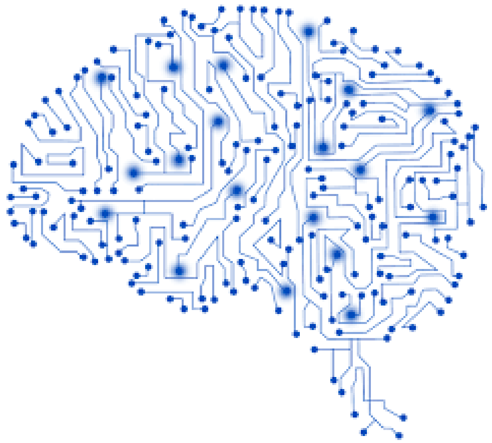
University of Minho
School of Engineering

ISLab



Dados e Aprendizagem Automática

Reinforcement Learning

Q-Learning vs SARSA

DAA @ MEI-1º/MiEI-4º – 1º Semestre

Bruno Fernandes, César Analide, Dalila Alves, Filipa Ferraz, Victor Alves

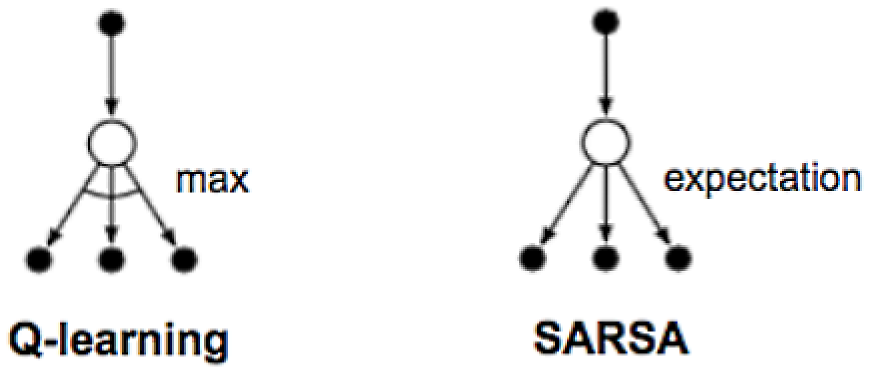
Part IX

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



What about Reinforcement Learning?

To implement our first RL algorithm, we will require:

