Towards Inherently Adaptive First Person Shooter Agents using Reinforcement Learning

Frank G. Glavin B.Sc., M.Sc.

Discipline of Information Technology
College of Engineering and Informatics
National University of Ireland, Galway

A thesis submitted for the degree of

Doctor of Philosophy

September, 2015

Research Supervisor: Dr. Michael G. Madden

Abstract

Reinforcement learning (RL) is a paradigm which involves an agent interacting with an environment. The agent carries out actions in the environment and receives positive reinforcement for actions that are deemed "good" and penalties for "bad" actions based on a reward signal. The goal of the learning agent is to maximise the amount of reward it receives over time.

This thesis presents several new behavioural architectures for controlling non-player characters (NPCs) in a modern first-person shooter (FPS) game using reinforcement learning. NPCs are computer-controlled players that are traditionally programmed with scripted, deterministic behaviours. We propose the use of reinforcement learning to enable the NPC to learn its own strategies and adapt them over time. We hypothesise that this will lead to greater variation in gameplay and produce less predictable NPCs.

The first contribution of this thesis is the design, development and testing of two general purpose Deathmatch behavioural architectures called Sarsa-Bot and DRE-Bot. These architectures use reinforcement learning to control and adapt their behaviour. We demonstrated that they could learn to play competently and achieve good performance against fix-strategy scripted opponents.

Our second contribution is the development of a reinforcement learning architecture, called RL-Shooter, specifically for the task of shooting. The opponent's movements are read in real-time and the agent chooses shooting actions based on those that caused the most damage to the opponent in the past. We carried out extensive experimentation that showed that the RL-Shooter architecture could produce varied gameplay, however, there was not a clear upward trend

in performance over time. This led to our third contribution which involved developing extensions to the SARSA(λ) algorithm called Periodic Cluster-Weighted Rewarding and Persistent Action Selection. We designed these to improve the learning performance of RL-Shooter and we demonstrated that the use of the techniques resulted in a clear upward trend in the percentage hit accuracy achieved over time.

Our final contribution is a skill-balancing mechanism that we developed, called Skilled Experience Catalogue, which is based on a by-product of the learning process. The agent systematically stores "snapshots" of what it has learned during the different stages of the learning process. These can then be loaded during the game in an attempt to closely match the abilities of the current opponent. We showed that the technique could successfully match the skill level of five different scripted opponents with varying difficulty settings.

I would like to dedicate this thesis to my family.
My mother, Ann, my sister, Paula, and my brother, John, have always been exceptionally encouraging and supportive.
Thank you for everything.

Acknowledgements

First and foremost, I would like to thank my research supervisor, Dr. Michael Madden, who has been an excellent research mentor and role model to me over the years. His continuous encouragement and expert guidance have been truly invaluable to me and I am forever grateful.
I would like to thank Dr. Padraig O'Flaithearta and Dr. Jonathan Shannon, two previous research colleagues of mine and now two great friends. The countless numbers of lunches, teas and discussions over the years really helped me out more than you may ever know.
I would like to thank my colleagues with whom I shared a research room. Catherine, Alan, Stephen, Darren, Ankita and Hamda, it was truly a joy to work in the presence of such enthusiastic and knowledgeable people.
Mark, Chris, Martina and all those that have worked in the ComputerDISC programming support centre, I am very grateful that we could share our programming knowledge and advance together as tutors over the years. I have also made some really great friends.
I wish to thank the members of my Graduate Research Committee, Dr. Colm O'Riordan, Ms. Karen Young and Dr. Sam Redfern and the Information Technology Discipline administrative and technical staff, Tina Earls, Mary Hardiman, Joe O'Connell and Peter O'Kane.
I would like to thank the developers of the Pogamut 3 toolkit, in particular, Michal Bida and Jacub Gemrot for the discussions, technical support and advice that they provided during development.
Finally, some of my friends that moved abroad a few years ago such as Daniel, Cormac and Colin, I really appreciate you all staying in touch and being so supportive.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification.

Frank G. Glavin
September, 2015

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The most successful applications of artificial intelligence (AI) in games to date have occurred in computer simulated versions of traditional board games. AI players of games such as Backgammon [Tesauro, 1995], Othello [Buro, 1997], Chess [Hsu, 2002], Checkers [Schaeffer *et al.*, 2007] and Go [Gelly & Silver, 2008] have reached a level of play comparable to, or exceeding that of, world champion human players. Since the turn of the century, the application of AI to commercial computer games[1] has received a lot of attention [Woodcock *et al.*, 2000]. There are several distinct differences when it comes to applying AI to modern commercial computer games as opposed to traditional board games. Some of these, as outlined by Spronck [2005], are as follows. Commercial computer games can involve fast-paced interaction with 3D environments, multiple objectives, partial observability and agents interacting in an adversarial setting whereas board games are generally slower, turn-based, and follow more simplistic rules. Players of commercial computer games often require a series of additional skills to complement the core objective of the game. For instance, a human player in an first-person shooter (FPS) game must learn how to navigate through the environment, collect items, and identify opponent players, through visual and auditory stimuli, before engaging in combat or completing other game objectives. This requirement of

[1] There are, of course, commercial simulations of traditional board games but here, when using the term *commercial computer game*, we are referring to modern computer games that take place in 3D environments such as first-person shooter (FPS) games which are described later in Section 1.1.2.

additional skills varies across different genres of commercial computer game and the game types found within these genres. Traditional board games, on the other hand, have more directed play in which knowledge of a simple rule set will usually suffice.

The goal of successful AI in commercial computer games is to provide entertainment to human players [Chan *et al.*, 2004; Lidén, 2003]. This contrasts with the goal of AI in traditional board games in which the objective is to defeat the human opponents [Spronck, 2005].

In this thesis, we are concerned with the application of AI to modern commercial computer games, a thriving research domain which has continued to receive considerable interest in recent years as predicted by Laird & Van Lent [2001]. For the remainder of this thesis, and unless otherwise stated, we will use the term *computer game* to refer to these types of games and not their traditional board game counterparts as mentioned above.

1.1 Overview

The concept of an artificially intelligent agent participating in a computer game has been around as long as computer games have existed. One of the first examples dates back to 1951 when a special purpose computer was developed, called *Nimrod*, for playing the game of Nim. This was showcased at the Exhibition of Science during the 1951 Festival of Britain [Mitchell, 2012]. The creators of Nimrod did not have entertainment in mind during development, rather their primary goal was to show off the mathematical ability of computers at the time [Donovan & Garriott, 2010]. Since the development of well-known games such as *Pong* and *Pac-Man* in the 1970s and 1980s, the core objective of using AI in games has been to increase the entertainment for players. Historically, these early games first appeared as arcade games and the amount of money the machines received was dependent on how much enjoyment the user was experiencing [Tozour, 2002]. Over the past fifteen years, the gaming industry has included applications of AI in various different forms [Yannakakis, 2012] and now has yearly revenue that exceeds that of the movie industry [Fang *et al.*, 2015]. Multi-objective 3D worlds with real-time decision-making that span across various genres make it an ideal

testbed for evaluating new AI techniques.

The purpose of this thesis is to investigate the use of reinforcement learning (RL), described in Section 2.2, to automate the learning of non-player characters (NPCs) in a first-person shooter (FPS) game and generate real-time adaption of behaviour. While the application presented in this work is specific to an FPS game, the findings are relevant to a plethora of similar domains that involve real-time learning in multi-objective and unpredictable environments.

1.1.1 Gaming Industry

The computer and video game industry has been going from strength to strength in recent years and with large-scale development projects, as well as billions of dollars' worth of sales, it is continuing to grow. A report in 2013 by market research firm Gartner [Blau *et al.*, 2013], which is summarised in the following press release [Gartner, 2013], stated that the total market spending for the video game industry was expected to grow by 19% to US$ 111 Billion in 2015, up from US$ 93 Billion in 2013. Handheld gaming is the only category expected to decline in sales while the others continue to grow, with mobile gaming being forecast to have the largest growth. The forecasted market revenue data for the worldwide video game industry is summarised in Table 1.1 below. The latest generation[1] of home video game consoles, Microsoft's *Xbox One* and Sony's *PlayStation 4*, were released onto the market in 2013.

Commercial game developers are primarily concerned with producing games that are entertaining for end-users in order to increase overall sales. They are also confined to certain development factors such as the scope of the project, quality assurance and memory requirements [Millington & Funge, 2009]. The increased processing power of these video game consoles means that more complex AI algorithms can be added to the games without it being detrimental to gameplay. In the past, game developers have been reluctant to include the latest AI techniques with most of the processing power being reserved for graphics and seamless gameplay. As processing power and storage is increased on PCs

[1]The release of video game consoles are categorised from the "first generation" all the way up to the current "eighth generation" on an incremental scale of hardware and software improvements.

Segment	2013	2014	2015
Video Game Console	$44,288M	$49,375M	$55,049M
Handheld Video Games	$18,064M	$15,079M	$12,399M
Mobile Games	$13,208M	$17,146M	$22,009M
PC Games	$17,772M	$20,015M	$21,601M
Total Video Game Market	$93,282M	$101,615M	$111,057M

Table 1.1: Worldwide video game market revenue from 2013 to 2015 in millions of US Dollars. (Source: [Gartner, 2013])

or gaming consoles, the integration of new AI techniques may no longer be as constrained as they have been in the past. In this competitive industry, novel AI and adaptive computer-controlled opponents are becoming a focal point for persuading players to purchase new games [Schwab, 2009].

1.1.2 First Person Shooter Games

The first-person shooter genre of computer games has existed for over twenty years and involves a human player taking control of a character, or avatar, in a 3D world and engaging in combat with other players, both human and computer-controlled. Human players perceive the world from the first-person perspective of their avatar and must navigate through the environment, collecting health items and guns, in order to find and eliminate their opponents. The most straightforward FPS game type is called a *Deathmatch* where each player must work by themselves with the objective of killing more opponents than anyone else. The game ends when the score limit has been reached or the game time limit has elapsed. An extension to this game type, *Team Deathmatch*, involves two or more teams of players working against each other to accumulate the most kills. Objective based games also exist where the emphasis is no longer on kills and deaths but on specific tasks in the game which, when successfully completed, result in acquiring points for the player's team. Two examples of such games are *Capture the Flag* and *Domination*. The former involves retrieving a flag from the enemies' base and returning it to the player's base without dying. The latter

involves keeping control of predefined areas on the map for as long as possible. All of these game types require, first and foremost, that the player is proficient when it comes to combat. Human players require many hours of practice in order to become familiar with the game controls, maps and to build up quick reflexes for accuracy. Replicating such human behaviour in computer-controlled bots is a difficult task considering that they do not have physical controllers and must read the game circumstances from the system as opposed to basing their decision-making on audio/visual stimuli and physical reflexes. It is only in recent years that gradual progress has been made using various AI algorithms, many of which are discussed in Section 2.4.3, to work towards accomplishing this task.

1.2 Motivations

As graphics in modern computer games approach photorealism, an emphasis for game developers is switching towards improving the intelligence of non-player characters [Spronck, 2005]. Physics engines and computer game graphics have seen steady progress over the years thanks to the theoretical advances in computer-generated imagery (CGI) and the increased processing power and memory available on high definition video game consoles and gaming PCs. AI in games, on the other hand, has seen the use of well understood and robust traditional techniques (see Section 2.3.1 for examples) with very little progress of academic research applied to commercial games [Yannakakis, 2012]. Some of the reasons for this can include a lack of trust and understanding of advanced AI techniques by commercial game developers and a lack of CPU resources available to process game AI. The growth of academic research into commercial games in the past decade coupled with the ever-improving computing resources of dedicated machines has slowly led to a bridging of the gap between the academic AI research community and game developers. We believe that research into AI techniques that can aid developers in creating NPCs that are self-contained, with the ability to learn and adapt themselves, are needed. This work will not only advance the use of AI in commercial computer games but also has the potential to lead to progress in operationally similar domains that involve interactive, real-time decision-making such as autonomous robotics.

1.2.1 Current Limitations of Game AI

When the computer-controlled opposition is too strong, human players can become frustrated with the gameplay. Likewise, opponents that are too weak result in predictable games in which human players do not feel challenged [Koster, 2013]. We believe that successful game AI requires techniques to be developed in which the NPCs can learn good tactics independently as well as being both unpredictable and adaptive to their surroundings. These are the traits that we believe are essential in generating behaviour that will result in an immersive and entertaining experience for human players. Human players typically want computer-controlled opponents to surprise them and attempt to defeat them in ways that they had not anticipated [Rouse III, 2010].

"*AI cheating*" in game development involves the NPC being given extra information during the game that is not available to human players. For instance, the NPC may be informed about the map and likely action spots or may auto-focus their weapon on the human player when they become visible. They must also make use of path-finding algorithms in order to quickly locate game objectives and traverse through the virtual world. Such "cheats" overcome the need for the NPC to have perfect visual and auditory sensors in order to play the game. This is not a problem if the cheating remains undetectable to human players but this can often not be the case [Conroy *et al.*, 2011]. AI cheating can have either a positive or negative effect on the gameplay depending on the perception of the human opponent. For example, it is unlikely that a human player will detect that the bot is cheating with extra knowledge about weapon selection or going to map "hot spots" whereas they are likely to detect cheating with the assisted aiming of weapons. The reason for this is that some cheat information is processed implicitly by the bot whereas other cheat information explicitly affects the bot's behaviour that is visible to a human player and is thus easier to detect. One way to reduce the noticeability of apparent cheating by a bot would be to introduce the elements of uncertainty and surprise that can be found when a human is playing the game. This behaviour is unlikely to be achieved if game developers continue to use traditional techniques, some of which are described in Section 2.3.1, and we believe that creating NPCs that can adapt to different

circumstances will appear more natural and less "super-human" as they will be prone to making mistakes while learning and improving over time.

1.2.2 Advances in Artificial Intelligence

The last decade has seen considerable growth in the availability of testbeds for carrying out AI experimentation, especially those related to gaming, as we will see in Section 3.1.3. Several different competitions are run yearly to advance the state-of-the-art. One of the most notable success stories in recent times are the two entries to the 2012 Bot Prize competition which surpassed the humanness rating of 50%. This competition, which is described later in Section 2.5.1, invites researchers to submit AI-driven FPS bots with the goal of the competition to appear as humanlike as possible. There are many other competitions and interesting testbeds, a selection of which will be detailed in Chapter 3.

This thesis focuses on the application of reinforcement learning in FPS games. This is a challenging domain that involves the learner being deployed in a multi-objective, dynamic environment where real-time decision making is necessary. It is hoped that addressing these problems in an interactive and multi-objective gaming environment could also contribute to advancing other domains such as robotics. Many reinforcement learning robotic tasks involve navigating through, and interacting with, environments with a large amount of uncertainty in order to complete one or more tasks. Continuous sensor readings are taken to describe the environment and the robot must decide which actions to take in real-time. Kober *et al.* [2013] stress the importance of a well-defined reward signal and note that this can be particularly difficult to achieve in robotics. The reward design is equally as important for an FPS agent and this thesis includes a discussion of our approach to reward design for each of the architectures that we developed.

1.3 Research Questions and Objectives

This thesis proposes the development and analysis of a series of different computer-controlled behavioural architectures that use reinforcement learning algorithms to produce NPCs for a commercial first-person shooter game. The goal of these behavioural architectures is to create NPCs that are capable of reacting to their environment in real-time and adjusting their behaviour over time, based on experience, to improve their overall performance. We have identified the following research questions and research objectives based on this initial proposal.

1.3.1 Research Questions

R.1 *Can an autonomous reinforcement learning agent use feedback from its environment to guide successful decision-making in a real-time adversarial 3-dimensional world with multiple objectives? What are the benefits and drawbacks of using such an approach?*

R.2 *Is reinforcement learning an effective technique for creating unpredictable and adaptable opponents in a competitive environment by learning to perform a task with a single objective?*

R.3 *What specially designed updating and rewarding mechanisms, if any, can be developed to improve the learning performance of a reinforcement learning agent?*

R.4 *Can reinforcement learning be used to aid the process of dynamically matching the skill level of an agent with the varied skill levels of several fixed-strategy opponents?*

1.3.2 Research Objectives

The following core research objectives were drawn up based on the aforementioned research questions. The successful completion of these objectives form part of the research contributions of this work which are described in the next section.

- Design, develop and implement an architecture(s) for a multi-skilled Deathmatch NPC that uses a combination of heuristic rules, based on human player knowledge, and reinforcement learning to facilitate adaptability and learning from experience. Carry out experimentation using a commercial FPS game in order to analyse the performance of the agent. This will allow us to determine if reinforcement learning is effective in this domain and enable us to identify the advantages and/or difficulties associated with such an approach. The completion of this research objective addresses research question R.1.

- Design, develop and implement an architecture for the specific task of shooting for a Deathmatch NPC. Carry out experimentation using a commercial FPS game in order to analyse the performance of the shooting task for the Deathmatch NPC. This will allow us to concentrate on a single task, with a single source of reward, and determine if such an architecture is better suited to facilitate learning and continous improvement with experience. The completion of this research objective addresses research question R.2.

- Design, develop and implement an extension, or extensions, to the current reinforcement learning paradigm to address any of the shortcomings that were identified from our earlier analyses. Carry out experimentation to determine the efficacy of such an approach. This will lead to the improvement in performance of learning to perform a single task using reinforcement learning. The completion of this research objective addresses research question R.3.

- Design, develop and implement a skill-balancing mechanism which incorporates the use of reinforcement learning in order to dynamically adjust the skill level of the NPC, during the game, to approximately match that of its current opponent. Carry out experimentation to test and validate this mechanism. This will allow us to investigate whether or not a reinforcement learning inspired approach can successfully balance gameplay against opponents of varying skill levels. The completion of this research objective addresses research question R.4.

1.4 Research Contributions

In this thesis, we present the design and implementation of several different learning architectures for enabling an NPC to acquire knowledge from experience while playing in an FPS game. We construct several different representations of the NPC's perception of the virtual world and design the actions that are available to it. We also develop specific reward signals which guide the decision-making agent towards good performance. The following describes the principal research contributions of this body of work. These are discussed in further detail in Section 8.1.

1. We carry out a survey of all of the relevant state-of-the-art literature.
2. We review and present various different testbeds, development toolkits and competitions that are used for carrying out AI experimentation in a variety of different domains.
3. We design, develop and present our *Sarsa-Bot* architecture that uses the SARSA algorithm [Rummery & Niranjan, 1994] to drive the logic of a Deathmatch NPC.
4. We design, develop and present our Danger Replenish Explore Bot (*DRE-Bot*) architecture, which uses the SARSA(λ) algorithm [Sutton & Barto, 1998], by implementing a series of refinements to the Sarsa-Bot architecture based on our experimental analysis.
5. We design, develop and present our *RL-Shooter* architecture, which is a single task architecture, to enable an NPC to adapt its shooting technique over time based on a dynamic reward signal from the amount of damage caused to opponents.
6. We design, develop and present our techniques of *Periodic Cluster-Weighted Rewarding* and *Persistent Action Selection*. We develop several refinements to our RL-Shooter architecture while implementing these techniques.

7. We design, develop and present our skill-balancing mechanism called *Skilled Experience Catalogue* that uses the agent's learning tables, stored from different levels of experience, to approximately match the skill level of an opponent whilst continually learning from in-game experience.

8. We carry out extensive experimentation and an empirical evaluation of each of the behavioural architectures and mechanisms that we developed.

9. Our findings are disseminated to the research community, peer-reviewed and published in international conferences and a journal as listed in the next section.

1.5 Publications

Some of the key aspects from this thesis have already been published as listed below.

- F. G. Glavin and M. G. Madden, "*Learning to Shoot in First Person Shooter Games by Stabilizing Actions and Clustering Rewards for Reinforcement Learning*" in CIG 2015, IEEE Conference on Computational Intelligence and Games, Taiwan, 2015, pp. 344-351.
  Parts of this publication appear in Chapter 7 of this thesis.

- F. G. Glavin and M. G. Madden, "*Adaptive Shooting for Bots in First Person Shooter Games using Reinforcement Learning*" in IEEE Transactions on Computational Intelligence and AI in Games, 2015, vol. 7, issue 2, pp. 180-192.
  Parts of this publication appear in Chapter 6 of this thesis.

- F. G. Glavin and M. G. Madden, "*DRE-Bot: A Hierarchical First Person Shooter Bot using Multiple SARSA(λ) Reinforcement Learners*" in 17th International Conference on Computer Games (CGAMES), 2012, pp. 148-152. [**Best Student Paper Award**]
  Parts of this publication appear in Chapter 5 of this thesis.

- F. G. Glavin and M. G. Madden "*Incorporating Reinforcement Learning into the Creation of Human-like Autonomous Agents in First Person Shooter Games*" in GAMEON 2011, the 12th annual European Conference on Simulation and AI in Computer Games, 2011, pp. 16-21.
  Parts of this publication appear in Chapter 4 of this thesis.

  Publications under Preparation:

- F. G. Glavin and M. G. Madden, "*Skilled Experience Catalogue: A First Person Shooter Skill-Balancing Mechanism based on the Knowledge Timeline of a Reinforcement Learning Agent*"
  The initial findings from this work are disclosed in Chapter 7 of this thesis.

1.6 Thesis Structure

This thesis consists of eight chapters, the first of which has introduced the research by providing information on the gaming industry and describing FPS computer games. This was followed by outlining the motivations for our work which described some of the current limitations with respect to AI in games and detailed a brief discussion of the advances in academic research in this area. We presented our research questions and objectives and then summarised the work that we carried out to address these. This was followed by listing the principal contributions from our body of work. We will now conclude this chapter with a summary of the content for each of the remaining chapters.

Chapter 2 provides a detailed review of all of the relevant literature to put this research into perspective. The first section provides an introduction into the broad area of AI in order to provide the reader with a solid foundation which defines the concept of AI, provides a brief look at its history and discusses some of the most notable success stories to date. This is followed by introducing the background details of the concept of reinforcement learning. We then describe some traditional approaches to game AI and provide summaries for machine learning techniques that are often combined with reinforcement learning. We review general applications of reinforcement learning before focusing specifically on AI

techniques used in the FPS genre of computer game. The chapter is concluded with a discussion.

Chapter 3 describes currently available software toolkits and testbeds that were investigated for carrying out research and experimentation with AI. This is followed by a discussion of why Pogamut 3 was chosen as the development toolkit and why the game Unreal Tournament 2004 was chosen as the environment. The important features of the toolkit and the game are also explained in detail.

Chapter 4 presents the development details of the Sarsa-Bot architecture for controlling an NPC in the FPS game Unreal Tournament 2004. This chapter begins by reiterating some of the motivations for this work and this is followed by a detailed description of each of the architecture components. The chapter concludes by detailing the experimentation that we carried out before providing an analysis and discussion of the results.

In Chapter 5, we discuss how the overall design of the Sarsa-Bot was refined to create the hierarchical DRE-Bot architecture which is composed of multiple learners that are triggered by high-level modes. Again, we describe the architecture in detail before proceeding to present and discuss the experimentation that we carried out.

Chapter 6 describes a specialised learning architecture, designed for a single task, that enables an NPC to learn how to competently shoot and adapt its technique over time based on building up knowledge from game experience. The focus of the bot's learning is based solely on the task of shooting so other tasks, such as turning to face opponents and navigation, have hard-coded rules. We carry out an extensive analysis, based on several categories of weapon and opponents with differing skill levels, before presenting our findings.

Chapter 7 introduces the reinforcement learning mechanisms that we developed called Periodic Cluster-Weighted Rewarding, Persistent Action Selection and Skilled Experience Catalogue. The first two techniques extend the traditional reinforcement learning framework to facilitate a reward allocation that is being carried out multiple times a second in the environment. These techniques are tested using a single weapon in the game. The final technique involves using a stored catalogue of shooting experience, gathered by a reinforcement learning agent, to approximately match the skill level of an opponent in a Deathmatch

game. We test a single instance of the technique against five opponents, with different skill levels, and present the results.

Chapter 8 provides a summary of all of the research that we have completed throughout this body of work. The results are then discussed in summary and our core contributions are presented. Possible directions for future work are then proposed before the thesis is brought to a close with some concluding remarks.

Chapter 2

Literature Review

This chapter provides a review of the state-of-the-art research and background information that is relevant to the work that we present in this thesis. We begin by defining artificial intelligence, briefly outlining its history and discussing some notable success stories. This is followed by introducing the theory behind reinforcement learning before presenting the existing algorithms that are most relevant to our work. We then summarise a selection of traditional approaches to applying artificial intelligence to games and discuss some machine learning approaches that can be combined with reinforcement learning. We categorise and summarise a series of reinforcement learning applications and we then focus on the first-person shooter genre of computer game before concluding the chapter with a discussion.

2.1 Artificial Intelligence

2.1.1 Definition of Artificial Intelligence

Artificial intelligence (AI) is a branch of computer science and engineering that is concerned with the study, design and creation of intelligent computer programs and machines. Russell & Norvig [2010] provide definitions for AI in which they organise it into four categories. These categories are: thinking humanly; acting humanly; thinking rationally; and acting rationally. These definitions will now be summarised based on the aforementioned text. The authors state that in order

to make a claim that a program can think like a human we must first understand the thinking process of humans. This can be achieved through a variety of ways such as introspection, psychological experiments and brain imaging. Only with a precise theory of the human mind does it become possible to express this theory as a computer program.

In order to test if a machine could act like a human, Alan Turing [Turing, 1950] proposed a test known as the *Turing Test*. In this test, a computer would be deemed intelligent if a human interrogator could not tell whether written responses came from a human or a computer. According to Russell & Norvig [2010], the computer would have to possess the capabilities of natural language processing, knowledge representation, automated reasoning, machine learning, computer vision and robotics in order to pass a rigorous full Turing Test.

A program must make use of logic in order to think rationally. The study of logic was initiated by the Greek philosopher Aristotle who proposed the use of syllogisms to provide argument structures that always yield correct conclusions when given correct premises. These are logical arguments in which a proposition is inferred from the premises. For example "Socrates is a man; all men are mortal; therefore Socrates is mortal". In the 19th century, logicians created a specific notation called *classical symbolic logic* for statements that describe all sorts of objects in the world. In 1965, programs already existed that, in principle, could solve any solvable problem in logical notation [Russell & Norvig, 2010]. The AI approach for rational thinking aims to build upon such programs. The authors identify two main obstacles with such an approach. Firstly, taking informal knowledge and attempting to state it in formal terms is not always a trivial task. Also, solving a problem in principle and solving a problem in practice are two very different things. In the absence of guidance for the reasoning, computer resources can quickly be used up even for small sized problems. In order for a computer agent to act rationally, it must act in order to achieve the best outcome, or, when there is an element of uncertainty, the best expected outcome. Perfect rationality is not feasible in complex environments.

2.1.2 The Beginning of Artificial Intelligence

The term "artificial intelligence" was coined by John McCarthy in 1956. He and a group of researchers had organised a two-month workshop at Dartmouth College during the summer of that year with the following proposal[1]:

> *We propose that a 2 month, 10 man study of artificial intelligence be carried out during the summer of 1956 at Dartmouth College in Hanover, New Hampshire. The study is to proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it. An attempt will be made to find how to make machines use language, form abstractions and concepts, solve kinds of problems now reserved for humans, and improve themselves. We think that a significant advance can be made in one or more of these problems if a carefully selected group of scientists work on it together for a summer.*

The workshop itself did not lead to any significant breakthroughs in the field but it did introduce some leading figures to each other. These were the people, including their students and colleagues, who would have the biggest impact in the field over the next 20 years [Russell & Norvig, 2010]. Since its inception in the 1950s, advancements in artificial intelligence have brought a series of success stories over the years. The next section will take a brief look at some of these applications.

2.1.3 Success Stories in Artificial Intelligence

The following is a list that we have devised of some notable examples of artificial intelligence being successfully applied in a variety of different domains. The purpose of these short summaries is to give the reader a general overview of some notable success stories and milestones that have occurred.

A computer backgammon player called *TD-Gammon* [Tesauro, 1992, 1995] was developed in the early 1990s which achieved a level of play close to the top

[1] http://www-formal.stanford.edu/jmc/history/dartmouth/dartmouth.html

human players in the world at the time. This used an artificial neural network (ANN) which was trained using a temporal difference learning algorithm called TD-Lamda [Sutton, 1988].

IBM's *Deep Blue* Supercomputer played a chess match, in 1997, against the then reigning World Chess Champion Gary Kasparov. Deep Blue became the first computer to beat a world champion in chess winning by a score of 3.5 to 2.5 in an exhibition match [Goodman & Keane, 1997]. Kasparov felt that he had witnessed playing against a "new kind of intelligence". The stock value of IBM increased by 18 billion US dollars as a result of this success. In more recent years, computers have been able to win matches more convincingly against human opponents.

Roomba is an autonomous robot vacuum cleaner that was commercially introduced in 2002 by the company iRobot. The robot vacuums the floor while navigating the living space. Roomba does not have to map out the floor that it is cleaning and it uses a variety of simple AI algorithms for real-time decision making based on the limitations of the environment. In February 2014, it was reported by the company website that over 10 million Roomba robots had been sold worldwide to date[1]. Forlizzi & DiSalvo [2006] present research on the use of the Roomba robot vacuum in the home and discuss how design can influence human-robot interaction.

A robotic car called *Stanley* won the 2005 DARPA Grand Challenge. The DARPA (Defense Advanced Research Projects Agency) Grand Challenge is a competition that is run for testing autonomous vehicles. Stanley was fitted with a radar, various cameras and laser rangefinders in order to sense the environment. Steering, acceleration and braking was carried out using on-board software [Thrun *et al.*, 2006]. Sebastian Thrun has led the development of technology for cars that drive themselves with Google's Driverless Car [Thrun, 2011] currently at the forefront of this research.

IBM's *Watson*, which is a result of the DeepQA Project [Ferrucci *et al.*, 2010], is a computer system based on AI techniques which can answer questions that are posed using natural language. Watson became known worldwide in 2011 when it competed on a special edition of the game show Jeopardy in order to test its

[1] `http://www.irobot.com/us/Company/About/Our_History.aspx` (Accessed March 2015)

abilities. Watson competed for three episodes against two opponents: the biggest all-time money winner and the player with the record for the longest winning streak on the game. The supercomputer came in first place winning a prize of one million US dollars which was donated to two charities. Baker [2011] provides details about Watson from its inception as an IBM project up to its performance on Jeopardy.

2.2 Reinforcement Learning Theory

Reinforcement learning (RL) is a branch of artificial intelligence in which a learner, often called an *agent*, interacts with an environment in order to achieve an explicit goal or goals [Sutton & Barto, 1998]. A (typically finite) set of states exist, called the *state space*, and the agent must choose an available action from the *action space* when in a given state at each time step. The approach is inspired by the process by which humans learn. The agent learns from its interactions with the environment, receiving feedback for its actions in the form of numerical rewards, and aims to maximise the reward values that it receives over time. The state-action pairs that store the expected value of carrying out an action in a given state comprise the *policy* of the learner. The agent must make a trade-off between exploring the effects of taking novel actions and exploiting the knowledge that it has acquired from earlier exploration.

Figure 2.1, below, shows the simple interaction between an agent and the environment. The agent takes an action in the environment and then receives notification of what state it is now in and what reward it has received. The environment can be non-deterministic which means that taking the same action in the same state on two different occasions may result in different outcomes. We assume, however, that the environment is *stationary* in that the probabilities of making transitions or receiving specific reward do not change over time [Kaelbling *et al.*, 1996]. Two common approaches to storing/representing policies in reinforcement learning are *generalisation* and *tabular*. With generalisation, a function approximator is used to generalise a mapping of states to actions. The tabular approach, which is used in our research, stores numerical representations

of all state-action pairs in a lookup table.

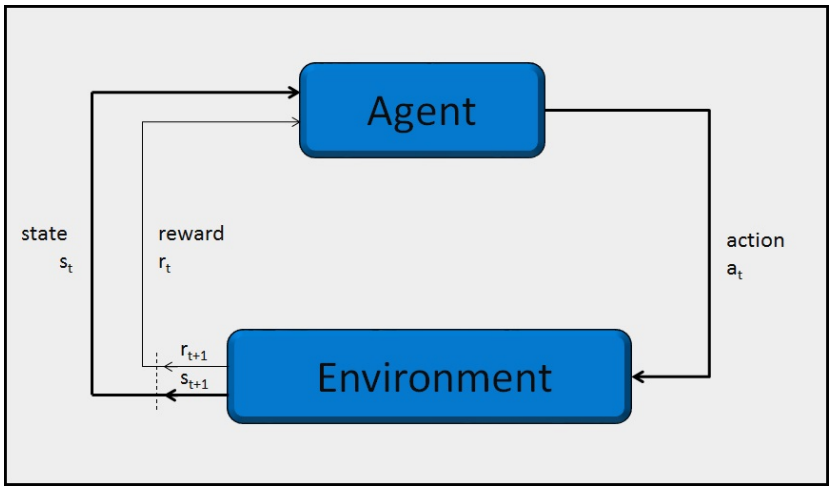

Figure 2.1: The interactions between the agent and the environment (Based on Figure 3.1 of Sutton & Barto [1998])

In addition to the agent and its environment, Sutton & Barto [1998] have identified four primary sub-elements that form an RL system. These are: a policy; a reward function; a value function; and a model of the environment. These will now be explained in brief.

Policy: This is a definition of the agent's behaviour in a given situation. It is essentially a mapping of the states being perceived by the agent to the actions that should be carried out while in those states. The policy, which can consist of a function, a lookup table or a search process, is the core element of a reinforcement learner which drives the decision-making.

Reward Function: This function assigns a single numeric reward value to each state (or state-action pair) in the environment to represent the desirability of being in the state (or the desirability of choosing an action in a particular state). It represents the goals of the reinforcement learning system. The individual reward values cannot be altered by the agent but can be used as the basis of altering the agent's policy.

Value Function: The value function represents the total amount of reward that an agent can expect to acquire from this state over future states. While

the reward function represents the immediate desirability to be in a state, the value function is an estimate of the long-term desirability based on the possible future states and corresponding reward available. These values are estimated and then re-estimated throughout the life span of the agent. Sutton & Barto [1998] propose a human analogy to explain the difference between a reward function and a value function. They propose that rewards are like pleasure, when high, and pain, when low whereas values correspond to a more farsighted notion of how pleased or displeased we are to be in the current state. The purpose of computing values, which are essentially predictions of reward, is to achieve more reward in the future.

Model: A model of the environment is something that mimics the behaviour of the environment so that the learner can consider future situations before they are actually experienced. For instance, the model may predict a resultant *next state* and *next action* from the current (given) state and action.

2.2.1 Markov Decision Processes

Markov Decision Processes (MDPs) provide a mathematical framework for reasoning about planning and acting in the presence of uncertainty [Bertsekas & Tsitsiklis, 1996; Puterman, 1994]. They are named after Andrei Markov whose research in the early 1900s led to the study of stochastic processes in various different applications [Ching & Ng, 2006]. A state is said to be *Markov*, or have the *Markov property*, if only the current state is required to make predictions about future states. As an example, we could look at the state representation of the configuration of all the pieces on a chess board. Each state is said to be Markov as it summarises all the relevant information that led to being in that state. Even though information pertaining to the sequence of moves that led to that point may be lost, all of the information required for the future of the game has been retained in the state representation. MDPs provide the formal basis for reinforcement learning methodology.

There are many possible formal definitions of MDPs which differ by small transformations of the problem. One example of an MDP, M, can be defined as the four-tuple (S, A, R, P) which consists of:

- S: The set of all possible states in the environment in which the agent can observe.
- A: The set of actions that may be executed at each time step by the agent.
- R: The reward function which provides the immediate reward value, which can be positive or negative, based on the action taken in the state.
- P: The state transition probabilities of the likelihood of transitioning to a particular state after taking an action.

Given a state, s, and a selected action, a, for a time interval, t, the *transition probability* to the next state, s' is written formally as follows:

$$P^a_{s,s'} = Pr\{s_{t+1} = s' | s_t = s, a_t = a\} \tag{2.1}$$

Given a state and action, s and a, and a next state, s', the reward function provides the *expected value* of the next reward and is written formally as:

$$R^a_{s,s'} = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\} \tag{2.2}$$

The quantities of $P^a_{s,s'}$ and $R^a_{s,s'}$ represent the *environment dynamic* and are the most important aspects of the dynamics of a finite MDP [Sutton & Barto, 1998]. The state space must provide enough information to allow predictions to be made about the environment. State abstraction can be used to avoid the "curse of dimensionality " [Bellman, 1957] which refers to the exponential growth of the problem as new states and actions are considered. The state space and action space design has a direct impact on the performance of the learner. Figure 2.2 shows a simplistic finite MDP with two states and two actions.

In this example, the entire state space and transition probabilities are fully observable. The dashed arrows represent stochastic transitions which are based on transition probabilities.

When in State 1, if action A1 is chosen there is an 80% chance that the agent will remain in that state and a 20% chance that it will transition to State 2. If action A2 is chosen then there is a 10% chance that it will remain in the same

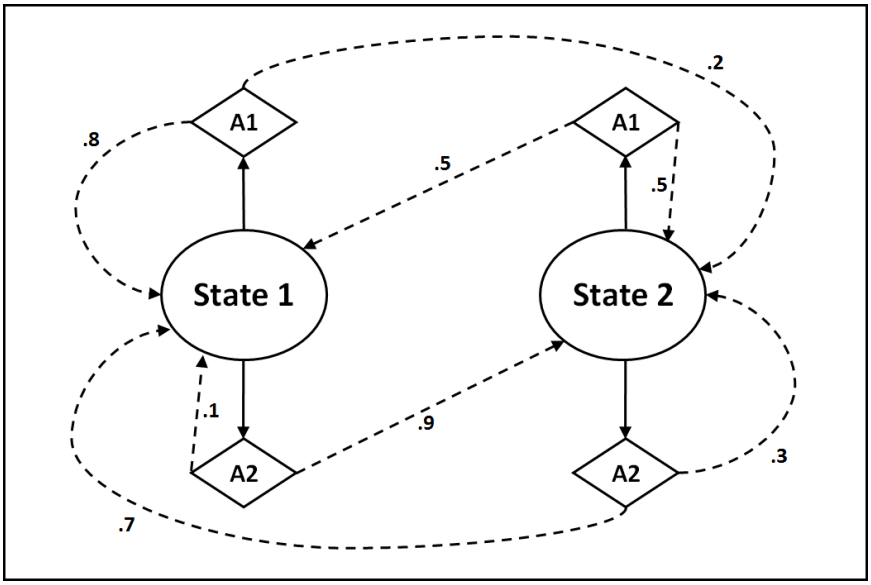

Figure 2.2: An example of a simple finite MDP with two states.

state and a 90% chance that it will transition to State 2. The actions chosen when in State 2 also have associated transition probabilities which represent the likelihood of staying in the same state or moving to the other state.

2.2.2 Dynamic Programming

Dynamic programming (DP), which was first developed by Bellman [1957], involves constructing solutions to sub-problems incrementally from smaller sub-problems and caching them to avoid re-computation. That is, the sub-problems are calculated just *once*, stored in memory and looked up any time they are needed [Russell & Norvig, 2010]. It is similar to the *divide and conquer* algorithm which recursively breaks down a problem into smaller sub-problems and then combines the solutions of the sub-problems to solve the overall problem, however, DP is required to have an *optimal substructure* and be made up of sub-problems that *overlap*. It must be possible to compute the optimal solution to the problem based on the solutions to the sub-problems. The sub-problems must also overlap in which case some sub-problem solutions can be cached and reused. If this overlap does not exist then the approach would be divide and conquer.

DP can be applied to a wide range of problems, see for example Bertsekas [1995], but here we are only concerned with using DP to compute optimal solutions to an MDP. There is an assumption that the learning agent has a perfect model of the environment. There is a guarantee that DP will find the optimal solution to the problem if it has complete knowledge of the environment and the scope of the problem involves evaluating *all* possible solutions. This is often not feasible with real world problems. A thorough discussion of DP algorithms, such as *Policy Iteration* and *Value Iteration*, for computing optimal solutions for an MDP can be found in Bertsekas [1995] and Sutton & Barto [1998].

In this thesis, we are concerned with a branch of reinforcement learning called *temporal difference* learning. In the next section we will discuss this concept and describe some of the associated algorithms.

2.2.3 Temporal Difference Learning

Temporal difference (TD) learning is concerned with learning to predict and using past experience to influence future behaviour. It is based on animal learning psychology and some of the earliest work, related to artificial intelligence, came from Samuel [1959], Michie [1961], Klopf [1972] and Holland [1975]. Sutton [1988] later provided the first formal results and proved the convergence of temporal difference methods as well as comparing them with supervised learning methods.

A TD learning system can learn directly from raw experience without having complete knowledge of the *environment dynamics*. Sutton [1988] states that the core idea of TD learning is to adjust predictions to match other, more accurate, predictions about the future. Updates of estimates are based on the difference between previous estimates and the new estimates, hence the name *temporal difference*. This process is known as bootstrapping[1] and can also be seen in dynamic programming. One of the advantages of TD learning over supervised learning is that the training examples that it uses can be taken directly from the temporal sequence of sensory inputs and thus no special supervisor is required. The calculations for TD learning methods are carried out sequentially at each time step

[1]The current value function estimate is used, in part, to generate the new estimate of the value function.

and therefore require less computational resources and storage than methods that need full task experience before learning can be carried out.

There are many forms of TD learning but each of them follow the same specification as described by Togelius [2007]. They carry out the following steps during each time-step of learning:

1. Calculate values for the set of possible actions based on a *value function*. Values can be assigned based on the current state (*action-value* approach) or by simulating the actions and basing the values on the simulated states (*state-value* approach).

2. Take the action with the highest calculated score (or a different action a small percentage of the time).

3. Receive a reward from the environment for carrying out this action.

4. The value function is updated based on the environment's feedback and the previous estimate of the value.

TD learning controllers can adapt during a single trial and are not required to wait until the trial ends before updating their policy. In the following sections we are going to present the most well-known and widely used TD learning algorithms.

2.2.3.1 Q-learning

The *Q-learning* algorithm was introduced by Watkins [1989] and was later followed by a convergence proof by Watkins & Dayan [1992]. It is a widely used reinforcement learning algorithm that can derive policies, based on *state-action value* estimates, in the absence of a complete model of the environment.

Q-learning is known as an *off-policy* TD control algorithm as it carries out updates to the estimates of the *optimal* action-value function independently of the policy that is currently being followed. The algorithm will carry out updates as if it is following a greedy policy (picking the action with the maximum Q-value) while it is actually following another policy. All that is required for the convergence of the algorithm is that all of the state-action pairs continue to be

Algorithm 1 Pseudocode for the Q-learning algorithm [Sutton & Barto, 1998].

1: Initialise $Q(s,\ a)$ arbitrarily (e.g. set all values to 0)
2: **repeat**
3: Initialise s
4: **repeat**
5: Choose a from s using policy derived from Q (e.g. ϵ-greedy)
6: Take action a, observe r, s'
7: $Q(s,\ a) \Leftarrow Q(s,\ a) + \alpha\ [r + \gamma\ max_{a'}\ Q(s',\ a') - Q(s,\ a)]$
8: $s \Leftarrow s'$
9: **until** (steps of single episode have finished)
10: **until** (all episodes have finished)

updated. Pseudocode for the Q-learning algorithm is shown in Algorithm 1. The algorithm begins with the initialisation of all of the Q-values, for example by setting them all to zero, before any learning starts. These initial values can also be set to values that are higher than any expected reward which will encourage the exploration of actions. The result of this would be that all actions would be selected several times before the value estimates converge. For every episode, a starting state is initialised and for each step of the episode, an action, a, is chosen from the current state, s, based on the action-selection policy being followed. One example of an action-selection policy is *ϵ-greedy* in which the action with the current highest Q-value is selected with probability 1 - ϵ (exploitation of knowledge) and a random action is selected with probability ϵ (exploration). The action is taken with the reward, r, and the successor state, s', being observed. The Q-value is then updated based on the maximum Q-value that can be achieved from the successor state s'. We can see from Line 7 of Algorithm 1 that this is carried out with the following update function:

$$Q(s, a) \Leftarrow Q(s, a) + \alpha[r + \gamma max_{a'} Q(s', a') - Q(s, a)] \tag{2.3}$$

In the update function shown in Equation 2.3, $r + \gamma max_{a'} Q(s', a')$ represents the learned value and is made up of the reward (r), the discount parameter (γ) and the estimate of the optimal future value ($max_{a'} Q(s', a')$). The discount parameter, γ, determines how important future rewards are. The closer the value is to 0, the more the agent will only consider current rewards whereas a value

close to 1 would mean the agent would be more focused on long term rewards. The difference between this learned value and the old value ($Q(s, a)$) is multiplied by the learning rate, α. The α value, which is between 0 and 1, determines how quickly newer information will replace older information. Setting α to 0 will result in the agent not learning anything whereas setting it to 1 will result in the agent only considering the most recent information. The resulting value is then added to the old Q-value estimate in order to give the result for the new estimate. The current state, s, is then updated with the value of the successor state, s'. This process is continued until all the steps of a single episode have finished and the algorithm continues to run until all of the episodes have finished.

2.2.3.2 SARSA

The SARSA[1] algorithm [Rummery & Niranjan, 1994] is an *on-policy* TD control algorithm. Being an on-policy method, the algorithm continually estimates Q-values for a specific behaviour policy while, at the same time, changing toward greediness with respect to the Q-values. The name SARSA comes from the quintuple (s, a, r, s', a') given the fact that the Q-value update function is based on the current state (s), the current action (a), the reward received (r), the successor state (s') and the successor action (a'). The SARSA algorithm learns directly from raw experience without any model of the environment's dynamics. The agent will interact with the environment and update the policy based on the actions that are chosen. This is in contrast to Q-learning which uses the optimal value function for updating the policy (see Line 7 of Algorithm 1). Pseudocode for the SARSA algorithm is shown below in Algorithm 2.

The Q-values are initialised arbitrarily, just as they were in the Q-learning algorithm, and then a starting state is read. An available action from this state is then selected based on the action-selection policy which is derived from the Q-values.

For every step of the episode, an action is taken with a reward and subsequent state being observed. An action is then chosen from this successor state based

[1]The algorithm was originally called "*modified Q-learning*", with the name SARSA first appearing in Sutton [1996].

Algorithm 2 Pseudocode for the SARSA algorithm [Sutton & Barto, 1998].

1: Initialise $Q(s,\ a)$ arbitrarily (e.g. set all values to 0)
2: **repeat**
3: Initialise s
4: Choose a from s using policy derived from Q (e.g. ϵ-greedy)
5: **repeat**
6: Take action a, observe r, s'
7: Choose a' and s' using policy derived from Q (e.g. ϵ-greedy)
8: $Q(s,\ a) \Leftarrow Q(s,\ a) + \alpha\ [r + \gamma Q(s',\ a') - Q(s,\ a)]$
9: $s \Leftarrow s'$
10: $a \Leftarrow a'$
11: **until** (steps of single episode have finished)
12: **until** (all episodes have finished)

on the policy. The Q-values are then updated based on the values from carrying out the successor action, a', while in the successor state, s'. The update function, from Line 8 of Algorithm 2, is as follows:

$$Q(s,\ a) \Leftarrow Q(s,\ a) + \alpha[r + \gamma Q(s',\ a') - Q(s,\ a)] \tag{2.4}$$

From the update function in Equation 2.4 we can see that, similar to Q-learning, the learned value is represented by $r + \gamma Q(s',\ a')$. This time, however, we are not using the estimate of the optimal future value rather we choose a new action from the successor state ($Q(s',\ a')$) using the action-selection policy. SARSA will learn the value of the policy currently being carried out by the agent whereas Q-learning learns the optimal policy of the agent independently of the actions that are being carried out. The remainder of the update function is the same as that of Q-learning. After the update, the current state is assigned the value of the successor state and the value of the current action is assigned the value of the successor action as seen in Line 9 and Line 10 of Algorithm 2. Again, this whole process continues until all the steps of a single episode have finished and the algorithm continues to run until all of the episodes have finished. The stopping conditions for individual episodes and the overall number of episodes will depend on the designer's implementation of the algorithm. Poole & Mackworth [2010] note that SARSA is a useful approach when optimising the value of an agent that is exploring. If one is carrying out offline learning, and using the

resulting policy in an agent that does not explore, Q-learning can be a more suitable approach.

2.2.3.3 SARSA(λ)

An extension to the SARSA algorithm was developed, called SARSA(λ) [Rummery, 1995; Sutton & Barto, 1998], which uses *eligibility traces* [Klopf, 1972] to speed up learning by allowing past actions to benefit from the current reward. An eligibility trace is a temporary record which indicates if a state-action pair has been visited[1] and signifies that it should be affected to some extent by the current reward (either positively or negatively).

Algorithm 3 Pseudocode for the SARSA (λ) algorithm [Sutton & Barto, 1998].

Initialise $Q(s,\ a)$ arbitrarily (e.g. set all values to 0)
Initialise $e(s,\ a) = 0$ for all $s,\ a$
repeat
  Initialise s
  Choose a from s using policy derived from Q (e.g. ϵ-greedy)
  repeat
    Take action a, observe $r,\ s'$
    Choose a' from s' using policy derived from Q (e.g. ϵ-greedy)
    $\delta \Leftarrow r + \gamma Q(s',\ a')$ - $Q(s,\ a)$
    $e(s,\ a) \Leftarrow 1$
    For all $s,\ a$:
      $Q(s,\ a) \Leftarrow Q(s,\ a) + \alpha\delta e(s,\ a)$
      $e(s,\ a) \Leftarrow \gamma\lambda e(s,\ a)$
    $s \Leftarrow s'$
    $a \Leftarrow a'$
  until (steps of single episode have finished)
until (all episodes have finished)

The use of eligibility traces can enable the algorithm to learn sequences of actions which could be beneficial in learning useful policies in FPS games. Eligibility trace values are stored in a separate look-up table ($e(s,a)$). Entries can be marked as eligible in one of the two following ways. These include either set-

[1] Action a has been carried out in state s

ting the trace value to 1 (*replacing trace*) or incrementing the trace value by 1 (*accumulating trace*), if and when the state-action pair has been visited. Sutton & Barto [1998] report that replacing traces, as opposed to accumulating them, can lead to better learning performance. The algorithm with replacing traces, for which the pseudo-code is provided in Algorithm 3, works as follows.

Firstly, the Q-values (set arbitrarily) and eligibility traces (set to 0) for all states and actions are initialised. The initial state, s, is read and then an action, a, is chosen from those available in this state based on the policy. Then, for every step of each episode, the action is carried out and a reward and successor state, s', is read. A successor action, a', is then chosen from this successor state based on the policy. The TD error, δ, is then calculated using the reward (r), discount parameter (γ) and the current and next state-action pairs. The eligibility trace for the current state-action pair is then assigned a value of 1 (replacing trace) to indicate that it is eligible to receive some of the reward. Next, the Q-values and eligibility traces for all states and actions are updated as follows. As we can see from Line 12 of Algorithm 3 each Q-value is updated by adding the new estimate, $\alpha\delta e(s,\, a)$, which consists of the learning rate, the TD error and the eligibility trace value for the current state and action, to it. If the state-action pair is not eligible for learning ($e(s,\, a) = 0$) then the Q-value will remain unchanged. Each eligibility trace value is also decayed using the discount parameter (γ), the eligibility trace parameter (λ) and the old value ($e(s,\, a)$). Again, this update will have no effect when $e(s,\, a) = 0$. Once this has completed, the current state, s, is set to the successor state, s', and the current action, a, is set to the successor action, a', as seen in Line 14 and Line 15. A comparison of the learning steps in the SARSA and SARSA(λ) algorithms, on a simple maze environment, is shown later in Figure 5.1 of Section 5.2.1.

2.3 Traditional Game AI and Machine Learning Techniques

The following sections provide summaries of the most prevalent traditional game AI techniques and also provide an overview of machine learning techniques that have been combined with reinforcement learning.

2.3.1 Traditional Approaches to Game AI

In this section we describe some of the traditional approaches to game AI that are used to create "intelligent" non-player characters in computer games. All of these techniques are well understood and have been used by game developers for many years. The purpose of this section is to give a general summary of some of the better known and often used techniques. The reader should see Champandard [2003], Millington & Funge [2009] or Kirby [2011] for a more thorough analysis of these and other game AI approaches.

2.3.1.1 Hard-coded Behaviour and Scripting

Hard-coding is the most straightforward way of implementing AI in computer games. If the hard-coded design fits the particular situation well, then it can produce fast, intelligent behaviour. If the code is not appropriate, however, then the resulting behaviour can be very poor and even lead to a disruption to a players suspension of disbelief [Kirby, 2011]. A simple example would be to have a list of conditional checks that have corresponding behaviours associated with them. For instance, if an agent in a FPS game has very little health left, it should only concentrate on finding health items. A human player may conversely risk engaging in combat depending on the situation. Hard-coding gives the programmer full low level control of the agent's behaviour. *Scripting* involves a further abstraction of hard-coded behaviours which are grouped together into specific tasks. Scripted actions can often become predictable and human players can exploit this weakness. The size and complexity of hard-coded and scripted behaviour could be seen as a drawback, especially for non-trivial game AI. It can become difficult to manage and update code bases as the code size and complexity grows.

2.3.1.2 Decision Trees

Decision trees, which were formulated by Schlaifer & Raiffa [1961] based on the work of Von Newmann & Oskar [1944], are tree-like directed graphs which are made up of *nodes* which are decision points. They involve starting at a single *root* node with the decision-maker traversing through the tree, based on the answers at each decision point, until they reach an outermost (*leaf*) node at which point the action to take has been decided.

