Plymouth University

School of Computing, Electronics and Mathematics

PRCO304

Final Stage Computing Project

2017/2018

BSc (Hons) Computer Science

Christopher Haynes

10542097

Deep Reinforcement Learning on the Atari 2600

Acknowledgments

First, I would like to acknowledge the advice provided by my supervisor Martin Beck throughout this project. This advice has often proved invaluable and provided me with a range of relevant material which I would have not discovered otherwise. I would also like to acknowledge some of the developers of the “Arcade Learning Environment”, Nicolai Czempin and Marc Bellemare, who happily provided me with details on the inner workings of the emulator and assisted me with debugging and defining the scope of my project. Finally, I would like to thank Nikita Kniazev, who created a Windows branch of the python “Arcade Learning Environment (Atari-Py)” which would work with “AI Gym”.

Abstract

This report examines how to approach solving complex, high-dimension control problems using reinforcement learning techniques. An investigation is carried out into the structure and purpose of Deep Q Networks, and other reinforcement learning algorithms. This leads into the design and development of a reinforcement learning agent which can successfully learn to play an Atari 2600, using only the screen’s pixel values as an input.

The agent was tested across a series of different games, with a range of different parameters, and compared to other similar academic agents. The developed agent was able to equal or surpass the ability of other similar reinforcement algorithms, and achieved superhuman performance in some games.

To try to increase the agent performance, two novel techniques were developed and tested with the results being written up in an academic paper. Although the maximum learning ability was not improved, some interesting behaviours were observed.

Table of Contents

Word Count = 10,672

Code URL = https://liveplymouthac-my.sharepoint.com/:f:/g/personal/christopher_haynes_students_plymouth_ac_uk/EoPn2aKbhbRNmN8CXLuIwKMBunbKkqZefoI3WVorew2nEA?e=OynUU2

Chapter 1: Introduction

The use of artificial intelligence (AI) is becoming more common across an expanding number of areas. Some popular, current examples are self-driving cars, face recognition technology and video analysis. All these technologies are now using a system with the ability to learn complex human behaviour and imitate it accurately. The current challenge for AI technologies is to find ways of improving their overall accuracy, whilst reducing the amount of time it takes for accurate behaviour to be learnt (referred to as training).

There are many areas in the field of machine learning which all take different approaches to the behaviour and operation of AIs. Supervised learning focuses on creating systems which can classify data based on information they have gained from historic, existing classified data. Unsupervised learning focuses on grouping data based on similarities without knowing any underlying information about individual data elements. This project aims to focus on reinforcement learning (RL) which uses AIs to choose actions based on the state of their environment and rewards they previously received for performing desired behaviour.

1.1 Reinforcement Learning

Reinforcement learning uses Markov decision processes (MDPs) to describe the environment. A MDP is a mathematical framework which can be used to model decision making. The main feature is that any given state of the system in an MDP contains all the relevant information from the history of prior states. This allows for actions to be chosen based on the current state without having to remember the history of prior states.

The three main components of reinforcement learning are states, actions and rewards. The environment, at any given time, is in a unique MDP state. The AI agent can then choose an action which causes the current state to transition into another state. If a certain action is taken in a certain state then the agent is presented a reward to either positively or negatively reinforce the actions chosen to lead to this state. A simple example of this model can be seen in a grid world;

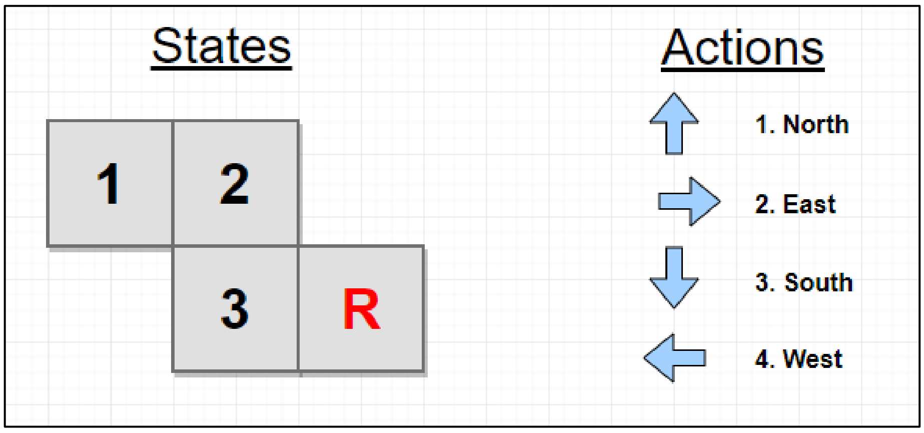


Figure 1: Grid world states, and available agent actions

First, assume that this is a deterministic environment (If action 3, move south, is chosen whilst the environment is in state 2 then the environment will always transition to state 3).

The agent will start at a random state, excluding the reward state "R". The agent will then choose an action based on the current state. If the action is not viable (e.g. the agent tries to move east from state 2), then the state remains the same after the action. Otherwise the environment will transition to the new state. If action 2 is chosen in state 3, then the reward is presented to the agent. This shows the agent that choosing action 2 in state 3 is the desired behaviour. The remaining challenge is distributing a discounted part of that reward to the surrounding states so that the agent can follow the trail of increasing rewards to the reward producing state.

One common approach to this is Q Learning, which uses a Q table of probabilities, known as q-values, to choose an action for a given state;

	Action 1, North	Action 2, East	Action 3, South	Action 4, West
State 1	0.1	0.6	0.2	0.1
State 2	0.1	0.1	0.5	0.3
State 3	0.3	0.5	0.1	0.1

Table 1: Example Q Learning Table

As can be seen from this example table, for each state-action pair there is a probability that the agent will select a particular action. The agent can either select the action probabilistically or can use a different selection policy such as "greedy" selection where only the action with the highest probability is chosen.

This table can be updated iteratively based on the reward received and using the Bellman equation;

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[reward + \gamma \max_{a} Q(sNext) - Q(s,a)\right]$$

Where;

- **s** is the current state of the environment.
- **sNext** is the state being transitioned to from the current state by the chosen action.
- **a** is the action selected by the agent.
- **Q(s, a)** is the q-value in the Q table based on the current state and chosen action
- **reward** is the reward presented for the state, action combination.
- **α** is the learning rate to make sure than any changes to the current q-value are only fractions of the calculated change.
- **γ** is the discount rate which limits the amount of the reward which can get passed to subsequent states. This value ensures that the reward from the goal state is spread out across the other states, decreasing the further a given state/action combination is from reaching the goal state.
- $\max_{a} Q\ (\mathbf{sNext})$ is the highest q-value from all the possible actions for the next state.

A single pass through the environment by the agent is referred to as an episode. An episode starts with a random starting state being selected, and ends when the agent reaches the goal state "R". Over multiple episodes the performance of the agent will increase as the Q table starts to more accurately represent the best actions to reach the goal state.

An episode could be described algorithmically as;

- Select a random starting state
- Whilst the agent has not reached the goal state:
    - Agent selects an action from the Q table using selected policy
    - The next state is determined from the current state and chosen action
    - Any reward for the current state and chosen action is recorded
    - The Q table is updated using the Bellman equation
    - The current state transitions to the next state

One area of reinforcement learning which is important to this project is partially observable Markov decision processes (POMDPs). With a POMDP, the agent is not able to directly determine current state, instead the agent must estimate the state based on observations which can be made about the environment.

For example, consider the grid world from figure 1. Rather than knowing the state of each square, the agent is only able to observe what "type" of square it is;

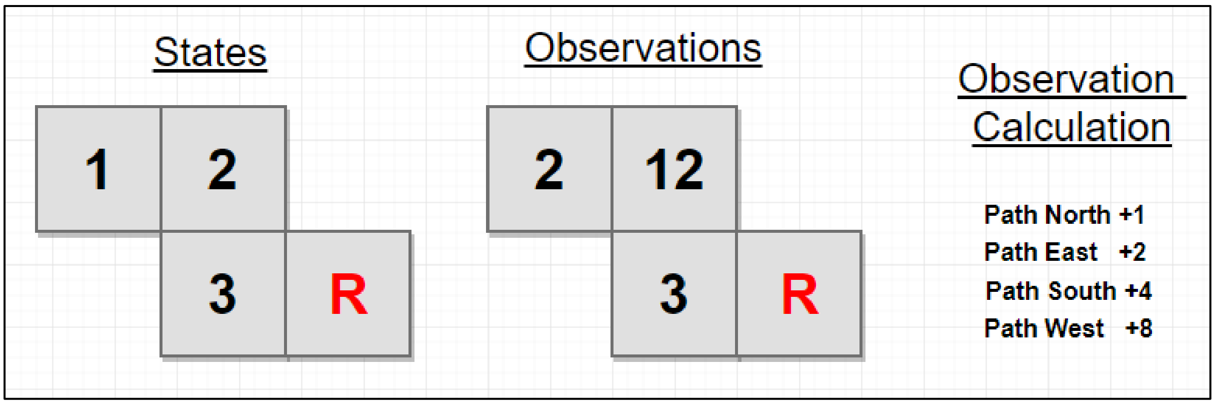


Figure 2: Grid world states and observations in a POMDP model

Here the agent uses an observation value to try to estimate the current state of the environment. Each observation value here is based on what viable "paths" lead out of the current state. For example, state 1 only has a viable path east so is given an observation value of 2, whereas state 2 has two viable paths, south and west, so it is given an observation value of 8 plus 4.

For this simple example world, the agent would perform as well using observations in the place of states. However, consider a world where there are multiple states that produce the same observation. For the agent to transverse that world successfully it would require some memory of previous transitions to better estimate the state. Ideally, the observation model would have enough short-term memory to remember enough transitions that it can produce a unique observation for each state. This is difficult to achieve in practice particularly for problems with a large or complex state space.

1.2 Accuracy Improvement Techniques

There are many techniques to increase the accuracy of action selection, a few are;

- Prioritising from which experiences the agent learns.
- Splitting the Q value into two parts, one part representing the value of the current state and the other part representing the advantage for taking the chosen action.

An additional approach to improving a reinforcement agent, or policy is self-play. This trains the AI by having it compete against itself either fictitiously as examined by Heinrich (2016) or directly as used by Silver et al (2017).

This technique is still relatively new and has not been fully explored. However, some advantages of using this system have been discovered. Weiring (2010) discusses that training with self-play produces a more competitive agent than training from using the observations of an expert. Silver et al (2017) also noticed this trend, and their results showed that a self-play reinforcement model can achieve a higher ranking in Go than any AI trained from existing domain knowledge (in this case, previous human moves and strategies).

1.3 Project Overview

This project aims to create a RL agent using a Deep Q Network to estimate the Q values. This agent will then be trained to play a range of Atari 2600 games using only a matrix of the screens pixels as the input, allowing the agent to learn "visually" much like a human player would. Once a stable agent has been developed, improvements will be applied and tested to increase the overall accuracy and reduce the required training time. Finally, a new novel technique will be implemented and experimental results will be produced to assess the effectiveness of this technique. The full project objectives and requirements are listed in Appendix C.

This report begins with a literature review, examining existing techniques and practices used within deep reinforcement learning. This is followed by the necessary planning and design of the agent, and descriptions of the specific implementations used in this project. It then describes the experiments carried out and concludes with the experimental results and an evaluation of the project.

Chapter 2: Literature Review

There are two areas that require further research in order make this project viable. The first is how to store the q-values or action selection probabilities, as for a problem of this scale a Q table would be too large to be useable. The other area is FSP which requires some way of storing previous actions for given states as well as the result that was achieved for that selection.

A common alternative to the Q table is Deep Q Learning, which uses a convolutional neural network to estimate Q values. As an input, the network takes the matrix of pixel values and as an output it provides a range of Q value estimates for each of the possible actions.

The way that Silver et al (2017) achieved FSP was using a Monte Carlo tree search. This is an algorithm which focuses on exploring the most promising moves. It records the total number of times a given node is visited and the number of times that visiting that node lead to a "victory". As more moves are made the existing nodes in the tree are updated and more leaves are added as more alternative paths are explored. By navigating through this tree from the root node to a leaf node, further information can be ascertained on the likelihood of a chosen action leading to an overall "victory".

By performing further research into these two areas, it should provide enough information to allow the design of the AI to begin.

2.1 Deep Q Learning

As mentioned above, using a Q table becomes limiting for more complex state spaces, as the amount of stored probabilities will be the product of the number of states and number of actions (assuming each action is viable for each state). Mnih et al (2015) describes this issue in his introduction to Deep Q Learning; *"While reinforcement learning agents have achieved some successes in a variety of domains, their applicability has previously been limited to domains in which useful features can be handcrafted, or to domains with fully observed, low-dimensional state spaces."*

Mnih et al (2015) also discusses the pitfalls of using a non-linear function approximator, such as a neural network, to represent the q-values. They state that it can cause unstable or divergent behaviour during training, which is due to two factors;

- Correlations present in the sequence of observations.
- Small updates to the neural network, during training, causing large changes in the policy and therefore the data distribution.

To combat this Mnih et al (2015) recorded a series of the agent's experiences, into a structure they called "experience replay". Each experience consisted of a state, action, reward and resultant state. During Q learning, batches of experiences were drawn uniformly at random from the experience replay, and the network updates were performed using this batch. This removes any correlations from consecutive observations and helps to ensure that the network is learning the entire problem domain instead of falling into local minima.

To main issue with training the deep Q network is that to derive the sum squared error a recorded result and a target result are required. This means that the same network is calculating a result and estimating a target. Therefore, as the network is updated it also moves the bounds of how it estimates the target. This can cause feedback loops and stop the network from improving its accuracy. The most common solution to this, applied by Mnih et al (2015) and Silver et al (2015), is to use two neural networks of the same architecture. One network behaving as the value network and estimating the Q value, and the other network becoming the target network, with the sole responsibility of estimating target values. To keep these networks similar but not identical, the target network is either updated periodically to match the value network, or slowly adjusted to towards the value network at each step.

An alternative approach to solving the unstable behaviour issue was suggested by Riedmiller (2005). His approach focuses on repeated training of the network over multiple iterations. This solution was impractical for larger networks as the training time would not be viable.

A similar structure for the neural network was used by Silver et al (2015), Heinrich (2016) and Mnih et al (2015). A convolutional neural network was used with a matrix of the screen pixels as the input. The output was a vector of probabilities relating to each possible input that could be given to the Atari.

The exact structure of the networks varied slightly between different authors, although there were some common factors;

- All the networks used at least two convolutional layers which were followed by a rectifier nonlinearity.
- All the networks used at least two fully connected layers for classification

Mnih et al (2015) did not use any pooling layers, as this would reduce the spatial relation of the input. For the agent to learn from the input, it needs to be aware of the spatial relationship between different on-screen objects. For example, in breakout, the distance between the paddle and the ball is an important factor in the agent being able to estimate the current state.

The Atari output is 210 x 160 x 3 RGB pixel values featuring 128 colours. This input was scaled in all implementations investigated. Silver et al (2015) reduces a frame to a greyscale image (210 x 160) and then down samples and crops until an area of 84 x 84 remains. A single state is then composed of four, consecutive, pre-processed frames stacked in a 3D matrix (84 x 84 x 4). This reduces the dimensionality of the input to the Q network, which reduces the processing time required. It also allows for a single state to contain the concept of “movement” which allows for more complex behaviours to be identified by the agent.

The structure of their network can be illustrated as;

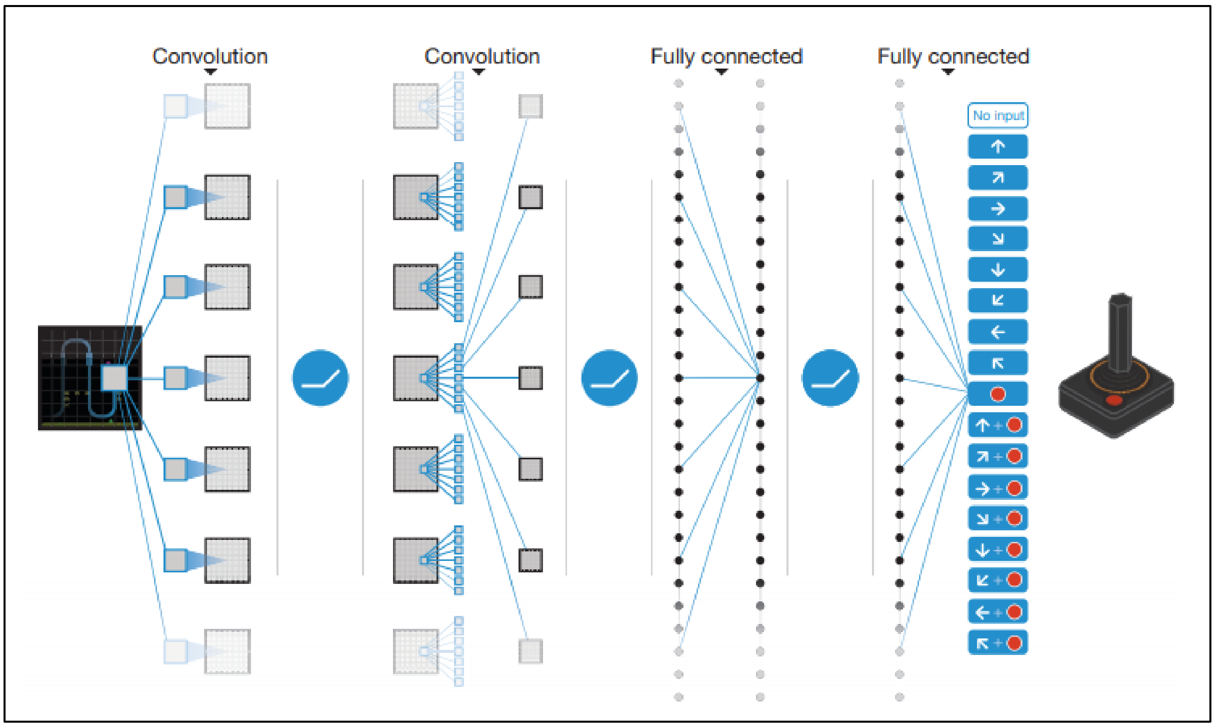


Figure 3: Schematic of convolutional neural network (Mnih 2015, p530)

The update of this network uses a variation of the Bellman equation, and trains in batches of 32 experiences, sampled at random from the experience replay.

2.2 Monte Carlo Tree Search

The concept of FSP is for the agent to simulate the result of the game from the current state multiple times to better adapt the policy for action selection. Silver et al (2017) uses a Monte Carlo tree search (MCTS) to create and expand a tree which contains the probabilities of given states leading to winning the game.

The structure of the tree consists of nodes representing the states, connected by edges which represent actions. An empty tree can be populated randomly, by selecting actions until the game reaches conclusion. At this point the winner of the game is decided and the path taken through the tree can be back-tracked to record the result in any visited node. Each node stores the number of times it has been visited as well as how many times traversing the node led to an expected win.

Yannakakis and Togelius (2018) discuss the advantage of MCTS because it focuses on expanding the most promising nodes rather than minimising the depth of the tree. This leads to the exploration of more paths which are likely to lead to an expected win.

The operation of a MCTS is described by Yannakakis and Togelius (2018) as consisting of four steps, selection, expansion, simulation and back propagation;

- **Selection** – A node is chosen for expansion. Starting at the root of the tree traversal occurs using an action selection policy such as greedy epsilon or UCB1. UCB1 considers more variables than greedy epsilon such as number of times nodes are visited and average reward of all successive child nodes.
- **Expansion** – Once a node is reached which has unexpanded children (a state of which some actions have not be attempted), one of the unexpanded children is chosen for expansion at random. Expansion consists of creating a new child node and connecting it via an edge to the expanding node.
- **Simulation** – The new expanded node then performs a simulation of the rest of the game to completion, this is often called a rollout or playout. As there is no domain knowledge for this section of the tree, actions are usually selected at random until the game reaches completion.
- **Backpropagation** – The reward is determined by the simulated result of the game and is passed back up the tree to each node which was visited during the current traversal until the root node is reached.

