Reinforcement Learning

Dr Rob Collins

Version 3, 21st August 2021

Introduction

This workshop focusses on Reinfocement Learning. The key idea here is to consider an agent that is exploring a world or envionment, to achieve some goal. The goal is often phrased in terms of a reward. the Agent is normally able to make some observation of the World, based on their current location (state) in that world.

We will be using the library Open AI Gym library to support this workshop. AI Gym is documented here: https://gym.openai.com/

In this workshop we will focus on the 'cart-pole' problem. That is, the problem of balancing a pole simply by moving its base left and right. The observations of the world include the position of the trolly that the pole rests on, the angle of the pole and its speed.

This workshop presents two attempts to solve the problem. The first is rather simple .. and in fact does not really succeed! However, it does illustrate how the AI Gymn software works.

The second approach uses 'Q-Learning'. Q-Learning depends on playing the game many, many times. Each time the game is played, the various 'states' of the game are recorded. For eachs state, those actions that lead to a succesful outcome are given a positive score. This positive score increases each time that the Agent has success performing a specific action in a specific state.

Step-by-step the Agent learns the best policy. That is, it learns which is the most succesful action to take in each state. The Agent is set up to select high-scoring actions more frequently than low-scoring actions. However, the Agent also has to do some 'exploration' - that is, sometimes it will take some random actions. This logic creates a balance between 'exploration' and 'exploitation'.

Instructions for Students

This workbook includes some empty code-blocks. In those cases you are required to create the code for the block based on your previous learning on this course. This activity is indicated in the workbook below with a '**Student Task**' indicator.

In working through the following notebook, please do the following:

1. Create an empty Jupyter notebook on your own machine
2. Enter all of the **Python code** from this notebook into **code blocks** in your notebook
3. **Execute each of the code blocks** to check your understanding
4. You **do not need to** replicate all of the explanatory / tutorial text in text (markdown) blocks
5. You **may** add your own comments and description into text (markdown) blocks if it helps you remember what the commands do
6. You **may** add further code blocks to experiment with the commands and try out other things
7. Enter and run as many of the code blocks as you can within the time available during class
8. **After class**, enter and run any remaining code blocks that you have not been able to complete in class

The numbers shown in the 'In [n]' text on your Jupyter notebook are likely to be different to the ones shown here because they are updated each time a block is executed.

Note to Tutor:

Use Environment 'ML_Course' for demonstration

1. Creation of the Learning Environment (World)

As always, we start by importing the required libraries. We have used numpy and pandas many time before in this course. The only new item here is 'gym' and this is the library for Open AI Gym.

In [40]:
```
import gym
import numpy as np
import pandas as pd
```

Open AI Gym provides a variety of different environments for exploring. The following block creates a 'Cart-pole' environment as introduced during the lecture. https://gym.openai.com/envs/CartPole-v1/

In [41]:
```
env = gym.make('CartPole-v1')
```

There is a requirement to re-set the environment before it is first used. It can also be reset at any time, if you want to write a programme that 'experiments' with different policies:

In [42]:
```
obs = env.reset()
obs
```

Out[42]:
```
array([-0.01810062, -0.02537872, -0.04474021,  0.00880797])
```

Observations are returned as an array of numbers. The function below simply prints that array in a more explicit format.

In [43]:
```
def showObs (theObs):
    print('Position       = ', theObs[0], '(0 = centre, >0 = means to the right)')
    print('Speed          = ', theObs[1])
    print('Angle          = ', theObs[2])
    print('Rotation speed = ', theObs[3], '(Positive means clockwise)')
```

In [44]:
```
showObs( obs )
```

```
Position       =  -0.018100623510893636 (0 = centre, >0 = means to the right)
Speed          =  -0.02537872041997895
Angle          =  -0.044740214591793816
Rotation speed =  0.008807966197100593 (Positive means clockwise)
```

The 'render()' function for each environment creates a window and draws a picture of the Environment and Actor. I have commented out this command in my programme (i.e. used a '#' to remove it) as I found that rendering the image crashed my programme. You might experiment by removing the '#' .. but please save your programme before you do so to prevent any loss!

In [6]:
```
# env.render()
```

Open AI Gym provides a variety of environments for experimentation. They each offer a common set of functions for learning about the specific environment. Thus, for example, you can learn about the range of Actions you can perform:

In [45]:
```
env.action_space
```

Out[45]:
```
Discrete(2)
```

The number of options in the action space is also given by ..

