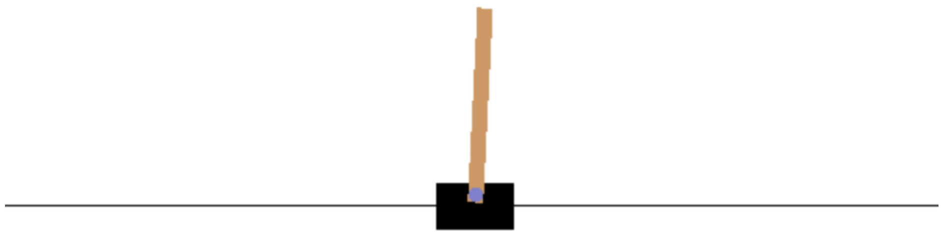


Using Deep Q-Learning to solve the Cartpole Problem

COMP 532 - ASSIGNMENT 2 ACCOMPANYING REPORT

Orion Assefaw, Student ID: 201530497, Email: o.assefaw@liverpool.ac.uk

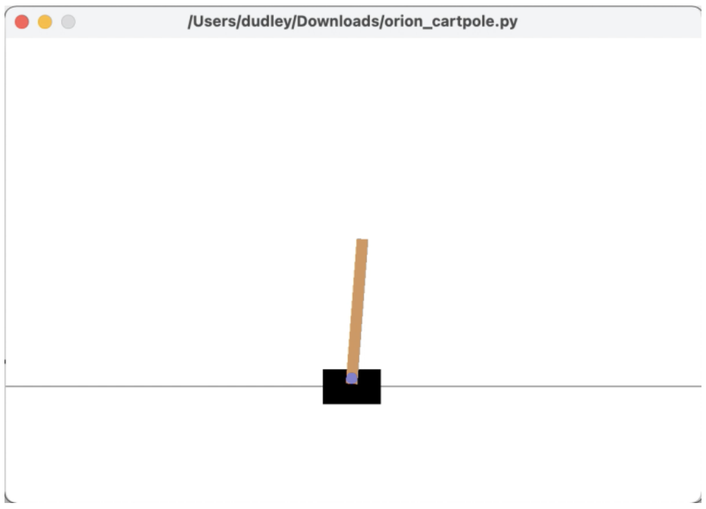
Introduction

In this assignment, a Deep Q-Network was built and trained to learn to balance a pole on a moving cart. This is widely known as the cart and pole problem.

To do so, OpenAI's Gym toolkit was used to set up the cart and pole environment. The python version employed to achieve this was version 3.8.

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

The first step was to import and load the game as seen below.



The Deep Reinforcement Learning Model:

As mentioned above, a deep Q-Learning model is used.

Network Architecture:

The neural network that was used has one input layer, two 'hidden' layers and an output layer. The model provides inputs from the observation space to the input layer, these are 'fed forward' to subsequent layers until the output layer, where the values in the output layer are used to select the action.

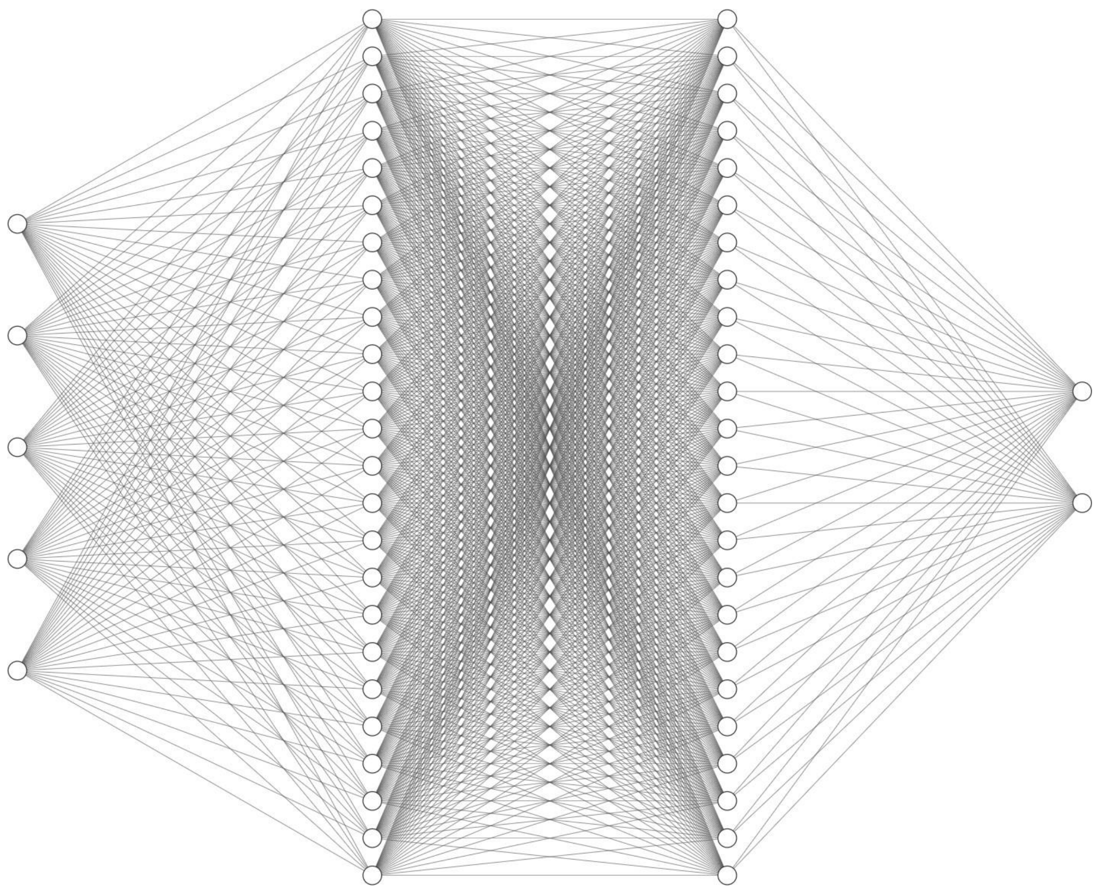
In the CartPole problem, there are 5 inputs (numbers that represent the state of the environment at that time, including the bias):

