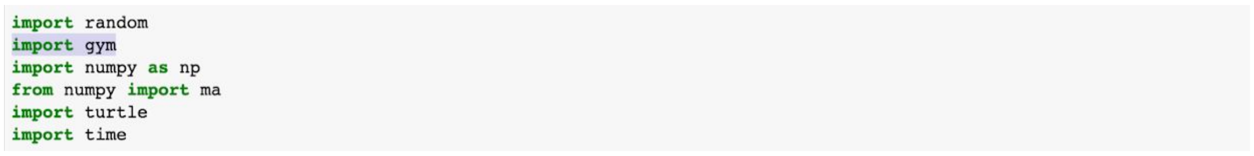
Harry Williams, 201374501 James McCarron, 201380393

COMP532: Assignment 2
Deep Q Network
J.McCarron, H.Williams

This assignment is concerned with creating a Deep Neural Network capable of learning and mastering a video game from the OpenAI gym; a toolkit that allows for easy implementation of RL algorithms.

SECTION 1: Importing OpenAI Gym

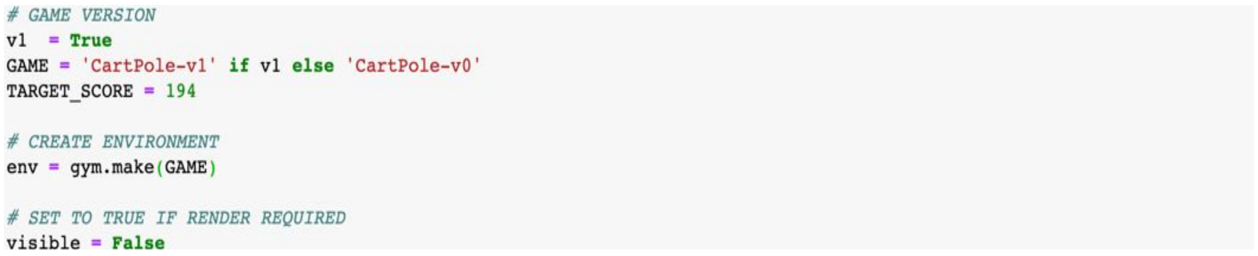
The OpenAI Gym library is imported and the game environment is selected, created and initialised. All these processes are shown in the figures 1.1, 1.2 and 1.3 below.



```
import random
import gym
import numpy as np
from numpy import ma
import turtle
import time
```

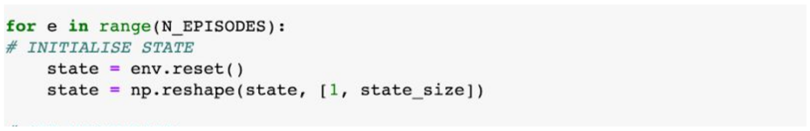
Figure 1.1 *Importing the OpenAI gym library.*

There are two possible environments that our agent can play. `Cartpole-v0` and `Cartpole-v1`. The only difference between these two environments is the maximum episode length. `Cartpole-v0` has a maximum episode length of 200 frames, whereas `CartPole-v1` has a maximum episode length of 500 frames.



```
# GAME VERSION
v1  = True
GAME = 'CartPole-v1' if v1 else 'CartPole-v0'
TARGET_SCORE = 194

# CREATE ENVIRONMENT
env = gym.make(GAME)

# SET TO TRUE IF RENDER REQUIRED
visible = False
```

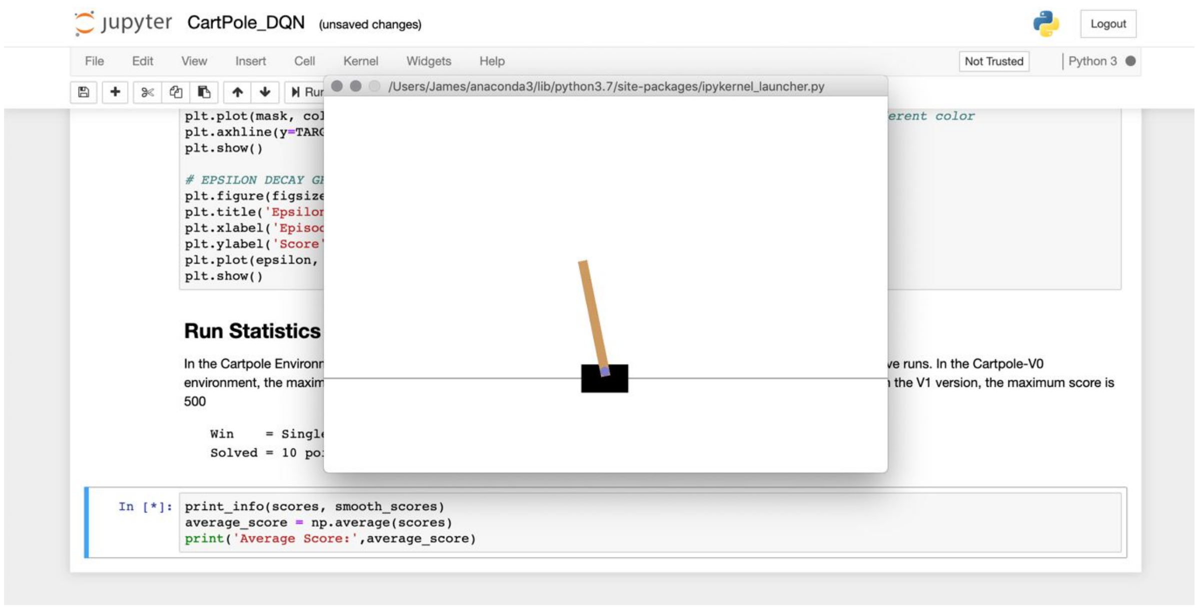
Figure 1.2 *Selecting & creating the game environment, CartPole v1 or v0.*



```
for e in range(N_EPISODES):
# INITIALISE STATE
    state = env.reset()
    state = np.reshape(state, [1, state_size])
```