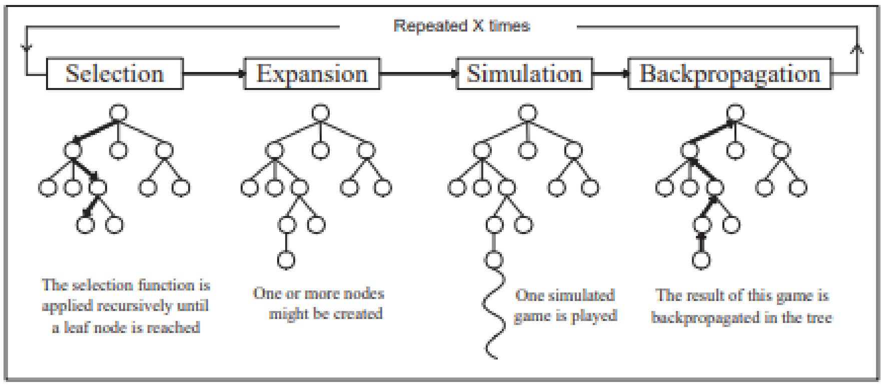

Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until a leaf node is reached
One or more nodes might be created
One simulated game is played
The result of this game is backpropagated in the tree

Figure 4: Scheme of MCTS (Chaslot 2007, p3)

The added value from the MCTS proved invaluable in allowing Silver et al to reach superhuman performance in the game of Go. However, compared to the Atari 2600, the state space of Go is very low. It may not be viable in terms of space or computation to perform a MCTS for this project. It may also be technically difficult to perform the simulation step, depending on how the Atari 2600 emulation is handled.

Chapter 3: Player Agent Development

3.1 Project Management

3.1.1 Project Plan Adaptation

Due to the specific and complex nature of this project, the project plan has been adapted and improved since the initial plan mentioned in the project initiation document (PID). Originally this project was going to focus on the aspect of training agents against one another, particularly in the game “Warlords”. However, during the early research stages, several barriers were discovered.

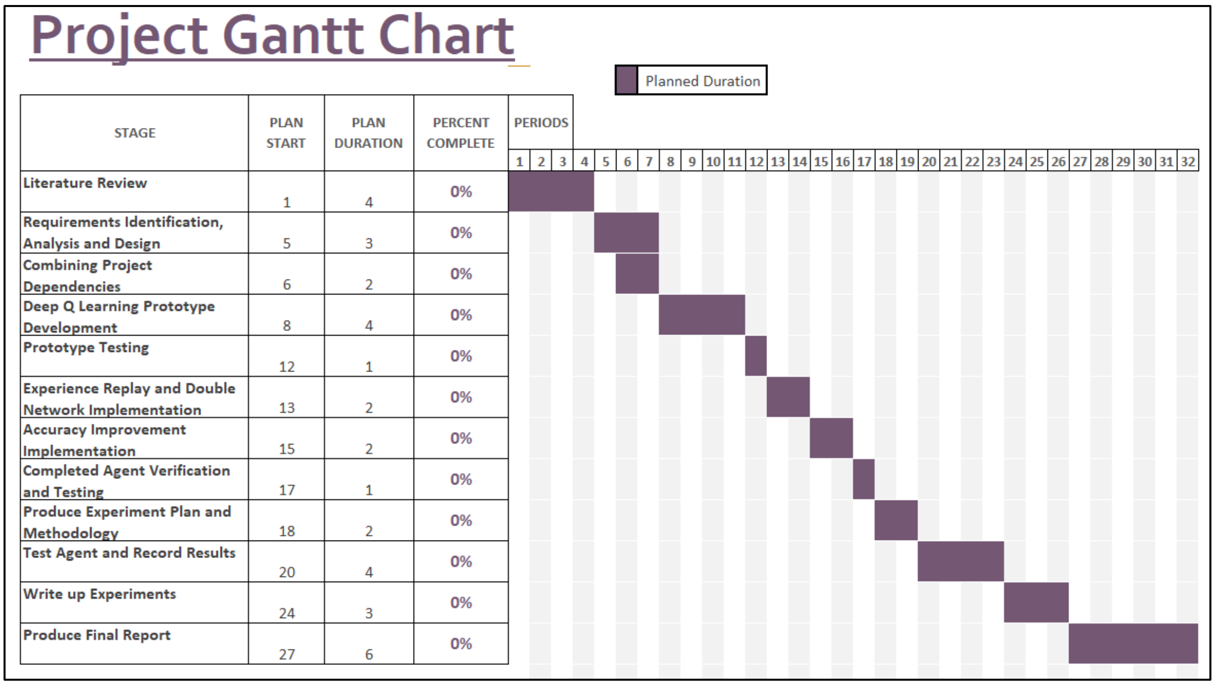
Firstly, the game warlords, could not be emulated using the intended API for RL environment modelling. After discussion with the creators of the API, it became clear that adding “Warlords” to their roster would take a large amount of time and would be out of the scope of this project. Instead, this project adapted to focus on training a general Atari 2600 agent which could play any game.

The second barrier was the required training time when training multiple agents against each other. Whilst examining the training times other researchers required for their agents, it became apparent that maintaining two or more agents would require an inordinate amount of memory and large amounts of computation. It was decided that this would add little value to the project for the training time it would require. Instead, only a single agent was trained at any time, and this allowed for full utilisation of the training systems resources.

Another slight change to the original plan was to consider the Agile framework for project management in place of the planned iterative waterfall model. After considering the number of changes to requirements which had occurred during the early stages of the project, it made sense to use a flexible framework which works well with iterative development in mind. This would allow for easier adaptation if requirements continued to change during development, and provide smaller working prototypes to be produced along the way to ensure viable progress was being achieved.

To ensure that work was being completed to a high quality and on schedule, highlight reports were produced each week detailing the work carried out and any barriers encountered. These reports were then discussed during a weekly meeting with the project supervisor, who provided guidance on the current state of the project and suggested areas in which the project could expand.

A new project plan was produced to reflect these changes;



Project Gantt Chart

▨ Planned Duration

STAGE	PLAN START	PLAN DURATION	PERCENT COMPLETE	PERIODS																															
				1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Literature Review	1	4	0%	▨	▨	▨	▨																												
Requirements Identification, Analysis and Design	5	3	0%					▨	▨	▨																									
Combining Project Dependencies	6	2	0%						▨	▨																									
Deep Q Learning Prototype Development	8	4	0%								▨	▨	▨	▨																					
Prototype Testing	12	1	0%												▨																				
Experience Replay and Double Network Implementation	13	2	0%													▨	▨																		
Accuracy Improvement Implementation	15	2	0%															▨	▨																
Completed Agent Verification and Testing	17	1	0%																	▨															
Produce Experiment Plan and Methodology	18	2	0%																		▨	▨													
Test Agent and Record Results	20	4	0%																				▨	▨	▨	▨									
Write up Experiments	24	3	0%																								▨	▨	▨						
Produce Final Report	27	6	0%																											▨	▨	▨	▨	▨	▨

Figure 5: Project Gantt Chart

3.1.2 Development Tools

API Selection

For emulating the Atari 2600 and modelling the reinforcement learning environment, Open AI Gym ("Gym") was selected. This is an API used by reinforcement learning researchers which provides standardised environments allowing results from different agents to be easily compared. This API handled the emulation of the Atari 2600 and provided a matrix of the screen pixels, and reward (in-game score) for every frame produced.

It was also decided to use an API for creating and managing the convolutional neural networks required, as designing and verifying a bespoke network would be out of the scope of this project. There was a range of options for this API, so the advantages and disadvantages were considered (See Appendix D) before Tensorflow was selected.

Programming Language

Once Open AI Gym had been selected as the core API of this project, the language had to be Python, as this is currently the only language which is supported. This had some advantages and disadvantages.

Using Python gave me access to a wide range of packages for many tasks (manipulating tensors, converting file types, easily parsing command line arguments etc). Lots of these tasks would have been far more difficult to achieve using a lower level programming language such as C++ (which was one of the other initial languages considered), and would like to have required me to produce extra code outside the scope of this project.

One of the main disadvantages of python is the large computational overhead, which makes it the less than ideal choice for such a demanding project. To overcome this, the number of steps in a single training trial was reduced to less than the amount specified in some of the research mentioned in my literature review.

IDE Selection

A range of integrated development environments (IDEs) were examined to determine which provided the most advantageous features when working with Python. The advantages and disadvantages of each were compared and Visual Studio with the Python plugin was selected.

3.1.3 Timing and Schedule

The start of this project took much longer than anticipated due to the difficulty in finding a set of compatible APIs and then running the planned game, "Warlords". The main requirement was to find a library or API which could emulate the Atari 2600, and could extract, the frames and score for the RL agent. "Gym" was an early choice, but would not run on Windows, so a wider range of emulators were investigated, none of which provided the required functionality. Eventually a work around was discovered so that "Gym" could be used. Further problems then ensued in running the planned game "Warlords" which was not supported by the emulator encapsulated within "Gym". This caused a large change to the requirements and scope of the project, which were reflected above in the project plan adaptation.

An early consideration was the risk of large training times limiting the scope of the project, and causing difficulties in debugging (See, risk assessment of PID, Appendix B2). The amount of training steps carried out in the research listed in the literature review varied from 10,000,000 to 50,000,000 for a single trial. However, there was no specific mention of how long one of these trials took to complete. Once the prototype agent was functioning, timings were recorded for different sections of the code, under different parameters to produce an estimate of how long it would take the agent to complete 1000 steps. This time varied significantly depending on several factors, with the most important factor being the size of the experience replay. The time for completing 1000 steps varied between 5.6 seconds and 32.3 seconds depending on these parameters. This large variation made estimating the total length of a trial difficult, so the average estimate for 1000 steps was assumed to be slightly higher than the mean of this range (20 seconds). From this estimate the amount of training time in a trial could also be estimated. For 50,000,000 steps it would take 11.5 days, and for 10,000,000 steps it would take 2.3 days to train one agent in one game. This training time would make it almost impossible to test a range of parameters and produce viable experimental results. To counter this, the number of steps in a single trail was set at 5,000,000 and the size of the experience buffer was significantly reduced to increase training time (at a cost of accuracy). These changes reduced the estimated training time for a trail to 23 hours.

3.1.4 Legal, Social and Ethical Issues

This project presented no legal, social or ethical issues, as it was exploring the design and operation of RL agents, rather than their real-world applications, which would contain a wider range of these issues. There are a range of safety issues involved with RL agents operating in the real-world (Leike et al, 2017), however none of them applied to this project.

3.2 Analysis and Design

3.2.1 User Stories

To make the requirements for this project more digestible, two sets of user stories were created. One set from the perspective of a user looking to run experiments on the agent with a variety of parameters. With the other set from the perspective of the agent and its requirements within the program.

User Story

1) A user must be able to launch the program, specifying parameters for their experiment, and be presented with the results. The acceptance criteria for this are;

- **A)** Can the user launch the program specifying which game, and how many episodes they wish to test?
  - **a.** If the user inputs invalid values when launching the program, are clear and concise exceptions displayed?
- **B)** Does the user have control over whether the game is rendered to the screen?
  - **a.** If the game is not rendered, can the user see the experiments progress from the console output?
- **C)** Are results automatically recorded and saved in a sensible format and location?
- **D)** Can the user control saving and loading agents if training is split across multiple program executions?
- **E)** Can the user load a trained agent and watch the progress with a fixed frame rate?
- **F)** Are the console arguments clearly defined with necessary help attributes and accompanying documentation?

Agent Story

1) An agent must be able to estimate the value of taking an action in a given state. The acceptance criteria for this are;

- **A)** Can the agent perform the required pre-processing on a frame to create a valid state?
- **B)** Can the agent select an action when given a state as an input?
- **C)** Based on received rewards can the agent intelligently update its action selection policy?

2) An agent must be able to remember past experiences (state, action, reward, next state), and recall these experiences using a uniform random selection and learn from them.

- **A)** Can the agent store a number of past experiences within a defined memory limit?
- **B)** Can a sample of these experiences be taken uniformly at random?
- **C)** Can the agent update its action selection policy using this batch of experiences in place of a single input?

3.2.2 Deep Q Network Design

As mentioned in the literature review, all Deep Q Networks investigated contained the following features;

- At least two convolutional layers with an increasing number of filters per layer (and a reduction in kernel size and stride)
- Linear rectifier used to limit the range of the outputs (ReLU)
- No pooling layers between convolutional layers, in order to maintain spatial relations
- At least two fully connected layers to reduce dimensionality and classify the output

The number of output nodes from the network needed to be equal to the number of viable control inputs for the selected game. This was easily achieved as “Gym” provided a function which would return a list of all the viable control options for a loaded game.

The size of the input depended on exactly what pre-processing was performed on the frame(s) before the state was presented to the network. Each state was to consist of 4 frames, each of which was converted to a greyscale image and scaled down by a factor of two. This gave the input a dimensionality of 105 x 80 x 4, where the dimensions are height, width, frame.

As Tensorflow was allowing me to use GPU parallelisation to increase performance I decided to use a large network architecture which was likely to produce better results at the cost of slightly longer computation time. The final architecture of my Deep Q Network was as follows;

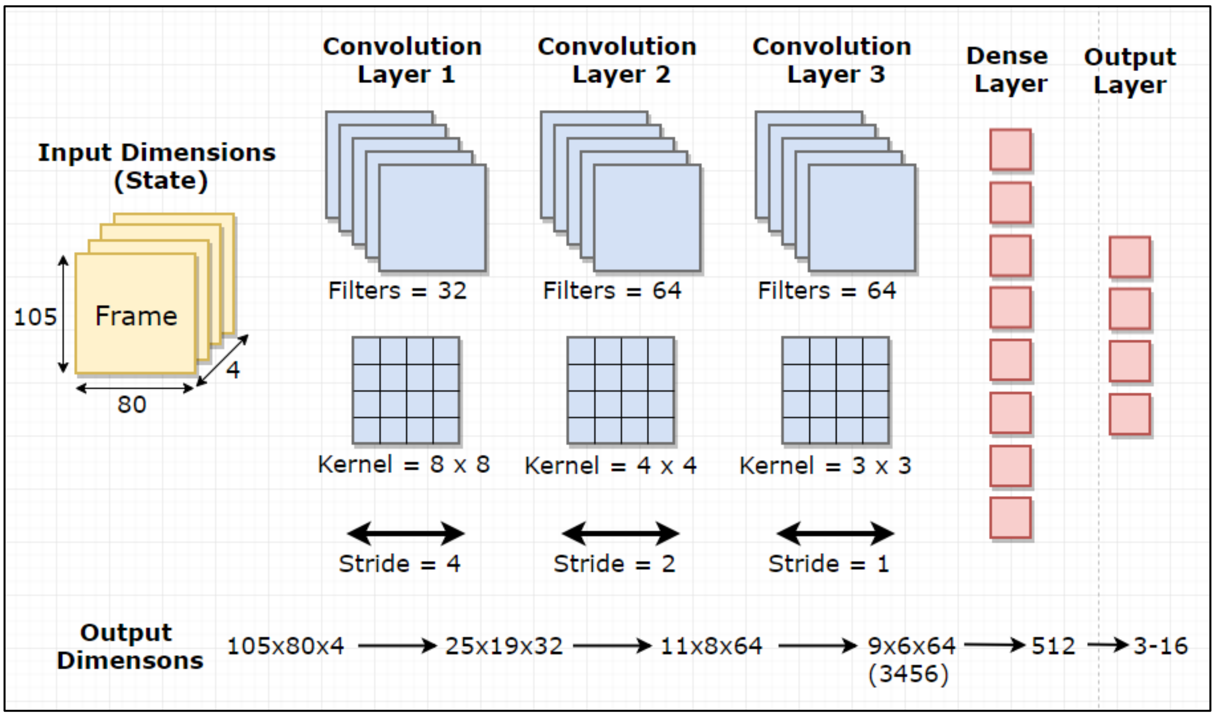


Figure 6: Deep Q Network Architecture

Figure 6 represents the architecture of the network, showing the dimensionality of the output after each layer. The convolutional layers and dense layer all use the rectifier linear unit (ReLU) on their respective outputs. The output layer just contains the raw estimated Q values for each action based on the presented state. These raw values can then be used in a range of ways, either using the SoftMax algorithm to provide a probabilistic range over the action space, or taking the maximum argument to implement a greedy action selection policy.

3.2.3 API Integration and Application

Some consideration was taken into how the chosen API's would integrate into the program structure, and what benefits and drawbacks they would present with their provided functionality.

Each of the APIs had a specific role within the program, and they required no direct communication between each other. "Gym" was responsible for;

- Emulating the Atari 2600
- Rendering the Atari frames to the screen
- Providing the range of viable inputs for a game
- Providing the current frame, reward and terminal state information to the agent

The main advantage of using "Gym" was that it was quickly becoming the standard platform for reinforcement learning problems. This made it easier to compare results with others and gave reassurance that the environment was a controlled constant. The main drawback was that it is still currently in Beta, and as such has some strange quirks, including; blank error messages, depreciating methods and little to no documentation. The Atari emulator within Gym was also intended to be used on a Linux distribution, so would not install on Windows. As a work around, a variant of the emulator with windows support was discovered on GitHub (https://github.com/Kojoley/atari-py). This version of the emulator was installed and "Gym" was directed to this version in place of its included emulator, which solved this issue.

Tensorflow was responsible for;

- Creating and managing the convolutional neural networks
- Taking "states" as inputs and producing Q values as outputs
- Parallelising operations on the GPU, where possible, for improved performance

Tensorflow had a huge range of benefits. It provided peace of mind that the created network layers would function as intended without the need for extensive testing. It was also likely to be designed in a far more optimised way than if the neural network had built from scratch for this project. Finally, there was the advantage of semi-autonomous GPU parallelisation which would utilise the GPU where possible, based on the structure of the network and shape of the provided inputs. This parallelisation allowed for incredibly fast batch updates, and vastly increased the overall potential speed of the agent. The only downside of Tensorflow was the long initial setup which required the installation and configuration of various drivers and plugins.

3.2.4 File I/O (Saving and Loading Agents and Results)

This project required two input/output functions. One for saving and loading the current state of the neural network, and one for recording results from experiments.

Tensorflow provided functionality for saving and loading individual variables or a graph representing the whole neural network. These functions can be wrapped within the main code of the agent and triggered based on user parameters. The following functionality was required;

- When a user sets the parameter to save the agent;
    - After each episode the neural network graph should be saved to file and named with a convention that includes the game name and episode number
- When a user sets the parameter to load an agent;
    - The user should specify which game is being loaded, and the number of the episode they wish to load from
    - These parameters are then used to create the path string required to load the neural network graph

It was decided that saving after each episode would provide the best balance between saving the model regularly and reducing overhead from I/O functions. The overhead from saving after each episode was minimal, and when timed using a double precision float, the time taken was 0.0.

The other file output functionality required was the results recorder. This class had a few requirements;

- At the start of a trial, a new csv file must be created for storing the results
    - This file must be named using a unique convention, containing the game name
    - There must be protection to stop files with the same name being overwritten
- At the end of each episode all relevant values should be stored in the csv file
    - There should be an initial line in the file providing headings for these values

For each episode, 5 values were targeted for recording; episode number, number of steps per episode, reward, game score and final epsilon value. These values provided enough information to assess the progression and learning rate of the agent. Other useful values that were not recorded could still be determined, such as total trial steps (as a cumulative sum of steps per episode) and number of training updates (as the result of trial steps modulus 4).

3.3 Implementation

3.3.1 Frame pre-processor

The first required element of the program was the ability to pre-process a raw frame of the game presented by "Gym" into a viable state. There were two main reasons for this pre-processing, firstly, it reduced the dimensionality of the inputs without removing any relevant information the agent would require. Secondly, it made a state represent multiple consecutive frames, so that the agent would consider the concept of movement and directionality.

Each step through the environment, one frame of the game would be presented to the agent. The agent would convert this image to greyscale, and down sample it by a factor of two before storing the frame into a 4-frame buffer. This buffer was then used to create the pre-processed state by concatenating the four stored frames.

