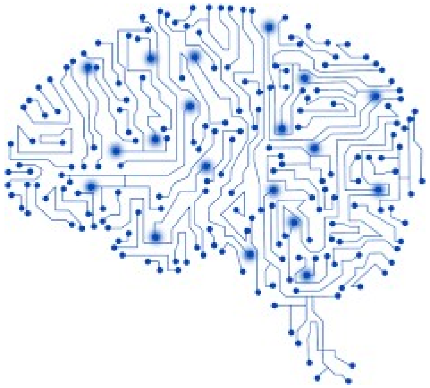
University of Minho
School of Engineering

ISLab



Dados e Aprendizagem Automática

Reinforcement Learning

Q-Learning vs SARSA

DAA @ MEI-1º/MiEI-4º – 1º Semestre

Cesar Analide, Bruno Fernandes, Filipa Ferraz, Filipe Gonçalves, Victor Alves

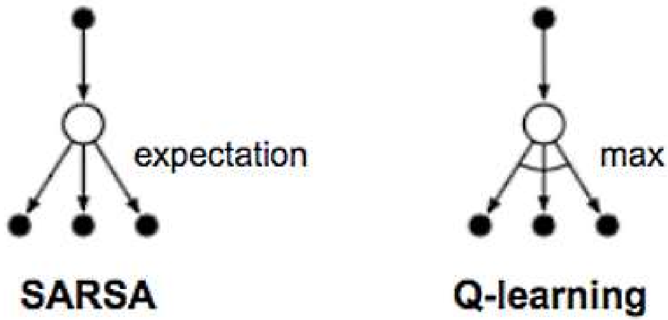
Part VIII

What about 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.

Based on the RL algorithms learned in this course, two methods come to mind:

- Q-learning $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$
- State-Action-Reward-State-Action (SARSA) $Q(s_t, a_t) \underset{a_{t+1}}{\leftarrow} Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) \underset{s_{t+1}}{-} Q(s_t, a_t)]$



What about Reinforcement Learning?

To implement our first RL algorithm we will require:

- To install OpenAI Gym library - use the Navigator or the Prompt:

  (Anaconda) conda install -c conda-forge gym

  (Pip) pip install gym

- You may also need Pyglet

  (Anaconda) conda install -c conda-forge pyglet

  (Pip) pip install gym

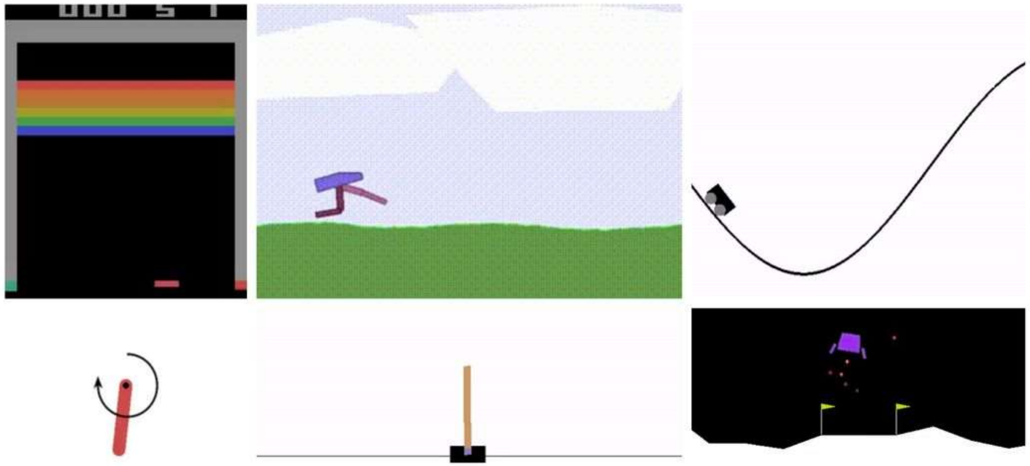
- And Pygame

  (Pip) pip install pygame

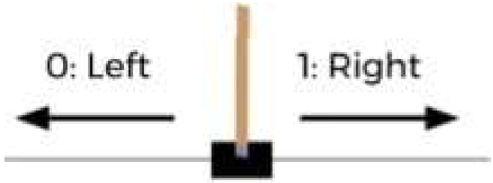
OpenAI Gym for Reinforcement Learning

Why?