Figure 1.3 *Initialising the game environment/state with env.reset().*

When `visible = True` in figure 1.2 the game will be rendered. See figure 1.4 below for snapshot of loaded game running as it trains.


jupyter CartPole_DQN (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
Python 3
/Users/James/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py

```
plt.plot(mask, col
plt.axhline(y=TARG
plt.show()

# EPSILON DECAY GR
plt.figure(figsize
plt.title('Epsilon
plt.xlabel('Episod
plt.ylabel('Score'
plt.plot(epsilon,
plt.show()
```

Run Statistics

In the Cartpole Environm ... ve runs. In the Cartpole-V0 environment, the maxim ... the V1 version, the maximum score is 500

```
Win    = Single
Solved = 10 po
```

```
In [*]: print_info(scores, smooth_scores)
        average_score = np.average(scores)
        print('Average Score:',average_score)
```

Figure 1.4 *Snapshot of game loaded and training,*

SECTION 2: Creating the Network

The Keras library was used to build the deep neural network. Keras components were imported as shown in figure 2.1 `from keras.models import Sequential` etc. A sequential model is used to build the neural network that will estimate the Q*(s,a). The network will consist of only `Dense` layers and the `Adam` optimizer will be used for stochastic gradient descent.

```
import random
import gym
import numpy as np
from numpy import ma
import turtle
import time

from collections import deque
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
```

Figure 2.1 *Importing Keras libraries.*

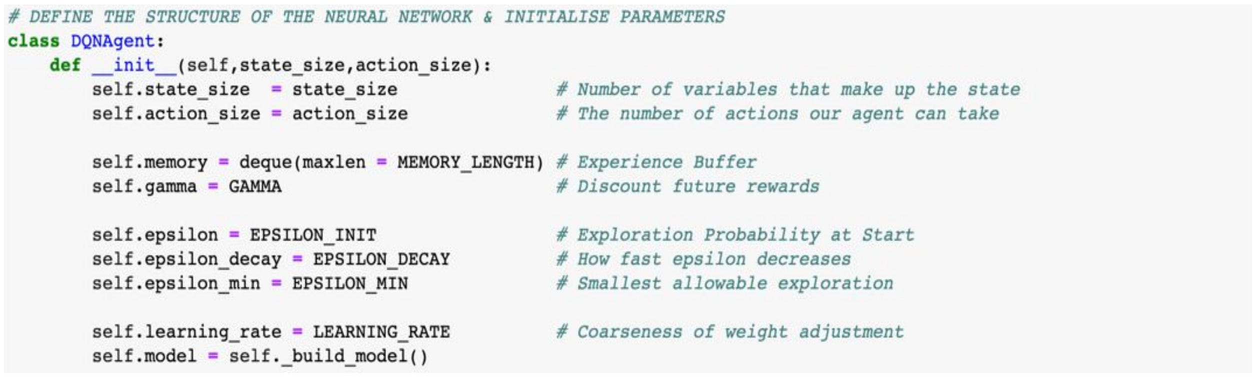
The network parameters are initialised in figure 2.2. The `self.state_size` parameter is the number of variables that make up the state which in the case of CartPole are:

Cart Position
Cart Velocity
Pole Angle Pole
Angular Velocity

`self.action_size` is the number of possible actions the agent can take, 2 in the case of CartPole, being right and left.

`self.gamma` is the discount factor for future rewards, used in calculating Q*.

`self.epsilon` is initialized here along with the decay rate `self.epsilon_decay` and the epsilon minimum `self.epsilon_min`, which will stop exploration when reached. This function is shown in figure 2.6.

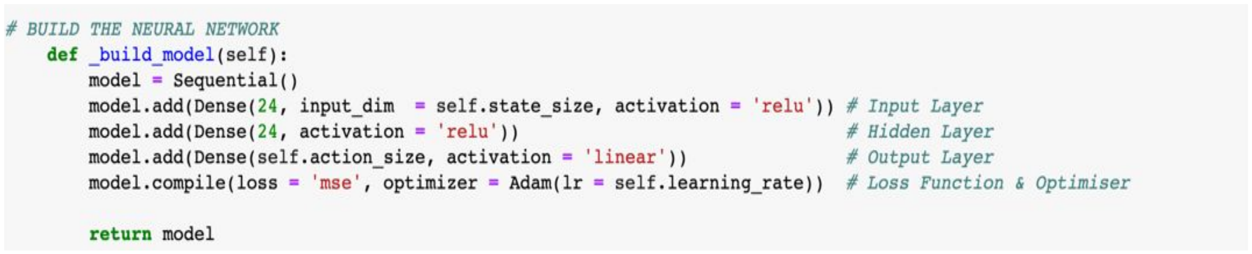


```python
# DEFINE THE STRUCTURE OF THE NEURAL NETWORK & INITIALISE PARAMETERS
class DQNAgent:
    def __init__(self,state_size,action_size):
        self.state_size  = state_size              # Number of variables that make up the state
        self.action_size = action_size             # The number of actions our agent can take

        self.memory = deque(maxlen = MEMORY_LENGTH) # Experience Buffer
        self.gamma = GAMMA                         # Discount future rewards

        self.epsilon = EPSILON_INIT                # Exploration Probability at Start
        self.epsilon_decay = EPSILON_DECAY         # How fast epsilon decreases
        self.epsilon_min = EPSILON_MIN             # Smallest allowable exploration

        self.learning_rate = LEARNING_RATE         # Coarseness of weight adjustment
        self.model = self._build_model()
```

Figure 2.2 Structure of the neural network.