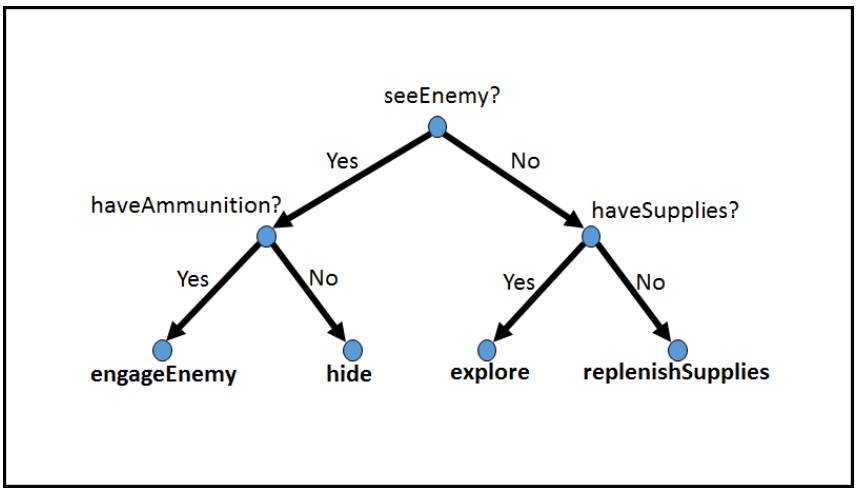

Figure 2.3: An example of a simple high-level decision tree for a first-person shooter NPC.

Figure 2.3 shows a simple high-level decision tree for a first-person shooter computer-controlled character. The first check, at the root of the tree, is whether or not an opponent is visible. If the opponent is visible and the character has ammunition in its weapon then it will engage in combat with the opponent. Otherwise, it will flee from combat and hide. If the character cannot see the opponent then it will check to see if it has sufficient supplies (health and ammunition). If the levels are sufficient it will continue to explore new areas of the map but if it lacks supplies then it will concentrate on finding them. The leaf nodes of this tree comprise scripted behaviour with a set of hard-coded rules to achieve the desired effect. A basic decision tree such as this will always produce the same

behaviour given the same circumstances. Decision trees are easy to understand and implement. They also have a fast execution time as the number of decisions considered (following a path in the tree) is a lot smaller than the total number of decisions in the tree.

2.3.1.3 Finite State Machines

A *finite state machine* (FSM), or *finite state automaton* as it is also called, is composed of a set of finite states and a set of transitions between these states which are predefined. The states are usually represented by descriptive words that summarise them such as *walking*, *sitting* and *driving*.

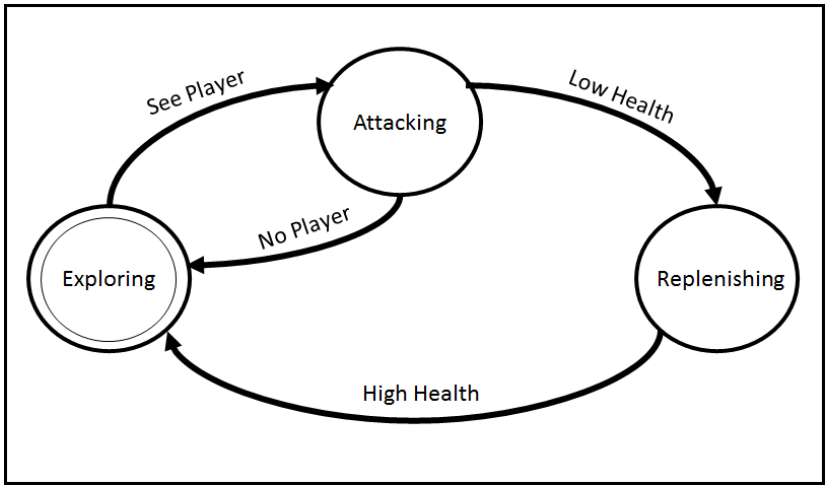

Figure 2.4: An example of a simple FSM for a first-person shooter NPC.

Some states cannot transition to others and the transitions are initiated by either the internal state of the agent or by a trigger from the environment. Figure 2.4 shows an example of an FSM for controlling an NPC in an FPS game. Each of the states are represented by circles with the arrows between them representing the transitions from one state to another. The starting state for the NPC is signified by the double circle around the "exploring" state. The text beside each transition indicates what caused the transition. In this example, the NPC will transition from exploring the environment, if it sees an opposing player, to an

attacking state. If its health drops below a threshold it will transition to a state in which it concentrates on replenishing its health. If no opposing player is visible then the NPC will revert back to the starting state of exploring providing that it has sufficient health.

2.3.1.4 Rule-Based Systems

Rule-based systems consist of a database of knowledge available to the AI and a mechanism for enforcing rules to make use of this knowledge. This usually consists of a series of "if-then" rules in which, if a condition is matched from the database, the specified rule will be executed.

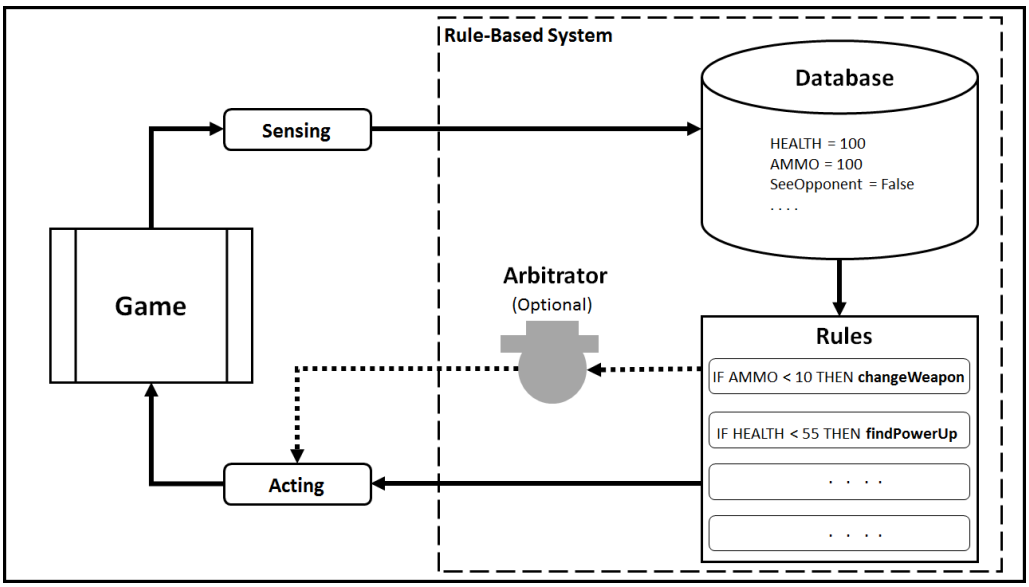

Figure 2.5: An illustration of a simple rule-based system.

This type of system can also include an *arbiter* which decides which action is chosen from each of the triggered rules. An illustration of a simple rule-based system is shown in Figure 2.5. Kirby [2011] notes that rule-based systems aim to bring out the best qualities of hard-coded AI and avoid constraining the game developer to partition the problem into a series of independent states such as those that make up an FSM. When the agent is making a decision, it reads the

current state of the world from the database and checks this information against the rules to see if they match. One or more of the matching rules can then be activated. It can be difficult to create rules that are representative of every situation and human expertise is often required in order to design these rules.

2.3.1.5 Fuzzy Logic

Fuzzy set theory involves the use of fuzzy sets whose elements have a *degree of membership* as opposed to their membership being either true or false. *Fuzzy logic* involves the use of logical expressions for describing the membership in fuzzy sets [Russell & Norvig, 2010]. Truth values of between 0 and 1 are calculated to represent the degree of membership of each element.

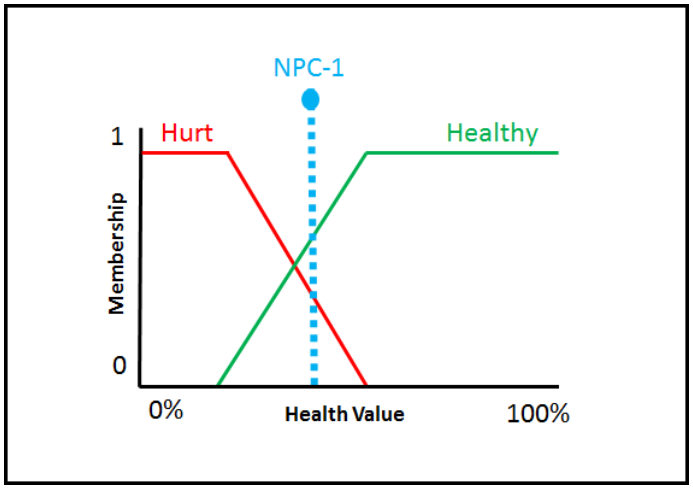

Figure 2.6: Numerical fuzzification of hit points in a first-person shooter NPC.

The term *fuzzification* is used for turning regular data into degrees of membership. Figure 2.6 shows the numerical fuzzification of the number of damage points for an NPC into the fuzzy sets of *hurt* and *healthy*. In this example, NPC-1 is 0.6 healthy and 0.4 hurt. Many different membership functions can rely on the same input data. It can be very complicated to manually produce fuzzy logic for the complex interactions of all the values that make up a computer game agent. Millington & Funge [2009] note that, while fuzzy logic has been relatively popular

in the games industry, it has largely been discredited as an effective technique by the academic AI community. The authors propose that this is due to the fact that it is always better to use probability for representing uncertainty. Further discussion of this can be found in Russell & Norvig [2010].

2.3.1.6 Blackboard Architectures

Blackboard architectures can be used for the coordination of actions from different decision maker sources. The blackboard system is made up of three different components which are: the decision making tools (called *experts*); the arbiter; and the blackboard. The architecture is illustrated, using the example of a first-person shooter NPC, in Figure 2.7.

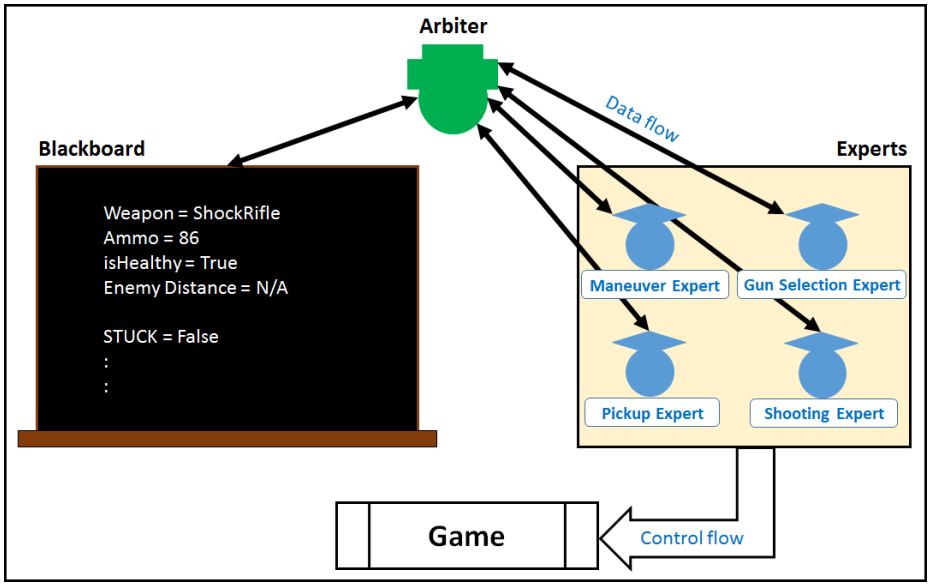

Figure 2.7: Illustration of a Blackboard Architecture.

The *blackboard* is a common knowledge base area of memory that each of the experts can read from and write to. The *arbiter* decides which expert currently has control of the blackboard. Each *expert* has to have a mechanism for alerting the arbiter when they require access. When they are granted access they can

then read from the blackboard, perform some logical computations and delete or write new values as required.

Blackboard architectures generally have a bad reputation among game AI developers due to the large code base, the required management code and the complicated data structures that they incorporate [Millington & Funge, 2009].

2.3.2 Machine Learning Technique Summaries

Section 2.4 and Section 2.5 provide a review of the literature which includes various different machine learning techniques that are both combined with reinforcement learning and applied to the logic of non-player characters in first-person shooter games. These techniques are summarised below to supplement the discussion in these sections.

2.3.2.1 Transfer Learning

Transfer learning (TL) involves transferring the knowledge acquired from the learning experience in one domain to a new, and similar, domain with a view to improving the learning performance. For instance, there are many real world applications in which it is expensive or even impossible to collect new training data for rebuilding learning models. Transfer learning can be used to reduce the need and effort to recollect such training data by transferring knowledge between the task domains. Pan & Yang [2010] present a survey of transfer learning and provide a discussion on the relationship between transfer learning and other related machine learning techniques. Taylor & Stone [2009] present a survey on the use of transfer learning methods applied to reinforcement learning tasks.

2.3.2.2 Case Based Reasoning

Case-based reasoning (CBR) is a problem solving paradigm in which a new problem is solved by using knowledge from a previously experienced *case*. Aamodt & Plaza [1994] note that the approach provides incremental learning since all of the experience is retained for each problem and can be immediately re-used for new problems. The authors also describe the CBR cycle as a four-step process:

1. *Retrieve*: Retrieve the memory cases relevant to solving the target problem.
2. *Reuse*: Adapt the solution from the previous case to the new situation.
3. *Revise*: Test the new solution on the target problem and revise if necessary.
4. *Retain*: Store the new case in memory once it has been successfully adapted.

The theory behind CBR is heavily based on the early work of Roger Schank [1983] (which was later revised and extended in Schank [1999]). Further reading on CBR can be found in Kolodner [2014].

2.3.2.3 Evolutionary Algorithms

Evolutionary Algorithms are algorithms that perform optimisation or learning tasks by using mechanisms inspired by biological evolution such as *selection*, *mutation* and *reproduction*. They have three main characteristics, as outlined by Yu & Gen [2010]:

- Population-based: A group of solutions are maintained, called a *population*, and these are used to optimise or learn the task at hand.
- Fitness-oriented: Each solution in the population is called an *individual* and each individual has a performance evaluation metric associated with it called a *fitness value*.
- Variation-driven: Traits from individuals in a population are combined in various ways to mimic the biological effect of genetic gene changes.

Moriarty *et al.* [1999] provide an overview of the application of evolutionary algorithms to reinforcement learning. Further reading on evolutionary algorithms can be found in Mitchell [1998] and Freitas [2013].

2.3.2.4 Neural Networks

The term "neural network" was traditionally used to refer to a network of biological neurons found in the human brain. A *neuron* is a cell in the brain that has the functionality of collecting, processing and sending electrical signals. In the field

of machine learning, the term refers to artificial neural networks (ANN) which are made up of artificial neurons or nodes. Two main types of neural network are *feed-forward* (acyclic) and *recurrent* (cyclic). Feed-forward neural networks can be either single layer, where the inputs are directly connected to the outputs, or multilayer where there are one or more hidden layers present. The information in a feed-forward neural network moves only in one direction from the input nodes to the output nodes as shown in Figure 2.8. Recurrent neural networks, on the other hand, have bi-directional data flow. Further reading on neural networks can be found in Russell & Norvig [2010].

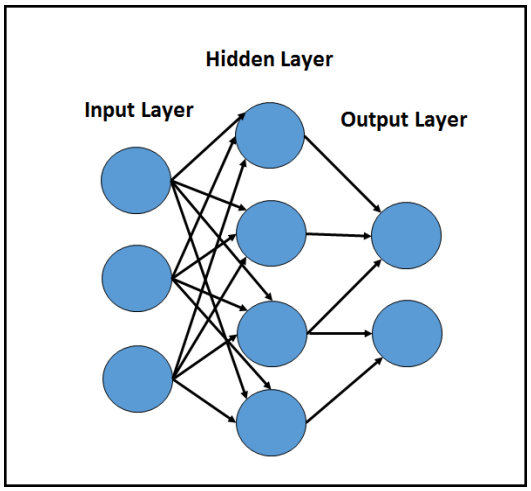

Figure 2.8: An example of a simple feed-forward neural network.

2.3.2.5 Support Vector Machines

A Support Vector Machine (SVM) [Vapnik, 1995, 2013] is a supervised learning algorithm used for classification or regression analysis. The following summary is based on a summary from this authors previous work in Glavin [2009]. There are two separate cases, one in which the data to be classified are linearly separable and the other in which the data cannot presently be linearly separated. In the linear case, the best separating hyperplane (maximum margin function) of the training samples that linearly separates the two classes is found as shown in

Figure 2.9.

The widest possible separating margin leads to maximal generalisation. The data vectors on each side of the hyperplane are called the *support vectors*. The hyperplane, used to classify new unknown samples, is a quadratic programming optimisation problem. When we deal with completely non-linear data we must transform the data into a higher dimensional space in order to achieve the linear separation.

This transformation or mapping of the data is achieved by using some function $\phi(\mathrm{x})$ as shown in Figure 2.10. Each sample is mapped to the new feature space and then the separating hyperplane can be computed. This process can be computationally expensive but it is possible to carry it out in an efficient manner when using what is known as the *Kernel Trick*. This method uses a kernel function to calculate the dot product in the new feature space as opposed to mapping each individual sample to a new set of features and then calculating the dot product. The separating hyperplane in the new feature space can then be calculated without explicitly mapping the data. Further reading on SVMs can be found in Russell & Norvig [2010].

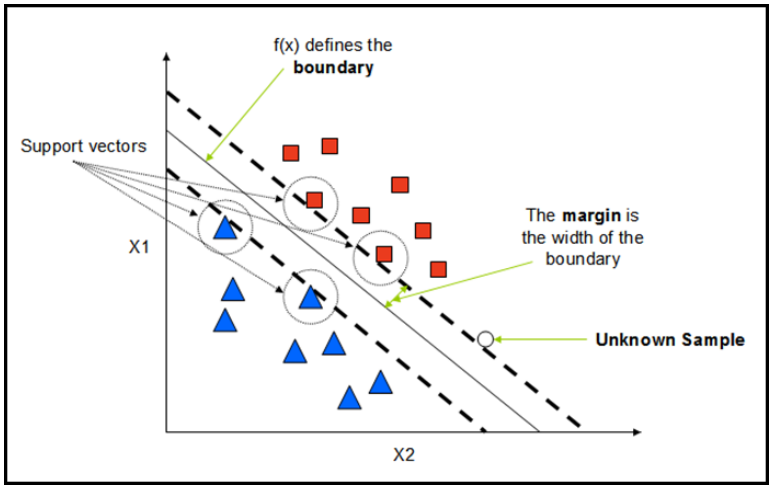

Figure 2.9: An example of a linear Support Vector Machine. [Glavin, 2009]

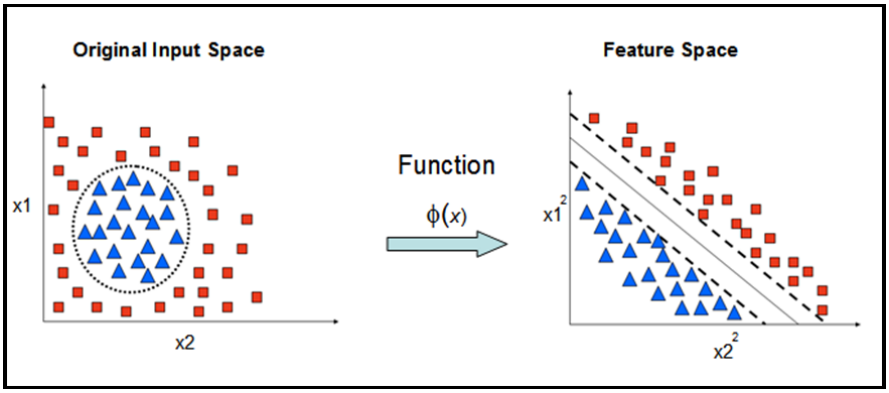

Figure 2.10: Mapping data to a higher dimensional feature space. [Glavin, 2009]

2.3.2.6 k-Nearest Neighbour

The k-Nearest Neighbour (kNN) classification algorithm is amongst the simplest of all machine learning algorithms and is an instance-based learning approach for approximating real-valued or discrete-valued target functions [Mitchell, 1997].

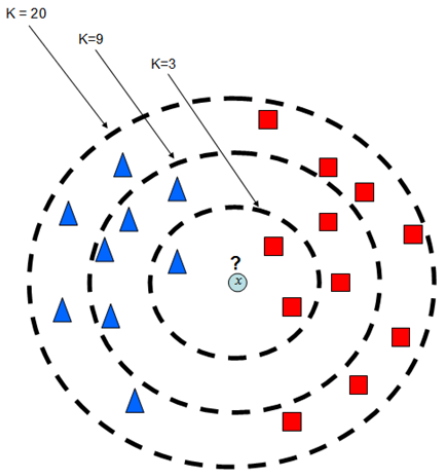

Figure 2.11: Classifying an example using the k-Nearest Neighbour algorithm. [Glavin, 2009]

An assumption is made that all instances correspond to points from an n-dimensional space. Training data for the algorithm are simply stored and when a new instance is presented, the "nearest" or most similar instances from the

neighbourhood are retrieved and used for classifying this new instance. The similarity between instances is measured using a distance metric such as *Euclidean* distance. An unknown instance, x, is classified by assigning the most common value of the k neighbours in the training set that are closest to it. This is known as a majority vote. The amount of neighbours to take into consideration is obviously a very important parameter. We can observe from Figure 2.11 that when k = 3, the test instance x would be classified as a red square. However, when k = 9, the majority vote would classify x as a blue triangle. Further reading on the k-Nearest Neighbour algorithm can be found in Mitchell [1997].

2.4 Applications of Reinforcement Learning

Reinforcement learning has been successfully used in a wide variety of diverse domains. In the following subsections we are going to present notable applications of reinforcement learning from the literature. Specifically, we will discuss reinforcement learning as applied to robotics and simulations; frameworks and utilities; and games and benchmark problems. We will delay our discussion of first-person shooter games, the domain of which this thesis is concerned with, until Section 2.5 in which it will be discussed in detail. In some of the following cases, reinforcement learning is combined with other machine learning techniques. These were summarised in the previous section in order to supplement the discussion.

2.4.1 Robotics and Simulations

In this section, we present some applications of reinforcement learning algorithms to physical robots and vehicles as well as computer simulations of these types of tasks. Robotics toolkits and testbeds that are available will be discussed later in Section 3.1.1.

Mahadevan & Connell [1992] presented an approach for automatically programming a behaviour-based robot called *OBELIX*. The authors describe two learning algorithms which combine Q-learning [Watkins & Dayan, 1992] with structural credit assignment techniques based on *statistical clustering* and *Hamming distance*. Statistical clustering propagates rewards across states based on a set of clusters for the actions, each of which defines the utility of taking that action in a specific class of states. The Hamming distance algorithm propagates rewards to other similar states based on the similarity metric of a weighted Hamming distance between states. The experimentation involved the use of both a simulated robot testbed and a physical wheeled robot for the task of moving boxes. The authors reported that the robot could learn to perform at a similar level to that of a human-programmed solution.

Madden & Nolan [1995] presented an adaptive controller, called *Icarus*, which used an extended version of the Q-learning algorithm [Watkins & Dayan, 1992]. The algorithm generated reactive control strategies in the application of controlling a vehicle in a simulated traffic environment. The task that was considered

by the authors was that of acceleration control. Experimentation was carried out which showed that Icarus successfully discovered an optimal strategy for the control task. The extension to the algorithm involved the selection of actions in new states being biased by previous experience. The use of the past experience bias resulted in an improved convergence to the optimal strategy.

Zhang & Dietterich [1995] applied the temporal difference algorithm TD(λ) [Sutton, 1988] to the application of space shuttle payload processing for NASA. This domain involves scheduling the tasks that must be carried out for installing and testing the payloads that are placed in the cargo bay of a space shuttle. The TD(λ) algorithm is applied to train a neural network (Section 2.3.2.4) for learning a heuristic evaluation function over the states. The goal with the scheduling tasks is to satisfy a set of constraints while also minimising the total duration of the schedule. The experimentation carried out showed that the proposed method out-performed the previous best algorithms for this task.

Ng *et al.* [2004] applied reinforcement learning to the autonomous flight of a model helicopter. They modelled the dynamics of the helicopter by allowing a human "pilot[1]" to control it and record the states and actions of the helicopter as it was flown. The pilot flew the helicopter for several minutes in which the 12-dimensional states and 4-dimensional control inputs were recorded. The main tool used for fitting was locally weighted linear regression. This model was then used to learn how to hover in place and perform a number of different manoeuvres. The authors used the *Pegasus* (Policy Evaluation-of-Goodness And Search Using Scenarios) reinforcement learning algorithm [Ng & Jordan, 2000] for carrying out the reinforcement learning phase. The Pegasus algorithm uses the same seeded random numbers to evaluate any policy using a simulator of the Markov Decision Processes dynamics. Standard search heuristics can then be used for finding the best performing policy. Further details on the Pegasus algorithm can be found in Ng [2003]. The learned policy was compared to that of the human pilot and it was shown to be able to fly the helicopter in a more stable manner. Ng *et al.* [2006] also applied reinforcement learning to a controller for the sustained autonomous *inverted* flight of a model helicopter.

Carreras *et al.* [2005] presented a hybrid behaviour-based scheme that uses re-

[1] A human that flies the model helicopter using a remote control.

inforcement learning to control Autonomous Underwater Vehicles (AUVs). The approach of the authors involved hybrid behaviour coordination of high level AUV control. They also proposed a new learning algorithm called *semi-online neural Q-learning* (SONQL) for learning the state-action mappings of the behaviours. The proposed algorithm used the Q-learning algorithm [Watkins & Dayan, 1992] with a neural network (Section 2.3.2.4) function approximator. Experiments were carried out on two AUVs for a target following task. Initial tests were performed with two behaviours (*tracking* and *recovery*) to demonstrate the feasibility of the SONQL algorithm. This was followed by additional tests with four different behaviours: *tracking*, *repulsion*, *recovery* and *tele-operation*. The results showed that the SONQL algorithm and the hybrid behaviour coordinator could help to increase the autonomy of the AUVs.

Taylor *et al.* [2008] introduced *Modelling Approximate State Transitions by Exploiting Regression* (MASTER) which is a method for automatically mapping the learning from one task to another similar one in a reinforcement learning problem. In the experimentation, the authors first used the SARSA(λ) algorithm [Sutton & Barto, 1998] to train the learner on the 2D Mountain Car[1] [Moore & Hall, 1990] problem and then learned an inter-task mapping to a 3D Mountain Car problem. The results of the experiments showed that the use of MASTER increased the speed of learning with the new target task.

Torrey *et al.* [2008] proposed to extend reinforcement learning by including the abilities of advice taking and transfer learning (Section 2.3.2.1). The implementation in this case was a variant of the SARSA(λ) algorithm [Sutton & Barto, 1998] that used a Support Vector Machine (Section 2.3.2.5) for function approximation. Advice was given to the algorithm in the form of a set of soft constraints on the Q-function. If certain conditions were met then these were automatically assigned the highest Q-values. Transfer learning was carried out using three different algorithms proposed by the authors. These were *Skill Transfer*, *Macro Transfer* and *Markov Logic Network Transfer*. Experimentation was carried out on the simulated RoboCup [Kitano *et al.*, 1997] tasks of *KeepAway*, *Breakaway* and *MoveDownField* with results showing that the process of learning could be sped up significantly.

[1] The Mountain Car task is also described in Sutton & Barto [1998] on page 214.

Kartoun *et al.* [2010] developed a new reinforcement learning algorithm which enables collaborative learning between a robot and a human. The algorithm, called CQ(λ), is based on the Q(λ)-learning algorithm [Peng & Williams, 1996] and looks to accelerate the learning process by making use of human knowledge and expertise. The CQ(λ) algorithm allows the robot to switch its collaboration level from autonomous to semi-autonomous (incorporating human guidance) based on a self-test of its learning performance. The algorithm is applied to a bag-shaking task in which a robot arm must shake a bag to release a tied knot and thus release the contents of the bag. The role of the human advisor (HA) is to provide guidance to the robot during policy selection. A digital scale is positioned under the robot and the reward function is automatically updated when an item, falling from the bag, is detected. As a result of this, the robot can sense when the performance is low and the HA can intervene with advice. The authors carried out a comparative evaluation of the CQ(λ) algorithm with the Q(λ)-learning algorithm for the bag-shaking task. The results showed that the CQ(λ) algorithm demonstrated a lower average time to extract the objects as well as having a higher reward level than that of the Q(λ)-learning algorithm. The authors, however, identified the issue of detecting whether or not the advice of the HA is beneficial to the robot. They proposed to introduce a "back up procedure" for the robot so that inadequate human suggestions could be erased and autonomous mode could be reinstated.

Barrett *et al.* [2010] demonstrated that the use of transfer learning (Section 2.3.2.1) could significantly speed up and improve the performance of reinforcement learning done entirely on a grounded robot. The authors note that working with grounded robots has a variety of difficulties such as environment or sensor noise, expensive failures (robot damage) and the time taken to perform tasks. In this case, the authors are using the SARSA(λ) algorithm [Sutton & Barto, 1998]. Experimentation was carried out involving a Sony *Nao* robot. The robot was placed in a sitting position and its task was to hit a ball as far as it could in a 45 degree angle with its right hand. The robot calculated its own reward signal based on the observations. The results showed that transfer learning could speed up the process of carrying out reinforcement learning on the physical robot.

Taylor *et al.* [2011] introduced *Human Agent Transfer* (HAT), which is an al-

gorithm that combines reinforcement learning, learning from demonstration and transfer learning (Section 2.3.2.1). The HAT algorithm consists of three steps: *demonstration*, *policy summarisation* and *independent learning*. For demonstration, the task is carried out by the agent being controlled by a human teacher with the agent recording all of the state-action transitions. The policy summarisation step involves using the state-action data from the demonstration step and deriving rules that summarise the policy. The agent uses reinforcement learning to carry out independent learning which is biased by the policy summary. Experimentation was carried out on the *KeepAway* task in the simulation of RoboCup [Kitano *et al.*, 1997] soccer. The results showed that even with just a few minutes of human demonstration the algorithm could increase the learning rate of the task by several hours.

Mülling *et al.* [2013] presented a framework for enabling a robot to learn table tennis based on its interactions with a human player. The authors developed a movement library for the robot from kinesthetic teach-in and imitation learning. Kober & Peters [2009] describe kinesthetic teach-in as "taking the robot by the hand", that is, performing the required task by moving the robot and recording the joint angles, velocities and accelerations that take place. The robot selects actions using their *Mixture of Motor Primitives* (MoMP) algorithm. They carried out experimentation using both a simulation and a physical robot which showed that the approach improved the performance of the robot enabling it to successfully play against a human opponent.

Kober *et al.* [2013] presented a comprehensive survey of the use of reinforcement learning for behaviour generation in robotics. The authors identified some of the key challenges in the domain and discussed some of the important contributions and success stories over the years. This work also included a discussion of some of the open questions in robotics reinforcement learning.

2.4.2 Frameworks and Utilities

This section describes a selection of frameworks and utilities that were developed for applying reinforcement learning. Various different toolkits and testbeds for carrying out experimentation are described later in Chapter 3.

Nason & Laird [2005] developed an extension to the Soar [Laird *et al.*, 1987] architecture, called *Soar-RL*, to enable it to carry out reinforcement learning. The authors describe Soar as a "*general cognitive architecture for developing systems that exhibit intelligent behavior*". The architecture has had many different versions since its initial creation in 1983[1]. The authors report that adding the reinforcement learning capability is achieved in two phases. Firstly, *numeric preference* values for existing rules are adjusted. Then, new rules are created that test different feature sets while creating numeric preferences. A numeric preference for an operator in the Soar architecture is generated by a rule which associates a numeric value with a particular set of features in working memory [Nason & Laird, 2005]. The authors carried out experimentation on two puzzle problems, namely, the *Missionaries and Cannibals* and *Eaters* puzzle and were able to demonstrate successful learning capabilities with the extended architecture.

Torrey *et al.* [2005] presented a framework for transferring knowledge (Section 2.3.2.1) from one reinforcement learning task to another through the use of advice-taking mechanisms. The authors discussed the importance of transferring knowledge in complex domains such as RoboCup [Kitano *et al.*, 1997] soccer. The framework that was presented involved a human providing advice to the learner and a mapping from old features and actions to ones in the new task in a RoboCup simulation. The authors discuss some of the application details of transferring knowledge from the subtask of *KeepAway* in RoboCup to the task of *BreakAway*.

Knox & Stone [2009] introduced the framework of *Training an Agent Manually via Evaluative Reinforcement* (TAMER) for allowing a human to interact with and shape an agent's policy through reinforcement signals. The authors define the process of shaping as interactively training an agent through signals of positive and negative reinforcement. Experiments using the TAMER framework were carried out on the classic game of Tetris and the Mountain Car [Moore & Hall, 1990] problem. The results suggested that the TAMER agent out performed an autonomous agent in the short term after very few trials. However, well-tuned autonomous agents were better at maximising peak performance after extensive trials. The authors proposed that future work would involve identifying how to

[1] http://soar.eecs.umich.edu

best use both human reinforcement and environment reward signals. The authors also suggested that it may be worthwhile to put more emphasis on human reinforcement at the beginning of learning and then the agent could be fine-tuned by switching the focus to environmental rewards at a later stage.

Lin & Wright [2010] presented an approach of automatically deriving state abstractions called *Evolutionary Tile Coding* (EvoTC) by using a genetic algorithm based on the work of Holland [1992]. This significantly reduces the state space so that reinforcement learning can be applied effectively. As with other adaptive tile-coding approaches, EvoTC begins with a single tile representing the entire state space. Splits are then introduced to increase the details of the abstraction. EvoTC uses an evolutionary algorithm (Section 2.3.2.3) to decide when and where the splits should take place. A comparative evaluation was carried out against the *Cerebellar Model Articulation Controller* (CMAC) [Miller *et al.*, 1990] and *Adaptive Tile Coding* (ATC) [Whiteson *et al.*, 2007] on two benchmark reinforcement learning problems. These were the Mountain Car [Moore & Hall, 1990] and Pole Balancing [Michie & Chambers, 1968] problems. The results showed that the EvoTC algorithm was able to create state abstractions more effectively on both of the problems.

Mehta *et al.* [2011] described an approach for automatically inducing task hierarchies from source problems and then transferring (Section 2.3.2.1) these hierarchies to target problems that have the same causal structure in their system dynamics. The authors' algorithm, called *Hierarchy Induction via Models and Trajectories* (HI-MAT), discovers the hierarchical structure of a problem by recursively partitioning the *Causally Annotated Trajectory* (CAT). The CAT is the original trajectory annotated with all of the causal edges between dummy start and end actions. Experimentation was carried out on the resource-gathering task in the game *Wargus*. This is a Real-Time Strategy (RTS) game in which players must gather items, build structures and battle against the opposing players. The experiments involved comparing three different approaches which were a non-hierarchical Q-learning [Watkins & Dayan, 1992] algorithm, a hierarchical RL algorithm called *MAXQ-0* with a manually created hierarchy and MAXQ-0 applied to the HI-MAT induced hierarchy. The results showed that the HI-HAT induced hierarchies performed comparably to the manually created ones and pro-

vided a more effective transfer than just a direct transfer of the value function.

Chali *et al.* [2011] described a reinforcement learning framework for a *complex question answering problem* using an extractive summarisation approach. The most important sentences were selected by using a modified linear gradient decent version of Watkin's Q(λ)-learning algorithm[1] [Watkins, 1989]. The task, as described in the Document Understanding Conference 2007, was as follows: "*Given a complex question (topic description) and a collection of relevant documents, the task is to synthesise a fluent, well-organised 250-word summary of the documents that answers the question(s) in the topic*". An ϵ-greedy policy was used by the authors during the training phase to balance between exploration and exploitation. A comparative evaluation was carried out using the proposed method against a Support Vector Machine (SVM) (Section 2.3.2.5) and a baseline method. The results showed that the reinforcement learning method was an effective approach which produced similar or improved results compared to the other two.

2.4.3 Games and Benchmark Problems

This section outlines some examples of reinforcement learning that were applied specifically to computer games and standard benchmark problems. This section does not include first-person shooter games as these will be discussed separately in Section 2.5.

Graepel *et al.* [2004] used the SARSA [Rummery & Niranjan, 1994] algorithm with a linear action-value function approximator to enable an agent to learn how to play the fighting game *Tao Feng*. The game involves two opponents taking part in a martial arts fight in a 3D environment. The characters that can be selected have one of 12 different fighting styles with over 100 actions available to the players. The authors designed states, such as "*opponent's physical state*" and "distance to opponent", and atomic actions such as *kick*, *punch* and *throw*. The authors also created actions by combining these atomic actions together in sequences and actions were categorised into *agreesive*, *defensive* and *neutral*. Two

[1] This should not be confused with Peng's Q(λ)-learning algorithm of the same name [Peng & Williams, 1996]

types of reward were defined, one of which encourages aggressive fighting and the other of which encourages a more "peaceful approach". The agent only receives a reward when the subsequent action has been successfully selected (which indicates that the previous action has completed). The authors carried out experimentation against the game's built-in AI. The results showed that good policies could be learnt which produced "interesting behaviour" and could exploit weaknesses with the built-in AI. The authors identified the importance of timing when fighting which is not currently captured by their learning agent.

Tan [2004] proposed an extension to a self-organising neural network architecture called FALCON (Fusion Architecture for Learning COgnition and Navigation). This is a cognitive model which allows an agent to interact and adapt in a dynamic environment. The system involves learning cognitive codes that associate states with actions that lead to a desirable outcome based on the reward signal. Experimentation was carried out on a mine-field navigation task. This was a 16×16 cell mine-field grid that contained 10 mines. Trials consisted of cycles of *sense*, *move* and *learn*. A trial ends when the agent reaches the goal, hits a mine or exceeds 30 cycles. The agents achieved almost a hundred percent success rate after a thousand trials.

Runarsson & Lucas [2005] carried out a comparative analysis between TD learning, using self-play gradient-descent, and co-evolutionary learning (CEL), using an evolution strategy, in the task of position evaluation for a small-board (5×5) game of *Go*. The standard grid size of Go is 19×19 but, given the rules, it can be played on a grid of any size. Players take turns placing stones on the board and the objective of the game is to have surrounded a larger total area of the board than their opponent by the end of the game. The authors carried out experimentation using the *GNUGo* 3.4[1] engine and reported that the TD learning approach learned at a much faster rate than the CEL approach, however, CEL was able to outperform TD learning when given enough time.

Ponsen *et al.* [2006] used hierarchical reinforcement learning (HRL) to learn a policy of navigation for a *worker unit* in the Real-Time Strategy (RTS) game Battle of Survival (BoS). RTS games involve players taking control of a civilisation, building structures and using military force to defeat other opposing civilisations

[1] http://www.gnu.org/software/gnugo/

in real-time. Worker units have to move to goal locations and therefore require effective navigation and the ability to avoid enemies. HRL involves breaking down complex tasks into simpler subtasks which can then be solved independently. Learning can also facilitate generalisation in which knowledge learned from one subtask can be transferred to other subtasks. A general overview of work in HRL can be found in Barto & Mahadevan [2003]. The authors carried out a comparative analysis of solving the navigation between both a *flat* and *hierarchical* representation of the task with the hierarchical representation decomposing the task into subtasks that solve single sub-goals independently. The HRL algorithm used is based on Hierarchical Semi-Markov Q-learning (HSMQ) [Dietterich, 2000]. The results showed that using a hierarchical representation led to reduced complexity and produced much better results than the flat RL algorithm.

Lucas & Togelius [2007] compared the use of TD learning (Section 2.2.3) and an evolutionary approach (Section 2.3.2.3) for training a neural network (Section 2.3.2.4) as a controller in a simple point-to-point car racing game. These were also compared with hand-coded strategy controllers. The task involved visiting as many way-points, in order, as possible within a given number of time steps. The points are initialised randomly and only the next three way-points are visible to the car at any time. Experimentation was carried out using different feature vectors and neural network variants for the tasks of *action-value* learning and *state-value* learning. The authors reported that evolution achieved a better final fitness and was more reliable than the reinforcement learning approach and that state-value learning produced much better results in each case.

Banerjee & Stone [2007] presented a game player agent which used reinforcement learning and transfer learning (Section 2.3.2.1) when interacting with a *General Game Playing* system [Pell, 1993] in order to transfer knowledge gained from one game and speed up the learning process in many other games. A game-tree look ahead structure was used to represent the features of the game. Experimentation was carried out in which the feature values were extracted using the game of *Tic-tac-toe* as the source. Then, the transfer learner was tested on three different target games which were *connect3*, *CaptureGo* and *Othello*. In all of the experiments, the learning speeds of a baseline learner were compared

to the Transfer Learner using the feature knowledge acquired from Tic-tac-toe. Three different types of opponent were tested against which were *ϵ-greedy*, *random* and *weak*. The results showed that game-independent features can be used for transferring state value information to provide better performance than *look ahead minimax* or *fixed heuristic* searches.

Burrow & Lucas [2009] carried out a comparative analysis on the use TD learning (Section 2.2.3) and an evolutionary algorithm (EA) (Section 2.3.2.3) for controlling a character in *Ms. Pac-Man*[1]. The authors use the TD(0) [Sutton, 1988] algorithm and an EA, with a population size of 30. This was EA(15+15) in which half of the population are kept and mutated versions of these then replace the remaining half. Further information on EA can be found in Beyer & Schwefel [2002]. A *state-value* controller was used in all of the experiments with two different function approximators being tested. These were a four-layer *interpolated table* and a Multi-Layer Perceptron (MLP). The results showed that evolving the MLP, using EA, outperformed the TD learning approach. The interpolated table was shown to be more reliable than the MLP approach with MLP only matching the performance of the interpolated table under certain reward schemes.

Sharifi [2010] used ScriptEase [McNaughton *et al.*, 2004] and the SARSA(λ) [Sutton & Barto, 1998] reinforcement learning algorithm to derive *appropriate behaviours* for companion NPCs that accompany the player character (PC) in Neverwinter Nights[2] (NWN). The author defines appropriate behaviour as "rapidly adaptive and human-like". The author carried out experimentation with different styles of PC (*independent*, *rogue*, *selfish*, *cautious*) and reported that the NPC could adapt its behaviour based on feedback from the PC which resulted in behaviour that was more natural and human-like.

Loiacono *et al.* [2010b] applied the Q-learning [Watkins & Dayan, 1992] algorithm to a Behavior Analysis and Training (BAT) [Dorigo & Colombetti, 1998] architecture to develop an overtaking policy for a controller in The Open Racing Car Simulator (TORCS[3]). The focus of the research was on two overtaking behaviours which were overtaking an opponent either on a straight stretch or a

[1] This game, and its associated competition, are described later in Section 3.1.5.

[2] Neverwinter Nights is a third-person role-playing game that was developed by BioWare and published by Atari.

[3] http://torcs.sourceforge.net

large bend; and overtaking an opponent on a tight bend. The authors carried out experimentation against the most advanced (best performing game AI) NPC that ships with the game. The authors reported that that the Q-learning BAT controller could produce overtaking behaviours that outperformed the strategy of the most advanced TORC NPC.

Amato & Shani [2010] applied reinforcement learning techniques to generate high-level behaviours in Civilisation IV which is a complex real-time strategy game. The authors followed the assumption that they were playing against a fixed-strategy opponent and three different learners were evaluated. These were Q-learning [Watkins & Dayan, 1992], Dyna-Q [Sutton, 1991] and Dyna-Q with factored state representations. The focus of the research was to select an effective strategy based on the current conditions of the game. The state space is defined as having four state features, namely, *population difference*, *land difference*, *military power difference* and *remaining land*. The results of the experimentation showed that reinforcement learning approaches outperformed the random and hand-coded fixed policies for the game.

Pena *et al.* [2012] presented two variants of the hybrid learning algorithms *WEREWoLF* and *WERESARSA* to create controllers in the "hand-to-hand combat" virtual environment of *vBattle* [Pena *et al.*, 2009]. These algorithms combine evolutionary techniques with reinforcement learning. The evolutionary techniques that are used to combine the reinforcement learners are *Differential Evolution* Algorithm (DE) [Storn & Price, 1997] and *Estimation of Distribution Algorithms* (EDA) [Larranaga & Lozano, 2002; Mühlenbein & Paass, 1996]. The authors carried out experimentation to compare the performance of the evolved controllers against several hard-coded opponents. The combination of the evolutionary and reinforcement learning techniques outperform the use of standard reinforcement learning. The authors emphasised the importance of the environment variables and reward function for providing enough information for the agent to guide the learning process.

Mnih *et al.* [2015] developed an artificial agent, called *deep Q-network* (DQN), which uses a deep neural network [Bengio, 2009] and reinforcement learning to learn control policies from different environments with minimal prior knowledge about the tasks. Specifically, DQN is based on the Q-learning algorithm [Watkins

& Dayan, 1992] and uses a Convolutional Neural Network (CNN) to learn a problem-specific representation and estimate a value function. The authors note that the successful integration of the reinforcement learning algorithm with the deep network architectures was dependent on the use of a replay algorithm which facilitated the storage and representation of recently experienced transitions. The agent was tested using the Arcade Learning Environment (ALE) [Bellemare *et al.*, 2013] on 49 different Atari 2600 games, receiving only the game pixels and score as inputs, and managed to outperform all previous algorithms and achieve a level of play comparable with professional human testers.

This work uses high-dimensional sensory inputs to learn successful policies in a wide range of classic games. In contrast, our work is concerned with reading important features of the environment from the system in a game that has multiple objectives, partial observability and includes multiple agents interacting in an adversarial setting.

There has been some critique of this work. For example, Liang *et al.* [2015] comment that since Mnih *et al.* [2015] only report a single independent trial per game, this affects the reproducibility of the experiments and makes it difficult to compare to other methods. They also argue that without an adequate baseline for what is achievable using simpler techniques, the cost to benefit ratios of more complex methods like DQN are harder to evaluate.

2.5 First Person Shooter Games

Since our research is primarily concerned with the control of an NPC in an FPS game, we are going to discuss some existing research from this domain in the following sections. Firstly, we briefly describe the Bot Prize competition which has been running for the last number of years. This is followed by a review of game AI techniques which include the use of reinforcement learning. Finally, we summarise other existing game AI approaches which have been applied to this genre of computer game.

2.5.1 Bot Prize Competition

In recent years, a competition was set up for testing the humanness of computer controlled bots in FPS games. This is called *Bot Prize* and the original competition took place on December 2008 as part of the IEEE Symposium on Computational Intelligence and Games[1]. The purpose of the competition was to see if computer controlled bots could fool expert judges into believing that they were human players in the FPS game Unreal Tournament 2004. This competition essentially acts as a Turing Test [Turing, 1950] (Section 2.1.1) for bots. The overall goal of the competition is to fool judges into believing that a bot is human at least 50 percent of the time. The original design [Hingston, 2009] of the competition involved a judge playing against two other opponents (one human and one bot) in 10 minute Deathmatch games. At the end of the game, the judge had to rank the opponents from 1 to 5 as follows:

1. This player is a not very human-like bot.
2. This player is a fairly human-like bot.
3. This player is a quite human-like bot.
4. This play is a very human-like bot.
5. This player is human.

The authors later reported that they had improved the design [Hingston, 2010] which made the process of judging part of the game. An existing weapon in UT2004 called the Link Gun was modified and used by players to judge other players. The primary mode of the gun should be used on players that are judged to be bots. Successfully identifying a bot by using the primary mode of the gun will kill the bot instantly and reward the shooter 10 points (as opposed to the regular 1 point per kill). However, if the player uses the primary mode of the gun on a human player then the shooter will die immediately and lose 10 points. The exact opposite of these rules apply when the gun is being fired in secondary mode where correctly identifying a human is rewarded but mistaking a bot for a human

[1] http://botprize.org/2008

is penalised. The judging powers of the gun are only enabled exactly once on each opponent. Shooting this gun at the same opponent more than once will have no effect after the first time. The Bot Prize competition ran for five years before finally being won by two teams in 2012. MirrorBot (52.2%) [Polceanu, 2013] and the UT^2 bot (51.9%) [Schrum *et al.*, 2011] surpassed the "humanness barrier" of 50%. These two winning entries are described later at the end of Section 2.5.3.

There was a break from the Bot Prize competition in 2013 but it returned as part of the 2014 IEEE Conference on Computational Intelligence and Games. A new method for assessing humanness was also introduced. The humanness ratio is now calculated using both *First-Person Assessment* (FPA) and *Third-Person Assessment* (TPA). The FPA is calculated using the in-game judging system and the TPA is calculated by external observers of video footage. A combination of both assessments make up the final humanness ratio.

2.5.2 Reinforcement Learning Approaches

Smith *et al.* [2007] developed an algorithm called RETALIATE (Reinforced Tactic Learning In Agent-Team Environments) for Unreal Tournament. The authors used the Q-learning algorithm [Watkins & Dayan, 1992] for learning winning policies in the *Domination* game type. This game type involves taking control of focal points on the map by positioning players next to them in order to earn points. The work that was carried out by the authors was concerned with co-ordinating the team behaviour, as opposed to learning behaviours of the individual players. Experimentation was carried out against three different teams with varying strategies. The results showed that the algorithm adapted well to the changing environments.

Hefny *et al.* [2008] created *Cereberus* which is a machine learning framework for the *Capture the Flag* (CTF) game type in first-person shooter games. This game type involves retrieving the opponent's flag from their base and returning it to your base to earn points. The framework uses neural networks (Section 2.3.2.4) to control the fighting behaviour of the bot. Separate feed-forward neural networks were developed for aiming, shooting and moving. Reinforcement learning

was used to select the high-level actions for each team member. Experimentation was carried out in three phases. Firstly, two static teams were played against each other to see if there was any bias based on the map design (positioning of team flags). Secondly, an adaptive opponent that begins with a defensive strategy was played against a static opponent. Finally, the policy derived by the adaptive team was used to play against the static team choosing only the best actions (greedy selection). The results showed that the adaptive team could modify its behaviour to beat the static team despite the flag positioning on the map being biased towards the static team.

Wang *et al.* [2009] proposed the use of FALCON [Tan, 2004], described in Section 2.4.3, for developing a computer-controlled agent in Unreal Tournament 2004. The authors built two FALCON networks, one for weapon selection and one for behaviour selection. The bot learned by using cognitive nodes which could be translated into rules by associating a state and a particular action with an estimated reward. The bots created these rules and learned how to play the game in real-time. The bot was tested by entering it into the Bot Prize competition in which the bot competed in order to convince the human judges that they were human by engaging in human-like behaviour. While the proposed bot did not win the competition, it did receive the highest game score and managed to fool some of the human judges.

McPartland & Gallagher [2011] applied the tabular SARSA(λ) [Sutton & Barto, 1998] reinforcement learning algorithm to a purpose built first-person shooter game. This research extended their previous work in this domain [McPartland & Gallagher, 2008a,b]. The algorithm was used to learn the controllers of navigation, item collection and combat individually. The authors developed a purpose built game as opposed to using a commercial game in order to reduce the processor overhead and increase the throughput of experiments. The experimentation that was carried out involved three different setups of the RL algorithm, namely, *HierarchicalRL*, *RuleBasedRL* and *RL*. HierarchicalRL used the controllers for navigation and combat by learning when to use which one based on the reward signal. The RuleBasedRL uses the navigation and combat controllers also but has predefined rules on when to use each. The RL setup learns the entire task of navigation and combat together by itself. The results showed that

reinforcement learning could be successfully applied to a simplified purpose-built FPS game. McPartland & Gallagher [2012b] extended their research by developing an interactive training tool in which human users can direct the policy of the learning algorithm. The bot follows its own policy unless otherwise directed by the user. They also investigated the outcome of having five commercial game developers use the interactive tool to train bots [McPartland & Gallagher, 2012a]. They concluded from their experiments that the training could produce bots with different behaviour styles in the simplified environment. The developers reported that the training tool had potential for use in FPS game development and they also identified several improvements that could be made.

Tastan & Sukthankar [2011] proposed an Inverse Reinforcement Learning (IRL) approach for teaching FPS bots to play through expert human demonstrations. The experiment testbed, which was constructed with the Pogamut toolkit (Section 3.2), acquires data from expert human players playing Deathmatch games. This data is then used to learn policies for *attack*, *exploration* and *targeting* opponents. IRL recovers rewards from policies by computing a candidate reward function that could have resulted in the demonstrated policy. This is in contrast to regular reinforcement learning which uses the reward signal to learn policies. In this approach, the most frequent actions executed by human players during the demonstrations are recorded. The reward model is then learned offline using a CPLEX solver based on a set of constraints extracted from the expert demonstration. Value Iteration is then used to create a policy from the reward vector which can be executed by the bots. The resulting bots were evaluated by comparing them against standard Pogamut (Section 3.2) example bots. Human judges were also used to evaluate the human-likeness of the bots by playing against them in Deathmatches and observing them in videos. The developed bots were rated more human-like than the Pogamut example bots.

Patel *et al.* [2011] examined the use of Q-learning [Watkins & Dayan, 1992] for evolving basic behaviours in a scaled down abstraction of the FPS game *Counter Strike* (Section 3.1.3). The bot evolves policies for learning how to fight (engaging with the enemy or ignoring them) and learning how to plant the bomb in the game. The evolved bot was evaluated against a static scripted opponent whom it was shown to outperform. The authors state that further tests would be needed

using the actual Counter Strike game and an evaluation against human players would be more beneficial.

Goyal & Pasquier [2011] developed a bot for Unreal Tournament 2004 which uses behaviour trees and the Q-learning [Watkins & Dayan, 1992] reinforcement learning algorithm. The bot was designed to carry out the core tasks of *Navigation* and *Shooting*. The behaviour tree for the Navigation task is made up of *Look for items*, *Look for players* and *Navigate randomly*. The Shooting task behaviour tree checks opponent range, weapon selection, shooting and finding cover. The Q-learning algorithm was only used for learning to shoot at the opponent. The state space was made up of five states which were discretised values for the bot's health. The action space consisted of six actions which included variations of shooting at the opponent. The authors ran short experiments in the Deathmatch game type against a single human opponent on two small maps. The results showed that even with the small size of the state-action space (30) many of of the state-action pairs were never visited. This would suggest that there were some flaws in the design of the states and actions for the shooting task.