- OpenAI Gym is an open-source toolkit for developing and comparing reinforcement learning algorithms
- OpenAI Gym library is a python library with a collection of environment that can be used with the reinforcement learning algorithms
- It has seen tremendous growth and popularity in the reinforcement learning community
- More information available here



OpenAI Gym's Environment

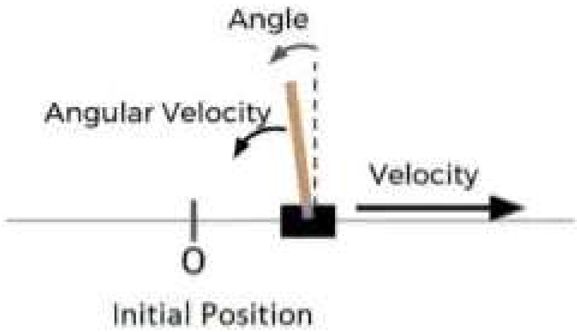


- An example of running an instance of the "CartPole-v1" (more info here) environment for 1000 time-step, rendering the environment at each step.

- A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum starts upright, and the goal is to prevent it from falling over.
    - A reward of +1 is provided for every time step that the pole remains upright.
    - The episode ends when the pole angle is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.

OpenAI Gym "CartPole-v1" environment is a numpy array with 4 floating point values:

1. Horizontal Position
2. Horizontal Velocity
3. Angle of Pole
4. Angular Velocity



OpenAI Gym's Functions

OpenAI Gym – functions:

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

```
import gym
env=gym.make("CartPole-v1")
env.reset()
for _ in range(1000):
    env.render()
    env.step(env.action_space.sample()) #take a random action
env.close()
```

Environment initialization

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

1000 time-steps are executed and rendered

OpenAI Gym 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
- **done (boolean):** Whether it's time to reset the environment again. Most tasks are divided into well-defined episodes and done being *True* indicates the episodes has terminated. For example, the pole tipped too far or the bot lost its last life
- **info (dict):** Diagnostic information useful for debugging, e.g., by containing the raw probabilities behind the environment's last state change

OpenAI Gym 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:

Code v1

```
import gym
env=gym.make("CartPole-v1")
env.reset()
for _ in range(1000):
    env.render()
    env.step(env.action_space.sample()) #take a random action
env.close()
```

Code v2 (still simple, yet more "complete"

```
import gym
env=gym.make("CartPole-v1")
env.reset()
for i_episode in range(20):
    observation=env.reset()
    for t in range(100):
        env.render()
        print(observation)
        action=env.action_space.sample()
        observation, reward, done, info=env.step(action)
        if done:
            print("Episode finished after {} time steps".format(t+1))
            break
env.close()
```

definition of number of episodes

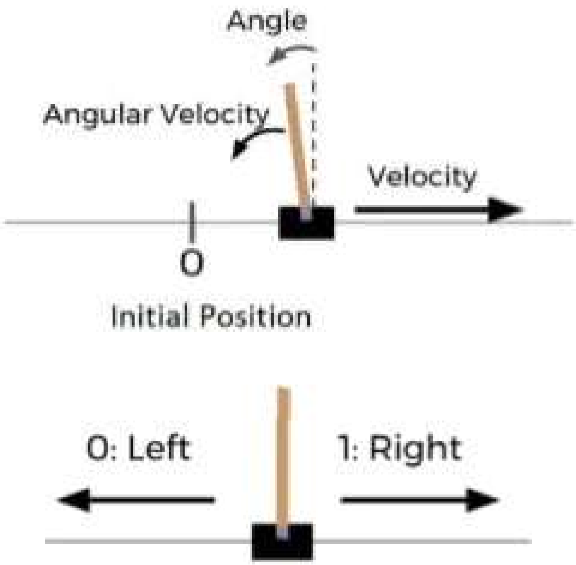
definition of number of time steps per episode

