

I hereby declare that this work has not been submitted for any other degree/course at this University or any other institution and that, except where reference is made to the work of other authors, the material presented is original and entirely the result of my own work at the University of Strathclyde under the supervision of Dr John Levine.

Investigating learning type algorithms for video games

Colin Cheung

Supervisor: Dr John Levine

Second Assessor: Dr Stephan Weiss

March 28, 2018

The copyright of this thesis belongs to the author under the terms of the United Kingdom Copyright Acts as qualified by University of Strathclyde Regulation 3.50. Due acknowledgement must always be made of the use of any material contained in, or derived from, this thesis.

Abstract

This project aims to investigate the application of Deep Q Networks (DQN) and its variants on the General Video Game AI (GVGAI) framework and compare it against Monte Carlo Tree Search (MCTS) and human controllers. The inspiration for this project was to find out how well a learning agent without a forward model could perform, especially when compared to other agents.

This project covers the implementation of DQN and the improvements: Double Deep Q Networks (DDQN), Prioritised Experience Replay (PER) and Dueling Networks and compares the performance of each. DDQN improved stability of learning and at the same time, performance. PER allowed for quicker training times, which allowed the agent to perform much better much faster. Furthermore it was seen in this project that Dueling Networks was outperformed by DDQN on the game of Aliens. This observation will likely be the case for most games in the GVGAI framework as the maximum number of possible actions for a game is five and Dueling Networks were shown to perform well in high action space environments. Overall, out of the four games tested: Aliens, Chase, Deceptizelda and Wafer Thin Mints, the DDQN agent was able to best MCTS in Deceptizelda, with similar scores for the others. This shows the DDQN agents is able to perform at a high level despite a lack of a forward model to predict future states.

The application of DQN to the GVGAI framework is the first of its kind and the initial results of this project may be beneficial for future research in this field. Furthermore the work discussed here in this project can be further expanded by researching cooperative two player games where both players interact with each other to work towards a common goal.

Contents

| | |
|---|-------------|
| Abstract | ii |
| List of Figures | v |
| List of Tables | viii |
| Acronyms | ix |
| Glossary | x |
| Preface/Acknowledgements | xii |
| 1 Introduction | 1 |
| 1.1 AI in popular culture | 1 |
| 1.2 Research Objectives | 2 |
| 1.2.1 Linking existing research | 2 |
| 2 Background Research | 4 |
| 2.1 Artificial Neural Networks | 4 |
| 2.1.1 Convolutional Neural Networks | 5 |
| 2.2 Reinforcement Learning | 7 |
| 2.2.1 Markov Decision Process | 7 |
| 2.2.2 Total Discounted Future Reward | 8 |
| 2.2.3 Q-Learning | 9 |
| 2.2.4 Deep Q Network | 10 |
| 2.3 Exploration vs Exploitation Problem | 11 |
| 2.4 DQN and Beyond | 12 |

Contents

| | | |
|----------|---|-----------|
| 2.4.1 | Experience Replay | 12 |
| 2.4.2 | Double and Dueling Networks | 12 |
| 2.4.3 | Epsilon-Greedy | 13 |
| 2.4.4 | Reward Function | 13 |
| 3 | Methodology | 14 |
| 3.1 | Tabular Q-Learning | 14 |
| 3.1.1 | Extension to Q Network | 15 |
| 3.2 | DQN | 17 |
| 3.2.1 | Deep Neural Network | 17 |
| 3.2.2 | Frame Skipping | 18 |
| 3.2.3 | Reward Clipping | 18 |
| 3.2.4 | Experience Replay | 19 |
| 4 | Experimentation | 20 |
| 4.1 | Frozen Lake | 20 |
| 4.1.1 | Tabular Implementation | 21 |
| 4.1.2 | Neural Network Implementation | 21 |
| 4.1.3 | Results | 22 |
| 4.2 | GVGAI | 22 |
| 4.2.1 | Aliens | 23 |
| 4.2.2 | Preprocessing the frames | 24 |
| 4.2.3 | Initialising replay memory | 26 |
| 4.2.4 | Implementation | 26 |
| 4.2.5 | Performance | 27 |
| 4.2.6 | Concerns | 29 |
| 5 | Improvements and Variations | 30 |
| 5.1 | Target Network | 30 |
| 5.1.1 | Evaluation | 31 |
| 5.2 | Prioritised Experience Replay | 34 |
| 5.2.1 | Sum Tree | 36 |
| 5.2.2 | Evaluation | 36 |

Contents

| | | |
|----------|---|-----------|
| 5.3 | Alternative Reward Function | 39 |
| 5.3.1 | Evaluation | 39 |
| 5.4 | Dueling Network | 42 |
| 5.4.1 | Evaluation | 44 |
| 6 | Application to other games | 47 |
| 6.1 | Monte Carlo Tree Search | 47 |
| 6.2 | Results | 49 |
| 7 | Conclusion | 51 |
| 7.1 | Further work | 51 |
| 7.1.1 | Alternative Action Selection Policies | 52 |
| 7.1.2 | Asynchronous Actor-Critic Agents | 53 |
| 7.1.3 | Hyperparameter Tuning | 54 |
| 7.1.4 | Increased Training Time | 54 |
| A | Pseudocode | 55 |
| B | Diagrams | 58 |
| C | Source Code | 60 |
| | Bibliography | 60 |

List of Figures

| | | |
|-----|--|----|
| 2.1 | Example of a single neuron in a neural network. Image from Cornell University: https://blogs.cornell.edu/info2040/2015/09/08/neural-networks-and-machine-learning/ | 4 |
| 2.2 | Example of an artificial neural network. A classic mutli-layer perceptron setup. Image from a blog by XenonStack. | 5 |
| 2.3 | Example of a CNN. Image from course notes by Standard University: http://cs231n.github.io/convolutional-networks/ | 6 |
| 2.4 | Example of a CNN with all the layers shown in detail. Image from a GitHub blog: https://shafeentejani.github.io/2016-12-20/convolutional-neural-nets/ | 7 |
| 2.5 | Reinforcement learning cycle. Image from a blog by JD Co-Reyes: https://ai.intel.com/openai/ | 8 |
| 2.6 | Left: Naive implementation, Right: Optimized implementation by Google DeepMind. Image from a blog by Tambet Matiisen. | 11 |
| 3.1 | Illustration of CNN architecture used in the paper. The output is one of 18 actions corresponding to combinations on the joystick. Image from Google DeepMind [1] | 18 |
| 4.1 | The grid world of Frozen Lake Image from OpenAI Gym. | 20 |
| 4.2 | Frames taken directly from the alien game in GVGAI. Green sprite: Alien, White sprite: Player, Grey sprite: Obstacle | 24 |
| 4.3 | Example input to the neural network, 4 images form a stack. | 26 |
| 4.4 | Rewards gained throughout the DQN learning process. | 28 |
| 4.5 | Total number of wins measured throughout the DQN learning process. | 28 |

List of Figures

| | | |
|------|--|----|
| 5.1 | Comparison of the rewards per episode of for DDQN and DQN | 32 |
| 5.2 | Comparison of the cumulative wins of for DDQN and DQN | 32 |
| 5.3 | Comparison of the mean Q values for DDQN and DQN | 33 |
| 5.4 | Comparison of the win rates between DQN and DDQN during different episode ranges. | 34 |
| 5.5 | Comparison of the rewards per episode of for the different implementations | 36 |
| 5.6 | Comparison of the cumulative wins for the different implementations . . | 38 |
| 5.7 | Comparison of the win rates against the standard DDQN during different episode ranges. | 38 |
| 5.8 | Comparison of the rewards per episode of for the different implementations. The moving averages were removed due to cluttering the plot. . . | 40 |
| 5.9 | Comparison of the cumulative wins for the different implementations . . | 41 |
| 5.10 | Comparison of the win rates against the old reward function during different episode ranges. | 41 |
| 5.11 | Top: Standard DQN with single stream of Q value. Bottom: Dueling Network where the value and advantage functions are split into separate streams, only to be summed at the output. Diagram from the paper <i>Dueling Network Architectures for Deep Reinforcement Learning</i> [2] . . | 43 |
| 5.12 | Comparison of the rewards per episode of for the two different implementations of the dueling network. | 44 |
| 5.13 | Comparison of the cumulative wins for the two different implementations of the dueling network. | 45 |
| 5.14 | Comparison of the max variant versus the mean variant during different episode ranges. Positive values indicate the max variant perform better and the mean variant worse, and also vice versa. | 45 |
| 5.15 | Comparison of the dueling network variants and DDQN + PER + New Reward | 46 |
| 6.1 | Example steps of Monte Carlo Tree Search | 48 |

List of Figures

| | | |
|-----|--|----|
| 7.1 | A diagram of four different actions the agent can take. The numbers represent arbitrary Q values. The height of the blue bars represent the probability of the action being selected. Note that the three bars on the right have the same height, this means they each have an equal probability of being selected. Image from a blog by Arthur Juliani. . . . | 52 |
| 7.2 | This diagram has the same Q values as 7.1. Note that the three bars on the right have differing heights proportional to their Q values. Image from a blog by Arthur Juliani. | 52 |
| 7.3 | A diagram of the A3C architecture. Image from a blog by Arthur Juliani. | 53 |
| B.1 | Results of the DQN created by DeepMind. Image from the paper <i>Human-level control through deep reinforcement learning</i> [1] | 58 |
| B.2 | Flowchart of the overall GVGAI algorithm | 59 |

List of Tables

| | | |
|-----|---|----|
| 3.1 | Initial Q-table values | 15 |
| 3.2 | CNN architecture used by DeepMind | 17 |
| 4.1 | Simple neural network example | 22 |
| 4.2 | Results of both implementations | 22 |
| 4.3 | Hyperparameters used for the DQN | 27 |
| 4.4 | Win percentages for different phases of the DQN learning | 29 |
| 5.1 | Win percentages for different phases of the DDQN learning | 33 |
| 5.2 | Win percentages for different phases of the DDQN + PER learning . . . | 37 |
| 5.3 | Win percentages for different phases of the DDQN + PER + New Re- ward learning | 40 |
| 6.1 | Performance of agents on different games. Each cell details the average score of the agent with the win rate in brackets. The DDQN includes Prioritised Experience Replay and the new reward function, the MCTS has access to a forward model to simulate future states. | 49 |

Acronyms

A3C Asynchronous Actor-Critic Agents. viii, 55

AI Artificial Intelligence. 1, 2, 23, 49–51, 53

CNN Convolutional Neural Network. vi, ix, 5–7, 11, 15, 17, 18

DDQN Double Deep Q Network. vii, ix, 32–39, 41, 43, 46, 48–52

DQN Deep Q Network. vii–ix, 14, 15, 17, 22, 23, 26–38, 45, 49, 59, 62

GPU Graphics Processing Unit. 25

GVGAI General Video Game AI. vi, 3, 22–25, 46

GVGP General Video Game Playing. 3, 22

I.I.D Independent and Identically Distributed. 12

MCTS Monte Carlo Tree Search. ix, 49–52

MDP Markov Decision Process. 7, 8, 14

PER Prioritised Experience Replay. vii, ix, 35, 36, 38, 41, 43, 44, 46, 48–51

SSO Serializable State Observation. 23, 52

UCT Upper Confidence Bound 1 applied to trees. 50

VGDL Video Game Definition Language. 23

Glossary

agent An autonomous entity that observes through sensors and interacts with its environment with an aim to achieve some objective. 7

episode A sequence of states, actions and rewards that result in a terminal state. 8–10

forward pass Passing input to a neural network and retrieving the resulting output.
11

policy Denoted as π , it is the rules an agent follow to decide what actions to take in a given state. 8, 10, 21

Preface/Acknowledgements

I would like to first, thank my project supervisor Dr. John Levine for giving me this wonderful opportunity to work on this field by accepting my project proposal. I would also like to thank Damien Anderson for the support he has given me throughout this project.

Lastly I'd also like to thank all my friends and family who have supported me throughout my years at Strathclyde.

Chapter 1

Introduction

1.1 AI in popular culture

The first Artificial Intelligence (AI) to beat a reigning world champion at chess was named Deep Blue [3]. The event, which occurred in 1997, was a huge step for AI as the public did not believe that an AI could ever beat a human expert [4]. Deep blue's implementation used traditional brute force techniques in conjunction with hand-crafted heuristics [4], it was said that it could evaluate up to 200 million moves per second [5]. However, as far as generalised AI was concerned, deep blue was a far cry from being able to play more than just chess alone. Over a decade later in 2011, IBM unveiled Watson, the very first computer to beat human champions on stage at the game of Jeopardy [6]. The game of Jeopardy is a much more different type of game than chess, it is a quiz game that tests general knowledge, Watson must then be able to comprehend the meaning behind clues before it can answer it. Fast forward a few years later in March 2016, AlphaGo, the AI created by Google's DeepMind, beat 18-time world champion Lee Seedol winning four out of five games [7]. The complexity of Go is many magnitudes greater than chess, with 10^{360} possible games in Go [8] versus 10^{120} in chess [9], in perspective the number of atoms in the observable universe is 10^{80} [10]. AlphaGo uses a novel combination of Monte Carlo Tree-Search, supervised learning and reinforcement learning to accomplish this feat [11]. Despite AlphaGo's achievements it still had human knowledge implanted when the AI initially learns to play the game. However just over a year later, DeepMind developed an improved version of AlphaGo named aptly AlphaGo Zero, which learns Go from scratch with zero human knowledge.

Chapter 1. Introduction

This version was able to beat its predecessor that won against Lee Sedol by a huge streak of a hundred games to zero. DeepMind extended this version of their AI to play both Chess and Shogi, beating the best AI's from both fields [12].

While these accomplishments from AlphaGo and the Zero variant were able to become the best player in their respective game, at the same time these types of games were very suitable for an AI to learn from. Chess, Go and Shogi share four convenient properties [13] for an AI:

1. Fully Deterministic - There is no randomness, each action will always have the same outcome.
2. Fully Observable - The entire game state is always known, in contrast in the game of Poker you do not know the hands of other players.
3. Access to simulations - Actions can be simulated in the game before they are played to evaluate how good it is.
4. Time for Deliberation - There is sufficient time to think before the next action.

1.2 Research Objectives

With this in mind, how well can an AI agent play a game where these four conditions are not always met? Particularly, an interesting area is an AI that can play not just one game, but multiple games. Following this, the main objectives are:

- Research, design and implement generalised video game playing learning type AI
- Investigate and implement improvements to these agents.
- Analyse its performance against other existing video game playing AI's.

1.2.1 Linking existing research

The idea of an agent being able to play multiple video games was formally introduced in the paper *General Video Game Playing* (GVGP) [14] in 2012 by a group of researchers including project supervisor, John Levine. This work led to the development

Chapter 1. Introduction

of a framework and competition known as the General Video Game AI (GVGAI) competition that was created to evaluate different types of agents. Originally the focus of the competition was to move away from using forward models, agents should be able to learn to play the games by only giving them the current state of the game; they will have to learn a forward model for themselves. However during that period it was difficult to produce such an agent without relying on a forward model and as such the competition was focused on those types of agents. Coincidentally, two years later in 2014, Google DeepMind revealed in the paper *Human-level control through deep reinforcement learning* [1], that their agent that was able to play a variety of games using the same algorithm at human level performance. This project serves to link these two different research papers by applying the agent used by Google DeepMind to the General Video Game AI framework without using a forward model.

Chapter 2

Background Research

This chapter aims to give a broad overview of the knowledge necessary to understand the work discussed later.

2.1 Artificial Neural Networks

An artificial neural network (aka. neural network) is simply a fancy name to describe a tool used to approximate a function. It was modelled after the human brain where there are billions of neurons connected in a huge network which gives us our intelligence [15].

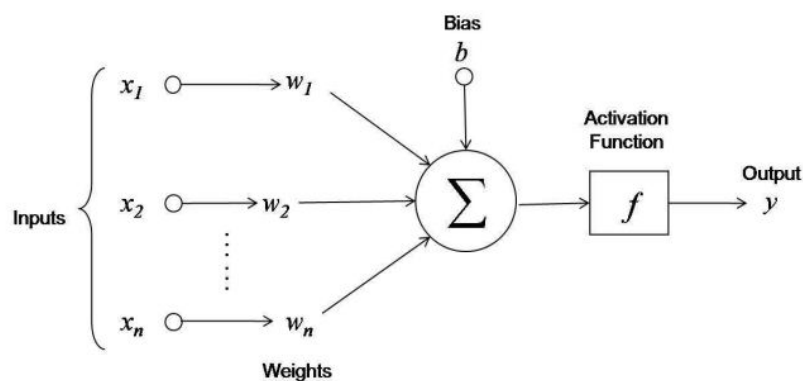


Figure 2.1: Example of a single neuron in a neural network. Image from Cornell University: <https://blogs.cornell.edu/info2040/2015/09/08/neural-networks-and-machine-learning/>

Figure 2.1 shows an example artificial neuron, an artificial neural network will consist many of such artificial neurons connected in one or more hidden layers. Since

every neural network will consist of an input and output layer, the main interests regarding the network's architecture are its hidden layers.

