Reinforcement Learning

Developing a battle tank for the RoboCode battle simulator

July 2009

by Chris van Run, student-id: 3020460
Supervisor: Frank Dignum
Bachelor thesis Artificial Intelligence
Faculty of Humanities
University of Utrecht

Abstract

The battle simulator RoboCode is a competitive Java programming challenge, wherein one can program a competitive battle tank. By exploring the possibilities of using machine learning, especially reinforcement learning, a learning battle tank is designed. to design a learning battle tank. A short survey of reinforcement learning in general is made and an in depth analysis of the RoboCode environment is performed. Finally the design is implemented and tested.

Preface

The subject of this thesis, RoboCode and reinforcement learning, is inspired by one of the courses given at the university of Utrecht. The final project of the course *'imperative programming'* (by Vincent van Oostrom) is to design a team of simulated robot tanks. The follow-up is a tournament where the victor is generally rewarded with an honorary candy bar, as reinforcement.

A digital version of this bachelor thesis, source code et cetera can be found online at http://www.phil.uu.nl/~run/thesis/.

The thesis is typeset in LaTeX. Diagrams and graphs are sketched using *OmniGraphSketcher* and *OmniGraffle*. The Java program has been developed in the IDE of *Eclipse* and the figures have been created using *Adobe Illustrator CS 3*.

The implemented robot is designed for RoboCode version 1.7.1.2.

Please note that all images are vectored and for a detailed view at a graph one can simply use the zoom function on your viewer.

My thoughts go out to Shoko, my lovely girlfriend, for her love, support and grammar checking abilities and to my friend Maaike for suggesting using RoboCode as a central subject for my thesis. Also I would like to thank my supervisor Frank for his subtle subtle understanding of my tendency to evade deadlines.

Chris van Run <Chris.vanrun@gmail.com>

Contents

Introduction i

I A survey of Reinforcement Learning and RoboCode 1

1 Reinforcement Learning 2
- 1.1 The Idea Behind Reinforcement Learning . . . 2
- 1.2 The Framework Constructed . . . 3
- 1.3 Ah, Optimality. 5
- 1.4 The Real World Is Harsh . . . 6
- 1.5 Exploration Versus Exploitation . . . 7

2 RoboCode Battle Simulator 9
- 2.1 An Introduction . . . 9
- 2.2 Other Work . . . 10

II Building the Robot 15

3 Analysis 16
- 3.1 Battle Simulator . . . 16
- 3.2 Observations On The Rules And Physics . . . 18

4 Design 23
- 4.1 The Reinforcement Learning Framework . . . 24
- 4.2 Action Selection . . . 30
- 4.3 Update Equations . . . 31

5 Implementation of the design in `Qbot` 33
- 5.1 The Java Program . . . 33
- 5.2 Testing . . . 36
- 5.3 Conclusion . . . 43

III Appendices 45

A The Rules Of The Game 46

B Event List 51

C Advanced Strategies 53

D Test Results 56

Introduction

How does one learn to perform a certain task? A question that appears simple at first, but proves to be quite a challange to answer even for the most arbitrary situations. Imagine a robot controller or an agent (as it is most commonly referred to by computer sciences) which is expected to perform a certain task. But then imagine simultaneously that the agent as no prior knowledge of the task or of any effect his actions will have. As example, take balancing a pole stick. The agent can influence the environment via it's output-actuators, e.g. move the pole x degrees to the right or left. It can observe the environment via input-sensors, a camera able to register angle of the pole. It receives a reward that resembles a performance measure on the task at hand. How should the agent use this feedback? How do we combine these four elements (input, output, action and reward) into a system that improves or learns over time? The study that explores solutions to this problem, and many others, is referred to as artificial intelligence.

In this thesis, I take the battle simulator of RoboCode as an environment to explore ways to implement machine learning in a somewhat more complex environment then that of (mere) pole stick balancing. In my study of artificial intelligence I frequently have gazed upon implementations of machine learning, but never really had a chance to build an implementation from the ground up. The goal is to design and implement a simulated robot tank that can be set against other tanks in a one on one fight. The robot should improve over time by trial-and-error interaction with the enemy tank, though I do not expect a simple design to beat the currently best ranking robot tank fighters.

Due to limitations on time and the structure of rewards in the battle simulator RoboCode, I have selected reinforcement learning as the machine learning category to base the design on. Reinforcement learning is a very broad term, but the the framework of learning presented by the literature is very dynamic and can be easily adapted. I expect it to be interesting to explore the problems that emerge while scaling a known reinforcement learning algorithm to a (more complex) practical implementation.

My thesis is divided into two parts. The first part concerns a short survey of reinforcement learning and RoboCode, the second part concerns the actual development of a robot. The development consists of the general software development phases; (1) analysis, (2) design and (3) implementation.

I hope you will enjoy reading the rest.

Part I

A Survey Of Reinforcement Learning and RoboCode

Chapter 1

Reinforcement Learning

In this chapter a short introduction to reinforcement learning is given, along with some notation conventions used in later sections. Sources for this section are *Reinforcement learning: A survey* (Kaelbling et al., 1996), *Artificial Intelligence - A Modern Approach* (Russell and Norvig, 2003) and *Reinforcement learning: An Itroduction* (Sutton and Barto, 1999).

1.1 The Idea Behind Reinforcement Learning

Reinforcement learning is not directly defined by what it does but rather by what it solves (the reinforcement problem). Imagine an agent set in an environment, in this environment he can and does certain actions. The environment, influenced by the chosen action, gives feedback to the agent. Usually along the lines of a performance measure constructed as reward or reinforcement. The goal of the agent is to get as much reinforcement or reward as possible. Hence the name *reinforcement* learning. The reinforcement learning problem can be described as the problem the agent faces when trying to learn a behaviour that maximises his reward. Generally the agent has no knowledge of how its actions will influence the environment and its future ability to choose actions[1]. It must learn this information by trial-and-error interaction with the environment. Since an single action can only be used to either *explore* or *exploit* the environment, a choice between the two must be made. Further on in this section we will return to this exploration-exploitation conflict.

A great property that reinforcement learning agents have, is that you only need to tell the agent his performance on a given task. That is, you only need to specify *what* needs to be done instead of explicitly detailing *how* it needs to be done. This sometimes leads to an agent surprising the designers by showing behaviour they would never have thought of.

The current field of reinforcement learning stems from three different threads that

[1] Prior knowledge varies. Some reinforcement learning algorithms integrate large quantities of knowledge into the agent before learning starts.

were pursued in the past (late 1950's). The first thread concerns learning by trial-and-error, this started from the psychology of animal learning. Usually the field of psychologist and biologists. The second thread comes from the problem of optimal control and finding its solution using value functions and dynamic programming. A field mainly occupied by computer scientists and mathematicians. The second thread lacked the learning approach the first thread was so keen on researching. A third thread, although smaller, concerns the use of temporal-difference learning. All three threads intertwined in the late 1980's to produce the modern field of reinforcement learning as presented in Russell and Norvig (2003).

The research for a solution for the reinforcement problem can be generally divided into two categories. The first is to search the space of behaviours in order to find one that performs well in the environment. Genetic algorithms and genetic programming are examples of this. The second is to use statistical techniques and dynamic programming methods to estimate the utility of taking actions when the environment is in a certain state. The main emphasis of this thesis will be on the latter rather then the former category.

1.2 The Framework Constructed

Time is usually a sequence of discrete time steps, $t = 0, 1, 2, 3, \ldots$. The interaction between the environment and the agent is best described by signals. As depicted in figure 1.1, the agent sends an *action* signal (a_t) containing a chosen action. The environment then sends two signals to inform the agent of the consequences of his action: a *state* signal (s_{t+1}) to show the changes the environment has undergone and a *reward* signal to inform the agent of its performance. The reward (r_{t+1}) is usually $\in \mathbb{R}$ As the agent is usually part of the environment it can be difficult to see where

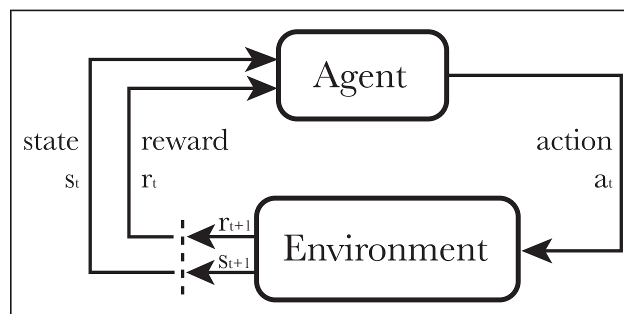

Figure 1.1: The agent-evironment interaction in reinforcement learning (Sutton and Barto, 1999)

the agent stops and the environment starts. The boundary is usually set at where the agent no longer has absolute control. It cannot declare what reward it should

receive nor can it directly influence the state change, therefore the reward and state signals are part of the environment.