Bot perception for each step, based on action taken

verify if episode is over

OpenAI Gym Observations

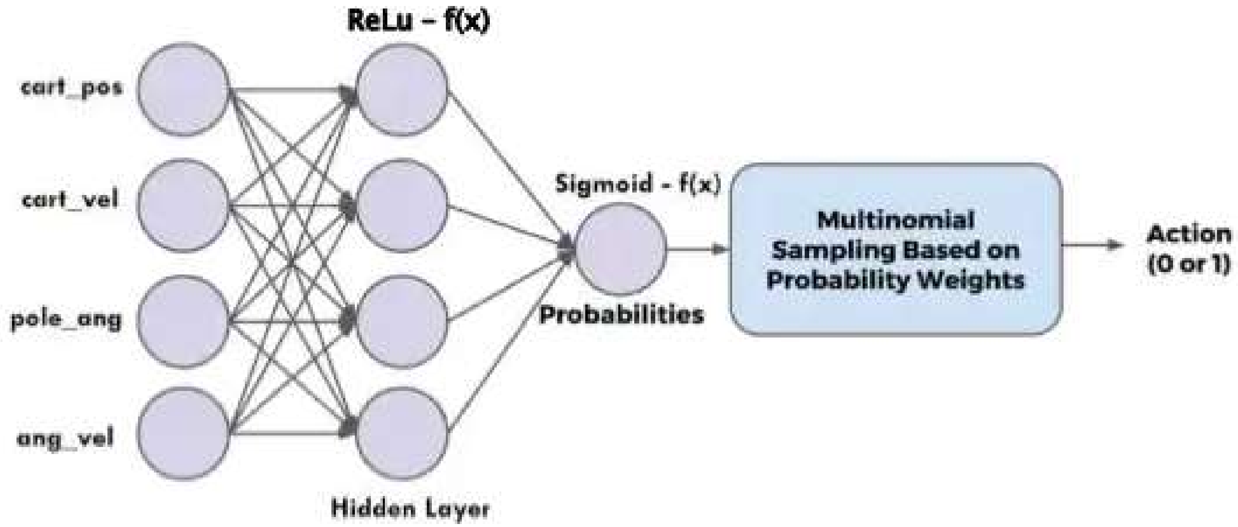
For the making of a hard-coded policy for a bot:



Code:

```
import gym
env=gym.make("CartPole-v1")
observation=env.reset()
for t in range(1000):
    env.render()
    #Defining a Hard-Coded Policy
    cart_pos, cart_vel, pole_ang, ang_vel = observation
    #Move Cart Right if Pole is Falling to the Right
    #Angle is measured off straight vertical line
    if pole_ang > 0:
        #Move Right
        action = 1
    else:
        #Move Left
        action = 0
    #Perform Action
    observation, reward, done, info=env.step(action)
env.close()
```

Neural Network in Reinforcement Learning



Implementing a RL Algorithm – Environment

Let's develop a Q-learning and SARSA model to solve this problem

```
import pygame
import gym
import numpy as np
import math
import matplotlib.pyplot as plt
%matplotlib inline
```

```
pygame 2.1.0 (SDL 2.0.16, Python 3.7.7)
Hello from the pygame community. https://www.pygame.org/contribute.html
```

Implementing a RL Algorithm – Environment

Prepare OpenAI Gym Environment

```
def prepare_env():
    #Environment creation
    env = gym.make("CartPole-v1")#, render_mode="human")

    #Environment values
    # Observation Space: [0] cart position along x-axis / [1] cart velocity / [2] pole angle (rad) / [3] pole angular velocity
    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)

    # Action Space: [0] push cart to the left / [1] push cart to the right
    print('Env. Action Space:', env.action_space)
    print('Env. Actions Space:', env.action_space.n)
    return env
```

Continuous min and max values for each observation variable, i.e., **[position of cart, velocity of cart, angle of pole, rotation rate of pole]**