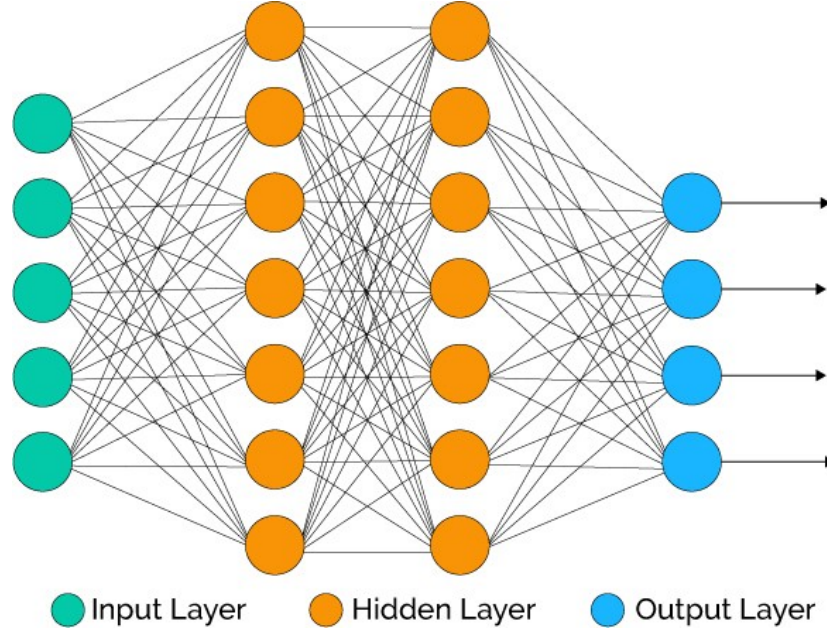


Figure 2.2: Example of an artificial neural network. A classic mutli-layer perceptron setup. Image from a blog by XenonStack.

A single artificial neuron performs the following actions: multiply the inputs by it's corresponding weight, summing the result of each multiplication, adding a bias, passing this sum through an activation function and finally producing a new output. The resulting output can then be passed on to another neuron, this process can be repeated many times. Figure 2.2 shows an example neural network with two hidden layers and 7 neurons in each hidden layer. A neural network learns by minimising a loss function and adjusting the values of it's weights using gradient descent and backpropagation algorithms. It is not necessary to go into detail on how the gradient descent and backpropagation algorithms work, the takeaway from this is that a neural network allows us to approximate a non-linear function.

2.1.1 Convolutional Neural Networks

A subclass of the artificial neural network, the Convolutional Neural Network (CNN) has demonstrated much success at analysing images [16]. A CNN can take as input a 3D-array of values representing an RGB image, as a result the neurons in a CNN

Chapter 2. Background Research

can be arranged in a 3D manner as shown in Figure 2.3 as opposed to the 2D manner shown in Figure 2.2

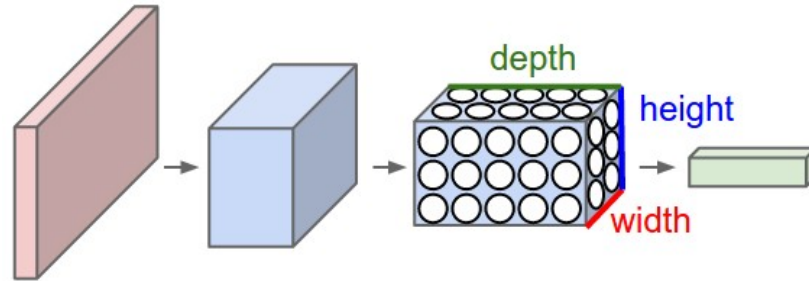


Figure 2.3: Example of a CNN. Image from course notes by Standard University: <http://cs231n.github.io/convolutional-networks/>

It consists of three main components:

- Convolutional Layer - The convolutional layer down-samples the original input space by applying a filter, a set of weights that are useful in detecting specific features in an image like eyes or ears. It computes the dot product between the input space and filters and the resulting computation is passed through a non-linear function typically a ReLU.
- Pooling Layer - The output of the convolutional layer is then passed into a pooling layer, this layer down-samples the input space once more and allows the network to learn translation invariance, i.e. the network can recognise different features of a tiger even if it doesn't appear in the same location in the image.
- Fully Connected Layer - Convolutional and Pooling layers typically come in pairs but right at the output we have one or more fully connected layers, this allows the network to map a function to the correct class of image. These fully connected layers are 2-dimensional, so the 3-dimensional convolutional and pooling layers are first flattened before they are connected.

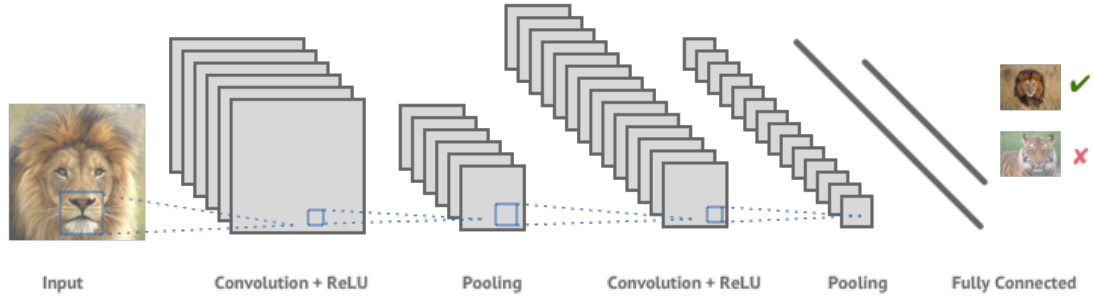


Figure 2.4: Example of a CNN with all the layers shown in detail. Image from a GitHub blog: <https://shafeentejani.github.io/2016-12-20/convolutional-neural-nets/>

While this may seem quite complex, the main points of the Convolutional Neural Network is that it repeatedly down-samples the input space using many different filters, through this action the network is able to learn more and more complex features and finally combining the information from all these filters to classify the image.

2.2 Reinforcement Learning

Reinforcement learning comes under a subcategory of machine learning. Simply put, it is the process of an agent interacting with its environment to maximise some reward [17]. In detail, the agent first observes the environment's state, it then performs an action in this state. The environment's state changes into a new state as a result of the action and a reward (or penalty) is received. By automating this process, the agent can learn from its experience with interacting with the environment and should be able to choose actions to maximise the total reward gained.

2.2.1 Markov Decision Process

Markov Decision Process (MDP) is a mathematical framework used to model a decision making process and is used in reinforcement learning [18]. It consists of a 5-tuple (S, A, P, R, γ) :

- S: Set of states - One observation from the environment is a single element of this set.
- A: Set of actions - Actions the agent can choose to perform to interact with the environment.

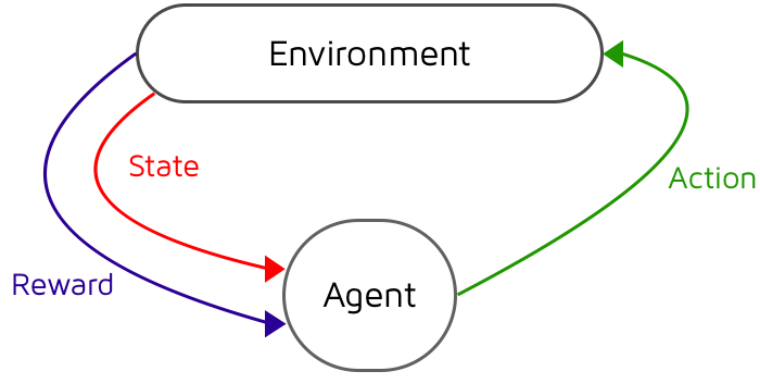


Figure 2.5: Reinforcement learning cycle. Image from a blog by JD Co-Reyes: <https://ai.intel.com/openai/>

- $P: P(s' | s, a)$ Probability transition model - What is the probability to end up in state s' whilst in state s and taking action a
- $R: P(r | s, a)$ Reward model - What is the reward r whilst in state s and taking action a
- γ : discount factor - A decimal value between 0 and 1 which places the importance between immediate and future rewards.

An Markov Decision Process for a single episode would look like:

$$s_0, a_0, r_0, s_1, a_1, r_1, \dots, s_{n-1}, a_{n-1}, r_{n-1}, s_n$$

Note that for our probability transition model $P(s' | s, a)$, the next state s' depends only on the current state s and action a taken. It does not depend on any previous states and actions, this is known as the Markov Property [19]. The ultimate goal of a reinforcement learning agent is to solve the MDP by finding the optimal policy π^* . This allows the agent to find the sequence of actions that allow it to maximise the **total discounted future reward**.

2.2.2 Total Discounted Future Reward

To explain what the total discounted future reward is, let us first specify the total reward for a given episode:

Chapter 2. Background Research

$$R = r_1 + r_2 + r_3 + \dots + r_n$$

We can then say that the total future reward from any point t is:

$$R_t = r_t + r_{t+1} + r_{t+2} + r_{t+3} + \dots + r_n$$

In a stochastic environment our rewards will never be the same if we repeat the same action for a given state. The further into the future the reward is, the more likely this is to be the case. As a result we use the **discounted future reward**:

$$R_t = r_t + \gamma \cdot r_{t+1} + \gamma^2 \cdot r_{t+2} + \dots + \gamma^{n-t} \cdot r_n \quad (2.1)$$

Here we see the discount factor γ from before in Section 2.2.1, when $\gamma = 0$ then it would really only consider the immediate reward from the next state. In contrast when $\gamma = 1$ it maximises future reward, this should only be used in a deterministic environment, but even then it can still lead to problems as the total future reward will tend to infinity. In practice, to prevent the reward from going to infinity and maximise future rewards, a value close to 1 is typically used. We also can further reduce eq. 2.1 in terms of itself:

$$R_t = r_t + \gamma \cdot \underbrace{(r_{t+1} + \gamma \cdot (r_{t+2} + \dots))}_{R_{t+1}} = r_t + \gamma \cdot R_{t+1} \quad (2.2)$$

This recursive function is relevant for the next section.

2.2.3 Q-Learning

So far we have described total discounted future reward, a good strategy for an agent is to maximise this reward. Following this let us now introduce a new function Q:

$$Q : S \times A \mapsto \mathbb{R}$$

It takes as input, the state and action, and returns a real value that indicates the quality of the state-action pair, in other words it's how good it is to take action a while in state s . We can calculate Q by:

$$Q(s_t, a_t) = r_t + \gamma \cdot \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \quad (2.3)$$

This is an extension to eq. 2.2 and is also known as a Bellman equation [20]. It's a nice way to specify how good a state-action pair is by stating it is the immediate reward r_t plus the discounted future reward for the next state and this allows for the function to maximise the total discounted reward. Intuitively, this means the agent can just pick actions that result in the greatest Q value:

$$\pi(s) = \arg \max_a Q(s, a)$$

Where π denotes the policy, the set of rules the agent follows, in this case the policy is selecting the action resulting in the maximum Q value for any given state. This in turn means the policy is maximising the total discounted future reward. Alternatively this can be interpreted as taking optimal actions from state s . Note that when one reaches the terminal state, the Q function would simply be the reward for reaching the terminal state:

$$Q(s_n, a_n) = r_n, \text{ where } n \text{ is terminal}$$

This is easy to see as there is no more potential future reward once an episode has finished, this also allows for the Q-function to propagate this value back toward previous states.

2.2.4 Deep Q Network

While Q-Learning can be used with a tabular approach but they don't scale well for large state spaces [21]. As a result we require a different approach to modelling the Q-function without relying on table. For this case we can make use of a function approximator as discussed in Section 2.1, also known as neural networks. The usefulness of neural networks is that it can generalise for any number of possible states as long as it is represented by a vector or matrix.

The next step is to define the appropriate representation of a state. We want states to be generalised well enough that it can work across multiple games, thus it is logical

to use the game’s screen pixels as input. As our input is an image we are particularly interested in a variation of neural networks called the Convolutional Neural Network (CNN) which have shown to have better performance in image classification [16]. Figure 2.6 shows how to structure the neural network for the Q-function.

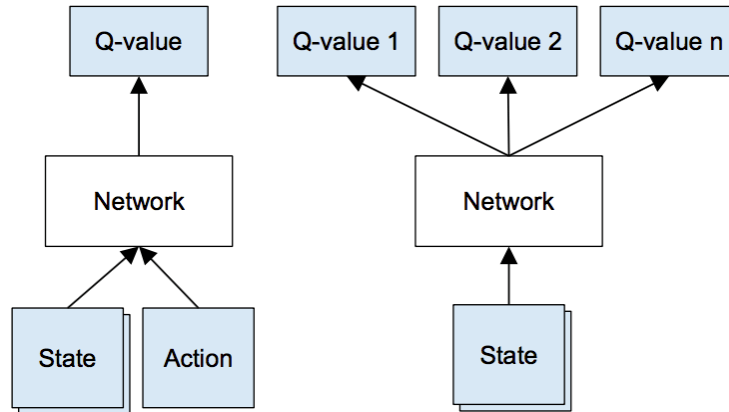


Figure 2.6: Left: Naive implementation, Right: Optimized implementation by Google DeepMind. Image from a blog by Tambet Matiisen.

While the Q-function does take a state and action as input, representing this the same way for a neural network would be suboptimal as it would require a resulting forward pass for each state-action pair. Instead Google DeepMind propose to only have the state as the input with the Q values for each action as the output. The result is that we only require a single forward pass for a state to determine all the Q values.

2.3 Exploration vs Exploitation Problem

Reinforcement learning can allow our agent to learn from its environment, however there is a fundamental issue that has not been addressed. At the start of learning, the agent knows nothing of its environment and must do some exploration before eventually exploit the knowledge it has learned. However when should it stop exploring and start exploiting?

A good example of this is the Multi-Armed Bandit problem [22]. In the problem, a gambler is playing N slot machines and wishes to maximise his profit (reward). During each trial the gambler faces a decision between playing the machine that has the current highest payout versus playing other machines in an attempt to find a machine with

greater payout. In reinforcement learning the agent faces this type of problem.

2.4 DQN and Beyond

Reinforcement learning when paired with a non-linear function approximator such as a neural network can be very unstable. There have been many improvements since the original paper that stabilise learning and ultimately improve performance.

2.4.1 Experience Replay

When a game is played the frames are immediately collected and trained at the end of the game, but this causes dependent training samples. As a result this highly correlates the dataset and violates any Independent and Identically Distributed (I.I.D) assumption for neural networks. Instead of immediately training on the frames of the game as it comes in, store it in memory to be sampled later. Doing so breaks the high correlation between frames. In the paper *Human-level control through deep reinforcement learning* [1] it suggested to sample from the memory uniformly. However, in a later paper *Prioritised Experience Replay* [23] showed that sampling experience replay by priority significantly improves training speeds and performance compared to the uniform sampling.

2.4.2 Double and Dueling Networks

The paper *Deep Reinforcement Learning with Double Q-learning* [24] proposed to split the Q-function neural network into two, one for selecting the action-values (Primary Network) and one for predicting the target values (Target Network). They then update the Target network to the Primary Network only periodically, and doing so improves the stability of training. Furthermore the paper *Dueling Network Architectures for Deep Reinforcement Learning* [2] has shown that a modified architecture named Dueling Network showed significant improvements in performance when the action space was especially large.

2.4.3 Epsilon-Greedy

The Epsilon-Greedy policy has been a standard as it is very simplistic. It works by selecting a random action with probability $1 - \epsilon$ otherwise select the max over all Q-values. The value of ϵ is annealed over time to a small value. This method gives a good solution to the Exploration vs Exploitation problem. The issue with Epsilon-Greedy is that when it selects a random action it does so using a uniform distribution, this is not the best solution as we can exploit the Q-values to make a more informative decision.

2.4.4 Reward Function

The reward function is perhaps the most critical component for reinforcement learning as it dictates what is considered "good" and "bad". However there has not been much focus in this area perhaps not wanting to add "hand crafted" features much like with the Deep Blue AI. DeepMind simply clips their reward to be in the range of -1 and +1, it helps with training the neural network but causes the network to be unable to tell the difference between a good state and a great state.

Chapter 3

Methodology

The primary objective is to research, design and implement a DQN agent. However due to the experimental nature of this work, it is necessary to incrementally build the system as to reduce complications in the future. This chapter covers the steps in order to achieve that goal.

3.1 Tabular Q-Learning

DQN is a combination of many different technologies and innovations, it is important to start simple. Thus before we attempt to run we should first learn how to walk, and in this case tabular Q-Learning is learning to crawl. As shown in Section 2.2.3 the Q-function takes as input the state and action and produces a corresponding Q-value to indicate the quality of the state-action pair. To implement this in principle we store all state-action pairs in a table. In this scenario we would use a 2D array to store n states and k actions resulting in a total of $n * k$ entries in the table. We can initialise the Q-table value to an arbitrary number, in Table 3.1 the initial value is 0.