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

  (Anaconda) `conda install -c conda-forge gymnasium`

  (Pip) `pip install gymnasium`

- You may also need Pyglet

  (Anaconda) `conda install -c conda-forge pyglet`

  (Pip) `pip install pyglet`

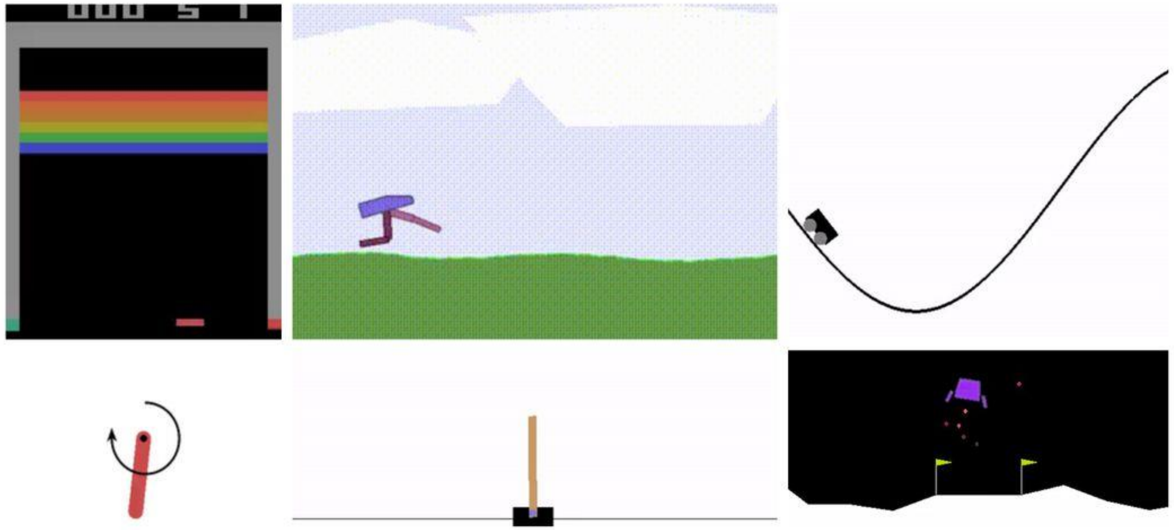
- And Pygame

  (Pip) `pip install pygame`

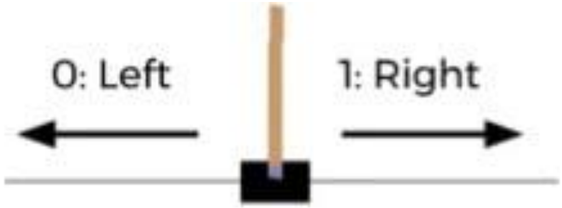
Gymnasium for Reinforcement Learning

Why?

- OpenAI Gymnasium is an open-source toolkit for developing and comparing reinforcement learning algorithms
- Gymnasium 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

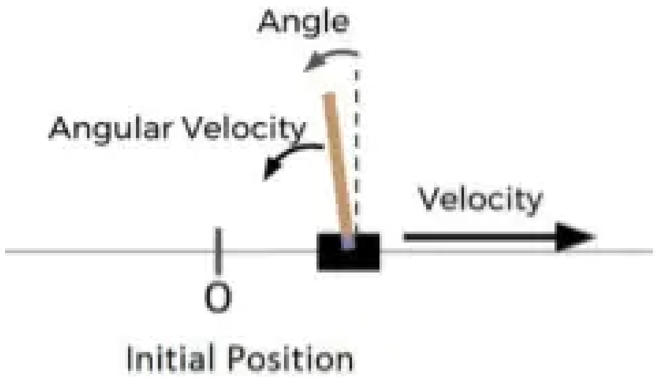


- An example of running an instance of the "CartPole-v1" (more info here) environment for 1000 time-step, rendering the environment at each step.

- 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

Environment initialization

```
import gymnasium as gym
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

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
- **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

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

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

Code v2 (still simple, yet more “complete”)

```python
import gymnasium as gym
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

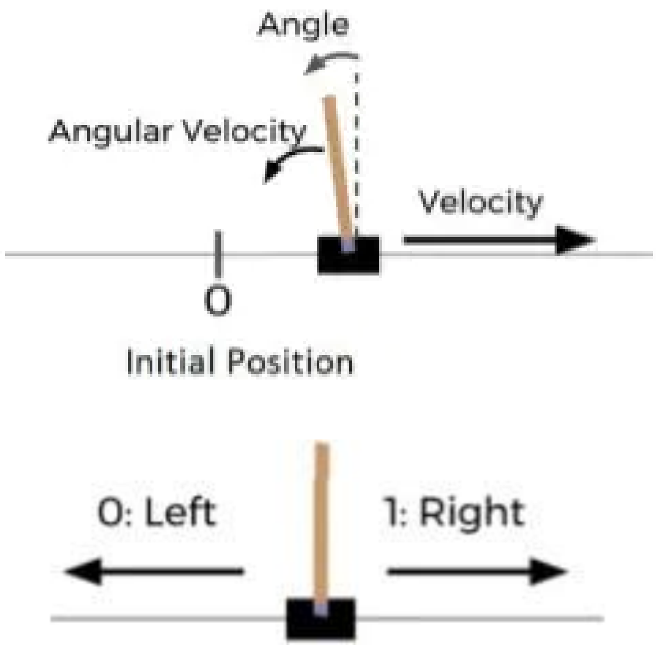
definition of number of time steps per episode

Bot perception for each step, based on action taken

verify if episode is over

OpenAI Gym Observations

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



Code:

```
import gymnasium as gym
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 #rigth
    else:
        action = 0 #left
    observation, reward, terminated, truncated, info = env.step(action)
env.close()
```

Implementing a RL Algorithm – Environment

Imports

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

Global Hiperparameters

```
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 – Environment

Let's preparate our environment and look at the Observation and Acton Spaces:

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

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.

Imagine spliting the number of possibilities into bins: in this case we will use 50 bins!