Number of possible actions, i.e., **0 (left)** or **1 (right)**

```
prepare_env()
```

```
Env. Observation Space:  Box([-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38], [4.8000002e+00 3.4028235e+38 4.1887903e-01 3.4028235e+38], (4,), float32)
Env. Observation Space - High: [4.8000002e+00 3.4028235e+38 4.1887903e-01 3.4028235e+38]
Env. Observation Space - Low: [-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38]
Env. Action Space: Discrete(2)
Env. Actions Space: 2
```

Implementing a RL Algorithm – Hyper-parameters

Prepare Reinforcement Learning Model Hyper-parameters

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

Number of episodes: applied for training the reinforcement learning 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

Implementing a RL Algorithm – Q-Table


```
#Q-Table of size theta_state_size * theta_dot_state_size * env.action_space.n
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)
```

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

50 Pole Angle States
50 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 angular velocity states) – increased number of states provides a higher resolution of the state space

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

Dict model stats to verify model learning progression

Implementing a RL Algorithm – Discretize State Results

When we execute step() it returns a continuous state. Discretised_state(state) function converts these continuous states into discrete states.

For training the RL model, the Pole Angle and Angulary Velocity features will be used

```
def discretised_state(state, theta_minmax, theta_dot_minmax, theta_state_size, theta_dot_state_size):
    #state[2] -> theta
    #state[3] -> theta_dot
    discrete_state = np.array([0,0])        #Initialised discrete array

    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))
```

i.e., Angle of Pole & Angular Velocity State

Continuous State of Angle of Pole

Continuous State of Angular Velocity

Discrete State for Angle of Pole & Angular Velocity

Implementing a RL Algorithm – Q-Learning


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

    while not done:
        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, done, _, _ = 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 done:
            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=i+1
        curr_discrete_state = new_discrete_state
        episode_reward += reward

    ep_rewards.append(episode_reward)
```

Initialize variables at start of episode, including acquisition of first observation after environment reset

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

Save episode_reward for model learning analysis

Implementing a RL Algorithm – Q-Learning (Cont.)


```
        i=i+1
        curr_discrete_state = new_discrete_state
        episode_reward += reward

    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()

#Plot Model evolution performance
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) #bottom right
plt.title('CartPole Q-Learning')
plt.ylabel('Average reward/Episode')
plt.xlabel('Episodes')
plt.show()
```

Append episode_reward on model episode rewards list (for further learning model analysis)

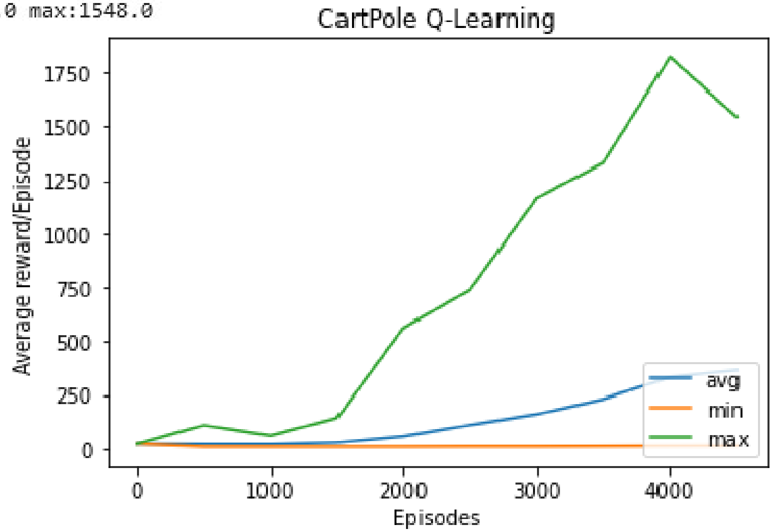
Append episode info. on episode rewards table dict

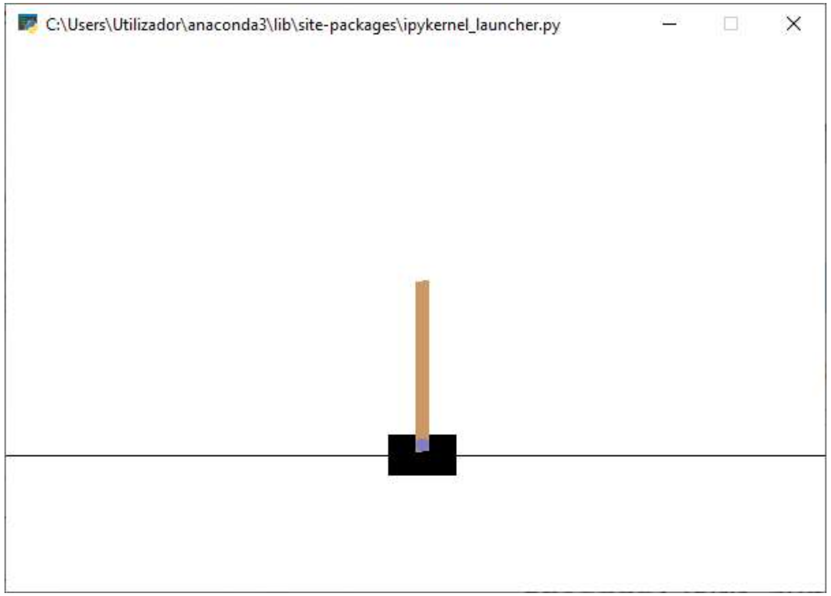
Based on episode rewards table, generate a plot to verify episode rewards evolution for each episode

Implementing a RL Algorithm – Q-Learning Results


