CS4287 Assignment 2

Sean Fitzgibbon - 19273444

Daniel Larkin - 19257503

Adam Butler - 19244967

Link to Google Colab (Jupyter Notebook):
https://colab.research.google.com/drive/1kqG39QawMjK3g4N8QZ9aimpXGXRll3hz?usp=sharing

Table of Contents

Why Reinforcement Learning?

The Cartpole environment is a part of OpenAI gym, which is a library of environments/problems that can be used to apply Reinforcement Learning Algorithms to them in order to test their efficiency/performance.

In the Cartpole environment, the bottom of a pole is connected to a cart, and the cart is free to move either left or right in order to balance the pole. If the pole falls below a certain angle, the game is lost.

We use Reinforcement Learning in this task because it involves a state which must assess rewards/penalties for each move, and must make a sequence of good decisions to maximize its reward. It isn't good enough to solve a single step of the process, it must take into account the future reward and optimize its choices based on this.

In Reinforcement Learning (RL), the agent will make observations and select actions based on the observations. These actions will ultimately lead to a maximum future reward, as prioritized by the agent. The reason for choosing RL over the likes of Supervised Learning (SL) is because in SL, the training data has the answer key attached to it, so the model would be trained with an 'answer sheet' alongside it, but in RL, the agent must decide and learn what the best action is by itself, with only its reward to base its decisions off. Essentially, RL learns by interacting with its environment, whereas SL bases its performance/learning off of pre-set data labels.

We want to maximize performance in the Cartpole environment, and we don't have labeled data to utilize SL, so we opt for RL in order to train an agent to interact with the environment, maximize its reward, and avoid failing the game to the best of its ability. For these reasons, Reinforcement Learning is the machine learning paradigm of choice for this task.

The Environment

The Cartpole environment is a Classic Control Environment from the OpenAI Gym library. In this environment, a pole is attached to a cart, and this cart can move either left or right. The goal in this environment is to balance the pole on the cart by moving it left or right, keeping the pole upright.

If the pole falls (reaches an angle 12° from the vertical), the episode will terminate. The episode will also terminate if the cart position moves more than 2.4 units from the centre of the environment (i.e. the edge of display).

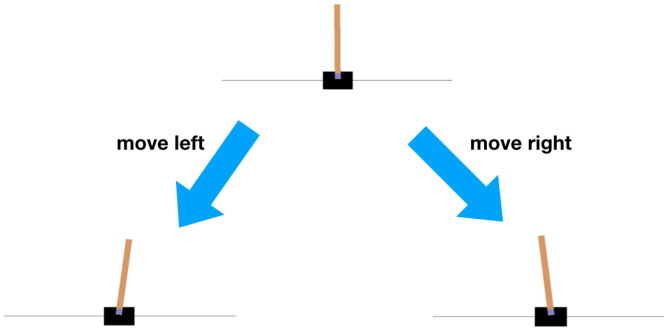

Figure 1: Cartpole environment with agent actions (Phy, 2019)

The environment is a 4-dimensional observation space, which includes the following:

- Cart velocity
- Cart position
- Pole angle
- Pole angular velocity

Also included in the environment is a 2-dimensional action space, allowing the agent to move the cart either left or right.

When interacting with the environment, the agent will receive a +1 reward for keeping the pole balanced for a step. It will receive a reward of 0 for an unsuccessful attempt. An episode terminates when the pole reaches an angle 12° from the vertical, or the cart moves more than 2.4 units from the centre of the environment.

The task is considered solved when the average reward is greater than/equal to 195.0 over 100 consecutive trials.

Implementation

Capture and Pre-processing of Data

Our Deep Q Reinforcement Learning network uses data from the simulated environment in which a pole is balanced on a cart. The data includes information on the current state of the cart, such as its position and velocity, as well as the current force applied to the cart. This data is used to train the network and enable it to learn to balance the pole in the simulated environment.

The network pre-processes data before using it for training. This preprocessing step involves cleaning the data, verifying it for accuracy, formatting it for consistency, and converting it into a suitable format for further analysis.

Our network cleans the data by removing any outliers or erroneous data points that may exist in the dataset. It also ensures that the data is formatted in a consistent fashion and is free of any errors or inconsistencies. Finally, it converts the data into a suitable format for further analysis.

