Getting Started with Reinforcement Learning and Open AI Gym
Solving the Mountain Car environment using Q-learning.

This is the third in a series of articles on Reinforcement Learning and Open AI Gym. Part 1 can be found here, while Part 2 can be found here.

Introduction
Reinforcement learning (RL) is the branch of machine learning that deals with learning from interacting with an environment where feedback may be delayed.

Although RL is a very powerful tool that has been successfully applied to problems ranging from the optimization of chemical reactions to teaching a computer to play video games, it has historically been difficult to get started with, due to the lack of availability of interesting and challenging environments on which to experiment.

This is where OpenAI Gym comes in.

OpenAI Gym is a Python package comprising a selection of RL environments, ranging from simple “toy” environments to more challenging environments, including simulated robotics environments and Atari video game environments.

It was developed with the aim of becoming a standardized environment and benchmark for RL research.

In this article, we will use the OpenAI Gym Mountain Car environment to demonstrate how to get started in using this exciting tool and show how Q-learning can be used to solve this problem.

This tutorial assumes you already have OpenAI Gym installed on your computer. If you haven’t done so, installation instructions can be found here for Windows and here for Mac or Linux.

**The Mountain Car Problem**
On the OpenAI Gym website, the Mountain Car problem is described as follows:

A car is on a one-dimensional track, positioned between two “mountains”. The goal is to drive up the mountain on the right; however, the car’s engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum.

The car’s state, at any point in time, is given by a vector containing its horizontal position and velocity. The car commences each episode stationary, at the bottom of the valley between the hills (at position approximately -0.5), and the episode ends when either the car reaches the flag (position > 0.5) or after 200 moves.

At each move, the car has three actions available to it: push left, push right or do nothing, and a penalty of 1 unit is applied for each move taken (including doing nothing). This means that, unless the can figure out a way to ascend the mountain in less than 200 moves, it will always achieve a total “reward” of -200 units.

To begin with this environment, import and initialize it as follows:

```python
import gym
env = gym.make('MountainCar-v0')
env.reset()
```

**Exploring the Environment**

Once you have imported the Mountain car environment, the next step is to explore it. All RL environments have a state space (that is, the set of all possible states of the environment you can be in) and an action space (that is, the set of all actions that you can take within the environment).
You can see the size of these spaces using:

```python
> print('State space: ', env.observation_space)
State space: Box(2,)

> print('Action space: ', env.action_space)
Action space: Discrete(3)
```

This tells us that the state space represents a 2-dimensional box, so each state observation is a vector of 2 (float) values, and that the action space comprises three discrete actions (which is what we already knew).

By default, the three actions are represented by the integers 0, 1 and 2. However, we don’t know what values the elements of the state vector can take. This can be found using:

```python
> print(env.observation_space.low)
[-1.2  -0.07]

> print(env.observation_space.high)
[0.6   0.07]
```

From this, we can see that the first element of the state vector (representing the cart’s position) can take on any value in the range -1.2 to 0.6, while the second element (representing the cart’s velocity) can take on any value in the range -0.07 to 0.07.

When we introduced the Q-learning algorithm in the first article in this series, we said that it was guaranteed to converge provided each state-action pair is visited a sufficiently large number of times. In this situation, however, we are dealing with a continuous state space, which means that there are infinitely many state-action pairs, making it impossible to satisfy this condition.

One way to address this problem is to use deep Q-networks (DQNs). DQNs combine deep learning with Q-learning by using a deep neural network as an approximator for the Q-function. DQNs have been successfully applied to developing artificial intelligence capable of playing Atari video games.
However, for a problem as simple as the Mountain Car problem, this may be a bit of overkill.

An alternative approach is to just discretize the state space. One simple way in which this can be done is to round the first element of the state vector to the nearest 0.1 and the second element to the nearest 0.01, and then (for convenience) multiply the first element by 10 and the second by 100.

This reduces the number of state-action pairs down to 855, which now makes it possible to satisfy the condition required for Q-learning to converge.

**Q-Learning Recap**

In the first article in this series, we went through the Q-learning algorithm in detail. When going though this algorithm, we assumed a one-dimensional state space, so our goal was to find the optimal Q table, \( Q(s,a) \).

In this problem, since we’re dealing with a two-dimensional state space, we replace \( Q(s,a) \) with \( Q(s1, s2, a) \), but other than that, the Q-learning algorithm remains more or less the same.

To recap, the algorithm is as follows:

1. Initialize \( Q(s1, s2, a) \) by setting all of the elements equal to small random values;
2. Observe the current state, \((s1, s2)\);
3. Based on the exploration strategy, choose an action to take, \( a \);
4. Take action \( a \) and observe the resulting reward, \( r \), and the new state of the environment, \((s1', s2')\);
5. Update \( Q(s1, s2, a) \) based on the update rule:

\[
Q'(s1, s2, a) = (1 - w)Q(s1, s2, a) + w(r + dQ(s1', s2', \text{argmax } a' Q(s1', s2', a')))
\]

Where \( w \) is the learning rate and \( d \) is the discount rate;
6. Repeat steps 2–5 until convergence.
Q-Learning in OpenAI Gym

To implement Q-learning in OpenAI Gym, we need ways of observing the current state; taking an action and observing the consequences of that action. These can be done as follows.