```
ep_rewards_table_qlearning = train_cart_pole_qlearning(EPISODES, DISCOUNT, EPISODE_DISPLAY, LEARNING_RATE, EPSILON)
```

```
Env. Observation Space:  Box([-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38], [4.8000002e+00 3.4028235e+38 4.1887903e-01 3.4028235e+38], (4,), float32)
Env. Observation Space - High: [4.8000002e+00 3.4028235e+38 4.1887903e-01 3.4028235e+38]
Env. Observation Space - Low: [-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38]
Env. Action Space: Discrete(2)
Env. Actions Space: 2
Episode:0 avg:21.0 min:21.0 max:21.0
Episode:500 avg:19.19 min:8.0 max:108.0
Episode:1000 avg:18.836 min:8.0 max:60.0
Episode:1500 avg:26.816 min:8.0 max:141.0
Episode:2000 avg:56.316 min:8.0 max:563.0
Episode:2500 avg:109.13 min:9.0 max:739.0
Episode:3000 avg:159.29 min:9.0 max:1168.0
Episode:3500 avg:228.932 min:10.0 max:1331.0
Episode:4000 avg:332.124 min:11.0 max:1823.0
Episode:4500 avg:365.758 min:12.0 max:1548.0
```





Implementing a RL Algorithm – SARSA


```
for episode in range(EPISODES):
    episode_reward = 0
    done = 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)

    while not done:
        new_state, reward, done, _, _ = 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 done:
            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

    ep_rewards.append(episode_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

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

a_{t+1} s_{t+1}

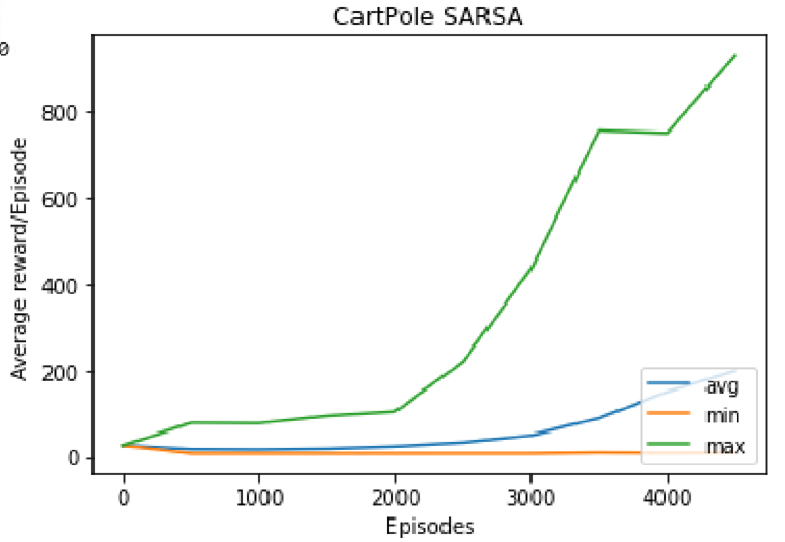
Implementing a RL Algorithm – SARSA (Cont.)