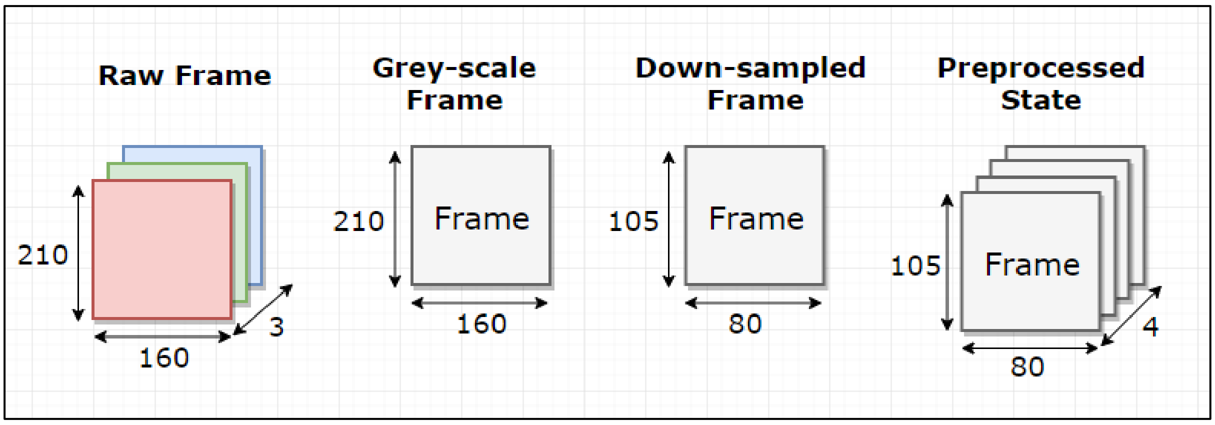


Figure 7: Pre-processing steps from raw frame to input state

Each frame was converted to grey-scale by taking the mean of the matrix across the second dimension, this results in a 210 x 160 frame. The grey-scale frame was then down-sampled by half by slicing the array in both directions taking only the even indices, outputting a 105 x 80 frame. All frames are added to the frame buffer and each step the current 4 frames have their dimensionality extended by 1 to 105 x 80 x 1. These frames are then concatenated across the second dimension.

```
PRE-PROCESSOR CLASS (frame)
        greyscale = MEAN (frame, second dimension)
        down-sampled = greyscale[evenIndices][evenIndices]
        RETURN down-sampled

AGENT CODE EXERT
        framebuffer = dequeue, length 4
        EVERY step:
                processedFrame = pre-processor(rawFrame)
                IF length (framebuffer) > 4
                        state = CONCANTANATE (framebuffer, second dimension)
                        DELETE framebuffer [0]
                framebuffer APPEND processedFrame
```

3.3.2 Deep Q Network Implementation and Training

The architecture of the Deep Q Network was based on the structure described in section 3.2.2. Tensorflow allowed for each layer of the network to be defined to these specifications, determining the input and output variable at each layer, which allowed for a processing pipeline to be formed.

A 4-dimensional tensor was used as an input placeholder. The first dimension would represent the index of a batch of states while the other three dimensions stored the state values. Using this structure allowed for an easy transition into batch updating when the experience replay was added. Also in preparation for batch updates, all the tensor functions implemented, performed their actions summed across an array so that training would work with either single values or n-sized batches.

Updating the network was achieved using the sum-squared error and a variation of the Bellman equation. In traditional Q Learning, the Bellman equation would be used to update a Q value based on the received reward and highest Q value of the next state;

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[reward + \gamma \max_{a} Q(sNext) - Q(s,a) \right]$$

However, to update the neural network, the loss or error needed to be calculated. This could be considered as the difference between the actual result and target result. The target result could be estimated as the reward received in the current state, plus the discounted network estimate of the action with highest Q value for the next state;

$$target = reward + \left[\gamma \ DQNestimate(\max_{a} Q(sNext)) \right]$$

The network could then calculate the actual Q Value range for the current state by running the state through the network, and multiplying the raw network output by a one-hot mask of the chosen action. This would provide just the raw estimate Q value for the selected action;

$$actual = SUM(NetworkOutput * OneHotActionMask)$$

Finally, the error could be described as the difference between the target and actual values. In this case, the sum squared error was taken to prioritise larger errors and provide an absolute value in place of a singed value.

$$sumSquareError = (target - actual)^2$$

An alternative to sum squared error (SSE) was described by Mnih et al (2015), who investigated mean absolute error (MAE), which takes the absolute of the difference between the actual and target values. This helps to limit large errors that may cause the network to overfit, however this also masks smaller values and is not differentiable at 0. The solution proposed by Mnih et al (2015) was to use a process called Huber loss, which would use the SSE for small errors and MAE for large errors. Although Huber loss was not implemented within the scope of this project it was considered as a potential improvement which could be used in future work.

The last step was updating the network with stochastic gradient decent, with the aim of minimising the sum squared error. This was achieved using the Adam algorithm (Kingma and Ba, 2014), which surpassed the performance of the route mean squared approach under most circumstances.

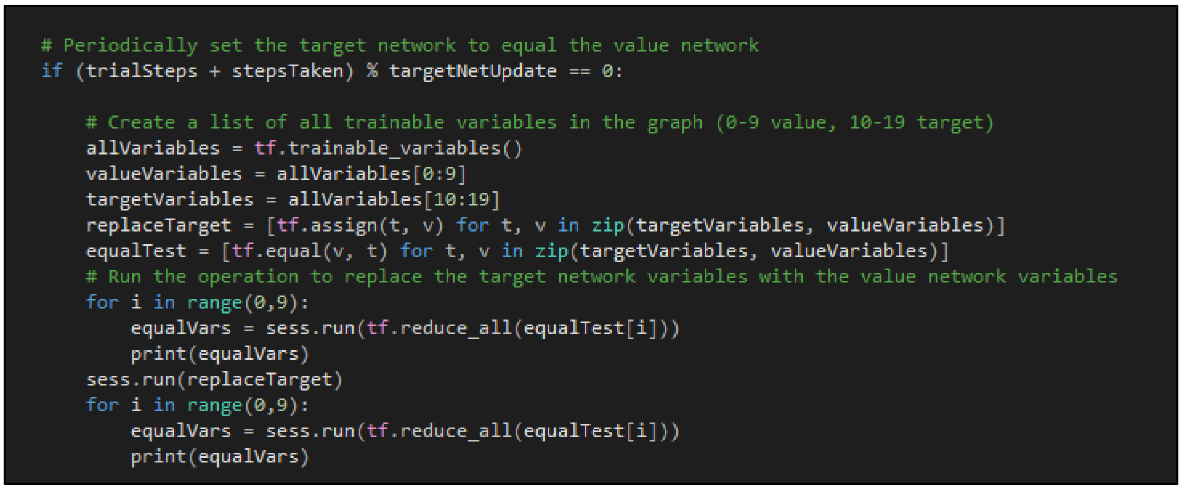
With the network training and pre-processing implemented, the agent was now able to achieve learning by updating the network after every step taken. This lead to converging behaviour and allowed for further optimisations to be implemented.

3.3.3 Target and Value Networks

One improvement discussed by Heinrich & Silver (2017) was the use of two identical Deep Q Networks, one for producing the estimated Q values for the current state and the other for producing the estimated target Q values for the next state. If just one network is used to estimate both the actual Q value and the target Q value, then after each step the target Q value would move as the network is updated. This can lead to feedback loops and cause the network updates to stagnate. By separating these roles into two networks this problem can be alleviated. The value network is updated regularly, but the target network is either slowly updated towards the current value network, or is set to be equal to the value network after a set number of steps have occurred.

Some alterations were made to the structure of the agent to account for these changes. At initialisation, two of the Deep Q Network class were created and both populated with random weights. The target network takes the responsibility of estimating the maximum Q value for the next state, whilst the value network performs all other network functionality for action selection.

Every 10,000 steps the target network is set to the equal the value network. Tensorflow did not have any functionality to clone these networks as they are both considered part of the same graph. The solution to this was to retrieve all the variables within the graph, and separate them into the variables for the value network and target network. An update function was then created so that Tensorflow could replace each of the target variables with the equivalent value variable. Another function was created which tested if the all the variables within the two networks were equal;



```
# Periodically set the target network to equal the value network
if (trialSteps + stepsTaken) % targetNetUpdate == 0:

    # Create a list of all trainable variables in the graph (0-9 value, 10-19 target)
    allVariables = tf.trainable_variables()
    valueVariables = allVariables[0:9]
    targetVariables = allVariables[10:19]
    replaceTarget = [tf.assign(t, v) for t, v in zip(targetVariables, valueVariables)]
    equalTest = [tf.equal(v, t) for t, v in zip(targetVariables, valueVariables)]
    # Run the operation to replace the target network variables with the value network variables
    for i in range(0,9):
        equalVars = sess.run(tf.reduce_all(equalTest[i]))
        print(equalVars)
    sess.run(replaceTarget)
    for i in range(0,9):
        equalVars = sess.run(tf.reduce_all(equalTest[i]))
        print(equalVars)
```

Figure 8: Updating and Testing Target Network

3.3.4 Experience Replay

This involves storing information that the agent has experienced during progression through the environment. Batches of these experiences can then be sampled to train the Deep Q Network, rather than just training on the current experience. This provides uncorrelated data for training, which is optimal for neural network updates, and significantly improves data efficiency, as each experience can be sampled multiple times (Zhang and Sutton, 2017).

For this project an experience consisted of a state, selected action, received reward and resultant state. These values needed to be recorded at every step and stored into a fixed size experience buffer.

A class was made for handling the experience replay;

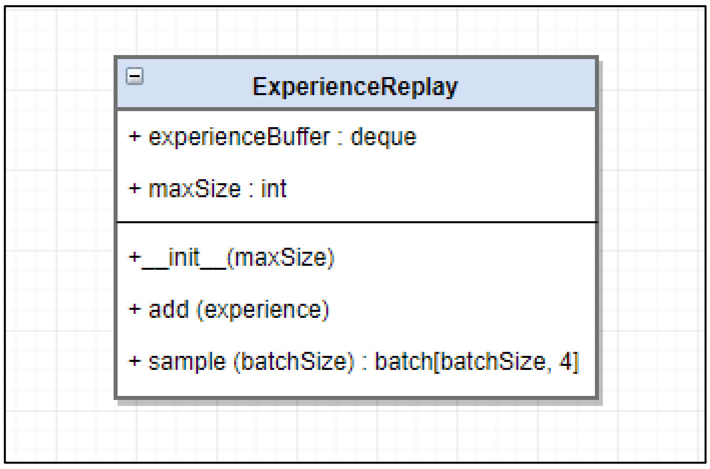

ExperienceReplay
+ experienceBuffer : deque
+ maxSize : int
+__init__(maxSize)
+ add (experience)
+ sample (batchSize) : batch[batchSize, 4]

Figure 9: Experience Replay Class Diagram

The buffer was made a deque to provide further optimisation and prevent shuffling when full. Sampling the buffer was initially achieved using the built-in array sample function, however it was discovered during testing that this function had an average efficiency of O(mn), where m is the buffer size and n is the requested batch size. In place, an index based selection method was created which could sample experiences with an average efficiency of O(n). For large buffers this significantly decreased the sampling time.

As the Deep Q Network was already structured to take n-sized batches, each sampled batch was sliced into sub-batches of states, actions, rewards and next states. These sub-batches were then fed into the network in place of the single inputs.

3.3.5 Program Structure and Execution

The program is structured as few nested elements;

- Trial – a full experiment on a single game, with a fixed set of parameters.
- Episode – one play through of the chosen game (starting from the initial state, continuing until all lives have been lost / terminal state has been reached).
- Step – the smallest measure of progression through an episode. Consists of receiving a frame from the game, selecting an action, recording the experience and performing any learning updates.

The following pseudocode describes the flow of an entire trial;

```
Initialise preProcessor
Initialise targetNetwork and valueNetwork
Initialise experienceReplay
Create results file

IF LOAD MODEL:
      valueNetwork = LOAD(modelRef)
      targetNetwork = valueNetwork

FOR EACH Episode:
      Initialise frameBuffer
      observation = GYM.FIRST_OBSERVATION

      WHILE NOT Terminal State:
            action = valueNetwork.MaximumArgument(state)

            IF Episode Steps > 4:
                  state = CONCANTANATE(frameBuffer)

            observation, reward = GYM.STEP(action)
            frame = preProcessor(observation)
            frameBuffer APPEND frame

            IF Episode Steps > 4:
                  DELETE frameBuffer [0]
                  nextState = CONCANTANATE(frameBuffer)

            experienceReplay APPEND ([state, action, reward, nextState])

            IF Episode Steps MOD Update Frequency == 0:
                  batch = experienceReplay.Sample(Batch Size)
                  maxActions = targetNetwork.RawOutput(batch[NextState])
                  targets = batch[Reward] + (discount * maxActions)
                  valueNetwork.Update(batch[State], batch[Action])

            IF Episode Steps MOD Target Update Frequency == 0:
                  targetNetwork = valueNetwork

      IF SAVE_MODEL:
            SAVE(valueNetwork)

      Save to file ([Episode Number, Steps, Reward, Score, Epsilon])
      PRINT Episode Info
```

The full structure of the program is shown in the UML diagram in Appendix E

Another consideration when designing the agent was the exploration, exploitation dilemma. For the agent to learn about all the possible scenarios in the state space and how different actions affect these states, it is advantageous for the agent to be explorative. However, to advance understanding of the domain, the agent must also choose the action which it believes will lead to a state with the most value.

The simplest solution to this is the greedy-epsilon policy. With this policy the agent selects the best action (1 – epsilon) of the time and selects a random action (epsilon) of the time. The epsilon value is often set to 0.1, so that 1 in 10 actions is selected at random. This allows the agent to make progress by choosing the action with the greatest value most of the time, but still allows for a degree of exploration. However, with larger, or more complex state spaces, this policy doesn't achieve great results.

For this project, a variation was used; annealing greedy-epsilon. A maximum and minimum epsilon value are determined as constants. The epsilon value starts at the maximum and anneals by a small amount every step until the minimum value is reached. Combined with the experience replay, this allows the agent to learn from a variety of explorative and exploitive experiences.

The full list of hyper parameters for the agent are;

Name	**Value**	**Description**
Alpha	0.0001	Learning rate for Adam algorithm
Gamma	0.99	Discount rate of subsequent states value
Epsilon Minimum	0.05	Lowest value epsilon can reach
Epsilon Maximum	1.0	Highest, and starting value of epsilon
Epsilon Delta	0.000002	Amount which epsilon anneals by each step
Replay Size	150000	Number of experiences stored in memory
Update Frequency	4	Number of steps between value network updates
Batch Size	32	Number of experiences sampled for one update
Target Update Frequency	10000	Number of steps between target network updates

The remaining functionality of the program was the command line argument parser. The full details of the available commands and operating procedure can be found in the user guide located in Appendix A.

3.4 Validation and Verification

3.4.1 Overview

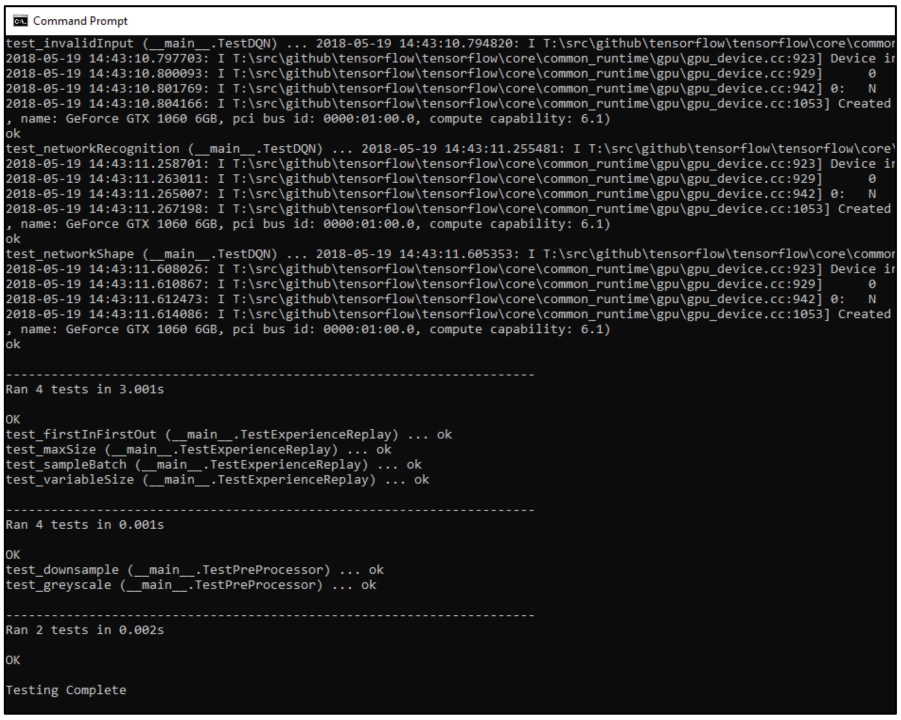
There were two stages to the validation and verification of this project. Firstly, each individual element of the program had to be investigated to ensure semantic correctness was achieved. For most of the independent classes, methods and functions, this could be achieved through unit tests. Remaining areas were manually checked to ensure that desired behaviour was being achieved for all edge cases.

The second stage was examining the performance and learning ability of the agent. This involved running several tests using a controlled set of hyper-parameters. The results from each test were recorded and could be used to confirm that learning is being achieved. These results also allowed a direct comparison to equivalent research, so that the efficiency of the created agent could be compared to other agents.

3.4.2 Unit Testing

The unit tests were controlled using the default Python "unittest" package, which provided a standard set of tools for creating dummy scenarios and testing assertions. To make best use of this package I spent some time investigating how to operate under this framework (Unit Testing Framework, 2018). The main purpose of the tests was to ensure that unexpected behaviour was accounted for, and that edge cases were examined thoroughly.

Tests were carried out on three of the main classes the agent used; Deep Q Network, Pre-Processor and Experience Replay. Running these tests on the final version of the agent produces the following results;



```
test_invalidInput (__main__.TestDQN) ... 2018-05-19 14:43:10.794820: I T:\src\github\tensorflow\tensorflow\core\commor
2018-05-19 14:43:10.797703: I T:\src\github\tensorflow\tensorflow\core\common_runtime\gpu\gpu_device.cc:923] Device ir
2018-05-19 14:43:10.800093: I T:\src\github\tensorflow\tensorflow\core\common_runtime\gpu\gpu_device.cc:929]      0
2018-05-19 14:43:10.801769: I T:\src\github\tensorflow\tensorflow\core\common_runtime\gpu\gpu_device.cc:942] 0:   N
2018-05-19 14:43:10.804166: I T:\src\github\tensorflow\tensorflow\core\common_runtime\gpu\gpu_device.cc:1053] Created
, name: GeForce GTX 1060 6GB, pci bus id: 0000:01:00.0, compute capability: 6.1)
ok
test_networkRecognition (__main__.TestDQN) ... 2018-05-19 14:43:11.255481: I T:\src\github\tensorflow\tensorflow\core\
2018-05-19 14:43:11.258701: I T:\src\github\tensorflow\tensorflow\core\common_runtime\gpu\gpu_device.cc:923] Device ir
2018-05-19 14:43:11.263011: I T:\src\github\tensorflow\tensorflow\core\common_runtime\gpu\gpu_device.cc:929]      0
2018-05-19 14:43:11.265007: I T:\src\github\tensorflow\tensorflow\core\common_runtime\gpu\gpu_device.cc:942] 0:   N
2018-05-19 14:43:11.267198: I T:\src\github\tensorflow\tensorflow\core\common_runtime\gpu\gpu_device.cc:1053] Created
, name: GeForce GTX 1060 6GB, pci bus id: 0000:01:00.0, compute capability: 6.1)
ok
test_networkShape (__main__.TestDQN) ... 2018-05-19 14:43:11.605353: I T:\src\github\tensorflow\tensorflow\core\commor
2018-05-19 14:43:11.608026: I T:\src\github\tensorflow\tensorflow\core\common_runtime\gpu\gpu_device.cc:923] Device ir
2018-05-19 14:43:11.610867: I T:\src\github\tensorflow\tensorflow\core\common_runtime\gpu\gpu_device.cc:929]      0
2018-05-19 14:43:11.612473: I T:\src\github\tensorflow\tensorflow\core\common_runtime\gpu\gpu_device.cc:942] 0:   N
2018-05-19 14:43:11.614086: I T:\src\github\tensorflow\tensorflow\core\common_runtime\gpu\gpu_device.cc:1053] Created
, name: GeForce GTX 1060 6GB, pci bus id: 0000:01:00.0, compute capability: 6.1)
ok

----------------------------------------------------------------------
Ran 4 tests in 3.001s

OK
test_firstInFirstOut (__main__.TestExperienceReplay) ... ok
test_maxSize (__main__.TestExperienceReplay) ... ok
test_sampleBatch (__main__.TestExperienceReplay) ... ok
test_variableSize (__main__.TestExperienceReplay) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.001s

OK
test_downsample (__main__.TestPreProcessor) ... ok
test_greyscale (__main__.TestPreProcessor) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.002s

OK

Testing Complete
```

Figure 10: Unit Test Results on Final Agent

The full details of the unit tests can be seen within the "UnitTests.py" script, and the procedure for launching unit tests is explained in the user guide (Appendix A).