This framework can be used to reduce the reinforcement problem to the problem of learning an mapping from a state s in the set of possible states ($\mathcal{S}$) to an action a in the set of possible actions ($\mathcal{A}$);

$$\mathcal{S} \Rightarrow \mathcal{A}$$

This (sort of) mapping is called a *policy* (π). As a convention the policy doesn't have an action as output, but rather the probability that an action will be selected given a state. This results in a joint probability distribution[2] over the actions.

$$\pi_t(a, s) \Rightarrow Pr(a|s)^{3}$$

Since π can (or should) change over time the time-index t is added. A policy π can then be used to map a state s to a single action a by setting $Pr(a|s) = 1$ and $Pr(x \text{ and } x \neq a|s) = 0$. By using this convention the framework allows for reinforcement learning to cope with probability distributions.

The goal of the agent can now be formalised as to find an optimal policy (π^*), optimal in the sense that it maximises the cumulative reward over the long run. The simplest way to calculate the cumulative reward R (from time step t onwards) is as follows:

$$R_t = r_{t+2} + r_{t+2} + r_{t+3} + \ldots + r_T$$

This is fine for agents that are only learning for a finite amount of time, with T being finite in episodic tasks. But if $T = \infty$, as in continuing tasks, the expected reward R_t could easily become infinite and the agent can no longer compare these. To cope with this, *discounting* is introduced. With discounting, the expected rewards received later on are considered to be of lesser importance.

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

With γ being the *discount rate* with values: $(0 \leq \gamma \geq 1)$. If $\gamma < 1$, the result of the infinite sum is finite as long as the reward sequence $\{r_k\}$ is bounded. If $\gamma = 0$, the agent only looks one step ahead. With γ getting closer to 1 the agent becomes more farsighted. Note that actually using $\gamma = 1$ would result in the expected cumulative reward discussed earlier.

The formalised goal now, is for the reinforcement agent to find an **optimal policy** (π^*)[4] that maximises the **expected discounted cumulative reward** on the long run.

[2] A joint probability distribution means that the sum of all distributed probabilities in a set equals 1.

[3] $Pr(A|B)$ = **Pro**bability of A given the fact B

[4] There can be more then one optimal policy, each having the same expected discounted cumulative reward

1.3 Ah, Optimality...

Most reinforcement learning algorithms using a statistical approach make use of a *state value function* to calculate the utility of a state: $V(s)$. That is, the expected discounted cumulative reward that is to be gained from being in an environment having the property of being in state s. This utility is not only dependent on the next reward the agent is going to receive but also on all the successor states, rewards and actions of that state. The influence that successors are going to have is determined by the policy the agent is going to use from state s onward. Please note that the policy (π) was defined as a probability distribution.

The relation between the utility of state s and its possible successor states s' is best described by the Bellman equation[5] for state values:

$$V^{\pi}(s_t) = \sum_{a \in \mathcal{A}} \pi(s_t, a) \sum_{s' \in \mathcal{S}} Pr(s' = s_{t+1}|s, a) \left[\mathfrak{R}^a_{ss'} + V^{\pi}(s')\right]$$

$\boxed{\mathfrak{R}^a_{ss'}}$ the expected next reward, when performing action a in state s resulting in state s'.

$\boxed{Pr(s' = s_{t+1}|s, a)}$ the transition probability of doing a in state s resulting in s' being the next state.[6]

The equation weighs the $V^{\pi}(s')$ via their probability of occurring after state s. This is much like a tree structure, with a few premises, the number of branches keep multiplying exponentially until leaves are reached, i.e. the final state s_T is reached. A similar relation and equation can be formulated for the *action-state* value function $Q^{\pi}(a, s)$ for the utility of performing action a in state s.

Let us return to the goal of the reinforcement agent, finding an optimal policy. The optimal policy π^* produces V^*, the optimal state value function. This is the true utility value of a state. Because V^* is the value function for a policy, it satisfies the Bellman equation stated above but can be reformulated[7] to form the *Bellman optimality equation* for state values:

$$V^*(s_t) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} Pr(s' = s_{t+1}|s, a) \left[\mathfrak{R}^a_{ss'} + V^*(s')\right]$$

As you will notice, the Bellman optimality equation is no longer dependent on the policy being followed. If the dynamics of the environment are known – i.e. *transistion*

[5]Named after Richard Ernest Bellman, an U.S. mathematician

[6]Most environments in reinforcement learning are considered to be stochastic in a static way. That is, performing an action a in state s_t will result in the environment being in state s_{t+1} by a fixed probability.

[7]See for details: (Sutton and Barto, 1999)

probabilities and $\Re^a_{ss'}$ – and the state set $\mathcal{S}$ and action set ($\mathcal{A}$) are finite, the Bellman optimality equation results in a set of equations. This equation set can be solved using any one of a variety of methods for solving sets of non-linear equations. This provides an omniscient agent with a V^*. The same could be done for $Q^*(s,a)$.

Constructing an optimal policy if value functions V^* or Q^* are known becomes trivial. If inn state s select the action a with that has the highest $Q^*(s,a)$. Unfortunately, the more practical applications of reinforcement learning cannot provide optimal value functions this easily and are generally left to approximate them as best they can.

1.4 The Real World Is Harsh

The Bellman optimality equation, when some premises are met, results in a set of non-linear equations. One of the premises is omniscient of the dynamics of the environment but this is generally not the case. The practical applied reinforcement learning agent is faced with the problem of not knowing what the exact consequences his actions will have and not knowing what rewards it is going to receive. Lack of omniscience is not the only thing real world agents tend to lack. Some real world reinforcement problems also have to cope with continuous state and action spaces instead of finite spaces. The agent has no choice but to approximate these value as best it can. Different approaches are taken in the field of reinforcement learning to do this, some which are discussed below.

Some methods approximate the value functions via a model. The model is set to simulate the dynamics of the environment and is made more accurate by trial-and-error. They do so in the hope that, given enough time, it will result in a perfect model. They then solve the linear-equations produced by the Bellman optimality equation. Methods that do this are generally named **dynamic programming** (DP) methods.

Other methods forgo working towards a *perfect* model, mostly due to the exceptional computational expenses that comes with building the *perfect*. **Monte Carlo** (MC) methods solve the reinforcement learning problem by averaging over the feedback provided by the environment through experience. They usually process local probability transactions instead of the complete set of probability transactions, resulting in far less computation.

The **temporal-difference** (TD) learning type of methods drop the usage of a model altogether. When an agent is using TD learning and finds the environment in state s_{t+1} it uses the reward signal of the environment to update its estimate of the preceding state's value ($V^*(s_t)$) by the following the very general equation:

$$\text{NewEstimate} \leftarrow \text{OldEstimat}e + \alpha\,[\text{Target} - \text{OldEstimate}]$$

The [Target − OldEstimate] can be considered the error of the OldEstimate. The update makes the estimate take a step towards the real optimal value with a step

size of α. How the exact update equation is calculated depends on which of TD learning methods is being used.

TD, MC and DP methods are not different methods of approximation per se, they can be combined resulting in hybrid methods. Usually it is even useful and desirable to use different types at the same time. For example, one could use one method for on-line computation while performing the task and another for off-line learning between the tasks. Reinforcement learning methods are mainly compared by the speed and guarantee of convergence towards an optimal policy.

1.5 Exploration Versus Exploitation

What in most real world applications of reinforcement learning holds is that, while the agent is trying to approximate the optimal value functions, the agents policy is already having an influence on the cumulative reward the robot is set to receive at the end of its task. Exploring every action in every state may ensure a speedy convergence toward the optimal value function but at the expense of the rewards it could be receiving by exploiting the knowledge it already has. The agent needs to find the delicate balance between exploring and exploiting the environment.

To do this a random factor can be introduced to the action selection system. With an **ϵ-greedy** approach the agent always selects the best action (to its knowledge) but with a chance of ϵ it picks another action at random. This approach works but results in the second best action having the same probability of being selected as the worse action. In situations where the worse action might have devastating consequences, this is something that should be avoided.

