THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

Deep Fighter: An RL Application to Street Fighter

By

Daniel Bairstow

Bachelor's Thesis

Submitted for the degree of Bachelor of Engineering with Honours
in the division of Electrical Engineering

submitted to

The University of Queensland

School of Information Technology and Electrical Engineering

Supervisors

Brian Lovell and Arnold Wiliem

June 9, 2019

Daniel Bairstow
daniel.bairstow@uq.net.au

June 9, 2019

Prof Michael Brünig
Head of School
School of Information Technology and Electrical Engineering
The University of Queensland
St Lucia, Q 4072

Dear Professor Strooper,

In accordance with the requirements of the degree of Bachelor of Engineering in the division of Electrical Engineering, I present the following thesis entitled "Deep Fighter: An RL Application to Street Fighter". This work was performed under the supervision of Prof. Brian Lovell and Prof. Arnold Wiliem.

I declare that the work submitted in this thesis is my own, except as acknowledged in the text and footnotes, and has not been previously submitted for a degree at The University of Queensland or any other institution.

Yours sincerely,

Daniel Bairstow.

Acknowledgments

I would like to thank both Professors Brian Lovell and Arnold Wiliem for their contribution and support throughout the project's process. Throughout the project Brian provided insight and assistance into my work as well as academics as a whole. Arnold assigned the original topic of the thesis, Reinforcement Learning. Arnold provided help in understanding the theory behind many of the new topics as well as providing research direction towards similar works I have based this project off. The many technical difficulties faced throughout the project were aided by Dr Danny Smith. Danny's help in solving Networking issues as well as setting up hardware was integral in the project's completion.

Abstract

Reinforcement Learning follows principles seen in animal psychology [15] in that it uses past experiences, through the use of a reward signal, to aid in determining optimal behaviour for a goal based problem. This subset of Machine Learning has seen a renaissance in recent years. With rapid advances in algorithms such as Deep Q-Learning [22], Reinforcement Learning has allowed the solving of problems such as human level Go play [16], originally thought to be years away from being achieved.

Such levels of performance have been aided in recent years by the use of neural networks in representing an agent to train. This project focuses on the application of Deep Q-Learning, the learning of a given action's value [18] that allows the agent to define optimal behaviour based on an environment's reward signal. This network was applied to the game of Street Fighter II: Turbo, where the character Ryu was trained in two separate agent architectures against an opponent Zangief character.

Over the course of 30,000 matches these agents displayed learned behaviour in the form of their own observable rule sets. Though not complex, these rule sets displayed perceivable thought processes behind how the agents attempted to maximize their reward. This was supported by practical results where one agent showed its capability to outperform a random agent both in terms of win rate and reward, proving the potential of learned rule sets to beat human defined rule sets.

Although no new ground was broken, this project showed the capabilities of Reinforcement Learning and provided insight into the theory and mechanics behind it. The lack of a large number of trained agents may have hampered the obtainment of more scientifically sound results, however the learned theory and academic skills gained throughout the project were worthwhile.

Summary of Notations

The following section details the meaning of the often used notation and acronyms throughout this report.

Acronyms

RL	Reinforcement Learning
DQL	Deep Q-Learning
DQN	Deep Q-Network
SD	Single Distribution
DD	Dual Distribution
TSD	Trained Single Distribution
TDD	Trained Dual Distribution
RSD	Random Single Distribution
RDD	Random Dual Distribution

Math

E[X]	Expectation of variable X
$max_a f(a)$	Select a such that function f(a) is maximized

Reinforcement Learning Elements

ϵ	Probability of taking random action in an ϵ-greedy policy
t	Discrete time step
s	State of the environment
r	Reward

π	Policy of a Reinforcement Learning agent
$\pi(a\|s)$	Probability of taking action a in state s
$v_\pi(s)$	Value of following policy from state s
$Q_\pi(s, a)$	Value of taking action in state s
θ	Network weights of a DQN
θ^-	Network weights of a DQN's target network

Contents

Acknowledgments **ii**

Abstract **iii**

Summary of Notations **iii**

List of Figures **x**

List of Tables **xi**

1 Thesis Scope **1**

1.1 Thesis Topic 1

1.2 Thesis Aim 2

1.3 Expected Challenges and Requirements 2

1.3.1 Coding a Deep Learning Model 2

1.3.2 Debugging a Complex Model 3

2 Theory **4**

2.1 Reinforcement Learning in Machine Learning 4

2.2 Reinforcement Learning in Deep Learning 7

3 Past works and applications **9**

3.1 Deepmind's Atari Deep Q-Learning 9

3.2 Deepmind's AlphaGo 10

3.3 Dota 2: A team focused strategy game 10

3.4 Gyroscope's Street Fighter II: Turbo . . . 11
3.5 Street Fighter as a Deep Q-Learning Application . . . 12
3.6 What is Street Fighter . . . 12
3.7 Street Fighter as an Environment . . . 13

4 Approach **14**
4.1 Reinforcement Learning Implementation . . . 14
4.1.1 The Agent . . . 14
4.1.2 Environment . . . 15
4.1.3 Reward Structure . . . 16
4.2 Resources . . . 17
4.2.1 Graphic Processing Units used . . . 17
4.2.2 Emulation . . . 17
4.3 Coding . . . 18
4.3.1 The Tensorforce Library . . . 18
4.3.2 EmuHawk . . . 19

5 Training **20**
5.1 Bugs and Errors . . . 20
5.1.1 State Scale on Flip . . . 21
5.1.2 Network Update Intervals . . . 21
5.1.3 Gradient Explosion . . . 21
5.2 Training Behaviour . . . 22
5.2.1 Jump Spamming . . . 22
5.2.2 Teaching Movement . . . 22
5.2.3 Attack Spam . . . 23
5.3 Final Training Architecture . . . 23

6 Results **25**
6.1 Testing Process . . . 25
6.2 Saliency . . . 27
6.3 Single Distribution Agent . . . 29

6.3.1 Reward Structure . . . 30
6.3.2 Observed Behaviour . . . 30
6.3.3 Agent Saliency . . . 32
6.3.4 Agent Statistics . . . 33
6.4 Dual Distribution Agent . . . 35
6.4.1 Reward Structure . . . 35
6.4.2 Observed Behaviour . . . 35
6.4.3 Agent Saliency . . . 36
6.4.4 Agent Statistics . . . 37
6.5 Single Distribution compared to Dual Distribution Agents . . . 38

7 Findings **40**
7.1 Agent Architecture . . . 40
7.1.1 Action Distributions . . . 40
7.1.2 Reward Structure . . . 41
7.1.3 Further Study . . . 41
7.2 Project Successes and Failures . . . 42
7.2.1 Project Aim . . . 42
7.2.2 Project Failures . . . 43
7.2.3 Academic Skills . . . 43

8 Conclusion **44**

Appendices **44**

A Agent Architecture **45**
A.1 Network Configuration . . . 45
A.2 Agent Configuration . . . 46
A.3 Agent Action Spaces . . . 48

B Results **49**

C Program Code **51**
C.1 Training program SFIIBeta.py 51
C.2 Runner.py Training Loop 52
C.3 Training A Deep Q-Network 53
C.4 Applying Saliency to Network 54

Bibliography **55**

List of Figures

2.1 Reinforcement Learning Feedback Loop 5

3.1 Replay Memory and Target Network importance 10
3.2 Street Fighter Gyroscope Agent Observations 11
3.3 Street Fighter II Player Screen 12

4.1 Character Models used in matches (left to right; Ryu, Zangief, ChunLi) . . 15
4.2 Emulation screenshots sent with different background elements removed . . 16

6.1 Chun Li match example state 27
6.2 Saliency of example Chun Li match state 27
6.3 A fresh (left) versus partly (right) trained network saliency plot 28
6.4 Total loss of agent during training 29
6.5 Histogram of actions chosen by the single distribution agent over time . . . 31
6.6 Saliency plot of SD Agent (left to right saliency A, B, C) 32
6.7 Saliency plot of DD Agent (left to right saliency A, B, C) 36
6.8 Comparison of average HP Difference 38

List of Tables

6.1 Single Distribution Agent's testing results 33
6.2 Dual Distribution Agent's testing results 37

B.1 Dual Distribution Agent's full testing results 49
B.2 Single Distribution Agent's full testing results 50

Chapter 1

Thesis Scope

Reinforced learning has seen great advances in recent years with its integration of neural networks. Regarded as some of the potential cornerstones of general artificial intelligence (AI), algorithms such as Deepmind's multi-purpose Atari deep Q-network [22] and AlphaGo's domination of professional players [3] have seen major breakthroughs in recent years. As more powerful computation and advances in the discipline continue to rise, RL is likely to play a vital role in the future of machine learning algorithms, specifically that of advancements in general AI.

This thesis covers a first attempt of exploring and applying reinforced learning in the implementation of a Street Fighter AI.

1.1 Thesis Topic

This thesis explores Deep Reinforcement Learning (DRL) through the design of a real-world execution of its principles. Reinforcement learning is a subset of machine learning, generally applied as a goal-oriented algorithm [22] used to define the preferred learned behaviour for a given task. Given the wide range of algorithms used in DRL, a choice was made given the chosen application of the thesis.

Due to the specific application and nature of Street Fighter, Deep Q-learning, a subset of DRL, was chosen as the focus of the thesis, with some other areas of RL and Deep Learning explored as the project advanced.

