THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

Machine Learning for Adaptive Computer Game Opponents

Jonathan Miles

This thesis is submitted in partial fulfillment of the requirements for the Degree of Master of Science at the University of Waikato.

December 2008

Abstract

This thesis investigates the use of machine learning techniques in computer games to create a computer player that adapts to its opponent's game-play. This includes first confirming that machine learning algorithms can be integrated into a modern computer game without have a detrimental effect on game performance, then experimenting with different machine learning techniques to maximize the computer player's performance. Experiments use three machine learning techniques; static prediction models, continuous learning, and reinforcement learning. Static models show the highest initial performance but are not able to beat a simple opponent. Continuous learning is able to improve the performance achieved with static models but the rate of improvement drops over time and the computer player is still unable to beat the opponent. Reinforcement learning methods have the highest rate of improvement but the lowest initial performance. This limits the effectiveness of reinforcement learning because a large number of episodes are required before performance becomes sufficient to match the opponent.

Acknowledgements

Many thanks to my supervisor Dr Tony Smith, whose interest in the application of machine learning to unconventional problems provided an interesting research topic and even afforded me the opportunity to conduct a study that let me play games and reference both *The Matrix* and *Max Payne* (the game) in my thesis (and really, what more could you ask for?). Not to mention the seemingly endless support and guidance offered (and the paper sacrificed for my numerous drafts, when do they stop counting in pages and start counting in trees?).

Thanks also to family and friends who have supported me during my studies.

Table of Contents

List of Figures

List of Tables

1 Introduction

This report investigates the use of machine learning techniques in a computer game (BZFlag) to produce a computer controlled player that adapts to an opponent's style of game-play. Added constraints are that the computer player is limited to the same in-game capabilities, degree of control, and information as the human opponent. The game used is BZFlag which provides competitive one-on-one game-play in a complex 3D environment.

Three machine learning techniques are tested; static prediction models, continuous learning, and reinforcement learning. Static models have the best initial in-game performance but are not able to beat the opponent. Continuous learning shows an improvement in performance over time, but the initial performance is less than that of static models and the rate of improvement drops as the duration of the experiments is extended. Reinforcement learning shows the highest rate of improvement, but has the worst initial performance. This highlights a limitation in reinforcement learning; that an extremely large number of iterations are often required before the performance becomes adequate.

1.1 Context in Games

Sweetser [2002] states:

> "The next industry breakthrough will be with characters that behave realistically and that can learn and adapt, rather than more polygons, higher resolution textures and more frames-per-second." (Cited in [Ponsen & Spronck, 2004])

Computer controlled players are used in all forms of computer games. This can involve anything from simple fixed behaviour to complex sets of rules designed to alter behaviour depending on the state of the game. Machine learning (ML) techniques have been used to create computer opponents and have shown success in simple games, such as board games and card games, but the complexity of

modern 3D games often limits the feasibility of creating computer players that use ML.

Methods used in modern 3D games generally provide an enjoyable experience to the human player. However, these systems are typically complex and are both time consuming and costly to produce. As games continue to increase the scope of their virtual worlds there is an increasing need for computer-controlled characters that can adapt to different situations and even develop unique 'personalities' to provide a more realistic environment for a human player. The use of ML techniques in complex 3D games to create computer-controlled characters is becoming a more popular area of research to solve this problem.

1.2 Context in Machine Learning

Manslow [2002] states:

> "It is anticipated that the widespread adoption of [machine] learning in games will be one of the most important advances ever to be made in game AI. Genuinely adaptive AIs will change the way in which games are played by forcing the player to continually search for new strategies to defeat the AI, rather than perfecting a single technique."

Games often use 'expert systems' to control computer character behaviour. An expert system uses a set of rules written by an 'expert' with domain knowledge. These systems are widely used as a way to provide access to domain knowledge without having an expert present 'in the flesh'. The rules are written by hand which is inherently time consuming and often requires extensive 'debugging' before the system is released for use. The rules are also fixed, making them unable to adapt and they typically repeat any mistakes that they might make.

Machine learning (ML) algorithms can help overcome these limitations. ML algorithms generally try to learn a function that maps input values to an output value. In supervised learning this is done by inferring a function from sets of

known examples. Alternatively, reinforcement learning allows an algorithm to learn a function by ‘trial and error’.

Complex 3D games provide an interesting, and somewhat unexplored, environment for ML research. Games often have strict requirements on CPU time and memory resources; whereas traditional ML research often involves an entire machine being devoted to a single algorithm. This may sound like ML techniques are not useful in computer games, but one must also consider that with these increased physical demands comes a decreased performance demand. That is to say; in a real-time game, decisions are made repeatedly (often second-by-second or faster), and this large number of decisions means a large number of ‘bad’ decisions can go unnoticed by a human user.

1.3 The Problem Domain

The problem domain addressed by this thesis is that of creating a computer opponent in BZFlag able to adapt to a human player’s style of game-play. An added constraint is that the computer opponent is not given an advantage over the human player. That is, the computer opponent must use the same controls and information that a human player would have in the same situation.

The overall aim of this study is to create a computer controlled opponent (using ML techniques) capable of adapting to a human player’s style of game-play, ideally resulting in a computer opponent that can beat the human player.

Creating a computer opponent includes first determining whether ML algorithms can be used in BZFlag without having a detrimental effect on game performance, then experimenting with various ML techniques to determine the in-game performance that can be achieved and adjustments to maximize performance. The ML approaches used in experiments are; static prediction models, continuous learning, and reinforcement learning.

1.4 Thesis Outline

Chapter 2 gives some background on computer players produced by game developers as well as systems created for academic studies. A brief overview of the WEKA machine learning workbench, the source used for many of the ML algorithms, is given, followed by a description of PIQLE and Connectionist which are two reinforcement learning frameworks.

Chapter 3 describes initial attempts to integrate an ML algorithm into BZFlag as a proof of concept. This includes separation of tank controls into steering, shooting, and rotation. The methods used to gather training data are discussed, as well as preliminary results which show that ML algorithms can be used to control shooting without affecting game performance. Modifications to data collection for the speed control are described, as well as the development of the online and offline training approaches which allow a wide range of ML algorithms to be used. Finally some limitations observed during experimentation are discussed.

Chapter 4 describes attempts to create a computer player that uses static prediction models to determine its actions. The limitations mentioned in Chapter 3 are described in more detail and solutions used are presented. Results using a static prediction model to control a single aspect of tank behaviour are presented, showing that the performance achieved is not terrible but is insufficient to beat the opponent. This is followed by results when two static models are used which show the performance using two static models can be better or worse than the performance either model alone. A problem observed when the models are not independent is discussed, as well as the solution used and the results obtained. These independent results show an improvement in performance over the previous results, but the computer player is unable to beat the opponent.

Chapter 5 describes attempts to improve performance achieved in Chapter 4 using continuous learning for one of the prediction models. The method used to select algorithm combinations is discussed, and results when continuous learning is used for short duration tests and longer duration tests are presented. The results from the short duration tests generally show an improvement in performance over time, but this is not maintained in the longer duration tests indicating there is a limit to the performance increased that can be gained by continuous learning.

Chapter 6 describes experiments that use reinforcement learning to control a tank. The initial configuration used with the Connectionist framework is described, and results are presented showing that the computer player does not demonstrate a steady improvement in performance. A problem with the behaviour of the tank using the initial configuration is described, as well as the changes made to overcome the problem and the results obtained which show a decline in the overall performance. The initial configuration used with the PIQLE framework is described along with results obtained which show very poor performance by the computer player. An altered configuration to improve performance is described and the results of a longer test run are presented which show a gradual performance improvement over time but the overall performance is less than that achieved when using Connectionist.

Chapter 7 gives a summary of this thesis and discusses achievements made during the research. These include; showing that machine learning algorithms can be used in a complex, modern game without having a detrimental effect on game performance, the use of only static prediction models to control a tank, and highlighting some limitations in both continuous learning and reinforcement learning when they are applied to games. Possible areas of future work are also discussed showing the research area using ML in computer games is vast and presents many avenues that can be explored.

2 Background

This chapter defines artificial intelligence in the context of computer games and describes some common approaches used by game developers and academic researchers. A general description of BZFlag is given, along with limitations that arise from using it for experimentation. A brief overview of the WEKA machine learning workbench is given, as well as an overview of reinforcement learning and the two reinforcement learning frameworks used in experiments.

2.1 Game AI

For the purposes of this discussion, artificial intelligence in games (game-AI) is defined as: a system to dictate the behaviour of a character inside a computer game, as distinct from characters controlled by a human user.

Game-AI can involve a multitude of different approaches, from rule-based expert systems to reinforcement learning agents. The development of game-AI can be separated into two categories based on the main objective in developing the game-AI. They are referred to here as commercial game-AI, which is done to create an opponent that is enjoyable to play against, and academic game-AI, which is done to create a computer player that plays the game well.

2.1.1 Commercial Game-AI

Commercially developed game-AI is perhaps the most prolific of game-AI systems. The term ‘commercial’ here is used to mean any computer game produced for its appeal to potential users (this includes games not necessarily made for profit, such as free or open source games). The main objective of game-AI in commercial games is to create computer players that a human player finds enjoyable to play against.

How enjoyable a player is to play against cannot be quantified directly and developers have many considerations when designing game-AI, often involving constraints at both upper and lower limits. For instance, human players want a computer opponent that is challenging to beat, but not so difficult that the game becomes frustrating. Human players also want game-AI that behaves 'realistically'; this can be things like 'taking cover' in first person shooter (FPS) games, or cooperating with other computer team-mates to meet objectives rather than behaving as a group of individuals.

Commercially developed game-AI is often complex, but the focus is on creating the appearance of learning (or adaptation) from the player's point of view. The game-AI itself usually behaves deterministically regardless of previous world state. In this sense the game-AI does not 'learn' how to play and typically repeats any mistakes it has made.

Three methods often used in commercial game-AI are; scripting, cheating, and rule-sets.[1] These distinctions are made here to aid discussion of game-AI techniques but modern games often combine these methods together in various ways.

Scripting

Scripting refers to a fixed 'script' that is created by a developer to dictate a non-player character's (NPC's) behaviour, where the 'script' is something set by a developer that does not take into account the current game state. One method of scripting is hand-coded instructions that dictate the exact position and actions of an NPC, another common method is 'way-points' for NPCs to use.

Hand-coding an NPC's actions has numerous limitations, most notably poor scalability. This also does not work well when the human player has a large amount of in-game freedom. For example, an NPC might be talking to the human player but facing another direction. Modern games still use this technique but,

[1] In this report scripted actions are separated from rule-sets that determine behaviour but among game developers 'scripting' is often used to refer to a combination of the two.

because of these limitations, long scripted scenes are often replaced with ‘cut scenes’.[2]

Way-points are often used in ‘death-match[3]’ style shooting games where destinations have multiple paths. Way-points are points placed in the ‘map’ by designers at strategic places, typically intersections of paths and half-way points between those intersections. The way-points can then be viewed as a graph which allows for faster path-finding algorithms in NPC navigation.

Firing points, a slight variation of way-points, are points placed on the map that dictate positions that are good strategically, such as areas with good cover for defence. This helps reduce the complexity of game-AI calculations and was used extensively in the FPS game *Halo* [Butcher & Griesemer, 2002 pg 22].

Cheating

Cheating is where an NPC is given an unfair advantage over the human player. This creates NPCs that are more difficult to beat without requiring complex calculations. Cheating can be narrowed into three subcategories; capability, ‘rubber-band game-AI’ and knowledge.

Capability refers to a difference in abilities between the human player’s character and NPCs. This can be a range of things depending on the game type. In an FPS game, for instance, the ‘harder’ opponents may have weapons that are not available to human players, or they may have more ‘hit points’ so they can survive more damage than human players. Another example is real-time strategy (RTS) games where a ‘harder’ computer opponent is given a better starting state such as more ‘units’ or more resources.

This technique sounds very simplistic but playtests carried out during development of *Halo* found that simply making enemy NPCs ‘tougher’ (i.e. able

[2] A ‘cut scene’ is where the human player’s controls are limited or disabled (‘cut’) and they observe what happens on the screen (like a movie), generally this is done to show scenes that develop characters or advance the plot of the game.
[3] Death-match games (also known as free-for-all or all-against-all games), are games where the objective is simply to kill as many opponents as possible within a given time limit.

to survive more damage) generally causes human players to think the NPCs are more intelligent [Butcher & Griesemer, 2002 pg 16].

'Rubber-band game-AI' (also known as 'catch-up AI') is often seen in racing and sports games, but is also used in other games. Rubber-band game-AI is a technique where the NPC's performance is adjusted to be similar to the human player's performance. For instance, in a racing game where the human player has a large lead (e.g. after the NPC has crashed into a wall), the NPC is able to catch up in a short amount of time, which would require the NPC's car to be going faster than the maximum speed permitted by the game (as though the two characters are connected by a rubber-band). Similarly, rubber-band game-AI can be applied in the opposite scenario, where the human player is doing poorly and the NPC reduces its performance so the human player still has a chance to win.

The idea of rubber-band game-AI is to regulate the game difficulty to match the human player's ability (and is often listed as a positive feature of the game). If done well this can make the game more enjoyable by ensuring the game is never 'too easy' or 'too hard', but often the NPC's 'miraculous' improvement in performance creates a feeling of unfairness and is less enjoyable to play against, or the other extreme where the NPCs 'dumbing down' makes the NPC too easy to beat and reduces the game's challenge.[4] Note that rubber-band game-AI is similar to capability cheats described previously, but rubber-band game-AI only affects the game when there is a large difference in performance between players. Once the NPC and human player are even (or close to it), rubber-band game-AI is suspended and the NPC's performance becomes normal again.

A good example of rubber-band game-AI is present in the well known *Need for Speed* racing game series to ensure the human player is never too far in front of (at least) one NPC. An example of rubber-band game-AI used in a genre other than racing is the third person shooter *Max Payne*, where the difficulty level of the NPCs is determined by the human player's performance. This technique is even

[4] Many gamers, typically of intermediate or advanced level, do not like any form of rubber-band AI in games because it can reduce the skill required to complete the game and is often seen as unrealistic and somewhat patronising.

mentioned in the game's publicity material as a feature of the game (referred to as "auto-adjusting gameplay").[5]

Knowledge-based cheats are often used in real-time strategy games (RTS) to enhance the performance of the game-AI. Knowledge cheats refer to the game-AI having access to more data than its human opponent does. For example, many RTS games use a mechanism called 'fog of war' that obscures large portions of the map for the player (unless one of the player's units is in the area), while the AI knows the exact layout of the map and the locations of the human player's units. Another example is an FPS game where the human player only has knowledge of what they can see (line-of-sight), whereas NPCs in the game know exactly where the human player is at all times.

Rule-Sets

Rule-set systems are similar to scripting discussed previously but allow for more variation based on the current environment. Rule-set systems use a set of rules (IF…THEN) that determine the NPC's actions based on the current environment. The ability to alter behaviour based on world state allows for variations in behaviour that cannot be achieved with scripting.

Rule-set systems are known as 'expert systems' in machine learning, where a human 'expert' uses domain knowledge to define what actions should be taken depending on the world state. Creating expert systems is inherently time consuming and often requires 'debugging' to adjust the rule-set. The rule-set is also highly specific to the situation, meaning new rule-sets must be created for each new game, and often different NPCs each require their own specific rule-set.

An example of rule-sets in game-AI is in *Halo*, which makes use of a rule-set for NPCs to complete their current goal (such as fight, hide, or search), though *Halo* also makes use of many other techniques as well [Butcher & Griesemer, 2002 pg

[5] Can be seen on the *Max Payne* homepage: http://www.rockstargames.com/maxpayne/main.html on the fourth slide ('Make your own levels').

21]. A simpler example is BZFlag which has two built-in NPCs that use only rule-sets to determine actions to take during a game (discussed further in Section 2.2).

2.1.2 Academic Game-AI

Academic game-AI refers to non-human players developed where the performance of the non-human player is the main focus. Unlike commercially developed game-AI (discussed in Section 2.1.1) there is little or no concern for the enjoyment of a human player. The goal is generally that the game-AI be capable of beating any human player (i.e. the world champion). To aid discussion academic game-AI is separated into three categories based on the games used; turn-based competition, real-time competition, and solo.

Turn-Based Competition

Turn-based games are perhaps some of the oldest games known to man and, not surprisingly, are popular as academic studies in machine learning. Turn-based competition games (TBCs) are games where two or more players take turns performing actions that alter the game's state. TBCs include most board games, card games, turn-based strategy games, and even some physically oriented games like Jenga.

TBCs can easily use traditional machine learning because each player must wait for their turn to perform an action, effectively giving a computer opponent ample time to determine its next action. Even if the decision time is limited (as is often the case when a computer plays against a human) the board state will not change until the action is taken, meaning that although the 'thought' time is limited the computer player is not punished for taking the maximum time allowed to determine its next move.

Many TBCs used in machine learning experiments are also 'perfect information' games, where the entire world state is known at all times. For example, in chess both players know the position of all pieces on the board at all times. 'Perfect

information' and relatively low time constraints often allow computer game-AI to use the 'brute-force' technique, where all game states that can be reached from the current state are computed, with the best possible move then being selected.

Deep Blue created by IBM is perhaps the most famous TBC academic game-AI system. *Deep Blue* played chess using the brute-force technique[6] and was able to beat the grand master at the time.

Tesauro's TD-Gammon is another example of an academic game-AI system for a TBC (backgammon). TD-Gammon uses temporal-difference learning to play backgammon. Temporal-difference learning is a form of reinforcement learning where learning is based on observed values that change over time (i.e. from one time-step to the next). TD-Gammon can learn to play backgammon successfully by playing repeated games against itself and, if combined with a shallow look-ahead function, is able to beat the top world players [Tesauro, 2002].

Real-Time Competition

Real-time competition games (RTCs) are games where all players carry out actions that affect the game state simultaneously. These are more complex than TBCs discussed previously and require actions to be chosen rapidly. The real-time nature of these games combined with the large number of variables involved makes the brute-force technique and some other machine learning techniques unfeasible.

Many computer games are RTCs, including most shooting games, RTS, and some racing games (if there is an opponent). Academic studies on RTCs often use RTS games to test machine learning performance. RTS games typically take a (relatively) long time to complete, and poor decisions are not as quickly 'punished' as they might be in other games (such as shooting games). This means that of all real-time games, RTS games are perhaps the least demanding on time.

[6] *Deep Blue* also had several thousand opening moves and end-game moves stored persistently, rather than having to compute them all repeatedly.

The focus of machine learning in RTCs is often limited to a single aspect of game-AI behaviour (such as path-finding or resource management). The games can also be changed to 'solo' games by removing the opponents (the algorithms are then scored by some metric, for example the amount of gold mined by the computer player after 10 minutes of game time).

One example of academic research in RTCs is the study done by Forbus et al. [2002] into the use of spatial reasoning to improve game-AI in RTS games. This aims to improve, among other things, path-finding in RTS games which typically use the A* algorithm and a variation of way-points (described in Section 2.1.1).

Another example is the annual RoboCup competition which aims to create a team of humanoid robots capable of beating a human team in a game of soccer by 2050. RoboCup has many categories based on the hardware used and is more of a robotics challenge, but also includes a simulation category which is based only on software and so falls into the academic game-AI RTC category.

Solo

Solo games are any games where there is no opponent, often using beat-the-clock style games such as racing games. In academic studies solo games are often used because they provide a static environment that is only changed by the agent's actions. This allows for 'incremental-improvement' systems, like reinforcement learning, to be used effectively.

One example of this is the Robot Auto Racing Simulator[7] (RARS) which was designed to provide researchers an easy way to apply machine learning algorithms to a racing game. Many academic studies have been done using RARS as a test environment, Cleland [2006] shows that an agent using reinforcement learning (Q-Learning) can learn to drive around a simple track and is able to beat basic heuristic robots.

[7] RARS has since been superseded by The Open Racing Car Simulator (TORCS) available online at http://torcs.sourceforge.net/

2.2 Objective

Game-AI has been developed for all types of computer games. Commercial game-AI is used in complex 3D games, but is costly and time consuming to create. It is also often highly tailored to one particular game. By contrast, academic game-AI often makes use of versatile machine learning techniques, but is generally applied to less complex games or learning is isolated to a particular task (such as path-finding).

This study aims to determine whether a computer controlled opponent can adapt to a human player's style of game-play in a complex 3D game. Furthermore the computer opponent must use the same level of information and control the human player is given (i.e. not cheating game-AI described in Section 2.1.1). Several machine learning techniques are used; static prediction models, continuous learning, and reinforcement learning.

2.3 BZFlag

BZFlag (short for BattleZone Flag) is a free, open source, and cross-platform multiplayer 3D tank battle game based on a previous game called *BattleZone* and released under the GNU LGPL. Using the terminology from Section 2.1 BZFlag is a commercial real-time competition game that uses rule-sets to control NPCs. The basic game-play of BZFlag is to have two or more tanks whose objective is to shoot each other, but there are several variations of this basic theme including teams, capture-the-flag (CTF), and 'rabbit hunt'.

2.3.1 Client-Server Architecture

BZFlag uses client-server architecture for all games, though both client and server programs can run on the same machine. The client can be considered a 'fat client', whereby a large amount of processing is done in the client program while the server program mainly handles synchronization of the game state between

multiple clients.[8] In this report BZFlag refers to the client program, while BZFS refers to the server program. All discussions of BZFlag and BZFS in this report refer to version 2.0.10.

2.3.2 World Configuration

The world configuration refers to the characteristics of the virtual world created by BZFS. This includes aspects such as size, obstacles, tank abilities, flags, and game-play modes. Due to the large number of parameters that can be set in BZFS only a brief overview is given here.[9]

Figure 2.1 Screenshot of BZFlag

[8] This approach, combined with the open source nature of BZFlag, makes it possible for a player to cheat by recompiling their client with altered code. As a result the 'division of labour' between the client and server may change in future versions.
[9] A thorough list of BZFS configuration settings is available online at http://my.bzflag.org/w/BZFS_Command_Line_Options

Figure 2.1 shows a screen created by BZFlag. The red and purple writing on the top half of the screen is score information. The orange squares show tank aiming information, with the smaller square showing where a shot would go if the tank fired one. The world in Figure 2.1 is randomly generated, with pyramids in blue and boxes in brown. The X-Y plane (ground) is green. Left of centre is the opponent tank in red. Bottom left shows the 'radar' that gives the positions of all other tanks (red dot) as well as all obstacles in the world (blue boxes). To the right of the radar is the message area which provides information such as server messages and chat facilities between players.

World Size

World size is the size of the virtual world created by BZFS. This is measured in 'BZFlag units' which have no real-world counterpart (though it is suggested that if the tank was life-sized one BZFlag unit would be approximately one meter).

The world size is set on the X and Y coordinate planes, the Z-axis size cannot be set by the user. The coordinates on all three axes can be positive or negative, so a world with a size of 200x200 is effectively 400x400 units (on the X-Y plane, green in Figure 2.1) with coordinates ranging from -200,-200 to 200,200.

The terrain is always flat, though obstacles can be placed within the world depending on the configuration. Terrain is uniform in all areas, meaning the characteristics (such as traction) are consistent regardless of position in the world. The world is enclosed on all four sides by 'walls' which cannot be damaged, destroyed, or breached in any way.

Obstacles

BZFlag has several types of obstacles that can be placed within the virtual world. These include boxes, cones, pyramids, arcs, and spheres. All obstacles are solid and cannot be moved or damaged by tanks regardless of the world configuration. Randomly generated worlds in BZFlag use only boxes and pyramids (randomly placed), but boxes are the only type of obstacle used for experiments described in this report.

Tank Abilities

Most tank characteristics are fixed (e.g. maximum speed), but some can be set by BZFS when starting the server, such as jumping and the shot-count. Jumping allows the player to 'jump' the tank upwards (increasing Z-axis values), which is often useful in dodging an opponent's shot. If jumping is turned off the tank stays on the 'ground' at all times (except when blown up by the opponent).

The shot-count is the number of shots each tank has available. This can be thought of as the number of chambers the turret has, where each chamber has to be reloaded after it has been fired. Each shot is reloaded independently of any other shots, with a fixed reloading delay of approximately four seconds.

Experiments discussed in this report have jumping disabled and the number of shots set to one for simplicity of testing.

Flags

Flags can be turned on or off in BZFS. If flags are turned on, BZFS randomly places several flags throughout the world at the start of a game. Both 'good' and 'bad' flags can be used, where a 'good' flag gives the player some kind of enhancement that makes game-play easier, while 'bad' flags do the opposite, making game-play harder for the player (often by manipulating the controls or making it easier for opponent tanks to shoot the player's tank).

There are a large number of flags available and they will not be listed here but, to give an idea of the effects, two ‘good’ flags are ‘Cloaking’ and ‘Shield’, while two ‘bad’ flags are ‘Left Turn Only’ and ‘Reverse Only’.

It should be noted that the flag used in CTF games is a special flag, which is not placed randomly[10] by the server and is neither ‘good’ nor ‘bad’ (as it has no effect on the tank abilities or controls).

Respawning

Respawning is a term in games that refers to a player’s character coming back to life after they have died in the game. Some games use fixed points (‘respawn points’) where the characters are placed after respawning, while other games place the character randomly in the world. BZFlag can use different types of respawning but the one used during experiments in this report is semi-random respawning. This attempts to find a position that is away from the opponent tank by randomly (using pseudorandom number generation) picking places in the world. A time limit of 10 milliseconds is used, after which the tank is placed in the world regardless of opponent position.

2.3.3 Game-Play

Games in BZFlag are generally one of two varieties; death-match and capture-the-flag (CTF). Death-match games are free-for-all games where each tank is trying to shoot every other tank. One variation of this is team death-match, which is the same as standard death-match except each tank is part of a team and is penalized for shooting team-mates. Another variation is ‘rabbit hunt’, where one player is the ‘rabbit’ and is hunted by all other players. When the rabbit is shot, the shooter becomes the rabbit and the process begins again, where the aim is to spend as much time as possible being the rabbit.

[10] The CTF flags are always placed on the team bases, the bases themselves however can be randomly placed on the map.

CTF always uses teams, although a team can be composed of a single player (BZFlag supports up to four teams). Each team has a 'base' on the map that has a flag bearing the team colour. The aim is to retrieve an opponent's flag and return it to the player's base.

2.3.4 Computer Players

BZFlag comes with two built-in computerized players. In this report they are referred to as basic-pilot and autopilot. Both players use rule-sets to determine behaviour, though the rule-sets of the two are different. Basic-pilot is the standard computer opponent during single player games. It has some simple dodging code but overall performs poorly and is easily beaten by a human player.

Autopilot exists to take over a human player's tank when desired (for instance, to answer the phone during a multiplayer game). Autopilot is superior to basic-pilot and easily beats basic-pilot in a one-on-one match. Autopilot uses a fixed rule-set that creates predictable behaviour and can be beaten by an intermediate human player[11] without much difficulty.

2.3.5 Limitations

Games used in academic studies, particularly those that deal with reinforcement learning, often increase the execution speed of the game because of the large amount of game-play required for learning. Unfortunately, the synchronization performed by BZFS makes it difficult to change the operating speed of BZFlag so some experiments in this study have a limited duration.

[11] All observations based on an 'intermediate human player' are from playing the game myself.

2.4 WEKA

WEKA is a machine learning workbench written in Java and released as open-source under the GNU GPL. WEKA is widely used in machine learning research so only a brief description is given here. For a more comprehensive description of WEKA and the algorithms included with it, see Witten & Frank [2005].[12]

WEKA includes various machine learning algorithms, data pre-processing tools, and applications for trialling learning algorithms on user provided datasets. The pre-processing tools include functions such as discretization or removal of attributes from a dataset.

WEKA uses a two step train-test approach. The first step is to 'train' the algorithm on a given dataset. Once the training completes, the learning algorithm is fixed (static) and does not change for the duration of the tests. The second step is the 'test' or 'prediction' phase, where the trained learning algorithm is used on the test dataset. The two datasets (test and train) can be the same dataset, separate datasets, or sub-sections of a larger dataset (such as in cross-validation tests).

2.5 Reinforcement Learning

Parts of this report use reinforcement learning. It is useful therefore to provide a definition of the term as well as a description of the two frameworks used. A detailed explanation of reinforcement learning is beyond the scope of this report, for more information see Sutton & Barto [1998].