Figure 2.3 below is the function that builds the neural network. The model is defined as sequential meaning that the network is built layer by layer. All layers in the network are 'Dense'. The input layer has `self.state_size` inputs, which in the case of CartPole is 4 as explained earlier. A Rectified Linear Unit (ReLU) activation function is used and it has 24 neurons.

The hidden layer of the network also has 24 neurons with a ReLU activation function. The ReLU function is shown in figure 2.4. With the ReLU function, if the input is less or equal to zero, there will be no activation. The output layer has as many outputs as there are actions (`self.action_size`) and uses a linear activation function. A linear activation is used to ensure that the outputs are not abstract. Mean Squared Error is used as the loss function and the Adam optimizer is used, using an earlier defined learning rate.



```python
# BUILD THE NEURAL NETWORK
    def _build_model(self):
        model = Sequential()
        model.add(Dense(24, input_dim  = self.state_size, activation = 'relu')) # Input Layer
        model.add(Dense(24, activation = 'relu'))                               # Hidden Layer
        model.add(Dense(self.action_size, activation = 'linear'))               # Output Layer
        model.compile(loss = 'mse', optimizer = Adam(lr = self.learning_rate))  # Loss Function & Optimiser

        return model
```

Figure 2.3 Building the network using Keras library.

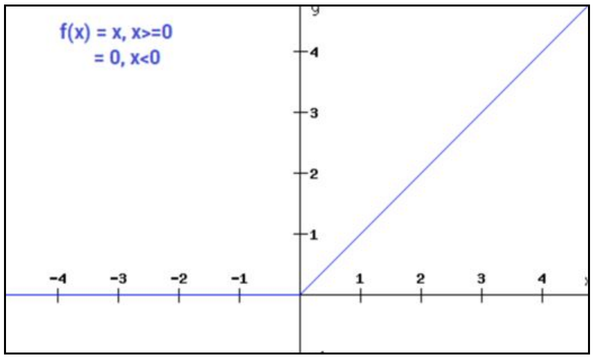
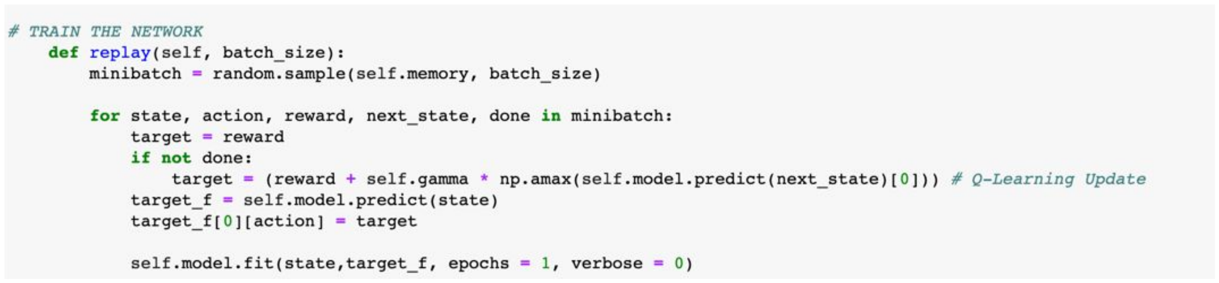


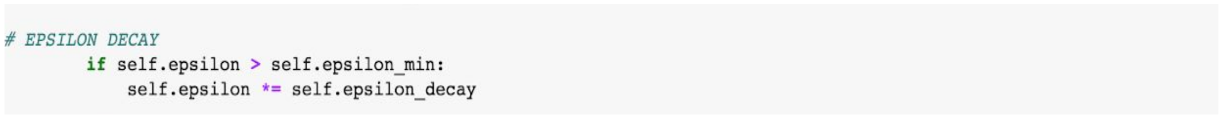
Figure 2.4 ReLU activation function.

Figure 2.5 demonstrates the function that trains the neural network. Markov Decision Process is used. If the episode is finished the target reward equals to the current reward. If the episode is running, the future rewards must be predicted using the current reward and the discounted future reward from the *next_state* using the neural network. The model is then fit to train `self.model.fit()`, using the current state and the future reward. For more explanation see section 4.



```
# TRAIN THE NETWORK
    def replay(self, batch_size):
        minibatch = random.sample(self.memory, batch_size)

        for state, action, reward, next_state, done in minibatch:
            target = reward
            if not done:
                target = (reward + self.gamma * np.amax(self.model.predict(next_state)[0])) # Q-Learning Update
            target_f = self.model.predict(state)
            target_f[0][action] = target

            self.model.fit(state,target_f, epochs = 1, verbose = 0)
```

Figure 2.5 Function to train the network using reinforced learning by prediction of the rewards of future states.



```
# EPSILON DECAY
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay
```

Figure 2.6 Epsilon decay if epsilon is above minimum

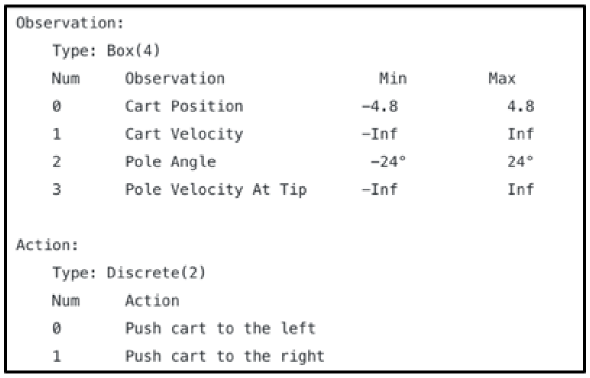
SECTION 3: Network I/O

For each frame of the game, we record the observation from the environment in a tuple with the structure. The observation, as shown in figure 2.3, is mapped to the neural network as input dimensions `input_dim = self.state_size` in the first neural network layer.