The network verifies the accuracy of the data by performing a range of different tests. These tests include checking for outliers, verifying the data against known standards or ranges, and checking for any inconsistencies or errors in the data. Any data points that are found to be outside of the established ranges or have errors are flagged and further investigated to ensure accuracy. The network also performs various other tests, such as running statistical tests on the data, to ensure that it is accurate and reliable.

Network Structure

Importing Libraries:

Keras-rl2 is integrated with Open AI Gym to assess and experiment with the Deep Q-Network (DQN) algorithm. **Matplotlib** is used for showing images and creating visualizations for the model's outcomes. **Gym**, a Open AI software, is applied to create the Cart Pole environment to produce and assess Reinforcement learning strategies. **Keras**, a high-level API, is employed with TensorFlow to construct and train advanced learning models.

```
import gym
import matplotlib.pyplot as plt
from keras import Sequential
from keras.layers import Input, Flatten, Dense
```

Figure 2: Importing libraries

Setting up the Cart Pole Environment with Open AI Gym:

The objective of the Cart Pole is to balance a pole attached to a cart, which can be moved in either direction (left or right) by a series of 0 or 1 actions.

The agent is provided with an array of four floats representing the angular position and velocity of the cart and pole.

The agent is rewarded with a scalar float value for their actions, with 0 corresponding to a "move left" action and 1 corresponding to a "move right" action.

```
#Load the CartPole environment from the OpenAI Gym suite
ENV_NAME = 'CartPole-v0'
env = gym.make(ENV_NAME)
```

Figure 3: Setting up the environment

The Neural Network

- The Neural Network model consists of an input layer, two hidden layers, and an output layer.
- The input layer is 1 observation vector, and the number of observations in that vector.

- We also add a flatten layer to the neural network
- The two hidden layers have 24 neurons each, and use a rectified linear activation function (ReLU). This activation function will convert any negative values to zero, and will maintain any positive values as they are. It is used to ensure that the weights of the layer can be updated effectively, preventing the issue of vanishing gradients.
- The output layer has the same number of neurons as the number of possible actions in the environment, and uses a linear activation function.

```
#Feed-Forward Neural Network Model for Deep Q Learning (DQN)
model = Sequential()
#Input is 1 observation vector, and the number of observations in that vector
model.add(Input(shape=(1,env.observation_space.shape[0])))
model.add(Flatten())
#Hidden layers with 24 nodes each
model.add(Dense(24, activation='relu'))
model.add(Dense(24, activation='relu'))
#Output is the number of actions in the action space
model.add(Dense(env.action_space.n, activation='linear'))
```

Figure 4: The neural network

The DQN:

Implementing DQN Algorithm by applying Linear Annealed - EpsGreedyQPolicy:

A policy is a set of rules that governs how an agent behaves in an environment. In this task, the policy is set to Linear Annealed with **Epsilon-Greedy Q-Policy** as the inner policy, and Sequential Memory for storing the results of actions and rewards for each action.

Epsilon-Greedy means that the best (greedy) option is chosen most of the time, but occasionally a random option which is unlikely to be chosen is selected. An exploration rate – epsilon – is set to 1 at the start of the Q function training, and is gradually reduced as the agent gains more confidence in the Q values.

```
policy =  LinearAnnealedPolicy(inner_policy=  EpsGreedyQPolicy(), attr='eps',
                        value_max=1.0, value_min=0.1,value_test=0.05,nb_steps=10000)
```

Figure 5: Epsilon greedy policy

Defining the Dqn Agent for DQN Model

```
dqn = DQNAgent(model=model,
               nb_actions=env.action_space.n,
               memory=memory,
               nb_steps_warmup=25,
               target_model_update=1e-2,
               policy=policy)
```

Figure 6: The DQN

- The model parameter is the neural network model which has been previously defined.
- The nb_actions parameter is the number of possible actions in the environment.
- The memory parameter is the Sequential Memory which stores the results of actions taken and the rewards received.
- The nb_steps_warmup parameter is the number of steps the agent takes before beginning to learn.
- The target_model_update parameter is the frequency at which the target model is updated.
- The policy parameter is the Linear Annealed Epsilon-Greedy Q-Policy which is used as the inner policy as mentioned above.

```
#We can use built-in tensorflow.keras Adam optimizer and evaluation metrics
from tensorflow.keras.optimizers import Adam
dqn.compile(Adam(learning_rate=1e-3), metrics=['mae','accuracy'])
```