[12] Information is also available online at http://www.cs.waikato.ac.nz/ml/weka/

2.5.1 Definition

Reinforcement learning (RL) is a method which matches a situation (world state) to an action in order to maximize some reward function. Furthermore the learner (agent) is not told the right action to take but rather must learn through trial and error which actions maximize the reward function [Sutton and Barto, 1998].

The lack of known 'correct' examples often results in slower learning than in supervised learning but, given sufficient learning time, RL is capable of exploring the entire search space and so is guaranteed to find the optimum solution (if one exists).

One method to achieve this is referred to as state-action pairs whereby all possible combinations of states and actions are kept in memory along with the observed reward for each state-action pair (i.e. the reward the agent received the last time the action was taken from that state). Another method used is similar to state-action pairs but uses a neural network to generalize the learning. This has the advantage of a lower memory requirement, since state-action pairs do not need to be kept in memory.

2.5.2 PIQLE

PIQLE (Platform Implementing Q-Learning) is a Java framework that is designed to separate problems from algorithms, allowing researchers to easily test new algorithms using standard problems or vice-versa.[13]

PIQLE includes implementations of various RL algorithms (generally those described in *Reinforcement Learning, an Introduction* [Sutton and Barto, 1998]), but because of time constraints only the state-action pair algorithm in PIQLE is used in this report.

The state-action pair method stores all combinations of states and actions along with the maximum expected reward for each state-action pair. PIQLE uses

[13] Only a brief overview is given here, for more information see the PIQLE homepage at http://sourceforge.net/projects/piqle

hashing to reduce the memory requirement so that only observed state-action pairs are stored, but the memory requirement can still be quite large.

The state-action pair approach works well on small or simplified problems but does not scale well to more complex areas. The reasons for this are firstly that a large number of states or actions (or both) increases the memory requirement, and secondly all state-action pairs must be visited repeatedly in order for the algorithm to converge.

PIQLE allows the number of actions available to be set on a state-by-state basis. This is particularly beneficial for use in the research described in this report because of the reloading delay (described in Section 2.2.2) which means a tank cannot fire in all world states.

2.5.3 Connectionist

Connectionist is a Java RL framework that uses Connectionist Q-Learning as described by Kuzmin [2002] where a neural network is used to allow generalization of the state-action pairs used for learning. It should be noted that PIQLE also has neural network based algorithms but Connectionist is experimented with first as it is less complex than PIQLE.[14]

Figure 2.2 shows the neural network at the centre with sensors on the left and actions on the right. The neural network has an arbitrary number of inputs (sensors) that represent the current world state, with the reward value received from previous states as an additional input. The output of the neural network corresponds to an action the agent can perform. The actions are fixed at the start of the experiment and it is assumed the actions are always available.

[14]Only a brief overview of Connectionist is given here, for information see the Connectionist homepage at http://elsy.gdan.pl/

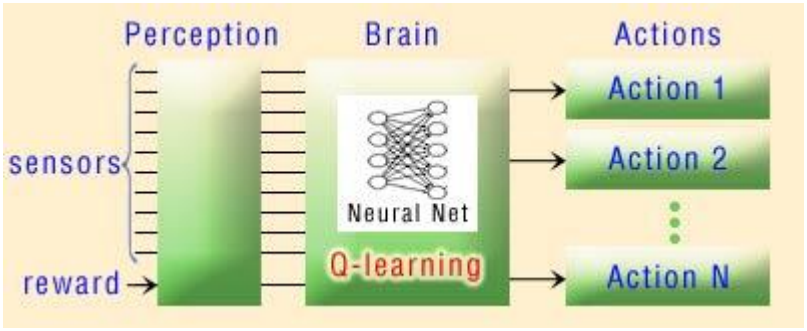

Figure 2.2 Connectionist Brain[15]

The neural network approach has the benefit of a reduced memory requirement over state-action pairs used in PIQLE (see Section 2.4.2). However, generalization by the neural network adds a level of complexity that can make it difficult to adjust for a particular learning problem.

Connectionist also allows for the neural network weights to be saved and restored during experiments. This allows for the neural network to be restored to a known 'good' state if the performance begins to deteriorate due to exploration of the search space.

The neural network approach[16] was used by Tesauro's TD-Gammon backgammon player (described in Section 2.1.2) which is capable of beating the top world players, proving that the neural network approach can be used successfully at least for simple 2D games.

[15] Obtained From http://elsy.gdan.pl/index.php?option=com_content&task=view&id=19&Itemid=32

[16] TD-Gammon uses the neural network approach to reinforcement learning, but does not use the Connectionist framework.

2.6 Chapter Summary

Game-AI has been developed for all kinds of computer games. Often the creation of these systems is both costly and time consuming. Research has been done using machine learning techniques to create computer opponents, but this is generally applied to simpler games. Use of machine learning techniques in a complex computer game raises many interesting questions and is largely an unexplored area of machine learning research.

This study aims to develop game-AI for BZFlag that is able to adapt to the game-play of a human opponent. An additional constraint is the game-AI will have the same in-game capabilities, information, and controls as a human player (i.e. not cheating). BZFlag is used because it provides competitive game-play in a complex 3D environment. The experiments described in this report make use of the WEKA machine learning workbench, and the PIQLE and Connectionist reinforcement learning frameworks.

3 Integration of Machine Learning in BZFlag

This chapter describes initial attempts to use machine learning (ML) algorithms to control a tank in BZFlag. It includes a description of how tank controls are separated, the selection of attributes used to train the algorithms, and development of online and offline approaches to training.

The goal at this stage is to determine whether an ML algorithm can be used to control a tank in BZFlag. This includes determining what attributes are available in BZFlag and confirming that an ML algorithm can be used to control a tank in real-time without having a detrimental effect on the performance of BZFlag. In-game performance of the ML-controlled tank is also observed but is of secondary importance at this point.

Section 3.1 explains how tank controls are separated into three categories; speed, shooting, and rotation. Section 3.2 describes the attempts to use an ML algorithm to control tank shooting and the observed in-game performance. Section 3.3 describes attempts to use an ML algorithm to control speed and the online and offline training approaches developed to accomplish it. Section 3.4 describes some limitations observed during the experimentation described in the previous sections. Section 3.5 is a brief summary of this chapter.

3.1 Separation of Controls

BZFlag allows players to control a tank inside the virtual world created by BZFS (discussed in Section 2.3). For this study controls are separated into three distinct categories; speed, shooting, and rotation. Separation simplifies the complexity of controlling a tank in the 3D environment in the hope that this improves an ML algorithm's ability to learn tank behaviour.

Speed is defined as the tank's velocity along the line it is facing. It is adjusted by setting a floating-point number representing the fraction of the maximum possible speed. This can be set to a maximum of 1.0 and a minimum of -0.5, with 1.0 being full speed ahead and -0.5 being full speed backwards (the tank can only go half as

fast in reverse). Changes to speed happen instantaneously (that is to say, to the user acceleration appears to be instantaneous).

Shooting is the ability to fire a projectile from the tank. Once fired the projectile continues along a straight-line path until it either hits something (an obstacle, tank, or wall) or reaches its maximum range. Tanks do not always have the ability to shoot because of the reloading mechanism (described in Section 2.3.2), unlike the speed and rotation controls which are always available. The shooting control is also different from speed and rotation in that it is a binary variable and thus can simply be toggled (to fire) when required.

Rotation is defined as the tank's ability to change its orientation in the virtual world. As with speed this is adjusted by setting a floating-point number representing the fraction of the maximum possible turn speed. This can be set to a maximum of 1.0 and a minimum of -1.0, where 1.0 is turning as fast as possible to the left and -1.0 is turning as fast as possible to the right. Unlike speed however, turning does not happen instantaneously, it takes time for the tank to rotate (approximately 8 seconds to turn 360 degrees).

3.2 Learning to Shoot

Shooting is selected as the first control to learn with an ML algorithm for two reasons; firstly it is a binary value and so does not require any discretization which simplifies the experiment, and secondly the effect of the algorithm on tank behaviour is the easiest of the three controls to observe during game-play.

3.2.1 Initial World Configuration

The initial world configuration used for testing has a size of 200. As described in Section 2.3.2, this creates a world that is 400x400 units (with coordinates from -200 to +200 on both the X and Y axis). This size was chosen arbitrarily but creates a world small enough that the tanks do not need to spend much time moving to find each other, yet large enough that the tanks cannot shoot each other from one side of the world to the opposite side (so some movement is still required).

To simplify the test, the only obstacle in the world is a single 10x10x10 square block at the centre of the world (coordinates 0,0). The standard re-spawning algorithm discussed in Section 2.3.2 is used to re-spawn dead tanks.

3.2.2 Gathering Training Data

Training data is gathered from a one-on-one match between autopilot and basic-pilot. The decisions made by autopilot are output at each time-step of the game. Table 3.1 shows the data recorded.

Name	Description
MyPosition (X,Y,Z)	The position of the autopilot's tank on the axis
MyVelocity (X,Y,Z)	The velocity of the autopilot's tank along the axis
EnemyPosition (X,Y,Z)	The position of the opponent's tank on the axis
EnemyVelocity (X,Y,Z)	The velocity of the opponent's tank along the axis
EnemyDistance	The straight-line distance from the centre of the autopilot's tank to the centre of the opponent's tank.
AngleDifference	The difference between the current rotation of the autopilot's tank, and the rotation which would point the autopilot's tank straight at the opponent's tank. (How far the autopilot tank must rotate to be facing the opponent tank)
isObscured	Boolean value – True if the opponent's tank is obscured behind an obstacle in the world, false otherwise.
Fire (Class value)	Boolean value – True when a shot is fired, false otherwise.

Table 3.1 Dataset Used to Train ML Algorithms to Control Shooting

The position attributes (*MyPosition* and *EnemyPosition*) are the absolute position of one of the tanks (autopilot or opponent) on the world axes. Using the world configuration described in Section 3.2.1, the X and Y coordinates have a range of -200 to +200, the Z coordinates have a minimum value of 0 and a maximum of approximately 30 (this is how high the tank goes when it explodes after being killed).

Velocity attributes (*MyVelocity* and *EnemyVelocity*) are the velocities of one of the tanks along the world axes. Using the world configuration described in Section 3.2.1 all velocity attributes (X,Y,Z) have a range of -25 to +25. As with world size this does not have a direct real-world unit of measure, but if one BZFlag unit is equal to one meter then a tank's maximum velocity is close to 25km/h.

EnemyDistance is the straight-line distance to the opponent's tank in BZFlag units. Using the world configuration described in Section 3.2.1 this has a minimum value of 0 and a maximum of approximately 565 (that is, if the two tanks are in opposite corners of the world, the hypotenuse of the triangle formed by two sides of the world is approximately 565 BZFlag units long).

AngleDifference is the difference in angle between the autopilot tank's current orientation, and the orientation that would have it facing straight at the opponent's tank. This is measured in radians and so has a minimum value of 0 and a maximum of approximately 3.14 (just under 180 degrees).

isObscured is a Boolean value that is true if the opponent's tank is obscured by an obstacle. In other words, it is true if there is no obstacle between autopilot and the opponent's tank (following a straight-line path).

The *EnemyDistance*, *AngleDifference*, and *isObscured* attributes are all generated by functions that are built-in to the autopilot's logic. All values except *isObscured* and *Fire* are numeric (floating-point) values. The two Boolean values, *isObscured* and *Fire*, are stored as nominal attributes with values "True" and "False".

3.2.3 Rule-Based ML Algorithm (PART)

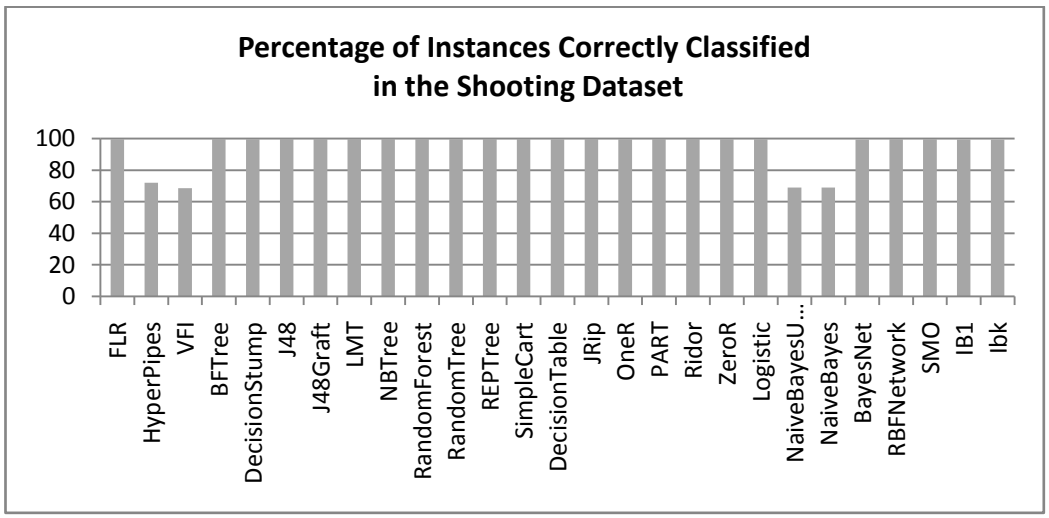

Figure 3.1 Percentage of Instances Correctly Classified in the Shooting Dataset

A large number of the machine learning algorithms in WEKA are trialled on the initial dataset. The percentages of correctly classified instances for each ML algorithm using 10-fold cross-validation on the dataset are shown in Figure 3.1. The dataset has over 150,000 instances so 10-fold cross-validation has over 15,000 instances in each fold.

The dataset has a large difference between the numbers of positive and negative examples because shots are fired relatively rarely (by several orders of magnitude) compared to other actions taken at each time-step. Figure 3.1 shows a majority of the learning algorithms perform extremely well on the dataset; this is most likely due to prediction of the majority class which is close to 99.5% of the dataset. ZeroR, for instance, which predicts the majority class for all instances scores close to 100%.

To correct this problem, the number of negative examples is reduced using random re-sampling so the numbers of positive and negative examples are approximately equal. The size of the balanced dataset is approximately 1500 instances which allows for 10-fold cross-validation to be used with around 150 instances per fold.

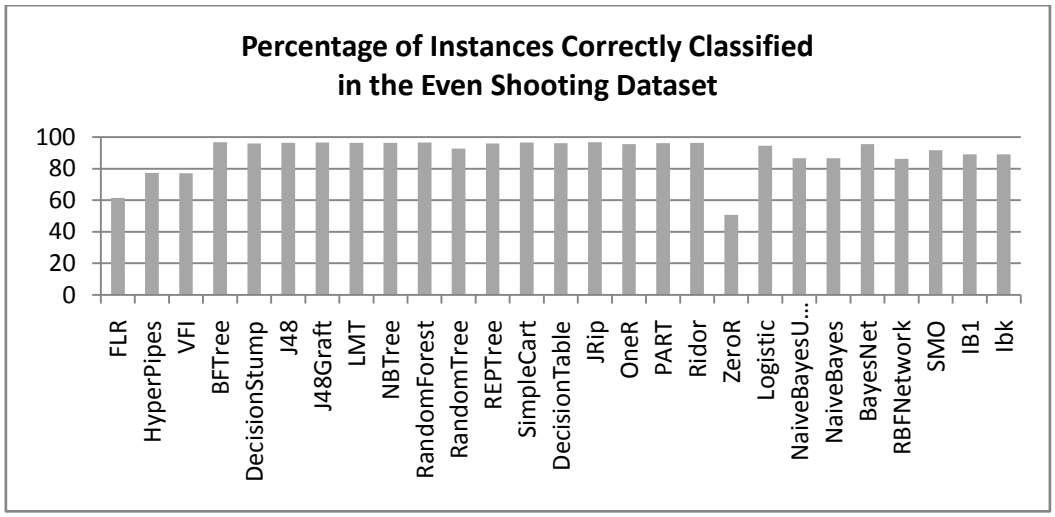

Figure 3.2 Percentage of Instances Correctly Classified in the Even Shooting Dataset

The percentages of correctly classified instances (using 10-fold cross-validation) for each ML algorithm on this reduced dataset are shown in Figure 3.2. This shows that ZeroR now scores close to 50% as expected but, despite balancing the dataset with equal numbers of positive and negative instances, a majority of the ML algorithms still score over 90%. Good performance from so many algorithms may indicate that the problem of shooting control is a relatively simple one; alternatively it could be an indication of over-fitting the data.

To check for over-fitting, one of the algorithms is used to decide the autopilot's actions during game-play. Observation of the tank's behaviour then reveals whether the algorithm has generalized enough to learn an adequate prediction model. The PART algorithm is selected because it is a rule-based learner and so can be easily integrated into the autopilot as a series of IF...THEN rules. It also has the smallest and least complex set of rules of all the rule-based learners.

Autopilot using rules generated by PART to control shooting (referred to as autopilot-PART), is capable of matching basic-pilot in a one-on-one match using the world configuration described in Section 3.2.1. Observation of the game-play however shows that shooting behaviour is inconsistent; situations considered similar by a human player can result in different shooting behaviour by autopilot-PART.

3.2.4 Human-Computer Shared Control

Autopilot-PART is modified to allow a human user to control the speed and rotation, while only the rules generated by the PART algorithm control shooting. This is done to better understand behaviour from the rules generated by the PART algorithm. This combination of human and autopilot-PART is adequate to beat basic-pilot due to the human player's ability to compensate for autopilot-PART's sometimes poor shooting performance.

Observation of the in-game behaviour of autopilot-PART reveals that shooting behaviour differs depending on where the tank is in the world. Inspection of the rule-set generated by the PART algorithm, some of which is shown in Figure 3.3, shows this is due to the algorithm using the position attributes as independent values for prediction rather than using the relationship between them (position values are bold in Figure 3.3). However, because autopilot-PART is capable of equalling the performance of basic-pilot, investigation is turned to the more complex area of speed control. That is to say, the objective of determining whether ML can control shooting is achieved. Analysis and improvement of shooting control is deferred to Chapter 4.

```
if(isObscured == false &&
AngleDifference > 0.0398 &&
EnemyDistance <= 211.52 &&
EnemyPositionZ <= 0.000313 &&
EnemyVelocityX <= -12.9125)

return false;
```

```
if(isObscured == false &&
EnemyDistance <= 35.6192 &&
EnemyPositionX > -29.2714 &&
EnemyPositionX <= 93.8645 &&
MyVelocityY <= 8.73996 &&
EnemyDistance > 13.8054)

return false;
```

Figure 3.3 Portion of PART Rule-Set

3.3 Learning to Control Speed

Autopilot-PART (described in the previous section) is able to equal the performance of basic-pilot. This shows that an ML algorithm can be used to control tank behaviour and investigation is now shifted to tank speed. Speed is potentially more complex than shooting because it is a floating point numeric value rather than a binary value.

Tests use the same world configuration described in Section 3.2.1. All data used in this section is gathered from a one-on-one match between the standard autopilot and a robot player using the same logic as the autopilot (referred to as robot-pilot) rather than basic-pilot used in Section 3.2.2. This is done because the standard autopilot can easily beat basic-pilot so matching the ML-controlled autopilot against a standard autopilot should give a better indication of how well the algorithm has learned tank behaviour.

3.3.1 Speed Dataset

When an algorithm makes use of the position attributes as individual values it results in inconsistent tank behaviour (discussed in Section 3.2.4). In order to prevent algorithms from using position attributes individually (rather than the relation between them) the *MyPosition* and *EnemyPosition* attributes are removed from the dataset and are replaced with *RelativePosition* attributes.

Name	Description
MyVelocity (X,Y,Z)	The velocity of the autopilot's tank along the X axis.
EnemyVelocity (X,Y,Z)	The velocity of the opponent's tank along the X axis.
RelativePosition (X,Y,Z)	The position of the opponent's tank on the X axis, relative to the autopilot's tank.
EnemyDistance	The straight-line distance from the centre of the autopilot's tank to the centre of the opponent's tank.
AngleDifference	The difference between the current rotation of the autopilot's tank, and the rotation which would point the autopilot's tank straight at the opponent's tank (How far the autopilot tank must rotate to be facing the opponent tank).
isObscured	Boolean value – True if the opponent's tank is obscured behind an obstacle in the world, false otherwise.
Angle	The current orientation of the autopilot's tank.
Speed (Class value)	The speed of the autopilot's tank.

Table 3.2 Dataset Used to Train ML algorithms to Control Speed

Table 3.2 shows the set of attributes used to train the ML algorithms to control tank speed. The *RelativePosition* values are the result of the opponent tank's position being subtracted from the autopilot tank's position on the respective axis. Using the world configuration described in Section 3.2.1, the X and Y values have a range of possible values from [-400 to +400], while the Z value has a range of approximately [-30 to +30]. As in Section 3.2.2 all values except *isObscured* are floating point numeric values.

The current orientation of autopilot's tank is also added to the dataset as this may affect the chosen speed (it was decided the angle information would most likely not be useful in the shooting control so it is left out of the shooting dataset shown in Table 3.1). It should also be noted that the current tank speed is not present in the dataset shown in Table 3.2; this is to prevent the possible problem of algorithms simply returning a value based on the tank's current speed, since a change in speed happens less often than maintaining the current speed.

3.3.2 ML Algorithm Results

Very few classification algorithms are able to predict numeric values, so in order to use a majority of the ML algorithms in WEKA the dataset must have a nominal class attribute. To achieve this, the class value (speed) is discretized. As described in Section 3.1, changes to tank speed happen almost instantaneously, this makes the descretization easier since almost all values in the dataset are either 1.0 (full speed ahead), 0.0 (stopped), or -0.5 (full speed backwards).

The value is discretized using the discretize filter available in WEKA with equal-width binning. Three bins are created to correspond with the observed speed values mentioned above, the bins generated by the filter are; [-∞ to -0.315415], [-0.315415 to +0.342293], [+0.342293 to +∞]. Given that the speed dataset is somewhat already separated into three classes these bins are deemed sufficient and no further testing with filter settings is carried out.

The same classification algorithms from Section 3.2.3 are tested on the new dataset with the discretized speed attribute. The dataset has over 1200 instances which provides over 120 instances per fold using 10-fold cross-validation.

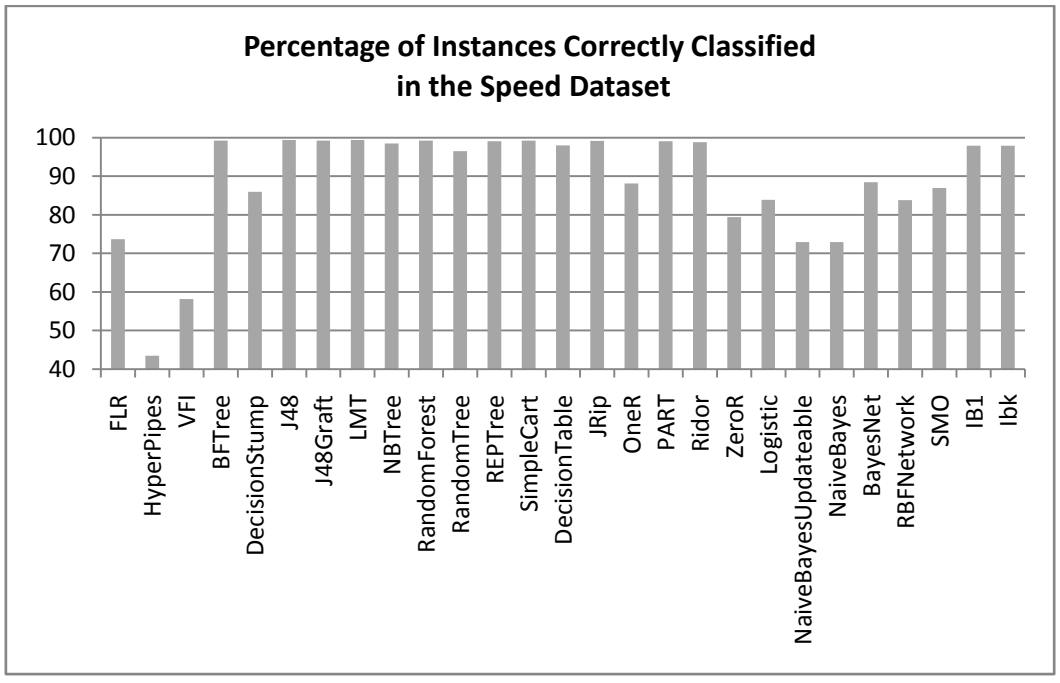

Figure 3.4 Percentage of Instances Correctly Classified in the Speed Dataset

Figure 3.4 shows that, as with shooting in Section 3.2.3, many algorithms perform very well on this dataset. JRip is the best performing algorithm by a small margin, and so is selected as the algorithm to test in BZFlag.

3.3.3 Online Training

The increased complexity of the rule-set created by JRip (mentioned in the previous section) over that created by PART (Section 3.2.3) makes it impractical to integrate the rule-set into the autopilot code. To allow JRip (or any other algorithm available in WEKA) to be used easily, BZFlag is modified to allow data used for classification to be sent over a TCP connection to a server program. After sending each instance the autopilot listens for the predicted value to be sent back over the same TCP connection.

WEKA does have some remote server capabilities but it was decided it was easier to write a new server program specifically for using machine learning algorithms from WEKA to control a tank in BZFlag. The server program is written in Java so it can make use of any of the algorithms in WEKA without modification. To differentiate this program from the BZFlag server, it is referred to as WEKA-Server.

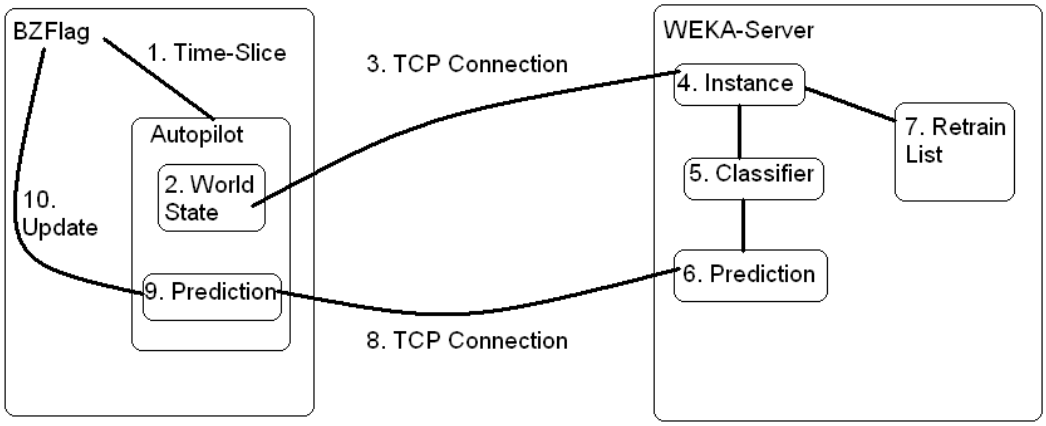

Figure 3.5 Communication Between BZFlag and WEKA-Sever

Figure 3.5 depicts the actions that take place for each decision made by the ML algorithm for the autopilot. This figure shows there are at least ten steps (some are condensed for simplicity) for every time-slice of game-play in BZFlag. The operation of each step is described in the following list:

1. BZFlag starts a new time-slice and calls on autopilot to update its speed and rotation, and to fire a shot if applicable.
2. Autopilot compiles a list of attributes describing the current world state of BZFlag, these values correspond to the attributes in the dataset (such as the one shown in Table 3.2). This also includes the class value (the value autopilot would choose).
3. The attributes (world state) are sent to WEKA-Server via the TCP connection.
4. WEKA-Server receives the attributes and creates a new Instance class (used by WEKA code).
5. The instance is passed to the ML algorithm.
6. The algorithm predicts the classification of the instance.
7. The new instance is added to the list of instances used to retrain the algorithm (when required).
8. The predicted value from Step 6 is sent via TCP to BZFlag.
9. Autopilot receives the predicted value.
10. Autopilot updates its speed\rotation\shooting accordingly.

Both sending and receiving in BZFlag uses blocking sockets so any delays caused by the operations do not disadvantage the autopilot (as the whole game blocks until the operations are complete).