- Angle of the pole
- Velocity of the cart
- Rotation rate of the pole
- Position of the cart
- The bias

there are 2 outputs:

- Move left
- Move right

The network (shown below) has an input layer (with 5 inputs), 2 hidden layers (each with 24 neurons), and an output layer (with 2 output neurons).



The Parameters:

- The learning rate α used was 0.001
- Discount factor γ of 0.95 was used
- Initial exploration rate ϵ of 1
    - However, the epsilon rate decayed over each timestep. This meant that as agent learns it starts to explore less and exploit more.
    - Initially the program will wait until enough experience has been gained (lots of exploration has been completed) before learning begins.

```
self.epsilon = self.epsilon if self.epsilon < 0.01 else self.epsilon*0.995
```

- r_{t+1} is the reward received when moving from state s_t to s_{t+1} for the cartpole this is +1 for every timestep where the pole remains upright.
- Number of episodes is 5000
- Max number of timesteps is 500

Forward Pass:

This is the part of the code that associates the observations, actions and rewards of the game to the network's input and output.

```
def forward(self, observation, remember_for_backprop=True):
        vals = np.copy(observation)
        index = 0
        for layer in self.layers:
            vals = layer.forward(vals, remember_for_backprop)
            index = index + 1
        return vals
```

- The RLAgent.forward function passes the input (the observation) to the network **layer.forward(vals, remember_for_backprop).** This collects the output of each layer and passes it to the next layer.

Loss Function:

- Reinforcement Learning theory helps us define the loss function as the squared difference between experimental and action values.
- $l_i(W_i) = (E_{s'}[r + \gamma \max_{a'} Q(s', a'; W_{i-1})] - Q(s, a; W_i))^2$

  $l_i(W_i) = $ (**experimental_values** - **action_values**)^2

$Q(s, a; W_i)$ - This is the agents current estimate of the value of each action in a given state, **action_value,** calculated using the variable **prev_obs**. This is used when selecting an action in our **select_action** function.

$\gamma \max_a Q(s', a', W_{i-1})$ - This is a prediction of the values of both actions from the next state **next_action_values**, calculated using the **new_obs** variable. This will change each time an action is taken and so is temporary.

$E_{s'}[r + \gamma \max_{a'} Q(s', a'; W_{i-1})]$ - This is the target value that the agent has learned from experience, **experimental_values,** calculated using the **next_action_values.**

- The difference between **experimental_values** and **action_values** will form the basis for updating the weights of our network, implemented in the backward function (which will be discussed below).

Activation Function - Rectified Linear Unit (ReLU)

```
def relu(mat):
    return np.multiply(mat,(mat>0))
```

- The **relu** function introduces non-linearity into the network. Negative inputs become 0 and positive inputs retain their value.

Backpropagation:

- Gradient update (with respect to Q-function parameters W_i)

Now that we have the loss function (shown above), calculated as the squared difference between experimental and action values, the optimization problem is the same as minimizing the error function in any other neural network, through backpropagation.

```
def backward(self, calculated_values, experimental_values):
  delta = (calculated_values — experimental_values)
  for layer in reversed(self.layers):
  delta = layer.backward(delta)
```

This function first calculates the difference between the calculated values (the predicted action_values) and the experimental value of taking an action in a state. This error is then 'propagated' backwards through each layer going from the last hidden layer to the input layer, and the weights are updated based on this error. Intuitively, each layer is told how much its estimate of its output differs from its expected output needed to generate the experimental values in the output layer.

The above function then calls the backward function of the NNlayer class as follows:

```
def backward(self, gradient_from_above):
        adjusted_mul = gradient_from_above
        if self.activation_function != None:
            adjusted_mul  =  np.multiply(relu_derivative(self.backward_store_out),gradient_from_above)
        D_i = np.dot(np.transpose(np.reshape(self.backward_store_in,(1, len(self.backward_store_in)))),
np.reshape(adjusted_mul, (1,len(adjusted_mul))))
        delta_i = np.dot(adjusted_mul, np.transpose(self.weights))[:-1]
        self.update_weights(D_i)
        return delta_i
```

Now, all that this function does is calculate and update:

- The actual derivative of the loss function with respect to weights at this layer.
- The calculated 'error' for the inputs to this layer (i.e., the output from the previous layer). This is then passed backward to the previous layer to enable its own derivative and delta calculations.

```
def relu_derivative(mat):
    return (mat>0)*1
```

- The **relu_derivative** function is used in the backpropagation step, for minimizing the errors (Loss Function).

Evaluation and Results

After just 230 episodes, it can be seen that significant progress has been madeand that the timesteps before failure are getting increasingly larger.

And once it reaches around 4500 episodes, the agent will actually stay upright until the maximum number of timesteps (500).

Initially, the learning started with a 300 timestep maximum and changed to a maximum of 500 timesteps. This slowed down the learning process at later episodes however also allowed for a less restricted demonstration of its performance.

References

- COMP532 Lecture 17
- Creating Deep Neural Networks, an Introduction to Reinforcement Learning