In [46]:
```
env.action_space.n
```

Out[46]:
```
2
```

In this case, there are only two possible actions. These are each represented by numbers .. 0 means 'push the trolly to the left' and 1 means 'push the trolly to the right'

In [47]:
```
push_left = 0
push_right = 1
```

We can also find out about the observations the Agent can make. In this case, there are four things that the Agent can observe, represented by four numbers. The four numbers represent the following information:

1. Position of the trolly .. where '0' is the centre of the environment and >0 means to the right
2. Speed
3. Agle of the pole from the verical
4. Rotation speed .. where a positive number means a clockwise rotation

AI Gym can tell us the maximum and minimum values for each of these observations:

In [48]:
```
env.observation_space.low
```

Out[48]:
```
array([-4.8000002e+00, -3.4028235e+38, -4.1887903e-01, -3.4028235e+38],
      dtype=float32)
```

In [49]:
```
env.observation_space.high
```

Out[49]:
```
array([4.8000002e+00, 3.4028235e+38, 4.1887903e-01, 3.4028235e+38],
      dtype=float32)
```

We can perform an Action in the environment using the 'step' function. For example, in the following block we take an Action which pushes the cart to the right.

In [50]:
```
action = push_right
obs, reward, done, info = env.step( action )
```

Having pushed the trolly to the right, we can Observe the World in its new state, by looking at the 'obs' variable.

In [51]:
```
showObs( obs )
```

```
Position       =  -0.018608197919293214 (0 = centre, >0 = means to the right)
Speed          =  0.17035536261865827
Angle          =  -0.0445640552678518
Rotation speed =  -0.2976485939158875 (Positive means clockwise)
```

The reward in this case is just 1. For Cart-pole, the Agent gets a reward of 1 for each move they make without the pole falling over. The objective is to keep the pole upright for as long as possible.

In [52]:
```
reward
```

Out[52]:
```
1.0
```

The environment will signal if the particular 'episode' is finished. For example, if the pole has fallen over. In this case it has not.

In [53]:
```
done
```

Out[53]:
```
False
```

Some environments provide some extra information. This might include how many 'lives' a player has. In the case of cart-pole there is no extra information available.

In [54]:
```
info
```

Out[54]:
```
{}
```

2. A simple attempt at controlling the cart-pole (Does not work!)

First, we will use a very simple policy to try to solve the cart-pole problem. This Policy seems like an obvious idea ...

1. If the angle of the pole is to the left, then push the cart to the left
2. If the angle of the pole is to the right, then push the cart to the right

This seems to be a good idea! But sadly, you will see that it is too simple and does not work at all well.

First, here is the policy, written as a function. If the angle part of the Observation ('obs[2]') is less than zero it returns 0, otherwise it returns 1:

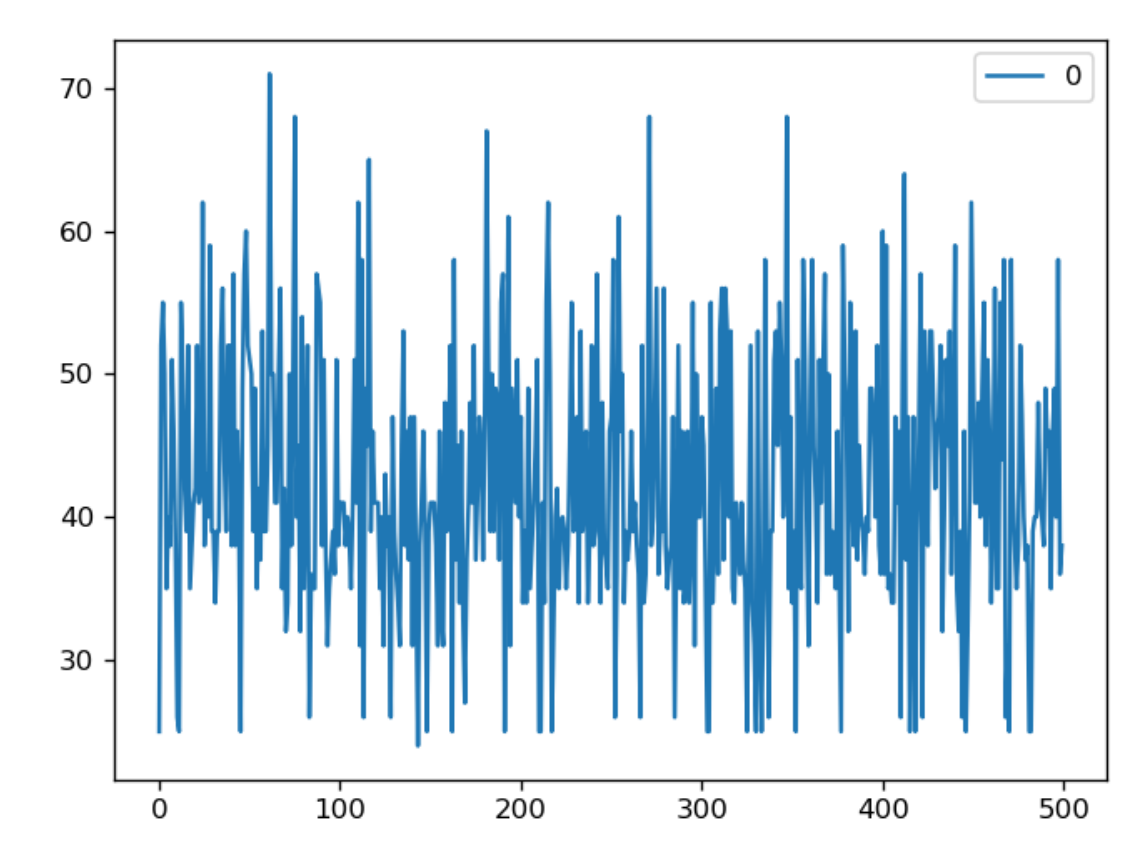
In [55]:
```
def simple_policy( obs ):
    angle = obs[2]
    return 0 if angle < 0 else 1
```