It should also be noted that whilst the TCP connection allows WEKA-Server to be run on any remote computer (accessible via a network), the increased delay caused by a network connection causes a lot of ‘jitter’ when watching the game so for all experiments WEKA-Server is run on the same machine as BZFlag (using the loopback interface for TCP connections).

The list mentioned in Step 7 (referred to hereafter as the retrain-list) contains all instances received by WEKA-Server since the start of the current experiment, with the class values being the values the standard autopilot would have used. The instances in the retrain-list are used to periodically retrain the algorithm when a sufficient number have been received.

Two values affect the retraining of the algorithm, the first is the maximum size of the retrain-list and the second is the retrain-threshold. In some experiments the total size of the retrain-list is limited to a fixed number, typically because the algorithm takes too long to retrain as the retrain-list becomes larger. Once the retrain-list reaches its maximum size each new instance replaces the oldest instance in the retrain-list.

The retrain-threshold specifies how large the retrain-list must be before the algorithm is retrained. After each retrain the retrain-threshold is increased until it reaches the maximum size of the retrain-list (if one is set). The initial retrain-threshold is set at 8 instances and is multiplied by 1.25 after each retrain. So retraining is done when the retrain-list has a size of 8, then 10, then 12, then 15, then 18, and so on. These values are somewhat arbitrary but were chosen empirically as this allows for the algorithm to be retrained often while the dataset is small and prone to poor representation of classes, but less often as the list becomes larger and the training time increases. Until the initial retrain-threshold is reached a default value is returned for all instances, this has a minimal effect on performance if the initial retrain-threshold is low because very little time elapses before the initial retrain-threshold is reached.

3.3.4 JRip

Section 3.3.2 shows JRip has (marginally) the best performance of the algorithms trialled on the speed dataset, so it is used to test the online training configuration described in the previous section. The experiments are run using the world configuration described in Section 3.3.

Observation of autopilot using JRip to control speed shows that if the retrain-list is limited to a maximum size of 500 then autopilot performs on-par with robot-pilot, but if the retrain-list is limited to a maximum size of 1000 then autopilot out-performs robot-pilot.

It was later discovered that robot-pilot had a flaw related to target selection in its implementation of the autopilot logic that resulted in it performing worse than the

standard autopilot. This does not affect the described performance of the online training configuration but means the in-game performance of the JRip algorithm observed at this stage is not reliable.

Some shortcomings of the online training approach became apparent during testing, namely a high demand on the CPU when retraining and, because blocking sockets are used for all communications between BZFlag and WEKA-Server, the game 'hangs' during periods of retraining.

3.3.5 Offline Training

To overcome the limitations of the online training configuration, a new configuration with classification separated from algorithm retraining is used. Operation is the same as that described in Section 3.3.3 except that instead of WEKA-Server retraining the algorithm the instances are sent via another TCP connection to another server program (referred to as ClassifierBuilder). ClassifierBuilder then retrains the algorithm when required and sends the newly trained algorithm back to WEKA-Server via the TCP connection. WEKA-Server uses the most recently received algorithm to classify incoming instances from BZFlag. Figure 3.6 shows the general operation of the offline training configuration.

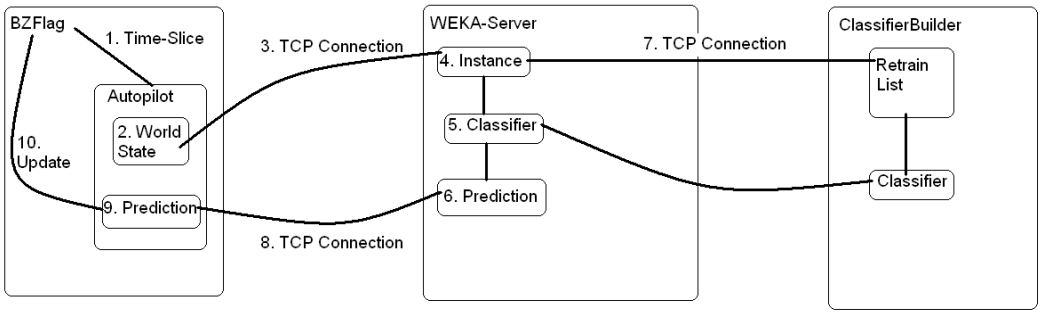

Figure 3.6 Communication Between BZFlag, WEKA-Server, and ClassifierBuilder

WEKA-Server communicates with ClassifierBuilder using a separate thread so BZFlag is unaffected by the delays of retraining the algorithm. This also allows for ClassifierBuilder to run on a separate machine without the delay of network communications affecting BZFlag and eases the load on the local CPU (though this is less of a concern if it is running on a multiple core machine).

The retraining delay no longer affects BZFlag so there is less need to put a limit on the size of the retrain-list. This means that retrain-threshold is the main value that determines when the algorithm is retrained. However, because BZFlag no longer halts during algorithm retraining, new instances are constantly being received by WEKA-Server. This becomes a major problem when retrain times become larger and cause WEKA-Server to hold an increasing number of instances, often causing WEKA-Server to run out of memory. For instance, say the retrain-list has 500 instances, the retrain-threshold is at 550, and WEKA-Server has 300 instances waiting to be sent to ClassifierBuilder. Using the retrain-threshold to determine algorithm retraining, 50 instances are sent from WEKA-Server to ClassifierBuilder, which starts retraining the algorithm. Meanwhile WEKA-Server still has 250 instances waiting to be sent to ClassifierBuilder, and will continue to receive more instances from BZFlag while ClassifierBuilder is retraining the algorithm.

To overcome this problem the retrain-threshold is ignored if a large number of instances (over 100) are still waiting to be sent from WEKA-Server to ClassifierBuilder, in which case all the instances are sent to ClassifierBuilder and then the algorithm is retrained. This does not always come into effect, as some algorithms retrain quickly even with large numbers of instances, but it is a necessity with algorithms that take a long time to retrain.

As with the online training approach described in Section 3.3.3, a default value is returned for all instances until WEKA-Server receives the first trained algorithm. The retrain-threshold used for the experiments conducted with the offline training approach is the same as that described in Section 3.3.2 (initial value of 8, multiplied by 1.25 after each retrain).

The offline training approach introduces several elements that can affect in-game algorithm performance. Firstly there is an increased delay due to network latency

when ClassifierBuilder is running on a remote machine. Secondly BZFlag does not halt during algorithm retraining and WEKA-Server has to use the 'old' algorithm until the retraining is complete. To determine how much effect this has on in-game performance a similar test to that described in the Section 3.3.4 is carried out with JRip controlling tank speed.

Observation of autopilot using JRip to control speed shows the offline training approach has no noticeable effect on in-game algorithm performance, with autopilot still able to out-perform robot-pilot. Most likely any detrimental effect caused by network and retraining delays is offset by the fact that the retrain-list does not need to have its size limited like it does in the online approach (described in Section 3.3.3).

As with the results described in Section 3.3.4, robot-pilot was later found to have a flaw in its implementation of the autopilot logic that caused it to perform worse than the standard autopilot. This does not affect the comparisons between the offline and online training approaches or the observed performance of the offline approach but means the observed in-game performance of the JRip algorithm at this stage is not reliable.

3.4 Limitations

The approach used during the initial experimentation discussed in this chapter suffers from some limitations. For example, all evaluations of in-game performance are done by human observation of game-play rather than an objective test. It is also possible for the game to enter a state of 'stale-mate', where the tanks become stuck in logic loops and are unable to kill each other, generally they are either in a state of 'indecision' on opposite sides of the obstacle in the world and are unable to 'choose' which side to go around, or they enter a state of 'Neo-Smith circling' when they get stuck side by side, continually turning the same direction, and are unable to shoot each other (since they turn at the same speed), similar to two dogs chasing each other's tail. Lastly there is the ability for 'spawn-camping' by a surviving tank that can potentially give an unfair advantage to one

side. These limitations and solutions used are discussed in more detail in the next chapter.

3.5 Chapter Summary

Chapter 3 describes several approaches to integrate an ML algorithm into BZFlag in order to control a tank. It is possible to use a hard-coded static model learned from standard autopilot behaviour to control tank shooting, but hard-coding a model becomes unfeasible with more complicated learning algorithms.

Online training can be used to avoid having to hard-code a prediction model but is detrimental to the performance of BZFlag due to the delays caused by algorithm retraining. Offline training overcomes the delay problem and allows an algorithm to be used to control a tank in BZFlag without compromising the performance of either the ML algorithm or BZFlag.

Having shown that machine learning algorithms can be used in a modern computer game without having a detrimental effect on game performance; investigation now turns to the performance of the algorithms and whether static prediction models are sufficient to beat an opponent. This is the subject of the next chapter.

4 Static Prediction Models

This chapter describes experiments to determine whether a tank controlled only by static prediction models can out-perform robot-pilot or a human player. This includes first determining how well a static prediction model can control a single aspect of tank behaviour when facing robot-pilot, then experimenting with combinations of static models to control all aspects of tank behaviour.

Section 4.1 discusses changes to the world configuration because of the limitations described in Section 3.4. Section 4.2 describes results obtained using a single static model to control one aspect of tank behaviour. Sections 4.3 expands on this, describing the results obtained when two static models are used to control different aspects of tank behaviour simultaneously. Section 4.4 discusses a problem observed with the predictions of one machine learning (ML) algorithm affecting the performance of another algorithm and the results of a proposed solution. Section 4.5 gives a brief summary of this chapter.

4.1 Solutions to Previous Limitations

The approach used in the experiments described in Chapter 3 suffers from some limitations (previously mentioned in Section 3.4), namely the lack of an objective performance comparison, the ability of the game to enter a ‘stale-mate’ state, and the ability of living tanks to gain an advantage over re-spawning tanks (‘spawn-camping’). This section describes these limitations in more detail and solutions implemented to overcome them.

4.1.1 Scoring

In the previous chapter all evaluations of in-game performance are done by human observation of game-play. This is inherently subjective and makes it difficult to compare the results of different experiments.

To overcome this issue an objective method for measuring performance is required. Some metrics such as shot accuracy were considered, but on its own this is not sufficient to score tank behaviour. For instance, player A may have 100% accuracy while player B has 50% accuracy. One is inclined to think that player A has better performance, but suppose player B has a rate-of-fire three times greater than player A. Now, despite having a lower accuracy, player B will have more hits.

A more reliable test method is to have the two players continue playing until the total kill-count reaches 100. That is, the sum of the number of deaths (hits) each tank has received reaches 100. The score of each player then gives a percentage of how likely they are to win against the opponent given any random starting state. In some cases the total kill-count may actually be 101, this is because the scores are updated during re-spawning and it is possible for both tanks to kill each other before the update, this is described in more detail in Section 4.1.3.

As described in Section 2.3.1, BZFlag uses client-server architecture. This makes it inherently multi-threaded and non-deterministic. This non-determinism means there is always a random element during in-game tests, which affects the reliability of scoring. To observe the variance of scores, ten test runs are conducted playing autopilot against robot-pilot (described in Section 3.3).

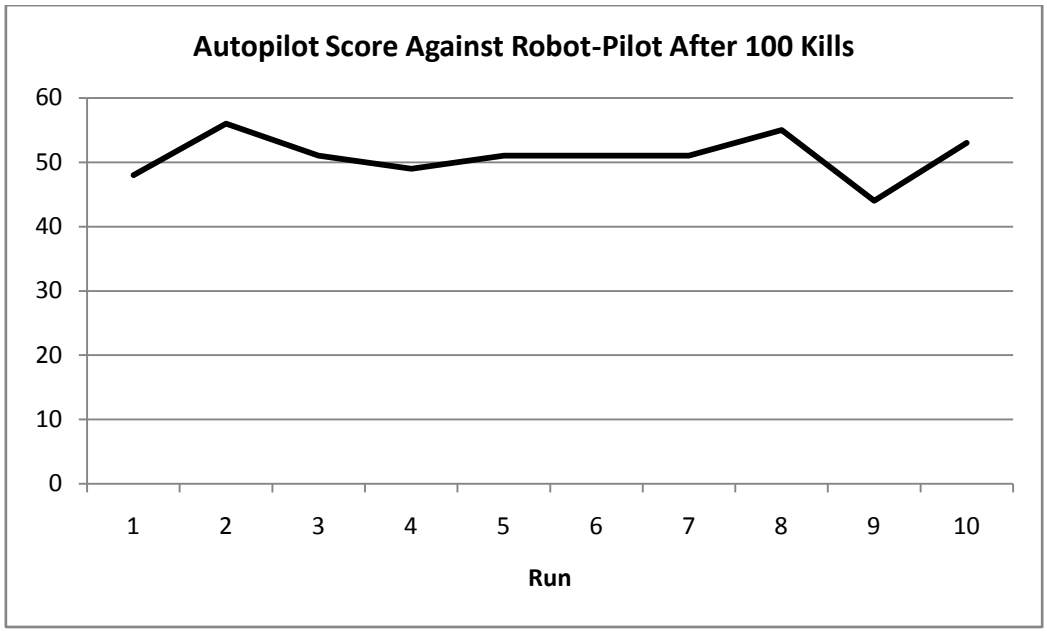

Figure 4.1 Autopilot Score Against Robot-Pilot After 100 Kills

The results in Figure 4.1 show that with two evenly matched opponents the score stays close to 50 as expected, though the score is not constant. The average score over all ten runs is 50.9, and the standard deviation is 3.446415. Assuming a normal distribution, 95% of runs will fall within approximately 7 points of the true average. Due to time constraints most of the experiments conducted use a single run (to 100 kills) to determine performance and these values are used as the base for performance comparisons.

When discussing tank performance in this report the terms ‘score’ and ‘points’ refer to the number of hits achieved by a single tank, whereas the terms ‘kills’, ‘total kills’, and ‘total kill count’ refer to the combined number of hits by both tanks. Descriptions of graph axes also use the term ‘N-Kill block’ where N is the number of kills represented by each data point, for instance each ‘run’ in Figure 4.1 is one 100-kill block.

4.1.2 Stale-Mate Conditions

The possibility for players to enter a stale-mate condition is a problem observed during the experiments described in Chapter 3. Stale-mate conditions are where, due to loops in the logic, the tanks become 'stuck' and are unable to kill each other. This is a quirk of BZFlag's implementation and is not present in games with a human opponent, but due to the large amount of game-play required it is impractical to use a human opponent for the tests. The 'stale-mate' condition typically occurs in two varieties, referred to as indecision and Neo-Smith circling.

Indecision occurs when tanks are on opposite sides of the single box in the world (using the world configuration described in Section 3.2.1). If the tanks are close to the centre of the box, the distance to the opponent is approximately equidistant around either side of the box. This often causes the tanks to quickly switch between going left and going right, while the opponent tank does the same. The switching results in both tanks staying near the centre of the block and the loop begins again.

Neo-Smith circling occurs in open ground areas of the world where the two tanks get very close to each other. This can happen because the autopilot continues moving towards the enemy even when it cannot fire (due to the reload delay described in Section 2.3.2). If the tanks manage to get side-by-side without shooting each other, they then start turning towards each other, often firing 'over the shoulder' of the opponent tank, but because they both turn at the same speed this behaviour continues infinitely.[17]

To prevent situations that lead to a state of indecision, the block (described in Section 3.2.1) is removed, resulting in a world that is a 400x400 plane. It is impossible to fully prevent Neo-Smith circling without rewriting the autopilot logic, so BZFlag is altered to kill off and re-spawn both tanks if no tank has died in the last 30 seconds. This time limit can also affect the game when tanks are not in a Neo-Smith circling state, but prevents the experiments from taking an excessively long time to complete. If the tanks are killed off because of the time

[17] The name Neo-Smith circling comes from a fight scene in the 1999 move *The Matrix*, which has a similar situation between two evenly matched opponents firing guns over each other's shoulder.

limit, scoring (described in Section 4.1.1) is not affected, the scores only count actual ‘hits’ by the tanks against each other.

4.1.3 ‘Spawn-Camping’

‘Spawn Camping’ is a term used by the online gaming community to mean ‘camping’ or waiting near a re-spawn point (discussed in Section 2.3.2) in order to kill an opponent as soon as they re-spawn. This gives the ‘camper’ an unfair advantage since the re-spawned player does not have time to react before being killed. Modern games often use random re-spawn points to avoid this issue, and indeed BZFlag also makes use of semi-random re-spawn points (described in Section 2.3.2). So, while ‘camping’ in the strict sense is not possible, autopilot generally continues moving at full speed while the opponent is dead and through ‘luck’ can come upon the recently re-spawned tank and gain an advantage.

BZFlag is altered to re-spawn both tanks after each kill to prevent this issue from unfairly affecting the results. This means the score keeping, as described in Section 4.1.1, is effectively the score of 100 random, isolated, one-on-one matches. It should also be noted that re-spawning after a kill does not happen instantly. This is because it is possible for a tank’s projectile to hit the opponent’s tank even after the tank that fired the shot has died, meaning the tanks can kill each other before either one has re-spawned.

4.2 Single Static Model

The large number of machine learning algorithms available in WEKA, and the fact that ultimately three tank controls have to be learned, make it unfeasible to try all possible combinations of algorithms and controls. To maximize the chance of finding a successful combination of algorithms and controls, the algorithms are first trialled on the relevant dataset using 10-fold cross-validation. Algorithms that perform well are then tested for in-game performance when controlling a single aspect of tank behaviour (using the scoring mechanism described in Section

4.1.1). The performance of each individual algorithm is then used to determine which algorithms are tested in combination.

4.2.1 Gathering Data

The datasets used in this chapter are obtained from a one-on-one match between a human player and robot-pilot. Data is gathered from both players to maximize an ML algorithm's ability to infer a generalised model of behaviour. The choices made by both players are recorded and combined randomly to form the datasets used. Both players' decisions are recorded in the hope that the algorithms will be able to create prediction models with sufficient generalisation to out-perform robot-pilot.

Name	Description	Shoot	Speed	Rot.
MyVelocity (X,Y,Z)	Velocity of player's tank along the axis.	**X**	**X**	**X**
EnemyVelocity (X,Y,Z)	Velocity of opponent's tank along the axis.	**X**	**X**	**X**
RelativePosition (X,Y,Z)	Position of opponent's tank on the axis, relative to player's tank.	**X**	**X**	**X**
EnemyDistance	Straight-line distance from the centre of player's tank to the centre of opponent's tank.	**X**	**X**	**X**
AngleDifference	Difference between the current rotation of the player's tank and the rotation which would point player's tank straight at opponent's tank (How far player's tank must rotate to be facing opponent's tank).	**X**	**X**	**X**
isObscured	Boolean value – True if opponent's tank is obscured behind an obstacle, false otherwise.	**X**	**X**	**X**
ShotRelative (X,Y,Z)	Position of opponent's projectile on the axis, relative to the player's tank (Missing if opponent does not have an active shot).	**X**	**X**	**X**
ShotVelocity (X,Y,Z)	Velocity of opponent's projectile along the axis (Missing if opponent does not have an active shot).	**X**	**X**	**X**
ShotDistance	Straight-line distance to opponent's projectile (Missing if opponent does not have an active shot).	**X**	**X**	**X**
MyRotation	Orientation of player's tank.	**X**	**X**	**X**
MySpeed	Current speed of player's tank.	**X**		**X**
FiringStatus	Integer value, tank can only fire when value is 1 (meaning 'ready').	**X**		
Fire (Class)	Boolean value – True when a shot is fired, false otherwise.	**X**		
Speed (Class)	Desired speed of player's tank.		**X**	
NewRotation (Class)	Desired rotation of player's tank.			**X**

Table 4.1 Datasets Used for Static Model Training

Shooting

Table 4.1 shows the data used in the shooting dataset (indicated with an 'X' in the shoot column). The first six rows in Table 4.1 are the same as those used in the speed dataset described in Section 3.3.1. To give a more accurate representation of world state and allow the algorithms to learn dodging behaviour, attributes are also included that relate to the opponent's projectile if one has been fired (*ShotRelative, ShotVelocity, ShotDistance*).

The *ShotRelative* attributes are similar to the *RelativePosition* attributes and have the same range of possible values. Each *ShotRelative* attribute is the position of the enemy's projectile subtracted from the player tank's position.

The *ShotVelocity* attributes are the velocities of the enemy's projectile along the respective axis, with the same range of values as the *MyVelocity* and *EnemyVelocity* attributes. *ShotDistance* uses the same function as *EnemyDistance* to compute the straight-line distance to the opponent's projectile. All three shot attributes (*ShotRelative, ShotVelocity, ShotDistance)* can have missing values (represented with a '?' in WEKA) if the enemy does not have an active shot (i.e. there is currently no projectile fired by the enemy in the world).

MyRotation is the current orientation of the player's tank in the world. This value is in radians and has a minimum value of 0 (0 degrees) and a maximum value of approximately 6.28 (just under 360 degrees).

MySpeed is the current speed of the player's tank. This is measured as a fraction of the tank's maximum speed with a range of [-1.0 to +1.0] (full speed reverse to full speed forwards). It should be noted that the tank can only go half as fast in reverse, so a *MySpeed* value of -1.0 does not mean the tank is travelling as fast as a value of +1.0, but rather means the tank is travelling at the maximum speed possible in that direction (forward or reverse).

FiringStatus is an integer value used in BZFlag to indicate the current tank status. In the world configuration used this has three possible values; 0, 1, and 2. A value of 0 means the tank is dead and is waiting to be re-spawned. A value of 1

indicates the tank is ready to fire. A value of 2 indicates that the tank is reloading and is therefore unable to fire.

The tank firing control is effectively binary (described in Section 3.2) so it does not require discretization and is simply converted to a nominal attribute for use with the ML algorithms present in WEKA.

Speed

Table 4.1 shows the data used in the speed dataset (indicated with an 'X' in the speed column). The *FiringStatus* attribute is not included as it has little to do with speed behaviour. *MySpeed* is also excluded because, as noted in Section 3.3.1, changes in speed happen less often than maintaining the current speed and so the algorithm might return a value based on the current speed for all instances, resulting in the tank never moving (because the initial speed is zero).

Changes to the speed of the tank happen almost instantly (as described in Section 3.3.2) so class values in the speed dataset are easily separated into three categories (forward, stopped, backwards). The discretization used is the same as that described in Section 3.3.2. It is possible for a human user to control the speed with less coarse stepping (by using a mouse or joystick), but this is sufficient to provide the algorithm the same degree of control that a human user has when using a keyboard.

Rotation

Table 4.1 shows data used in the rotation dataset (indicated with an 'X' in the rot. column). One notable difference is inclusion of the *MyRotation* attribute. This is included because, unlike speed, rotation is not set with an absolute value. In BZFlag setting the tank's desired speed to 1.0 (full speed ahead) results in the tank accelerating and instantly achieving top speed, however the tank's orientation is altered by setting the desired turn speed, not the desired orientation, so it is less likely to have a detrimental effect on performance.

NewRotation is discretized into three groups; -1.0 (turn left) 0.0 (go straight) and 1.0 (turn right). Rotation values are more evenly spread than speed values so the discretization bins are set by hand to [-∞ to -0.01], [-0.01 to 0.01], and [0.01 to +∞]. As with speed, a human user can use more precise inputs but this gives the algorithm a similar degree of tank control that a human user has with a keyboard.

Five datasets are created for testing. Rotation and speed both have two datasets created; one with all instances produced during the game (referred to as the 'full' dataset), the other created using random re-sampling on the full dataset to reduce the number of majority-class instances, resulting in a dataset with approximately even numbers of all class values (referred to as the 'even' dataset). The shooting dataset has a considerably higher number of negative instances (several orders of magnitude more, as noted in Section 3.2.3) so only the even dataset is used for shooting experiments.

4.2.2 Algorithm Selection

The list of learning algorithms trialled on the datasets described in Section 4.2.1 is less extensive than that used in Chapter 3. This is because of the time constraints and the number of datasets, which are larger than those in Chapter 3. With this in mind, ML algorithms that train relatively quickly are favoured over those that take longer to train; however, for completeness, some common algorithms, such as SMO, are also included.

The algorithms used for in-game testing are determined by checking the percentage of correctly classified instances on the dataset using 10-fold cross-validation. Both the full and even datasets described in the previous section are used for trailing ML algorithms to control speed and rotation. Note that ZeroR, which predicts the majority class for all instances, is not considered for in-game use but is included in the test-set to determine a lower-bound for each dataset (that is to say, if an algorithm scores worse than ZeroR it should not be considered at all).

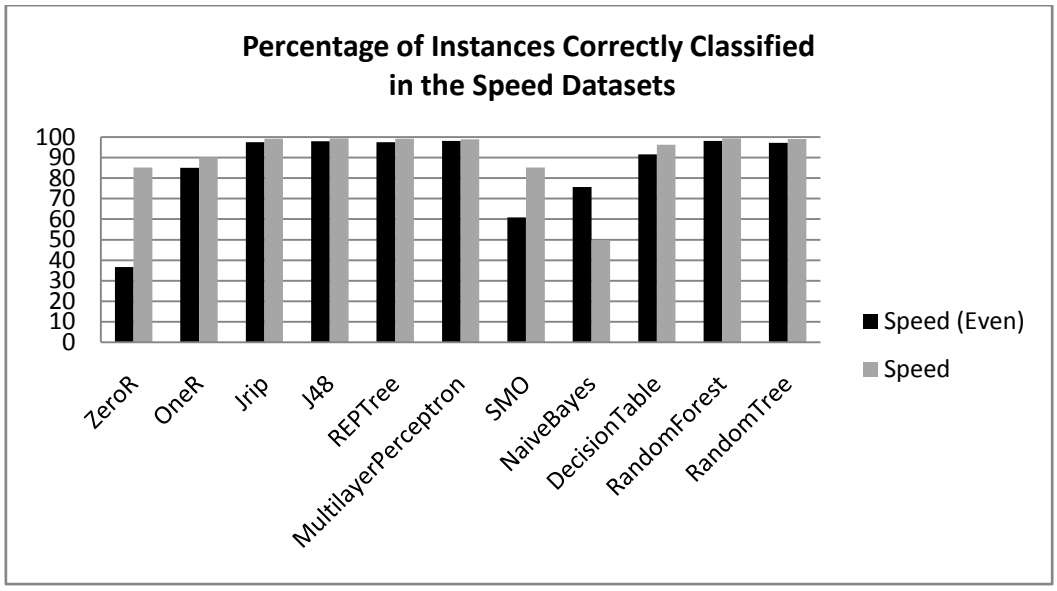

Figure 4.2 Percentage of Instances Correctly Classified in the Speed Datasets

Figure 4.2 shows the percentages of instances correctly classified in the speed datasets (using 10-fold cross-validation). These results demonstrate that using the even dataset for training generally results in algorithms that perform worse (if only slightly) than those trained on the full dataset. Based on these results only the algorithms trained on the full dataset are considered. The five algorithms with the best performance are selected for in-game testing. These are; RandomForest, J48, REPTree, JRip, and RandomTree (best to worst). The in-game performance of these algorithms is discussed in Section 4.2.3.

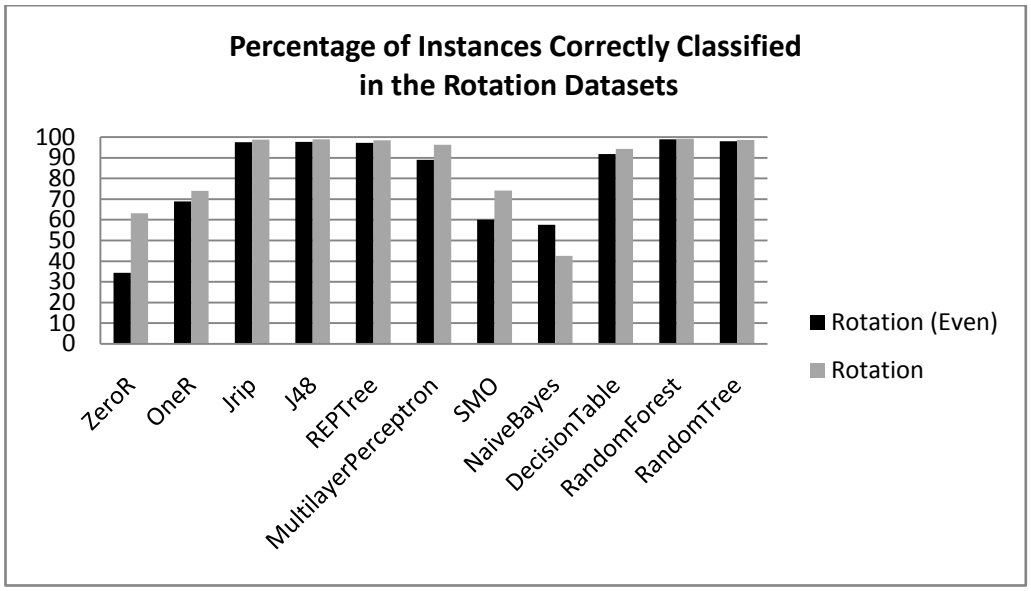

Figure 4.3 Percentage of Instances Correctly Classified in the Rotation Datasets

Figure 4.3 shows that, as with the results in Figure 4.2, generally algorithms trained on the even dataset perform no better than algorithms trained on the full dataset. Because of this, only the algorithms trained on the full dataset are considered for in-game testing. The five algorithms with the best performance are selected for testing in-game performance. These are; RandomForest, J48, JRip, RandomTree, and REPTree (from best to worst). In-game performance of these algorithms is discussed in Section 4.2.3.