3.4.3 Manual Testing

Manual testing was carried out to examine more complex scenarios, in place of testing individual methods. This also allowed for command line arguments and file I/O to be tested thoroughly. Results from these tests helped to verify that the user stories were being achieved correctly. These tests were carried out over the course of the implementation of the agent, and the recorded results are available in Appendix F.

3.4.4 Agent Ability and Training Efficiency Tests

To investigate the learning ability of the agent, tests were run across 5 games; Pong, Breakout, Sea Quest, Ms Pacman and Space Invaders. For these tests the hyper-parameters were kept constant as described in the table in section 3.3.5 and each game was trained from scratch with a new agent. Each trial ran for 5,000,000 steps and the raw results recorded can be found in Appendix G.

The results from these trials, including a rolling average trendline, are presented below;

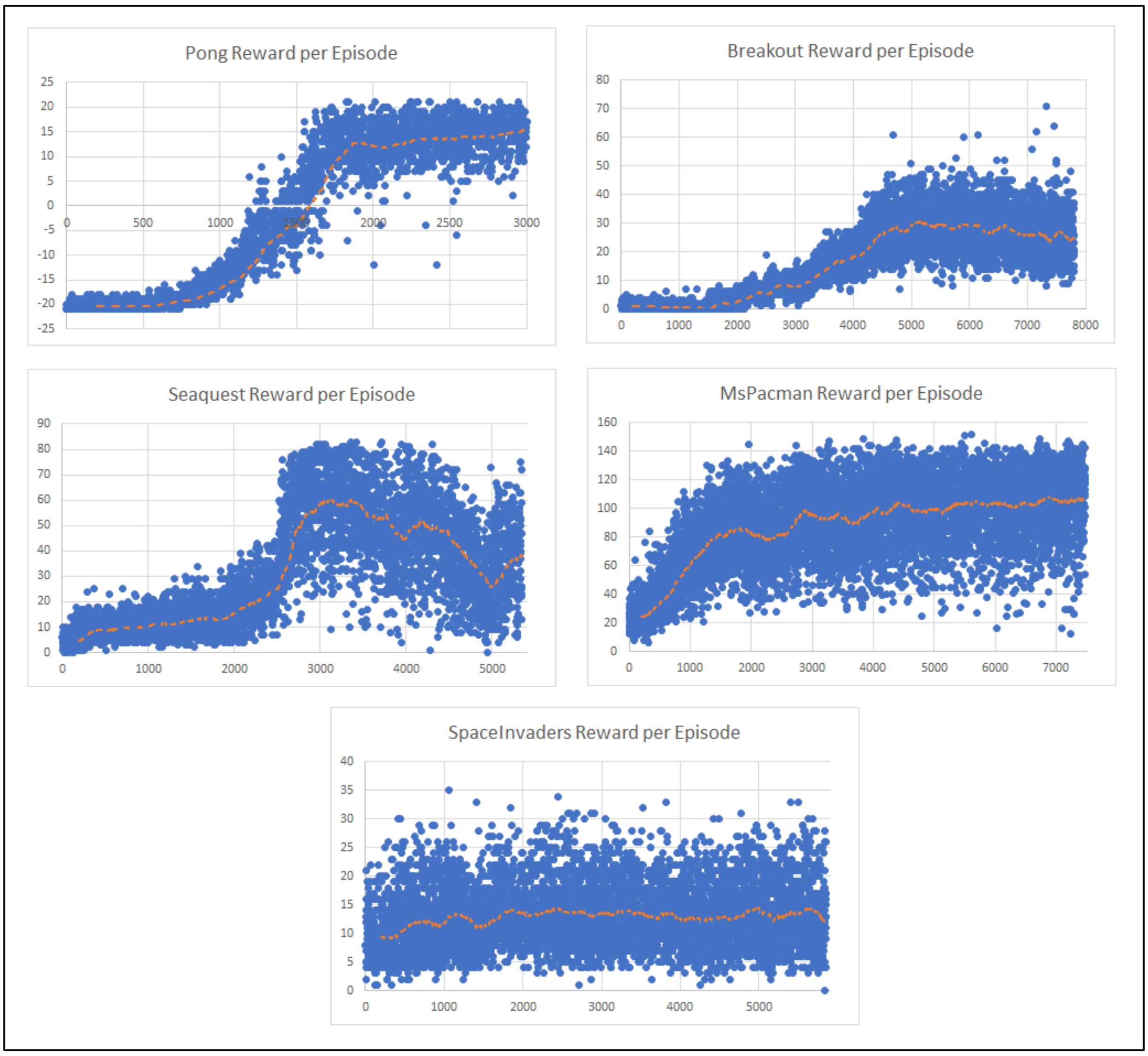


Figure 11: Agent reward per episode over 5,000,000 steps for a series of games, including trendlines of a rolling average over 200 episodes

From the graphs in figure 11, it can be seen that learning was achieved for 4 out of the 5 games tested. The performance of the agent in different games varies depending on a few factors.

The Pong agent, achieved the best results, and after training for 5,000,000 steps was able to win 499 games out of 500. The Breakout and Sea Quest agents also achieved promising results and were able to learn complex behaviour about the state space. The Ms Pacman agent achieved a degree of learning, but very quickly plateaued and was not consistent in achieving high scores. The worst performer was the Space Invaders agent which didn't manage to improve its performance compared to random behaviour.

The poor performance in Ms Pacman and Space Invaders was likely due to reward saturation. Both games present the agent with common rewards, for every "pill" collected in Ms Pacman and for every ship destroyed in Space Invaders. This reward saturation makes it difficult to train the Deep Q Network as too many states have rewarding behaviour to accurately identify the best action at a given time.

To compare the performance of my agents to others, I used values from Mnih "Playing Atari with Deep Reinforcement Learning" (2013) to produce the following table;

	Pong	**Breakout**	**Sea Quest**	**Space Invaders**
Random	-20.4	1.2	110	179
Sarsa (Bellemare, 2012)	-19	5.2	665	271
Human	-3	31	28010	3690
DQN (Mnih, 2013)	20	168	1705	581
DQN (This Project)	**15.4**	**39.2**	**786.6**	**181.9**
DQN BEST (Mnih, 2013)	21	225	1740	1075
DQN BEST (This Project)	**21**	**320**	**1660**	**900**

The top section of the above table compares the average scores for different games using different action selection policies;

- "**Random**" is uniformly randomly selected actions.
- "**Sarsa**" is the average performance of an alternative reinforcement learning algorithm (Bellemare, 2012).
- "**Human**" is the average of the median performance of players with two hours experience.
- "**DQN (Mnih)**" is the average performance of the Deep Q Network agents specified by Mnih (2013).
- "**DQN (This Project)**" is the average performance (over the last 200 episodes) of the agents created for this project.

The bottom section of the above table contains the highest scores achieved in each game by the agents specified by Mnih and the agents created for this project.

This table demonstrates that the agents produced for this project were able to achieve superhuman performance in both Pong and Breakout. It also shows that this implementation could get better maximum results, in Pong and Breakout, than a comparable Deep Q Network. Although the average performance of this project's agents was below that achieved by Mnih, this is likely due to the reduction in training steps, and a smaller experience replay.

3.4.5 Continued Improvement Plan

Although the produced agents managed to learn the complex state spaces of some Atari games, there is still room to improve most of their performances. Two of the most valuable variations of Deep Q Networks are Double DQNs and Duelling DQNS.

Double DQNs aim to reduce overestimation of certain action values by allowing the value network to select an action and then using the target network to generate the estimated Q value for that action. Separating the target generation from the action selection substantially reduces the level of overestimation (Van Hasselt, 2015).

Duelling DQNs aim to improve performance by separating the Q Value into two streams. As a Q value represents the value for taking an action in a given state [Q(s, a)], it can be represented instead as the value of the current state [V(s)] and the advantage of the chosen action [A(a)]. By using an action selection policy that combines the value of the state and the advantage of an action, a better performance can be achieved (Wang, 2015).

There are also some improvements that could be made, to the program, in terms of user control;

- A trail could be run for either a pre-defined number of steps or number of episodes.
- For long trials it would be advantageous to stop execution at any point without causing exceptions.
- It could be possible to save the experience replay, so that a model can be reloaded with a pre-filled experience buffer. The downside to this would be huge memory requirement for saving experiences.

Chapter 4: Exploration vs Exploitation Experiment

4.1 Introduction

Using the agent designed for this project, the author has decided to attempt a novel technique for improving learning ability. This experiment was written up as an academic paper, in the LNCS Springer style, and can be found in Appendix H. This chapter contains the main premise, methodology and conclusion of the experiment, but more detail can be found in the attached paper.

As described in section 3.3.5, the exploration, exploitation dilemma, considers how best to balance the amount an agent takes random actions to increase its knowledge of a state space, with the amount of calculated actions to increase its reward.

4.2 Existing Research

The greedy epsilon policy used in this project, and other similar examples (Mnih, 2013), is often referred to as an annealing epsilon. By starting with totally random behaviour and slowly taking more calculated actions over time, a good balance of exploration and exploitation are achieved. The draw back to this technique is that once the epsilon has reached the minimum, learning can stagnate, and the agent can quickly fall into local minima.

An alternative approach is the adaptive greedy epsilon policy (Tokic, 2010). This approach considers allowing the epsilon value to vary during runtime based on error calculated by the neural network. If the error starts to grow larger, then the epsilon value will start to proportionally increase. The theory is that this will allow the agent to increase exploration relative to the localised state space it has reached through exploitation. Tokic (2010), achieved results which showed that this policy can achieve better than annealing epsilon for the one-armed bandit problem. However, there was little evidence of this approach, or any similar approach, being tested on high-dimensional complex state spaces, such as the Atari 2600.

4.3 Hypothesis

The two new approaches are based on the principle that an agent adapts its behaviour based on the received reward. This suggests that if the agent can self-assess how it is performing, by considering the change in reward over time, it should be able to alter its own epsilon value to vary exploration when it is most needed.

The hypothesis is that using these approaches will allow for a better average reward than using annealing epsilon, as the agent should be able to escape local minima and produce less noisy results. However, it is also expected that the overall number of training steps needed, to reach a similar level of performance as the default agent, will be far higher.

4.4 Methodology

The same algorithm is used for both techniques considered, however each method has a different way of determining the new value of epsilon.

The general approach is for the agent to record the total reward at the end of each episode. After a fixed number of episodes, the mean of these rewards is calculated and saved in a different list. If the agent is already using the minimum epsilon value and once two mean rewards have been saved, the gradient between them is calculated. If this gradient is 0 or negative, then the epsilon is triggered to change. The way it is changed depends on the approach;

Stepped Annealing Epsilon (SAE)

The epsilon value is raised up to a pre-determined threshold, and anneals back to the minimum over several steps.

Variable Epsilon (VE)

The epsilon value is set proportionally, based on the average reward gradient, a constant modifier and the maximum epsilon.

4.5 Conclusions

Although the average and best performance of the tested agents over 5,000,000 steps did not surpass the ability of the control agent, there was some notable improvements.

The overall noise within a set of episodes seems lower when using one of the variable epsilon techniques. One reason for this could be that the agent is experiencing a wider range of states which are similar to each other, so learning in a particular localisation is maximised. To fully justify this, a statistical analysis would have to be done, considering the average variation across a rolling section of the results.

Unlike the control agent, the variable epsilon agents suffered less of a drop off after a plateau had been reached in performance. This was likely due to the increased exploration preventing the agents getting stuck in feedback loops.

These techniques showed some promising behaviour and could use further investigation. It would be useful to examine their performance over a larger training trial (perhaps 10,000,000 steps). It would also be worth spending some time tweaking the hyper-parameters specific to the epsilon value management, in order to find the optimum setup.

Chapter 5: Project Evaluations and Conclusion

5.1 Project Outcome

The overall project outcome can be seen as successful; an agent was successfully designed to the required specifications, and it managed to achieve learning on a range of Atari 2600 games with only a high-dimensional visual input.

All of the user stories, for both the user and the agent (section 3.2.1), were also achieved, meaning that the full intended functionality was achieved.

Finally, the outcome of the project objectives (Appendix C) can be considered individually;

Ref	Title	Evaluation Criteria
1	Nonlinear Function Approximator	A Deep Q Network was designed and verified. See Sections; 3.2.2 and 3.4
2	Experience Replay Implementation	An experience replay class was created and integrated into the final agent. A variable number of experiences could be reordered and sampled from. Optimisations were also achieved during Validation and Verification. See Sections 3.3.4 and 3.4
3	Deep Learning Agent Development	Combining all the necessary components into a working Deep Q Agent was achieved, and the agent was able to achieve stable learning across a range of games. See Section 3.3
4	Learning Performance Analysis	The created agent was tested across a range of games, the results were recorded, compared to similar research and evaluations and conclusions were drawn. See Section 3.4.4
5	Investigate Novel Technique for Agent Improvement	Two new techniques were developed, tested, analysed and written up in a Springer styled academic paper. See Chapter 4, Appendix H

5.2 Post-Mortem

The start of this project was the most difficult section. It was hard to decide what was the best approach, and with the amount of freedom available, making definitive decisions was difficult. For example, although I knew that I wanted to examine deep reinforcement learning, it was tough to settle on a specific algorithm and problem domain. This indecision at the start of the project led to unclear and poorly defined requirements. Due to this there was a lot of change during the start of the project, and week on week the goal posts seemed to shift. I should have been stricter about defining the core of the project, and this would have helped make my initial requirements much more accurate.

Although the potential risk of large training times was considered from the start of this project, it was not until the design phase that tests were performed to work out an estimate for training times. This meant that the scope of the project had to adapt by reducing the number of steps in a training trial and planning less tests. Ideally, these estimates should have been carried out during the project planning stage, so that a better schedule could have been produced. This would also have helped in reducing changing requirements.

The two main APIs of the project were great to work with, and the time spent in selecting them paid off. They both provided exactly the required functionality for this project, and demonstrated that a detailed software selection policy is worth the time spent.

Producing user stories from both the perspective of the end-user and the agent was very useful in breaking down all the required functionalities of the program. They were periodically used like a check list to ensure that all the required program behaviour was being developed as needed.

Thanks to advice from my project supervisor, I was careful to not get dragged into spending too much time tuning hyper-parameters. Once the program was working successfully, there was a large temptation to tweak these parameters for improved performance.

Overall, I enjoyed working on this project and I learnt a lot about the field of reinforcement learning. It also opened my eyes to the advantages of good project management practices. It was also interesting to look deeper into performing academic level research and producing formal papers.

References

Bellemare, G. M et al (2012). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research,* 47:253–279, 2013.

Chaslot, G. M. J. B et al. (2007), Progressive Strategies for Monte-Carlo Tree Search.'Information Sciences 2007', WORLD SCIENTIFIC, pp. 655--661.

Heinrich, J. & Silver, D. (2016), 'Deep Reinforcement Learning from Self-Play in Imperfect-Information Games'.

Kingma, D. P. & Ba, J. (2014), 'Adam: A Method for Stochastic Optimization'.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D. & Riedmiller, M. (2013), 'Playing Atari with Deep Reinforcement Learning'.

Mnih, V et al. (2015), 'Human-level control through deep reinforcement learning', *Nature* **518**(7540), 529--533.

Riedmiller, M. (2005), Neural Fitted Q Iteration First Experiences with a Data Efficient Neural Reinforcement Learning Method' Machine Learning: ECML 2005', Springer Berlin Heidelberg, pp. 317--328.

Silver, D et al (2017), 'Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm'.

Tokic, M. (2010), Adaptive-Greedy Exploration in Reinforcement Learning Based on Value Differences' KI 2010: Advances in Artificial Intelligence', Springer Berlin Heidelberg, pp. 203--210.

Unit Testing Framework (2018) — Python 3.6.5 documentation. [ONLINE] Available at: https://docs.python.org/3/library/unittest.html. [Accessed 27 April 2018].

Van Hasselt, H.; Guez, A. & Silver, D. (2015), 'Deep Reinforcement Learning with Double Q-learning'.

Wang, Z.; Schaul, T.; Hessel, M.; van Hasselt, H.; Lanctot, M. & de Freitas, N. (2015), 'Dueling Network Architectures for Deep Reinforcement Learning'.

Wiering, M. A. (2010), 'Self-Play and Using an Expert to Learn to Play Backgammon with Temporal Difference Learning', *Journal of Intelligent Learning Systems and Applications* **02**(02), 57--68.

Yannakakis, G. N. & Togelius, J. (2018), *Artificial Intelligence and Games*, Springer.

Zhang, S. & Sutton, R. S. (2017), 'A Deeper Look at Experience Replay', arXiv:1712.01275v3.

Appendices

A) User Guide

Deep Reinforcement Agent for Atari 2600

This program is for performing deep reinforcement learning experiments on the Atari 2600. The program can launch a range of Atari games, and then train an agent how to play using a pre-determined set of hyper parameters. The user can determine several program parameters based on the experiment they wish to carry out.

Program Requirements

This program requires the following software to be installed

- Python 3.6 or later (Recommended: Anaconda for easy installation)
- *(Python Package)* Gym
- *(Python Package)* Tensorflow
- Microsoft Build Tools for Visual Studio 2017 (or 2015)

Windows Installation Procedure

To prepare a Windows system to run this program, the following steps should be followed;

With Anaconda installed on the machine, open a Windows Command Prompt

At the command prompt, enter the following commands;

```
conda install git
conda update --all
conda clean -a
pip install git+https://github.com/Kojoley/atari-py.git
pip install gym[atari]
pip install --upgrade tensorflow
```

If you have a Nvidia GPU and want to optimise the performance of the program, then the GPU version of tensorflow should be installed. This requires a few extra downloads and configuration steps which can be found in Tensorflow's documentation;

https://www.tensorflow.org/install/install_windows

Launching a Training Session

Download all the required project files and save them in a directory of your choice.

Open a Windows Command Prompt and navigate to the directory

```
C:\Users\John> cd desktop\agent

C:\Users\John\Desktop\Agent>
```

A basic training session can be launched with the following command

```
C:\Users\John\Desktop\Agent> python agent.py
```

This will launch the program with all the default settings, and ask the user which game they wish to test on. Type the name of the game you wish to test and press enter to begin.

Command Line Arguments

For complex control of the program several command line arguments can be used to set certain parameters;

Name	Abbreviation	Description	Default Value
--gameName	-gn	Name of the game to be loaded into the program	N/A
--episodeCount	-ec	The number of episodes to be ran in this trial	10
--render	-r	Should the game be rendered to the screen?	False
--noTrain	-nt	Should the model be trained?	True
--save	-s	Should the model be saved?	False
--load	-l	Load a trained model by episode number. [1]	0

An example of using these arguments to launch an experiment on the game "Breakout" for 1000 episodes, with the game being rendered to screen and the model being saved would look like;

```
C:\Users\John\Desktop\Agent> python agent.py -gn Breakout -ec 1000 -r -s
```

Performing Unit Tests

A series of unit tests are bundled with the program to ensure expected functionality.

These tests can be initiated from the Windows Command Line of the program location;

```
C:\Users\John\Desktop\Agent> python unittests.py
```

Notable File Locations

Within the main directory there are two folders, *"DQNModels"* and *"Results"*.

[1] *"DQNModels"* contains any saved models and the file names are referenced by episode name so they can easily be loaded. This folder is already populated with some trained models to demonstrate the agent ability.

"Results" contains the saved data from each episode completed within the program. The results files are .csv files which can be easily read and formatted by Excel. Each file is named based on the game name and the date of the performed experiment.