One would like to have a probability distribution over the actions that takes into account the different values they are perceived to have. The **softmax** approach does this by using a Boltzmann distribution[8] in state s:

$$\frac{e^{Q(a_1,s)/\tau}}{\sum_{b=1}^{n} e^{Q(a_b,s)/\tau}}$$

Where τ is called the temperature. If the temperature is set very high the actions are selected almost equiprobable. Set to a low temperature the probability is more dependant on the action value. When $\tau \to 0$ the softmax approach becomes near greedy selecting only the actions with the highest value.

A Boltzmann distribution as action selection system synergies nicely with the as a probability distribution defined policy. Convergence of the value function can still be guaranteed, even when using a random factor for exploration. Namely by greedy in the limit of infinite exploration; if the agent is given enough time, the value functions will still converge.

[8]Named after Ludwig Eduard Boltzmann, an Austrian physicist

This concludes the survey of reinforcement learning framework, conventions and methods. The following section is dedicated to analysing the task environment of the RoboCode battle simulation.

Chapter 2

RoboCode Battle Simulator

In this section the RoboCode battle simulator is shortly introduced. A by a survey of previous projects follows, wherein robots are being designed and implented using some form of machine learning.

2.1 An Introduction

RoboCode[1] is an environment in which robots battle each other. It has been developed by Matthew A. Nelson at the end of 2000 as a personal gimmick. After Nelson accepted a position at IBM, RoboCode temporary had a official status under the department of IBM AlphaWorks. In 2005 it was released as an open source project at SourceForge[2]. It has seen a lot of input from the community and is currently being updated and maintained by Flemming N. Larsen.

RoboCode is an abbreviation of 'robot code'. The robots simulate small tanks, which can be programmed using the programming language of Java. The programmers can construct the code robots use to perform actions but, in general, have no direct influence on an ongoing battle. Robots can compete in a team or deathmatch battles. Building a team brings more types of robots to the playground, but this paper will focus on single robot teams. The basic RoboCode package comes with a standard set of robots. The robots that have the prefix `sample.` are part of this standard set.

Apart from sporadic organising competitions the community also has an up-to-date ranking of the programmed robots. This RobotRumble has multiple divisions determined by the codesize of the robot and the type of battle. (e.i. MicroBot, MacroBot and MegaBot & one-on-one, ten-on-ten or teambattles) There also exists an online robot repository[3] that contains all uploaded robots. The RobotRumble and the RobCode repository can be used to find able adversaries for robots. They also provide

[1] http://robocode.sourceforge.net/
[2] http://sourceforge.net/
[3] http://robocoderepository.com/

an extensive source of 'hand coded' algorithms that can be used as an inspiration for home build robots. The robots that are referred to later on in this thesis, can all be found and downloaded from the repository.

The RoboCode environment is governed by game rules and a physics system abiding to simple formulas. If the reader is not familiar with these, it is recommended to read appendix A (Gameplay and physics) on page 46 for a clear understanding of the basic principles governing robots in RoboCode.

2.2 Other Work

The RoboCode battle simulation is used as an educational tool on many different parts of the globe. It does not come as a surprise that there are several other projects wherein artificial intelligence (AI) methods are being implemented to design adaptive robots. The RoboCode repository and RobotRumble mainly consist of undocumented implementations. In this section an overview of some of the documented projects is presented. The goal is to get a broad idea about the projects, for the details and exact methodology of every algorithm the interested reader is referred to the original paper.

Genetic Algorithms And Programming

Eisenstein (2003) explores how genetic programming (GP)[4] can best be applied to produce controllers based on subsumption and behaviour oriented languages such as REX. Developing a variation on REX, TableRex, RoboCode was used as a test case. The robots were evolved against the sample robots as well as against the more advanced robot *SquigBot*. Four training scenarios were constructed using the variables: *Number of adversaries* (one or more) and *starting position* (fixed or random). In the simplest setup, *one adversary and fixed starting location*, robots emerged that could easily defeat the sample and advanced robots. Using *random staring positions*, the robots required more time to be evolved into controllers of the same performence.

When using *multiple adversaries and a fixed starting position*, robots failed to evolve efficient movement strategies. With *multiple adversaries and a random staring position* the robots would take an exponentially long time to evolve in anything that could beat but the easiest sample robots. Eisenstein noticed that robots were rarely evolved to take advantage of their gun. One explanation given by Eisenstein was the fact that firing only costs Health and without good targeting using the gun had virtually no benefit for the robots. Eisenstein has the following to say about targeting:

[4]Genetic algorithms and genetic programming are inspired by the natural evolution of organisms, but instead tries to evolve robot controllers. By using a large populations of controllers, genetic programming simulates evolution by letting the highest scoring members of that population combine to produce the next generation. By allowing random permutation of the controllers the population ideally produces an optimum controller.

> "Targeting is a difficult problem, even for the hand-coded robots. Bullets move at a top speed of 20 units per tick, and tanks can move as fast as 8 units per tick. Thus, the velocity of the target tank must be taken into account. In addition, the time required to rotate the gun turret must also be factored in. Even if the movement of the target is totally predictable, accurate targeting requires some complex mathematics. To make matters worse, the simulator will tell a robot whether its shot hit or missed, but it provides no information about how close the shot came."

It can be concluded that the targeting issue could be a subject for improvement. One could, for example, use an artificial neural network (ANN)[5] to train a targeting controller.

In Shichel et al. (2005) the designers undertook a first attempt to introduce evolutionarily designed robots into an international RoboCode competition. The purpose was to see how these genetic evolved robots would hold against human made 'hard coded' robots. As an extra challenge they chose the division of one-on-one HaikuBot challenge, where their code is limited to four instances of a semicolon (four lines). Evolving the robots against the top players of past HaikuBot challenges resulted in a individual robot. The robot took third place out of 27 entrees. Showing that GP can be used to evolve adequate controllers. The question remains though, whether GP can be used to produce robots to compete in competitions that allow robots with a larger codesize.

Hybrid Systems

From the Department of Computer Science, at the University of Aalborg, several documented implementations of artificial intelligence (AI) have been found. During the DAT3 seminar, two notable projects produced AI related robots. Both of these use a hybrid system, i.g. multiple AI algorithms.

In Gade et al. (2003) a modular hybrid agent architecture is developed, selecting the right type of machine learning (AI algoritm) for each of the main properties of RoboCode. The goal was to develop a robot (named `Aalbot`) that performed well in a melee battle[6]. The architecture framework used is generally referred to as *fine grained subsumption*. This hybrid architecture is an architecture that is neither pure reactive nor pure deliberative; taking advantage of both extremes.

The architecture consists of three components, i.g. a world state, subsumption layers

[5] An artificial neural networks consist of interconnected artificial nerve cell. Used in artificial intelligence to solve certain problems, it is used by cognitive modelling to test theories of the brain. It has been shown to excel in recognising patterns. Consisting of an input, hidden and output layer of nodes it trained by using examples of desired input/output.

[6] A melee battle is where multiple robots fight in the same battlefield. In this case, four; `Aalbot`, `Peryton`, `SquigBot` and `sample.Walls`

and an arbitrator component. The subsumption layers are each designed to tackle a certain controlproblem of RoboCode, using a set of modules contained within each module. As example the problems *Targeting* and *Target Selection* are two different layers. The layers can all add whishes to a list, i.e. 'Kill robot `A`'. This wish list, after been edited by every layer, is then presented to the arbitrator compontent. The arbitor translates it into command the RoboCode robot framework can parse. The world state represents a fading memory accessible for all layers, making information readily available.

For some of the modules hard coded strategies are being used, due to their efficiency. Other modules use AI algorithms;

An artificial neural network (ANN) module for targeting and shooting the gun.

A reinforcement learning (RF) module for target selection.[7]

A genetic algorithm (GA) was used for the movement and radar control modules.

It should be noted that testing adaptive hybrid systems can be difficult. Showing proof that part of the system improves over time is best done by eliminating influence from all the other subsystems. For example, the targeting system using the ANN can be implemented and can then be trained against an array of (only) moving robots. Showing an increasing frequency of bullet impacts can suffice to show that the ANN is learning. But testing target selection is far to depend on the performance of other subsystems to implement this kind of testing. Gade et al. showed that the implemented RF module (using the other adaptive subsystems) improved towards the convictions they had about optimal targeting. This made using the RF module obsolete as they could have hand coded their convictions of optimal targeting into the robot.