It is interesting to note the top five algorithms are the same for both rotation and speed datasets. This may be due to similarity of the data within the dataset, as they both have a large number of attributes in common (shown in Table 4.1).

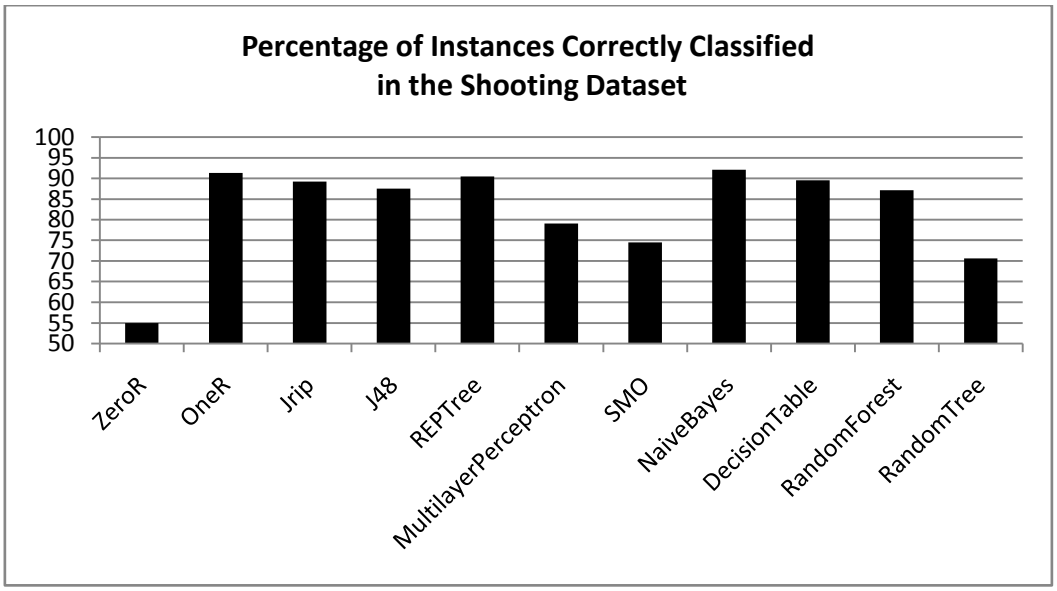

Figure 4.4 Percentage of Instances Correctly Classified in the Shooting Dataset

The results shown in Figure 4.4 use only the even shooting dataset. This is because the full shooting dataset has a large number of negative instances (by several orders of magnitude), so an algorithm that simply predicts the majority class can score over 95% (as noted in Section 4.2.1).

Based on the results shown in Figure 4.4, the five algorithms with the best performance are selected for testing in-game performance. These are; OneR, NaiveBayes, REPTree, DecisionTable, and JRip (best to worst). In-game performance of these algorithms is discussed in Section 4.2.3.

4.2.3 In-Game Performance

Using the results described in Section 4.2.2, the algorithms with the best performance are used to control a single aspect of tank behaviour in BZFlag, while the autopilot controls the remaining two aspects. All results displayed in this section include the performance of autopilot (labelled ‘Autopilot’) in order to accurately compare the performance achieved by the algorithms. All results are also ordered (left to right, best to worst) based on the algorithms’ performance discussed in the previous section.

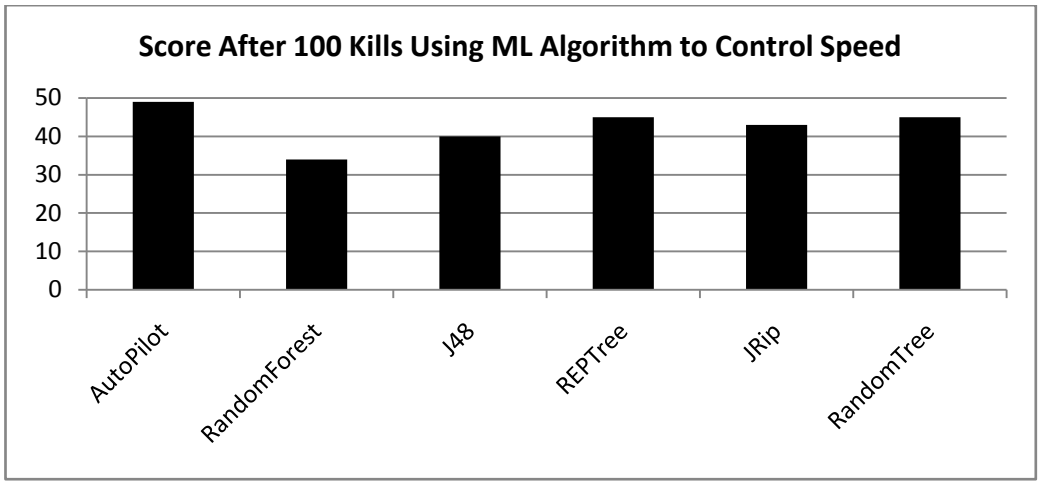

Figure 4.5 Score After 100 Kills Using ML Algorithm to Control Speed

Figure 4.5 shows the performance of algorithms in cross-validation tests is not necessarily matched when the algorithm is used for tank control. For instance RandomForest, the algorithm with the best performance when tested on the dataset in the previous section, is the worst performing algorithm in Figure 4.5. This may be an indication the algorithm has over-fit the training dataset, resulting in a prediction model that does not generalise enough to perform well during in-game tests.

The in-game performance of the algorithms does not equal that of autopilot, but the scores are not abysmal. Except for RandomForest all algorithms tested score 40 points or higher against robot-pilot, with the best performing algorithms of REPTree and RandomTree both scoring 45 points.

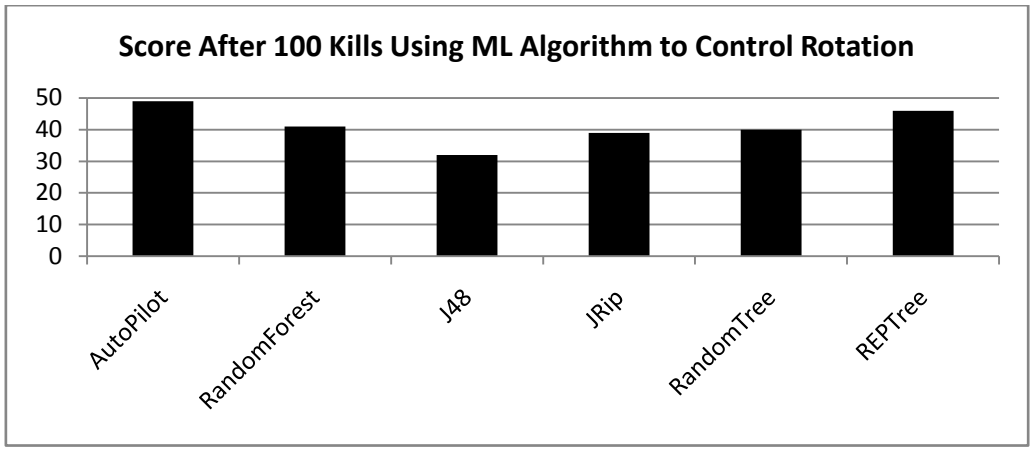

Figure 4.6 Score After 100 Kills Using ML Algorithm to Control Rotation

Figure 4.6 shows that, as with the speed results displayed in Figure 4.5, none of the algorithms are capable of matching autopilot's performance, however three of the five algorithms still manage to score 40 points or higher against robot-pilot. It is interesting to note that REPTree is the best performing algorithm in Figure 4.6 and is best-equal in Figure 4.5, possibly indicating the two controls are similar problems.

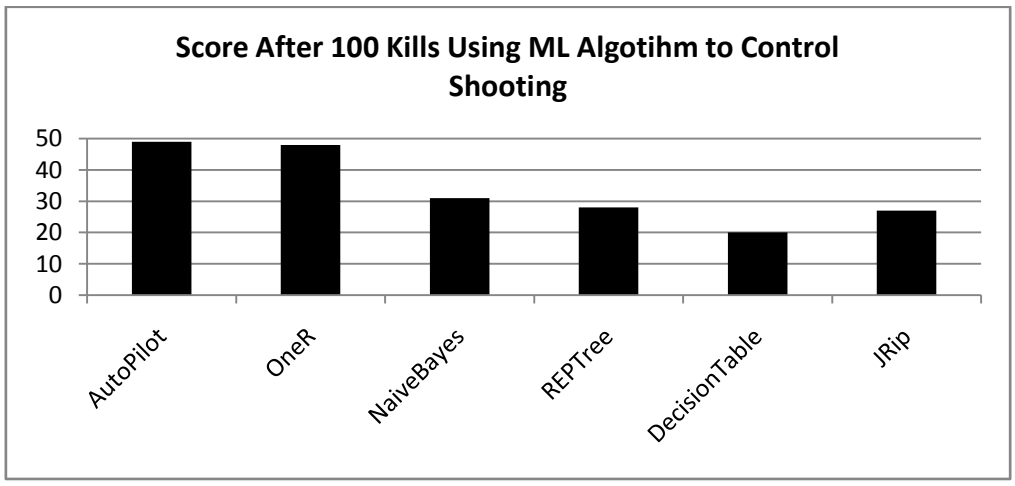

Figure 4.7 Score After 100 Kills Using ML Algorithm to Control Shooting

Figure 4.7 shows that OneR clearly performs better than the other algorithms in the test-set, coming very close to the performance of autopilot with a score of 48 against the robot-pilot, versus autopilot's score of 49.

The remaining algorithms performed poorly in comparison; NaiveBayes is the only other algorithm to score more than 30 points against robot-pilot. This is a particularly interesting result because shooting is a binary control and is expected to be the easiest of the three controls to learn. The high performance of OneR, a classification algorithm that creates a prediction based on a single attribute, indicates the other algorithms' poor performances may be caused by the simplicity of the problem. That is to say; OneR creates a simple prediction model but the other algorithms develop overly complex models for the relatively simple problem, resulting in poor performance.

4.2.4 Observations

ML algorithms with the best performance on the dataset do not necessarily have the best in-game performance. This may be due to over-fitting the training data to some degree rather than problems with the testing itself. All the algorithms that perform well in-game also perform well in the cross-validation tests, indicating it is an adequate method of reducing the number of algorithms selected for in-game performance testing.

4.3 Dual Static Models

None of the algorithms tested in the previous section are capable of out-performing robot-pilot in a one-on-one competition when they control one aspect of tank control. However, it is unclear what performance can be achieved when two or all three control aspects are handled by ML algorithms.

This section describes experiments to determine the best performing two-algorithm combinations for tank control, while autopilot controls the third aspect of tank behaviour.

4.3.1 Algorithm Selection

Due to the time consuming nature of in-game tests and the large number of possible combinations of controls and ML algorithms, some algorithms discussed in the previous section are removed from the test-set.

Figure 4.5 shows the difference between the best and worst performing algorithms used to control tank speed is 11 points. Similarly Figure 4.6 shows the difference between the best and worst performing algorithms used to control rotation is 14 points. Figure 4.7 however, shows the difference between the best and worst performing algorithms used to control shooting is 28 points. Because of this large difference in shooting ML algorithm performance, only the best three are kept in the test-set. These are OneR, NaiveBayes, and REPTree (best to worst in-game performance).

4.3.2 In-Game Performance

Using two ML algorithms to control different aspects of tank behaviour is more complex than the single algorithm tests described in Section 4.2.3 because there are a large number of possible combinations. Tests described in this chapter

include the following control pairs; speed+shooting, rotation+shooting, and speed+rotation.

Speed and Shooting

Figure 4.8 shows configurations using OneR to control shooting consistently out-perform tanks using NaiveBayes and REPTree to control shooting. This is expected given the vast difference in the in-game performance of the algorithms (shown in Figure 4.7).

Interestingly NaiveBayes, which performs better than REPTree when autopilot controls tank speed, performs worse than both OneR and REPTree in all combinations tested. This seems to indicate NaiveBayes over-fits to situations that autopilot creates and is unable to generalize when presented with the different situations encountered when an ML algorithm controls tank speed.

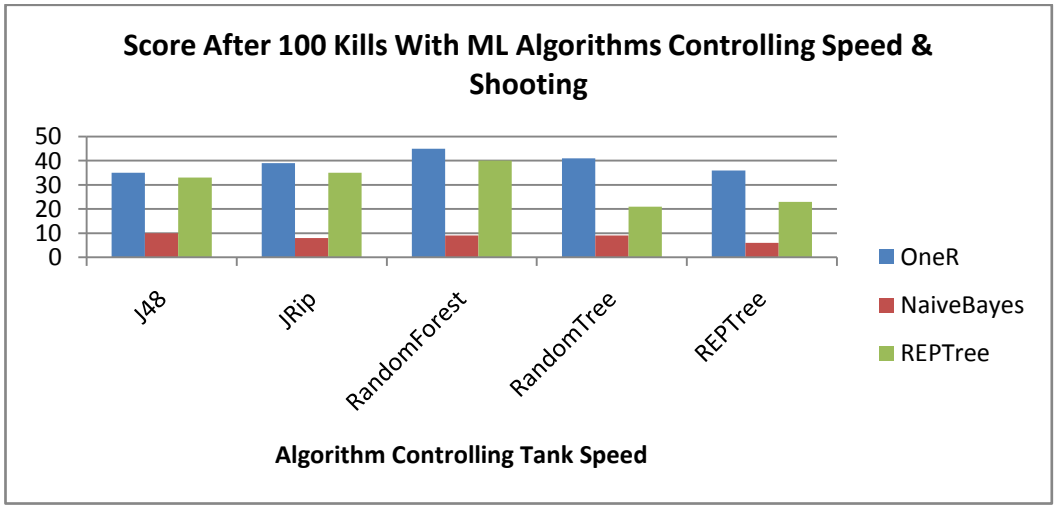

Figure 4.8 Score After 100 Kills With ML Algorithms Controlling Speed & Shooting

Rotation and Shooting

Figure 4.9 shows the scores obtained by tanks using ML algorithms to control tank rotation and shooting, while autopilot controls speed. The combinations which use OneR to control tank shooting consistently perform better in Figure 4.8, but here combinations with REPTree have similar or better performance in two of the five combinations. Similar to the speed+shooting results in Figure 4.8, NaiveBayes does poorly when used in combination with other algorithms controlling tank rotation.

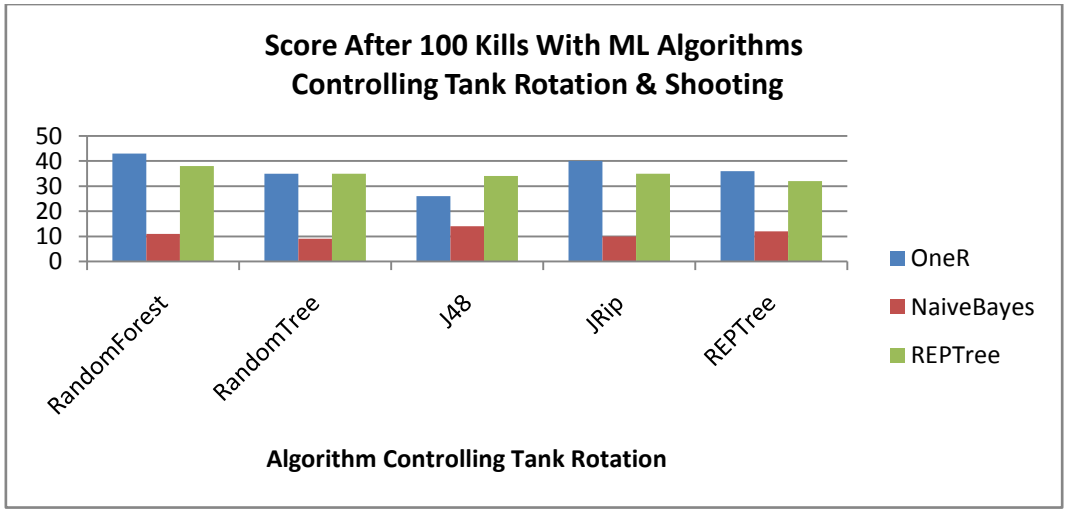

Figure 4.9 Score After 100 Kills With ML Algorithms Controlling Tank Rotation & Shooting

Speed and Rotation

Figure 4.10 shows the scores obtained by tanks using algorithms to control tank speed and rotation, while autopilot controls shooting. The combination using algorithms to control both speed and rotation clearly results in an extremely poor performance compared to results discussed previously in this section.

The results in Figure 4.8 and Figure 4.9 show that tanks using ML algorithms to control shooting and either speed or rotation can typically score at least 30 points in one combination against robot-pilot. Here however the combination using RandomTree to control speed and RandomForest to control rotation is the only combination that scores more than 10 points against robot-pilot.

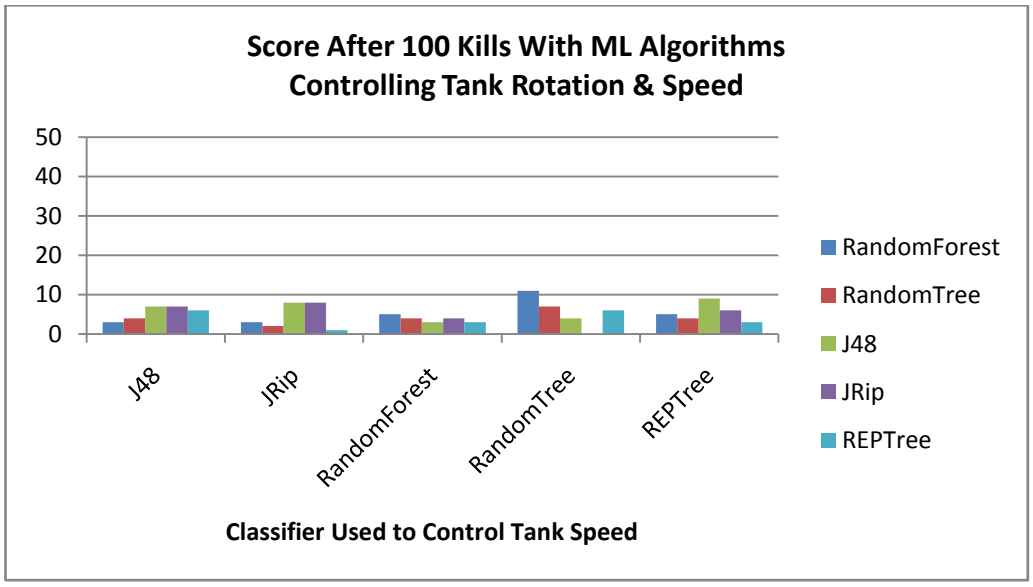

Figure 4.10 Score After 100 Kills With ML Algorithms Controlling Tank Speed & Rotation

4.3.3 Observations

The results shown in Figure 4.8 and Figure 4.9 both show it is possible for a combination of two ML algorithms achieve a better performance than either algorithm individually. For instance the combination shown in Figure 4.8, where RandomForest is used to control tank speed and REPTree is used to control shooting achieves a higher score than that of RandomForest (Figure 4.5) or REPTree (Figure 4.7) alone.

This indicates the combination of three ML algorithms may improve performance over that shown in Figure 4.10, but based on the results in Figure 4.8 and Figure 4.9 it is unlikely that any improvement achieved would be sufficient to match robot-pilot.

4.4 Independent Models

Inspection of the datasets described in Section 4.2.1 shows the poor performance shown in Figure 4.10, when ML algorithms are used to control both speed and rotation, may be caused by the prediction of one algorithm being used as attributes for another algorithm (either directly or indirectly). This section discusses experiments to determine if removing some of these attributes from the datasets is sufficient to increase the in-game performance of the algorithms.

4.4.1 Dataset Changes

Some attributes are removed from the datasets described in Section 4.2.1 to prevent the prediction of one ML algorithm being used as the input of another ML algorithm.

At the same time some attributes that do not contain useful information are also removed from the datasets. This includes all attributes related to the Z-axis (both velocity and relative position) because, although tanks can be allowed to ‘jump’

(described in Section 2.3.2), this is disabled during testing so changes on the Z-axis only occur when a tank blows up (at which point the living tank's actions become rather irrelevant until the dead player re-spawns, at least in a one-on-one match).

isObscured is also removed from all the datasets because, while this attribute contains useful information during the experiments described in Chapter 3, once the obstacle in the world is removed (described in Section 4.1.2) this value is always 'false' because there are no obstacles to obscure the opponent's tank.

Name	Description	Shoot	Speed	Rot.
MyVelocity (X,Y)	Velocity of player's tank along the axis.	**X**		**X**
EnemyVelocity (X,Y)	Velocity of opponent's tank along the axis.	**X**	**X**	**X**
RelativePosition (X,Y)	Position of opponent's tank on the axis, relative to player's tank.	**X**	**X**	**X**
EnemyDistance	Straight-line distance from the centre of player's tank to the centre of opponent's tank.	**X**	**X**	**X**
AngleDifference	Difference between the current rotation of the player's tank and the rotation which would point player's tank straight at opponent's tank (How far player's tank must rotate to be facing opponent's tank).	**X**	**X**	**X**
ShotRelative (X,Y)	Position of opponent's projectile on the axis, relative to the player's tank (Missing if opponent does not have an active shot).	**X**	**X**	**X**
ShotVelocity (X,Y)	Velocity of opponent's projectile along the axis (Missing if opponent does not have an active shot).	**X**	**X**	**X**
ShotDistance	Straight-line distance to opponent's projectile (Missing if opponent does not have an active shot).	**X**	**X**	**X**
MyRotation	Orientation of player's tank.			**X**
FiringStatus	Integer value, tank can only fire when value is 1 (meaning 'ready').	**X**		
Fire (Class)	Boolean value – True when a shot is fired, false otherwise.	**X**		
Speed (Class)	Desired speed of player's tank.		**X**	
NewRotation (Class)	Desired rotation of player's tank.			**X**

Table 4.2 Datasets Used for Independent Static Model Training

Shooting

Table 4.2 shows the attributes used in the shooting dataset (indicated with an 'X' in the shoot column). Attributes that relate to the Z-axis (*EnemyVelocityZ*, *RelativePositionZ*, *ShotRelativeZ*, and *ShotVelocityZ*) which are present in the dataset shown in Table 4.1 have been removed, as well as the now irrelevant *isObscured* attribute.

Both *MyRotation* and *MySpeed* attributes, which are present in the dataset shown in Table 4.1, have also been removed. Though neither value is the direct output of an ML algorithm (they measure the tank's current values, not its 'desired' values given by the algorithms), they are obviously strongly affected by predictions of the other algorithms, and so are removed to ensure the independence of the shooting ML algorithm.

Speed

The speed dataset shown in Table 4.1 does not include the *MySpeed* attribute. This is because, as discussed in Section 3.3.1, changes in speed happen less frequently than continuation of the current speed. If the algorithm uses this fact, it may predict a value based on the current speed that results in the tank never moving (because the initial speed is zero).

The *MyVelocity* attributes however are still present in the dataset shown in Table 4.1. The *MyVelocity* attributes are not directly affected by the speed ML algorithm's predictions because they refer to the tank's velocity along the world axes, whereas the speed ML algorithm controls the tank's speed in the direction it is facing. However, it is possible the performances of the speed ML algorithms are worse because they are using these attributes for prediction. Both cross-validation and in-game tests are performed to determine if inclusion of the *MyVelocity* attributes has a detrimental effect on speed ML algorithm performance.

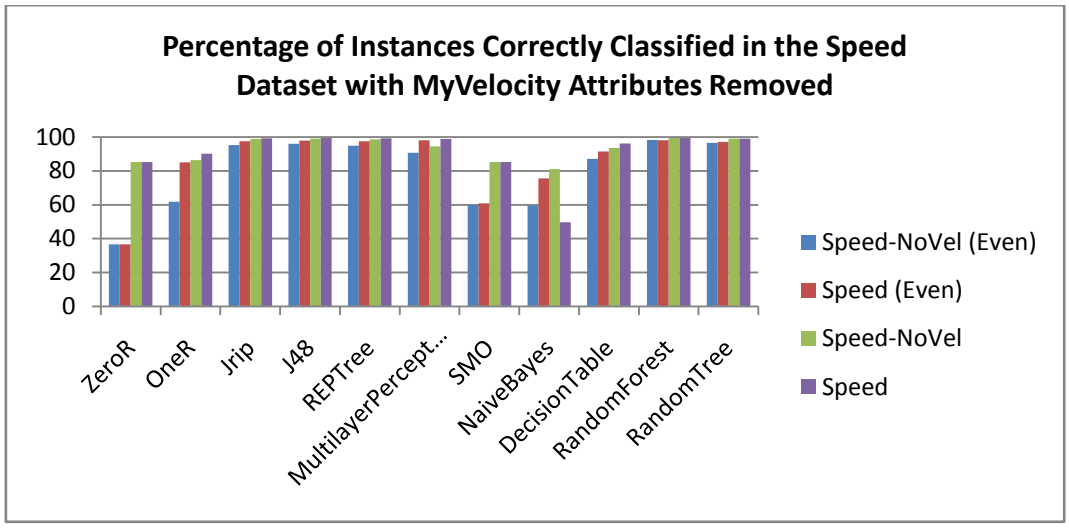

Figure 4.11 Percentage of Instances Correctly Classified in the Speed Dataset with MyVelocity Attributes Removed

Figure 4.11 shows the percentage of correctly classified instances in the speed dataset (using 10-fold cross-validation). The algorithms are trained on the same dataset shown in Table 4.1 with all three *MyVelocity* attributes removed (indicated with 'NoVel'). For ease of comparison the results obtained on the speed dataset shown in Table 4.1 are duplicated in Figure 4.11. As in Section 4.2.2, both the 'full' dataset and the 'even' dataset (with approximately even numbers of all classes) are tested.

These results do not show a strong indication the performance is improved by the removal of the *MyVelocity* attributes, with all algorithms performing the same or in some cases worse than the results obtained on the datasets that include *MyVelocity*. However, many of the algorithms classify more than 90% of the instances correctly, so achieving much higher performance is difficult. To get a more indicative measure of any performance difference the algorithms trained on the dataset with the *MyVelocity* attributes removed are tested for in-game performance.