```
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-Table


```
def train_cart_pole_qlearning(EPISODES, DISCOUNT, EPISODE_DISPLAY, LEARNING_RATE, EPSILON):
    #Prepare OpenGym CartPole Environment
    env = prepare_env()

    #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)

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

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

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

Dict model stats to verify model learning progression

Implementing a RL Algorithm – Q-Learning


```python
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)

    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)
```

Initialize variables at start of an episode

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

Bot executes selected action and acquires observation from new state

Implementing a RL Algorithm – Q-Learning


```
    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

ep_rewards.append(episode_reward)
```

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


```python
    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()

return ep_rewards_table
```

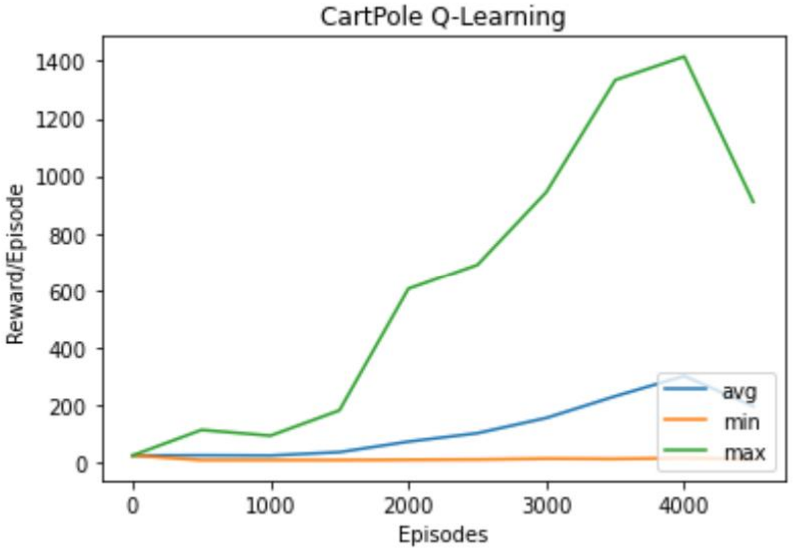
Append episode information on episode rewards table dict

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]
Episode:0 avg:24.0 min:24.0 max:24.0
Episode:500 avg:24.558 min:8.0 max:113.0
Episode:1000 avg:23.786 min:8.0 max:93.0
Episode:1500 avg:35.608 min:8.0 max:181.0
Episode:2000 avg:72.214 min:9.0 max:605.0
Episode:2500 avg:101.596 min:10.0 max:690.0
Episode:3000 avg:154.694 min:13.0 max:944.0
Episode:3500 avg:230.346 min:12.0 max:1334.0
Episode:4000 avg:300.564 min:15.0 max:1416.0
Episode:4500 avg:195.47 min:11.0 max:910.0
```



Implementing a RL Algorithm – SARSA


```
def train_cart_pole_sarsa(EPISODES, DISCOUNT, EPISODE_DISPLAY, LEARNING_RATE, EPSILON):
    #Prepare OpenGym CartPole Environment
    env = prepare_env()

    #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)

    #For stats
    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)
```

Implementing a RL Algorithm – SARSA


```python
    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

    ep_rewards.append(episode_reward)
```

Based on Exploration constant, select random action or action with highest Q-value for <u>next state</u>

