DEPARTMENT OF COMPUTER SCIENCE

Reinforcement Learning Framework for Robocode

Andrew Lord

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Wednesday 15th May, 2013, CS-MEng-2013

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Andrew Lord, Wednesday 15th May, 2013

Contents

List of Figures

List of Tables

List of Algorithms

Executive Summary

The game of Robocode involves creating virtual robots, which can fight each other within a virtual arena. Reinforcement Learning concerns an agent interacting with its environment so as to maximise the cumulative reward that it receives. This project involves the development of a robot that uses Reinforcement Learning to train multiple aspects of its behaviour during the game. The robot is included within a framework application, which allows people to create their own Reinforcement Learning controlled robots, play them in Robocode and collect results of their performance. The stochastic environment of Robocode brings with it many challenges for the Reinforcement Learning implementations to overcome, with custom representations being needed for the robot's states and actions.

Reinforcement Learning algorithms allow an effective Artificial Intelligence to be learned by the agent itself. The applications of Reinforcement Learning to Robocode have been explored in the past, however, it has mostly involved controlling one minor aspect of behaviour. This project goes beyond this, exploring the application of Reinforcement Learning in more complex ways. This involves performing a thorough investigation into different Reinforcement Learning algorithms, action selection methods, amounts of exploration and state-action representations.

The performance of the Reinforcement Learning robot was explored extensively against different opponents. When the robot uses both Reinforcement Learning movement and targeting it is able to achieve a win rate of 95% against the simple opponent `Corners`. This also includes significantly outperforming the most advanced movement strategy developed, Wave Surfing. When placed against the more advanced opponent, `LinearWave`, the dodging rate is comparable to that of Wave Surfing. This shows how RL has the potential to perform as well as the most sophisticated hand-coded strategies. It was also found that the strategy produced is a sufficient challenge for a human to defeat. The trained RL movement makes it difficult to hit, especially at range. This shows how RL can be used to produce an Artificial Intelligence for humans to play against, giving Reinforcement Learning applications within video games.

To apply Reinforcement Learning to a problem, currently it is performed manually, possibly with help from a library. This can often be very complex and time consuming. The framework application solves this situation, providing a Robocode environment with Reinforcement Learning capabilities already integrated. It is also designed to educate the user about Reinforcement Learning and how it can be applied to problems.

The main achievements of the project are as follows:

- 50 hours have been spent collecting and understanding material on Reinforcement Learning and previous work that combines it with Robocode.
- More than 6500 lines of source code have been written in Java. They comprise a robot for Robocode and a separate framework application with a GUI.
- The framework application allows experiments to be ran on Robocode, updating the graphs and statistics in real-time, whilst also averaging all of the collected results over any number of runs.
- The capabilities of the Robocode control API have been augmented to allow battles to be ran for a specified number of time-steps, a task the API was not designed for.
- Over 200 hours of experiments and tests have been conducted.
- A thorough investigation has been made, that includes exploring the effect of different Reinforcement Learning algorithms and action selection methods. The performance of the produced robot has been analysed against a variety of hand-coded opponents and against a human.

Supporting Technologies

- A framework developed by Dr Tim Kovacs was used as the basis for the Reinforcement Learning implementations in the developed robot.
  - Classes were used for Q-Learning, Sarsa, Monte Carlo methods, ε-greedy and Boltzmann Softmax. The Q-Table data structure was also used to store computed action-value estimates.
  - Extensive changes were made to add support for the dynamic learning rate. This involved re-writing the RL update rules, which is the central component of the Q-Learning, Sarsa and Monte Carlo methods classes.
  - Alterations were made to the Q-table data structure to store visit counts needed to compute the dynamic learning rate and support was added for outputting of the Q-table and saving it to a file.
  - Custom classes were written to handle the environment, actions and states, rather than using any provided.
  - The framework is available at: https://www.cs.bris.ac.uk/Teaching/Resources/COMSM0305/secure/rl/rl.library.html.
- Tutorials on the Robocode Wiki were used to create the hand-coded movement and targeting behaviours, due to them being widely accepted techniques and only added for comparison to Reinforcement Learning solution.
  - All the techniques were integrated into the robot design, which required changes to be made to them.
  - The Robocode Wiki is available at: http://robowiki.net/.
- To plot data within the framework application, JFreeChart was used.
  - All storing and manipulation of the data was developed separately. The library was used for plotting the final calculated points alone.
  - The library is available at: http://www.jfree.org/jfreechart/
- An XHML and CSS 2.1 renderer called Flying Saucer was used in the framework application to display help pages, due to the Java built-in HTML capabilities being too limited. The library is available at: https://code.google.com/p/flying-saucer/.

Notation and Acronyms

AI	:	Artificial Intelligence
ANN	:	Artificial Neural Network
API	:	Application Programming Interface
CS	:	Computer Science
CSV	:	Comma-Separated Values
DP	:	Dynamic Programming
GUI	:	Graphical User Interface
MC	:	Monte Carlo
ML	:	Machine Learning
RL	:	Reinforcement Learning
TD	:	Temporal-Difference
USC	:	University of Southern California
XML	:	Extensible Markup Language
$a \in A$	:	a is a member of the set A.
a_t	:	The action taken in time t
A	:	Set of all actions
$A(s)$	:	Set of actions that can be taken in state s
$\arg\max_x f(x)$	:	The set of values of x for which $f(x)$ achieves its largest value
$\max(a, b)$	:	Returns the highest out of a and b
$\max_x f(x)$	:	The largest value of $f(x)$
$P(x\|y)$	:	Conditional Probability of x being true given y is
$P^a_{ss'}$	:	The probability of transitioning to state s' by taking action a in state s
$Q(s, a)$	:	Action-value for taking action a in state s
r_t	:	The reward received at time t
$R^a_{ss'}$	:	The return given by transitioning to state s' by taking action a in state s
s_t	:	The state that the agent is in at time t
S	:	Set of all states
$V(s)$	:	State-value for being in state s
α	:	Learning rate
ε	:	Exploration rate (for ε-greedy action selection)
$\sum_a^b$	:	Summation from a to b
γ	:	Discount rate
$\pi(s)$	:	The action given by policy π when in state s
τ	:	Temperature (for Softmax action selection)

Acknowledgements

I would like to thank my project supervisor, Dr Tim Kovacs, for his support and guidance throughout the duration of the project and for providing his Reinforcement Learning library. Thank you also to Melanie Grewcock for the emotional support she provided during the project and to Thomas Pickering, Alex Sheppard and Philip Tattersall for providing user feedback on the Reinforcement Learning framework application. Finally, I would like to thank Flemming Larsen who provided invaluable advice concerning adapting the running of Robocode to the needs of the project.

Chapter 1

Contextual Background

This project concerns the use of Reinforcement Learning (RL) algorithms within the context of the game Robocode[33]. To describe the context of the project, this introductory chapter will cover some high-level background information, before motivating the project's aims and objectives.

1.1 Background

1.1.1 Robocode

Robocode is a Java game in which robot competitors are placed into a virtual battlefield to fight each other. It was started by Matthew A. Nelson in late 2000[32], before being completed and released by IBM AlphaWorks in July 2001. Robocode was later made open source at the start of 2005 and added for download on SourceForge[44]. Development continued on versions of Robocode to add features and fix bugs, before Flemming Larsen took over development from Matthew A. Nelson in 2006[11].

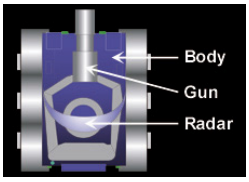


Figure 1.1: Anatomy of a Robocode robot, consisting of a main body, a radar and a turret. [38]

In Robocode, a robot consists of a main body, a radar and a turret[38], as seen in Figure 1.1. The main body provides movement capabilities, allowing the robot to navigate the battlefield, whilst hopefully avoiding the walls and enemy bullets. By using its radar, the robot can scan the battlefield for opponents, gathering information about those it detects. The turret can then be turned in order to aim and fire at an opponent with a chosen fire-power.

When it was developed, the primary focus of Robocode was on being a programming game, designed for people to create robots which can then compete in battle[32]. Through the process of creating a robot, developers learn about Java and game Artificial Intelligence (AI). With the event-driven approach a robot is designed, the programmer is able to control how the robot responds to situations which occur during the battle, such as scanning an enemy or driving into a wall[37]. This allows an AI to be formed, through a collection of rules to control the robot's behaviour, whilst the robot has no direct control over the game itself.

A wide variety of different robots have been developed and made available online[23]. This involvement from the community has resulted in many well documented techniques to become known, such as Wave Surfing[40] movement. Many robots which are developed therefore use variations of known techniques that are found in other robots.

For the technical details of Robocode, how it works and the different robot strategies available, see Section 2.1.

1.1.2 Machine Learning

The field of AI studies the "*design of intelligent agents*"[21], where an entity uses its intelligence to act within an environment, such as a character moving around a building. Alan Turing, an influential figure in AI, put forward the Turing Test in 1950[51], which demonstrated how learning was a crucial aspect of AI. The inclusion of learning brought about Machine Learning (ML) as a branch of AI, which aims to develop algorithms that can improve through experience. This idea was formalised by Mitchell[16, p.2]:

> **Definition 1.** *"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."*

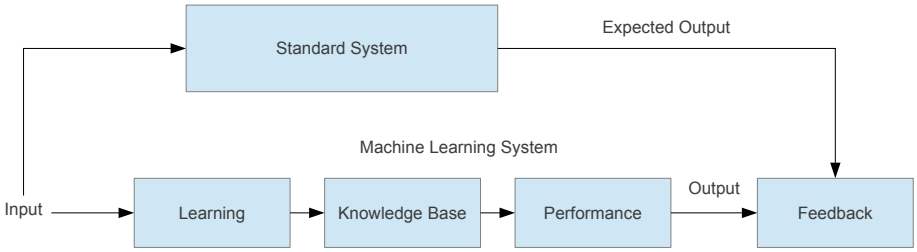


Figure 1.2: Comparison between an ML system and a standard system. ML involves learning, accumulated knowledge and performance based on the knowledge.

ML is constructed of a series of phases, which are shown in Figure 1.2 to be: learning, knowledge, performance and feedback. The algorithm first undergoes some form of learning process with the aim of gaining new information. A main knowledge base is kept, which is consulted and added to, maintaining a store of all the information known by the algorithm. This accumulated knowledge can then be used to perform actions, producing output and feeding into the evaluation and feedback.

To allow a robot in Robocode to improve from fighting an opponent, it is possible for an ML algorithm to be applied. An example for learning movement would be defining task T as accurately dodging the enemy bullets, performance measure P as the dodging rate the robot manages to achieve and experience E as the firing pattern of the enemy. If the dodging rate P improves after experience E, then the robot has managed to learn how to move in a better way. ML is a type of algorithm of which there are many different approaches, such as Decision Trees, Genetic Programming and RL.

1.1.3 Reinforcement Learning

RL is an area of machine learning which is inspired by the study of reinforcement in behaviourist psychology and operant conditioning[46]; the idea that humans and other animals repeat behaviours that produce positive rewards and avoid those which produce negative rewards[42]. Sutton and Barto define RL as[49, Chapter 1.1]:

> **Definition 2.** *"learning what to do, how to map situations to actions, so as to maximize a numerical reward signal."*

An RL learning agent aims to take actions in an environment, such as a maze, to maximise the reward it receives. By following this process, the agent hopes to reach its goal, such as getting to the end of a

maze or defeating an opponent in a game. The inspiration of reinforcement can be seen in the way the agent takes preference to actions that produce the most reward.

It is possible to represent the world the agent is in as a series of states[49, Chapter 3.1], where each state contains a set of parameter values, such as the position and orientation of a robot. Each state is a unique combination of parameter values, meaning that every unique value that a parameter can take results in another possible state. By performing actions, the agent is able to transition between the states, with some of the actions being more desirable than others, based on the state that the agent is taken to. Through the environment responding with rewards, the agent can attempt to distinguish the highest quality actions, from those that it should avoid. The agent selects actions that maximise its cumulative reward[49, Chapter 3.2], in order to reach its goal.

When a problem is to be solved using RL, the state and action representations need to be defined. Choosing these representations carefully is crucial to the effectiveness of the algorithm, as a poorly designed representation could prevent the agent from learning sufficiently. It is important the total number of possible states is minimised by reducing the number of values each parameter can take. If there are too many states, the learning process will be much slower, as the agent will have more quality estimates to improve. To achieve this, techniques are used such as avoiding the inclusion of any redundant parameters and segmenting continuous parameters into discrete sets of values.

An example representation to learn how to move a robot, might involve the states containing the location of the robot and the actions indicating a direction to move in. The RL algorithm would aim to move in the best direction, based on where the robot is within the world.

1.2 Aims

This project is divided into two main components; a robot capable of learning its behaviour through the use of RL and a framework application that allows people to create their own RL robots. The aim is for the robot to learn both movement and targeting behaviours using RL, where each part has a separate RL implementation containing state and action representations and a collection of parameters.

To determine how best to apply RL to Robocode, extensive research will need to be undertaken into the wide range of different RL algorithms that are available. Therefore, research of general RL will be combined with work completed on using RL in the Robocode context (Chapter 2). Any knowledge gained from the research and background reading will be used to adapt the RL algorithms for the Robocode environment (Section 3.1.1). This will involve dealing with the non-deterministic nature of the game and the event-driven manner in which a robot is programmed.

Once RL has been fully integrated into the robot, state and action representations will need to be designed for each of the different learning tasks. This will require a thorough investigation into the possible variables to include and the effect they have on the quality and speed of learning. To ensure the best results are achieved, all the RL parameters will need to be optimised, exploiting known characteristics of the Robocode environment. Whilst selecting representations and parameter values attention will be given to the speed of the RL calculations due to the limited period of time a robot is given to perform its turn (Section 3.1.3.3 and Section 3.1.4.3).

RL has applications in many more areas beyond Robocode, so any conclusions drawn have wider implications than just learning how to design a robot. To discover more about RL and Robocode the developed robot will be used to experiment with different algorithmic set-ups (Section 4.1.2 and Section 4.1.3). By playing the robot against a variety of different opponents, the performance of the robot will be analysed and the extent to which RL can outperform hand-coded AI or humans will be observed (Section 4.1.8 and Section 4.1.9).

The framework application will have a Graphical User Interface (GUI), enabling users to design and create their own robots that have RL controlling their behaviour. Through a connection to Robocode, it will be possible to play the created robots in games against a chosen opponent. Whilst the game is running, the framework will collect and plot the results in real-time, allowing the robot's performance to be observed as it changes over time (Section 3.2). A user can then run experiments by altering the RL parameters and viewing the effect it has on performance. To support this running of games and collecting

results in real time, a method will need to be developed due to it not being inherently designed into the Robocode Application Programming Interface (API).

By the conclusion of the project, the intention is to have a fully working robot developed, that can learn movement and targeting behaviours. This robot could then be customised through the framework application to allow a range of different robots to be created and used for a range of experiments on the Robocode environment.

1.3 Motivation and Project Importance

AI is a constantly developing field, becoming increasingly present in consumer products such as the Apple iPhone[2] and the Microsoft Xbox Kinect[15]. Games have always played an important role in AI research and can be traced back as far as 1952, when a program to play draughts was developed[4]. The AI in early games was relatively simple, employing exhaustive searches of possible moves or using heuristics[4], due to more advanced techniques being either unknown or requiring complex programming that was not available at the time. This did not change significantly until the late 1970s when arcade games like Space Invaders[45] made the concept of AI opponents popular. As computational power grew, the use of ML within game AI became increasingly possible, meaning the exploration of RL within a game context is an important topic. Its use would enable much more complex game AI, that could be learnt either in advance or in real-time against the player.

Previous work has been performed in the past on using RL within Robocode, however, it focuses on applying one type of RL to a specific aspect of the robot's behaviour. Examples include Ashish Uthama[52], who used RL to learn which from a set of guns was the best to use against a given opponent, and Morten Gade et al[5] who used RL to learn which opponent was the best to target in a multi-opponent battle. This project goes beyond what has been done before with RL and Robocode, investigating the application of RL to learn both movement and targeting. Previous work has involved using RL to learn an aspect of behaviour, however, the effect of using multiple RL implementations to learn different behaviours simultaneously remains unexplored. This is an interesting area of research, due to the computational and complexity demands and because the RL implementations may have an unexpected effect on each other. By learning multiple behaviours through RL, the robot becomes more adaptive, but training both at the same time may have an impact on the learning performance.

Previous work on using RL with Robocode concerns using RL to produce robots with more effective control mechanisms, whereas this project focuses more on how RL applies in the Robocode environment to gain conclusions about RL and the game in general. AI often attempts to replicate or even outperform humans, therefore playing the RL robot against hand-coded AI or a human provides interesting research conclusions. It is always an impressive result whenever RL, a learning algorithm, can achieve better or comparable performance to either a handwritten AI or a human.

Software platforms are available for developing and experimenting with AI techniques, such as OpenNERO [8]. A platform such as this makes it possible to switch between different AI techniques to observe the effect it has on performance. However, this highlights the focus of OpenNERO, which is more on trying out different AI techniques, rather than experimenting with them from a scientific perspective. This would involve the averaging of results over a sequence of runs, then computing statistics and plotting graphs of performance. The RL framework is a novel concept within the area of RL software, which is designed with running these full experiments in mind. It allows customisation of the robots and for graphs of results to be produced from any experiments that are ran.

Currently, RL algorithms are applied to problems either manually or through an RL library. Both options have drawbacks, with implementing a full RL implementation being a very large task and it being difficult to find an RL library that has all the features which are needed[10]. When the RL algorithm is applied to Robocode, it has to fit into the event-driven structure of the robot and cannot be too time intensive. This means that if a researcher wishes to run RL experiments on Robocode, they would need to spend a large amount of time getting the programming environment set-up, rather than being able to just focus on the experiments themselves. The framework application proposed is therefore original and will provide a needed method for running RL experiments on Robocode.

Users of the framework application will be required to put consideration into how RL should be set-up, meaning the framework provides an extension to the learning focus Robocode was intentionally given.

The benefit to the user is that the task of developing a robot with RL is made much easier, allowing them to focus on its performance without needing full working knowledge of game AI techniques. With the desire to teach users about RL, the framework grants the user the ability to tweak all the RL parameters and supplies user friendly access to RL.

Dr Tim Kovacs, the supervisor of the project, has shown interest in using the framework, due to RL being a major component to the Learning in Autonomous Systems unit he teaches. An assignment could be constructed which uses the framework application. This would allow students to learn about RL in an exciting way, without needing to invest the large amount of time required to develop a full RL implementation. Research, such as that of McGovern et al[13], suggests that the inclusion of a game problem motivates students more and makes the course material more engaging. The framework application could therefore be used to teach students about RL, how to represent it and how to evaluate its performance.

1.4 Challenges and Significance

Many existing RL algorithms are well established, meaning it would not be worthwhile to develop a new RL algorithm. Instead suitable RL algorithms are found that can be applied effectively to Robocode, with potential algorithms being chosen based on their performance. Integration of the algorithms into the robot design inevitably involves alterations needing to be made, these are to handle the non-deterministic and event-driven environment.

To design the state and action representations, understanding of how to map a set of parameters representing the robot and its environment to a single state or action are needed. There is a state for each combination of the parameter values, meaning that the more values each parameter can take, the higher the total number of states. The presence of too many possible states results in a much slower learning process and usually means there is an abundance of redundant information being stored. To minimise the number of possible states, techniques such as the removal of redundant parameters and segmentation of the parameter values are needed. Choosing the best set of parameters to use and segmentation applied to each of them is a central challenge to the application of RL to a problem. A high quality representation, which uses a sufficient level of detail, allows the RL algorithm to learn at an adequately fast rate.

Being able to play created robots in Robocode directly is one of the core features of the framework application. The Robocode API allows games to be controlled from another Java class, however, the API does not provide access to the robots which are in the game. To train a robot for a fixed number of time-steps, the robot needs to be capable of ending the game as soon as the number of time-steps has been reached. The robot also needs to communicate with the framework application, to provide results so that they can be plotted in real-time. These tasks are outside the capabilities of the Robocode control API and so present a complex challenge to be overcome.

Evaluating the performance of the RL robot can be carried out using a selection of metrics, such as cumulative reward, dodging rate and hit rate, whilst playing against a variety of opponents. However, assessment of the framework application is more complicated, due to many aspects of it lacking numerical metrics to apply. The goals of the framework application entail being user friendly, educational and quicker to use than performing tasks manually, so user testing must be involved in some form. However, thought is needed into the best way to determine its effectiveness.

1.5 Objectives

The high-level objective for the project is to perform a thorough investigation into the application of RL to Robocode and to develop a framework application to allow created robots to be evaluated in Robocode. The investigation will cover areas of RL not previously covered and will therefore provide insight into how it performs in the Robocode environment.

The specific goals of the project are:

1. Research and understand the theory surrounding RL and identify how it can be applied best to the Robocode environment.
2. Implement a robot for Robocode that uses RL to control movement and targeting. The robot should be fully customisable, including alterable parameters and support for alternative hand-coded movement and targeting techniques.
3. Investigate how RL performs on Robocode.
   (a) How to best represent the state-action space.
   (b) Analyse different algorithm configurations.
   (c) Observe the effect of learning multiple behaviours simultaneously.
   (d) Evaluate the performance of the robot against a variety of opponents, including:
      i. Hand-coded opponents
      ii. Interactive opponent controlled by a human.
4. Develop a framework application for creating RL robots, playing them in games and producing statistics of their performance. It should be both user-friendly and educational.

Chapter 2

Technical Background

This chapter intends to provide a detailed technical basis for the project and the problem being investigated. The technical background of Robocode (Section 2.1) and Reinforcement Learning (Section 2.2) will be discussed, whilst also covering the benefits of teaching AI through the use of game problems (Section 2.3.2).

2.1 Robocode

2.1.1 Control

For controlling Robocode, either the GUI can be used or a Java control API. Each of the methods will be used for this project, the GUI for testing during development of the robot and the API for running games of Robocode from the framework application.

2.1.1.1 GUI

All aspects of Robocode can be utilised through the GUI, from running battles and configuring settings, to editing robots in a built-in editor.

A very simplistic editor is accessible through the GUI, where the source code for robots can be altered and compiled. There is also the option of loading pre-compiled robots into the game, which allows the source code to be developed elsewhere. Once the robot has been properly loaded in, battles can be set up, where chosen robots can be played against each other for a given number of rounds. This is performed through a create battle window. The view shown during the battle is seen in Figure 2.1, the robots can be seen moving around the battlefield and firing at each other, with an option to enable a graphical view of the robot's radar scanning arc. For the purpose of this project, the default battlefield size of 800×600 pixels will be used, however, the size can be changed when starting a battle.

Other controls that are available to the user through the GUI include a slider to control playback speed and the ability to open an output console for each robot. The slider does not provide robots more or less time per turn, but controls the playback speed, as if the game were a simulation.

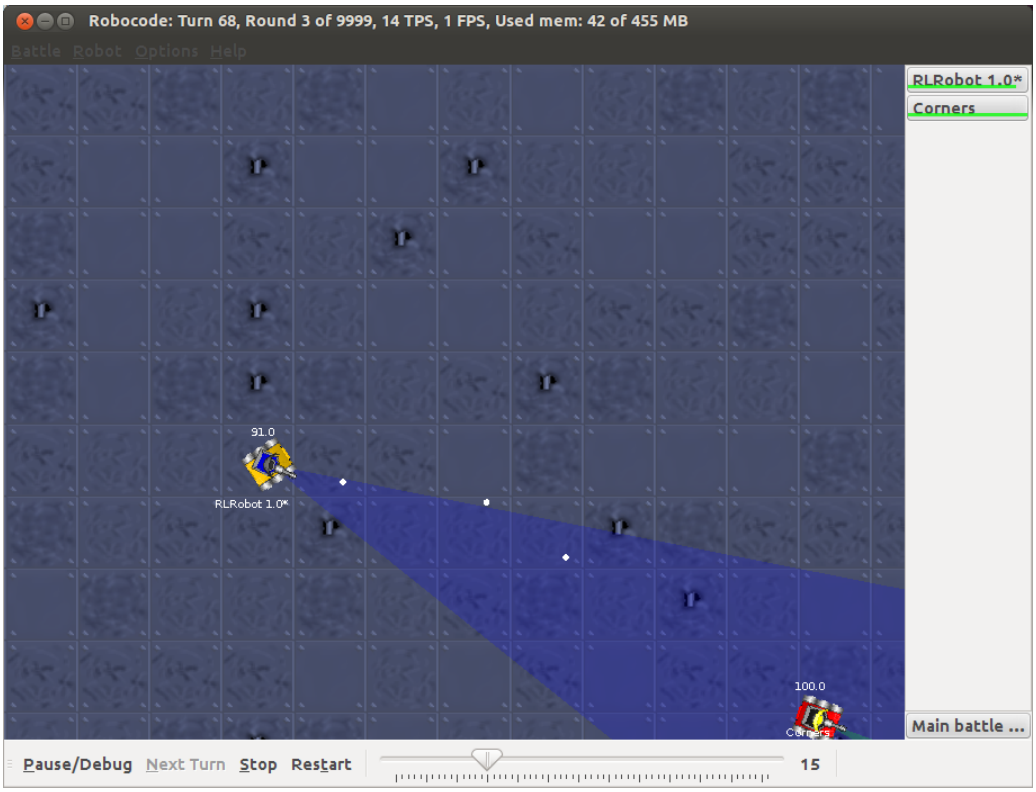


Figure 2.1: Screen capture of the Robocode GUI application showing a battle between the RL robot and the `Corners` robot. The blue cone represents the scanning arc of the radar and it can be seen that bullets have been fired.

2.1.1.2 API

Included in the Robocode API is a control package[34], which allows the game to be controlled from an external application. The Robocode engine is loaded from a local installation, which can then be provided with a specification for the battle, including the battlefield dimensions, the number of rounds to play for and the robots which will compete. Once the battle has been started, a custom battle listener can be defined which monitors the game as it progresses, collecting events fired by Robocode. This allows the external application to react to events, such as a round ending or the battle finishing.

2.1.2 Scoring

Each battle consists of a sequence of rounds, where each round starts with the competitors being placed onto the 2D battlefield at random locations. They begin with their energy replenished, ready to fight until only one robot survives. Rounds are all independent, meaning that performance in one round cannot affect that of later rounds. Upon completion of the game, the scores for each robot are displayed in order, with the winner being the robot with the highest total score. Robocode takes many factors into account on top of the number of opponents defeated, allowing all aspects of the robot's behaviour to contribute to the score it receives. The winner will therefore need to be effective at movement, targeting and scanning, rather than being able to dominate in one area.

The number of rounds each robot ranked 1st, 2nd or 3rd is also displayed alongside the other statistics. However, it provides no contribution to the score and is merely to show how long the robot survived.

Statistic	Description
Total Score	The sum of the other scores; determines the rank order each robot receives at the end of the battle.
Survival Score	When a robot dies in battle, each other robot that is still alive receives 50 points.
Last Survivor Bonus	The last robot alive at the end of a round receives 10 bonus points for each robot that died during the round.
Bullet Damage	For each point of damage done to an enemy using bullets, the shooter receives 1 point.
Bullet Damage Bonus	When a robot defeats an enemy using bullets, a bonus of 20% of all the damage done to that enemy is received.
Ram Damage	For each point of damage done to an enemy through ramming, 2 points are awarded.
Ram Damage Bonus	When a robot defeats another by ramming, a bonus of 30% of all the damage done to that enemy is received.

Table 2.1: Breakdown of the different statistics that contribute to a robot's score at the end of a battle in Robocode. A score from each category is taken to produce an overall score.
[39]

2.1.3 Distance and Time

Robocode is an episodic, turn-based game, where each unit of time (or tick) is referred to as a turn[36]. The GUI application separates frames from turns, meaning that if turns are completing faster than the frames can be drawn, then some frames can simply be skipped without affecting the performance of the game[35]. Each time the robot is given a turn, they have a limited amount of time in which to select a sequence of independent actions. All actions are then performed in order at the end of the turn. The time limit imposes an extra level of complexity for any controlling algorithm to run in time.

Distances are measured using a double precision representation of pixels[36]. This allows fractions of a pixel to be quantified, rather than being restricted to purely integer distances. A scaling factor is also applied to all distance calculations to allow the battle window to fit on the screen. Avoiding the use of pixels in distances provides much higher accuracy when moving a robot around the battlefield.

2.1.4 The Robot

The two different robot types available are the `Robot` or the `AdvancedRobot`, which allows many actions to be taken per turn and output to be performed through writing to files[24]. The features of the `AdvancedRobot` will be needed for an RL algorithm to function correctly, so will be used for this project.

Actions available to the main robot body are moving forwards and backwards or turning left and right[38]. By combining these into a sequence, the robot can navigate around the battlefield at will. On top of the body is mounted a turret, which the robot can rotate left and right in order to fire bullets at the enemy. The final part of the robot's anatomy is the radar, which is attached to the gun and can be turned left and right to scan for other robots. Each part can be turned independently of the others, just as tank would be controlled by a driver, a gunner and a scanner. If each part works together correctly the robot is able to detect enemies and use that information to fire upon them, whilst moving to dodge their bullets.

To react to situations which occur during a battle, the robot can receive events from the environment. These events include being hit by a bullet and scanning an enemy, where responding to the latter would allow the robot to turn the gun to face them and fire (Listing 2.1). The `AdvancedRobot` class allows custom events to be used[24], meaning that the robot can theoretically react to anything it wishes.

```
public void onScannedRobot(ScannedRobotEvent event) {
        //Deal with scanning a robot, e.g. target and fire.
}

public void onHitByBullet(HitByBulletEvent event) {
        //Deal with being hit by an enemy bullet.
}
```

Listing 2.1: Example AdvancedRobot events dealing with scanning an opponent and being hit by a bullet.

2.1.5 Movement

When the main body of the robot moves forward it accelerates at a rate of 1 unit/turn^2, until maximum velocity has been reached, which is 8 units/turn[36]. The body can decelerate faster, at 2 units/turn^2. Velocity and acceleration are handled implicitly by the engine, based on the distance a robot attempts to move in a turn[36].

Velocity is calculated using acceleration and time, as demonstrated in Equation 2.1.

$$velocity = acceleration \times time \tag{2.1}$$

Equation 2.2 shows how the velocity can then be used to compute the distance to move the robot.

$$distance = velocity \times time \tag{2.2}$$

Aside from the movement of the robot, its velocity also affects the speed at which other parts of the robot can rotate; the higher the velocity, the slower the rotation speed of the body, gun and radar[36].