When testing the ANN they found that the gun performed 7 to 8 times against the robot `SpinBot`. Relative to firing at a random angle. Further testing of the RF module was mainly meant to approximate the balance between exploration and exploitation. Gade et al. concluded on a variable for the probability distribution for action selection. The GA module did not show the steady increase in the average fitness of individual as the designers optimistically expected. It is unclear what the exact reasons for the lack of efficient evolution is. As stated in the report, one of the possible reasons for this are the choices the designers made for the set of terminals and functions. Considering the success of GA in Eisenstein (2003) and Shichel et al. (2005) this could very well be the case. This shows that special care should be taken when selecting the input and output for an algorithm. After training `Aalbot` with generally poor performing robots, it performed adequately against the more advanced robots, ranking second in the earlier mentioned melee battle.

[7] Specifically Q-learning with a variable probabilistic approach for choosing actions

Another documented implementation from the university of Aalborg comes from Frøjhær et al. (2004). Yet another hybrid hierarchical architecture is used to implement a variety of AI methods. One aspect that differs from `Aalbot` is that the robot from Frøkjær et al. was designed so that it could work within a team of robots. The robot was designed to use each machine learning method on the hierarchical level the method was best suited for. In figure 2.1 on page13, the outline is depicted. The model has three overall layers, namely *General Strategy*, *Strategy Modules* and *Interaction Modules*.

The *General Strategy* uses a Bayesian network (BN)[8] to choose between an offensive or a defensive strategy. It does so by feedback from the *Strategy Modules* (*Retreat, Attack, Acquire target*) which tell the Bayesian network the probability of certain factors. Lastly, the actual behaviour is determined by the interaction modules for each of the actuators (*Radar, Gun, Move(body) and Team*).

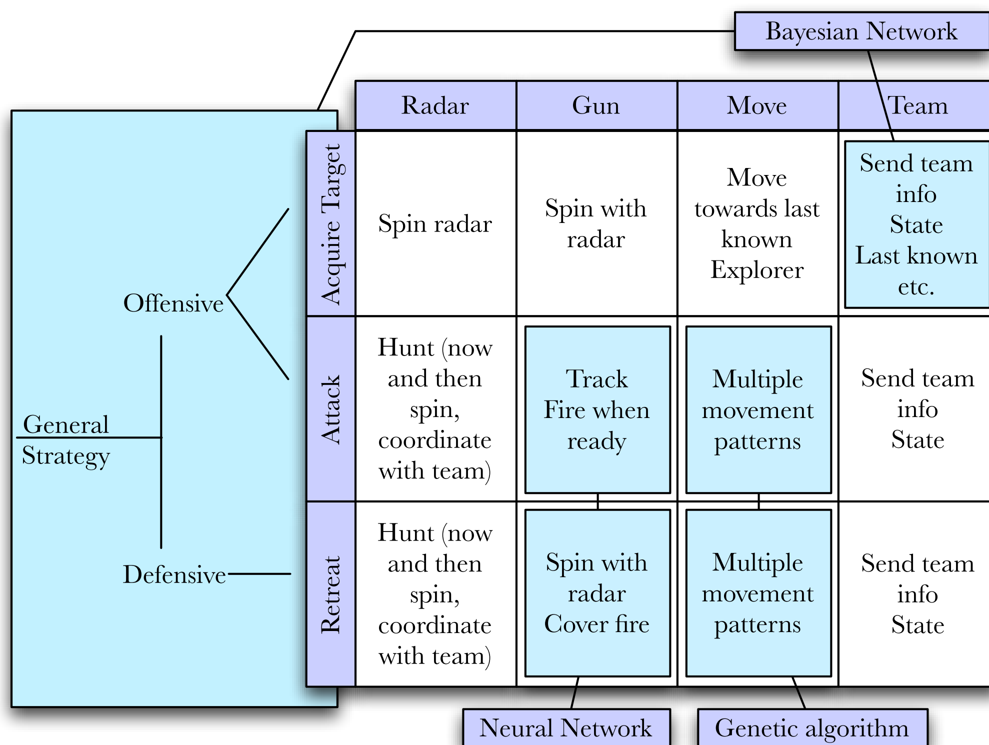

Figure 2.1: An outline of the module design in (Frøjhær et al., 2004)

[8]A Bayesian network is a probabilistic graphical model that represents a set of random variables and their conditional independencies via a graph (i.e. the nodes resemble variables and the edges the (in)dependencies. Given certain conditions the network can be used to compute the probabilities of events.

A neural network (ANN) is trained for shooting/firing the *Gun*, a genetic algorithm (GA) for *Move*(body) and a separate BN is used for target selection. For the *Radar* a hand code strategy is being used. This project is very detailed on giving background information on the used algorithms and is very thorough for the selecting procedure for the best input for the algorithms from the sensor data RoboCode offers. I will discuss some of the selection criteria and segmentation techniques in a later section. Due to the limited time on behave of Frøkæ et al. only the movement (GA) and shooting (ANN) modules were implemented and tested with succes. Both showed improvement over time. The implementation of the target and strategy selection module (BN) was never completed and therefore never tested.

Considerations

Gade et al. (2003) make the following observation about machine learning and RoboCode:

> "Although some variations of the pattern matching methods have been implemented, the interest in applying advanced ML to Robocode is not overwhelming. One explanation could be that the simulation environment in Robocode is not too difcult for humans to fully understand (the only obstacles are the walls, there is a limited number of actuators and little nondeterminism etc.), making the explicit expression of behaviour code possible. Consequently many traditional 'hand coded' techniques have proven very effective, and in addition require less development time to achieve good results."

Many machine learning algorithms also suffer from a large set of variables that need to be tuned extensively to produce effective learning. But these variables often have (sometimes unexpected) relations to each other. Setting one variable might cause another variable to change in its former optimal value. This makes applying machine learning challenging, especially in hybrid architectures with multiple AI algorithms running at the same time. Each with its own set of variables.

Part II

Building A Robot
Analysis
Design
Implementation

Chapter 3

Analysis

The analysis section is meant to provide information to base design decisions on. After a short survey of the battle simulator, to view the limitations of the RoboCode engine, some observations are made on the game rules and physics.

3.1 Battle Simulator

The RoboCode battle simulator uses Java threads to run the battles. Apart from the main thread, the battle manager, every robot has its own thread. It receives input from the battle manager thread via an event queue. Each robot is given its own event queue. Events generated are listed in appendix **??**, but one should think of events like `RobotScannedEvent` or `HitByRobotEvent`. The logical code, i.e. the non-reactive behaviour, is specified in the *run* method, an integrated method part of a thread.

Some different types of classes can be extended to create a new robot. The most commonly used is `AdvancedRobot` as it allows queueing actions, custom events and interaction with the file system.

Access to the file system by the robot threads is limited through the Java security manager[1] that the RoboCode engine utilises. The robot thread is only allowed to read and write to any files located in a unique directory, but the combined size of all files in that directory can never exceed 200,000 bytes. If it does exceed the limit, the robot is automatically killed. These restrictions have been set to prevent robots from 'hacking' their way to victory, either by using large files that can crash the file system or by using malicious memory injection. At the beginning of every round the robot threads are reset and any objects, that are not defined to be static, are lost. To store information between battles the robot can only use the limited storage in its data directory.

During the rounds and at every turn the robot threads are woken up. The main thread then waits for the robot to finish its turn. That is, until the robot performs a blocking action call (e.g. *execute()*) or a set amount of computational time is used.

[1] The security manager is part of the Java virtual machine and can be disable for testing purposes.

In the latter case, the robot's turn is terminated immediately, this is called 'skipping a turn'.

The exact loop the engine goes through during runtime is as follows;

1. Battle graphics are (re)painted
2. All robots execute their code until they take action (and then paused)
3. Time is updated (time = time + 1)
4. All bullets move and check for collisions
5. All robots move (heading, acceleration, velocity, distance, in that order)
6. All robots perform scans
7. All robots are resumed to take new action
8. Each robot is processing its event queue

The next section details observations on the game rules and physics of the battle simulator.

3.2 Observations On The Rules And Physics

Just detailing the rules and phyics of the RoboCode environment (as in appendix A on 46) doesn't give true insight on how these affect battles or the programming of controllers for these robots. When designing artificial intelligence one looks at recurrence of certain episodes. These are an indication of when an performance measure can be made. How often can a robot fire a bullet? How many ticks does a bullet take to travel the entire battlefield? How long does one radar sweep take? In this section a few observations and calculations are performed to estimate the length of these episodes.

Normally every part of the robot turns along with the part it is mounted on, i.e. radar → gun → body. But this can be disabled, so that every part moves independently.