1.2 Thesis Aim

Generally, the aim of any Deep Learning solution to a problem is to approach human level intelligence regarding completing a task. In this case such an achievement is difficult to expect, given the complexity of Street Fighter and the high skill ceiling of human players. As such, rather than beating a human, it was hoped that the thesis would be able to learn behaviour capable of beating a computers' predefined rules designed by a human. This would prove the thesis's success in the form of a learned set of "rules" beating a human's defined set of rules, such as the coded computer controlled fighters.

1.3 Expected Challenges and Requirements

Before beginning the thesis proper, potential challenges encountered throughout the thesis were analysed so as to streamline the process of solving these challenges as they arose. The main expected challenges and the respective requirements for overcoming them are outlined here.

1.3.1 Coding a Deep Learning Model

Despite having knowledge of the theories behind RL, the execution of the project requires coding knowledge of Machine Learning libraries that I had not used before/was unfamiliar with. Finding appropriate libraries and resources to better understand these libraries was integral. Coding well-structured software for this application would be difficult otherwise, due to the lack of experience in the area, potentially harming future progress. It was likely that progress would not be consistent, with several errors in code or structure of project likely to occur at some point. In addition, using software that allows control of the model's hyperparameters and other training variables would assist in experimentation.

1.3.2 Debugging a Complex Model

In addition to the lack of experience, using third-party libraries to assist in the project was beneficial as such libraries were likely to be bug-free. This cut down on time spent debugging code and software that is completely new. Even so such errors and bugs were likely to occur and were dealt with appropriately. Debugging the complex model that this project was likely to become required proper understanding of the used library's functions as well as ample data that could be used for debugging. Ensuring both were accounted for aided in locating and fixing any source of errors.

Chapter 2

Theory

Before any practical progress could be made it was important to build a good understanding of the concepts and theory behind Reinforcement Learning and its subsequent building blocks of Machine and Deep Learning. This section covers the found theory and principles of RL, Deep Learning and DQN's, from the basics of Reinforcement Learning to its application through the use of a Deep Network.

2.1 Reinforcement Learning in Machine Learning

Similar to Unsupervised and Supervised Learning, Reinforcement Learning is a subset of Machine Learning. Supervised and Unsupervised generally consist of some kind of data analysis focusing on finding generalizations or structure in a data set. Reinforcement Learning, on the other hand, attempts to maximize a reward signal returned to an agent acting within an environment. This relationship between agent and reward forms a goal-seeking problem similar to how humans learn behaviour, though not nearly as complex [15].

Generally, a Reinforcement Learning problem can be modeled as a Markov Decision Process (MDP) problem, with the environment and agent forming the state and action space respectively, where an action in the current state results in a new state [11] and a respective reward signal. This relationship is seen in figure 2.1.

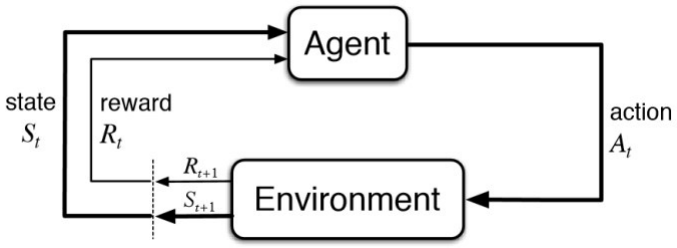

Figure 2.1: Reinforcement Learning Feedback Loop

The reward signal returned from the environment is critical in defining an agent's behaviour and its policy. An environment's reward allows the agent to differentiate between good and bad actions and as such should accurately define the goal of the given task. Even a maximized reward may not necessarily mean the task has been completed correctly if the rewards were not task-appropriate. Sparse reward systems, an agent that is given non zero rewards rarely, can similarly challenge the learning process[12].

In RL an agent's decision process is defined by its policy $\pi(a|s)$. Ideally this policy can be thought of as a mapping between any given state and the respective ideal action to maximize the reward signal. One method of maximizing a reward signal is through policy iteration, where optimal policy is found to maximize the reward. For a small action and state space this would be a simple lookup table, however, for more complex problems this quickly become unfeasible. For a given policy the reward is given as shown in equation 2.1.

$$[R_s^a = E[r_{t+1}|s_t = s, a_t = a] \tag{2.1}$$

Extrapolating on the policy gives the value function $v_\pi(s)$, a prediction of the expected, accumulative, discounted future reward [12] by following a given policy. Hence the optimal policy $\pi*$ will give the optimal value function $v_*(s)$, a maximized cumulative reward. As seen in the decomposed Bellman equation 2.2, the equation uses a discount factor $\gamma \in [0, 1]$ to control how much future rewards are taken into account. This function is recursive, with s' representing the next state to be fed into the value function until a terminal state

is reached, where the value is simply the reward.

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{2.2}$$

Similar to the value function, the action-value function $Q_\pi(s, a)$ measures the cumulative reward for taking action a in state s then following the policy. Assuming an ideal policy this allows actions to be chosen at each time step as the action with the maximum value.

$$Q_\pi(s, a) = \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} Q(s', a')] = r + \gamma \max_{a'} Q(s', a') \tag{2.3}$$

$$Q^*(s_t, a_t) = max_\pi E[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|s_t, a_t, \pi] \tag{2.4}$$

"Learning" can be applied to either the policy, value function, or action-value function. When trying to learn the Q value, what is known as Q-learning, equation 2.5 is used to apply loss.

$$L = r + \gamma max_{a'} Q(s', a') - Q(s, a) \tag{2.5}$$

however before this function may be used for for selecting the best action, the optimal policy must first be found, as shown in equation 2.4. Finding this ideal policy is exhaustive in large action-state spaces, requiring too many state-action transitions to be explored. As such the Q function can be approximated using some form of function approximation in Q-learning, a form of temporal difference learning [23] to be discussed in the next section.

2.2 Reinforcement Learning in Deep Learning

Function approximation is used when state-action spaces are too large [12]. In linear function approximation, a function can be approximated by a set of weights applied to each input (state) component of the function. However, for complex nonlinear functions, nonlinear function approximation is needed. Deep Neural Networks (DNN) allow this.

A DNN consists of several layers with each layer feeding into the next. Each layer is made up of a number of weights applied to its input in tandem with an activation function applied to its output [14]. The use of the activation function gives the required nonlinear behaviour. The learning component of DNNs is the finding of weights for the NN to exhibit desired behaviour. In the case of function approximation the NN's weights are being updated to minimize the loss seen in equation 2.5.

Using this method, the Q function can be approximated to Q* function independent of policy [18] using what is called a Deep Q-Network (DQN) with weights defined by θ, an extension of Q-Learning. If through learning the Q function can be accurately generalized, the agent may choose actions at each time step by following the Q function rather than a defined policy.

In large action-state spaces DQL forms an exploration versus exploitation problem [19] as generalization requires the exploration of many new state-action pairs, while exploiting its learned behaviour. This is achieved in practice using a predefined policy ϵ-greedy, where a random action is chosen irregardless of Q with a probability of $\epsilon \in [0, 1]$ [20], as seen in equation 2.6. Generally this value begins high during early training to explore a suitable range of state and action space, and is lowered as the Q functions converges on the optimal Q*.

$$\pi(s, \epsilon) = \begin{array}{c} \text{random action, } U[0,1] < \epsilon \\ max_a Q(s, a), \text{ otherwise} \end{array} \tag{2.6}$$

Previously, applying Deep Q-Learning by itself was problematic, learning often being unstable and the approximator diverging when using nonlinear approximation [21]. However recent years have seen breakthroughs in solving this problem through the introduction of several features to how a DQN learns. This project focuses mainly on the architecture introduced by Deepmind's Atari Agent [22]. This architecture introduced

what is known as replay memory and a target network.

Rather than storing then feeding a batch at time step intervals into the model to apply optimization, replay memory works by storing a tuple containing the agent's observations at every time step, the state, action, reward and resulting state at a given time step (s_t, a_t, r_t, s_{t+1}). When model updates are made a batch is randomly sampled from replay memory. This sampled batch of tuples (s, a, r, s') contains all the information the model requires to perform optimization on a given time step, as equation 2.7 shows. The use of replay memory removes correlation of samples used in training and smooths over changes in the sample distribution [22].

The Target Network of a DQN acts as the "target" of the DQNs updates, with θ^- acting as the network weights. At intervals of τ time steps, the Target Network's parameters are updated to the DQNs base network's weights θ. By doing this, the target used in updates in equation 2.7 is stabilized and not changing constantly as the network updates [8] increasing the learner's stability.

$$L(\theta) = (r + \gamma max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta))^2 \tag{2.7}$$

Recent years have seen these features in DQNs built upon with new techniques such as prioritized replay, Double Q-Learning, Long Short Term Memory and other techniques [9]. Whilst by comparison DQNs are far less efficient and have poorer performance, only a basic DQN with Experience Replay and a Target Network were used in this project.

Chapter 3

Past works and applications

Reinforcement learning has a wide array of applications. Typically, applications which can easily fit RL's structure of a reward function and environment to act are the easiest tasks to adapt RL to. Games and Robotics are two of the easiest problems RL can be applied to as they both typically represent the required structure. Games have been used in recent years as a test bed for RL techniques [12], with many recent advances being made by numerous applications in Atari [9][22] and the board game Go [2][16]. This section highlights the performance and importance of RL as a Machine Learning application.