Wang & Tan [2015] used a self-organising neural network that performs reinforcement learning, called FALCON [Tan, 2004], to control NPCs in Unreal Tournament 2004. The bot, called *FALCONBot*, employed two reinforcement learning networks to learn both behaviour modeling and weapon selection. Experimentation was carried out against two sample Pogamut (Section 3.2) bots, *advancedBot* and *hunterBot*, and showed that FALCONBot could learn weapon-selection to the same standard as the hard-coded expert human knowledge. The bot was also shown to be able to adapt to new opponents on new maps if its previously learned knowledge was retained. The implementation used the inbuilt shooting command for combat, with random deviations added to the direction of the shooting, in an effort for it to appear human-like.

2.5.3 Other Artificial Intelligence Approaches

Laird & Duchi [2000] examined some of the factors that make a bot more human-like by running a series of experiments with their *Soar QuakeBot* [van Lent & Laird, 1999] architecture. This bot was developed for the game *Quake 2* and is

based on the *Soar* [Laird *et al.*, 1987] architecture (Section 2.4.2). All of the internal knowledge for the bot to play the game is encoded in Soar rules. There are 715 rules that are based on the bot sensors and motor actions. The Quake 2 game engine updates the world ten times a second. The Soar architecture runs asynchronously to this executing its decision cycle 30 to 50 times a second. Further information on the Soar QuakeBot can be found in Laird [2001]. The authors set up several Deathmatch games with the Soar QuakeBot and they recorded videos from the first-person perspective of the bot. Video recordings were also taken of five human players of varying skill levels. A panel of eight judges were then asked to view the recordings and make judgments on the humanness and skill levels of all of the players, both human and computer-controlled. The results showed that *decision time* and *aiming skill* were reported as being the critical factors in making bots appear human-like. The other factors that were tested, which were deemed not to be critical, were *aggressiveness* and *number of tactics*.

Cole *et al.* [2004] proposed the use of a genetic algorithm (Section 2.3.2.3) to evolve parameter values for NPC bots in the FPS game *Counter Strike* (Section 3.1.3). This research is focussed on the game type called *defuse mission* in which teams of players are tasked with preventing their opponents from planting and detonating a bomb at specified locations on the map. The game is won by successfully defusing the bomb or killing all players on the opposing team. The authors decided that the parameters to tune would be *weapon selection* and *aggressivity*. They then evolved these parameters before deploying the resulting bots against hard-coded bots with expert knowledge. The results show that the bots with the evolved parameters could play at the same level as the bots tuned with expert knowledge. This indicated that the algorithm could tune the bot's parameters as well as those that were manually tuned by a human. The authors reported that such an approach could reduce game development time and no longer require the programmer to be an expert in the game's strategy.

Thurau *et al.* [2004] used a Neural Gas [Martinetz *et al.*, 1993] algorithm to create a grid way-point map for generating human-like movement in a 3D gaming environment. The authors report that this is essentially an improved k-Means algorithm that has shown good performance for learning topologies in the past [Martinetz & Schulten, 1991]. Quake 2 was used as the environment and the

authors trained the agent using observation data of actual human player movements in the game. Potential field forces were learned for the agent to guide its movement patterns in the environment. Experimentation showed that the agent was successful in imitating human movements from the training data.

Priesterjahn *et al.* [2007] presented an approach to evolve artificial players in the game *Quake 3*. An evolutionary algorithm (Section 2.3.2.3) was used by the agents to learn the input and output rules. A comparative analysis against the hard-coded bots, provided by Quake 3, showed that the evolutionary method was able to out-perform these on all of the difficulty settings.

Auslander *et al.* [2008] developed an agent called *CBRetaliate* in Unreal Tournament 2004. This work aimed to enhance the RETALIATE algorithm [Smith *et al.*, 2007], described in 2.5.2, by introducing the use of Case-Based Reasoning (CBR) (Section 2.3.2.2). The algorithm partitions the space into regions with associated Q-tables for each region. The authors report that the agent has two key features which are that it uses a time window to compute similarity and stores (and can reuse) Q-tables for continuous problem solving. Relevant Q-tables can be loaded if similar circumstances are encountered. They carried out experimentation on the Domination game type and the results showed that the use of CBR could speed up the adaption process to changing opponent tactics.

Soni & Hingston [2008] developed bots for Unreal Tournament 2004 by training neural networks (Section 2.3.2.4) using recorded actions from games involving human players. A feed-forward and a recurrent neural network were developed to create two individual bots. These bots and the hard-coded fixed-strategy native bots from the game were then tested against human opposition and the human players filled out a survey after the games were completed. The results from the survey showed that the human players found the bots that were controlled by the neural networks to be more human-like, less predictable and more challenging than their hard-coded counterparts. A correlation analysis of the answers also showed that the players' perceptions of the bots were highly correlated with each other.

Hirono & Thawonmas [2009] developed an Unreal Tournament 2004 bot called *ICE* which uses Finite State Automaton, described earlier in Section 2.3.1.3, that has two states. These states are the *item-collection* state and the *battle* state.

The bot enters the battle state when an opponent is in sight and switches to the item-collection state when any of the following conditions are met: kills current opponent, loses sight of opponent and has low ammunition/health, cannot find opponent at last seen location. The bot came in second place in the 2008 Bot Prize competition although the authors note that there was a significant difference between the human-likeness ratings of the bots compared to the human players. After the competition, the bot was improved based on five important factors of human-likeness identified by the authors. These were items acquisition, attacking method, weapon change, combat against multiple opponents and combat movement locus. Experimentation was carried out by emulating the conditions of the Bot Prize competition and analysing positive and negative comments about the bots from human observers. Two out of the five factors, *item acquisition* and *attacking method*, were reported as improved based on an evaluation of human judge comments yet some further problems were also identified.

Arrabales *et al.* [2009] developed a consciousness-based cognitive architecture called CERA-CRANIUM for controlling a FPS NPC which is based on *Global Workspace Theory* (GWT) [Baars, 1988] and the *Multiple Draft Model* (MDM) [Dennett, 1991]. It was implemented using two main software components: CERA, which is a control architecture that is structured in layers, and CRANIUM, which creates and manages parallel processes in shared workspaces. Perception processes are generated by CERA using the services provided by CRANIUM. Initial experimentation was carried against rule-based and Q-learning [Watkins & Dayan, 1992] bots in Deathmatch games. While the bot performed well, the authors noted that the current implementation produced poor decision-making on many occasions due to a lack of complex learning abilities.

Cothran & Champandard [2009] described the implementation that resulted in winning the 2009 Bot Prize competition. The winning bot was called *sqlitebot* and it used a sqlite database to both track the locations of kill/death events (hot spots) and store the cross-visibility of *NavPoints* (Section 3.3.3.5) for the use of evasive behaviour. The bot was an extension to the *AdvancedBot* from the Pogamut (Section 3.2) example bots. When the bot kills another player, or is killed by another player, the location of the incident is recorded in the database and can be used later when the bot is looking for a target location. For evasive

behaviour, the bot always chooses a *NavPoint* that is currently outside of the enemy's visibility set when its health reaches a minimum threshold. This *NavPoint* information is populated in a pre-game bot map survey step which logs all of the required information. Further additions included adding random fuzziness to combat decisions and delaying reaction times to incoming projectiles by 200 milliseconds.

Petrakis & Tefas [2010] proposed the use of a feed-forward neural network (Section 2.3.2.4), trained with back propagation, for weapon selection in the game Unreal Tournament 2004. In order for the bot to select the best weapon available it needs to be able to predict the approximate damage that each of the weapons could do based on the current situation. Nine neural networks were created for different weapons in the game. Each one had an input layer of three neurons which were the *distance* from the enemy, the *angle* between them and the *velocity* of the enemy. There were 50 neurons in the hidden layer and one output layer which represented the estimated damage of the weapon. Experimentation was carried out against native bots with varying degrees of difficulty and the authors reported a major improvement in the bot score compared to random weapon selection.

Pao *et al.* [2010] proposed a method for detecting game bots based on dissimilarity measurements between the trajectories of bots and human users. The authors observed a large amount of dissimilarity between the data traces of the trajectories of bots and humans. With certain human behaviour patterns being difficult to mimic, this can be used as a signature to detect bots. The approach uses a dimensionality reduction technique called *Isomap* [Tenenbaum *et al.*, 2000] to find appropriate space for the trajectory representation and then uses a supervised classification algorithm (Smooth Support Vector Machine (Section 2.3.2.5) or k-Nearest Neighbour (Section 2.3.2.6)) to make the decision. An evaluation was carried out by conducting a case study on the game Quake 2. The results showed that a high level of detection accuracy could be achieved when using a relatively small trace of just a few hundred seconds.

Fernández & O'Valle [2011] described two decision tree-based approaches for controlling the behaviour of bots in Unreal Tournament 2004. Decision trees were discussed earlier in Section 2.3.1.2. The first uses a traditional scripting process

where the strategy is manually coded according to the programmer's expertise with an aim to maximise player satisfaction. The second approach uses evolutionary programming (Section 2.3.2.3) to automatically generate the game AI. Both of these approaches were tested against the *HunterBot* which is an example bot that is packaged with the Pogamut IDE (Section 3.2) and an evaluation was carried out based on the opinions of five spectator judges. Five rounds were observed and the evolved bot was judged to be more human-like 16 times of out the 25 (5 rounds with 5 judgements). The authors conceded that this evaluation was subjective and emphasised the need for further work in evaluating the human-likeness of computer controlled characters.

Hindriks *et al.* [2011] proposed connecting the GOAL agent programming language, which programs rational cognitive agents, to the game Unreal Tournament 2004 to facilitate what they call "Unreal Goal Bots". These are *Belief-Desire-Intention* (BDI) [Rao *et al.*, 1995] agents. BDI is a software model developed for programming intelligent agents. It separates the activity of selecting a plan (from a plan library) from the execution of the current plan that is running. In this sense the agents can balance the time spent deliberating about plans with actually executing them. The creation of the plans is a separate task which is carried out by the system designer. The authors developed an interface to manage the agent-bot interaction using the Pogamut toolkit (Section 3.2). They encountered several issues, the most important of which they state are efficiency and scalability.

Cardamone *et al.* [2011] addressed the problem of automatically designing maps for FPS games in which evolutionary algorithms (Section 2.3.2.3) are applied to optimise a fitness function which is based on the players fighting time. The authors state that the goal of their work is to evolve maps that will potentially lead to more interesting gameplay. In order to run the game simulations for the experimentation, *way-point* generating and acceleration of the game speed had to be addressed. Human designers usually place way-points[1] on the map manually so the authors had to implement an algorithm for automatic way-point generation. Firstly, all free cells on the map were identified and then a way-point

[1]Coordinates used for computer controlled bots to navigate the map (Also called *Nav-Points*).

was placed on every free cell. These were then connected, where possible, and all the resources from the map were aligned onto this grid of way-points. The game was sped up by disabling graphical rendering which resulted in 10 minute game simulations taking place in 10 seconds. The results showed that playable FPS maps could be evolved but the authors note that the fitness function depended on the control logic of the default bots in the game. Another drawback mentioned was user studies were not carried out to validate the efficiency of the fitness function. The authors proposed that these issues would be addressed in future work.

Fountas *et al.* [2011] developed a system which uses a *global workspace architecture* implemented in spiking neurons to control a bot in Unreal Tournament 2004. Global workspace architecture, which was first proposed by Baars [1988], is a model of consciousness. Separate processes that are working in parallel can be coordinated to produce a single stream of control. Each of these separate parallel processes compete with each other to place their information in the global workspace. Background information on spiking neurons can be found in Maass [1997]. The system, developed by the authors, is designed to create a bot that produces human-like behaviour and is based on control circuit theories of the brain. The authors report that it is the first system of this type to be deployed in a dynamic real-time environment. The bot interfaces with the UT2004 game environment and implements wall-following and shooting modules using spiking neural networks run on the NeMo [Fidjeland & Shanahan, 2010] simulator. The bot was being developed ahead of the Bot Prize competition held as part of the Computational Intelligence in Games conference in 2011.

Thawonmas *et al.* [2011] developed a bot for the 2011 Bot Prize competition called *ICE-CIG2011* with a view to having human-like tactic selection and the ability to judge opponents as being either computer or human-controlled. The approach used *Neuro Evolution of Augmenting Topologies* (NEAT) [Stanley & Miikkulainen, 2002] with a combination of Neural Gas [Martinetz & Schulten, 1991; Martinetz *et al.*, 1993] and the k-Nearest Neighbour (Section 2.3.2.6) algorithm for learning tactic selection from a human player that acts as a trainer. Judging opponents is based on the closest match of an unknown opponent to one that is known. Majority voting is carried out using the k-Nearest Neighbour

algorithm. Experimentation was carried out using the same conditions as the Bot Prize competition which showed that ICE-CIG2011 had promising human-likeness and its judging ability was superior to human players in some cases.

Acampora *et al.* [2012] developed a cross-platform architecture which takes into account emotions, personality and dynamic action-selection with a view to improve the NPC human likeness in computer games. This is achieved by merging theories from psychology, namely the emotional models OCC [Ortony *et al.*, 1990] and OCEAN [McCrae & John, 1998], with *Timed Automata based Fuzzy Controllers* (TAFCs) [Acampora, 2010]. OCC and OCEAN are used for modelling the personality and emotions whereas the TAFCs are used as a decision making system which analyses the environment and emotional state in order to select suitable actions. The approach was developed using the *Fuzzy Markup Language* (FML), a hardware independent programming language, to facilitate the design and implementation of NPC bots using their "emotional engine". This merging of time, emotions and personality is called *timed emotional intelligence*. Experimentation was carried using Unreal Tournament 2004 where the developed bot played in Capture the Flag games against a native fixed-strategy bot. Video footage of the gameplay was recorded as well as gameplay in which humans were playing. Human judges were then asked to rate how human-like they believed the players in each video were. Preliminary results, based on the judges' votes were reported to show "a good level of human-likeness".

Schrum *et al.* [2011] developed a bot for Unreal Tournament 2004, called UT^2, which uses *multiobjective neuroevoloution* to learn skilled combat behaviour and filters the available combat actions to promote human-like behaviour. The bot can also take advantage of a database of human navigation traces to assist in getting the bot unstuck if its navigation fails. The behaviour-based architecture uses a list of behaviour modules which are ordered by priority. The most high priority module is to become unstuck. Other modules include acquiring items, observing opponents, chasing and engaging opponents. The *combat* controller uses neuroevolution which evolves artificial neural networks (Section 2.3.2.4), where the fitness function is designed to encourage human-like traits in gameplay. For its shooting strategy, the bot shoots at the location of the opponent with some random added noise. The amount of noise added is dependent on the distance

from the opponent and its relative velocity with more noise being added as the distance and relative velocity values increase. The *observe* module was added so that the bot could appear more human-like while judging opponents or groups of opponents. The implementation was an updated version of an entry to the Bot Prize competition ahead of Bot Prize 2011. Full development details and an analysis of the bot's performance in Bot Prize can be found in a chapter written by Schrum *et al.* [2012].

MirrorBot, developed by Polceanu [2013], records opponents' movements in real-time and if it encounters what it perceives to be a non-violent player it will trigger a special behaviour of *mirroring*. The bot then proceeds to mimic the opponent by playing back the recorded actions after a short delay. The actions are not played back exactly as recorded to give the impression that they are being independently selected by the bot. MirrorBot has an aiming module to adjust the bot's orientation to a given focus location. If the opponent is moving then a "future" location will be calculated based on the opponents velocity and this will be used to target the opponent. In the absence of a target, MirrorBot will focus on a point computed from a linear interpolation of the next two navigation points. The authors do not report any weapon-specific aiming so it is assumed that this aiming module is used for all guns despite the large variance in how different guns shoot. The decision on which weapon to use is based on its efficiency and the amount of available ammunition.

2.6 Discussion

We believe that the task of adversarial combat for NPCs, as found in the first-person shooter genre of computer games, provides an interesting challenge for reinforcement learner agents. The stochastic environment, which can be populated by both friendly and opposing players, requires real-time decision-making with instantaneous consequences. Even in the simplest of game types, such as Deathmatch, the agent still has many different parallel objectives. For instance, the agent must navigate the environment, find and kill opposing players, maintain levels of health and ammunition, pick up the most powerful weapons and take

evasive action to stay alive. As we have seen from the previous sections, many researchers have focused on evolving good NPC behaviour over many generations and then deploying the final learned behaviour in a static deterministic NPC. Our research, on the other hand, is concerned with continuous in-game, reactive play in which the NPC will adapt its strategy and always strive to receive as much reward as possible. While some work has been carried out in the literature, as seen earlier in Section 2.5.2, in which both Q-learning and SARSA(λ) have been applied to first-person shooter games, we believe that there is a large scope for improvement in how the agent interacts with the environment and the overall design of a reinforcement learning behavioural architecture.

2.7 Chapter Summary

In this chapter we introduced the concept of AI which included providing a definition, a brief look at its history and a discussion of some of the notable success stories throughout the years. This provides background information for the reader in order to put our research into perspective. We then discussed traditional approaches to game AI which summarised various approaches which have been used in commercial game AI deployment over the years. This highlighted the static and predictable nature of these "tried and tested" techniques. This was followed by providing summaries for a selection of machine learning techniques which are commonly used in combination with reinforcement learning, the topic of which is the core technique used in this thesis and which was introduced in the subsequent section. We outlined the underlying theories and presented the existing algorithms which are most relevant to our research. Applications of reinforcement learning from the literature, from different domains, were then presented to give the reader an impression of its broad applicability. We then focused on the first-person shooter genre of computer game which is the domain in which we address in this thesis. We discussed the application of reinforcement learning algorithms and other artificial intelligence approaches to this genre of game and concluded the chapter with a brief discussion.

Chapter 3

Artificial Intelligence Testbeds

This chapter begins by describing some of the software packages and testbeds that are available for carrying out research using AI. Specifically, we focus on software toolkits and testbeds that can be used to evaluate reinforcement learning techniques. This is followed by a discussion of why the Integrated Development Environment (IDE), Pogamut 3, and the computer game Unreal Tournament 2004 were chosen for carrying out this research. Finally, both the IDE and the game are then described in detail.

3.1 Software Toolkits and Testbeds

There are many different software packages and testbeds, both freely available and commercial that can be used for developing and deploying reinforcement learning agents. These include, but are not limited to, robots being trained to carry out tasks, software packages including various algorithms and evaluation methods, commercial and open-source computer games, board game simulations, purpose-built environments and AI competitions. We will now take a closer look at these categories.

3.1.1 Robotics and RoboCup

Robotics and AI have been the subject matter of many science fiction novels and films over the years. These days, intelligent robots are not limited to fiction and

have applications in many different domains including military [Nath & Levinson, 2014], education [Danahy *et al.*, 2014], the service industry [Bajcsy, 2014] and manufacturing [Bryzek *et al.*, 2006]. Earlier, in Section 2.4.1, we reviewed some of the applications of reinforcement learning to robotics. Reinforcement learning can enable a robot to learn how to behave based on trial-and-error interactions with its environment. Kober *et al.* [2013] present a survey for exploring the use of reinforcement learning for behaviour generation in robotics. They list some of the success stories in this domain and go on to describe some of the challenges associated with the use of reinforcement learning in robotics such as the curse of dimensionality [Bellman, 1957] and the authors also note that "*samples can be expensive due to the long execution time of complete tasks, required manual interventions, and the need for maintenance and repair*".

Researchers and students wishing to delve into the world of robotics can build and program a fully functional robot using LEGO Mindstorms [Cuéllar & Pegalajar, 2014]. These include a small computer control system, a set of modular sensors and a set of motors. The robots are built to the user's specification using LEGO parts and can be as complex or as simplistic as required. The MindStorm website[1] regularly runs competitions with different themes to promote design innovation in different areas. This is just one example of many programmable robotics kits that are available on the market today [Dragos, 2013].

Another interesting initiative for robotics is the *RoboCup* competition [Kitano *et al.*, 1997] which was founded in 1997 with the goal of "developing by 2050 a Robot Soccer team capable of winning against the human team champion of the FIFA World Cup". The competition, which is run yearly, has since evolved into a robotics competition for several different categories including disaster rescue, human-robot interaction in the home, robots in work-related scenarios and a competition for young students called *RoboCupJunior*. The core competition is still the soccer tournament which promotes research into cooperative multi-robot systems that are fully autonomous. Developers can test their algorithms on a simulation of the competition or the decision logic can be loaded onto a physical robot which has multiple sensors. Stone *et al.* [2006] provide a code repository with a set of tools and tutorials for researchers to use the *Keepaway* soccer do-

[1] http://mindstorms.lego.com

main as a benchmark. The Keepaway task is a sub-problem of the RoboCup tournament in which one team must retain possession of the football from another team while operating within a limited region. The goal of the opposing team is to intercept the ball. Figure 3.1 below shows an example of Sony Nao robots taking part in the RoboCup competition.

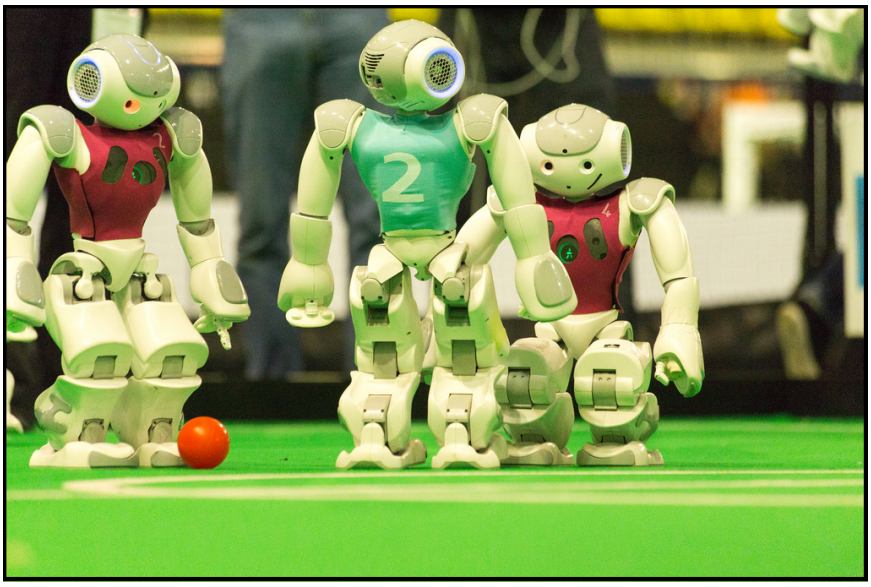

Figure 3.1: Sony Nao bots taking part in the RoboCup competition (Source: [RoboCup, 2014]).

3.1.2 Software Packages

The following are software packages that have been developed for running reinforcement learning algorithms on test environments and analysing the results.

The *RL-Toolbox* [Neumann, 2005] is an open-source framework, written in C++, for running reinforcement learning algorithms. It includes some standard reinforcement learning algorithms and also provides general interfaces for implementing new algorithms. It contains tools for creating discrete state spaces for learning tasks and carrying out parameter selection. It should be noted that the tool is no longer updated by the author but it is still available to download at

the time of writing.

CLSquare (CLS^2, Closed Loop System Simulation) [Hafner & Riedmiller, 2005] is a software package for testing reinforcement learning algorithms on benchmark problems such as the Cart Pole [Michie & Chambers, 1968] and Mountain Car [Moore & Hall, 1990] problems. The latest version of the software can be downloaded from the authors' website[1] and includes a variety of controllers including table-based Q-Learning.

The *Waikato Environment for Knowledge Analysis (WEKA)* [Hall *et al.*, 2009] is a unified workbench that gives researchers access to state-of-the-art machine learning techniques. While the main focus of WEKA is for running classifier and filter algorithms, it can also be used in conjunction with other frameworks such as the Platform for Implementing Q-Learning Experiments (Piqle) [Comité & Delepoulle, 2005] Java framework for designing and testing reinforcement learning experiments. It allows the user to design new algorithms or problems and also includes many of the standard reinforcement learning algorithms and some sample problems.

Tanner & White [2009] developed *Reinforcement Learning Glue (RL-Glue)* which is a standard interface for connecting agents, environments and experimental programs regardless of the programming language that each of them are written in. It includes a set of guidelines for the reinforcement learning community so that the sharing and comparison of components (agents/environments/experiments) can be carried out with greater ease. It essentially acts as a "glue" for connecting each of the components and eliminating the need to rewrite code. There are several programming language codecs[2] for connecting components written in different languages such as Java, C++ and Matlab.

Ertel *et al.* [2009] developed a robot learning toolkit, called the *Teaching-Box*, which includes a Java library of reinforcement learning and supervised learning algorithms. It includes a variety of environments to test the algorithms on and can provide visualisations of the experiments. It also supports the previously described RL-Glue interface.

[1] http://ml.informatik.uni-freiburg.de/research/clsquare

[2] The authors note that codecs are the language-specific software for allowing creations from a particular language to connect to RL-Glue.

Papis & Wawrzynski [2013] created a platform for running reinforcement learning algorithms on a series of benchmark problems called *dotRL*. It was developed using the .NET framework and includes algorithms such as SARSA and Q-learning. It also contains several benchmark problem environments such as Cart-Pole Swing Up [Michie & Chambers, 1968] and Acrobot [Connell & Mahadevan, 1993]. Further algorithms and environments can be added by the user by implementing an interface and the software contains several tools for running and analysing experiments.

3.1.3 Commercial and Open-Source Computer Games

At the turn of the century, Laird & Van Lent [2001] noted that "*Just as computers have inexorably gotten faster, computer game environments are becoming more and more realistic worlds, requiring more and more complex behavior from their characters*". Since then, many commercial computer games have been used in academic research to test state-of-the-art algorithms.

GameBots [Kaminka *et al.*, 2002] is a virtual reality platform that allows users to create and evaluate agents in a 3D environment. GameBots is made up of two components. Firstly, it is an open-source extension of the Unreal Tournament game engine and has a socket-based API so that agents can be created and deployed in the Unreal Tournament series of games. Secondly, it has development tools and sample code available for users. Unreal Tournament 2004, which was chosen for this research project, will be discussed later in Section 3.3.

Gorman *et al.* [2007] developed the *Quake 2 Agent Simulation Environment (QASE)* Java API after identifying a lack of a fully-featured API for carrying out academic research using a commercial game. The authors state that they used an FPS game as the testbed as it provides a "*comparatively direct mapping of human decisions onto agent actions*". This differs from other game genres where factors other than the player's decision-making process are taken into account during game-play. For instance, in an adventure game the player would need to recognise game objects and have conversations with other characters.

Choi *et al.* [2007] use *Urban Combat*[1], a simulation built on top of the open

[1] http://ailab.wsu.edu/uct/

source game *Quake 3*, as a testbed for carrying out experimentation on their cognitive architecture called Icarus. Urban Combat provides an interface for computer-controlled players to be deployed in the environment and carry out actions in real-time. A graphics interface is also provided to allow human players to execute actions in the environment. Detailed logging is included which enables users to carry out behavioural analysis of the computer-controlled players.

Another game in which bots can be deployed is called *Counter-Strike*. This game, which was originally a modification of the FPS game *Half-Life*, has a large community of users that create modifications for the game [Kücklich, 2005]. Floyd [2006] created PODBot which is an open source (GPL) metamod[1] plugin for adding bots to the game. Metamod is a DLL manager that sits between the Half-Life Engine and Half Life game modifications. Users can design their own bots and then deploy them in the game.

Bellemare *et al.* [2013] developed the *Arcade Learning Environment* (ALE) which is a testbed that provides an interface to hundreds of different Atari 2600 games in order to evaluate AI techniques. The learning environment, including documentation and tutorials, can be downloaded from the developers website[2].

The emergence of smartphones and tablets with high-speed processors has led to the development of mobile games with PC-quality graphics [Tach, 2014]. AI being deployed in apps and games running on hand-held devices will undoubtedly see a lot more attention in the coming years.

3.1.4 Board Games and Purpose-Built Games

Board games have long been considered benchmark problems for AI. Fully deterministic, perfect information games such as chess [Goodman & Keane, 1997] and checkers [Schaeffer, 2008] have been reported as success stories with computer-controlled players learning how to play at grandmaster level. Other games such as poker, which contain incomplete information, have not yet reached the same performance levels of top human players although there have been some recent advances, for example, with heads-up limit Texas holdem [Bowling *et al.*, 2015].

[1] http://metamod.org/
[2] http://www.arcadelearningenvironment.org/downloads/

The non-deterministic, perfect information game Backgammon was one of the early success stories of AI [Tesauro, 1992, 1995]. Papahristou & Refanidis [2013] have developed a program called *AnyGammon* which extends the board game of backgammon so that users can play on different board sizes. The game still retains the original rules but the authors state that smaller board sizes can make it easier to analyse the effectiveness of different AI algorithms. Maze environments [Madden & Howley, 2004] and simple games such as *Tic-Tac-Toe* [Sutton & Barto, 1998] are also often used as testbeds. Researchers can also create purpose-built environments, see for example McPartland & Gallagher [2011], in order to have full low-level control of their environment.

3.1.5 Artificial Intelligence Competitions

The following is a selection of gaming competitions, which were organised and run over the past decade, for testing AI algorithms.

General Video Game Playing [Genesereth *et al.*, 2005; Levine *et al.*, 2013; Perez *et al.*, 2015] competitions involve creating controllers for general video gameplay. Learning agents must play many different general games, the details of which are unknown to the designers when submitting their entries. The purpose of the competitions is to test Artificial General Intelligence (AGI) [Goertzel & Pennachin, 2007].

The Annual Computer Poker Competition, which is run by the AAAI Conference on Artificial Intelligence [Littman & Zinkevich, 2006], is a competition which allows entrants to deploy their computer-controlled poker agents in many different variants of the game of poker.

The *Ms Pac-Man* screen-capture competition [Lucas, 2007], which is based on the Ms Pac-Man arcade game, has been run since 2007. The players read screen captures from the game and their objective is to obtain the highest score. The competition was modified in 2011 to allow the participants to control both Ms Pac-Man and the ghosts and was renamed the *Ms Pac-Man vs Ghosts* competition [Rohlfshagen & Lucas, 2011].

The *Simulated Car Racing Championship* [Loiacono *et al.*, 2010a], which uses

the open-source racing game *The Open Racing Car Simulator* (TORCS[1]), is a competition for testing AI controllers in a racing environment. Competitors in the championship receive points depending on where they place in the races and can receive additional points for racing the fastest lap and receiving the smallest amount of damage. The cars have various sensors and effectors for enabling automated control. Older racing competitions, such as the 2007 IEEE CEC Simulated Car Racing Competition [Togelius *et al.*, 2008], used much more simplistic software in terms of graphics and game mechanics.

The *StarCraft* AI Competition[2] invites the submission of bots to battle in the retail version of the real-time strategy game *StarCraft: BroodWar*. A survey of AI approaches in StarCraft and a review of the different competitions can be found in Ontanón *et al.* [2013].

Hingston [2009, 2010] created the *Bot Prize* Competition for testing the humanness of computer-controlled characters in the first-person shooter game Unreal Tournament 2004. We described the details of this competition earlier in Section 2.5.1.

The *Mario AI Benchmark* [Karakovskiy & Togelius, 2012] is a benchmark problem for reinforcement learning and other AI techniques based on a public domain clone of the Nintendo game *Super Mario Bros.* It is an open-source platform game and was used as the Turing Test track for the 2012 Mario AI Championship [Shaker *et al.*, 2013]. The competition involved human judges watching recorded gameplay clips and judging players as being either human-controlled or not.

3.1.6 Discussion

There are many interesting domains and available testbeds for carrying out AI experimentation as shown in the previous sections. For this research project, we have chosen the game Unreal Tournament 2004 and the Pogamut 3 IDE for the following reasons. Firstly, Unreal Tournament 2004 is a commercial game and, although it was released over a decade ago, it still encompasses all of the challenges associated with FPS games today. Secondly, the game involves real-

[1] http://torcs.sourceforge.net
[2] http://starcraftaicompetition.com

time multiobjective decision-making and such an environment is highly suitable for testing the adequacy, and identifying the limitations, of using reinforcement learning to autonomously control NPCs. Thirdly, the IDE, which has been in development for many years and is consistently updated, facilitates and simplifies the process of creating and deploying bots on a game server. This ensures that the majority of the development effort from our research can focus on designing the learning architectures, running experimentation and analysing the results.

3.2 Pogamut 3

Pogamut 3 [Gemrot *et al.*, 2009] is an open-source platform toolkit for creating virtual agents in the 3D game environment of Unreal Tournament 2004. It makes use of *UnrealScript* for developing external control mechanisms for the game. The main objective of Pogamut 3 is to simplify the coding of actions taken in the environment, such as path finding, by providing a modular development platform.

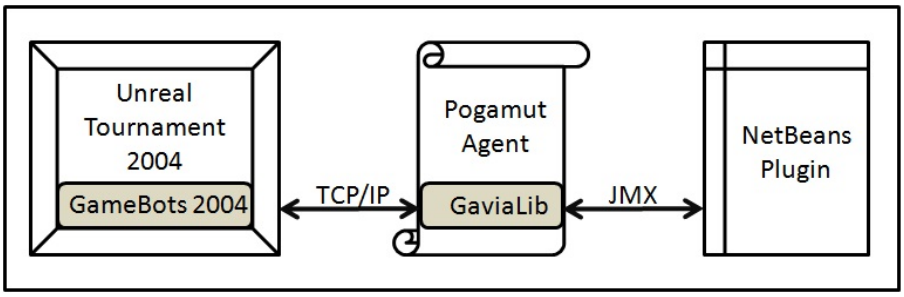

Figure 3.2: The Pogamut 3 architecture based on Fig 1. of Gemrot *et al.* [2009]

Pogamut 3 integrates five main components: Unreal Tournament 2004, GameBots2004, the GaviaLib Library, the Pogamut Agent and the NetBeans IDE. This is illustrated in the architecture diagram in Figure 3.2. In the following sections we will briefly discuss each of these components. For a more detailed explanation see Gemrot *et al.* [2009].

3.2.1 Environment: Unreal Tournament 2004

The game Unreal Tournament 2004, discussed in detail in Section 3.3, is the virtual environment used for running the agents built with the Pogamut 3 IDE. The authors state that they chose UT2004, over similar FPS games such as Halo 2 and Quake 3, because it had a bigger following of community modders and, at the time, it had a "better graphical experience". Users can also edit the game using UnrealScript and there are a lot of pre-built objects, maps, and a level editor included with the game.

3.2.2 GameBots2004

The authors developed a customised extension for the original Gamebots [Kaminka *et al.*, 2002] testbed for connecting the Pogamut 3 toolkit to the UT2004 game. *GameBots2004* exports game information through a TCP/IP text-based protocol that allows users to connect to the game through a client-server architecture. Users can design and develop the bot's control mechanism and this acts as the client in the architecture. GameBots2004 then acts as the server. GameBots2004 can be used as a standalone component, without interacting with any of the other Pogamut 3 components, so a user could create their own development environment to interact with it if they wished.

3.2.3 GaviaLib Library

The *GaviaLib* library is a Java library that was created to work as a generic interface for connecting to different virtual worlds. The only assumption about the environment that is being connected to is that it works with objects and can provide information from events that occur. GaviaLib provides an abstract agent implementation for controlling the agent using a Java technology called Java Management Extensions (JMX)[1]. It also provides the interface to the world for notifying the agent of changes to objects and events as they occur. The GaviaLib architecture is shown in Fig. 3.3 below.

[1] http://docs.oracle.com/javase/tutorial/jmx/

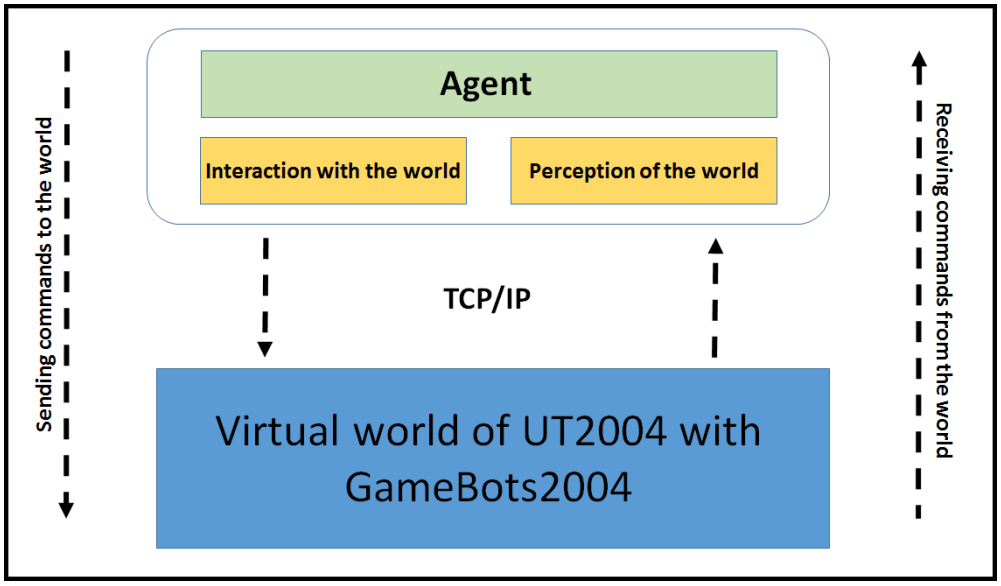

Figure 3.3: The GaviaLib architecture based on Fig 3. of Gemrot *et al.* [2009]

3.2.4 Pogamut Agent

The Pogamut agent classes are used to program the bots that are to be deployed in the game. Several templates and examples exist from basic navigation bots to more complex agents. The bot's decision-making can be controlled directly from writing Java code or by using a plugin application called the POSH planner. This is an independent action-selection planner, aimed at helping beginners with their initial experiments, and functions as a plugin to the Pogamut 3 toolkit.

3.2.5 NetBeans IDE

The Integrated Development Environment was developed as a NetBeans[1] plugin. It communicates with the agent via JMX and there are various sample projects and example agents available that can be used as templates. The IDE also assists the user with the management of UT2004 servers and provides a list of running agents with access to the agent's variables and general properties. Step-by-step debugging and log viewers are also included to assist developers.

[1] https://www.netbeans.org

3.3 Unreal Tournament 2004

3.3.1 Overview

Unreal Tournament 2004 (UT2004) is a commercial first-person shooter game that was developed primarily by Epic Games and Digital Extremes and released in 2004 [Liandri, 2014]. It is a multiplayer game that allows players to compete with other human players and/or computer controlled bots.

Figure 3.4: Cover art of the game Unreal Tournament 2004

It includes ten different game modes which range from team-based games, such as Capture the Flag and Domination, to "every person for themselves" style games such as Deathmatch where the objective is simply to kill every other player in the game. A wide variety of diverse maps and weapons are available to choose from in the game. The game is built upon the Unreal Engine and the availability of UnrealScript, for simple high-level programming of the game, has led to a dedicated following of software modifiers (modders) and new content developers. *Mutators* can be written and applied to games. These are scripts that are written to apply user-defined changes to a game and can effect aspects such as the health levels of players, available pickups and gun availability during games. Almost everything in UT2004, excluding the graphical and physics part of the engine,

can be modified by the user.

3.3.2 Suitability

UT2004 is a highly customisable game with a large number of professional developer and user-made maps, a wide variety of weaponry and a series of different game types from solo play to cooperative team-based play. The game was released over ten years ago much to critical acclaim [Liandri, 2014] and, although computer game graphics in general have since been vastly improved, the FPS formula and foundations of gameplay remain the same in current state-of-the-art FPS games. Accessibility to the mechanics of the game is simplified through the use of the GameBots2004 and Pogamut 3 IDE. Modern commercial FPS games such as the *Call of Duty*[1] or *Battlefield*[2] series are multi-million dollar projects so their source code remains closed, however, any demonstrable advances in the game-AI of UT2004 would be equally applicable to the latest cutting-edge FPS games given that the underlying structure of the games remain analogous.

3.3.3 Technical Details

The following sections will describe some of the technical details for UT2004 such as the different game types, maps and weaponry available. The game's units of measurements, the experiment environment, native fixed-strategy opponents and the inbuilt navigation system will also be discussed.

3.3.3.1 Game Types, Maps and Weapons

There are eleven inbuilt default *game types* to choose from in UT2004. The game type determines how players interact with each other and the environment. Some of these game types, such as Deathmatch and Team Deathmatch, involve the sole objective of using various weapons to eliminate opponents. Other game types include parallel objectives where players must fight the opponents but also control different areas of the map or capture enemy flags for example. Users can

[1] http://www.callofduty.com/
[2] http://www.battlefield.com/

also define their own custom game types with tailor-made rules and scenarios.

Each game is played in a three dimensional environment called a *map*. There are many different default maps, of varying sizes and complexity, that are included with the game and it is also possible for users to build their own custom maps using the Unreal Editor application that comes with the game. Mutators, as mentioned earlier, can also be developed and applied to any map. These can directly affect the number of pickups available, weapons available and the levels of health for the players.

There are a wide variety of weapons available to players which are used to inflict damage on their enemies. Each weapon, details of which can be found in Liandri [2014], has a *primary* and *secondary* mode. For a human player, primary mode is a left mouse click and secondary mode is a right mouse click. These modes have different outcomes depending on the weapon. For instance, the Assault Rifle is a machine gun in primary mode and launches grenades in secondary mode whereas the Sniper Rifle shoots a high powered sniper round in primary mode and scopes in to a magnified view of the opponent as the secondary mode.

3.3.3.2 Unreal Units of Measurement

Unreal Tournament 2004 has its own unit of measurement called an *Unreal Unit* (UU). These units are used when measuring distance and velocity. The standard running speed for players peaks at 440 UU per second. 1 UU is approximately equivalent to 2cm in the real world[1]. The player avatars in the game are 88 UU high (58 UU when crouched) which means that the real world height is 1.76 metres. Angles are measured by using *Unreal Rotation Units* (URU). The rotation values returned from the system range from 0 to 65536 (2^{16}). That is, a full circle is represented as 65536 URUs which is equivalent to 360 degrees. The following formula is used to convert URU to degrees:

$$degrees = (URU \ / \ 65536) \ * \ 360$$

[1] http://udn.epicgames.com/Three/ChangingUnits.html

3.3.3.3 Server and Experiment Setup

The game settings and server can be configured by editing the configuration files in the *System* folder of the UT2004 game directory. Servers can be started from the command line by typing "ucc server" followed by a list of parameters. Alternatively, there are default batch files for running different types of servers that can be found in the system folder. Once the server has been started, both user-defined bots and *Native* bots can be added to games. Native bots are computer-controlled bots that ship with the game and have eight different skill levels. The Native bot's perception and reaction speeds are influenced by the skill level setting, as described in the next section. Human users can also act as spectators to these games or join the game as a player if they wish by using the NetBeans plugin as shown below in Figure 3.5.

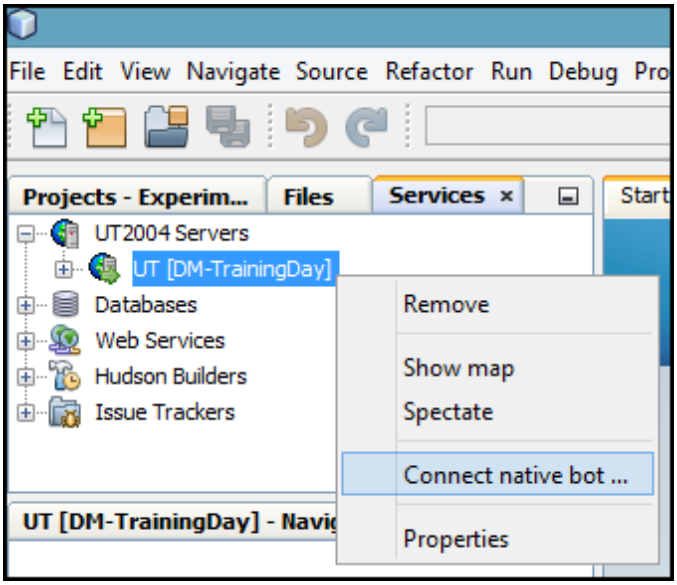

Figure 3.5: Connecting native bots using the server plugin in NetBeans.

Bots developed using the Pogamut toolkit have a *logic* method which is called, by default, four times a second. This value can be edited using the configuration file but reducing this value results in the bot having a slower reaction time and thus can reduce its performance during the games. There are various dif-

ferent inbuilt methods for checking the status of a bot such as reading health and ammunition values, absolute positioning and sensing opponents and items (simulating hearing and seeing) in the environment. The reinforcement learning architectures that were developed, as described in the next three chapters, make use of the information provided by these methods.

3.3.3.4 Native Bot Skill Levels

There are eight different pre-programmed native bot skill levels in UT2004 that are designed to increase the challenge for human players as the skill level is increased. High-level descriptions of the attributes associated with each of these skill levels, as reported by Unreal [2007], are listed in Table 3.1.

3.3.3.5 Bot Navigation

Each of the maps that are included with the UT2004 game include navigation point graphs, or *NavPoints*, which can be used by the bot to safely traverse the environment.

Figure 3.6: NavPoint graph from the map Albatross in UT2004. (Source: [Kadlec *et al.*, 2014])

When there is a direct path from one NavPoint to another then they are

connected by edges as shown in Figure 3.6. *Ray casting* is another technique that can be used in the absence of a NavPoint graph or when then NavPoint graph does not give enough information. This involves declaring and initialising *rays* of a fixed length which protrude from the bot. These are invisible to opponents in the game and will signal if they are currently intersecting with map geometry or other players in the environment so that the bot can react accordingly. There are inbuilt methods in the game for dodging, jumping, walking, running and crouching. The Pogamut toolkit developers provide an online resource called the Pogamut 3 Cookbook [Kadlec *et al.*, 2014] which provides both basic and advanced tutorials for developing and deploying bots in UT2004.

3.4 Chapter Summary

This chapter has outlined a variety of different development toolkits and testbeds that can be used for carrying out AI experimentation. It has provided a comprehensive summary of those that are closely related to our research, that is, those in which reinforcement learning algorithms can be applied, gaming testbeds and game AI competitions. In this chapter, we also discussed the rationale for choosing the Pogamut 3 IDE and the UT2004 FPS game before going on to summarise the core technical details for both.

Skill	Attributes
Novice	60% of regular running speed, will not move during combat unless very weak, limited perception with 30° field of view, shooting aim can range 30° off target, slow to turn (turning is based on the *ReactionTime* variable).
Average	70% of regular running speed, slightly higher shooting accuracy, turns slightly faster than novice.
Experienced	80% of regular running speed, will move and fire simultaneously, 40° field of view, can turn by more than 1/2 per second.
Skilled	90% of regular running speed, can double jump, 60° field of view, turns more than 5/8 per second.
Adept	Run at full speed, will dodge enemy fire, will close in on enemy, aim "leads" the target, 80° field of view, turn almost 3/4 per second.
Masterful	Impact jump use, shock combo use, will switch target enemies during combat, 100°field of view, turn almost 7/8 per second.
Inhuman	Will dodge enemy aim, no limits on perception, 120° field of view, turn all the way around in less than 1 second.
Godlike	All strategies and tactics are engaged, Omni-perceptive: seeing all players and objects even without line of sight, highest level of accuracy with less than 1° off target, turns 1 and 1/2 times per one second.

Table 3.1: UT2004 native bot skill attributes. (Source: [Unreal, 2007])

Chapter 4

Sarsa-Bot Architecture

In this chapter, we describe our reinforcement learning behavioural architecture, called *Sarsa-Bot*[1], that we developed to control an NPC in the FPS game Unreal Tournament 2004. Firstly, we give an overview of the problem at hand and discuss the short-comings of traditional techniques used for computer-controlled characters. This is followed by a discussion of the motivating factors for carrying out this work. We then provide a description of the architecture design and present an analysis of the experimentation that we carried out. The chapter concludes with a discussion and a summary of the work.

4.1 Overview

Firstly, we would like to note that this chapter is the first of three chapters in which we describe reinforcement learning behavioural architectures that we developed for controlling NPCs in Unreal Tournament 2004. Each of these are standalone projects with each successive architecture evolving based on the experimental analysis of the previous one. Secondly, the motivations that we describe in the next section and those described earlier, in Section 1.2, are the overall motivations that apply to each of these three architectures.

As discussed earlier in Section 1.1.2, modern FPS games take place in multi-objective, photo-realistic environments where players, both human and computer-

[1]Some of the work in this chapter was first published in Glavin & Madden [2011]

controlled, compete by engaging in combat and completing tasks. Traditionally, game developers have used scripted techniques, such as Finite State Machines (Section 2.3.1.3) and Fuzzy Logic (Section 2.3.1.5), to control the NPCs. These have the benefit of being simplistic and are a "tried and tested" approach as far as development is concerned, however, they can often lead to predictable gameplay that can be exploited by knowledgeable and experienced players. Controlling an FPS bot in a multi-objective 3D environment is certainly a difficult task, even for a human. Constant real-time decision-making is required for a variety of different tasks such as path-finding combat, retreating and completing game objectives. We are proposing the use of reinforcement learning to develop bots that can learn their own strategy from experience and continually adapt how they interact with opponents over time. The design of such an architecture involves careful consideration of *states*, *actions* and *rewards*. States comprise all of the information that the bot perceives from the environment to describe its surroundings and provide it with situational information. Actions are control statements for interaction with the environment and rewards result in either positive or negative feedback for the bot based on the outcome of its decision-making.

4.1.1 Motivation

The computer games industry has seen enormous growth in recent times [Blau *et al.*, 2013] and there is now an ever-growing need for challenging and immersive AI in modern computer games. NPCs take on many different roles depending on the genre of the game. For instance, adventure or sandbox games may include NPCs to enhance the realism of the environment. These can consist of crowds of characters going about their daily lives and interacting with each other. In adversarial games, such as first-person shooters, NPCs directly interact with human players either as allies or opponents with their competency forming a large part of the overall gameplay mechanics. Predictable, flawless or (conversely) constantly poor performing NPCs can have a detrimental effect on a human players enjoyment of the game [Koster, 2013]. For this reason, we believe that NPCs should be adaptive, spontaneous and learn from experience in order to succeed at entertaining. To enable such behaviour, we are proposing the use of reinforcement

learning in which the NPC will base its decision-making on the feedback that it receives through experience. This will allow the bot to adapt to different situations and varied opponent behaviour with a view to making it less predictable. Its decision-making will be based on the interaction with the opponents in the environment as opposed to deterministic scripting. We believe that producing this adaptive behaviour will move us a step closer to generating human-like behaviour in NPCs. Here, when we speak of human-like behaviour, we are referring to the fundamental ability of humans to learn from their mistakes and adjust their behaviour over time. There have been many different approaches in the literature, some of which were discussed earlier in Section 2.5, to produce and measure human-like behaviour in FPS games.

The inspiration for this research was stemmed from a love for FPS games and observing a niche for improving the behaviour of computer-controlled players in these games. In particular, the success of the Bot Prize competition (described in Section 2.5.1) and the early work of McPartland & Gallagher [2011], who used reinforcement learning on a simplistic purpose-built FPS, were two early sources of inspiration. The three dimensional environment of a first-person shooter game provides a challenging domain for testing artificial intelligence algorithms and we believe that advancements in this area could be applicable to other real-time decision-making domains such as robotics. Our work also contributes to identifying appropriate evaluation procedures and quantifying real-time learning as well as outlining our design process for developing suitable states, actions and rewards.

4.2 Sarsa-Bot Architecture Design

For a human, learning to effectively play an FPS game is a difficult task that takes time and patience to master. In order to develop useful strategies and tactics, the player must observe his or her success by monitoring the outcomes of their individual actions. Over time, the human player builds up a knowledgebase of information which helps guide them towards good decision-making and winning behaviours in the future. We have designed the Sarsa-Bot architecture to enable the NPC to decide for itself which actions it should take depending on the cir-

cumstances that it finds itself in. A reward signal, which we designed to promote effective Deathmatch behaviour, is included to guide the bot's decision-making as it learns through experience. The following sections provide details about the implementation, the algorithm used and how the bot perceives and interacts with the environment.

4.2.1 Implementation Details

We developed the Sarsa-Bot in the Java programming language using the Pogamut 3 IDE [Gemrot *et al.*, 2009] which was described earlier in Section 3.2. We used the *NavigationBot* and *HunterBot* examples [Kadlec *et al.*, 2014], which are included as example projects with the Pogamut toolkit, as a basis for development. These examples contain basic navigation and shooting modules. The *logic* module acts as the brain of the bot and is called periodically, 4 times a second by default, by an internal thread associated with the bot. All of the decision-making required for the bot to act by itself must be contained inside this method. The bots can be tested by playing against *native* Unreal Tournament 2004 bots which have scripted, fixed strategies or by competing against human players. Firstly, a game server must be configured and executed. The developed bot, human players and/or native opponents can then be added from inside the Pogamut plug-in in NetBeans (see Section 3.3.3.3). Humans can spectate the games that are running on the server and the games can also be stored as gameplay videos so that the behaviour of the players can be analysed. The SARSA algorithm [Rummery & Niranjan, 1994], which was described in Section 2.2.3.2, was embedded in the logic of the bot and we developed several utility modules to enable the bot to carry out in-game actions. We will now take a look at the details of the architecture. A full overview of the Pogamut 3 IDE and the Unreal Tournament 2004 game technical details can be found in Section 3.2 and Section 3.3 respectively.

4.2.2 On the use of the SARSA Algorithm in Sarsa-Bot

The algorithm that we use for the Sarsa-Bot implementation is the SARSA algorithm. This algorithm is very straightforward and is well suited to problems that involve continuous decision-making. We firstly implemented the algorithm

using a simple, purpose-built maze environment in order to test and validate its functionality. Once this had been completed, we integrated the algorithm into the logic of the bot. In order for the algorithm to work, a state space representation, an action space representation and a reward signal had to be designed and implemented. The choices that we made for designing these components are described in the following sections.