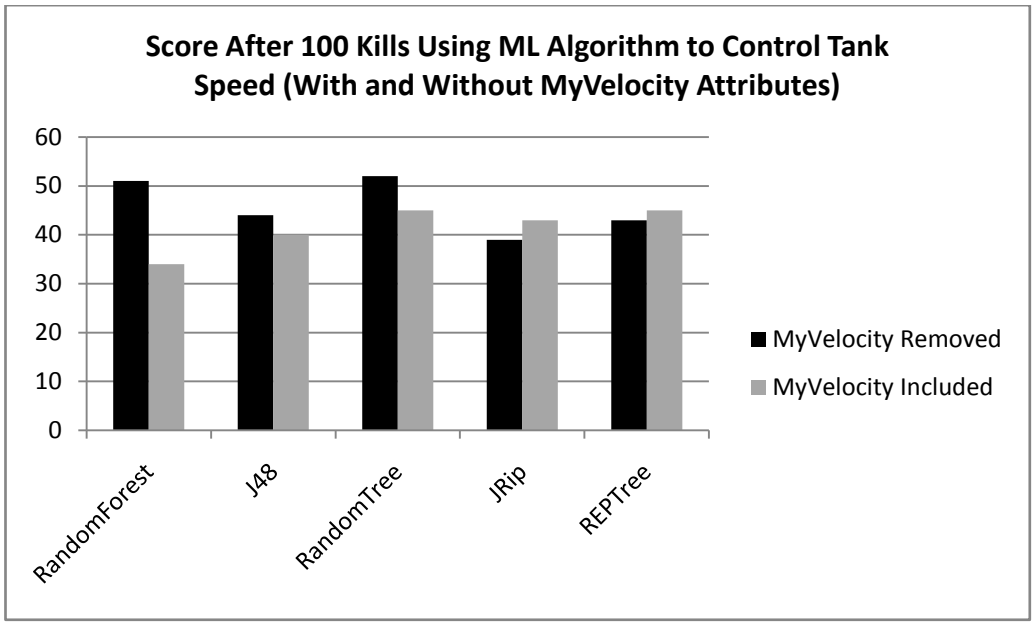

Figure 4.12 Score After 100 Kills Using ML Algorithm to Control Tank Speed (With and Without MyVelocity Attributes)

Figure 4.12 shows the results of the same five algorithms used for the tests described in Section 4.2.3. Only the full datasets are tested since the algorithm trained on the full dataset generally performs better in cross-validation tests. For ease of comparison the results from Section 4.2.3 (Figure 4.5) are duplicated in Figure 4.12.

These results show that removing the *MyVelocity* attributes from the training dataset can improve an algorithm's in-game performance. The change in performance is most notable in the RandomForest and RandomTree algorithms, both of which are capable of matching the in-game performance of robot-pilot when trained on the dataset with *MyVelocity* attributes removed. JRip and REPTree perform slightly worse with *MyVelocity* attributes removed but the decline in performance is minor compared to the improvement of the other algorithms.

Table 4.2 shows the data used in the speed dataset (indicated with an 'X' in the speed column). The speed dataset has all attributes removed that relate to the Z-axis, as well as the *isObscured* attribute. *MyRotation* is also removed from the dataset to ensure the algorithm's independence. Based on the results discussed

previously and shown in Figure 4.11 and Figure 4.12 the *MyVelocity* attributes are also removed from the dataset.

Rotation

Table 4.2 shows the dataset used for training algorithms to control tank rotation (indicated with an 'X' in the Rot. column). As with the other two datasets in Table 4.2 all the attributes relating to the Z-axis are removed, as well as *isObscured*. *MySpeed* is also removed from the dataset to ensure the independence of the rotation ML algorithm.

It should be noted that while the speed dataset shown in Table 4.2 has *MySpeed* removed, the rotation dataset still includes *MyRotation*. This is because, while the tank's speed does not change frequently, rotation changes relatively often and, as discussed in Section 3.1, *MyRotation* measures the tank's actual orientation, not its turning speed as given by the algorithm.

4.4.2 Single Static Model

Figure 4.12 indicates the difference a change to the dataset can have on ML algorithm performance. Because of this, the algorithms are tested again using both 10-fold cross-validation and in-game tests.

All experiments in this section use the same data as in Section 4.2.1, meaning that all instances are the same but some attributes have been removed (discussed in the previous section). The world configuration is the same as that used previously in this chapter (described in Section 4.1).

For the sake of completeness, some ML algorithms overlooked in the first half of this chapter are added to the test-set. PART is included due to its good performance during tests described in Section 3.2.3. Two versions of IBk, a nearest-neighbour algorithm, are also included in the test-set. One version uses 1

nearest-neighbour, the other uses a setting of 2 nearest-neighbours (labelled as 'IBk(2)').

Speed

Figure 4.13 shows the percentages of correctly classified instances in the speed dataset described in Section 4.4.1 (using 10-fold cross-validation). As with the experiments described in Section 4.2.2, two datasets are used, one with all the instances included and the second (indicated with '(Even)') with approximately even numbers of each class.

The results discussed in Section 4.2.3, and those in Figure 4.12, show that algorithms with similar performance on the dataset in cross-validation tests do not necessarily have similar performance during in-game tests. Because of this, algorithms trained on the even dataset are considered independently, regardless of the performance of the algorithm trained on the full dataset.

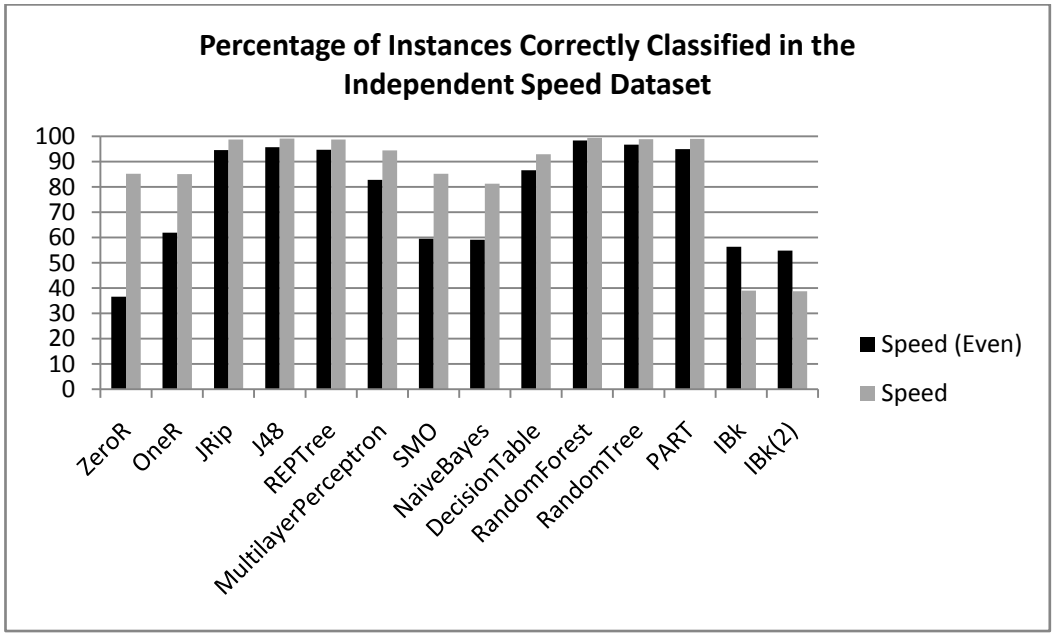

Figure 4.13 Percentage of Instances Correctly Classified in the Independent Speed Dataset

Also because of the difference between the cross-validation test results and the in-game test results, any algorithm that scores over 90% in the cross-validation test is tested for in-game performance rather than taking the top five algorithms (as in the experiments in Section 4.2).

Figure 4.14 shows in-game results of the ML algorithms trained on the datasets described in Section 4.4.1. For easier comparison, the relevant results from Figure 4.5 are duplicated here as 'Previous Results'. Note that MultilayerPerceptron(Even) did not score above 90% in the cross-validation test described previously and so is not included in the in-game test-set.

These results show the removal of attributes (discussed in Section 4.4.1) has generally improved performance of the algorithms during in-game tests. The results also show that while algorithms trained on the even dataset have performance similar or worse than algorithms trained on the full dataset, the in-game results show that some algorithms have better in-game performance after being trained on the even dataset.

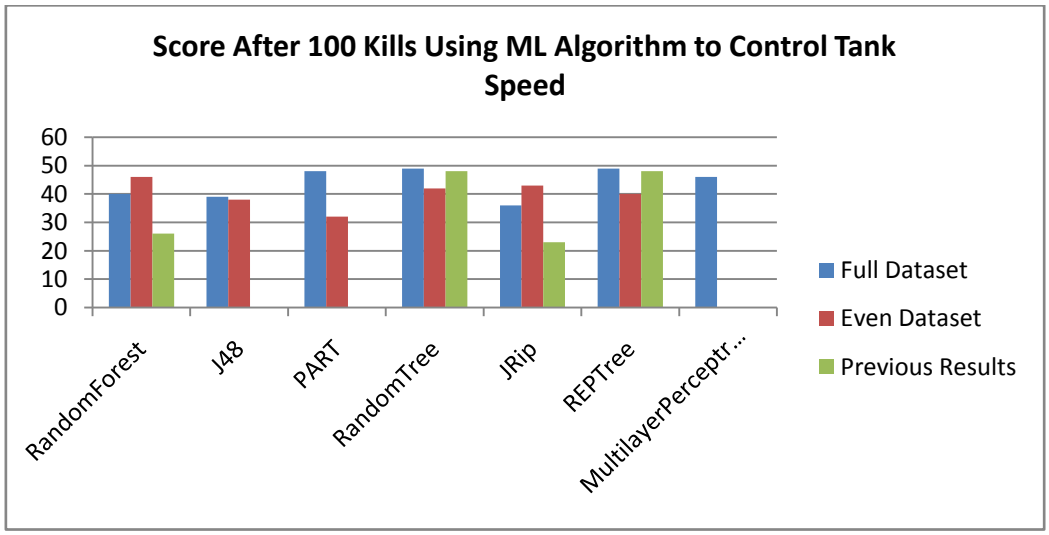

Figure 4.14 Score After 100 Kills Using ML Algorithm to Control Tank Speed

Rotation

Figure 4.15 shows very little difference between the algorithms trained on the even and full datasets. This is probably due to the even spread of the rotation data; whereas the speed data has a non-uniform distribution (full speed forward is the majority class).

Experiments described in this chapter show that ML algorithms can have different levels of in-game performance despite having similar scores in cross-validation tests. Because of this, algorithms trained on both the even and full datasets are considered independently. As with speed described previously, all algorithms that score higher than 90% in the cross-validation test are tested for in-game performance.

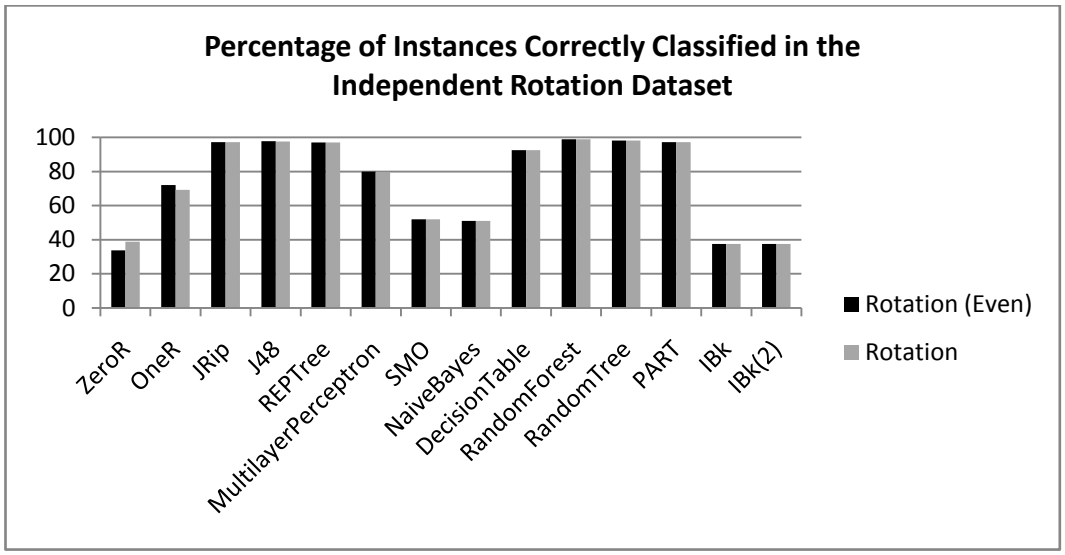

Figure 4.15 Percentage of Instances Correctly Classified in the Independent Rotation Dataset

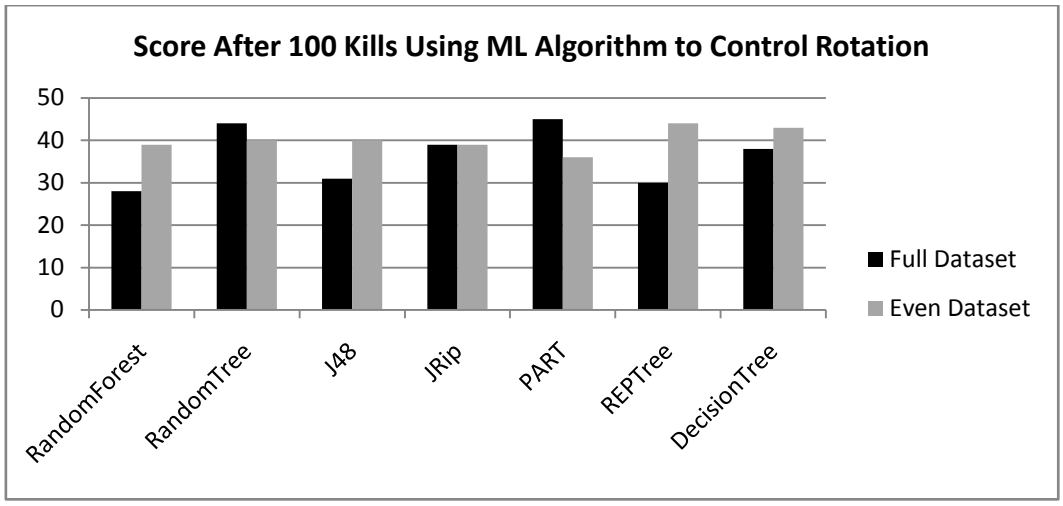

Figure 4.16 Score After 100 Kills Using ML Algorithm to Control Rotation

Figure 4.16 shows that, as with speed discussed previously, despite almost identical performance of the algorithms during cross-validation tests, there is often a difference in in-game performance between algorithms trained on the full dataset and those trained on the even dataset. The figure also shows that, despite the rotation data being evenly spread across all classes, some algorithms still perform better when trained on the even dataset. This is most notable in REPTree and RandomForest, with a difference of at least 10 points between the algorithms trained on the even and full datasets.

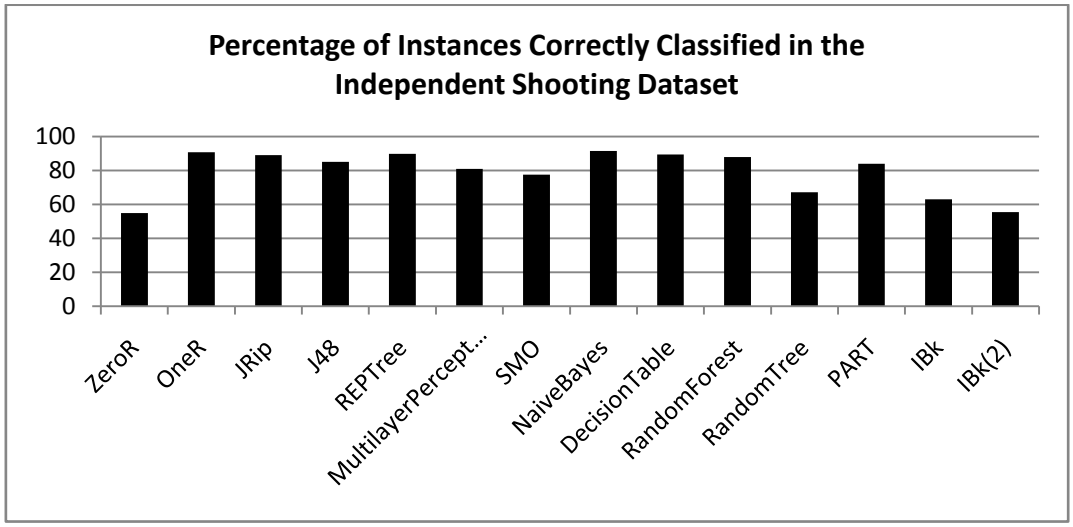

Figure 4.17 Percentage of Instances Correctly Classified in the Independent Shooting Dataset

Shooting

Figure 4.17 shows results of the cross-validation tests of the algorithms trained on the independent shooting dataset (shown in Table 4.2). Because of the large number of negative instances in the shooting dataset (as mentioned in Section 3.2.3), only the even dataset is used for training.

Unlike the speed and rotation results discussed previously, very few algorithms score higher than 90% in the cross-validation tests. Because of this, the top five algorithms are tested for in-game performance instead of only those above 90%.

Figure 4.18 shows in-game results of the algorithms that score the five highest results in the cross-validation tests described previously. These results are similar to those shown in Figure 4.7, with OneR performing better than any of the other algorithms tested and, unlike the results in Figure 4.7, OneR now actually out-performs robot-pilot in the one-on-one match.

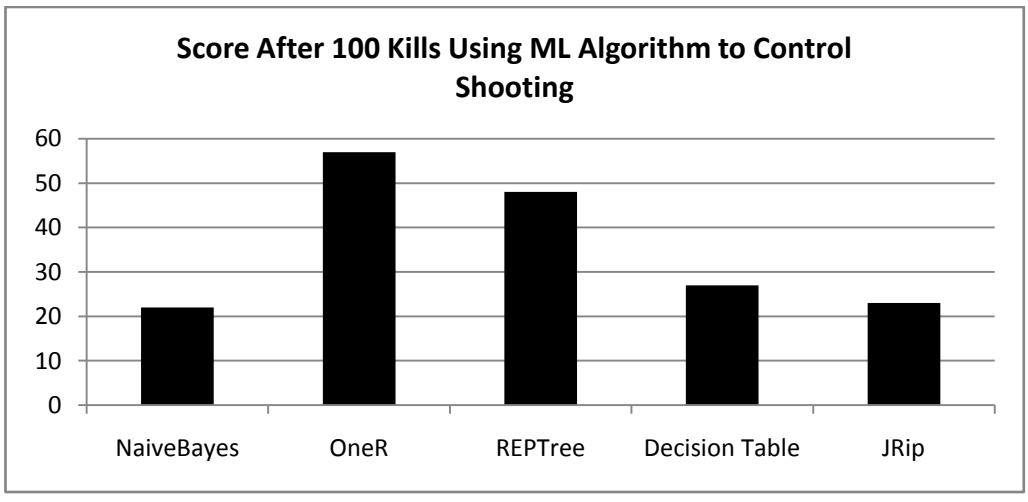

Figure 4.18 Score After 100 Kills Using ML Algorithm to Control Shooting

```
AngleDifference:
< 0.0686315    -> True
>= 0.0686315  -> False
```

Figure 4.19 Rule-Set Created By OneR

Inspection of the rule-set created by OneR, displayed in Figure 4.19, reveals that it makes use of the *AngleDifference* attribute. This shows an excellent form of generalization, where robot-pilot has a much more complex consideration before firing a shot.

Another difference from the results in Figure 4.7 is that REPTree now performs almost equal against robot-pilot. This is a good indication that REPTree over-fits when trained on the dataset shown in Section 4.2.1, so removal of irrelevant attributes improves the algorithm's in-game performance.

4.4.3 Dual Static Models

The results presented in the previous section show the in-game performance improves when irrelevant attributes are removed from the training datasets. The results for both speed and rotation, displayed in Figure 4.14 and Figure 4.16, show it is possible for an ML algorithm to control one aspect of tank behaviour sufficiently well to match robot-pilot's performance. The shooting results, displayed in Figure 4.18, show that it is even possible for a tank using an ML algorithm to handle one aspect of tank control to out-perform robot-pilot.

This section discusses experiments to determine if this level of performance can be maintained or exceeded by using two ML algorithms together to control two aspects of tank behaviour simultaneously.

Algorithm Selection

The number of algorithms tested in combination has to be limited because of the time consuming nature of testing and the large number of combinations available. Only OneR and REPTree are used to control shooting. This is for two reasons; firstly it limits the number of combinations to test, and secondly because of the large difference in the performance of the shooting ML algorithms shown in Figure 4.18.

To reduce the time needed for testing, only the speed+shooting and shooting+rotation combinations are considered at this stage. These results are then used to determine which ML algorithms perform well together and the combinations to use for controlling all three aspects of tank behaviour in the next section.

As described in the Section 4.4.2, in-game performance of an algorithm can be quite different depending on whether it is trained on the full or even dataset. Because of this, ML algorithms trained on the full dataset are considered independently of those trained on the even dataset.

Speed and Shooting

Figure 4.20 shows in-game performance of the ML algorithm combinations. Note that MultilayerPerceptron is not tested using the even dataset because it scores less than 90% in the cross-validation tests shown in Figure 4.13.

Unlike the individual in-game speed results shown in Figure 4.14, where algorithms trained on the even dataset out-perform algorithms trained on the full dataset in almost half the tests, here we see that algorithms trained on the full dataset generally out-perform algorithms trained on the even dataset. The only exceptions to this are RandomTree+OneR which performs better when trained on the even dataset, and PART+REPTree which performs about the same regardless of the dataset.

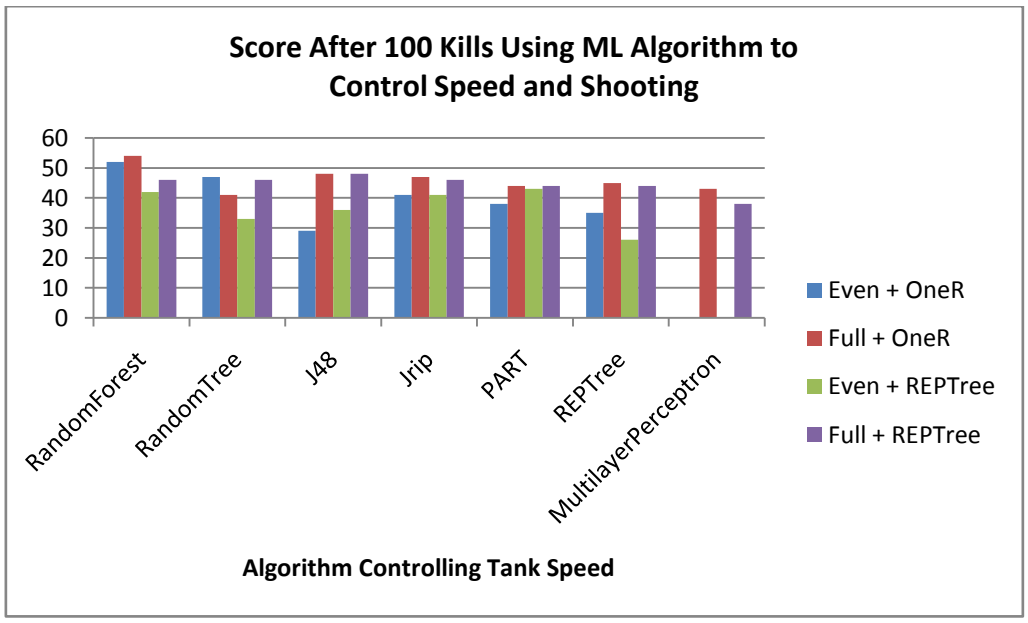

Figure 4.20 Score After 100 Kills Using ML Algorithms to Control Speed and Shooting

This difference from the results in Figure 4.14 could indicate the non-uniform distribution of speed instances is an important component for ML algorithm performance, or alternatively the difference may indicate that algorithms trained on the full dataset perform better because it has more instances available and thus the algorithms are better equipped to infer the correct response in situations not presented by autopilot. Further experimentation could be done to determine the exact reason but this is outside the scope of this report and remains a possible topic for future work.

The results in Figure 4.20 also show an improvement over those in Figure 4.8, with all algorithms managing to score over 40 points in at least one configuration. The combination of RandomForest and OneR however is the only one that manages to out-perform robot-pilot (though only slightly with a score of 52).

Shooting and Rotation

Figure 4.21 shows in-game performance of the shooting+rotation ML algorithm combinations. These results show a small improvement over the results given in Figure 4.9, with four combinations scoring at least 40 points against robot-pilot. The only combination that comes close to equalling robot-pilot's performance is DecisionTable trained on the even dataset using OneR to control shooting with a score of 48.

The shooting+rotation combinations do not have the same level of in-game performance increase as the speed+shooting combinations discussed earlier; however, as noted in Section 4.3.3, it is possible for a combination of algorithms to perform better than either of the algorithms alone. So, it is possible that while the shooting+rotation combinations do not out-perform robot-pilot, the combination of all three ML algorithms may improve in-game performance.

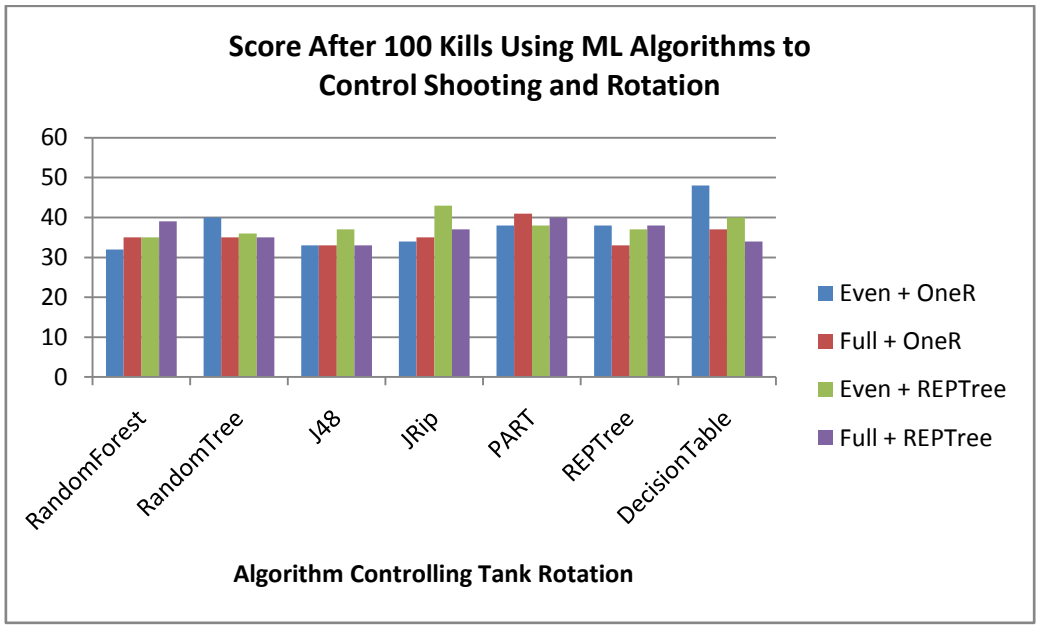

Figure 4.21 Score After 100 Kills Using ML Algorithms to Control Shooting and Rotation

4.4.4 Triple Static Models

Based on the in-game results described in the previous section, the best two-algorithm combinations are combined into three-algorithm combinations. For example; Figure 4.20 shows that JRip performs best with REPTree when trained on the full speed dataset, while Figure 4.21 shows that DecisionTable performs best with REPTree when trained on the even rotation dataset. Based on this the JRip(Even)-REPTree-DecisionTable(Even) speed-shoot-rotation combination is tested, but combinations such as JRip-REPTree-DecisionTable, and JRip(Even)-REPTree-DecisionTree are not tested.

To reduce the time required for in-game tests, the combinations are first tested up to 30 kills. The best performing combinations are then tested again up to 100 kills.

Figure 4.22 shows in-game performance of the combinations which use REPTree to control shooting. Although these results indicate an improvement over the results in Figure 4.10, they are still quite poor. The random nature of BZFlag means the score after 30 kills between two evenly matched opponents may not be close to 15, because it often takes a longer run for scores to even out (as more world states are encountered). However, it is not unrealistic to expect a score of at least 10 if the tanks are equally matched.