Random integer is chosen using `act()`, if the integer is <= `self.epsilon` do a random action, if not choose the best action according to the neural network as shown in figures 3.2 and 3.3. Actions and observations are shown in figure 3.1. As shown in figure 2.3, the action is mapped to the output from the network via the `self.action.size` parameter.

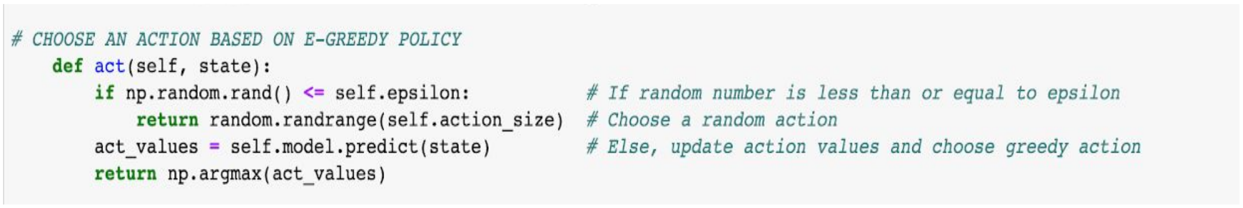
Our agent receives 1 point of reward for every frame of the game. The game is ended in one of three ways.

- Pole angle goes ± 15° from vertical
- The cart goes ± 2.4 from the center
- The maximum episode length is reached



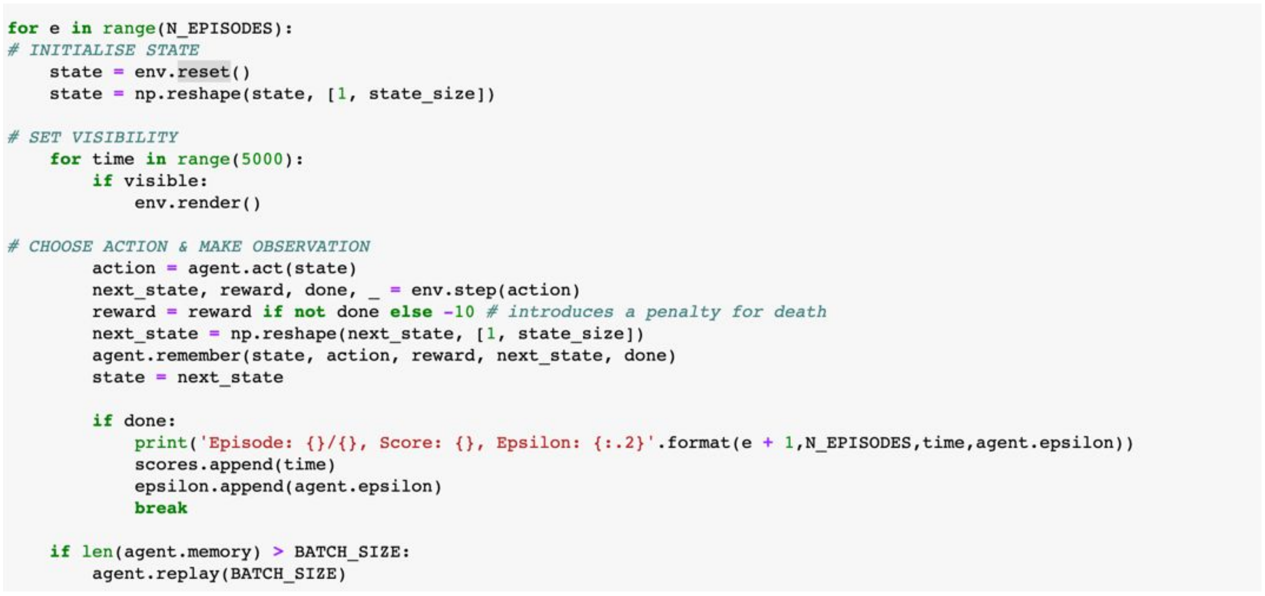
```
Observation:
    Type: Box(4)
    Num     Observation               Min         Max
    0       Cart Position             -4.8          4.8
    1       Cart Velocity             -Inf          Inf
    2       Pole Angle                 -24°         24°
    3       Pole Velocity At Tip      -Inf          Inf

Action:
    Type: Discrete(2)
    Num     Action
    0       Push cart to the left
    1       Push cart to the right
```

Figure 3.1 Observations and Actions



```
# CHOOSE AN ACTION BASED ON E-GREEDY POLICY
    def act(self, state):
        if np.random.rand() <= self.epsilon:          # If random number is less than or equal to epsilon
            return random.randrange(self.action_size)  # Choose a random action
        act_values = self.model.predict(state)         # Else, update action values and choose greedy action
        return np.argmax(act_values)
```

Figure 3.2 Function to choose action.



```
for e in range(N_EPISODES):
# INITIALISE STATE
    state = env.reset()
    state = np.reshape(state, [1, state_size])

# SET VISIBILITY
    for time in range(5000):
        if visible:
            env.render()

# CHOOSE ACTION & MAKE OBSERVATION
        action = agent.act(state)
        next_state, reward, done, _ = env.step(action)
        reward = reward if not done else -10 # introduces a penalty for death
        next_state = np.reshape(next_state, [1, state_size])
        agent.remember(state, action, reward, next_state, done)
        state = next_state

        if done:
            print('Episode: {}/{}, Score: {}, Epsilon: {:.2}'.format(e + 1,N_EPISODES,time,agent.epsilon))
            scores.append(time)
            epsilon.append(agent.epsilon)
            break

    if len(agent.memory) > BATCH_SIZE:
        agent.replay(BATCH_SIZE)
```