3.1 Deepmind's Atari Deep Q-Learning

As mentioned, Deepmind's Atari player provided the features of DQNs this project initially planned to use. In its own application, Deepmind displayed the usefulness of these features, performing at or above human-level in 29 of 49 different games [22]. Deepmind's model utilized an emulator screen shot fed into a convolutional stack followed by a fully connected layer across all 49 games. In addition to this performance, the application showed the importance of Replay Memory and a Target Network as shown in figure 3.1.

Game	With replay, with target Q	With replay, without target Q	Without replay, with target Q	Without replay, without target Q
Breakout	316.8	240.7	10.2	3.2
Enduro	1006.3	831.4	141.9	29.1
River Raid	7446.6	4102.8	2867.7	1453.0
Seaquest	2894.4	822.6	1003.0	275.8
Space Invaders	1088.9	826.3	373.2	302.0

Figure 3.1: Replay Memory and Target Network importance

3.2 Deepmind's AlphaGo

Deepmind's most recent network, AlphaGo Zero, utilises Policy and Value function approximation [3]. AlphaGo shows RL's usefulness in extremely large state, action spaces, Go having a state space of around 10^{174} as compared to Chess's 10^{120}. Deepmind designed their most recent network, AlphaGo Zero, which utilized a Policy trained to imitate Monte Carlo Tree Search [2], allowing it to effectively explore the state space without the heavy computation of a tree search.

Combining this Policy method and a Value function predicting the winner, AlphaGo Zero was capable of beating the previous version, Alpha Go Master, which defeated the strongest human professional players 60-0 in 2017 [2]. The first computer program to beat a human professional player was AlphaGo in 2015 [16], showing just how quickly improvements in the field have been made.

3.3 Dota 2: A team focused strategy game

Whilst these works have not seen long term improvements due to their recent application, they have showed promise into how RL can be applied to complex tasks made up of many sub tasks. Dota 2 is a Multiplayer Online Battle Arena involving five players working together to destroy the enemy's base. Dota's complexity not only comes from its game mechanics but also from its use of five separate players acting within the same environment.

The OpenAI Five team utilized five separate neural networks, each acting as a different player on the same team, with no explicit communication between them other than

environmental observations. Despite the complexity and the use of separate agents, the team beat the previous world champion team 2-0 [13]. For a complex team-based game which requires real tactics this was a substantial accomplishment, especially the display of key strategies professionals would use despite learning from scratch.

3.4 Gyroscope's Street Fighter II: Turbo

As part of the Samsung Developer Conference, Gyroscope, a machine learning and data science software company [7], constructed a Deep Q-Network capable of playing the fighting game Street Fighter II: Turbo [4]. Gyroscope modelled the network's architecture by reading RAM values of observations a human would make, as seen in figure 3.2. The available actions were chosen as a combination of two button inputs, with the network being left to discover the more obscure moves such as action combos for special attacks.

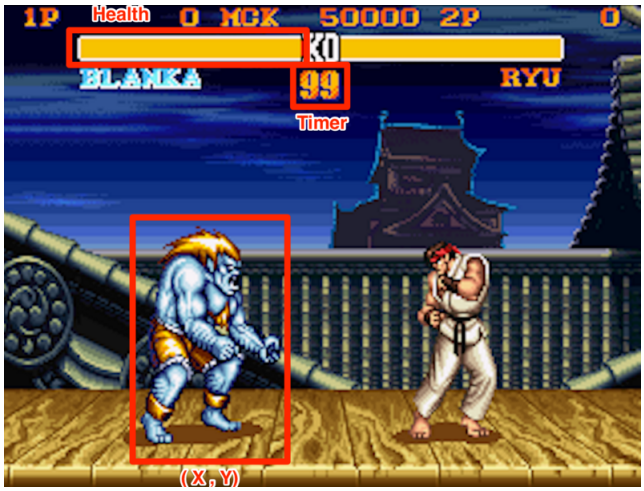

Figure 3.2: Street Fighter Gyroscope Agent Observations

Several tricks were applied to increase training efficiency. Speed running tools were used to increase the frame rate the agent could play at, decreasing necessary training time for an agent. The reward function was applied by reading the health values of opponents from RAM and calculating the change in health between each time step. Each AI was trained for 8 hours, finishing with an 80% win rate against the hardest difficulty computer

opponents, displaying character-specific strategy and use of special moves. Whilst not ground breaking, this application shows the usefulness of DQNs in other types of games and acts as a baseline for the project's own application.

3.5 Street Fighter as a Deep Q-Learning Application

Before DQL can be applied to an application, a framework is required to fit into the structure of a Deep Q-Network and its various components: Environment, Agent, state-action spaces, reward function etc. This section describes the initial approach to representing Street Fighter as an RL Environment as well as how the Network was implemented.

3.6 What is Street Fighter

Street Fighter is a series of 2-D fighting games first released in 1987 that has seen mass popularity with over 41 million sales worldwide [1]. In Street Fighter opponents utilize a combination of attacking, blocking and movement to attempt to reduce the opposing player's health to zero before their own does, as seen in figure 3.3. With six different attack and eight directional commands, there are 48 valid button combinations, not including special moves that require a specific sequence of player commands.

Figure 3.3: Street Fighter II Player Screen

3.7 Street Fighter as an Environment

Part of the reason Street Fighter was chosen was its simple agent-environment architecture it can form in a DQN. The agent, one of the two onscreen characters, carries out different character moves (actions) with the aim of beating the other player acting, the reward.

As human control normally consists of two simultaneous actions, e.g. jumping and kicking at the same time, for the 14 possible commands there are 32,768 (2^15) possible actions that could form the action space, including no character input as a command. This action space can realistically be cut down to 63 distinct actions. The state space of the environment, all possible pixel configurations the game can take, is extremely large for even a small screen. Street Fighter has a native resolution of 256 by 224 RGB pixels, meaning there are 172,032 input values. This can be cut down slightly by down sampling screen shots of the emulation.

In addition to the large action and state space, the cast of 12 characters, each with generally distinct move sets, introduced some prospective variety into the potential learned techniques of the agent.

Chapter 4

Approach

This section outlines the approach taken to applying DQL to the Street Fighter game, including both its structure in RL as well as the required resources and code for implementation.

4.1 Reinforcement Learning Implementation

To apply Reinforcement Learning to a problem, it first has to be adapted to an appropriate format to represent both the network being trained on and the problem as an Environment and Agent relationship. This section covers how the project was modelled to form this relationship.

4.1.1 The Agent

The Agent was chosen to be Player 1 with the fighter Ryu chosen as its character to play. Ryu was chosen as we were most familiar with his character in the series and due to a pretty straightforward strategy built into his move set. Ryu utilizes a range of long-distance attacks to keep his opponent at a safe distance, combined with strong throws when opponents get too close. Ryu also has a distinctive character model compared to his opponents during training, as seen in figure 4.1, which should improve visualization of the Environment.

The Agent's action space, set of available actions, was changed throughout the project as new observations on the agent's performance were made. Generally the agent's network utilized a last additional layer to get the number of network outputs to match the number of actions in the action space. This meant that any agent with a different action space had to restart training.

Figure 4.1: Character Models used in matches (left to right; Ryu, Zangief, ChunLi)

4.1.2 Environment

As the environment is defined as the space within the agent is acting, the environment is the emulation of a Street Fighter match. Matches were played with a 90 second time limit, with the agent giving a new action to perform every 4 frames. Matches were played at 30fps, giving 7.5 actions per second of real time play. The training opponent was a computer controlled character (CPU) Zangief of varying degrees of difficulty pulled from the games files. Training against a CPU allowed training speed to be greatly increased as the agent could play at above human speeds.

Every 4 frames, the environment would update the agent by sending both a screenshot of the game, as seen in figure 4.2, as well as a set of RAM values of the emulation to be used in the reward calculation. As figure 4.2 shows, the screenshot could be represented with varying degrees of "noisy" information, elements of the screenshot not important for learning. As such throughout training different representations of the environment were used to try to speed up training. Another technique used to speed up training was to flip the match half the time, allowing the agent to experience matches where it views itself

on the right side of the stage when starting.

Ryu's training opponent was chosen as Zangief due to their obviously different character colour schemes and size as seen in figure 4.1.

Figure 4.2: Emulation screenshots sent with different background elements removed

4.1.3 Reward Structure

Reward structure, as mentioned in section 2.1, is vital in defining the agent's optimal behaviour to achieve a goal, in this case winning. In an ideal case the reward would be given dependent on whether an agent wins or loses. However, due to matches taking at least 20 seconds to finish (150 time steps), learning the right state-action transitions to reach a terminal state without some kind of tree search similar to AlphaGo's [16] would be far too complex to learn.

As such, rewards aimed to describe winning behaviour, rather than indicating a win or loss. For this reason rewards were non-sparse, rewarding good and bad behaviour as frequently as possible. The characteristics of what to reward/not reward were defined throughout the project, though typically rewards were based around elements of the game, e.g. blocking, moving, gaining HP advantage, hitting attacks etc.

4.2 Resources

Two main resources were required to implement the Deep Q-Network to be trained. The DQN required both an agent and environment to work with, the agent containing the network and the environment running a game of Street Fighter. The network was built using Graphic Processing Units (GPUs) supplied by UQ connected to Ubuntu machines, whilst the speed running tool EmuHawk was used to emulate the game on a Windows computer.