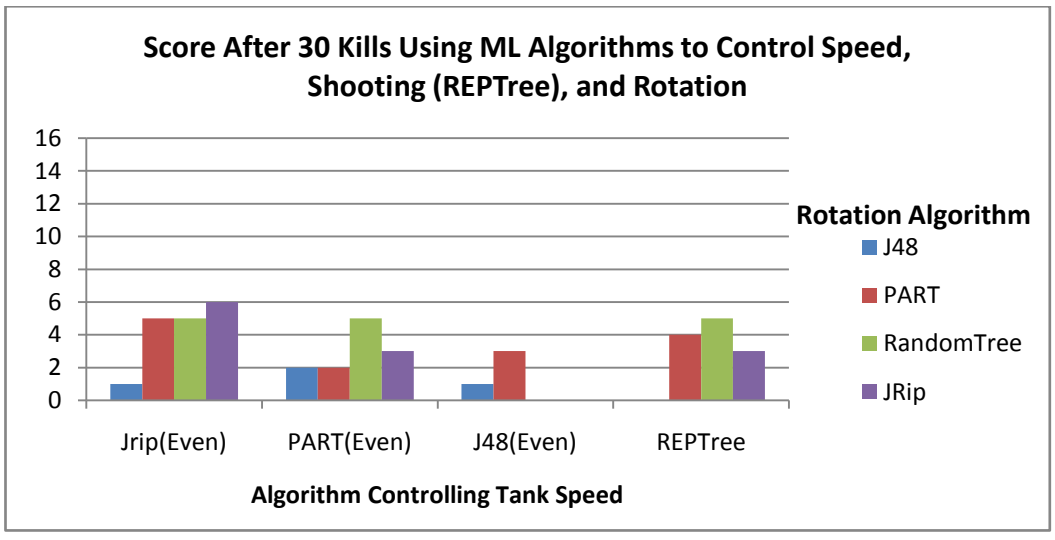

Figure 4.22 Score After 30 Kills Using ML Algorithms to Control Speed, Shooting (REPTree), and Rotation

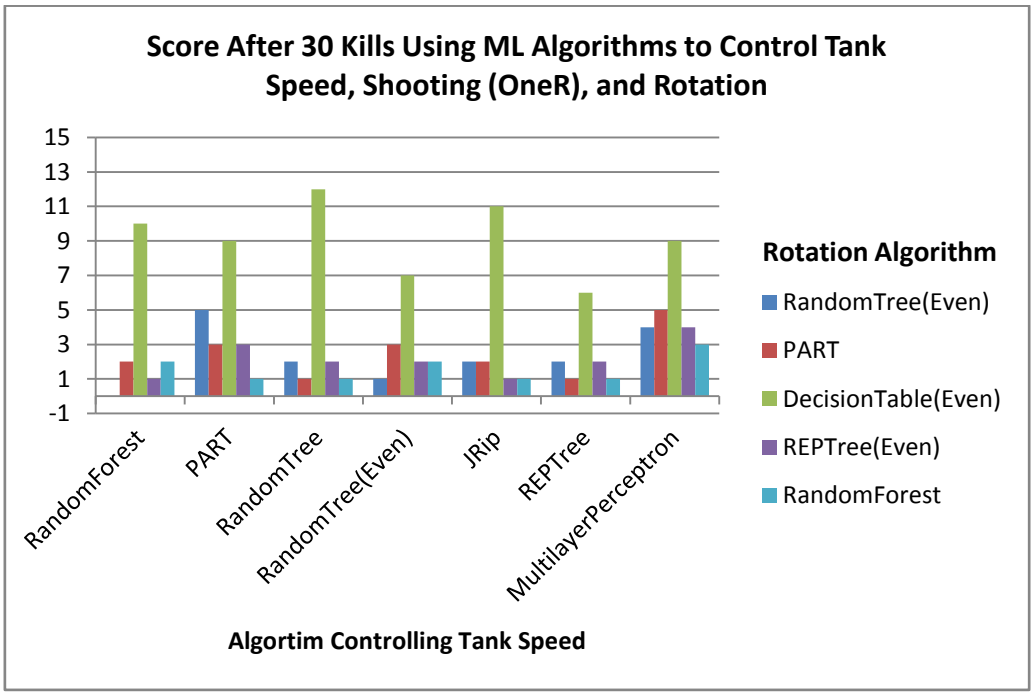

Figure 4.23 Score After 30 Kills Using ML Algorithms to Control Tank Speed, Shooting (REPTree), and Rotation

Figure 4.23 clearly shows the best combinations are those that use OneR to control tank shooting and DecisionTable(Even) to control rotation. All combinations with these two algorithms out-perform or match all other combinations shown in Figure 4.22 and Figure 4.23.

Based on the results in Figure 4.23, all combinations that score more than 8 points are used for a full in-game test up to 100 kills.

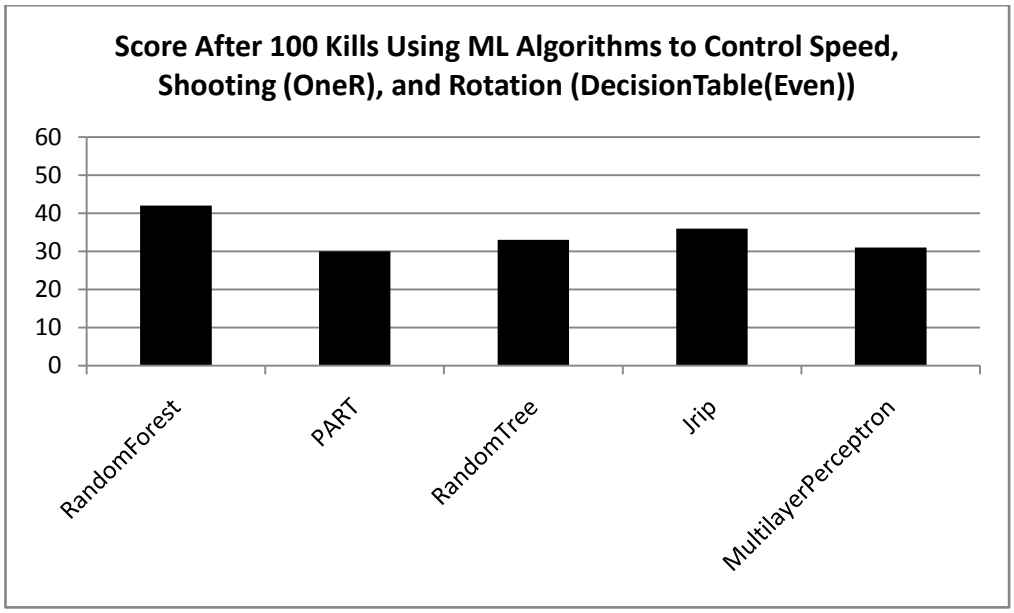

Figure 4.24 Score After 100 Kills Using ML Algorithms to Control Speed, Shooting (OneR), and Rotation (DecisionTable(Even))

Figure 4.24 shows the best performing combinations are only capable of scoring around 40 points against robot-pilot and, given that robot-pilot can be beaten by even an intermediate human player, are unsuitable to be used as an opponent for a human player.

4.5 Chapter Summary

Chapter 4 describes experiments to control a tank in BZFlag using static prediction models. It is possible to equal the performance of autopilot using a static model to control one aspect of tank behaviour, and in some cases it is even possible to out-perform robot-pilot.

A combination of two ML algorithms can perform better than either algorithm alone, but none of the combinations tested are able to easily out-perform robot-pilot. The performance when using three ML algorithms to control all aspects of tank behaviour is worse, with none of the combinations tested being able to match the performance of the standard robot-pilot.

It is possible that further ‘fine tuning’ of algorithm parameters or attributes in the training datasets could improve in-game performance. However, since all three-algorithm combinations tested are unable to match robot-pilot it seems unlikely that any performance improvements achieved by this would be sufficient to easily beat robot-pilot, and therefore be suitable for testing against human players.

5 Continuous Learning

This chapter describes experiments to determine whether the in-game performance achieved in Chapter 4 can be improved using continuous learning (CL) with one of the machine learning (ML) algorithms. This includes selection of the algorithm combinations and testing in-game performance.

Section 5.1 describes the configuration used to train the ML algorithms. Section 5.2 discusses selection of algorithms used for testing in-game CL. Section 5.3 discusses the results of short duration in-game tests and Section 5.4 goes on to discuss the results of longer duration tests. Section 5.5 gives a brief summary of this chapter.

All experiments in this chapter use the same world configuration and scoring mechanism described in Section 4.1 (400x400 plane, jumping and flags disabled, no obstacles).

5.1 Offline Training Configuration

The configuration used to train the ML algorithms is similar to that described in Section 3.3.5; BZFlag sends world state information to WEKA-Server and awaits a response. WEKA-Server then uses the most recently trained ML algorithm to determine the value to send to BZFlag. All instances received by WEKA-Server are sent to ClassifierBuilder. ClassifierBuilder retrains the ML algorithm when required and sends the updated version to WEKA-Server.

The configuration used for experiments in this chapter differs from that in Section 3.3.5 in that ClassifierBuilder starts with some instances already loaded. The starting instances are from the relevant datasets (described in Section 4.4.1). Both even and full datasets are used depending on the algorithm (except for shooting which always uses the even dataset). WEKA-Server starts with an ML algorithm trained on the initial instances. Each instance sent from WEKA-Server to ClassifierBuilder also has its class value set to the predicted value from the ML algorithm in WEKA-Server.

Put another way, the ML algorithm starts with a number of instances for training and then adds new training instances with its predicted class value. As one might expect, this does not improve performance if the algorithm is already performing poorly but, if the algorithm performs sufficiently, performance can be improved using this method [Vega and Bressan, 2003].

5.2 Algorithm Selection

No studies could be found that suggest any particular ML algorithms are better suited to CL than any other ML algorithms. To confirm this, a small test is conducted using CL with the rotation ML algorithm while the standard autopilot controls speed and shooting. The algorithms tested are the same as those shown in Figure 4.21, using the dataset that gives the best performing static model for the initial training.

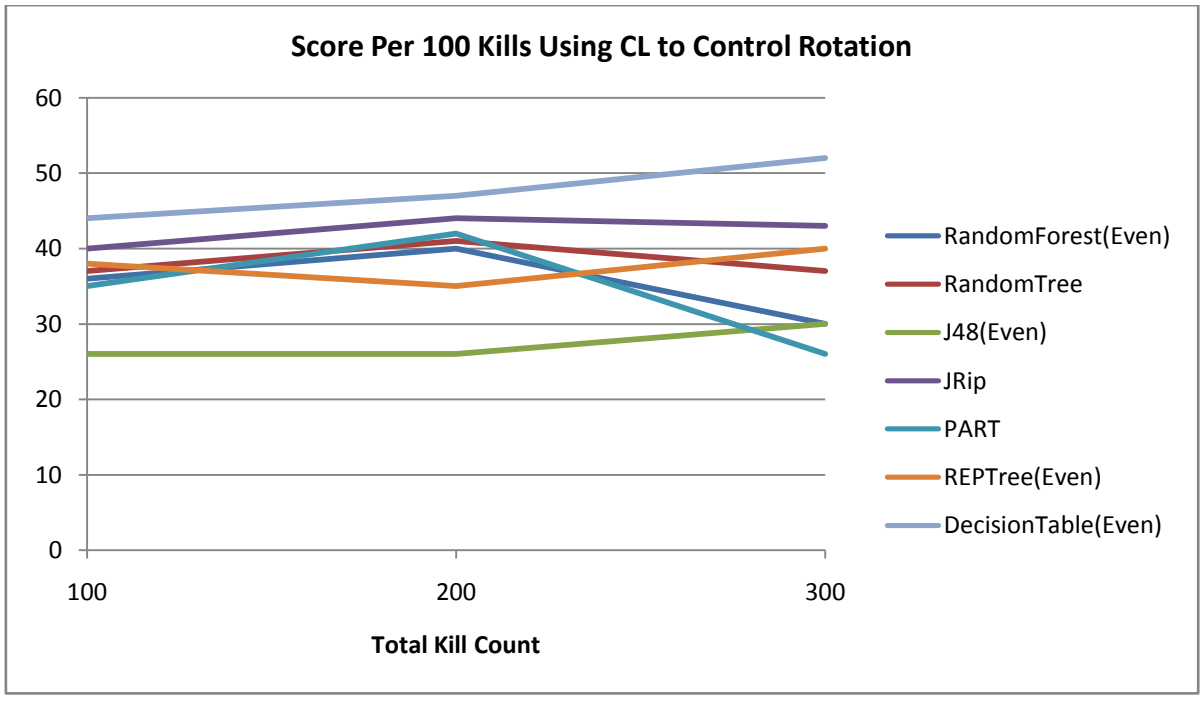

Figure 5.1 Score Per 100 Kills Using CL to Control Rotation

Figure 5.1 shows in-game performance using CL to control rotation with (Even) after the algorithm name indicating the even dataset is used. Each in-game test is run until the total kill count reaches 300, this number is arbitrary but allows for the effect of CL to become apparent without taking an excessive amount of time. The score is recorded after each 100 kills.

Figure 5.1 also shows DecisionTable(Even) is the best performing algorithm tested. This is expected because, as noted in Section 5.1, any performance increases during CL are dependent on the initial performance of the algorithm itself, and the results in Figure 4.27 show that DecisionTable(Even) is the best performing rotation ML algorithm.

The performance of DecisionTable(Even) also improves steadily during the test, whereas some of the other algorithms show more volatile changes in performance that may indicate a reduced ability to improve long term performance.

These results are by no means a comprehensive study of each algorithm's suitability for CL, but with no studies found to indicate the contrary it is assumed that algorithms with the best performance in experiments discussed in Section 4.4.4 are likely to have the best performance when used with CL.

5.3 Short Duration In-Game Testing

Each test is run until the total kill count reaches 300 with the score recorded after each 10 kills. This creates 30 data points, referred to as 10-kill blocks. The ability for each algorithm to improve performance is determined by plotting these data points and comparing the slope of the linear trend lines. 300 is an arbitrary number but is selected to be large enough that increases in performance are detectable but small enough so as not to require excessive time for testing.

Section 5.2 states the assumption that algorithms that perform well in the experiments described in Section 4.4.4 are the most likely to improve performance using CL. Based on this assumption there is little to be gained by re-testing performance of single and dual algorithms for in-game performance, so the

best three-algorithm combinations are tested using CL to determine if the performance improves over that achieved using static models (described in Section 4.4.4). All combinations tested in this section use DecisionTable(Even) as the rotation algorithm and, although OneR is clearly the best performing shooting algorithm, some combinations that use REPTree to control shooting are also included for completeness.

Only one of the three algorithms uses CL for these tests. This is because it simplifies testing and it is unlikely that a CL algorithm which performs poorly when combined with static models will perform better when combined with other CL algorithms.

Experiments in this chapter use CL on the speed and rotation ML algorithms only. This is because experiments in Chapter 4 show that these two controls have the most detrimental effect on in-game performance, and thus are the areas that must be improved in order to match the performance of robot-pilot.

5.3.1 Continuous Speed Learning Algorithms

CL is first trialled for algorithms that control speed. All combinations trialled in this section use DecisionTable(Even) to control rotation.

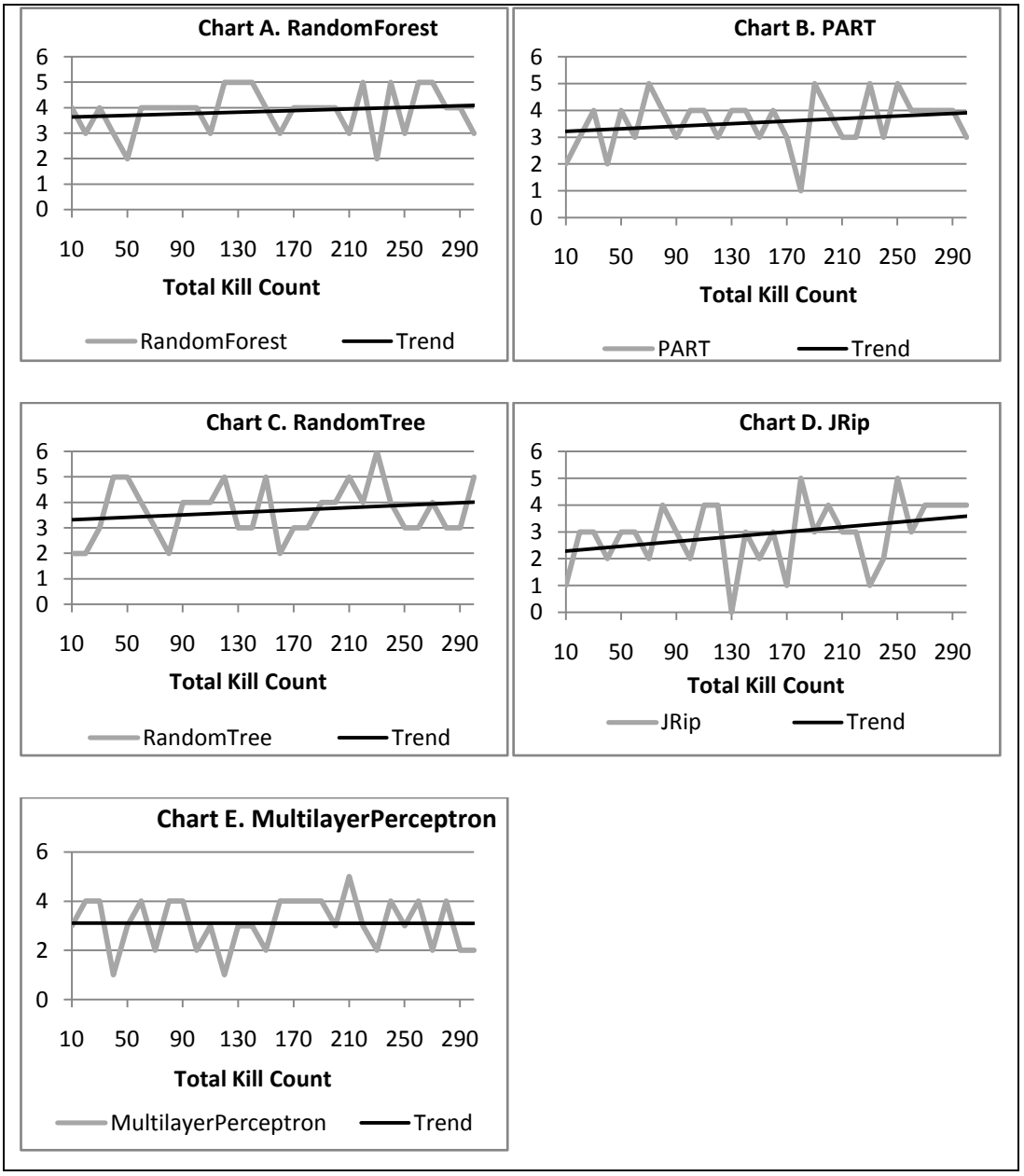

Figure 5.2 Points Scored Using CL to Control Speed

Combinations Using OneR to Control Shooting

Figure 5.2 shows the performance when CL is used to control speed and OneR to control shooting. In Chart A the grey line shows the score achieved by autopilot (i.e. how many times autopilot shot the opponent during the last 10 kills), with the black line showing the linear trend of the score. If the two tanks have similar performance the score should vary around five points. The five combinations tested have scores closer to three points, indicating worse performance than robot-pilot. Chart C shows the combination using RandomTree is the only one of the five that manages to score more than five points in any 10-kill block, but the overall score is still not adequate to match robot-pilot.

Chart B and Chart D show that both PART and JRip have sudden drops in performance in a few 10-kill blocks. The exact reason for this is unclear but it may be simply an 'unlucky' set of random starting states for the particular algorithm.

The linear trend lines show that, although all the combinations tested are unable to match the performance of robot-pilot, the performance does increase over time when using CL. The one exception to this is MultilayerPerceptron shown in Chart E, which has more or less constant performance regardless of the duration.

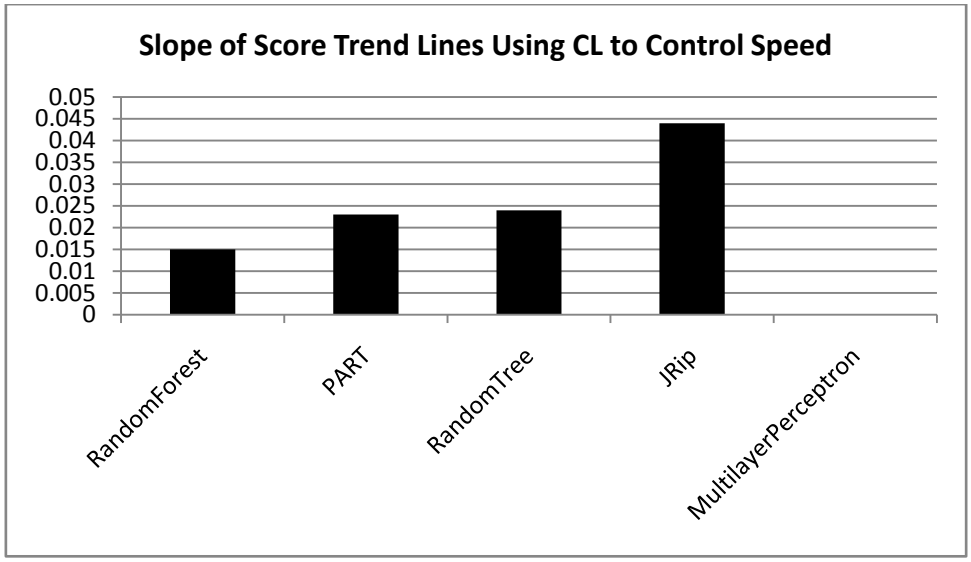

Figure 5.3 Slope of Trend Lines Using CL to Control Speed

For easier comparison the slopes of the trend lines in Figure 5.2 are displayed together in Figure 5.3. JRip clearly has the largest performance increase, with a slope over twice as large as the other algorithms.

MultilayerPerceptron has a slope of zero, indicating the performance does not change regardless of the duration (as noted previously). The reason for this is not known but may indicate the algorithm is not suited for CL.

The remaining three algorithms have similar slopes. Interestingly RandomForest does worse than RandomTree despite the similarity of the algorithms. The underlying reasons why some algorithms show greater performance increases using CL are beyond the scope of this report and remains a possible topic for future work.

Combinations Using REPTree to Control Tank Shooting

Figure 5.4 shows the performance of the combinations when CL is used to control speed and REPTree to control shooting. Comparing Figure 5.4 with Figure 5.2 shows the choice of algorithms used in the combination can have a large impact on the effectiveness of CL. This is expected because an algorithm's ability to improve performance depends heavily on its initial performance (as noted in Section 5.2) and Section 4.3.2 shows the performance of a single algorithm is affected by other algorithms used in combination with it.

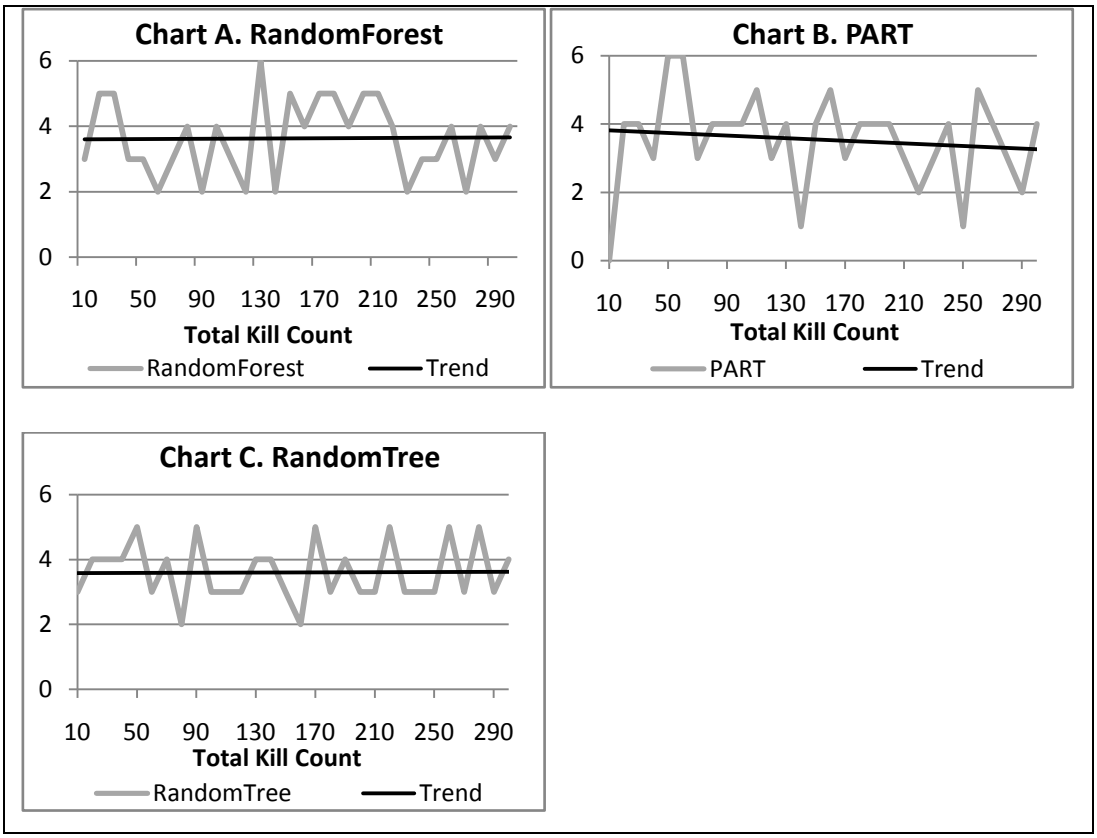

Figure 5.4 Points Scored Using CL to Control Speed (REPTree for Shooting)

These results also help confirm the assumption made in Section 5.2; the best performing static model combinations are the most likely to perform well when using CL. The best performing combinations in Section 4.4.4 are those that use OneR to control shooting and that result emerges again here.

Interestingly the performance of some combinations in Figure 5.4 perform better than their OneR counterparts in Figure 5.2, yet the linear trend lines in Figure 5.4 show the performance has little, if any, improvement and even declines over time in Chart B. This is difficult to explain but may indicate that REPTree does not allow the CL speed algorithm to improve beyond the initial performance. This is not investigated because there is negligible performance increase in the combinations using REPTree for shooting so it is not useful for improving tank performance but could be a topic of future work.

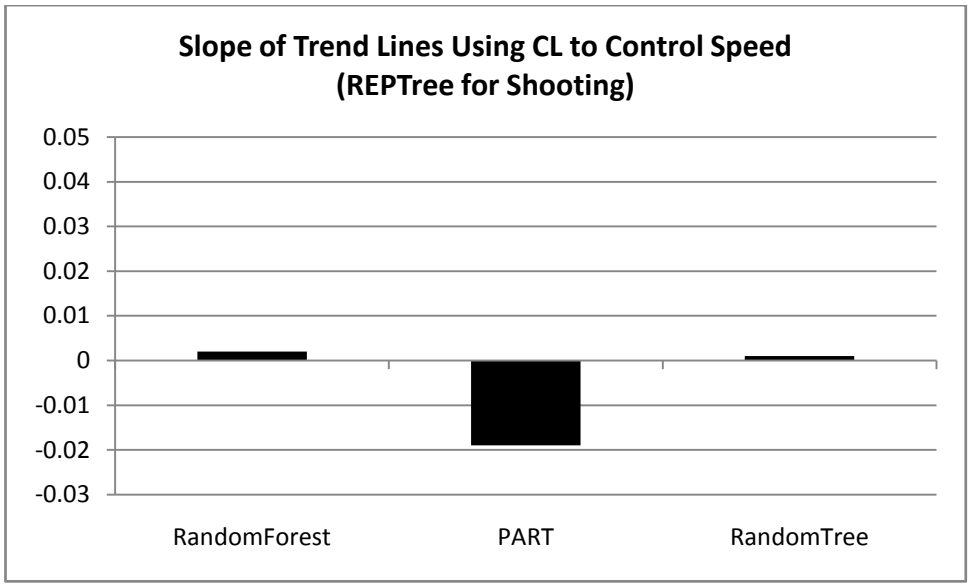

Figure 5.5 Slope of Trend Lines Using CL to Control Speed (REPTree for Shooting)

For easier comparison, Figure 5.5 shows the slopes of the trend lines shown in Figure 5.4. This confirms the performance of the combinations using REPTree for shooting have a much lower rate of improvement than the combinations using OneR shown in Figure 5.3. RandomForest and RandomTree show a minimal level of increase, while the performance of the combination using PART declines over time.

These results are consistent with expectations based on the results discussed in Section 4.4.4 that show combinations using OneR clearly out-perform combinations which use REPTree to control tank shooting.

5.3.2 Continuous Rotation Learning Algorithm

Section 5.3.1 shows the combinations that perform best in the experiments described in Section 4.4.4 are the same ones that perform best when one of the algorithms uses CL. As a result, DecisionTable(Even) is the only algorithm used for the tests described in this section.

The results in Section 5.3.2 also show the performance achieved by CL can be strongly affected by the other ML algorithms used in the combination. Because of this, and the results discussed in Section 5.3, no combinations that use REPTree to control shooting are tested, so all experiments in this section use OneR to control shooting.