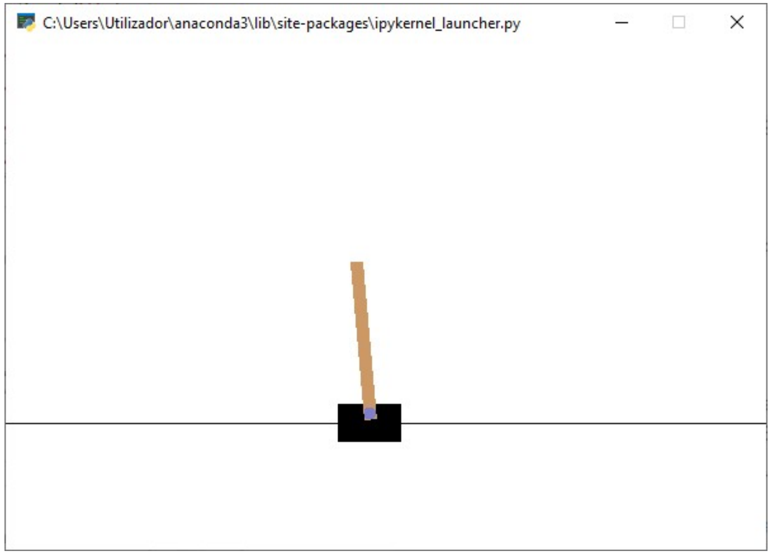

```
        curr_discrete_state = new_discrete_state
        action = new_action

        episode_reward += reward

    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()

#Plot Model evolution performance
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) #bottom right
plt.title('CartPole SARSA')
plt.ylabel('Average reward/Episode')
plt.xlabel('Episodes')
plt.show()
```