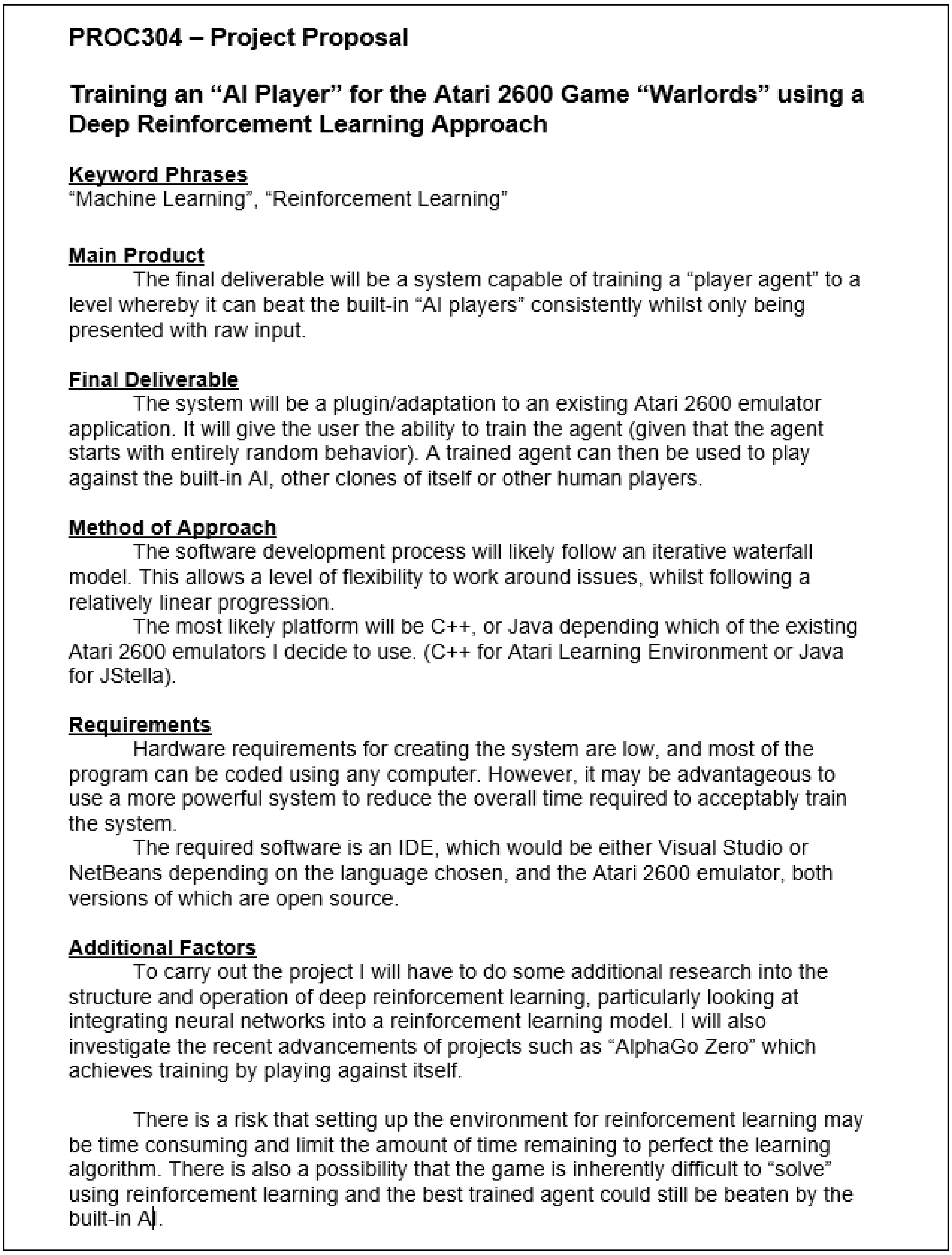
List of Compatible Atari 2600 Games

A full list of game names compatible with this program are;

- Alien
- Asterix
- Asteroids
- Atlantis
- BattleZone
- BeamRider
- Berzerk
- Bowling
- Boxing
- Breakout
- ChopperCommand
- CrazyClimber
- Defender
- DemonAttack
- ElevatorAction
- Enduro
- FishingDerby
- Freeway
- FrostBite
- Gravitar
- Hero
- IceHockey
- Jamesbond
- Kangaroo
- Kaboom
- Krull
- KungFuMaster
- MontezumaRevenge
- MsPacman
- NameThisGame
- Phoenix
- Pitfall
- Pong
- PrivateEye
- Qbert
- Riverraid
- RoadRunner
- Robotank
- Seaquest
- Solaris
- SpaceInvaders
- StarGunner
- TimePilot
- UpNDown
- Venture
- YarsRevenge
- Zaxxon

B) Project Management Documents

B1) Project Proposal



PROC304 – Project Proposal

Training an "AI Player" for the Atari 2600 Game "Warlords" using a Deep Reinforcement Learning Approach

Keyword Phrases
"Machine Learning", "Reinforcement Learning"

Main Product
The final deliverable will be a system capable of training a "player agent" to a level whereby it can beat the built-in "AI players" consistently whilst only being presented with raw input.

Final Deliverable
The system will be a plugin/adaptation to an existing Atari 2600 emulator application. It will give the user the ability to train the agent (given that the agent starts with entirely random behavior). A trained agent can then be used to play against the built-in AI, other clones of itself or other human players.

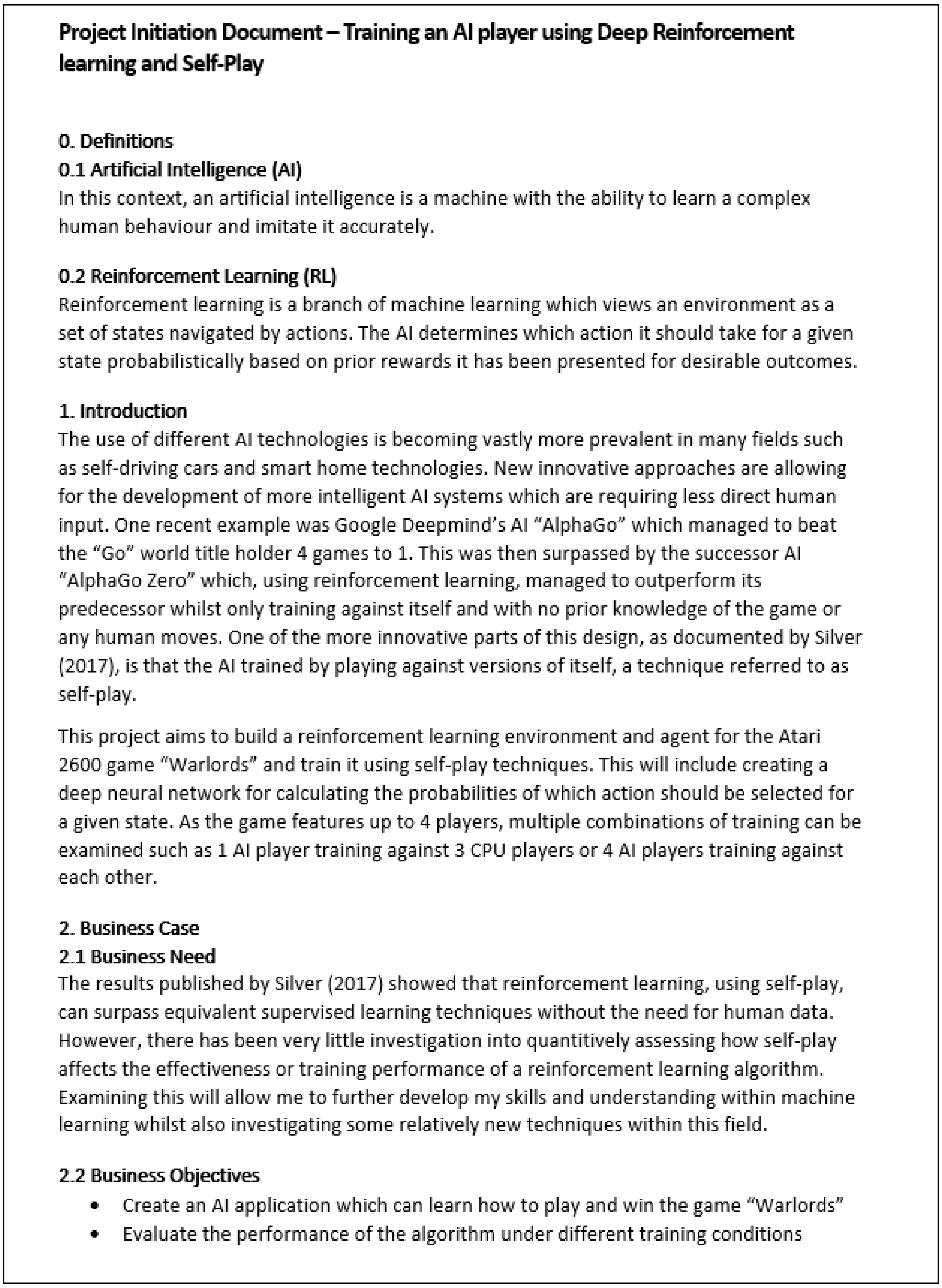
Method of Approach
The software development process will likely follow an iterative waterfall model. This allows a level of flexibility to work around issues, whilst following a relatively linear progression.
The most likely platform will be C++, or Java depending which of the existing Atari 2600 emulators I decide to use. (C++ for Atari Learning Environment or Java for JStella).

Requirements
Hardware requirements for creating the system are low, and most of the program can be coded using any computer. However, it may be advantageous to use a more powerful system to reduce the overall time required to acceptably train the system.
The required software is an IDE, which would be either Visual Studio or NetBeans depending on the language chosen, and the Atari 2600 emulator, both versions of which are open source.

Additional Factors
To carry out the project I will have to do some additional research into the structure and operation of deep reinforcement learning, particularly looking at integrating neural networks into a reinforcement learning model. I will also investigate the recent advancements of projects such as "AlphaGo Zero" which achieves training by playing against itself.

There is a risk that setting up the environment for reinforcement learning may be time consuming and limit the amount of time remaining to perfect the learning algorithm. There is also a possibility that the game is inherently difficult to "solve" using reinforcement learning and the best trained agent could still be beaten by the built-in AI.

B2) Project Initiation Document (PID)



Project Initiation Document – Training an AI player using Deep Reinforcement learning and Self-Play

0. Definitions

0.1 Artificial Intelligence (AI)

In this context, an artificial intelligence is a machine with the ability to learn a complex human behaviour and imitate it accurately.

0.2 Reinforcement Learning (RL)

Reinforcement learning is a branch of machine learning which views an environment as a set of states navigated by actions. The AI determines which action it should take for a given state probabilistically based on prior rewards it has been presented for desirable outcomes.

1. Introduction

The use of different AI technologies is becoming vastly more prevalent in many fields such as self-driving cars and smart home technologies. New innovative approaches are allowing for the development of more intelligent AI systems which are requiring less direct human input. One recent example was Google Deepmind's AI "AlphaGo" which managed to beat the "Go" world title holder 4 games to 1. This was then surpassed by the successor AI "AlphaGo Zero" which, using reinforcement learning, managed to outperform its predecessor whilst only training against itself and with no prior knowledge of the game or any human moves. One of the more innovative parts of this design, as documented by Silver (2017), is that the AI trained by playing against versions of itself, a technique referred to as self-play.

This project aims to build a reinforcement learning environment and agent for the Atari 2600 game "Warlords" and train it using self-play techniques. This will include creating a deep neural network for calculating the probabilities of which action should be selected for a given state. As the game features up to 4 players, multiple combinations of training can be examined such as 1 AI player training against 3 CPU players or 4 AI players training against each other.

2. Business Case

2.1 Business Need

The results published by Silver (2017) showed that reinforcement learning, using self-play, can surpass equivalent supervised learning techniques without the need for human data. However, there has been very little investigation into quantitively assessing how self-play affects the effectiveness or training performance of a reinforcement learning algorithm. Examining this will allow me to further develop my skills and understanding within machine learning whilst also investigating some relatively new techniques within this field.

2.2 Business Objectives

- Create an AI application which can learn how to play and win the game "Warlords"
- Evaluate the performance of the algorithm under different training conditions

3. Project Objectives

Order	Title	Description
1	Literature Review	Investigation of existing reinforcement learning techniques through a literature review. A focus will be taken on Deep Q Networks, Monte Carlo tree search and batch updating theory.
2	Deep Learning RL Agent Prototype	Creation of a RL agent prototype using deep neural networks for probability look up. With the ability to be trained against the built-in CPU players in the game.
3	Self-Play RL Agent Development	Adaption of prototype system to allow for simultaneous self-play training for up to 4 AI players.
4	Testing and Comparison	Performing a series of experiments to test the ability of the algorithm (speed to win a game, total games won over time etc), and the ability of the algorithm with respect to training time required. Tests will be repeated using different numbers of AI players using self-play and the results will be compared and contrasted.
5	Creation of paper on effect of self-play training for reinforcement learning algorithms	A paper will be written describing the results of the testing stage and drawing conclusions on the usefulness and adaptability of self-play training.

4. Initial scope

- Core
  - Design the structure of the RL algorithm including functions for state transition and rewards.
  - Design the deep neural network for state based probability calculation using a Monte Carlo search tree
  - Build the prototype agent based on chosen designs.
  - Integrate the agent with the Atari 2600 emulator using a reinforcement learning API such as JStella.
  - Build the final RL agent including improvements and fixes from the prototype as well as the ability to train through self-play
  - Test at least two different training patterns and compare them using metrics such as those discussed in the project objectives.
  - Write the paper discussing the use of self-play in training reinforcement learning agents

- Optional
    - Consider a wider range of training probabilities to give more diverse results when evaluating self-play

5. Method of approach

The project will start with the in-depth literature review to ensure that the correct level of knowledge into the subject is acquired before design begins.

The software development process will likely follow an iterative waterfall model. This allows a level of flexibility to work around issues, whilst following a relatively linear progression.

6. Initial project plan

Stage	Expected Start Date	Expected Completion Date	Products/Deliverables/Outcomes
Initiation		02/02	PID
Literature Review	03/02	10/02	Literature review write up detailing work carried out in the field relevant to the design required in this project.
Deep Learning RL Agent Prototype	11/02	09/03	Software demo which can be trained to improve performance and successfully interact with the emulator.
Self-Play RL Agent Development	10/03	09/03	Final software product which can train against itself whilst integrated with the emulator.
Testing and Comparison	10/03	24/03	Test results, graphs, tables and descriptions of the experiment parameters used.
Creation of paper on effect of self-play training for reinforcement learning algorithms	25/03	13/04	Paper on effect of self-play training for reinforcement learning algorithms
Final report writeup	14/04	19/05	Final report writeup along with all required appendices

6.1. Communication plan

Each week highlight reports will be sent to the project supervisor as is the requirement of the PRCO304 module. A further face to face meeting will also take place once a week for at least the first term of the year.

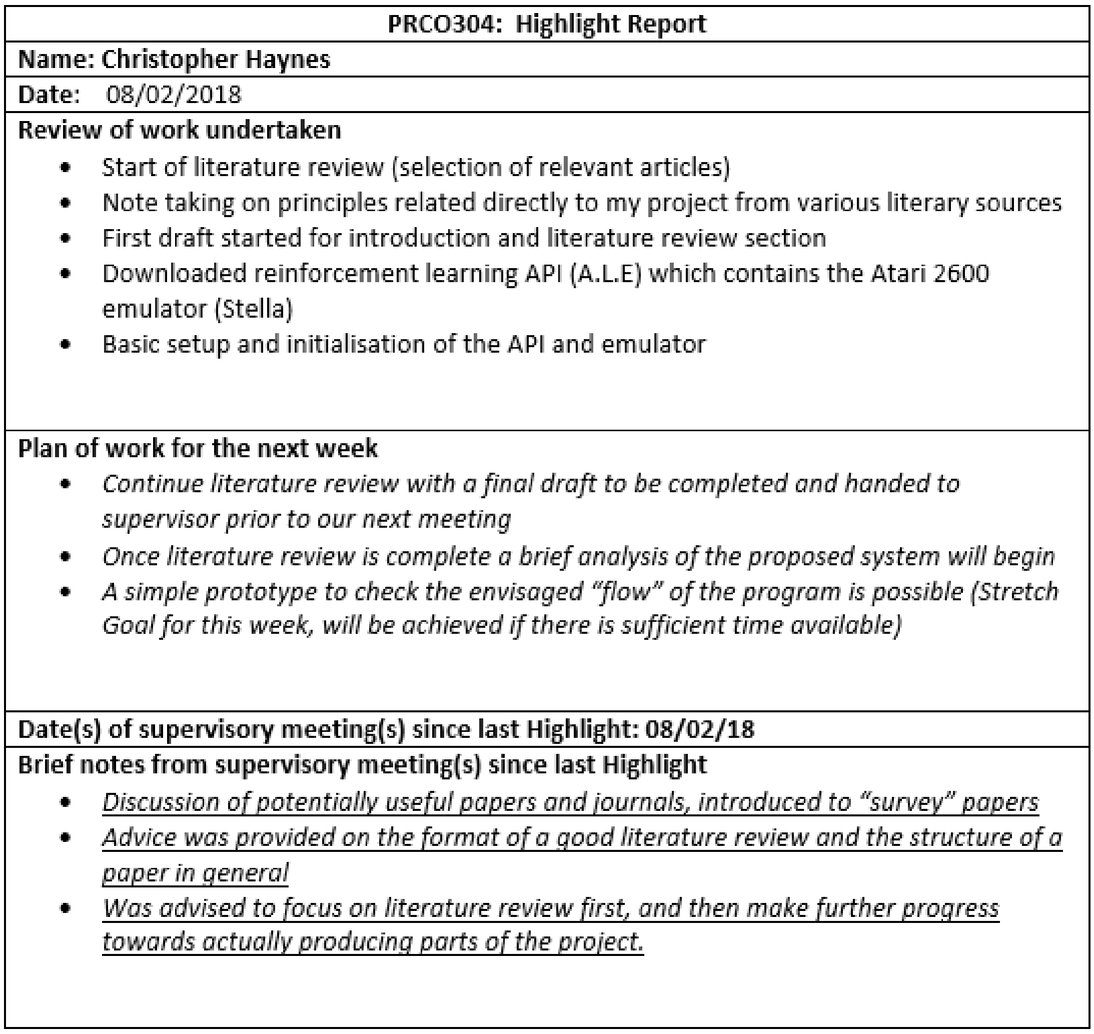
7. Initial risk list

Risk	Management Strategy
Schedule Overrun	The original schedule will be examined often to ensure that timings are met. Any changes or deviations from the plan will cause the project supervisor to create an exception plan in the event of more than one week's slippage.
Difficulty learning/using the development technologies	Prototyping the initial agent will allow for the project to progress in smaller steps with the aim of making the difficult technologies more digestible.
Large training times required	An alternative computer is available to run 24/7 training sessions, if required, to allow for multiple agent combinations to be trained at the same time across two machines.
Technology Failure	The program code will be managed with Github and will follow the standard Git branching policy. Additional manual back up of files and program code will be made to a cloud service and secondary hard disk.

8. References

Silver,D et al. (2017): Mastering the game of Go without human knowledge. In: Nature, Jg. 550, Nr. 7676, p. 354 - 359.