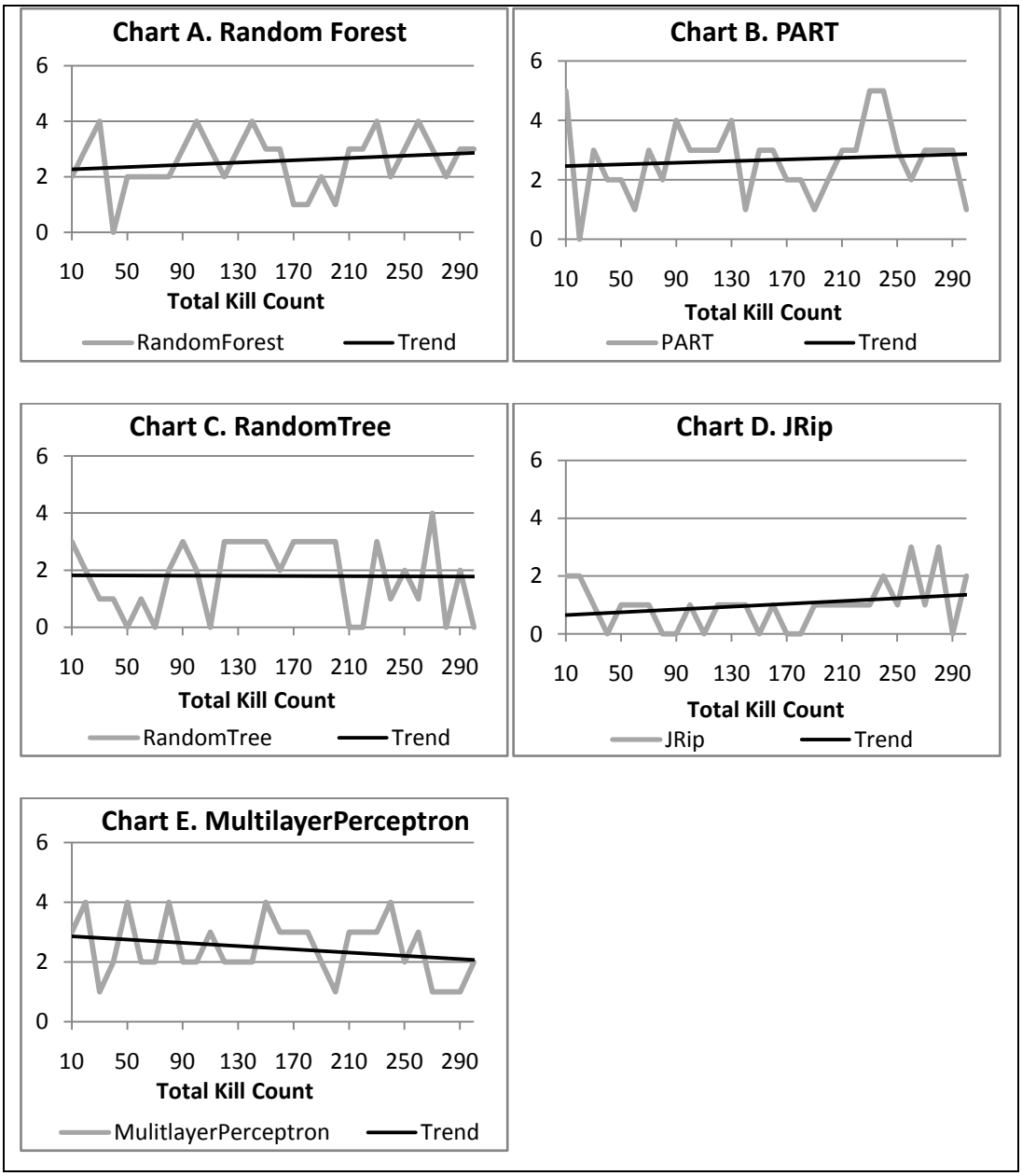

Figure 5.6 Points Scored Using CL to Control Rotation

Figure 5.6 shows the performance of combinations that use OneR to control shooting and CL on the rotation algorithm (DecisionTable(Even)). The speed and shooting algorithms are the same as the static models described in Section 4.4.4. The performance of these combinations is generally worse than combinations that use CL speed algorithms (Figure 5.2). In Figure 5.2 all the combinations manage a score of five in at least one 10-kill block, yet in Figure 5.6 PART (Chart B) is the only combination that manages to score five points in any 10-kill block.

Three of the five combinations tested still have a performance increase over time, but the performance is worse than when CL is used to control speed (in Figure 5.2).

For easier comparison, the slopes of the trend lines in Figure 5.6 are displayed together in Figure 5.7. This shows a similar ordering to the slopes in Figure 5.3 with the exception that RandomTree performs much worse in Figure 5.7. JRip has the largest performance increase in both tests, though the increase is not as large in Figure 5.7. MultilayerPerceptron performs poorly in both tests, which may indicate the algorithm is not well suited for CL. The exact reason for this is outside the scope of this report and remains a possible topic for future work.

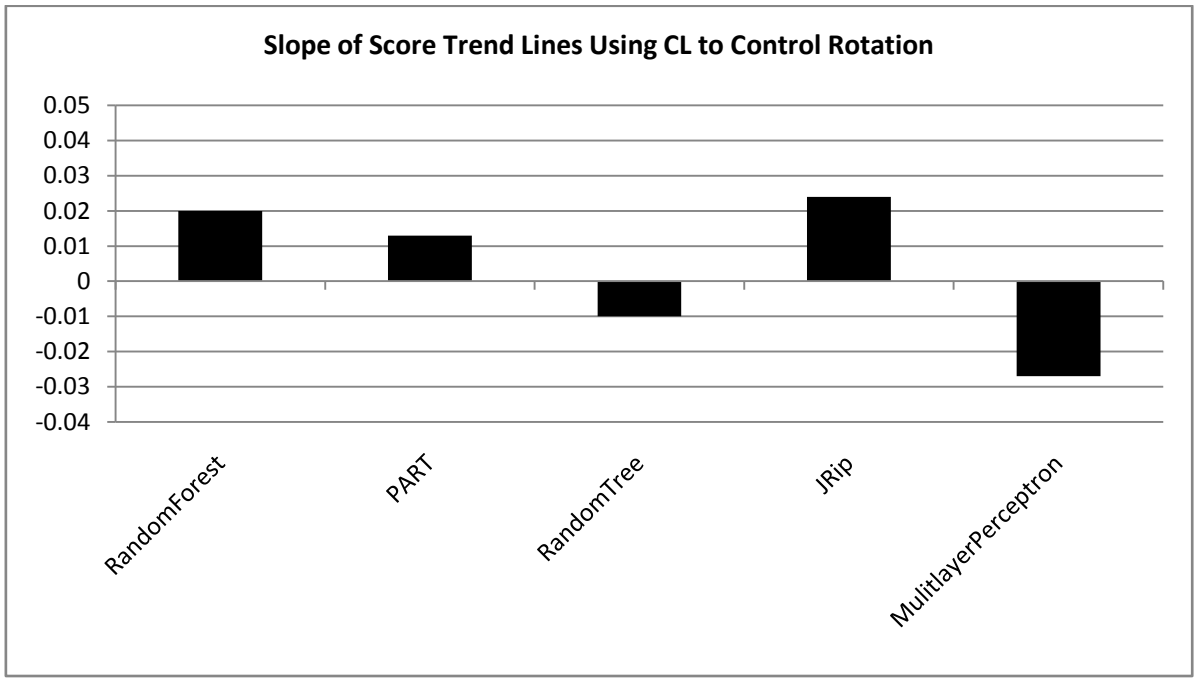

Figure 5.7 Slope of Trend Lines Using CL to Control Rotation

5.4 Long Duration In-Game Testing

Using results from Section 5.2 and Section 5.3, the combinations that perform best with CL are tested until the total kill count reaches 1000. This gives more time for improvements from CL to become apparent.

As with the experiments in the previous sections, only one of the three algorithms uses CL because if the algorithm cannot improve its performance when combined with two static models it is unlikely that it will do better when combined with other CL algorithms.

5.4.1 Continuous Learning Speed Algorithm

Based on the results shown in Figure 5.3, the three combinations with the highest rates of improvement are; JRip-OneR-DecisionTable(Even), RandomTree-OneR-DecisionTable(Even), and PART-OneR-DecisionTable(Even) (speed-shoot-rotation, best to worst). To determine the performance increase that can be gained these three combinations are tested for in-game performance until the total kill count reaches 1000.

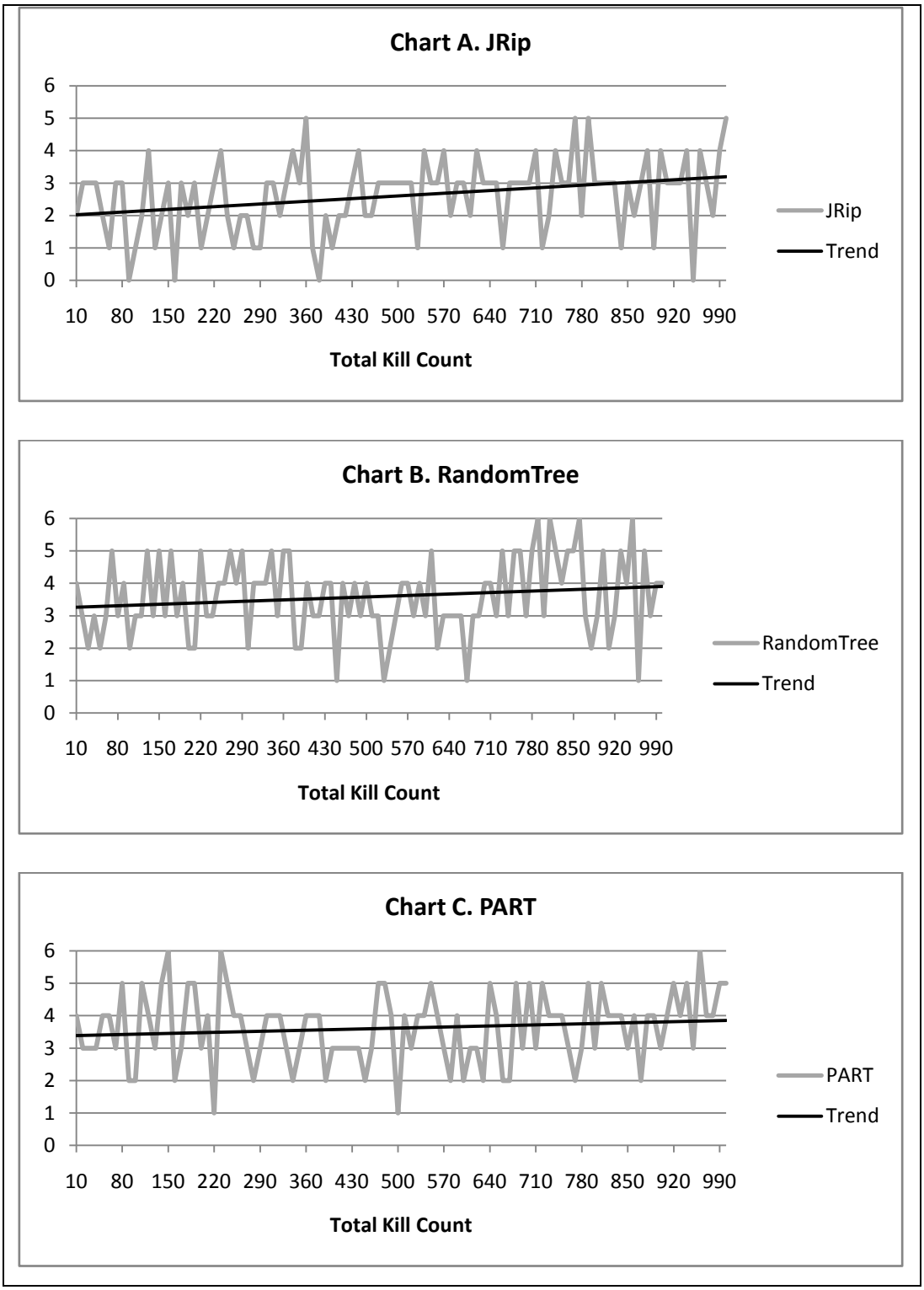

Figure 5.8 Points Scored Using CL to Control Speed (Long Duration)

Figure 5.8 shows the performance when CL is used to control speed. All the combinations use OneR to control shooting and DecisionTable(Even) to control rotation (static models the same as those described in Section 4.4.4).

Chart B in Figure 5.8 shows the combination using RandomTree has the best performance of the combinations tested, managing to score six points against robot-pilot several times towards the end of the experiment. All three combinations show an improvement over time, though none of them are able to match the performance of robot-pilot during testing.

Figure 5.9 shows the slopes of the trend lines in the charts in Figure 5.8 for easier comparison. Figure 5.9 shows the performance increase during long duration testing is far lower than it is in the short duration testing discussed in Section 5.3. This may indicate that in a CL system the performance increase drops off as the duration increases, however the rotation algorithm may not exhibit this effect.

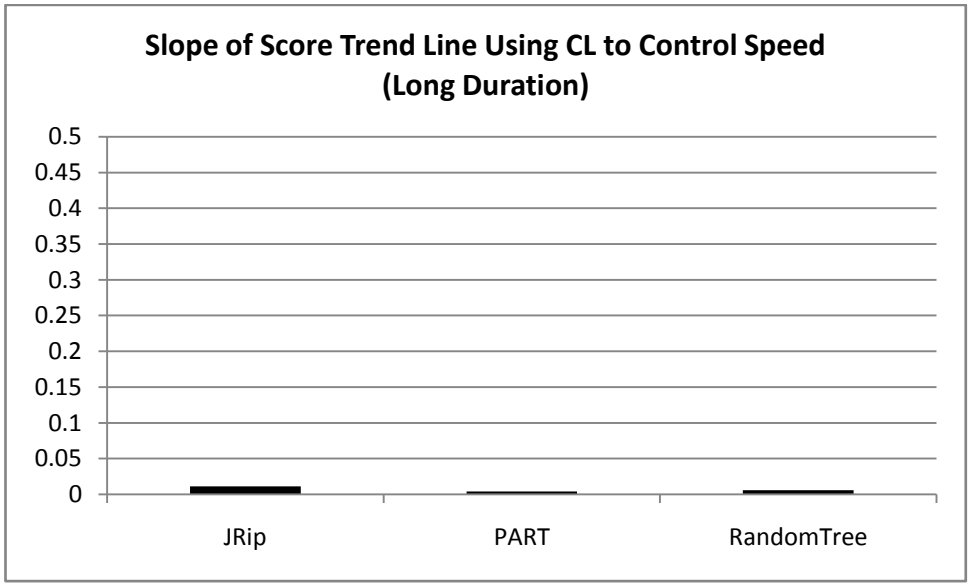

Figure 5.9 Slope of Trend Line Using CL to Control Speed (Long Duration)

5.4.2 Continuous Rotation Learning Algorithm

The results in Figure 5.7 show the combinations with the highest rates of improvement that use CL on the rotation algorithm are; JRip-OneR-DecisionTable(Even), RandomForest-OneR-DecisionTable(Even), PART-OneR-DecisionTable(Even) (speed-shoot-rotation, best to worst). These combinations are tested for in-game performance until the total kill count reaches 1000 to determine the performance increase.

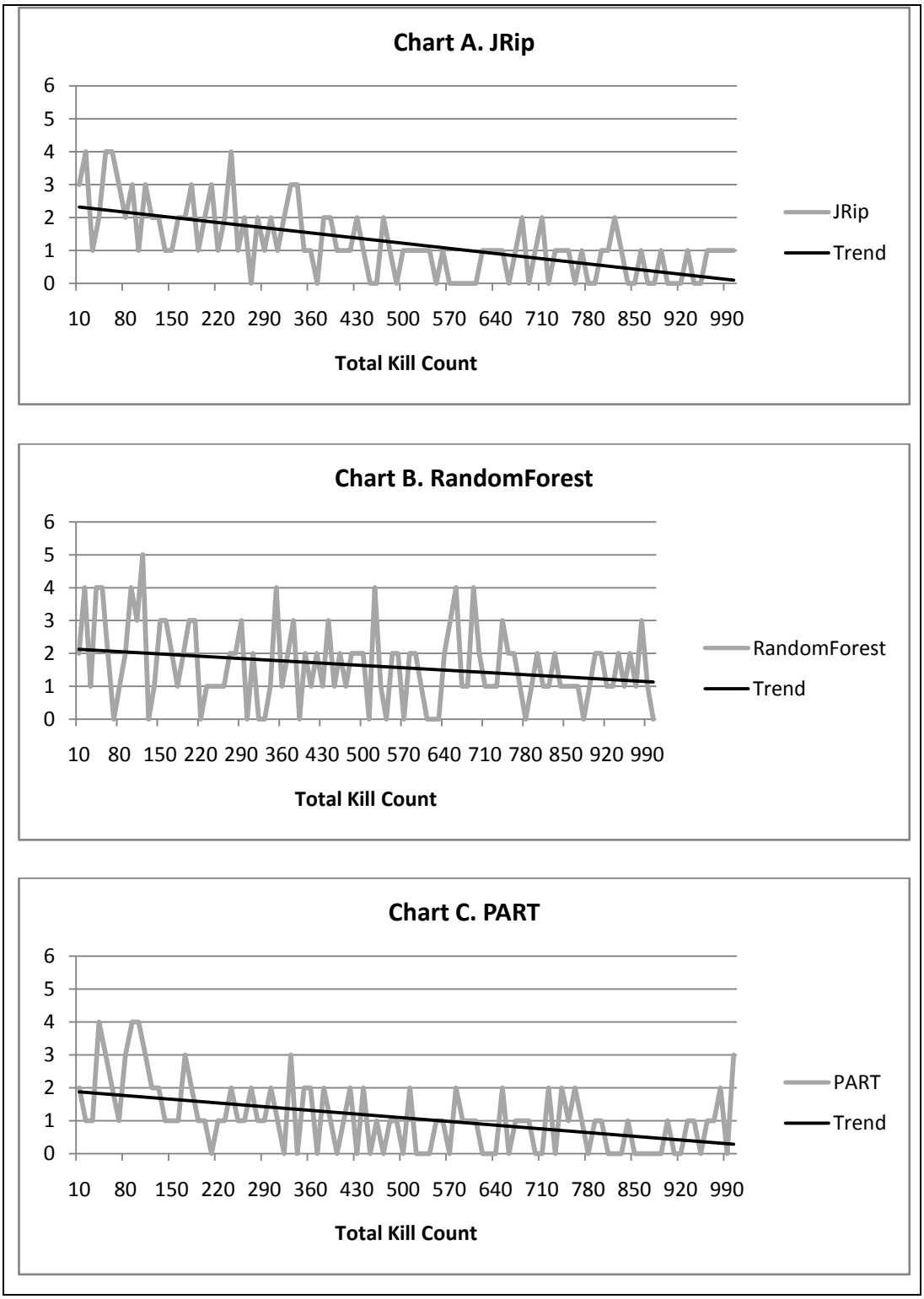

Figure 5.10 Points Scored by Using CL to Control Rotation (Long Duration)

Figure 5.10 shows the performance when CL is used to control rotation. All three combinations use DecisionTable(Even) to control rotation and OneR to control shooting. The performance of all three combinations is quite low, with the combination that uses RandomForest (shown in Chart B) the only one to score more than five points in any 10-kill block, but this happens only once.

For easier comparison the slopes of the trend lines of the three graphs shown in Figure 5.10 are reproduced in Figure 5.11. This clearly shows that all three combinations using CL on the rotation algorithm (DecisionTable(Even)) have a decrease in performance as the duration of the test continues.

This may be because the algorithm only learns from its own predictions, meaning that if the algorithm performs sufficiently it will eventually converge and the performance increase drops (as in Section 5.4.1), or if the algorithm does not perform sufficiently the accuracy of the model decays and ultimately results in a decline in performance over time (as in this section).

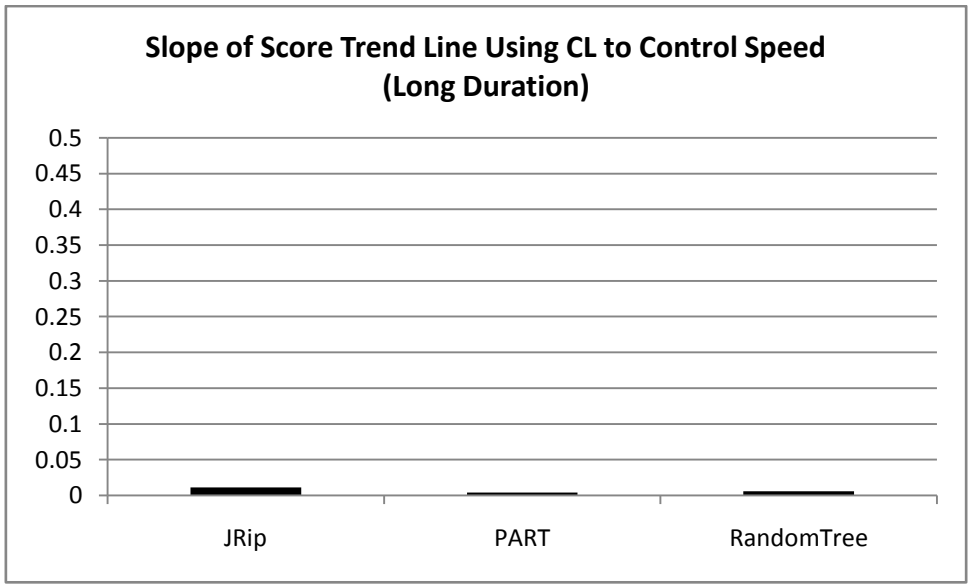

Figure 5.11 Slope of Trend Line Using CL to Control Speed (Long Duration)

5.5 Chapter Summary

This chapter describes experiments to improve the in-game performance achieved in Chapter 4 by using continuous learning on one of the ML algorithms. It is shown that in-game performance can be increased using continuous learning during short duration tests.

When used in long duration tests the rate of improvement observed for short tests is not maintained, dropping close to zero or becoming negative. This may indicate that an algorithm with sufficient performance converges, resulting in a ‘plateau’ in the rate of improvement, while the performance of an algorithm that does not perform sufficiently decreases over time. It is possible that the results in this chapter could be improved with further experimentation, but even with the short duration tests the performance gain is not sufficient to out-perform or even match robot-pilot.

This indicates that continuous learning can be used during short tests to improve performance of an ML algorithm, but over time the rate of improvement drops. Given the results shown in the long duration tests it is unlikely that the performance could be improved sufficiently to match robot-pilot so no further experimentation is conducted using CL.

6 Reinforcement Learning

This chapter describes experiments to determine if a reinforcement learning (RL) agent can learn to control a tank in BZFlag. Two reinforcement learning frameworks are used; Connectionist and PIQLE. These two frameworks are chosen because both are written in Java and as such can easily be integrated into the WEKA-Server program described in Section 3.3.3.

Section 6.1 describes Connectionist and outlines configurations used and results observed. Similarly Section 6.2 describes PIQLE, configurations used and results observed. Section 6.3 is a brief summary of this chapter.

6.1 Connectionist

Recall from Section 2.5.3 that Connectionist is a Q-Learning framework which uses a neural network to implement learning and dictate the actions of an agent. Because values are used as inputs to a neural network they do not have to be discretized.

Available actions are a combination of speed (forward, stop, backward), rotation (left, straight, right), and shooting (shoot, hold), based on the class values discussed in Section 4.2.1. This gives a total of 18 possible actions (3 x 3 x 2 = 18). For example, Forward-Straight-Shoot and Backward-Right-Hold are two possible actions. This gives the agent the same degree of control that a human user has when using a keyboard to control a tank. The list of actions is set at the start of the experiment and all 18 actions are assumed to be available at every time-step.

Connectionist works in two phases. In the first phase it receives all input values (sensors in Figure 2.2), as well as the reward value from the actions previously taken which is used to adjust the neural network. In the second phase the output of the neural network is produced and the agent carries out the specified action.

The ‘brain’ in Connectionist has four parameters that are set at the start of the experiment; Alpha, Gamma, Lambda, and Random Actions (RA). Alpha is the ‘learning rate’ or ‘step size’ of the algorithm. The learning rate is how much effect each new update has on previously learned values. The alpha parameter can be set in the range [0.0 to +1.0]. If the alpha is set too high the algorithm may not converge because of the large adjustments made at each step, while if it is set too low the agent may learn very slowly (or not at all if the value is 0.0).

Gamma is the ‘discount’ of the expected reward for future actions and can be set in the range [0.0 to +1.0]. A value close to zero makes the agent more ‘myopic’, focusing on immediate reward values, while a value close to one makes the agent more ‘far-sighted’, focusing on an expected reward in the future [Sutton and Barto, 1998].

Lambda is the eligibility trace forget rate and can be set in the range [0.0 to +1.0]. This causes actions taken closer to receiving a reward to be rewarded (or punished) more strongly than actions taken earlier.

RA is the probability of selecting a random action rather than the best known action. This can be set anywhere from 0% to 100%, but typically a low value (5% to 10%) is used to ensure the agent continues to explore the state space without sacrificing too much ‘exploit’ behaviour.

6.1.1 Initial Configuration

Name	Description	Conn.	PIQLE
MyVelocity (X,Y)	Velocity of the agent's tank along the axis.	X	
EnemyVelocity (X,Y)	Velocity of the opponent's tank along the axis.	X	
RelativePosition (X, Y)	Position of the opponent's tank on the axis, relative to the agent's tank.	X	X
EnemyDistance	Straight-line distance from the centre of the agent's tank to the centre of the opponent's tank.	X	
AngleDifference	Difference between the current rotation of the agent's tank, and the rotation that would point the agent's tank straight at the opponent's tank (i.e. How far the agent's tank must rotate to be facing the opponent tank).	X	X
ShotRelative (X,Y)	Position of the opponent's projectile on the axis, relative to the player's tank (zero if the opponent does not have an active shot).	X	
ShotVelocity (X, Y)	Velocity of the opponent's projectile along the axis (zero if the opponent does not have an active shot).	X	
ShotDistance	Straight-line distance to the opponent's projectile (zero if the opponent does not have an active shot).	X	
MySpeed	Current speed of the player's tank in the virtual world.	X	
MyRotation	Current orientation of the player's tank in the virtual world.	X	
FiringStatus	Integer value, tank can only fire when value is 1 (meaning 'ready').	X	X

Table 6.1 Attributes used for Reinforcement Learning

The world configuration is the same as that used in Section 4.1, with a size of 200 (400x400 plane) and no obstacles. Table 6.1 shows the inputs that are given to the Connectionist agent at each time-step (marked with an ‘X’ in the Conn. column). The values and ranges are the same as those described in Section 4.2.1. The only change is that missing values are represented with 0.0 because Connectionist cannot handle missing inputs.

The initial parameters for the Connectionist ‘brain’ are; alpha 0.5, gamma 0.9, lambda 0.2, RA 10%. These values are selected as a starting point because they are the used in one of the demo programs provided with Connectionist called ‘wanderbot’. The reward scheme used is +1 for killing the opponent and -1 for being killed. All other times the reward is 0.

6.1.2 Results

Figure 6.1 shows the score achieved by the tank controlled by Connectionist every 10 kills up to 15,000 kills. The black line is a moving average with a sliding window of 50 data points (i.e. 50 10-kill blocks or 500 kills). The moving average shows some variance in the performance over time with no sign of a steady increase. The moving average also shows the performance reaches a maximum at around 3200 kills, and gradually decreases after 11,000 kills possibly indicating the agent is starting to explore a poor area of the search space.

Observation of game-play shows the Connectionist tank fires as often as possible (immediately after each reload), regardless of the opponent's position. This is obviously detrimental to performance but the reward scheme described in Section 6.1.1 does not discourage such behaviour.