4.2.3 Sarsa-Bot State Space

We designed, tested and redesigned the state space several times in order to balance promoting the ability to learn with complexity management. Choosing important features from the environment with the correct amount of abstraction to avoid the "curse of dimensionality" [Bellman, 1957] is a challenging task. The state-action space grows considerably as features are added so the careful selection of the most important features is critical. For this reason, we identified the most significant information needed by the player. We determined this information to be the level of health, the level of ammunition and whether or not the bot could see an opposing player. We discretised the levels of health and ammunition to include five levels for each. This enables us to calculate state values by checking what range they currently lie in. For example, if the bot has health greater than or equal to 100 it will return 0, if it has health in the range of 80 to 99 it will return 1, and so on. A graphical representation of the state space is shown below in Figure 4.1 and the formula used to return the state, s, that the bot is in at any given time.

The *getSeeOpponent* function will return 0 if the opponent is not visible and 1 if the opponent is visible. The *getCurrentAmmo* function will return a value from 0 to 4 depending on the range in which the current ammunition level is in. The *getCurrentHealth* function works in the same fashion as *getCurrentAmmo* by returning a value from 0 to 4. There are a total of 50 possible states as the formula will return a number between 0 and 49, inclusive. The result that is returned for the current state is based on the information from the function calls that the bot reads from the system.

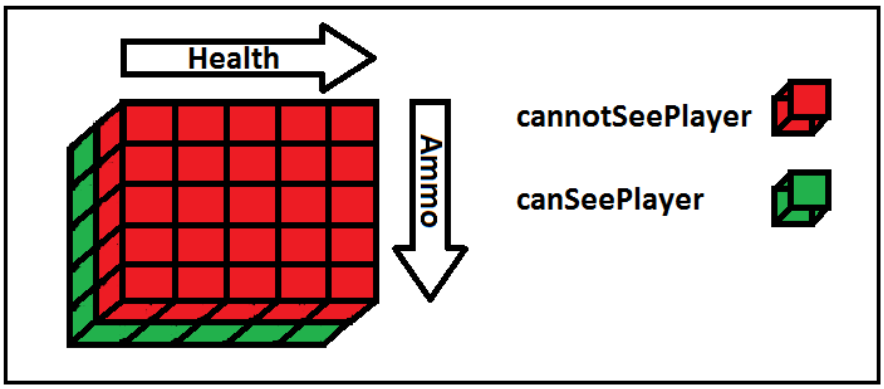

Figure 4.1: A graphical representation of the state space used.

$$s = (getSeeOpponent() \times 25) + (getCurrentAmmo() \times 5) + getCurrentHealth()$$

4.2.4 Sarsa-Bot Action Space

The next task that we had to complete was to design the actions that would be available to the bot. As mentioned earlier, these actions are control statements and the algorithm must learn which actions earn the most reward in the specific states. We designed eight actions in total which meant that the state-action Q-value table would consist of 400 entries (50 states multiplied by 8 actions). It was decided that the actions had to be relatively high level in order to create a bot which had a sufficient level of play. For instance, we deemed actions such as *takeStepForward*, *lookLeft* etc. as being too low level. These were replaced with actions such as *lookForPlayer* in which the bot traverses through the map until it spots an opposing player. This was achieved by using the inbuilt *NavPoint* navigation system in the Pogamut IDE which was described in Section 3.3.3.5.

All of the actions that are available to the bot are listed in Table 4.1. The *lookForPlayer* action traverses through the map one NavPoint at a time. The bot also keeps track of the last location that an enemy was seen and will change the focus of its navigation if it hears a noise. The *lookForPickup* action will continually move around, similar to the previous action, but it will also move the bot towards any pickup items in its field of view. The *shootPrimary* and

shootSecondary actions will engage any visible opponent with either the primary or secondary mode of the current weapon that they are using (see Section 3.3.3.1 for more details).

Action	The bot will:
lookForPlayer	Move through the map stopping when it sees a player.
lookForPickup	Search for and move to the nearest visible item.
shootPrimary	Shoot at any visible player in primary mode.
shootSecondary	Shoot at any visible player in secondary mode.
dodge	Perform a dodging maneuver in a random direction.
jump	Perform a jump to a random height.
changeWeapon	Change its weapon to another from the inventory.
goToLastSeenPlayer	Go to the location of the last visible player.

Table 4.1: Actions available to the Sara-Bot.

The bot will only shoot its weapon if a player is visible. The bot will perform a dodging maneuver when the *dodge* action is selected. This is performed in a random direction and can consist of either a single or a double[1] jumping dodge. The *jump* action again has some randomness built in with regards to the amount of height jumped and whether a single or double jump is carried out. The bot can change to another weapon by selecting the *changeWeapon* action. The bot will only change to a weapon that is currently loaded with ammunition. Finally, the bot can travel to the recorded location of the last seen player using the *goToLastSeenPlayer* action. If there is no location recorded, the bot will revert to the *lookForPlayer* action.

In this implementation, an emphasis was put on simplicity for the state space and the action space. The bot also has the benefit of some hard-coded knowledge such as not shooting when there is no visible target and using *NavPoint* navigation.

[1] A double jump is activated by choosing to jump while already mid-air from an original jumping action.

4.2.5 Sarsa-Bot Reward Signal

We developed a reward signal, which is a scalar value that is made up of the accumulated values from a series of system status checks[1]. These checks and their corresponding reward values are listed in Table 4.2.

Status	Reward:
seeOpposingPlayer	+10
hasJustKilledOpposingPlayer	+10,000
isCausingDamage	+1,000
hasDamageBonus	+100
isHealthy	+10
hasCollectedPickUp	+50
isNotHealthy	-10
isColliding	-10
hasJustDied	-1,000
isBeingDamaged	-200

Table 4.2: Rewards received depending on current status.

All of the checks are Boolean and will return either True or False. When the bot carries out an action in a specific state, all of these checks are run in real-time and the bot receives the calculated reward so that the corresponding Q-value can be updated in the Q-table. If the *seeOpposingPlayer* status check returns true, a reward value of 10 will be added to the reward total. A large reward value of 10,000 is added if the system registers that the bot has killed an opponent whereas 1,000 is deducted if the bot is killed by an opponent. 1,000 is added if the bot is causing damage and 200 is deducted if the bot is receiving damage. The bot will continuously receive +10 while it has sufficient health but this becomes -10 if the health drops below a threshold level. It receives 100 if it

[1]There is a direct comparison of this approach with a simple reward in Section 4.3.5

has caused enough damage to opponents that it receives a damage bonus from the system. It will receive 50 for successfully collecting a pickup from the map and 10 is deducted if the bot collides with any of the walls or obstacles on the map.

We designed these status checks and their values based on the perspective of a human player. Firstly, we identified the main objectives in a Deathmatch competition for a player and then proposed a weighting of how important each of the different game events were and how much of an impact they had on the player. From here, we established a numerical value to represent each event. The values reported are those that produced the best performances during our initial trial runs.

4.2.6 Discussion of Design Choices

The Sarsa-Bot is designed to function as an adaptable NPC based on our implementation of high-level actions using reinforcement learning with a simplistic state space representation. It receives rewards (or penalties) from multiple sources in the environment and this is used to guide its decision-making. Given that the bot bases its behaviour entirely on feedback from the environment, it produces behaviour that is less repetitive and predictable than traditional, deterministic scripting. With this implementation, however, it can be difficult to continually improve its capability with carrying out *specific* tasks. Reward is received as an accumulation of many checks from the environment as described earlier. This can lead to a credit assignment issue where specific actions are being reinforced by rewards that may not be necessarily related to them. The Sarsa-Bot, however, will always strive to receive the most reward it can from the environment. It is, of course, limited by the design of the states, actions and rewards but can still function as an adaptable opponent that changes its behaviour based on a generalised view of good game-play from the accumulated reward signal. We will now take a look at the experimentation that we carried out using the Sarsa-Bot architecture.

4.3 Sarsa-Bot Experimentation

In this section, we describe the experiments that we carried out in the early stages of this research using the Sarsa-Bot architecture which we developed using decision-making logic based on the SARSA algorithm. We begin by outlining the experiment details and presenting the results. This is followed by a discussion and summary in which we also describe some of the challenges and issues that were identified during this time.

4.3.1 Experiment Details

The initial experiments consisted of 1-vs-1 Deathmatch games against a human player. The sole objective of this game type is to kill the opposing player using a variety of guns, some of which can be picked up from the environment and some of which the player is equipped with at the beginning of the game. All of the games, both against human and against computer-controlled players, are played out in real time by connecting to a localhost server on a Windows computer. How quickly we can expect the agent to adapt in practice is dependent on the size of the state-action space. The Sarsa-Bot has 400 entries in its state-action table as there are 50 states and 8 available actions. In contrast to this, RL-Shooter (described later in Chapter 6) has 38880 state-action values based on 6 categories of weapon. In the case of RL-Shooter, 350 thirty minute games were played against three levels of opponent to ensure that the majority of states were visited. Such lengthy runs were not required with Sarsa-Bot. The human trials took under an hour to complete with the follow up experimentation being carried out as 5 minute games as described below.

These games were played on the smallest map in the game, called Training Day, which is designed for two to three players. This is a suitable map for testing the architecture as it removes any large search times between players, minimises the risk of the bots navigation getting stuck in obscure geometry and encourages almost constant combat. The layout of this map is shown in Figure 4.2.

The first stage of the experimentation involved a human player playing against the Sarsa-Bot. Every time the bot died (one episode) the current state of the Q-table was stored to a file, in order to keep track of the learning that was occurring

as the Sarsa-Bot was gaining experience.

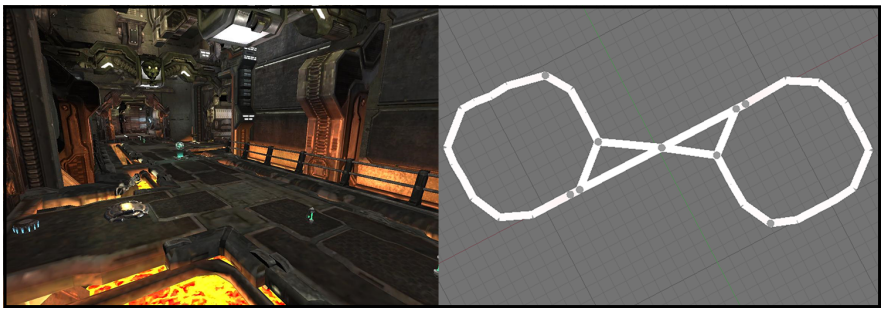

Figure 4.2: Training Day map and a bird's eye view of its layout.

Sampling from these "human vs bot" experiments, we took the Q-tables with varying degrees of experience. The first of these Q-tables corresponds to when the bot has no learning experience at all whereas the last one corresponds to the bot's Q-table having played and died 140 times against a human player. These Q-tables summarise all of the learning of the Sarsa-Bot, and any one of them can be loaded at the beginning of a game in order for the Sarsa-Bot to start the game with some experience. In order to identify if, in fact, any learning was occurring, we play the Sarsa-Bot at different levels of experience (which we will call XP[1]) against two Level 3 fixed-strategy bots from the game. Two opponents were used to increase the occurrences of player interactions. We ran one hundred five minute games with different levels of XP in order to average the results. It is important to note that once we loaded the XP Q-tables, we froze learning and did not allow the Sarsa-Bot to update the table during the game against the fixed bot. Therefore the bot was using the experience it had built up in the past playing against a human player but was not learning in real-time as it played against the fixed-strategy bots. The reason for this was to identify if the bot was, in fact, learning more effective strategies from experience. Another important note is that the bot with no experience (*0-XP*) had its Q-table populated with random values so that these could be frozen like the other levels of XP. Therefore the *0-XP* bot essentially represents the bot choosing actions at random. The same random values were used for the *0-XP* bot in all of the experiments. The

[1] 1-XP corresponds to the experience gained after dying once

averaged results from these games are shown in the following section.

4.3.2 Results

The first measurement that we took from the games was the average time (in seconds) that the Sarsa-Bot spent shooting per game. These results are shown in Figure 4.3.

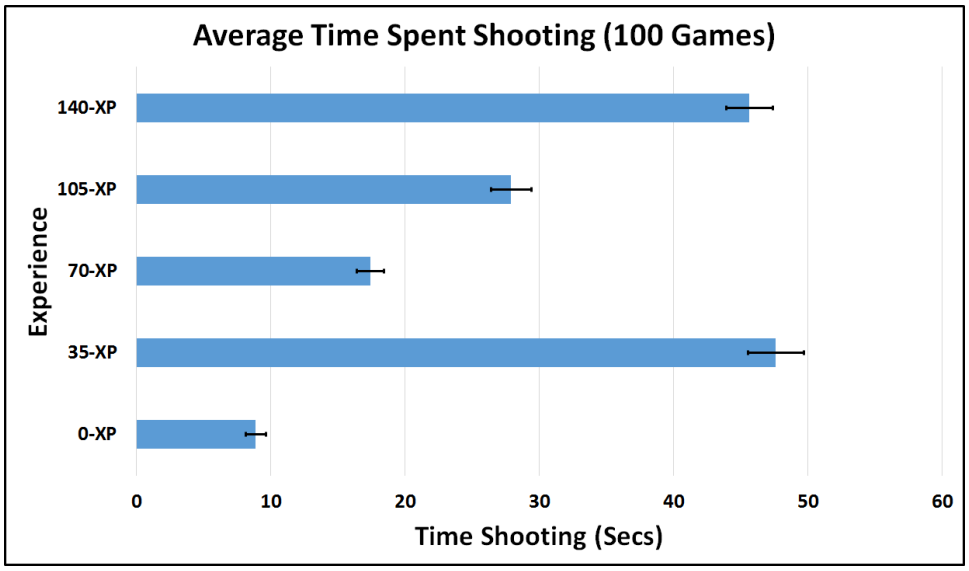

Figure 4.3: Mean and standard error of time spent shooting per game over 100 games.

Standard error of the mean (SEM) bars are shown in all the graphs and these represent an estimate of how far the sample mean is likely to be from the population mean. The SEM, σ_M, is calculated as follows.

$$\sigma_M = \frac{\sigma}{\sqrt{N}} \tag{4.1}$$

σ is the standard deviation of the original distribution and N is the sample size. Two of the eight actions available to the bot involve shooting and it is clearly important to learn when to select these actions in order to improve its

performance and kill the opponents. From the results we can see that the *0-XP* bot, which is choosing actions at random, spends the least amount of time shooting its weapon, only doing so for less than 10 seconds on average per game. One surprising result is that the *35-XP* bot spends the most amount of time shooting (48 seconds) out of all of the five bots which is marginally ahead of the most experienced *140-XP* bot (46 seconds). This would suggest that the strategy that was frozen at *35-XP* involved choosing the shooting actions in the most common states that were encountered. As learning continues, it appears to have replaced this strategy with another one by the time it reaches *70-XP* and *105-XP* as the amount of time shooting is lessened with these strategies. This suggests that some of the other actions could be becoming incorrectly assigned with the large reward associated with killing an opponent.

The average kills that each bot achieved are shown in Figure 4.4. These results show a similar pattern to the time spent shooting which would be expected.

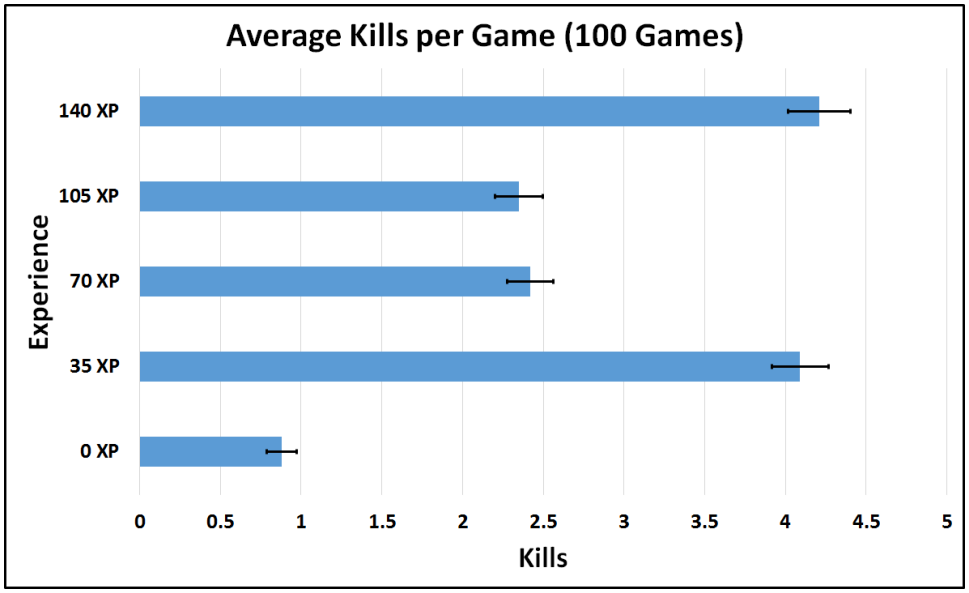

Figure 4.4: Mean and standard error of kills per game over 100 games.

There are, however, some notable differences. The bot with the most experience (*140-XP*) manages to marginally achieve the most kills per game even

though it shoots its gun, on average, less than the *35-XP* bot. The secondary shooting mode of the Assault Rifle launches a grenade which can kill an opponent on impact. The *140-XP* could have developed a strategy with the efficient use of this mode which would achieve more kills with less shots fired.

Picking up items during gameplay is an important part of the game mechanics as it replenishes health and ammunition as well as arms the player with more powerful weapons. Figure 4.5 shows the average number of weapons collected per game for each of the five bots.

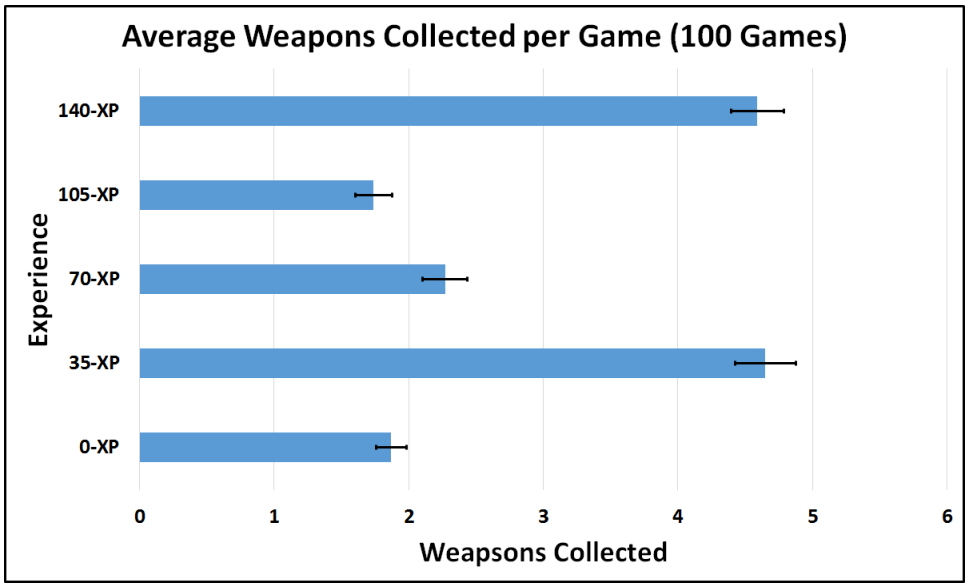

Figure 4.5: Mean and standard error of weapons collected per game over 100 games.

35-XP and *140-XP* average 4 to 5 weapon pick-ups per game whereas the others only collect approximately 2. These results could be interpreted in two ways. The first interpretation is that the bots that are more proficient at choosing shooting actions would kill the opponents more often, staying alive for longer, and therefore have more opportunities to pick up weapons while they are alive. The second interpretation is that the bot chooses to go to pick-up points more often resulting in acquiring more powerful guns which in turn are used to kill the opponents. Players drop their current weapon on the map when they are

killed. If the bot then followed the path over the dead opponent it would pick up its weapon so the first interpretation of the results is more likely accurate. The *0-XP*, which acts at random, manages to pick up 2 weapons per game which suggests that the bots will pick-up weapons by chance as they are traversing the map.

Table 4.3 shows the pick-ups per game for weapons, ammunition and health items for each level of experience for comparison. We can observe that the same trend occurs for picking up ammunition and health in which the best performing bots pick up more on average. Pick-ups help the bots to stay alive for longer and, likewise, the longer the bot is alive the more chance it will have of picking up items in its path.

Experience	**Weapons**	**Ammunition**	**Health**
140-XP	4.59	5.61	8.00
105-XP	1.74	1.08	1.66
70-XP	2.27	1.08	1.54
35-XP	4.65	5.08	8.6
0-XP	1.87	1.57	2.27

Table 4.3: Average pick-ups per game for weapons, ammunition and health items.

Figure 4.6 shows the total reward received per game for each of the different levels of experience. The trends in this figure are as expected given the large rewards associated with injuring or killing the opponents. The random actions of *0-XP* receive the least amount of reward per game. The highest average total reward was achieved by the bot with the most experience. This bot has had more time to explore new actions and exploit the knowledge that it has built up. The fact that the total reward does not increase uniformly as experience is increased could be a result of the strategy deployed by the human opposition. We discuss this in more detail in the next section.

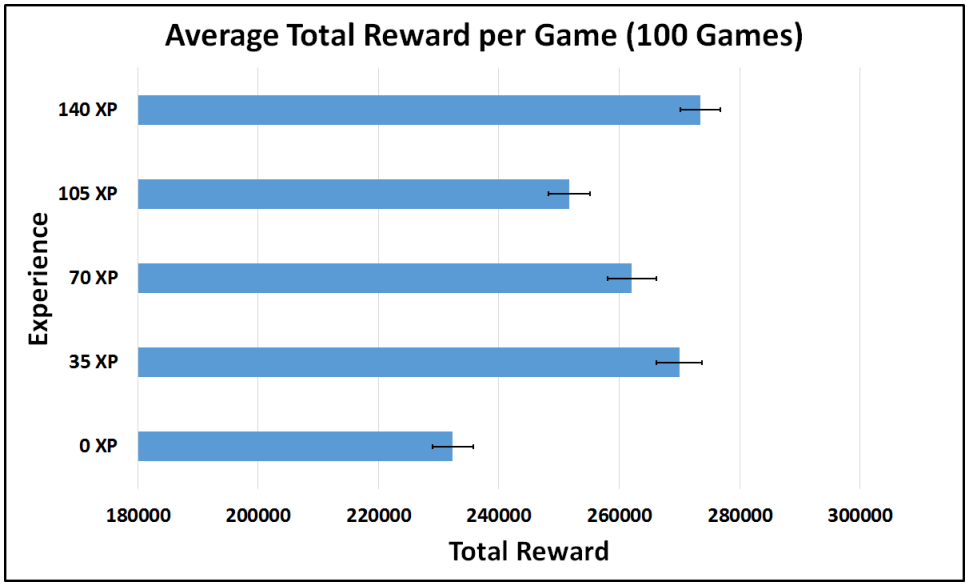

Figure 4.6: Mean and standard error of total reward received per game over 100 games.

4.3.3 Discussion

One challenging aspect of this work is in deciding how best to evaluate the performance of the proposed bot. Some issues arise when we begin to train the bot by playing many games against a human player. The human player for instance, can quickly became familiar with the game controls and learn how to play well. This can lead to instances in which the human player "goes easy" on the bot in order to give it more of a chance when, in fact, the bot is making poor decisions and inadvertently being rewarded for them. Also, if the human player takes a ruthless approach then they can kill the bot several consecutive times without giving it a chance to learn, for example, that shooting a player on sight is a good strategy. From our experiences we now believe that experiments involving human opposition would need to be both extensive and varied. Connecting the bot, as if it was human, to an online network could be a useful approach to explore in the future. This work has successfully used fixed-strategy bots as a benchmark against which to assess the learning progress of the Sarsa-Bot. By freezing the learning experience at different stages we have shown that different strategies are

being deployed by the Sarsa-Bot as it gains experience. We have also identified that there is not a uniform increase in performance over time which could be due to the misattribution of reward.

The results presented here are from initial experiments that took place in order to validate the use of a reinforcement learning algorithm into the logic of an agent in a FPS game. These results have given rise to very interesting questions about meaningful evaluation procedures and complex implementation issues, as were raised in Section 4.2.6.

4.3.4 Analysis of Findings

In conventional RL settings, the agent can choose a single action in each state, which leads to a new state and a possible reward. However, in the setting we are considering, the bot must make decisions in real-time (at a rate of 4 time steps per second) and when it decides to perform an action, it may take several time steps to complete, such as performing a dodge or a jump. While one action is underway, the bot can decide to take another action, so that two actions may continue at once. Therefore, when a reward is received, it may not relate to the most recently taken action. For example, if the bot accidentally jumps off of a ledge, then fires its gun in the next time step, and subsequently dies from the jump, it will incorrectly associate the negative reinforcement from dying with the action of firing the gun.

While we have been able to demonstrate some success in designing a bot that can learn from experience as it plays against a human, we believe that it is necessary to extend the standard reinforcement learning framework to deal with the real-time, multiple-action, complex-reward setting that is required for these games and such developments will be of value to other real-time scenarios in which reinforcement learning can be applied. In the next chapter, we describe how we restructured and developed refinements for this architecture in an attempt to address some of the shortcomings that we identified.

4.3.5 Multi-Factor versus Simple Reward Signal

	With Multi-Factor Reward Avg (Std Dev)	**With Simple Reward Avg (Std Dev)**
Deaths	138.70 (± 17.60)	140.15 (± 13.81)
Suicides	1.10 (± 1.12)	1.55 (± 0.94)
Killed Opponent	5.85 (± 1.95)	6.050 (± 1.93)
Killed by Opponent	137.60 (± 17.55)	138.60 (± 13.73)
Kill Streak	1.15 (± 0.37)	1.30 (± 0.57)
Weapons Collected	16.75 (± 4.17)	15.50 (± 4.89)
Ammo Collected	7.60 (± 2.66)	7.35 (± 2.87)
Health Collected	15.85 (± 3.79)	15.25 (± 3.93)
Adrenaline Collected	29.55 (± 5.46)	26.80 (± 7.88)
Time Moving	1841.82 (± 135.15)	1813.72 (± 132.71)
Time Shooting	17.39 (± 2.59)	18.67 (± 2.89)
Traveled Distance	276093.11 (± 19143.00)	283171.88 (± 15840.58)

Table 4.4: Performance measures for Multi-Factor Reward versus Simple Reward. The differences are not statistically significant at the 10% level.

In this section, we present a comparison and statistical significance test between the Sarsa-Bot using the 10-factor reward signal, described earlier in Table 4.2, and one that uses a simple kill/death reward signal. The simple reward signal is +1 for an enemy kill, -1 for the agent being killed and 0 otherwise. The experimentation involved twenty runs of both Sarsa-Bots separately playing against two Level 2 native bots in hour-long Deathmatch games. The learning rate, α, was set to 0.2 and the discount parameter was set to 0.8. An ϵ-greedy strategy with ϵ set to 0.2 was used as the action selection policy. These parameters were described earlier in Section 2.2.3.2. The purpose of this comparison is to see how much influence, if any, the reward signal has on the overall performances of the Sarsa-Bot.

	P-Value	Confidence
Deaths	0.7735	22.64%
Suicides	0.1776	82.23%
Killed Opponent	0.7466	25.34%
Killed by Opponent	0.8420	15.79%
Kill Streak	0.3302	66.98%
Weapons Collected	0.3899	61.00%
Ammo Collected	0.7768	22.32%
Health Collected	0.6259	37.41%
Adrenaline Collected	0.2085	79.15%
Time Moving	0.5110	48.90%
Time Shooting	0.1508	84.92%
Traveled Distance	0.2106	78.94%

Table 4.5: Statistical significance test of differences between a multi-factor and simple reward. The differences are not statistically significant at the 10% level.

From Table 4.4 we can see that the averages and standard deviations in each performance measure are very similar when using both the multi-factor reward and the simple reward design as described above. Since the bot only has a simplistic perception design, in which it is measuring its own levels of health and ammunition as well as opponent visibility, the reward values do not appear to be directly affecting the overall performance that it achieves.

We ran a two-tailed unpaired t-test on all of the performance values and the results are presented in Table 4.5. This shows that there are no statistical differences in the performance results, at the 10% level, when different reward values are used.

4.4 Chapter Summary

This chapter has described the development details of our Sarsa-Bot behavioural architecture for controlling NPCs in the game Unreal Tournament 2004 using reinforcement learning. The motivation for this work is to produce bots that can

adapt their behaviour from in-game experience and gear their decision-making towards receiving the most reward possible. The goal for facilitating the bot to learn by itself is to produce diverse and unpredictable bot behaviour. The bots' high-level learning behaviours are complimented with hard-coded human knowledge such as not shooting the gun when there is not a target in sight. We consider such hard-coded logic to model the kinds of instructions that a novice player would be given which are separate from the strategies that the player will learn through playing the game. The second half of this chapter outlined our experimentation setup in which we trained the Sarsa-Bot against a human opponent. Learning was then frozen and the Sarsa-Bot was deployed against fixed-strategy scripted opposition in order to analyse the learning performance of the agent. This provided evidence of learning and raised some questions regarding the assignment of reward when it is being received from multiple sources. We also ran experiments comparing the multi-factor reward with a more simple reward and concluded that there were no statistical differences in the performance results which meant that the reward values were not having a direct impact on performance in the current architecture design. This led to the identification of a number of refinements for the behavioural architecture which are described and analysed in the next chapter.

Chapter 5

DRE-Bot Architecture

In this chapter, we describe our *Danger Replenish Explore* reinforcement learning behavioural architecture, called *DRE-Bot*[1], that we developed to control an NPC in the FPS game Unreal Tournament 2004. We give an overview of the architecture design which includes some implementation details, a discussion on the use of the SARSA(λ) algorithm, the multiple learner hierarchy and the new mode-switching mechanism. We then present our refined design of the states, action and rewards. We outline our experimentation methodology before presenting and analysing the results. We conclude the chapter with a discussion and a summary of the work presented.

5.1 Overview

Traditional approaches to game AI, some of which were described in Section 2.3.1, can often lead to predictable gameplay that can be exploited by knowledgeable players. As mentioned in the previous chapter, the control of an FPS bot in a multi-objective 3D environment is a difficult task with constant real-time decision making being required for a variety of different tasks such as path-finding and combat. The content of this chapter and the previous one aims to address our first research question and first research objective from Chapter 1 (Section 1.3). This involves developing bots that can learn their own strategy and continually

[1]Some of the work in this chapter was first published in Glavin & Madden [2012]

adapt over time, as opposed to being given strict deterministic rules. This also involves testing the capability of the bot and outlining the benefits and challenges of such an approach.

5.2 DRE-Bot Architecture Design

In this section we describe the use of the SARSA(λ) algorithm in the DRE-Bot architecture and then discuss the new implementation details such as the introduction of multiple learners and a high-level mode-switching mechanism. We also present details of the refined states, actions and rewards.

5.2.1 On the use of the SARSA(λ) Algorithm in DRE-Bot

One of the first updates to the design involved us adding the use of *eligibility traces* to the algorithm. That is, we used the SARSA(λ) algorithm [Sutton & Barto, 1998], described in Section 2.2.3.3, as opposed to the original one-step SARSA used in the Sarsa-Bot architecture. The reason for this was its reported success in the literature for similar problems. Most notably, McPartland & Gallagher [2011] used the SARSA(λ) algorithm on a purpose-built FPS game, as described earlier in Section 2.5.2, and reported on its success. Their experimentation was carried out using a simplified implementation of an FPS game. One of our objectives, described earlier in Section 1.3, is to see if such reported success would also be evident when using a commercial 3D FPS game and if this algorithm would be suitable for creating a competent computer-controlled player in such a game.

Eligibility traces, which were described in Section 2.2.3, keep a record of recently visited state-action pairs and mark them as being eligible for undertaking learning changes. They can considerably decrease the amount of time it takes for learning as each trial will provide much more information about how the algorithm reached the goal. A comparison between the use of eligibility traces and omitting them is shown in Figure 5.1, which is based on Figure 7.12 from Sutton & Barto [1998]. In this simple grid-world example, the one-step method only strengthens the last action of the sequence. The eligibility trace method, however, strengthens many actions of the sequence. The degree at which the strengthening is occurring

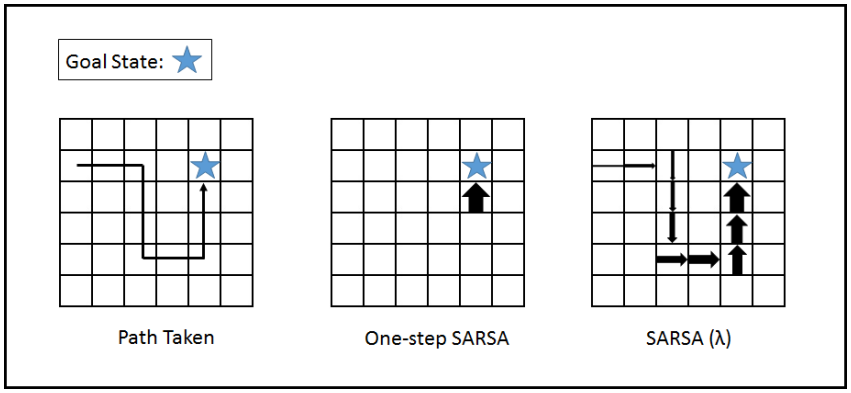

Figure 5.1: Comparison of one-step SARSA and SARSA(λ). Based on Figure 7.12, page 181 of Sutton & Barto [1998]

is signified by the size of the respective arrows. The extent of the strengthening is dependent on the discount parameter, γ, and the eligibility trace parameter λ. The SARSA(λ) algorithm implementation was firstly tested using a simple maze environment for verification purposes. Once this was competed, we incorporated the changes into the logic of the DRE-Bot.

5.2.2 Implementation Details

This section describes the inclusion of different learning components in the DRE-Bot architecture and a high-level mechanism for switching between these during gameplay. The bot assesses its current circumstances and then selects a suitable learner in order to decide what actions to take.

5.2.2.1 Multiple Learner Hierarchy

One of the first refinements that we made to the Sarsa-Bot's implementation was to include multiple learners. Each of these learners work independently from one another and each of them assumes control based on a high level controller that carries out checks to determine which situation the bot is currently in. This hierarchical architecture makes it possible to add additional learners to the logic of the bot or to remove existing learners. The Danger, Replenish and Explore learners all work independently from one another. If we removed the Danger learner

then the bot would only concentrate on replenishing supplies and navigating the map. Conversely, we could add new learners for game-specific objectives. It is also possible to freeze the learning in certain *modes*[1] while continuing to learn in other modes if necessary. Creating individual learners for specific situations increases the speed at which overall learning occurs and also makes it easier to refine the different states and actions associated with particular modes. Given that FPS games involve various scenarios with multiple objectives, this type of hierarchical architecture is beneficial for organising the functionality of the bot.

5.2.2.2 DRE-Bot Mode-Switching

We identified three high-level scenarios (which we will refer to as "modes") of FPS games. These were *Danger*, *Replenish* and *Explore* mode. A mode-switcher carries out conditional checks to determine which mode the bot is currently in.

The DRE-Bot architecture switches between modes as follows. The first check is whether or not the bot has enough ammunition and health. If either are below a certain level, that is, when ammunition and/or health is at a low (40%) or critical (20%) level regardless of seeing an opponent or being damaged, then the DRE-Bot automatically enables *Replenish* mode. If the levels of health and ammunition are sufficient, the next check is to see if an opponent is visible or if the bot is currently being damaged. If neither of these return True then the bot is in *Explore* mode by default. Each of these modes consist of one independent learner which has its own states, actions and rewards as illustrated in Figure 5.2.

Each of the learners in this implementation use the SARSA(λ) algorithm and each has its own Q-value lookup table. One of the benefits of this multiple-learner architecture is that different learning algorithms could be used for the different modes if it was deemed to be advantageous to the learning process. However, for this implementation, the same algorithm is used in all three modes. The reason for this is that we would like to test the reinforcement learning algorithm on each of the different modes.

[1] Modes are the high-level scenarios which are described in Section 5.2.2.2

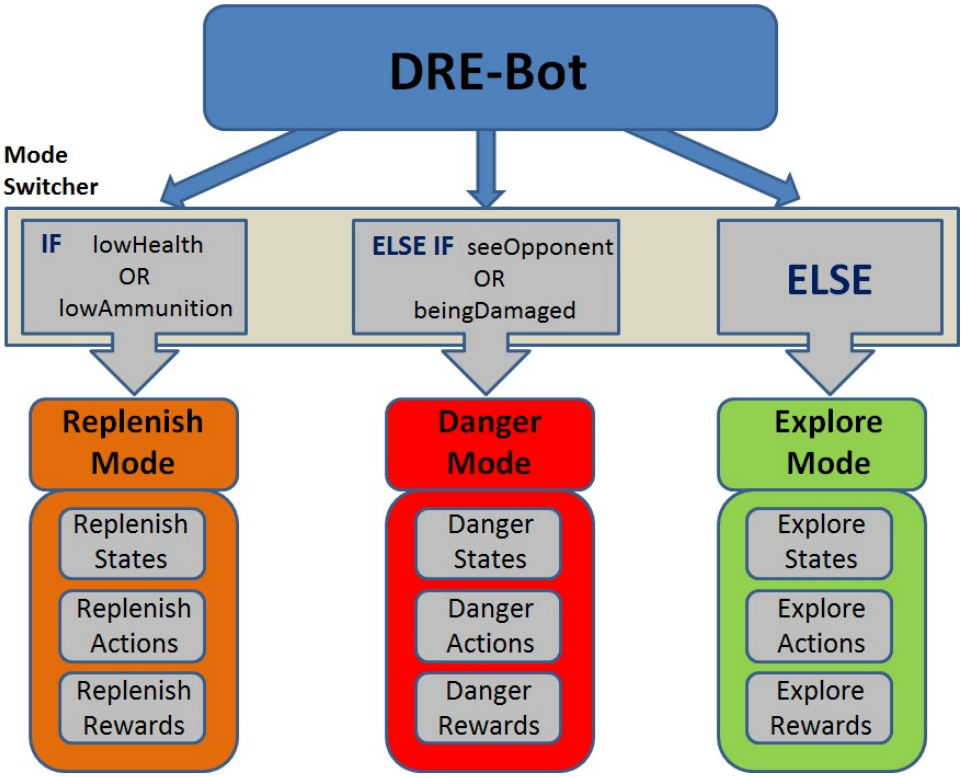

Figure 5.2: DRE-Bot multiple learner architecture

5.2.3 State, Action and Reward Design

The states are made up of a series of checks from the environment. Checks such as "*Is an opponent visible?*", "*Am I being hit?*", "*Do I have low health?*" are combined together to make up the individual states. All of these checks correspond to the bot's perception of the world around it. Every new state that is added increases the complexity of the learner. For this reason, it was important to design high-level states that capture the most important information from the game. Too few states could lead to consistently poor performance whereas too many could drastically increase the amount of time it takes to learn an effective policy. In the following sections we discuss the individual states, actions and rewards for each of the modes.

The actions that we designed for the bot consist of single activities which can be carried out during the game. Examples of these include activities such as

continuous movement, jumping, shooting and changing weapon. The number of actions and their corresponding complexity will have a direct impact on the proficiency of the bot. The bot is, of course, limited by the choices that it has available to it and will carry out the actions exactly as they are defined. If the bot has a large number of actions to choose from then it will go through a longer period of performing poorly as it explores the best actions to choose in each state. The state-action space will grow considerably as actions are added. We have designed a limited number of high-level actions, based on human knowledge of the game, in order to enable the DRE-Bot to compete in a Deathmatch game. The actions available in each of the modes are sophisticated and high-level. We believe that this is an important part of the design in order for the DRE-Bot to competently function in the environment. We want the bot to learn the best actions to take in the given circumstances and have provided it with enough choices to adapt its behaviour with the long-term goal of maximising its reward. In this sense, it is designed to be analogous to a novice human player that understands the dynamics of each action but still must learn to use them at appropriate times to produce an effective strategy.

The reward[1], similar to the multi-factor reward of Sarsa-Bot, is acquired through an accumulation of reward checks. Each check, from all of the modes, are listed below with their corresponding values in Table 5.1. These are listed together with a tick indicating that this check is run in the specific mode. For instance, in *Explore* mode, the checks for causing and receiving damage or killing/dying would not be run as these are only considered in *Replenish* and *Danger* mode. If a reward check returns True then the value for that reward is added to the total. The reward is then returned as a single number which represents the reward received for that time step. The rewards checks include: being healthy or not, colliding or not, moving or not, seeing an opposing player or not, causing or receiving damage, killing or being killed, picking up items and gaining adrenaline. Adrenaline can be gained by either picking up *pills* on the map or completing tasks such as a *killing spree* (multiple successive kills without dying) or ending an opponent's killing spree. Gaining adrenaline is given a substantial reward as

[1]We also carried out a direct comparison between this multi-factor and a simple reward in Section 5.3

it is indicative of good play in the game.

The following sections outline the states and actions that are available for each the three different modes.

Check	Reward	Danger	Replenish	Explore
isHealthy	+ 0.0001	✓	✓	✓
isNotHealthy	- 0.0001	✓	✓	✓
isNotColliding	+ 0.00001	X	X	✓
isColliding	- 0.00001	X	X	✓
isMoving	+ 0.00001	X	X	✓
isNotMoving	- 0.00001	X	X	✓
seeOpposingPlayer	+ 0.0001	✓	✓	X
isCausingDamage	+ 0.1	✓	✓	X
isBeingDamaged	- 0.1	✓	✓	X
killedOpponent	+ 1	✓	✓	X
killedByOpponent	- 1	✓	✓	X
pickedUpItem	+ 0.1	X	✓	X
gainedAdrenaline	+ 0.2	✓	✓	X

Table 5.1: DRE-Bot reward signal

5.2.4 Danger Mode

Firstly, in Table 5.2, we can see the different checks and values which are used to represent the *Danger* state. *BeingHit* will return True if the bot is currently being damaged by an opponent. If the bot is colliding with map geometry then the *Bumping* check will return True. *HearingNoise* also returns a value of either True or False. The *Distance* check measures how far an opponent is from the bot. These values are *near*, *medium*, *far* or *no* which represents that no opponent is visible at the moment. The discretised values for distances are hard-coded and we designed them specifically for the map that is currently being played. There

are 32 different *Danger* states in total ($2 \times 2 \times 2 \times 4$).

Check	Values
BeingHit	True / False
Bumping	True / False
HearingNoise	True / False
Distance	near / medium / far / no

Table 5.2: DRE-Bot Danger states

In *Danger* mode, the bot can shoot its primary weapon, shoot its secondary weapon, go to the location of the last opponent that was seen (if any), stop all movement, dodge in a random direction, jump with a random amount of elevation, turn and face a visible opponent player or turn randomly, and change its weapon.

Action	The bot will:
ShootPrimary	Shoot opponent in primary mode.
ShootSecondary	Shoot opponent in secondary mode.
LastSeenOpponent	Go to the last seen opponent.
StopMovement	Stop all movement completely.
Dodge	Perform dodging maneuver.
Jump	Perform random jump.
FacePlayerOrTurn	Face opponent or turn randomly[1].
ChangeWeapon	Change to a different weapon.

Table 5.3: DRE-Bot Danger actions

These actions are summarised in Table 5.3. In a similar fashion to the Sarsa-Bot, there are some hard-coded rules about taking actions such as not being able

[1]The bot will turn randomly if it cannot currently see an opponent

to shoot a gun without an opponent being visible. The *Danger* actions are designed to deal with scenarios in which the bot is being damaged and in danger of being killed by an opponent. For this reason, the actions include ones for engaging in combat, avoiding enemy fire, facing enemies and moving towards them. The bot is designed to continue to engage in combat unless is it not equipped to do so. If the DRE-Bot's health or ammunition drops below a certain level it will automatically switch to *Replenish* mode.

5.2.5 Replenish Mode

The checks and values for the *Replenish* states are shown in Table 5.4. These include checks for seeing an enemy, seeing a pickup and hearing a pickup. It is important to include seeing an enemy in this mode as the bot still needs to defend itself when it is looking to replenish its health and ammunition.

Check	**Values**
SeeEnemy	True / False
SeePickup	True / False
HearPickup	True / False
Levels	LA/LH/LA&LH/CA/CH/ CA&CH/CA&LH/LA&CH

Table 5.4: DRE-Bot Replenish states

The levels of health and ammunition are also taken into account. It should be noted that only one of these has to be below the threshold for the bot to be in *Replenish* mode therefore the levels for health can be *low* (LH), *critical* (CH) or *OK* (in which case it has ammunition that has a critical or low level). Ammunition also has the levels *low* (LA), *critical* (CA) or *OK*. Low corresponds to having 40% remaining whereas critical corresponds to having 20% remaining with *OK* again representing sufficient levels of ammunition. There are 64 possible states when in *Replenish* mode ($2 \times 2 \times 2 \times 8$).

The *Replenish* actions are listed in Table 5.5. These include shooting the primary weapon, shooting the secondary weapon, continuous movement, going to a visible pickup, recording the location of a visible item, going to an item from those that are stored and turning away from a visible opponent to escape. These actions are designed to deal with scenarios in which the bot has low health or ammunition and needs to replenish their supplies. Shooting actions were also included so that the bot could defend itself in such situations. These two shooting actions are the same as those found in the *Danger* learner.

Action	**The bot will:**
ShootPrimary	Shoot opponent in primary mode.
ShootSecondary	Shoot opponent in secondary mode.
Move	Move continuously straight ahead.
GoToPickup	Go to pickup, if visible.
RecordItem	Record location of visible pickup.
GoToKnownItem	Go to recorded pickup location.
EscapeOpponent	Turn from opponent and run.

Table 5.5: DRE-Bot Replenish actions

5.2.6 Explore Mode

The checks and values for the *Explore* states are shown in Table 5.6. This mode is designed for when the bot is not in any danger and has good health and ammunition. There are only two checks carried out with 6 possible states altogether (3×2). The movement check can return: *walking*, *running* or *stopped*. Whether or not the bot is currently crouched is also checked. The *Explore* state space represents features of how the bot is moving around the environment.

The final table of actions, Table 5.7, lists the actions that are available to the bot while in *Explore* mode. These include constant motion around the map while running, constant motion around the map while walking, turning left, turning

right, stopping all movement and switching between a crouched/un-crouched position.

Check	Values
Movement	Walk / Run / Stopped
Crouched	True / False

Table 5.6: DRE-Bot Explore states

Action	The bot will:
RunAround	Move continuously while running.
WalkAround	Move continuously while walking.
TurnLeft	Turn left by a random amount.
TurnRight	Turn right by a random amount.
StopMovement	Stop all movement completely.
Crouch/Uncrouch	Switch between crouched or un-crouched.

Table 5.7: DRE-Bot Explore actions

5.2.7 Discussion

The DRE-Bot architecture is a result of several refinements made to the initial Sarsa-Bot architecture. Firstly, multiple individual learners have been introduced to deal with specific scenarios in the game. Each of these bot modes have their own state space, action space and reward signal. These work independently from each other and can be added or removed as required. The three modes of the current implementation include: being in danger and having to engage in combat, having low supplies and needing to gather more, exploring the map without any concerns. Secondly, we implemented the updated version of the SARSA algorithm which uses eligibility traces (SARSA(λ)). When a state-action pair is visited it is

marked as being eligible to receive some of the reward from future updates. This eligibility decays over time. Finally, the state representations were improved to provide more information for the bot. The new states were designed based around the idea of the bot perceiving different scenarios of gameplay. For each scenario, or mode, the information most relevant is presented to the bot. The actions and rewards were also updated and these are also mode-specific. The following sections describe the experimentation that we carried out using the DRE-Bot architecture.

5.3 DRE-Bot: Multi-Factor vs Simple Reward

	DRE-Bot Multi-Factor Avg (Std Dev)	**DRE-Bot Simple Avg (Std Dev)**
Deaths	221.15 (± 10.70)	220.60 (± 13.67)
Suicides	14.30 (± 3.39)	14.20 (± 3.50)
Killed Opponent	**216.25 (± 22.31) ***	202.50 (± 23.15)
Killed by Opponent	206.85 (± 10.87)	206.40 (± 13.27)
Kill Streak	6.20 (± 1.40)	5.85 (± 1.79)
Weapons Collected	**169.95 (± 29.07) ****	152.40 (± 23.72)
Ammo Collected	118.40 (± 15.14)	116.15 (± 10.64)
Health Collected	145.20 (± 21.68)	144.05 (± 19.25)
Adrenaline Collected	**78.65 (± 24.89) ****	64.30 (± 16.50)
Time Moving	3100.09 (± 42.80)	3086.92 (± 90.81)
Time Shooting	1869.10 (± 45.27)	1851.87 (± 69.91)
Traveled Distance	964059.25 (± 18754.74)	962636.30 (± 26913.40)

Table 5.8: Performance measures for DRE-Bot using the multi-factor and simple reward. Level of confidence: *** = 99%, ** = 95%, * = 90%

The first set of experiments involve comparing the performance results of DRE-Bot using a multi-factor reward with the use of a simple reward. The motivation of these experiments is to see if the mult-factor reward leads to any

significant performance improvements over a simple reward signal. The multi-factor reward was described earlier in Table 5.1. The simplistic reward is +1 for an enemy kill, -1 for the agent being killed and 0 otherwise.

The experimentation involved twenty runs of both DRE-Bots (one using multi-factor reward and one using a simple reward) separately playing against two Level 2 native bots in hour-long Deathmatch games. The total experiment time for carrying out this comparison was 40 hours. Apart from the reward signal, all settings were identical for each bot. The learning rate, α, was set to 0.2 and the discount parameter, γ, was set to 0.8. An ϵ-greedy strategy with ϵ set to 0.2 was used as the action selection policy. These parameters were described earlier in Section 2.2.3.2. The eligibility trace parameter, λ, was set to 0.9.

We can see from Table 5.8 that the DRE-Bot using the multi-factor reward is never worse than the simple reward and performs slightly better, with statistical significance, in some performance measures. The number of asterisks in the table indicates the statistical significance level of the differences with α equal to 0.01 (***), α equal to 0.05 (**) and α equal to 0.1 (*) from a two-tailed unpaired t-test. These results show that having a reward signal that uses multiple sources of reward leads to performances with a marginal statistically significant improvement when it comes to killing the opponent, collecting weapons and collecting adrenaline. For this reason, we use the multi-factor reward design for the experimentation in the remainder of this chapter.

5.4 DRE-Bot vs Sarsa-Bot

In order to verify that the changes to the architecture have led to an improvement in performance, we directly compared the performances of DRE-Bot and Sarsa-Bot using the same methodology. In this case, both were using the multi-factor reward signal. The experiment settings and parameters are as they were in the previous section with twenty runs of both bots separately playing against two Level 2 native bots in hour-long Deathmatch games. We can see from Table 5.9 that while the DRE-Bot does die more times than the Sarsa-Bot (both from the opponent and suicides) it manages to significantly out-perform it on each of the other different performance measures at a 99% confidence level using a two-tailed

unpaired t-test. It is likely that the Sarsa-Bot dies less than the DRE-Bot as it spends a lot less time moving, travelling only a third of the distance of DRE-Bot on average, and is therefore likely to have less encounters with opponents during the matches.

	DRE-Bot Avg (Std Dev)	**Sarsa-Bot Avg (Std Dev)**
Deaths	221.15 (± 10.70)	**138.70 (± 17.60) *****
Suicides	14.30 (± 3.39)	**1.10 (± 1.12) *****
Killed Opponent	**216.25 (± 22.31) *****	5.85 (± 1.95)
Killed by Opponent	206.85 (± 10.87)	**137.60 (± 17.55) *****
Kill Streak	**6.20 (± 1.40) *****	1.15 (± 0.37)
Weapons Collected	**169.95 (± 29.07) *****	16.75 (± 4.17)
Ammo Collected	**118.40 (± 15.14) *****	7.60 (± 2.66)
Health Collected	**145.20 (± 21.68) *****	15.85 (± 3.79)
Adrenaline Collected	**78.65 (± 24.89) *****	29.55 (± 5.46)
Time Moving	**3100.09 (± 42.80) *****	1841.83 (± 135.15)
Time Shooting	**1869.10 (± 45.27) *****	17.40 (± 2.59)
Traveled Distance	**964059.25 (± 18754.74) *****	276093.11 (± 19143.00)

Table 5.9: Performance measures for DRE-Bot in comparison to Sarsa-Bot. Level of confidence: *** = 99%, ** = 95%, * = 90%

5.5 DRE-Bot Parameter Search

5.5.1 Experiment Details

These experiments involve connecting the DRE-Bot to a Deathmatch server against two fixed-strategy bots from the game. The games were played on the Training Day map and the games ended when the DRE-Bot had died 200 times.

Each single game run is played out in real time with the duration of the game being approximately one hour. The game time is approximate as the stopping condition of each game is dependent on the death count of the DRE-Bot. Given the hierarchical nature of DRE-Bot, it is expected to adapt quickly in practice as a result of the small number of states and actions. Both the discount parameter, γ, and the eligibility trace parameter, λ, were varied to note what effect they had on the overall performance of the SARSA(λ) algorithm. The learning rate, α, was fixed at 0.2 for all games. The learning rate determines how quickly new information overrides older information. As the value approaches 1 it will only consider recent information whereas if it was set to 0 it would not learn anything. Setting the value to 0.2 means that the bot will learn slowly from experience. An ϵ-greedy action-selection strategy with ϵ set 0.2 was also used. Setting ϵ to 0.2 means that random actions will be chosen 2 out of every 10 times. The results reported here are averaged over five runs of the sixteen games (each individual game varies the lambda (λ) and gamma (γ) parameters).

5.5.2 Results

The discount rate (γ) and eligibility trace (λ) parameters are varied as shown in Table 5.10 below. There are 16 combinations of both parameters with each having the values: 0.0, 0.3, 0.6 or 0.9. The results are presented and analysed in five different sections. The first four sections each look at a fixed discount rate with varying lambda values. For each game, averaged over five runs, we will observe the minimum, maximum and average time alive for the bot. We also report the minimum, maximum and average reward received for the duration of a single life. The *kill-death difference* (KDD), calculated by subtracting the total number of deaths from the total number of kills, is also reported and graphed for each game. The fifth section presents a comparative analysis of all of the results together.