To define the positions and orientations within the battlefield, the Cartesian Coordinate System is used[55], where the origin $(0, 0)$ is in the bottom left corner. Any angles are then measured clockwise[36], resulting in 90^o or $\pi/2$ radians corresponding to East. Therefore, a robot heading at 180^o that wishes to move at 200^o will need to turn by 20^o. Figure 2.2 illustrates how positions and orientations are measured in Robocode.

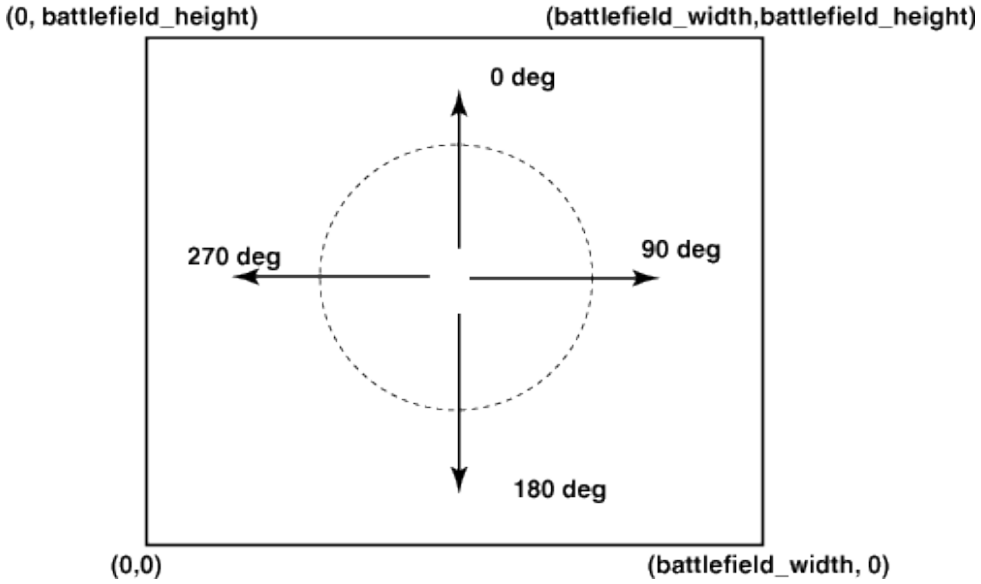


Figure 2.2: Cartesian coordinate system, which is used for positions and orientations within Robocode. Angles are measured clockwise from North, the x-axis runs to the right and the y-axis runs upwards. [36]

2.1.6 Energy and Gun Heat

Robots start with 100 energy, which can then be lost in different situations, leading to them being destroyed if their energy level is depleted. Situations in which energy can be lost are: collisions with a wall or another robot, being hit by an enemy bullet and when firing the gun (will be regained if the bullet hits). When firing the turret, the power at which to fire is chosen as a value between 0.1 and 3.0. Firing the gun then results in an amount of energy being lost equal to the fire power chosen[35]. Upon the bullet hitting the enemy, energy is recovered according to Equation 2.3, more than was lost by firing[36].

$$energyRegained = 3 \times firepower \tag{2.3}$$

An `AdvancedRobot` loses energy according to Equation 2.4 when it collides with a wall, the amount for which is dependent on the velocity at which the impact occurs[36].

$$energyLost = |velocity| \times 0.5 - 1 \tag{2.4}$$

However, all robots take a fixed amount of 0.6 damage when they drive into another robot. If a robot depletes its own energy through collisions to zero, it will become disabled and will be unable to move or fire. Full control capabilities are restored if some energy is regained through an airborne bullet hitting its target. However, if this does not happen quickly enough, the robot will be destroyed[35].

The damage dealt upon a bullet hitting an opponent is proportional to the power at which it was fired. To calculate the amount, Equation 2.5 is used, where the higher the fire-power, the higher the damage[36]. This means the process of selecting a firepower presents a trade-off between the risk of losing more energy and the reward of dealing greater damage.

$$damage = 4 \times bulletPower + 2 \times max(bulletPower - 1, 0) \tag{2.5}$$

To prevent robots from firing constantly, gun heat is generated, by Equation 2.6, proportionally to the power it was fired at[36]. In order to fire again, the gun must first cool back down to zero, which occurs at a rate of 0.1 heat per turn. Robots could start a round close together, so to provide a delay before bullets can be fired, all guns begin hot.

$$gunHeat = 1 + (bulletPower/5) \tag{2.6}$$

2.1.7 Robot Strategy

There are many different strategies that can be used to govern scanning, moving, targeting and firing, each of which will be discussed in turn.

2.1.7.1 Scanning

The process of scanning involves turning the radar, with the purpose of detecting opponent robots. Information collected can then be utilised to target them and potentially move to avoid them.

Spinning Radar is the most simplistic approach, where 360^o sweeps of the radar are repeatedly performed [28]. This can be quite effective in melee battles, which are those against multiple opponents, due to it being likely that there are opponents spread across the battlefield. However, in 1-vs-1 battles with just a single opponent, it is a very wasteful strategy, due to a majority of the time being spent scanning empty space[5, p.9].

To ensure that the radar keeps track of all the opponents, an alternative strategy is *Oldest Scanned*[28]. The robot stores when it last saw each of the opponents and will then switch the scanning direction of the radar to rotate it towards the opponent it has not seen for the longest amount of time. This has the advantage of avoiding scanning the whole battlefield each time, however, it merely alters the scanning direction and so in 1-vs-1 battles will still result in scanning many areas of empty battlefield. If the

technique attempted to scan the areas of the battlefield the enemy was likely to be within then it would be much less wasteful.

Locking onto an enemy is a very efficient way of utilising the radar in 1-vs-1 battles. Once *Turn Multiplier Lock*[30] has seen the enemy, it aims the radar to face their last known location. A scanning arc is then performed either side of this to ensure the enemy cannot escape the lock-on. This results in a very small area of the battlefield being scanned and allows the robot to always know the location of the enemy. However, it would likely result in poor performance in melee battles, where it is often advantageous to switch between targets[5, p.10].

2.1.7.2 Moving

Through a combination of *straight-line* and *circular* movement, a rudimentary movement strategy can be developed[5, p.11]. Highly predictable schemes such as these, will have poor performance against advanced robots, due to heading and velocity making it reasonably simple to predict their future positions. Making *random* movements could improve it, however, they make it harder to track the opponent and are more likely to result in the occurrence of wall collisions[5, p.11].

A more versatile approach is *Anti-Gravity*, which is based on an algorithm known as *potential field motion planning*[3]. It is a flexible strategy in which a series of gravity points are placed on the battlefield. If the location is deemed safe then the robot will be attracted towards it, however, if the location is dangerous then the robot will be repelled away instead[20]. Through using this technique the robot is able to avoid enemy robots and stay in safe areas. `TheArtOfWar` is an example of a robot which uses Anti-Gravity movement and was one of the best robots in Robocode for a period of time[54].

Wave Surfing has quickly become one of the most popular movement strategies, first utilised by a robot called `Shadow`[1] and then later by the current world champion `DrussGT`[43]. It involves predicting when bullets have been fired and then moving to avoid them. Due to robots being unable to detect bullets in the air, they must instead monitor enemies for a drop in their energy level, which predicts that a bullet has been fired. The bullet's trajectory is still unknown, so instead a wave is formed, which expands out from the location at which it was fired. Information taken from past bullets can be used by the robot to deduce the enemy's targeting technique and to then move to the safest point on each wave[40]. Wave Surfing is a very effective strategy of minimising damage, as it focuses on avoiding bullets explicitly.

2.1.7.3 Targeting

The most basic aiming approach is *Head-On Targeting*, where the gun is simply directed straight at the opponent and fired[26]. If the opponent is moving then this will rarely work very well, however, it in melee battles it can be quite effective.

Aiming of the gun needs to account for the distance between the robot and the opponent, to allow bullets to hit the enemy[5, p.12]. To achieve this, a predictive scheme is needed that can target where it believes the enemy will be once the bullet arrives. *Linear Targeting* makes the assumption that the opponent will continue moving at the same heading and with the same velocity, using this to aim the bullet for the target's future position[27]. *Circular Targeting* is a similar technique designed for enemies which move in a circular motion, assuming that they will continue with the same turn rate and velocity[25]. It is possible to use pattern matching to recognise the opponent's movement style and then use the appropriate targeting scheme to aim for their future positions[5, p.13].

Incorporating a form of ML is a natural evolution to predictive targeting techniques. *Neural Targeting* performs this using one or more Artificial Neural Networks (ANN)[29]. It has the advantage of learning dynamically, so is able to work against different enemy movement behaviours.

2.2 Reinforcement Learning

2.2.1 Concepts

RL places a learning agent within an environment with which it can interact. The agent selects actions to take, which the environment responds to by presenting a new situation and giving a reward[49, Chapter 3.1]. Formally, the agent has a *policy* that maps states of the environment to actions which can be taken in those states[53]. Let S represent the set of all states and A be the set of all actions. A policy is defined as[16, p.369]:

$$\pi : S \rightarrow A \tag{2.7}$$

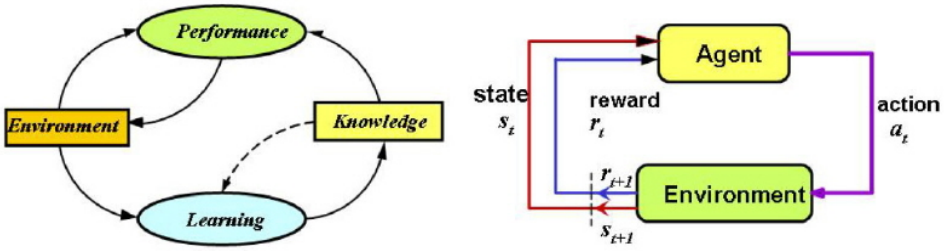


Figure 2.3: RL system model, showing the interaction with the environment. More specifically, RL consists of an agent taking actions and receiving back a new state and reward from the environment. [17]

Continual interaction occurs between the agent and the environment (Figure 2.3), meaning that at time-step t the agent will be in state s_t, where $s_t \in S$. Through action selection, the agent chooses an action a_t to take, where $a_t \in A(s_t)$ and $A(s_t)$ is the set of actions which can be taken at state s_t[49, Chapter 3.1] (Equation 2.8). When the policy π chooses an action to take:

$$\pi_t(s, a) \rightarrow P(a_t = a | s_t = s) \tag{2.8}$$

In the following time-step, $t+1$, the environment responds to the chosen action with a reward, $r_{t+1} \in \mathbb{R}$, and the agent moves into a new state, s_{t+1}[49, Chapter 3.1] (Figure 2.3). The new state that the agent moves into, s_{t+1}, is presented by the environment through the use of a transition function. This function is a mapping from a given state and action to a new state, where $\delta(s, a)$ produces the new state s_{t+1} when $s_t = s$ and $a_t = a$[16, p.370]. The transition function is written as[5, p.138]:

$$\delta : S \times A \rightarrow S \tag{2.9}$$

The reward function used by the environment is a mapping from a given state and action to a real value[5, p.138]:

$$R : S \times A \rightarrow \mathbb{R} \tag{2.10}$$

The objective of an RL algorithm is to use these rewards and seek to maximise the expected return, which is a function of the sequence of rewards[49, Chapter 3.3]. In order to decide how important future rewards are, the agent utilises a concept known as *discounting*. When the agent selects actions, it aims to maximise the sum of the *discounted rewards*[58, p.41], whose values are weighted based on how far into the future they are received. Discounting is controlled by the *discount rate* parameter, γ, where $0 \leq \gamma \leq 1$. When γ is 0 only immediate rewards are taken into account and as γ approaches 1, future rewards are increasingly considered. The discounted return is[49, Chapter 3.3]:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.11}$$

A value function represents the merit to the agent of being in a particular state, or taking a given action in a given state. Most RL algorithms attempt to estimate these value functions in order to determine which actions should be taken. To represent the value of simply being in a particular state, there is the *state-value function*, $V^\pi(s)$. This is the expected return, $E_\pi\{\}$, when starting at state s and following the policy π [49, Chapter 3.7]:

$$V^\pi(s) = E_\pi\left\{R_t|S_t = s\right\} = E_\pi\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t = s\right\} \tag{2.12}$$

Similarly, to represent the value of being in a particular state and taking a given action, there is the *action-value function*, $Q^\pi(s,a)$. This is the expected return when starting at state s, taking the action a and then following policy π[50, p.336]. The action-value function is defined as[49, Chapter 3.7]:

$$Q^\pi(s,a) = E_\pi\left\{R_t|S_t = s, a_t = a\right\} = E_\pi\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k+1}|s_t = s, a_t = a\right\} \tag{2.13}$$

By receiving rewards and maintaining an average for each state encountered, the action-value function can be estimated, which will eventually converge on each state's true value, $V^\pi(s)$. This leads to the ultimate aim of an RL algorithm, to find a policy which achieves the most reward over a large period of time, the *optimal policy* π^*[16, p.371]. There may be multiple optimal policies, however, they will all share the same *optimal state-value function*, $V^*(s)$ and *optimal action-value function*, $Q^*(s,a)$. An optimal policy is therefore[53, p.273]:

$$\pi^* \equiv \arg\max_\pi V^\pi(s) \qquad \forall s \in S \tag{2.14}$$

The optimal state-value function, $V^*(s)$, will return the maximum discounted cumulative reward when starting from any state and following the optimal policy[7, p.248]. This is written as[49, Chapter 3.8]:

$$V^*(s) = \max_\pi V^\pi(s) \qquad \forall s \in S \tag{2.15}$$

The optimal action-value function, $Q^*(s,a)$, will give the maximum discounted cumulative reward when starting from any state and taking an action from that state, then following the optimal policy[49, Chapter 3.8]:

$$Q^*(s,a) = \max_\pi Q^\pi(s,a) \qquad \forall s \in S, \quad \forall a \in A(s) \tag{2.16}$$

Unfortunately, computing the optimal value functions directly requires perfect knowledge of the environment's transition function, δ, and reward function, R[49, Chapter 3.9], both of which are not usually available. Even if they were obtainable, the computational power required to compute and utilise the information would often be too great, due to the amount of data that needs to be processed. RL attempts to overcome these problems through approximation of the value functions to discover the optimal policy.

2.2.2 Action Selection and Exploration

Action selection is the process of choosing an action, a, in a given state, s, from the available actions $A(s)$. One method is to simply select the action with the highest value, $Q(s,a)$, known as being greedy[5, p.141]:

$$a_t = \arg\max_a Q(s,a) \tag{2.17}$$

Utilising a purely greedy action selection approach, leads the agent to exploit the current $Q(s, a)$ estimates. The result of *exploitation* is that the estimates may converge to a local optimum, unable to improve[16, p.379]. This highlights a central problem in RL, that some state-action pairs may never occur, due to either the agent not visiting certain states or some actions not being performed. The problem arises if the policy is deterministic, as only one action in each state will be chosen[49, Chapter 5.2]. It is important to ensure a variety of actions have a chance of being selected, to allow a global optimum to be found[5, p.142]. Exploration allows the problem to be overcome, whereby different *non-greedy* actions are chosen, in order to explore the state-action space and gather new information[16, p.369]. There are multiple action selection methods that utilise *exploration*, including ε*-greedy* and *Softmax*.

2.2.2.1 ε-Greedy

With ε-greedy, for a majority of the time greedy actions are selected, however, with a small probability ε, a uniformly random action is taken instead. As the number of time-steps increases, the number of times each action is taken will tend towards ∞, allowing the $Q(s, a)$ values to converge to the optimal, $Q^*(s, a)$[49, Chapter 2.2]. Formally, a uniform random number, $0 \leq \xi \leq 1$, is consulted at each time-step, allowing the agent to select an action with a fixed probability, $0 \leq \varepsilon \leq 1$. The rest of the time the agent picks an action greedily based on the $Q(s, a)$ values. The ε-greedy policy can be defined as[50, p.337]:

$$\pi(s) = \begin{cases} \textit{uniform random action} \text{ from } A(s) & \text{if } \xi < \varepsilon \\ \arg\max_{a \in A(s)} Q(s, a) & \text{otherwise} \end{cases} \tag{2.18}$$

The advantage of ε-greedy over the purely greedy approach can be clearly seen. It allows a balance between exploration and exploitation to take place. The agent can exploit the knowledge it has gained about the quality of state-action pairs, but can then also learn new information by sometimes taking exploratory actions. However, the result is that, when exploring, the worst action is as likely to be chosen as the second best action and there are situations were choosing the worst actions may have negative consequences[49, Chapter 2.3].

2.2.2.2 Softmax

With Softmax, rather than switching between exploring and exploiting, each action instead has a probability of being picked, based on its current value estimate. The method will exploit, through the greedy action having the highest probability and then explore, through each other action having a non-zero probability, weighted by their value estimates. The probability of choosing an action is taken from a Boltzmann distribution[49, Chapter 2.3], where τ is a parameter called *temperature*. A policy using Boltzmann Softmax can be written as[50, p.337]:

$$\pi(a|s) = P(a_t = a|s_t = s) = \frac{e^{\frac{Q(s,a)}{\tau}}}{\sum_b e^{\frac{Q(s,b)}{\tau}}} \tag{2.19}$$

High temperature values make the probabilities for all actions almost identical, whereas low temperature values cause the probabilities to become more dispersed[50, p.338]. This means that as $\tau \to 0$, Softmax action selection becomes increasingly greedy.

2.2.2.3 Comparison

The relative performance of ε-greedy and Softmax will be different depending on the problem being solved. In practise ε-greedy can be easier to use, due to the ε parameter being quite simple to set based on intuition, compared to setting τ which requires knowledge of the exponential function[49, Chapter 2.3].

2.2.3 Algorithms

To solve a problem through RL there are many different methods that can be used, each with its own advantages and disadvantages. Possible methods can be easily divided into three classes, *Dynamic Programming*, *Monte Carlo methods* and *Temporal-Difference Learning*.

2.2.4 Dynamic Programming

Dynamic Programming (DP) is a classical mathematical approach to solving the problem, which uses a complete model of the environment to compute the optimal policy. For this reason it has limited use in RL applications, due to a complete model of the environment not usually being available. However, DP covers the theory that other types of RL are based upon. MC methods and TD use many of the same techniques as DP, so it will be discussed to allow them be understood.

2.2.4.1 Markov Decision Processes

Perfect knowledge of the environment is provided as a *Markov Decision Process* (MDP), a process that holds the *Markov Property*. Sutton and Barto define the Markov Property as[49, Chapter 3.5]:

Definition 3. *"A state signal that succeeds in retaining all relevant information"*

If a process possesses the Markov Property it means that the conditional probability of future states depends purely upon the present state and not on previous states[58, p.39], as it holds all the required information from those before it. In situations where there are a finite number of states and actions, the process becomes a *finite MDP*, a central concept to most RL techniques. For a finite MDP the probability of transitioning to each successor state, s', for any state, s, and action, a, is given by[49, Chapter 3.6]:

$$P^a_{ss'} = P(s_{t+1} = s' | s_t = s, a_t = a) \tag{2.20}$$

The expected return from taking an action to move from one state to another can then be determined through using perfect knowledge of the environment's reward function:

$$R^a_{ss'} = E\left\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\right\} \tag{2.21}$$

2.2.4.2 Value Iteration

To solve a problem, the optimal policy is needed, which DP obtains through storing approximations of the value functions, V^* or Q^*, and then iteratively improving them[41, p.6]. This process involves computing a value function, V^π, from an arbitrary policy, π, through Policy Evaluation and then using this to obtain an improved policy through Policy Improvement. DP performs this through the Bellman optimality equation for V^*, which shows that the value of a state when following the optimal policy equals the expected return for the best action from the state[49, Chapter 3.8]:

$$V^*(s) = \max_a E_{\pi^*}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\right\} \tag{2.22}$$

$$= \max_a E_{\pi^*}\left\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s, a_t = a\right\} \tag{2.23}$$

$$= \max_a E\left\{r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a\right\} \tag{2.24}$$

$$= \max_{a \in A(s)} \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^*(s')] \tag{2.25}$$

Policy Evaluation involves computing a new approximation for the optimal value function, based on the current policy[57, p.46], updating towards the Bellman optimality equation. There is a sequence of approximations of the value functions $\{V_k\}$ that as $k \to \infty$ will converge on V^π. $\pi(s,a)$ represents the probability of taking action a when in state s and following policy π. Policy Evaluation is defined as[49, Chapter 4.1]:

$$V_{k+1}(s) = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V_k(s')] \qquad \forall s \in S \tag{2.26}$$

To produce each approximation V_{k+1}, the value of each state s is updated to a new value. The old values of the successor states of s are combined with the associated expected rewards to form the new value[49, Chapter 4.1]. DP performs a *full backup*, where the values of all the successor states are used[7, p.249], compared to a *sample backup*, where just a sample successor state is taken into account.

Once the value function, V^π, has been determined for a policy π, then it is possible to search for a new better policy. Policy Improvement is a process whereby the value function of a previous policy allows the best (greedy) actions for each state to be selected to form a better policy[58, p.47]. Policy Improvement is defined as[49, Chapter 4.2]:

$$\pi'(s) = \arg\max_a E\{r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t = s, a_t = a\} \tag{2.27}$$

$$= \arg\max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^\pi(s')] \tag{2.28}$$

The optimal policy can be determined through successive steps of Policy Evaluation and Policy Improvement, as demonstrated by Figure 2.4.

$$\pi_0 \xrightarrow{\text{E}} V^{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} V^{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi^* \xrightarrow{\text{E}} V^*,$$

Figure 2.4: Policy Iteration, involving repeated steps of Policy Evaluation (E) and Policy Improvement (I). The process aims to obtain the optimal value function, shown as V^*.
[49, Chapter 4.3]

Policy Iteration involves doing a full Policy Evaluation each time, meaning it is an inefficient method for determining the optimal policy, requiring many loops through the set of states for each iteration. In an attempt to improve this *Value Iteration* was formed, where only one backup of each state is performed in the evaluation phase (Algorithm 2.1 [49, Chapter 4.4]).

Initialise V arbitrarily, e.g. $V(s) = 0, \forall s \in S^+$
repeat
  $\triangle \leftarrow 0$
  forall $s \in S$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V(s')]$
    $\triangle \leftarrow \max(\triangle, |v - V(s)|)$
  end
until $\triangle < \theta$ *(a small positive number)*
Output a deterministic policy, π, such that
  $\pi(s) = \arg\max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V(s')]$

Algorithm 2.1: Value Iteration algorithm, combining Poliy Evaluation and Policy Improvement. These are interposed to prevent multiple sweeps through all of the states as was the case with Policy Iteration.

2.2.4.3 Summary

DP is a very well developed mathematical approach to solving a problem. However, due to its requirement for a complete model of the environment and its computational demands, it remains mostly a theoretical technique within RL, it is however used extensively outside RL. Monte Carlo methods and Temporal-Difference Learning are based on DP, using many of the same techniques, whilst attempting to approximate without needing perfect knowledge of the environment.

2.2.5 Monte Carlo Methods

Monte Carlo (MC) methods sample sequences of states, actions and rewards through interaction with the environment. This allows optimal behaviour to be formed without requiring complete knowledge of the environment. The experience is divided into a set of episodes, with the state-value approximations and the policy being updated when an episode terminates. MC methods have to unfortunately wait until the end of an episode to perform an update, due to needing the complete sample returns on which to base it[14, p.48].

The underlying concept of MC methods consists of averaging the returns received from visiting a state, which eventually converges on the state's true value[49, Chapter 5.1]. MC methods form episodes by following a policy π, where each occurrence of a state s is referred to as a visit and the first visit is the first time a state is visited in a given episode. To produce estimates for the value of state s when following policy π, $V^{\pi}(s)$, there are two different methods, known as the *every-visit MC method* and the *first-visit MC method*. Every-visit averages over all the visits to s, whereas, first-visit only averages returns from the first visits to s[49, Chapter 5.1]. It can be seen that MC therefore performs a type of Policy Evaluation, as was found in DP.

The MC methods discussed so far estimate V^*, which would be sufficient if a model was available. The algorithm looks ahead and chooses an action that leads to the best reward. If this model is not present, then it is better to estimate Q^* instead, the action-value function. To estimate action values, $Q^{\pi}(s, a)$, the return values are averaged for visits to s in which action a was taken[49, Chapter 5.2]. The problem with estimating Q^* is that some state-action pairs may never occur, resulting in no set of returns to average. This problem can be overcome through exploration, to try and observe more of the state-action space[16, p.369]. One method to handle exploration in MC methods is through an assumption called *exploring starts*, where each episode starts with a state-action pair and every pair has a non-zero probability of being selected[48, p.4].

Just as in DP, Policy Improvement can be performed within MC methods by making the policy greedy for the current value function. This greedy policy is one that chooses an action with the highest Q-value (Equation 2.29), for each state:

$$\pi(s) = \arg\max_{a} Q(s, a) \tag{2.29}$$

The complete MC algorithm does not completely calculate Q^{π_k} each time, due to this requiring potentially infinite states, but instead simply updates towards it[49, Chapter 5.3]. The *Monte Carlo ES* algorithm alternates on each episode between evaluation and improvement to converge on the optimal policy and value function (Algorithm 2.2 [49, Chapter 5.3]).

2.2.5.1 On-Policy and Off-Policy Control

The exploring starts assumption may not always hold, meaning that an alternative is needed to ensure that exploration takes place. Approaches have resulted in two different types of algorithm, *on-policy* and *off-policy*. The former evaluates and improves the same policy as used to control the behaviour of the agent[22, p.1], meaning it aims to find the best policy that still explores. However, the latter separates the *behaviour policy* from the *estimation policy*. This means that it aims to find the best greedy policy, whilst the behaviour policy continues to perform exploration[22, p.1].

Initialise, for $\forall s \in S$, $\forall a \in A(s)$
  $Q(s,a) \leftarrow$ arbitrary
  $\pi(s) \leftarrow$ arbitrary
  $Returns(s,a) \leftarrow \varnothing$
repeat
  Generate an episode using exploring states and π
  forall s, a *appearing in the episode* **do**
    $R \leftarrow$ return following the first occurrence of s, a
    Append R to $Returns(s,a)$
    $Q(s,a) \leftarrow average(Returns(s,a))$
  end
  forall s *in the episode* **do**
    $\pi(s) \leftarrow \arg\max_a Q(s,a)$
  end
until *no more episodes*

Algorithm 2.2: Monte Carlo ES algorithm, which makes the Exploring Starts assumption as a form of exploration. The algorithm alternates between evaluation and improvement for each episode, aiming to obtain the optimal policy and value function.

2.2.5.2 Summary

MC methods can learn the optimal policy and value functions purely through sampling experience. However, they do not perform updates to the value estimates based on other estimates, like DP. Instead, MC methods average the return from visiting states, meaning that estimates can only be improved at the end of an episode. This makes MC methods harder to apply in continuous environments and results in slower convergence than TD approaches, which update continuously.

2.2.6 Temporal-Difference Learning

Temporal-Difference (TD) Learning combines elements of MC methods and DP, to produce an effective and commonly used approach to solving problems through RL. Like MC methods, TD learns from experience, given by following a policy π and updating its estimate of V^π[49, Chapter 6.1]. MC methods use the actual return so can only update at the end of an episode, however, TD only needs to wait until the next time-step. The simplest form is TD(0), which updates its state-value estimates based on those of the successor states[57, p.48], known as *bootstrapping*. The update is controlled by a learning rate parameter, α, which corresponds to the extent to which newly acquired information overrides older information. The updating that occurs in TD is referred to as a sample backup, due to the fact that only the value of a sample successor and the reward achieved is used to compute the updated value[49, Chapter 6.1]. The update rule for TD(0) is:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{2.30}$$

Learning the optimal state-value function, V^*, as would be the aim for the TD(0) update rule (Equation 2.30), requires a complete model of the environment to predict the result of every possible state-action pair. The alternative is to instead sample from the environment and utilise the action-value function, $Q(s,a)$, often referred to as the Q-Function[16, p.374].

An example problem is demonstrated in Figure 2.5, where the agent needs to reach the goal square, labelled as G. RL assigns a $Q(s,a)$ estimate to each of the arrows, shown in the left-hand diagram. It is possible for the $Q(s,a)$ values to converge to the $V^*(s)$ values for the states, which are displayed in the right-hand diagram. Through varying the actions that are taken, to gain more information on which to base a greedy policy, it is therefore possible to obtain the $V^*(s)$ values of the optimal policy. For the example problem, the optimal policy is presented in Figure 2.6.

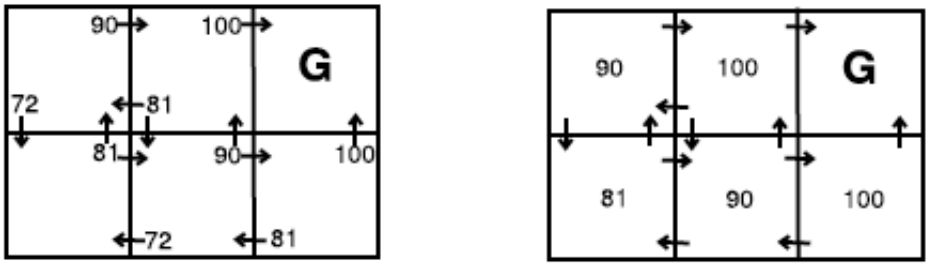


Figure 2.5: Example environment with 6 states. The goal state is denoted as G and the valid actions between states are demonstrated arrows. The $Q(s, a)$ values for an optimal control policy, π^* are drawn on the left diagram and the $V(s)$ values for the same policy are shown on the right diagram.
[5, p.139]

Figure 2.6: An optimal control policy, π^* for the example environment. Showing the best actions to take to move between the states of the environment, towards the goal state G.
[5, p.139]

The optimal policy can therefore be formed through learning the action-value function[60, p.281]:

$$\pi^* = \arg\max_a Q^*(s, a) \qquad \forall s \in S \tag{2.31}$$

Extending TD learning into a full RL control algorithm requires the issue of exploration to be handled, of which approaches fall into on-policy and off-policy, as for MC methods. The on-policy technique *Sarsa* and the off-policy technique *Q-Learning* will be discussed.