B3) Highlight Reports



PRCO304: Highlight Report

Name: Christopher Haynes

Date: 08/02/2018

Review of work undertaken

- Start of literature review (selection of relevant articles)
- Note taking on principles related directly to my project from various literary sources
- First draft started for introduction and literature review section
- Downloaded reinforcement learning API (A.L.E) which contains the Atari 2600 emulator (Stella)
- Basic setup and initialisation of the API and emulator

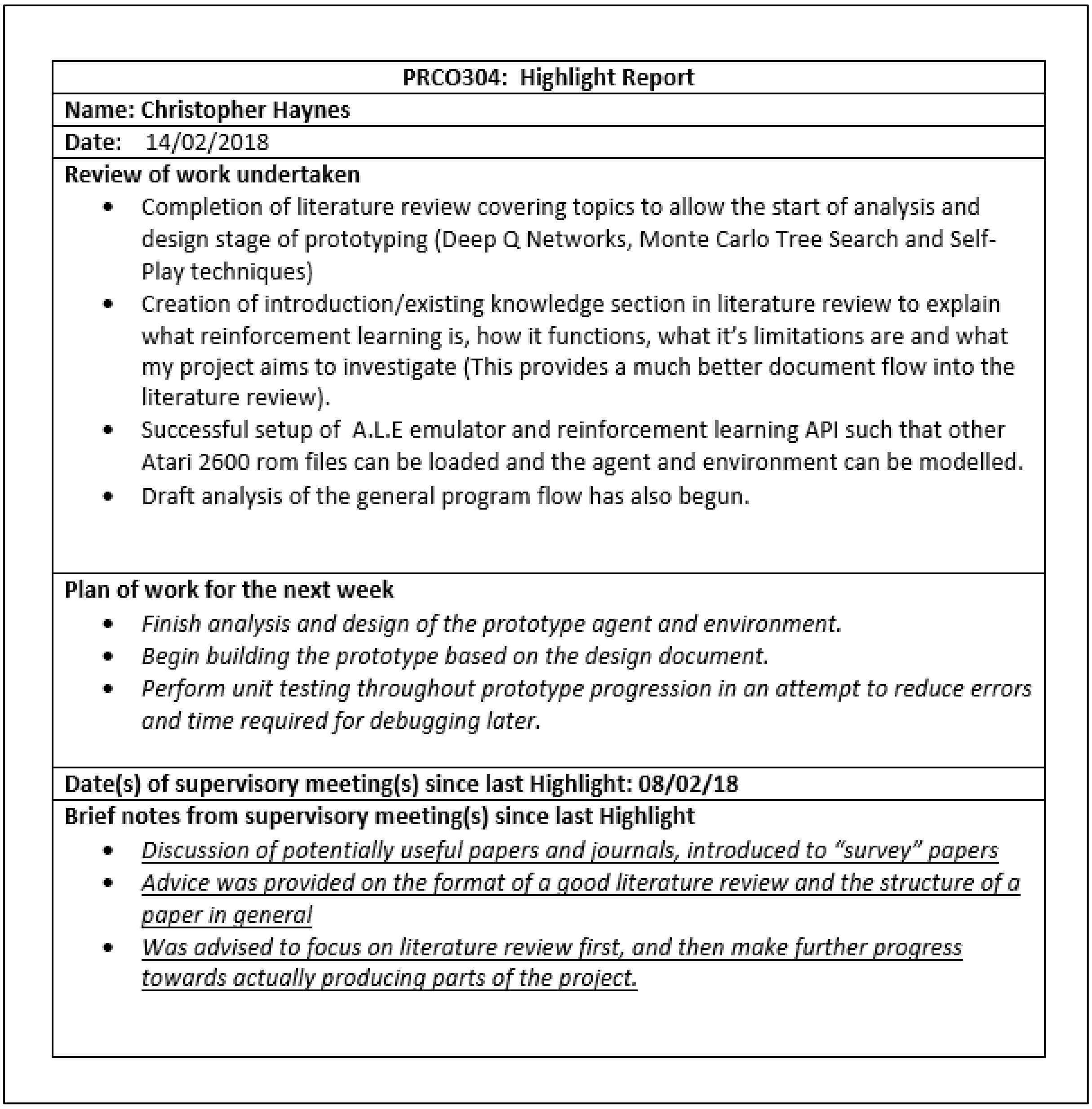
Plan of work for the next week

- *Continue literature review with a final draft to be completed and handed to supervisor prior to our next meeting*
- *Once literature review is complete a brief analysis of the proposed system will begin*
- *A simple prototype to check the envisaged "flow" of the program is possible (Stretch Goal for this week, will be achieved if there is sufficient time available)*

Date(s) of supervisory meeting(s) since last Highlight: 08/02/18

Brief notes from supervisory meeting(s) since last Highlight

- *Discussion of potentially useful papers and journals, introduced to "survey" papers*
- *Advice was provided on the format of a good literature review and the structure of a paper in general*
- *Was advised to focus on literature review first, and then make further progress towards actually producing parts of the project.*



PRCO304: Highlight Report
Name: Christopher Haynes
Date: 14/02/2018

Review of work undertaken

- Completion of literature review covering topics to allow the start of analysis and design stage of prototyping (Deep Q Networks, Monte Carlo Tree Search and Self-Play techniques)
- Creation of introduction/existing knowledge section in literature review to explain what reinforcement learning is, how it functions, what it's limitations are and what my project aims to investigate (This provides a much better document flow into the literature review).
- Successful setup of A.L.E emulator and reinforcement learning API such that other Atari 2600 rom files can be loaded and the agent and environment can be modelled.
- Draft analysis of the general program flow has also begun.

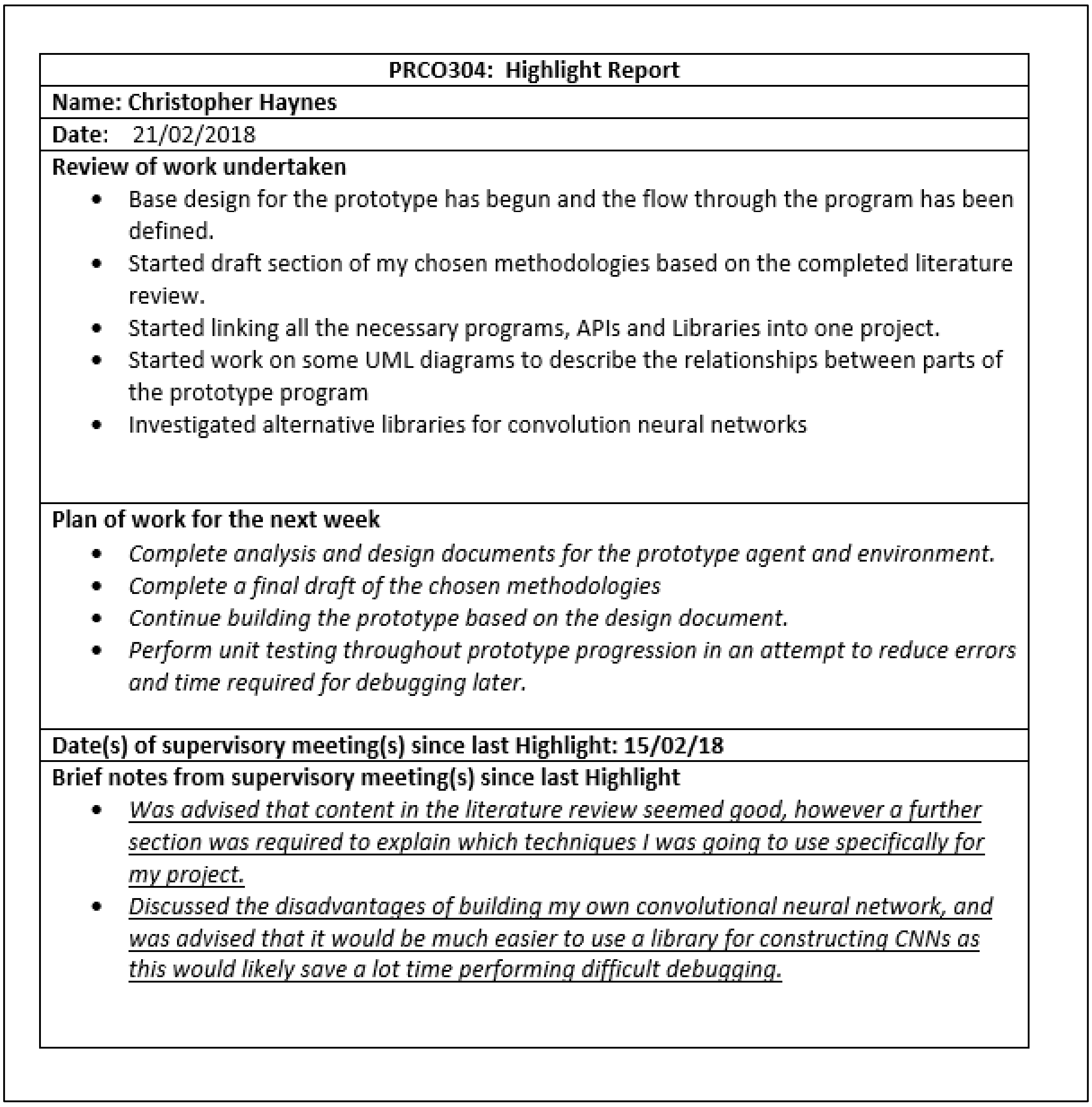
Plan of work for the next week

- *Finish analysis and design of the prototype agent and environment.*
- *Begin building the prototype based on the design document.*
- *Perform unit testing throughout prototype progression in an attempt to reduce errors and time required for debugging later.*

Date(s) of supervisory meeting(s) since last Highlight: 08/02/18

Brief notes from supervisory meeting(s) since last Highlight

- *Discussion of potentially useful papers and journals, introduced to "survey" papers*
- *Advice was provided on the format of a good literature review and the structure of a paper in general*
- *Was advised to focus on literature review first, and then make further progress towards actually producing parts of the project.*



PRCO304: Highlight Report

Name: Christopher Haynes

Date: 21/02/2018

Review of work undertaken

- Base design for the prototype has begun and the flow through the program has been defined.
- Started draft section of my chosen methodologies based on the completed literature review.
- Started linking all the necessary programs, APIs and Libraries into one project.
- Started work on some UML diagrams to describe the relationships between parts of the prototype program
- Investigated alternative libraries for convolution neural networks

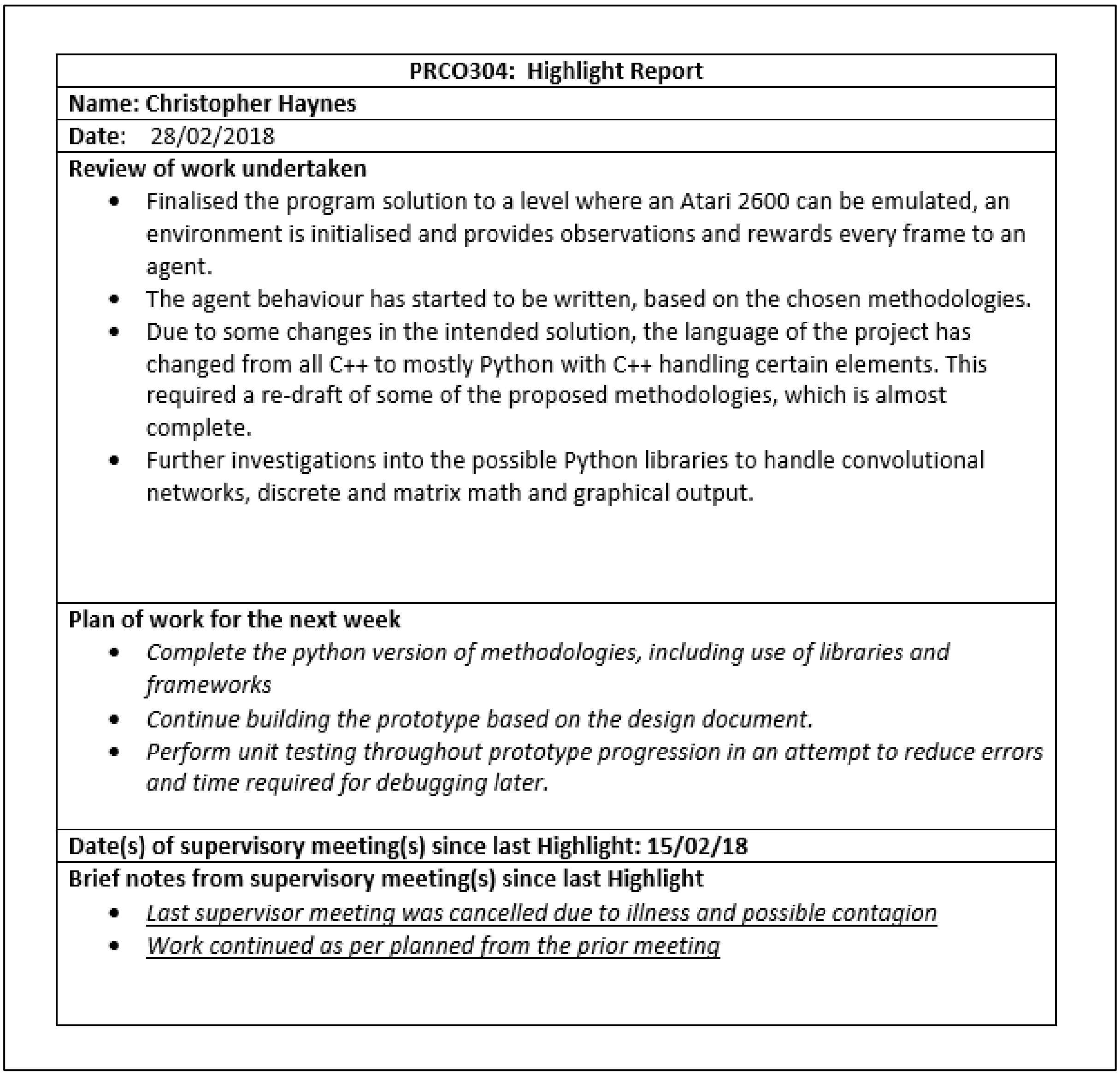
Plan of work for the next week

- *Complete analysis and design documents for the prototype agent and environment.*
- *Complete a final draft of the chosen methodologies*
- *Continue building the prototype based on the design document.*
- *Perform unit testing throughout prototype progression in an attempt to reduce errors and time required for debugging later.*

Date(s) of supervisory meeting(s) since last Highlight: 15/02/18

Brief notes from supervisory meeting(s) since last Highlight

- *Was advised that content in the literature review seemed good, however a further section was required to explain which techniques I was going to use specifically for my project.*
- *Discussed the disadvantages of building my own convolutional neural network, and was advised that it would be much easier to use a library for constructing CNNs as this would likely save a lot time performing difficult debugging.*



PRCO304: Highlight Report

Name: Christopher Haynes

Date: 28/02/2018

Review of work undertaken

- Finalised the program solution to a level where an Atari 2600 can be emulated, an environment is initialised and provides observations and rewards every frame to an agent.
- The agent behaviour has started to be written, based on the chosen methodologies.
- Due to some changes in the intended solution, the language of the project has changed from all C++ to mostly Python with C++ handling certain elements. This required a re-draft of some of the proposed methodologies, which is almost complete.
- Further investigations into the possible Python libraries to handle convolutional networks, discrete and matrix math and graphical output.

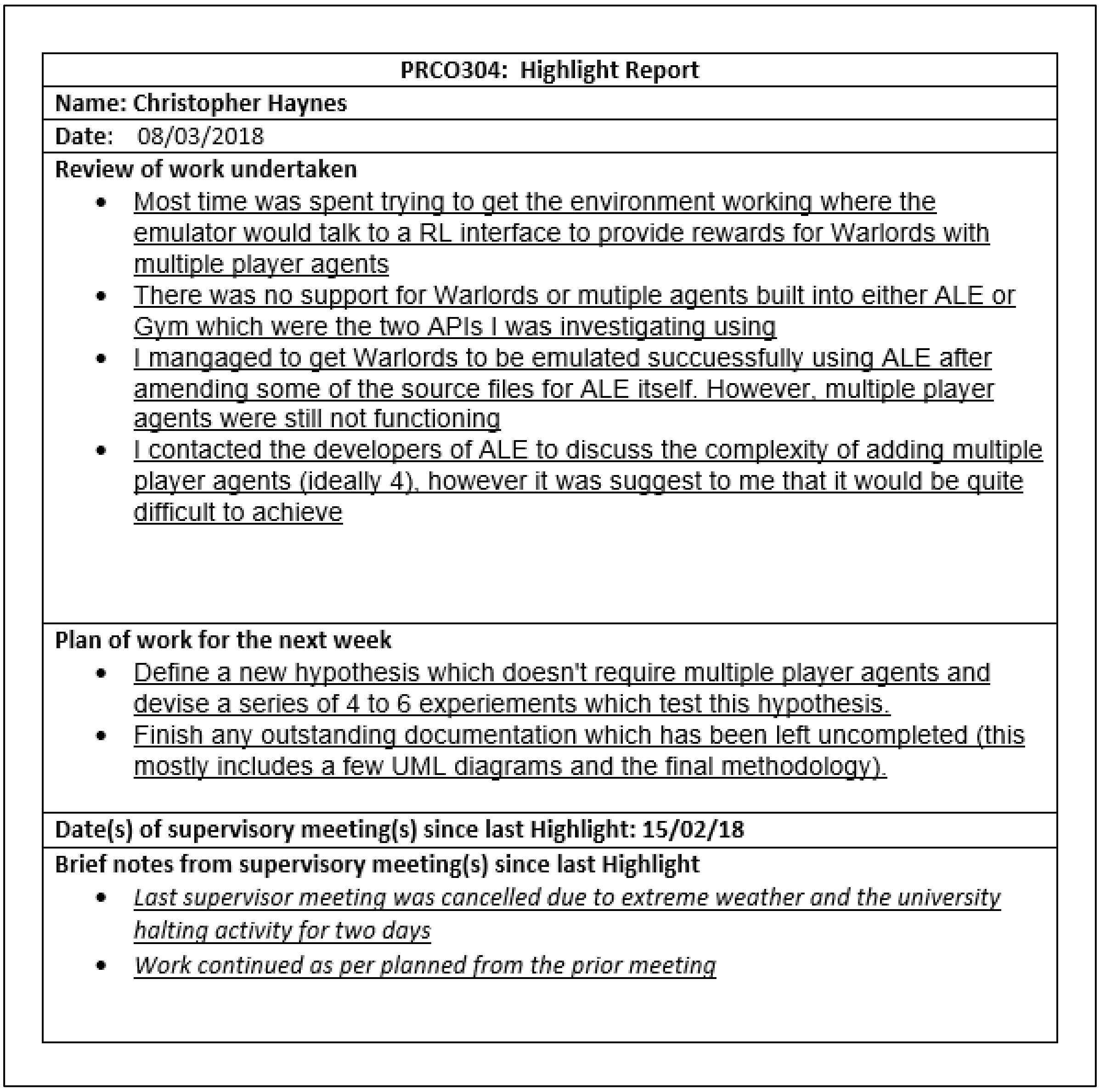
Plan of work for the next week

- *Complete the python version of methodologies, including use of libraries and frameworks*
- *Continue building the prototype based on the design document.*
- *Perform unit testing throughout prototype progression in an attempt to reduce errors and time required for debugging later.*

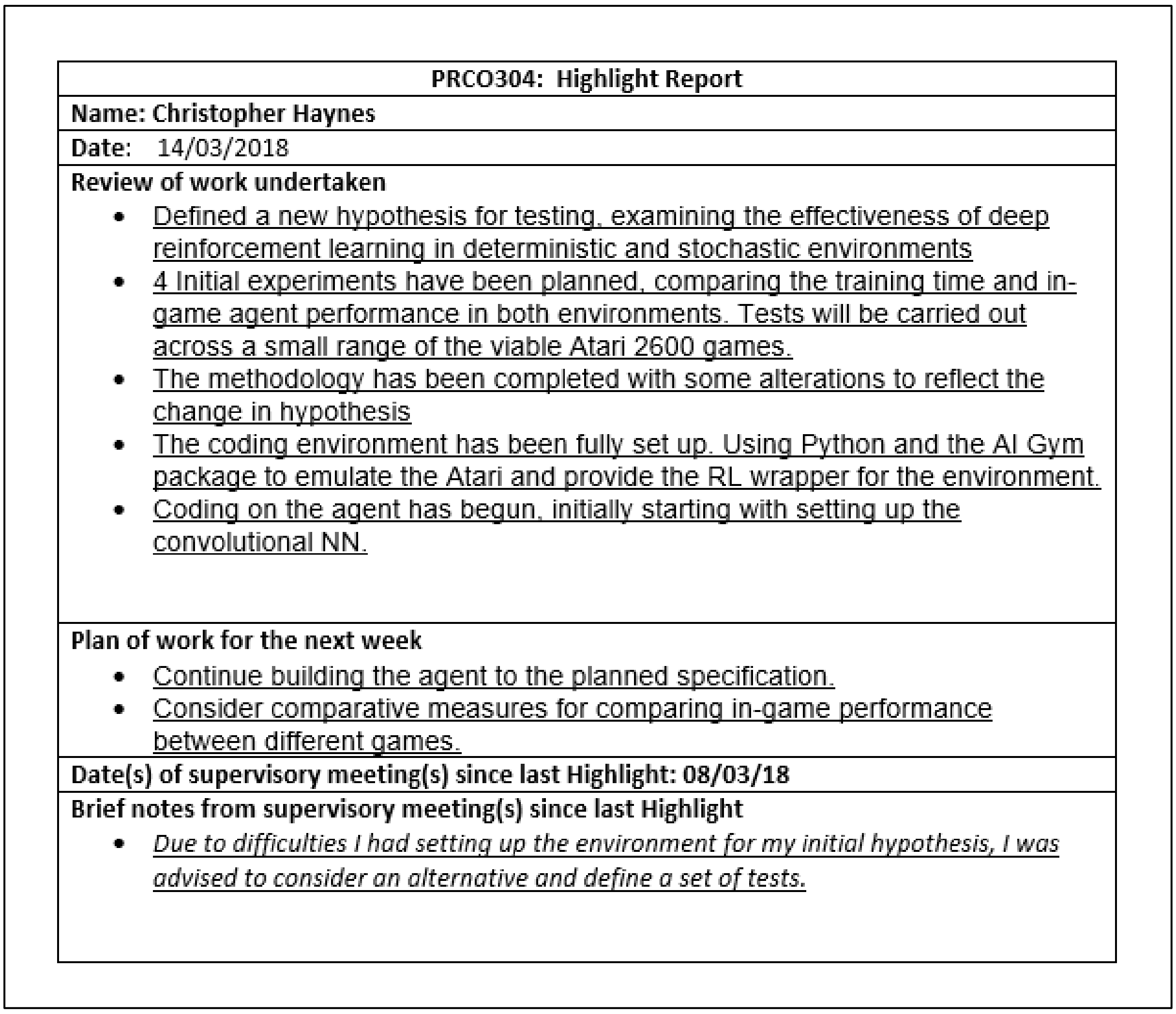
Date(s) of supervisory meeting(s) since last Highlight: 15/02/18

Brief notes from supervisory meeting(s) since last Highlight

- *Last supervisor meeting was cancelled due to illness and possible contagion*
- *Work continued as per planned from the prior meeting*



PRCO304: Highlight Report
Name: Christopher Haynes
Date: 08/03/2018
Review of work undertaken • Most time was spent trying to get the environment working where the emulator would talk to a RL interface to provide rewards for Warlords with multiple player agents • There was no support for Warlords or mutiple agents built into either ALE or Gym which were the two APIs I was investigating using • I mangaged to get Warlords to be emulated succuessfully using ALE after amending some of the source files for ALE itself. However, multiple player agents were still not functioning • I contacted the developers of ALE to discuss the complexity of adding multiple player agents (ideally 4), however it was suggest to me that it would be quite difficult to achieve
Plan of work for the next week • Define a new hypothesis which doesn't require multiple player agents and devise a series of 4 to 6 experiements which test this hypothesis. • Finish any outstanding documentation which has been left uncompleted (this mostly includes a few UML diagrams and the final methodology).
Date(s) of supervisory meeting(s) since last Highlight: 15/02/18
Brief notes from supervisory meeting(s) since last Highlight • *Last supervisor meeting was cancelled due to extreme weather and the university halting activity for two days* • *Work continued as per planned from the prior meeting*



PRCO304: Highlight Report

Name: Christopher Haynes

Date: 14/03/2018

Review of work undertaken

- Defined a new hypothesis for testing, examining the effectiveness of deep reinforcement learning in deterministic and stochastic environments
- 4 Initial experiments have been planned, comparing the training time and in-game agent performance in both environments. Tests will be carried out across a small range of the viable Atari 2600 games.
- The methodology has been completed with some alterations to reflect the change in hypothesis
- The coding environment has been fully set up. Using Python and the AI Gym package to emulate the Atari and provide the RL wrapper for the environment.
- Coding on the agent has begun, initially starting with setting up the convolutional NN.