Now we have a loop that plays the game multiple times. Each time the game is played we add up the total reward in 'episode_rewards'. At the end of a specific game, we take this total reward for the game and add it to a list called 'totals'. Thus, when the game is finished we will be able to plot a graph of all of the games to see how well this policy worked.

In [56]:
```
totals = []
for episode in range(500):
    obs=env.reset()
    episode_rewards = 0
    for step in range(200):
        action = simple_policy(obs)
        obs, reward, done, info = env.step(action)
        episode_rewards += reward
        # env.render()                        # Remove the comment character (#) to draw the environment each
        if done:
            break
    totals.append( episode_rewards )
```

Now we can print a line chart showing the score for each episode ..

In [57]:
```
totals_chart = pd.DataFrame(totals)
ax = totals_chart.plot.line()
```



There is clearly no learning occuring. The graph above looks random! It certainly does not get better as each episode is played.

Also, the scores are not very good. They can be summarised like this:

In [58]:
```
totals_chart.describe()
```

Out[58]:

	0
count	500.000000
mean	42.248000
std	9.101438
min	24.000000
25%	36.000000
50%	40.000000
75%	49.000000
max	71.000000

When I ran this code, the highest score was 72. The average was only 41.

So it is clear that we need a better strategy if we are going to learn to play this game!

2. Q-Learning

As always ... we start by loading the requried libraries:

In [59]:
```
import math                                # Used for number conversion
%matplotlib notebook
from matplotlib import pyplot as plt       # Used to plot the chart
```

As above .. we are going to use the 'Cart-pole' environment

In [60]:
```
env = gym.make('CartPole-v0')
```

2.1 Defining the Q-table

We need to define the 'Q-table'. This table has two parts:

1. A part representing the state of the Actor, and
2. A part containing a score for each of the possible actions that can be taken in that state.

As described in the lecture, the 'Cart-pole' environment represents its state as continuous numbers. We can't build a Q-table based on continuous variables as it would be vast. So these numbers need to be broken down into discrete 'buckets'.

The 'buckets' tuple, defines the number of discrete 'groups' (buckets) that each of the 4 state variables are divided into.

Note that the number of buckets for each state variable was developed experimentally. That is, there was no deep theory or mathematics that lead to the following number of buckets. Rather, lot's of different numbers were tried until one of them worked!

Student task : After you have completed this workshop, come back to this point and modify the number of buckets to see what impact it has on the performance of the learning algorithm.

In [61]:
```
buckets=(1, 1, 6, 12,)          # down-scaling feature space to discrete range
buckets
```

Out[61]:
```
(1, 1, 6, 12)
```

The we use numpy, to create a multi-dimensional array to store each of the state - action pairs:

In [62]:
```
Q = np.zeros(buckets + (env.action_space.n,))
```

The block above is quite complicated to read unless you are very familiar with Python. So let me explain ...