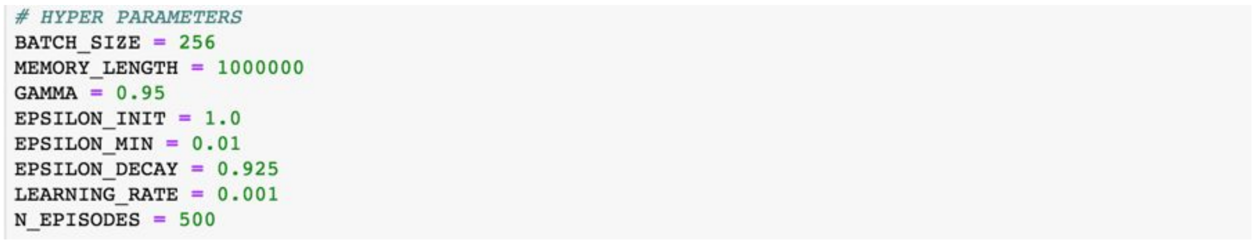
Figure 3.3 Function to train the network using reinforced learning by prediction of the rewards of future states.

SECTION 4: Deep RL Model

We chose to implement a Deep Q Network (DQN) to train our agent. A DQN combines the tabular learning method of Q-Learning with the additional computational power of a neural network. In Cartpole, four different variables make up the state and there are two possible actions that our agent can take: move left, or move right.

However, we do not update the Q values for every single state. In order to increase the stability of our network, we are using implementing the Q learning algorithm on a randomly sampled mini-batch of experiences. This is to increase the diversity of experiences the agent learns from and reduces the risk of the agent plateauing on suboptimal strategies or exhibiting forgetful behaviour.

Our network hyper parameters have a number of notable points. The first of which is the high `BATCH_SIZE` and `MEMORY_LENGTH` variables. We found that this helped the network learn quickly, but then stabilise over tests with higher episodes. We also implemented a decaying epsilon value which dramatically improved the agent's learning ability.



```
# HYPER PARAMETERS
BATCH_SIZE = 256
MEMORY_LENGTH = 1000000
GAMMA = 0.95
EPSILON_INIT = 1.0
EPSILON_MIN = 0.01
EPSILON_DECAY = 0.925
LEARNING_RATE = 0.001
N_EPISODES = 500
```

Figure 4.1 Model Hyperparameters

This was achieved by defining an a starting epsilon value, a minimum epsilon value, and a decay rate. For each episode, the epsilon value was reduced by the decay rate until it reached the minimum. This allows for lots of initial exploration at the beginning of the run of tests, but allows the agent to pursue a greedy policy later over the course of the training period.

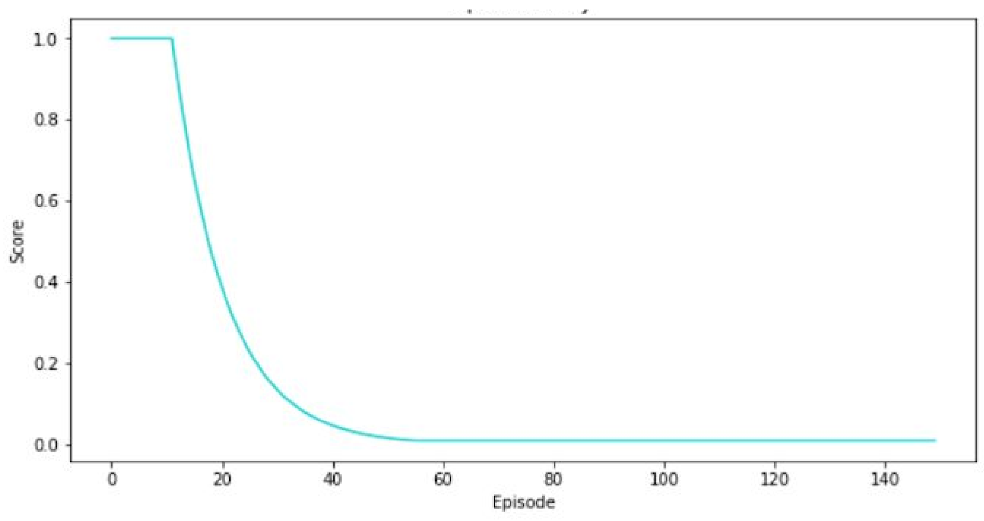


Figure 4.2 Decaying Epsilon

SECTION 5: Experiment Results

This section will outline the results of the scores obtained by the agent. The supplementary materials provided in the submission folder includes an edited highlight video that shows our agent in the process of learning, and subsequently completing the game repeatedly and with ease.

After several days of hyperparameter adjustment, we eventually resolved the dilemma of speed against stability and had an agent that consistently managed to solve the Cartpole environment.

Below are a number of matplotlib graphs that detail our agent's performance. The graphs appear in pairs, one showing the raw score data, and the other showing a 10 point moving average. The reason for twofold; firstly it gives a better visual representation of the agents learning process, and secondly, is that the cartpole documentation describes 'solving' the environment as getting an average score of 195+ over 10 consecutive timesteps.