Remember from Section 2.2.1 that in an MDP we store transitions as a tuple (s, a, r, s') . The update rule for the Q-function is now:

$$Q(s, a) = Q(s, a) + lr \cdot (r + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a)) \quad (3.1)$$

Where:

s : the current state

| State \ Action | 1 | 2 | ... | k |
|----------------|----------|----------|-----|----------|
| 1 | 0 | 0 | ... | 0 |
| 2 | 0 | 0 | ... | 0 |
| 3 | 0 | 0 | ... | 0 |
| \vdots | \vdots | \vdots | ... | \vdots |
| n | 0 | 0 | ... | 0 |

Table 3.1: Initial Q-table values

a : the action for the current state

r : the reward after performing action a in current state s

s' : the next state

lr : the "learning rate" a value that determines how fast the function converges.

When s is terminal the function is then:

$$Q(s, a) = Q(s, a) + lr \cdot (r - Q(s, a)) \quad (3.2)$$

3.1.1 Extension to Q Network

Instead of using a table to store the Q-values we can use a feed-forward neural network to attempt to approximate the Q-function. Doing so will help give some experience in implementing the CNN later on for DQN's. This however may cause complications as the state space is particularly small and is expected that the neural network will take much longer to converge. A network with one hidden layer containing n neurons would be our table equivalent.

The input of the state is represented by 1-hot encoding, each possible location in the game is represented by numbers 1 through n and the location of where the agent currently resides is set to 1. The hidden layers are simply n neurons that combined with the k neurons at the output give us a sort of table equivalent to the neural network.

Chapter 3. Methodology

The k neurons at the output represent the Q value of that action for a given input. Furthermore, experience replay requires to store transitions of the form:

$$transition = (s, a, r, s', d) \quad (3.3)$$

Where:

d : a binary value indicating if s' is terminal

An issue with this is that the format of the transition requires to know the next state. However for the current state it is not possible to retrieve the next state, after all it can't predict the future. As a result, instead of attempting to predict the future, delay transitions by one timestep so that s' is simply the current state and s is the previous state:

$$transition = (s_{t-1}, a_{t-1}, r_{t-1}, s_t, d) \quad (3.4)$$

Storing these transitions as the agent plays the game allows it to sample from a distribution later on during training and not directly train on it as it comes in, this is briefly mentioned in Section 2.4.1. Lastly the d variable is necessary for determining which update function to use, eq. 3.1 or eq. 3.2.

The update function for the neural network slightly differs from that of the tabular approach. The network needs to minimise some loss function, formally this function is:

$$L = \left[\underbrace{r + \gamma \cdot \max_{a'} Q(s', a)}_{\text{target}} - \underbrace{Q(s, a)}_{\text{prediction}} \right]^2 \quad (3.5)$$

This is simply a regression task. Given a single transition (s, a, r, s', d) it is relatively straightforward to calculate this loss function. Firstly compute the forward pass for state s then compute a second forward pass for state s' . Get the max Q-value from the output of the second pass, use this value to set the target action a from the first forward pass using the generic Q-function $r + \gamma \cdot \max_{a'} Q(s', a')$, where $\max_{a'} Q(s', a')$ is the max Q-value from the second forward pass.

3.2 DQN

Stepping from tabular Q-Learning and implementing the version of the DQN Google DeepMind proposed in their paper [1] can be broken in multiple sections.

3.2.1 Deep Neural Network

A deep neural network is simply a neural network with multiple hidden layers. In particular DeepMind proposed to use a Deep Convolutional Neural Network for their network architecture. The specifics are shown in Table 3.2

| Layer | Output shape | Filter size | Stride |
|--------|--------------|-------------|--------|
| input | 84x84x4 | | |
| conv1 | 20x20x32 | 8x8 | 4 |
| conv2 | 9x9x64 | 4x4 | 2 |
| conv3 | 7x7x64 | 4x4 | 1 |
| fc1 | 512 | | |
| output | 18 | | |

Table 3.2: CNN architecture used by DeepMind

As discussed in Section 2.1 and Section 2.1.1 the primary interests here are the four hidden layers, the three convolutional layers (conv1-3) and the fully connected layer (fc1). As far as convolutional neural networks go, this is a standard setup except there are no pooling layers. The pooling layers are unnecessary, detrimental even, unlike the case with classifying different images of tigers, we don't want translation invariance. This is due to the fact that for specific games we need to know precisely where a particular object is in the image. For example in the game of Pong we don't want to discard information regarding the position of the ball.

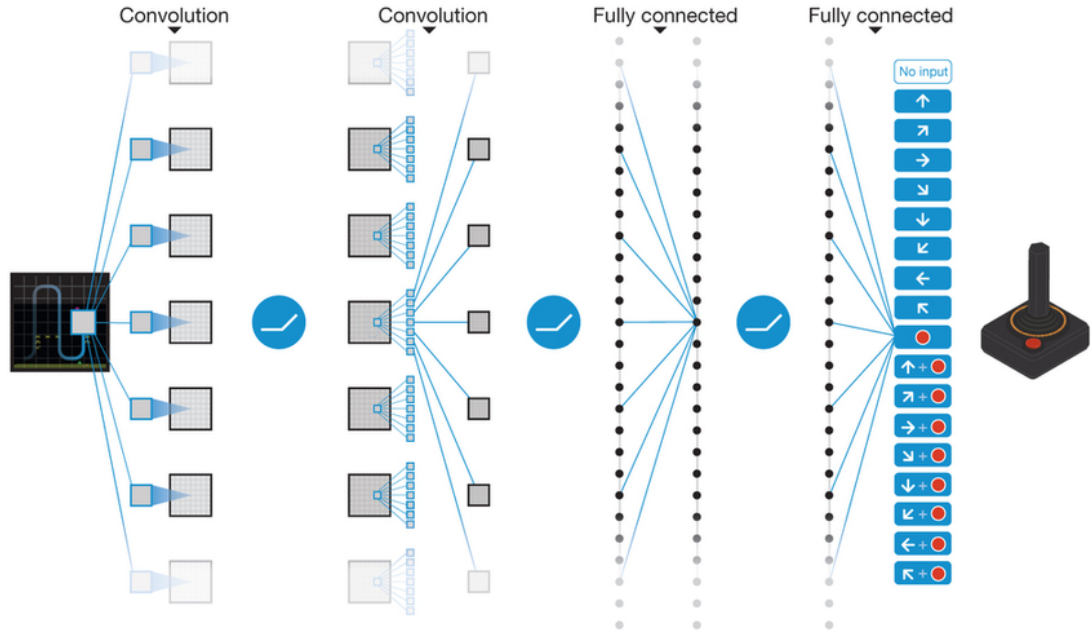


Figure 3.1: Illustration of CNN architecture used in the paper. The output is one of 18 actions corresponding to combinations on the joystick. Image from Google DeepMind [1]

3.2.2 Frame Skipping

The input to the neural network also differs slightly. Instead of feeding in an RGB image to the network, DeepMind applies what they call frame-skipping. Each frame of the game is converted into gray-scale, four of these images form a stack and is fed into the neural network. This is the reason behind the input shape being $84 \times 84 \times 4$. This method however only produces a single action per four frames, DeepMind chooses to repeat this action for the next four frames until a new action is computed. They state that it allows the agent to play at a significantly faster rate without increasing the runtime by much [1]. It could also be inferred that logically speaking an agent does not necessarily need to perform an action at each frame of the game. Furthermore by stacking multiple frames it gives the network information about the velocities of the objects in the game.

3.2.3 Reward Clipping

In the paper *Human-level control through deep reinforcement learning* [1], the agent was made to play a variety of different games. It uses the game score as an intermediate state

Chapter 3. Methodology

reward function, however the game score itself varies drastically throughout different games. To tackle this problem DeepMind proposes to simply assign a reward of +1 and -1 for all positive and negative score changes respectively and rewards of 0 are left unchanged. This simply means that score change of +10 will still result in a constant +1, however this approach is inherently flawed as the agent will not be able to differentiate rewards of different magnitudes. The paper brings up this issue but does not address the problem, however it seemed to still perform relatively well in a large variety of games DeepMind tested the agent on.

3.2.4 Experience Replay

Details of the implementation were discussed in Section 3.1.1. This method of storing transitions and then sampling at a later time solves multiple problems:

- Breaks correlations between transitions
- Allows old transitions to be revisited, effectively a memory bank
- Improves learning speed by training on mini batches

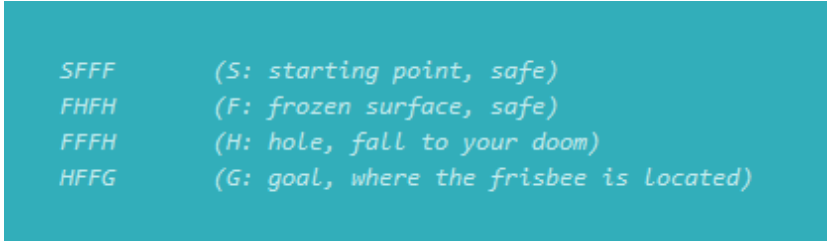
The default implementation uses a uniform distribution to sample these transitions, in the DeepMind paper they brought up this issue and that sampling through another method would be worth looking into.

Chapter 4

Experimentation

This chapter focuses on applying the theory discussed earlier into practice, details of the implementations and the results are shown in each section.

4.1 Frozen Lake



| | |
|-------------|--|
| <i>SFFF</i> | <i>(S: starting point, safe)</i> |
| <i>FHHH</i> | <i>(F: frozen surface, safe)</i> |
| <i>FFFF</i> | <i>(H: hole, fall to your doom)</i> |
| <i>HFFG</i> | <i>(G: goal, where the frisbee is located)</i> |

Figure 4.1: The grid world of Frozen Lake Image from OpenAI Gym.

Since using tables with Q-Learning practically only works for small state spaces, let us define the environment called Frozen Lake, it is a text based game available on OpenAI gym. The game is simple, it is a 4x4 grid world as shown in Figure 4.1, the player starts at the tile marked S (starting) and it's objective is to reach the tile marked G (goal). At any point the player can move in the four cardinal directions as long as it does not result in going out of bounds. The F (frozen) tiles are safe to traverse, however upon reaching a H (hole) tile the game ends and the player loses.

4.1.1 Tabular Implementation

Following from Section 3.1, it is worth noting that for this game we only need to give a reward on a terminal state either +1 or -1 depending on the outcome, any intermediate states will result in $r = 0$. Finally we use the Epsilon-Greedy policy to select our actions, the exploration rate is annealed over time so that at the end of training it is close to zero. Details of the results of training are shown in Table 4.2. The pseudocode for this approach is shown in Algorithm 1.

Algorithm 1 Tabular Q-Learning

```

1: initialise Q table to arbitrary values
2: initialise learning rate  $\alpha$ 
3: initialise exploration rate  $\beta$ 
4: initialise discount factor  $\gamma$ 
5: for each training episode do
6:   observe initial state  $s$ 
7:   while game not finished do
8:     get random number  $x$ 
9:     if  $x < \beta$  then
10:      select action from Q-table  $\max_a Q[s][a]$ 
11:     else
12:      select a random action
13:      perform action above and obtain reward  $r$  and new state  $s'$ 
14:      if  $s'$  is not terminal then
15:         $Q[s][a] = Q[s][a] + \alpha \cdot (r + \gamma \cdot \max_{a'} Q[s'][a'] - Q[s][a])$ 
16:      else
17:         $Q[s][a] = Q[s][a] + \alpha \cdot (r - Q[s][a])$ 
18:       $s = s'$ 
19:    $\beta = \beta - \text{decay rate}$ 

```

4.1.2 Neural Network Implementation

This does not differ much from the tabular approach except that transitions are now stored in replay memory and of course it requires a neural network to replace the table. The specifics of the neural network are shown in Table 4.1 and the pseudocode is shown in Algorithm 5, found in Appendix A.

| Layer | Output shape |
|--------|--------------|
| input | 16 |
| dense1 | 16 |
| output | 4 |

Table 4.1: Simple neural network example

4.1.3 Results

Overall the results seem promising. Shown in Table 4.2 are the results after training each implementation on 2000 episodes and then tested on a further 500 episodes, the performance is measured using win rate (the percentages of games won).

| Type | Win % |
|----------------|-------|
| Tabular | 76% |
| Neural Network | 70% |

Table 4.2: Results of both implementations

While the neural network did perform worse, this was expected as neural networks are slower to converge to the optimal. The success of these results can be seen as significantly important as the next stage is to properly implement the DQN.

4.2 GVGAI

General Video Game AI (GVGAI) [25] is a framework that will allow us to implement and evaluate the DQN agent. It is a framework developed by researchers to test different types of algorithms on a multitude of games. Its goal is to encourage research to develop an agent that could perform well on not just a single game but a variety of games. The origins of the competition first began with the paper *General Video Game Playing* [14] and has since developed into a framework and competition. The framework provides the tools to easily create a game using the Video Game Definition Language (VGDL).

Chapter 4. Experimentation

The games created in the GVGA framework are tailored especially for AI agents and are designed to test them, even adding elements to purposely lure the AI into a bad situation. The competition is hosted for these types of games and for the agents that use the forward model, however since the learning agent does not have a forward model and the competition time limits are too strict it will not be used in this project. The framework allows for easy implementation of AI agents by providing a simplistic abstract class, specifics are shown in Algorithm 2.

Algorithm 2 Abstract Agent Class

- 1: ▷ This function is called at the start of every game. Useful for initialising variables.
 - 2: **function** INIT(sso, timer)
 - 3:
 - 4: ▷ The most important function, this function is called in-between transitions and the agent returns the next move for this state.
 - 5: **function** ACT(sso, timer)
 - 6:
 - 7: ▷ Called at the end of a game, the agent returns an integer between 0-2 inclusive to select the next level to be played.
 - 8: **function** RESULT(sso, timer)
-

The parameters for each function are the Serializable State Observation (SSO) to access state information and timer to check the amount of time elapsed. The SSO provides information on the game’s current state through various class members, however for the purpose of this project only the image stored in this class will be used.

4.2.1 Aliens

Although in GVGA it is called aliens, it is closer to the classic game of Space Invaders. The rules for this particular version is simple, aliens spawn from the top left corner of the screen, these sprites move horizontally until they reach the end of the screen at which point they travel down vertically and move in the opposite horizontal direction. This process is repeated until either the alien contacts the player or the player kills all the aliens. An example of the game in action is shown in Figure 4.2.

The choice to select aliens as the first game to test the DQN was due to multiple factors. Firstly the game has an upper-bound on the number of steps a game can last for, that is when the last alien contacts the player. Secondly, the actions required for the agent to learn are limited to three: move left, shoot, move right. Lastly the game



(a) The blue projectile is a laser from the enemy aliens.



(b) The tiny yellow dot is the projectile from the player's spacecraft

Figure 4.2: Frames taken directly from the alien game in GVGAI. Green sprite: Alien, White sprite: Player, Grey sprite: Obstacle

has a very intuitive scoring system, each kill on an alien grants 2 points and the win condition is to destroy all aliens. Thus it would be logical to assume maximising the score would result in a winning outcome, however there is a slight catch on how the scoring system is implemented. Shooting and destroying a grey obstacle will also grant 1 point, this may have consequences in how the agent learns to play the game.

4.2.2 Preprocessing the frames

GVGAI allows the agent to see the state of the game in an image format. The alternative is a serialised format, however this will not be used. The aliens game produces RGB images of dimensions 300x110 pixels, the size of a single image is then:

$$\begin{aligned}
 \text{image size} &= \text{length} * \text{width} * \text{bit-depth} \\
 &= 300 * 110 * 24 \\
 &= 99 \text{ kB}
 \end{aligned}$$

Recall in Section 3.2.2 that the input to the neural network will be a sequence of images, in this case it is 4, this gives the network information on the velocity of objects. Thus to store 100,000 samples in experience replay will take:

Chapter 4. Experimentation

$$\begin{aligned} \text{total memory used} &= \text{size per frame} * \text{frames per sample} * \text{no. samples} \\ &= 99 \text{ kB} * 4 * 100,000 \\ &= 39.6 \text{ GB} \end{aligned}$$

The amount of memory used is impractical, furthermore the dimensions of the image causes the network to have much more parameters to learn, 16.7 million to be precise. The number of parameters affects the learning time and the amount of memory used in the training phase. Incidentally despite using a relatively high performance GPU (GTX 980) with 4GB Video RAM it was unable to run in this configuration and simply crashed. Thus it is necessary to reduce the number of parameters it is necessary to reduce the image dimensions and there are two simple methods of doing this:

- Downscale - Reduce the dimensions of the image by a factor of 2, the resulting dimensions is 150x55.
- Greyscale - Convert the RGB image into greyscale, thus reducing the bit-depth to 8.