4.2.1 Graphic Processing Units used

At the project's beginning, two Nvidia Quadro P620 GPUs were available, although their 2GB of memory subjected training to some limitations. Due to their small size, the size of the network as well as the replay memory were limited, as was the computation speed of deciding the action at each time step. During the last two months of the project, two larger GPUs were made available with 6 and 12GB of memory, increasing training speed by 60% and allowing more flexibility with memory usage. This allowed two networks to be trained at the same time, allowing some experimentation of DQL's hyperparameters and reward structure.

4.2.2 Emulation

After considering Gyroscope's own DQL application, EmuHawk [17] was chosen to play Street Fighter on for several reasons. EmuHawk allowed:

- Reading of RAM values
- Sending/Receiving local host socket packets
- Speeding up frames per second
- Open source code for altering emulation loop
- Saving/Loading of states of emulation allowing matches to reset

These features allowed the agent to be run on a Linux machine whilst communicating with the Emulator on a Windows machine. Using this setup games could be played at over 30 fps, allowing faster training.

4.3 Coding

The coding requirements of this project was vital as it involved the explicit application of DQL to the problem. Due to the intimidating nature of the problem when starting out, the reinforcement learning library Tensorforce [10] was selected due to its ease of use. The program language python was chosen due to its ease of use and documentation in tensorflow.

For a full overview of how code was applied see Appendix C, where pseudo code of each major program used is shown.

4.3.1 The Tensorforce Library

The Tensorforce library is a reinforcement learning library built using tensorflow capable of applying several different RL algorithms to a given problem. The library's use of agent and network configuration files allows hyperparameters of the agent, such as batch size and learning rate, as well as the architecture of the network used to easily be changed. Tensorforce has three component classes it uses in training: an agent (which contains the network model), an environment (which handles state/reward fetching and action execution) and a runner which handles the training loop.

Street Fighter specific environment and runner classes were coded by hand to deal with communicating with the EmuHawk emulation. The runner's loop generally consists of:

1. Pass state through agent to get action
2. Execute action
3. Retrieve resulting state and reward
4. Store result in batch

As well as the runner and environment, a new agent class was built on the default Deep Q-Agent of the library to allow the agent's code to be changed. This allowed several checks and print statements to be implemented to more easily observe agent behaviour at its most base level, including the agent's Q-values which were needed for other functionality such as state Saliency.

4.3.2 EmuHawk

As mentioned in section 4.2.2, EmuHawk is an open source emulation library written in C#. Although lua functionality allowed the learning loop to run there were drastic losses in speed waiting for the lua script to execute once EmuHawk received an action to execute, causing a bottleneck in training speed. As such the source code of EmuHawk was altered to automatically run the learning loop, communicating with the agent through sockets set up at the program's start.

Chapter 5

Training

Over the course of the project there were many "jump starts", as new bugs and errors coming to light could cause the previously trained models to be in essence useless. Due to this as well as problems in weight initialization, only two fully trained models were completed. Though this number is not quite as high as expected when beginning, as part of the goal was to experiment with RL by altering training conditions, the two models both gave different insights into RL and its application.

5.1 Bugs and Errors

Byte Overflow in Emulator RAM

During training, rewards were calculated using RAM values read directly from the Emulator's core. As these values were read as bytes during the final hit when player's health dropped to 0, the read RAM value would actually "underflow" to 255 rather than 0. This caused the final reward, calculated using HP difference, to be very large and of the opposite signage than what would have been rewarded. This was fixed simply using a check in the agent's side of the code.

5.1.1 State Scale on Flip

Every second state the screenshot of the emulation was flipped. Typically the state is represented as values from 0 to 1 by scaling pixels from 0-255 to 0-1, however this operation was not performed when the state was flipped, causing the agent's network to try to work with two different inputs on vastly different scales. This meant any loss calculations were completely different from one episode to another. This was fixed by scaling down to 0-1 in both cases.

5.1.2 Network Update Intervals

During training, samples were read into a batch that once full were stored into memory. If the current time step was then divisible by the frequency parameter, an update was performed by sampling a batch. The agent configuration file was an altered version of an example from Tensorforce, where a frequency of four, batch capacity of 1000 and batch size of 31 were used.

The problem arose when it was realized that essentially, for every 1000 samples there was a 1 in 4 chance that 31 samples were used to train the network: this essentially meant only 0.775% of samples were used in training, a minuscule number. This problem, once noticed, was fixed by using a smaller batch capacity and a frequency of 1, updating every time a batch was stored in memory.

5.1.3 Gradient Explosion

Gradient explosions were the most prominent and bothersome of encountered errors mostly due to there being no real solution, as well as the randomness to their occurrence. Gradient explosions majorly hamstrung training before the larger GPUs were used for training, as training on the smaller GPUs could explode days into training whereas the larger GPUs only took a matter of minutes. Using the larger GPUs was vital in re-initializing weights enough times to finally get an initialization that worked. Over the course of the last two weeks of testing on the larger GPUs, only a single agent was fully trained without this error.

5.2 Training Behaviour

5.2.1 Jump Spamming

When beginning training, a dual distribution (DD) agent was used with each distribution covering movement or attack actions. The reward function at this stage gave a scalar reward for how the difference in health (HP) between players has changed since the last time step. After training for an extended period, Ryu was spamming jumps and kicking in mid air. At the time, this was seen as poor spammy behaviour as there was little strategy, hence jumping was removed from the action space.

5.2.2 Teaching Movement

Teaching movement was one of the difficulties in crafting the reward function. After using HP difference as a reward, Ryu showed little inclination towards moving other than in randomly performing his special moves. As such a small reward was given for moving, dependent on whether Ryu moved towards or away from Zangief. Initially, the magnitude of the reward for correct movement was twice that of the wrong movement, however this caused some problems where Ryu could abuse the reward system.

The first problem was found where Ryu would constantly take the Left movement action in both flipped and non-flipped states. Although understanding the core reason behind how an agent acts is difficult, a hypothesis can be made. It was suspected that due to positive rewards being twice that of negative ones, if Ryu constantly moves in one direction his net reward over a flipped and non-flipped episode will always be positive, hence seeing it as good behaviour. This was fixed by making negative rewards twice as large as positive movement rewards.

The second problem occurred when Ryu would trick the reward function to stop giving movement rewards if Ryu held down an attack action. Essentially, part of the calculating movement rewards checked whether Ryu was only inputting a movement action. However, once an attack action has been executed in game, holding it down has no further effect until that action is no longer used. As such, once an attack action was executed, Ryu could move freely until he let go of that action, circumventing the movement reward.

Both of these abuses of the reward functions shows the agent's reliance on the reward function for determining behaviour, regardless of whether this behaviour is actually considered to be a winning strategy.

5.2.3 Attack Spam

The problem of attack spam came into play when the learned behaviour was to constantly spam long-reaching attacks with little movement. This was problematic as the larger HP different rewards outweighed utilising other game mechanics, such as moving and blocking, which gave much smaller rewards. The idea was to implement a spam reward, negatively rewarding if a hit did not do damage. Not only did this behaviour cause attacks to not be spammed with no chance of them hitting Zangief, but it also allowed other actions, such as moving, to be explored as they were more viable when Zangief wasn't close.

5.3 Final Training Architecture

Agent training took place over 30,000 episodes using a training plan against varying levels of AI difficulty. ϵ-greedy was linearly annealed from 99.9% to 20% randomness over 4 million time steps, after which it was set at 20% for the remainder of training.

Training was separated into three 10,000 episode parts, where the agent Ryu's opponent's difficulty was increased from level 3 to level 5 then to level 8 respectively, 8 being the highest difficulty. The reason this was done was to ensure Ryu was capable of learning data in his initial exploration phase, where his actions are chosen with high randomness. Starting Ryu against a level 8 AI would have harmed any learning, as level 8 Zangief cheats by reading player input. This is similar to AlphaStar where beating the hardest difficult AI required the exploitation of the AI by the player [24], something the agent is unlikely to discover without some additional features.

After fixing the update intervals error described in section 5.1.2, a batch capacity of 100 was chosen with a batch size of 50 to accommodate for GPU memory limitations, ensuring a sample efficiency of 50%.

The Network architecture was a straightforward convolutional stack followed by a 256 size dense layer and a final linear layer to reduce output size to match the action

space. Each layer before the final one utilized a rectified linear unit as its activation function. This architecture was based off Deepmind's original Atari network [22]. The full configurations can be seen in Appendix A.

The states being fed into the network from the environment were altered to reduce the noisy background in the game. This is seen as the rightmost screenshot in figure 4.2. Although this did remove the health bar from the screenshot, the fenced area made training difficult as Ryu would always start in this area.

Two agents were trained, a dual and a single distribution agent. The dual distribution agent's action space covered all possible Street Fighter moves, with a distribution for movement and attack actions respectively. The single distribution agent on the other hand utilized a hand crafted action space built over the project's course. This action space was crafted to cut down on the search space for the agent while leaving moves Ryu would be more likely to use, such as those a good Ryu player would use. The action spaces of each respective agent are shown in Appendix A.

Chapter 6

Results

In the end, due to the presence of gradient explosions in all new agents, as well as previous errors causing previous models to be invalid, only two fully trained agents were finished. These two final agents were a Single (SD) and Dual (DD) Distribution network, both using slightly different hyperparameters and reward functions. This section compares the performance and behaviour of these agents.

6.1 Testing Process

For testing purposes both agents as well as a completely random version of both agents were run through a testing set of characters. Due to both trained agents and the CPU opponents being deterministic, having no randomness in action selection, there were only a few possible game outcomes.