Figure 7: Compiling the DQN, using Adam as the optimizing function

Here we are using the Adam optimizer from the TensorFlow Keras library to compile the Deep Q-Network (DQN). The Adam optimizer uses a learning rate of 1e-3, which is a small, fixed rate at which the model learns. Additionally, two metrics are used to measure the performance of the DQN: mean absolute error (MAE) and accuracy.

```
#Finally fit and train the agent
history = dqn.fit(env, nb_steps=50000, visualize=False, verbose=2)
```

Figure 8: Fit the model to 'history'

To start training the DQM model, we will use the "model.fit" method to train the data and parameters with nb_steps=50000.

Q-Learning Update Applied to Weights

The Q-Learning update applied to weights in the Cartpole environment is a form of temporal difference learning. This involves updating the weights based on the difference between the current predicted value and the expected future reward. The weights are then adjusted based on the difference to maximize the expected future reward. Essentially, this method means that the model will ultimately be more likely to take actions that lead to higher rewards, and less likely to take actions leading to lower rewards.

As stated above, the error is calculated by the difference between the expected Q-value of the current state-action pair and the observed Q-value. The expected Q-value is calculated with the Bellman equation: Q(S, A) = R + γ * max(Q(S', A')). This equation states that the expected Q-value for a State-action pair is equal to the reward for action A in state S, plus the maximum expected Q-value of all possible actions from the next state S'.

- Q(S,A) = expected value of state-action pair
- R = reward for action in state
- γ = discount
- S' = next state
- A' = action that can be taken from S'
- max(Q(S',A')) = max. Expected value over all actions from next state

The observed Q-value is the value actually observed after taking the action. We can use these to calculate the error by getting the difference between the two values.

Upon calculating the error, we can update the weights of the algorithm with an algorithm like gradient descent, where the weights are updated to reduce error between the expected and observed Q-values, improving performance of the algorithm over many iterations/episodes.

In our implementation, the Q-learning weight update is performed when we 'fit' and 'compile' the model, as seen in figure 2 below.

```
from tensorflow.keras.optimizers import Adam
dqn.compile(Adam(learning_rate=1e-3), metrics=['mae','accuracy'])

#Finally fit and train the agent
history = dqn.fit(env, nb_steps=50000, visualize=False, verbose=2)
```

Figure 9: Compiling and fitting the model

The update is performed by the ‘fit’ method by training the agent when interacting with the environment, and updating the weights based on the observed rewards and Q-values. It is performed using the Adam optimization algorithm, as specified in the ‘compile’ method. This algorithm will adjust the weights of the model to minimize the error, as calculated above.

Independently Researched Concepts

Maximisation Bias:

Maximization bias in DQN can be addressed by using experience replay. Experience replay stores past experiences and samples from them randomly to generate new experiences. This helps to prevent the agent from becoming overly focused on a single experience or action, as it is able to sample from the stored experiences. Additionally, using a target network can help to reduce the effects of maximization bias, as the agent is not updating its estimates of the value of each action during exploration.

```
memory=memory,                  # experience replay memory
nb_steps_warmup=25,             # how many steps are waited before starting experience replay
```

Figure 10: Maximization Bias

Dueling Network Architectures:

Dueling network architectures in DQN refer to a type of neural network architecture that splits the network into two streams, one for the state-value function (V) and one for the action-advantage function (A). The V stream is used to estimate the value of a given state, while the A stream is used to estimate the advantages of taking a particular action in that state. This architecture allows the agent to more accurately estimate the value of a given state, and make more informed decisions about which action to take.

Random Seed Initialisation:

Random seed initialization in DQN is the process of setting a random seed for the agent to use when exploring the environment. This random seed is used to determine which actions the agent should take when exploring. By setting a random seed, the agent's exploration will be consistent across multiple runs of the same environment. This allows for more reliable results when analyzing the DQN's performance.

Results

The first plot in Figure 11 shows the reward achieved by the agent over time when interacting with the environment. This plot shows how initially, the agent is trying to succeed but failing to achieve a solid result, but as the episodes progress, the agent gets better at balancing the pole on the cart, eventually leveling out and achieving a maximum reward over consecutive episodes.

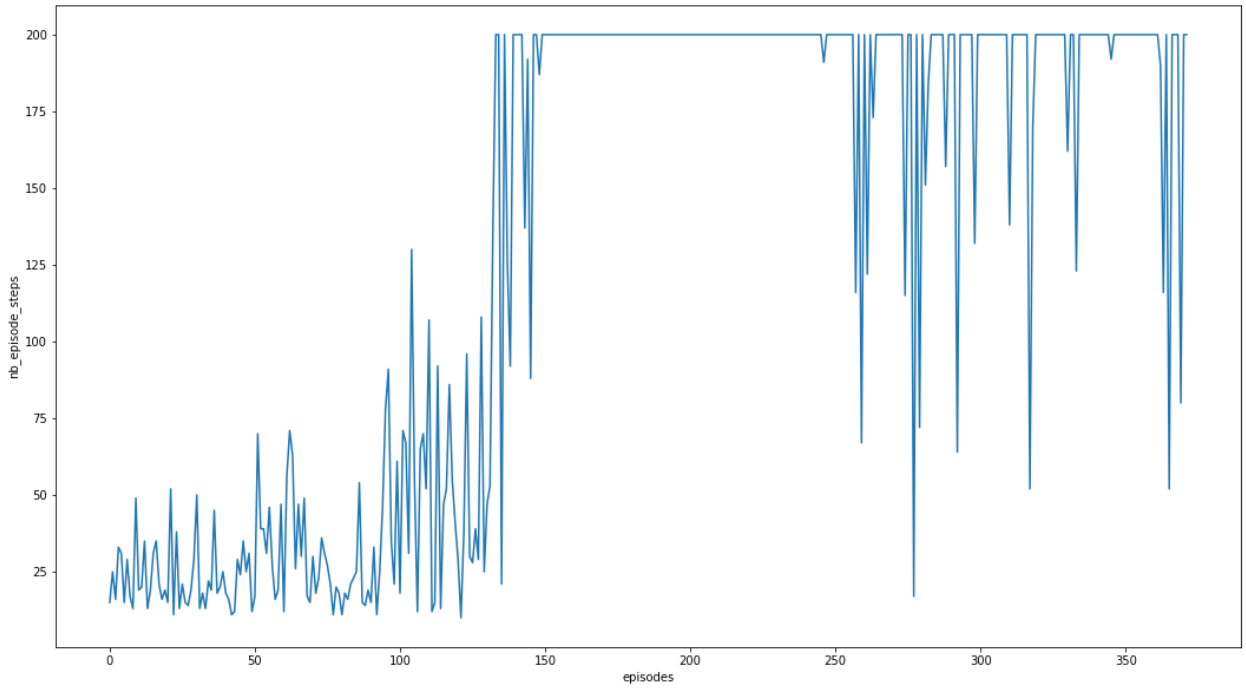

Figure 11: Reward over episodes for agent during training

This next plot in Figure 12 shows the action distribution for the DQN agent during training. We found that this plot is slightly irrelevant, since we only have 2 actions taken and after running numerous times, it is almost always near 50/50 for each action.

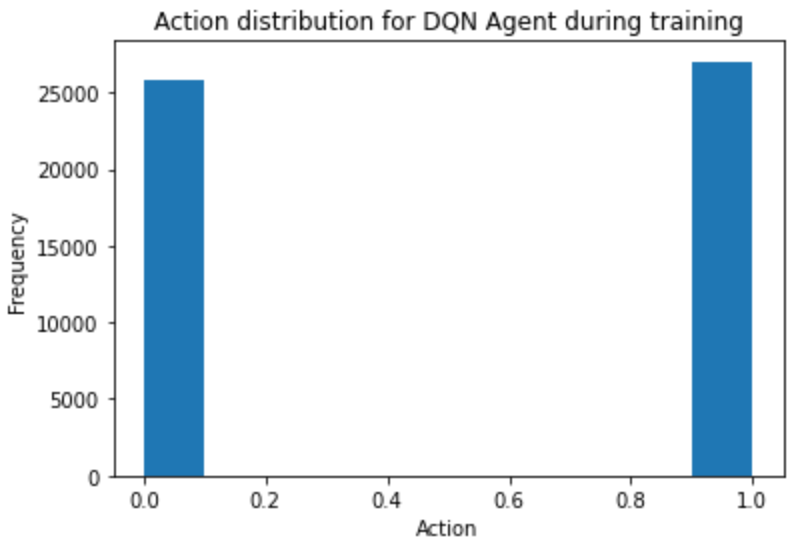

Figure 12: Action Distribution for DQN Agent during training

This plot was achieved through the creation of an ActionLogger class, which we attach to the model.fit function in order to log the actions the agent takes. We then access the log and create the plot. The first bar represents '0', the action for moving left. The second bar represents '1', the action for moving right. Ultimately, this plot was removed for lack of relevance, and it seemed to cause our agent to under-perform, as we were achieving a lower reward towards the end of the network. It was a similar case with an 'Observation Distribution' plot that we tested. Code snippets for the plot above can be seen below in Figure 13, 14 and 15.

```
from rl.callbacks import Callback

class ActionLogger(Callback):
    def __init__(self, interval=1):
        self.interval = interval
        self.actions = []

    def on_step_end(self, step, logs={}):
        if step % self.interval == 0:
            self.actions.append(logs['action'])
```

Figure 13: ActionLogger class

```
logger = ActionLogger(interval=1)
```

Figure 14: Creating logger var

```
#Finally fit and train the agent
history = dqn.fit(env, nb_steps=50000, callbacks=[logger], visualize=False, verbose=2)
```

Figure 15: Add ActionLog to 'fit' function

Evaluation of Results

The results above show the agent has successfully learned to solve the Cartpole problem by the end of its lifetime. Upon running tests on 100 episodes, we achieve a reward of over 195.0 for each episode consecutively, meaning that the agent has 'solved' the problem.

We managed to assess the reward over time using the plot shown in Figure 3, and were able to realize an issue using this plot when we attempted to add an Action Logger. This plot was also used to deduce whether the agent was performing well, and if it had 'solved' the Cartpole problem by the end of the cycle of episodes. It is clear that at the beginning, the agent is exploring and failing frequently, but towards the final episodes, the agent is solving most tasks and only failing sometimes.

We also added an Action Logger to assess actions taken by the agent in the environment, the results of which are seen in Figure 4. Ultimately, we felt this was irrelevant and just caused issues with the overall performance of the agent. With more time, we would have liked to discover why this occurred, as it seems strange that a log would cause an issue with the overall reward.

This network has proven to be an effective method for solving the Cart Pole task. The performance of the reinforcement learning agent can be evaluated in several ways. One way is to measure the accuracy of the agent's predictions and actions. This can be done by assessing the accuracy of the agent's predictions and by measuring the time it takes to complete the task. Additionally, the robustness of the agent can be evaluated by assessing its performance in different scenarios.

Finally, the performance of the agent can be compared to other reinforcement learning agents, such as comparing its performance to a baseline agent or to a different agent designed for the same task.The network was able to achieve a high accuracy in predicting the pole's position and velocity, and it was able to complete the task in a timely manner. Additionally, the network was robust, as it was able to perform well in different scenarios. Overall, these results suggest that this network is a promising approach for solving the Cart Pole task and that the metrics are relevant.

References

Phy, V. (2019). *Reinforcement Learning Concept on Cart-Pole with DQN*. [online] Medium. Available at: https://towardsdatascience.com/reinforcement-learning-concept-on-cart-pole-with-dqn-799105ca670 [Accessed 05 Dec. 2022].

Goel, K. (2022). *Supervised Learning vs Unsupervised Learning vs Reinforcement Learning*. [online] Intellipaat Blog. Available at: https://intellipaat.com/blog/supervised-learning-vs-unsupervised-learning-vs-reinforcement-learning/ [Accessed 07 Dec. 2022].

Gymlibrary.dev. (2022). *Cart Pole - Gym Documentation*. [online] Available at: https://www.gymlibrary.dev/environments/classic_control/cart_pole/ [Accessed 15 Dec. 2022].

Phy, V. (2019). *Reinforcement Learning Concept on Cart-Pole with DQN*. [online] Medium. Available at: https://towardsdatascience.com/reinforcement-learning-concept-on-cart-pole-with-dqn-799105ca670 [Accessed 15 Dec. 2022].