2.2.6.1 Sarsa, On-Policy Control

On-policy requires that the algorithm estimate $Q^\pi(s, a)$ for the current behaviour policy, which can be performed with a similar update method as for TD(0). Interaction in the world results in a sequence of state-action pairs, which can lead to a terminal state where the episode ends. The Sarsa action-value function update is then performed after taking an action from each non-terminal state[49, Chapter 6.4]:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{2.32}$$

It can be seen that the update involves a sequence of $\{s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}\}$, which forms the acronym SARSA, after which the algorithm is named. Sarsa continuously interleaves Policy Evaluation with Policy Improvement,where evaluation involves estimating Q^π for a behaviour policy π and improvement involves changing the policy towards being greedy with respect to the current Q^π estimates[49, Chapter 6.4].

Sarsa updates the estimates corresponding to the policy currently being followed by the agent, allowing the algorithm to approach the optimal policy and the optimal action-values[47, p.2]. Due to updating

and following the same policy, convergence to the optimal policy requires one be found that includes exploration (Algorithm 2.3 [49, Chapter 6.4]).

Initialise, $Q(s, a)$ arbitrarily
repeat for each episode
  Initialise s
  Choose a from s using policy derived from Q (e.g. ε-greedy)
  repeat for each step of episode
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g. ε-greedy)
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$ and $a \leftarrow a'$
  until *s is a terminal state*
until *no more episodes*

Algorithm 2.3: Sarsa algorithm, which continually estimates Q^π for the behaviour policy π and changes π towards being greedy.

2.2.6.2 Q-Learning, Off-Policy Control

Q-Learning estimates $Q^\pi(s, a)$ for a policy independent to the one being followed. The $Q(s, a)$ values are approximated iteratively, performing updates whenever the agent interacts with the environment[56, p.86]. Through updating in this way, the algorithm is sampling the transition and reward functions one step ahead from the current state, converging towards the optimal policy. The Q-Learning update rule is[49, Chapter 6.5]:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2.33}$$

If prior knowledge is known about the environment, this can be built into the initial $Q(s, a)$ values assigned to the state-action pairs[59, p.8]. However, otherwise the general approach is for them to be all initialised to the same value, usually zero.

The difference between Sarsa and Q-Learning can be clearly seen in the update rules. Sarsa uses its policy to perform an action then uses the reward and Q-value of the successor state to perform the update. However, Q-Learning instead uses a separate evaluation policy, allowing the behaviour policy to maintain exploration, whilst the $Q(s, a)$ value updates can still use a purely greedy policy (Algorithm 2.4 [49, Chapter 6.5]). The policy still determines which state-action pairs are visited, however, for Q-Learning to converge on the optimal policy all that is required is that updates continue to happen.

Initialise, $Q(s, a)$ arbitrarily
repeat for each episode
  Initialise s
  repeat for each step of episode
    Choose a from s using policy derived from Q (e.g. ε-greedy)
    Take action a, observe r, s'
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$
  until *s is a terminal state*
until *no more episodes*

Algorithm 2.4: Q-Learning algorithm, which approximates the optimal action-value function Q^*, whilst following a separate policy.

2.2.6.3 Summary

TD learning is an alternative to DP methods, that avoids the need for perfect knowledge of the environment, by interacting with it instead. Value estimates are updated through bootstrapping, based on the value estimates of the successor states. The act of using estimates as the basis for the new estimates allows TD learning to improve them continuously, making it more suitable for continuous environments

than MC methods. Its simplicity and minimal amount of computation have made TD learning the most widely used form of RL[49, Chapter 6.9].

2.3 Related Work

2.3.1 Robocode and Reinforcement Learning

2.3.1.1 Choosing Firepower

Nielsen and Jensen[18] attempted to learn the optimal fire-power with which to fire an ANN gun on a Robocode robot, using RL implemented with the Q-Learning algorithm.

The algorithm divided the environment into states, where a state stored the distance the opponent was away from the learning agent. Actions taken by the agent consisted of different levels of power that the turret could be fired with. Nielsen and Jensen determined that the best power with which to fire the turret could be decided from the maximum $Q(s, a)$ value for a given state without needing the knowledge of discounted maximum future value. For this reason the Q-Learning update rule (Equation 2.33) was modified to:

$$Q(s_t, a_t) = (q - \alpha)Q(s_t, a_t) + \alpha R(s_t) \qquad \alpha = 0.1 \tag{2.34}$$

A complex reward function was used, where positive reward was given for damage dealt and energy regained, conversely negative reward was given for gun heat generated and energy drained. Rewards were given upon a bullet: hitting an opponent, hitting a wall or hitting another bullet. The values were taken directly from Robocode, meaning that a higher fire-power bullet would produce higher positive rewards and lower negative rewards. It was hoped that RL would manage to strike a balance between risk and reward.

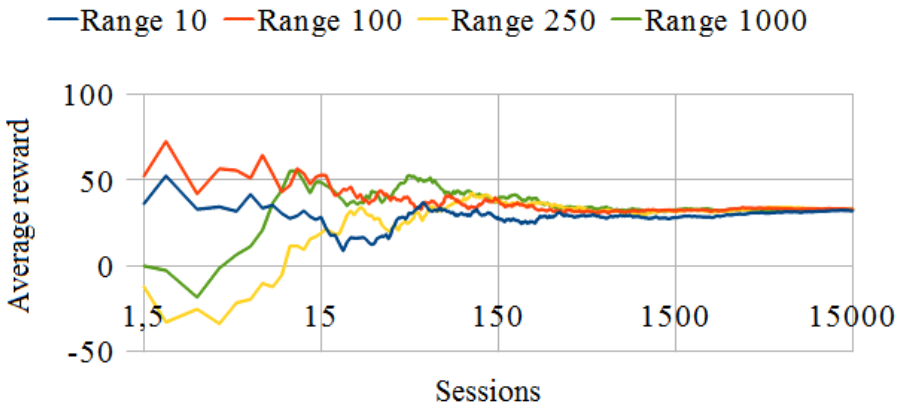


Figure 2.7: Average reward received when bullets are fired at different ranges over 15,000 sessions, by the firepower selection agent developed by Nielsen and Jensen. The rewards converged to the same value for all ranges, showing an effective firepower is found for each range.

To evaluate the performance of the RL algorithm an experiment was ran. First, 32 bullets were fired at the opponent, with power randomly selected between 0.1 and 3, and the $Q(s, a)$ values updated. Then, 32 bullets were fired using the gained information and $Q(s, a)$ estimates stopped being updated. This two phase process was then repeated $15,000$ times. The rewards received over time, for shots fired at different ranges, were recorded. Nielsen and Jensen found that rewards over time converged to the same value for all ranges (Figure 2.7) and that bullets should be fired with high power in almost all situations, rather than just at close range as they had expected would be the case. They concluded that high fire-power was optimal in general due to the high hit-rate of the ANN-controlled gun, 32.69%.

Unfortunately, no rigorous evaluation of the RL algorithm was undertaken, through comparing the results to those collected from a non-ANN gun. It would have been interesting to see how the fire-power would be altered if the gun achieved a lower hit-rate for long range shots. Also, the RL fire-power selection was not compared to a non-RL technique, which would have allowed the effectiveness of the algorithm to be observed. These extra evaluation approaches would allow more interesting conclusions to be drawn on the application of RL to this aspect of Robocode.

2.3.1.2 Choosing the Type of Gun

Uthama[52] used RL through the Q-Learning algorithm to learn, from a selection of different guns, the best type to use against a given opponent. RL was chosen due to the flexibility it provides in allowing guns to be changed whenever needed, to adapt to the opponent.

The different types of gun that were made available to the agent were an ANN gun and a spray gun, which fires out three bullets, one directly at the enemy and one slightly either side. Actions which could be taken included firing the ANN gun, with a power of 0.5, 1, 2 or 3, or firing the spray gun, with a power of 1 or 2. The states then recorded the accuracy of the robot in a period of time, chosen as the number of hits from the last five bullets fired. Due to an ANN gun needing to train before it can achieve a hit rate close to its optimum, the $Q(s, a)$ values were initialised in favour of it being selected. This would ensure it was given the chance to improve before the $Q(s, a)$ values were updated in favour of the spray gun. Once the $Q(s, a)$ values had been initialised, the agent began choosing actions, balancing greedy and exploratory moves, whilst updating the $Q(s, a)$ value estimates.

The RL algorithm was evaluated by playing against Crazy, a robot which moves around in a peculiar pattern, for over 100 rounds. Uthama observed that the ANN-controlled gun achieved a poor hit rate of only 14%, whereas the spray gun managed 22% and using RL achieved 26%. This showed that the use of RL enabled the agent to switch guns, resulting in an increase in hit accuracy. Observing the final $Q(s, a)$ values showed this to be the case, as the RL agent learnt to reward the use of the spray gun against Crazy. An interesting observation made was how the exploration factor played a vital role in the RL algorithm's performance. Uthama discovered that a purely greedy approach resulted in the agent not discovering to switch to the spray gun (due to the initial $Q(s, a)$ values being in favour of the ANN gun). However, altering the amount of exploration seemed to have no effect on performance. Regrettably the RL algorithm was not tested with any other forms of gun or against any other opponents to see how it performed, this would have allowed for a more detailed analysis and for the results to be verified further.

2.3.1.3 Selecting Between Targets

Gade et al.[5] used RL, through the Q-Learning algorithm, to learn the optimal target selection policy, whilst playing in multi-opponent battles. The goal chosen for the RL algorithm was *"a robot, targeted by the target selection module, is killed"*, which allowed the agent to receive a reward whenever its target was defeated. The hope was that through achieving the goal consecutively, the robot would be able to proceed to win the round. It would need to be capable of handling the situation where the target is defeated by another robot.

The states store the energy level of each enemy, the distance to each enemy and the probability of hitting each enemy. To reduce the total number of possible states, each of the parameter's values was segmented into a series of bins. For example probability was represented with three values: unlikely (0 to 0.4), likely (0.4 to 0.9) and very likely (0.9 to 1). If the number of possible states was too large the Q-Learning algorithm would take too long to converge, which justified the segmentation of the values. The actions available to the agent were merely the different targets to aim for. To select actions a probabilistic approach was taken shown in Equation 2.35, which is an adapted form of the Softmax action selection method in Equation 2.19 (Section 2.2.2.2):

$$\pi(a|s) = P(a_t = a|s_t = s) = \frac{\mathrm{k}^{Q(s,a)}}{\sum_b \mathrm{k}^{Q(s,b)}} \tag{2.35}$$

The parameter k is a constant, which is greater than zero, that controls how the $Q(s, a)$ values affect the probability of the action being selected. As k increases, the probability of the action with the highest $Q(s, a)$ value being selected also increases, where $k = 1$ results in all actions being equally likely.

Two different measures were used to monitor the performance of the RL algorithm: cumulative reward and the action selection pattern. Maximising cumulative reward is the overall aim of RL and so observing it ensured that the implementation of Q-Learning was correct. The action selection pattern indicated the type of policy the RL algorithm had learned. Reward is accumulated during a round and then plotted on the graph shown in Figure 2.8. Gade et al. found that over 500 rounds the cumulative reward increased continually, showing that the $Q(s, a)$ values of the visited states increased over each round. The reward increasing over time demonstrated that Q-Learning discovered how to better choose the target.

When observing the action selection pattern, the k parameter was altered to examine how the policy handled different amounts of exploration. A higher value of k represented less exploration and more exploitation of the $Q(s, a)$ values. It was observed that as the value of k increased, the importance of the target's energy became more significant, with the RL algorithm favouring the target with the lowest energy. Contrary to what was expected, as the value of k increased, targets with a lower probability of being hit tended to be chosen over targets with a higher probability. It was concluded that this was due to reward being given if the target was killed, regardless of whether the robot could hit them or not. This could result in the robot choosing a target with low energy which it can then not hit, over a closer opponent which it could have hit. It would have been interesting if Gade et al had ran experiments with an alternate reward function to determine if this was the case and if it could have been resolved.

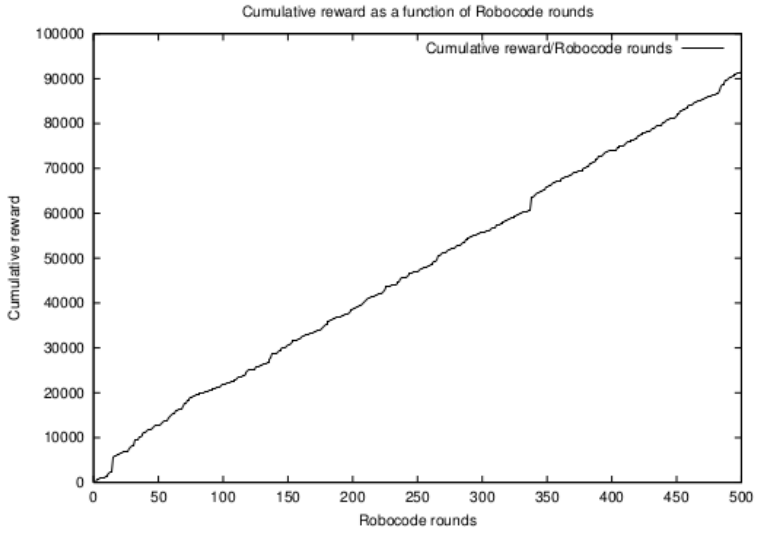


Figure 2.8: Reward received over 500 rounds by the target selection agent developed by Gade et al. Q-Learning improves, showing that the agent learns how to better choose a target.

Gade et al provided a reasonably detailed analysis of the effect exploration had on performance, however they could have used more values and may have managed to improve the algorithm further. Also, the RL target selection was not compared to a non-RL approach to observe the improvement that the use of RL provided.

2.3.1.4 Behaviour Based on Danger Level

Livne[12] experimented with the ability for Q-Learning and other TD methods to learn a full robot behaviour based on the current level of danger. This behaviour involves the controlling of movement, targeting and scanning. The aim was to compare the performance achieved with various different robots, including a non-RL solution.

The first attempt to learn a full robot behaviour involved the state containing information about: robot heading, gun heading, radar heading, robot location, robot energy, enemy location, enemy energy and

whether the robot was hit by a bullet or a wall in the last turn. The actions the agent was capable of taking included: movement direction, movement distance, turning angle, gun turning angle, radar turning angle and fire-power. Clearly this representation was much too descriptive, containing a significant number of different parameters and too much information for the agent to learn in a reasonable amount of time, if at all.

A second attempt reduced the amount of information being used significantly, deriving the state from just the ratio between the enemy's energy and the energy of the robot. This ratio was representative of the level of danger the robot was currently in. A set of pre-defined actions were then provided to the robot, that involved moving, turning, firing and scanning. They were designed to deal with different levels of danger through ranging from defensive to offensive actions. Reward to the agent was based on the ratio between the enemy's and the robot's change in energy level between turns.

Four robots were developed to test how well RL had been applied to the problem. The first robot, `SampleBot`, had a fixed, pre-defined behaviour involving no learning. The second, `QLearningBot`, was the first learning robot, that used the Q-Learning algorithm. The third, `TDLearningBot`, used an alternative TD algorithm that updated $Q(s, a)$ values based on multiple future steps rather than just the successor state. The fourth and final robot, `ExplorerBot`, was an adaptation of the third that randomly selected an action from a certain number of the actions with the highest $Q(s, a)$ estimates. Robots were placed against each other as pairs, where each ran for one hundred battles of ten rounds each.

When each robot was played against the non-learning robot, it was observed that the learning robots started with a winning rate of 20%, which by the end had improved to an average of 80%. It could be seen that `ExplorerBot` managed to achieve a higher final winning rate than the other robots at 84% compared to 78%, however, it had a slower learning rate. This was because `ExplorerBot` performed more exploration than the other algorithms due to its action selection approach, meaning it had a chance to overcome more local optimums, but resulted in slower learning. The final observation made was that learning against a more advanced robot resulted in better overall performance.

Livne performed detailed comparisons between the different learning robots, which showed the effect exploration had on algorithm performance. The RL robots were contrasted with a complete non-RL robot, allowing it to be clearly seen that the RL solution could beat the hand-coded approach. However, Livne could have attempted to learn the robot behaviour without using pre-defined actions for the agent to simply choose between.

2.3.2 Teaching AI and Programming Through Games

Zyda and Koenig[63] investigated the efforts at University of Southern California (USC) of using games to improve the teaching of Computer Science (CS) degree programmes. Their belief was that games would provide motivation for the students and that they would increase enrolment, improve retention and allow students to be better taught. USC went on to incorporate games into the regular CS units, both to backup topics discussed in the lectures and as the basis for assignments. It was discovered that the addition of games into the course increased the application rate substantially and boosted the enrolment figures twofold.

Zyda and Koenig[63] discussed the utilisation of games to teach AI concepts at USC. This was decided due to AI being both a core topic to any CS programme, whilst also being one of the main concepts used in games development. Games now use a wide range of AI techniques, including path-planning, search methods, decision trees and neural networks. This opens up the possibility for many areas of AI to be demonstrated and taught through games, which also provides significant opportunities for AI research.

McGovern et al.[13] developed a Java gaming framework, which they used to teach an AI course at the University of Oklahoma, believing that the use of games would motivate students and enable them to see more quickly how the algorithms work. They found that students gained an appreciation for AI by completing game-related assignments, incorporating the AI concepts taught on the course. The framework that was developed included a variety of different games, ranging from Pacman, through to Mancala, an ancient African game. The games such as Pacman provided an environment to learn about search algorithms such as A^*, with an assignment also being set that encouraged the use of genetic algorithms. McGovern et al. concluded that students found it hard to apply algorithms before, but

that the gaming framework allowed them to easily apply them to new domains and gain considerable confidence in AI topics.

2.3.3 Summary

It can be seen that a variety of previous work has been completed on using RL with the Robocode environment. Most of the work has involved learning a small aspect of the robot's behaviour, such as selecting a target in the case of Gade et al. Although Livne used RL to learn a full behaviour, it involved simply choosing between pre-defined actions and situations. This means that an in-depth analysis on learning all the main aspects of behaviour (movement and targeting) through RL is needed. This would involve applying RL to multiple problems, learning either consecutively or simultaneously.

Games, such as Robocode, provide a strong learning environment for AI and programming in general. They have been found to engage students and allow them to understand complex AI concepts more easily. The Java gaming framework developed by McGovern et al allowed their students to learn about AI through applying the algorithms to new game domains. It can therefore be seen that the development of a framework application for Robocode would provide a strong learning environment for RL.

Chapter 3

Project Execution

This chapter provides details of how the RL robot (Section 3.1) and the framework application (Section 3.2) were implemented. The design of each part will be discussed, including any design decisions which were made.

3.1 Reinforcement Learning Robot

Previous work concerning the use of RL with Robocode has focused on learning various different aspects of the robot's behaviour. The approach taken in this project is instead to learn a full robot behaviour, separated into movement and targeting. This goes far beyond the work of Livne[12], who allowed the robot to learn a full behaviour by choosing between a set of pre-defined actions. Instead, the robot controls each aspect of its behaviour through a separate RL algorithm.

A strong movement strategy is crucial for winning, which makes learning to do this an interesting problem. It has been heavily studied in the past, resulting in a large number of hand-coded techniques to choose between. Predicting the robot's future position is simple if the movement strategy is too basic, which is the reason for methods such as Anti-Gravity and Wave Surfing. Both techniques are very effective at avoiding danger, however, Wave Surfing was developed much later and has since become one of the most popular movement strategies. To learn a movement strategy, avoiding damage from enemy bullets and navigating to an advantageous position on the battlefield must be balanced, by monitoring how the opponent moves. Hand-coded techniques lack the ability to adapt to different enemy types, providing RL the advantage because it can.

In order to defeat an opponent a robot needs to be fully capable of targeting and firing accurately. The time it takes bullets to travel across the battlefield, coupled with the complexity of the more advanced movement techniques, mean it is not as simple as aiming straight at the enemy and firing a bullet. Well studied hand-coded approaches focus mainly on predicting the opponent's future position and then firing towards that point. However, robots are not guaranteed to move in the same way throughout the battle, meaning that RL has the benefit of being able to adapt targeting based on the enemy's behaviour. Targeting provides a new application of RL to a Robocode robot which has not been previously studied.

If both movement and targeting were both learnt by a single RL implementation, the state and action representations would need to include all of the information required to control both behaviours. This would result in a very complex implementation, which would learn slower due to the larger number of possible states and actions. For this reason each aspect of the robot's behaviour is controlled independently and situated in a separate component of the robot. The scanner rotates the radar, which can then provide the other components with any information gathered from detected opponents. Moving and turning of the main body is handled by the driver, who can use the information from the scanner to avoid enemies and their bullets. The gunner manages the rotating and firing of the turret, which involves choosing the fire-power to use and then firing upon detected opponents. Additionally, there is a logger that handles outputting to files and loading from properties files. The main robot combines these

different components together, allowing them to communicate, behave collaboratively and react to any situations that occur.

3.1.1 Application of RL to Robocode

In order to integrate RL into the robot, an implementation was used that was extended from a Java RL library developed by Dr Tim Kovacs[9]. The library allows learning agents and environments to be set up, so that RL experiments can be ran. For an experiment to be ran, various components are required: an environment, an agent, an action selection policy and possibly a Q-Function (action-value function). All the information about the problem that the agent is attempting to solve is contained within the environment, including actions that can be taken and the states the agent could be in. Custom state and action representations are used, to allow the library to be applied to different problems. The agent interacts with the environment by implementing either Q-Learning, Sarsa or MC methods and using either ε-greedy or Boltzmann Softmax for action selection. To store the action-value estimates, a Q-Table is used, which contains entries for each state-action pair.

Usually in RL, upon an action being completed, the agent advances straight to the next state. However, within Robocode a robot has no direct control over the game, meaning it cannot advance between states in this way. Instead the robot's state is determined as soon as the opponent has been scanned and then an action is selected to be carried out. The current time-step lasts until the robot has finished performing its chosen action. To accommodate this changes were made to the RL implementation, with advancement to the next state and the next action being chosen as soon as the opponent is scanned and the previous action has completed. Achieving this involved weaving RL time-steps and updates into the event-driven manner in which a robot is written.

To make the robot as customisable as possible, the logger component loads a properties file in at the start of the battle, altering aspects of the robot. Rather than fixing which algorithms would be used to perform RL, Q-Learning, Sarsa and MC methods were all supported. Even though it was expected that MC methods would produce lower performance, due to not updating continuously, the option was included to allow interesting experiments to be ran at the end. Both ε-greedy and Boltzmann Softmax action selection techniques were supported for the same reason.

The stochastic Robocode environment means that performing a given action in a given state does not result in the same successor state each time. This is due to the robot having no control over how the enemy behaves and the continually changing world (robots and bullets moving around). Recall the $Q(s, a)$ estimate update rules for Q-Learning (Equation 2.33) and Sarsa (Equation 2.32).

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \quad \text{(2.33 revisted)}$$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \quad \text{(2.32 revisted)}$$

The learning rate, α, is used to control the extent to which the old estimate affects the new one. Basing estimates on previous estimates handles non-determinism to an extent, however, to improve it further instead of using a fixed learning rate, a dynamic learning rate is proposed. This changes based on the number of times a particular state-action pair, (s, a), has been visited so far, denoted $visits(s, a)$. The dynamic learning rate is defined as:

$$\alpha_t = \frac{1}{1 + visits(s_t, a_t)} \quad (3.1)$$

Therefore, as a state-action pair is visited more often, the old $Q(s, a)$ estimate has a greater influence on the new estimate. Non-determinism means that on each visit an action could result in a different successor state with a different $Q(s, a)$ estimate. Using the dynamic learning rate ensures that the $Q(s, a)$ estimates can still converge[16, p.382], as each visit will have less of an effect on the update.

To give RL a better opportunity to learn the best policy it can, it was decided that a focus would be given to 1-vs-1 battles, where the robot is matched against a single opponent. This would allow the robot to focus its attention on just one opponent.

3.1.2 Scanning

Control of the radar is the first main component of the robot. There are many accepted hand-coded techniques for scanning, ranging from the simple Spinning Radar to the more complicated Turn Multiplier Lock. The Spinning Radar spends the majority of its time scanning empty space, which Oldest Scanned attempts to improve upon this by rotating towards a specific enemy. However, as it simply alters the scanning direction it is still better designed for melee battles than for 1-vs-1. A significantly better choice is the Turn Multiplier Lock Radar, which maintains a lock on the opponent. This makes for a very efficient strategy in 1-vs-1 and means the robot is always aware of the opponent's location.

Applying RL to radar control seemed unnecessary, due to the Turn Multiplier Lock strategy being optimal in a 1-vs-1 scenario. Once it has locked on, the opponent can be scanned every turn without much movement of the radar, so there is little that could be improved. If RL were to be used, there would not be much for it to base its state representation upon, due to most information not being gained until the opponent has been scanned. Also, there is no need for the scanning strategy to adapt to different enemy behaviours. For these reasons it was decided that RL would not be applied to radar control and that the Turn Multiplier Lock Radar would be used instead.

At the beginning, the scanner simply performs a complete sweep to locate the opponent for the first time. From that point onwards, the radar can simply be rotated towards the enemy's last location. The angle between the radar's facing direction and the bearing to the enemy is computed and the radar is then rotated by twice that angle. This has the effect of performing a scanning arc around the opponent's position so that they cannot move to escape the lock-on. The full procedure can be seen in Listing 3.1.

```
double turn_angle = robot.getHeading() + enemy.getBearing()
                          - robot.getRadarHeading();
turn_angle = normalRelativeAngle(turn_angle);
robot.setTurnRadarRight(2 * turn_angle);
```

Listing 3.1: Turn Multiplier Lock Procedure, which calculates the angle it needs to turn to face the enemy. The radar is then turned twice this distance, scanning an area around the enemy and ensuring lock-on is maintained.

3.1.3 Movement

The task of moving and rotating the robot body is handled by the driver component using RL.

3.1.3.1 Analysis

A good movement strategy needs to deal with the many different situations that may occur during a battle, as best it can. The situation the robot finds itself in should dictate how it moves, which is why many hand-coded techniques are unsuccessful. For example, if the robot has a wall in close proximity it should move more cautiously than if it is in the centre of the battlefield. This is because crashing into a wall causes the robot to take damage and being stuck in a corner will make it a much easier target for the enemy's turret. Likewise, the robot should attempt to keep the enemy from getting close, as aiming at close range is much simpler. An analysis of the game highlights that when deciding where to move the robot should keep aware of: the opponent, the walls and the corners.

Each threat that the robot needs to keep aware of highlights many parameters that describe the world, as shown in Table 3.1. For example, to avoid moving into the walls, the robot can use its position and heading, combined with the locations of the walls to calculate which direction would move it away from the wall. For RL to learn the movement behaviour, it can form a state representation from these different parameters.

The aim of the RL algorithm is to learn the optimal movement policy, π^*, by following a goal and evaluating the quality of actions that are taken. Working towards the goal set and following the rewards that are given to the algorithm allows the action-value estimates to be improved and progress to be made

Threat	Useful Parameters
Walls	Robot's position
	Robot's heading direction
	Battlefield width and height
Corners	Robot's position
	Robot's heading direction
	Positions of the corners
Enemy	Distance to the enemy
	Angle to the enemy
	Enemy's position
	Enemy's heading direction
	Enemy's velocity
	Enemy's energy
	Robot's energy
	Whether the enemy has fired
	Bullets predicted positions

Table 3.1: Threats for the movement controller to keep aware of and the parameters the robot could find useful to deal with them.

towards π^*. A general goal which could be used is: *"Select a movement so that the agent goes on to win the current round"*. This would allow the agent to be rewarded whenever the robots wins a round. However, limiting the algorithm to rewards and RL updates at the end of each round would make the learning process very sluggish. Additionally, it makes the incorrect assumption that success is always due to the quality of movement control. These weaknesses emphasise that the goal should allow updates to occur during the round and should reward the performance of the movement controller alone.

An improved goal is: *"Select a movement so that the robot does not take any damage"*. The agent would be rewarded and $Q(s, a)$ estimates updated, whenever it makes a move. Damage can be received in several different ways: by driving into a wall, by colliding with an enemy and by being hit by a bullet. If the goal of not taking any damage is continuously achieved, the robot will go on to win the round as the opponent will use up all of its energy. This improved goal therefore includes the original one in a more detailed form. It was chosen to be used, due to it allowing updates to occur continuously and because it focuses on the performance of just the movement control.

3.1.3.2 RL Algorithm

To determine whether Q-Learning, Sarsa or MC methods should be used, a comparison was made between them. It was expected that Q-Learning would perform the best, due to it updating continuously and being an off-policy algorithm. This means that it updates more often than MC methods and the policy it learns is purely greedy, without exploration built-in, like the Sarsa on-policy approach. For these reasons Q-Learning was used. The results and discussion can be found in the Evaluation Chapter (Section 4.1.2).

3.1.3.3 State Representation

To allow the agent to build knowledge of the best way to move, a state representation is required, which actions can then be assigned to. These states will correspond to the different situations the robot finds itself in and will rely on *sensor input* and *collected input*. Sensor input contains information about the

world, such as the robot's position and heading direction. Collected input contains all the information that has been collected and calculated by the various components of the robot, such as the bearing to enemy and the opponent's position. Table 3.1 presented many of the different sensor inputs and collected inputs that are available for use.

If the state was derived by simply collecting together all of the quantifiable information available to the robot, the number of possible values each parameter could take would be:

$$\begin{array}{ccc} \text{Enemy's heading direction} & = & [0; 360] \\ \text{Enemy's position} & = & \{0, \ldots, 800\} \times \{0, \ldots, 600\} \\ \text{Enemy's velocity} & = & [-8; 8] \\ \text{Enemy's energy} & = & [0; 100] \\ \text{Robot's heading direction} & = & [0; 360] \\ \text{Robot's position} & = & \{0, \ldots, 800\} \times \{0, \ldots, 600\} \\ \text{Robot's velocity} & = & [-8; 8] \\ \text{Robot's energy} & = & [0; 100] \\ \text{Distance to enemy} & = & [0; \sqrt{800^2 + 600^2}] \\ \text{Angle to enemy} & = & [-180; 180] \\ \text{Whether the enemy has fired} & = & [0, 1] \end{array} \tag{3.2}$$

The total number of states using all of the parameters could be calculated as follows:

$$|[0; 360]|^3 \times (800 \times 600)^2 \times |[-8; 8]|^2 \times |[0; 100]|^2 \times |[0; \sqrt{800^2 + 600^2}]| \times 2 \tag{3.3}$$

Even if all values were discretised to integer values, there would still be a very high number of possible states:

$$360^3 \times (800 \times 600)^2 \times 16^2 \times 100^2 \times \sqrt{800^2 + 600^2}^2 \times 2 = 5.503765708 \times 10^{28} \tag{3.4}$$

With a high number of possible states, the robot finding itself in a state it has already seen becomes much rarer and can result in it taking a long time for value estimates to converge. To prevent this, it is important that the number of possible states is reduced, through removing redundant parameters and *segmentation*.