5.5.2.1 Game 1 to Game 4 (Discount 0.0)

Setting the discount parameter to 0 makes the bot opportunistic in that it is only concerned with current rewards. The best performance, when the discount pa-

Game	γ	λ	Game	γ	λ
1	0.0	0.0	**5**	0.3	0.0
2	0.0	0.3	**6**	0.3	0.3
3	0.0	0.6	**7**	0.3	0.6
4	0.0	0.9	**8**	03	0.9
Game	γ	λ	**Game**	γ	λ
9	0.6	0.0	**13**	0.9	0.0
10	0.6	0.3	**14**	0.9	0.3
11	0.6	0.6	**15**	0.9	0.6
12	0.6	0.9	**16**	0.9	0.9

Table 5.10: Individual games that vary the γ and λ parameters.

Discount γ : 0.0				
Game Number (λ):	**1 (0.0)**	**2 (0.3)**	**3 (0.6)**	**4 (0.9)**
Average Time Alive:	15.5	16.8	16.5	16.2
Min Time Alive:	2.1	2.6	1.9	2.3
Max Time Alive:	79.6	85.0	63.9	75.2
Average Reward:	1.5	1.9	2.1	1.9
Min Reward:	-1.9	-1.7	-1.9	-1.8
Max Reward:	13.7	14.6	14.9	15.8
Final Kill Difference:	75	129	144	132

Table 5.11: Statistics for games 1 to 4 with γ set to 0.0.

rameter was set to this value, was achieved in Game 3 which received a final KDD of 144 as shown in Table 5.11. It should be noted that the lambda parameter does not have any effect when the discount parameter is set to 0 as the eligibility trace update will alway equate to 0 in this case. Game 2 and Game 4 resulted in similar final KDDs of 129 and 132 respectively. The worst performance of the four bots was from Game 1. The final KDD of this bot was almost half that of the others with a value of 75. The average reward that it received was also notably

lower than the others. The average amount of time that each bot stays alive does not correlate exactly to performance (w.r.t KDD), however, the worst performing bot remains alive, on average, for a second less than the others. Figure 5.3, below, shows the KDD for Game 1 to Game 4 that are averaged over five runs.

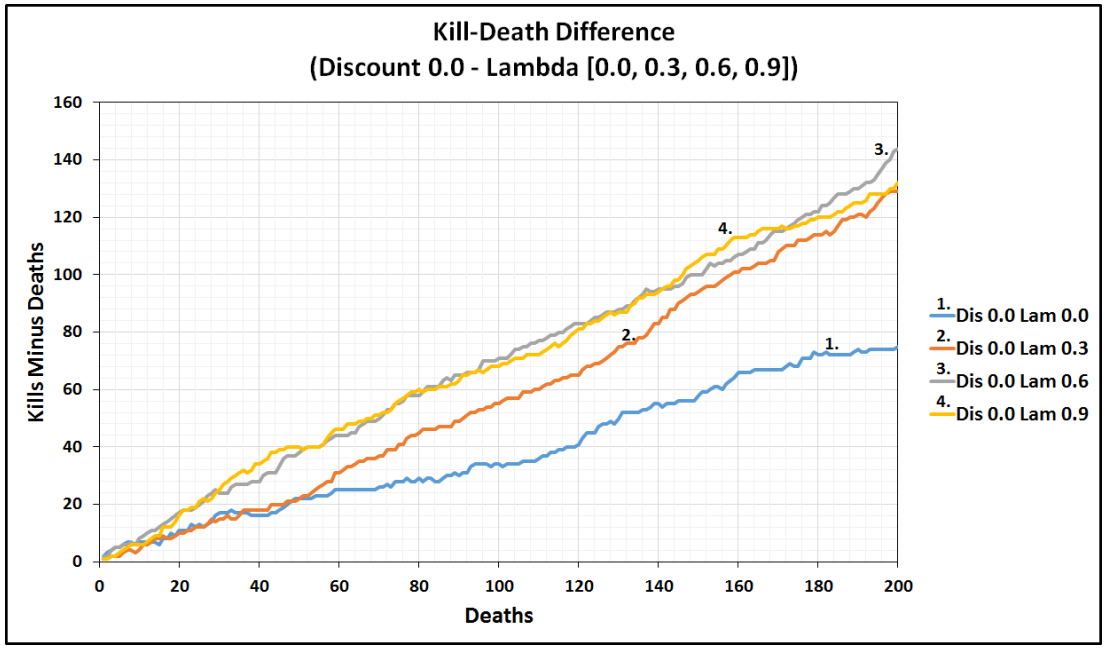

Figure 5.3: Kill-death differences for games 1 to 4 averaged over five runs.

Each of the bots achieve a positive KDD on average at the beginning of the games and this continues to increase, at different rates, as the games progress. After approximately 15 deaths, the games with the higher values for lambda (Game 3 and Game 4) begin to outperform the other two (Game 1 and Game 2). Game 2, however, gradually improves its performance towards the end of the game whereas Game 1 does not reach the performance levels of the others.

5.5.2.2 Game 5 to Game 8 (Discount 0.3)

Table 5.12, above, shows the resulting statistics for Game 5 to Game 8. During these games the discount parameter was set to 0.3. While these bots are not as

Discount γ : 0.3				
Game Number (λ):	**5 (0.0)**	**6 (0.3)**	**7 (0.6)**	**8 (0.9)**
Average Time Alive:	15.9	15.8	15.7	16.2
Min Time Alive:	2.3	2.3	2.2	2.6
Max Time Alive:	71.6	65.7	80.8	91.5
Average Reward:	1.9	1.5	1.8	1.9
Min Reward:	-1.9	-1.9	-1.8	-1.9
Max Reward:	14.9	14.9	15.0	16.0
Final Kill Difference:	127	83	113	132

Table 5.12: Statistics for games 5 to 8 with γ set to 0.3.

"optimistic" as those in the previous set of games, there is still a greater emphasis on the most recent reward as opposed to striving for a large long-term reward (which is the case as the discount parameter approaches 1.0). Game 5 and Game 8 achieved the best performances with KDDs of 127 and 132 respectively. As with the previous set of games the average time alive for each of the bots remains consistent at approximately 16 seconds. The maximum duration that the bots managed to stay alive for were each over a minute with the best performing bot, from Game 8, staying alive for over a minute and a half. Figure 5.4 displays the KDDs for Game 5 to Game 8. Game 5, Game 7 and Game 8 have a similar performance level up until midway through the game. At this point, the performance of Game 7 begins to degrade slightly but the others retain a closely matched performance. The KDD of Game 6 breaks away from the others after approximately 30 deaths and does not manage to achieve the same level of performance as the others for the duration of the game. This is likely due to poor decision-making at the beginning of the game and the slow rate of learning.

5.5.2.3 Game 9 to Game 12 (Discount 0.6)

Table 5.13 shows the results for games 9 to 12 which were averaged over five runs. Unlike the previous sets of games, the final KDD for each of these games, with the discount parameter set to 0.6, are very similar. The final KDD for each game

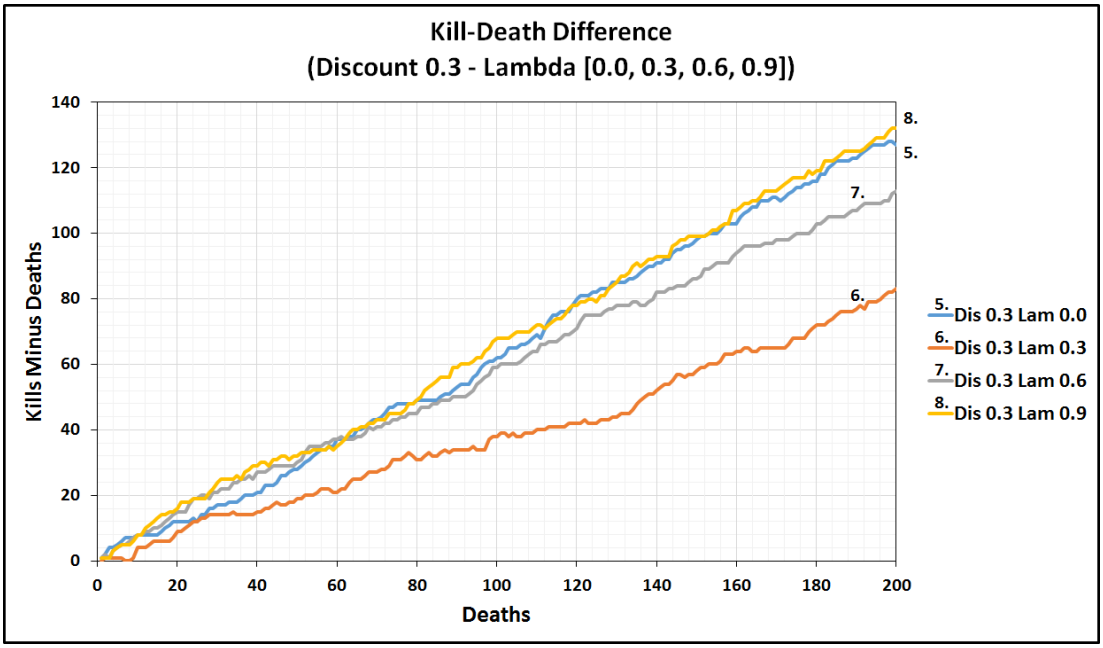

Figure 5.4: Kill-death differences for games 5 to 8 averaged over five runs.

Discount γ : 0.6				
Game Number (λ):	**9 (0.0)**	**10 (0.3)**	**11 (0.6)**	**12 (0.9)**
Average Time Alive:	15.8	16.3	16.4	15.8
Min Time Alive:	1.4	2.4	2.6	2.3
Max Time Alive:	94.5	74.1	82.2	62.7
Average Reward:	1.7	1.8	1.8	1.8
Min Reward:	-1.9	-1.9	-1.9	-1.8
Max Reward:	17.7	14.7	14.6	13.6
Final Kill Difference:	111	119	118	115

Table 5.13: Statistics for games 9 to 12 with γ set to 0.6.

are within 9 kills of each other with three of the four games being within 4 kills of each other. The average duration for the life of each bot is again approximately 16 seconds. The bot that manages to stay alive for the longest and receive the

most reward for a single life has the lowest final KDD although the values fall within a similar range of each other as mentioned earlier. We can see the progress of the KDDs for each game, over the 200 deaths, in Figure 5.5 below. Game 9 starts to outperform the others after 20 deaths but the performance drops mid-way through the game and it ends up with the worst KDD of the four games although there is not a large difference overall. Game 10, Game 11 and Game 12 have almost identical KDDs until 100 deaths when Game 10 begins to weaken in performance before recovering and marginally having the best KDD at the end. Game 9 slightly outperforms the others in the early stages but as the game progresses it goes from being the best performing to the worst performing after 200 deaths.

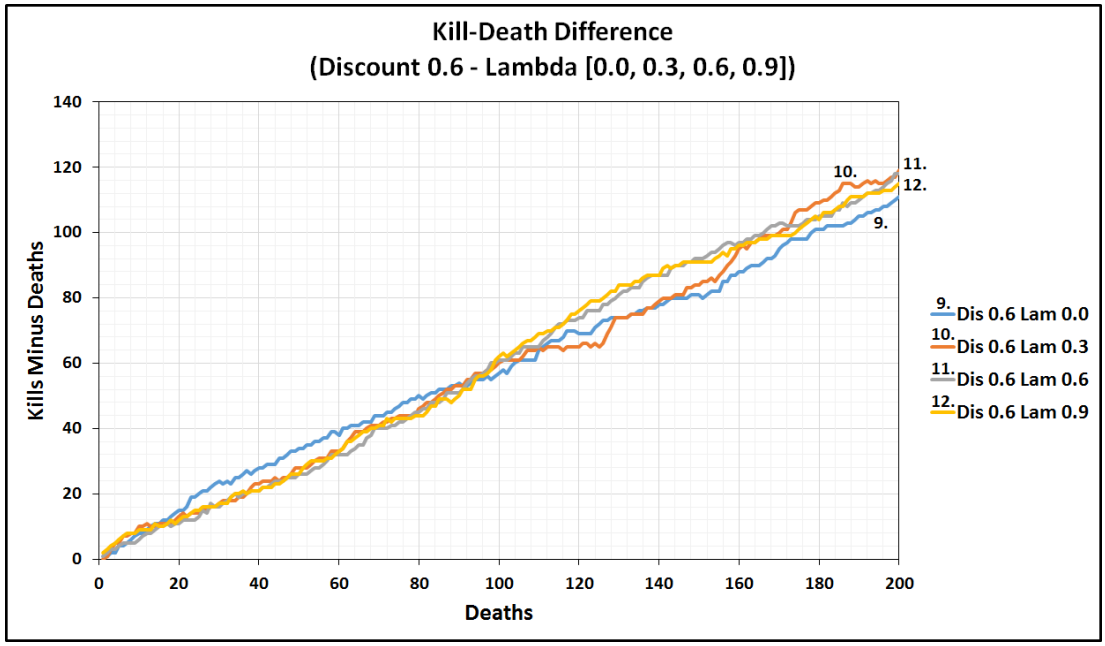

Figure 5.5: Kill-death differences for games 9 to 12 averaged over five runs.

Discount γ : 0.9				
Game Number (λ):	**13 (0.0)**	**14 (0.3)**	**15 (0.6)**	**16 (0.9)**
Average Time Alive:	15.8	15.8	15.9	16.1
Min Time Alive:	2.1	2.3	2.4	1.6
Max Time Alive:	72.95	79.9	77.8	75.7
Average Reward:	2.0	1.9	1.3	1.7
Min Reward:	-1.7	-1.8	-1.8	-1.9
Max Reward:	16.2	13.7	14.4	15.2
Final Kill Difference:	142	133	67	101

Table 5.14: Statistics for games 13 to 16 with γ set to 0.9.

5.5.2.4 Game 13 to Game 16 (Discount 0.9)

The results from the final set of games, Game 13 to Game 16, are shown in Table 5.14. The average time alive continues to be approximately 16 seconds as we have seen in all of the previous sets of games. The largest average reward that we have seen from all of the games is achieved by Game 13 in which lambda is set to 0.0 (disabling eligibility traces). This game also receives the second highest KDD overall. The maximum time alive for the bot in each of the four games falls between 70 and 80 seconds. The KDD for each of these games is plotted in Figure 5.6. Firstly, we can see that three of the games, Game 13, Game 14 and Game 16 all begin with a very similar performance before diverging after 90 deaths. Game 15 begins the game worse than the other three and this continues throughout the 200 deaths.

5.5.2.5 Overall

This section discusses the results from all 16 games and shows them plotted together. These include the average reward, the final KDD and the average time alive. A random baseline indicator which represents the level of game-play achieved when actions are selected completely at random is also included. These levels were observed by running five games of 200 deaths in which learning was

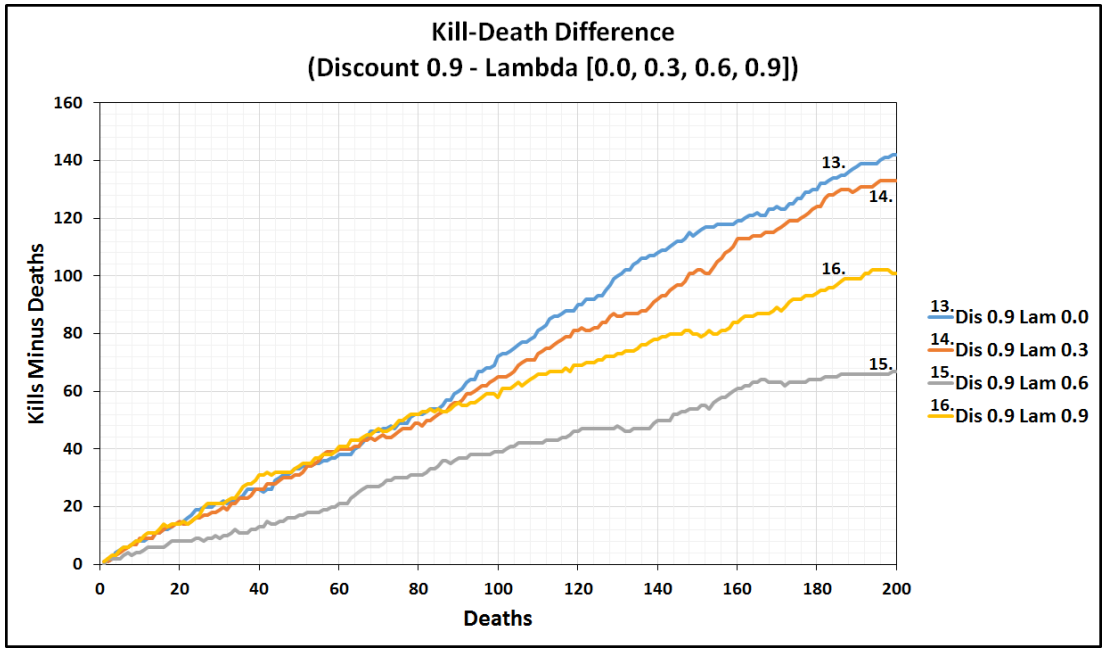

Figure 5.6: Kill-death differences for games 13 to 16 averaged over five runs.

disabled and a random number generator was used to select actions. The bot managed to have more kills than deaths in four out of the five games with random action-selection but the number of kills and reward received was much less than when learning was enabled in the majority of the cases. The decomposition of the tasks into modes is enabling the bot to perform quite well even when selecting actions at random. For instance, when the bot is in *Danger* mode, two out of the seven actions available to it involve shooting at the opponent which results in the bot being able to kill the opponent, or at least cause damage, which results in a substantial reward. The logic of the bot still uses the SARSA(λ) algorithm but it is using 100% exploration for the random runs. The first collection of results, that are shown below in Figure 5.7, are the average rewards received per life.

The average level of reward received per life when actions are selected at random is 0.75. The value is more than doubled in all but two of the bots that have learning enabled. Just two of the bots receive an average reward of more than 2. These are Game 3 and Game 14 and their parameter values are very

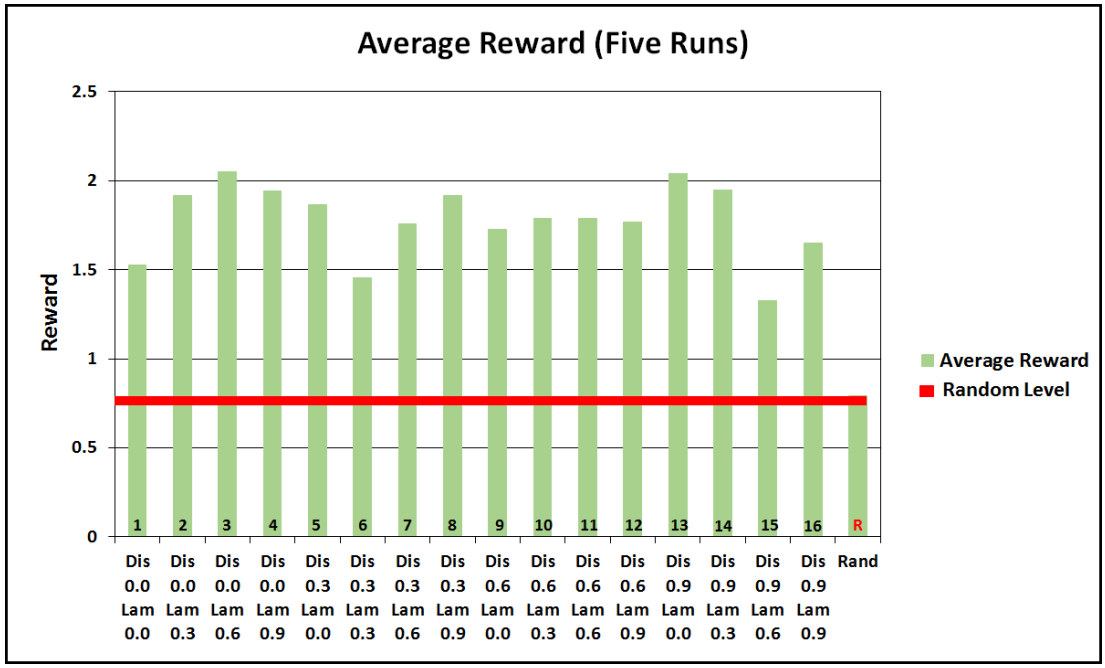

Figure 5.7: Average reward for each combination of γ and λ.

different. Game 3 has the discount parameter set to 0 and the lambda parameter set to 0.6 whereas Game 14 has the discount parameter set to 0.9 and the lambda parameter set to 0. The average reward is consistent for all four values of lambda when the discount parameter is set to 0.6. This trend cannot be seen for the other discount parameter values.

The average final KDD for all of the games are shown in Figure 5.8 below. The average random level is a KDD of 12.6. Just one of the five random games ended with a negative KDD. As mentioned earlier, the limited actions to choose from meant that the bot could still kill the opponent even though it was selecting actions at random. All of the bots, except for Game 15, finished with an average KDD of over 80 with the majority falling in the range of 100 to 140. The two bots with the highest average reward achieved a final KDD of just over 140.

The final illustration of all of the results plotted together is that of the average time alive which is shown in Figure 5.9. All games, including the random action-selection, fall within a range of approximately 15.5 to 16.5. As we have seen from

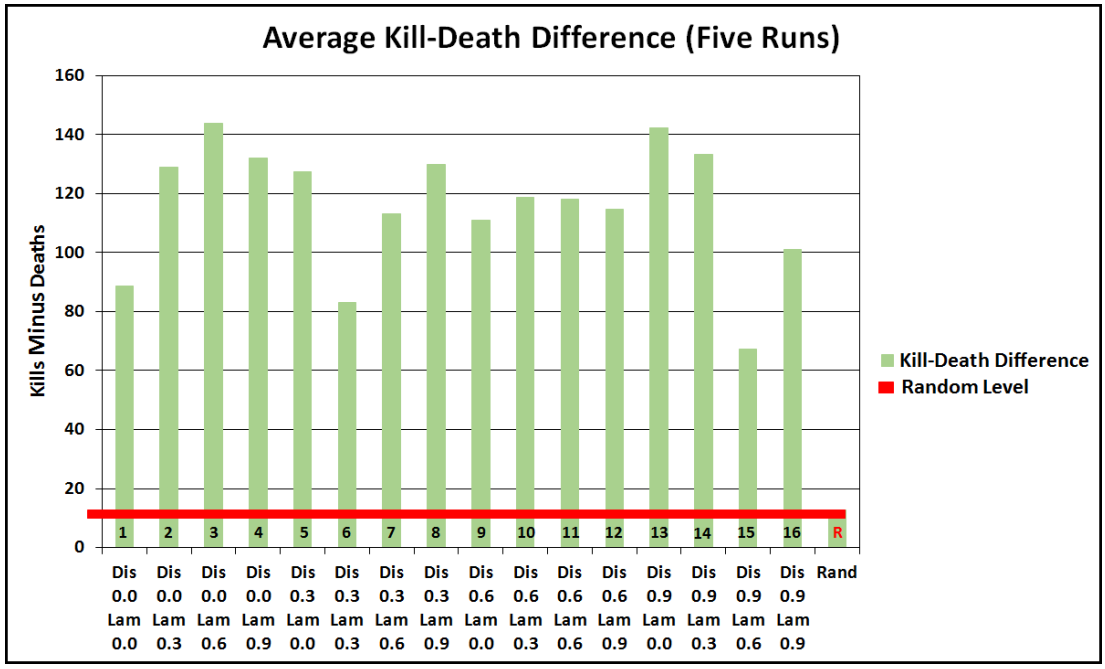

Figure 5.8: Average kill-death difference for each combination of γ and λ

.

the previous sections, the bots can stay alive for up to a minute and a half but on average they fall within a very similar range. This is expected as the small map Training Day was chosen so that the competing bots would almost always be in sight of each other. The fact that it takes 16 seconds on average for the bots to see an opponent and either kill them or die themselves is not surprising given the size of the map.

5.5.3 Discussion

The reason that the learning performance appears to be relatively insensitive to the algorithm parameters could be a result of the implicit randomness in the game and the fact that the DRE-Bot has a high-level, simplistic design. The map used is also very small and encourages almost constant combat between players. DRE-Bot is shown to be learning in a simple Deathmatch scenario and can achieve good performance with respect to the KDD achieved. The performance is much better

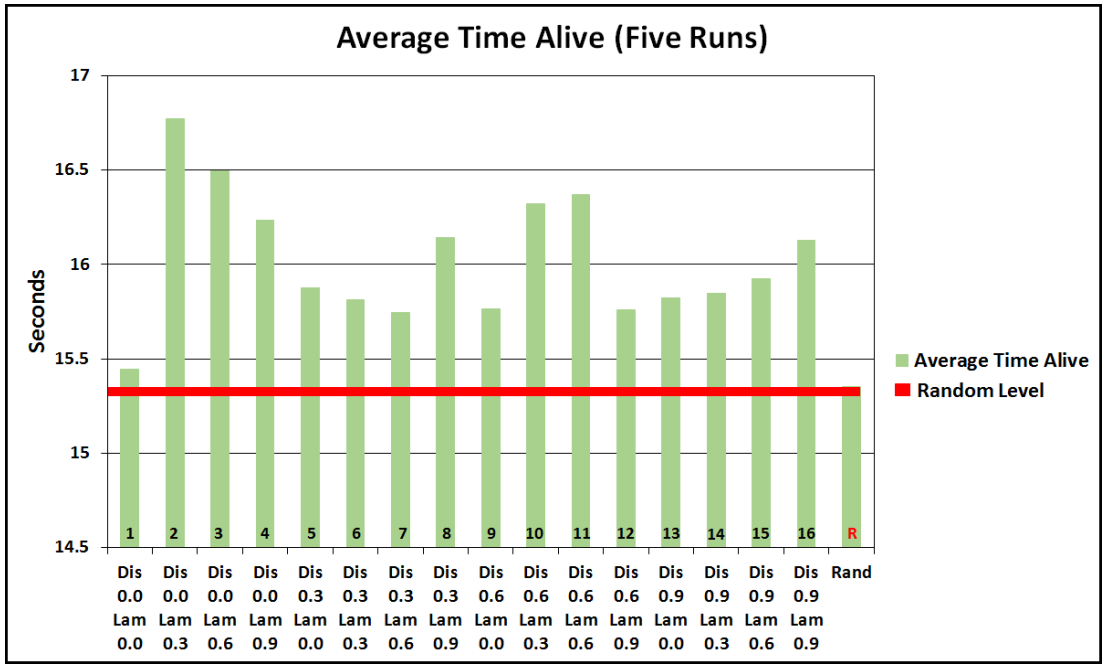

Figure 5.9: Average time alive for each combination of γ and λ.

than the baseline random action-selection performance although we have seen that selecting actions randomly still results in a small positive KDD. The results show that DRE-Bot architecture can produce varied performances but there does not appear to be any clear correlation present between the values of the γ and λ parameters and the performance levels achieved. This is evidenced by the large variation observed in performances over multiple runs.

5.5.4 Analysis of Findings

This section has described experimentation that was carried out using the DRE-Bot architecture. The experiments involved carrying out multiple runs of games on the Training Day map in which the discount and lambda parameters were varied. The purpose of this was to identify if the values of these parameters consistently had a direct impact on the performance of the bot and to observe the levels of performance that the bot could achieve against fixed-strategy opponents.

While the DRE-Bot did perform much better than the baseline random bots, which indicates that some useful strategies are being learnt, there was no clear evidence that the choice of value for the γ and λ parameters had a direct impact on the overall performance achieved.

5.6 Chapter Summary

This chapter has introduced and described our hierarchical DRE-Bot behavioural architecture for controlling NPCs in a first-person shooter game. The bot's decision-making is controlled by the SARSA(λ) algorithms and the architecture consists of multiple learners which are activated depending on the circumstances of the game. We described the overall design of the states, actions and rewards before going on to present the details of the experimentation that we carried out. Firstly, we compared a multi-factor reward design with a simple reward design which showed that the multi-factor reward approach led to marginal statistically significant performance improvements and then we proceeded to compare the performances of DRE-Bot and Sarsa-Bot using the same methodology. These experiments verified that the changes to the architecture had indeed led to significant performance improvements overall. Finally, we presented the results of 16 different games, over multiple runs, in order to analyse if the gamma and lambda parameter values has a noticeable effect on the overall performance. We have concluded that while the DRE-Bot can produce varied performances and comfortably beat the fixed-strategy opponents, reinforcement learning may be more suited to a task with a single objective and single source of reward. For this reason, we switched the focus of our research to learning the task of shooting which we will describe in the next chapter.

Chapter 6

RL-Shooter Architecture

This chapter describes our shooting architecture, called *RL-Shooter*[1], that enables an NPC to learn how to competently shoot and adapt its technique over time based on game experience. The focus of the bot's learning is based solely on shooting so other tasks such as turning to face opponents and navigation are hard-coded into the bot's logic. The purpose of this is to address our second research question, from Section 1.3, which is to investigate whether or not reinforcement learning is more suited to an individual task, with a sole objective, as opposed to the multi-objective behavioural architectures described in the last two chapters.

6.1 Overview

In current state-of-the-art commercial first-person shooter games, computer controlled bots can often be easily distinguishable from those controlled by humans. Tell-tale signs such as failed navigation,"sixth sense" knowledge of human players whereabouts and deterministic, scripted behaviours are some of the causes of this. We propose, however, that one of the biggest indicators of non-humanlike behaviour in these games can be found in the weapon shooting capability of the bot. Consistently perfect accuracy and "locking on" to opponents in their visual field from any distance are indicative capabilities of bots that are not found in human players. Traditionally, the bot is handicapped in some way with either a

[1]Some of the work in this chapter was first published in Glavin & Madden [2015a]

timed reaction delay or a random perturbation to its aim, which does not adapt or improve its technique over time. We hypothesise that enabling the bot to learn the skill of shooting through trial and error, in the same way a human player learns, will lead to greater variation in gameplay and produce less predictable non-player characters. FPS games require human players to have quick responses, good hand-eye coordination and the ability to memorise complicated game controls. The purpose of the RL-Shooter shooting architecture is to mimic a human player firing the weapon and adapting the shooting technique over time as experience is gained. The behaviour produced is based on a dynamic reward signal from the amount of damage caused to opponents.

6.1.1 Motivation

Yannakakis & Hallam [2005, 2004] showed how AI quality can affect a players enjoyment in games that involve interaction with computer-controlled opponents. They concluded that more enjoyment comes with non-trivial adaptive opponents, which means that they exhibit a variety of different behavioural styles and are able to respond to changing human strategies. Human players consistently adapt their playing strategy, increase their knowledge through experience and try to find new ways to out-wit their opponents. Artificially generating these skills and traits in a computer-controlled bot is a complicated task which needs a lot of consideration. While bots can easily be programmed to flawlessly carry out the tasks required to play in a FPS game, this is not the goal of developing effective AI opposition in these games. The overall aim is to design a bot in such a way that a human player would not be able to detect that the opponent is being controlled by a computer. In this sense, bots cannot be perfect and must be occasionally prone to bad decision making and "human errors" while at the same time learning from their mistakes.

6.1.2 Implementation Details

The RL-Shooter bot is designed with some hard-coded knowledge and is not required to learn everything about the game mechanics itself. For instance, we designed the bot to only shoot if an opposing player is visible. This prevents

the bot from needlessly shooting and wasting ammunition when there are no opponents in sight. It also has an embedded hard-coded strategy of focusing on the closest player if there are multiple targets in its field of vision. It will only attempt to shoot if it currently has ammunition, that is, it will not stand and point the gun at the opponent unless it can cause damage. The bot will continually navigate around the map, using the inbuilt NavPoint system, when the conditions for shooting are not met. These conditions are seeing an opposing player and having ammunition to engage in combat.

6.2 RL-Shooter Design

RL-Shooter bot is an architecture that enables an NPC to adjust its shooting technique based on rewards that it receives from the environment for successfully causing damage to an opponent. The success of the reinforcement learning algorithm relies on the design of suitable states, actions and rewards. This section provides a detailed description of how the states, actions and rewards were designed for the task of learning to shoot. The state and action space for the current architecture was designed specifically for the map Training Day, the layout of which is shown in Figure 6.1 below. This map was chosen due its small size and tendency to encourage almost constant combat between players. Since the reinforcement learning architecture in this chapter is only concerned with shooting, the smaller map prevents players from having to excessively travel throughout the map before encountering opponents.

6.2.1 On the use of the SARSA(λ) Algorithm in RL-Shooter

The architecture uses the SARSA(λ) algorithm, as described in Section 2.2.3, once again to control decision-making. The learning process, as embedded in the bot shooting logic, is illustrated in Fig. 6.2. If the bot has just started to shoot, it will read the current state (s) from the environment and select an action (a) to perform. A reward (r) is then received from the system based on damage caused and the next state (s') is read from the environment. The bot then chooses the

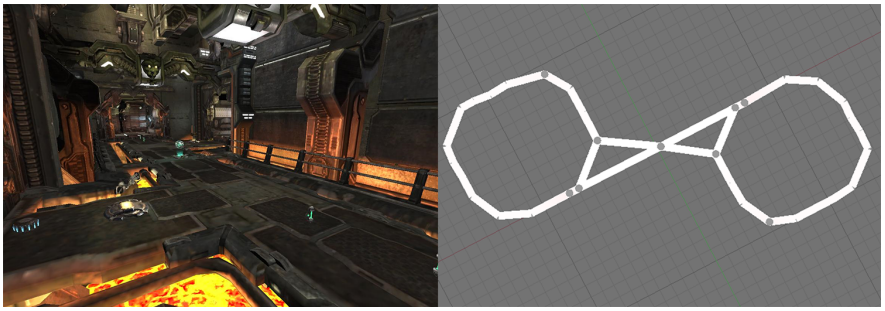

Figure 6.1: Training Day map and a bird's eye view of its layout.

next action (*a'*) to carry out and the policy is updated using this quintuple of information (*s a r s' a'*). If the bot continues shooting, the current state and action will be updated with the values from Step 4 and Step 5. If the bot stops shooting, due to killing an opponent or dying, then the shooting episode is ended. The next time it starts to shoot it will begin by reading the initial state directly from the system.

6.2.2 RL-Shooter States

The state space is inspired by how humans perceive enemy players during FPS combat. We have taken into account features for the enemy's *distance*, *movement*, *speed* and *rotation* relative to the bot.

State	Distance	Player Widths
Close	0 - 510 UU	$0 \leq d < 15$
Medium	510 - 1700 UU	$15 \leq d < 50$
Far	>1700 UU	≥ 50

Table 6.1: RL-Shooter discretised distance values

Each avatar in the game has an absolute location on the map represented by X, Y and Z coordinates in UU. The X and Y values are "flat" while the Z value represents the height of the character. We measure the distance of the bot to the

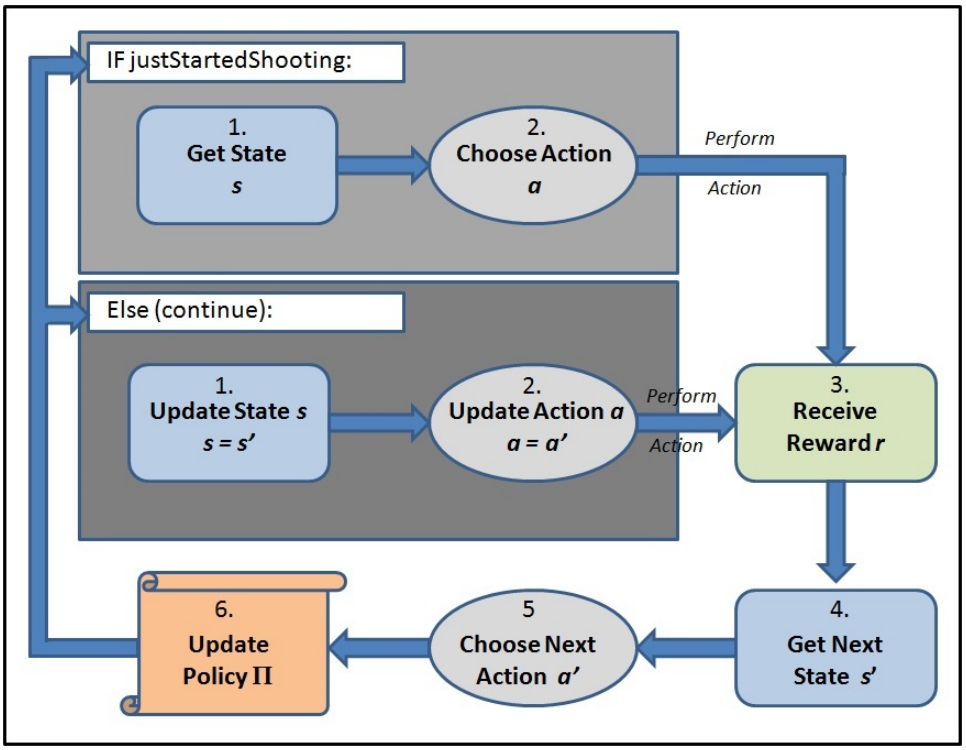

Figure 6.2: RL-Shooter shooting logic using SARSA(λ).

enemy and discretise these values into the ranges of "close", "medium" and "far" as detailed in Table 6.1. The enemy is said to be close to the bot if the bot's current location falls inside a perimeter of 510 UU surrounding the enemy. We chose this value as it is equivalent to 15 player widths. The enemy is a medium distance from the bot if its location falls between 15 and 50 player widths from the bot and anything over 50 is considered far. The relative speed of the enemy can be "regular" or "fast" as shown in Table 6.2.

State	Total Relative Velocity
Regular	0 - 800 UU/sec
Fast	>800 UU/sec

Table 6.2: RL-Shooter discretised speed values

The maximum relative velocity that is recorded from both the RL-Shooter bot and opponent running directly at each other is 880 UU. Players in the game usually move towards each other during combat with the majority of the readings falling between 700 UU and 880 UU. For this reason, a recorded value of above 800 UU is discretised as being fast. The impact from explosive weapons such as grenades can also cause players to temporarily have a higher velocity reading than usual. We only take the X and Y coordinates of the velocity vector into account when calculating relative velocity. The total velocity of the enemy relative to the bot is calculated and if this value falls below a certain threshold then the enemy is said to be moving at a regular speed. Anything above this threshold is treated as fast. The direction that the enemy is moving relative to facing the RL-Shooter bot is also taken into account. The values for this state representation are shown in Table 6.3. Three checks are carried out to determine how the enemy is moving. Firstly, the enemy can be moving towards or away from the bot or not moving in either of those directions. The enemy can also be moving either left or right or not moving in either of these directions. The enemy can be jumping or not when moving in any direction and is stationary when not moving in any direction. In our definition of "stationary", the enemy can still be jumping on the spot.

Check	**Values**
T/A Direction:	Towards / Away / None
L/R Direction:	Left / Right / None
Jumping	Yes / No

Table 6.3: RL-Shooter discretised movement direction values

There are 6 discrete values, shown in Figure 6.3, for representing the direction in which the opponent is facing. These values consist of Front Right One (FR1), Front Right Two (FR2), Back Right (BR), Back Left (BL), Front Left Two (FL2) and Front Left One (FL1). The enemy will not always move in the same direction as it is facing but knowing which direction it is facing could be useful for anticipating the enemy's sequence of movements.

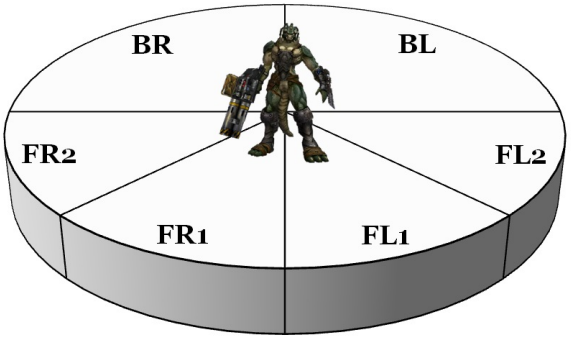

Figure 6.3: RL-Shooter discretised values for the enemy's rotation

The bot also takes into account whether the weapon they are using is an "instant hit" weapon or not. This means that there is no apparent delay from when the weapon is fired to hitting the target. Examples of such weapons are the sniper rifle and lightening gun which instantly hit their target once fired. Other guns shoot rockets, grenades and slow moving orbs which take time before hitting the target. The complete state space for shooting includes 1296 different states using the aforementioned checks. These are summarised in Table 6.4 below.

Attributes	**Number of values**
Distance	3
Velocity	2
Jump	2
Direction	9
Rotation	6
InstantHit	2
Total	$3 \times 2 \times 2 \times 9 \times 6 \times 2 =$ **1296**

Table 6.4: RL-Shooter shooting states

6.2.3 RL-Shooter Actions

The actions that are available to the bot involve variations on how it shoots at enemy targets. We have identified six different categories of weapons to account for the variance in their functionality.

The *Instant Hit* category is for weapons that immediately hit where the crosshairs are pointing once the trigger has been pulled. The primary mode[1] of the Sniper Rifle, Lightning Gun and Shock Rifle all belong to this category. The Sniper Rifle and Lightning Gun do not have a shooting secondary mode but activate a zoomed in scope view for increased precision. This scoped secondary mode, however, only applies to human players in the game.

The primary mode of the Assault Rifle and both modes of the MiniGun are examples of the *Machine Gun* category which spray a consistent flow of bullets. The game includes an inbuilt spread of bullets to mimic the recoil that would occur when shooting a machine gun.

The *Projectile* category is made up of guns which shoot explosive projectiles. These include grenades from the secondary mode of the Assault Rifle and Flak Canon as well as an exploding paste from the Bio Rifle in primary mode. The secondary mode of the Bio Rifle is used for charging the weapon to produce a larger amount of paste which damages players when they come into contact with it.

Slow Moving guns, which shoot the likes of rockets and "orbs" involve a delay from when they are shot to when they reach the target. Examples of these guns include the secondary mode of the Shock Rifle and the primary mode of the Rocket Launcher and the Link Gun.

Close Range weapons are those that should be used when in close proximity to an opposing player. The Flak Canon is an example of this type of weapon which shoots a spread of flak (primary mode) that is very effective at close range. The Shield Gun, used as a last resort weapon for defence, deals a small amount of damage from close range in primary mode and acts as a shield deflecting enemy fire in secondary mode.

The final category of weapons, *Other*, includes all other weapons in the game

[1] All weapons have a primary and secondary mode activated by left and right mouse clicks, respectively, as described in Section 3.3.3.1.

that have not been identified in one of the previous categories. The weapon categories are summarised in Table 6.5 below.

Instant Hit	**Machine Gun**
Sniper Rifle (Primary (No Sec.))	Assault Rifle (Primary)
Lightning Gun (Primary (No Sec.))	Mini Gun (Primary)
Shock Rifle (Primary)	Mini Gun (Secondary)
Projectile	**Slow Moving**
Assault Rifle (Secondary)	Shock Rifle (Secondary)
Flak Canon (Secondary)	Rocket Launcher (Primary)
Bio Rifle (Primary (No Sec.))	Link Gun (Primary)
Close Range	**Other**
Flak Canon (Primary)	
Shield Gun (Primary)	(All other guns picked up)
Shield Gun (Secondary)	

Table 6.5: The different types of gun available to the bot

Each category of gun has five actions associated with it in the current implementation. This results in 6480 state-action pairs for each category of gun or 38880 state-action pairs in total. The actions available for each gun are listed in Table 6.6. The shooting actions for the bot involve receiving the planar coordinates of the enemy's location and then making slight adjustments to these or shooting directly at that area. The *Head*, *Mid* and *Legs* take the X-axis and Y-axis values directly and the Z-axis value is set to head height, the midriff and the legs of the opponent accordingly. Shooting *left* and *right* involves skewing the shooting in that direction by incrementing / decrementing the X value of the target location by a fixed number of UU. In the current implementation, the amount of skew added comes from fixed values that are specific to the weapon type. A possible improvement for future implementations could involve dynamically determining this skew based on the relative velocity of the opponent and the nature of the weapon being used.

Gun	Instant	Machine	Projectile	Slow	Close	Other
Action 1	Head	Player	Player	Player	Head	Head
Action 2	Mid	Location	Location	Left	Mid	Mid
Action 3	Legs	Head	Above	Left-2	Legs	Legs
Action 4	Left	Left	Above-2	Right	Left	Left
Action 5	Right	Right	Above-3	Right-2	Right	Right

Table 6.6: Shooting action directions for specific gun types

The *Player* action uses the inbuilt targeting which takes an enemy player as an argument and continually shoots at that player, regardless of their movement. This is essentially "locking on" to the opponent but since actions are chosen multiple times a second by the reinforcement learner this should not be apparent to the human opposition. Experienced human players can often be very accurate on occasion, just not constantly flawless. The *Location* action shoots directly at the exact location of the opponent. There are three variants of shooting above the opponent (*Above*, *Above-2* and *Above-3*) which differ by the distance above the player with *Above-3* being the highest. These "Above" actions are designed so that the bot can find the correct height above the opponent so that the resulting trajectory will lead to causing damage. The further away the opponent is will require more height in order for the aim to be successful. *Left-2* and *Right-2*, which are found in the *Slow* category, provide a bigger adjustment in each direction to account for the slow moving ammunition.

Unlike previous work in the literature [Polceanu, 2013; Schrum *et al.*, 2011; Wang & Tan, 2015], our shooting mechanism is being refined over time as the bot learns with in-game experience. A percentage of random selection of actions is used to enable exploring in the policy and the rate of this is decreased over time at which point the bot begins to rely on the knowledge that it has built up. We will see the changes in the rate of exploration (wrt the ϵ-greedy action-selection policy) for the experimentation later in Table 6.7. The large initial exploration rate facilitates a quicker search for the best actions by trying a larger number

of actions. We also have an inbuilt mechanism to select actions that have not been selected before during exploration. Human players constantly adapt and learn what works best and then try to reproduce these actions as often as they can. Mistakes are, of course, made from time to time which are being accounted for here with random action-selection occurring a percentage of the time during exploration. At the early stages of learning, the bot will not know the best actions to take so each of them are as likely to be chosen.

Weapon selection for the bot is taken from hard-coded rules based on priority tables of close, medium and far combat. The best weapons to use at these distances are stored in the tables and these are based on human knowledge of the weapon capabilities. The use of these tables were inspired by a similar system in the UT^2 bot [Schrum *et al.*, 2012]. The bot will use the best available weapon that it has, according to these tables, based on the current distance from the opponent. Weapon selection in itself is a task that could be learned but our current research is focused on shooting technique so we have opted to use human knowledge for weapon selection.

6.2.4 RL-Shooter Rewards

The reward that the bot receives is dynamic and related directly to the amount of damage caused by the shooting action. The bot receives a small penalty of -1 if the action taken does not result in causing any damage to an opponent. This ensures that the bot is always striving towards the long term goal of causing the most damage that it can, given the circumstances.

6.2.5 Discussion

To the best of the author's knowledge, this architecture is the first to use the SARSA(λ) reinforcement learning to enable NPCs to specifically learn and adapt the skill of shooting over time based on in-game experience. This approach is novel in that the bot will constantly adjust its shooting technique as it gathers knowledge of what works well and what does not. This approach is inspired by how humans learn to play these games. With the RL-Shooter behavioural architecture, we have developed a state space and action space representation to

facilitate the bot's perception of opponents in the environment by reading key details. We have also tailored suitable actions to react in given circumstances. Reading important features of the opponents combat movements coupled with the "damage caused" reward signal are then used to drive real-time, knowledge-based decision making. Traditional approaches to NPC shooting in FPS games involve limiting the ability of the bot by incorporating either a delay before shooting or purposefully missing the target to simulate lower ability from the bot. The main drawback of this approach is the lack of adaptation. Once a human player forms a strategy to beat such an opponent, there is no longer a challenge and the gameplay becomes highly predictable. Our approach constantly adapts the shooting technique based on in-game experience. In order to advance the state-of-the-art, we believe that computer-controlled opponents should adapt to their surroundings and improve with experience over time. Enabling learning in individual tasks, such as shooting, certainly leads to less predictable gameplay. The bot initially *explores* the outcome of all of the actions that it has available and then, as it gains experience over time, it *exploits* the knowledge that it has built up.

As mentioned earlier in Section 6.2, RL-Shooter was tailored specifically for the small Training Day map in UT2004. The architecture could, however, be refined to work equally and consistently between all maps by introducing very few changes. For instance, we discard the Z value when reading the relative velocity in the map Training Day, given the flat nature of the map's geometry. This, of course, has the consequence of ignoring some relevant velocity information when the opponent is jumping and dodging so in order for the state space to be more complete and representative of all maps, with complex geometry of varying sizes, this value would have to be re-introduced. Also, the discretised values for distance are specific to the Training Day map. Game logs were used to determine the min, max and average distances of opponents during combat. These values were then used to create an approximate notion of "close", "medium" and "far" for the specific map. More generalised values would be required for the same distance categories to be applicable in all maps.

6.3 RL-Shooter Experimentation

6.3.1 Experiment Details

Three individual RL-Shooter bots were trained against native scripted opponents from the game with varying difficulty. These native bots ship with the game and each of them has a hard-coded scripted strategy that dictates how they behave. A discussion of these bots and a list of all the skill levels available can be found earlier in Section 3.3.3.4 and at the *Liandri Archives: Beyond Unreal* website[1]. In our experiments we use three skill levels, Novice, Experienced and Adept:

- **Opponent Level 1 (Novice)** - 60% movement speed, static during combat, poor accuracy (can be 30 degrees off target), 30 degrees field of view.
- **Opponent Level 3 (Experienced)** - 80% movement speed, can strafe during combat, better accuracy and has faster turning, 40 degrees field of view.
- **Opponent Level 5 (Adept)** - Full speed, dodges incoming fire, tries to get closer during combat, 80 degrees field of view with even faster turning.

Each experiment run involved the RL-Shooter bot competing against three opponents that have the same skill level as each other, on the Training Day map in a series of thirty minute games. As discussed earlier, Training Day is a small map which encourages almost constant combat between opponents. We chose this map since the focus of our experimentation was on the shooting capabilities of the RL-Shooter bot. A total of three experiment runs took place, one for each of the opponent skill levels mentioned above. There was no score limit on the games and they only finished once the thirty minute time limit had elapsed. Each time the RL-Shooter bot was killed the state-action table was written out to a file. These files represent a "snapshot" of the bot's decision-making strategy for shooting at that moment in time. Each bot starts out with no knowledge (Q-value table full of 0's) and then, as the bot gains more experience, the tables become more populated and include decisions for a wider variety of situations.

[1] http://liandri.beyondunreal.com/Bot

The amount of exploration being carried out in the policy of the learners was dependent on the values from Table 6.7.

Lives	Exploration Rate
0 - 9,999	50%
10,000 - 19,999	40%
20,000 - 29,999	30%
30,000 - 39,999	20%
40,000 - 49,999	10%
50,000 or greater	5%

Table 6.7: Exploration Rate of the RL-Shooter Bot

For the first 10,000 lives the bot is randomly selecting an action half of the time. The other half of the time it is using knowledge that it has built up from experience (choosing the action with the greatest Q-value based on previous rewards received). During exploration, we included a mechanism for choosing randomly from actions which have not been selected in the past to maximise the total number of state-action value estimates that are produced. The exploration rate is reduced by ten percent every ten thousand lives until is remains static at five percent once the bot has been killed over fifty thousand times.

6.3.2 Results: 60,000 Lives

This section and the next one present the experimentation results from two different perspectives. In this section, we look at the different trends that occur with the bot having lived and died 60,000 times. This is followed, in Section 6.3.3, by analysing *the same results* from the perspective of the 30 minute games that were played, as opposed to the individual lives. The Level 5 skilled opponent had played 350 games as its death count approached 60,000. For this reason, our comparative game analysis of the three skill levels is carried out over 350 games.

In this section we look at the results and statistics gathered from each of the

RL-Shooter bots playing against opponents with different skill levels (Level 1, Level 3 and Level 5 opponents). From here on, we will refer to the RL-Shooter bot playing against Level 1 opponents as RL-Shooter-1 and the other two, playing against Level 3 and Level 5 opponents, as RL-Shooter-3 and RL-Shooter-5 respectively. We analyse the results of the bots having lived through 60,000 lives with a decreasing exploration rate as described in Table 6.7.

Firstly, in Table 6.8 below, we can see the total kills, deaths and suicides accumulated over the 60,000 lives for each bot. This table also shows the kill-death (KD) ratio which computes how many kills were achieved for each death (either by the other player or by suicide). RL-Shooter-1 has a KD ratio of 1.87:1 with almost 20% of its deaths coming from suicides. Suicides occur in the game when the bot uses an explosive weapon too close to an opponent or wall and can also occur if a bot falls into a lava pit. Although the Training Day map is small, there are three separate areas where bots can fall to their deaths.

Opponent Skill Level	**Total Kills**	**Total Deaths By Others**	**Total Deaths By Suicide**	**KD Ratio**
Level 1	112420	48299	11701	1.87:1
Level 3	63934	52994	7006	1.07:1
Level 5	40466	54136	5864	0.67:1

Table 6.8: Total kills, deaths, suicides and kill-death ratio

The RL-Shooter-3 bot appears to be more evenly matched with its opponents and has a KD ratio of 1.07:1. Deaths by suicide correspond to 12% of the bot's overall deaths. The number of suicides appears to be directly linked to the number of kills which suggests that the majority of suicides are inflicted by the bot's own weapon as opposed to falling into a pit as mentioned earlier. This is confirmed further by the reduced suicide rate (10%) and kill totals for the RL-Shooter-5 bot. The RL-Shooter-5 has a negative KD ratio with 0.67 kills to every death.

Table 6.9 shows the average and standard deviation of hits, misses and reward received per life for the 60,000 lives. A *hit* is recorded each time the bot shoots its weapon and causes damage to an opponent. A *miss* is recorded when the

weapon is fired but fails to cause any damage. The *reward* corresponds to the exact amount of damage inflicted on an opponent for the current hit or -1 if no damage resulted from firing the weapon.

