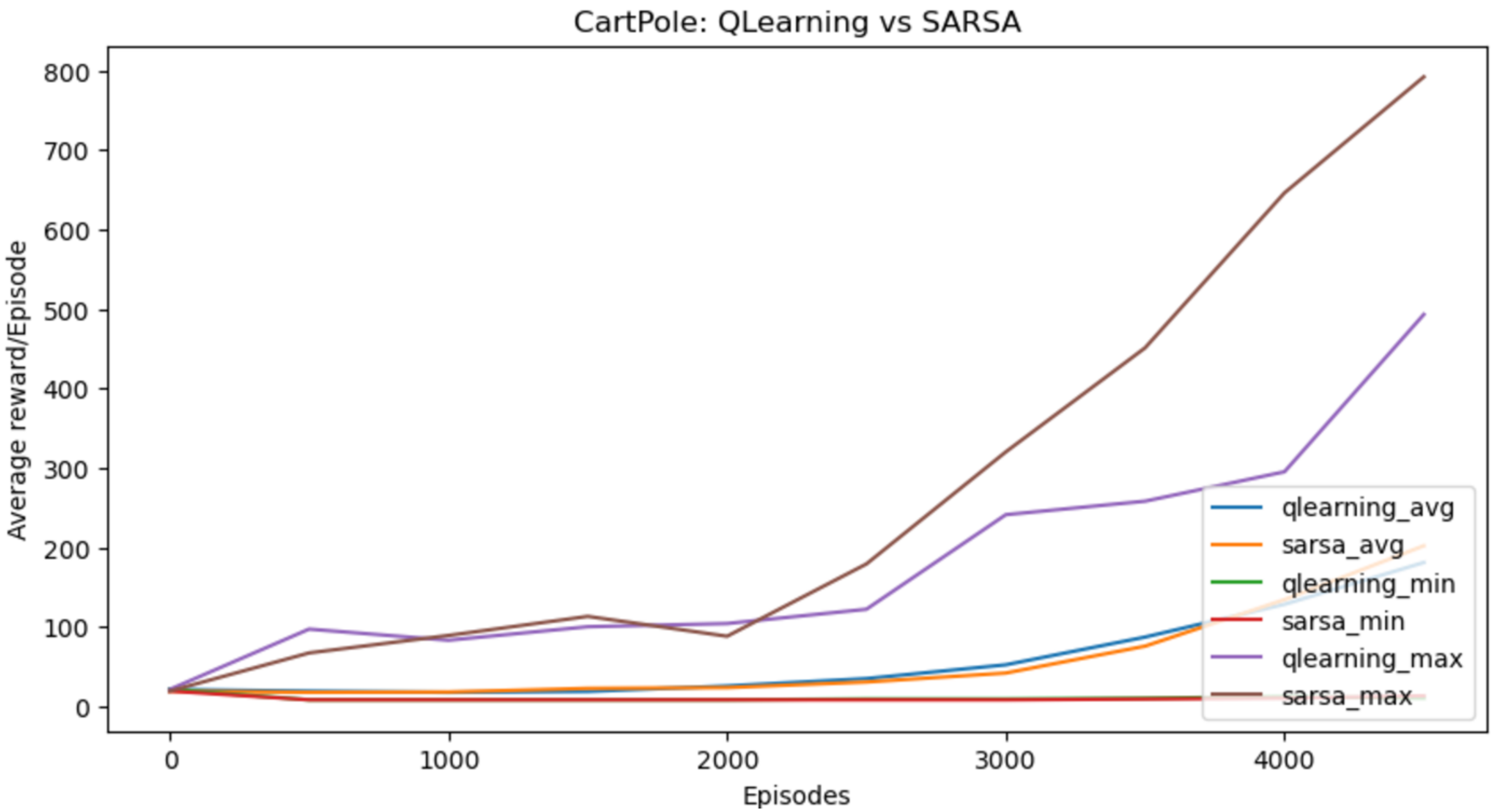
SARSA - Results


```
ep_rewards_table_sarsa = train_cart_pole_sarsa(EPISODES, DISCOUNT, EPISODE_DISPLAY, LEARNING_RATE, EPSILON)
Env. Observation Space:  Box([-4.8               -inf -0.41887903        -inf],
[4.8               inf 0.41887903        inf], (4,), float32)
Env. Observation Space - High: [4.8               inf 0.41887903        inf]
Env. Observation Space - Low: [-4.8               -inf -0.41887903        -inf]
Env. Action Space: Discrete(2)
Env. Actions Space: 2
Episode:0 avg:19.0 min:19.0 max:19.0
Episode:500 avg:17.448 min:8.0 max:67.0
Episode:1000 avg:17.814 min:8.0 max:89.0
Episode:1500 avg:22.402 min:8.0 max:113.0
Episode:2000 avg:23.406 min:8.0 max:88.0
Episode:2500 avg:30.432 min:8.0 max:179.0
Episode:3000 avg:41.69 min:8.0 max:320.0
Episode:3500 avg:75.542 min:9.0 max:451.0
Episode:4000 avg:134.048 min:10.0 max:646.0
Episode:4500 avg:201.876 min:13.0 max:792.0
```