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

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

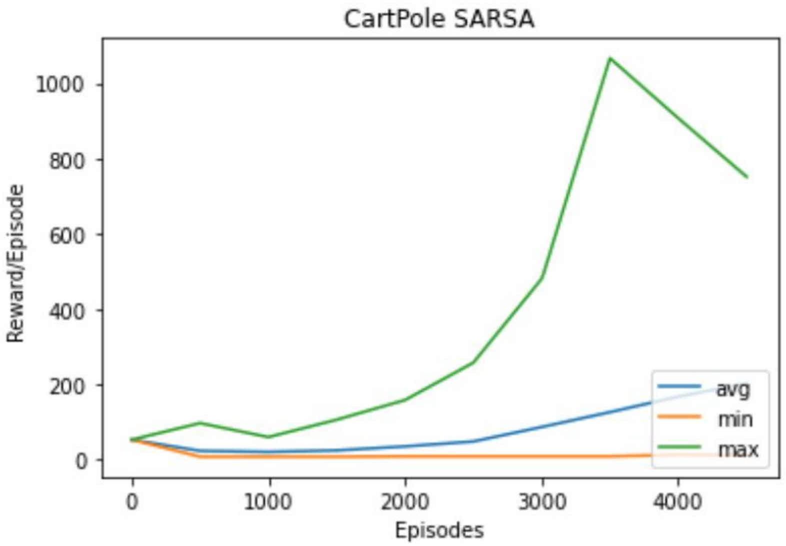
Implementing a RL Algorithm – SARSA


```
    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()

return ep_rewards_table
```

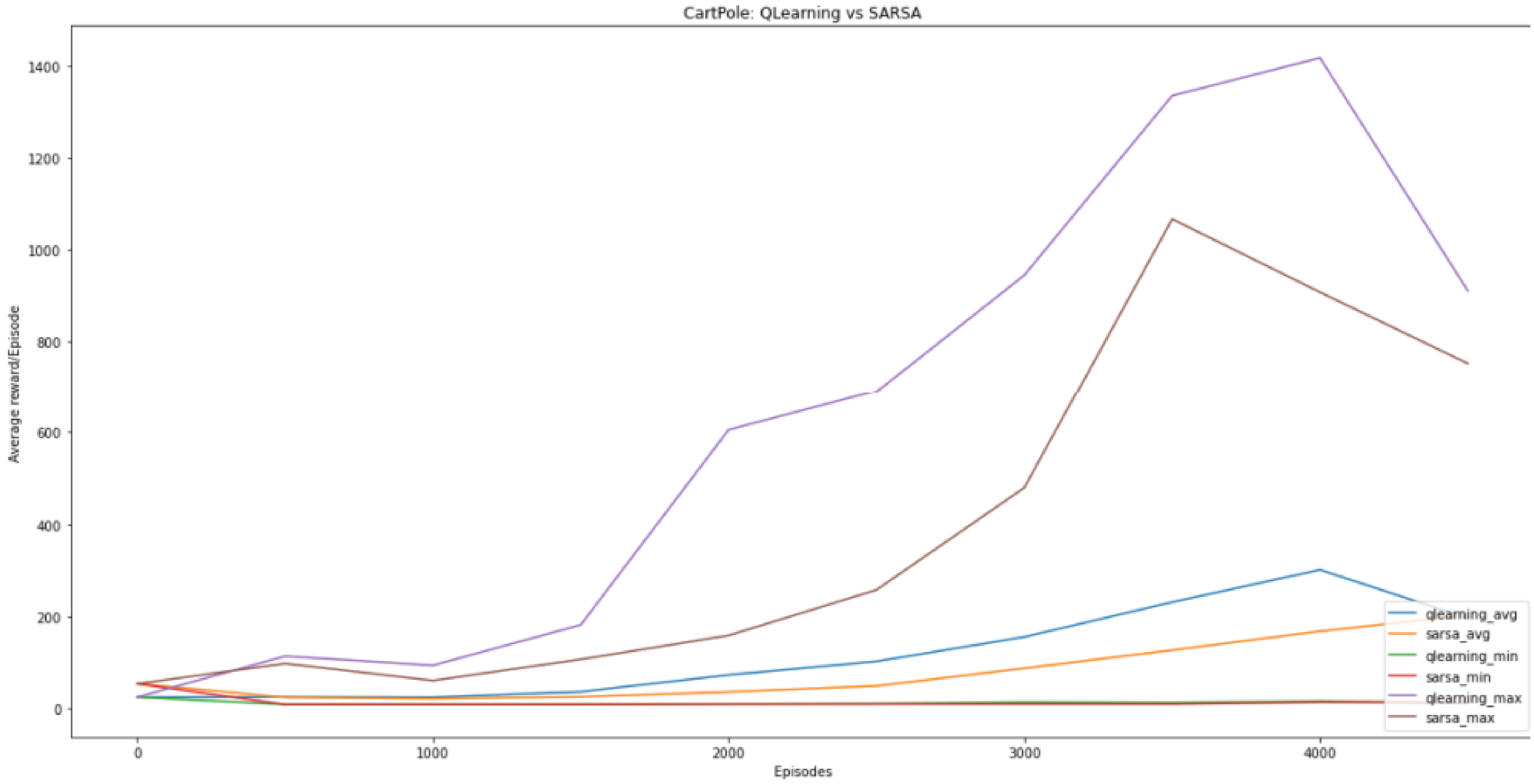
Implementing a RL Algorithm – SARSA Results


```
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]
Episode:0 avg:53.0 min:53.0 max:53.0
Episode:500 avg:23.506 min:8.0 max:97.0
Episode:1000 avg:20.756 min:8.0 max:60.0
Episode:1500 avg:24.712 min:8.0 max:106.0
Episode:2000 avg:35.276 min:9.0 max:158.0
Episode:2500 avg:48.318 min:9.0 max:257.0
Episode:3000 avg:86.654 min:9.0 max:479.0
Episode:3500 avg:126.016 min:9.0 max:1066.0
Episode:4000 avg:167.182 min:13.0 max:907.0
Episode:4500 avg:202.04 min:12.0 max:752.0
```