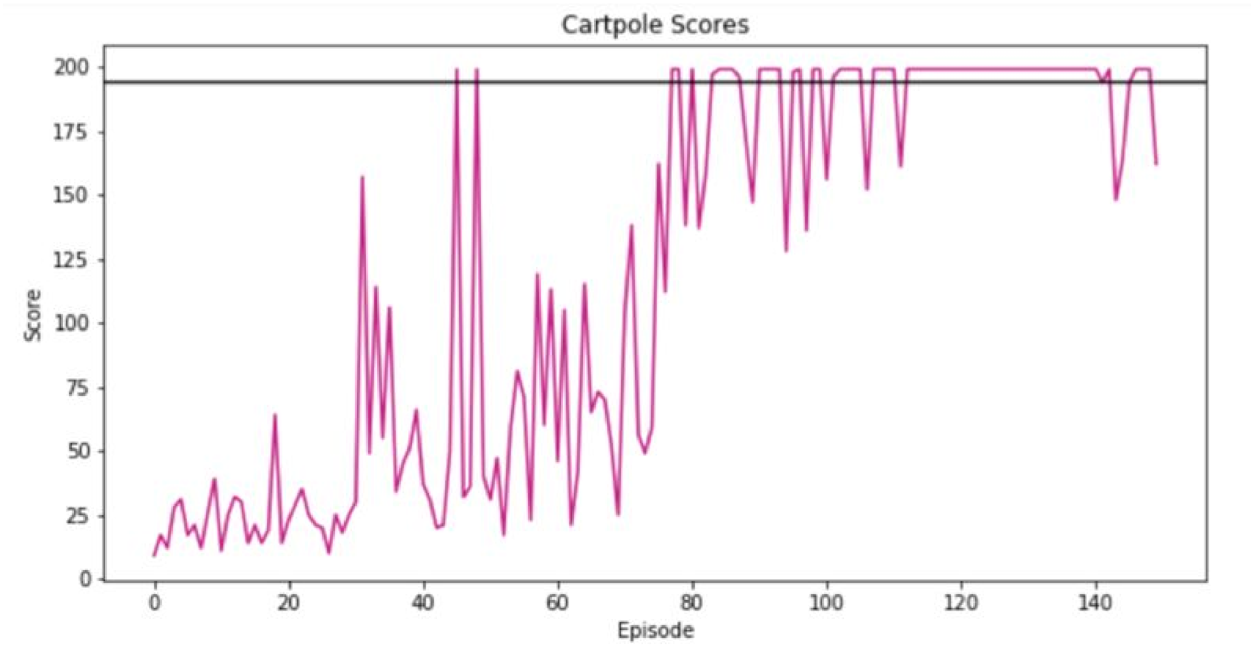


Figure 5.1 Agent's Scores over 150 Episodes

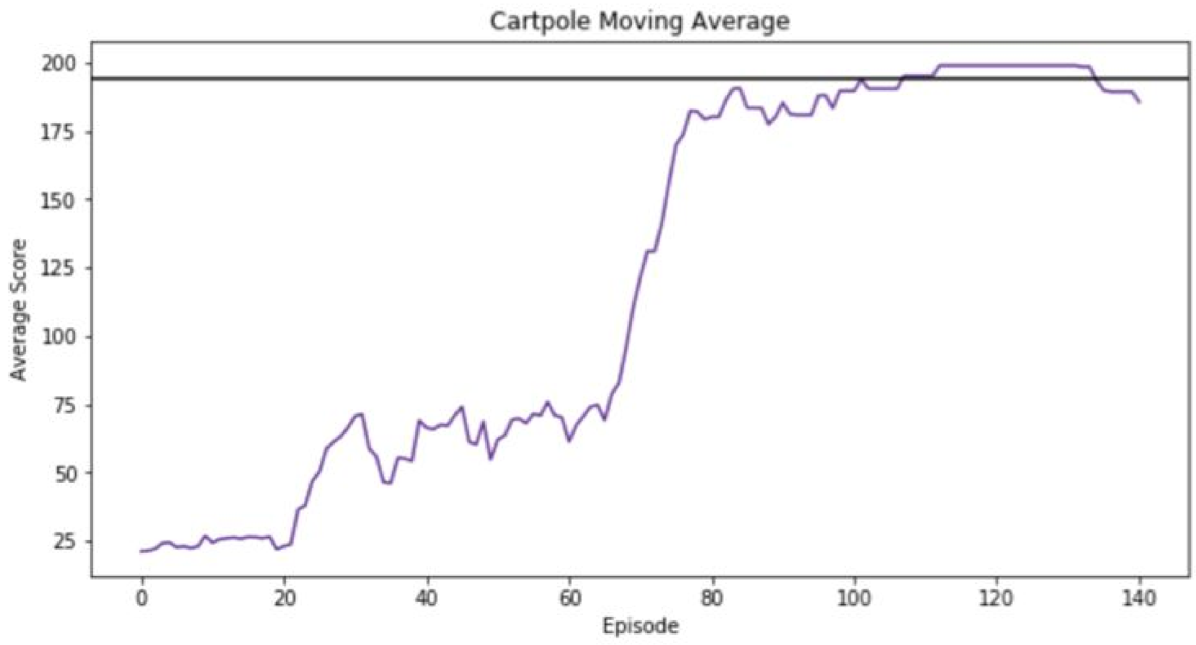


Figure 5.2 10 Point moving average over 150 runs

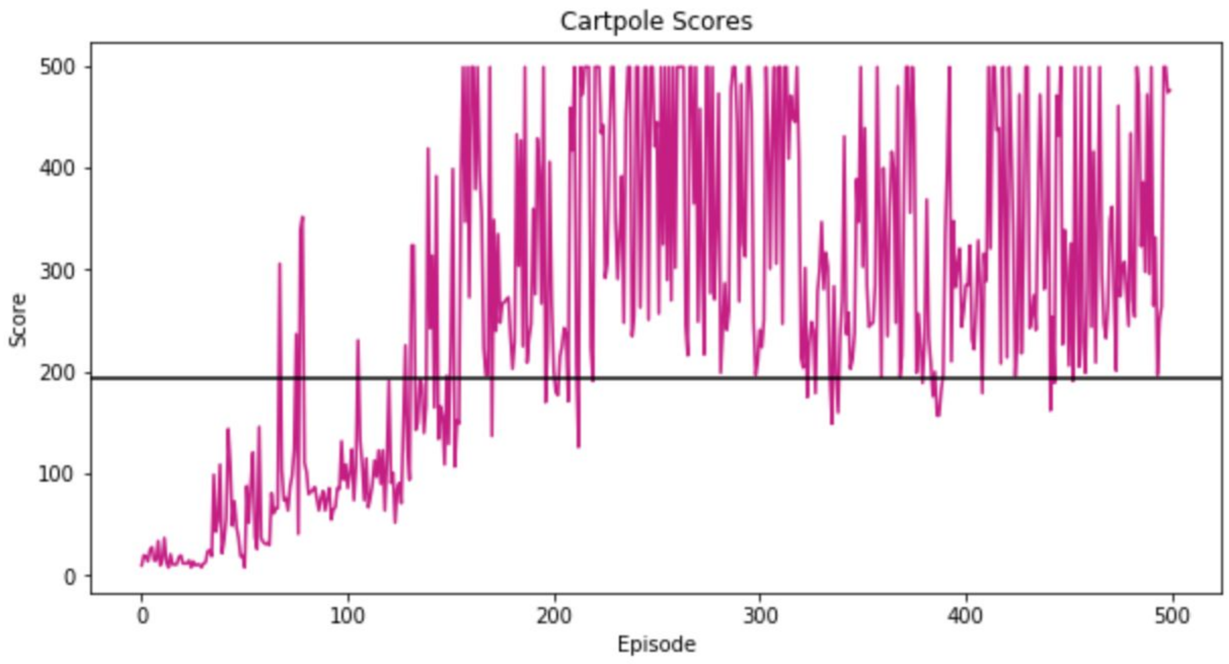


Figure 5.4 *Agent's score in v1 Environment over 500 Episodes*

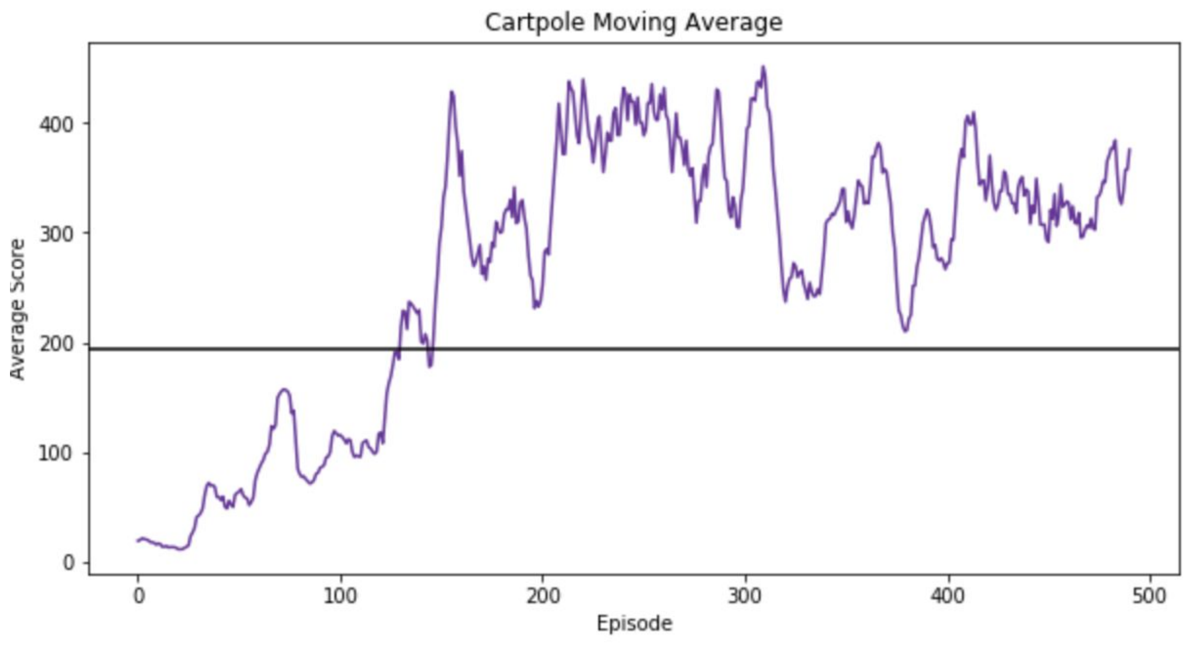


Figure 5.4 *Agent's Average Score in v1 Environment over 500 Episodes*