The first redundant parameter is the robot's velocity. This is because sensor input is only collected once an action has completed and so the robot's velocity will be zero at this point. Using the robot's position, the angle to the enemy and the distance to the enemy, the opponent's location can be easily represented. This makes the enemy's position the second redundant parameter, especially considering using the other three parameters provides a lower number of possible values. Through observing robot velocities, it can be seen that they mostly increase to the maximum as the robot moves and then return to zero again at the end of the turn. Using the opponent's velocity would therefore not provide much useful information to the movement controller and is therefore the third redundant parameter. The robot's energy and the opponent's energy are the fourth and fifth redundant parameters. Whether the robot or the opponent have high or low energy has no bearing on the moves to take that will avoid damage. Thus the robot will choose its moves regardless of these values. A robot is capable of rotating its turret and main body independently of each other, meaning that the opponent will not necessarily fire in its direction of travel. The enemy's heading direction is therefore the sixth redundant parameter, as the agent is more interested in the direction of incoming bullets than which way the opponent is facing.

The final parameter to be considered is whether the enemy has fired or not recently. If it was present, the agent could perhaps learn to move away from the opponent when a bullet has been fired. However, the parameter provides much less information than is required for a full bullet avoidance strategy. This would involve predicting the trajectories of bullets that are fired, rather than just the knowledge that a bullet has been fired recently. Therefore, reasons exist for both removing and keeping the parameter. To determine which was best an experiment was carried out, which found that inclusion of the parameter decreased the overall performance of the RL algorithm. This was because the agent received less reward than it did without it present. The results and associated discussion can be found in the Evaluation Chapter (Section 4.1.4.1).

After removing all the mentioned redundant parameters, those remaining include: the robot's position, the robot's heading direction, the angle to the enemy and the distance to the enemy. The number of possible states has therefore been reduced to:

$$360^2 \times 800 \times 600 \times \sqrt{800^2 + 600^2} = 6.2208 \times 10^{13} \tag{3.5}$$

To further reduce this number, segmentation must be used. The process consists of assigning a mapping from a large set of values to a smaller set of groups or bins. Exact details of how this mapping is achieved are different depending on the parameter being segmented. Some inputs require a large number of bins to achieve an adequate level of accuracy, whereas some can be represented with a lower number. When intuition is insufficient, experimenting with multiple possible mappings is required to determine which is best.

In the following paragraphs, the mappings that were chosen to represent the parameters are discussed.

Robot Position. The robot's position consists of (x, y) coordinates, which lie within the bounds of the battlefield (800×600). Initially x can therefore take any value within $\{0, \ldots, 800\}$ and y can take any value within $\{0, \ldots, 600\}$. Position is in desperate need of segmentation, as otherwise it has a very high number of possible values.

Each axis could be divided into bins of size 100, splitting the battlefield into a selection of squares. This would require 8 values for x and 6 values for y, resulting in $8 \times 6 = 48$ different values that the robot's position could take. The mapping function splits up the battlefield as shown in Figure 3.1.

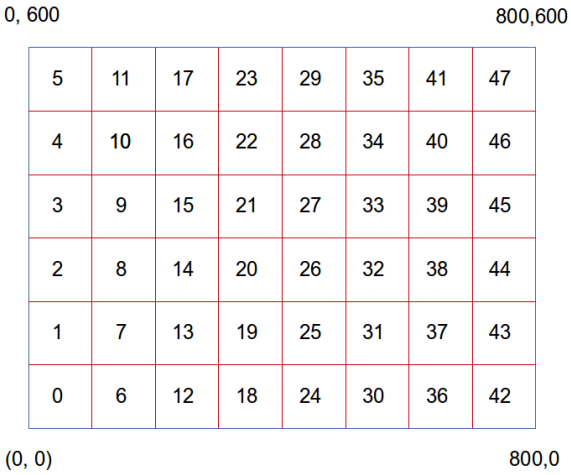


Figure 3.1: Position segmentation for the movement state that uses 48 bins, dividing up the battlefield into a grid of squares.

Many of the squares would result in the robot wanting to move in the same way and it is best to remove these redundant values so that the overall number of states can be reduced. The main area of danger on the battlefield is the edges, where the robot is at risk of colliding with the walls. Once away from the edge, the robot is safe from damage through wall collisions and can instead focus on other parameters to dodge the enemy and its bullets. Therefore, an alternative method would be to divide the battlefield into five different areas: one for each of the edges and then one for the centre. The mapping function for this is seen in Equation 3.6 and it would divide the battlefield as shown in Figure 3.2.

$$p(x,y) = \begin{cases} 0 & \text{if } 0 \leq x \leq 150 \\ 1 & \text{if } 450 \leq y \leq 600 \\ 2 & \text{if } 650 \leq x \leq 800 \\ 3 & \text{if } 0 \leq y \leq 150 \\ 4 & \text{otherwise} \end{cases} \tag{3.6}$$

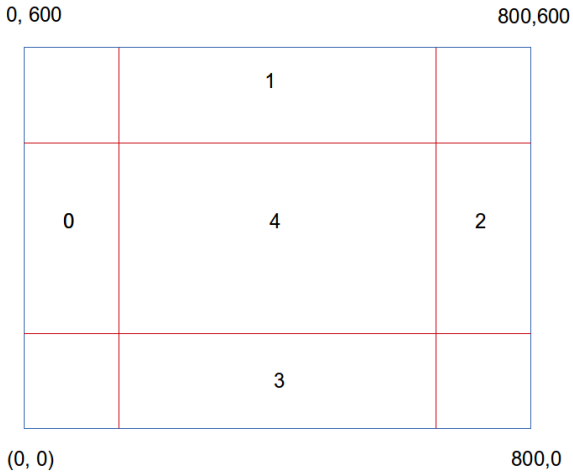


Figure 3.2: Position segmentation for the movement state that uses 48 bins, separating the central area of the battlefield from the edges.

It can be seen that when the robot is in the corners, the position will fall into whichever bin is checked first. The upper-right corner for example could be in bin 1 or 2. If the robot is in a location covered by bin 1, on average it would be reasonable to move to the right, however, in the upper-right corner this could result in crashing into the wall. This demonstrates how the corners are insufficiently distinguished by this mapping. The action the robot will want to take when in a corner is not the same as the actions it would take when in a position along the wall. For this reason, a further approach would be to make the corners separate from the edges with the mapping function in Equation 3.7, which would separate the battlefield as in Figure 3.3.

$$p(x,y) = \begin{cases} 0 & \text{if } 0 \leq x \leq 150 \text{ and } 150 < y < 450 \\ 1 & \text{if } 650 \leq x \leq 800 \text{ and } 150 < y < 450 \\ 2 & \text{if } 150 < x < 650 \text{ and } 450 \leq y \leq 600 \\ 3 & \text{if } 150 < x < 650 \text{ and } 0 \leq y \leq 150 \\ 4 & \text{if } 150 < x < 650 \text{ and } 150 < y < 450 \\ 5 & \text{if } 0 \leq x \leq 150 \text{ and } 450 \leq y \leq 600 \\ 6 & \text{if } 650 \leq x \leq 800 \text{ and } 450 \leq y \leq 600 \\ 7 & \text{if } 0 \leq x \leq 150 \text{ and } 0 \leq y \leq 150 \\ 8 & \text{if } 650 \leq x \leq 800 \text{ and } 0 \leq y \leq 150 \end{cases} \tag{3.7}$$

When in the corners, the robot should learn to move diagonally back towards the centre of the battlefield. If elsewhere along the edges, it will likely learn to move perpendicular to the wall. Whenever the robot is in the main central area, it will use the other parameters to avoid the opponent robot, secure in the knowledge that it is away from the walls. This mapping was therefore chosen to represent the robot's position.

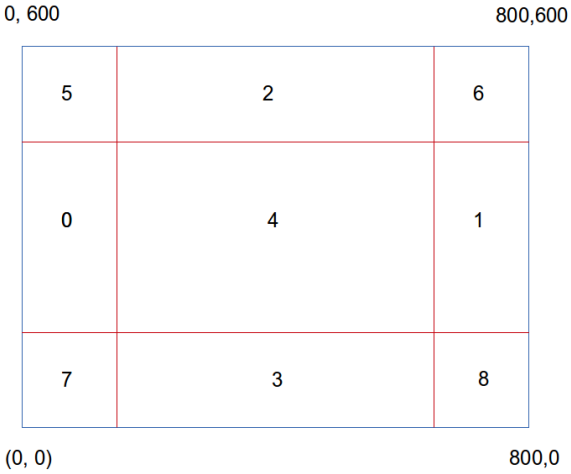


Figure 3.3: Position segmentation for the movement state that uses 9 bins, separating the central area of the battlefield from the edges and handling the corners separately.

Robot Heading Direction. The robot's heading direction consists of an angle relative to the world, stored in degrees to make specifying the bins easier. Without segmentation, the heading can initially take any value within $\{0^o, \ldots, 360^o\}$.

Deciding which direction to move in requires the robot to know the way it is facing, meaning if the number of bins is too small, the robot will not be able to distinguish from many directions. The approach taken was therefore to divide the full 360^o into the eight basic compass directions: North, North-East, East, South-East, South, South-West, West and North-West. The mapping function would be:

$$h(a) = \begin{cases} 0 & \text{if } 338 < a \leq 22 \\ 1 & \text{if } 22 < a \leq 68 \\ 2 & \text{if } 68 < a \leq 112 \\ 3 & \text{if } 112 < a \leq 158 \\ 4 & \text{if } 158 < a \leq 202 \\ 5 & \text{if } 202 < a \leq 248 \\ 6 & \text{if } 248 < a \leq 292 \\ 7 & \text{if } 292 < a \leq 338 \end{cases} \tag{3.8}$$

This mapping function separates the heading directions as shown in Figure 3.4.

Reducing the mapping further would result in using only four directions: North, East, South and West. However, this would be too restrictive and would prevent the robot from correctly determining which way to travel in. For example, a heading of 46^o is in the North-Eastern direction, but the smaller mapping would count this as East. It is inevitable that this would result in the robot not moving correctly and would likely reduce overall performance. Using all eight directions provides much better accuracy without causing there to be too many possible states. Therefore, the segmentation from above was used, representing the eight basic directions with eight bins.

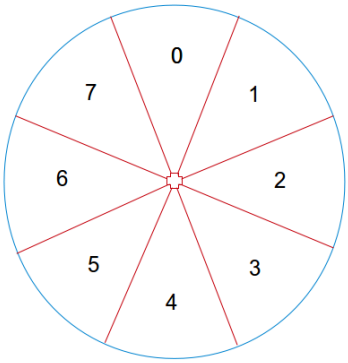


Figure 3.4: Robot heading segmentation for the movement state that uses 8 bins, dividing the 360^o into the eight basic compass directions.

Angle to the Enemy. The enemy's bearing is obtained as an angle relative to the robot's heading direction. It can initially take values within $\{-180^o, \ldots, 180^o\}$, so to make specifying the bins easier it was converted to be an absolute angle within $\{0^o, \ldots, 360^o\}$. Using the angle to the enemy allows the robot to determine the enemy's location and the direction incoming bullets will arrive from. The same mapping as for the robot's heading direction was used, due to it providing sufficient detail (Equation 3.8).

Distance to the Enemy. The distance from the robot to the enemy is the length of the vector joining their positions, which is maximal when the robot and opponent are in diagonally opposite corners. In this instance the Euclidean distance would be calculated as:

$$\text{max distance} = \sqrt{800^2 + 600^2} = 1000 \tag{3.9}$$

Thus distance to the enemy can initially be within $\{0, \ldots, 1000\}$. The simplest method for dividing the distance range would be to split it into bins of size 100, resulting in ten different bins. The mapping function would be:

$$d(m) = \left\lfloor \frac{m}{100} \right\rfloor \tag{3.10}$$

This separates the distance as shown in Figure.

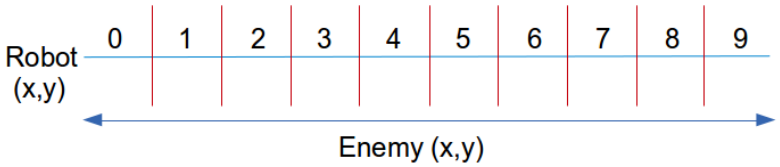


Figure 3.5: Distance to enemy segmentation for the movement state that uses 10 bins. Dividing the full distance into equal parts.

Many of the distances will have little effect on how the robot should behave. For example, when a distance is far, it does not matter to the robot's movement strategy whether its value is 800 or 900. Likewise, even for a distance that is closer, whether it is 300 or 400 is not very important. Therefore, it is possible to reduce the number of bins that are used by representing the distance with the mapping shown in Equation 3.11, which divides the distance as illustrated in Figure 3.6.

$$d(m) = \begin{cases} 0 & \text{if } 0 \leq m \leq 50 \\ 1 & \text{if } 50 < m \leq 150 \\ 2 & \text{if } 150 < m \leq 300 \\ 3 & \text{if } 300 < m \leq 450 \\ 4 & \text{if } 450 < m \leq 700 \\ 5 & \text{if } m > 700 \end{cases} \tag{3.11}$$

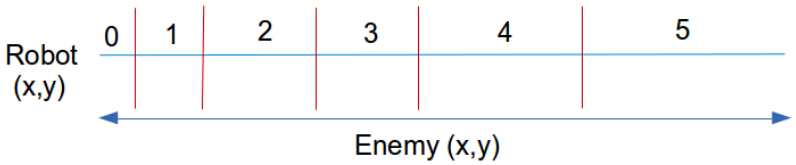


Figure 3.6: Distance to enemy segmentation for the movement state that uses 6 bins. Bin sizes increase as the distance increases, showing how differences in far away distances are less important.

Using the smaller representation, the robot can establish if a distance is: very close, close, quite close, middle range, quite far or very far. When the enemy is close the robot should learn to move away from it to avoid damage and if the enemy is far away the robot will try to get into medium range to make targeting it easier. This mapping captures the different distances the robot needs to be aware of and so was used to represent the distance to the enemy.

Result. After removing the redundant parameters the number of possible states was lowered to 6.2208×10^{13}. The application of segmentation to the remaining parameters allows the number of possible states to be reduced even further, down to a more manageable number:

$$9 \times 8 \times 8 \times 6 = 3456 \tag{3.12}$$

This lower number of possible states will allow the agent to learn a much better movement behaviour than before.

3.1.3.4 Action Representation

Actions comprise of a movement in a specific direction, so to achieve this the representation needs to include the angle to turn and the distance to move for. As for the states, every possible value cannot be included as it would prevent the RL algorithm from converging in any reasonable time. With a large number of possible actions, the chance of a state-action pair occurring multiple times is much lower, thus $Q(s, a)$ estimates would take a longer time to converge towards their optimal values. To ensure this does not happen the number of possible actions needs to be reduced

Rather than allowing all angles, they can be limited by applying segmentation. Moving in the eight basic compass directions, as put forward in Section 3.1.3.3, would allow the robot to navigate around the battlefield. It could be argued that more flexibility in the available directions would allow the robot to better avoid incoming bullets, however, adding extra actions would negate the benefits by slowing down the learning process. The angles that the eight directions correspond to are: 0^o, 45^o, 90^o, 135^o, 180^o, 225^o, 270^o and 315^o. This corresponds to the following mapping:

$$d = \begin{cases} 0 & \text{for } 0^o \\ 1 & \text{for } 45^o \\ 2 & \text{for } 90^o \\ 3 & \text{for } 135^o \\ 4 & \text{for } 180^o \\ 5 & \text{for } 225^o \\ 6 & \text{for } 270^o \\ 7 & \text{for } 315^o \end{cases} \tag{3.13}$$

Only when an action completes does an RL update and an action selection take place. Therefore, it is advantageous for movements to take as little time as possible to perform. For this reason it was decided that movements would have a constant distance of 100. Longer distances would reduce the flexibility of movement, with the agent performing less actions and RL updates so learning more slowly. Whereas shorter distances would allow more moves to be taken, but the robot will be easier to hit due to it not moving very far. The best distance will therefore strike a balance between the two, which 100 does sufficiently.

3.1.3.5 Action Selection Method

It is important that exploration is maintained to allow the optimal movement policy to be formed. A comparison was made between ε-Greedy and Boltzmann Softmax with different amounts of exploration to decide which was best to use, the results for which can be found in the Evaluation Chapter (Section 4.1.3). It was found that ε-Greedy and exploring 10% of the time provides the best balance between exploitation and exploration, allowing the RL algorithm to make the most improvement to the reward.

3.1.3.6 Rewards

In Section 3.1.3.1 the goal for the RL algorithm was chosen as: *"Select a movement so that the robot does not take any damage"*. For this to be achievable, rewards need to be given to the agent based on the actions it takes. Whenever damage was taken a penalty (negative reward) was given, which will allow the agent to learn that damage should be avoided.

Being hit by a bullet is the most common form of damage as it is a possibility regardless of where on the battlefield the robot is. It is also the main situation that the agent should learn to avert. The reward given was a constant amount, $-r$.

Whenever the robot is towards the perimeter of the battlefield, damage can be taken by colliding with the wall. If in close proximity with a wall, movement flexibility is reduced and so the robot is also more susceptible to damage from incoming bullets. Due to their being increased likelihood of taking damage when in the vicinity of a wall, the reward given was higher than when being hit by bullets, $-2r$.

Driving into the opponent causes damage to be received, however, it is much less likely to occur than the other two situations. It could be argued that it causes damage to the enemy and so could be used to defeat them, but this goes against the agent's goal and so should not be rewarded. Besides, being close to the opponent increases the probability of the robot being hit by a bullet, due to the short distance making it harder to dodge. The effect of being near the opponent is similar as for walls and so the same reward was given, $-2r$.

Rather than the absolute values of the rewards, it is the relationship between the different rewards that is important. The constant reward, r, was therefore chosen to be 1. This means that the rewards are: -1 for being hit by a bullet, -2 for driving into a wall and -2 for driving into the opponent. These can be granted numerous times in one round, so if the robot is hit by 3 bullets and hits the wall it will have a reward of -5. There was the option of supplying a positive reward if the robot was not damaged by a bullet, however, without it the robot would get 0, which is higher than an otherwise negative reward anyway.

3.1.3.7 Implementation

Control flow of the movement component involves three phases: initialisation, movement and output. When the battle begins initialisation is performed, where all aspects of the RL robot are set up. A Q-table was used to store the $Q(s, a)$ estimates, which is a data structure that contains an entry for each state-action pair. These estimates were all initialised to zero, due to there being no reason to give preference to specific states or actions.

Upon initialisation being completed, actions can start being performed. When an enemy has been detected by the scanner, information about the enemy is transmitted to the movement component to form the current state. Except when it is the first action of the round, the $Q(s, a)$ estimate is updated for the previous state-action pair using any accumulated rewards, then an action is chosen. Whenever the robot is damaged, the appropriate reward is given to the agent, which is reset to zero whenever an RL update occurs.

During the battle, reward is stored for each 2000 time-steps, so that afterwards the learning performance can be analysed. At the end of the battle, the logger outputs the rewards and the computed Q-table to a file.

3.1.3.8 Alternative Hand-Coded Techniques

To allow comparisons to be made, the hand-coded technique Wave Surfing was implemented into the RL robot driver. Incorporating it into the existing design involved the driver checking which technique was controlling it and calling the correct movement method. As mentioned in Section 3.1.1, a properties file was used to switch between the different control techniques.

3.1.4 Targeting

The gunner controls the turning and firing of the turret using RL.

3.1.4.1 Analysis

In order for bullets to hit the opponent, selecting an angle to aim the turret needs to take the travel time of the bullet into account. This angle will be different depending on the situation, basing it on which direction the enemy is travelling in, the enemy's velocity, the enemy's bearing and the distance to the enemy.

To form the optimal targeting policy, π^*, a possible goal to use is: *"Aim the turret so that the robot defeats the opponent"*. Using this goal would allow the agent to be rewarded whenever the opponent is killed, which in a 1-vs-1 battle would be the end of the round. This unfortunately has similar drawbacks to the general goal for movement, *"Select a movement so that the agent goes on to win the current round"*, discussed in Section 3.1.3.1. Only rewarding the agent after the opponent is defeated will make the learning process very slow, so it is much more advantageous to allow rewards to be given more regularly.

A better goal is: *"Aim the turret so that when a bullet is fired it hits the opponent"*. Assuming the robot does not take too much damage, if this goal is repeatedly achieved, it will result in the opponent being defeated. Due to being able to reward whenever a bullet hits or misses, more regular updates can be made to the $Q(s, a)$ estimates allowing a faster learning process. For this reason the second goal was chosen to be used.

3.1.4.2 RL Algorithm

Choice of RL algorithm was discussed for movement control in Section 3.1.3.2. For the same reasons Q-Learning was chosen for targeting control.

3.1.4.3 State Representation

It was discussed in Section 3.1.4.1 how targeting the opponent depends on their: velocity, heading direction, distance away and bearing. There are many other parameters which could also be used, such as: enemy's position, enemy's energy, robot's heading, robot's position, robot's velocity and robot's energy. However, the targeting strategy would not be affected by parameters concerning the robot's own situation within the world, so all parameters about the robot were discounted. Even though it is possible the enemy's movement strategy may change based on their energy level, this is very unlikely and so it was not used. The final parameter, which is the enemy's position, provides no additional information that the enemy's bearing and distance away do not already provide and so was not used either. If the parameters are discretised to integers, the number of possible states using this representation is therefore:

$$\begin{array}{ccc} \text{Enemy's heading direction} & = & [0; 360] \\ \text{Enemy's velocity} & = & [-8; 8] \\ \text{Distance to enemy} & = & [0; \sqrt{800^2 + 600^2}] \\ \text{Angle to enemy} & = & [-180; 180] \end{array} \tag{3.14}$$

$$|[0; 360]|^2 \times |[-8; 8]| \times |[0; \sqrt{800^2 + 600^2}]| = 360^2 \times 16 \times \sqrt{800^2 + 600^2} = 2.0736 \times 10^9 \tag{3.15}$$

Removal of the redundant parameters was already carried out before counting the number of possible states, so to reduce the number further segmentation must be used. The following paragraphs describe the mappings that were chosen to represent the different parameters.

Enemy Heading. The enemy's heading direction is obtained as an absolute orientation relative to the world, which is within $\{0^o, \ldots, 360^o\}$. Taking it into account should allow the robot to aim bullets in front of the opponent's facing direction. The mapping chosen was the same used for the robot heading in Section 3.1.3.3. This segmentation allows the agent to distinguish between eight different enemy headings, which is sufficient to determine an aiming direction.

Enemy Velocity. The enemy's velocity consists of a value within $\{-8, \ldots, 8\}$, where velocity is negative when the enemy is reversing. When the opponent is moving faster, the robot will need to aim further in front if it wishes the bullet to hit. To capture this in the mapping, possible velocities can be separated into three categories: slow, medium and fast. This resulted in the following mapping function:

$$v(s) = \begin{cases} 0 & \text{if } 0.0 \leq |s| \leq 4.0 \\ 1 & \text{if } 4.0 < |s| \leq 7.0 \\ 2 & \text{if } |s| > 7.0 \end{cases} \tag{3.16}$$

Distance From Enemy. Distance to the enemy was previously used in Section 3.1.3.3), as part of the movement state representation. The same mapping could be used for targeting:

$$d(m) = \begin{cases} 0 & \text{if } 0 \leq m \leq 50 \\ 1 & \text{if } 50 < m \leq 150 \\ 2 & \text{if } 150 < m \leq 300 \\ 3 & \text{if } 300 < m \leq 450 \\ 4 & \text{if } 450 < m \leq 700 \\ 5 & \text{if } m > 700 \end{cases} \tag{3.11 revisted}$$

However, unlike for movement, large changes in distance have a very large effect on the action which should be taken. The further away the opponent is, the further in front of them the turret should be aimed, meaning distances of 450 and 600 require very different angles to be chosen. For this reason the following mapping is possible instead:

$$d(m) = \begin{cases} 0 & \text{if } 0 \leq m \leq 50 \\ 1 & \text{if } 50 < m \leq 150 \\ 2 & \text{if } 150 < m \leq 250 \\ 3 & \text{if } 250 < m \leq 375 \\ 4 & \text{if } 375 < m \leq 500 \\ 5 & \text{if } 500 < m \leq 600 \\ 6 & \text{if } 600 < m \leq 700 \\ 7 & \text{if } 700 < m \leq 800 \\ 8 & \text{if } m > 800 \end{cases} \tag{3.17}$$

This divides the distance as shown in Figure 3.7.

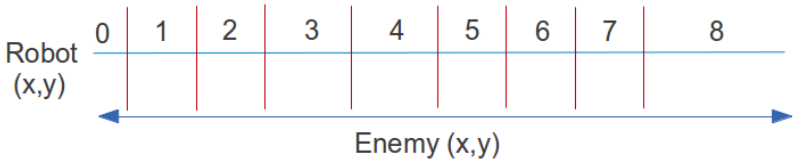


Figure 3.7: Distance to enemy segmentation for the targeting state that uses 9 bins.

By expanding the number of bins, possible distances can be much more accurately represented. This should allow the robot to more precisely select the angle at which to aim the turret.

Angle to the Enemy. The angle to the enemy was also used previously for the movement state representation, discussed in Section 3.1.3.3. It is a bearing taken relative to the robot's heading and so is within $\{-180^o, \ldots, 180^o\}$.

Instead of including the angle to the enemy in the state representation, it was decided that it would instead be used to form the actions. Any angles used would be offsets from the enemy's bearing, which allows full accuracy to be maintained. This will be discussed further when the action representation is defined in Section 3.1.4.4.

Result. Before segmentation was applied the number of possible states was 2.0736×10^9, which has been reduced to:

$$8 \times 3 \times 9 = 216 \tag{3.18}$$

This number of states is significantly lower than for the movement state representation and should allow the RL algorithm to converge in a fraction of the time.

3.1.4.4 Action Representation

Actions consist of an angle by which to offset the turret from the enemy's bearing and then the power that it should be fired with. With this in mind, allowing every possible angle between 0^o and 360^o would clearly be over zealous. Many angles would result in shooting a bullet in the opposite direction of where the enemy is. Once these nonsensical angles have been removed, segmentation can be used to limit the number of different angles further.

Aiming either too far in front or too far behind the enemy's direction of motion will result in a miss. A mapping can be used that allows a selection of different angles to be chosen either side of the opponent:

$$a = \begin{cases} 0 & \text{for } -16^o \\ 1 & \text{for } -12^o \\ 2 & \text{for } -8^o \\ 3 & \text{for } -4^o \\ 4 & \text{for } 0^o \\ 5 & \text{for } 4^o \\ 6 & \text{for } 8^o \\ 7 & \text{for } 12^o \\ 8 & \text{for } 16^o \end{cases} \tag{3.19}$$

The robot should learn to use negative angles when the opponent is reversing and positive angles when they are driving forwards. The angle of 0^o is also included in case the robot is pitted against an opponent which stops moving at any point.

Nielsen and Jensen found that maximum fire-power was best in almost all situations[18]. It was therefore decided that instead of the firepower being within the action representation, full firepower would simply be used. The only exception is when the robot's energy is so low that missing the shot would deplete the robot's energy to zero. Firepower was chosen based on the following equation:

$$bullet\ power = \min(3.0, robot\ energy) \tag{3.20}$$

3.1.4.5 Action Selection Method

The choice of action selection method was discussed in Section 3.1.3.5. As for movement control, the ε-greedy technique was chosen with ε set to 0.1, as it provides a good balance between exploration and exploitation.

3.1.4.6 Rewards

For the goal, *"Aim the turret so that when a bullet is fired it hits the opponent"*, rewards need to be given whenever a bullet hits something. In order to ensure that bullets manage to hit the opponent, negative rewards should be given upon the bullet missing and positive rewards given upon the bullet hitting. The three situations that can occur are: the bullet hits the opponent, the bullet hits a wall, the bullet hits another bullet in the air.

If the bullet hits the opponent then the agent has successfully chosen the correct angle to fire at. The agent needs to learn to favour actions that result in a hit and so the reward given was 10.

When a bullet hits a wall or another bullet it means that it has missed and the robot will lose energy. A neutral reward of 0 could be given, to represent no damage being done by the bullet. However, the robot should actively avoid angles that result in a miss as losing energy puts the robot closer to losing the round. To highlight this, a reward of -5 was given for a miss.

3.1.4.7 Implementation

Initialisation of the gunner involves setting up the agent, action selection method and Q-table. Due to there being no states or actions which needed higher $Q(s, a)$ estimates, all entries in the Q-table were set to zero to begin with.

As soon as initialisation has been performed and the opponent has been scanned, actions can be performed. Whenever a bullet hit or miss event occurs, the appropriate reward is given to the gunner, for RL updates to be based upon. The current state is determined using the information provided by the scanner, which the action selection policy then uses to select an action to take. The angle the action corresponds to is then added to the enemy's bearing. Once the turret has been rotated by this amount, it is fired.

The gunner keeps track of the reward received, passing the results to the logger when the battle ends. This data is outputted to a file as it was for the driver.

3.1.4.8 Alternative Hand-Coded Techniques

A variety of different hand-coded targeting techniques were implemented alongside the RL solution. It was decided that these would be: Linear Targeting, Circular Targeting, Head-On Targeting and Random Targeting. The only technique not already mentioned in the Technical Background (Chapter 2) is *Random Targeting*[31]. This technique works by calculating the arc which will result in hitting the opponent, based on bullet travel speed, and then choosing a random angle within that arc.

Adding the different hand-coded methods into the gunner design involved adding a conditional switch between the different guns when it was to be aimed and fired. As mentioned in Section 3.1.1, a properties file was used to choose between the different techniques.

3.1.5 Summary

Each of the components that make up the RL robot have been discussed, from design through to implementation. RL was not applied to every aspect of the robot, due to RL not being best to use if an optimal hand-coded approach already exists, as was the case with scanning. It is much more advantageous to focus on areas in which RL has the opportunity of improving over alternative approaches.