This change reduced the number of parameters to 1.5 million and the memory usage is now:

$$\begin{aligned} \text{image size} &= 150 * 55 * 8 \\ &= 8.25 \text{ kB} \\ \text{total memory used} &= 8.25 \text{ kB} * 4 * 100,000 \\ &= 3.3 \text{ GB} \end{aligned}$$

Figure 4.3 shows the image reduction and the whole image stack that is fed into the network as input. It is worth noting that currently the GVGAI framework first writes the images to disk before sending the agent the image through main memory. It is logical to assume that there is significant overhead for this repeated process of writing every single frame for every single game to backing storage.

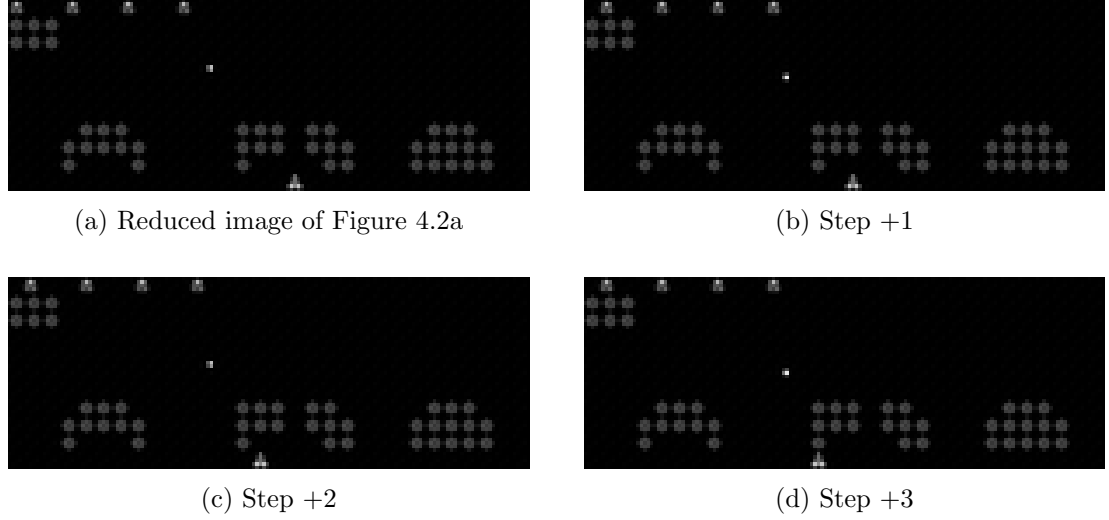


Figure 4.3: Example input to the neural network, 4 images form a stack.

4.2.3 Initialising replay memory

In Section 3.1.1 the network was trained immediately after the episode ended. This may have worked out for Frozen Lake as the game itself was simplistic, however this has the issue of highly dependent transitions as it was all gathered from the same episode. To really utilise the full effect of replay memory, it should be first initialised partially with random transitions. That is to say, use a random agent to play the game for k warm-up transitions, storing it in replay memory, and then begin training after this process is complete.

4.2.4 Implementation

Taking everything discussed so far into account, it is now possible to implement the DQN. The pseudocode for this implementation is shown in Algorithm 6 located in Appendix A, furthermore to visualise the algorithm Figure B.2 located in Appendix B shows the flow chart of the overall algorithm. For Algorithm 6, line 21 shows the case of repeating the same action in-between image stacks (Section 3.2.2) and line 28 the procedure for training the network is the same as in Algorithm 5. Table 4.3 shows all the different hyperparameters used for the DQN. Ideally it would have been better to have the replay memory size be at least double of what it is currently, unfortunately due to hardware constraints this value is sufficient given the circumstances.

| Parameter | Value |
|-------------------------------------|---------|
| Discount factor γ | 0.99 |
| Learning rate | 0.00025 |
| Mini-batch size | 32 |
| Replay memory size | 100,000 |
| Warm up stacks | 20,000 |
| Frames per stack | 4 |
| Frame downscaling factor | 2 |
| Exploration rate max | 1 |
| Exploration rate min | 0.1 |
| Episodes until exploration rate min | 1000 |

Table 4.3: Hyperparameters used for the DQN

4.2.5 Performance

To measure performance it is logical to plot the reward as a time series plot, this is due to the Q-function attempting to maximise the total discounted future reward as discussed in Section 2.2.3. As a result we expect to see the reward increase as time progresses. In Figure 4.4 we can see that this is indeed the case, however it is not very apparent.

Figure 4.5 shows the cumulative wins of the agent, this is a good indicator of how the agent performs throughout the learning process. Recall that for the Epsilon-Greedy policy the exploration rate affects the probability of selecting random actions vs selecting the action with the maximum Q value. Here the value of the exploration rate is initially set at 1 and is linearly annealed to 0.1 over 1000 episodes. This means that after 1000 episodes there is still an element of randomness, but this allows the agent to refine its existing policy. Figure 4.5 shows an increasing curve shape which

Chapter 4. Experimentation

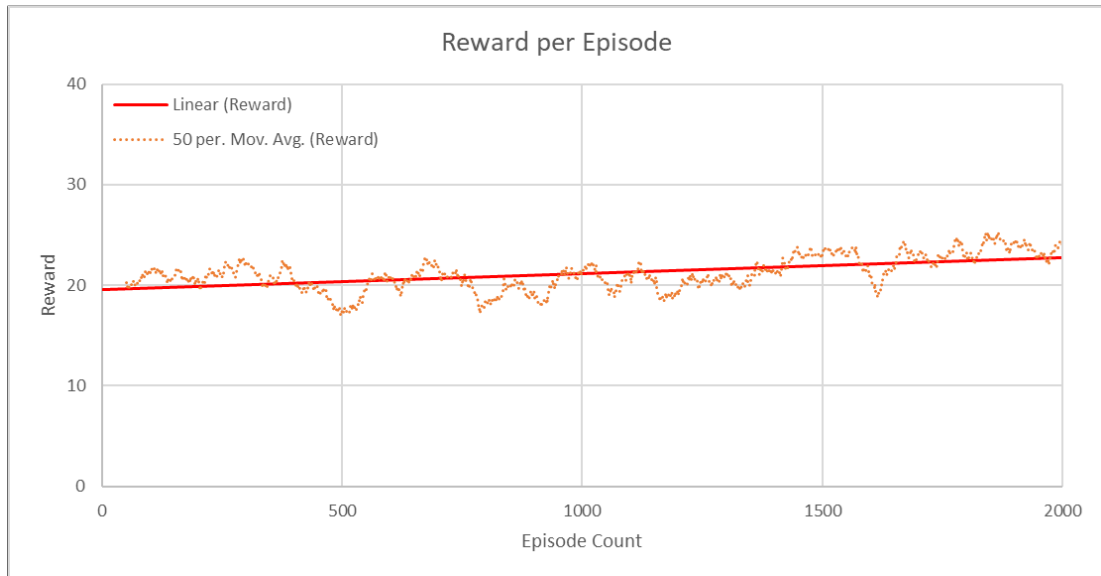


Figure 4.4: Rewards gained throughout the DQN learning process.

indicates that it has learned to win more games as the exploration rate decreases. In contrast a straight line for this plot would mean it has learned nothing.

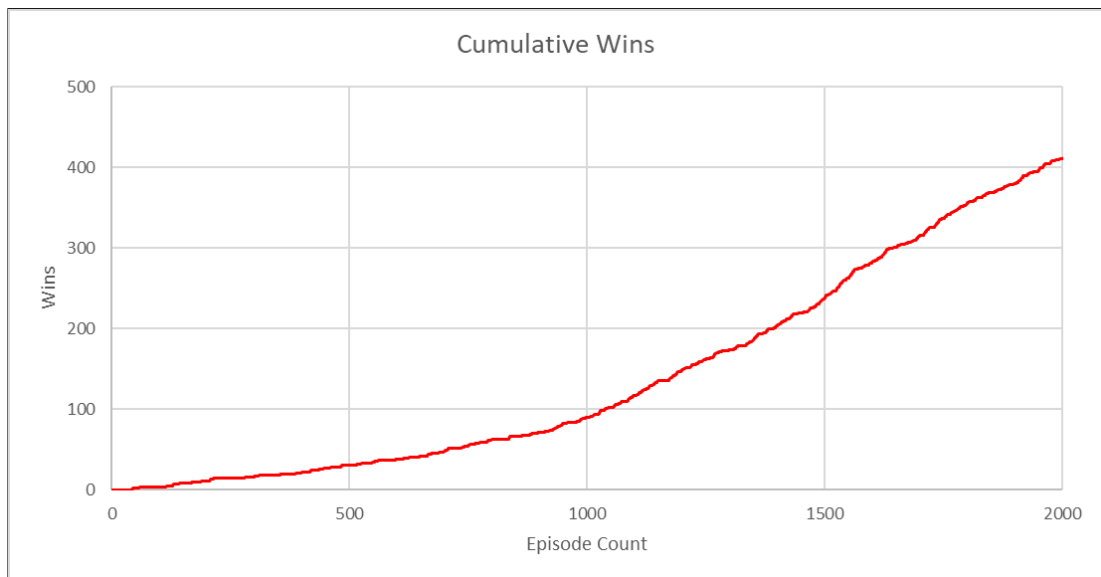


Figure 4.5: Total number of wins measured throughout the DQN learning process.

To summarise, Table 4.4 shows the percentages at different stages of the learning process. We see that there is a significant difference between the random moves played in the range 1-500 and the exploitation moves in the 1500-2000 range.

| Episode Range | Win % |
|---------------|-------|
| 1-500 | 6% |
| 500-1000 | 12% |
| 1000-1500 | 29% |
| 1500-2000 | 35% |

Table 4.4: Win percentages for different phases of the DQN learning

4.2.6 Concerns

Despite the success of the agent being able to play the Aliens game, it doesn't do it very well. While the performance is significantly better than the random agent, it still has a long way to go. Furthermore the whole process of training the DQN agent for 2000 episodes took approximately 12 hours. This is problematic when attempting to test different parameters, ideas or network architectures. Therefore it will only be possible to test the agents and any improvements/variations on a small set of different games due to time constraints.

Chapter 5

Improvements and Variations

This chapter focuses on improving the existing implementation of the DQN. Given the results shown in Section 4.2.5 it is clear that it still has a long way to go before the agent can be said to be any good.

5.1 Target Network

The regular DQN tends to overestimate the Q-values [24], in particular this is due to the max operation performed as shown in eq. 2.3. We can see this problem in a situation where for an arbitrary state, all actions have the same true Q value. However due to the noisy environment the predicted Q values for each action will be different. The max operation causes the action with the highest positive error to be picked and get propagated to subsequent states. This is the positive bias aka. value overestimation and can affect the stability of the algorithm. A proposed solution to this problem is to split the Q function into maximising action and value estimation as shown in eq. 5.1.

$$Q_1(s, a) = r + \gamma \cdot Q_2(s', \arg \max_a Q_1(s', a)) \quad (5.1)$$

To implement this in practice Q_1 (Primary Network) and Q_2 (Target Network) would be separate but identical neural network architectures. This is shown in Algorithm 3, it can be seen that Q_1 is focused on predicting the maximising action, while Q_2 is to estimate the actual Q values. During the training procedure only Q_1 will be updated, however to keep the networks in sync it is necessary to update Q_2 but only

periodically. For example after n timesteps set the weights of $Q_2 = Q_1$, in this specific case, the target network was updated every 5000 stacks.

Algorithm 3 Double-Q Training

```

1: function TRAIN()
2:   sample minibatch of transitions( $s, a, r, s', d$ ) uniformly from replay memory
3:   for each minibatch sample do
4:     if  $s'$  is not terminal then
5:        $f_2 = \text{forward pass using } Q_1 \text{ with } s' \text{ as input}$ 
6:        $f_3 = \text{forward pass using } Q_2 \text{ with } s' \text{ as input}$ 
7:        $\text{target} = r + \gamma \cdot f_3[\arg \max_a f_2]$ 
8:     else
9:        $\text{target} = r$ 
10:     $f_1 = \text{forward pass using } Q_1 \text{ with } s \text{ as input}$ 
11:     $f_1[a] = \text{target}$ 
12:  train network on batch

```

While the Double Deep Q Network (DDQN) does not always improve performance it was shown in the paper *Deep Reinforcement Learning with Double Q-learning* [24] that it substantially improves stability of learning which allows for learning of more complex tasks.

5.1.1 Evaluation

To compare how well DDQN performs against DQN we train on the same parameters and number of episodes. There is only one additional hyperparameter for DDQN which is the target network frequency, this frequency is set to update the target network every 5000 stacks. Figure 5.1 shows that there is a significant difference between the rate at which the reward gained increases as more episodes are played. Figure 5.2 shows again there is a significant difference to how much the DDQN agent wins over the DQN. Note that both agents perform similarly at the beginning when the exploration rate is high, however it can be seen how different their exploitation of the environment is after the exploration has settled down to the minimum at around the 1000 episode point.

While these improvements to performance are a good sign, the original motivation for DDQN was that the original DQN would overestimate the Q values. It would then be a better idea to compare the different Q values for both of these implementations. Figure 5.3 shows the comparisons for the mean Q values between the DDQN and

Chapter 5. Improvements and Variations

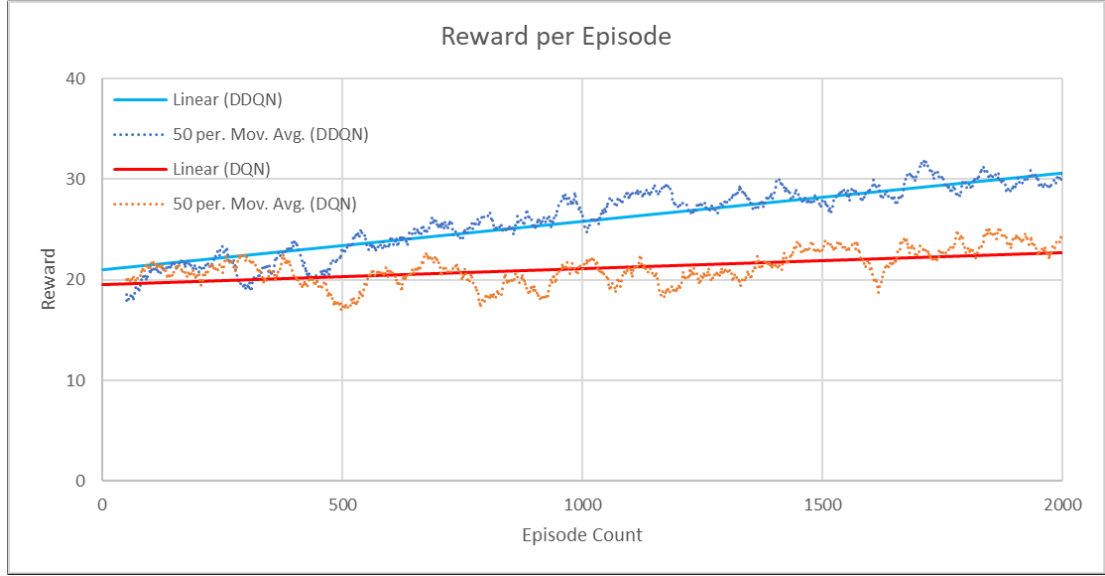


Figure 5.1: Comparison of the rewards per episode of for DDQN and DQN

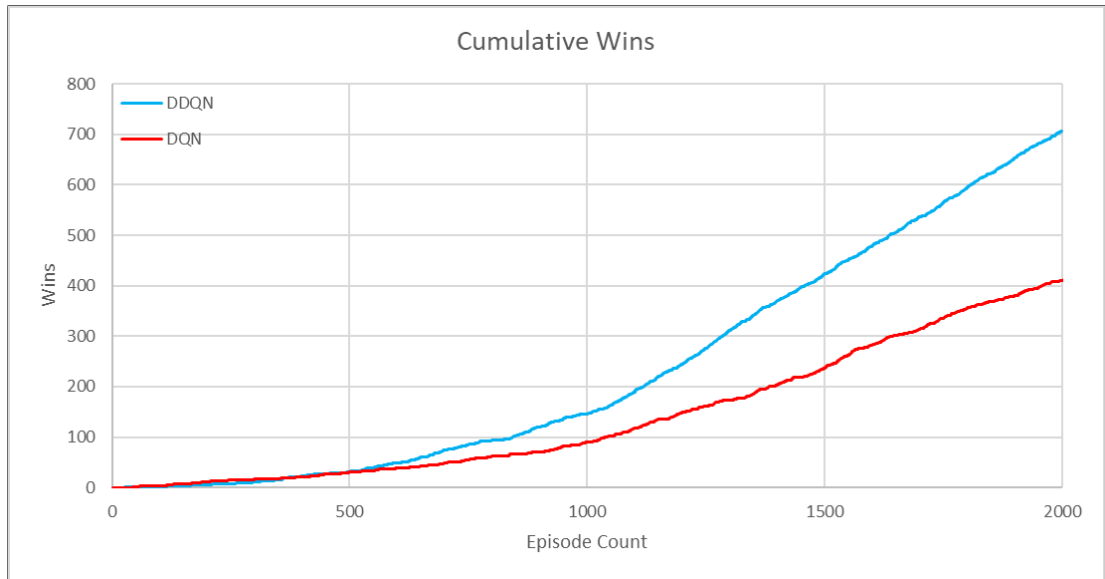


Figure 5.2: Comparison of the cumulative wins of for DDQN and DQN

DQN. It can be seen that the DQN initially estimates the Q values to be very high, this may be due to their random initial starting conditions however we see that for the DQN its estimation of the Q values are far more jittery than the DDQN. This is most likely the max operator selecting less than optimal actions with high positive error. Despite this, both implementations still settle down to similar Q values near the end of training, it would be interesting to see if extending the training time would make these plots diverge, however due to time constraints this is not given priority.