Opponent Skill Level	**Hits Avg (Std Dev)**	**Misses Avg (Std Dev)**	**Reward Avg (Std Dev)**
Level 1	9.82 (±7.42)	26.84 (±18.46)	79.82 (±83.00)
Level 3	7.30 (±5.69)	21.41 (±11.68)	47.71 (±50.38)
Level 5	4.70 (±3.79)	17.83 (±8.73)	25.06 (±33.37)

Table 6.9: Average and standard deviation values after 60,000 lives

RL-Shooter-1 fires the most shots per life on average with 36.66 (27% hits; 73% misses). This would be expected as weaker opposition would afford the bot more time to be shooting, both accurately and inaccurately. The shots per life and accuracy decrease as the skill level of the opposition increases with RL-Shooter-3 shooting an average of 28.71 shots (25% hits; 75% misses) and RL-Shooter-5 shooting an average of 22.53 shots (21% hits; 79% misses). These results are illustrated in Table 6.10.

Opponent Skill Level	**Hit Percentage**	**Miss Percentage**
Level 1	27%	73%
Level 3	25%	75%
Level 5	21%	79%

Table 6.10: Percentages of hits and misses over the 60,000 lives

While the level of shooting inaccuracy may seem quite high for all of the bots, they are all still performing at a competitive standard as evidenced earlier by Table 6.8. It is important to remember that hits are only recorded when the bot is shooting and the system indicates that it is currently causing damage. All

other shots are classified as misses. The actual damage caused by individual hits also varies greatly depending on the gun type used and the opponent's proximity to explosive ammunition from certain guns.

Table 6.11, below, lists the minimum, maximum and median values of the hits, misses and rewards over the 60,000 lives. The minimum numbers of hits and misses for each of the levels was zero. This is a result of the bot spawning into the map and being killed before it has a chance to fire its weapon.

Opponent Skill Level	**Hits Min|Max|Med**	**Misses Min|Max|Med**	**Reward Min|Max|Med**
Level 1	0 | 72 | 8	0 | 232 | 22	-50 | 1261 | 56
Level 3	0 | 46 | 6	0 | 115 | 19	-50 | 538 | 35
Level 5	0 | 39 | 4	0 | 99 | 17	-49 | 322 | 16

Table 6.11: Minimum, maximum and median values after 60,000 lives

The maximum numbers of hits, misses and rewards are again closely dependent on the opposition skill level and the large difference between these and the median values shows the amount of variance from life to life.

The reward, as mentioned earlier, corresponds directly to the amount of damage that the bots successfully inflict on their opponents. The value is set to 0 at the beginning of each life and then accumulates based on any damage caused or is decremented by 1 for shots that do not result in any damage. The results for each of the skill levels do not show a clear upward trend for total reward per life received over time. There could be many reasons for this. The ammunition from the different guns that can be picked up from the map cause varying degrees of damage upon successfully hitting an opposing player. While the RL-Shooter bots are learning different strategies for each of the different types of gun, they have no control over which weapon they have available to them during each life. They are prioritising the use of the more powerful weapons when they are available but during many lives, as evidenced by the shooting time average data from Table 6.12, they have not acquired these more powerful weapons. The small number of actions available for each gun type could also be a reason behind performing well

in the earlier games even when selecting randomly. On some occasions, the bot received a substantial total reward during its lifetime but it is inconclusive as to whether this was occurring randomly, given the nature of the game (real-time, multiple opponents, small map etc.), or whether the bots were improving their action selection as they experienced new situations and then took advantage of this knowledge when these situations occurred at a later stage.

6.3.3 Results: Thirty Minute Games

This section analyses the results and statistics based on individual games as opposed to the lives of the bots which we looked at in the previous section. Specifically, we look at 350 games, each with a duration of 30 minutes, for the three different opponent skill levels. All of the following results and statistics are reviewed on a "per game" basis.

The first table below, Table 6.12, lists some game statistics averaged over the 350 games. RL-Shooter-1 collected nearly twice as many weapons on average as the other two bots. All players in the game start each life with an Assault Rifle and must pick up additional weapons and ammunition from different points around the map. The Assault Rifle is a weak weapon and is only used when a better weapon is not available. Playing against lesser opposition gives the bot many more opportunities to pick up different weapons and also replenish their ammunition with pick-ups.

	Level 1	**Level 3**	**Level 5**
Weapons Collected	171.12	105.85	98.43
Ammunition Collected	72.54	57.74	57.36
Time Moving (mins)	20.81	20.66	20.30
Distance Travelled (UU)	489464.34	472513.43	459385.39

Table 6.12: Averages per game after 350 games (30 minute time limit)

All three bots spent the same amount of time moving which was just over 20

minutes. This would be expected as they were all using the same navigation modules which did not include any learning. Time spent not moving would include the short delays between when the bot is killed and when it spawns back to life on the map. The average distance travelled for each bot over the 350 games is also shown and this is measured in UU[1]. RL-Shooter-1 travels 16951 UU more than RL-Shooter-3 per match, on average, while RL-Shooter-3 travels 13128 UU more than RL-Shooter-5. This would suggest that as the skill level of the opponents increase the bots have less opportunity to traverse the map and thus miss out on important pick-ups.

	Level 1	**Level 3**	**Level 5**
Total Time Shooting (mins)	16.67	17.10	15.68
Shooting Assault Rifle (mins)	11.07	14.25	13.88
Shooting Shock Rifle (mins)	2.24	1.413	0.85
Shooting Flak Canon (mins)	1.92	0.74	0.54
Shooting Mini Gun (mins)	1.39	0.67	0.39
Shooting Shield Gun (mins)	0.03	0.01	0.002

Table 6.13: Shooting time averages after 350 games (30 minute time limit)

Table 6.13 shows the average amount of time shooting (in minutes) per game and also lists the shooting time for each of the individual guns. From the table, we can see that RL-Shooter-3 spends the most time shooting on average and also spends the most time using the Assault Rifle. RL-Shooter-1 does not use this default gun as much as the other bots because it is able to pick up stronger weapons from the map. The Shield Gun, which the bots also spawn with, is seldom used in any case as this is a "last resort" weapon which helps the bot to defend itself while searching for a more effective weapon. The small map with multiple opponents meant that the bots rarely got into a situation where the Shield Gun was the only remaining option.

[1]Unreal Units are described in Section 3.3.3.2

Opponent Skill Level	Killed Avg (Std Dev)	Killed By Avg (Std Dev)	Suicides Avg (Std Dev)
Level 1	251.08 (±4.71)	100.94 (±14.36)	24.75 (±8.46)
Level 3	178.02 (±11.51)	140.33 (±7.73)	18.78 (±4.61)
Level 5	135.06 (±10.61)	168.73 (±9.38)	18.54 (±4.34)

Table 6.14: Average and standard deviation values after 350 games

Table 6.14 shows the average kills, deaths by others (Killed By) and suicides from the 350 games. One fifth of the deaths to the RL-Shooter-1 bot were self-inflicted. Aside from this fact, the bot managed to keep an impressive 2:1 kill-death ratio. RL-Shooter-3 bot was more closely matched with its opponents (1.12:1 KD) where as RL-Shooter-5 bot had a negative kill-death ratio of 0.72:1. RL-Shooter-3 and RL-Shooter-5 had very similar suicide rates to each other.

Opponent Skill Level	Killed Min|Max|Diff	Killed By Min|Max|Diff	Suicides Min|Max|Diff
Level 1	177 | 287 | 110	78 | 121 | 43	12 | 43 | 31
Level 3	139 | 213 | 74	118 | 167 | 49	8 | 36 | 28
Level 5	89 | 158 | 69	125 | 190 | 65	3 | 33 | 30

Table 6.15: Minimum, maximum and difference values after 350 games

The minimum, maximum and difference (between min and max) of Kills, Killed By and Suicides after the 350 games are shown in Table 6.15. This table gives an idea of the range of variance between games when playing against each of the skill levels.

Another indicator of performance in FPS games is known as a Kill Streak. This is a record of the total number of kills that a bot can make without dying. The maximum Kill Streak was recorded for each of the games and is shown in Fig. 6.4.

The highest Kill Streak per game for RL-Shooter-1 usually falls between 7 and 10 for each game. This appears to change, however, as more shooting experience

is acquired and falls between 11 and 16 on many occasions reaching even as high as 20. RL-Shooter-3 usually achieves maximum Kill Streaks of either 5 or 6 but again these increase over time with the highest that it reaches being 11.

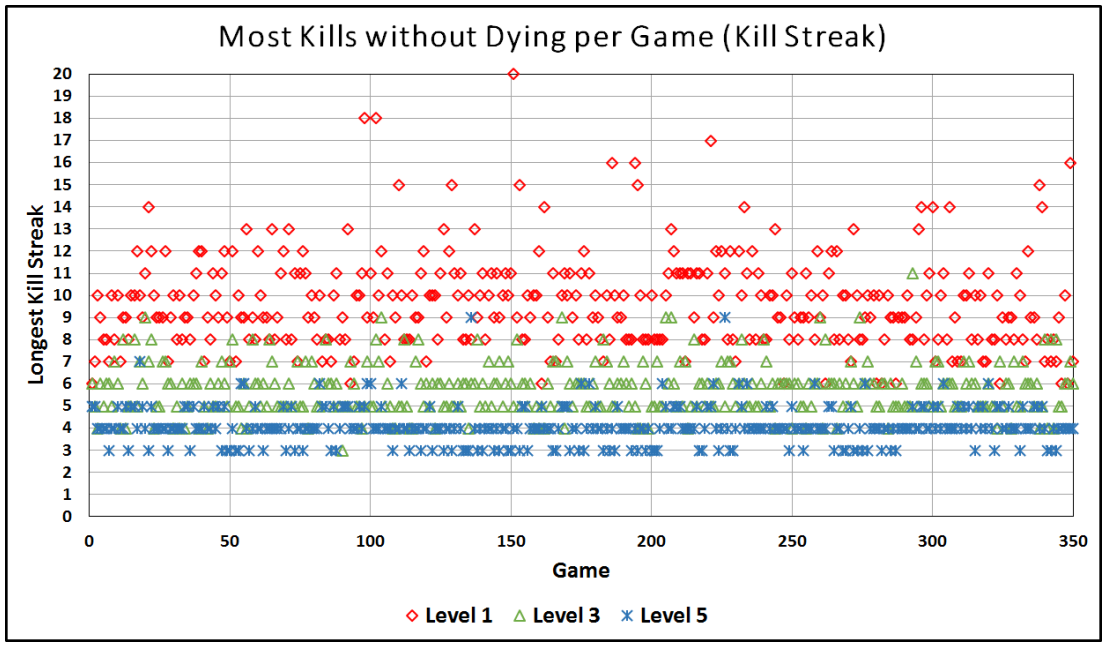

Figure 6.4: Longest kill streak per game for each of the opponent skill levels.

RL-Shooter-5 is less successful at achieving high Kill Streaks with the majority of them being either 3 or 4. It does, however, manage to achieve a Kill Streak of 9 on two occasions.

Fig. 6.5 shows the total number of kills that the RL-Shooter bots achieved in each of the 350 games. A clear separation of the results can be seen from the graph. RL-Shooter-1 manages to kill opponents in the range of 200 to 300 times each game. This range drops to between 150 and 200 for RL-Shooter-3 and again drops to falling mostly between 100 and 150 for RL-Shooter-5. Improvements in performance over time, while not significant, are more evident against the Level 3 and Level 5 opponents. This would suggest that the RL-Shooter-1 bot learns the best strategy to use against the weaker opponent at an early stage and then only ever matches this, at best, in the subsequent games.

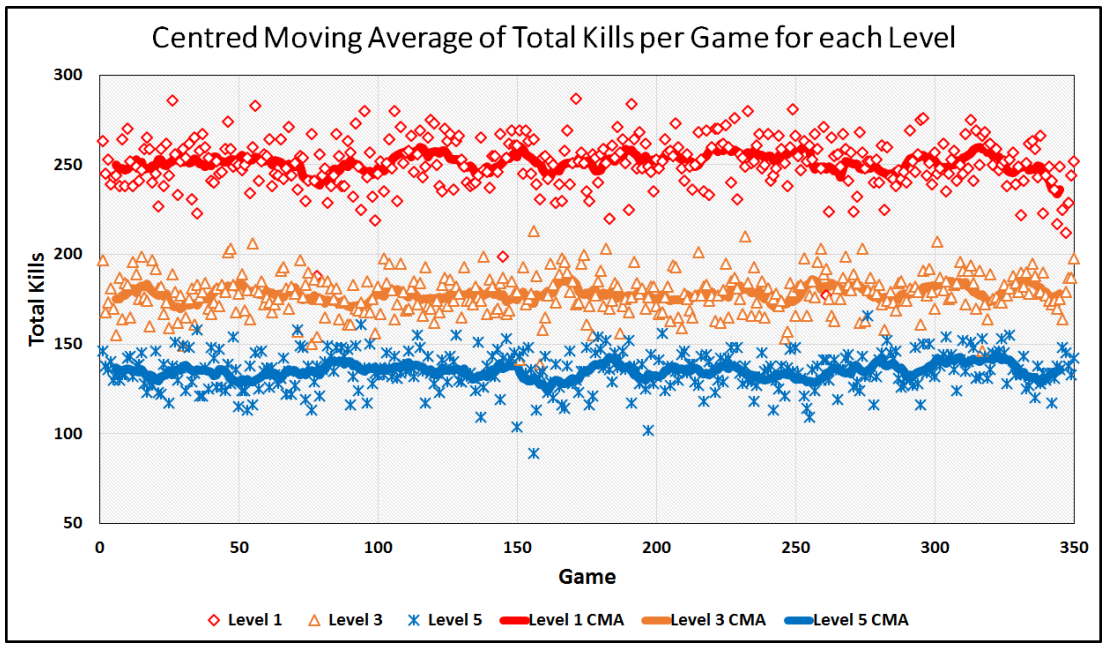

Figure 6.5: Total number of kills per game and Centred Moving Average of kills for each of the opponent skill levels.

The total number of deaths from the same 350 games are shown in Fig. 6.6. There is once again a clear separation of the data based on the skill level. While the number of deaths of the RL-Shooter-5 bot mostly fall between 160 and 180, there are a number of occasions midway through the games in which they fall within the range expected of a Level 3 bot (120 to 160). The number of deaths for the RL-Shooter-1 bot are quite evenly dispersed between 80 and 120 throughout all of the games. In order to investigate the presence of any trends in the data, Fig. 6.5 and Fig. 6.6 also show the Centred Moving Average (CMA) of the total kills and deaths, respectively. We use an 11-point sliding window for the CMA, so each point on the graph represents the average of the 11 samples on which it is centred. RL-Shooter-1 is the most consistent when it comes to kills in the early games. It appears to be gradually increasing the number of kills until a dip in performance around Game 80. It then slowly begins to recover before the total kills begin to fluctuate up and down. It is just beginning to recover from another

dip in performance in the final games. The other two bots, RL-Shooter-3 and RL-Shooter-5 show a similar fluctuating pattern with total kills. There appears to be little evidence to suggest that the total kills are improving consistently over time.

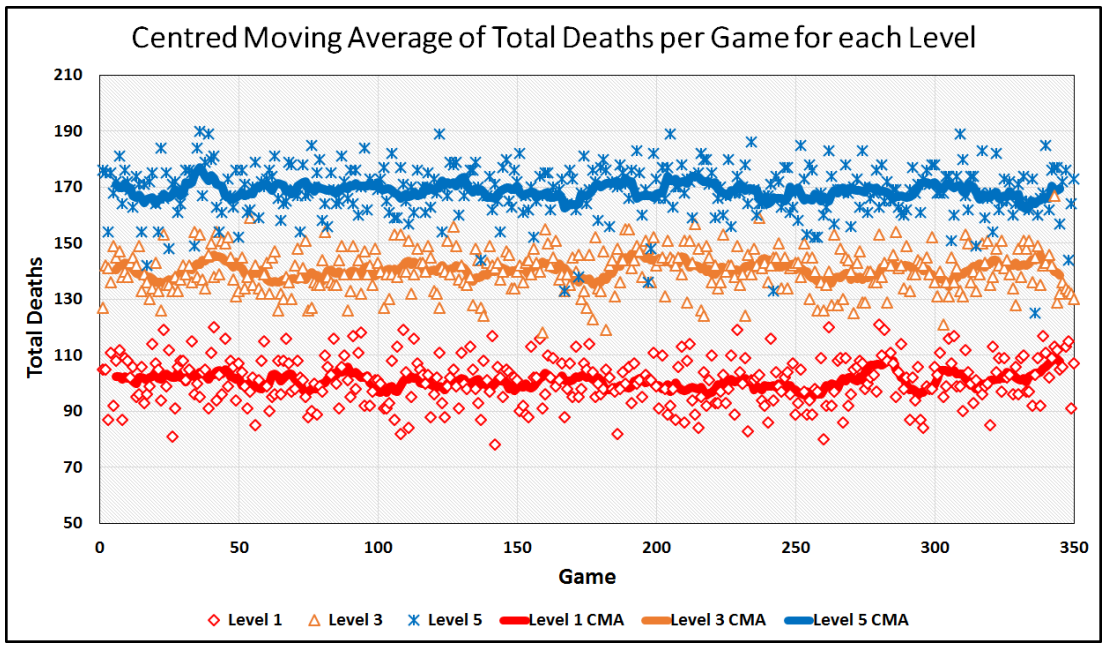

Figure 6.6: Total number of deaths per game and Centred Moving Average of deaths for each of the opponent skill levels.

This can also be said of the total deaths which show a similar amount of variance. We attribute this to the fact that the bots are choosing from a small subset of actions at each time step. The bot can be successful when randomly choosing from these actions. Although the best actions will not become apparent until the bots have built up experience, they may choose successful actions at an early stage given their limited choices.

6.3.4 Discussion

The experiments reported in this chapter involved deploying the RL-Shooter bot against native fixed-strategy opponent bots with different skill levels. The reason for pitching the RL-Shooter bot against scripted opponents was to ensure that all of the games were played against opponents of a set skill level to facilitate a direct comparative analysis and make it easier to detect any possible trends in performance. This would be much more difficult to achieve with human opponents given the inherent variance in human gameplay and the amount of time that would be needed to run all of the games (with the same human players). The results presented here are a comprehensive baseline against which any future improvements can be measured.

6.4 Analysis of Findings

Reviewing the overall results that are presented in the preceding sections, the main trends that can be observed are:

- The RL-Shooter bots are able to perform at about the same level as the "Experienced" opponent. For example, its kill-death ratio against Level 3 opponents is approximately 1:1.
- When pitched against weaker opponents, the RL-Shooter bots perform better and when pitched against stronger opponents they perform worse; this can be seen in all of the results presented.
- From Figures 6.6 and 6.5, there is not a clear pattern of the RL-Shooter bots continually improving in performance over time.

These results indicate how challenging it is for a bot with its relatively limited perception abilities and narrow range of actions to improve its performance over time. In our continuing work on this research topic, we believe it will be important to focus on developing new mechanisms by which we can improve the ability of the bots to demonstrate learning. We will also review and refine our state representations, action representations, and reward design. Simplifying the

experiment setup and limiting the learning to a single gun could also make it easier to identify learning trends over time. We take this into account during the development of our techniques in Chapter 7.

6.5 Chapter Summary

This chapter has described our architecture for enabling NPCs in an FPS game to adapt their shooting technique using the SARSA(λ) reinforcement learning algorithm. The amount of damage caused to the opponent is read from the system and this dynamic value is used as the reward for shooting. Six categories of weapon were identified and, in the current implementation, the bot has a choice of five hand-crafted actions for each. The bot reads the current situation that it finds itself in from the system and then makes an informed decision, based on past experience, as to what the best shooting action is. The bot will continually adapt its decision-making with the long term objective of inflicting the most damage possible to opponents in the game. This chapter has also described experimentation carried out using the RL-Shooter architecture and this was followed by a discussion and an analysis of the findings.

Chapter 7

Reinforcement Learning Mechanisms

In the first half of this chapter, we present a new method for reinforcement learning updates[1] and reward calculations for our RL-Shooter architecture. The updates are carried out periodically, after each shooting encounter has ended, and a new weighted-reward mechanism is used which increases the reward applied to actions that lead to damaging the opponent in successive hits in what we term "hit clusters". This work addresses research question R.3 from Section 1.3. In the second half of this chapter, we propose a skill-balancing mechanism which is based on a by-product of the learning process. The agent systematically stores "snapshots" of what it has learned during the different stages of the learning process. These can then be loaded during the game in an attempt to closely match the abilities of the current opponent. This work addresses research question R.4 from Section 1.3.

7.1 Periodic Cluster-Weighted Rewarding

In this section, we extend the development of our RL-Shooter behavioural architecture, as described in the previous chapter, by introducing the techniques of *Periodic Cluster-Weighted Rewarding* (PCWR) and *Persistent Action Selection*

[1] Some of this work was first published in Glavin & Madden [2015b]

(PAS). We describe the motivation behind, and the core functionality of, the techniques before presenting and discussing the experimentation that we carried out. The results from this section are then used as the basis for the work that is presented in Section 7.2.

7.1.1 Motivation

The purpose of this work is to improve upon the learned shooting performance of the RL-Shooter bot that was presented in Chapter 6. Our goal is to show a clear upward trend in accuracy and performance over time as the learner gains experience. As discussed throughout this thesis, we believe that FPS games provide an ideal testbed for carrying out experimentation using reinforcement learning. The actions that the players carry out in the environment have an immediate and direct impact on the success of their gameplay. Decision-making is constant and instantaneous with players needing to adapt their behaviour over time in an effort to improve their performances and outwit their opponents.

For the research presented here, we are concentrated solely on learning the actions of shooting a single weapon. We have limited the experimentation to a single gun in order to closely analyse the trend in performance over time.

7.1.2 States, Actions and Rewards

In this section, we describe the refinements that we applied to our RL-Shooter shooting architecture to learn how to shoot a single weapon in the game and produce a clear upward trend in shooting accuracy over time. The weapon chosen was the *Assault Rifle*. This weapon, with which each player is equipped when they spawn, is a machine gun that is most effective on enemies that are not wearing armour, and it provides low to moderate damage. The secondary mode of the weapon is a grenade launcher. We use a "*mutator*" to ensure that the only weapon available to the players on the map is the Assault Rifle; this is a script that changes all gun and ammunition pickups to that of the Assault Rifle when it is applied to the game. The architecture is only concerned with the primary firing mode of the gun in which a consistent spray of bullets is fired at the target. Actions could, of course, be tailored for both modes of each weapon in the game.

The game produces a spread of bullets, when the gun is fired, to imitate the natural variance of the bullets when firing a machine gun. That is to say the gun will not always hit *exactly* where it was aimed. We hypothesised at the outset that the bot will still be able to learn in the presence of this variance and will adjust its technique as a human player would.

For the refined state space, we increased the number of discretised relative velocity values, the number of relative rotation values and the number of discretised distance values for the bot's perception of the opponent. The refined action space includes eleven different shooting directions across the opponent at four different heights. The reward signal no longer uses the damage values from the system, as described in Section 6.2, and now uses set values. These changes are described in detail in the following sections.

7.1.2.1 States

The states are made up of a series of checks based on the relative position and speed of the nearest visible opponent. Specifically, the relative speed, relative moving direction, relative rotation and distance to the opponent are measured. The velocity and direction values of the opponent are read from the system before being translated into the learner bot's point of view.

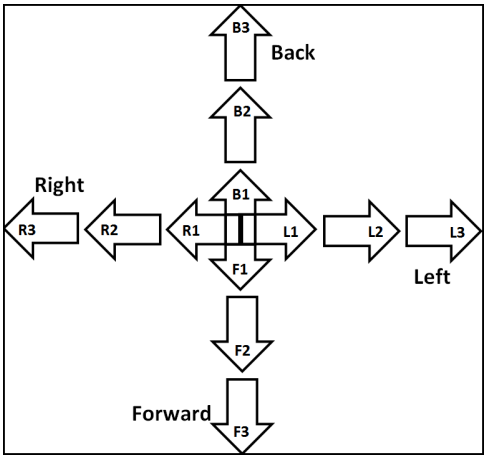

Figure 7.1: The relative velocity values for the state space.

The opponent's direction and speed are recorded relative to the bot's own speed and from the perspective of the learner bot looking directly ahead. The opponent can move forwards (F), backwards (B), Right (R) or Left (L) relative to the learner bot, at three different discretised speeds for each direction as shown in Figure 7.1. UT2004 has its own in-game unit of measurement called an Unreal Unit (UU), as described earlier in Section 3.3.3.2, that is used in measuring distance, rotation and velocity. In our velocity state representation, *Level 1* is from 0 to 150 UU/sec, *Level 2* is from 150 to 300 UU/sec and *Level 3* is greater than 300 UU/sec. The bot can be moving in a combination of forward or backward and left or right at any given time. For instance, one state could be R3/F1 in which the opponent is moving quickly to its right while slowly moving forward. There are six forward/backward moving states and six left/right moving states. The bot being stationary is another state so there are thirty seven possible values for the relative velocity states.

The direction that the opponent is facing (rotation) relative to the bot looking straight ahead is also recorded for the state space. This is made up of eight discretised values. There are two back facing rotation values which are Back-Left (BL) and Back-Right (BR). There are six forward facing rotation values, three to the left and three to the right, with each consisting of thirty degree segements as shown in Figure 7.2.

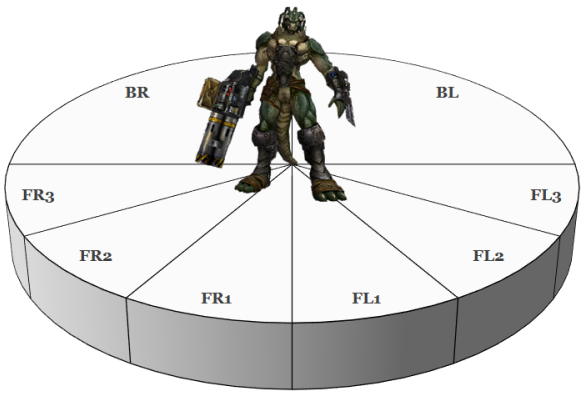

Figure 7.2: The relative rotation values for the state space.

The distance of an opponent to the bot is also measured and is discretised into the following values: “close”, “regular”, “medium” and “far” as shown in Table 7.1 below. These values are map-specific and were determined after observing log data and noting the most common values for opponent distance that were recorded.

The state space was designed to provide the bot with an abstract view of the opponents movements so that an informed decision can be made when choosing what direction to shoot.

State	Distance
Close	0 - 500 UT
Regular	500 - 1000 UT
Medium	1000 - 1500 UT
Far	>1500 UT

Table 7.1: The discretised distance values to an opponent for the state space.

7.1.2.2 Actions

The actions that are available to the bot are expressed as different target directions in which the bot can shoot, and which are skewed from the opponent’s absolute location on the map. The character model in the game stands approximately 50 UU wide and just under 100 UU in height and can move at a maximum speed of 440 UU/sec while running. (The system can, however, record higher velocity values on occasion when the bot receives the direct impact of an explosive weapon such as a grenade.) The amount of skew along the X-axis (left and right) and Z-axis (up and down) varies by different fixed amounts as shown in Figure 7.3. The Z-axis skews have four different values which range from the centre of the opponent to just above its head. The X-axis skews span from left to right across the opponent with the largest skews being 200 UU to the left and 200 UU to the right. These actions were designed specifically for the Assault Rifle weapon.

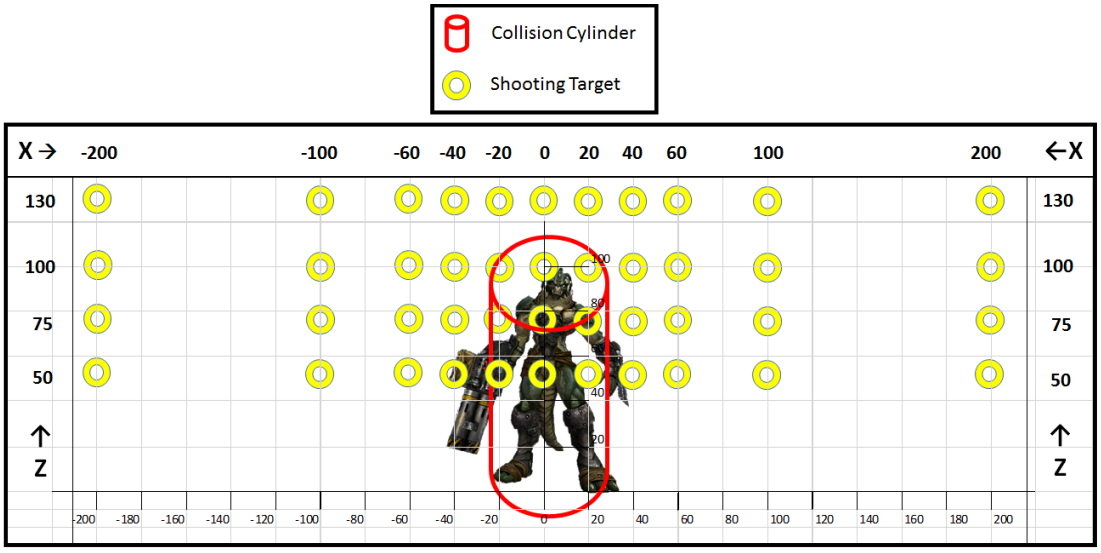

Figure 7.3: A visualisation of the shooting actions available to the bot.

7.1.2.3 Rewards

The bot receives a reward of 250 every time the system records that it has caused damage to the opponent with the shooting action. If the bot shoots the weapon and does not hit the opponent it receives of penalty of -1. These values were chosen as they produced the best results during a series of parameter search runs in which the reward value was modified. The reward is adjusted depending on the proximity of hits to other hits when the PCWR technique (which will be described in Section 7.1.3.2) is enabled. If the technique is disabled then the bot will receive 250 for every successful hit.

7.1.3 Implementation Details

In this section, we discuss the use of the SARSA(λ) algorithm and the values of the algorithm parameters that we used. We then present our technique of Periodic Cluster-Weighted Rewarding, for shaping the reward received, which is followed by a discussion of Persistent Action Selection that involves maintaining the same action choice over multiple time steps.

7.1.3.1 On the use of the SARSA(λ) Algorithm

The shooting architecture uses the SARSA(λ) [Sutton & Barto, 1998] reinforcement learning algorithm with two variations to its conventional implementation. The bot calls the *logic* method (which drives its decision-making) approximately every quarter of a second. This is required so that the bot is capable of reacting in real-time. It is possible to adjust how often the bot calls the logic method but if it is increased to half a second or a second then the bot becomes visibly less responsive. We have identified that if the bot is selecting four shooting directions a second there is sometimes a credit assignment problem in which the reward for a successful hit will be incorrectly assigned to a different action that was selected after the initial action that caused the damage. This is due to a delay in the time it takes to register a hit on the opponent after selecting an action. We address this problem through the use of Persistent Action Selection, discussed later in Section 7.2, by ensuring that when an action is chosen, it remains the chosen action for a set number of time steps before a new action is chosen. The algorithm is still run four times a second with the perceived state changing at the this rate, however, the action-selection mechanism is set up to repeatedly select the same action in groups of three time steps. The states, and their corresponding state-action values, will continue to change at each time step but the action chosen will remain the same over three time steps. The reward can also be adjusted by PCWR as described in the next section.

We initialised all of the values in the state-action and eligibility trace tables to zero and used the following parameters for the SARSA(λ) algorithm. The learning rate, α, was set to 0.7. The discount parameter, γ, was set to 0.5. The eligibility trace, λ, was set to 0.9. The significance of these parameters were discussed earlier in Section 2.2.3. The ϵ-greedy action-selection policy was used with the exploration rate initialised at 20%. This was reduced by 3% every one hundred deaths and when it reaches 5% it remains at this level. This percentage determines how often the bot will explore new actions as opposed to exploiting previously learned knowledge. During the early stages of learning we encourage exploration of different shooting strategies. We do not reduce exploration below 5% to ensure the behaviour does not become predictable and that new strategies

can be explored a small percentage of the time later in the learning process. A detailed explanation of the SARSA(λ) algorithm can be found earlier in Section 2.2.3.3. In our refined shooting architecture, the states and actions for each step are stored and the updates are carried out in sequence once the shooting period has ended. This enables us to shape the reward using PCWR if required. The process is illustrated in Figure 7.4.

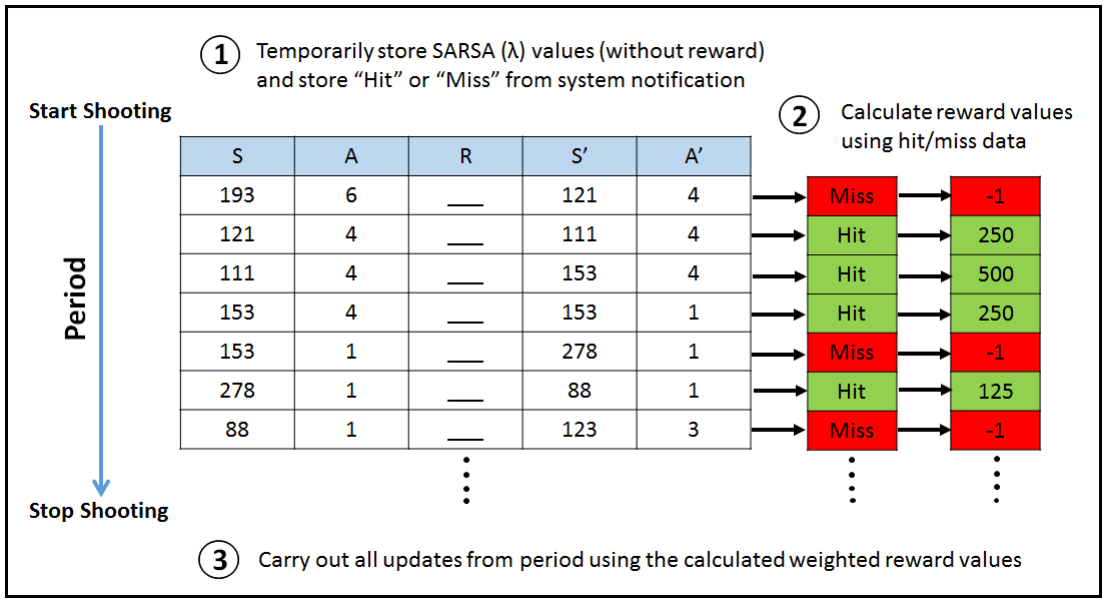

Figure 7.4: An illustration of how Periodic Cluster-Weighted Rewarding works.

7.1.3.2 Periodic Cluster-Weighted Rewarding

Periodic Cluster-Weighted Rewarding (PCWR) involves weighting the percentage of reward that is applied to an action that successfully caused a damage "hit" to an opponent, while hits that occur in clusters of other hits will receive greater reward. If a single standalone hit occurs then the action receives half of the reward. The purpose of this is to promote behaviour that is indicative of good FPS gameplay. This is achieved by providing more reinforcement to actions which resulted in groups of hits on the opponent. If a number of hits occur in a

row, the two outermost (the hits that occur next to misses) receive the full 250 reward[1]. All of the other hits inside the cluster receive double the reward value (500). This is illustrated in Figure 7.5.

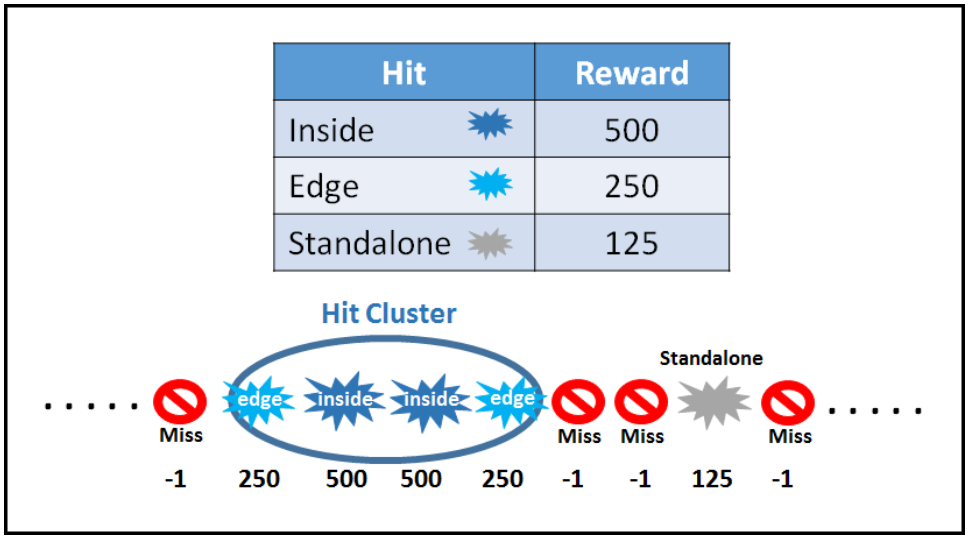

Figure 7.5: An illustration of hit clusters and reward allocation for hits.

The purpose of this is to increase the positive reinforcement of actions that lead to the bot causing a significant amount of damage to the opponent. From when the bots starts shooting to when it stops we have termed a shooting "period". During this period, all of the states, actions and hit/miss values are recorded as they happen. Once the shooting period ends, all of the Q-table updates are carried out in sequence with the cluster weighting having been applied to the reward value.

7.1.3.3 Persistent Action Selection

As mentioned earlier, the bot reads information about its current state from the system four times a second. These short time intervals are required to enable the bot to perceive and react to situations in real-time. Persistent Action Selection

[1]The value 250 was chosen for the full reward as it produced the best results in preliminary runs in which various values were tested between 1 and 1000.

(PAS) involves choosing an action and then keeping this as the selected action over multiple time steps. The states are still changing at each time step, and being read every 0.25s, but the actions that are selected are being persisted over multiple time steps. The purpose of this is to minimise the occurrences of mis-attribution of reward in a setting where a new action could have been selected before the reward was received for the previous action. Persisting with the same action also naturally amplifies the positive or negative reinforcement associated with that particular action. If it is an action that does not lead to any hits then it will be less likely chosen in the future. Although the actions persist over n time steps, the bot's response time will continue to be every 0.25s. This is illustrated for 3 time steps in Figure 7.6.

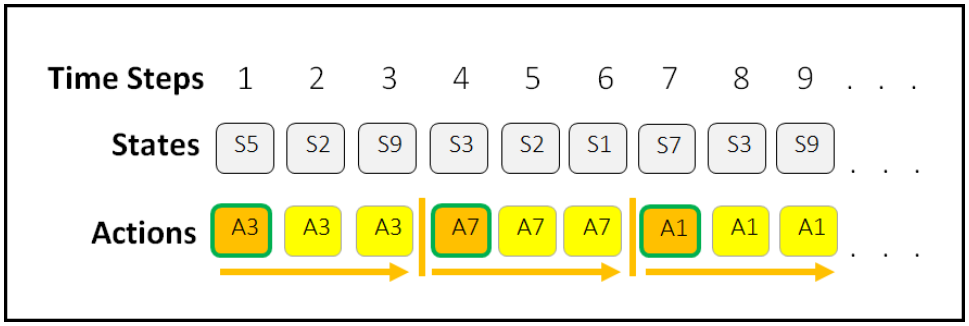

Figure 7.6: An illustration of the workings of Persistent Action Selection.

Therefore, since actions are specified relative to the current location of the opponent, the shooting direction will change as the opponent moves, even when the action is being persisted. Intervals in the range of 2 to 10 were all tested and the value 3 was chosen as it produced the best hit accuracy and the shooting behaviour looked most natural to human observers.

7.1.4 Experimentation

The experimentation that we describe in this section includes the analysis of four different variations of the shooting architecture. These include PCWR enabled with PAS enabled over three time steps, PCWR enabled with actions selected

at every time step, PCWR disabled with PAS enabled over three time steps and PCWR disabled with actions selected every time step. From here on, these will be abbreviated as *PCWR:Yes_PAS:Yes*, *PCWR:Yes_PAS:No*, *PCWR:No_PAS:Yes* and *PCWR:No_PAS:No* respectively. Ten individual runs were carried out for each of these variations in order to analyse the averaged results. All of the games were played out in real-time.

7.1.4.1 Details

Each run involves the RL bot (one of four variations as described earlier) taking part in a Deathmatch game against a single Level 3 (Experienced) fixed-strategy opponent and continually playing until it has died 1500 times. These games were played out in real time with each individual game time taking approximately 7.5 hours to complete. All of the runs took place on the *Training Day* map. This is one of the default maps from the game that is designed for two to three players and encourages almost constant combat. The map was chosen specifically for this reason, to minimise the amount of time that players would spend searching for each other. The only gun available for the duration of the game is the *Assault Rifle*, which is a low powered weapon that shoots a consistent flow of bullets, and which each player is equipped with when they appear on the map. The accumulation of kills and deaths are recorded and the number of hits, misses and rewards per life are also recorded.

7.1.4.2 Results and Analysis

Table 7.2 shows the average hits, misses and rewards per life that the bot achieved when PCWR and PAS were both enabled and disabled. The values shown are averaged over 10 runs in each case. The first observation that we can make is that the average hits per life is much greater when the actions are persisted over 3 time steps. When PAS is enabled the bot achieves almost double the number of hits per life whereas there is little change in the number of misses per life. We can also see that the average total reward per life is almost halved when PCWR is enabled. This is the result of a large number of isolated hits occurring during the games in which the bot only receives 50% of the reward for the hit. Since the

reward scheme is different when PCWR is enabled and disabled, reward averages would not be expected to be within a similar range. Of the two instances in which PCWR is disabled, the average reward is doubled when the PAS technique is enabled.

Technique	Hits	Misses	Reward
PCWR:Yes_PAS:Yes	25.64	40.94	3753.60
PCWR:No_PAS:Yes	26.22	42.18	6512.72
PCWR:Yes_PAS:No	12.99	39.31	1530.19
PCWR:No_PAS:No	13.39	39.48	3307.29

Table 7.2: Averages per life for hits, misses and rewards.

The overall percentage shooting accuracy for each scenario is listed in Table 7.3. This accuracy is averaged over 10 runs for each and is calculated as the hits divided by the total shots taken. When PCWR is enabled using PAS, the bot achieves slightly better accuracy than when it is disabled. We can once again see a large difference in performance between when the PAS technique is enabled and when it is disabled. This table also lists the best kill streak achieved by each of the bots over all of the runs.

Technique	Accuracy	Kill Streak	Hours Alive
PCWR:Yes_PAS:Yes	38.51%	15	17.55
PCWR:No_PAS:Yes	38.33%	14	17.95
PCWR:Yes_PAS:No	24.84%	7	12.90
PCWR:No_PAS:No	25.32%	7	13.04

Table 7.3: Overall average percentage accuracy, maximum kill streak and average hours alive per game.

A kill streak is achieved when the bot kills an opponent several consecutive times without dying itself. Although we have not designed the bot to maximise

the skill streak that it can achieve, we can observe that it is a direct result of learning to proficiently use the weapon. The PCWR:Yes_PAS:Yes bot achieved a maximum kill streak of 15 on 3 of 10 game runs. The PCWR:No_PAS:Yes bot achieved a maximum kill streak of 14 whereas the two bots that choose an action every time step both only managed to reach a maximum kill streak of 7. The final information from this table is the average total time alive for each of the bots for the 1500 lives. Proficient shooting skills result in the bot staying alive for longer and there is an average difference of between 4 and 5 hours when actions are persisted over 3 time steps as opposed to being selected every time step.
Table 7.4 shows the average, minimum and maximum final kill-death ratio after 1500 lives over 10 runs. The kill-death ratio is calculated by dividing the number of kills the bot achieves by the number of times it dies. For instance, if the bot kills 5 times and has died 4 times then its kill-death ratio would be 1.25:1.

Technique	**Average**	**Min**	**Max**
PCWR:Yes_PAS:Yes	2.17:1	2.01:1	2.36:1
PCWR:No_PAS:Yes	2.18:1	2.05:1	2.28:1
PCWR:Yes_PAS:No	0.82:1	0.78:1	0.93:1
PCWR:No_PAS:No	0.86:1	0.77:1	0.92:1

Table 7.4: Average, minimum and maximum final kill-death ratio after 1500 lives over 10 runs.

On average the bots that use PAS over 3 time steps kill the opponent over twice as often as they die. The two bots that select a new action every time step always die more times than they are able to kill. The single best kill-death ratio overall was achieved by the PCWR:Yes_PAS:Yes bot, however, it performs slightly worse on average than the PCWR:No_PAS:Yes bot. Kill-death ratio gives a good high-level indication of performance in FPS games. The trend for the percentage of hits over time, as learning is occurring, for each of the bots is shown in Figure 7.7. These values are averaged over the 10 games for each bot with the points on the graph also being averaged in 10-point buckets. Thus, there are 150 points on the graph to depict the hit percentages over the 1500 deaths.

The first observation that we can make from the illustration is the separation in performance depending on whether PAS over 3 time steps is enabled or not. When actions are selected in every time step, the performance begins at about 20 percent hit accuracy and finishes just over 25 percent. However, the hit accuracy rises to 40 percent when PAS is enabled. This graph shows no clear distinction between the performance of enabling or disabling PCWR, as the averaged results fall within the same range.

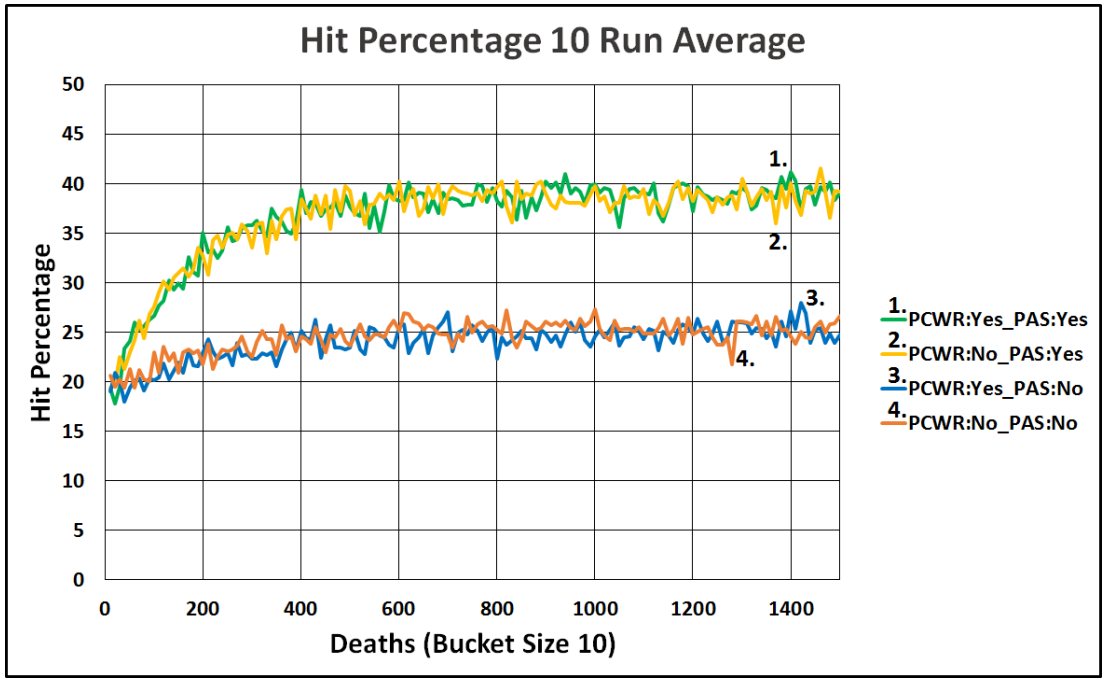

Figure 7.7: Percentage of hits for each variation of the architecture.

The following illustration, in Figure 7.8, shows the average reward received per life for the PCWR:Yes_PAS:Yes bot. As before, these values are averaged over 10 games with the points on the graph also being averaged in 10-point buckets. Thus, there are 150 points on the graph to depict the reward received over the 1500 lives. This illustration shows that the bot is, on average, receiving more reward as it learns from experience over time.

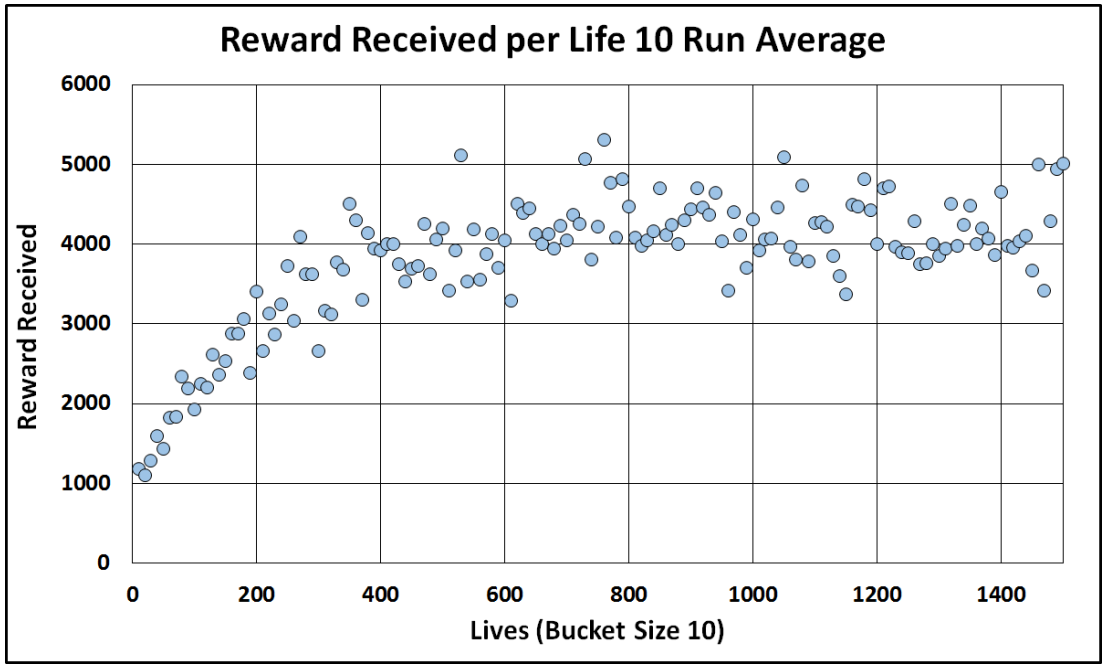

Figure 7.8: Reward received per life for the PCWR:Yes_PAS:Yes bot.

7.1.5 Discussion

These results show clear evidence that the RL shooting architectures are capable of improving a bot's shooting technique over time as it learns the correct actions to take through in-game trial and error. The bot updates its state-action table, which drives its decision-making, after every shooting incident. These are carried out as mass updates once the shooting period has ended. The heat maps in Figure 7.9 show the percentage of actions selected by the bot at the following stages: PCWR:Yes_PAS:Yes after 150 lives; PCWR:No_PAS:No after 1500 lives; PCWR:Yes_PAS:Yes after 1500 lives. The shooting actions are those that were illustrated earlier in Figure 7.3, which span eleven directions across the opponent at four different height levels.

The heat maps clearly show the strategies that are adopted by each of the bots at the different stages of learning and the difference that selecting actions

PCWR:Yes_PAS:Yes Percentage Actions Choosen after 150 Lives										
1.42	1.58	1.96	1.98	1.92	1.72	2.15	2.22	2.34	2.83	2.22
1.84	2.06	2.44	2.51	2.57	1.39	2.77	2.22	1.91	1.94	1.86
1.58	2.36	2.62	2.12	3.35	2.57	2.06	3.33	2.20	3.21	2.06
2.48	2.03	2.98	2.05	2.20	2.27	2.77	2.72	2.74	2.13	2.34
PCWR:No_PAS:No Percentage Actions Choosen after 1500 Lives										
1.48	1.47	1.63	1.51	1.57	1.74	1.55	1.54	1.58	1.52	1.49
1.55	1.60	2.06	1.84	2.42	1.92	2.07	1.91	1.86	1.67	1.58
1.48	1.72	2.75	3.09	3.83	3.95	3.80	2.92	2.35	2.25	1.41
1.51	2.55	2.69	3.51	4.31	4.90	4.01	2.97	2.39	2.23	1.81
PCWR:Yes_PAS:Yes Percentage Actions Choosen after 1500 Lives										
0.39	0.52	0.54	0.46	0.62	0.52	0.49	0.47	0.41	0.49	0.36
0.47	0.37	0.68	0.80	0.92	1.01	0.67	0.58	0.51	0.50	0.38
0.45	0.68	1.24	2.35	5.81	5.78	6.63	1.62	0.95	0.62	0.53
0.43	1.01	2.10	5.10	14.80	15.93	14.89	3.87	2.03	0.54	0.49

Figure 7.9: Percentage of overall shooting actions selected after 150 lives and after 1500 lives.

over multiple time steps can make. The diagrams show the percentage of time each shooting target was selected: at early stages (top of figure), shooting actions are widely dispersed; after learning with SARSA(λ) over 1500 lives (middle graph) the bot shows a clear preference for shooting where opponents are most likely to be found and avoiding other areas such as corners; and after learning with SARSA(λ) including our PAS and PCWR techniques, it shows even stronger trends in shooting preferences. We have demonstrated that selecting the same action over multiple time steps can lead to much better performance than when

new actions are selected every time step. Our PCWR technique, when enabled with PAS, achieved the maximum kill streak, highest overall accuracy and maximum kill-death ratio, however, it will need further refinements to validate its usefulness as, on average, its performance was similar or worse than when the technique was disabled. On the other hand, enabling PAS provides clear performance benefits, despite being a simple technique.

Having produced a consistent upward trend in the bot's shooting performance, we used these techniques to run a series of training games to populate a catalogue of experience for our skill-balancing mechanism which we describe in the next section.

7.2 Skilled Experience Catalogue

In this section, we introduce a skill-balancing mechanism called *Skilled Experience Catalogue* (SEC) that we developed which adjusts the learner bot's proficiency with a weapon based on its current performance against an opponent. Firstly, a catalogue of experience, in the form of stored Q-tables, is built up by playing a series of training games. Once the bot has been sufficiently trained, the catalogue acts as a timeline of experience. If the bot is performing poorly, it can jump to a later stage in the timeline in order to be equipped with more knowledge. Likewise, if it is performing significantly better than the opponent it will jump to an earlier stage. The bot continues to learn in real-time while it is playing but its base knowledge is adjusted, as required, by loading the most suitable Q-tables.