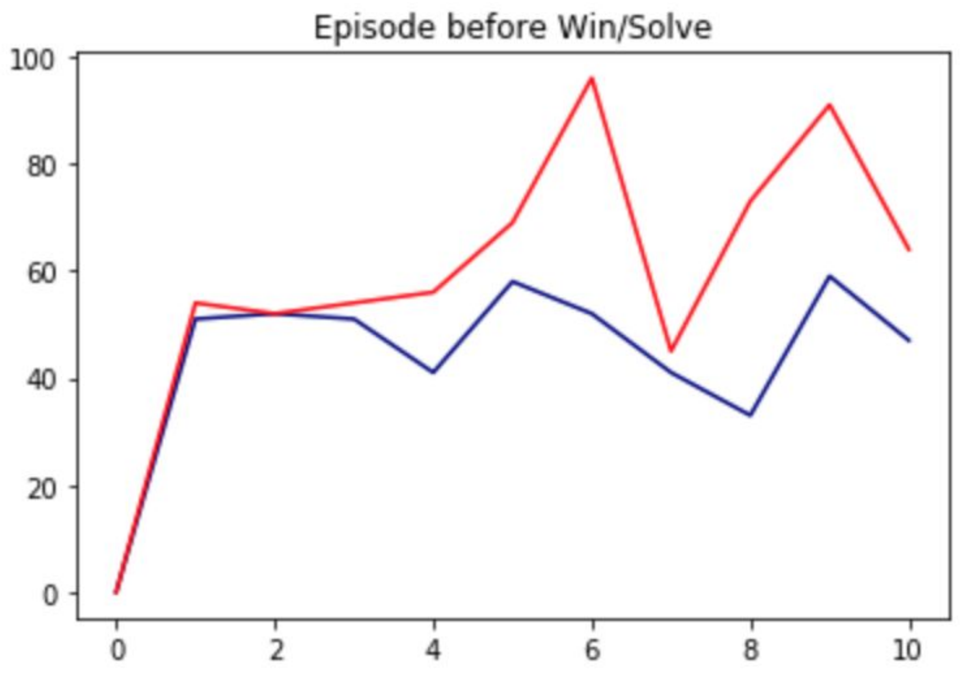


Figure 5.3 Blue = Won, Red = Solved

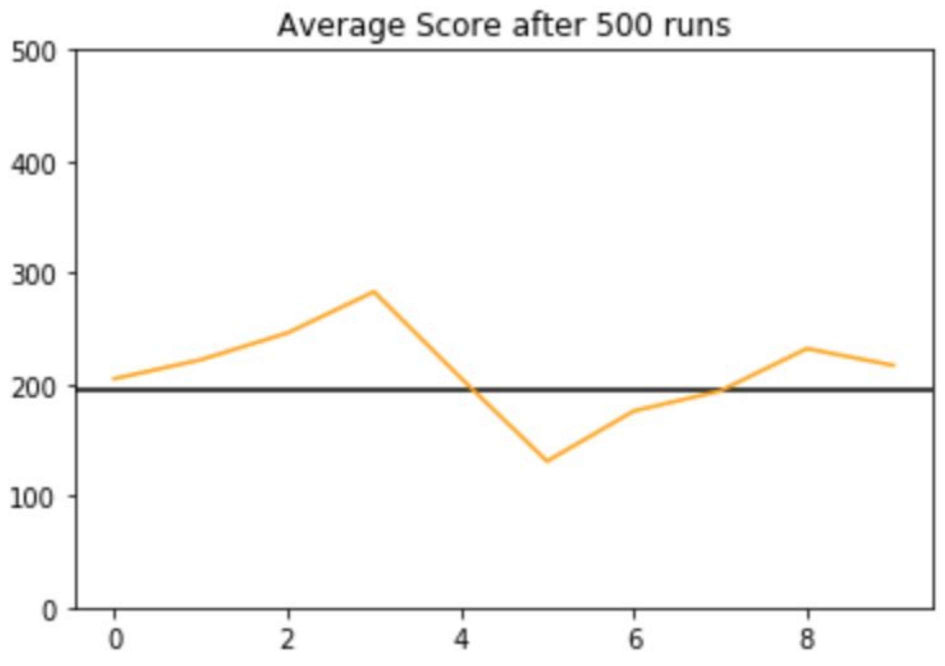


Figure 5.3 Average score after 500 runs

One aspect that requires further investigation, is the occasional catastrophic forgetting that our agent seems to exhibit. This manifested itself as a significant crash knee, where our agent