'np.zeros' can be used to create arrays of many dimensions and fill them with zeros. Thus, a simple example would be:

In [63]:
```
np.zeros(5)
```

Out[63]:
```
array([0., 0., 0., 0., 0.])
```

Which creates a 1-dimensional array with 5 elements.

Or we could produce a more complex array like this:

In [64]:
```
array_shape = (2,3)
np.zeros(array_shape)
```

Out[64]:
```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

Which uses the tuple '(2,3)' to define an 2 x 3 array.

We can go further:

In [65]:
```
array_shape = (2,3,4)
np.zeros(array_shape)
```

Out[65]:
```
array([[[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]],

       [[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]]])
```

The tuple '(2,3,4)' defines an array of 2x3x4 ... and so on.

Now, we need an array that has space for each of the state variables (as defined in buckets) and also space for the possible actions. The following tuple gives us an array of the right shape:

In [66]:
```
buckets + (env.action_space.n,)
```

Out[66]:
```
(1, 1, 6, 12, 2)
```

So at this point we can create an array as required:

In [ ]:
```
# initialising Q-table
Q = np.zeros(buckets + (env.action_space.n,))
Q
```

2.2 Defining other controlling variables

The following block sets the value of the various constants that define the behviour of the programme. You learnt about some of these during the lecture.

In [68]:
```
n_episodes = 5                  # Set to 200 training episodes
min_alpha = 0.1                  # learning rate
min_epsilon = 0.1                # exploration rate
gamma = 1.0                      # discount factor
ada_divisor = 25                 # Adaptation rate for alpha and gamma
max_env_steps=None,
monitor=False
```

2.3 Translating the state from Continuous to Discrete values

We now need to define a function that takes an observation represented as a continuous number, and translates it into a discrete version. The discrete version will be used as an index into the Q-table:

In [69]:
```
def discretize(obs):
    upper_bounds = [env.observation_space.high[0], 0.5, env.observation_space.high[2], math.radians(50)]
    lower_bounds = [env.observation_space.low[0], -0.5, env.observation_space.low[2], -math.radians(50)]
    ratios = [(obs[i] + abs(lower_bounds[i])) / (upper_bounds[i] - lower_bounds[i]) for i in range(len(obs))]
    new_obs = [int(round((buckets[i] - 1) * ratios[i])) for i in range(len(obs))]
    new_obs = [min(buckets[i] - 1, max(0, new_obs[i])) for i in range(len(obs))]
    return tuple(new_obs)
```

We can use this function to translate the previous observation from this:

In [70]:
```
showObs( obs )
```

```
Position       =  -0.1991363197099241 (0 = centre, >0 = means to the right)
Speed          =  -1.1891467948066015
Angle          =  0.2104728442263587
Rotation speed =  1.4678484017842104 (Positive means clockwise)
```

To a discrete version:

In [71]:
```
discrete_obs = discretize(obs)
showObs( discrete_obs )
```