Essentially when the CPU starts a match it chooses a specific starting move, from there both opponents act deterministically resulting in always the same outcome for that starting move. As such for each opponent the trained agent only had a number of preset outcomes, meaning win percentage was limited in its accuracy. Random agents were played for 100 episodes against each testing opponent as they did not have this problem.

Both trained agents were tested for enough episodes to make the deterministic results obvious in each CPU. The testing CPU's used where level 3, 6 and 8 Zangief, and a level 3 Chun Li. Chun Li was chosen as she shared a similar black background as the Zangief

match as well as Chun Li's very different character model compared to Ryu as seen in figure 6.1.

Figure 6.1: Chun Li match example state

Chun Li also offers some insight into the network's attention to features. Despite both having black backgrounds, Chun Li's stage has background bicycle riders, introducing new features that could interfere with the agent's learned strategy, as shown in figure 6.2 where attention is given to the background cyclist.

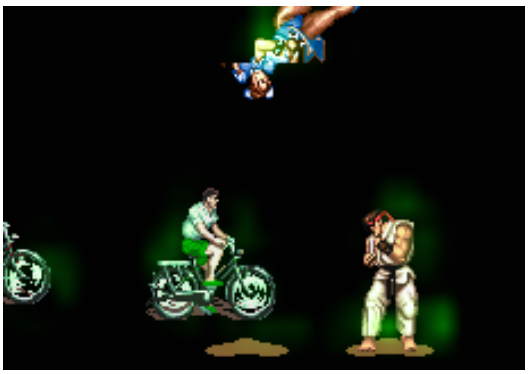

Figure 6.2: Saliency of example Chun Li match state

6.2 Saliency

Whilst an RL agent is effective in solving problems there is normally very little inherent knowledge in the agent's network of what strategy is used in solving the problem. However, using what is known as a saliency map, a colour map can be used to show what regions of

the network's input are most important in determining its output, essentially the network's "attention".

Saliency essentially maps different regions of the input to how much the output changes if these regions were altered [6]. This was achieved by adapting the code provided in the Visualizing and Understanding Atari Agents paper [5]. A pseudo code explanation of the adapted code is provided in Appendix C.

Given a state from the environment, the original output of the network can be found. This original is then compared to the network's output after changing different regions of the input using a Gaussian blur. This produces a mapping of how much the output changes in different regions, allowing a colour mapping to be applied to the original input. This colour mapping shows what the network pays attention to as seen in figure 6.3.

Figure 6.3: A fresh (left) versus partly (right) trained network saliency plot

A Ryu versus Guile match was used as it had a non-black background, hence a fresh agent would have no saliency where the input is black as blurring black does nothing. In the left image the fresh agent, trained for 0 time steps, has high saliency throughout the entire background. This is due to the initialized weights having giving no attention to a single feature in particular. Comparing this to the partly trained agent, saliency is exclusively high around the two character models.

This saliency plot shows the effects of training as the network has learned the features of the state that it needs to pay attention to. Using such saliency plots on the trained agents, some insight into how complex the network's understanding of the game state is can be gained.

6.3 Single Distribution Agent

Training of the single distribution agent produced training loss detailed in figure 6.4. As the figure shows, there is a large spike at the beginning of training which settles down to lower levels. While the loss doesn't converge to 0 as would be seen in some typical Machine Learning applications, this is likely due to the large rewards returned by the agent causing large losses when the Q-value is wrong.

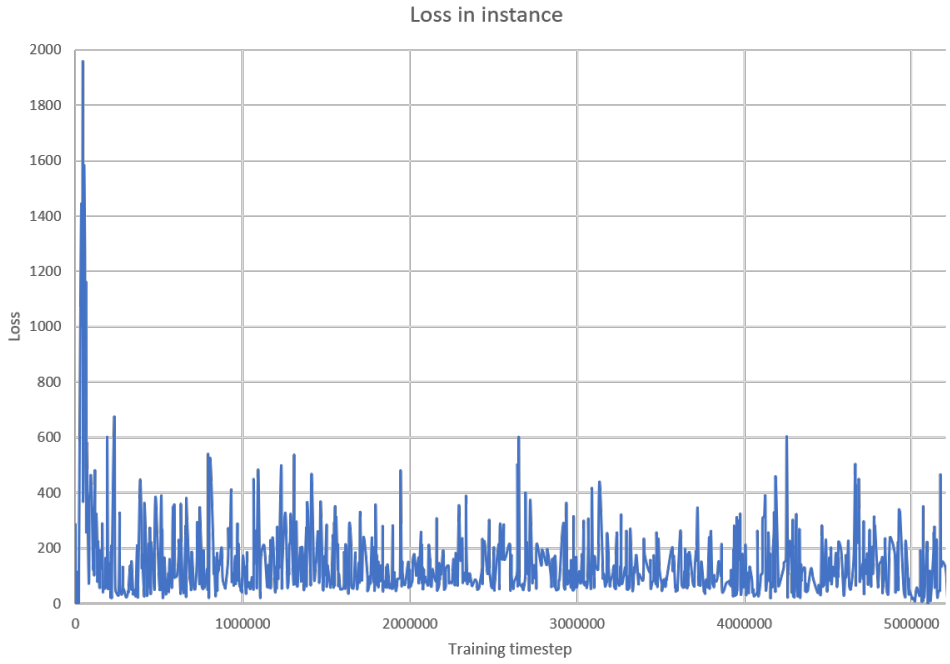

Figure 6.4: Total loss of agent during training

6.3.1 Reward Structure

The trained single distribution (TSD) agent used scalar rewards as calculated as the sum of the following conditions:

1. Change in HP gap: $2(HPgap_t - HPgap_{t+1})$
2. Moving towards opponent: 4
3. Moving away from opponent: -5
4. Missing attack: -1.6
5. Game won: +5
6. Game lost: -5

This reward structure focused on allowing the agent to differentiate between good and better rewards by using rewards of varying scale dependant on the action, which was provided by the HP gap reward. Movement rewards attempted to teach the network character positions while a spamming reward was added to reduce the likelihood of over estimating attack action's Q-value, allowing the agent to explore non attack actions.

6.3.2 Observed Behaviour

Of the two agents the single distribution (SD) network showed a much more obvious strategy. Similar to a human designed AI, the agent seemingly acted using a subset of rules detailed below. These rules are ordered in the apparent order the agent executed them.

1. If Ryu not in centre of the screen move towards the centre.
2. If at centre of the screen crouch down.
3. If Zangief walks within range perform a low kick.
4. If Zangief attempts to do an attack from the air perform a high kick, knocking him out of the air.

5. If Zangief gets close perform a grab/throw attack.

6. If Zangief attacks attempt to block.

This strategy can be seen in a histogram of the agent's action distribution shown in figure 6.5. The numbers of the x-axis are the agent chosen actions defined by its action space in Appendix A. As the figure shows, the actions 0 (No action, 1 (Down), 2 (Left), 3 (Right) and 4 (Down,Right) are predominately chosen from the movement actions. From the attack actions the most predominant throughout training is action 11 (high kick), 12 (low kick) and 14 (throw). These are the same actions described in the above behaviour.

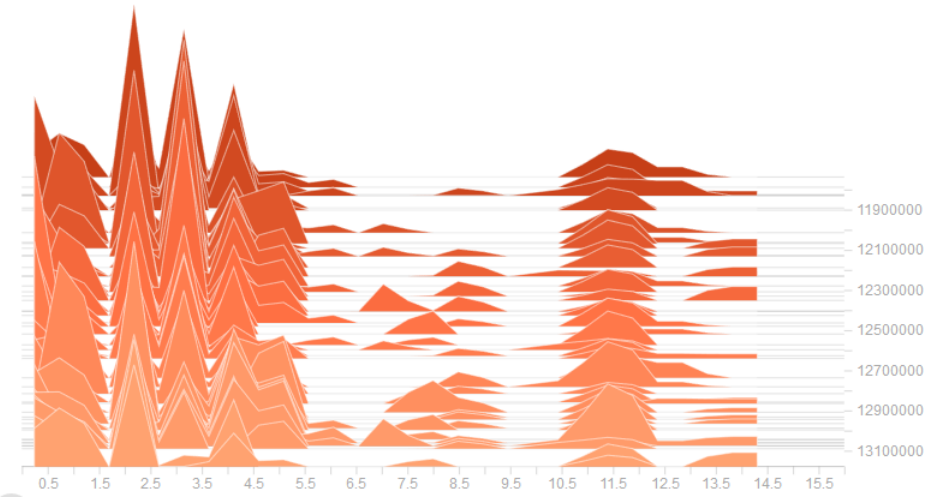

Figure 6.5: Histogram of actions chosen by the single distribution agent over time

This strategy actually utilized the technique Ryu is designed for: "Zoning" opponents by using his long reaching attacks and throws to keep them at a distance. In addition, the agent was capable of moving long distances towards his opponent, though his movement appeared to focus more on keeping Ryu at the centre of the screen rather than near Zangief. When Zangief approached, Ryu would continuously move left then right, appearing to be expecting an attack he could block.

Observing the network's Q values showed values rising and falling with each time step within a general range of -40 to 100. However, the lack of any elements in the state to

give the agent an idea of how long the match has been going for, e.g. health bar or timer, means that the Q-value does not go down as the game nears its end as would be expected in an ideal learned Q-function.