Plan of work for the next week

- Continue building the agent to the planned specification.
- Consider comparative measures for comparing in-game performance between different games.

Date(s) of supervisory meeting(s) since last Highlight: 08/03/18

Brief notes from supervisory meeting(s) since last Highlight

- *Due to difficulties I had setting up the environment for my initial hypothesis, I was advised to consider an alternative and define a set of tests.*

PRCO304: Highlight Report
Name: Christopher Haynes
Date: 21/03/2018
Review of work undertaken • The conv neural net prototype has been built. It was tested on a standardised constant set of data to ensure consistency. • The RL agent can now input a matrix of pixels and received reward and be returned a probability range over a set of actions. • Worked on improving neural net training from the agent. • Started structure of Monte Carlo Tree Search
Plan of work for the next week • Continue building the agent to the planned specification. • Begin experiments and start recording results • Start considering the structure of the final report
Date(s) of supervisory meeting(s) since last Highlight: 08/03/18
Brief notes from supervisory meeting(s) since last Highlight • *I was advised to test components of the project, such as the neural net, using a constant set of data.* • *I should be careful in my report to make distinctions between the software engineering aspect and the research aspect of the project.*

C) Project Objectives

Ref	Title	Descriptions	Requirements
1	Nonlinear Function Approximator	Consider Deep Reinforcement literature to assist in discovering and designing a Q Value estimator.	See Chapter 2.1 “Deep Q Networks”
2	Experience Replay Implementation	Design a structure that can hold a predetermined number of agent “experiences”. These experiences can then be used in random batches to train the nonlinear function approximator.	See Chapter 2 “Literature Review” and Chapter 3.3.4 “Experience Replay”
3	Deep Learning Agent Development	Create a full reinforcement learning agent, capable of achieving learning from only the raw screen output of the Atari 2600. It should use the previously developed elements as components.	See Chapter 3.2.1 “User Stories”
4	Learning Performance Analysis	Perform verification and validation on the agent, testing that it can learn across a range of games. Record detailed results and compare them to equivalent agents in academic research.	See Chapter 3.4.4 “Agent Ability and Training Efficiency Tests”
5	Investigate Novel Technique for Agent Improvement	Create a novel technique for improving the learning ability of an agent. Implement the technique, record relevant results and write up the experiment in a LNCS Springer style paper	See Chapter 4 “Exploration vs Exploitation Experiment”

D) CNN API Selection Document

Tensorflow

Strengths	**Weaknesses**
Developed by Google, and used on similar deep learning projects. Adaptive GPU parallelisation and distributed architecture available “out-of-the-box”. Excellent documentation and wide range of supporting websites, tutorials etc.	Slower than Theano on average. Lack of full control over low-level functions. Less functionality than some competitors.
Opportunities	**Threats**
The GPU parallelisation could help to reduce training times and increase the number of viable experiments. The language seems understandable and strikes a nice balance between high and low-level concepts.	Some useful functionality, that is not known at this stage, could be required. This may involve creating extra functions outside the scope of this project.

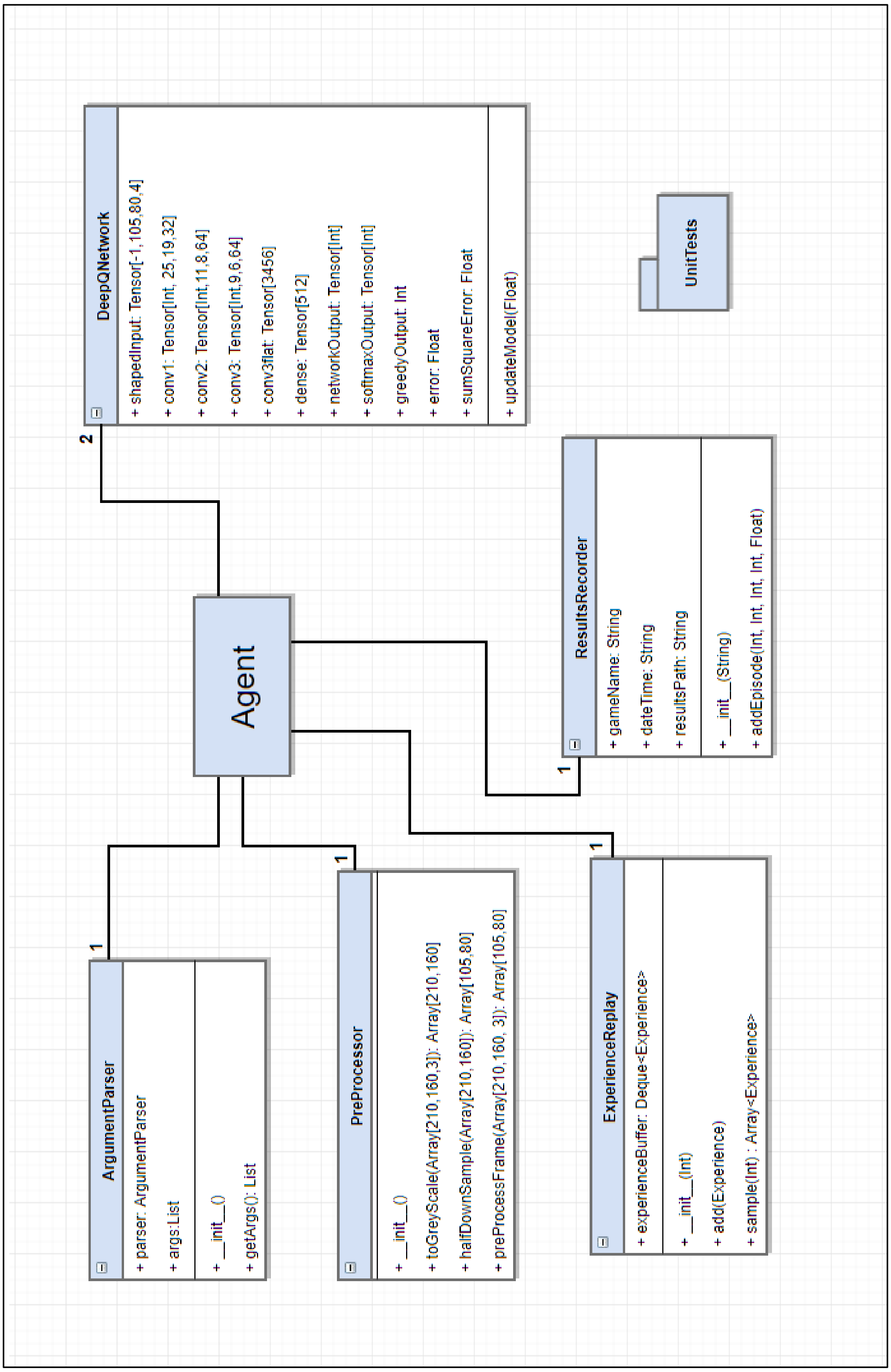
Theano

Strengths	**Weaknesses**
Can perform quick tensor operations. Supports some GPU parallelisation. One of the oldest and most well used deep learning libraries. Good level of supporting documentation.	Low-level language. Code is often produced using a higher-level wrapper and not directly.
Opportunities	**Threats**
The low-level, modular nature could be useful in designing more complex network structures.	The language is complex with little high-level functionality and may require many common functions to be designed from scratch.

Keras

Strengths	**Weaknesses**
Wrapper that allows either a Tensorflow or Theano backend. Modular and minimalistic. High-level language.	Too structured. Not as flexible in terms of variable network design.
Opportunities	**Threats**
Using a high-level wrapper could reduce the complexity of designing the network and save development time.	Too high-level to allow the level of control needed in designing bespoke neural networks.

E) General Architecture Diagram


DeepQNetwork
+ shapedInput: Tensor[-1,105,80,4]
+ conv1: Tensor[Int, 25,19,32]
+ conv2: Tensor[Int,11,8,64]
+ conv3: Tensor[Int,9,6,64]
+ conv3flat: Tensor[3456]
+ dense: Tensor[512]
+ networkOutput: Tensor[Int]
+ softmaxOutput: Tensor[Int]
+ greedyOutput: Int
+ error: Float
+ sumSquareError: Float
+ updateModel(Float)
UnitTests
2
Agent
ResultsRecorder
+ gameName: String
+ dateTime: String
+ resultsPath: String
+ __init__(String)
+ addEpisode(Int, Int, Int, Int, Float)
1
1
1
1
ArgumentParser
+ parser: ArgumentParser
+ args:List
+ __init__()
+ getArgs(): List
PreProcessor
+ __init__()
+ toGreyScale(Array[210,160,3]): Array[210,160]
+ halfDownSample(Array[210,160]): Array[105,80]
+ preProcessFrame(Array[210,160, 3]): Array[105,80]
ExperienceReplay
+ experienceBuffer: Deque<Experience>
+ __init__(Int)
+ add(Experience)
+ sample(Int) : Array<Experience>

F) Manual Testing

Manual Testing Results and Changes						
Ref	**Title**	**Description**	**Data**	**Expected Result**	**Actual Result**	**Changes**
1	Agent Communication with DQN	Does the DQN accept a pre-processed state from the Atari environment?	state = (105, 80, 4)	Output int value in the range of viable control outputs	ValueError raised.	Input was placed in an array before submission to allow for batch size alteration
2			state = [(105, 80, 4)]		Action was output from the network, but still in a tensor object	The output was cast to an int, so it could be used as a valid action choice
3			state = [(105, 80, 4)]		Valid integer returned	None
4		Is a descriptive error message displayed when an input of incorrect shape is presented?	state = [(210, 160, 3)]	ValueError raised with error message describing expected input shape and actual input shape	Expected message returned	None
5	Creating States and maintaining Frame Buffer	Does the frame buffer get no longer than 4 frames?	frameBuffer	When the frame buffer is full, for every new frame added, the oldest frame should be deleted	Buffer occasionally grows beyond the specified bounds	Seperated the processes for random action selection by greedy epsilon and intital behaviour
6			frameBuffer		Buffer size does not grow beyond 4 between steps	None
7		Are frames concantanated into the correct state shape?	frame = (105, 80)	When combining the four frames it should produce a state with the required DQN input shape of (105, 80, 4)	Error. The frames can not be joined as they do not share the same dimensionality with the required output	Extended the dimenson of each frame in the buffer to (105, 80, 1)
8			frame = (105, 80, 1)		The concantination occurs over the wrong axis and produces a (420, 80, 1) output.	Specified which axis to concantante the frames over
9			frame = (105, 80, 1)		Expected state shape of (105, 80, 4) achieved	None

10	Experience Buffer Validation	Can a batch of experiences be split into valid sub-batches for network input?	trainingBatch	Each slice of the training batch should be valid as a relevent DQN batch input	State and nextState both failed due to dimensional mis-match (batchSize, 4)	The sub-batches were stacked to alter the expected shape to (batchsize, 105, 80, 4)
11			trainingBatch		Expected network output, unit test confirms batch updating	None
12	Result Recorder Validation	Is a csv file created at the expected loaction, with the correct name?	N/A	Within the "Results" directory, a new csv file should be created with the game name and current date	Correct file loaction and naming convention	None
13		It should not be possible to overwrite an existing results file under any circumstances	N/A	When creating new files for the same game on the same day, a new file should be made instead of overwritting the exisiting file	Original file is overwritten	Added more detail to the file nameing procedure (itemised numbering) to ensure that no overwritting can occur
14			N/A		New files are created for each test execution, with no overwritting	None
15	Saving and Loading DQN Model Validation	A loaded model should be in the same state as the saved model	valueNetwork, targetNetwork	When loading a model, both networks should be equal on load	Only the valueNetwork is set correctly, the targetNetwork is initialised randomly still	On load, the targetNetwork is updated to match the valueNetwork
16			valueNetwork, targetNetwork		Both networks variables are equal	None
17		A saved model should be stored in the "DQNModels/{gameName}/ directory, and be loadable via the last episode number	valueNetwork, targetNetwork	A model can be saved, and then loaded using the last episode number	The model is saved in the correct location and correctly loaded using the episode reference	None

18	Command Line Arguments	Can the user try to run an experiement with less episodes than the loaded episode number?	episodeCount = 5, LOAD_REF = 10	If the episode count is less than the loaded episode, an exception should occur	The program does not any episodes in the trial	The semantics of the episode count were changed to represent how many episodes are required ontop of the loaded episode
19			episodeCount = 5, LOAD_REF = 10		5 episodes are ran, starting from episode 11 and up to episode 15	None
20		The user should not be able to lanuch the agent with an invalid game name	GAME_NAME = "nonValidName"	If an invalid name is entered then the user must be prompted to enter a valid name before the trial begins	The program attempts to launch the Gym environment and then crashes.	The user input is checked against a list of valid game names to ensure compatability
21			GAME_NAME = "nonValidName"		If the user doesn't know a valid name, then an endless loop can occur	A list of valid names is printed to the console if the user enters an invalid name
22			GAME_NAME = None	If the program is launched with no name provided then a different message is displayed asking the user to select from the list of valid games	There is no way to launch a trial until a valid game name has been selected	None
23		All combinations of flags and command line arguments should be viable	N/A	If the program is launched with any amount of flags it opperates normally. No flags or arguments should contradict.	The program launches as intended under all flag and argument combinations	None

G) Experiment Results

Due to the size of these results, they have been submitted along with the code submission. The full tables can be found at the code URL, in the “Results/FinalResults” directory.

Validation and Verification Test Results

Pong – Pong-FINALRESULTS.csv

Breakout – Breakout-FINALRESULTS.csv

Seaquest – SeaQuest-FINALRESULTS.csv

Ms Pacman – MsPacman-FINALRESULTS.csv

Space Invaders – SpaceInvaders-FINALRESULTS.csv

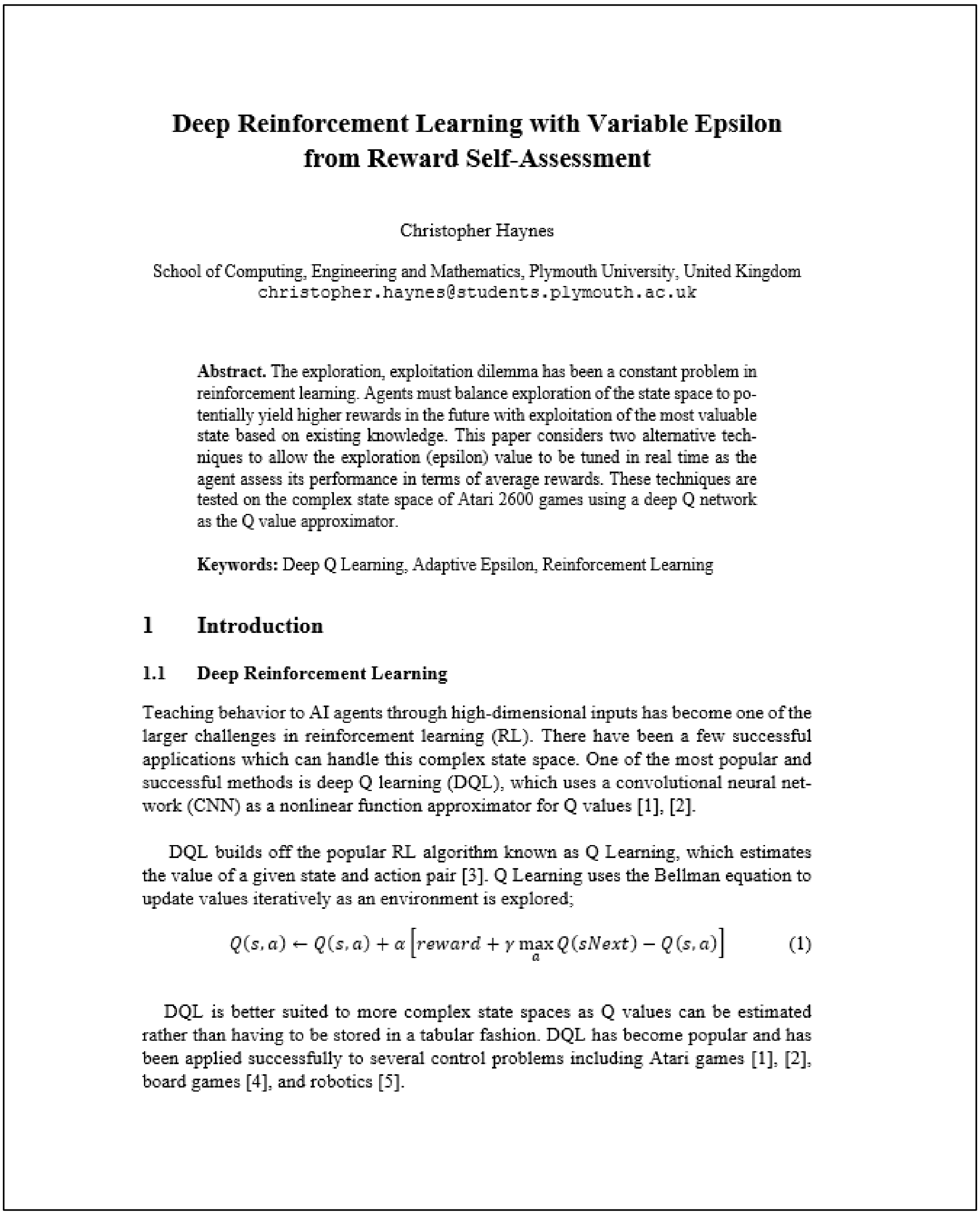
Breakout (No Experience Replay) – Breakout-FINALRESULTS-NO-ER.csv

Epsilon Experiment Results

Stepped Annealing Epsilon Agent – Breakout-SAE-FINALRESULTS.csv

Variable Epsilon Agent – Breakout-VE-FINALRESULTS.csv

H) Springer Styled Paper



Deep Reinforcement Learning with Variable Epsilon from Reward Self-Assessment

Christopher Haynes

School of Computing, Engineering and Mathematics, Plymouth University, United Kingdom
christopher.haynes@students.plymouth.ac.uk

Abstract. The exploration, exploitation dilemma has been a constant problem in reinforcement learning. Agents must balance exploration of the state space to potentially yield higher rewards in the future with exploitation of the most valuable state based on existing knowledge. This paper considers two alternative techniques to allow the exploration (epsilon) value to be tuned in real time as the agent assess its performance in terms of average rewards. These techniques are tested on the complex state space of Atari 2600 games using a deep Q network as the Q value approximator.