At many points there were multiple approaches that could be taken. Sometimes it was possible to decide between these options through analysing the problem and removing nonsensical solutions. However, this was not always possible and so the different options needed to be implemented and chosen between based on empirical results, which is often the case in ML. Experiments were ran to observe the results received with the different approaches, which are discussed in the Evaluation Chapter (Section 4.1).

3.2 Reinforcement Learning Framework Application

After completion of the RL robot, it was included in an RL framework application, named *RoboLearn*. Previous work on using games[63] and gaming frameworks[13] to teach AI has shown that they allow people to better grasp the complicated AI techniques. The intention was therefore for RoboLearn to do the same for RL by using Robocode as a learning environment. By designing the framework for running experiments, making adaptations to the algorithms and observing their performance under different conditions, RoboLearn has a very different focus to the previous examples. This should allow people to learn about RL through experimenting with the algorithms on the Robocode environment.

Playing created robots in Robocode through the framework required various changes to be made to their design. Development of RoboLearn included the following sections: ability to create, store and display the robots; ability to control Robocode by running battles; collecting of results; plotting of results in real-time and help pages to explain all of the RL concepts.

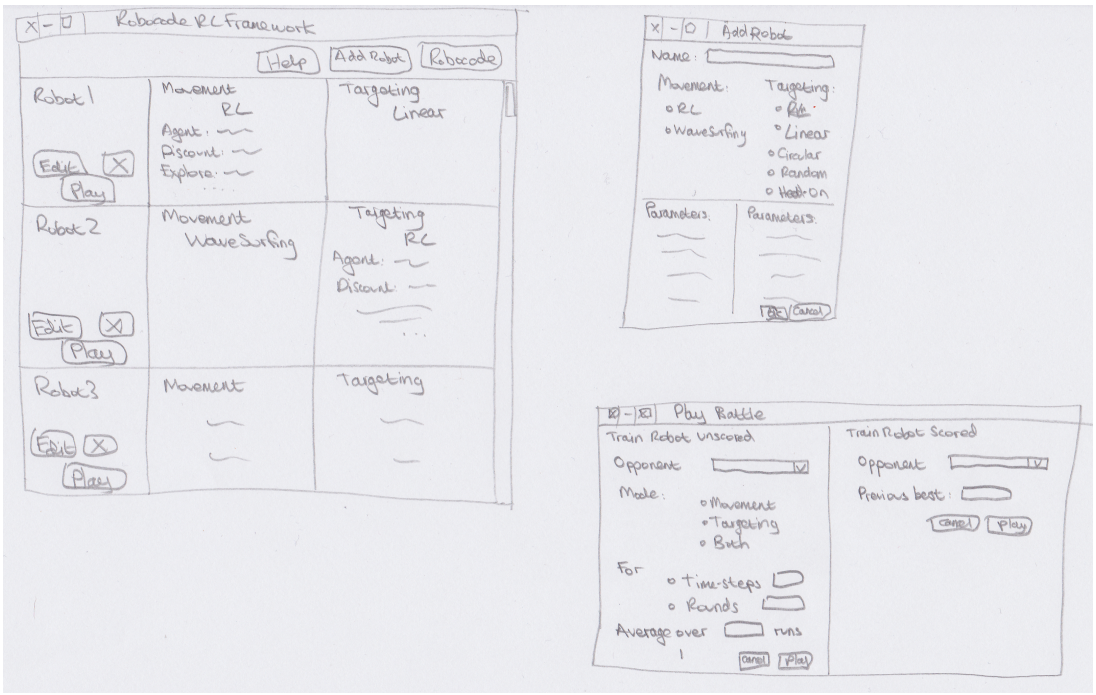
3.2.1 GUI

To allow access to the functionality of RoboLearn through a GUI the first main task involved designing how it would look. Due to the project objectives stating that the application should be user-friendly, it was important that the layout made it easy to use and understand. To achieve this, it was decided that the main window would contain two areas; a panel of buttons across the top to perform all the main actions and below it a list of all the robots which have been created.

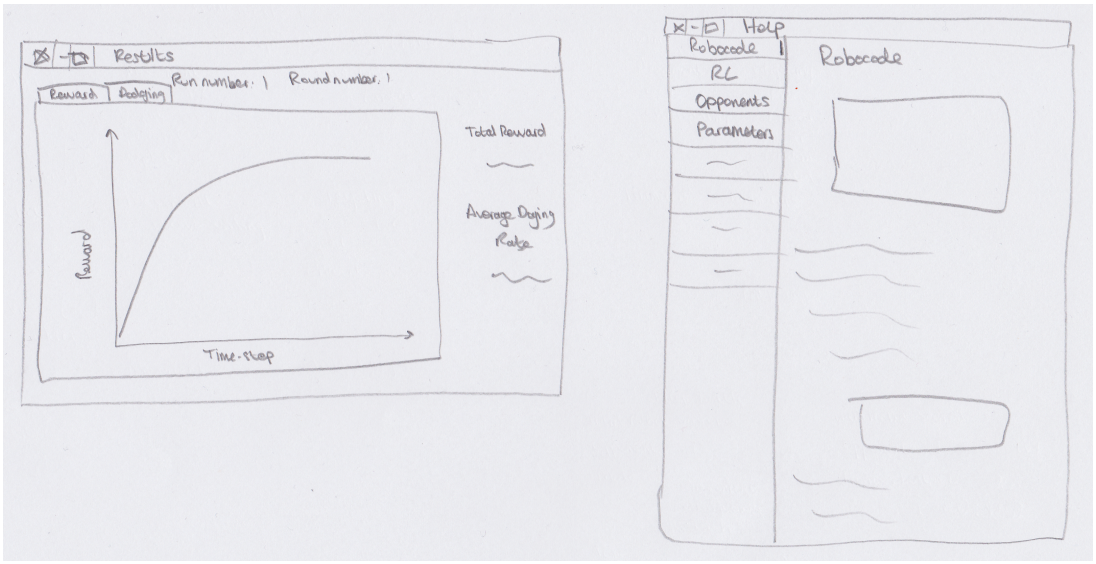
All the information relating to a robot and actions to be performed upon them are found within their own panel. Then, all the general actions to perform within the framework are contained in the button panel across the top. This ensures that the user knows where to go when they want to complete a specific task.

Along with the main interface, a series of additional windows were needed for: robot creation, robot modification, running battles, displaying results received from Robocode and help pages containing information about RL and Robocode. Whenever settings needed to be selected, controls such as drop-down lists and radio buttons were preferred over text boxes, due to them making the task of data entry quicker and easier for the user.

Sketched designs were produced for the windows that would make up the GUI, which can be seen in Figure 3.8. Once the designs had been produced, the GUI was developed to match. During development a small number of minor changes were made to the original designs. These involved removing un-needed buttons and altering text used for labels and titles. Using the sketches ensured that the final GUI produced matched with how it was originally envisioned.



(a) Main, create robot and play battle windows.



(b) Results and help windows.

Figure 3.8: Sketched designs of the RoboLearn windows. The final GUI was developed to match the designs, with minor changes.

3.2.2 Robot Creation

Once the graphical front-end had been produced, the functionality behind it needed to be implemented. The first task was robot creation, which would involve collecting the settings chosen by the user and storing these in a project file. For this to be possible, a file format needed to be decided upon, with the options including: XML, plain-text CSV or a database.

1. **CSV:** is the most basic option, where all the data for each robot is stored together on one line of the file. Each parameter or setting is then separated using a delimiter, such as commas or semicolons. This has the benefit of being a very simple format to create and parse, but it does not allow for relationships to be formed between data items through structuring the data.
2. **XML:** uses tags stored in a tree-like structure. This means that structured data is allowed, maintaining the relationships between data items. All the data for each robot is stored within a tag, where each aspect of the robot is a child tag. This makes it simpler to create, parse and maintain.
3. **Database:** such as SQLite. Databases are much more complicated than the previous options, however, they provide many more advanced features. These include the option to run queries on the data.

It was decided that the extra features provided by a database would not be required and that implementing one would be excessive considering the amount of data being stored. Therefore, XML was the best option, due to it providing better parsing and storing of data than CSV, whilst being flexible and requiring much less maintenance than a full database.

An XML parser was developed, to handle maintenance of the project file in which all robots would be stored. Methods were included to allow: searching for specific tags, finding child tags and allowing the contents of the XML to be edited. These were needed when collecting the information for a certain robot and for altering robot parameters.

The XML for an example robot with RL movement and Linear Targeting can be seen in Listing 3.2. A clear structure is used, with all the data for the robot being stored within a single tag. Inside this there is a tag to contain all of the movement parameters and the targeting parameters. It can be seen in Listing 3.2 that due to the robot using Linear Targeting, the targeting parameters tag is therefore empty. The project file collects together all of the various robot tags.

```
<robot>
        <name>TestRobot1</name>
        <driver>rl</driver>
        <movement>
                <gamma>0.66</gamma>
                <policy>egreedy</policy>
                <tau>0.10</tau>
                <agent>ql</agent>
                <epsilon>0.42</epsilon>
                <initialq>0.00</initialq>
        </movement>
        <gunner>linear</gunner>
        <targeting/>
</robot>
```

Listing 3.2: XML for an example robot in RoboLearn that uses RL movement and Linear Targeting. The XML would then be included in the main project file.

The robot creation window is used to create a robot from which the XML is generated. A screen capture of this creation window is displayed in Figure 3.9.

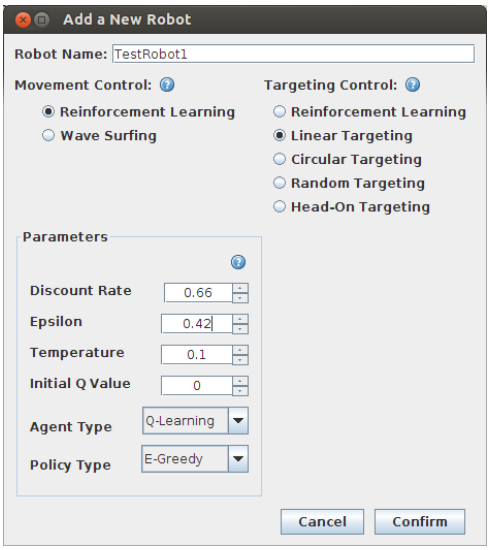


Figure 3.9: Screen capture of the robot creation window.

3.2.3 Displaying Robots

Whenever a new robot is created, a panel also appears for it in the GUI, which was designed along with the main interface. The panel includes: the name of the robot, movement settings, targeting settings and then a set of buttons to perform actions on the robot. Using the panel the robot can be edited, deleted or played in Robocode. Whenever a robot is added, modified or removed, an event is fired within the framework that makes the necessary changes to both the GUI and the XML project file. This ensures that both are always updated and correct. When the application runs, the project file is automatically loaded up, meaning that RoboLearn is ready to use straight away. A screen capture of the main window can be seen in Figure 3.10.

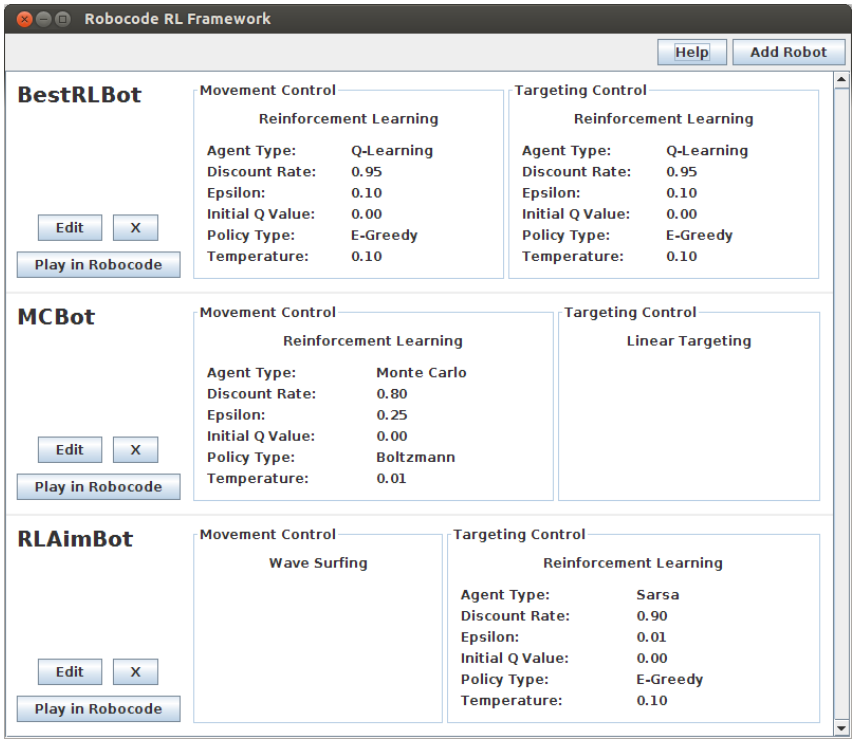


Figure 3.10: Screen capture of the main RoboLearn window.

3.2.4 Robocode Control

The Robocode control API allows Robocode to be executed and controlled from an external application [34]. Rather than requiring users to install and set up an installation of Robocode, one was included within RoboLearn, which could then be loaded by the Robocode engine. Using it this way involved supplying a specification for the battle, describing both the battlefield and the robots to be included. To allow results to be collected, a custom `BattleObserver` was developed, which was supplied to the Robocode engine to listen for battle events which are fired.

The robot specification supplied to the Robocode engine includes the package names of the robots to include, which means they must already be present in the Robocode robot database. This posed a potential problem, as it would not be possible for RoboLearn to add new robots to the robot database file. An alternative method for playing the created robots in Robocode therefore needed to be devised. The RL robot can be configured through a properties file, so every parameter which could be altered was made tunable by one of the properties. Instead of having a separate compiled robot for each one in RoboLearn, the main RL robot was added to the robot database through the Robocode GUI application. The robots could then be included in a battle by selecting the RL robot and copying their settings to the properties file.

When running RL experiments it is important to be able to plot the reward and other statistics over time to observe the learning process. The number of time-steps that have occurred are counted within the RL robot, meaning that the statistics along with time-steps must be transferred to RoboLearn. Then when the correct number of time-steps have passed, the battle can be ended. Support for communication between the external application and one of the robots is well beyond the capabilities of the control API available, so a method of achieving this needed to be designed. A feature of the Java programming language is that system properties can be set[19], so the robot could signal RoboLearn or pass data to it through them. To retrieve the data, RoboLearn checks the appropriate properties within the battle listener after each turn has completed.

When a battle is started through RoboLearn the user has the choice of specifying a number of rounds or a number of time-steps. If rounds are chosen, then the battle will be ended by the engine automatically upon that number being completed. However, if time-steps are selected, the battle listener must abort the battle upon the correct number of time-steps being reached. The Robocode API was also not designed for battles to be aborted within the battle listener. It was discovered that it was only possible when a thread was executed which aborted the battle instead.

3.2.5 Collecting and Plotting Results

Rather than just displaying the results at the end of the battle, it was decided that they would appear in real-time as it plays. To collect this information, the RL robot signals that a new result is available through the system properties and then sets another to the result value. RoboLearn checks the properties repeatedly so that as soon as it is signalled it can collect the data. This form of communication is not required to compute the dodging rate of the driver and the hit rate of the gunner, but only for the reward. This is because the battle listener is able to iterate over all of the bullets that have been fired every turn to check if they hit or miss.

Once results are available the battle listener sends them out through an event to any source that is waiting for them, one of which is the results window. Upon receiving a result event, the results window plots the data on a graph and then displays running totals and averages beside it. To handle the plotting, the library JFreeChart was used[6]. The library allows a data set to be linked to a specific graph panel, so that new data can be added by simply altering the data set.

When producing results, it is common to run an experiment multiple times and average the data collected from each. Support for doing this directly in RoboLearn was added, by asking for the number of runs to average over when starting a battle. For the first run, results are plotted exactly as they were before, but for the other runs, the user will see the graph update and average in real-time. The result for a given time-step is calculated as the average of those taken from each run for that time-step.

A screen capture of results being plotted can be seen in Figure 3.11.

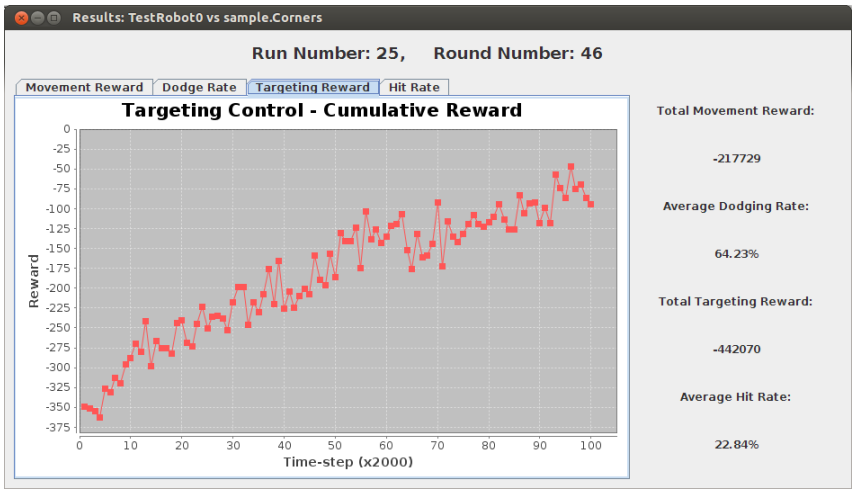


Figure 3.11: Screen capture of the results window. When results are received they are plotted and included in the overall statistics shown on the right.

3.2.6 Help and Information Pages

The objectives stated that RoboLearn should be both user-friendly and educational to use. To make it educational the user was provided with a set of help and information pages to teach them about RL. Icons were placed next to each RL concept, so that the user could use it as a shortcut to the appropriate help page to find out more. For example, clicking the icon next to the RL parameters when creating a robot, would navigate to a page explaining the function of each parameter.

The help and information pages were written in HTML and then displayed in the window using an XHTML renderer called Flying Saucer[61]. Using HTML allowed pages to be laid out quickly and easily, including text formatting and images. The help page on Robocode can be seen in Figure 3.12.

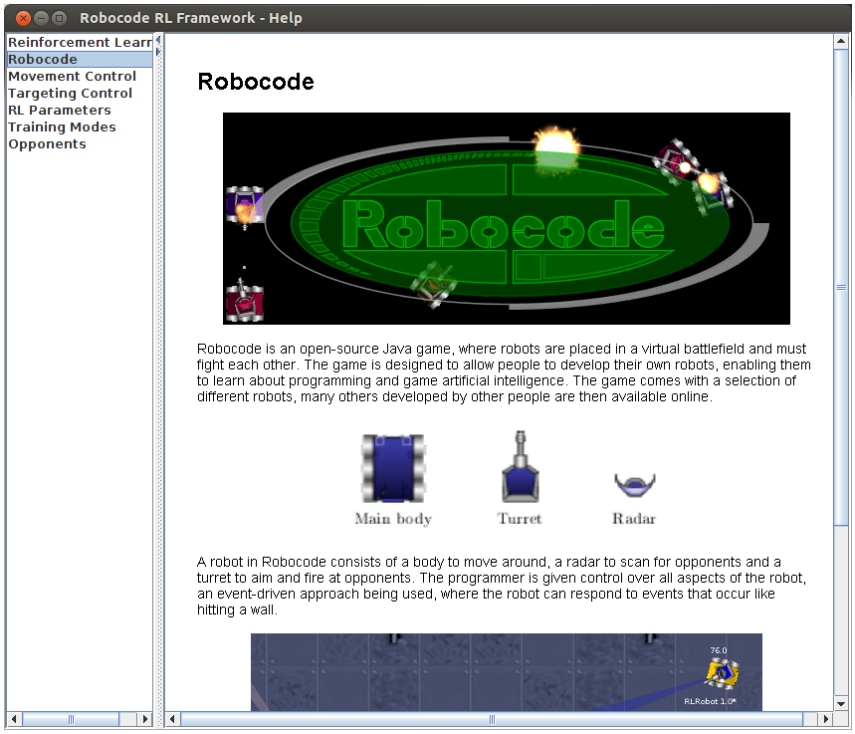


Figure 3.12: Screen capture of the help and information window, displaying the page containing details of Robocode. Help pages can be selected by clicking a topic on the left-hand side.

3.3 Summary

To design the RL implementations many decisions were made, with a focus given to reducing the total number of possible states through the amount of information used in the state and action representations. Techniques such as segmentation and removing redundant parameters were used to reduce the state-action space. This had the effect of speeding up learning and allowing the RL algorithm to converge in a reasonable amount of time.

Following the RL robot, the RL framework application, RoboLearn, was created. It is a user-friendly and educational way of creating RL robots for the Robocode environment. The framework allows users to run RL experiments on Robocode, whilst learning more about RL in the process.

To determine how successful the RL robot and RoboLearn are, they will both need to be thoroughly evaluated. This will involve exploring the parameters and algorithms used. This evaluation can be found in the next chapter.

Chapter 4

Critical Evaluation

This chapter evaluates both the RL robot and RoboLearn, the RL framework application. Alternative options will be compared for any design choices that were made and then the overall performance results achieved will discussed. These will allow the extent to which the project has been a success to be determined.

4.1 Reinforcement Learning Robot

First, the RL robot will be evaluated, which will involve a sequence of experiments exploring the reward received with a variety of different robot set-ups. The overall performance achieved by the robot will then be compared to that managed by the hand-coded robots `Corners` and `LinearWave`.

The Robocode environment is stochastic, which means that any results achieved are not identical for each run of the experiment. To account for this, experimental results are obtained by averaging over 50 runs, to ensure that the performance of the robot is fairly represented. Unless otherwise stated, results are collected during each window of 2000 time-steps up to a total of 200,000 time-steps. For example to calculate reward for the first result, the reward is summed for 2000 time-steps and then stored or plotted.

Many of the experiments involve comparing different parameter values or algorithmic set-ups. However, when this is not the case there is a default set-up for the robot that is used. Both movement and targeting are controlled by RL using Q-Learning and ε-greedy action selection. The RL parameters used are:

- Discount rate, $\gamma = 0.95$
- Exploration rate, $\varepsilon = 0.1$
- Learning rate, $\alpha =$ dynamic, based on the number of visits to the state-action pair (Equation 3.1).

$$\alpha_t = \frac{1}{1 + visits(s_t, a_t)} \qquad \text{(3.1 revisted)}$$

During the evaluation, the robot will be played against a selection of different hand-coded robots. These include: `LinearWave`, `Corners` and `Stationary`.

- **LinearWave:** is a combination of Wave Surfing movement and Linear Targeting. It is the most advanced robot used in the project, due to Wave Surfing being very efficient at dodging incoming bullets. Linear Targeting is also very accurate at predicting future enemy positions using their heading and velocity.
- **Corners:** is a simple robot which moves towards one of the corners. Once there it rotates its turret to face an opponent and fires. Upon being hit by a bullet, it stops, moves to the next corner and resumes firing. This behaviour is then repeated until the round ends.

- **Stationary:** is a robot that does nothing, so it is stationary throughout the battle and does not fire at its opponents.

Wave Surfing is one of the current most popular movement techniques and by 2010 each of the top 40 best robots in the world used a form of it[40]. For this reason, except in experiments against a variety of opponents, the default opponent is `LinearWave`. It is expected that training the RL robot against these techniques will demonstrate its overall performance and prepare the robot for battles against the other robots available.

Movement and targeting both exist as separate RL implementations within the robot. For this reason, they are trained independently, to ensure the performance of one cannot affect the other in any way. When learning movement, aiming and firing of the turret is disabled and when learning targeting, Wave Surfing is used to control the robot's motion.

4.1.1 Reward

The first experiment involves collecting the reward received by the RL robot for both movement and targeting. Whenever the robot receives a reward it is added to a total, which is then published and reset back to zero every 2000 time-steps. Intervals of 2000 time-steps were chosen for the reward as they provide good clarity of results. By collecting this information, rather than observing just the quality of the final policy learned, the speed at which the RL implementations improve can be observed. Due to learning taking place against a new opponent without any prior knowledge, the speed at which the RL agent can improve is a strong indicator on its effectiveness. A later experiment will explore pre-training the $Q(s, a)$ estimates, judging purely the quality of the final policy learned (Section 4.1.6).

RL movement was trained using the default set-up to produce the rewards seen in Figure 4.1.

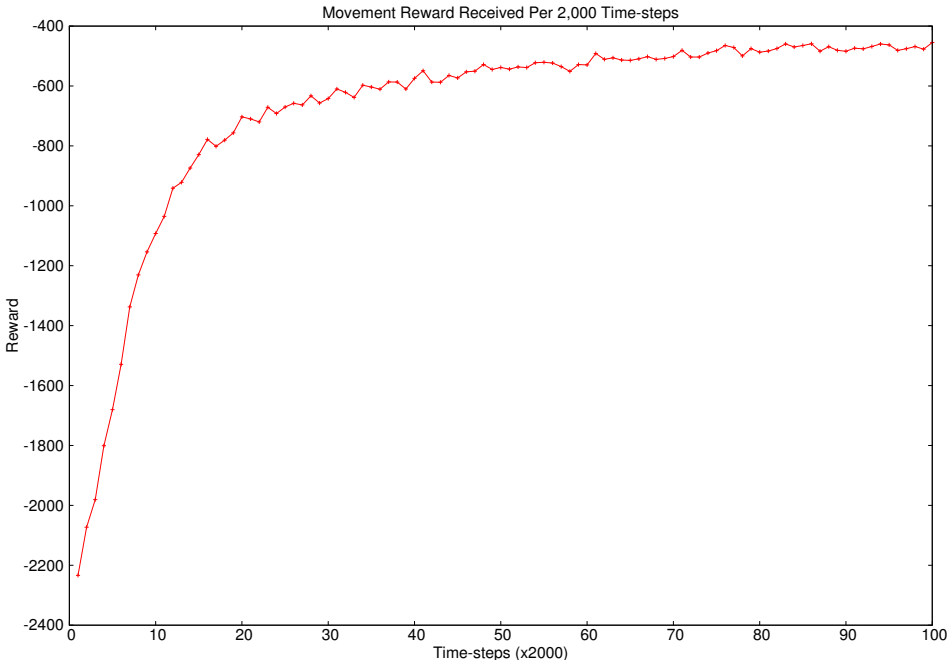


Figure 4.1: Reward received by the movement controller when using Q-Learning for movement only and trained against `LinearWave`. Q-Learning is able to adapt to the robot's movement and increase its reward over time.

The reward quickly rises to -941 in 24,000 time-steps, before the rate of increase slows for the remainder of the experiment. Even at the slower rate, the reward still manages to increase much further to -455 after the full 200,000 time-steps. Towards the end of the experiment, the reward appears to stop changing, showing that it is maximising the reward. The initial learning phase (sharp increase in reward), is due to the agent learning to avoid the damage from colliding with the walls or the opponent. Taking damage in these ways carries a lower reward and is easier to avoid, so the robot can learn to achieve this quickly. Afterwards, the agent has to focus on learning to counter the opponent's Linear Targeting technique,

which is much more complicated. For this reason, the learning process slows down, however, a large improvement is made regardless.

Maximising the reward is the central aim of any RL implementation, which this experiment has shown to be occurring after 200,000 time-steps. This has therefore demonstrated the RL movement implementation to be capable of learning effectively.

Next, RL targeting was trained and the reward collected, which is shown in Figure 4.2 as the red series.

Figure 4.2: Reward received by the targeting controller when using Q-Learning for targeting only and trained against `LinearWave`, `Corners` and `Stationary`. Q-Learning does not improve its targeting of `LinearWave` but is able to improve against the simpler `Corners` and `Stationary`.

For `LinearWave`, as shown by Figure 4.2 the reward does not increase to any significant extent, it merely varies around -480. Clearly the RL algorithm fails to learn to counter the opponent's Wave Surfing movement, which unfortunately suggests either the implementation is incorrect or it is just insufficient to defeat Wave Surfing. To explore this further, the experiment was ran against two opponents with simpler movement behaviours: `Corners`, which moves towards the corners of the battlefield, and `Stationary`, which does not move. The results against these two opponents are the green and blue series in Figure 4.2.

In general the agent receives more reward against the stationary robot (blue series) and `Corners` (green series). They are both much simpler than Wave Surfing, making it easier to hit them and predict what they will do in the future. The RL implementation manages to learn how to oppose the movement strategies better than it did for Wave Surfing, shown by the increasing reward throughout. However, there are many more sharp increases and decreases in the reward, with the most being seen against the stationary robot. This suggests that the agent is finding states it has not visited before much more often and is taking bad actions from them. A function approximator may be able to help solve this problem, but it is beyond the scope of this project (Section 5.3.3). Regardless, the increase in reward demonstrates that RL implementation is able to learn and improve its targeting policy, but that it is not good enough to do so against the advanced Wave Surfing method.

4.1.2 Q-Learning, Sarsa and Monte Carlo Methods

One aspect of designing the RL implementations for movement and targeting was selecting the algorithm which would be used. Therefore, the next experiment makes a comparison between the available options:

Q-Learning, Sarsa and MC methods. This will allow the most suitable algorithm for the Robocode environment to be verified. Conclusions will be drawn on the RL algorithms in general, so will apply to both movement and targeting. Therefore, as movement showed better improvement of the reward against `LinearWave` than targeting did, only results achieved by movement will be used to perform the comparison.

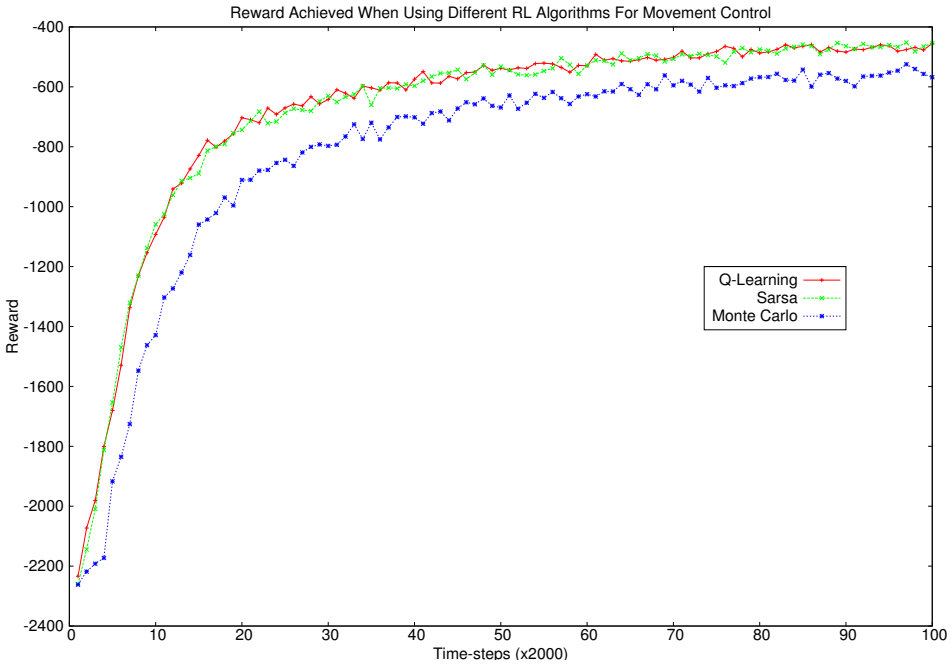


Figure 4.3: Reward received by the movement controller when three RL algorithms are used to train just movement against `LinearWave`. The results achieved by Q-Learning and Sarsa are essentially the same and outperform MC Methods.