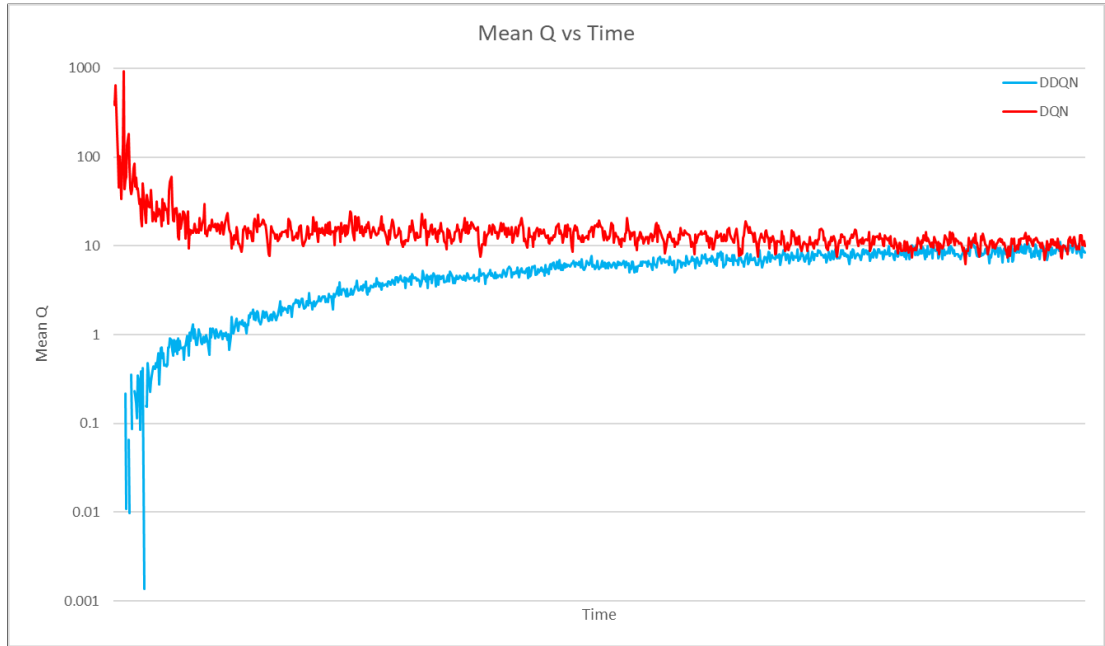


Figure 5.3: Comparison of the mean Q values for DDQN and DQN

Table 5.1 shows the performance of the DDQN measured by win rate for different episode ranges. The win percentage comparison against DQN is shown in Figure 5.4. Note that for the range 1-500 DQN actually performs slightly better but during this episode range, both agents are mostly playing random moves, thus we can't say with certainty that this means anything. The most interesting area is after 1000+ episodes as this is where the exploration rate settles down, we can

| Episode Range | Win Rate |
|---------------|----------|
| 1-500 | 3.0% |
| 500-1000 | 17.4% |
| 1000-1500 | 47.4% |
| 1500-2000 | 56.6% |

Table 5.1: Win percentages for different phases of the DDQN learning

see there is a significant difference between the two implementations, with the DDQN being able to better exploit it's knowledge. Overall this is an improvement, with greater stability of learning the new implementation was able to outperform the standard DQN, however there is still further room for improvement.

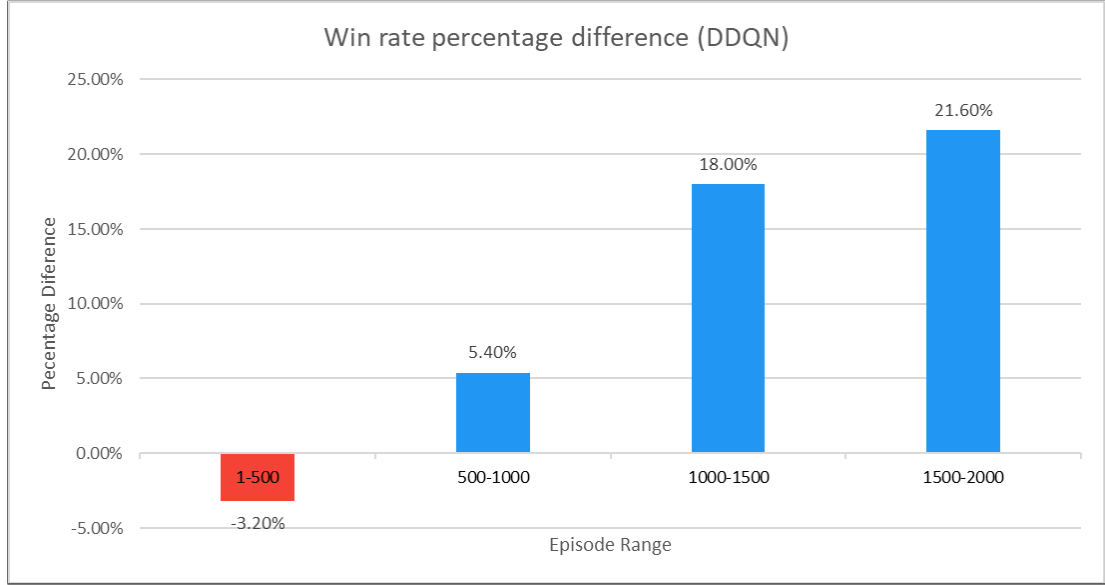


Figure 5.4: Comparison of the win rates between DQN and DDQN during different episode ranges.

5.2 Prioritised Experience Replay

With the original experience replay, transitions are sampled uniformly but this may not be the best approach. This issue was also acknowledged in the original paper: *Human-level control through deep reinforcement learning* [1], and the paper *Prioritized Experience Replay* [23] provides a solution to this. This is due to the fact that some states can be more utilised than others, i.e we can learn more from an interesting state than a dull state. However sampling uniformly disregards this and assumes all states are of equal value despite this not being the case. To remedy this issue we can assign some value to each transition as an indicator of its value and sample with priority, this leads to the name Prioritised Experience Replay (PER). However the next issue is how to assign this value? A solution to this problem is to use the network's error for a transition. This is because the error tells us how confident the network is for a given state, with a higher error telling us that it is more surprised about this transition. Recall that the loss from Equation 3.5 is simply the squared error, and we want the absolute error as the sign doesn't matter.

Chapter 5. Improvements and Variations

$$error = | \underbrace{r + \gamma \cdot \max_{a'} Q(s', a)}_{\text{target}} - \underbrace{Q(s, a)}_{\text{prediction}} | \quad (5.2)$$

However for Equation 5.2 the target refers to the standard DQN. While it is possible to implement PER for DQN, it will only be implemented for the DDQN due to the lengthy time it takes for training and because the DDQN has shown to perform better. Thus the error for DDQN is:

$$error = | \underbrace{r + \gamma \cdot Q_2(s', \arg \max_a Q_1(s', a))}_{\text{target}} - \underbrace{Q_1(s, a)}_{\text{prediction}} | \quad (5.3)$$

One approach to PER is using proportional prioritisation, which firsts converts the error into priority:

$$p = (error + \beta)^\alpha \quad (5.4)$$

Where:

α : a value between 0 and 1 which controls the amount of prioritisation used, with $\alpha = 0$ being the uniform case.

β : a small positive constant that will ensure that all transitions will not have zero priority

Finally priority is translated to probability of being chosen during the replay phase. The probability for sample i being picked for replay is:

$$P(sample_i) = \frac{p_i}{\sum_k p_k} \quad (5.5)$$

Where:

$k = 1, \dots, N$

N = total number of samples

5.2.1 Sum Tree

Before with DQN and DDQN the transitions were stored in a deque, this worked because we wanted old transitions to be replaced with new ones if the deque capacity was exceeded, however this won't work with priorities. A fast and efficient solution for this would be to use an unsorted sum tree data structure [26]. A sum tree is a binary tree where the parent's nodes are the sum of its children, with the samples themselves stored in the leaf nodes. This allows gives $O(1)$ for insertion/updates and $O(\log n)$ for sampling.

5.2.2 Evaluation

As with Section 5.1.1 the same parameters were used to compare the new implementation against both DQN and DDQN. We can see from Figure 5.5 that the new implementation performs marginally better than standard DDQN.

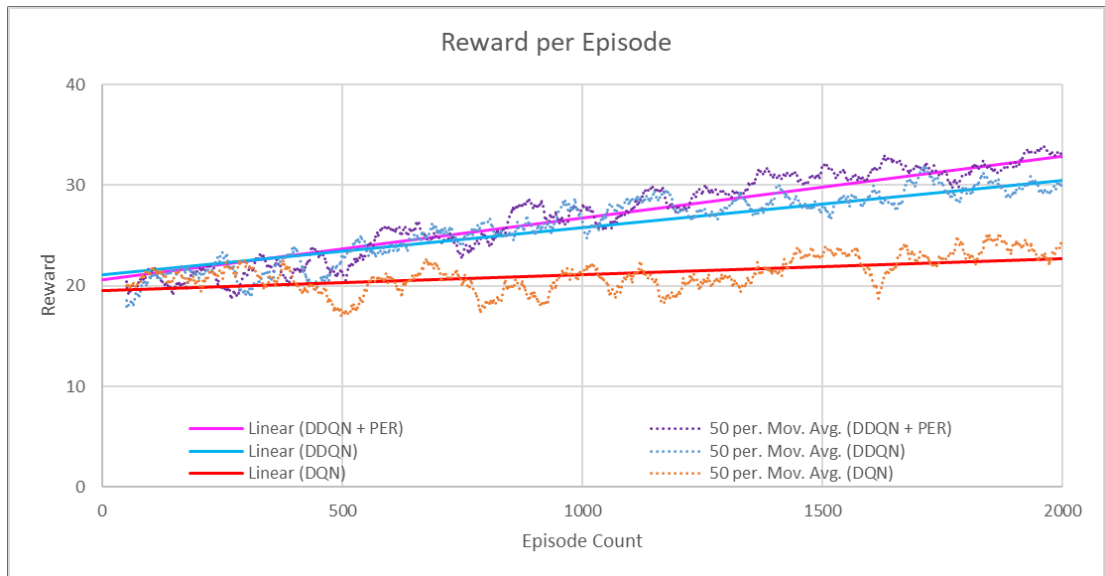


Figure 5.5: Comparison of the rewards per episode of for the different implementations

As such, we should also be expecting an improvement in the win rates, however Figure 5.6 gives a different impression. It would seem that despite gaining a higher on average reward over the 2000 episodes than DDQN it won less games on average in the same time frame Figure 5.7 shows a large contrast between these two implementations. The 1-500 episode range can be ignored again due to randomness but the

Chapter 5. Improvements and Variations

remaining episodes ranges show that with PER it performed significantly worse than with standard DQN for win rates. Furthermore while the 1000-1500 episode had a performance difference of over -15% it seemed to correct that in the next 500 episodes.

Overall, despite gaining more average reward, it would seem that it had a negative impact on the win rate. An explanation for this would be due to how the reward function was structured. Recall that from Section 4.2.1, a positive score of +1 is given to destroying an obstacle and +2 score for destroying an alien. However due to the reward clipping as discussed in Section 3.2.3, it values these positive rewards equally. Since destroying an obstacle doesn't help the agent win the game but it is given as much reward as destroying an alien which does, it's clear why this degradation

in win rate is happening. The reason why a positive score is given for destroying an obstacle is due to the deceptive nature of the game, and it seems that the agent has fallen into a fallacy of believing that maximising the score of the game gives greater performance. Amidst this negativity, it has at the very least shown that PER gives a higher average reward, if only the reward was more directly correlated with winning the game then it should perform much better.

| Episode Range | Win rate |
|---------------|----------|
| 1-500 | 3.4% |
| 500-1000 | 12.6% |
| 1000-1500 | 31.8% |
| 1500-2000 | 54.2% |

Table 5.2: Win percentages for different phases of the DDQN + PER learning

Chapter 5. Improvements and Variations

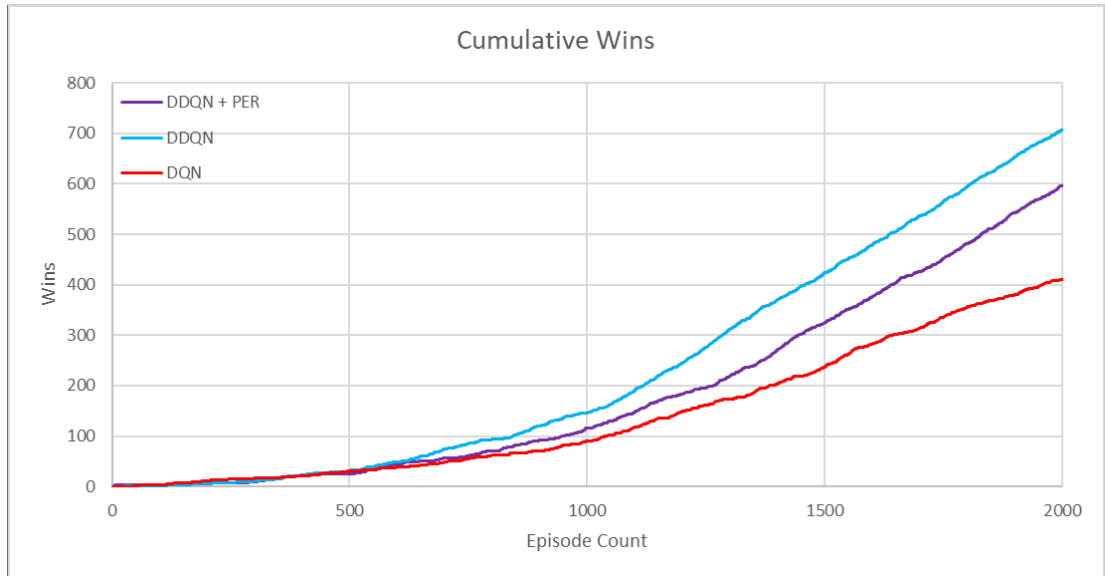


Figure 5.6: Comparison of the cumulative wins for the different implementations

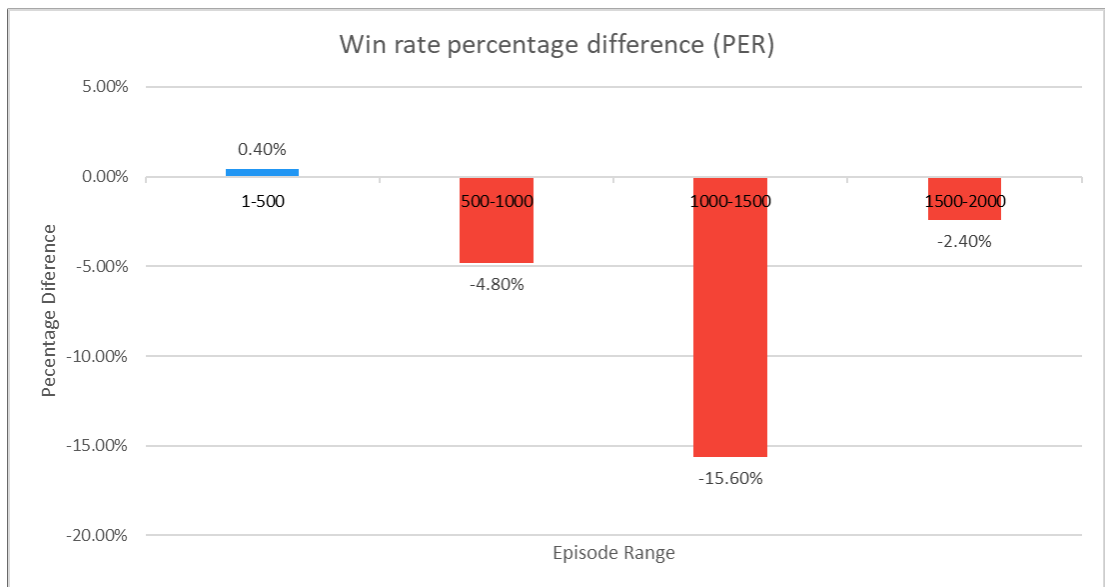


Figure 5.7: Comparison of the win rates against the standard DDQN during different episode ranges.