6.3.3 Agent Saliency

Figure 6.6 shows the saliency plots of three separate environment states. State A is the initial starting state of the environment. The plot focuses predominantly on Ryu's shoulder, perhaps as Zangief is too far away to consider moving or attacking.

Plot B shows the network's seeming reliance on the fire extinguisher. It is hypothesized that Ryu was using the fire extinguisher as a frame of reference to stay in the centre of the screen, as he was not necessarily paying attention to Zangief's position in how he moved.

Plot C shows clear attention on Zangief's hand mid-attack. As this attack is one Zangief makes often, it appears the network could be recognizing the hand sticking out from his body as being a "tell" for the attack, allowing Ryu to block as he is shown doing.

Figure 6.6: Saliency plot of SD Agent (left to right saliency A, B, C)

6.3.4 Agent Statistics

Running the TSD agent through our test matches gave the agent's final quantitative performance, as seen in table 6.1. The most important feature of a successfully trained agent is that it performs better than its random version. This would imply that the agent's learned strategy is better than a completely random one, as any untrained agent could use a random strategy.

	Trained Agent			**Random Agent**		
Opponent	**HP diff**	**Reward**	**Win rate**	**HP diff**	**Reward**	**Win rate**
Zangief3	172.25	396.25	100%	68.5	-259.2	87%
Zangief6	48	92.4	80%	-107	-444.5	3%
Zangief8	-95	-195.8	0%	-145.1	-486.8	0%
ChunLi3	-49	-160.7	0%	17	-553.4	54%

Table 6.1: Single Distribution Agent's testing results

In the trained agent, while the agent fails at winning any matches against Zangief8, the agent has very good performance across the board against the lower-level Zangiefs, beating the random single distribution (RSD) agent in all cases. In both 3rd and 6th level matches the cumulative reward is positive in all but one case, where Ryu loses, as seen in Appendix B. This trend is seen in the remaining losing matches, where a negative reward is always given.

From this trend we can demonstrate the appropriateness of the Environment's reward function. That is, a greater reward implies that Ryu wins by a greater margin, as shown by the final HP difference of the players. However, Ryu winning does not imply a greater reward, as shown by the Random Agent's results where Ryu won on average by 68.5 HP against a 3rd level Zangief, but still had a very negative reward.

Whilst the agent performs well against the lower level opponents, Zangief8 beats the agent in every match. Ryu's strategy described in section 6.3.2 simply can not be applied to beat an 8th level Zangief, even after 10,000 episodes of training against him. Though this result appears negative it is not unwarranted.

Street Fighter's built-in CPUs use a set of rules to decide an action by reading RAM values of the game's current state, similar to our own agent. However, at higher levels

Zangief is capable of reading RAM and selecting an action instantaneously. This creates a common problem in Human versus AI games, where the AI cheats to create artificial difficulty. AlphaStar saw similar difficulties against harder level AIs as they could read human inputs to "cheat" [24].

An 8th level opponent may in fact have impeded training. As Ryu could not find an appropriate strategy to win, we risked destroying the previous winning strategy found in the lower level opponents. This results in a kind of "overfitting" to an unfair environment such as an 8th level Zangief.

The Chun Li opponent also saw a similar losing streak in the TSD Ryu, though perhaps for different reasons. As figure 6.2 shows, Chun Li's stage has more interference of background noise, where information the agent doesn't actually need is still displayed, such as the cyclists. This noise likely interfered with the network's feature extraction, resulting in a different strategy to the one Ryu used against Zangief.

This is a form of overfitting to Zangief's states, meaning the agent is not generalized for all of Street Fighter. To apply the agent to more than one opponent, training should likely be revised to alternate between different characters, so as to learn the important features in any state.

To note is the much greater reward in the trained agent compared to the random agent in ChunLi3, despite the random one winning far more games. This is likely caused by spamming actually being beneficial in the random agent, despite causing a heavy reward penalty. It is likely that Chun Li is a more difficult opponent for Ryu, as both random agents had a lower win rate compared to the 3rd level Zangief.

6.4 Dual Distribution Agent

6.4.1 Reward Structure

The trained dual distribution (TDD) agent differed from the SD agent, where rather than a scalar reward, all non-zero rewards were scaled to ± 1 values. The following conditions were used to decide the agent's reward, with the conditions detailed below:

1. Change in HP gap: ± 1
2. Moving towards/away from opponent: ± 1
3. If current reward is 0: -0.1
4. Game won: +5
5. Game lost: -5

The aim of the clipped rewards was to test agent behaviour when multiple good actions are measured equally. Of note is the lack of a spam reward, which was instead replaced with a "do nothing" reward, incentivizing the agent to be proactive in beating the other player.

6.4.2 Observed Behaviour

Observed behaviour was harder to characterize as a rule set compared to the single distribution's agent. This was due to the dual distribution (DD) having more randomness due to the final action being decided from two different set of actions.

The DD agent still closed the distance between players before attacking with his favorite low kick, however there were some oddities in the agent's behaviour once the distance was closed. Rather than attacking the moment Zangief was in range, Ryu would often use attacks when Zangief was not nearly close enough, or would use the wrong attacks when he was in range.

This could imply that the lack of a spamming reward caused the actions to not be differentiated, with the agent not favoring the attacks that were more likely to hit.

6.4.3 Agent Saliency

Saliency was slightly different for the DD as there were two sets of network outputs. As such green and blue colours were used to show the saliency of movement and attack actions respectively.

State A of figure 6.7 shows attention on Zangief for attack actions, with Ryu's feet having attention for a mix of both attack and movement actions. It is likely that Ryu's feet could be used as a tell of what Ryu's current action is, in this state standing still. Zangief's attention, on the other hand, implies Ryu's decision process regarding whether to use a specific attack or not, as in this starting position he would be too far away to hit with an attack.

State B shows heavy movement saliency on Zangief's body as he moves towards Ryu. As Ryu moving relative to Zangief's position has already been observed, this likely shows the network's attention to how Zangief moves. In this state Ryu is executing a spinning kick special move, which causes him to be stuck in this attack until it finishes. Observing the highlighted foot of Ryu and the past saliency plots, it seems that the network focuses on appendages and other parts of the character model that stick out from the character's body.

State C again shows high saliency on character appendages. Zangief when moving has high saliency, although in this case he has some attack saliency as well as movement. This possibly shows how much more attention is paid to attack actions when Zangief is closer to Ryu, compared to state B where Zangief only has movement saliency.

Figure 6.7: Saliency plot of DD Agent (left to right saliency A, B, C)

6.4.4 Agent Statistics

Running the TDD agent through our test matches gave the agent's final quantitative performance, as seen in table 6.2. The two most important results of the TDD agent compared to the TSD agent is the difference in reward trend and the different win rates against the test CPUs. Observing the rewards in the trained agent, the agent's reward

	Trained Agent			**Random Agent**		
Opponent	**HP diff**	**Reward**	**Win rate**	**HP diff**	**Reward**	**Win rate**
Zangief3	6.25	-7	50%	88.1	-29.1	94%
Zangief6	-47.5	7	0%	-59.9	-34.8	16%
Zangief8	-144	17	0%	-120.9	-32.5	0%
ChunLi3	36.6	-36.4	60%	37	-54.2	63%

Table 6.2: Dual Distribution Agent's testing results

does not follow the trend of a larger reward always representing a win. This is evident when comparing Zangief3 and Zangief8, where whilst the agent never beats Zangief8, its reward is significantly larger than Zangief3's, despite having a higher win rate. A similar relationship is seen in the trained agent's other matches.

This implies that the reward structure of clipping rewards to ± 1 causes the important reward in the SD agent (changes in HP difference) to be treated equally as other rewards. This means the reward function likely no longer prescribes winning behaviour as well as the SD agent's did. For example multiple small, fast attacks could be seen as being better as they give $+1$ rewards faster, despite a bigger attack doing more damage overall.

This explains the poor win rates of the trained agent, as Ryu is only capable of beating Zangief3 and ChunLi3. Even these win rates are sub-optimal compared to the random dual distribution (RDD) agent, where the win rate is higher despite the reward being lower than the trained agent's in all cases. This again implies the inadequacy of the reward function to describe winning behaviour.

6.5 Single Distribution compared to Dual Distribution Agents

Although the previously described agents show a general learning of the environment's reward structure, both agents utilize slightly different models in terms of reward function and agent architecture. This section compares the results of each trained agent and attempts to draw conclusions as to how the differences in how agents were modeled affected training.