Figure 4.3 shows that both Q-Learning (red series) and Sarsa (green series) produce very similar results, a rapid increase in reward up to 22,000 time-steps before slowing down. Relative performance of the Q-Learning and Sarsa algorithms differs for each problem being solved, due to the differences between on-policy and off-policy learning. Q-Learning is off-policy, so it updates a different policy to the one being followed for behaviour, often resulting in many optimistic exploratory actions being taken. This can be very detrimental to the performance of the agent in many environments, however, in Robocode the large number of moves taken before the end of each round means that the robot has the opportunity to recover from any bad moves taken. It can be seen that both Q-Learning and Sarsa achieve almost the same performance within Robocode, with the only exception being that the results for Sarsa are slightly less stable (more jumps in the reward). This instability is caused by Sarsa learning a policy that maximises the reward whilst still exploring, which means whenever the exploratory moves are worse and result in less reward, the overall reward drops.

It can be seen in Figure 4.3 that the initial learning phase for MC methods (blue series) is slightly shallower than for the other algorithms. MC methods only perform updates to the value estimates at the end of an episode (the round), rather than after each time-step. This means that actions taken during a round cannot take advantage of updated estimates, making improvement slower than if updates are made continuously. Exploration is a real problem for RL, with it not being possible to estimate the true action values without it. The reward achieved by MC methods is lower in general showing that the policy that is learnt is not as good as those produced by Q-Learning and Sarsa.

The results clearly show that the difference between Q-Learning and Sarsa is minimal and MC methods produce worse performance than both. Q-Learning was chosen due to the reward being slightly more stable than for Sarsa.

4.1.3 Action Selection Methods and Exploration

To ensure that sufficient exploration of the state-action space is performed, an action selection method needed to be chosen. The next experiment contrasts the ε-greedy and Boltzmann Softmax techniques, which select actions in a slightly different way. Exploration is controlled in ε-greedy through the ε parameter (Section 2.2.2.1), whereas Boltzmann Softmax uses τ (Section 2.2.2.2). The reward that is received by the two different methods will be compared, with various different values of ε and τ. For the same reason as in the previous experiment (Section 4.1.2), movement alone will be used to perform the comparison.

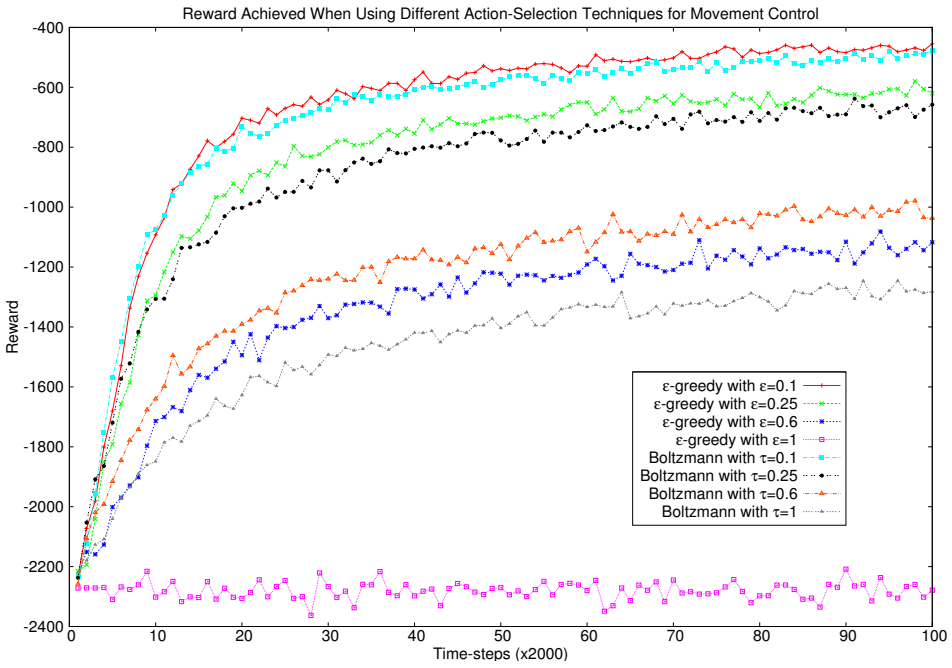


Figure 4.4: Reward received by the movement controller when using Q-Learning for movement only, with different action selection techniques and trained against `LinearWave`. Q-Learning learns the most when using ε and an exploration rate of 0.1.

The graph presented in Figure 4.4 demonstrates the reward received when using both ε-greedy and Softmax Boltzmann, with values of 0.1, 0.25, 0.6 and 1.0 for the ε and τ parameters. It can be seen that the series which is above the rest corresponds to ε-greedy using an ε value of 0.1 (red series). The effect of increasing ε can be seen for values of 0.25 (green series), 0.6 (dark blue series) and 1 (pink series). When more exploration takes place (higher ε), lower overall performance is achieved. This is shown by a slower rate of increase in the initial learning phase and the reward settling at a lower value. Also, the reward becomes more unstable, with increasing numbers of jumps in the reward. Exploratory moves are randomly selected, meaning that they could potentially be bad moves to take and result in less reward to be received. For this reason the ε value needs to capture a balance between exploration and exploitation, which 0.1 is seen to do the best.

Increasing the value of τ for Softmax Boltzmann has a similar effect as increasing the value of ε for ε-greedy. Higher values of τ result in the reward increasing slower and beginning to converge at a lower reward. As $\tau \to 0$, action selection becomes increasingly greedy, whereas higher values of τ will result in more exploratory behaviour. Therefore, as τ increases the agent receives less reward, due to more bad actions being taken as the agent explores.

With ε and τ set to 0.1, in the initial learning phase Boltzmann performs better than ε-greedy, through the reward increasing at a faster rate. In this phase the agent is mainly learning to avoid wall and opponent collisions, so there is a large difference in reward when a collision occurs to when one does not. This results in the same difference to be seen in the $Q(s, a)$ estimates. When Boltzmann assigns probabilities to each action, the differences in the estimates will ensure that actions that do not lead to a collision are more likely to be selected. Whereas for ε-greedy when taking an exploratory move, each action is just as likely, which could result in less reward being received. After the reward increase slows,

the same does not occur as instead ε-greedy performs much better. When learning to avoid enemy bullets, taking the same action in a given state does not result in the same new state, due to the non-determinism of the Robocode environment. This means the difference in the $Q(s, a)$ estimates for an action which results in being hit and one which does not is less distinct, which makes Boltzmann less effective than before.

For the Robocode environment, it can be seen that the best performance is achieved by ε-greedy with ε as 0.1, hence it was chosen to be used.

4.1.4 Movement Component State Representation

When designing the RL movement state and action representations, there were often multiple possible approaches that were considered. The next experiment consists of various parts, each exploring a different aspect of the movement component's state and action representations.

4.1.4.1 Whether the Enemy Has Fired.

When designing the movement state representation in Section 3.1.3.3, a possible parameter to include was whether the enemy had just fired or not. The parameter could enable the robot to learn to avoid bullets, although this would likely require the bullet trajectories. To decide whether it should be included, the reward received when training movement control with the parameter will be compared to the reward collected without it.

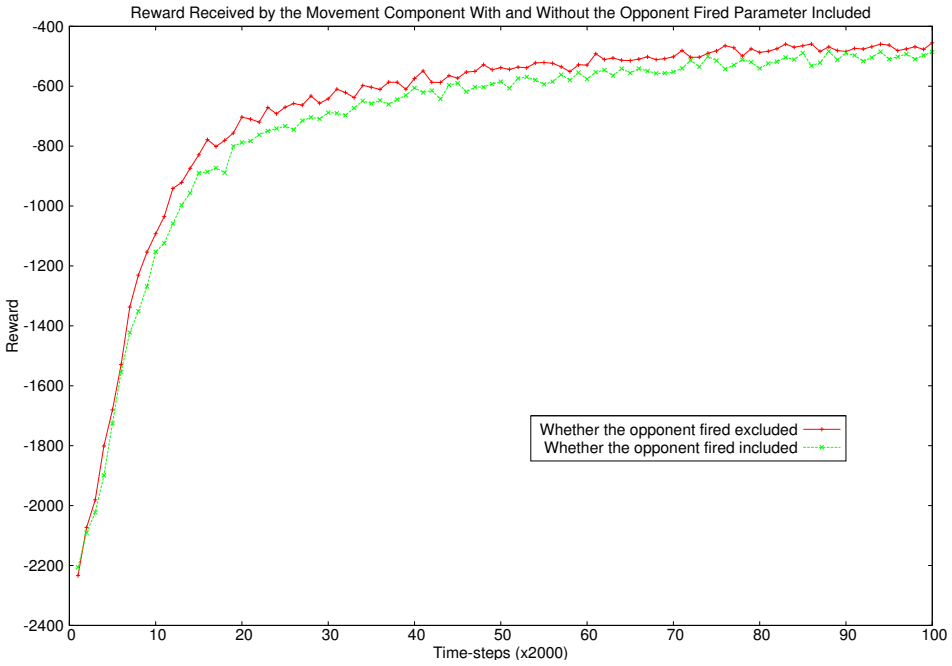


Figure 4.5: Reward received by the movement controller when using Q-Learning for movement only, with and without the opponent has fired parameter and trained against `LinearWave`. Q-Learning improves less when the parameter is included.

Figure 4.5 shows that initially the same rate of improvement is achieved with or without the parameter. During this learning phase the agent is learning to avoid colliding with the walls, for which whether the opponent has fired or not has no effect. However, after 14,000 time-steps the agent begins to learn slower with the extra parameter, eventually beginning to converge around a lower reward.

This shows that once the agent starts to learn how to avoid the opponent's bullets, whether the opponent has fired or not being included results in slower learning and worse actions being taken. A bullet takes time to travel across the battlefield, meaning that moving as soon as it is fired will not necessarily allow the robot to avoid it. By doing so the robot is simply more likely to drive into a different bullet and

receive a lower reward. If the parameter has no bearing on which actions the agent should select and the agent cannot learn to ignore it, then its presence will simply result in worse actions being chosen. Due to the agent receiving less reward, it demonstrates that the extra parameter results in a lower quality policy being learned, in which worse actions are taken. Due to the lower performance achieved, whether the opponent has just fired was not included in the movement state representation.

4.1.4.2 Position.

To limit the number of possible values for the position in the movement state representation (Section 3.1.3.3), segmentation was used. There were three different segmentations suggested that put the position into either 5, 9 or 48 different bins. It was anticipated that the 5-bin segmentation would not adequately segment all areas of the battlefield, the 48-bin representation would have too many redundant bins and that the 9-bin version would provide a good balance between the two. To determine if this is the case, the reward will be collected when training movement control with each of the segmentations.

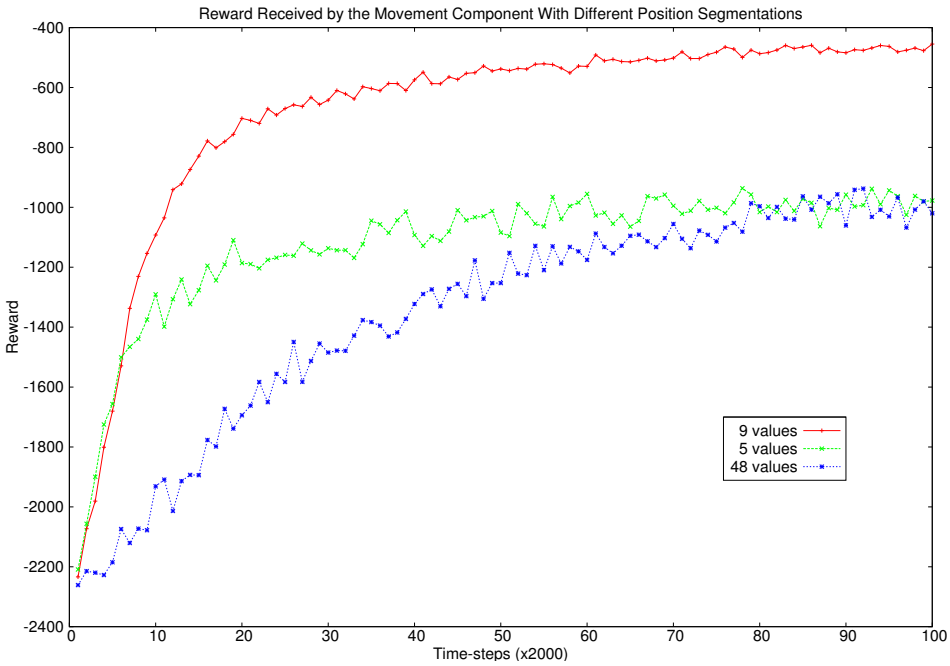


Figure 4.6: Reward received by the movement controller when using Q-Learning for movement only, with three different position segmentations and trained against `LinearWave`. Q-Learning improves the most when the 9-bin segmentation is used, the others result in less reward being collected.

The graph seen in Figure 4.6 demonstrates that relative performance of the three segmentations is as predicted. Using the lowest number of bins (green series) allows reward to increase initially at the same rate as for the 9-bin version, however, after 14,000 time-steps it begins to converge around a much lower value. Without adequately segmenting the corners of the battlefield, the agent cannot learn properly what to do and is more likely to drive into the wall. This means that after partially learning how to avoid the walls and the opponent, improvement halts and begins to converge with a lower quality policy. The difference in reward between using the 9-bin segmentation and the 5-bin version, shows the importance of properly representing all of the areas of the battlefield.

Conversely, when using the much larger 48-bin segmentation (blue series) there is no initial learning phase of sharp improvement. Instead the reward increases at a more gradual pace and then starts to converge around the same reward as with 5 bins. There are significantly more possible states the agent could be in when using the larger segmentation, which causes a much slower learning process. When the agent has to update so many extra states, the $Q(s, a)$ estimates will improve at a significantly lower rate and require many more moves to be taken. This means that it is only worthwhile to add extra possible states, if they would require the agent to select different moves. Additionally, in many of these states the robot would potentially want to perform the same action, which shows how much redundancy is contained within

the segmentation. This means the agent is working harder to update all of the states, whilst gaining no benefit from their presence.

The 9-bin segmentation receives significantly more reward than the two alternatives and also achieves the fastest improvement in reward. For these reasons it was chosen to represent the robot's position.

4.1.4.3 Distance to the Enemy.

When designing the segmentations to express the distance to the opponent in Section 3.1.3.3, two alternatives were put forward, consisting of either 6 or 10 bins. It is preferable in RL to reduce the number of possible states to be speed up learning, so it was envisaged that the 6-bin segmentation would produce the best performance. To guarantee this, the reward will be recorded when training movement control with each of the segmentations.

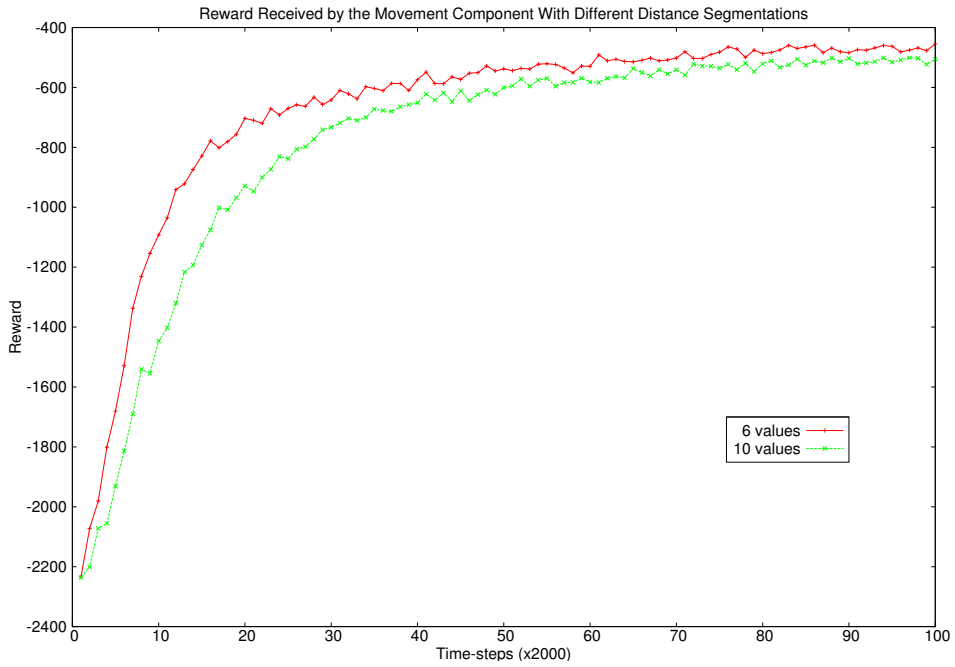


Figure 4.7: Reward received by the movement controller when using Q-Learning for movement only, with two different distance segmentations and trained against `LinearWave`. Q-Learning improves the most when the 6-bin segmentation is used.

Figure 4.7 illustrates how the reward can be improved at a faster rate with less bins (red series). Even though the larger segmentation allows the agent to distinguish between a greater number of distances, it also results in more information being stored. The agent is unable to learn at the same speed when it has so many more states to optimise the action value estimates for. Therefore, the 6-bin segmentation was used.

4.1.5 Targeting Component State

4.1.5.1 Distance to the Enemy

For the targeting component distance plays a larger role than for movement. This is because when aiming the turret, how far away the enemy is will strongly affect the angle it should be aimed in order to hit. When using distance for movement in Section 3.1.3.3 it was expected that using a smaller segmentation would allow the reward to improve more quickly. In Section 3.1.4.3 two different segmentations were proposed for encoding the distance to the opponent in the targeting state representation, using 6 bins and 9 bins. Contrary to movement, it was believed that using the larger 9-bin segmentation would allow the agent to better determine the angle to aim the turret than with the 6-bin segmentation.

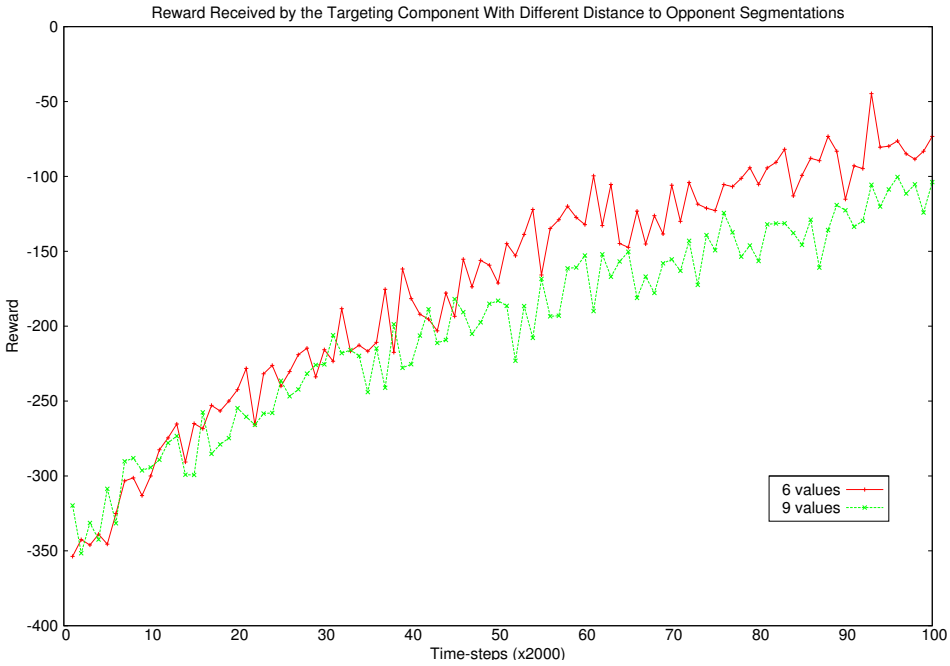


Figure 4.8: Reward received by the targeting controller when using Q-Learning for targeting only, with two different distance segmentations and trained against `Corners`. Q-Learning improves the most when the 6-bin segmentation is used, contrary to what was expected.

The results in Figure 4.8 contradict this by showing that, as for movement in Section 4.1.4.3, using the smaller segmentation entitles the agent to reach a higher reward than it does when using more bins. Similar performance is achieved by both for the first 60,000 time-steps, however, after this point the 9-bin segmentation starts to improve at a slower rate. The reward achieved by the two representations is much closer than it was for movement control, showing how distance is more important to the targeting component. The extra information added to the state representation unfortunately still slows down the learning process, making the smaller segmentation more effective. For this reason the 6-bin version was used.

4.1.6 Pre-Trained Action-Value Estimates

Once the robot has been trained, the computed Q-table can be saved to a file so that it can be loaded at the beginning of a later battle, giving the agent its previously learned knowledge. If this is performed and the agent selects only greedy moves, instead of including exploration, the quality of the agent's learned policy can be determined. Pre-training of the $Q(s, a)$ estimates is needed to play the robot against hand-coded opponents (Section 4.1.8) and a human (Section 4.1.9) using its policy learned in advance. The next experiment observes the effect on the reward received by the movement component of using the pre-trained policy. $Q(s, a)$ estimates that are loaded in are trained for 200,000 time-steps beforehand.

Figure 4.9 demonstrates that starting with pre-trained $Q(s, a)$ estimates causes more reward to be received from the beginning. The knowledge that the robot starts with enables it to select the high quality actions, without having to discover them through experience. Taking purely greedy actions also allows the robot to fully exploit all the gained knowledge without taking exploratory actions that could produce less reward to be received. The variations in the reward, shown by sharp drops and increases, relate to the robot finding itself in states it had not come across very often or at all during the pre-training phase. This means that it will likely select poorer actions without being able to improve the $Q(s, a)$ values of the available actions through exploration.

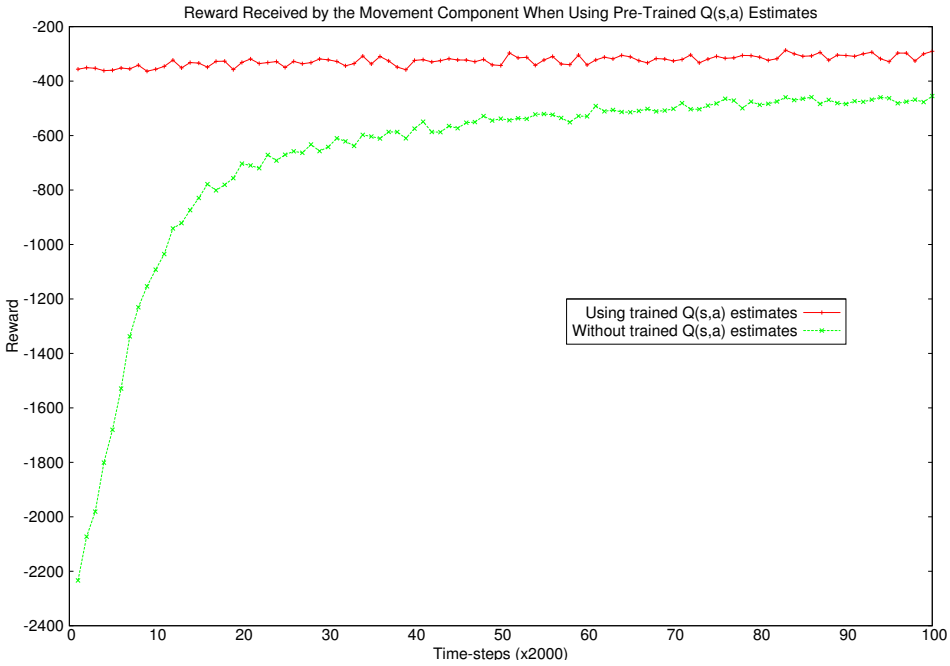


Figure 4.9: Reward received by the movement controller when using Q-Learning with and without pre-trained $Q(s, a)$ estimates and trained against `LinearWave`. Q-Learning collects more reward from the beginning when pre-trained $Q(s, a)$ estimates are used.

To visualise the actions that the final trained policy chooses to take, a battle was viewed in real-time for the RL robot against `LinearWave`. The first observation to make is that the movement control policy is very effective at dodging the Linear Targeting gun. After each move forwards it performs, it turns sharply, which means the direction it moves in often changes. At many points the path it travels also becomes curved or circular. Due to the robot taking preference to actions that alter its movement direction, the enemy's Linear Targeting gun often fires bullets either side of the RL robot. In many rounds this results in `LinearWave` being disabled, due to spending all of its energy. The policy produced by RL targeting was also observed and it was found that it manages to adapt and learn to fire bullets in front of the opponent's direction of motion. However, the Wave Surfing technique is very efficient at altering its movements and RL targeting does not manage to cope with this.

4.1.7 Multiple Reinforcement Learning Implementations

Within the RL robot there is two different RL implementations, one controlling movement and the other in charge of targeting. As mentioned at the start of the robot evaluation in Section 4.1, each is trained independently of the other to ensure that one does not affect the performance of the other. For the next experiment, the impact of training multiple RL implementations simultaneously will be investigated. This involves surveying the performance of the movement and targeting components when they are training at the same time, with the aim of discovering any differences in the learning process.

By default, the movement component is trained with targeting deactivated. For the first part of the experiment, this will be compared to when is targeting is controlled at the same time by RL and Linear Targeting.

The results presented in Figure 4.10 reveal that the type of targeting being used has a strong effect on the reward obtained by the RL movement implementation. Using either form of targeting means that, for the first 14,000 time-steps, more reward is received. At this point the reward achieved when using no targeting continues to increase at a much higher rate than using either of the targeting strategies. For the entirety of the experiment, using RL targeting and Linear Targeting results in the same improvement speed, but with less reward being collected when RL targeting is used. The best overall performance is accomplished with targeting deactivated, due to the very short time in which RL and Linear Targeting outperform it.

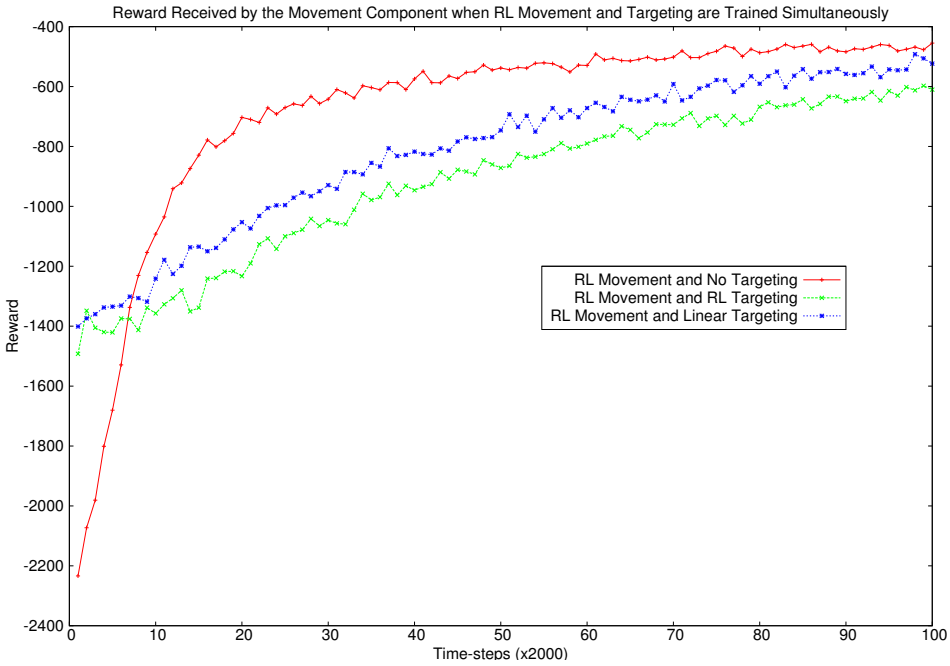


Figure 4.10: Reward received by the movement controller when using Q-Learning for movement and different forms of targeting, trained against `LinearWave`. Q-Learning improves more quickly when no targeting is used, but at the beginning collects less reward.

Targeting Control Technique	Number of New Episodes
None	6387
RL	7999
Linear Targeting	7461

Table 4.1: The number of new episodes that occur when using Q-Learning for movement and different forms of targeting, trained against `LinearWave`. Using targeting results in a higher number of episodes occurring.

When bullets are fired by the RL robot, each round will end significantly sooner than with targeting deactivated. This relates to the low accuracy achieved by both RL targeting and Linear Targeting against the enemy's Wave Surfing movement. Each round marks the start of a new episode within the RL algorithm, where the number of new rounds that occur with each form of targeting can be seen in Table 4.1. The trend shows that the less reward that is received, the higher the number of new episodes that occur. This shows how learning a higher quality movement policy allows the robot to avoid more bullets and survive for longer. Using a more inaccurate form of targeting results in more energy being lost from missing bullets and the robot dying sooner. Therefore, Linear Targeting results in more energy being lost this way than with no targeting and with RL targeting even more is lost. Having a greater number of new episodes results in less effective learning taking place and then choosing more bad moves results in even more new episodes occurring. It can be seen that the RL movement implementation can learn much better by just deactivating the targeting controller.

When the targeting component is being trained, movement is usually controlled by Wave Surfing. For the second part of the experiment, a comparison will be made to when targeting is controlled by RL, with and without pre-trained $Q(s, a)$ estimates.

It is demonstrated in Figure 4.11, that throughout the experiment the use of Wave Surfing to control movement (red series) instead of RL, allows the RL targeting implementation to receive more reward. Training RL movement and targeting simultaneously (green series) results in less reward being received, however a comparable rate of improvement to the reward is still achieved. Pre-training the RL movement action-value estimates (blue series) means that reward is initially lower, but increases at a faster rate,

matching that of Wave Surfing after 200,000 time-steps.