Implementing a RL Algorithm – SARSA Results

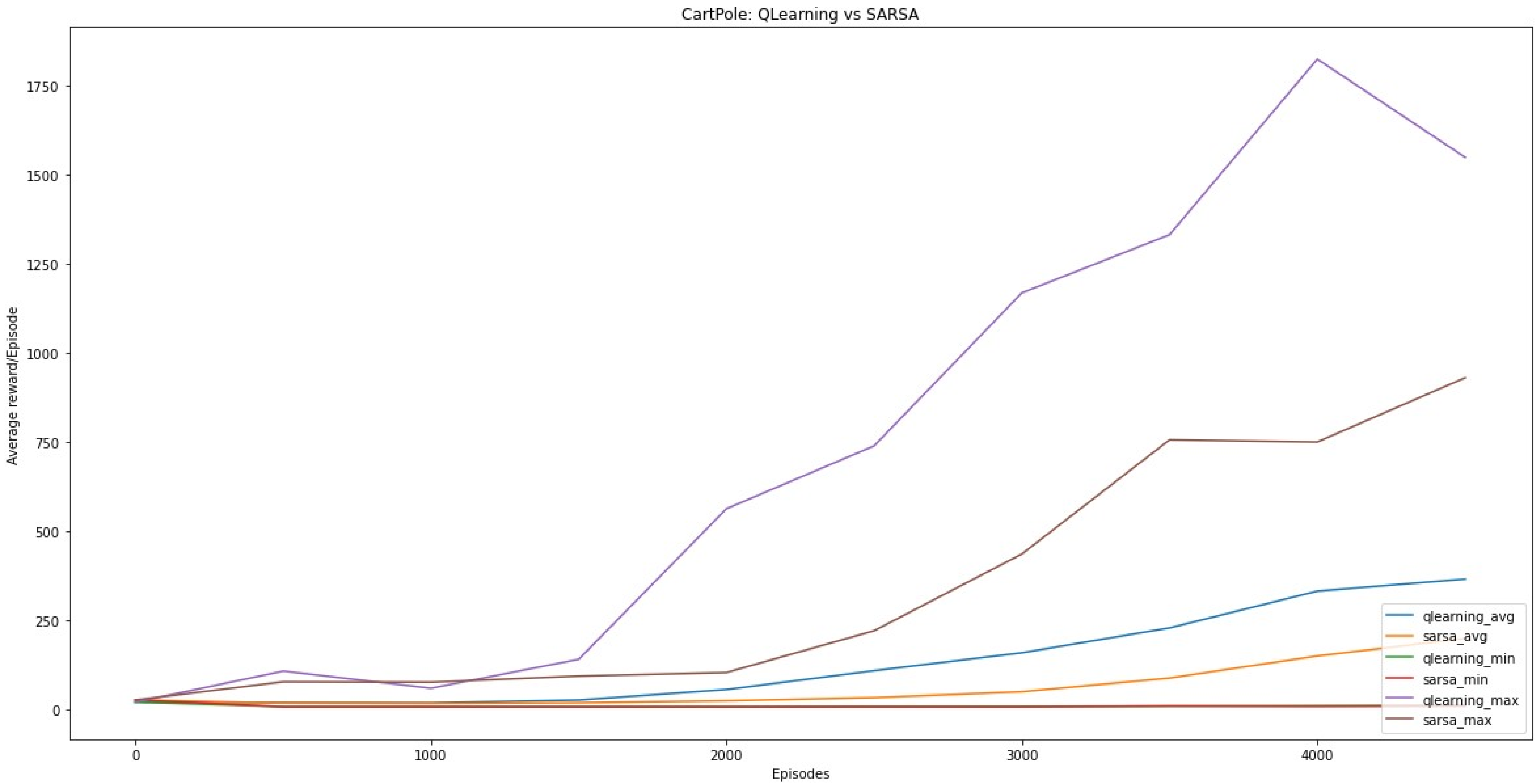

```
#SARSA
ep_rewards_table_sarsa = train_cart_pole_sarsa(EPISODES, DISCOUNT, EPISODE_DISPLAY, LEARNING_RATE, EPSILON)
```

```
Env. Observation Space:  Box([-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38], [4.8000002e+00 3.4028235e+38 4.1887903e-01 3.4028235e+38], (4,), float32)
Env. Observation Space - High: [4.8000002e+00 3.4028235e+38 4.1887903e-01 3.4028235e+38]
Env. Observation Space - Low: [-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38]
Env. Action Space: Discrete(2)
Env. Actions Space: 2
Episode:0 avg:27.0 min:27.0 max:27.0
Episode:500 avg:18.116 min:8.0 max:78.0
Episode:1000 avg:17.202 min:8.0 max:77.0
Episode:1500 avg:19.304 min:8.0 max:94.0
Episode:2000 avg:24.608 min:8.0 max:104.0
Episode:2500 avg:33.646 min:8.0 max:221.0
Episode:3000 avg:50.208 min:8.0 max:436.0
Episode:3500 avg:88.592 min:10.0 max:756.0
Episode:4000 avg:150.384 min:9.0 max:750.0
Episode:4500 avg:199.208 min:10.0 max:930.0
```