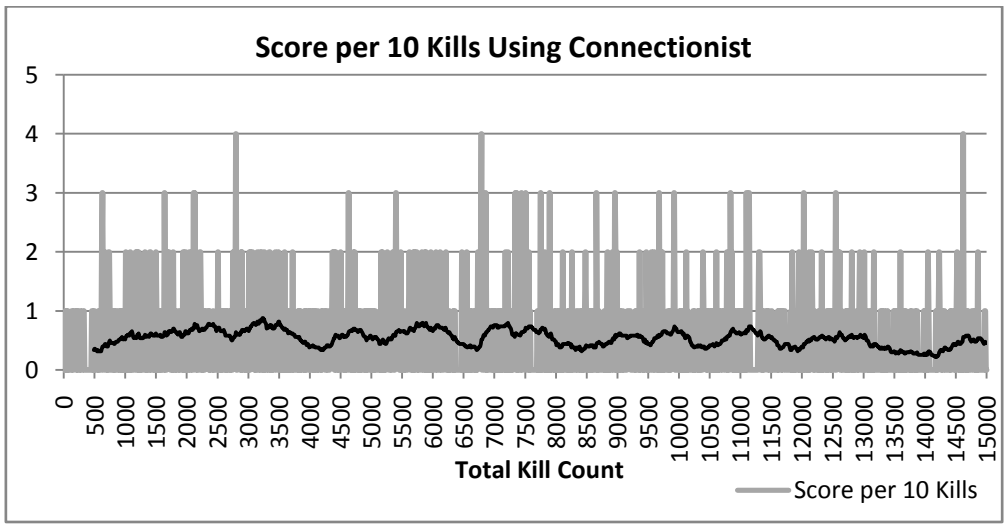

Figure 6.1 Score per 10 Kills Using Connectionist

6.1.3 Altered Configuration

The reward scheme is altered to 'punish' reloading time to deter the agent from firing as often as possible. The new reward scheme is +1 for killing the opponent, -1 for being killed, -0.1 for all states when the tank is reloading, and 0 in all other states.

The parameters are also altered to help stabilize learning. Alpha (learning rate) is reduced to 0.1 to reduce fluctuations in performance, lambda is increased to 1.0 to make the Connectionist tank as far-sighted as possible (given the large number of time-steps needed to kill the opponent), and RA is reduced to 5% to also help reduce fluctuations in performance. Gamma is left at 0.9.

6.1.4 Results

Due to time constraints the altered configuration is only run until the total kill count reaches 4000. Figure 6.2 shows the performance using the altered configuration is much worse than the performance using the initial configuration. The moving average appears to increase slightly over time, but without extending the duration of the experiment it is difficult to determine the long term trend.

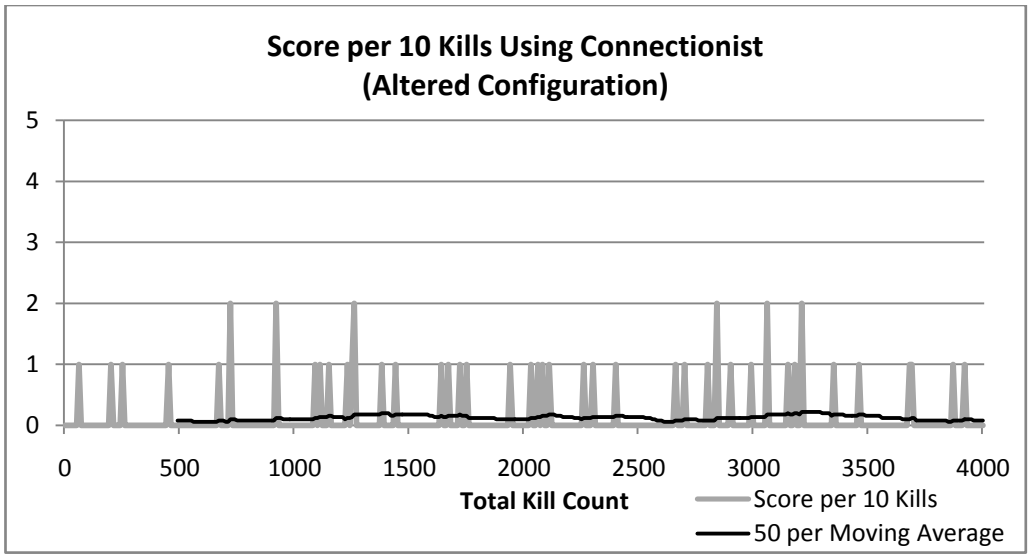

Figure 6.2 Score per 10 Kills Using Connectionist (Altered Configuration)

However, it seems unlikely the performance would have a sudden increase if the duration was extended, thus it would take a large number of episodes before the performance would match that in Figure 6.1.

Observation of the game-play shows the tank does not fire as often as it does when using the configuration described in Section 6.1.1. The poor performance seems to indicate a lack of ‘lucky’ shots that occur when the tank fires as often as possible, rather than a drop in ‘intelligent’ performance.

6.1.5 Backups

Backups of the neural network’s weights (described in Section 2.5.3) are introduced in an attempt to improve performance. This is done by comparing the score achieved after each 100 kills. If the score is higher than, or equal to, the maximum score achieved, the maximum value is adjusted and the current weights are saved, otherwise the last saved weights are restored.

6.1.6 Results

Figure 6.3 shows performance when backups are used. Performance clearly improves over that shown in Figure 6.2, even during the first 1000 kills. However, performance is not as good as that in Figure 6.1, with the agent never scoring more than three points in any 10-kill block. The moving average shows less variance than that in Figure 6.1. This may indicate a more ‘stable’ performance, possibly indicating more intelligent game-play rather than ‘lucky’ shots.

The moving average does not show a steady increase in performance, indicating it would take an extremely large number of episodes before the performance would approach that of robot-pilot, let alone a human opponent.

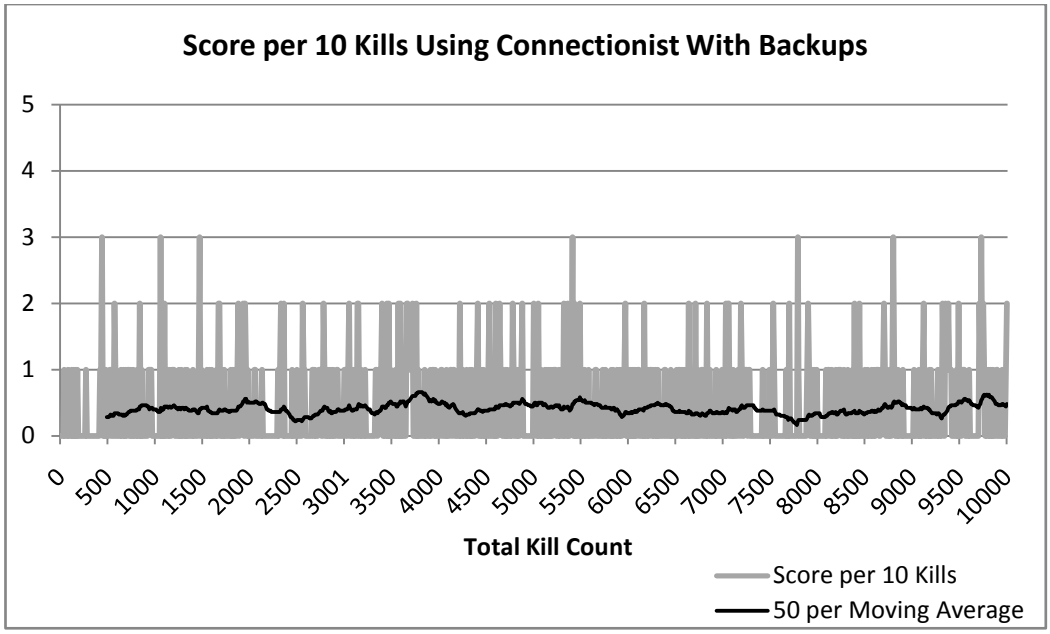

Figure 6.3 Score per 10 Kills Using Connectionist with Backups

6.1.7 Remarks

The initial configuration used with Connectionist produced the highest scores, but also has the largest variance. The high scores may indicate exploitation of robot-pilot's behaviour because, whilst a human player could do very well against an opponent that fires as often as possible (regardless of the human player's position), robot-pilot has relatively simple dodging behaviour and tends to head towards the opponent at all times, thus making it more susceptible to 'lucky' shots.

This 'lucky shot' exploitation behaviour could be investigated further, but the Connectionist agent performs far worse than robot-pilot so it would require an extremely large number of episodes to match the performance of robot-pilot. Time constraints make such experiments unfeasible during this study so this is left as a possible topic for future work.

The performance of a Connectionist agent in BZFlag has two main limitations. The first is that all actions are available for the agent at all times. This means the

agent cannot take into account times during game-play when shooting is not available and alter its behaviour accordingly. The other limitation is the Connectionist 'brain' is designed for environments with instant (or very quick) reward values, but BZFlag can have very long episodes (in terms of time-steps) before either tank dies, making it difficult to correctly assign reward values to the appropriate actions.

6.2 PIQLE

PIQLE is a Q-Learning framework that uses state-action pairs to determine the action for an agent to take. The state-action pairs are stored in a hash table to reduce memory requirements. State-action pairs need to match future occurrences exactly and, because of the large number of states possible in complex environments, can require that attributes be discretized. A more detailed description of PIQLE is given in Section 2.5.2.

PIQLE allows for available actions to be determined for each world state. Thus there is less need to penalize the agent for firing inappropriately so the reward mechanism is the same as that used in Section 6.1.1 (+1 for kill, -1 for being killed).

PIQLE also uses the notion of 'terminal' states. This allows for long episodes to be terminated only when one of the tanks is killed, thus state-action pairs that lead to a reward should be correctly rewarded. Note that because rewards are only received in 'terminal' states the reward will never be 0 like the Connectionist reward scheme.

6.2.1 Initial Configuration

The world configuration is the same as that described in Section 6.1.1. Only a small subsection of the possible attributes are used to represent the world state. This simplifies initial testing and (as noted in Section 6.2) the state-action pairs must match exactly, so reducing the number of variables results in faster convergence for the algorithm.

Table 6.1 shows the values used to represent world state for the PIQLE agent (marked with an 'X' in the PIQLE column). This is a much smaller set of attributes than those used previously but is sufficient to distinguish world states with minimal memory requirements.

The *RelativePosition* attributes are rounded off to the nearest 10, giving 81 possible values using the world configuration described in Section 4.1 (size 200, no obstacles). The *AngleDifference* attribute is rounded to the nearest 0.5, giving a range of 12 values (0 to +6.0). *FiringStatus* is an integer value with only three possible values so is left unchanged.

This rounding off (or discretization) is used as an initial starting point for tests to limit the time and memory requirements but may limit the agent's maximum performance, for instance *AngleDifference* is only accurate to 0.5 radians (approximately 29 degrees) which may limit the ability of the agent to target the opponent.

6.2.2 Results

Figure 6.4 shows the performance of the PIQLE agent using the configuration described in Section 6.2.1. The experiment is stopped after 2000 kills because of the agent's poor performance, scoring zero for most of the experiment.

The reason for this poor performance is most likely the large number of unique world states available. Using the configuration described in Section 6.2.1, the world can have over 23,000 unique states. Even assuming that the agent can never fire (i.e. always reloading) the agent has nine actions available, resulting in over two million unique state-action pairs. This is detrimental to reinforcement learning where the agent must visit each state-action pair numerous times in order for the algorithm to converge.

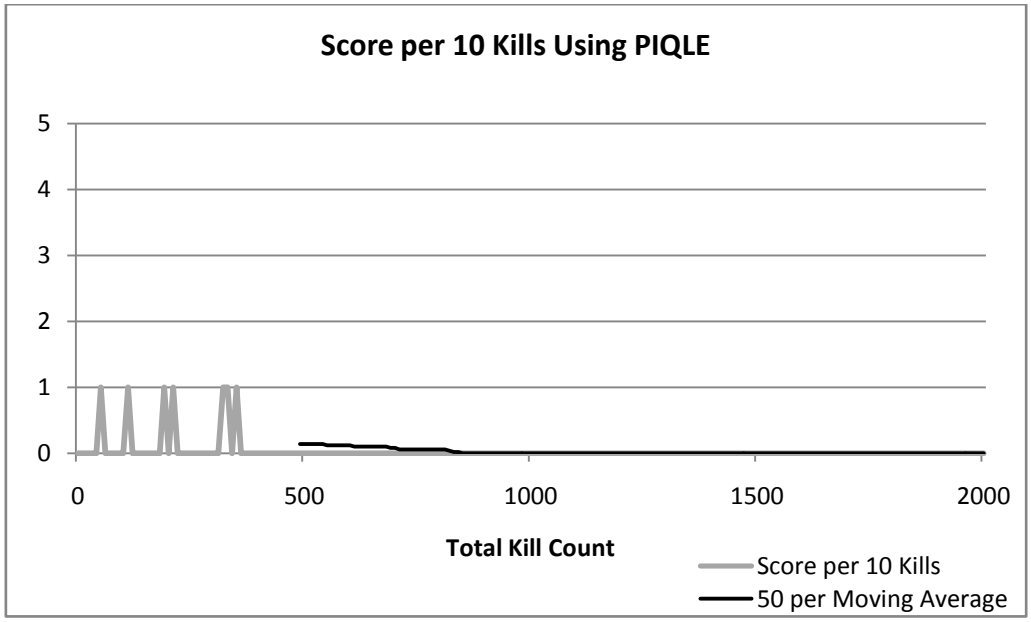

Figure 6.4 Score per 10 Kills Using PIQLE to Control Tank

6.2.3 Changes to Configuration

The configuration is changed to overcome the limitations observed in Section 6.2.2. The *RelativePosition* values shown in Table 6.1 are rounded off to the nearest 40. This gives a range of 21 possible values (-400 to +400) using the world configuration described in Section 4.1. This decrease in the number of possible values results in close to 16,000 unique world states and, with 18 possible actions, gives a maximum of close to 286,000 unique state-action pairs.

6.2.4 Results

Figure 6.5 shows in-game performance of the PIQLE agent with the configuration changes. The experiment is also run for a much longer duration to allow each state-action pair to be visited more often. The data points are now 100-kill blocks because of the large amount of data. The performance of the agent is clearly improved over the previous configuration shown in Figure 6.4.

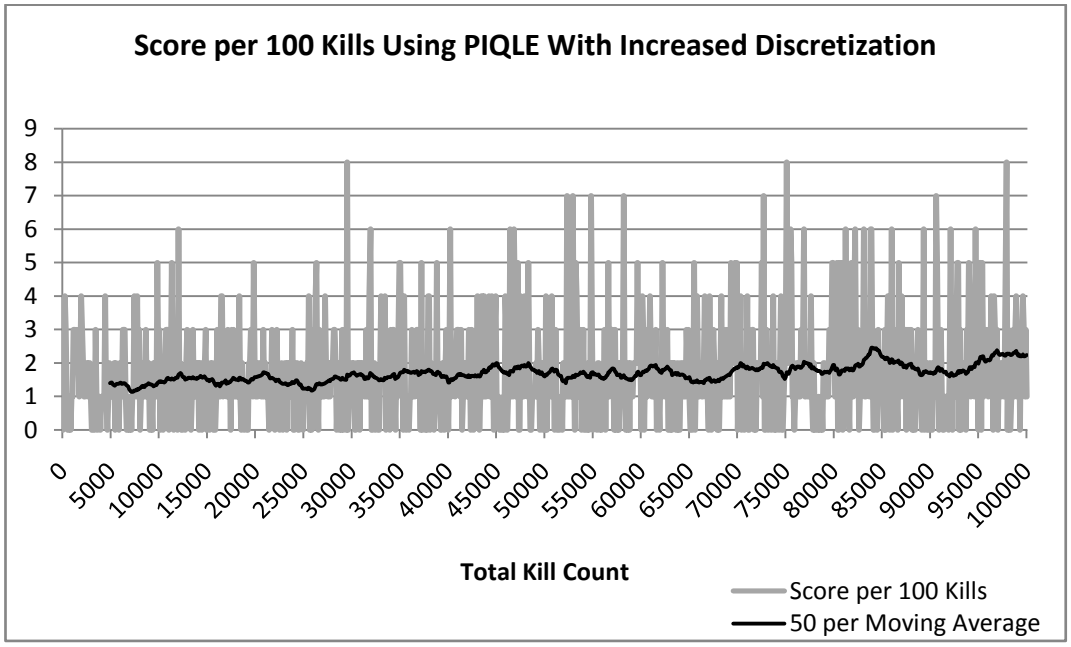

Figure 6.5 Score per 100 Kills Using PIQLE with Increased Discretization

Figure 6.5 also shows a gradual increase in performance over time, with the agent scoring six points once in the first 25,000 kills, but scoring six points several times (and even seven and eight points) during the last 25,000 kills. The moving average confirms this gradual increase, with a value between one and two up to around 80,000 kills, after which the value is often over two.

For easier comparison Figure 6.6 shows the performance of Connectionist from Figure 6.1 and the performance of PIQLE from Figure 6.5. The PIQLE values are taken from the last 15,000 kills in Figure 6.5. This shows that while the PIQLE agent may have a more reliable increase in performance, its overall performance is less than the Connectionist agent.

This seems to indicate that, if given sufficient runtime, the PIQLE agent's performance would improve faster than the Connectionist agent, but the Connectionist agent would have superior performance during the start of the test.

The performance of both agents is still far below that of robot-pilot however. Even with the best performance of the two agents (in Figure 6.6) the Connectionist agent scores a maximum of 13 points against robot-pilot, while PIQLE scores a maximum of only eight.

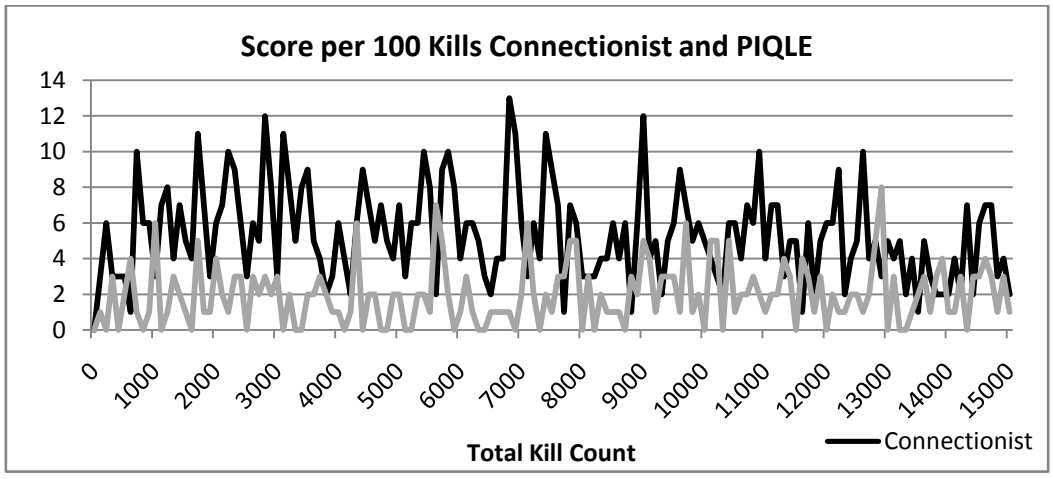

Figure 6.6 Score per 100 Kills Connectionist and PIQLE

6.3 Chapter Summary

Reinforcement learning shows the highest rate of performance increase of all the methods discussed in this report, but the initial performance is lower than methods that use supervised learning. Even assuming the rate of improvement remained constant it would take many hundreds of thousands, possibly millions, of episodes before the performance of the reinforcement agents would approach that of robot-pilot, let alone a human player.

This may be acceptable for training against a computerised opponent, but it calls into question whether a reinforcement learning agent that has learned to beat a human opponent could adapt fast enough when faced with a new opponent.

Unfortunately time constraints make it unfeasible for further experimentation with reinforcement learning during this study and it remains an open topic for future work.

7 Summary and Future Work

7.1 Summary

This thesis investigates the use of machine learning (ML) techniques to develop a game-AI system capable of adapting to a human opponent when given the same degree of control and information as the human player.

Numerous game-AI systems have been developed but academic game-AI systems often focus on relatively simple games, either 2D board games or simplified 3D games. Game-AI created by game developers is used in all manner of games, but the focus is not on learning but rather on providing the 'appearance of learning' to the human player. This somewhat unexplored area of AI systems in competitive 3D environments provides an interesting area of ML research.

Chapter 3 describes initial attempts to integrate an ML algorithm into BZFlag to control a tank and describes many issues that arise, such as the selection of data and algorithms to use during experiments. The online training approach is also described which highlights some constraints placed on an ML algorithm when it is used in a real-time 3D game and the offline training approach developed to help overcome those limitations.

Chapter 4 describes attempts to create an AI system using static prediction models. The single models provide similar performance to robot-pilot in many cases. When using two models the performance is similar to single models except combinations of rotation and speed controls which result in extremely poor performance. This is believed to be a result of the dependency of the algorithms on each other; that is, the outputs of one algorithm influence the inputs of the other algorithm. The datasets were altered to prevent this effect which did not have a detrimental effect on performance of the one and two algorithm combinations though due to time considerations the speed and rotation combinations were not tested again. Combinations using three algorithms to control the tank are tested, and the performance is not terrible but is less than that of robot-pilot and it was deemed unlikely that tuning the algorithm parameters or

the training dataset would provide a sufficient increase in performance to match a human player.

Chapter 5 describes experiments using continuous learning (CL) to improve the performance observed in Chapter 4. A brief test is conducted that confirms the static prediction models with the best performance are the most likely to improve performance when CL is used. The three-algorithm combinations with the best performance in Chapter 4 are tested again using CL on one of the algorithms for a short duration. The results are generally positive and several combinations show an improvement in performance over time. The experiments are then conducted again for a longer duration. CL on the speed algorithm shows much lower rates of improvement which may indicate there is a 'plateau' effect to performance gains from CL. The longer duration experiments using CL on the rotation algorithm however show a decline in performance over time, indicating there is a limit to how much improvement can be gained using CL.

Chapter 6 describes experiments using reinforcement learning (RL) to create an agent that is able to adapt to the opponent's game-play. Initial experiments using the neural-network approach with Connectionist show a decline in performance over time and observation of the game-play shows that the agent fires as often as possible (given reloading delays) regardless of the opponent's position. The reward function is altered to slightly penalize firing, which has the desired effect of reducing the number of shots fired by the agent. The performance with the altered reward function shows a small improvement over time, though the overall score is lower because of fewer 'lucky' shots. Backups of the neural network weights are used with the aim of improving the performance, and although these increase the overall score achieved the rate of improvement is unchanged. This poor improvement rate may be a result of limitations (such as all actions being available at all times) due to the implementation of the Connectionist framework, rather than the neural network approach itself but this remains a topic for future work.

PIQLE is experimented with to determine whether better performance can be achieved using the state-action pair RL method. The initial configuration has very poor performance. This shows a limitation of the state-action pair method; each

state-action pair must be visited several times in order for the algorithm to converge. This means a large number of possible states can require either a very long runtime or alternatively quite coarse discretization of attributes. The second experiment is conducted using both a longer duration and coarser discretization, with the results showing an increased rate of improvement but the performance is still far from being able to match robot-pilot, let alone a human player.

7.2 Conclusions

This research shows that ML techniques can be used in a modern game with a complex 3D environment without have a detrimental effect on game performance. Experiments in Chapter 4 show that static prediction models used in isolation can give similar or even better performance than a rule-based agent. It is difficult to create an agent using only static prediction models that can out-perform an intermediate human player, but with fine tuning the algorithm parameters and selection of data the static model approach may be useful as an alternative means for creating simple computer opponents.

Chapter 5 shows that CL can be used to improve the performance of a single static model, but this improvement is both small and short-lived, making it of little benefit for use in game-AI.

The experiments with RL in Chapter 6 show the most promise, with the highest rate of performance increase, but the experiments also highlight a limitation of RL; it requires thousands or millions of episodes in order for the learning to converge. Because of the demands this places on time it was not possible to investigate the use of RL more thoroughly, but this does raise questions about the usefulness of RL for game-AI; because RL ‘learns’ very slowly, it may require many games against a human opponent before it successfully adapts to the human’s strategy. Human players however are unlikely to play hundreds of identical games against a computer opponent (particularly because it performs poorly during initial learning), so it seems RL could be useful in producing an

agent that can play the game well, but may not be able to adapt to a new opponent quickly.

This also serves to highlight a fundamental difference between ML applied to traditional problems and ML applied to game-AI; traditional machine learning is generally geared towards finding the optimum solution. Reinforcement learning is guaranteed to explore the entire search space (provided sufficient runtime and random exploration is not stopped), and as such is proven to converge on the optimum solution. When playing against a human opponent however, a sub-optimal solution found quickly is far more preferable to an optimal solution found over a large number of iterations.

7.3 Future Work

There are many directions for further investigation mentioned in this thesis as they arise, some of which are observations from results that raise interesting questions but are outside the main goals of this thesis.

The requirement for fast adaptation by game-AI may limit the usefulness of RL, one possible solution to this that could be investigated is to use some form of genetic algorithm to produce the agent's behaviour. RL could be used to produce agents that perform well in different situations (such as attack or defence) and then the agent used against human opponents could be created by combining known 'good' behaviours.

Using a large number of players in 'death-match' style gameplay may speed up learning, since the number of episodes in a given amount of time will be increased. This could be particularly useful if genetic algorithms are used. That is, a 'knock out' type competition could be used, where each time an agent dies they are replaced by a new agent derived from the current two top agents.

Team games are another possible configuration for testing. All the experiments described in this report use one-on-one games but it may be possible that using

ML techniques with teams of agents may be able to produce simpler, emergent behaviour that performs better than single-player agents.

The separation of controls used in the experiments of Chapter 3 through Chapter 5 simplifies decision making for the separate algorithms but it may also hinder the creation of an overall strategy. An alternative approach is to use a hierarchical system of decision making. For instance, the agent first decides between the high-level concepts of ‘fight or flight’ based on the current world state. Once that decision is made, it leads to a lower-level decision, such as run or hide. Eventually the bottom level of the hierarchy has a direct action (such as move forward, turn left, or shoot). This approach is more complex to set up, but allows for simple decision making at each level while giving the agent an overall strategy to follow.

All the approaches used in this report and those described in this section give the agent the same level of control and information as the human player. It might be beneficial however, to have the agent start with an advantage over the opponent player (such as the ‘cheating’ game-AI described in Section 2.1.1). Once the agent can beat the opponent using an advantage the advantage can gradually be reduced and eventually removed. This could aid in learning because the agent may find a successful strategy more quickly when it is given an advantage over its opponent.

References

Butcher, C. and Griesemer, J. (2002) *The Illusion of Intelligence*. Talk given at the Game Developers Conference, March 23, 2002. Slides and notes available online at http://halo.bungie.org/misc/gdc.2002.haloai/talk.html (Retrieved on 15 December 2008).

Cleland, B.G. (2006) *Reinforcement Learning For Racecar Control.* Masters thesis. Department of Computer Science, University of Waikato.

Forbus, K.D., Mahoney J.V., Dill, K. (2002) *How Qualitative Spatial Reasoning Can Improve Strategy Game AIs.* IEEE Intelligent Systems, vol. 17, no. 4, pp. 25-30, Jul/Aug, 2002.

Kuzmin, V. (2002) *Connectionist Q-Learning in Robot Control Task.* In: *Scientific proceeding of Riga Technical University 5.serija. Datorzinatne. Information technology and management science, 10. sejums*. Riga, Latvia.

Manslow, J. (2002) in Rabin, S. (ed) *AI Game Programming Wisdom*. (2002) Charles River Media, Inc.

Ponsen, M., & Spronck, P. (2004). *Improving adaptive game AI with evolutionary learning.* In: Q. Mehdi, N. Gough, S. Natkin, and D. Al-Dabass (eds.): *Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004)* (pp. 389-396). University of Wolverhampton.

Sutton, R.S. and Barto, A.G. (1998) *Reinforcement Learning: An Introduction.* Bradford Book, The MIT Press, Massachusetts.

Tesauro, G. 2002. *Programming backgammon using self-teaching neural nets*. *Artif. Intell.* 134, 1-2 (Jan. 2002), 181-199.

Vega, V. S B., Bressan, S. (2003) *Continuous naive bayesian classifications*. In Proceedings of the International Conference on Asian Digital Libraries. 279--289.

Witten, I.H. and Frank, E. (2005) *Data Mining: Practical machine learning tools and techniques*. 2nd Edition, Morgan Kaufmann, San Francisco, 2005.