would suddenly drop from a relatively stable optimal performance to a incredibly low plateau of between 7 and 9 points for several hundred runs. This behaviour is very rare, and we were not able to identify what was causing it; however, it did not occur in the Cartpole-v0 environment.

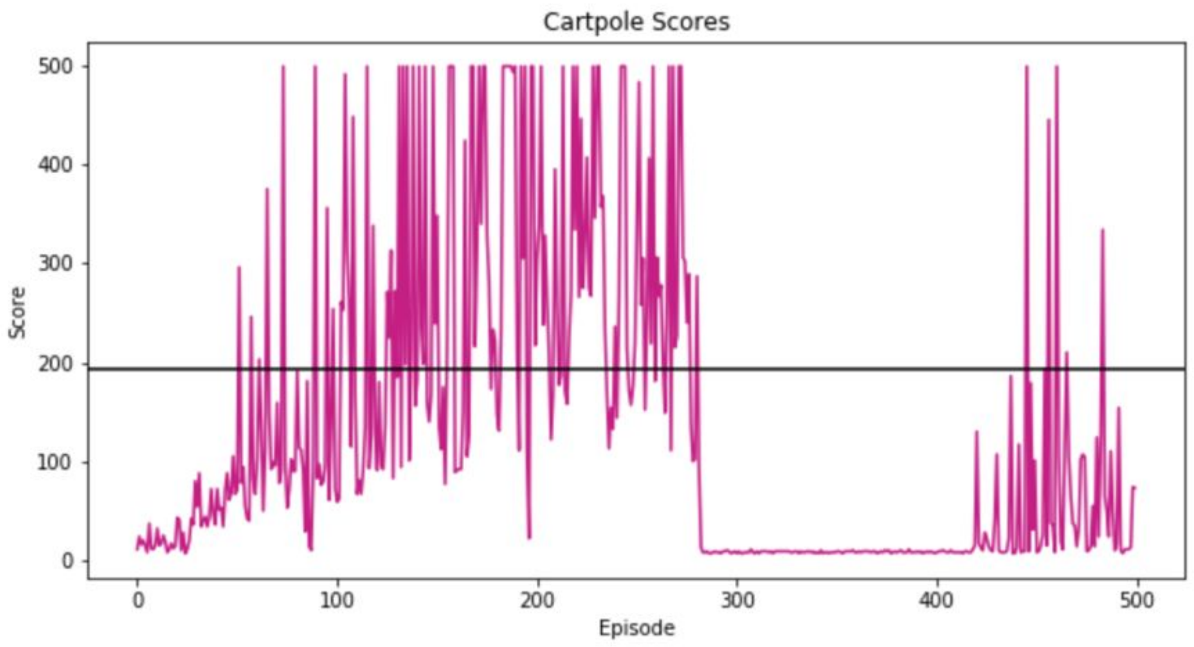


Figure 5.5 Catastrophic Forgetting Behaviour in v1