5.3 Alternative Reward Function

As discussed in Section 5.2.2, given that the current reward function is deceptive, some adjustments need to be made. In addition to clipping the rewards, we can directly weight all the rewards for an episode dependent on the outcome of the game. In particular, assign a penalty of -0.5 for every transition if the outcome of the game resulted in a loss, otherwise leave everything unchanged. This means we can no longer simply store each transition into replay memory directly as they come in, instead we store these transitions in a temporary location until the end of the episode where the outcome is determined. Afterwards the rewards of each transition are altered if the result was a loss and then it is stored in replay memory. Note that it is not necessary to alter the original reward function, we only change the reward values if the game results in a loss. The pseudocode is shown in Algorithm 4. Note that on line 9, we check to see if subtracting 0.5 would result in the reward for that transition to be below -1. This is because we want our reward ranges to be between +1 and -1 like before.

Algorithm 4 Storing transitions

```

1: function STORE_TRANSITION( $s, a, r, s', d$ )
2:   append transition ( $s, a, r, s', d$ ) to temporary storage
3:
4:   if  $s'$  is terminal then
5:     for each transition  $t$  in temporary storage do
6:
7:       if game was lost then
8:          $t_{reward} = t_{reward} - 0.5$ 
9:         if  $t_{reward} < -1$  then
10:            $t_{reward} = -1$ 
11:
12:       store  $t$  in replay memory

```

5.3.1 Evaluation

Figure 5.8 shows the average reward gained per episode, the moving averages were removed to improve readability. We can see that the altered reward function returns a lower overall average reward than DDQN + PER. This is expected since rewards

Chapter 5. Improvements and Variations

| Episode Range | Win Rate |
|---------------|----------|
| 1-500 | 4.0% |
| 500-1000 | 22.2% |
| 1000-1500 | 58.0% |
| 1500-2000 | 68.2% |

Table 5.3: Win percentages for different phases of the DDQN + PER + New Reward learning

are highly punished on a loss. Figure 5.9 shows that the new reward function wins more on average than any of the other implementation, this is empirical evidence that the deceptive scoring system actually led to a poorer performing system. However it should be noted that this altered reward function is flawed because it punishes all states equally on a loss. The loss of the game may be attributed to only a handful of states, in this specific case that was when obstacles were being destroyed. Ideally the best case scenario would be to remove the deceptive scoring system altogether, however given the circumstances, this altered reward function is an acceptable compromise.

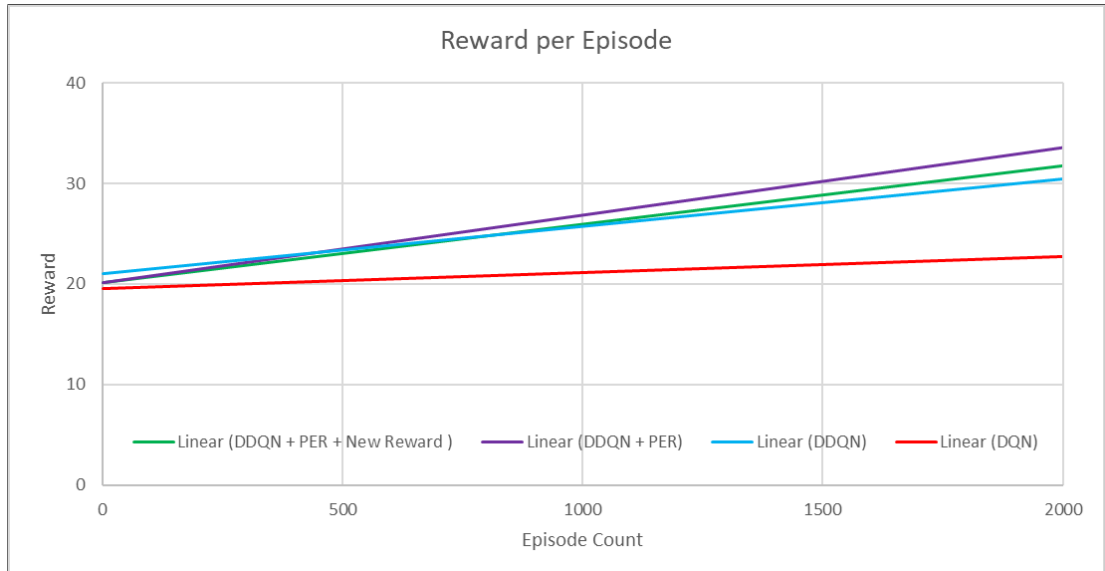


Figure 5.8: Comparison of the rewards per episode of for the different implementations. The moving averages were removed due to cluttering the plot.

Chapter 5. Improvements and Variations

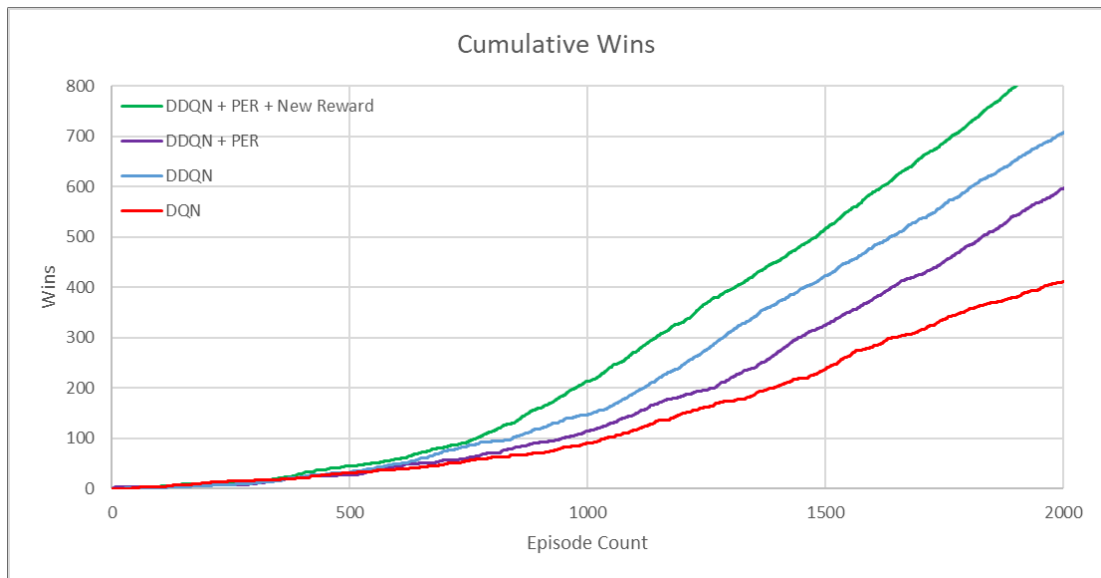


Figure 5.9: Comparison of the cumulative wins for the different implementations

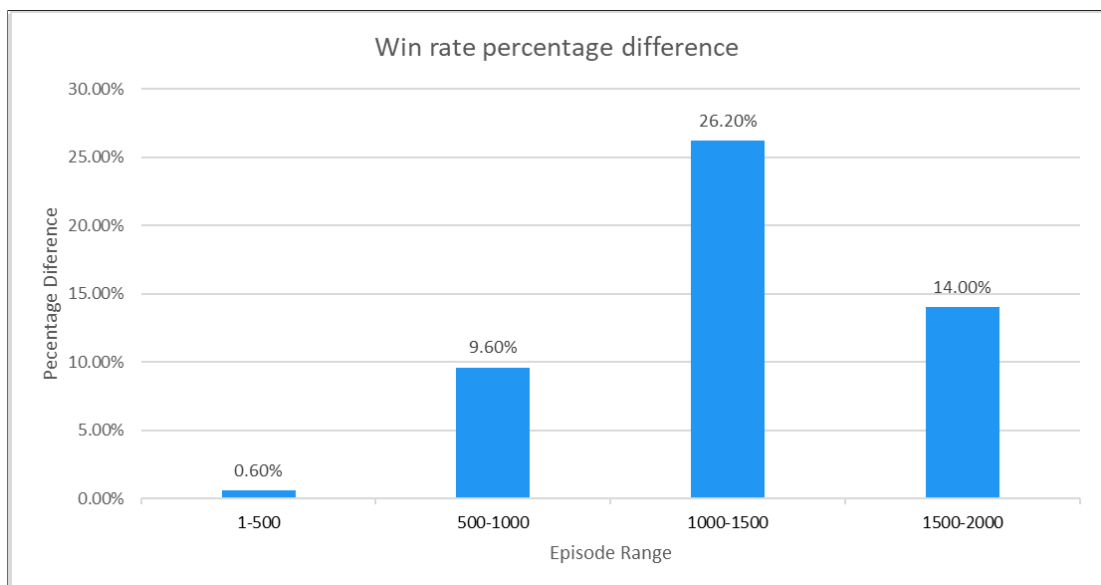


Figure 5.10: Comparison of the win rates against the old reward function during different episode ranges.

5.4 Dueling Network

The Q function $Q(s, a)$ tells us how good it is to take action a whilst in state s . This function however can be further split into two more fundamental functions:

$$Q(s, a) = V(s) + A(a) \quad (5.6)$$

Where $V(s)$ is the value function and tells us how good it is to be in state s , whilst the advantage function tells us how good it is to take action a in comparison to other actions. With Dueling Networks it separately computes the value and advantage function separately and finally summing at the output, this can be seen in Figure 5.11. Although this action of computing functions separately only to just sum them again seems pointless, the agent may not always need to care about both the value and advantage functions at the same time. Consider the following example: the agent is at a point in the game where it is close to winning the game. Thus simply being in this state gives a high reward which is associated with the value function, however the advantage differs for each action depending on how it will affect the agent from this point on.

There are three different implementations for the Q function when using dueling networks, here we will use the mean and max variants of the dueling network:

$$Q(s, a) = V(s) + (A(a) - \frac{1}{|A|} \sum_{a'} A(a')) \quad (5.7)$$

$$Q(s, a) = V(s) + (A(a) - \max_{a'} A(a')) \quad (5.8)$$

Since the dueling network only changes the structure of the neural network we will continue to use PER and the new reward function.

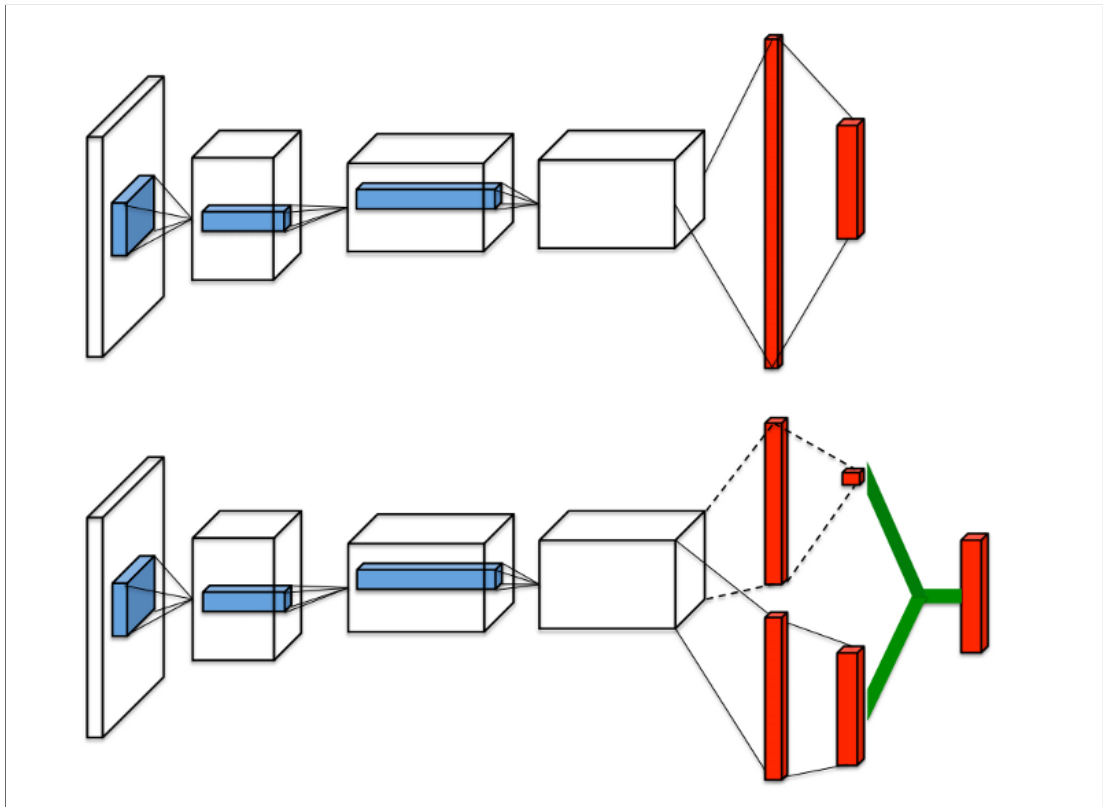


Figure 5.11: Top: Standard DQN with single stream of Q value. Bottom: Dueling Network where the value and advantage functions are split into separate streams, only to be summed at the output. Diagram from the paper *Dueling Network Architectures for Deep Reinforcement Learning* [2]

5.4.1 Evaluation

As shown in Figures 5.12, 5.13 and 5.14, the performance of the mean variant performs slightly better as time progresses. This is likely due to the max operator again maximising the noisy advantage values. Overall the two implementations perform relatively similarly, however when compared to the DDQN + PER + New Reward, there is a significant difference of over 10% in win rate for the 2000-2500 episode range as shown in Figure 5.15. Whilst there was some expectation that the dueling network would outperform the DDQN architecture (and it did for the other episode ranges), it is important to note that the original paper *Dueling Network Architectures for Deep Reinforcement Learning* [2] was aimed at environments with large action choices, 18 in the Atari platform. On the other hand the game of Aliens has a mere 3 actions and the maximum possible in the GVGAI framework is 5.

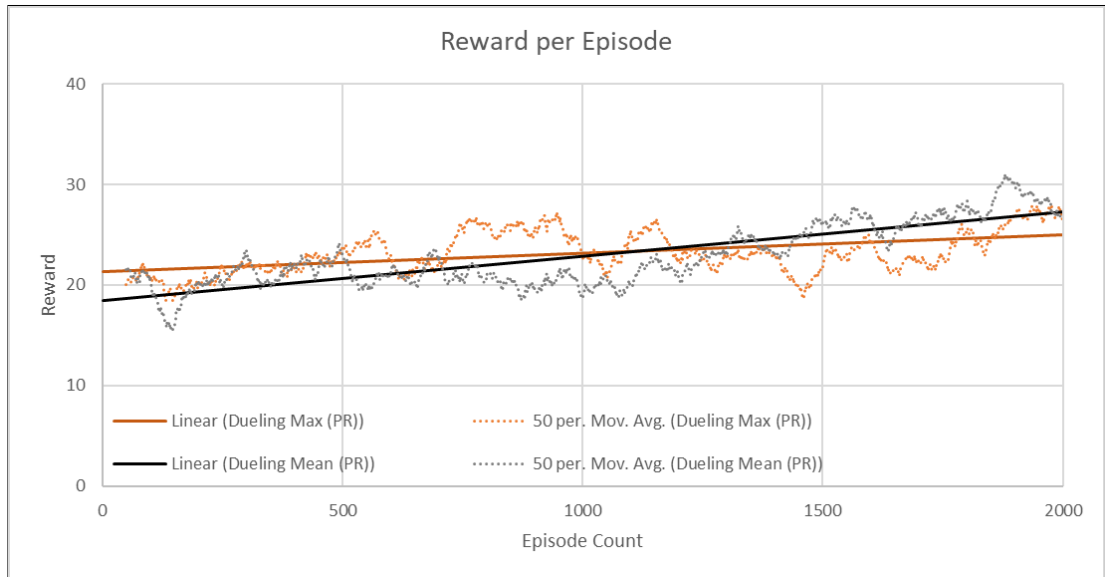


Figure 5.12: Comparison of the rewards per episode of for the two different implementations of the dueling network.