The initial state of an environment is returned when you reset the environment:

```
> print(env.reset())
array([-0.50926558, 0.])
```

To take an action (for example, `a = 2`), it is necessary to “step forward” the environment by that action using the `step()` method. This returns a 4-tuple giving the new state, reward, a Boolean indicating whether or not the episode has terminated (due to the goal being reached or 200 steps having elapsed), and any additional information (this is always empty for this problem).

```
> print(env.step(2))
(array([-0.50837305, 0.00089253]), -1.0, False, {})
```

If we assume an epsilon-greedy exploration strategy where epsilon decays linearly to a specified minimum (`min_eps`) over the total number of episodes, we can put all of the above together with the algorithm from the previous section and produce the following function for implementing Q-learning.

```python
import numpy as np
import gym
import matplotlib.pyplot as plt

# Import and initialize Mountain Car Environment
env = gym.make('MountainCar-v0')
env.reset()

# Define Q-learning function

def QLearning(env, learning, discount, epsilon, min_eps, episodes):
    # Determine size of discretized state space
    num_states = (env.observation_space.high - env.observation_space.low) *
```
np.array([10, 100])
num_states = np.round(num_states, 0).astype(int) + 1

# Initialize Q table
Q = np.random.uniform(low = -1, high = 1,
                      size = (num_states[0], num_states[1],
                              env.action_space.n))

# Initialize variables to track rewards
reward_list = []
ave_reward_list = []

# Calculate episodic reduction in epsilon
reduction = (epsilon - min_eps)/episodes

# Run Q learning algorithm
for i in range(episodes):
    # Initialize parameters
    done = False
tot_reward, reward = 0, 0
    state = env.reset()

    # Discretize state
    state_adj = (state - env.observation_space.low)*np.array([10, 100])
    state_adj = np.round(state_adj, 0).astype(int)

    while done != True:
        # Render environment for last five episodes
        if i >= (episodes - 20):
            env.render()

        # Determine next action - epsilon greedy strategy
        if np.random.random() < 1 - epsilon:
            action = np.argmax(Q[state_adj[0], state_adj[1]])
        else:
            action = np.random.randint(0, env.action_space.n)

        # Get next state and reward
        state2, reward, done, info = env.step(action)

        # Discretize state2
        state2_adj = (state2 - env.observation_space.low)*np.array([10, 100])
        state2_adj = np.round(state2_adj, 0).astype(int)
# Allow for terminal states
if done and state2[0] >= 0.5:
    Q[state_adj[0], state_adj[1], action] = reward

# Adjust Q value for current state
else:
    delta = learning*(reward +
        discount*np.max(Q[state_adj[0],
                         state_adj[1]]) -
        Q[state_adj[0], state_adj[1], action])
    Q[state_adj[0], state_adj[1], action] += delta

# Update variables
    tot_reward += reward
    state_adj = state2_adj

# Decay epsilon
if epsilon > min_eps:
    epsilon -= reduction

# Track rewards
    reward_list.append(tot_reward)

if (i+1) % 100 == 0:
    ave_reward = np.mean(reward_list)
    ave_reward_list.append(ave_reward)
    reward_list = []

if (i+1) % 100 == 0:
    print('Episode {} Average Reward: {}'.format(i+1, ave_reward))

    env.close()
return ave_reward_list

# Run Q-learning algorithm
rewards = QLearning(env, 0.2, 0.9, 0.8, 0, 5000)

# Plot Rewards
plt.plot(100*(np.arange(len(rewards)) + 1), rewards)
plt.xlabel('Episodes')
plt.ylabel('Average Reward')
plt.title('Average Reward vs Episodes')
plt.savefig('rewards.jpg')
plt.close()
For tracking purposes, this function returns a list containing the average total reward for each run of 100 episodes. It also visualizes the movements of the Mountain Car for the final 10 episodes using the `env.render()` method.

The environment is only visualized for the final 10 episodes, rather than for all episodes, because visualizing the environment dramatically increases the code run time.

Suppose we assuming a learning rate of 0.2, a discount rate of 0.9, an initial epsilon value of 0.8, and a minimum epsilon value of 0. If we run the algorithm for 500 episodes, at the end of these episodes, the car has started to figure out that it needs to rock back and forth to gain the momentum necessary to ascend the mountain, but can only make it about halfway up.

If we increase the number of episodes by an order of magnitude to 5000, however, by the end of the 5000 episodes the car is able to ascend the mountain perfectly, almost every time.

Plotting the average reward vs the episode number for the 5000 episodes, we can see that, initially, the average reward is fairly flat, with each run terminating once the maximum 200 movements is reached. This is the exploration phase of the algorithm.
Nevertheless, in the final 1000 episodes, the algorithm takes what it’s learned through exploration and exploits it in order to increase the average reward, with the episodes now ending in less than 200 movements, as the car learns to ascend the mountain.

This exploitation phase is only possible because the algorithm was given sufficient time to explore the environment, which is why the car was unable to climb the mountain when the algorithm was only run for 500 episodes.

**Summary**

In this article, we have demonstrated how RL can be used to solve the OpenAI Gym Mountain Car problem. To solve this problem, it was necessary to discretize our state space and make some small modifications to the Q-learning algorithm, but other than that, the technique used was the same as that used to solve the simple grid world problem in the first article in this series.

But this is just one of the many environments available to users in Open AI Gym. For readers interested in trying out the skills they have learned in this article on their own, I recommend experimenting with any of the other Classic Control problem (available here) and then moving on to the Box 2D problems.

By continually modifying and building on the Q-learning algorithm, it should be possible to solve any of the environments available to users of OpenAI Gym. Nevertheless, as with everything, the first step is learning the basics. This is what we have succeeded in doing today.