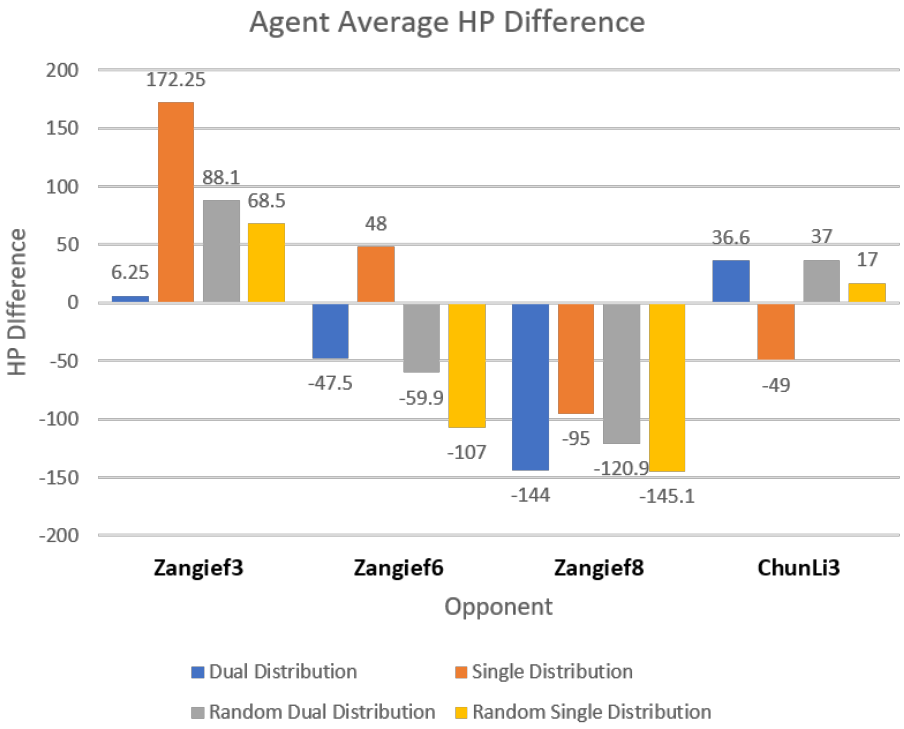

Figure 6.8: Comparison of average HP Difference

The graph showing the average HP difference at the end of matches, shown in figure 6.8, was used to compare trained agents. HP difference was used rather than reward as the reward structure used different scales in either agent, whilst HP difference can be seen as a measure of how much an agent wins or loses by.

Across all agents, the TSD agent performed the best against Zangiefs of all levels, especially Zangief3 where the agent reached near perfect game (176 HP difference) results. The only case in which the TSD agent under performed was the ChunLi3 match, being the only agent with a win rate of less than 50%.

The TDD agent, on the other hand, performed worse on average than its random counterpart. Only against Zangief6 did the TDD barely beat RDD's score by around 12, in all other cases performing at, or below, the RDD's score.

The random agents on the other hand, saw the RDD beating RSD's score in every match. This implies that an SD agent is inherently outclassed by a DD agent. Likely due to the DD agent's action space covering all possible character moves, whereas the SD action space was hand-selected to reduce the search space.

From these observations we may conclude that the TSD agent using a better reward structure was capable of high success against its training partner Zangief. However, the agent has shortcomings when it comes to new opponents, perhaps indicating some form of overfitting to Zangief matches.

The TDD agent, however, may have learned to maximize its reward function, although the learned behaviour does not necessarily encompass "winning" behaviour. This leads us to believe that the reward function was at fault, as the random agent performance of both SD and DD architectures shows DD's capabilities to outperform an SD agent.

Chapter 7

Findings

This project has explored the trained agents of two differing architectures by applying Deep Q-Learning. This section attempts to summarize the main findings of the project in terms of Reinforcement Learning and its applications and how these findings relate to the original purpose of the project.

7.1 Agent Architecture

7.1.1 Action Distributions

As we showed in the results of the random agents, the number of action distributions appears to affect the performance of the agent. Whether this effect is a direct cause of the number of distributions, or whether it is due to the different size in action space between the two agents, is unclear.

It was thought when choosing to train a single distribution agent that it would work better than a dual distribution. This was thought to be the case as an action with large calculated loss (equation 2.7) in a single distribution implicates one bad action from a single distribution, whereas in a dual distribution this would implicate two bad actions, one from each distribution. Even if one of the actions was optimal, the other chosen action could be wrong.

One of the shortcomings of this project is that the TSD agent was "hamstrung" by

limiting its action space. Though at the time during training this was seen as necessary to efficiently train the agent, the end result does not give the agent all the capabilities another human or CPU player would have.

Further exploration of how the number of distributions and the size of action space affect training would be required before the findings in this report should be considered substantial.

7.1.2 Reward Structure

Section 2.1 briefly discussed the theory behind how rewards represent good behaviour. The findings of the two trained agents support some of this theory. Specifically, a reward must correctly encompass winning behaviour for learned behaviour to be considered winning behaviour, and the scale of a reward is important in differentiating the best action when considering two good actions - that is, they both give positive rewards.

In our TDD agent, reward was maximized despite the agent performing poorly in terms of win rates across the board. Our TSD agent, however, performed at a significantly higher level than the lower-level Zangief CPUs, while still having a high returned reward. Due to the lack of baseline agents, we cannot state that the TDD agent's poor performance is specifically due to its reward structure and not some part of its agent architecture. However, the reward of the agent is flawed and as such would have impeded training.

7.1.3 Further Study

Although this project was able to compare varying elements of DQL architecture, the sample size of fully trained agents is simply too small to make any concrete statements on our findings. As an extension to this project, more agents could be trained to provide more solid insight into how agent architecture affects learning. Some potential methods of extending this project could include:

- Reward structure and scale: Alter what gets rewarded and by how much.
- State elements: Adding background elements, such as a player health bar, would be a prospective way to study whether DQL can form an adaptive rule set, one that

changes in response to change in health, for example.

- Network Distributions: Compare agents of varying network distributions that all have the same action space.
- A generalized Street Fighter agent: Train Ryu against multiple different characters using different stages, in the hopes of building an agent that could play against any character effectively.
- A form of Generative Adversial Network: Similar to Alpha Go, training two agents against one another in a "survival of the fittest" style match would potentially increase the opponent skill ceiling above 6th level Zangief. This could fix the problem of an 8th level Zangief cheating to win.
- Update learning algorithm: Already our used DQN with only replay memory and a target network is three years old since Deepmind's first Atari DQN [22]. Recent works have seen major improvements in DQL's effectiveness that could be explored.

7.2 Project Successes and Failures

Over the course of eleven months the project has been a long process of successes and failures. Not only academic knowledge has been gained, skills involving the research, application and methodology of a completely new concept, such as, reinforcement learning have been very interesting to learn.

7.2.1 Project Aim

As the results gathered in this report show, we successfully built and taught a reinforcement learning problem to an agent. Although the number of trained agents were not as high as initially hoped, some of the findings remain interesting and open some doors if reinforcement learning is to be explored further.

The best of the two agents, the single distribution agent, also met our original goal of showing that a learned rule set could outperform a defined rule set. Although this specific

application of Street Fighter may not have much direct real world worth, this finding shows RL's potential uses in the future. Most any problem that can be defined using the environment-agent architecture of RL could be used to find the optimal behaviour without the time-consuming nature of other approaches such as a random search [25].

7.2.2 Project Failures

The core failures of the project lie in its lack of depth in trained agents. Without a number of agents to compare results, many important observations on agent architecture can not be confirmed. This is due in part to the decisions made in training as to what agent architecture to train, as well as due to time constraints caused by the constant gradient explosions in models.

Despite this failing scientific results, the project has been a success in teaching new skills.

7.2.3 Academic Skills

Due to the lack of a known expert in the field at the University of Queensland, much of the content and background of the project was self-researched. Although supervisors were able to help in understanding some of the harder theory and helped guide research in the right direction, much of the theory and background of RL as well as how it was applied to the chosen problem was done without guidance. Although at times challenging, this taught some valuable lessons in how to research and apply knowledge from other applications to our own.

Overall the experience has been very enjoyable and though many problems were faced such as those detailed in section 5.1, these problems allowed the project to grow in new ways towards its final product.

Chapter 8

Conclusion

Throughout the planning and execution of this project, we have demonstrated not only insight into the topic of Reinforcement Learning but also its application to a real world problem. Through the training of two agents of varying architectures, we have shown the capability of Reinforcement Learning in a Deep Q-Learning setting to equal or even outperform a human written program at performing a task.

Though the lack of trained agents hindered both project progress and evaluation, the project has given strong insights into how agent architecture such as reward function and network distributions can affect training. Other agent features such as agent hyperparameters remain mostly unexplored due to lack of trained agents. Of the explored areas, reward function appears to be the most clearly important in defining desired agent behaviour. Whilst the number of action distributions does appear to affect training, the exact nature of this effect remains unclear.

While not groundbreaking, this work has shown a good baseline understanding of Reinforcement Learning and its application. From the work accomplished in this project we have demonstrated both practical and theoretical knowledge, which may be extended in future works as described above, so as to better understand how the underlying mechanics of Reinforcement Learning can affect the trained agent.

Appendix A

Agent Architecture

The network configuration files used to build the agents in tensorforce are shown below.