Keywords: Deep Q Learning, Adaptive Epsilon, Reinforcement Learning

1 Introduction

1.1 Deep Reinforcement Learning

Teaching behavior to AI agents through high-dimensional inputs has become one of the larger challenges in reinforcement learning (RL). There have been a few successful applications which can handle this complex state space. One of the most popular and successful methods is deep Q learning (DQL), which uses a convolutional neural network (CNN) as a nonlinear function approximator for Q values [1], [2].

DQL builds off the popular RL algorithm known as Q Learning, which estimates the value of a given state and action pair [3]. Q Learning uses the Bellman equation to update values iteratively as an environment is explored;

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[reward + \gamma \max_{a} Q(sNext) - Q(s,a) \right] \qquad (1)$$

DQL is better suited to more complex state spaces as Q values can be estimated rather than having to be stored in a tabular fashion. DQL has become popular and has been applied successfully to several control problems including Atari games [1], [2], board games [4], and robotics [5].

1.2 Exploration, Exploitation Dilemma

The exploration, exploitation dilemma has remained an unresolved issue in the field of RL. An agent should maximize exploration to discover states which yield higher future rewards. At the same time the agent should maximize exploitation of currently known valuable states to maximize the current reward [6]. This conflict is difficult to balance and a range of different approaches have produced varying results [1], [4].

An exploration value (epsilon) is often used to control the probability of how often an agent should take a random move. This value can simply be set as a constant low occurrence (0.1), so that the agent always has a small amount of exploration as it learns.

2 Related Work

2.1 Annealing Epsilon

Other successful approaches to controlling the epsilon value have been investigated with the most popular being annealing epsilon, which is used in a range of RL applications [1], [4], [7].

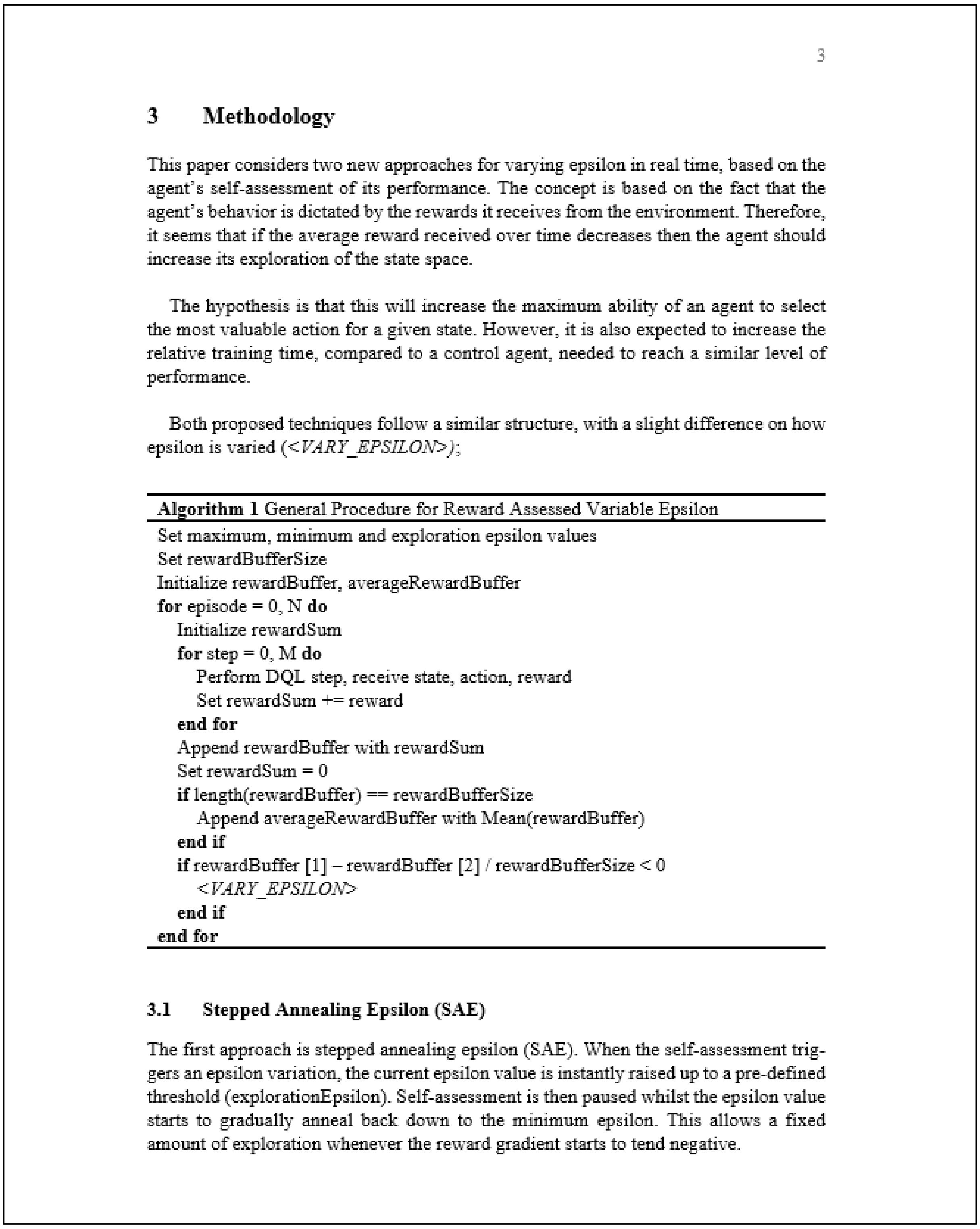
This approach defines a maximum and minimum epsilon value. The agent starts with the maximum value which anneals by a small amount each step until the minimum value is reached. The maximum is often a value of 1.0 and the minimum in the range between 0 and 0.1. This allows for the agent to explore the environment with a variable level of random behavior and experience a wide range of different states. The main disadvantage to this technique is that once the agent reaches the minimum epsilon value, exploration stagnates. This can lead to agents missing the discovery of simple behaviors which can cause learning to plateau.

2.2 Adaptive Epsilon

A variation on annealing epsilon is Adaptive epsilon greedy exploration [8]. This model seeks to adapt the epsilon value in real-time to better balance the levels of exploration and exploitation throughout the training cycle of an agent.

This model adapts the epsilon policy based on the temporal-difference-error calculated when updating the CNN. If the error has a positive gradient, then the epsilon value is proportionally increased to help localized exploration.

This provides the agent with a mostly exploitive behavior, but allows for pockets of exploration when improvements start to dwindle. This can prove advantageous in situations where some states require a level of exploitation to be reached, but still need to be further explored to increase performance.



3 Methodology

This paper considers two new approaches for varying epsilon in real time, based on the agent's self-assessment of its performance. The concept is based on the fact that the agent's behavior is dictated by the rewards it receives from the environment. Therefore, it seems that if the average reward received over time decreases then the agent should increase its exploration of the state space.

The hypothesis is that this will increase the maximum ability of an agent to select the most valuable action for a given state. However, it is also expected to increase the relative training time, compared to a control agent, needed to reach a similar level of performance.

Both proposed techniques follow a similar structure, with a slight difference on how epsilon is varied (*<VARY_EPSILON>)*;

Algorithm 1 General Procedure for Reward Assessed Variable Epsilon

```
Set maximum, minimum and exploration epsilon values
Set rewardBufferSize
Initialize rewardBuffer, averageRewardBuffer
for episode = 0, N do
    Initialize rewardSum
    for step = 0, M do
        Perform DQL step, receive state, action, reward
        Set rewardSum += reward
    end for
    Append rewardBuffer with rewardSum
    Set rewardSum = 0
    if length(rewardBuffer) == rewardBufferSize
        Append averageRewardBuffer with Mean(rewardBuffer)
    end if
    if rewardBuffer [1] – rewardBuffer [2] / rewardBufferSize < 0
        <VARY_EPSILON>
    end if
end for
```

3.1 Stepped Annealing Epsilon (SAE)

The first approach is stepped annealing epsilon (SAE). When the self-assessment triggers an epsilon variation, the current epsilon value is instantly raised up to a pre-defined threshold (explorationEpsilon). Self-assessment is then paused whilst the epsilon value starts to gradually anneal back down to the minimum epsilon. This allows a fixed amount of exploration whenever the reward gradient starts to tend negative.

3.2 Variable Epsilon (VE)

The second approach is variable epsilon (VE). When self-assessment triggers an epsilon variation, the current epsilon is increased at a negative proportional amount of the average reward gradient. This reward then anneals back down to the minimum level before self-assessment continues. This should provide a more accurate response to any drop off in rewards.

$$\varepsilon = -\left(\frac{\delta RewardSum}{\delta SampleSteps} \cdot 2\right) MAX\varepsilon \qquad (2)$$

4 Experiments

Tests were carried out on the complex state space of the Atari 2600. A DQL agent was used as the control example with the two variations created for the approaches under test. The hyper-parameters were kept constant with only the epsilon variables changing between the different agents under test.

Table 1. Constant Hyper-parameters for all agents

Parameter Name	Value	Description
Alpha	0.0001	Learning rate for stochastic gradient decent
Gamma	0.99	Discount rate for subsequent state values
ReplaySize	150000	Number of experiences sorted in memory
UpdateFrequency	4	Number of steps between value network updates
BatchSize	32	Number of experiences sampled in one update
TargetUpdateFrequency	10000	Number of steps between target network updates

Each agent was trained for 5,000,000 steps using their own approach to control the epsilon value during training. Breakout was selected as the main game for experimentation. Although superhuman performance could be achieved, in this game, with the standard DQL and epsilon policy, the average score was suboptimal and there was a lot of noise in the results. The aim is to compare the performance and variability of each of these agents over this training period.

4.1 Results

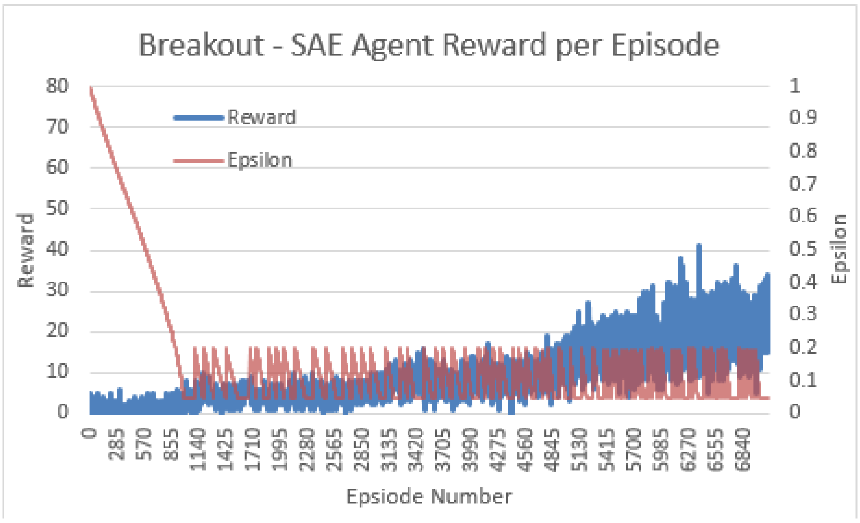


Fig. 1. Results from training the SAE agent in Breakout over 5,000,000 steps. The epsilon line shows how the exploration value steps up and anneals down during training

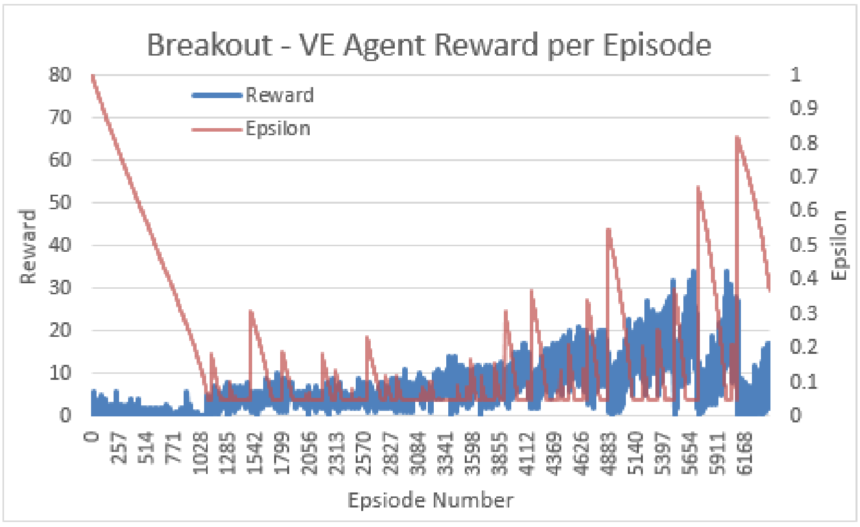


Fig. 2. Results from training the VE agent in Breakout over 5,000,000 steps. The epsilon line shows how the exploration value dynamically steps and anneals down during training.

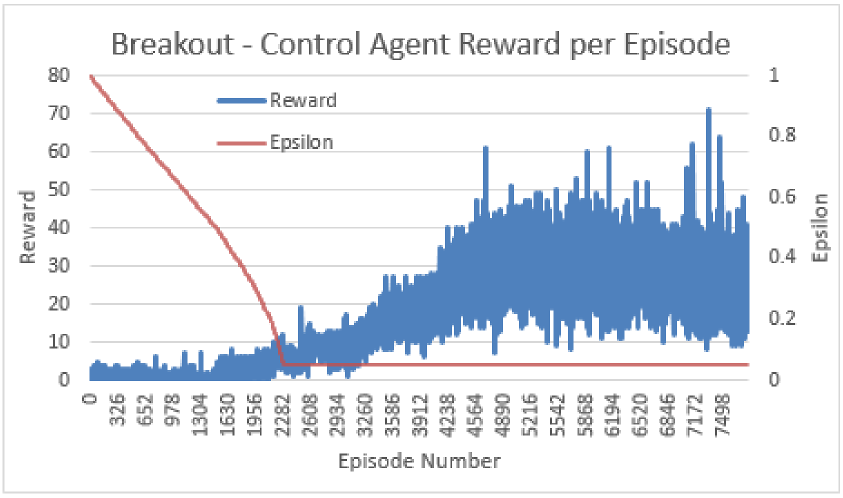


Fig. 3. Results from training the control agent in Breakout over 5,000,000 steps. The epsilon line shows how exploration anneals from the start down to a minimum exploration value.

Table 2. Agent Results Comparison (Averages are means taken over the last 200 episodes)

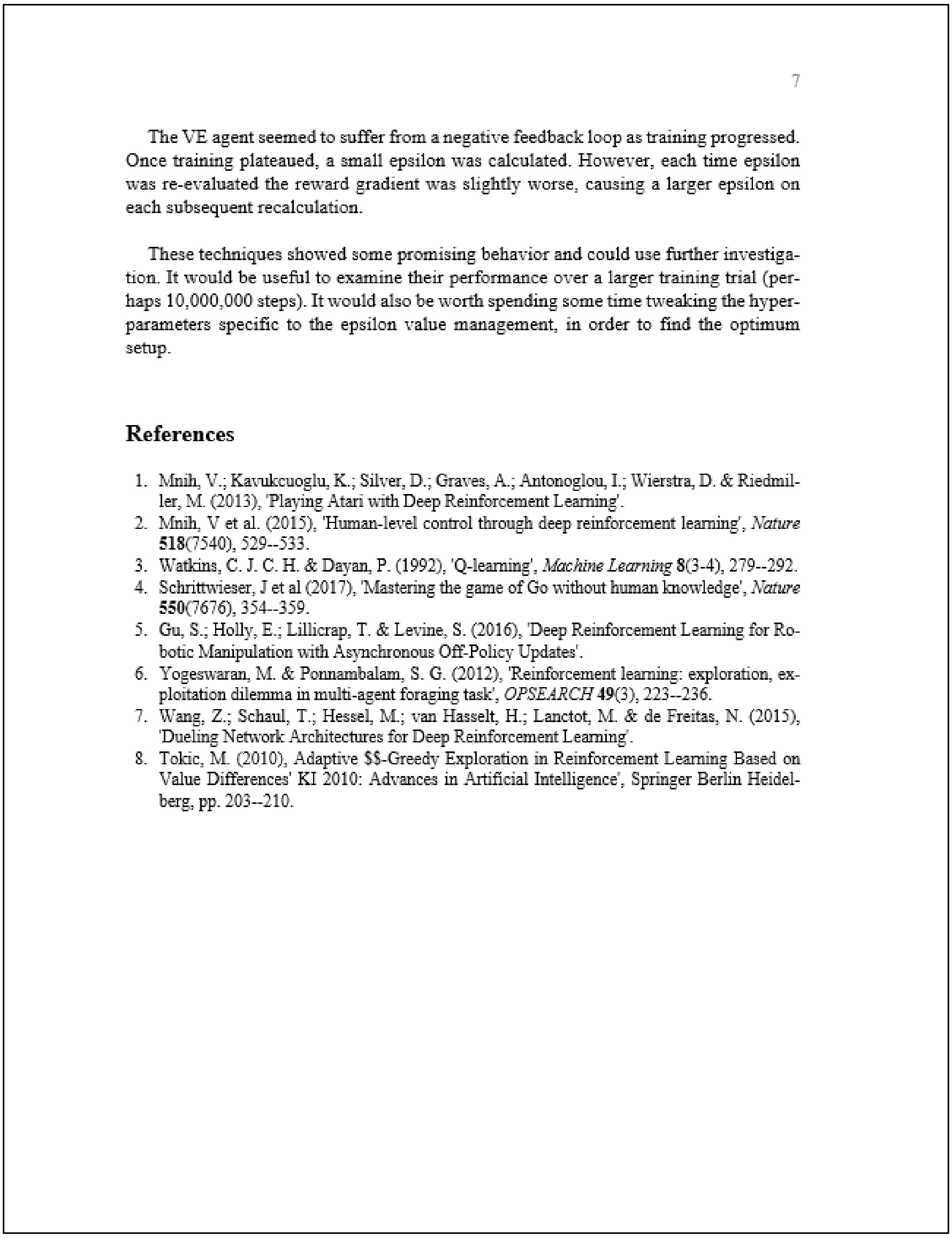
Agent Name	Average Score	Best Score
Control	39.2	320
Step Annealing Epsilon	27.3	74
Variable Epsilon		

5 Conclusion

Although the average and best performance of the tested agents over 5,000,000 steps did not surpass the ability of the control agent, there was some notable improvements.

The overall noise within a set of episodes seems lower when using one of the variable epsilon techniques. One reason for this could be that the agent is experiencing a wider range of states which are similar to each other, so learning in a particular localization is maximized. To fully justify this, a statistical analysis would have to be done, considering the average variation across a rolling section of the results.

Unlike the control agent, the variable epsilon agents suffered less of a drop off after a plateau had been reached in performance. This was likely due to the increased exploration preventing the agents getting stuck in feedback loops.



The VE agent seemed to suffer from a negative feedback loop as training progressed. Once training plateaued, a small epsilon was calculated. However, each time epsilon was re-evaluated the reward gradient was slightly worse, causing a larger epsilon on each subsequent recalculation.

These techniques showed some promising behavior and could use further investigation. It would be useful to examine their performance over a larger training trial (perhaps 10,000,000 steps). It would also be worth spending some time tweaking the hyper-parameters specific to the epsilon value management, in order to find the optimum setup.

References

1. Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D. & Riedmiller, M. (2013), 'Playing Atari with Deep Reinforcement Learning'.
2. Mnih, V et al. (2015), 'Human-level control through deep reinforcement learning', *Nature* **518**(7540), 529--533.
3. Watkins, C. J. C. H. & Dayan, P. (1992), 'Q-learning', *Machine Learning* **8**(3-4), 279--292.
4. Schrittwieser, J et al (2017), 'Mastering the game of Go without human knowledge', *Nature* **550**(7676), 354--359.
5. Gu, S.; Holly, E.; Lillicrap, T. & Levine, S. (2016), 'Deep Reinforcement Learning for Robotic Manipulation with Asynchronous Off-Policy Updates'.
6. Yogeswaran, M. & Ponnambalam, S. G. (2012), 'Reinforcement learning: exploration, exploitation dilemma in multi-agent foraging task', *OPSEARCH* **49**(3), 223--236.
7. Wang, Z.; Schaul, T.; Hessel, M.; van Hasselt, H.; Lanctot, M. & de Freitas, N. (2015), 'Dueling Network Architectures for Deep Reinforcement Learning'.
8. Tokic, M. (2010), Adaptive $$-Greedy Exploration in Reinforcement Learning Based on Value Differences' KI 2010: Advances in Artificial Intelligence', Springer Berlin Heidelberg, pp. 203--210.