Rounds And Battles

The default duration of a battle is ten rounds. After a battle is fought the scoring of every robot is calculated. After every round the robots are once again placed on the battlefield with renewed energy. The number of turns in a single round can differ greatly and depends on the robots. Two evenly matched robots can last more then 4500 turns in a single round. The duration of an round is never infinite. If none of the robots hit each other (by way of a bullet or collision) for 450 consecutive turns, they start losing $\frac{1}{10}$ energy every turn. (450 turn is the default setting and can be customised). Since firing always costs energy and the energy returned from hitting an enemy robot is always lower then this, no round can last for ever. Winning a battle is therefor an episodic task. The maximum duration is dependent on the number of robots, .i.e. the cumulative amount of energy around.

Radar Observations

How long does one radar sweep take? How much can one sweep scan? The radar mounted on top of the gun can only scan in the direction the radar is pointing and has a maximum distance of 1200 pixels. In a small battlefield this means that you scan every robot in the direction the radar is pointed in. When two enemy robots are standing directly behind each other, the first one does not prevent scanning of the second. There is no limitation on the numbers of robots that can be scanned in one sweep. During one tick the radar can be rotated to scan a cone shaped area of the battlefield.

The radar can rotated at a maximum of $\frac{45°}{turn}$. If the parts are not set to move independent, the maximum rotation of the highest mounted part of the robot is dependent on maximum rotation of the lower parts, i.e. the gun (20°) and body(10°). The maximum rotation speed of the radar is $75° = 45° + 20° + 10°$ and the minimum rotation speed is $15° = 45° - 20° - 10°$. At a maximum rotation speed it takes $\lceil \frac{360}{75} \rceil = 5$ turns to scan the entire battlefield and at a minimum rotation speed it

takes $\lceil \frac{360}{15} \rceil = 24$ turns.

The radar can get the following information from scanning a robot: distance, energy, heading, bearing, name and velocity. Note that it cannot directly get the exact location, but this can be easily calculated using the bearing and distance. Also see the event list in appendix B on page 51

Gun And Bullet Observations

Firstly, it should be noted that bullets cannot be seen. They can not be scanned while in flight and the only way a robot can know of a fired bullet, is by detecting a slight drop in the *energy* of the enemy between two radar scans.

Secondly, the speed, generated gun heat and damage of a bullet depend on the *firepower*[2], see the equations on page 49. When firing, the gun generates heat and has to be cooled down before it can be fired again. RoboCode lowers the gun heat by 0.1 each tick, but the amount can be altered.

To get a general idea about how the firepower influences the travel time of the bullet and the fire frequency see tables 3.1 and 3.2 below. For easy and relative comparison, the numbers of pixels a robot can travel during the episode are displayed.
What follows from this, is that for firing at a moving target that is far away, a

Table 3.1: Traveltime of a diagonal shot bullet on a 800x800 battlefield

Firepower	∼Traveltime (*ticks*)	Displacement of target (*pixels*)
$0.1^{(min)}$	58	464
1	66	528
1.5	73	584
$3.0^{(max)}$	103	824

Table 3.2: Minimal delay between firing using default gun heat drop (0.1)

Firepower	Delay (*ticks*)	Displacement of target (*pixels*)
$0.1^{(min)}$	2	16
1	4	32
1.5	5	40
$3.0^{(max)}$	8	64

firepower of ~ 0.1 gives the target far less time to evade then firing with ~ 3.0. So when designing a firing strategy, a low *firepower* might be efficient when firing from

[2] $(0.1 < Firepower < 3.0)$

a certain distance.

Thirdly, as stated before, the damage a bullet impact does to a target is also determined by $firepower$. See equations on page 49 for the exact formula. The relation between gained/cost *energy* and *damage* to the target is best displayed using a graph (figure 3.1). Please note, that although a bullet shot with a low $firepower$ deals less damage then one shot with a high $firepower$ it can be fired more frequently. For good comparison the y-axis shows the *energy/tick* instead of the raw *energy* gain or loss calculating in the frequency of the bullets.

If the robot can be reasonably certain it can predict the target's future location, it is effective to use at least $firepower > 1$, as this does extra damage. The graph shows that, if every bullet hits, firing at maximum $firepower$ will bring the targets energy down faster. But if the target is very versatile and evades a lot, the robot loses more energy the stronger the utilised $firepower$ is.

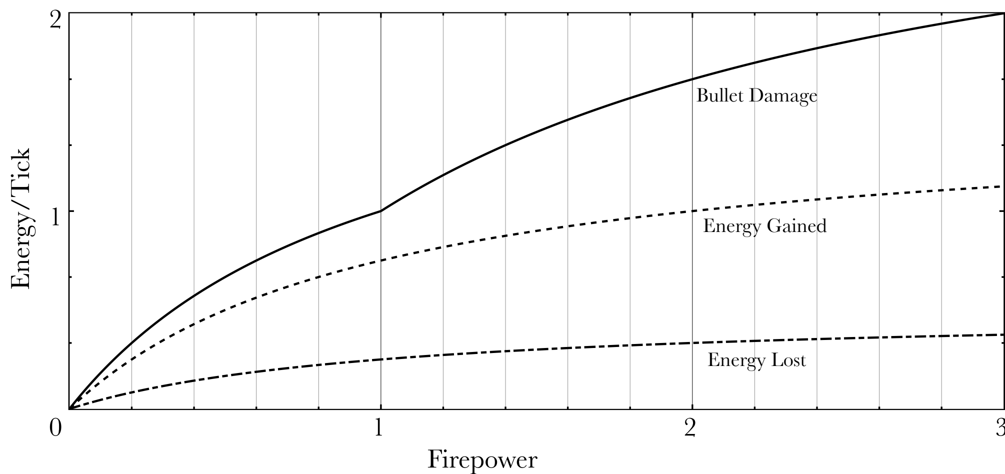

Figure 3.1: Energy gained and energy losses per tick set against firepower, using default gun heat drop of (0.1)

Movement Observations

How long does it take to cross a battlefield? As stated on 48, the maximum movement speed is 8 $\frac{pixel}{turn}$. With the maximum acceleration it takes 8 turns to accelerate to a velocity of 8 $\frac{pixel}{turn}$ moving 36 *pixels* while accelerating. It takes 4 turns to decelerate when moving at maximum velocity, moving 20 *pixels* while doing so.

If the distance is smaller then 56 *pixels* the equation for the number of turns(t)

it takes to move distance (d) of $pixels$ is:

$$t = (8 + 4) + \frac{d - (36 + 20)}{8} \quad (3.1)$$

What this means for the travel time across a battlefield can be seen in table 3.3.

Battlefieldsize (pixels)	Traveltime (turns)
$400x400^{min}$	55
800x800	105
$5000x5000^{max}$	630

Table 3.3: Turns for moving along one side of the battlefield

Observation Summary

Concluding on the earlier observations, the following can be observed for events on a battlefield of 800 x 800 $pixels$. A robot takes;

~100 turns to move from one side to the next

~60 turns to shoot a bullet with $firepower = 0.1$ across the battefield

~100 turns if fired with $firepower = 3.0$

~50 shots in 100 turns with $firepower = 0.1$ dealing 20 damage while gaining 15 Energy

~13 shots in 100 turns with $firepower = 3.0$ dealing 28 damage while gaining 117 Energy

Usually the environment is classified by properties as defined by Russell and Norvig (2003). The task environment properties that hold for RoboCode are as follows;

Partially Observable the robot sensors provide limited information

Strategic the environment is deterministic except for the actions of other robots

Sequential current actions reflect upon the cumulative reward

Semidynamic the game pauses while the robot has limited time to select an action

Continuous for example, the state of the environment contains angles n where $n \in \mathbb{R}$

Multiagent a battle can be fought between an unlimited number of robots

These values and properties will be used to provide information for the following design chapter. Especially machine learning methods that need to evaluate the system

after certain events can use this information to determine the length of certain events. The observations made in this chapter can also be used to develop and evaluate strategies already available via the community.

Chapter 4

Design

This chapter is meant to provide details on the design and the design choices before I commence the implementation and testing. First the overall design is explained. Then, because the design is based on the reinforcement learning framework provided in part I, the reinforcement signal designs are explored. Detailing the update equation will be at the near finish of this chapter.

Overall design

Several projects already exist that implement some form of machine learning into a robot, see section 2.2 on page 10. Some of the projects use a single machine learning algorithm to map the raw events of RoboCode onto the most basic level of actuators. That is, for example, turning the gun or moving the robot forward.