7.2.1 Motivation

Koster [2013] states that the "Holy Grail" of game design is one in which the challenges are constant and require varied skills from the player. The authors propose that this also includes a perfect difficulty curve which adjusts itself, as required, to closely match that of the player. We believe that modern computer games can often lack flexibility with regards to difficulty settings and this can lead to mismatches between the player's ability and the overall difficulty of the game. Dynamic Difficulty Adjustment (DDA) [Hunicke & Chapman, 2004] involves identifying the player's performance and skill level, and then dynamically adjusting the difficulty level accordingly. The goal of this is to ensure that the game remains challenging and can cater for many different players of varying skill levels.

The approach that we are proposing is novel in that it is using a by-product of the bot's learning process to create *milestones* which represent the knowledge acquired at the different stages of learning. The learning tables of the bot are written out to files, at different stages, in order to keep a sequential catalogue of experience. The bot can then jump to the most appropriate base stage of learning to coincide with the skill level of the current opponent while continuing to adapt based on its in-game experience. Our approach does not require manually optimising parameters to represent different skill levels. Conversely, we are sampling

from the natural learning progress of the agent over time.

7.2.2 Implementation Details

The bot must play a series of games against an opponent in order to populate the catalogue of experience. Each time the bot dies it will write out its Q-value table to a file. Once this training period has been completed, milestone Q-tables (from set intervals of the learning timeline) are selected and placed into the SEC. We have chosen 100 deaths to represent a milestone in our initial implementation. That is to say that a milestone Q-table is added to the SEC after 100 deaths, 200 deaths, 300 deaths and so on. Figure 7.10 illustrates how changing the milestone affects the skill level of the bot.

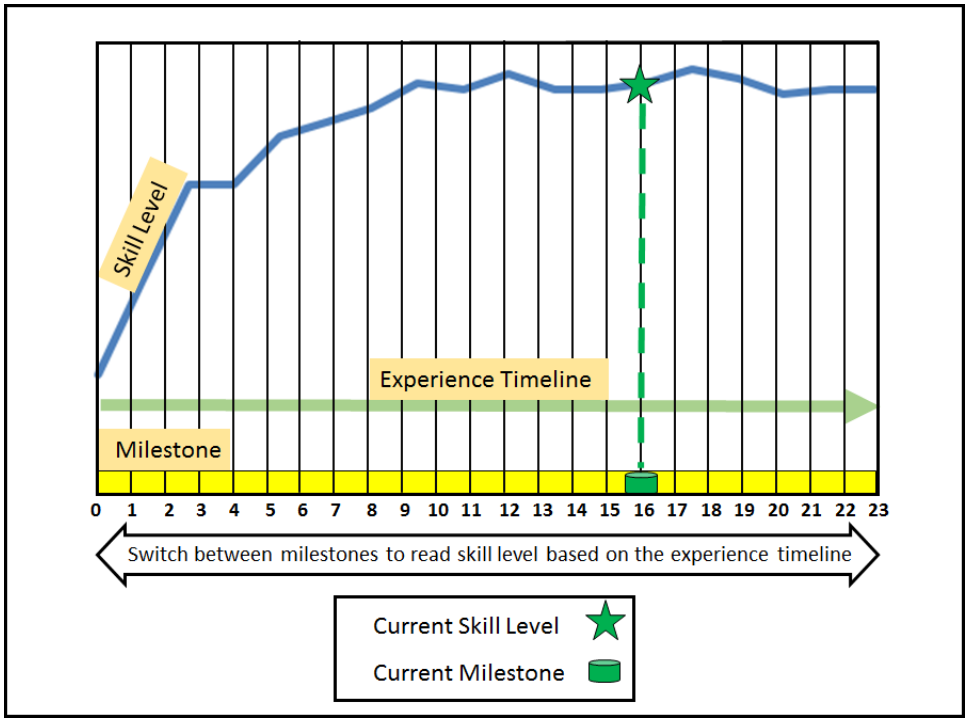

Figure 7.10: A graphical illustration of Skilled Experience Catalogue.

Once the SEC has been populated with the desired number of milestones, then the mechanism for changing milestones has to be set. Our initial implementation

uses a simple positive/negative threshold for kill-death differences (KDDs) to drive the switching between milestones. Specifically, if the KDD between the bot and the opponent exceeds 5, the learning table will switch back to the previous milestone. It will continue to step back through the milestones after every death while the KDD remains above 5 or until it has reached the first milestone (no knowledge). If the KDD falls below -5, the next milestone will be chosen from the SEC after every death until either the KDD returns to the *match range* ($-5 \geqslant KDD \leqslant 5$) or the highest milestone has been reached. The current milestone will remain unchanged while the KDD remains within the match range.

The learning algorithm that is controlling the bot is as described in Section 7.1.3. Both PCWR and PAS are enabled and all of the algorithm settings remain as described in that section.

7.2.3 Experimentation

The first step of our experimentation involved training the bot by playing it against a Level 5 opponent. For these experiments we enable both PCWR and SEC while setting the parameters as described in 7.1.3. We ran one hundred 1-vs-1 Deathmatch (30 minute) games on the Training Day map. The resulting data per game, for how often the bot killed the opponent versus being killed by the opponent (WasKilled) is illustrated in Figure 7.11. The linear trend of the performance over time is also plotted. We can see from the graph that the learner bot performs poorly during the early games in which it dies much more often than killing the opponent. This would be expected as the bot needs time to experience all of the game states and experiment with the different actions. The bot begins to outperform the opponent at the half way point of the 100 game. The linear trends in performance cross paths after 75 games at which point the trend for killing the opponent more than dying is observed.

Figure 7.12 shows the number of wins and losses that the bot recorded against the Level 5 opponent during the 100 training games. The bot has to play 15 games before it manages to defeat the Level 5 opponent. The bot manages to beat the opponent much more frequently when it passes the half way point of the 100 games. This verifies that the bot is in fact learning how to play more proficiently

as it gathers experience. The bot wins 39 out of the 100 games with 28 of these wins occurring in the second half of the games played. During the 100 games, the bot died almost 9000 times. As a result of this we stored 90 milestone Q-tables. These were from no experience (empty Q-table) up to having died 8900 times in increments of 100.

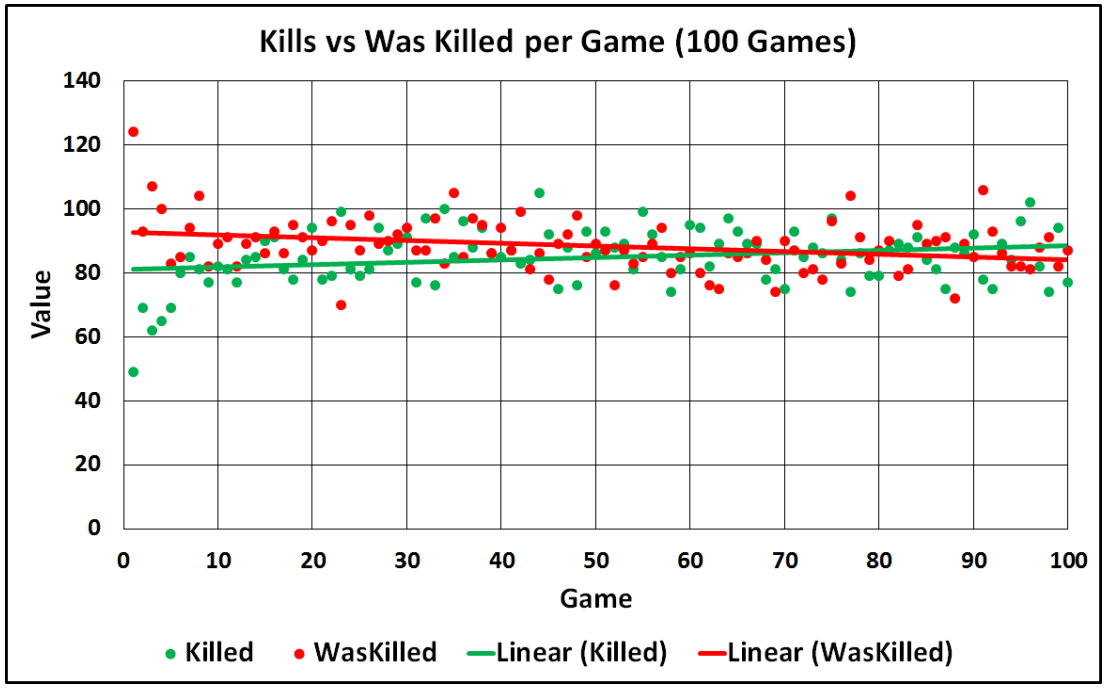

Figure 7.11: Killed vs WasKilled results for the one hundred 30 minute training games.

Table 7.5 compares the average kills, deaths and KD ratio from the first 25 games (First Quarter - Q1) to the last 25 games (Fourth Quarter - Q4) in order to test if the enabled learning is leading to a statistically significant improvement in the average performance in the latter stages of learning. The ** in the table entries signifies that there are statistical differences with significance level α equal to 0.05 using a unpaired two-tailed t-test. It is important to note that in each successive game the bot begins with the knowledge that it has built up from all of the previous games. Therefore the examples are not strictly independent (as

memory from each game is persisting) but it appears that there is sufficient evidence from Figure 7.11 to consider them suitable for this direct comparison. The settings of the individual 30-minute games and the opponents remain consistent throughout the 100 games. Both sample sets comprise individual games that take place in a period of 12.5 hours of learning.

The purpose of this comparison is to check for a statistically significant difference in the average performance between these two learning periods (early and later learning) in order to verify that the bot continues to improve its performance over time. For instance, if there was no significant difference between the bot's average performance during the period of 0 to 12.5 hours and the bot's average performance during the period of 37.5 to 50 hours then we could conclude that the effect of learning on the bot's performance had already plateaued during the earlier stages of learning. Table 7.5 shows that this is not the case and that the average number of kills achieved and the average kill-death ratio has improved, at a 95% confidence level, in the later learning period. We can see from Figure A.11 that the bot continues to encounter new states throughout all of the training games and therefore may have an opportunity to learn a better policy in the latter stages of the training phase.

	Q1: Games 1 to 25 Avg (Std Error)	**Q4: Games 75 to 100 Avg (Std Error)**
Kills	79.20 (± 2.09)	**84.56 (± 1.46) ****
Deaths	91.60 (± 2.02)	87.52 (± 1.46)
KD Ratio	0.88 (± 0.04)	**0.98 (± 0.03) ****

Table 7.5: Comparison of performance between Q1 and Q4. Level of confidence: *** = 99%, ** = 95%, * = 90%

From here on in we will refer to the bot that uses the SEC as *SEC-Bot*. The SEC-Bot begins each new game that it plays with no experience and then increases or decreases its knowledge as required, based on the threshold, by using the Q-table milestones.

Table 7.6 shows the number of times that the SEC-Bot won, lost and drew

against the first 5 levels of the native fixed-strategy bots. Details about the native bots were described earlier in Section 3.3.3.4. The SEC-Bot won 39, lost 30 and drew 6 of the 75 games that were played.

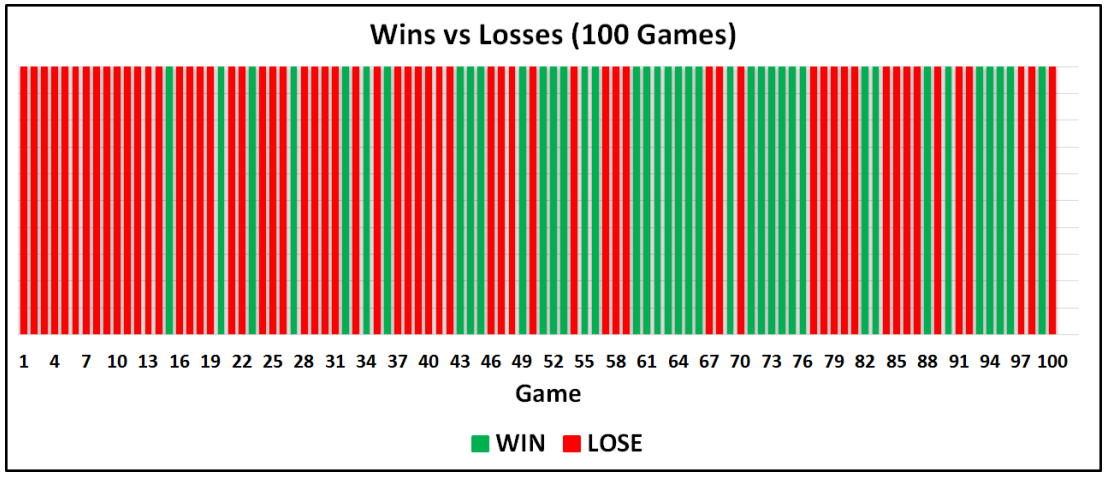

Figure 7.12: Games won and lost during the one hundred 30 minute training games.

Skill	Win	Lose	Draw
Level 1	8	6	1
Level 2	9	6	0
Level 3	8	4	3
Level 4	7	6	2
Level 5	7	8	0

Table 7.6: Wins and losses for the SEC-Bot against each of the 5 levels.

This is a positive indication that the bot is managing to successfully balance the gameplay against the different levels of opponent using a single catalogue of experience. The following figures, from Figure 7.13 to Figure 7.17, show the results of the SEC-Bot playing a total of 15 thirty minute games against five

different levels of opponent. The purpose of this is to observe how well it can match the opponent's skill level, with respect to killing and being killed, by using the SEC mechanism. From the results we can see that the SEC-Bot manages to closely match the kill rate of the opponent over each of the different levels. Figure A.1 to Figure A.5 in Appendix A contrast the kill-death ratios and kill-death differences of all five opponents when SEC is both enabled and disabled.

The milestone changes that occurred for each level of opponent are illustrated in Figures A.6 to Figure A.10 in Appendix A. From these, we can see that, for Level 1, the SEC-Bot never increased to a higher milestone than 0. At the beginning of each game, it started with no knowledge and then only required the in-game learning experience to remain balanced with this opponent. It did, however, have to reset the 0 milestone (clear all knowledge) on many occasions as it was out-performing the Level 1 opponent at the early stages of learning.

The SEC-Bot had to increase the milestones on just two occasions during the Level 2 games whereas the changes were much more prevalent against the more difficult opponents (Level 3, Level 4 and Level 5). The SEC-Bot had to almost immediately rise up through the milestones (once it fell below the threshold), before stopping at the highest point, while playing against the Level 5 opponent. This explains the larger variance that is seen in Figure 7.17 compared to the other results. It is important to recall that the SEC-Bot was initially trained using this level of opponent and it took almost 50 games before it had enough knowledge and managed to start convincingly defeating the opponent.

7.2.4 Discussion and Future Work

The preliminary results for SEC are very promising and show that, through the use of a simple threshold mechanism and "snapshots" from the learning timeline, we can closely match the level of five different fixed-strategy opponents (with varying degrees of proficiency) using a single instance of the SEC mechanism. A successful training phase, in which the SEC-Bot plays a series of games against a single opponent, is a crucial aspect of the success of the skill-balancing mechanism. The SEC-Bot requires a performance timeline that has a clear upward trend

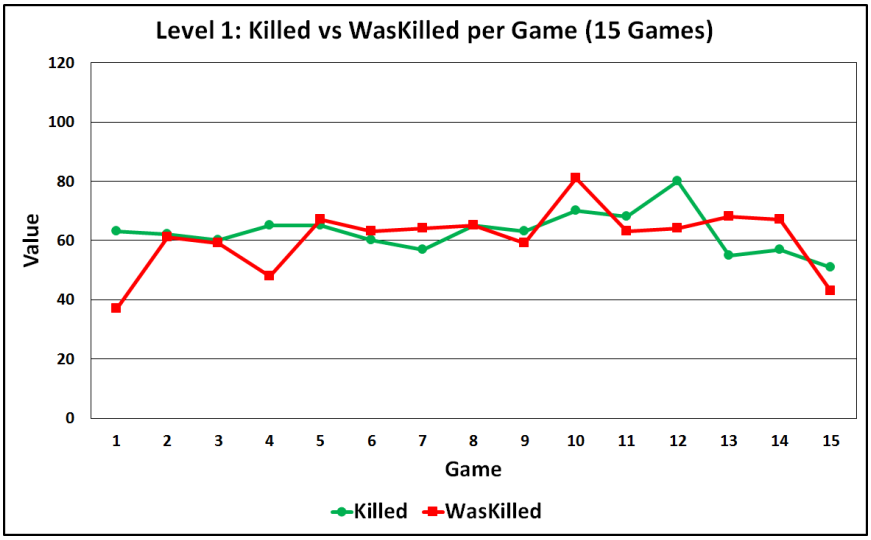

Figure 7.13: Killed vs WasKilled against a Level 1 opponent with SEC enabled.

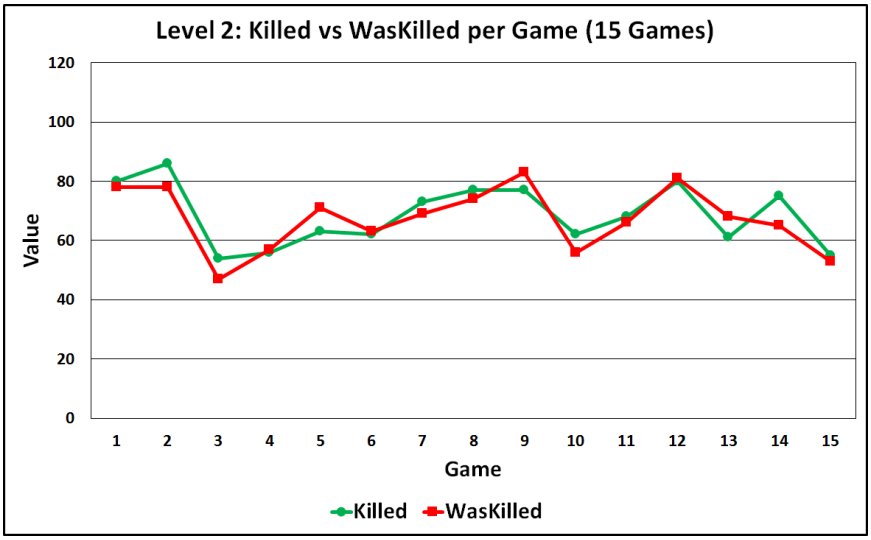

Figure 7.14: Killed vs WasKilled against a Level 2 opponent with SEC enabled.

so that higher milestones explicitly produce better performances. We achieved this upward trend in performance from our earlier research that we described in Section 7.1 (See Figure 7.7).

Some possible refinements that could be applied to the SEC-Bot are as follows.

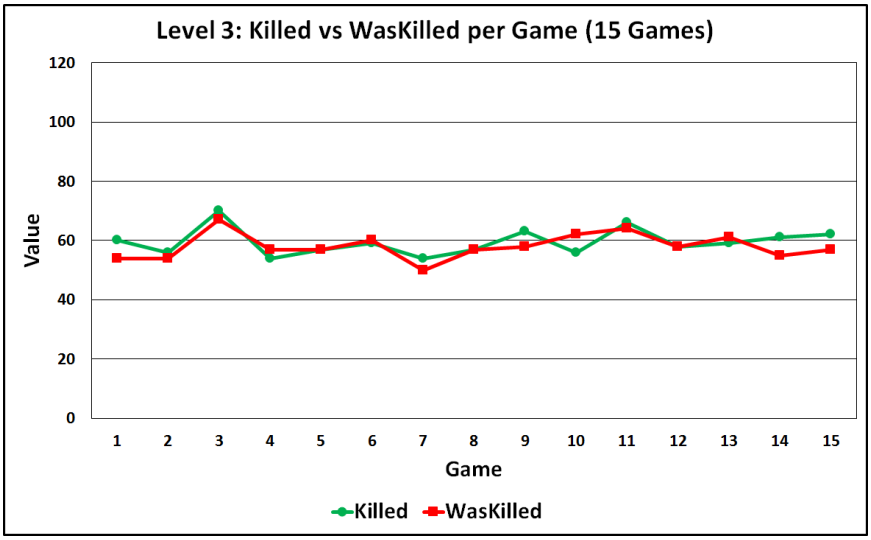

Figure 7.15: Killed vs WasKilled against a Level 3 opponent with SEC enabled.

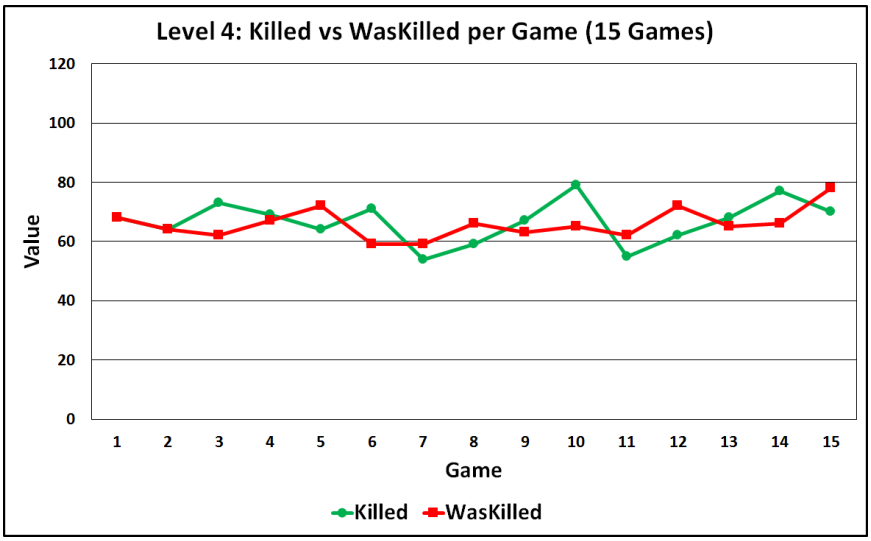

Figure 7.16: Killed vs WasKilled against a Level 4 opponent with SEC enabled.

The bot currently begins each game at the very first milestone which means that it has no knowledge. It may work better to introduce a strategy in which it recalls the most recently used milestone from the last game or selects one at random.

Variations to the threshold used and the milestone switching strategy could

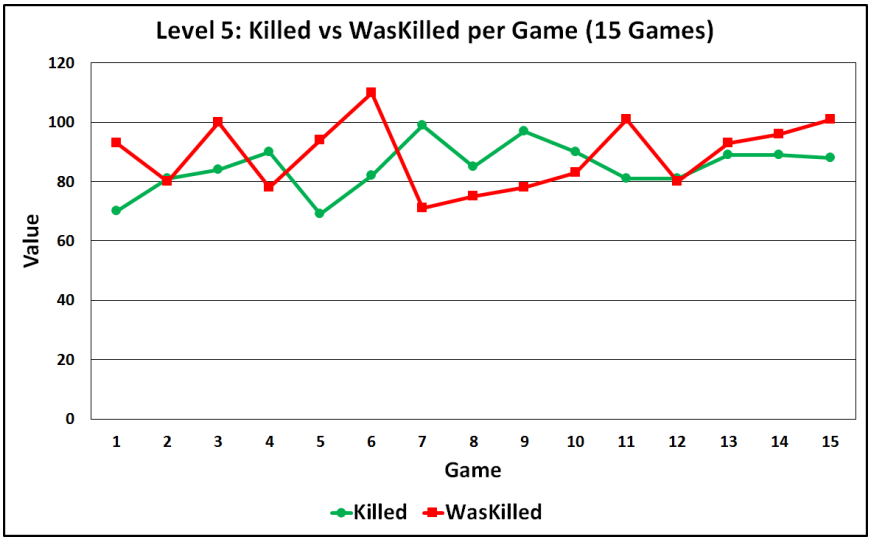

Figure 7.17: Killed vs WasKilled against a Level 5 opponent with SEC enabled.

also be improved. The size of the threshold could be optimised and the rate at which the bot traverses milestones could be adjusted. For instance, one strategy could be to move through several milestones at once to coincide with how far outside the threshold the bot currently is. If the bot is currently three deaths outside the threshold (KDD = -8) then it could move in increments of three through the milestones to speed up the process of balancing the skill level. In the current implementation the SEC-Bot will only change one milestone at a time.

The criteria for selecting appropriate milestones is also another interesting task. Careful performance analysis could aid in the process of milestone creation to determine definitive points of performance improvement which may not follow systematic increments. For instance, we may wish to select a larger number of milestones during the earlier stages of learning and select fewer milestones as the learning begins to plateau.

A milestone/player caching system could be introduced for using the SEC against multiple opponents. Each opponent would have an ID and an associated milestone so that the SEC-Bot could switch to an appropriate milestone based on the current opponent. We designed the initial system to balance play with just a single opponent.

Finally, the SEC-Bot could be trained against different levels of human players as opposed to fixed-strategy bots. Another approach could be to deploy the SEC-Bot on a server, over the Internet, to train it against both the human and computer-controlled players that it encounters.

7.3 Chapter Summary

This chapter has presented and critically analysed some novel techniques that we developed for using reinforcement learning to enable a bot to increase proficiency with a weapon over time in a first-person shooter game. These research efforts were geared towards improving on our previous work from Chapter 6. We also described using a by-product of this learning experience, which we have termed *milestones*, in order to adjust the proficiency of the bot based on real-time performance statistics in an attempt to balance the gameplay. Specifically, the three techniques that were described were Periodic Cluster-Weighted Rewarding (PCWR), Persistent Action Selection (PAS) and Skilled Experience Catalogue (SEC). We limited the experimentation to a single gun in order to closely analyse the results and we succeeded in producing a clear upward trend in performance over time. This was then used to form the basis of our skill-balancing mechanism which, using a single instance of the architecture, was shown to successfully mimic the skill level of five different fixed-strategy opponents.

Chapter 8

Conclusions

In this concluding chapter of the thesis we begin by summarising all of the work that we carried out by presenting a list of our research contributions. This is followed by discussing the main findings and conclusions of our work. Some ideas and directions for future work are then presented before the thesis is brought to a close with some concluding remarks.

8.1 Summary of Completed Work

The following is a list that summarises all of the work that we carried out during this research. Each of these items also represent the principal contributions of this body of work.

- *Carry out a state-of-the-art literature review.* This included a broad look at artificial intelligence in general and then focused on the background information and theory of reinforcement learning. Traditional approaches to game AI were discussed and various machine learning techniques, which are used in conjunction with reinforcement learning, were summarised. We reviewed the general state-of-the-art applications of reinforcement learning before focusing on game AI techniques that are used in the first-person shooter genre of computer games. We also presented a review of relevant testbeds and toolkits that are available for carrying out experimentation using artificial intelligence techniques.

- *Develop a general-purpose NPC behavioural architecture using reinforcement learning for an FPS game.* This "proof of concept" work began with careful consideration of FPS environmental and gameplay features. This included identifying the most relevant information required by the agent and proposing tasks that could be learned. We designed the states, actions and rewards whilst embedding the reinforcement learning algorithm into the logic of the NPC. This work addressed our first research question R.1.

- *Refine the NPC architecture based on experimentation results and analysis.* This involved partitioning the logic of the bot into separate modes to deal with specific circumstances, embedding a reinforcement learning algorithm that uses eligibility traces, as well as refining the design of the states, actions and rewards. Specifically, the bot was afforded more information from the game state and could switch between high-level modes for sensing danger, needing to replenish supplies and exploring the environment. This work addressed our first research question R.1.

- *Develop a behavioural architecture to specifically learn the task of shooting.* We designed and implemented states, actions and rewards for learning the sole task of shooting. The state space was designed to emulate the bot's view of the environment from the first-person perspective as a human player would. This included the features of relative speed, relative moving direction, relative rotation and distance to the nearest visible opponent. Tailor-made actions were designed for several different categories of weapon and the reward for successful hits was taken directly from the damage caused by the specific weapons as recorded by the system. This work addressed our second research question R.2.

- *Analyse any drawbacks identified and propose solutions for the shooting architecture.* We introduced the concepts of Periodic Cluster-Weighted Rewarding and Persistent Action Selection to address the issues with reward allocation that were identified in the RL-Shooter architecture. We also streamlined the experimentation process by concentrating on a single weapon in order to closely analyse the changes in performance over time.

This work addressed our third research question R.3.

- *Develop a reinforcement learning inspired game-balancing mechanism.* We developed a game-balancing mechanism, called Skilled Experience Catalogue, which uses a catalogue of experience in the form of stored Q-tables. The catalogue of experience includes the different stages of learning that are used to closely match the current opponent's ability. The bot generates a timeline of experience by playing a series of training games against an opponent and recording its current knowledge table after each death. Once these games are completed, a group of milestones are selected from the timeline to represent the different stages of learning. The bot then balances the gameplay with the current opponent by switching between these milestones, based on a threshold, while at the same time continuing to learn from in-game experience. This work addressed our fourth research question R.4.

- *Publish our findings in international conferences and journals.* The results of our work were disseminated to the research community, peer-reviewed and subsequently published as listed in Section 1.5.

8.2 Research Findings

Our research led to the design, development and testing of the following behavioural architectures and learning mechanisms:

1. Sarsa-Bot (Chapter 4)
2. DRE-Bot (Chapter 5)
3. RL-Shooter (Chapter 6)
4. Periodic Cluster-Weighted Rewarding (Chapter 7 Section 1)
5. Persistent Action Selection (Chapter 7 Section 1)
6. Skilled Experience Catalogue (Chapter 7 Section 2)

At the beginning of this dissertation, in Section 1.3, we proposed four research questions with corresponding research objectives. In this section, we answer each of these questions by discussing the findings from our research and providing an overall summary of the work.

8.2.1 Research Question R1

R.1: *Can an autonomous reinforcement learning agent use feedback from its environment to guide successful decision-making in a real-time adversarial 3-dimensional world with multiple objectives? What are the benefits and drawbacks of using such an approach?*

We have shown, through the development and testing of the Sarsa-Bot and DRE-Bot architectures, that it is possible to control an NPC using reinforcement learning in an FPS game. Experimentation on static, fixed-strategy opponents provided us with evidence of learning and also raised some questions regarding the assignment of reward when it is being received from multiple sources. While the bots drive their general performances based on maximising the reward they receive, and have the benefit of improving their performances in real-time, they have a limited representation of their surroundings and there is the possibility of delayed reward being assigned to the incorrect state-action pairs. These bots still adapt their behaviour, based on experience from playing against opponents, but we concluded that reinforcement learning would be more suited to learn an individual task that has a single source of reward. For this reason, the focus of our research switched to learning the single task of shooting a weapon.

8.2.2 Research Question R2

R.2: *Is reinforcement learning an effective technique for creating unpredictable and adaptable opponents in a competitive environment by learning to perform a task with a single objective?*

The RL-Shooter architecture addressed the specific task of shooting and categorised the learning task into specific weapon types. A single source of reward is read from the amount of damage that is caused to the opponent. The architecture was tested extensively against three levels of native UT2004 opponents and evaluated on a "per game" and "per life" basis. The results showed that the bot could perform at the same level of an "experienced" native bot. Even though the reward was being read from a single source, the reward attribution problem was still evident from the experimentation results. This lead to the development of new mechanisms for correctly assigning reward to the actions that caused the damage.

8.2.3 Research Question R3

R.3: *What specially designed updating and rewarding mechanisms, if any, can be developed to improve the learning performance of a reinforcement learning agent?*

Periodic Cluster-Weighted Rewarding involves applying a weighting to the reward for hitting an opponent based on its proximity to other recorded hits. The same actions are chosen every three time-steps, through the use of Persistent Action Selection, as opposed to the old architecture in which a new action was selected every time step (at four time steps a second). We carried out experimentation in which opponents could only use a single gun, the Assault Rifle, and the results showed clear evidence of an upward trend in the performance of hit accuracy over time. An analysis of the game logs also showed that the bot had learned to select the best actions more often after building up experience. We illustrated this evidence by using heat maps (see Figure 7.9 in Chapter 7) which show how frequently each action was selected at the different stages of learning.

8.2.4 Research Question R4

R.4: *Can reinforcement learning be used to aid the process of dynamically matching the skill level of an agent with the varied skill levels of several fixed-strategy opponents?*

We designed Skilled Experience Catalogue based on the premise that there is a progressive timeline which begins with poor performances and ends with good performances as the agent learns how to perform a task over time. Consecutive "snapshots" of the Q-tables, which control decision-making, are stored as files during the learning process. This essentially generates a catalogue of proficiency which can then be used to jump to a desired period of learning in order to match the ability of the current opponent. If the current opponent cannot match the level of skill of the SEC learner bot, and is performing poorly, then an earlier version of the Q-table can be loaded when the score difference exceeds a certain threshold. We trained our SEC-Bot against a Level 5 fixed-strategy opponent by playing 100 half hour games and then proceeded to play against five different levels of opponent and successfully balanced the gameplay with all of them.

8.2.5 Overall

This thesis has presented several behavioural architectures which incorporate reinforcement learning into the logic of an NPC in an FPS game. The process of design, implementation and evaluation has been described in detail and has shown promising results for embedding automated decision-making, based on feedback from the environment, into an agent in a complex virtual world. Adversarial combat games provide a challenging domain for deploying artificial intelligence. Instantaneous decision-making and reactive behaviours are required to competently compete with opponents. We have approached this overall research problem incrementally from three different perspectives. These include two general-purpose Deathmatch agents, a controller for learning the task of shooting and a mechanism for balancing gameplay against opponents of varying proficiency. Throughout this thesis we have described the low-level detail of all

of the architectures that we designed and developed. It is hoped that these ideas will inspire researchers to continue to advance the state-of-the-art in artificial intelligence being applied to non-player characters. In the future we hope that our SEC technique, which has shown the most promising results, can be further developed in order to contribute to the overarching goals of producing non-player characters which are challenging, adaptive and unpredictable in their behaviour. In particular, this technique could be applied to any scenario in which reinforcement learning can be used with functionality added to store and load the agents memory (whether in the form of a Q-table, a set of neural network weights, or another form) and sample from this catalogue in order to match a specific threshold of performance. Therefore the technique has a lot of potential in wide variety of domains and with the use of various different learning algorithms. There are many different paths that could be taken with regards to future work. These are described in the next section.

8.3 Directions for Future Work

The following are some ideas and possible directions for future work related to this research. We believe that there is room for further refinement with each of the architectures with respect to the design of the states, actions and rewards as well as addressing some of the issues that were identified. Different learning algorithms could also be introduced and combined with the current implementations.

There are several other individual tasks in an FPS game that could be learned such as navigation, item collection and fleeing from opponents. There are also tasks that are specific to certain game types such as taking control of an area in the game type *Domination* and retrieving enemy flags in *Capture the Flag*. These could be developed and combined with our RL-Shooter architecture.

It would be worthwhile to investigate tailoring our architectures to work in other game environments and genres. There are many other environments, some of which were described in Section 3, which could be used to test each of the architectures. An investigation into whether these environments could facilitate an easier integration of the NPC learning logic would be worthwhile. We believe

that other similar game genres such as Third Person Shooters (TPS), in which the avatar is fully visible on-screen, would be a good starting point.

Further experimentation could be carried out and extended to include different maps and game settings. Limiting the weapons and supplies as well as adjusting the levels of health could also be applied. Trials could be run with human opposition to evaluate the level of entertainment produced by the architectures. Finally, an entry could be submitted to the Bot Prize competition (see Section 2.5.1), or an equivalent competition, in order to evaluate its performance against other approaches.

Periodic Cluster-Weighted Rewarding requires further investigation as a reward shaping technique. This could involve analysing the states and actions that were visited during the period of updates and incorporating this knowledge into the reward shaping process. The process of halving or doubling the reward based on the proximity of hits to each other could be refined and further evaluated.

Additional refinements and applications of SEC could also be identified and developed. The idea of having a catalogue of progress for a reinforcement learning problem could apply to any problem that requires adjusting the progress of learning to match a specified standard with the use of a threshold. Further work could improve the saving of the Q-tables by introducing a database to store the different stages of learning. We have also listed several other possible refinements for this technique in Section 7.2.4.

While it is beyond the scope of this thesis to assess how the SEC technique could improve the entertainment value and challenge for human players in an FPS game, we now propose a methodology by which this could be done. Firstly, a group of FPS players, ranging from very little experience to experienced, should be identified and categorised based on their abilities. These could be undergraduate computer science students or members of gaming/computer societies from the university. An initial "Call for participation" should be circulated to students which will outline the details of the user trials which will take place. Those interested will fill out a short survey and subsequently be categorised into Beginner, Intermediate and Expert groups. We envisage having 10 players in each group. Each group will be split in half with one half playing against an opponent with SEC enabled and the other half playing against an opponent with SEC disabled.

Each human player will play five 10-minute Deathmatch games against a single opponent and will fill out a questionnaire based on their experience once these games have completed. This will allow us to assess the impact of the SEC technique on the gameplay from the perspectives of players will varying skill levels.

8.4 Concluding Remarks

The goal of our research project was to investigate the use of reinforcement learning to drive the behaviour of an NPC in an FPS game. Over the past decade, many research efforts have concentrated on optimising the behaviour of FPS NPCs using, for instance, evolutionary algorithms and neural networks to train the NPC offline and then to deploy it when it has reached a sufficient level of behaviour. The focus of our research was to develop behavioural architectures that would facilitate the continual in-game adaption of the NPC. We believe that the feedback from the environment, and opponents, should have a direct impact on the agent's decision-making analogous to the learning process of a human player. Our research led us on a path of discovery with regards to the suitability of reinforcement learning in the context of an FPS game. We began with two general purpose architectures, called Sarsa-Bot and DRE-Bot, and then focussed specifically on the task of shooting with our RL-Shooter architecture. We then refined the perception of the bot in the environment and concentrated on a single weapon in order to generate a clear upward trend in performance (hit accuracy) over time. This led to our idea of sampling from the bot's natural timeline of progress in order load an appropriate level of competency to match the current skill level of the opponent. Although our shooting and skill-balancing architectures have shown the most promising results, we believe that the journey has been both informative and fruitful in terms of presenting new ideas for applying reinforcement learning in FPS games to produce more interesting and adaptive opponents.

The study of artificial intelligence is a vibrant research area that has applications in many different domains, as shown earlier in Chapter 2. The goal of creating intelligent programs and machines has existed since its inception. There

have been many success stories over the years and computer gaming domains have become established as an ideal platform for testing the latest techniques. Although our research has focussed on FPS games, we believe that the approaches discussed and the new techniques developed in our body of work are relevant and applicable to a wider scope of domains that involve real-time learning in a complex, multi-objective setting.

Appendix A

Additional Results

The following are additional results from our SEC experimentation which we described in Section 7.2.3. Figure A.1 to Figure A.5 clearly show that enabling SEC balances the gameplay with a KD Ratio of 1:1 in each case.

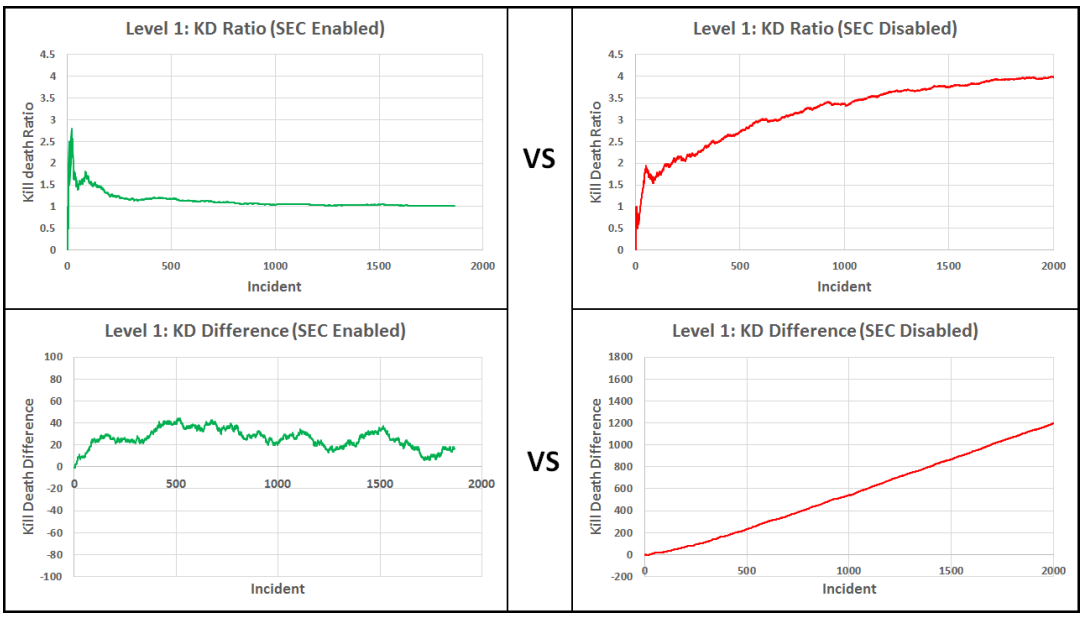

Figure A.1: Level 1 KD Ratio and KD Difference which shows that the max KD Difference is approx. 40 (SEC Enabled) and approx. 1200 (SEC Disabled).

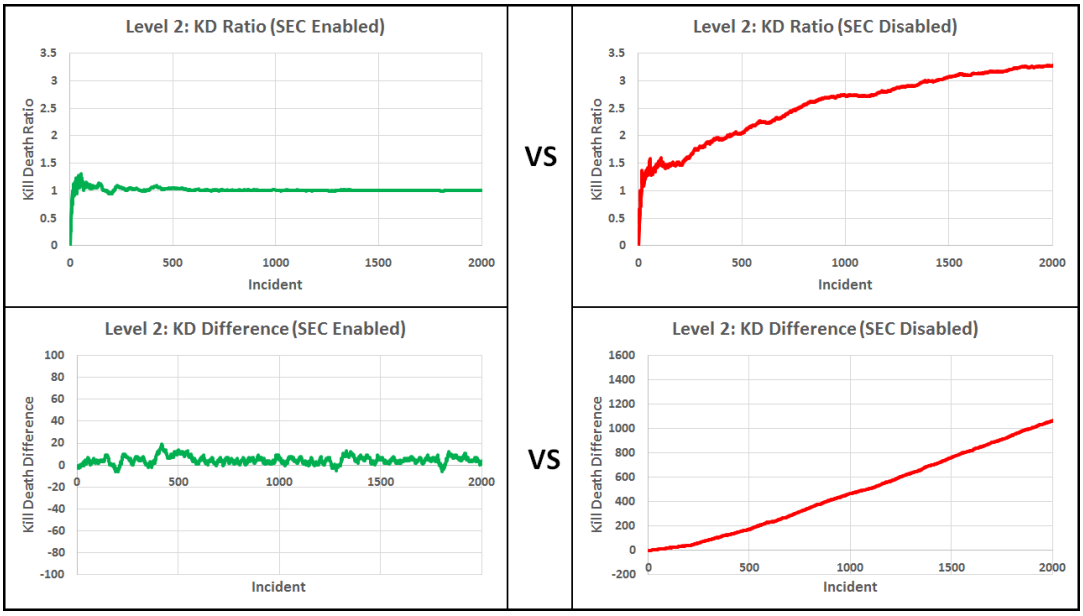

Figure A.2: Level 2 KD Ratio and KD Difference which shows that the max KD Difference is approx. 20 (SEC Enabled) and approx. 1050 (SEC Disabled).

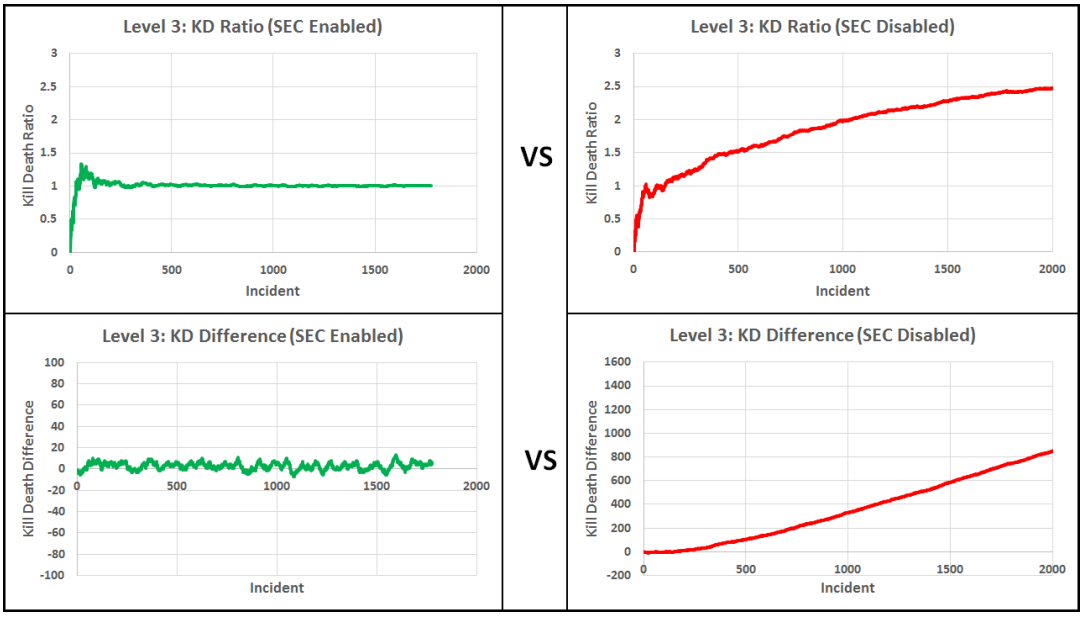

Figure A.3: Level 3 KD Ratio and KD Difference which shows that the max KD Difference is approx. 10 (SEC Enabled) and approx. 920 (SEC Disabled).

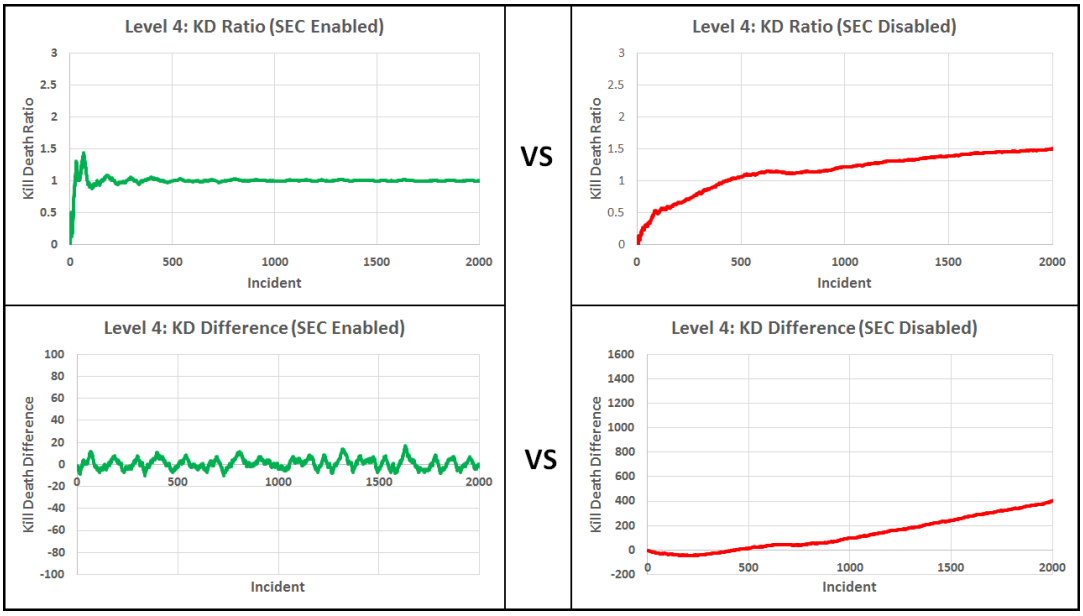

Figure A.4: Level 4 KD Ratio and KD Difference which shows that the max KD Difference is approx. 20 (SEC Enabled) and approx. 400 (SEC Disabled).

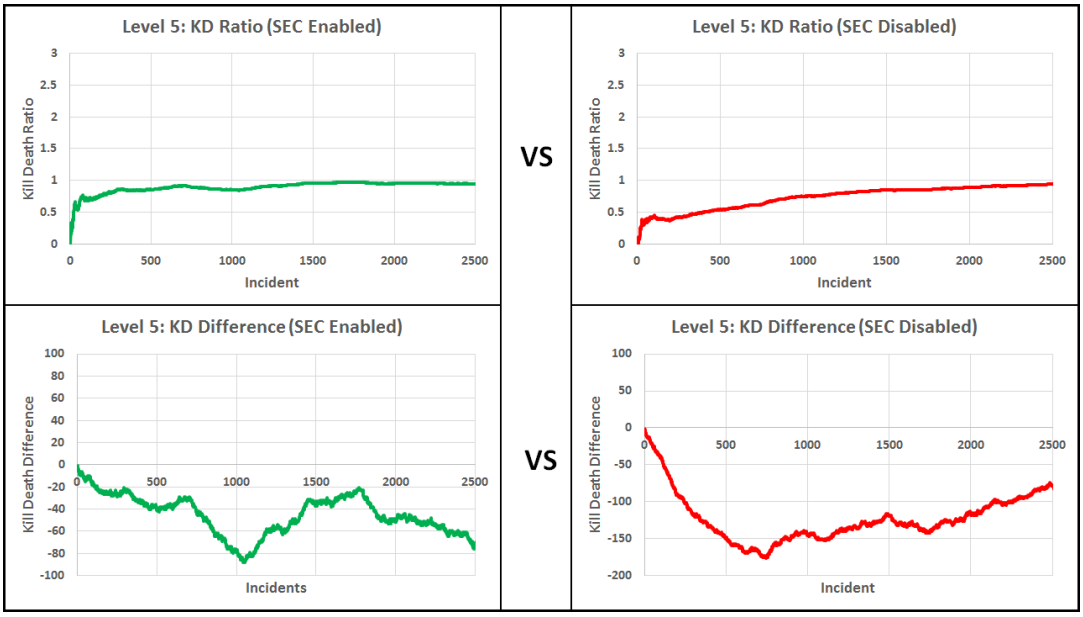

Figure A.5: Level 5 KD Ratio and KD Difference which shows that the max KD Difference is approx. -85 (SEC Enabled) and approx. -175 (SEC Disabled).

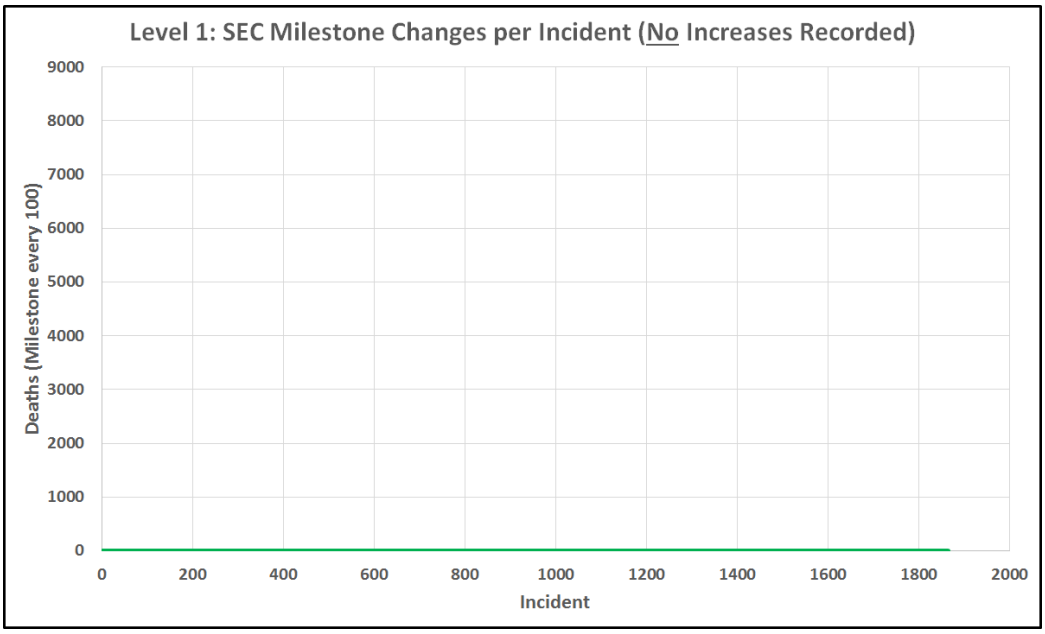

Figure A.6: Level 1: SEC milestone changes. From incident 15 onwards, Q-values are reset to 0 after each incident and SEC still keeps a positive KD Difference.

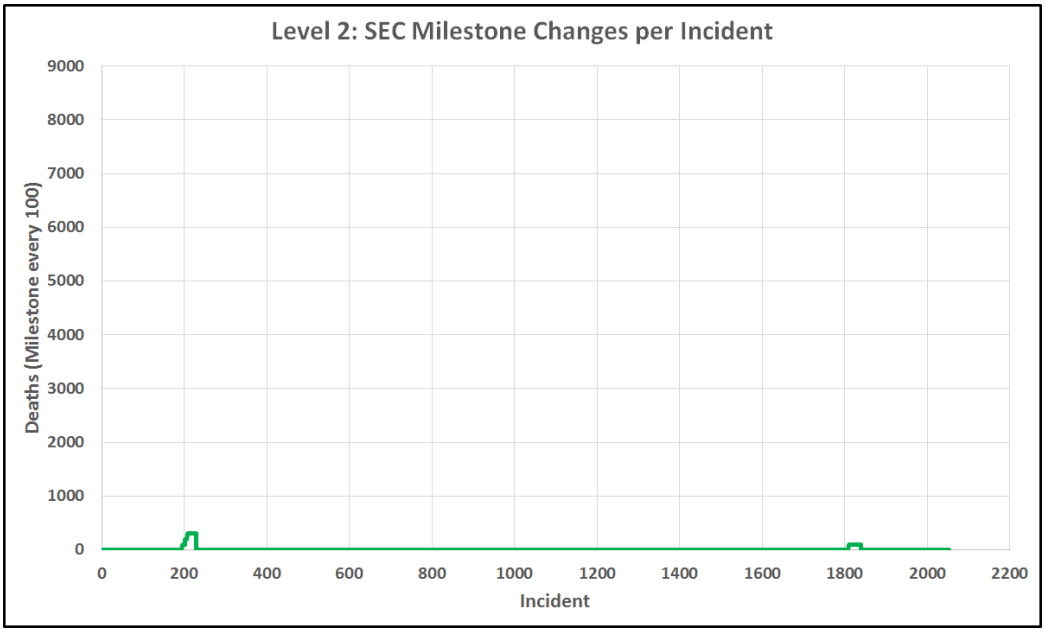

Figure A.7: Level 2: SEC changes to higher milestones on two occasions. Resets (0s), that are not illustrated here, happen 37% of the time.