Comparing Q-Learning vs. SARSA

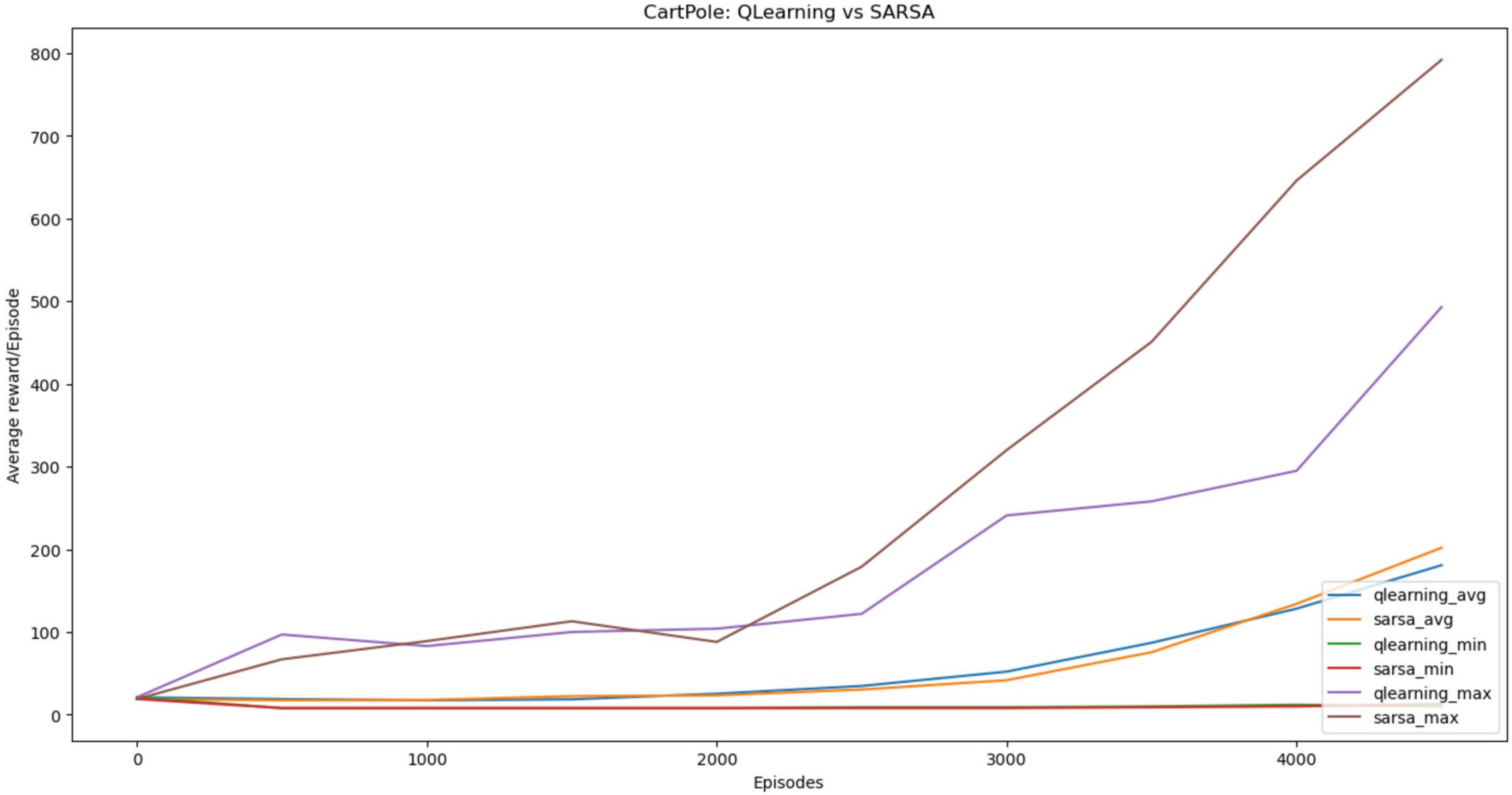


Comparing Q-Learning vs. SARSA

On comparing the graphs of **SARSA** and **Q-Learning** we observe:

- For this run, the **reward converges to a larger value** in the case of **SARSA** than in the case of Q-Learning. This is possibly due to the action selection step where an epsilon-greedy policy is applied. In **Q-Learning**, the **action corresponding to the largest Q-value is selected**.
- The **maximum reward** is obtained by the agent in **4500 episodes for Q-Learning** and **for SARSA** in the case of cart pole.
- Training both models with **more episodes and optimizing its hyperparameters** could provide further increases on the decision-making performance. More experiments could be tested by **adapting the input features** and **changing the number of states per feature**.

Comparing Q-Learning vs. SARSA



Hands On