This gives a highly adaptive robot, but one that is very slow in the adapting. If given enough time the robot controller using it can learn to counter any robot it faces. The drawback is that it loses precious time before it shows any sign of even the simplest effective strategy. When facing a single opponent and the only goal is to win, eventually, this does not pose a problem. But when facing an unfamiliar robot the trained controller generally loses really quick and requires a large amount of training before it can perform adequately again.

Drawing inspiration from the many effective hand coded strategies that exist in ranked RoboCode robots and the conclusions of the reports from other projects, I suggest a high-level machine learning design. Others use machine learning to learn via the interpretation of the low level sensors input and control of the low level actuators. By using tested behavioural strategies as actuators (output) and highly pre-processed sensor data for input the resulting robot controller could adapt its behaviour through trial-and-error. Reinforcement learning is presumed to be adequately suited for this.

Instead of selecting a certain action every turn, the robot controller will select one behavioural strategy to govern a part of the robot for a certain amount of turns. After which it will evaluate its selection and re-select (new) strategies. Since the robot's anatomy consists of three parts: the radar, the gun and the body, the strategies will

be divided among these three categories. A choice for an action will in fact be a composition of strategies into a triplet. Not all behaviour of the robot is provided by the selected strategy. Some behaviour is designed to be deliberately omitted or hard coded into the robot, like firepower, these are listed later on in this chapter.

This high level design has several benefits as well as some drawbacks.

Benefits

A benefit from the design is that the robot controller will not require a lot of training before it can battle adequately. Right from the start it will be able to function, although not completely effective. Ideally, the robot controller will quickly find the strategy that works best against the enemy's strategy and learns to use a possible counter strategy. By using pre-processed sensor data the possible number of states can be reduced considerably. This allows for less computations and quicker convergence of a value function to its optimal value.

Drawbacks

The robot will require a set of strategies and these will have to be composed by human insight. This is, as with anything really, subjected to human error. Selecting the wrong or to few elements for the set might cripple the adaptive powers of the controller. In general I think the adaptive power of the robot is speed up by this design, but by a manner that limits its adaptive flexibility. Some of the strategies, if not all, might require constant updates from the environment, this comes with an extra computational cost. The strategy selection procedure should try and avoid this. Design choices will also have to be made in respect to the pre-processing of the sensor data. This is also subject to human error.

4.1 The Reinforcement Learning Framework

Action Signal

As mentioned in the earlier section, a single action equals a strategy triplet, one category for each part of the robot. Ideally the complete set of strategies consists of strategies that have a counter-strategy relationship to each other.

If the enemy can counter gun strategy *gun-A*, we would like to have a strategy *gun-B* that will always work when strategy *gun-A* is being countered. This should also hold the other way around. That is, if gun strategy *gun-B* is being countered, gun strategy *gun-A* should still be effective. The general idea is, that there should not exist a counter strategy that counters all available strategies of the robot any given time.

The source of the strategies will be (if not explicitly provided for) either the

robowiki[1] or the higher ranking robots from the *RoboCode repository*[2]. For convenience and due to limited resources, the numbers of strategies is limited to twelve, that is four for each of the three categories. A default strategy is added to each of the categories to provide a passive strategy for the controller. Using this passive strategy might not make much sense for the radar behaviour, but for the gun and body it might be useful. Movement of the body can disrupt the aiming accuracy for the gun. Not firing the gun might be useful in a situation where the enemy can easily be rammed to death, since this gives bones points.

Some testing occurred before selecting the strategies, the general gun and body strategies each differ greatly in the community. However, the strategy for controlling the robot's radar (in an one on one fight) is always the same. A perfect lock can easily be achieved and is the best way to use the radar. Given this fact the choice is made to let the controller always use perfect radar lock strategy. In table 4.1, all strategies that have been considered for the radar are listed. In the tables 4.2 and 4.3 the chosen gun and body strategies are listed.

Table 4.1: Scanning Strategies

Infinity Scan	Just spin the radar around as fast as you can, scanning every part of the battlefield as fast as possible
Perfect Scan	Find the enemy robot. Then turn the radar along with the targets movements
Gun heat Scan	Locks onto a target if the gun has low gun heat, but spins it around like Infinity Scan otherwise
Disabled Radar	Do not use the radar at all

State signal

The design will eventually introduce some kind of value function to let the robot learn via reinforcement learning. If no function approximation is being used, the most logical way to represent a function would be a look-up-table, or a map. The idea is to have the robot controller remember the learned information from the last match and utilise this in the next one. This means that the information needs to be stored in the limited storage space for the robot. This implies a limit on the table entries and therefore on the number of states and state-action pairs.

RoboCode has a lot of variables that are continuous in value. For example the Cartesian co-ordinates on the battlefield or the heading angle of the robot. When the state space is not finite, the mapping cannot be either.

This can be counter by discarding all the information and input the robot does not need to decide on a strategy and use segmentation on the the remaining sensor

[1] http://robowiki.net/
[2] http://robocoderepository.com/

Table 4.2: Targeting Strategies (Gun)

Head-On Targeting	Using the radar's information to fire at the enemy's last known location.
Circular Targeting	Using the radar's information to predict the future position of the target. It uses turn rate, speed and direction of the enemy.
GuessFactor Targeting	A more advanced type of targeting. If a target is scanned, it can only move for a certain distance given the speed of a bullet and the distance of the enemy (maximum escape angle). By constructing a guess factor the robot aims somewhere in this region. Also see appendix C on page 53 for more information.
Disabled Gun	Do not use the gun at all.

Table 4.3: Movement Strategies (Body)

Random Movement	Moves the robot perpendicular to the enemy, oscillating at a random frequency and slowly closer to the enemy.
Ram Movement	Move towards the target at full speed and ram into him. Move back and repeat previous action.
WaveSurfing Movement	A more advanced movement strategy. By estimating when a enemy's bullet would reach the robot it reacts accordingly. It estimates by way of waves radiating from the location the enemy robot shot from, hence the name. Also see appendix C on page 53 for more information.
Disabled Body	Do not move the body at all.

input. The latter entrails dividing the infinite value range of a sensor into a finite number of value intervals. Because the robot is designed to function on a higher level I think it is possible to segment the values to a high extend without losing a large part of its learning capacity.

After the segmentation the state space should have some properties that are useful when learning. For example, using the exact co-ordinates of the robots would be very inefficient. When using the co-ordinates the robot would differentiate between states that are alike except for the fact that one of the states has a clockwise turned battlefield, as seen in Figure 4.1. The state parameters in table 4.4 are suggested

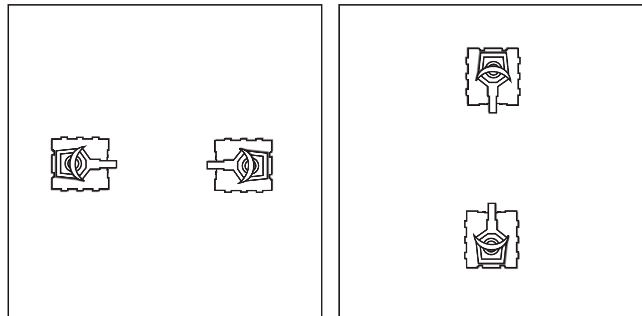

Figure 4.1: Two situations the robot should not differentiate between

to be used in the state representation. The distance between the robots and the general location is meant to provide the useful information that could otherwise be deduced from the exact Cartesian co-ordinates or the robots. The energy of both robots should suffice to provide the knowledge of how the battle has progressed, e.g. if one robot has the upper hand and how both robots have been performing until the current state.

Now that the sensor input variables have been determined, the segmentation of these have to be considered. It is assumed that the lower the maximum number of possible states, while still retaining all the relative data, the better the robot controller will be able to learn. Skipping ahead to the implementation, it should be noted that there is also a limit on the storage capacity of each robot for between battles, namely 200,000 bytes. If we store the values from the value function as a *float* Java type integer (32 bits) we can store up to 50,000 different values. Also see *Table Storage* on page 35.

This is the upper limit, it is presumed we can do with far less. In the rest of this section the parameters and their segments are elaborated.