Implementing a RL Algorithm – Q-Learning vs SARSA

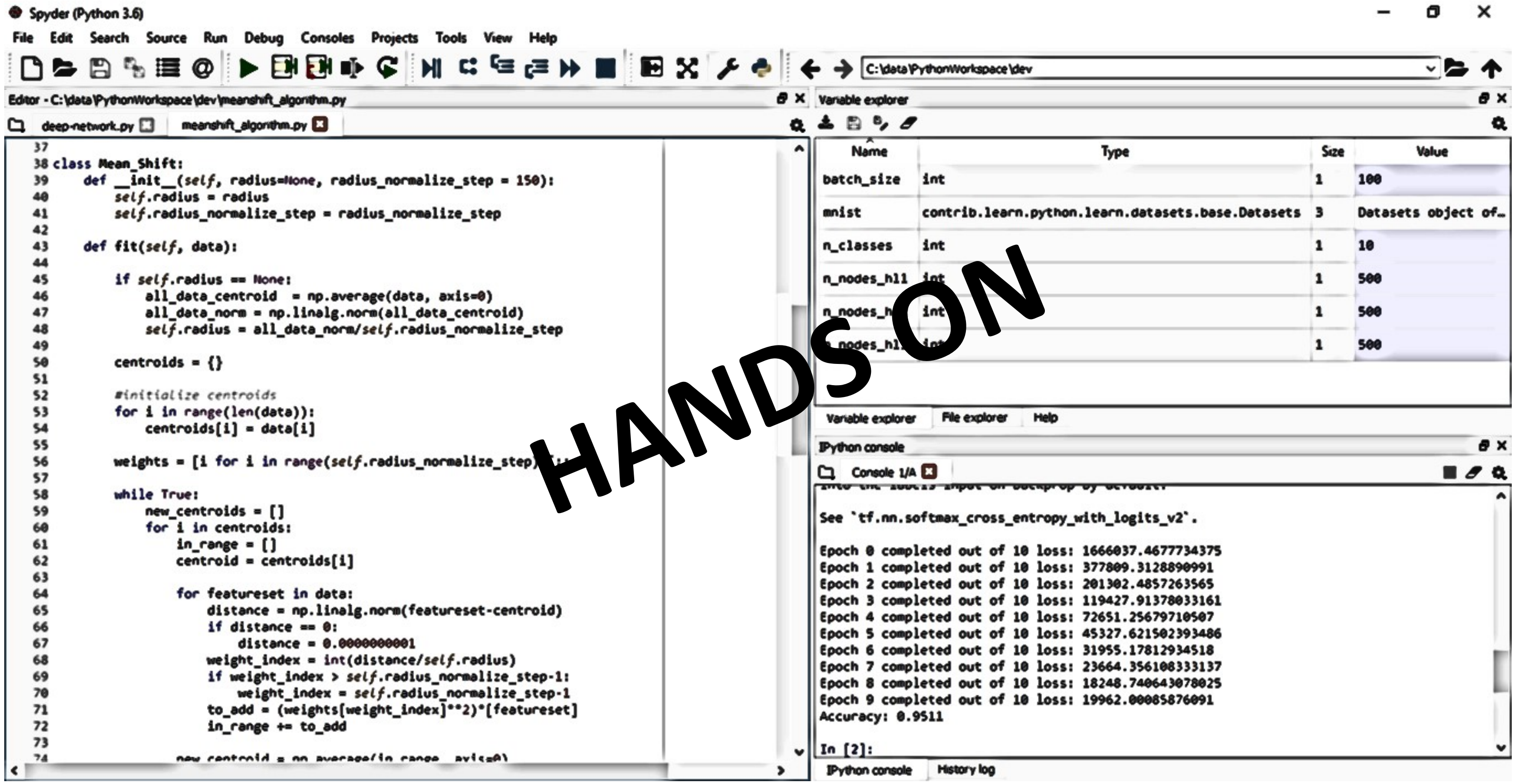


Implementing a RL Algorithm – Q-Learning vs SARSA

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

- The reward converges to a larger value in the case of Q-Learning than in the case of SARSA. This is possibly due to the action selection step. In Q-Learning, the action corresponding to the largest Q-value is selected. This therefore can cause a higher reward value to be obtained in the long run.
- The maximum reward is obtained by the agent in 4000 episodes for Q-Learning and 4500 episodes for SARSA in the case of cart pole.
- Training both models with more episodes and optimizing its hyper-parameters 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.

Hands On


Spyder (Python 3.6)
File Edit Search Source Run Debug Consoles Projects Tools View Help
Variable explorer
IPython console
HANDS ON