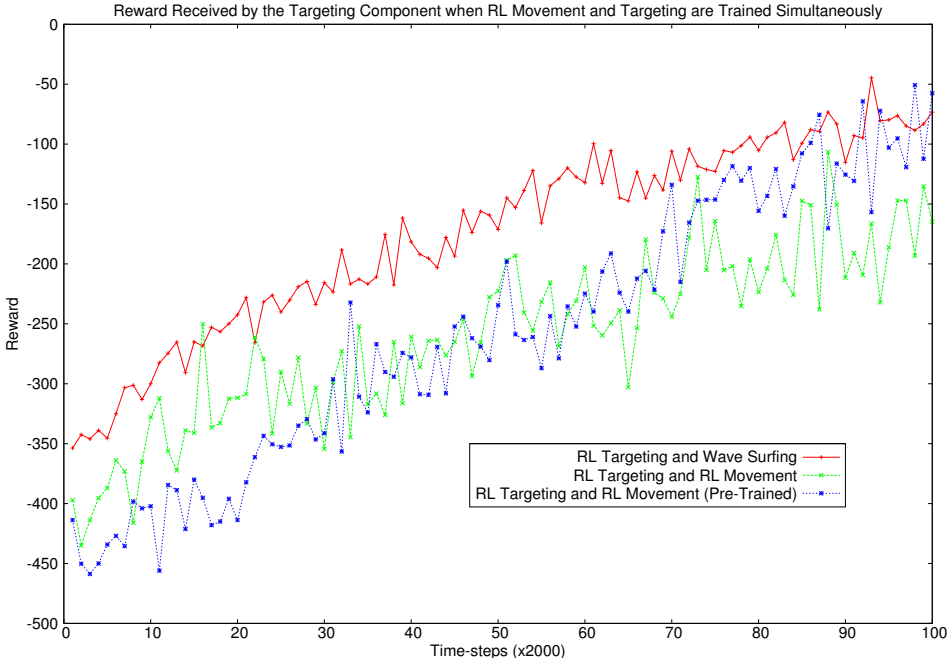


Figure 4.11: Reward received by the targeting controller when using Q-Learning for targeting and different forms of movement, trained against `Corners`. Q-Learning improves the most when pre-trained RL movement is used, however, the most reward is collected when movement is controlled by Wave Surfing.

Movement Control Technique	Number of Movements
Wave Surfing	208868
RL	19902

Table 4.2: The number of movements made when using Q-Learning for targeting and different forms of movement, trained against `Corners`. Wave Surfing makes significantly more moves than RL does.

The RL targeting state includes the distance to the enemy, which can change based on how the robot moves, which means that the movement control technique has a strong influence on the RL targeting. It can be seen in Table 4.2 that Wave Surfing performs around ten times the number of moves as RL movement, which shows that increased movement frequency results in higher targeting accuracy. By moving more often, it is more common for the state to change and an update to be made to a different state. This has the effect of exploring the battlefield more and giving the agent greater opportunities to find the higher quality actions to take for targeting control. For this reason, it is clearly best for movement to be handled by Wave Surfing whilst the RL targeting is being trained.

4.1.8 Performance Against Hand-Coded Opponents

Previous experiments have observed the learning process that the RL implementations undergo. The next experiment will instead focus on how effective the policies learned by the RL robot are at dodging bullets and hitting the opponent. RL implementations will be trained in advance for 200,000 time-steps, so that the learned policies can be used. By comparing combinations of RL movement, Wave Surfing, RL targeting and Linear Targeting, the performance of the RL solutions can be seen relative to the advanced hand-coded techniques. Battles will be ran against both `LinearWave` and `Corners` for 10,000 rounds each, whilst monitoring the dodging and hit rates achieved, as well as how many rounds are successfully won.

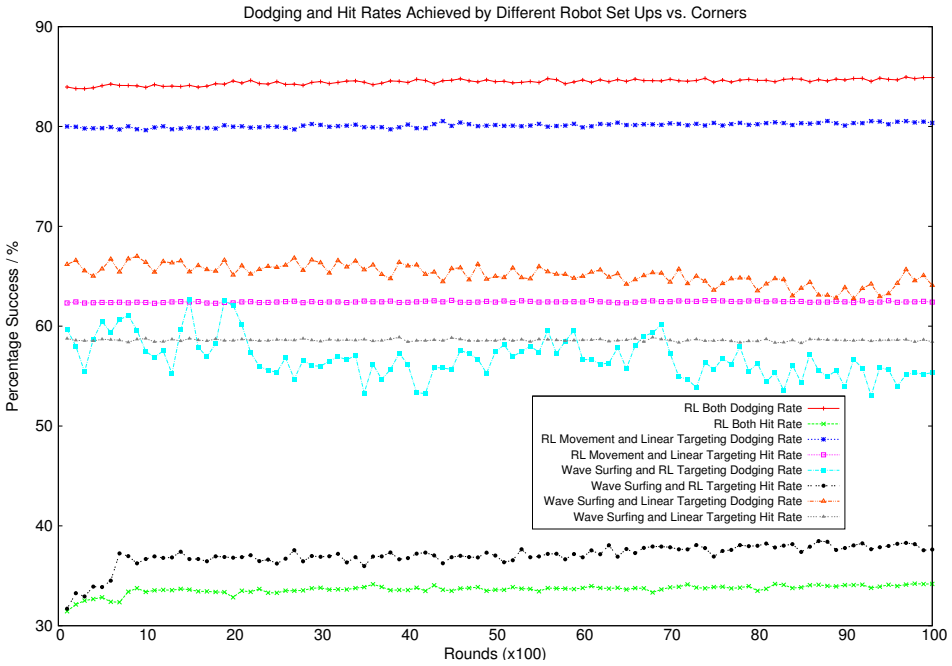


Figure 4.12: Dodging and hit rate achieved by different robot set-ups against the `Corners` opponent. RL movement produces the highest dodging rate and Linear Targeting achieves the highest hit rate.

Figure 4.12 shows that against `Corners`, RL movement managed a vastly higher dodging rate than Wave Surfing, regardless of the targeting method used. The best dodging rate was achieved whilst targeting was also handled by RL. Table 4.3 contains the average dodging rate and hit rate each robot set-up accomplished; RL movement managed a $15-27\%$ improvement in dodging rate over Wave Surfing. Conversely, even though RL targeting attained a hit percentage of 33.6% or 37%, Linear Targeting was $22-28\%$ better. Interestingly, Linear Targeting worked best when grouped with RL movement, whereas RL Targeting was more effective with Wave Surfing. The `Corners` robot moves in a straight line towards a corner and then fires upon its enemy, before moving to the next corner when hit. It is clear from the results that the RL movement solution's ability to adapt allows it to better handle this changing behaviour. Due to the higher standard deviation on the dodging rates of Wave Surfing, a T-Test is carried out to ensure the difference between RL movement and Wave Surfing is significantly different. A two-samples T-Test is performed using the best average dodging rates achieved by RL movement and Wave Surfing, 84.47% and 65.19% respectively. The computed two-tailed p-value is less than 0.0001 (7.143×10^{-143}), which demonstrates that the average dodging rates of the two methods are significantly different. This supports the conclusion that RL movement has outperformed Wave Surfing.

Robot Set-up	Dodging Rate Mean	Dodging Rate Standard Deviation	Hit Rate Mean	Hit Rate Standard Deviation
RL Movement and RL Targeting	84.47%	0.2682	33.64%	0.4459
RL Movement and Linear Targeting	80.11%	0.2183	62.44%	0.06576
Wave Surfing and RL Targeting	56.81%	2.021	37.04%	1.116
Wave Surfing and Linear Targeting	65.19%	0.9966	58.59%	0.109

Table 4.3: Means and standard deviations for the dodging and hit rates achieved by different robot set-ups against the `Corners` opponent. RL movement achieves the highest average dodging rate with the lowest dodging rate standard deviation. Linear Targeting manages the highest average hit rate and the hit rate lowest standard deviation.

When battled against `LinearWave`, the results collected are very different. It can be seen in Figure 4.13 that, with either targeting technique, RL movement achieved a higher dodging rate than Wave Surfing did when paired with RL targeting. Initially, Wave Surfing and Linear Targeting managed the highest dodging rate, but strangely after around 6000 rounds it began to drop. This continued for the remainder of the experiment and for the last 1600 rounds the same dodging rate was achieved as by RL movement. Wave Surfing is being played against another robot using Wave Surfing, so perhaps this starts to have an impact on performance later in the experiment. The results in Table 4.4 show that both techniques attained very similar average dodging rates. Therefore, a T-Test is performed using the best average dodging rates achieved by RL movement and Wave Surfing, 83.09% and 86.66% respectively. The two-tailed p-value calculated is less than 0.0001 (1.43×10^{-41}), showing that the dodging rates are significantly different. If they were not it could be argued that Wave Surfing only achieved a better dodging rate out of chance. However, due to this not being the case the observation made is that Wave Surfing achieves slightly better performance.

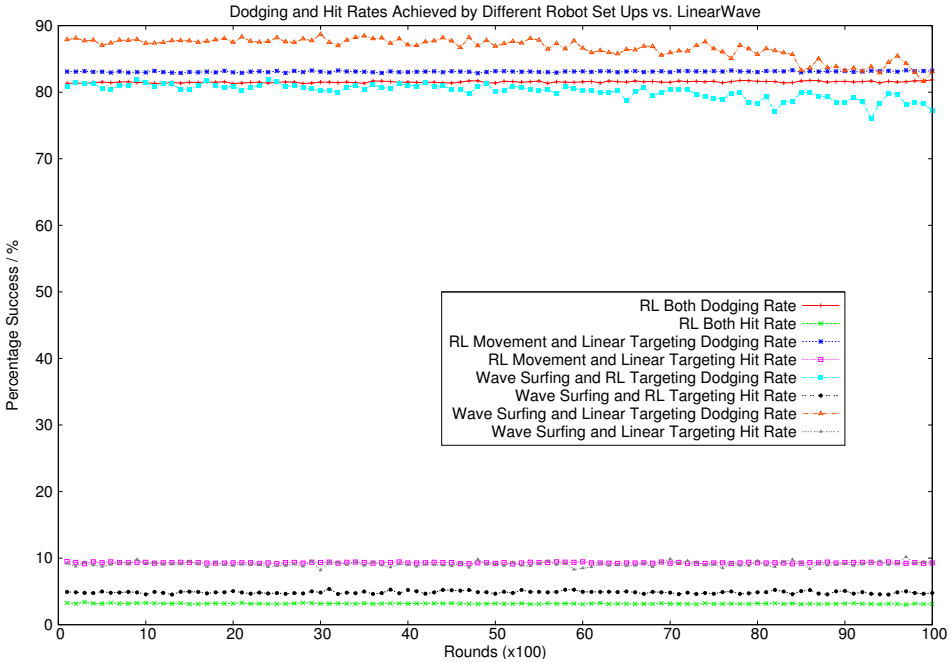


Figure 4.13: Dodging and hit rate achieved by different robot set-ups against the `LinearWave` opponent. Wave Surfing initially achieves the highest dodging rate, but later drops to the same as RL movement. Linear Targeting achieves the best hit rate.

Robot Set-up	Dodging Rate Mean	Dodging Rate Standard Deviation	Hit Rate Mean	Hit Rate Standard Deviation
RL Movement and RL Targeting	81.53%	0.1202	3.17%	0.05773
RL Movement and Linear Targeting	83.09%	0.09185	9.307%	0.07342
Wave Surfing and RL Targeting	80.15%	1.085	4.865%	0.1901
Wave Surfing and Linear Targeting	86.66%	1.553	9.09%	0.3405

Table 4.4: Means and standard deviations for the dodging and hit rates achieved by different robot set-ups against the `LinearWave` opponent. Wave Surfing achieves a slightly better average dodging rate than RL movement at the cost of higher standard deviation. The best average hit rate is accomplished by Linear Targeting.

The hit rates were much closer together than against `Corners`, with Linear Targeting only being $4-6\%$ better. These statistics demonstrate how Wave Surfing is very efficient at avoiding incoming bullets, even from the predictive Linear Targeting technique. It is very impressive that the RL implementation manages a dodging rate that is so close to the powerful Wave Surfing technique, even against such an advanced opponent. Being proficient in adapting to the enemy behaviour is what makes RL a viable technique, as it can learn a movement strategy that exploits how the opponent's Linear Targeting works.

Figure 4.14 displays the number of wins that each robot set-up managed against `LinearWave` and `Corners`. This allows it to be determined whether the policies learned by RL are effective enough to defeat their opponent. The trend seen in Figure 4.14 aligns with the dodging rates and hit rates shown in Figure 4.12 and Figure 4.13. RL movement with either form of targeting earned the most wins against `Corners`, whereas Linear Targeting was best against `LinearWave`, either with RL movement or Wave Surfing.

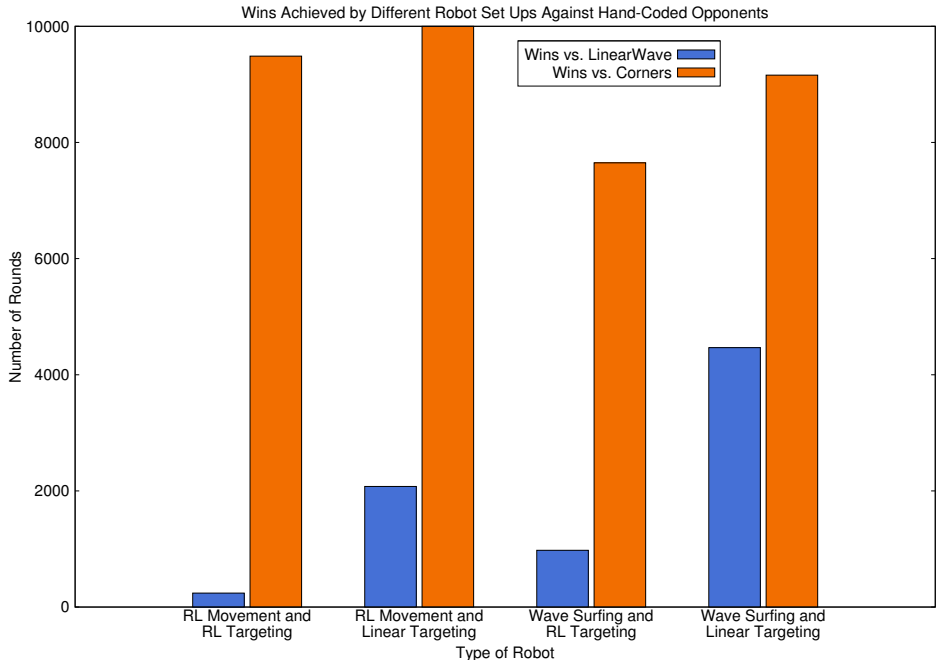


Figure 4.14: Wins achieved by different robot set-ups against `LinearWave` and `Corners`. RL movement with either form of targeting manages the most wins against `Corners`, but Wave Surfing and Linear Targeting achieve the most against `LinearWave`.

Against `Corners`, even though the hit rate achieved by Linear Targeting was $22-28\%$ better than that of RL targeting (Table 4.3), when coupled with RL movement more rounds were won than by Wave Surfing and Linear Targeting. This truly demonstrates the effectiveness of RL movement and its ability to adapt to the enemy's behaviour. It is a strong example of RL being capable of both defeating a hand-coded AI opponent and outperforming a hand-coded AI.

Performance against `LinearWave` is very different, due to the RL targeting struggling against the Wave Surfing movement, allowing Linear Targeting to earn many more wins when paired with both forms of movement control. Even though less wins are received, the results in Figure 4.13 and Table 4.4 show that RL can achieve a comparable dodging rate to the hand-coded Wave Surfing technique and a reasonably close hit rate to that of Linear Targeting. This demonstrates that RL has the potential to perform as well as, if not better than advanced hand-coded AI techniques. Considering Wave Surfing is the result of years of research into movement control by the Robocode developer community, this is an impressive achievement.

4.1.9 Performance Against A Human Opponent

In the previous experiment (Section 4.1.8) the performance of the RL robot was observed against the hand-coded opponents `Corners` and `LinearWave`. The next experiment will provide a more qualitative evaluation of the produced policies, through playing the robot controlled by RL movement and targeting

against a human using the `Interactive` robot. As in Section 4.1.8 the RL implementations will be trained in advance for 200,000 time-steps, except they will be trained against `LinearWave`. Training the robot completely against a human would take a significant amount of time, so it is expected that the sophisticated `LinearWave` will adequately prepare the robot instead. Ten rounds will be played, to observe how effective the produced AI is and also to analyse how realistically it appears to behave. The robot will be controlled by the author, who has previous experience of controlling characters in video games, this will allow the performance to be seen in a realistic setting.

It can be seen very quickly that the RL targeting is not able to hit the human player very often, due to it being quite simple to make changes in movement direction and force a miss. Conversely, the RL movement policy is still very effective, moving around sufficiently and making it not trivial to hit. It does appear to find itself in close proximity to the edges of the battlefield more often than was expected. Occasionally poor quality actions are taken and the RL robot moves into a wall instead of away from it, or drives head-on into a bullet. This would need to be improved if the AI was to be used for a published video game. However, overall the movement provides sufficient challenge to hit and makes it interesting to play against. From the ten rounds played, 10 were won by the author, due to it being quite simple to avoid the bullets and either defeat the opponent through it getting disabled or by hitting it sufficiently.

Due to the movement policy being much more effective than the targeting, a battle was played against the robot using RL movement and Linear Targeting. It is seen that the robot becomes significantly more difficult to defeat, when it has a more accurate targeting strategy. Careful movements have to be made to avoid many of the bullets and the RL robot's movement policy still make it hard to hit, especially when it is not at close range. Out of the ten rounds played, the RL robot won 5 and the author won 5, however, victory was awarded to the author as more damage was delivered across the ten rounds.

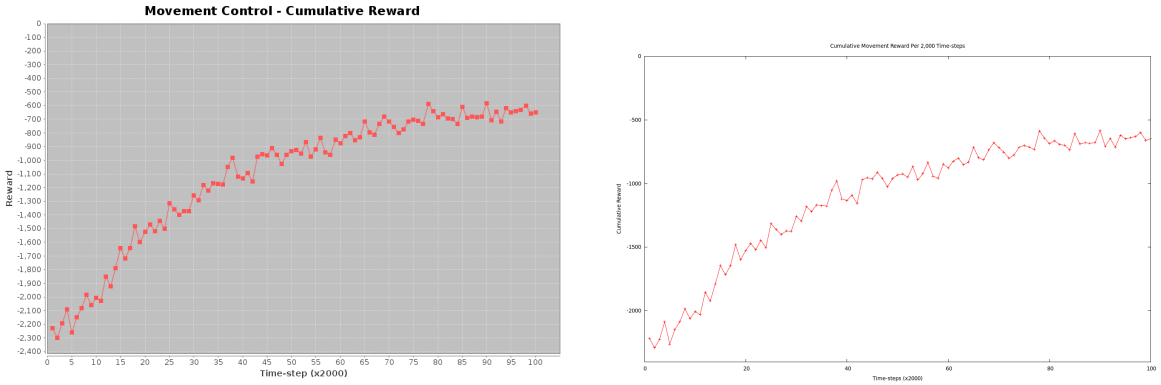
Overall, the performance of the RL movement policy produced is very impressive, making the opponent hard to hit at times. The RL robot using both RL movement and targeting is similar to an AI of "Easy" difficulty found in most video games and when the gun is switched to Linear Targeting it becomes closer to a "Hard" difficulty AI. Therefore, it can be seen that there would be a place for both of the produced policies if the AI was to be used for a video game. The AI does not need to be perfect, as that would make the game impossible to win. Therefore having an AI that can alter its difficulty through using different targeting strategies, is very useful. This shows the RL robot developed to be a complete success, managing to provide sufficient challenge for a human to defeat.

4.2 RoboLearn

The RL framework application, RoboLearn, will now be evaluated. This will involve observing how much faster tasks are to complete when RoboLearn is used and also receiving some user feedback.

The ability of RoboLearn to average results over a sequence of runs was invaluable when completing the experiments on the RL robot. By using RoboLearn, robot settings could be tweaked, the battles run and all the results collected, with significantly less effort than if doing it manually. However, before it can be trusted it is important to check that the results produced by RoboLearn are correct. To achieve this the same battle was ran with RoboLearn and directly in Robocode, then the results obtained from each were compared to make sure they were the same. Figure 4.15 shows a graph of rewards when using RoboLearn (Figure 4.15(a)) and Robocode directly (Figure 4.15(b)), where it can be seen that they are equivalent.

RoboLearn is intended to allow users to run experiments on the Robocode environment quickly and easily, without having to spend a large amount of time configuring and programming. It is used with the intent to learn more about RL and its application to Robocode. This highlights the areas RoboLearn focuses on: being user-friendly, educational and quicker to use than performing the tasks manually. To evaluate its success, a series of tests are designed to determine how long it takes to complete individual tasks. User feedback is also gathered to ascertain the extent to which RoboLearn successfully meets the mentioned usability criteria.



(a) Results plotted by RoboLearn

(b) Results output from Robocode

Figure 4.15: Cumulative reward graphs produced by playing a battle in RoboLearn and Robocode.

4.2.1 Task Completion Speed

There are many key tasks that need to be completed when running experiments using RL on Robocode. These include: creating a robot, tweaking the robot's RL parameters, running a battle against a given opponent and producing results which are averaged over a sequence of runs. Whilst being timed, each of the tasks is completed both using RoboLearn and manually with Robocode and a spreadsheet. By comparing the results from each it can be determined which method is the most efficient. Timing and completion of the tasks are performed by the author.

Test 1: A new robot is set up that uses RL to control movement and Linear Targeting for aiming of the turret. The RL parameters that are used:

- Agent type =Sarsa
- Discount rate, $\gamma = 0.9$
- Exploration rate, $\varepsilon = 0.2$
- Policy type = ε-greedy
- Initial Q Value = 0

Test 2: Parameters of the movement control for the already created robot are altered to the following:

- Agent type = Q-Learning
- Discount rate, $\gamma = 0.94$
- Policy type = Boltzmann
- Temperature, $\tau = 0.15$

Test 3: The robot tweaked in Test 2 is played in a battle for 200,000 time-steps against `LinearWave`, training the RL movement with targeting disabled. Without RoboLearn it is not possible to run a battle for a number of time-steps, so the output is monitored and once the 200,000 time-steps have been completed the battle can be stopped. A graph of the results is produced.

Test 4: After running a battle five times, the results received are collected together and averaged. This battle is the same as in Test 3, so trains the RL movement for 200,000 time-steps against `LinearWave`. A graph of the results is produced.

The chosen tests rigorously evaluate all features of RoboLearn, to observe how much faster the framework allows tasks to be completed. Each test is performed five times to ensure the times are accurate.

After completing the four tasks both manually and using RoboLearn, it is clear from Figure 4.16 that using RoboLearn allows tasks to be completed much quicker. Table 4.5 shows that all four tests were completed using RoboLearn in $26.6 - 34.3\%$ less time, which supports the claim made in Section 1.3 that RoboLearn is made to run experiments more quickly and easily than doing it manually. As the tasks get longer, the more time that is saved by using RoboLearn, which is due to running battles and collecting the results being much more complicated when using Robocode directly. For example, to complete test 4 manually, the battle is ran five times, after each the results are copied to a spreadsheet and then at the end the average dataset is produced and plotted. Conversely, this task with RoboLearn simply involves selecting five runs and waiting as the average is calculated and plotted in real-time automatically.

Task Completion Method	Test 1: Create robot (Time / s)	Test 2: Alter robot (Times / s)	Test 3: One battle (Time / s)	Test 4: Average results (Time / s)
Manually	19.2	16.3	276.6	1406.7
RoboLearn	14.1	11.8	186.3	923.9
	-26.6%	-27.6%	-32.6%	-34.3%

Table 4.5: Times taken to complete each of the four tests manually and using RoboLearn. Each task is completed faster by using RoboLearn.

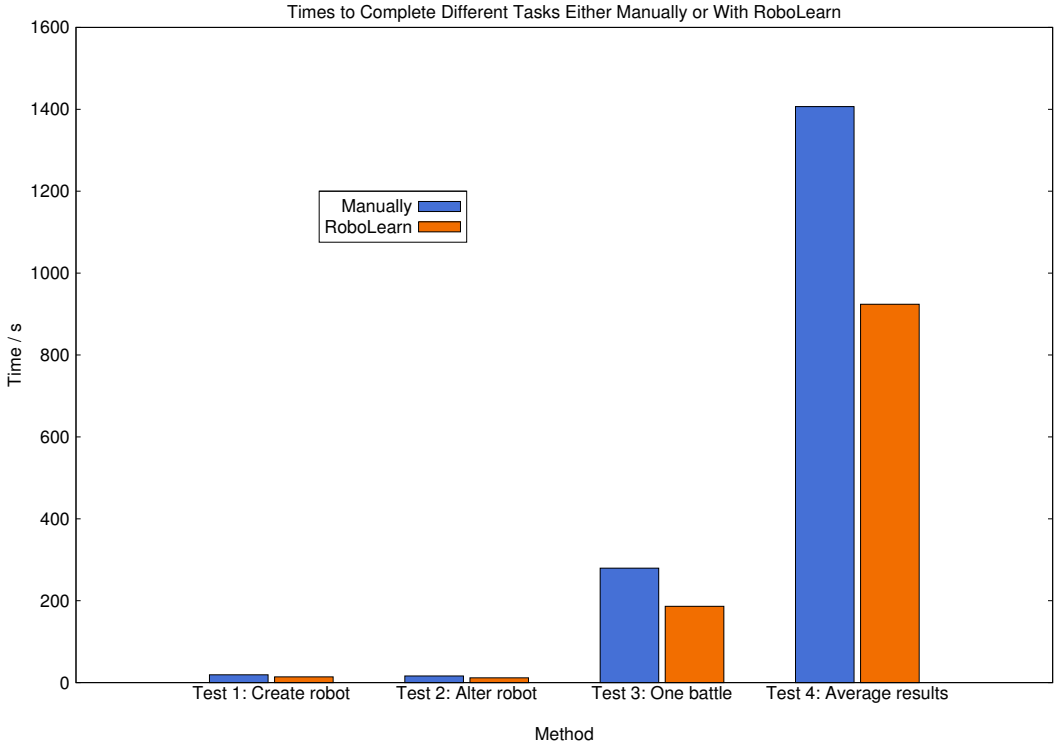


Figure 4.16: Times taken to complete each of the four tests manually and using RoboLearn. When using RoboLearn each task can be completed more quickly.

4.2.2 User Feedback

Assessing how easy to use or how educational a software application is, can be performed through user testing. This involves potential users trying the application, forming their opinion of it and then providing constructive feedback. Carrying out an extensive user study, involving interviews, observation and recording, with hundreds of candidates is very time consuming and beyond the scope of this project.

However, some benefits of user feedback can be utilised with a lower number of users. Three candidates, who are all Computer Science students at the University of Bristol, are provided with RoboLearn and asked to complete a sequence of tasks. The tasks are: create a robot, modify a robot and then play the robot in a battle with a chosen opponent. No direction is given, in order for it to be revealed how clear RoboLearn is to use. Information about RL, parameter settings and Robocode is given to each user through the RoboLearn help and information pages. Afterwards, each candidate is asked for qualitative feedback through their comments on RoboLearn, including which features they like, how easy it is to use and any improvements they would make. The comments received were very positive, whilst also supplying some useful improvements.

- **User 1:** *"After exploring the interface, I found it to be intuitive and well laid-out, making it easy to find features and settings. I particularly like the help files, which are both informative and detailed, giving all the needed information about RL and Robocode. When training the RL robot there is ample data and visualisations presented, making it clear what is going on and how well the robot is doing. The application is easily configurable, with all parameters and settings only a few clicks away and then being able to run battles allows for rapid feedback of any parameter changes made."*
- **User 2:** *"When asked to do certain tasks, from looking at the interface it is always clear where to go to achieve a certain result. I really like the help pages that are available through clicking on the shortcuts next to any terminology. The information provided makes it easier to understand what certain parameters to do and how everything works. If I was to make any suggestions, the button to delete a robot is an 'X', which is unclear and so should be labelled as 'Delete' instead. Also, I think perhaps the main interface could simply be a list of robots, with more details provided when one is clicked on."*
- **User 3:** *"The application's GUI is clear and intuitive; it is obvious how to access specific features without having to refer to any documentation. It offers a wealth of configurable parameters, conveyed clearly, allowing users to tweak the simulated robots to precise specifications. Should the user require further explanation of any of the application's capabilities, help boxes are universally available across the UI.*

  On the downside, the functionality of 'X' button alongside each robot is needlessly unclear - I feel it would be more sensible to change the button's text to 'Delete'. In addition, there is no indication of the application's progress as it simulates battles. Even though any extrapolated time metric would be unreliable given that rounds do not last for a predetermined amount of time, some indication of the number of rounds which have been completed in relation to the total would be an easy-to-implement yet useful feature. Finally, the application in its current form cannot be controlled from the command line. The ability to configure, launch and manipulate the program from a shell script would add another layer of flexibility on top of the program's already adaptable interface, allowing the user to run and gather data on interrelationships between any combination of parameters they care to imagine."

Strengths	Improvements
Clear and intuitive interface	Robot deletion button should be 'Delete' instead of 'X'
Informative and effective help pages	Progress indication during battles
Useful data and visualisations provided during training	Simpler main interface as list of robot names
Easily and extensively configurable	Allow experiments to be ran through scripts
Rapid feedback on effect of parameter changes	

Table 4.6: Summary of qualitative user feedback received on RoboLearn. A selection of strengths and improvements were mentioned.

Table 4.6 contains a summary of the feedback received. It was mentioned by all three candidates that the interface is intuitive and that the help pages are very informative and useful. This demonstrates that RoboLearn is both user-friendly and educational. The quality and breadth of the data and visualisations displayed during training, along with how configurable it is, were commended. By acknowledging these

aspects of RoboLearn, the comments support its usefulness as a tool for running RL experiments on Robocode.

The first two improvements seen in Table 4.6 were were implemented into RoboLearn straight away. Having an indicator of progress on the results window during a battle allows the user to determine how close to completion an experiment is. However, it could be disputed that the third suggested improvement would make the interface significantly less user friendly. A large amount of the information would would require extra work to access. Another reason for presenting the robot's parameters in the main interface is that the user may not recall each robot from its name alone. This means that the robots can be easily distinguished by having their settings printed in the main window. The final improvement of allowing scripts to be run via scripts would require extensive changes to be made to RoboLearn and is beyond the scope of this project. However, it would allow much more flexibility when running sets of different experiments, so it has been recorded as a planned future extension for RoboLearn.