Distance Segmentation (3 Segments)

The distance between two robots is divided into three ranges: *CLOSE*, *MEDIUM* and *FAR*. Conform the following distribution:

Table 4.4: Suggested state parameters, practical range on a 800 by 800 battlefield

State Parameter	*Practical Range*	Suggested Segmentation
Distance To Target	$[0 \leftrightarrow 1130]$	$[0 \leftrightarrow 200 \leftrightarrow 600 \rightarrow \infty]$
Energy Target	$[0 \leftrightarrow 100]$	$[0 \leftrightarrow 10 \leftrightarrow 25 \leftrightarrow 45 \leftrightarrow 100]$
Energy Robot	$\Uparrow$	$\Uparrow$
Locatio nTarget	$[0 \leftrightarrow 800, 0 \leftrightarrow 800]$	{EDGE, CORNER, CENTRE}
Location Robot	$\Uparrow$	$\Uparrow$

$$\text{Distance}(p) \begin{cases} \text{CLOSE} & \text{if } p \leq 200 \\ \text{MEDIUM} & \text{if } 200 < p \leq 400 \\ \text{FAR} & \text{if } 400 < p \end{cases}$$

With p being the real distance between the robots. If the robots have more than half of the battlefield's width between them, they are considered to be FAR apart.

Energy Segmentation (4 Segments)

The philosophy for the energy segmentation is that as the energy levels get lower, the importance of the difference between energy levels increases. The energy level spectrum has more bins as the energy gets lower. The energy of a robot is divided into 4 ranges conform the following distribution:

$$\text{Energy}(e) \begin{cases} \text{I} & \text{if } e \leq 10 \\ \text{II} & \text{if } 10 < e \leq 25 \\ \text{III} & \text{if } 25 < e \leq 45 \\ \text{VI} & \text{if } 45 < e \leq 100 \end{cases}$$

With e being the real energy of the robot. Or shown in a more graphical way:

General Location Segmentation (3 Segments)

The philosophy behind the general location segmentation is that when a robot is near the edge or in a corner on the battlefield, it has less flexibility in its movement. This gives the robot a disadvantage and should be taken into account in the state parameter. Together with the distance the robot can tell whether it stands in the same corner as the enemy. The segmented areas are as follows:

When a robot drives directly at the wall at maximum speed, it must start turning it's body at around 150 distance from the wall in order to evade it at maximum speed.

This will be the limit of when the robot will in the EDGE area. If it comes within 212 ($\approx \sqrt{150^2 + 150^2}$) distance of a corner it is considered to be in a CORNER area. Everything else is considered CENTER. These areas are shown in figure 4.2.

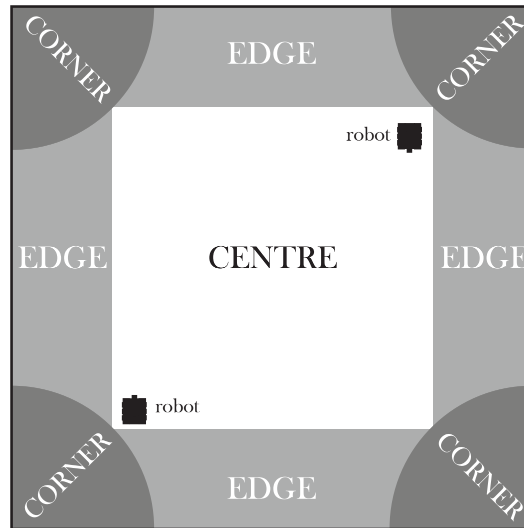

Figure 4.2: General Location Segmentation

Storage capacity

With the above number of bins for each state parameter, the total size of the state space becomes:

$$3 \times 4 \times 4 \times 3 \times 3 = 432$$

This is an excellent size considering the maximum of 50,000 determined earlier. But in reinforcement learning we usually work with tuples of actions and states. Omitting the radar strategy, there are 16 possible strategy tuples (4^2). The number of possible action state tuples reaches 7,000 ($432 \times 16 = 6912$), still well below the maximum.

Next is the design of the reward signal.

Reward signal

The reward signal is provided to the robot via the RoboCode engine itself. After the match, the engine constructs the score of the robots based on the scoring rules[3] of RoboCode. But does so only at the end of every battle. The total score of the battle consists of the sum of the score perceived every round. Since the end of the round

[3]See page 46 in appendix A

is the only way to accurately calculate the performance, due to survival bonus, the reward signal is set to give a numerical reward based on the scoring at the *end of every round*.

No other performance measure is used in the reward signal in order to prevent sub-optimal behaviour due to not learning with the actual reward. For example, if the robot would receive extra reward for firing as few bullets as possible, it might stop shooting altogether.

4.2 Action Selection

How often the robot has to choose an action is very important to establish a learning scheme. If set too often, the controller takes forever to learn the consequences of its actions on the long run. But if it only has a choice once every battle, the controller's choice will not have any influence at all. Since very little happens in a single turn, choosing an action every turn would result in the former. Then what is the optimal frequency?

In theory, the frequency of choosing an action should synchronise with the frequency on which the strategies can be evaluated or said to have any result. When evaluating a gun strategy, the strategy should have the time to fire a set of bullets and see if these hit or not. The same goes for a body strategy evaluation. That is, a duration long enough so that bullets can reach the controller in order for the strategy to show how well it can dodge them and evade walls at doing so.

Following the observations made in analysis of the game physics in section 3.2, it takes 100 turns to fire a bullet diagonal across the battlefield with the highest firepower (slowest bullet). Taken into account that none of the smart robots will keep standing in the corners for long, it can be estimated that firing a bullet and having it travel across a reasonable distance will take approximately 80 turns. A gun can fire at a minimum frequency of once every 8 turns, that is at maximum firepower. If 3 bullets are allowed to be fired, the strategy can best be evaluated after 104 turns ($= 80 + (3 \times 8)$). To be on the safe side, the frequency of choosing a strategy is set to once in every 105 turns.

Actions will be selected using the Boltzmann probability distribution mentioned earlier on page 7:

$$\text{Probability of selecting action } a_1 \text{ given state } s = \frac{e^{Q(a_1,s)/\tau}}{\sum_{b=1}^{n} e^{Q(a_b,s)/\tau}}$$

With Q being the state-action value function, discussed later on. The parameter τ that controls the amount of exploration that needs to be set. For early rounds it can be a considered a good idea to let the controller explore all its actions. But when reaching the last rounds of the battle it should exploit more. This can be done by slowly lowering the temperature. The exact value will have to be tested in the implementation chapter.

After having discussed the design of the reinforcement learning framework the next section continues designing how the robot could actually learn using the framework.

4.3 Update Equations

Using a policy independent temporal-difference (TD) update equation, the robot controller will learn to map state-action tuples to the estimated utility. The above mentioned action selection will select actions by a probability mostly determined to this state-action value.

The one-step Q-Learning as introduced by Watkins (1989) and modified by Sutton and Barto (1999) will be used to improve the mapping toward the optimal value function.

The Q-Learning update equation is defined by:

$$\overbrace{Q_{t+1}(s_t, a_t)}^{\text{new value}} \leftarrow \underbrace{Q_t(s_t, a_t)}_{\text{old value}} + \alpha \overbrace{\Big[\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{max_a(Q_t(s_{t+1}, a_t))}_{\text{max future value}} - \underbrace{Q_t(s_t, a_t)}_{\text{old value}} \Big]}^{\text{error}}$$

Before implementing one-step Q-Learning, the framework is tested. To see if the obviously good states have, relative to the obviously bad ones, a high value and vice versa. This testing of the framework is done by using the TD(0) update equation, which is as follows

$$V_{t+1}(s_t) \leftarrow V_t(s_t) + \alpha \left[r_{t+1} + V_t(s_{t+1}) - V_t(s_t) \right]$$

The variable that needs to be tested for both the update equations is the α or the step-size variable. This determines the amount with which the old estimated value is to be updated given the newly received reward, this is generally referred to as the *error*. If set it to $\alpha_k(a) = \frac{1}{k}$, that is the kth selection of action a, there is absolute certainty the value function will converge to the optimal value function. But only by the laws of great numbers, which means (almost always) very *very* slow. In practice a constant value for α is used. This does not ensure exact convergence since it can lead to constant overshooting the targeted optimal value function, but in practical implementations this is not a real issue. The tested values will be around 0.1 as the value has proven itself in other practical implementations.

To summarise, the values of τ and α need to be adjusted and tested to see what the best values are.

Hard Coded Behaviour

The firepower of the fired bullets is designed to be beyond the control of the robot controller. As observed in section 3.2 the firepower can best be low if the target is

far away and set to maximum if the target is very close. In synergy with the distance segmentation the firepower will be calculated as follows:

$$\text{firepower}(distance) = \begin{cases} 3.0 & \text{if target is CLOSE } (< 200) \\ 1.9 & \text{if target is MEDIUM } (> 200 \text{ and } < 400) \\ 0.1 & \text{if target is FAR } (> 400) \end{cases}$$

This is the only aspect of behaviour the robot controller has no direct influence on.