Chapter 5. Improvements and Variations

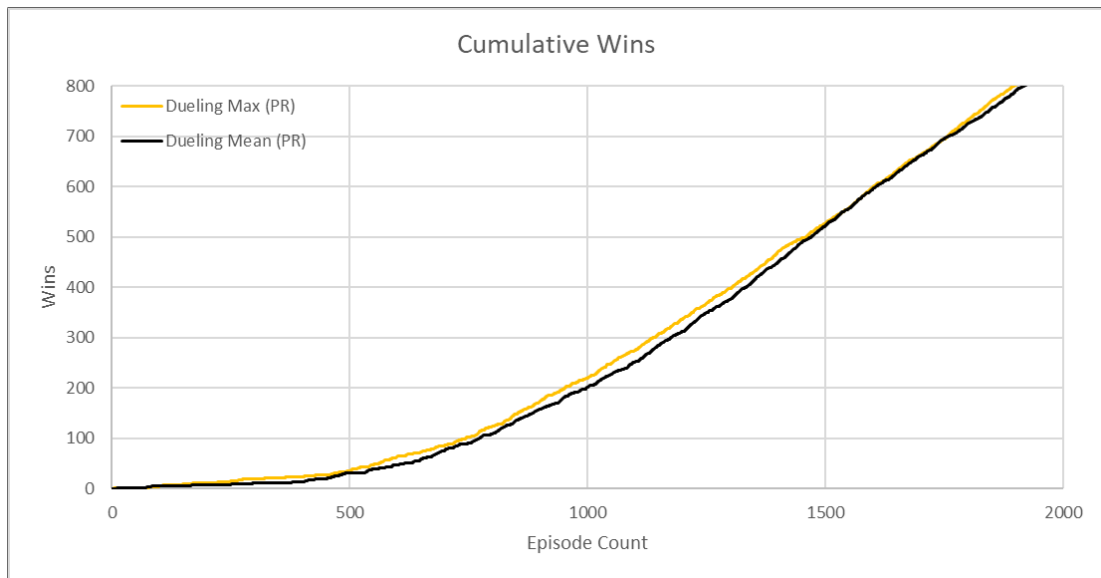


Figure 5.13: Comparison of the cumulative wins for the two different implementations of the dueling network.

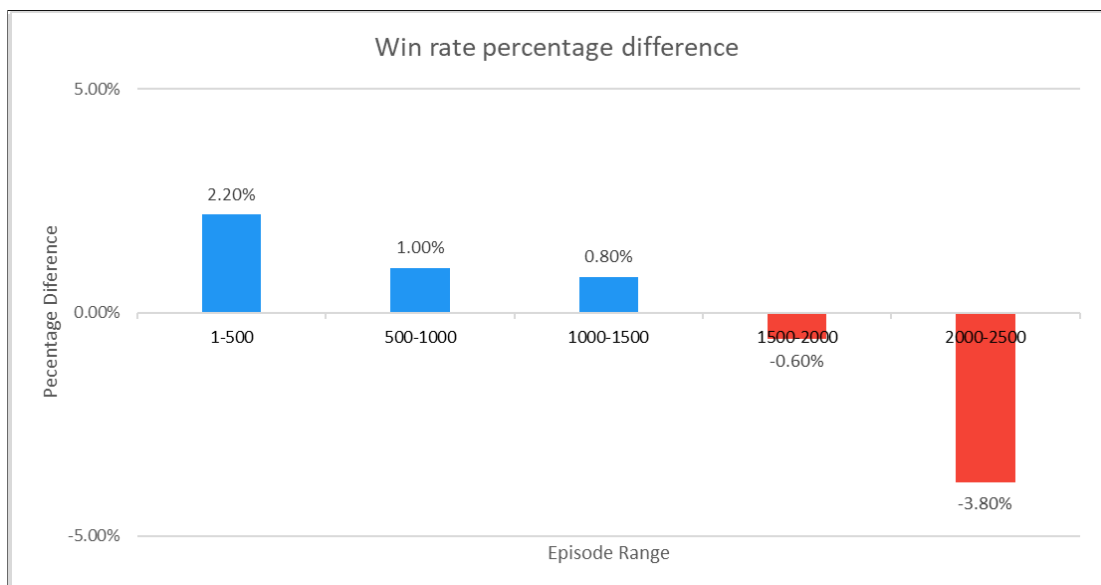


Figure 5.14: Comparison of the max variant versus the mean variant during different episode ranges. Positive values indicate the max variant perform better and the mean variant worse, and also vice versa.

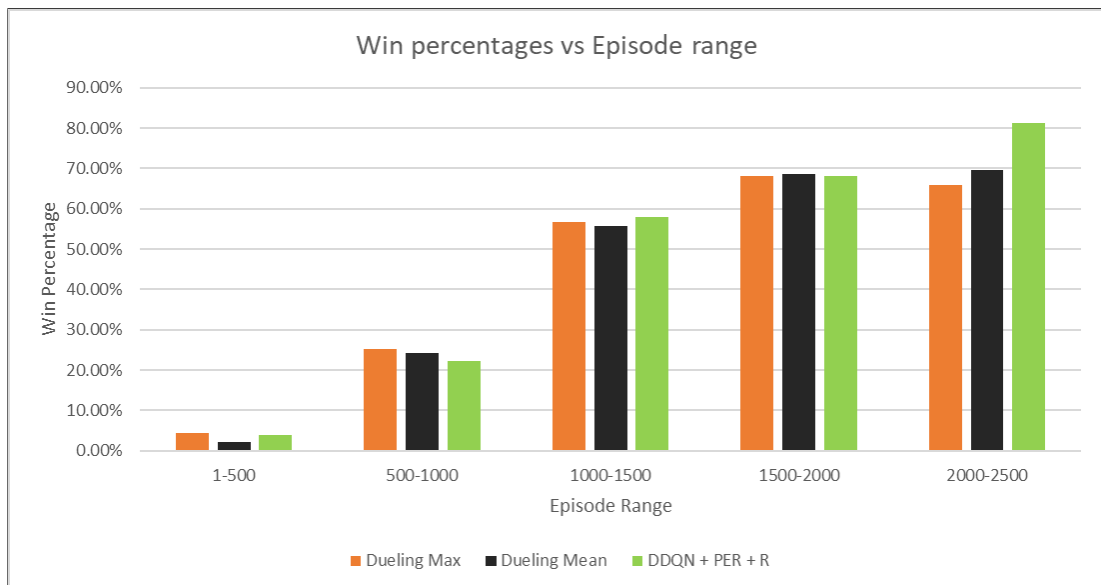


Figure 5.15: Comparison of the dueling network variants and DDQN + PER + New Reward

Chapter 6

Application to other games

Up until now, all experimentation has been done on the game of Aliens. One of the original goals of the project was to implement a generalised video game playing AI. To achieve this objective will require testing the agent on a variety of games. Unfortunately due to the time constraints caused by long training times, only a select few games could be tested on a single learning type agent. Ideally all of the deep Q network agents (DQN, DDQN and Dueling) would be tested including all the different variations (PER and reward function), however only the DDQN + PER + New Reward was selected as it performed the best out of all the other agents in the previous experiments. Each game selected has elements of deception; a higher score does not directly correlate with winning.

6.1 Monte Carlo Tree Search

The DDQN is compared against Monte Carlo Tree Search (MCTS). It is a type of heuristic tree searching algorithm and was employed by AlphaGo. MCTS is a popular algorithm due to its efficiency and applicability to a large variety of games and domains. The MCTS algorithm has four simple steps:

1. Selection - Starting from the root node, select child nodes until reaching a leaf node.
2. Expansion - While the leaf node is not terminal, create one or more child nodes and select one of these new nodes c .

Chapter 6. Application to other games

3. Simulation - Play random moves from node c until a condition is met e.g terminal state has been reached aka. rollouts.
4. Backpropagation - The result is propagated back up the tree to update connected nodes.

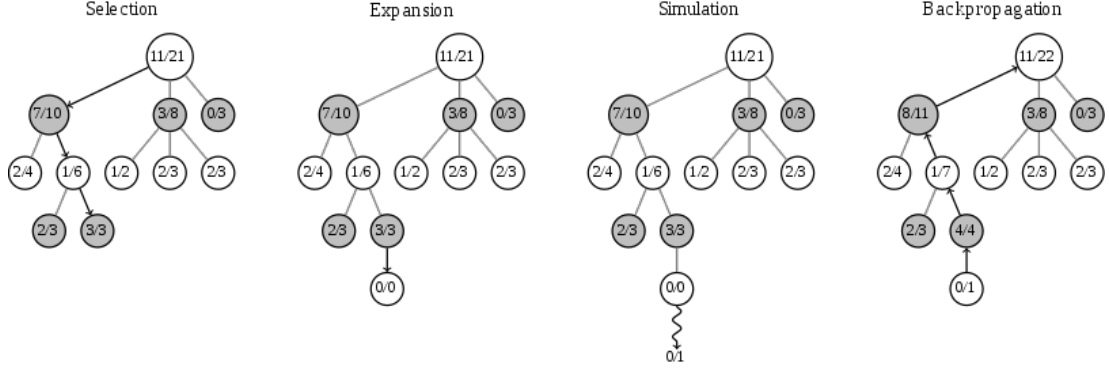


Figure 6.1: Example steps of Monte Carlo Tree Search

MCTS also tackles the exploration vs exploitation problem (Section 2.3) however it uses the Upper Confidence Bound 1 applied to trees (UCT) formula [27] for each node:

$$UCT = \frac{w_i}{n_i} + c \cdot \sqrt{\frac{\ln N_i}{n_i}} \quad (6.1)$$

Where:

w_i : the number of wins for the current node after move i

n_i : the number of simulations for the current node after move i

N_i : the total number of simulations for the current node after move i

c : a constant exploration parameter

Overall the MCTS algorithm is a good generalised AI for video game playing. Its node selection strategy means that the most promising nodes are visited first and more frequently than less important states similar to how DDQN uses PER (Section 5.2).

| Controller \ Game | Aliens | Chase | Decepti-zelda | Wafer Thin Mints |
|-------------------|-----------------|---------------|-----------------|------------------|
| Human | 42.6 (100%) | 1.2 (0%) | 10.6 (80%) | 11 (0%) |
| DDQN | 56.3 (82.6%) | 0.7 (0%) | 1.99 (99%) | 10.2 (33%) |
| MCTS | 79.2 (100%) | 3.5 (3.7%) | 1.91 (84.4%) | 9.9 (75.5%) |

Table 6.1: Performance of agents on different games. Each cell details the average score of the agent with the win rate in brackets. The DDQN includes Prioritised Experience Replay and the new reward function, the MCTS has access to a forward model to simulate future states.

6.2 Results

DDQN and MCTS both played 50 episodes of each game, whilst Human played 10, and their averages were taken. Table 6.1 shows the controllers performance on a variety of games and it can be seen that MCTS beats the learning agent every time except on Deceptizelda. However it is important to note that there are several major differences between the two AI agents:

- Access to simulations - MCTS has the ability to simulate future moves, it can avoid unfavourable future states. DDQN does not have this ability and must learn from experience.
- Time limitations - MCTS is given 40ms to decide its next action, DDQN has no time limit and was given up-to 24 hours to train.
- Independent actions - MCTS can select any action during each frame of the game, DDQN employs frame skipping (Section 3.2.2) and as a result actions are repeated for four frames.
- Differing Inputs - MCTS uses the SSO that contains explicit information about the game state, DDQN is given only the pixels of the game and current score.

Chapter 6. Application to other games

It is most likely the lack of a forward model to simulate future states is the cause of the large difference in performance. In addition, frame skipping plays a factor in causing the DDQN to perform poorly since it will be unable to ever reach certain states. However given that AlphaGo uses a combination of both agents, finding a method to combine them will give better performance than either individual as evidenced by AlphaGo's superhuman performance. On the other hand, the DDQN does indeed show human level performance even if the environments were specifically deceptive.

Chapter 7

Conclusion

Overall the investigation into learning type agents, specifically Q-Learning, showed promising results. The results drawn from this project give empirical evidence that while the Q-Learning agent didn't perform the best out of the competition, it still performed remarkably well considering the environments did not have the convenient features as discussed in Section 1.1. It is important to note that this type of learning agent uses the same inputs as humans and learns in a similar way as humans; through trial and error. There is no forward model to simulate anything, and despite this, the learning agent was able to produce similar results to MCTS. This also means that learning agents can be applied to games where there is no forward model or serialised state, incidentally most games for humans are like this.

Reinforcement learning is a major step forward for generalised video game playing AI however there are major drawbacks. These types of agents require tremendous amount of resources, both computational and temporal, and while it can play theoretically any game it has to be trained individually on that game. It also suffers from the same problems as many other AI when dealing with large state spaces.

7.1 Further work

In search of methods to further increase performance of the agent there are a few things yet to consider and due to time constraints was not discussed and implemented in the project.

7.1.1 Alternative Action Selection Policies

As discussed in Section 2.4.3, Epsilon-Greedy uses a uniform distribution to select the random actions when it does not use its current max Q value action. This can be seen in Figure 7.1.

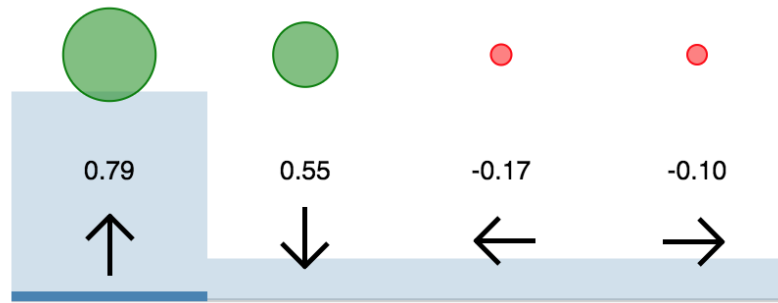


Figure 7.1: A diagram of four different actions the agent can take. The numbers represent arbitrary Q values. The height of the blue bars represent the probability of the action being selected. Note that the three bars on the right have the same height, this means they each have an equal probability of being selected. Image from a blog by Arthur Juliani.

Alternatively, instead of having equal probability on the remaining actions, it is possible to assign the probability of the action being selected proportional to the Q value for that action. This can be seen in Figure 7.2.

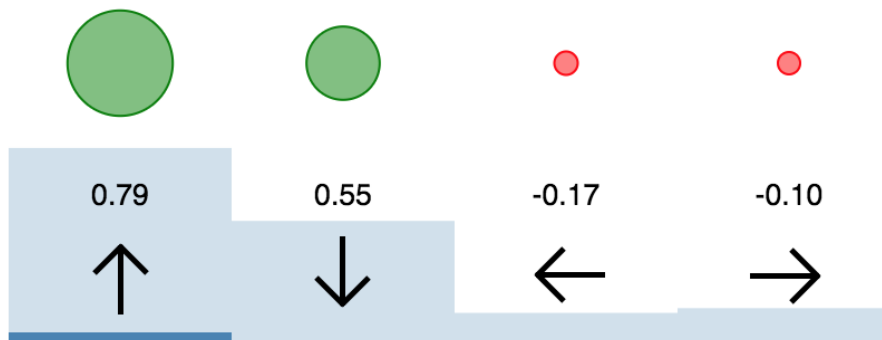


Figure 7.2: This diagram has the same Q values as 7.1. Note that the three bars on the right have differing heights proportional to their Q values. Image from a blog by Arthur Juliani.

7.1.2 Asynchronous Actor-Critic Agents

Instead of using just a single agent to learn, why not have multiple agents playing at the same time? Asynchronous Actor-Critic Agents also known as A3C, is a method of using concurrent parallel agents to update a global network. The paper *Asynchronous Methods for Deep Reinforcement Learning* [28] showed that it reduced training speeds by approximately half. This approach differs slightly however, from value iteration methods like Q-learning, the network will now estimate a value function $V(S)$ (how good it is to be in state s) and a policy $\pi(s)$ (a set of probabilities for each action).

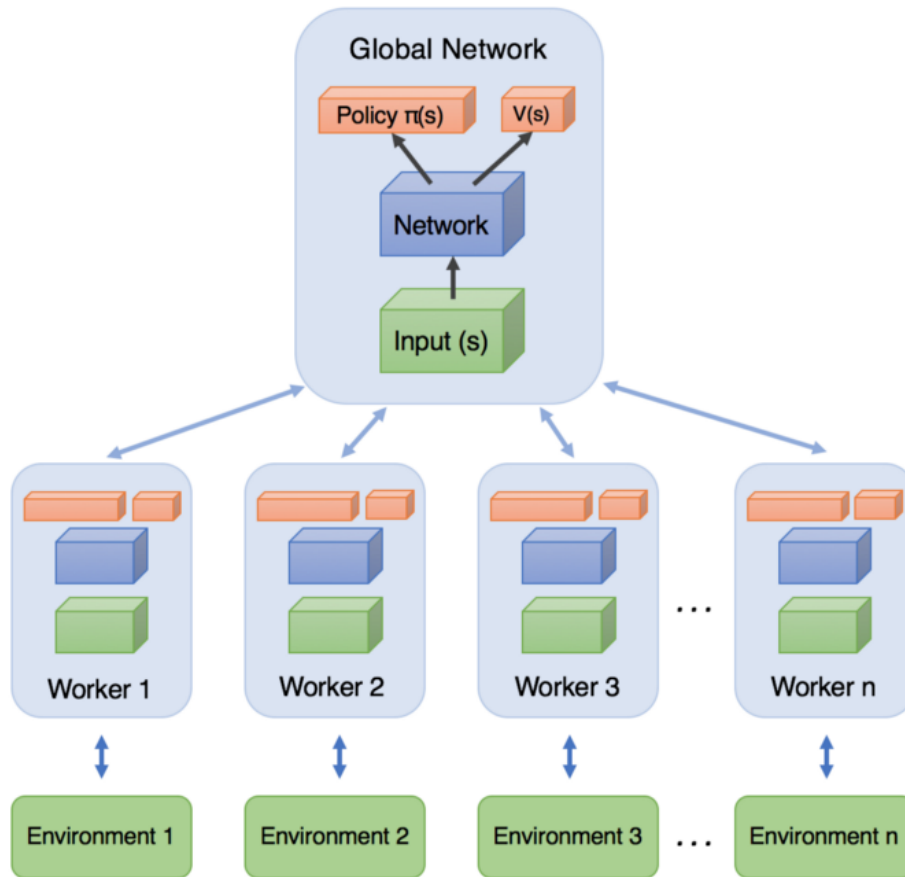


Figure 7.3: A diagram of the A3C architecture. Image from a blog by Arthur Juliani.