Chapter 5

Conclusion

Over the course of the project, the applications of RL in the Robocode environment have been discussed. This involved a thorough analysis of how RL could be used to learn strategies for both movement and targeting. RL offers the advantage of adaptability, which hand-coded techniques do not, meaning it can alter its behaviour depending on the enemy it is trained against. The main task of applying RL to the robot behaviours involved designing the states and actions of the agent, which would allow it to work towards its defined goals. This complex process involved deciding on the parameters which should be used and then using segmentation to limit the number of possible values each could take. Performing this allowed the number of possible states the agent could be in to be reduced, which speeds up the learning process. Although having too few states throws out too much information and prevents RL from finding a good policy, so a balance had to be reached between the two. Once the robot was completed, the RL framework RoboLearn was developed. It allows users to create their own RL robots, tune the parameter values and then run experiments on the Robocode environment, including averaging results over sequences of runs to produce graphs and statistics of performance. RoboLearn was designed to be user-friendly and educational, whilst making the task of experimenting on Robocode with RL faster and easier than doing it manually. By scrutinising a selection of research which has been completed on Robocode and RL in the past, it was clear which areas of investigation needed more work. To this end, a variety of experiments were performed with the developed RL robot, evaluating its performance with different parameter values and conditions. The main conclusions made are:

- Q-Learning and Sarsa have been shown to achieve similar performance when learning the robot's movement control (Section 4.1.2). MC methods resulted in much lower performance, with a slower improvement of the reward and convergence around a lower value. Q-Learning is preferable due to it resulting in a more stable reward.
- When selecting the action selection method, ε-greedy with an exploration rate of 0.1 was proven to achieve the best overall performance (Section 4.1.3). The difference in the $Q(s, a)$ estimates for an action that leads the robot to be hit and one which does not are not distinct enough for Boltzmann Softmax to be more effective.
- Different segmentations for state parameters were compared, where the best performance was achieved when an appropriate balance was struck between minimising the number of possible values and accurately representing the information given by the parameter (Section 4.1.4).
- It was determined that training RL movement and targeting at the same time results in both achieving lower performance (Section 4.1.7). It demonstrates that each RL implementation is more effective when they are trained independently.
- Comparing the time taken to complete tasks with and without RoboLearn demonstrated that it is quicker and easier to use than having to complete the tasks manually (Section 4.2.1).
- User feedback of RoboLearn showed it to be a success, mentioning how it has an intuitive interface, educational information pages, useful data visualisations and is easily configured (Section 4.2.2).
- A robot that uses RL to control both movement and targeting is able to achieve a win rate of 95% against the hand-coded robot `Corners` (Section 4.1.8). It significantly outperforms the most

sophisticated movement strategy developed, Wave Surfing, by managing a dodging rate which is $15 - 27\%$ higher.

- Against the advanced `LinearWave` opponent, RL manages a comparable dodging rate to that achieved by Wave Surfing (Section 4.1.8). This shows how RL has the potential to perform as well as the most advanced hand-coded AI techniques.
- When the robot is played against a human, it manages very reasonable performance. Using RL targeting it is quite simple to defeat, but is the level of an "Easy" difficulty AI. However, when Linear Targeting is used, the robot becomes much harder to defeat. The movement policy makes it difficult to hit, especially at range, which combined with the Linear Targeting allowed it to win 50% of the rounds played. This shows the RL robot developed to be a success and that it is able to provide challenging game-play and to at times outperform the movement strategy a human is able to perform.

5.1 Status of Objectives

Research and understand the theory surrounding RL and identify how it can be applied best to the Robocode environment.

Significant research was performed into RL and the various algorithms and action selection techniques available, which was discussed extensively in Chapter 2. This gained knowledge was used to adapt an RL implementation for the Robocode environment, including the addition of a dynamic learning rate in the RL update rules (Section 3.1.1).

Implement a robot for Robocode that uses RL to control movement and targeting. The robot should be fully customisable, including alterable parameters and support for alternative hand-coded movement and targeting techniques.

Analysis of the Robocode environment was used to design effective state and action representations, to achieve defined goals for learning movement and targeting control (Section 3.1.3.3 and Section 3.1.4.3). The produced robot is able to maximise its received reward when played against hand-coded robots, showing the RL implementations to be correct (Section 4.1.8). Parameters and control techniques can be chosen through properties files, that the robot loads in at the start of the battle.

Investigate how RL performs on Robocode.

Segmentation and removal of redundant parameters were used to limit the state-action space and produce sensible state and action representations (Section 3.1.3.3 and Section 3.1.4.3). Alternative segmentations were explored, with empirical results being collected to compare their performance (Section 4.1.4 and Section 4.1.5).

A wide range of experiments were carried out that investigated the effect of different: RL algorithms (Section 4.1.2), action selection techniques (Section 4.1.3) and amounts of exploration (Section 4.1.3) on the robot's performance, the effect of training movement and targeting simultaneously was also explored (Section 4.1.7). These all enabled a variety of useful conclusions to be drawn.

An experiment was performed that compared the dodging rates and hit rates achieved by robots controlled by RL and hand-coded techniques (Section 4.1.8). RL was able to outperform a hand-coded AI against the simple `Corners` opponent and achieve comparable performance against the more advanced `LinearWave` opponent.

Develop a framework application for creating RL robots, playing them in games and producing statistics of their performance. It should be both user-friendly and educational.

RoboLearn was developed successfully, allowing users to create robots by choosing their parameter values and control techniques. Experiments can then be ran on Robocode using the created robots, included averaging of results over sequences of runs and producing statistics and visualisations of the results (Section 3.2). User feedback was collected that showed RoboLearn to be clear and intuitive to use, whilst providing informative help pages and being easily configurable (Section 4.2.2).

5.2 Comparison to Previous Work With RL on Robocode

In Section 2.3.1, previous work using RL to learn particular behaviours for a Robocode robot were discussed. Except Livne[12], the RL solutions described were not compared to an alternative non-RL approach, such as a hand-coded solution. This meant that the improvement gained through RL was not determined. The most detailed investigation was performed by Livne, who compared a small selection of robots, observing the effect of exploration and some different versions of TD Learning. However, there were still many areas of investigation not covered by any of the authors.

The robot produced for this project learns multiple aspects of its behaviour through independent RL implementations. This had not been done before, which allowed many interesting observations to be made, discussed in Section 4.1. This project also provided a much more rigorous evaluation of the produced RL robot and a more extensive investigation into the application of RL to Robocode. This involved comparing the performance achieved by Q-Learning, Sarsa and MC methods, then also contrasting the ε-greedy and Boltzmann action selection techniques. These experiments had not been made previously on Robocode, with most authors just using Q-Learning. The evaluation made of the robot included comparing to advanced hand-coded alternatives and a human.

5.3 Future Work

RL is a very large area of study, which is applicable to a wide variety of problems. During the project there were options left unexplored and conclusions made that form interesting areas of future research. These are presented here as extensions to the completed work, with the intention of providing new directions for others to explore.

5.3.1 Support for RoboLearn to Run Experiments From Scripts

When user feedback was collected, the ability to run experiments from scripts was suggested. This would involve major changes to RoboLearn, but would allow more flexibility. When a large volume of results are being collected, often a sequence of experiments would need to be ran. The addition of running from scripts would allow this to be set up in advance and then run to completion without needing interaction from the user at any point. Use of the GUI would then be there for when interactive use of RoboLearn was required or when only a single experiment was to be ran.

5.3.2 Dynamic Exploration

When evaluating different action selection techniques in Section 4.1.3 it was discovered that Boltzmann Softmax initially improved the reward at a faster rate than ε-greedy. Whereas ε-greedy later overtook it and achieved the best overall performance. It would therefore be interesting to explore the use of a dynamic form of action selection and exploration. This would involve adapting the exploration rate and the action selection technique used, based on conditions within the battle and the performance of the agent. By adapting to the situation, the agent would be able to combine the benefits of the different approaches so that it can achieve even better performance.

5.3.3 Function Approximation For Storing Action-Value Estimates

Rather than storing $Q(s, a)$ estimates in a Q-table and then retrieving them when needed, another option is to use Function Approximation. When a table is used, the number of states and actions needs to be limited to prevent it getting too large. However, instead the state-action space can be generalised through a Function Approximation technique such as: Coarse Coding[49, Chapter 8.3.1], Tile Coding[49, Chapter 8.3.2] or ANNs[62]. This is a huge area of research which, when applied to RL and Robocode, could provide significant performance gains, through the ability to store more data in the state and action representations.

Bibliography

[1] ABC. Shadow - RoboWiki. [Online] Available at: http://robowiki.net/wiki/Shadow. [Accessed May 11, 2013].

[2] Apple Inc. Apple (United Kingdom) - iOS 6 - Use your voice to do even more with Siri. [Online] Available at: http://www.apple.com/uk/ios/siri/. [Accessed May 11, 2013].

[3] J. Borenstein and Y. Koren. The Vector Field Histogram - Fast Obstacle Avoidance for Mobile Robots. *IEEE Journal of Robotics and Automation*, 7(3):278–288, 1991.

[4] J. Copeland. A Brief History of Computing. [Online] Available at: http://www.alanturing.net/turing_archive/pages/Reference%20Articles/BriefHistofComp.html. [Accessed May 11, 2013].

[5] M. Gade et al. Applying Machine Learning to Robocode. Technical Report E2-216 DAT3, Aalborg University, 2003. Available at: http://www.dinbedstemedarbejder.dk/Dat3.pdf. [Accessed May 11, 2013].

[6] JFreeChart. JFreeChart. [Online] Available at: http://www.jfree.org/jfreechart/. [Accessed May 12, 2013].

[7] L.P. Kaelbling, M.L. Littman, and A.W. Moore. Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research*, (4):237–285, 1996.

[8] I. Karpov. OpenNERO - game platform for Artificial Intelligence research and education - Google Project Hosting. [Online] Available at: https://code.google.com/p/opennero/. [Accessed May 11, 2013].

[9] T. Kovacs. Overview of the Java RL Library. [Online] Available at: https://www.cs.bris.ac.uk/Teaching/Resources/COMSM0305/secure/rl/rl.library.html, 2007. [Accessed May 12, 2013].

[10] T. Kovacs and R. Egginton. On the analysis and design of software for reinforcement learning, with a survey of existing systems. *Machine Learning*, 84(1-2):7–49, 2011.

[11] F. N. Larsen. User:FlemmingLarsen - RoboWiki. [Online] Available at: http://robowiki.net/wiki/User:FlemmingLarsen. [Accessed May 11, 2013].

[12] A. Livne. Intellibot Reinforcement Learning in RoboCode. Technical report, Israel Institute of Technology, 2006.

[13] A. McGovern, Z. Tidwell, and D. Rushing. Teaching Introductory Artificial Intelligence through Java-Based Games. *Proceedings of Second Symposium on Educational Advances in Artificial Intelligence*, 2011.

[14] K.E. Merrick and M.L. Maher. *Motivated Reinforcement Learning*. Springer-Verlag Berlin Heidelberg, 2009.

[15] Microsoft. Xbox Kinect Motion Sensors - Games, Photos & News. [Online] Available at: http://www.xbox.com/en-GB/Kinect. [Accessed May 11, 2013].

[16] T. M. Mitchell. *Machine Learning*. McGraw-Hill Science/Engineering/Math, 1997.

[17] Next Generation Networking Systems Laboratory. Cognitive Networking - Next Generation Networking Systems Laboratory. [Online] Available at: http://sierra.ece.ucdavis.edu/

(S(smzyjgyn0pi0ns55ag5fs145))/Print.aspx?Page=Cognitive%20Networking. [Accessed May 11, 2013].

[18] J.L. Nielsen and B.F. Jensen. Modern AI for games: Robocode. Technical report, ITU Copenhagen, 2010.

[19] Oracle. System Properties (The Java Tutorials - Essential Classes - The Platform Environment). [Online] Available at: http://docs.oracle.com/javase/tutorial/essential/environment/sysprop.html. [Accessed May 12, 2013].

[20] A. Owens. Secrets from the Robocode masters: Anti-gravity movement. *IBM developerWorks*, 2002. [Online] Available at: http://www.ibm.com/developerworks/java/library/j-antigrav/.

[21] D. Poole, A. Mackworth, and R. Goebel. *Computational Intelligence: A Logical Approach*, chapter 1, pages 1–2. Oxford University Press, Inc., 1998.

[22] D. Precup, R.S. Sutton, and S. Dasgupta. Off-Policy Temporal Difference Learning with Function Approximation. In *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, pages 417–424, 2001.

[23] Robocode Repository. Bots: Summary. [Online] Available at: http://robocoderepository.com/Categories.jsp. [Accessed May 11, 2013].

[24] Robocode. AdvancedRobot (Robocode 1.8.1.0 API). [Online] Available at: http://robocode.sourceforge.net/docs/robocode/robocode/AdvancedRobot.html. [Accessed May 11, 2013].

[25] Robocode. Circular Targeting - RoboWiki. [Online] Available at: http://robowiki.net/wiki/Circular_Targeting. [Accessed May 11, 2013].

[26] Robocode. Head-On Targeting - RoboWiki. [Online] Available at: http://robowiki.net/wiki/Head-On_Targeting. [Accessed May 11, 2013].

[27] Robocode. Linear Targeting - RoboWiki. [Online] Available at: http://robowiki.net/wiki/Linear_Targeting. [Accessed May 11, 2013].

[28] Robocode. Melee Radar - RoboWiki. [Online] Available at: http://robowiki.net/wiki/Melee_Radar. [Accessed May 11, 2013].

[29] Robocode. Neural Targeting - RoboWiki. [Online] Available at: http://robowiki.net/wiki/Neural_Targeting. [Accessed May 11, 2013].

[30] Robocode. One on One Radar - RoboWiki. [Online] Available at: http://robowiki.net/wiki/One_on_One_Radar. [Accessed May 11, 2013].

[31] Robocode. Random Targeting - RoboWiki. [Online] Available at: http://robowiki.net/wiki/Random_Targeting. [Accessed May 12, 2013].

[32] Robocode. Robocode - RoboWiki. [Online] Available at: http://robowiki.net/wiki/robocode. [Accessed May 11, 2013].

[33] Robocode. Robocode home. [Online] Available at: http://robocode.sourceforge.net/. [Accessed May 11, 2013].

[34] Robocode. robocode.control (Robocode 1.8.1.0 API). [Online] Available at: http://robocode.sourceforge.net/docs/robocode/robocode/control/package-summary.html. [Accessed May 11, 2013].

[35] Robocode. Robocode/FAQ - RoboWiki. [Online] Available at: http://robowiki.net/wiki/Robocode/FAQ. [Accessed May 11, 2013].

[36] Robocode. Robocode/Game Physics - RoboWiki. [Online] Available at: http://robowiki.net/wiki/Robocode/Game_Physics. [Accessed May 11, 2013].

[37] Robocode. Robocode/My First Robot - RoboWiki. [Online] Available at: http://robowiki.net/wiki/Robocode/My_First_Robot. [Accessed May 11, 2013].

[38] Robocode. Robocode/Robot Anatomy - RoboWiki. [Online] Available at: http://robowiki.net/wiki/Robocode/Robot_Anatomy. [Accessed May 11, 2013].

[39] Robocode. Robocode/Scoring - RoboWiki. [Online] Available at: http://robowiki.net/wiki/Robocode/Scoring. [Accessed May 11, 2013].

[40] Robocode. Wave Surfing - RoboWiki. [Online] Available at: http://robowiki.net/wiki/Wave_Surfing. [Accessed May 11, 2013].

[41] G. A. Rummery. *Problem Solving with Reinforcement Learning*. PhD thesis, Cambridge University, 1995.

[42] D. L. Schacter, D. T. Gilbert, and D. M. Wegner. *Psychology*, page 19. Worth Publishers, Inc., 2nd edition, 2012.

[43] Skilgannon. DrussGT - RoboWiki. [Online] Available at: http://robowiki.net/wiki/DrussGT. [Accessed May 11, 2013].

[44] SourceForge. Robocode - Browse Files at SourceForge.net. [Online] Available at: http://sourceforge.net/projects/robocode/files/. [Accessed May 11, 2013].

[45] Square Enix. Space Invaders. [Online] Available at: http://www.spaceinvaders.net/#history. [Accessed May 11, 2013].

[46] R. S. Sutton. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts, 1984.

[47] R. S. Sutton. Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding. In *Advances in Neural Information Processing Systems 8*, volume 8, pages 1038–1044, 1996.

[48] R. S. Sutton. Open Theoretical Questions in Reinforcement Learning. *Proceedings of the Fourth European Conference on Computational Learning Theory*, pages 11–17, 1999.

[49] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998. Available at: http://webdocs.cs.ualberta.ca/~sutton/book/ebook/the-book.html. [Accessed May 11, 2013].

[50] M. Tokic and G. Palm. Value-Difference Based Exploration: Adaptive Control between Epsilon-Greedy and Softmax. In J. Bach and S. Edelkamp, editors, *KI 2011: Advances in Artificial Intelligence*, volume 7006 of *Lecture Notes in Computer Science*, pages 335–346. Springer Berlin Heidelberg, 2011.

[51] A. M. Turing. Computing Machinery and Intelligence. *Mind*, LIX(236):433–460, 1950.

[52] A. Uthama. Robot Tank with Neural Networks and Reinforcement Learning. Technical Report EECE 592, University of British Columbia, Vancouver, 2005. Available at: http://www.ece.ubc.ca/~ashishu/tech/592_Robocode_Ashish_Uthama.doc. [Accessed May 11, 2013].

[53] H. van Hasselt and M. Wiering. Reinforcement Learning in Continuous Action Spaces. *Proceedings of the 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning*, pages 272–279, 2007.

[54] R. Vermette. TheArtOfWar - RoboWiki. [Online] Available at: http://robowiki.net/wiki/TheArtOfWar. [Accessed May 11, 2013].

[55] M. I. Voitsekhovskii. Cartesian coordinates. *Encyclopedia of Mathematics*. [Online] Available at: http://www.encyclopediaofmath.org/index.php/Cartesian_coordinates.

[56] L. Waltman and U. Kaymak. A Theoretical Analysis of Cooperative Behavior in Multi-agent Q-Learning. In *Proceedings of the 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning*, pages 84–91, 2007.

[57] T. Wang, M. Bowling, and D. Schuurmans. Dual Representations for Dynamic Programming and Reinforcement Learning. *Proceedings of the 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning*, pages 44–51, 2007.

[58] C. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, 1989.

[59] S.D. Whitehead and L.J. Lin. Reinforcement Learning of Non-Markov Decision Processes. *Artificial Intelligence*, 73(12):271–306, 1995.

[60] M.A. Wiering and H. van Hasselt. Two Novel On-policy Reinforcement Learning Algorithms based on TD-methods. In *IEEE International Symposium on Dynamic Programming and Reinforcement Learning ADPRL 2007*, pages 280–287, 2007.

[61] P. Wright and P. Brant. Flying Saucer - XML/XHTML and CSS 2.1 renderer in pure Java - Google Project Hosting. [Online] Available at: https://code.google.com/p/flying-saucer/. [Accessed May 12, 2013].

[62] Z. Zainuddin and O. Pauline. Function Approximation Using Artificial Neural Networks. *International Journal of Systems Applications, Engineering & Development*, 1(4):173–178, 2007.

[63] M. Zyda and S. Koenig. Teaching Artificial Intelligence Playfully. Technical report, University of Southern California, 2008.

Appendix A

User Testing Consent Forms and Feedback Agreements

A user testing consent form was prepared that would inform each participant of what was expected of them during the testing process. It would also ensure that they consented to being part of the study and demonstrated to them their rights as a participant. Consent forms were signed by each of the three testing candidates.

During the process of user testing the feedback of each participant was recorded. It was then included within a feedback form which was then signed by the candidate. This was performed to make sure the recorded feedback accurately represented the participant's opinions of the application.

All three consent forms are included in Figure A.1, Figure A.3 and Figure A.5. The three feedback agreements are displayed in Figure A.2, Figure A.4 and Figure A.6.

User Testing Consent Form

Study administrator is: Andrew Lord

Participant is: [redacted]

Participant number: 1

This is a study about a Reinforcement Learning framework application called RoboLearn for the designers final year project thesis. It is intended for anyone who wants to learn about Reinforcement Learning and wishes to run experiments with it on the game Robocode. The goal is to make the framework application appealing, intuitive, user friendly and informative. Your participation will help achieve this goal.

In this session you will be working with a working prototype. You will be asked to perform tasks a typical user might do, such as using the application to design robots, edit robots and play them in games of Robocode to view the results. The designer will sit in the same room, observing, taking notes and instructing on what task to complete next. If you are stuck there are a selection of help and information pages available for use.

All information collected in the session will only be used for the purpose of the final year project thesis and will not be stored beyond that point. When your feedback is used in the thesis it will be confidential and will not include your name.

This is a test of the software, not to test you. You will be asked for feedback to find out what aspects are confusing, so they can be made better. You may stop your participation in the study at any time.

Statement of Consent

I have read the description of the study and of my rights as a participant. I voluntarily agree to participate in the study.

Print Name: [redacted]

Signature: [redacted]

Date: 14/05/2013

Figure A.1: User 1 consent form.

User Testing Feedback Form

Study administrator is: Andrew Lord

Participant is: [redacted]

Participant number: 1

Feedback Given

After exploring the interface, I found it to be intuitive and well laid-out, making it easy to find features and settings. I particularly like the help files, which are both informative and detailed, giving all the needed information about RL and Robocode. When training the RL robot there is ample data and visualisations presented, making it clear what is going on and how well the robot is doing. The application is easily configurable, with all parameters and settings only a few clicks away and then being able to run battles allows for rapid feedback of any parameter changes made.

Statement of Consent

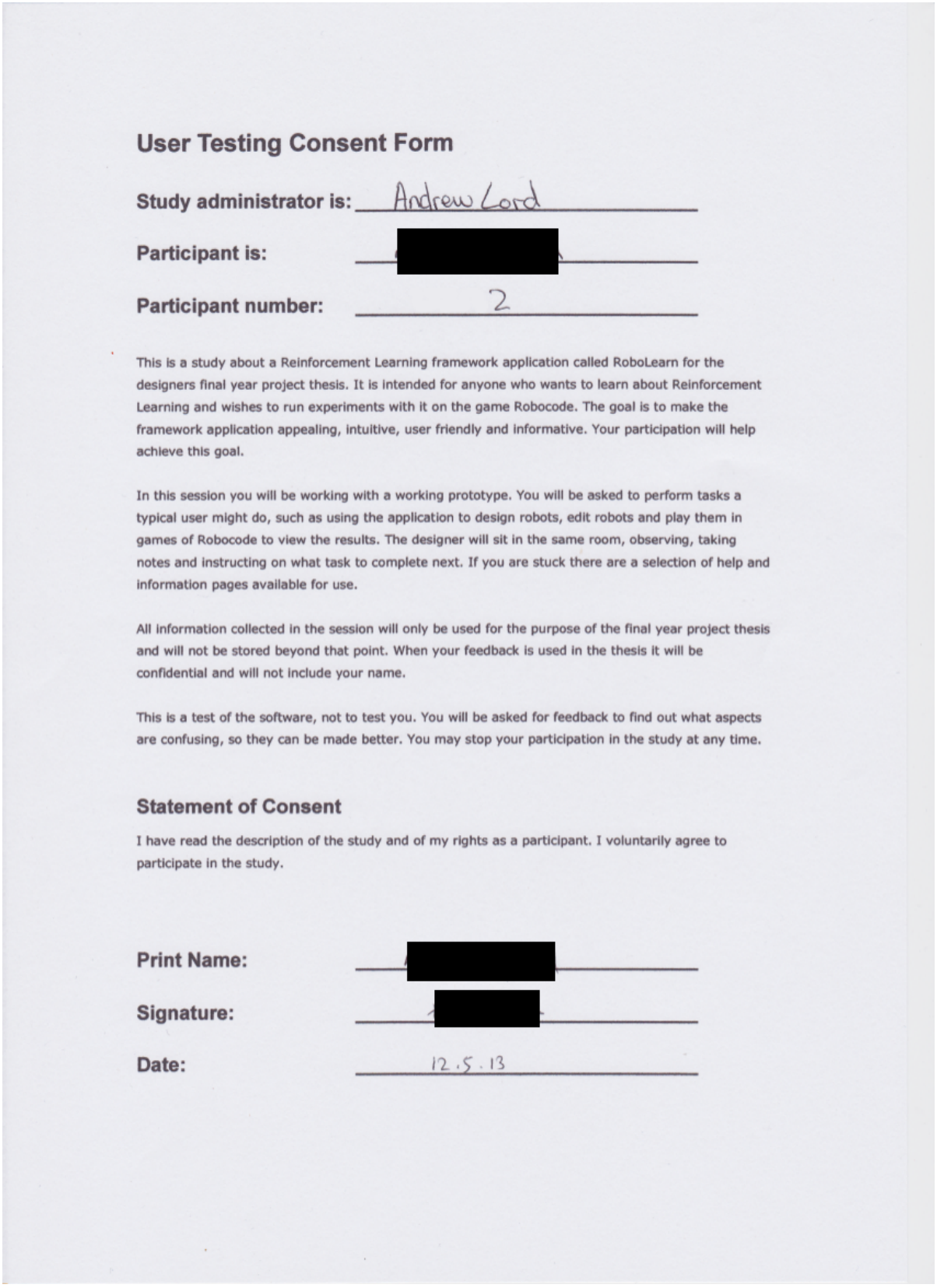
I confirm that the feedback contained above is accurate and describes my opinion of the application. I voluntarily agree for my feedback to be used for the purposes of the study.

Print Name: [redacted]

Signature: [redacted]

Date: 14/05/2013

Figure A.2: User 1 feedback agreement form.



User Testing Consent Form

Study administrator is: Andrew Lord

Participant is: [redacted]

Participant number: 2

This is a study about a Reinforcement Learning framework application called RoboLearn for the designers final year project thesis. It is intended for anyone who wants to learn about Reinforcement Learning and wishes to run experiments with it on the game Robocode. The goal is to make the framework application appealing, intuitive, user friendly and informative. Your participation will help achieve this goal.

In this session you will be working with a working prototype. You will be asked to perform tasks a typical user might do, such as using the application to design robots, edit robots and play them in games of Robocode to view the results. The designer will sit in the same room, observing, taking notes and instructing on what task to complete next. If you are stuck there are a selection of help and information pages available for use.

All information collected in the session will only be used for the purpose of the final year project thesis and will not be stored beyond that point. When your feedback is used in the thesis it will be confidential and will not include your name.

This is a test of the software, not to test you. You will be asked for feedback to find out what aspects are confusing, so they can be made better. You may stop your participation in the study at any time.

Statement of Consent

I have read the description of the study and of my rights as a participant. I voluntarily agree to participate in the study.

Print Name: [redacted]

Signature: [redacted]

Date: 12.5.13

Figure A.3: User 2 consent form.

User Testing Feedback Form

Study administrator is: Andrew Lord

Participant is: [redacted]

Participant number: 2

Feedback Given

When asked to do certain tasks, from looking at the interface it is always clear where to go to achieve a certain result. I really like the help pages that are available through clicking on the shortcuts next to any terminology. The information provided makes it easier to understand what certain parameters to do and how everything works. If I was to make any suggestions, the button to delete a robot is an 'X', which is unclear and so should be labelled as 'Delete' instead. Also, I think perhaps the main interface could simply be a list of robots, with more details provided when one is clicked on.

Statement of Consent

I confirm that the feedback contained above is accurate and describes my opinion of the application. I voluntarily agree for my feedback to be used for the purposes of the study.

Print Name: [redacted]

Signature: [redacted]

Date: 12.5.13

Figure A.4: User 2 feedback agreement form.

User Testing Consent Form

Study administrator is: Andrew Lord

Participant is: [redacted]

Participant number: 3

This is a study about a Reinforcement Learning framework application called RoboLearn for the designers final year project thesis. It is intended for anyone who wants to learn about Reinforcement Learning and wishes to run experiments with it on the game Robocode. The goal is to make the framework application appealing, intuitive, user friendly and informative. Your participation will help achieve this goal.

In this session you will be working with a working prototype. You will be asked to perform tasks a typical user might do, such as using the application to design robots, edit robots and play them in games of Robocode to view the results. The designer will sit in the same room, observing, taking notes and instructing on what task to complete next. If you are stuck there are a selection of help and information pages available for use.

All information collected in the session will only be used for the purpose of the final year project thesis and will not be stored beyond that point. When your feedback is used in the thesis it will be confidential and will not include your name.

This is a test of the software, not to test you. You will be asked for feedback to find out what aspects are confusing, so they can be made better. You may stop your participation in the study at any time.

Statement of Consent

I have read the description of the study and of my rights as a participant. I voluntarily agree to participate in the study.

Print Name: [redacted]

Signature: [redacted]

Date: 14/05/13

Figure A.5: User 3 consent form.

User Testing Feedback Form

Study administrator is: Andrew Lord

Participant is: [redacted]

Participant number: 3

Feedback Given

The application's GUI is clear and intuitive; it is obvious how to access specific features without having to refer to any documentation. It offers a wealth of configurable parameters, conveyed clearly, allowing users to tweak the simulated robots to precise specifications. Should the user require further explanation of any of the application's capabilities, help boxes are universally available across the UI.

On the downside, the functionality of 'X' button alongside each robot is needlessly unclear - I feel it would be more sensible to change the button's text to "Delete". In addition, there is no indication of the application's progress as it simulates battles. Even though any extrapolated time metric would be unreliable given that rounds do not last for a predetermined amount of time, some indication of the number of rounds which have been completed in relation to the total would be an easy-to-implement yet useful feature. Finally, the application in its current form cannot be controlled from the command line. The ability to configure, launch and manipulate the program from a shell script would add another layer of flexibility on top of the program's already adaptable interface, allowing the user to run and gather data on interrelationships between any combination of parameters they care to imagine.

Statement of Consent

I confirm that the feedback contained above is accurate and describes my opinion of the application. I voluntarily agree for my feedback to be used for the purposes of the study.

Print Name: [redacted]

Signature: [redacted]

Date: 14/05/13

Figure A.6: User 3 feedback agreement form.