Implementing a RL Algorithm – Q-Learning vs SARSA

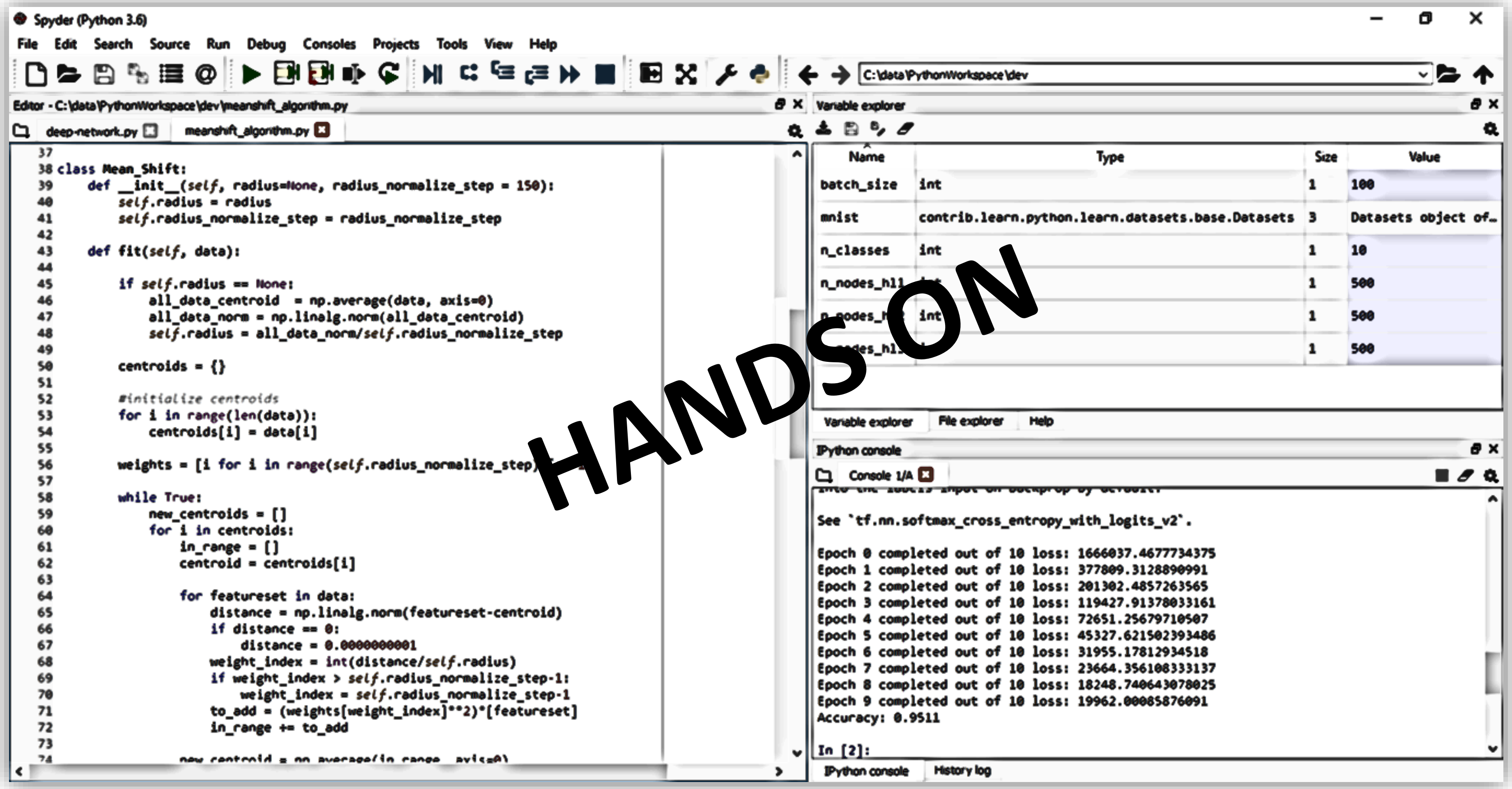


Implementing a RL Algorithm – Q-Learning vs SARSA

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 4350 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
C:\data\PythonWorkspace\dev
Editor - C:\data\PythonWorkspace\dev\meanshift_algorithm.py
deep-network.py
meanshift_algorithm.py
class Mean_Shift:
def __init__(self, radius=None, radius_normalize_step = 150):
self.radius = radius
self.radius_normalize_step = radius_normalize_step
def fit(self, data):
if self.radius == None:
all_data_centroid = np.average(data, axis=0)
all_data_norm = np.linalg.norm(all_data_centroid)
self.radius = all_data_norm/self.radius_normalize_step
centroids = {}
#initialize centroids
for i in range(len(data)):
centroids[i] = data[i]
while True:
new_centroids = []
for i in centroids:
in_range = []
centroid = centroids[i]
for featureset in data:
distance = np.linalg.norm(featureset-centroid)
if distance == 0:
distance = 0.0000000001
weight_index = int(distance/self.radius)
if weight_index > self.radius_normalize_step-1:
weight_index = self.radius_normalize_step-1
to_add = (weights[weight_index]**2)*[featureset]
in_range += to_add
HANDS ON
Variable explorer
Name
Type
Size
Value
batch_size
int
1
100
mnist
contrib.learn.python.learn.datasets.base.Datasets
3
Datasets object of…
n_classes
int
1
10
500
500
500
Variable explorer
File explorer
Help
IPython console
Console 1/A
See `tf.nn.softmax_cross_entropy_with_logits_v2`.
Epoch 0 completed out of 10 loss: 1666037.4677734375
Epoch 1 completed out of 10 loss: 377809.3128890991
Epoch 2 completed out of 10 loss: 201302.4857263565
Epoch 3 completed out of 10 loss: 119427.91378033161
Epoch 4 completed out of 10 loss: 72651.25679710507
Epoch 5 completed out of 10 loss: 45327.621502393486
Epoch 6 completed out of 10 loss: 31955.17812934518
Epoch 7 completed out of 10 loss: 23664.356108333137
Epoch 8 completed out of 10 loss: 18248.740643078025
Epoch 9 completed out of 10 loss: 19962.00085876091
Accuracy: 0.9511
In [2]:
IPython console
History log