```
Position       =  0 (0 = centre, >0 = means to the right)
Speed          =  0
Angle          =  4
Rotation speed =  11 (Positive means clockwise)
```

It is these values that will be used to find a location in the Q-table.

2.4 Selecting an Action based on the Q-Table

The next function selects an action in a particular state. Remember that the programme does not always select the 'best' option in the Q-Table! Remember in the lecture that Mario sometimes has to open a random door, even if he knows that one of the doors has a reward.

This is called the 'explore / exploit' trade-off. As well as exploting what we already know, we also sometimes need to take some risks and explore new options. This is the route of learning!

The amount of times we 'explore' rather than 'exploit' is defined by the 'epsilon' variable. So the following function will sometimes choose the best option from the Q-table, and sometimes select a random action, depending on the value of Epsilon.

The 'best' option is found using the 'np.argmax' function. 'np.argmax' find the position of the largest value. Thus:

In [72]:
```
np.argmax([1,12,1,60,2,4])
```

Out[72]:
```
3
```

In the largest value is at position '3' ... remember that in Python, arrays are numbered from 0!

Student task : Experiment by modifying the above function until you understand what it does.

So here is a function that selects next action:

In [73]:
```
# Choosing action based on epsilon-greedy policy
def choose_action(state, epsilon):
    return env.action_space.sample() if (np.random.random() <= epsilon) else np.argmax(Q[state])
```

2.5 Learning! .. Updating the Q-Table based on the reward

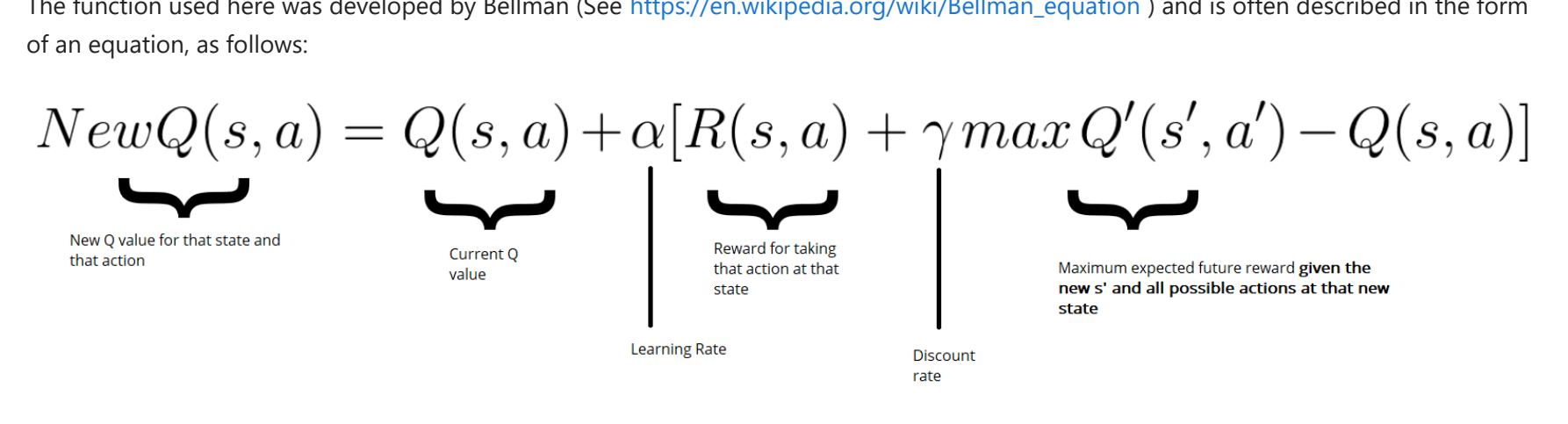
This next function is the heart of our learning algorithm. After we take an action we review the reward. This function needs to know the state that we were in **before** taking the action, the new state and the reward that was received.

In this function, 'alpha' is the learning rate. The value of alpha controls the influence that new information has on the Q-table.

'gamma' is the discount factor. The discount factor controls the impact of immediate reward compared with delayed reward.

The function used here was developed by Bellman (See https://en.wikipedia.org/wiki/Bellman_equation) and is often described in the form of an equation, as follows:

$$NewQ(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma\, max\, Q'(s',a') - Q(s,a)]$$



In [74]:
```
# Updating Q-value of state-action pair based on the update equation
def update_q(state_old, action, reward, state_new, alpha):
    Q[state_old][action] += alpha * (reward + gamma * np.max(Q[state_new]) - Q[state_old][action])
```

2.6 Adaptive rates for Learning and Exploration