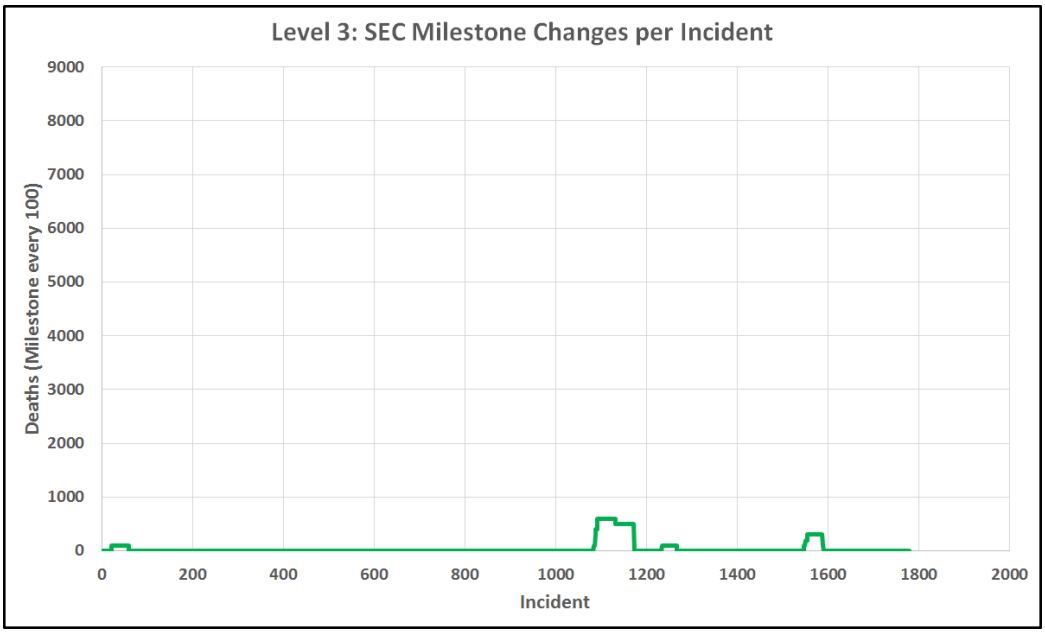

Figure A.8: Level 3: SEC changes to higher milestones on four separate occasions. Resets (0s), that are not illustrated here, happen 23% of the time.

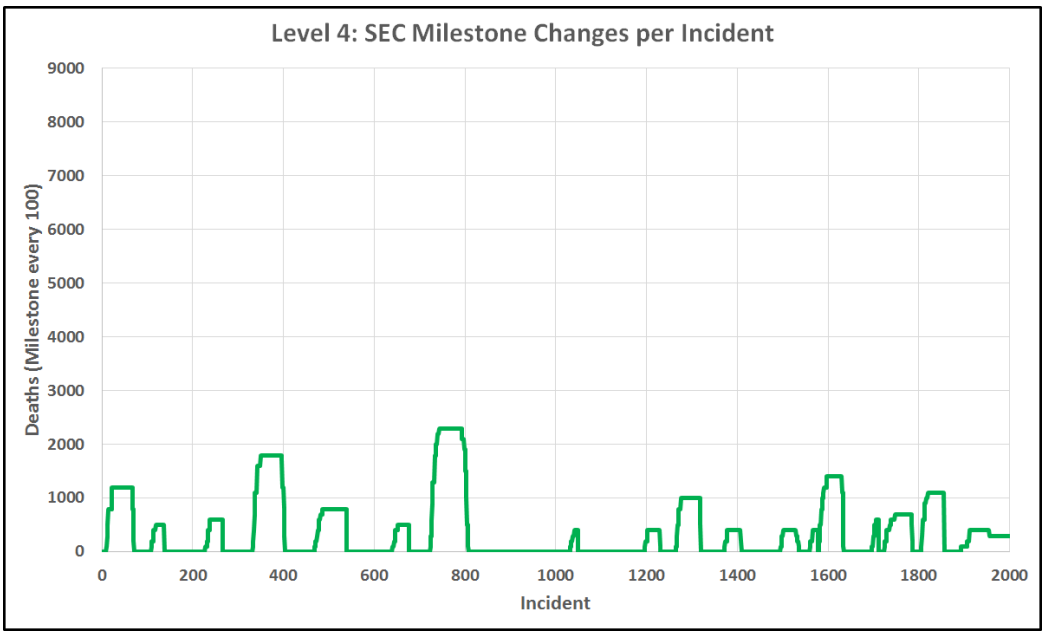

Figure A.9: Level 4: SEC changes to higher milestones on many occasions. Resets (0s), that are not illustrated here, happen 16% of the time.

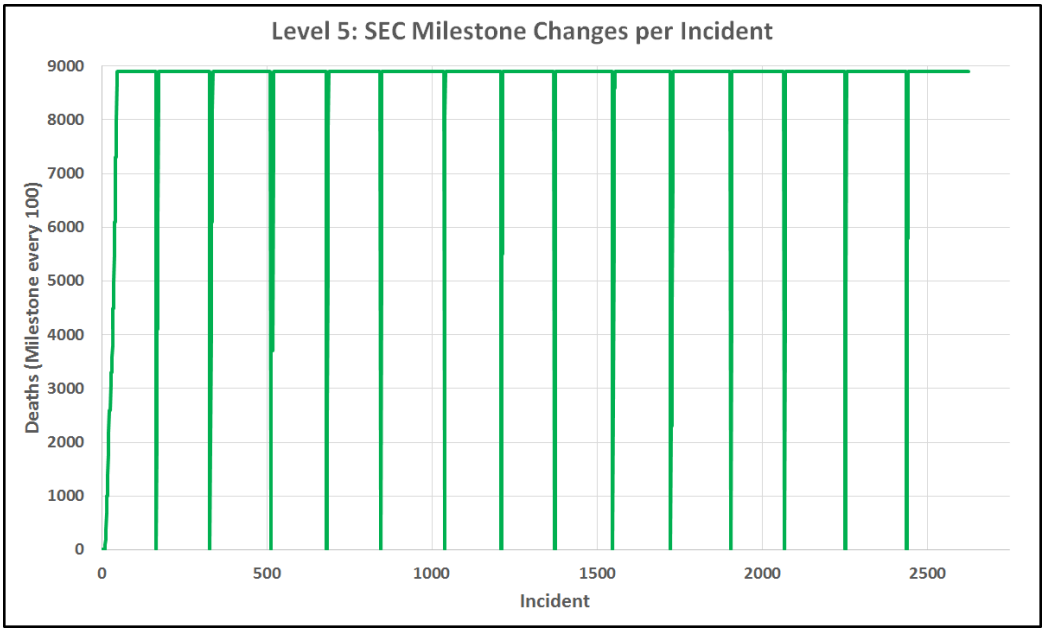

Figure A.10: Level 5: SEC continuously changes to the top milestone in every game. No resets of Q-values to 0 took place.

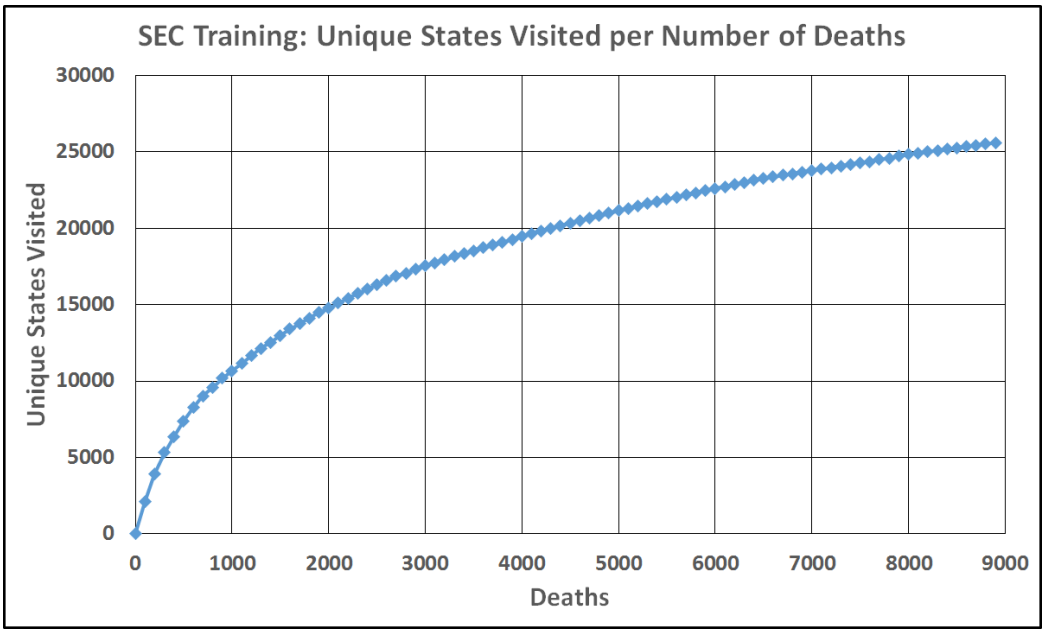

Figure A.11: Number of unique states visited per death during the SEC training game which shows that the SEC-Bot is always encountering new states.

Appendix B

List of Abbreviations

The following are a list of abbreviations that are used throughout this thesis.

AI . . . . . Artificial Intelligence

AGI . . . . . Artificial General Intelligence

ALE . . . . . Arcade Learning Environment

ANN . . . . . Artificial Neural Network

ATC . . . . . Adaptive Tile Coding

AUV . . . . . Autonomous Underwater Vehicle

BAT . . . . . Behaviour Analysis and Testing

BDI . . . . . Belief Desire Intention

BL . . . . . Back Left

BoS . . . . . Battle of Survival

BR . . . . . Back Right

CA . . . . . Critical Ammunition

CAT . . . . . Causally Annotated Trajectory

CBR . . . . . Case Based Reasoning

CEL . . . . . Co-Evolutionary Learning

CGI . . . . . Computer Generated Imagery

CH . . . . . Critical Health

CLsquare . . Closed Loop System Simulation

CMA . . . . . Centred Moving Average

CMAC . . . . Cerebellar Model Articulation Controller

CTF . . . . . Capture The Flag

DARPA . . . Defense Advanced Research Project Agency

DDA . . . . . Dynamic Difficulty Adjustment

DP . . . . . Dynamic Programming

DRE . . . . . Danger Replenish Explore

EA . . . . . Evolutionary Algorithm

EvoTC . . . . Evolutionary Tile Coding

FALCON . . Fusion Architecture for Learning COgnition and Navigation

FL . . . . . Front Left

FPA . . . . . First Person Assessment

FPS . . . . . First Person Shooter

FR . . . . . Front Right

FSM . . . . . Finite State Machine

HA . . . . . Human Advisor

HAT . . . . . Human Agent Transfer

HI-MAT . . . Hierarchy Induction via Models and Trajectories

HRL . . . . . Hierarchical Reinforcement Learning

HSMQ . . . . Hierarchical Semi-Markov Q-Learning

IDE . . . . . Integrated Development Environment

IRL . . . . . Inverse Reinforcement Learning

JMX . . . . . Java Management Extensions

KD . . . . . Kill-Death

KDD . . . . . Kill-Death Difference

kNN . . . . . k-Nearest Neighbour

LA . . . . . Low Ammunition

LH . . . . . Low Health

MLP . . . . . Multi Layer Perceptron

MASTER . . Modelling Approximate State Transitions by Exploiting Regression

MDP . . . . . Markov Decision Process

MoMP . . . . Mixture of Motor Primitives

NEAT . . . . Neuro Evolution of Augmenting Topologies

NN . . . . . Neural Network

NPC . . . . . Non-Player Character

PAS . . . . . Persistent Action Selection

PC . . . . . Player Character

PCWR . . . . Periodic Cluster-Weighted Rewarding

PEGASUS . Policy Evaluation-of-Goodness And Search Using Scenarios

PIQLE . . . . Platform for Implementing Q-Learning Experiments

QASE . . . . Quake 2 Agent Simulation Environment

RETALIATE REinforced TActic Learning In Agent Team Environments

RL . . . . . Reinforcement Learning

RTS . . . . . Real Time Strategy

SARSA . . . State Action Reward State Action

SEC . . . . . Skilled Experience Catalogue

SVM . . . . . Support Vector Machine

TAMER . . . Training an Agent Manually via Evaluative Reinforcement

TD . . . . . Temporal Difference

TL . . . . . Transfer Learning

TPA . . . . . Third Person Assessment

TPS . . . . . Third Person Shooter

TORCS . . . The Open Racing Car Simulator

UT2004 . . . Unreal Tournament 2004

URU . . . . . Unreal Rotation Unit

UU . . . . . Unreal Unit

WEKA . . . . Waikato Environment for Knowledge Analysis

XP . . . . . Experience

References

Aamodt, A. & Plaza, E. (1994). Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. *AI communications*, **7**, 39–59. 37

Acampora, G. (2010). Exploiting Timed Automata-based Fuzzy Controllers and Data Mining to Detect Computer Network Intrusions. In *IEEE International Conference on Fuzzy Systems (FUZZ)*, 1–8, IEEE. 67

Acampora, G., Loia, V. & Vitiello, A. (2012). Improving Game Bot Behaviours through Timed Emotional Intelligence. *Knowledge-Based Systems*, **34**, 97–113. 67

Amato, C. & Shani, G. (2010). High-Level Reinforcement Learning in Strategy Games. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems*, 75–82. 54

Arrabales, R., Ledezma, A. & Sanchis, A. (2009). Towards Conscious-like Behavior in Computer Game Characters. *Symposium on Computational Intelligence and Games*, 217–224. 63

Auslander, B., Lee-Urban, S., Hogg, C. & Munoz-Avila, H. (2008). Recognizing the Enemy: Combining Reinforcement Learning with Strategy Selection using Case-Based Reasoning. *Advances in Case-Based Reasoning*, 59–73. 62

Baars, B.J. (1988). *A Cognitive Theory of Consciousness*. Cambridge University Press. 63, 66

Bajcsy, R. (2014). Robots Are Coming. *Communications of the Association for Computing Machinary*, **57**, 42–43. 71

Baker, S. (2011). *Final Jeopardy: Man vs. Machine and the Quest to Know Everything*. Houghton Mifflin Harcourt. 19

Banerjee, B. & Stone, P. (2007). General Game Learning Using Knowledge Transfer. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 672–677. 52

Barrett, S., Taylor, M.E. & Stone, P. (2010). Transfer Learning for Reinforcement Learning on a Physical Robot. In *9th International Conference on Autonomous Agents and Multiagent Systems-Adaptive Learning Agents Workshop (AAMAS-ALA)*. 46

Barto, A.G. & Mahadevan, S. (2003). Recent Advances in Hierarchical Reinforcement Learning. *Discrete Event Dynamic Systems*, **13**, 41–77. 52

Bellemare, M.G., Naddaf, Y., Veness, J. & Bowling, M. (2013). The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, **47**, 253–279. 55, 75

Bellman, R.E. (1957). *Dynamic Programming*. Princeton University Press, Princeton. 22, 23, 71, 92

Bengio, Y. (2009). Learning Deep Architectures for AI. *Foundations and Trends® in Machine Learning*, **2**, 1–127. 54

Bertsekas, D.P. (1995). *Dynamic Programming and Optimal Control*, vol. 1. Athena Scientific Belmont, MA. 24

Bertsekas, D.P. & Tsitsiklis, J.N. (1996). Neuro-Dynamic Programming (Optimization and Neural Computation Series, 3). *Athena Scientific*, **7**, 15–23. 21

Beyer, H.G. & Schwefel, H.P. (2002). Evolution Strategies–A Comprehensive Introduction. *Natural Computing*, **1**, 3–52. 53

Blau, B., Erensen, J., Huy Nguyen, T. & Verma, S. (2013). Forecast: Video Game Ecosystem, Worldwide, 4Q13 @ONLINE. https://www.gartner.com/doc/2606315/forecast-video-game-ecosystem-worldwide. 3, 89

Bowling, M., Burch, N., Johanson, M. & Tammelin, O. (2015). Heads-up Limit Holdem Poker is Solved. *Science*, **347**, 145–149. 75

Bryzek, J., Roundy, S., Bircumshaw, B., Chung, C., Castellino, K., Stetter, J.R. & Vestel, M. (2006). Marvelous Mems. *Circuits and Devices Magazine, IEEE*, **22**, 8–28. 71

Buro, M. (1997). The Othello Match of the Year: Takeshi Murakami vs. Logistello. *ICCA Journal*, **20**, 189–193. 1

Burrow, P. & Lucas, S.M. (2009). Evolution versus Temporal Difference Learning for Learning to Play Ms. Pac-Man. In *IEEE Symposium on Computational Intelligence and Games (CIG)*, 53–60, IEEE. 53

Cardamone, L., Yannakakis, G.N., Togelius, J. & Lanzi, P.L. (2011). Evolving Interesting Maps for a First Person Shooter. In *Applications of Evolutionary Computation*, 63–72, Springer. 65

Carreras, M., Yuh, J., Batlle, J. & Ridao, P. (2005). A Behavior-Based Scheme Using Reinforcement Learning for Autonomous Underwater Vehicles. *IEEE Journal of Oceanic Engineering*, **30**, 416–427. 44

Chali, Y., Hasan, S. & Imam, K. (2011). A Reinforcement Learning Framework for Answering Complex Questions. In *15th International Conference on Intelligent User Interfaces*, 307–310, ACM. 50

Champandard, A.J. (2003). *AI Game Development: Synthetic Creatures with Learning and Reactive Behaviors*. New Riders. 31

Chan, B., Denzinger, J., Gates, D., Loose, K. & Buchanan, J. (2004). Evolutionary Behavior Testing of Commercial Computer Games. In *Congress on Evolutionary Computation (CEC2004)*, vol. 1, 125–132, IEEE. 2

Ching, W.K. & Ng, M.K. (2006). *Markov Chains*. Springer. 21

Choi, D., Konik, T., Nejati, N., Park, C. & Langley, P. (2007). A Believable Agent for First-Person Perspective Games. In *3rd Artificial Intelligence and Interactive Digital Entertainment International Conference*. 74

Cole, N., Louis, S.J. & Miles, C. (2004). Using a Genetic Algorithm to Tune First-Person Shooter Bots. In *Congress on Evolutionary Computation (CEC2004)*, vol. 1, 139–145, IEEE. 61

Comité, F. & Delepoulle, S. (2005). PIQLE: A Platform for Implementation of Q-learning Experiments. In *NIPS Workshop: Reinforcement Learning Benchmarks and Bake-offs II*. 73

Connell, J.H. & Mahadevan, S. (1993). *Robot Learning*. Kluwer Academic Publishers. 74

Conroy, D., Wyeth, P. & Johnson, D. (2011). Modeling Player-like Behavior for Game AI Design. In *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology*, 9, ACM. 6

Cothran, J. & Champandard, A. (2009). Winning the 2K Bot Prize with a Long-Term Memory Database using SQLite @ONLINE. http://aigamedev.com/open/articles/sqlite-bot/. 63

Cuéllar, M. & Pegalajar, M. (2014). Design and Implementation of Intelligent Systems with LEGO Mindstorms for Undergraduate Computer Engineers. *Computer Applications in Engineering Education*, **22**, 153–166. 71

Danahy, E., Wang, E., Brockman, J., Carberry, A., Shapiro, B. & Rogers, C.B. (2014). LEGO-based Robotics in Higher Education: 15 Years of Student Creativity. *International Journal on Advanced Robot Systems*, **11**, 27. 71

Dennett, D.C. (1991). *Consciousness Explained*. Little, Brown and Co. 63

Dietterich, T.G. (2000). An Overview of MAXQ Hierarchical Reinforcement Learning. In *Abstraction, Reformulation, and Approximation*, 26–44, Springer. 52

Donovan, T. & Garriott, R. (2010). *Replay: The History of Video Games*. Yellow Ant Lewes, UK. 2

Dorigo, M. & Colombetti, M. (1998). *Robot Shaping: An Experiment in Behavior Engineering*. MIT press. 53

Dragos, C. (2013). 47 Programmable Robotic Kits @ONLINE. http://www.intorobotics.com/47-programmable-robotic-kits/. 71

Ertel, W., Schneider, M., Cubek, R. & Tokic, M. (2009). The Teaching-Box: A Universal Robot Learning Framework. In *International Conference on Advanced Robotics (ICAR)*, 1–6, IEEE. 73

Fang, F., Sun, Y. & Leu, J. (2015). Profit Analysis for Video Game Releasing Strategies: Single-player vs. Multiplayer Games. *Journal of Supply Chain and Operations Management*, **13**, 58. 2

Fernández, A.J. & O'Valle, J.L. (2011). Decision Tree-based Algorithms for Implementing Bot AI in UT2004. In *Foundations on Natural and Artificial Computation, Lectures Notes in Computer Science*, 383–392. 64

Ferrucci, D., Brown, E., Chu-Carroll, J., Fan, J., Gondek, D., Kalyanpur, A.A., Lally, A., Murdock, J.W., Nyberg, E., Prager, J. *et al.* (2010). Building Watson: An Overview of the DeepQA Project. *AI magazine*, **31**, 59–79. 18

Fidjeland, A.K. & Shanahan, M.P. (2010). Accelerated Simulation of Spiking Neural Networks using GPUs. In *International Joint Conference on Neural Networks (IJCNN)*, 1–8, IEEE. 66

Floyd, C. (2006). Ping of Death Bot (PodBot) @ONLINE. http://podbotmm.bots-united.com/doc_v3/index.html. 75

Forlizzi, J. & DiSalvo, C. (2006). Service Robots in the Domestic Environment: A Study of the Roomba Vacuum in the Home. In *Proceedings of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction*, 258–265, ACM. 18

Fountas, Z., Gamez, D. & Fidjeland, A.K. (2011). A Neuronal Global Workspace for Human-like Control of a Computer Game Character. In *IEEE Conference on Computational Intelligence and Games (CIG)*, 350–357. 66

Freitas, A.A. (2013). *Data Mining and Knowledge Discovery with Evolutionary Algorithms*. Springer Science & Business Media. 38

Gartner (2013). Gartner Says Worldwide Video Game Market to Total $93 Billion in 2013 @ONLINE. http://www.gartner.com/newsroom/id/2614915. xi, 3, 4

Gelly, S. & Silver, D. (2008). Achieving Master Level Play in 9 x 9 Computer Go. In *23rd AAAI Conference on Artificial Intelligence*, vol. 8, 1537–1540. 1

Gemrot, J., Kadlec, R., Bida, M., Burkert, O., Pibil, R., Havlicek, J., Zemcak, L., Simlovic, J., Vansa, R., Stolba, M., Plch, T. & C., B. (2009). Pogamut 3 Can Assist Developers in Building AI (Not Only) for Their Videogame Agents. In: Agents for Games and Simulations. Lecture Notes in Computer Science, 1–15, Springer. xiv, 78, 80, 91

Genesereth, M., Love, N. & Pell, B. (2005). General Game Playing: Overview of the AAAI Competition. *AI magazine*, **26**, 62. 76

Glavin, F.G. (2009). *A One-Sided Classification Toolkit with Applications in the Analysis of Spectroscopy Data*. Master's thesis, College of Engineering and Informatics, NUI Galway. xiv, 39, 40, 41

Glavin, F.G. & Madden, M.G. (2011). Incorporating Reinforcement Learning into the Creation of Human-Like Autonomous Agents in First Person Shooter Games. In *GAMEON 2011, the 12th annual European Conference on Simulation and AI in Computer Games*, 16–21. 88

Glavin, F.G. & Madden, M.G. (2012). DRE-Bot: A Hierarchichal First Person Shooter Bot using Multiple Sarsa(λ) Reinformcement Learners. In *17th International Conference on Computer Games (CGAMES)*, 148–152. 108

Glavin, F.G. & Madden, M.G. (2015a). Adaptive Shooting for Bots in First Person Shooter Games Using Reinforcement Learning . *IEEE Transactions on Computational Intelligence and AI in Games*, **7**, 180–192. 134

Glavin, F.G. & Madden, M.G. (2015b). Learning to Shoot in First Person Shooter Games by Stabilizing Actions and Clustering Rewards for Reinforcement Learning. In *IEEE Conference on Computational Intelligence and Games)*, 344–351. 159

Goertzel, B. & Pennachin, C. (2007). *Artificial General Intelligence*, vol. 2. Springer. 76

Goodman, D. & Keane, R. (1997). *Man versus Machine: Kasparov versus Deep Blue*. H3 Publications, Cambridge, MA. 18, 75

Gorman, B., Fredriksson, M. & Humphrys, M. (2007). The QASE API: A Comprehensive Platform for Games-Based AI Research and Education. *International Journal of Intelligent Games & Simulation*, **4**. 74

Goyal, A. & Pasquier, P. (2011). Human-like Bots for Unreal Tournament 2004: A Q-learning Approach to Refine UT2004 Strategies. Tech. rep., School of Interactive Arts and Technology, Simon Frase University, Surrey. 60

Graepel, T., Herbrich, R. & Gold, J. (2004). Learning to Fight. In *International Conference on Computer Games: Artificial Intelligence, Design and Education*, 193–200. 50

Hafner, R. & Riedmiller, M. (2005). Case Study: Control of a Real World System in CLSquare. In *Proceedings of the NIPS Workshop on Reinforcement Learning Comparisons, Whistler, British Columbia, Canada*. 73

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P. & Witten, I.H. (2009). The WEKA Data Mining Software: An Update. *Special Interest Group on Knowledge Discovery and Data Mining Explorations Newsletter*, **11**, 10–18. 73

Hefny, A.S., Hatem, A.A., Shalaby, M.M. & Atiya, A.F. (2008). Cerberus: Applying Supervised and Reinforcement Learning Techniques to Capture the Flag Games. In *Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*. 57

Hindriks, K.V., Van Riemsdijk, B., Behrens, T., Korstanje, R., Kraayenbrink, N., Pasman, W. & De Rijk, L. (2011). Unreal Goal Bots. 1–18, Springer. 65

Hingston, P. (2009). A turing test for computer game bots. *Computational Intelligence and AI in Games, IEEE Transactions on*, **1**, 169–186. 56, 77

Hingston, P. (2010). A new design for a turing test for bots. In *IEEE Symposium on Computational Intelligence and Games (CIG)*, 345–350, IEEE. 56, 77

Hirono, D. & Thawonmas, R. (2009). Implementation of a Human-like Bot in a First Person Shooter: Second Place Bot at BotPrize 2008. *Proceedings of Asia Simulation Conference*, 1–5. 62

Holland, J. (1975). Adaption in Natural and Artificial Systems. *Ann Arbor MI: The University of Michigan Press*. 24

Holland, J.H. (1992). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA. 49

Hsu, F.H. (2002). *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press. 1

Hunicke, R. & Chapman, V. (2004). AI for Dynamic Difficulty Adjustment in Games. In *Challenges in Game Artificial Intelligence AAAI Workshop*, vol. 2, 91–96. 176

Kadlec, R., Gemrot, J., Bida, M., Havlicek, J., Pibil, R., Zemcak, L. & Vansa, R. (2014). Pogamut 3 Cookbook @ONLINE. http://pogamut.cuni.cz/pogamut_files/latest/doc/tutorials/. xiv, 85, 86, 91

Kaelbling, L.P., Littman, M.L. & Moore, A.W. (1996). Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research*, **4**, 237–285. 19

Kaminka, G.A., Veloso, M.M., Schaffer, S., Sollitto, C., Adobbati, R., Marshall, A.N., Scholer, A. & Tejada, S. (2002). Gamebots: A Flexible Testbed for Multiagent Team Research. *Communications of the ACM*, **45**, 43–45. 74, 79

Karakovskiy, S. & Togelius, J. (2012). The Mario AI Benchmark and Competitions. *IEEE Transactions on Computational Intelligence and AI in Games*, **4**, 55–67. 77

Kartoun, U., Stern, H. & Edan, Y. (2010). A Human-Robot Collaborative Reinforcement Learning Algorithm. *Journal of Intelligent & Robotic Systems*, **60**, 217–239. 46

Kirby, N. (2011). *Introduction to Game AI*. Cengage Learning. 31, 34

Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I. & Osawa, E. (1997). RoboCup: The Robot World Cup Initiative. In *1st International Conference on Autonomous Agents*, AGENTS '97, 340–347, ACM. 45, 47, 48, 71

Klopf, A.H. (1972). Brain Function and Adaptive Systems: A Heterostatic Theory. Tech. rep., (No. AFCRL-SR-133). Air Force Cambridge Research Labs Hanscom Air Force Base. 24, 29

Knox, W.B. & Stone, P. (2009). Interactively Shaping Agents via Human Reinforcement: The TAMER Framework. In *5th International Conference on Knowledge Capture*. 48

Kober, J. & Peters, J. (2009). Learning Motor Primitives for Robotics. In *IEEE International Conference on Robotics and Automation (ICRA'09)*, 2112–2118, IEEE. 47

Kober, J., Bagnell, J.A. & Peters, J. (2013). Reinforcement Learning in Robotics: A Survey. *The International Journal of Robotics Research*, 0278364913495721. 7, 47, 71

Kolodner, J. (2014). *Case-Based Reasoning*. Morgan Kaufmann. 38

Koster, R. (2013). *Theory of Fun for Game Design*. O'Reilly Media, Inc. 6, 89, 176

Kücklich, J. (2005). Precarious Playbour: Modders and the Digital Games Industry. *Fibreculture*, **5**. 75

Laird, J. & Van Lent, M. (2001). Human-level AI's killer application: Interactive computer games. *AI magazine*, **22**, 15. 2, 74

Laird, J.E. (2001). It Knows what You're Going to Do: Adding Anticipation to a Quakebot. In *Proceedings of the 5th International Conference on Autonomous agents*, 385–392, ACM. 61

Laird, J.E. & Duchi, J.C. (2000). Creating Human-like Synthetic Characters with Multiple Skill Levels: A Case Study using the Soar Quakebot. In *AAAI Fall Symposium on Simulating Human Agents*, 75–79. 60

Laird, J.E., Newell, A. & Rosenbloom, P.S. (1987). Soar: An Architecture for General Intelligence. *Artificial Intelligence*, **33**, 1–64. 48, 61

Larranaga, P. & Lozano, J.A. (2002). *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*, vol. 2. Springer Science & Business Media. 54

Levine, J., Congdon, C.B., Ebner, M., Kendall, G., Lucas, S.M., Miikkulainen, R., Schaul, T., Thompson, T., Lucas, S.M., Mateas, M. *et al.* (2013). General Video Game Playing. *Artificial and Computational Intelligence in Games*, **6**, 77–83. 76

Liandri, A. (2014). Unreal Tournament 2004 @ONLINE. http://liandri.beyondunreal.com/Unreal_Tournament_2004. 81, 82, 83

Liang, Y., Machado, M.C., Talvitie, E. & Bowling, M. (2015). State of the art control of atari games using shallow reinforcement learning. *arXiv preprint arXiv:1512.01563*. 55

Lidén, L. (2003). Artificial Stupidity: The Art of Intentional Mistakes. *AI game programming wisdom*, **2**, 41–48. 2

Lin, S. & Wright, R. (2010). Evolutionary Tile Coding: An Automated State Abstraction Algorithm for Reinforcement Learning. In *AAAI Workshops*, 42–47. 49

Littman, M. & Zinkevich, M. (2006). The 2006 AAAI Computer Poker Competition. *ICGA Journal*, **29**, 166. 76

Loiacono, D., Lanzi, P.L., Togelius, J., Onieva, E., Pelta, D., Butz, M.V., Lönneker, T.D., Cardamone, L., Perez, D., Sáez, Y., Preuss, M. & Quadflieg, J. (2010a). The 2009 Simulated Car Racing Championship. *Computational Intelligence and AI in Games, IEEE Transactions on*, **2**, 131–147. 76

Loiacono, D., Prete, A., Lanzi, P.L. & Cardamone, L. (2010b). Learning to Overtake in Torcs using Simple Reinforcement Learning. In *IEEE Congress on Evolutionary Computation (CEC)*, 1–8, IEEE. 53

Lucas, S.M. (2007). Ms Pac-Man Competition. *ACM Special Interest Group on Genetic and Evolutionary Computation*, **2**, 37–38. 76

Lucas, S.M. & Togelius, J. (2007). Point-to-Point Car Racing: An Initial Study of Evolution versus Temporal Difference Learning. In *IEEE Symposium on Computational Intelligence and Games (CIG)*, 260–267, IEEE. 52

Maass, W. (1997). Networks of Spiking Neurons: The Third Generation of Neural Network Models. *Neural Networks*, **10**, 1659–1671. 66

Madden, M.G. & Howley, T. (2004). Transfer of Experience between Reinforcement Learning Environments with Progressive Difficulty. *Artificial Intelligence Review*, **21**, 375–398. 76

Madden, M.G. & Nolan, P.J. (1995). Application of AI Based Reinforcement Learning to Robot Vehicle Control. In *International Conference on Applications of Artificial Intelligence in Engineering*. 43

Mahadevan, S. & Connell, J. (1992). Automatic Programming of Behavior-based Robots using Reinforcement Learning. *Artificial intelligence*, **55**, 311–365. 43

Martinetz, T. & Schulten, K. (1991). *A "Neural-Gas" Network Learns Topologies*. University of Illinois at Urbana-Champaign. 61, 66

Martinetz, T., Berkovich, S. & Schulten, K. (1993). Neural Gas Network for Vector Quantization and its Application to Time-Series Prediction. *IEEE Transactions on Neural Networks*, **4**, 558–569. 61, 66

McCrae, R.R. & John, O.P. (1998). An Introduction to the Five-Factor Model and its Applications. *Personality: critical concepts in psychology*, **60**, 295. 67

McNaughton, M., Cutumisu, M., Szafron, D., Schaeffer, J., Redford, J. & Parker, D. (2004). ScriptEase: Generative Design Patterns for Computer Role-Playing Games. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, 88–99, IEEE Computer Society. 53

McPartland, M. & Gallagher, M. (2008a). Creating a Multi-Purpose First Person Shooter Bot with Reinforcement Learning. In *IEEE Symposium On Computational Intelligence and Games (CIG)*, 143–150, IEEE. 58

McPartland, M. & Gallagher, M. (2008b). Learning to be a Bot: Reinforcement Learning in Shooter Games. In *Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*. 58

McPartland, M. & Gallagher, M. (2011). Reinforcement Learning in First Person Shooter Games. *IEEE Transactions on Computational Intelligence and AI in Games*, **3**, 43–56. 58, 76, 90, 109

McPartland, M. & Gallagher, M. (2012a). Game Designers Training First Person Shooter Bots. In *AI 2012: Advances in Artificial Intelligence*, 397–408. 59

McPartland, M. & Gallagher, M. (2012b). Interactively Training First Person Shooter Bots. In *IEEE Conference on Computational Intelligence and Games (CIG)*, 132–138. 59

Mehta, N., Ray, S., Tadepalli, P. & Dietterich, T. (2011). Automatic Discovery and Transfer of Task Hierarchies in Reinforcement Learning. *AI Magazine*, **32**, 35–50. 49

Michie, D. (1961). Trial and Error. *Science Survey, Part*, **2**, 129–145. 24

Michie, D. & Chambers, R.A. (1968). BOXES: An Experiment in Adaptive Control. *Machine intelligence*, **2**, 137–152. 49, 73, 74

Miller, W., Glanz, F.H. & Kraft, L. (1990). CMAC: An Associative Neural Network Alternative to Backpropagation. *Proceedings of the IEEE*, **78**, 1561–1567. 49

Millington, I. & Funge, J. (2009). *Artificial Intelligence for Games*. CRC Press. 3, 31, 35, 37

Mitchell, B.L. (2012). *Game Design Essentials*. John Wiley & Sons. 2

Mitchell, M. (1998). *An Introduction to Genetic Algorithms*. MIT press. 38

Mitchell, T.M. (1997). *Machine learning*. McGraw Hill International. 41, 42

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G. *et al.* (2015). Human-level Control through Deep Reinforcement Learning. *Nature*, **518**, 529–533. 54, 55

Moore, A.W. & Hall, T. (1990). *Efficient Memory-Based Learning for Robot Control*. Ph.D. thesis. 45, 48, 49, 73

Moriarty, D.E., Schultz, A.C. & Grefenstette, J.J. (1999). Evolutionary Algorithms for Reinforcement Learning. *J. Artif. Intell. Res. (JAIR)*, **11**, 241–276. 38

Mühlenbein, H. & Paass, G. (1996). From Recombination of Genes to the Estimation of Distributions I. Binary Parameters. In *Parallel Problem Solving from Nature (PPSN) IV*, 178–187, Springer. 54

Mülling, K., Kober, J., Kroemer, O. & Peters, J. (2013). Learning to Select and Generalize Striking Movements in Robot Table Tennis. *The International Journal of Robotics Research*, **32**, 263–279. 47

Nason, S. & Laird, J.E. (2005). Soar-RL: Integrating Reinforcement Learning with Soar. *Cognitive Systems Research*, **6**, 51–59. 48

Nath, V. & Levinson, S.E. (2014). *Autonomous Military Robotics*. Springer. 71

Neumann, G. (2005). *The Reinforcement Learning Toolbox, Reinforcement Learning for Optimal Control Tasks*. 72

Ng, A.Y. (2003). *Shaping and Policy Search in Reinforcement Learning*. Ph.D. thesis, University of California, Berkeley. 44

Ng, A.Y. & Jordan, M.I. (2000). PEGASUS: A Policy Search Method for Large MDPs and POMDPs. In *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, 406–415, Morgan Kaufmann Publishers Inc. 44

Ng, A.Y., Kim, H.J., Jordan, M.I. & Sastry, S. (2004). Autonomous Helicopter Flight via Reinforcement Learning. In S. Thrun, L.K. Saul & B. Schölkopf, eds., *Advances in Neural Information Processing Systems 16 (NIPS 2003)*, MIT Press, Cambridge, MA, USA. 44

Ng, A.Y., Coates, A., Diel, M., Ganapathi, V., Schulte, J., Tse, B., Berger, E. & Liang, E. (2006). Autonomous Inverted Helicopter Flight via Reinforcement Learning. In *Experimental Robotics IX*, 363–372, Springer. 44

Ontanón, S., Synnaeve, G., Uriarte, A., Richoux, F., Churchill, D. & Preuss, M. (2013). A Survey of Real-time Strategy Game AI Research and Competition in Starcraft. *Computational Intelligence and AI in Games, IEEE Transactions on*, **5**, 293–311. 77

Ortony, A., Clore, G.L. & Collins, A. (1990). *The Cognitive Structure of Emotions*. Cambridge University Press. 67

Pan, S.J. & Yang, Q. (2010). A Survey on Transfer Learning. *Knowledge and Data Engineering, IEEE Transactions on*, **22**, 1345–1359. 37

Pao, H.k., Chen, K.t. & Chang, H.c. (2010). Game Bot Detection via Avatar Trajectory Analysis. *IEEE Transactions on Computational Intelligence and AI in Games*, **2**, 162–175. 64

Papahristou, N. & Refanidis, I. (2013). AnyGammon: Playing Backgammon Variants using any Board Size. In *Foundations of Digital Games*, 410–412. 76

Papis, B. & Wawrzynski, P. (2013). dotRL: A Platform for Rapid Reinforcement Learning Methods Development and Validation. In *Computer Science and Information Systems (FedCSIS)*, 129–136, IEEE. 74

Patel, P., Carver, N. & Rahimi, S. (2011). Tuning Computer Gaming Agents using Q-Learning. *Computer Science and Information Systems (FedCSIS)*, 581–588. 59

Pell, B. (1993). *Strategy Generation and Evaluation for Meta-Game Playing*. Ph.D. thesis, University of Cambridge Ph. D. thesis. 52

Pena, L., Ossowski, S. & Pena, J.M. (2009). vBattle: A New Framework to Simulate Medium-Scale Battles in Individual-per-Individual Basis. In *IEEE Symposium on Computational Intelligence and Games (CIG2009)*, 61–68, IEEE. 54

Pena, L., Ossowski, S., Pena, J.M. & Lucas, S.M. (2012). Learning and Evolving Combat Game Controllers. In *IEEE Conference on Computational Intelligence and Games (CIG)*, 195–202, IEEE. 54

Peng, J. & Williams, R.J. (1996). Incremental Multi-step Q-learning. *Machine Learning*, **22**, 283–290. 46, 50

Perez, D., Samothrakis, S., Togelius, J., Schaul, T., Lucas, S., Couëtoux, A., Lee, J., Lim, C.U. & Thompson, T. (2015). The 2014

General Video Game Playing Competition. *IEEE Transactions on Computational Intelligence and AI in Games*. 76

Petrakis, S. & Tefas, A. (2010). Neural Networks Training for Weapon Selection in First-Person Shooter Games. In *Artificial Neural Networks (ICANN)*, 417–422, Springer. 64

Polceanu, M. (2013). MirrorBot: Using human-inspired Mirroring Behavior to Pass a Turing Test. In *IEEE Conference on Computational Intelligence in Games (CIG)*, 1–8. 57, 68, 143

Ponsen, M., Spronck, P. & Tuyls, K. (2006). Hierarchical Reinforcement Learning with Deictic Representation in a Computer Game. In *Proceedings of the 18th Belgium-Netherlands Conference on Artificial Intelligence*, 251–258. 51

Poole, D.L. & Mackworth, A.K. (2010). *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press. 28

Priesterjahn, S., Kramer, O., Weimer, A. & Goebels, A. (2007). Evolution of Human-Competitive Agents in Modern Computer Games. In *IEEE Congress on Evolutionary Computation*, 777–784, IEEE. 62

Puterman, M.L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons. 21

Rao, A.S., Georgeff, M.P. *et al.* (1995). BDI agents: From Theory to Practice. In *First International Conference on Multi-agent Systems (ICMAS)*, 312–319, San Francisco. 65

RoboCup (2014). Official RoboCup Brazil 2014 Website @ONLINE. http://www.robocup2014.org. xiv, 72

Rohlfshagen, P. & Lucas, S.M. (2011). Ms Pac-Man versus Ghost Team CEC 2011 Competition. In *IEEE Congress on Evolutionary Computation (CEC)*, 70–77, IEEE. 76

Rouse III, R. (2010). *Game design: Theory and practice*. Jones & Bartlett Learning. 6

Rummery, G. & Niranjan, M. (1994). On-line Q-learning using Connectionist Systems. Tech. rep., University of Cambridge, Engineering Department. 10, 27, 50, 91

Rummery, G.A. (1995). *Problem Solving with Reinforcement Learning*. Ph.D. thesis, Engineering Department, Cambridge University. 29

Runarsson, T.P. & Lucas, S.M. (2005). Coevolution versus Self-Play Temporal Difference Learning for Acquiring Position Evaluation in Small-Board Go. *Evolutionary Computation, IEEE Transactions on*, **9**, 628–640. 51

Russell, S.J. & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*. Prentice Hall series in Artificial Intelligence, Prentice Hall. 15, 16, 17, 23, 35, 36, 39, 40

Samuel, A.L. (1959). Some Studies in Machine Learning using the Game of Checkers. *IBM Journal of research and development*, **3**, 210–229. 24

Schaeffer, J. (2008). *One Jump Ahead: Computer Perfection at Checkers*. Springer Science & Business Media. 75

Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P. & Sutphen, S. (2007). Checkers is Solved. *Science*, **317**, 1518–1522. 1

Schank, R.C. (1983). *Dynamic Memory: A Theory of Reminding and Learning in Computers and People*. Cambridge University Press. 38

Schank, R.C. (1999). *Dynamic Memory Revisited*. Cambridge University Press. 38

Schlaifer, R. & Raiffa, H. (1961). Applied Statistical Decision Theory. 32

Schrum, J., Karpov, I.V. & Miikkulainen, R. (2011). UT^2: Human-like Behavior via Neuroevolution of Combat Behavior and Replay of Human Traces.

2011 IEEE Conference on Computational Intelligence and Games (CIG), 329–336. 57, 67, 143

Schrum, J., Karpov, I.V. & Miikkulainen, R. (2012). *Human-Like Combat via Multiobjective Neuroevolution*. Springer. 68, 144

Schwab, B. (2009). *AI Game Engine Programming*. Cengage Learning. 4

Shaker, N., Togelius, J., Yannakakis, G.N., Poovanna, L., Ethiraj, V.S., Johansson, S.J., Reynolds, R.G., Heether, L.K., Schumann, T. & Gallagher, M. (2013). The Turing Test Track of the 2012 Mario AI Championship: Entries and Evaluation. In *IEEE Conference on Computational Intelligence in Games (CIG)*, 1–8, IEEE. 77

Sharifi, A. (2010). *Generating Adaptive Companion Behaviors Using Reinforcement Learning in Games*. Master's thesis, University of Alberta. 53

Smith, M., Lee-Urban, S. & Munoz-Avila, H. (2007). RETALIATE: Learning Winning Policies in First-Person Shooter Games. In *17th Innovative Applications of Artificial Intelligence Conference*, vol. 22, 1801–1806. 57, 62

Soni, B. & Hingston, P. (2008). Bots Trained to Play like a Human are More Fun. In *IEEE International Joint Conference on Neural Networks (IJCNN) at the IEEE World Congress on Computational Intelligence*, 363–369, IEEE. 62

Spronck, P.H.M. (2005). *Adaptive Game AI*. UPM, Universitaire Pers Maastricht. 1, 2, 5

Stanley, K.O. & Miikkulainen, R. (2002). Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation*, **10**, 99–127. 66

Stone, P., Kuhlmann, G., Taylor, M.E. & Liu, Y. (2006). Keepaway Soccer: From Machine Learning Testbed to Benchmark. In *RoboCup 2005: Robot Soccer World Cup IX*, 93–105, Springer. 71

Storn, R. & Price, K. (1997). Differential Evolution–A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. *Journal of global optimization*, **11**, 341–359. 54

Sutton, R.S. (1988). Learning to Predict by the Methods of Temporal Differences. *Machine learning*, **3**, 9–44. 18, 24, 44, 53

Sutton, R.S. (1991). Dyna, An Integrated Architecture for Learning, Planning, and Reacting. *ACM Special Interest Group on Artificial Intelligence Bulletin*, **2**, 160–163. 54

Sutton, R.S. (1996). Generalization in Reinforcement Learning: Successful Examples using Sparse Coarse Coding. *Advances in neural information processing systems*, 1038–1044. 27

Sutton, R.S. & Barto, A.G. (1998). *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning, MIT Press. xiv, xv, 10, 19, 20, 21, 22, 24, 26, 28, 29, 30, 45, 46, 53, 58, 76, 109, 110, 165

Tach, D. (2014). Android's Next Release will include GPU-focused Tech for PC Quality Graphics @ONLINE. http://www.polygon.com/2014/6/25/5842246/android-gpu-extension-pack-pc-graphics. 75

Tan, A.h. (2004). FALCON: A Fusion Architecture for Learning, Cognition, and Navigation. *IEEE International Joint Conference on Neural Networks*, 3297–3302. 51, 58, 60

Tanner, B. & White, A. (2009). RL-Glue: Language-Independent Software for Reinforcement-Learning Experiments. *The Journal of Machine Learning Research*, **10**, 2133–2136. 73

Tastan, B. & Sukthankar, G. (2011). Learning Policies for First Person Shooter Games using Inverse Reinforcement Learning. *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 85–90. 59

Taylor, M., Kuhlmann, G. & Stone, P. (2008). Autonomous Transfer for Reinforcement Learning. In *Proceedings of the 7th international Joint Conference on Autonomous Aagents and Multiagent Systems-Volume 1*, 283–290, International Foundation for Autonomous Agents and Multiagent Systems. 45

Taylor, M.E. & Stone, P. (2009). Transfer Learning for Reinforcement Learning Domains: A Survey. *The Journal of Machine Learning Research*, **10**, 1633–1685. 37

Taylor, M.E., Suay, H.B. & Chernova, S. (2011). Using Human Demonstrations to Improve Reinforcement Learning. *Artificial Intelligence*, 66–71. 46

Tenenbaum, J.B., De Silva, V. & Langford, J.C. (2000). A Global Geometric Framework for Nonlinear Dimensionality Reduction. *Science*, **290**, 2319–2323. 64

Tesauro, G. (1992). *Practical Issues in Temporal Difference Learning*. Springer. 17, 76

Tesauro, G. (1995). Temporal Difference Learning and TD-Gammon. *Communications of Association for Computing Machinery*, **38**, 58–68. 1, 17, 76

Thawonmas, R., Murakami, S. & Sato, T. (2011). Believable Judge Bot that Learns to Select Tactics and Judge Opponents. In *IEEE Conference on Computational Intelligence and Games (CIG)*, 345–349. 66

Thrun, S. (2011). Google's Driverless Car. *Ted Talk, Ed*. 18

Thrun, S., Montemerlo, M., Dahlkamp, H., Stavens, D., Aron, A., Diebel, J., Fong, P., Gale, J., Halpenny, M., Hoffmann, G., Lau, K., Oakley, C., Palatucci, M., Pratt, V., Stang, P., Strohband, S., Dupont, C., Jendrossek, L.E., Koelen, C., Markey, C., Rummel, C., van Niekerk, J., Jensen, E., Alessandrini, P., Bradski, G., Davies, B., Ettinger, S., Kaehler, A., Nefian, A. & Mahoney, P. (2006). Stanley: The Robot that Won the DARPA Grand Challenge: Research Articles. *Journal of Intelligent and Robotic Systems*, **23**, 661–692. 18

Thurau, C., Bauckhage, C. & Sagerer, G. (2004). Learning Human-like Movement Behavior for Computer Games. In *From Animals to Animats 8: International Conference on Simulation of Adaptive Behavior*, vol. 8, 315, The MIT Press. 61

Togelius, J. (2007). *Optimization, Imitation and Innovation: Computational Intelligence and Games*. Ph.D. thesis, University of Essex. 25

Togelius, J., Lucas, S., Thang, H.D., Garibaldi, J.M., Nakashima, T., Tan, C.H., Elhanany, I., Berant, S., Hingston, P., MacCallum, R.M. *et al.* (2008). The 2007 IEEE CEC Simulated Car Racing Competition. *Genetic Programming and Evolvable Machines*, **9**, 295–329. 77

Torrey, L., Walker, T., Shavlik, J. & Maclin, R. (2005). Knowledge Transfer via Advice Taking. *3rd international Conference on Knowledge Capture*, 217. 48

Torrey, L., Walker, T., Maclin, R. & Shavlik, J. (2008). Advice Taking and Transfer Learning: Naturally Inspired Extensions to Reinforcement Learning. In *AAAI Fall Symposium on Naturally Inspired AI*. 45

Tozour, P. (2002). The Evolution of Game AI. *AI game programming wisdom*, **1**, 3–15. 2

Turing, A.M. (1950). Computing Machinery and Intelligence. *Mind*, **59**, 433–460. 16, 56

Unreal, W. (2007). Unreal Wiki @ONLINE. http://wiki.beyondunreal.com/Legacy:Bot_Mind. xi, 85, 87

van Lent, M. & Laird, J. (1999). Developing an Artificial Intelligence Engine. 60

Vapnik, V. (1995). *The Nature of Statistical Learning Theory*. Springer-Verlag: New York. 39

Vapnik, V. (2013). *The Nature of Statistical Learning Theory*. Springer Science & Business Media. 39

Von Newmann, J. & Oskar, M. (1944). Theory of Games and Economic Behaviour. 32

Wang, D. & Tan, A. (2015). Creating Autonomous Adaptive Agents in a Real-time First-Person Shooter Computer Game. In *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 7, 123–138. 60, 143

Wang, D., Subagdja, B., Tan, A.H. & Ng, G.W. (2009). Creating Human-Like Autonomous Players in Real-Time First Person Shooter Computer Games. In *21st Annual Conference on Innovative Applications of Artificial Intelligence (IAAI'09)*, 173–178. 58

Watkins, C.J.C.H. (1989). *Learning from Delayed Rewards*. Ph.D. thesis, University of Cambridge, England. 25, 50

Watkins, C.J.C.H. & Dayan, P. (1992). Q-learning. *Machine Learning*, **8**, 279–292. 25, 43, 45, 49, 53, 54, 57, 59, 60, 63

Whiteson, S., Taylor, M.E. & Stone, P. (2007). *Adaptive Tile Coding for Value Function Approximation*. Computer Science Department, University of Texas at Austin. 49

Woodcock, S., Laird, J. & Pottinger, D. (2000). Game AI: The State of the Industry. *Game Developer Magazine*, **8**. 1

Yannakakis, G. & Hallam, J. (2005). A Generic Approach for Obtaining Higher Entertainment in Predator/Prey Computer Games. *Journal of Game Development*, **1**, 23–50. 135

Yannakakis, G.N. (2012). Game AI revisited. In *Proceedings of the 9th Conference on Computing Frontiers*, 285–292, ACM. 2, 5

Yannakakis, G.N. & Hallam, J. (2004). Evolving Opponents for Interesting Interactive Computer Games. *From Animals to Animats*, **8**, 499–508. 135

Yu, X. & Gen, M. (2010). *Introduction to Evolutionary Algorithms*. Springer Science & Business Media. 38

Zhang, W. & Dietterich, T. (1995). A Reinforcement Learning Approach to Job-Shop Scheduling. In *International Joint Conference on Artificial Intelligence*, vol. 14, 1114–1120. 44