7.1.3 Hyperparameter Tuning

Hyperparameters are variables in the system that affect the training process. Modifying these parameters can affect the system’s performance, and finding the optimal parameters to maximise performance can be a difficult challenge. The hyperparameters used in the project were based mainly on the original paper *Human-level control through deep reinforcement learning* [1], and some educated guessing. A guaranteed method to find the best parameters would be to use a systematic approach like grid search to find the most optimal parameters for the agent, this basically trains the agent using different permutations of parameters. However considering the amount of time the training process takes and how many hyperparameters there are, this method is impractical. The most promising hyperparameters to look experiment with are: the size of replay memory and the rate at which the exploration rate reduces to the minimum. In this project both hyperparameters were set to a relatively low value due to constraints on available resources, increasing these values may help significantly.

7.1.4 Increased Training Time

A simple and effective solution that will likely give the agent increased performance is to simply allow it to train longer. In this project the agents were allowed to train for a period of 12-18 hours and it was clear from the results that the agent had yet to converge to it’s optimal performance. Allowing the agents an additional 24 hours should be enough to allow the agent to settle down to it’s peak performance.

Appendix A

Pseudocode

Appendix A. Pseudocode

Algorithm 5 Neural Network Q-Learning

```
1: initialise experience replay memory  $M$ 
2: initialise learning rate  $\alpha$ 
3: initialise exploration rate  $\beta$ 
4: initialise discount factor  $\gamma$ 
5: for each training episode do
6:   observe initial state  $s$ 
7:   while game not finished do
8:     get random number  $x$ 
9:     if  $x < \beta$  then
10:      select action from  $Q$ -table  $\max_a Q[s][a]$ 
11:    else
12:      select a random action
13:      perform action above and obtain reward  $r$  and new state  $s'$ 
14:      store transition  $(s, a, r, s', d)$  into replay memory  $M$ 
15:     $s = s'$ 
16:     $\beta = \beta - \text{decay rate}$ 
17:
18:    if length of  $M > \text{batch size}$  then
19:      TRAIN()
20:
21: function TRAIN()
22:   sample minibatch of transitions  $(s, a, r, s', d)$  uniformly from replay memory
23:   for each minibatch sample do
24:     if  $s'$  is not terminal then
25:        $f_2 = \text{forward pass using state } s' \text{ as input}$ 
26:        $\text{target} = r + \gamma \cdot \max_{a'} f_2$ 
27:     else
28:        $\text{target} = r$ 
29:        $f_1 = \text{forward pass using state } s \text{ as input}$ 
30:        $f_1[a] = \text{target}$ 
31:   train network on batch
```

Appendix A. Pseudocode

Algorithm 6 GVGAI DQN

```
1: function INIT(sso, timer)
2:   initialise experience replay memory  $M$ 
3:   initialise warm up steps  $k$ 
4:
5: function ACT(sso, timer)
6:   image = get preprocessed frame
7:   add preprocessed frame onto stack
8:   if images in stack == 4 then
9:     currentState = get images from stack
10:    action = get action from DQN with currentState as input
11:
12:    if previous state is not null then
13:      store transition (prevState, prevAction, prevReward, currentState, 0) in
replay memory
14:
15:    prevState = currentState
16:    prevAction = action
17:    prevReward = get reward for this state
18:    clear image stack
19:
20:  else
21:    action = previous action
22:  return action
23:
24: function RESULT(sso, timer)
25:   prevReward = calculate reward for terminal state
26:   store transition (prevState, prevAction, prevReward, currentState, 0) in replay
memory
27:   if size of  $M > k$  then
28:     TRAIN()
```

Appendix B

Diagrams

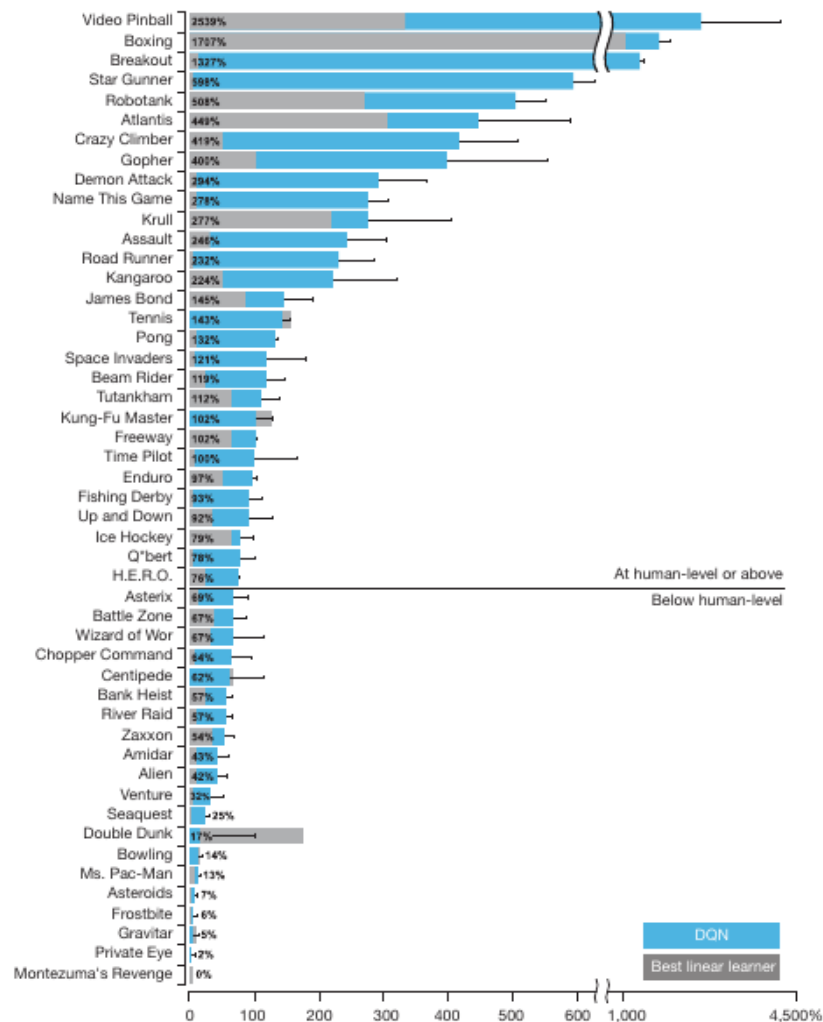


Figure B.1: Results of the DQN created by DeepMind. Image from the paper *Human-level control through deep reinforcement learning* [1]

Appendix B. Diagrams

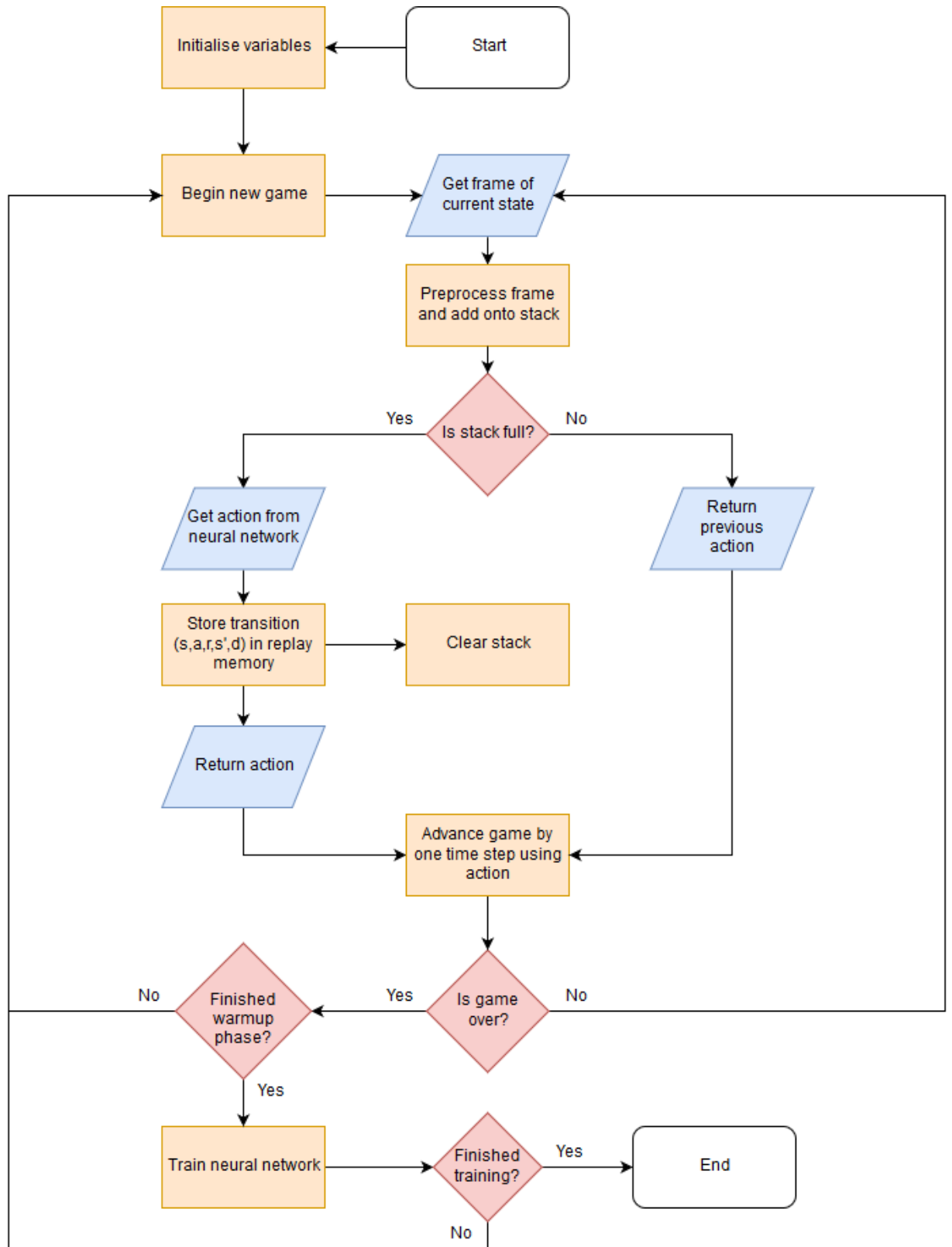


Figure B.2: Flowchart of the overall GVGAI algorithm

Appendix C

Source Code

The Python source code used in this project is available at the following GitHub repository: <https://github.com/ColinCee/GVGAI-Dissertation>

Bibliography

- [1] “Dqn,” Feb 2015, accessed: 2018-02-03. [Online]. Available: <https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf>
- [2] Z. Wang, T. Schaul, M. Hessel, H. v. Hasselt, M. Lanctot, and N. d. Freitas, “Dueling network architectures for deep reinforcement learning,” Apr 2016, accessed: 2018-02-03. [Online]. Available: <https://arxiv.org/abs/1511.06581>
- [3] “Deep blue versus garry kasparov,” Jan 2018, accessed: 2018-02-03. [Online]. Available: https://en.wikipedia.org/wiki/Deep_Blue_versus_Garry_Kasparov
- [4] C. Piech, “Deep blue,” 2012, accessed: 2018-02-03. [Online]. Available: <http://stanford.edu/~cpiech/cs221/apps/deepBlue.html>
- [5] “Deep blue faq,” Feb 2001, accessed: 2018-02-03. [Online]. Available: <https://www.research.ibm.com/deepblue/meet/html/d.3.3a.html>
- [6] A. Gabbatt, “Ibm computer watson wins jeopardy clash,” Feb 2011, accessed: 2018-03-13. [Online]. Available: <https://www.theguardian.com/technology/2011/feb/17/ibm-computer-watson-wins-jeopardy>
- [7] S. Borowiec, “Alphago seals 4-1 victory over go grandmaster lee sedol,” Mar 2016, accessed: 2018-02-03. [Online]. Available: <https://www.theguardian.com/technology/2016/mar/15/googles-alphago-seals-4-1-victory-over-grandmaster-lee-sedol>
- [8] “Go and mathematics,” Jan 2018, accessed: 2018-02-03. [Online]. Available: https://en.wikipedia.org/wiki/Go_and_mathematics
- [9] “Shannon number,” Jan 2018, accessed: 2018-02-03. [Online]. Available: https://en.wikipedia.org/wiki/Shannon_number

Bibliography

- [10] “Observable universe,” Jan 2018, accessed: 2018-02-03. [Online]. Available: https://en.wikipedia.org/wiki/Observable_universe#Matter_content
- [11] A. Karpathy, “Alphago, in context – andrej karpathy – medium,” May 2017, accessed: 2018-02-03. [Online]. Available: <https://medium.com/@karpathy/alphago-in-context-c47718cb95a5>
- [12] David, Hubert, Thomas, Schrittwieser, Julian, Ioannis, Lai, Matthew, Guez, Arthur, and et al., “Mastering chess and shogi by self-play with a general reinforcement learning algorithm,” Dec 2017, accessed: 2018-03-13. [Online]. Available: <https://arxiv.org/abs/1712.01815>
- [13] A. Karpathy, “Alphago, in context,” May 2017, accessed: 2018-02-03. [Online]. Available: <https://medium.com/@karpathy/alphago-in-context-c47718cb95a5>
- [14] J. Levine, C. B. Congdon, M. Ebner, G. Kendall, S. M. Lucas, R. Miikkulainen, T. Schaul, and T. Thompson, “General video game playing,” 2012, accessed: 2018-03-24. [Online]. Available: <http://www.idsia.ch/~tom/publications/dagstuhl-gvgp.pdf>
- [15] “Neural networks and machine learning,” Sep 2015, accessed: 2018-02-04. [Online]. Available: <https://blogs.cornell.edu/info2040/2015/09/08/neural-networks-and-machine-learning/>
- [16] “Convolutional neural network,” Feb 2018, accessed: 2018-02-06. [Online]. Available: https://en.wikipedia.org/wiki/Convolutional_neural_network#Image_recognition
- [17] M. Alzantot, “Deep reinforcement learning demystified (episode 0),” Apr 2017, accessed: 2018-02-05. [Online]. Available: <https://medium.com/@m.alzantot/deep-reinforcement-learning-demystified-episode-0-2198c05a6124>
- [18] “Markov decision process,” Feb 2018, accessed: 2018-02-05. [Online]. Available: https://en.wikipedia.org/wiki/Markov_decision_process
- [19] “Markov property,” Jan 2018, accessed: 2018-02-05. [Online]. Available: https://en.wikipedia.org/wiki/Markov_property

Bibliography

- [20] “Bellman equation,” Jan 2018, accessed: 2018-02-06. [Online]. Available: https://en.wikipedia.org/wiki/Bellman_equation
- [21] A. Juliani, “Simple reinforcement learning with tensorflow part 0: Q-learning with tables and neural networks,” Aug 2016, accessed: 2018-02-06. [Online]. Available: <https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-0-q-learning-with-tables-and-neural-network>
- [22] “Multi-armed bandit,” Feb 2018, accessed: 2018-02-23. [Online]. Available: https://en.wikipedia.org/wiki/Multi-armed_bandit
- [23] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, “Prioritized experience replay,” Feb 2016, accessed: 2018-02-03. [Online]. Available: <https://arxiv.org/abs/1511.05952>
- [24] H. v. Hasselt, A. Guez, and D. Silver, “Deep reinforcement learning with double q-learning,” Dec 2015, accessed: 2018-03-03. [Online]. Available: <https://arxiv.org/abs/1509.06461>
- [25] “The gvg-ai competition,” accessed: 2018-02-25. [Online]. Available: <http://gvgai.net/>
- [26] “Let’s make a dqn: Double learning and prioritized experience replay,” Feb 2018, accessed: 2018-03-04. [Online]. Available: <https://jaromiru.com/2016/11/07/lets-make-a-dqn-double-learning-and-prioritized-experience-replay/>
- [27] “Monte carlo tree search,” Mar 2018, accessed: 2018-03-17. [Online]. Available: https://en.wikipedia.org/wiki/Monte_Carlo_tree_search#Exploration_and_exploitation
- [28] Volodymyr, A. Puigdomènech, Mirza, Mehdi, Alex, T. P., Harley, Tim, David, Koray, and et al., “Asynchronous methods for deep reinforcement learning,” Jun 2016, accessed: 2018-03-17. [Online]. Available: <https://arxiv.org/abs/1602.01783>