As explained in the lecture, a common strategy in reinforcement Learning is to have the values of alpha and epsilon decay over a period of time. That means that the Actor adapts its learning behaviour the longer the game is played.

In [75]:
```
# Adaptive learning of Exploration Rate
def get_epsilon(t):
    return max(min_epsilon, min(1, 1.0 - math.log10((t + 1) / ada_divisor)))
```

In [76]:
```
# Adaptive learning of Learning Rate
def get_alpha(t):
    return max(min_alpha, min(1.0, 1.0 - math.log10((t + 1) / ada_divisor)))
```

2.8 The main learning loop

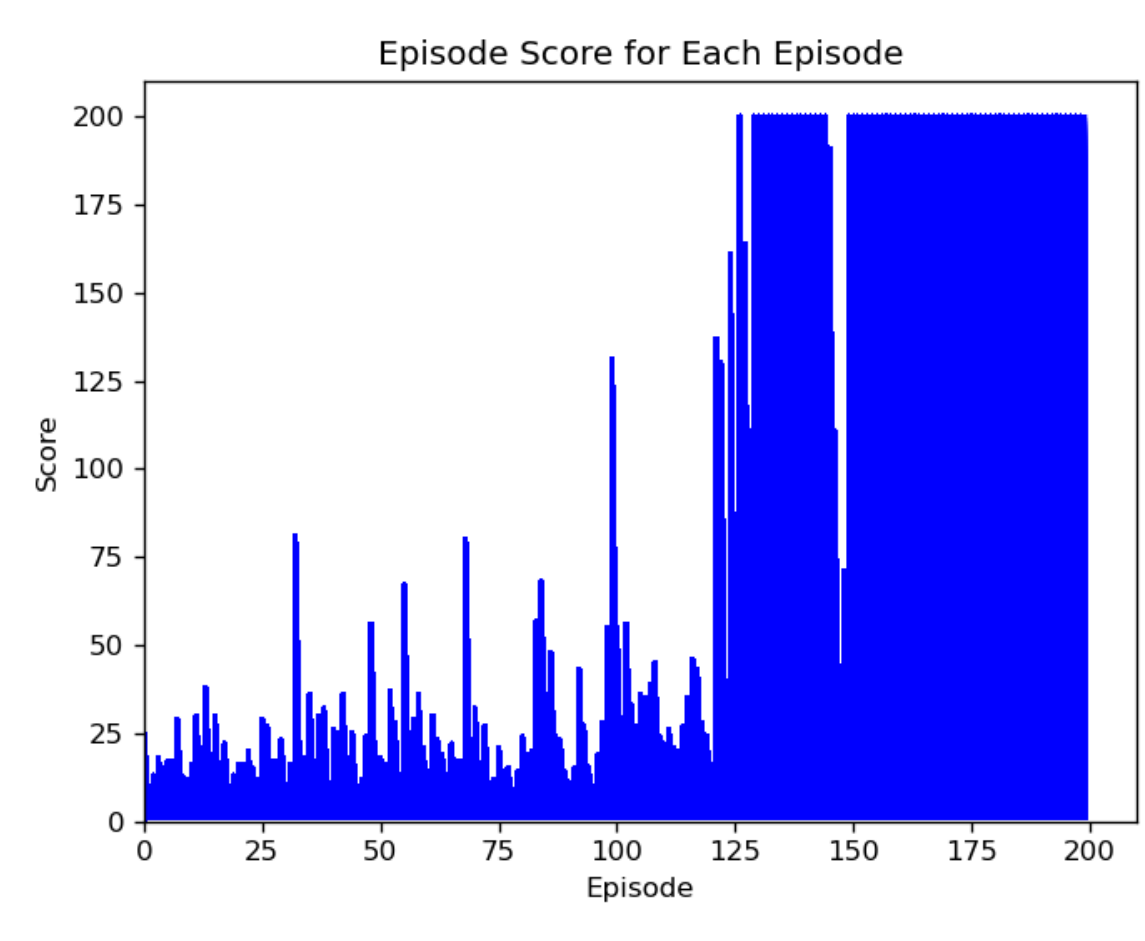
Having defined all of the above equations, the final learning loop is rather simple. The game is played many times - as defined by the variable 'n_episodes'. Each time a game is played the programme does the following:

1. The state is observed
2. The learning and discount rates are calculated
3. The programme tries to balance the pole

Balancing the pole means:

3.1 Selecting an action 3.2 Observing the new new state of the system 3.3 Obtaining the reward 3.4 Based on that response .. updating the Q-table 3.5 Continue doing this until either the pole falls over, or the game ends (200 steps)

In [39]:
```
n_episodes = 200                   # (copy to make class demo faster! <200 )
y_chart = []                      # Stores the y Axis data
x_chart = []                      # Stores the score for each run (x values)
plt.figure(100)                   # Gives the plot a unique number (name)
plt.xlim((0,210))                 # Max size of the x axis
plt.ylim((0,210))                 # Max size of the y axis
plt.title('Episode Score for Each Episode')      # Chart title
plt.ylabel('Score')               # Label for the y axis
plt.xlabel('Episode')             # Label for the x axis

for e in range(n_episodes):             # Run many episodes (defined by n_episodes)
    # As states are continuous, discretize them into buckets
    current_state = discretize(env.reset())

    # Get adaptive learning alpha and epsilon decayed over time
    alpha = get_alpha(e)
    epsilon = get_epsilon(e)

    done = False
    i = 0
    total_reward = 0

    while not done:
        # Render environment
        # env.render()        # Comment out to make the code run faster

        # Choose action according to greedy policy and take it
        action = choose_action(current_state, epsilon)
        obs, reward, done, _ = env.step(action)
        new_state = discretize(obs)

        # Update Q-Table
        update_q(current_state, action, reward, new_state, alpha)
        current_state = new_state
        i += 1
        total_reward += reward
        y_chart.append(total_reward)
        x_chart.append(e)
        plt.plot(x_chart,y_chart, color='blue')
        plt.show()
        plt.pause(0.01)
    env.close()
```



In [ ]: