Computer Science and
Artificial Intelligence
COC257
B623995

REINFORCEMENT LEARNING FOR ADAPTIVE CONTROL SYSTEMS

by

Jordan A. Phillips

Supervisor: Dr. A. Soltoggio

Department of Computer Science
Loughborough University

April 2019

1. Acknowledgements

I would like to say thank you to my supervisor, Dr. Andrea Soltoggio, for guidance and support through out the process of this project.

2. Abstract

The discussion and evaluation in this report will be focused around the design of algorithms used in reinforcement learning tasks. I will be discussing and evaluating some well-known algorithms that have come to light recently and more-over, looking at the research with the Atari system, by Google DeepMind, with their Deep Q-learning Network.

Since the definition of an Adaptive Control System (ACS) is in-line with that of the Q-learning algorithm definition, the algorithms that can be produced will provide the outcome desired by the ACS. I aim to adapt Q-learning algorithms or other reinforcement learning algorithms for playing Atari games, which I will only focus on Pong, Breakout and Asteroids. *I will however, discuss the CartPole game available in OpenAIGym as an introduction to my understanding.*

Contents

3. Introduction

3.1 Motivation

The area of deep learning has brought about new applications of data such as the progression of speech recognition, which formerly was limited by inaccuracy, is now able to be more accurate with the help of deep learning[5]. By inputting sound waves in to a neural network you can produce an output of what the person may be saying, most speech recognition code uses Recurrent Neural Networks (RNN) to look at the current state of a sound wave and then predict the likelihood of the next letter. The model gets better at 'understanding' the flow of the language since letters like 'gxz' do not cohere to a 'regular' English language word and so you want to avoid predicting the letter 'g' followed by the letter 'x'[5].

In more recent times, DeepMind which was founded in London 2010, recently having been acquired by Google in 2014, published their Nature paper on the topic of deep reinforcement learning. The paper, produced in early 2015[6], showed some amazing statistics for playing the Atari 2600 library of games using the deep Q-learning network which was is a combination of reinforcement learning and deep neural networks. The paper showed that a game such as Breakout (Figure 1, Below), has a human (professional games tester) score of 31.8 and with DQN it is 401.2(26.9). This showed significant traction in being a great base for learning environments that were not known to the agent but only knowing the actions you could take given a state[6].

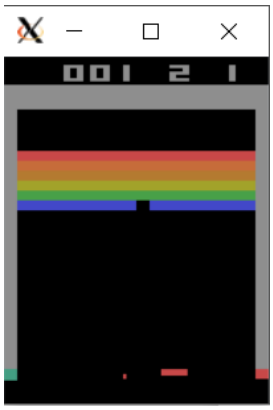

Figure 3.1: The Atari 2600 playing Breakout (Visualized by Xming)

The results of my testing will hopefully allow me to further my knowledge of the topic but also provide some meaningful data to which further analysis can be made of my study, by others, in to the area of deep learning.

3.2 Aims of the Project

3.2.1 Outcomes

The outcome of this project will provide an analysis of the algorithm design and parameter choice of playing the games on the Atari 2600 as well as looking at how different environments that have similar inputs such as Pong and Breakout (2 total actions) can vary in their learning process.

The simulations of games will be run on cloud services, since there is prior knowledge in using Google Cloud Platform it will be used to run the algorithms on. The tasks will be memory hungry due to storing of data that is being used in the networks and the size of arrays could be very large if the agent is running for a long period of time.

The success of the project relies in the comparisons between the environments, the differences with the agents performance upon being placed in a different environment is key to the deeper understanding of how the networks operate and manage given that new environment.

A final success criteria for this project is to further my own understanding of neural networks, deep learning, Python and virtual environments.

3.2.2 Objectives

1. Devise a literature review, write up the methods for deep reinforcement learning and research algorithmic techniques.
2. Set up a virtual machine (Google Cloud Platform) to run the algorithms on.
3. Use Jupyter Notebook to learn how to implement algorithms and display output for the implementation.
4. Develop an algorithm for playing the OpenAI Cartpole environment to gain an understanding of neural networks and algorithm design.
5. Implement the algorithm on Cartpole. Analyse the findings of the implementation.
6. Play Pong and Breakout from the Atari library.
7. Analyse how the games Pong and Breakout perform with the same hyperparameters and check online for the optimal parameters, analyse why the hyperparameters are set to the values that they are.
8. If possible use the algorithm on the Atari Asteroids environment. Analyse the findings with the Asteroids environment.
9. Provide a conclusive end to the report and detail to what the findings mean and judge the success of the project based on the requirements having been met.

3.3 Basic Reinforcement Algorithm Principles

A introductory algorithm for reinforcement learning is Q-learning. Q-learning is a type of algorithm that does not require a model to run, a policy tells the agent what the best expected value is at a stage so the agent can take the action that gets the highest reward. Figure 3.2 below shows a maximising Q-learning algorithm;

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

Figure 3.2: Q-learning equation with a maximising policy[25]

The goal of the Q-learning algorithm and reinforcement learning in general is to maximise future reward. Figure 3.2 shows that an agent is currently at the 'old value' from the equation, the agent wishes to progress in the environment as so to maximise the future reward[25]. The learned value is what the agent wants to maximise, more specifically the Q-value of the next state that is achieved by taking action a. The estimate of the optimal future value comes from a Q-table which is an array of Q-values relating to state and action pairs given the current location.

The discount factor is used to lower the value of the future Q-value since it is a prediction and the further you predict the Q-value the less likely success is. At each step t, the probability to succeed is denoted by γ, the discount factor[25]. The value of γ is a number between 0 and 1.

A learning rate is used to see how the new information is compared to the old information that the agent has acquired[25].

4. Literature Review and Discussion

The topic of deep learning has vast amounts of published papers, journals, books and online resources for understanding the area, implementations of algorithms and agents and new research of the area. A fundamental diagram of reinforcement learning is shown by Figure 4.1 (Below).

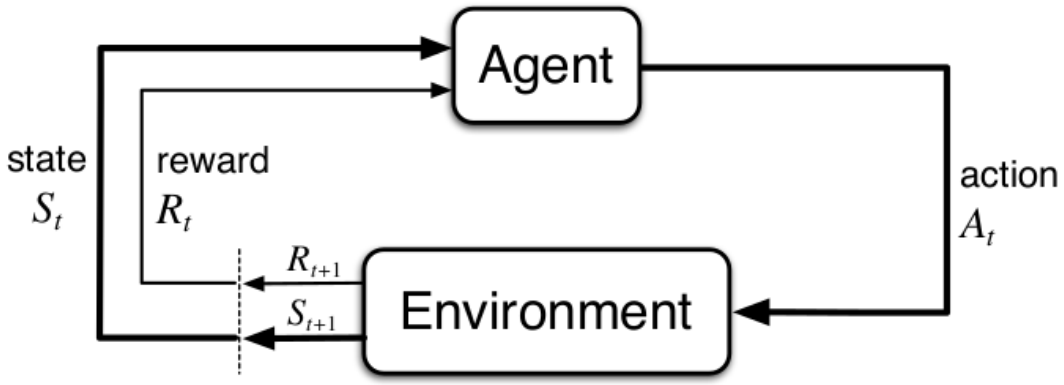

Figure 4.1: Agent-Environment Loop: Reinforcement Learning: An Introduction 2nd Edition, Richard S. Sutton and Andrew G. Barto [Image taken from source 3]

Figure 4.1 is again, a classic diagram of how reinforcement learning works. The agent begins in state S_t and performs action A_t. The environment reacts to this action, A_t and returns a new state, S_{t+1} and a reward R_{t+1} to the agent. The loop iterates until the environment has been solved or terminated[3].

Depending on how the agent is programmed, the agent may be based upon a greedy approach. A greedy approach is where the agent will take some action A_t and look for R_{t+1} where R_{t+1} is max(R) (highest immediate reward), the state-action pair, with state S_{t+1} is then chosen which corresponds to max(R) and the agent continues this until terminated or the game is won or lost.

4.1 Reinforcement Learning

Reinforcement learning is one of four main categories of machine learning, the others being supervised learning, unsupervised learning and self-supervised learning, according to (Chollet 2017, p.95)[7]. Reinforcement learning is where an agent gets a reward, either positive or negative based on the action taken in a given state of the environment. The goal of the agent is usually to maximize some cumulative reward. When DeepMind published their Nature paper on *Human-level control through deep reinforcement learning*[6], they showed that by using reinforcement learning algorithms they can surpass human-level play on games such as Pong, Breakout and Space Invaders. In this paper, they discuss their hyperparameters for running the agent, they used an informal search on a select number of games to get values that could be applied to all the games.

DeepMind use a single algorithm for all games on the Atari 2600 in order to meet the goal of general artificial intelligence, a widely sought-after achievement in the field of artificial intelligence[6]. Using the concept of a deep neural network discussed in the next section, 4.2 on page 5, the network can 'identify' object properties and structures from raw input pixel data.

4.2 Deep Neural Networks

A deep neural network is a neural network that contains layers between input and output, the number of layers between these relates to how 'deep' the neural network is. Deep neural networks, in terms of reinforcement learning, must work without there being a supervisor or teacher to help the network in categorising their findings. According to the paper by (J.Schmidhuber, 2014-15, p.93)[9], deep neural networks can suffer from vanishing or exploding gradients. Vanishing and exploding gradients are caused by values being multiplied or handled in some way which makes them move to infinity or down to zero and therefore messing up parts of the neural network weights and predictions of future values. As well as this, '*cumulative backpropagated error signals may shrink rapidly or grow out of bounds*'[9] according to Schmidhuber.

4.3 Convolutional Neural Networks

Convolutional neural networks (CNN) are a subset of DNN, they are good for finding features in image data[8,10]. CNNs can extract basic features like edges, ends and corners in images, each unit in a layer gets input from a previous layer to which it overviews a larger area or neighbourhood of pixels or values which culminates to an edge or property of the overall image at some level[8].

CNNs have proven success in ability to work in game environments such as the 'computer Go' game by (Clark and Storkey, 2014) who showed *move prediction accuracy of 41.1% and 44.4%*[11] which had surpassed previous state-of-the-art algorithm techniques on this problem. The game of Go relies heavily on pattern recognition to predict board control[11]. The Atari 2600, while there isn't 'control' implied in most games, there is the same element of prediction needed to foresee good 'moves' that the agent should take.'

4.4 Jupyter Notebook Development

Jupyter notebook, formerly IPython, is a useful environment for algorithm design and code debugging[23]. Jupyter uses the Python programming language, which is a very popular algorithmic and mathematical language. The modular nature of Jupyter means that you can crop and change parts of code that you can test in subsections to see how it affects the result.

It is easy to get to grips with the Python language using modular sections of code rather than having to comment out bits of code you do not want to be executed like in Notepad++ for example. Jupyter notebook is particularly useful for this project as the algorithms will need to have output that you can readily manipulate. The modular nature of Jupyter can be shown in Figure 4.2 below:

```
k = 0
count = 0
for i in score_array :
    if i > k :
        count = count + 1

print ("Number of games the AI won : " + str(count))
```

```
Number of games the AI won : 44
```

```
no_ep = score_array.index(int(max(score_array)))
print(no_ep)
print(max(score_array))
```

```
21428
10.0
```

Figure 4.2: Snip of Jupyter Notebook, showing two separate cells that can interact with each other or be run separately.

4.5 Summary

In summary, there is a vast amount of literature, videos, research and journals on the topic of deep learning and this study should provide a valuable addition to the existing material. Using the existing topics however this paper will create new links for algorithms and how they interact with the environment that they are utilised in. This paper will also seek to affirm previously stated conclusions on some tested problems to verify its integrity of a research task. A result of 18.9 score for Pong was achieved with DeepMind's DQN that was tested against the human player which only scored 9.3. If the algorithm discussed in this paper can score higher than 9.3 and get in the range of 18.9 or in games then this study will have some success in producing results similar to the pioneering research done by DeepMind[6].

Many papers and sources use similar hyperparameters that have already proven to be the most successful in given environments and so the paper will not aim to test hyperparameters[1,5,6,12] in this way, it will try to reproduce some of the results found in other sources. This paper will mainly focus on how these parameters come about given the conditions for the environment as not much is discussed in detail about this topic on many sources.

5. Plan

5.1 Methodology

The project will use an Agile methodology of software engineering, the Agile approach fits the project best since the sections of development from the Gantt chart seen in section 5.2 on page 7 is a series of sprints that should not be overrun. The Agile method suits this project due to the nature of software development and the unpredictable nature of bugs and changes in the requirements of the project.

The Agile method allows the project to be split up in sprints, the research sprint is quite long, most other sprints are manageable with only a couple weeks but it is important to get research done and out of the way in order to progress.

The project made use of a Trello board to keep track of the individual tasks that were more detailed than in the Gantt chart, see Figure 5.1. The Trello board was more dynamic than the Gantt chart and this meant that the project could flow more naturally as research was not limited to Week 9 for 4 weeks as it happened continuously throughout. The majority of research for algorithms did however, take place during those four weeks. The Trello board can be seen in section 5.3 on page 8.

5.2 Gantt Chart

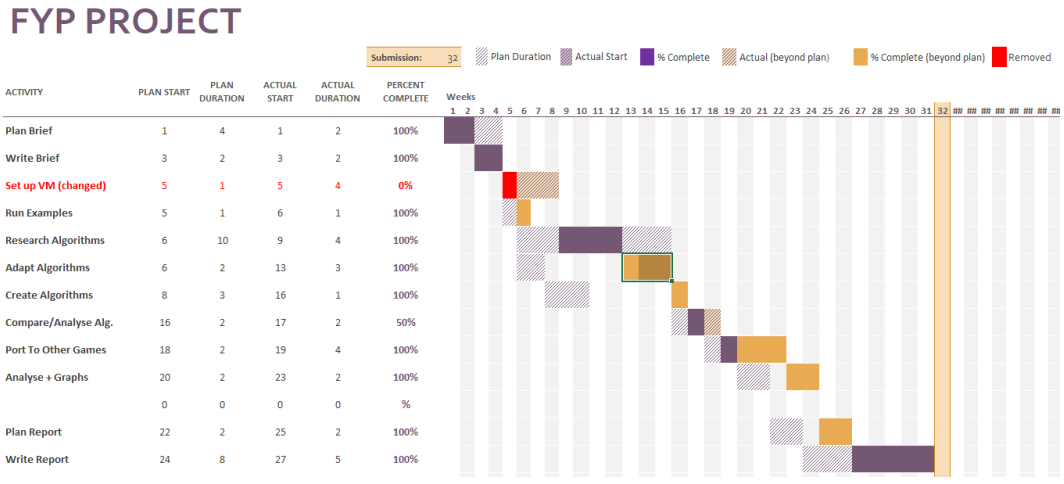

Figure 5.1: Final Gantt chart that was used throughout the project to keep progress on time.

5.3 Trello

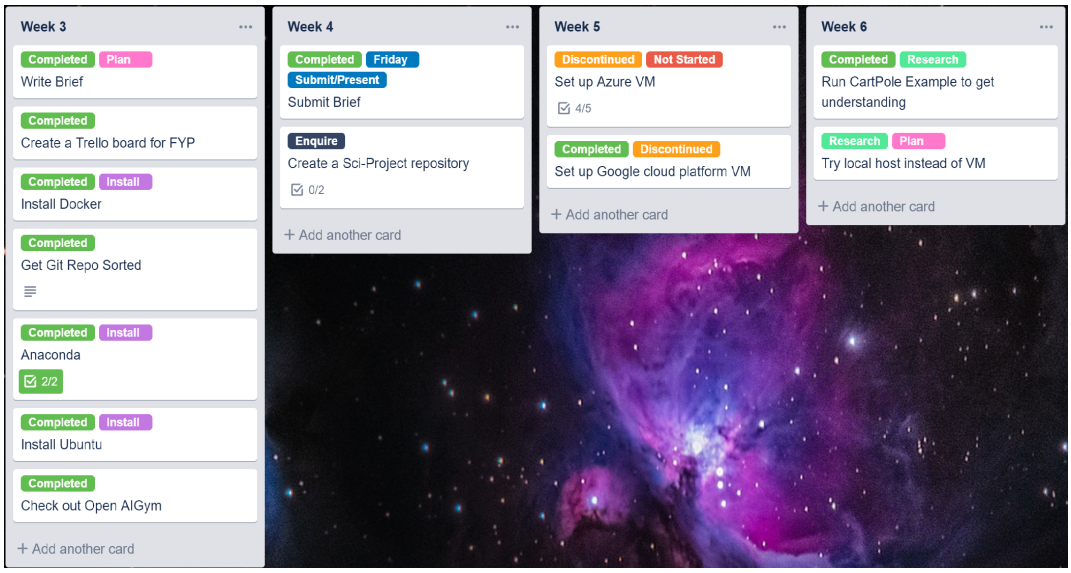

Figure 5.2: The segment of a Trello board showcasing the weeks 3 to 6 in more detail than the Gantt chart in Figure 5.1.

The Trello board works well with Agile development as you can easily track progress through the week with the cards and labels that Trello provides. Cards can be moved from different weeks and assigned checklists in some cases should a card have more detailed requirements. As noted in 5.1 it is unclear of what happens in weeks 5 - 8 on the Gantt chart, since the overall goal of those weeks was changed throughout the weeks it occupies. A breakdown of the same section of the Trello board is more detailed. The Trello shows that whilst no progress was made on the VM installation, since complications caused both VM services to be inoperable, there was a suggestion of trying to host the algorithms locally. The solution is discussed further in section 6.1 on page 11.

Trello labelling helps with categorising tasks and helps later on when report writing as at each week there is all the information in the Trello cards of how the project evolved and progressed.

5.4 Testing of Algorithms

The algorithms are tested as they are written, using Jupyter Notebook, the code can be segmented in to chunks. This allows the code to not only be tested as separate chunks to help with debugging, but also allows the code to be run more effectively when changing parts of the code itself.

```
1  no_ep = score_array.index(int(max(score_array)))
2  print(no_ep)
3  print(max(score_array))
```

```
25394
15.0
```

```
1  print(len(score_array))
```

```
200383
```

Figure 5.3: Example of segmented code in Jupyter Notebook. (Breakout, indexing of maximum score)

As you can see from Figure 5.3, the code is more efficient. Since you do not want to run the length of the *score_array* each time you execute the code, it would be inefficient (the code has completed learning and this is extra analysis of the results) to run it when you wanted to see just the number of episodes *no_ep*.

Using Jupyter notebook as a development for testing and debugging saved time and computational power that was otherwise, in short supply, since the algorithms were run on a local machine in the final iteration of development.

5.5 Risk Assessment

The risk assessment will be using a grading method of risk called RAG (Red, Amber Green), where RED = high risk, AMBER = medium risk and GREEN = low risk[20].

5.5.1 Virtual Machines

The risk for setting up virtual machines is high, if the virtual machines are unable to run the algorithms it could mean that the software, made for Ubuntu, may be incompatible with the Windows 10 machine. The virtual machines are more powerful than this current laptop and so the algorithms will take much longer to run and thus may extend the algorithm testing sprints by long time. Virtual machines will need all the dependencies for the software installed and this may cause issues with compatibility and version control of the software being used by the algorithm.

Another added risk of virtual machines is using the budget of the cloud services up and not being able to continue the algorithm due to the budget of computation exceeding the computation allowed for a user. The RAG for setting up virtual machines is RED.

5.5.2 Deep Learning Algorithms

Deep learning algorithms make use of certain hyperparameters that are configured for optimal performance. If the hyperparameters are not set correctly, the algorithm will begin to behave unfavourably. This might cause exploding gradients or over-fitting and not learn the correct action to take given a current state, as learning rates being too high can cause the wrong relationships of states and actions to be made. The risk for deep learning algorithms is GREEN since there are plenty of tested hyperparameters available on many sites[1,6,12,16].

5.5.3 Cloud Services

Cloud services can be unreliable and may go down, this can cause the algorithm to halt and will need to be reset completely. Resetting the algorithm will be costly to time most of all, some algorithms can run for weeks to learn depending on the power of computation of the machine it is being run on. Cloud services are generally backed up by lots of clusters of machines and a cluster going down will not affect your algorithm however, there is still a risk, even with a company like Google. This risk is AMBER since should an error occur, even if not likely, it could cause weeks of setback.

5.5.4 Bugs and Errors

Bugs are almost certain with any code and so there is always the risk of losing time due to bugs persisting and not being identified straight away. Through using Jupyter notebook, bugs should be easily identified and since code is modular within Jupyter notebook (see Figure 5.3 for an example) you can test sections of code for errors and bugs. This risk is GREEN as most bugs are addressed with a few lines of code added or removed.

5.5.5 Algorithm Design

Algorithm design is important to get right as you may end up running an agent for an extended period of time and it is not learning correctly, this is aided by using the correct hyperparameters mentioned in section 5.5.2. However, using the correct optimizer and algorithm design is just as important[14, 19]. The environment that the algorithm is placed in dictates the method used to solve it, as seen in section 4.3, the convolutional neural networks work better for image data than other networks[8,10]. The risk for getting the right algorithm design for an environment is important to get right, however it is widely known what type of network works best in what type of environment[8,10]. The risk is thus, relatively low, this is a GREEN level of risk.

6. Algorithm Development and Code Implementation

6.1 Proposed algorithm for learning

In this paper the psuedocode algorithm below is based on the algorithm mentioned in 7.1 on page 14 as well as (Siraj Raval, 2017)[1, 12]. The goal of this algorithm is to fit the Atari 2600 'Pong' environment and then use it within the other games on the Atari 2600 system.

Algorithm 1: Atari AI

Result: Play and learn from the specified Atari 2600 environment and meet a goal for playing

Input: Screen pixel data;

Output: Graph of played games;

import dependencies;

env = gym.make('Pong');

state_size = env.observation_space;

action_size = env.action_space;

batch_size = b;

learning_rate = 1e-4;

gamma = 0.99;

decay_rate = 0.99;

episode_number = 0;

observation = env.reset();

while *True* **do**

    env.render();

    observation, reward = env.step(action);

    if *done* **then**

        episode_number += 1;

        discount_weights(gamma, learning_rate);

        observation = env.reset();

        if *reward >= 18* **then**

            print Pong AI is now better than human

        end

        if *reward == 21* **then**

            print Game won and no points lost!

        end

    end

end

6.2 Short Introduction and Setup

The algorithm and the code in its entirety is written in Python, for CartPole the version, Python 3, was used however for compatibility reasons, the Pong and breakout versions used Python 2.7.15rc1 since it allowed more libraries to be used with Xming. The study will be using the Jupyter Notebook framework for ease of testing and the most current version will be used. Mentioned in more detail in section 7.3 on page 18, the initial plan was to run the algorithm using the power of cloud computing on some virtual machine that hosted Linux OS. The paper initially discusses using Google Cloud Platform (GCP) at the time was a familiar interface and sought to reduce computational time and memory significantly as mentioned in section 3.2 on page 2.

GCP offers a starter of £250 for any project which was assumed to be enough for running the algorithm on the different environments and in time to produce the output before the £250 was used up in memory and/or time of the virtual machine usage. Upon setting the virtual machine up, it became apparent that although the budget for operating the VM was enough there was no way of rendering the result of the agent through the platform. Permission errors that could not be solved due to the terms of service for using the Google machines, did not allow the screen to connect to the local device.

Other virtual machines are available from other providers such as Azure, yet upon creating an account for Azure, the resources available were more restricted and had less budget. As well as budget, Azure had similar issues with permission errors so another solution had to be found. This set the project back many weeks from the Gantt charts original projection as a solution was hard to find.

After finding a tutorial, the installing of the Ubuntu application onto the Windows 10 system allowed rendering of the environment through a program called Xming. Xming allows the Ubuntu VM to display the environment by exporting the display of the 'localhost' to the Ubuntu VM. The Ubuntu VM was essentially its own environment that was separate from the local machine which meant that the dependencies for OpenAIGym and the Atari 2600 libraries could all be installed on the optimal versions without using the Windows 10 equivalent.

For the CartPole demo, Keras was used for creating the model of the neural network, Keras is is one of the leading high-level neural network APIs[13]. Since Keras is written in Python it makes Keras good for processing numerical data and it supports neural network computation engines[13]. In the CartPole demo, Keras uses the TensorFlow back-end which is a large machine learning library.

Keras allows layers of a network to be added in simply and in the example in the next section, 7.1 on page 14 the model being used is a sequential model for networks and each layer is dense. Sequential means that the model is a linear stack of layers and that processing happens layer by layer[13]. The Keras API for CartPole uses the Adam optimizer[14] which is an algorithm used to update

weights in a network and *'computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients'*[14]. The adaptive learning rates mean that performance improves[14].

The algorithms being considered[1, 12] use NumPy, which is a large package for Python used to construct arrays and matrices effectively and allow fast operations to be executed with those structures[15]. Working with pixel input matrices and large arrays makes NumPy necessary for the fast computation needed for learning in networks.

Matplotlib, the extension for NumPy, allows 2-Dimensional graphs to be displayed and used for analysis towards the end of the paper to view performance of the agent in the environments. Previous knowledge of matplotlib will assist in the implementation of the necessary code for the graphical output. No complex graphs are needed for the analysis so matplotlib can provide all the graphical capability needed.

The OpenAI gym toolkit library contains all the environments and demonstrations of basic problems to attempt to solve[16]. The gym library gives the environments a shared interface to allow algorithms to be created for one game and to be used on all games[16]. Gym contains all the environments from the Atari 2600 library of games as well as robotic simulation, 2D and 3D, algorithmic environments and classical control environments such as CartPole and MountainCar[16]. DeepMind also used the gym library for their revolutionary paper on reinforcement learning, which this study aims to follow on from[6].

6.3 Parameters

The Parameters for the CartPole demonstration were already implied to be optimal and so it was not changed, although some testing was carried out for understanding and parameters were changed to different values. In an example the learning rate, gamma and the epsilon decay were changed to see how this affected the overall learning and performance of the CartPole agent. Changing the learning rate and keeping the other properties the same yielded mixed results, the errors of the agent were blown up and the agent became erratic with massive pole movements and an overall worse score to end with after 100 episodes. The table of testing the CartPole parameters is shown in table 7.1 on page 16.

7. Reinforcement Learning in Specified Environments

7.1 CartPole Demo - *Understanding Environments*

This study will be using TensorFlow, Keras and Jupyter to create the neural networks and run the code for testing the algorithms. For example, in the CartPole demo, which this paper will talk through and evaluate in this section, the Keras API front end for TensorFlow allows setting the network layers to Dense, this is essentially a matrix vector multiplication. The matrix is full of trainable parameters which are updated when a loss function is calculated for the weight of the neurons in the network.

The CartPole demonstration was taken from a YouTube Video series by (Jon Krohn, 2018)[1]. The paper will start by discussing the parameters surrounding the game itself and which parameters have been set up around it for learning and playing the game.

Parameters:

- *state_size*, this is always 4 in the CartPole game because the game is very basic, since the only states are: Location of the cart, Velocity of the cart, Angle of the Pole and the Angular velocity of the pole.

- *action_size*, the agent can only ever take 1 of 2 actions at each decision, either move left or right.

- *batch_size*, used in the selection of memories and for learning with backpropogation.

- *n_episodes*, the number of times the agent will attempt the game.

The Agent:

The agent playing the CartPole game will need to learn from its previous attempts but not be slowed down computationally by the memory retrieval. A few class parameters are set up for the agent, including the way in which an agent explores. The agent has an exploration rate, epsilon, which is initialized as 1.0. This value of epsilon will decrease over time by a factor of 0.995 per decision down to a minimum of 0.01. The lower the value of epsilon, the less likely the agent is to explore, so

as the value of epsilon decreases to the minimum, the agent will get to a point where it only explores new actions in a state 1% of the time, as a result it will instead, exploit the past experiences of the agent.

The neural network is constructed by the Keras API to be a sequential class structure, where layers are added in the order of computation, as stated by (Jovian Lin Ph.D., 2017), there are limitations to this model, where agents that reuse layers cannot be given sequential models to work from. Since the CartPole agent is very basic and does not require previous layers to operate successfully, the sequential model is adequate[2]. The sequential model is also *Dense*, this means that the neurons (24 in this example for code 7.1):

$$\textit{'model.add(Dense(24, input\_dim = self.state\_size, activation='relu'))'}[1] \tag{7.1}$$

are trainable since the matrix will be multiplied by a vector based upon the calculated loss function, based on how well the agent is performing. This is known as backpropagation.

A function *remember*[1], will save current information, this is the state, action, reward and the next state as well as if the agent is done. An *act*[1] function is used to 'decide' what to do upon being in a state. The *act*[1] function randomizes a number and if this is less than or equal to the epsilon then a random action is taken for that state, this is the exploratory nature of the agent. If the number generated is greater than epsilon the agent will guess an action based upon the remembered data from prior playthroughs depending on the max reward of a playthrough.

The last major function of this agent is the *replay*[1] function, this function takes the *batch_size* parameter set up earlier and draws up a sample of previous playthrough data based on the *batch_size*, the larger the value, the longer it will take to decide but will be more accurate. The reward is predicted if the agent is not done with a playthrough, i.e. it has not died or won. The target for the agent is calculated on this expected reward and thus the future reward can be calculated based upon this target of where the agent calculates it will end up. The neural network is fed the information relating to the next state and then it predicts the value of the next state.

In the *replay* function, there is a part of the Q-learning algorithm, the formula finds the maximum value of the next state given the current state. The equation, 7.2 below, shows the target which is the learned value from the neural network when the current state Q(S, A) is inputted.

$$target = (\ Q(S, A) + \gamma * max(\ Q(S_{t+1}, A))) \tag{7.2}$$

In the next environment sections this paper will aim to implement my own algorithm from section 6.1 on page 11.

7.1.1 Results

Parameters for CartPole Testing			
Learning Rate	**Gamma**	**Epsilon Decay**	**Average Score**
0.0010	0.9500	0.9950	213.3
0.0100	0.9500	0.9950	081.7
0.0001	0.9500	0.9950	021.3
0.0010	0.6000	0.9950	019.0
0.0010	0.9999	0.9950	089.2
0.0010	0.9500	0.9999	038.2
0.0010	0.9500	0.9000	117.5

Table 7.1: Results of changing parameters on the CartPole agent.

Table 7.1 shows the tested parameters of the CartPole game. The first row is the base case of parameters which was seen as the optimal testing parameters[1]. This was confirmed to show the highest average score for the 100 games that were played and so confirms the best parameter optimisation suggested earlier.

Table 7.1 also shows that both increasing and decreasing the learning rate have a negative effect on the average score of the agent. Though it can be assumed that because the learning rate was so low, the agent may have simply required more games to get to the same average[17]. Figure, 7.1, on page 16, shows how the learning rate relates to loss[18].

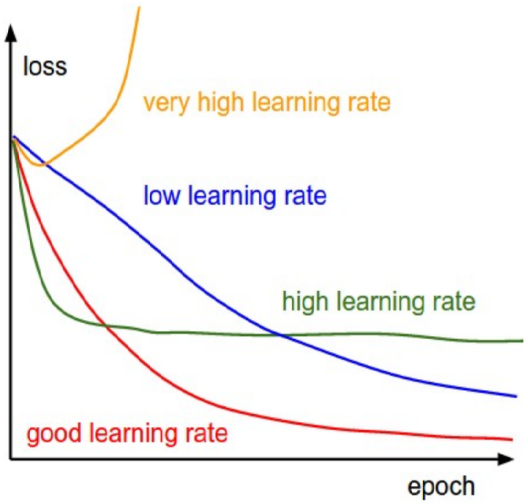

Figure 7.1: Learning rates affecting loss of function, image credit [18].

Figure 7.1 on page 16 shows the epoch against loss, where the loss is the inconsistency between ac-

tual value and the predicted value. High learning rates increase loss, as shown, since you emphasise on the errors made by the agent as the learning rate is a multiplier for some weight. Low learning rates are slow to converge since the multiplier of the weight is smaller it takes longer to realise the actual weight of the action.

From Table 7.1, we see that where learning rate is highest (0.0100), on row 2, the average score is less than the base case (row 1). A lower score has occurred since the agent may have been wrongly trained in the process and where a bad weight has occurred it was multiplied by a large weight and so it wasn't eliminated as a bad action to take in that state.

Contrasted to the higher learning rate, when the learning rate was lowered (0.0001), on row 3, the agent was so slow to converge that the agent seemed to perform poorly trying to balance the pole of the cart. This is also confirmed by Figure 7.1[18].

Figure 7.1 shows that both increasing and decreasing gamma worsen the average score of the agent compared to the base case. The same is said for the decay of epsilon. When epsilon decay is increased the agent will be less likely to explore quicker than the base case, this means that the agent may not try new actions. The value of epsilon is the chance that the agent will do some random action, epsilon = 1.0 means that the agent will choose a random action at a chance of 100%. If epsilon = 0.1 means that the agent will choose a random action 10% of the time.

7.1.2 Conclusions

To conclude the CartPole code, it achieved the goal of helping with a greater understanding of how a Q-learning algorithm works and what the parameters do within the algorithm to make it work. It shows that whilst you can change the parameters yourself and see the results of the changes first-hand, it is more efficient to take trialed and tested hyperparameters. For a reinforcement learning algorithm especially, since there are many online sources that have got tested parameters and reinforcement learning iterates over the parameters multiple times and so you want to avoid exploding or vanishing gradients[9].

7.2 Atari 2600 System

The Atari 2600 was released on September 11, 1977. The Atari console had major success with games such as Pac-Man. The controller for the console consisted of a joystick with a button, the joystick had eight possible unique directions that allowed for more than the standard: up, down, left, right, combining these to create new directions like up-left and down-right. In this study this paper will look at the three games, Pong, Breakout and Asteroids. The first two, are games that take place in an environment that has 3 (Breakout having 4 technically) actions to take in any given state of that environment. In the case of Pong, the player can choose to complete actions:

Pong: [Up, Down, NoOperation],
Breakout: [Up, Down, NoOperation, Fire],
Asteroids can have 18 possible actions for a given environment:
[Left, LeftFire, Left-Down, LeftUp, LeftDownFire, LeftUpFire] are just the actions containing 'Left'.

7.3 Pong

7.3.1 Initial Plan

This section started out trying to turn the Cartpole example that had been worked with prior to this into a Pong variant. However as the process of converting began to change the Cartpole demo it became almost too different that it seemed pointless trying to adapt it when the code was nearly completely different. In an example, the Cartpole environment works with known values about the environment and Pong takes pixels as input, this difference alone meant that a new approach had to be taken.

As implementation of the algorithm began it was quickly realised that it wasn't going to work with the Windows 10 machine. The Atari environment rendering 'env.render()' works only with Linux and so another setback came about. It took a long time of researching and failing to realise the current setup. Instead of rendering on the local machine, a program called Xming was used, this software allows the algorithm to render a window within my machine that can work directly with Linux. To get around having to install a new OS on the local Windows 10 device, a tutorial on how to install Ubuntu on a Windows machine, that can use Atari OpenAI gym, on YouTube (YT) was used. (Talitha Trippel, 2017) The tutorial created awareness of a Windows store application which is essentially a local virtual machine (VM) application which contained the main core of the Ubuntu system.

The algorithm used to run Pong was not original, discussed in 7.3.2 on page 19, though the concepts and workings were still understood, it was decided that it would be used for the Atari games since it had proved to be successful in the video it was presented on. The algorithm, comes from (Siraj Raval, 2017)[12] and is available on YouTube and GitHub. This algorithm helped to create the psuedocode algorithm that was amalgamated in section 6.1 on page 11.

The game of Pong is an 80x80 pixel grid for the algorithm that was used, the 80x80 is the minimum environment size that the game space can fit inside. Figure 7.2 on page 19, shows the Pong environment rendered by Xming.

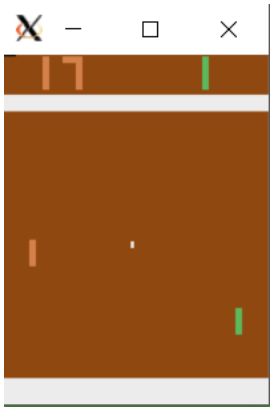

Figure 7.2: The Atari 2600 playing Pong (Visualized by Xming)

A seen in Figure 7.2 (Above), the AI agent is the green paddle and the Atari 2600 agent is the orange paddle. When the score of the game reaches 21, the environment is reset and the game is won by the player who scores 21. As mentioned, the game of Pong has three available actions given at any state of the environment, being; Up, Down and No Operation.

The Pong environment contains data that is not needed when playing the game itself, for example, the colour of the game-space is not important during the model and so you can preprocess the image to get a more streamlined version of the image. This preprocessing allows for faster computation of the neural network and better use of storage as even by just changing the image to black and white can allow for only a third of the memory to be used since the RGB scale, (0 - 255, 0 - 255, 0 - 255), is converted/reduced to (0, 255) greyscale.

7.3.2 Implementation

The algorithm used for the Pong implementation was drastically different from the CartPole algorithm due to the environments being too dissimilar. The agent is trained using stochastic policy gradients, this means that the outcome of an event is not certain (stochastic) and the policy is trained by observations of an environment[19]. The agent performs similarly to the agent in Figure 4.1 on page 4, however the agent has an agenda, a policy. This policy is used to decide which action to take based upon the current state and policy, where P is the policy: $P(a \mid s)$, a is the action and s is the state. The policy, P is the probability of taking the action a given the state s[19]. Since this policy, P is stochastic, the output is the possibility of the action taken.

In the Pong implementation, a package, Pickle is used. The pickle package for Python allows Python object hierarchy to be converted into a byte stream[12], pickling allows the data that is pickled to be cached easily for retrieval later on. Instead of using Keras API like the CartPole implementation, the Pong algorithm uses lower level TensorFlow which is just a more configurable Keras. Keras uses TensorFlow but obstructs most of it to make it easier for the user to add network layers for

example[13].

The Pong algorithm configures five hyperparameters, *H*, *batch_size*, *learning_rate*, *gamma* and *decay_rate*. *H* is the number of hidden layer neurons, the *learning_rate* has been chosen to be 1e-4 so that it will converge. The discount factor, *gamma*, has been chosen at 0.99 so that later rewards will be exponentially less important and the *decay_rate* to be 0.99 as well which is used in the RMSProp (leaky) optimizer. A final hyperparameter *batch_size* is set up to do a parameter update after every 10 episodes[12].

The Pong algorithm uses the RMSProp (Root Means Squared) (leaky) optimizer, this optimizer updates the Adagrad adaptive learning rate method to reduce the volatile nature of the Adagrad learning rate. By using a moving average of squared gradients, RMSProp updates the learning rate of weights of the model based upon the decaying average of squared gradients[18]. The RMSProp optimizer restricts the fluctuations with the weights and so the agent is more likely to make better decisions as the weights are more uniform[21].

The Pong algorithm uses the sigmoid activation function to keep values between 0 and 1 for the prediction of the probability for some input[12].

Dissimilar to the CartPole game, Pong has a more complex environment, the Pong environment is from the Atari library and has an RGB (0 - 255, 0 - 255, 0 - 255), this preprocessing, function *prepro*, mentioned earlier erases the colour to turn the input grayscale. As well as removing colour, the preprocessing function crops the original Atari screen size from 210*160 pixels to 80*80[12]. The fewer pixels to assign weights and do calculations on the less time the agent will take to learn. The preprocessing removes the background since it is consistent throughout the game and does not change based on actions, unlike games such as Montezuma's Revenge where the background must be kept since it does change. The main parts of the environment such as the ball and paddles are kept so that the weights can be applied to those pixels, the 80*80 frame is then converted to a one dimensional vector for easier and faster computation and processing.

A function *discount_rewards* takes a one dimensional array of rewards and computes the discounted reward, initially set to zeros. A variable, *running_add*, stores the sum of all rewards for that vector where the previous value is multiplied by the value of *gamma*. The calculated sum is then added to the discounted reward matrix and returned.

The next part of the implementation of code contains the forwards and backwards pass to adjust weights and compute the error respectively. The forwards pass uses ReLU (Rectified Linear Unit), which is an activation function which in this case turns values to zero if they are less than zero to keep linearity. In the forward pass, the weights are analysed and the model decides should the paddle

move up or move down.

The backwards pass takes the array of hidden states *eph* and the gradient *epdlogp*. The weights are calculated by using the dot product of the transpose of the hidden states and the gradient. The weights are put through each layer to be combined with the input observation and then the set of weight derivatives are returned.

To create the environment of Pong, as seen in Figure 7.2, the environment is assigned the OpenAI gym Pong, by the code 7.3:

$$env = gym.make("Pong - v0") \tag{7.3}$$

$$observation = env.reset() \tag{7.4}$$

The code, 7.4, is used to reset the environment to the very initial state of the game, S_0.

In this section of code three 'bookkeeping' arrays are set up to be used for the analysis later when the agent is done and some other arrays for gradients, hidden states, observations and rewards are set up. A few other variables are initialised such as episode number to use for averages and keeping track how many games have been played by the agent.

The training of the agent is contained within a while loop that never ends *While: True*, ensuring that the agent plays uninterrupted so that it can be left in the background for a long period of time and not need any micromanaging. Initially the current observation of the environment is preprocessed and if the loop has not just started (agent has been iterating through states), a difference between the previous and the current observation (frame) is calculated.

The probability of actions and hidden states are calculated from the forwards pass and then the action is decided upon by code 7.5:

$$action = 2, if : np.random.uniform() < aprob, else 3 \tag{7.5}$$

Code 7.5 shows how the probability of choosing action 2 or 3 is based upon the probability of choosing a sampled action, if *aprob* is larger than the random number between 0 and 1 then do action 3. The gradient is updated with the choice of the action so that is encouraged to be taken. After the action has been selected the environment must be 'stepped' by that action, stepping the environment returns 4 values, the observation of the step, the reward, if the agent is done and extra info. A reward is appended to the reward array after the step so that the previous action reward is stored.

When the agent is done, the episode number is increased, all arrays are updated such that the values from the end of the game are submitted to the array. The discounted expected reward is calculated

backwards through the agents playthrough and the rewards are standardised to help control the gradient[12].

A concept called 'Advantage' is used to describe how good the action is compared to the average of all actions using the discounted reward[12]. The gradient is then calculated from the backpropagation of the advantage and hidden states, this is then applied to all the weights in the model. When the episode number is a multiple of 10, the RMSProp controls the gradient and makes it less volatile. After the gradient has been normalised and the values for analysis have been recorded in arrays, the observation is reset to S_0 and the agent plays through the environment again.

7.3.3 Results

The Pong agent was very successful, it managed to get 44 wins in 22800 games, however after 22800 games the agent became unresponsive and a memory leak caused the agent to remain in the 'NoOperation' action for another 3206 games before it was noticed. The agent preceded to score 0, for all those games. This can be shown from Figures 7.3 and 7.4 below:

```
1  plt.plot(episode_array, score_array, 'ro')
2  plt.axis([1, 22800, -21, 21])
3  plt.show()
```

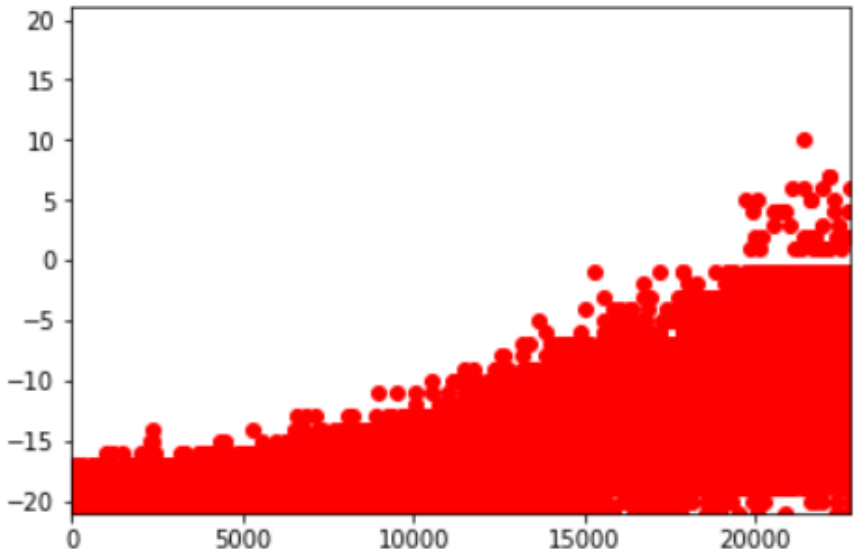

Figure 7.3: The Pong agent after 22800 games.

```
1  print(len(score_array))
```

```
26006
```

```
1  plt.plot(episode_array, score_array, 'bo')
2  plt.axis([1, 26006, -21, 21])
3  plt.show()
```

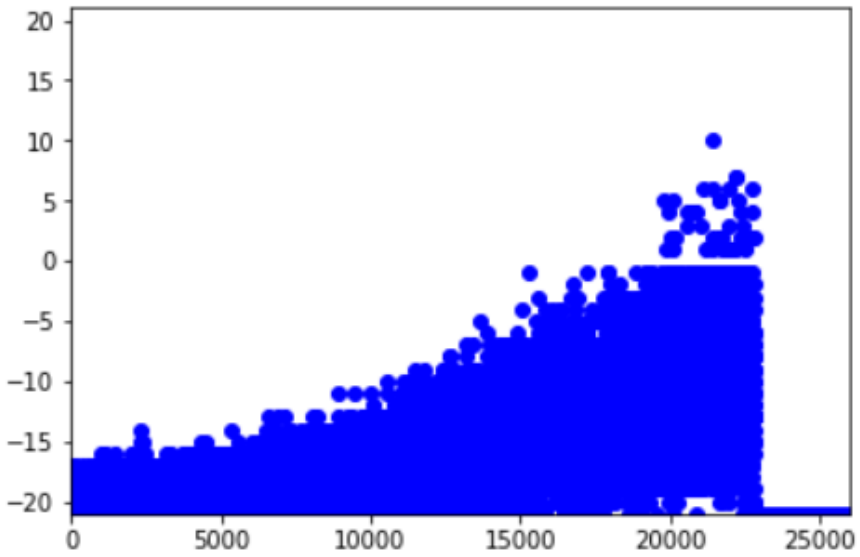

Figure 7.4: The Pong agent after 26006 games.

As shown by the Figures, 7.3 and 7.4, the agent was learning at an increasing rate, as more games were played the score got increasingly higher. From Figure 7.4, it is plain to see that the agent was unresponsive. The line through both graphs on the score = 0 line marks a boundary. The games above the 0 line are the games that the agent won against the Atari agent. The number of games won was 44, with the maximum score being 10. The score of 10 means the difference between the agent and the Atari agent, where the agent scored 10 more points than the Atari, beating it 21 to 11. The index of the maximum score of 10 had occurred at game number 21428, when it beat the Atari agent.

The curve of the graph showed exponential growth of score as more games were played, the agent was demonstrating better moves more and more often. The running mean score of the agent began at -21 and had improved to -10.654563 before the agent had run out of memory.

7.3.4 Conclusions

To conclude the Pong agent, from Figure 7.3, it seemed to be getting better at choosing the best action given its current state more and more often than previous attempts. This exponential rise picking the better move meant the agent was becoming 'more intelligent' with choice of actions. It should also be noted that in Figure 7.3 the number of complete losses (scoring no points) the agent experienced started to decrease. For games 21000 to 22000 there was no complete losses made by the agent and it had scored at least 1 point for roughly 1000 games. The gap at the bottom of the graph signifies that the agent was making less critical mistakes and losing less games without scoring.

The agent running out of memory meant that another simulation would not fit in with the Gantt chart as this agent had already been running for 12 days and would cause the project to slip behind schedule. The decision was made to leave the Pong agent alone since the graph had proved to show some success in the learning algorithm applied to the Pong environment.

The original success criteria for Pong had thus failed and it meant that testing the agent against the DeepMind average of 18.9[6] could not be a possibility in this projects time frame. Though local success of an increased average and a graph showing exponential growth did allow the project to move on.

7.4 Breakout

7.4.1 Initial Plan

The original concept of the Pong algorithm[12] gave a naivety to the ease for transferring the algorithm to the Breakout environment. Whilst the algorithm remained relatively similar to the simple trial and error approach to integrating the Pong algorithm to the breakout environment brought about countless errors and frustration at something that should have been, on the surface, 'easy'.

The Breakout environment as seen in Figure 3.1 on page 1 shows a somewhat similar environment to the Pong environment, where the agent is essentially bouncing a ball back and trying not to let it go behind the agent otherwise the agent will be losing 'lives'. The Breakout agent only has five lives and when the lives run out, the game is reset and the agent tries again. The Pong agent however has 21 lives and then they run out the environment is reset, the Pong games are a lot longer than Breakout as a result.

7.4.2 Implementation

The initial test for the use of the new environment on the original algorithm given by Raval[12] had been specifically designed for the Pong environment and so adjustments had to be made in order to

work with the Breakout environment.

Upon seeing the unchanged integration of the algorithm, immediately the issue was clear, as shown in Figure 7.5:

Figure 7.5: The Breakout Environment with no 'Fire' action.

The agent was moving left and right continuously without any ball, upon listing all the available actions for the Breakout game, the 'Fire' button was used to spawn the ball in to the environment. Implementing the 'Fire' action became a hard task since the probability for actions was based upon the action being true or false. The 'Fire' action, to release the ball, was implemented after every frame that had passed, this way the environment remained a decision of left or right like the Pong environment and it also, the 'Fire' action had no effect unless used when the ball had been reset and so implementing it after every frame was doing nothing in frames it should not be used in.

The 'Fire' action had to happen at the start of every new frame since you do not want to move unless there is a ball as you do not know which direction it will spawn. The addition of a new step as shown from code 7.6:

$$\begin{aligned} &env.render() \\ &observation, reward, done, info = env.step(1) \\ &reward\_sum+ = reward \end{aligned} \tag{7.6}$$

this code 7.6, means that at the start of every frame, the environment is created with *render()* and the values are extracted by *step(1)* where the 1 corresponds to the 'Fire' action. The step() releases the ball and the reward for releasing the ball is added even though there is no reward for completing this action.

The next implementation error came from the normalisation of the discounted reward, the discounted reward was being exploded towards infinity and vanished towards 0. Debugging revealed there was

an error in the divide calculation for standard deviation. The error was caused by the new action and observation being implemented into the reward matrix when there was a reward of 0, since this was always being implemented and you can not divide by 0. To get around this error, the mean and standard deviation were removed and instead used the *sklearn.preprocessing import normalize* package to normalise the discounted reward which fixed the issue.

Another issue foreseen and not tested this time was in preprocessing, in the Pong implementation, the image is squashed and de-colourised. The Breakout implementation needed the colours as they had meaning, in the Breakout environment the different colours are represented by score shown in Table 7.2 below;

Breakout Brick Colour Scores	
Brick Colour	**Score**
Blue	1
Green	1
Yellow	4
Dark Orange	4
Redish Orange	7
Red	7

Table 7.2: Breakout brick colour scoring upon breaking.

this scoring differs from the original version of Breakout[22]. The de-colourisation had to be removed for allowing the agent to distinguish colour. The less preprocessing done by the function means that it will take longer to go through the network and may take up more memory and processing power at runtime than Pong.

Another adaptation that was made for the Breakout version of the Pong algorithm was that the Breakout agent used a local mean (mean of the past 100 games) instead of a mean of the entire history of the agents performance, like Pong. The breakout version of the algorithm removed this lengthy calculation to compromise for the extra processing for the frames in Breakout.

The Breakout version maintained a vast similarity with the exception of the few additions to code, the idea to create a general agent for the Atari library was missed slightly due to the changes that had to be made to the code. Upon execution of the code, the script threw system errors and some signing with the reward had to be amended in order to run, this signing was related to the discounted reward and was quickly resolved.

7.4.3 Results

The results for breakout were somewhat underwhelming, as shown by Figure 7.6 and Figure 7.7:

```
1 plt.plot(episode_array, score_array, 'bo')
2 plt.axis([1, 123185, 0, 16])
3 plt.show()
```

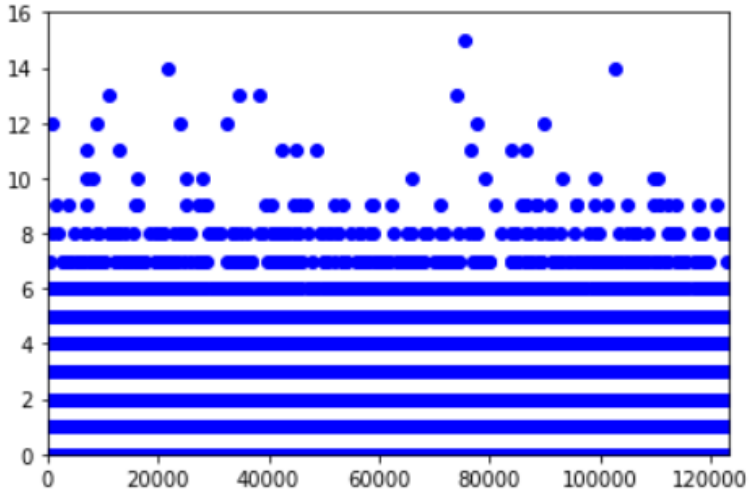

Figure 7.6: The first run of Breakout with the preprocessing cutting out a lot of the input (x-axis - Episode Number, y-axis - Game Score).

From Figure 7.6, we can see that the agent was occasionally scoring highly on the task however this graph is almost meaningless since the environment the agent was looking at was cropped and de-colourised, this meant it was effectively a waste of time. The graph does still show that whilst the agent may have been getting a high score on occasion, it is able to get this score even when it is wrongly allocating weights and values to the model. This graph is useful later in the conclusion section, 7.4.4, of this chapter on page 28.

The agent from Figure 7.6 ran for 123185 games before it was stopped when the mistake was realised. The agent had not completed the board or got a significant score in that time. The maximum score the agent had achieved was 15 of which it only achieved once, considering the incorrect preprocessing, the agent could seemingly manage to rebound the ball. From Figure 7.6, there is no visible trend from the graph and it is seemingly random. Figure 7.6 shows the amount of low scoring games did not improve, neither did the amount of high scoring games.

Figure 7.7 shows that even with the correct preprocessing technique, the agent was unable to score any higher than 15, with no real trend in the graph, it is hard to say if this agent was an improvement from the previous agent based on the graph.

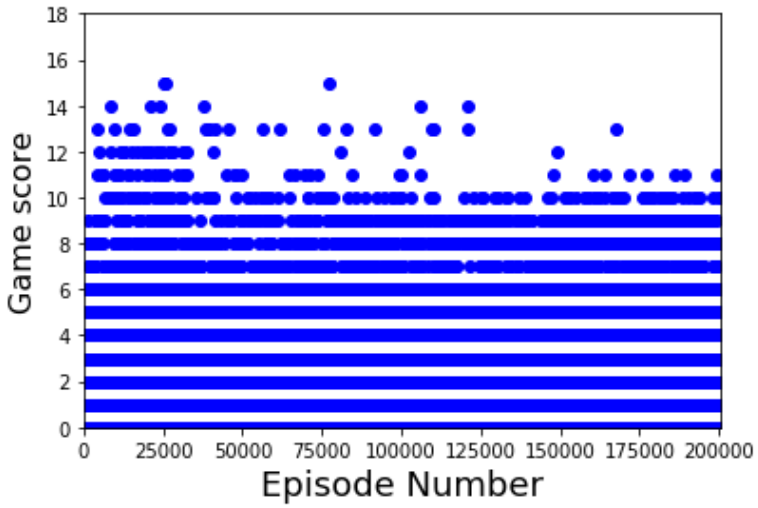

Figure 7.7: The finalised Breakout test Graph with all the discussed changes from section 7.4.2 on page 24.

7.4.4 Conclusions

The main conclusion from the Breakout tests were that the agent did not perform as expected, graphically. In the code, a tracker for the average score across the recent Breakout games was kept. The average started at roughly 1 point per 5 lives, the average after 200383 games was 2.65 points per 5 lives, whilst the increase in performance is there in terms of numerical representation, the actual result of a 1.65 score increase over 200383 games is rather insignificant.

The agent managed to get an average of 2.51449797025 score over the entirety of the playthroughs. The maximum score of 15 was first achieved in game 25394. From Figure 7.7, the agent appears to have gotten worse in the last 50000 games.

The Atari environment is not known to the agent like the CartPole agent, the Atari agents are fed raw pixel data and the CartPole agent uses metrics to adjust the movements of the cart, allowing for Q-values to dictate the action taken at any given time. The Atari agents use the RMSProp optimiser due to its benefits of image processing and the CartPole agent uses Adam since it benefits that environment[12,21].

8. Challenges and Discussion

8.1 Environment Challenges

The environment for Breakout was initially assumed to be the same as Pong, the naivety of this approach led the project to fall behind in time. On the Gantt chart in section 5.2 on page 7, the delay this caused is shown by weeks, 19 to 23, the original task was projected to take 2 weeks, yet instead it took 4 weeks and overran by 4 weeks since the start of the task was a week later than projected also.

Finding ways to set up the Atari environments using a Windows 10 machine proved to be more difficult than imagined. The original plan was to use a virtual machine and use the power of the machine to run algorithms, when the rendering of the environment failed to SSH to my local machine, a new approach to run the agent on Windows 10 was taken. Installing all the packages on to the Ubuntu application from the Windows 10 store was easy but took a long time, a couple days and compatibility errors with Python meant that it had to be restarted. Originally Python 3 was used for the virtual machines yet not all Python 3 compatible packages worked with Windows 10 and so came the eventuality of reverting to Python 2.7.15rc1.

8.2 Technical Challenges

During the course of the process, there were many hurdles that had not been factored into the original risk assessment. During the initial research and trialing out the Jupyter Notebook, the machine being used for the running of algorithms had a fatal error and had to be sent away to be fixed. Time on the project was pushed to the side whilst the computer was being repaired. The Atari environment is marketed only for Linux[16] and this initially dissuaded the use of Windows and plans were drawn up to use virtual machines on cloud services. The cloud service VMs fell through and the project was set back again.

8.3 Personal Development

The project was not immediately accessible, Python was a new programming language and the topic of neural networks and reinforcement learning were not understood well enough to know how both concepts work. There was no background of using notebooks such as Jupyter and skills were slow to develop through self-teaching of most concepts. A large amount of time went into learning what networks are and how they are composed, watching YouTube videos[1,12] and getting detailed explanations of what key concepts are helped with the learning but since the project was already underway the time spent learning was again not factored into the time frame of the project. Through the use of sprints and Trello, the project was able to move around and blocks of work could be later

brought into the picture once the theory and technical knowledge was acquired.

8.4 Algorithm Success

The algorithm that was used for Pong and Breakout had varied success, the original objectives for success were to mimic that of the DeepMind paper[6] and neither agents had proved successful in either environment. The Pong agent was more successful than the Breakout agent, as shown by Figures 7.3 and 7.7 on pages, 22 and 28 respectively. The Pong environment is a success, since the algorithm worked until the machine hosting the algorithm ran out of memory as shown by Figure 8.1, the machine was slow and eventually stopped.

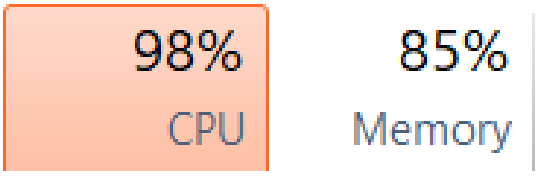

Figure 8.1: CPU usage and memory usage of the machine when the Pong algorithm was running.

The Pong algorithm was successful however in learning how to play and beat an opponent (44 total times) at the game. The fact that the Pong agent went from moving randomly to successfully hitting the ball back and scoring points meant that the algorithm could be continued and assumed that it would eventually reach the level of DeepMind average of 18.9 score[6]. The algorithm only managed an average of -10, but from the trend of the graph in Figure 7.3 it looks to have been on track to reach that goal of 18.9 in some amount of time.

8.4.1 Improvements to the Algorithm

To make improvements to the algorithm, the obvious improvement is to allocate more memory to the algorithm so that it can get to a stage where the score is comparable to the DeepMind score. A suggestion for this is to use a cloud service and get around the rendering issue with not showing the game but just showing the metrics that the game is returning. A more powerful machine would not only allow the algorithm to run longer but also much more efficiently, when the algorithm was running, the render of the game through Xming was ‘human’ pace and thus games were slow to progress, especially with Pong where games could last for 41 points.

A further improvement to the algorithm could be to remove the low level configuration and use a higher level model such as Keras, this would make adding more layers and configuring network structure far easier. Keras was used in the CartPole example and used to define the network structure and number of layers the network had, the algorithm used for the Atari games did this but in a more complex way.

8.4.2 Hyperparameters of the Algorithms

The hyperparameters for both of the algorithms were tried and tested[1,6,12]. During the project some hyperparameter optimisation was attempted as shown by table 7.1 on page 16, however the parameters were already optimal. The hyperparameters used in most algorithms are widely known to be optimal for the algorithms that they are used in, changing them usually has a negative impact as shown from the table.

8.5 Project Success

The overall goal of the project was to look at different algorithms and look at how different environments can affect the behaviour of algorithms, this paper does achieve this goal with mild success. The two algorithms that were looked at were very different and the CartPole algorithm had a higher level of abstraction, the API, Keras was useful for quickly adding together a network and configuring a lot of the background tasks for the user. Whereas the Atari algorithm used different optimisers and had a lower level of abstraction. The Atari algorithm used RMSProp and the CartPole algorithm used the Adam optimiser, since the two environments have different requirements, the same optimiser for both would be inappropriate.

The project will be useful to the wider research and scope of reinforcement learning to help provide information on why certain environmental differences affect the behaviour of agents.

The 8th objective for this project was not able to be achieved within the time due to push backs of a large scale discussed earlier, Asteroids is a very complex environment, 18 actions possible for any given state, and would have been very challenging to implement and adapt the algorithm to get it to work within that environment. DeepMind found little success in Asteroids, managing an average score of only 1629 compared to their expert human game tester at 13157[6]. Due to aforementioned setbacks and complications, the testing of Asteroids was abandoned.

8.6 Further Study and Considerations

To continue the study, the algorithm would have to be generalised to the other Atari environments and compare it to the use of different optimisers and parameters that may provide benefits within the more complex environment.

A further study of interest is if the memory can be shared between games, the reason for choosing the games of Pong and Breakout were not only based upon naivety of similar environments but that the games have similar player rules. Both in the Breakout and Pong the player or agent must rebound a ball to score points, it may be possible to use the knowledge of the Pong environment to get a ‘head start’ at Breakout. If the preprocessing for both environments removed half of the screen then the

games are almost identical. There does not seem to be anything on the topic for this and especially in the topic of Atari. This is of interest as humans do not begin any task completely ‘in-the-dark’ and upon coming across a new challenge, previous knowledge about a different situation or environment is applied to help overcome the problem.

8.7 Conclusion

Reinforcement learning is the newest branch of machine learning[7], and with such it has the most potential to affect different areas of the world. Using reinforcement learning effectively to get better at tasks such as speech recognition and pattern recognition can help people make use of data that maybe otherwise not as useful like audio data for phones may not have been used before but now it can be put into databases and used for speech analysis and recognition for smart home applications like Amazon Alexa[24].

The ability for a reinforcement learning algorithm to successfully play games and beat humans in every game but learn from each game and take knowledge that has been learned from a game and apply it in successful ways to other environments may just be the last algorithm needed for playing games.

9. References

[1] John Krohn, (2018). *Deep Q Learning Networks*, Youtube: LiveLessons.
Online Video. Available online at: <https://www.youtube.com/watch?v=OYhFoMySoVs> 0:00 - 1:12:14
(Consulted 30 December 2018)

[2] Lin, J. Ph.D., (2017). *Keras Models: Sequential vs. Functional* Available online at: <https://jovianlin.io/keras-models-sequential-vs-functional/> (Consulted 5 January 2019)

[3] Francis, J., (2017). *Introduction to reinforcement learning and OpenAI Gym* Available online at: <https://www.oreilly.com/learning/introduction-to-reinforcement-learning-and-openai-gym> (Consulted 15 January 2019)

[4] Talitha Trippel, (2017). *Install OpenAI Gym Atari on Windows*
Online Video. YouTube. Available online at: <https://www.youtube.com/watch?v=HbMhdxd3qMQt=352s 0:00 - 11:39>
(Consulted 19 February 2019)

[5] Geitgey, A., (2016). *Machine Learning is Fun Part 6: How to do Speech Recognition with Deep Learning*
Available online at: <https://medium.com/@ageitgey/machine-learning-is-fun-part-6-how-to-do-speech-recognition-with-deep-learning-28293c162f7a>
(Consulted 17 March 2019)

[6] Demis Hassabis et al., (2015) *Human-level control through deep reinforcement learning* Deep-Mind - February 2015
Available online at: <https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf>
(Consulted 27 December 2018)

[7] Chollet, F (2017) *Deep Learning with Python* Chapter 4, p95. Manning Publications (30 Nov. 2017) New York, United States of America.

[8] LeCun, Y., Bottou, L., Bengio, Y. Haffner, P. *Gradient-based learning applied to document recognition*. p6. Proc. IEEE 86, 2278–2324 (1998).

[9] Schmidhuber, J., 2014-2015 *Deep learning in neural networks:An overview* p93 University of Lugano SUPSI, Galleria 2,6928 Manno-Lugano, Switzerland. Published by Elsevier Ltd in 2014.

[10] Wikipedia 2019, updated 21 April 2019,
Available online at: <https://en.wikipedia.org/wiki/Convolutional_neural_network>
(Consulted 27 April 2019)

[11] Clark, C., Storkey, A., (2015) *Teaching Deep Convolutional Neural Networks to Play Go*. p2. University of Edinburgh, Scotland (2015) Accessible online at: <https://arxiv.org/pdf/1412.3409.pdf>
(Consulted 19 March 2019)

[12] Siraj Raval, (2017) *How to Beat Pong Using Policy Gradients (LIVE)* Online Video, YouTube, Available online at: <https://www.youtube.com/watch?v=PDbXPBwOavc> (Consulted 30 December 2018)

[13] Heller, M., (2019) *What is Keras? The deep neural network API explained* InfoWorld 2019,

Available online at: <https://www.infoworld.com/article/3336192/what-is-keras-the-deep-neural-network-api-explained.html> (Consulted 20 March 2019)
[14] Kingma, D., Ba, J., (2015) *Adam: A Method for Stochastic Optimization*, 3rd International Conference for Learning Representations, San Diego, 2015. Available online at: <https://arxiv.org/abs/1412.6980> (Consulted 19 March 2019)
[15] The Scipy community. (2017) *What is NumPy?*, Enthought 2017, Available online at: <https://docs.scipy.org/doc/numpy-1.13.0/user/whatisnumpy.html> (Consulted 23 March 2019)
[16] OpenAI, (2019) *Getting Started with Gym* OpenAI 2016, Available online at: <http://gym.openai.com/docs/> (Consulted 23 March 2019)
[17] Zulkifli, H., (2018) *Understanding Learning Rates and How It Improves Performance in Deep Learning*, Towards Data Science (2018), Available online at: <https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10> (Consulted 24 March 2019)
[18] Karpathy, A. Ph.D., (2015) *CS231n Convolutional Neural Networks for Visual Recognition*, Stanford University, United States of America (2015), Available online at: <http://cs231n.github.io/neural-networks-3/> (Consulted 24 March 2019)
[19] Hui, J., (2018) *RL—Policy Gradient Explained* Medium Coorperation, United States of America (2018), Available online at: <https://medium.com/@jonathan_hui/rl-policy-gradients-explained-9b13b688b146> (Consulted 25 March 2019)
[20] Veromann, V., (2019) *RAG+B Traffic Light Rating System* weblog post, Available online at: <https://blog.weekdone.com/ragb-traffic-light-rating-system-expanding-established-design-patterns/> (Consulted 30 March 2019)
[21] Gandhi, R., (2018) *A Look at Gradient Descent and RMSProp Optimizers*, Towards Data Science, 2018, Available online at: <https://towardsdatascience.com/a-look-at-gradient-descent-and-rmsprop-optimizers-f77d483ef08b> (Consulted 2 April 2019)
[22] Wikipedia, *Breakout (video game)*, wiki article, updated 6 April 2019, Available online at: <https://en.wikipedia.org/wiki/Breakout_(video_game)> (Consulted 20 April 2019)
[23] Shen, H., (2014) *Interactive notebooks: Sharing the code*, Nature, 2014 Macmillan Publishers Limited, Available online at: <https://www.nature.com/news/interactive-notebooks-sharing-the-code-1.16261> (Consulted 21 April 2019)
[24] Wikipedia, *Amazon Alexa*, wiki article, updated 25 April 2019, Available online at: <https://en.wikipedia.org/wiki/Amazon_Alexa> (Consulted 25 April 2019)
[25] Wikipedia, *Q-learning*, wiki article, updated 28 April 2019, Available online at: <https://en.wikipedia.org/wiki/Q-learning> (Consulted 29 April 2019)