A.1 Network Configuration

```
[
    {
        "type": "conv2d",
        "size": 8,
        "window": 3,
        "stride": 1
    },
    {
        "type": "conv2d",
        "size": 16,
        "window": 3,
        "stride": 1
    },
    {
        "type": "conv2d",
        "size": 32,
```

```
        "window": 3,
        "stride": 1
    },
    {
    "type": "conv2d",
        "size": 1,
        "window": 1,
        "stride": 1
    },
    {
        "type": "flatten"
    },
    {
        "type": "dense",
        "size": 512
    }
]
```

A.2 Agent Configuration

```
{
    "type": "sf_agent",
    "update_mode": {
        "unit": "timesteps",
        "batch_size": 50,
        "frequency": 1
    },
    "memory": {
        "type": "replay",
        "capacity": 1000,
        "include_next_states": true
```

```
    },
    "optimizer": {
        "type": "clipped_step",
        "clipping_value": 0.001,
        "optimizer": {
            "type": "adam",
            "learning_rate": 1e-3
        }
    },
    "batching_capacity": 100,
    "discount": 0.99,
    "entropy_regularization": null,
    "target_sync_frequency": 200,
    "target_update_weight": 1.0,
    "actions_exploration": {
        "type": "epsilon_anneal",
        "initial_epsilon": 0.9999,
        "final_epsilon": 0.2,
        "timesteps": 4000000
    },
    "saver": {
        "directory": null,
        "seconds": 600
    },
    "summarizer": {
"steps": 100,
"directory": null,
        "labels": ["graph", "total-loss", "actions", "variables",
        "reward", "gradients", "losses", "distribution",
        "print_configuration"]
    },
```

```
    "execution": {
        "type": "single",
        "session_config": null,
        "distributed_spec": null
    }
}
```

A.3 Agent Action Spaces

Single distribution action space:

```
["NA", "Down", "Left", "Right", "Down,Right", "Down,Left", "A",
"B", "L", "R", "X", "Y", "Down,A", "Left,X", "Right,X"]
```

Dual distribution action space:

```
Movement:
['NA', 'Down', 'Left', 'Right', 'Down,Right', 'Down,Left']
Attack:
['NA', 'A', 'B', 'L', 'R', 'X', 'Y']
```

Appendix B

Results

HPDiff	**Reward**	**Win**	**Win**
Zangief3	117	21	1
Zangief3	33	-50	1
Zangief3	-71	-3	0
Zangief3	-54	4	0
Zangief6	-176	17	0
Zangief6	-109	3	0
Zangief6	-73	1	0
Zangief6	-22	13	0
Zangief8	-139	6	0
Zangief8	-155	28	0
Zangief8	-138	17	0
ChunLi 3	123	-73	1
ChunLi 3	142	-3	1
ChunLi 3	-102	14	0
ChunLi 3	39	-55	1
ChunLi 3	-19	-65	0
ChunLi 3	-1	-51	0
ChunLi 3	-100	-282	0

Table B.1: Dual Distribution Agent's full testing results

	HPDiff	**Reward**	**Win**
Zangief 3	176	423	1
Zangief 3	161	330	1
Zangief 3	176	408	1
Zangief 3	176	424	1
Zangief6	45	35	1
Zangief6	80	186	1
Zangief6	-16	-34	0
Zangief6	127	260	1
Zangief6	4	15	1
Zangief 8	-86	-120	0
Zangief 8	-119	-228	0
Zangief 8	-64	-226	0
Zangief 8	-84	-157	0
Zangief 8	-80	-204	0
Zangief 8	-137	-240	0
ChunLi 3	-46	-149	0
ChunLi 3	-1	-51	0
ChunLi 3	-100	-282	0

Table B.2: Single Distribution Agent's full testing results

Appendix C

Program Code

This section describes some pseudo code of the programs used in the project. All mentioned programs were run in Python and were supplied to the supervisor before project submission.

C.1 Training program SFIIBeta.py

This program was built as the key training program, taking care of setting up the agent, connection to the environment, as well as the runner class tensorforce uses in its training loop. This program was built to take command line arguments that could change any of the following:

- Episodes to train for.
- Whether to train agent.
- Training plan i.e. who the agent plays against during training and for how long.
- Whether to apply any randomness to training using ϵ-greedy.
- Network and agent configuration files to use.

The pseudo code for this program is the following:

```
Check command line arguments are valid
Initialize Agent class using network and agent configuration files
Wait for emulator to connect to the SFIIEnvironment class (Environment.connect())
Begin training loop using Runner.run()
Save and terminate agent
```

C.2 Runner.py Training Loop

Once begun the Runner's training loop in Runner.py doesn't end until the agent has be trained for the predefined number of episodes. This training loop utilised the runner class as a go-between the agent and environment. The following pseudo code uses Agent and Environment to represent the classes used in the Runner class.

Episodes trained for E = 0
while E < episodes to train for **do**
    Episode time step T = 0
    S_T = Environment.reset() //Resets match to starting state
    while Terminal is False **do**
        A_T, Q-values = Agent.act(S_T) //Retrieve action and Q-values
        S_{T+1}, Terminal, R_T = Environment.execute(A_T)
        Agent.observe(Terminal, R_T) //Adds observation to batch
        T += 1
    end
    E += 1
end

C.3 Training A Deep Q-Network

The following pseudo code describes the training of a Deep Q-Network in tensorforce and was adapted to suit the Street Fighter environment [12].

Input: the game's full pixel values
Output: Q action value function (from which we obtain policy and select action)
Initialize replay memory D
Initialize an empty batch B
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
for episode = 1 to M **do**
    for t = 1 to T **do**
        Following ϵ-greedy policy, select a_t
        Execute action a_i in emulator and observe reward r_t and state s_{t+1}
        Store transition (s_t; a_t; r_t; s_{t+1}) in B
        If B is at capacity **do**
            Place contents of B into D
            Sample random batch of transitions (s_j; a_j; r_j; s_{j+1}) from D
            Set

$$y_j = \begin{cases} r_j, & \text{if episode terminates at step j+1.} \\ r_j + \gamma max_{a'} \hat{Q}(s_{j+1}, a'; \theta^{-}), & \text{otherwise.} \end{cases} \tag{C.1}$$

            Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ w.r.t. the network parameter
        end
        Every C steps reset $\hat{Q}$= Q, i.e., set θ^{-}= θ
    end
end

C.4 Applying Saliency to Network

Applying Saliency utilised a modified training loop as detailed in the Runner.py pseudo code. Where the Agent.act() call is taken, the Q-values from this state are used to calculate saliency as seen below.

Q-values of current state Q_t
Current state s_t
for positions (x,y) at intervals across the state **do**
    Create Gaussian filter G blurred around the x,y position
    Occlude state s_t using mask G
    Recalculate Q-values Q_{tnew} using blurred state
    Calculate mean squared error E between original and altered Q-values
end
Normalize E values between 0 and 128
Add normalized E to a channel of the Red, Green, Blue state s_t

Bibliography

[1] Capcom. Game series sales, 2018.

[2] K. Simonyan I. Antonoglou A. Huang A. Guez T. Hubert L. Baker M. Lai A. Bolton Y. Chen T. Lillicrap F. Hui L. Sifre G. Van Den Driessche T. Graepel D. Silver, J. Schrittwieser and D. Hassabis. Mastering the game of go without human knowledge. *Nature*, 550, 2017.

[3] Deepmind. Alphago, 2018.

[4] A. Fletcher. How we built an ai to play street fighter ii - can you beat it? *Gyroscope Software*, 2017.

[5] Sam Greydanus. Visualize atari. https://github.com/greydanus/visualize_atari, 2018.

[6] Sam Greydanus, Anurag Koul, Jonathan Dodge, and Alan Fern. Visualizing and understanding atari agents. *CoRR*, abs/1711.00138, 2017.

[7] Gyroscope. Gyroscope: Deliver adaptive, personalized game experiences, 2017.

[8] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Ian Osband, et al. Deep q-learning from demonstrations. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[9] Steven Kapturowski, Georg Ostrovski, Will Dabney, John Quan, and Remi Munos. Recurrent experience replay in distributed reinforcement learning. In *International Conference on Learning Representations*, 2019.

[10] Alexander Kuhnle, Michael Schaarschmidt, and Kai Fricke. Tensorforce: a tensorflow library for applied reinforcement learning. https://github.com/tensorforce/tensorforce, 2017.

[11] E. Levin, R. Pieraccini, and W. Eckert. Using markov decision process for learning dialogue strategies. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*, volume 1, pages 201–204 vol.1, May 1998.

[12] Yuxi Li. Deep reinforcement learning: An overview. *CoRR*, abs/1701.07274, 2017.

[13] OpenAI. Openai five. https://openai.com/five/, 2019.

[14] Jurgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[15] Hanan Shteingart and Yonatan Loewenstein. Reinfocement learning and human behaviour. *Current Opinion in Neurobiology*, 2014.

[16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[17] Open Source. Bizhawk. https://github.com/TASVideos/BizHawk, 2016-2019.

[18] R. S. Sutton and A. G. Barto. Reinforcement learning: An introduction. *The MIT Press*, 2017.

[19] Sebastian B. Thrun. Efficient exploration in reinforcement learning. Technical report, 1992.

[20] Michel Tokic. Adaptive ϵ-greedy exploration in reinforcement learning based on value differences. In Rüdiger Dillmann, Jürgen Beyerer, Uwe D. Hanebeck, and Tanja Schultz, editors, *KI 2010: Advances in Artificial Intelligence*, pages 203–210, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[21] John N. Tsitsiklis and Benjamin Van Roy. An analysis of temporal-difference learning with function approximation. Technical report, IEEE Transactions on Automatic Control, 1997.

[22] D. Silver A. A. Risi J. Veness M. G. Bellemare A. Graves M. Ridemiller A. K. Fidjeland G. Ostrovski S. Petersen C. Beattie A. Sadik I. Antonoglour H. King D. Kumaran D. Wierstra S. Legg V. Mnih, K. Kavukcuoglu and D. Hassabis. Human-level control through deep reinforcement. *Nature*, 518, 2015.

[23] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, pages 2094–2100, 2016.

[24] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy P. Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. Starcraft II: A new challenge for reinforcement learning. *CoRR*, abs/1708.04782, 2017.

[25] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. *CoRR*, abs/1707.07012, 2017.