Chapter 5

Implementation of the design in Qbot

In this final chapter the design is implemented in the robot named Qbot. The first section is to layout the details of the robot controller, followed by some preliminary testing of the state value function. After that, testing of the complete robot is done followed by a conclusion and discussion.

5.1 The Java Program

Since Qbot is implemented in Java, an object orientated programming language, a object orientated approach has been chosen. A simplified overview can be seen in figure 5.1.

The first object in the overview is Qbot which *extends* AdvancedRobot, the standard advanced robot class. This is the actual robot.

Because RoboCode is event driven and the robot has no access to the RoboCode engine itself, every object that requires to be updated needs to receive them via Qbot. A distributor, InformationDistributor is implemented to distribute all the information and events received by Qbot. Via the distributor the Learner constructs a Scoring and a WorldView which in sequence produce a *reward* and a *state* from the environment.

The Learner gives feedback to Qbot in the form of a triplet of strategies. The strategies are stored and managed in the StrategyCollection,. Each strategy has a link to Qbot. Via this link a strategy can issue commands to the robot, but does this only if the strategy control status is set to *active*.

When setting up a new battle the ValueFunction object is constructed and the look-up table is loaded from the file system. At the end of a battle the ValueFunction writes the look-up table back to the file system.

An approximation of the program flow can be seen in table 5.1 on page 35.

Figure 5.1: Overview of the Java objects

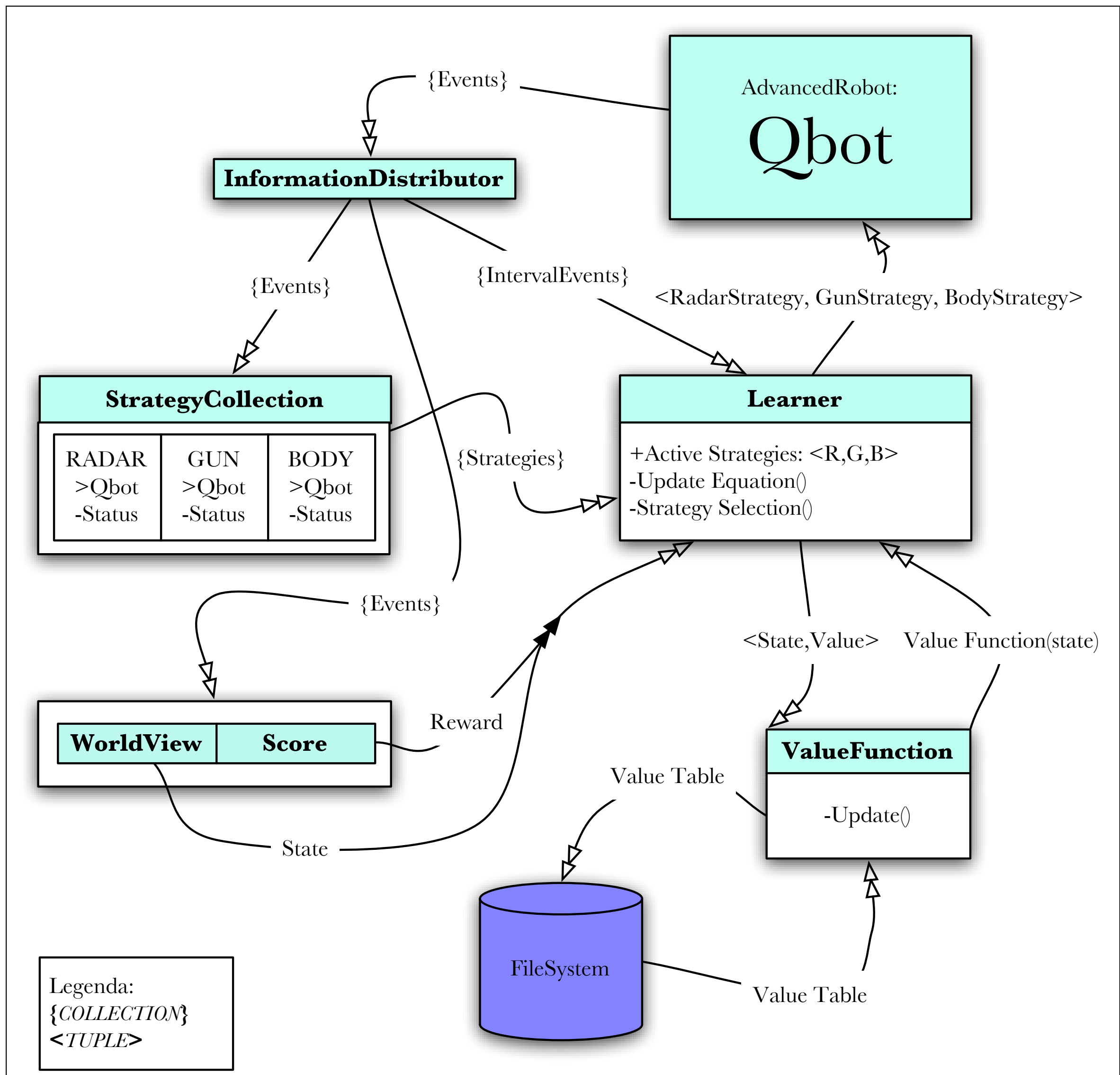

Table 5.1: Program flow

1. The battle is set up, all objects are constructed via `Config.java`:
   (a) All objects are linked to the `InformationDistributor`.
   (b) The `ValueFunction` imports the value function.
   (c) The `Leaner` initialises the start *state* and selects its starting *strategies*.
2. During every round:
   (a) Every turn the selected strategies are allowed to send control commands to the robot.
   (b) At a set interval and at the end of the round the `Learner` requests the score and a state from `Scoring` and `WorldView`. It then re-evaluates via `ValueFunction` and selects new strategies.
   (c) At the start of the round `Scoring` and `WorldView` are reset.
3. At the end of the battle the `ValueFunction` exports the value function's look-up table.

Table Storage

The storage of the look-up table is somewhat more advanced and deserves to be elaborated. For preliminary testing, state values need to be stored. As data structure an hash map is chosen, only the used array is stored.

When the `Learner` requests a state value by providing a `State`, the index of its value in the array is calculated by a hash index function. To understand how the index hash function works, imagine having a set of digits ($\{a, b, \ldots, y, z\}$) which all have a different value range. The hash index function that maps a set of these values to to an unique value between 0 and the maximum possible values minus one (e.g. an effective array index) is as follows:

$$\text{hash index} = a + \Theta(a)[b + \Theta(b)[\cdots[y + \Theta(y)z]]$$

With $\Theta(a)$ being the number of possible values that a can have. In the normal decimal system this would be 10 (range: $0 \leftrightarrow 9$). If trying to map two decimal integers x and y to an index of $0 \leftrightarrow 99$, the hash index function would be $x + 10y$.

A `State` is a container for 5 state parameters, 3 of which have 3 different segments (distance and locations) and 2 of the parameters have 4 different segements (energy). The hash index function for a State with distance: d, enemy's energy: e^{enemy}, Qbot's energy: e^{Qbot}, enemy's location: l^{enemy} and Qbot's location: l^{Qbot} would become:

$$\text{hash index} = d + 3(e^{\text{enemy}} + 4(e^{\text{Qbot}} + 4(l^{\text{enemy}} + 3l^{\text{Qbot}})))$$

By using this (perfect) index hash function, storing any state information becomes obsolete and we can get away with only storing the values. I believe this is the most efficient way of saving the value function. Since the values are being stored as a *float* (32 bits) and the maximum storage capacity is 200,000 bytes. Excluding any overhead generated by the array, the robot can store a maximum of

$$\frac{200,000 \times 8}{32} = 50,000$$

entries. Given the segments for each parameter, there is a maximum of 432 possible states. The preliminary testing should not face any storage problems.
For final testing, storage of action-state pairs is required. With the radar strategy: s^{RADAR}, the gun strategy: s^{GUN} and the body strategy: s^{BODY} the hash index function becomes:

$$s^{\mathrm{RADAR}} + 1(s^{\mathrm{GUN}} + 4(s^{\mathrm{BODY}} + 4(d + 3(e^{\mathrm{enemy}} + 4(e^{\mathrm{Qbot}} + 4(l^{\mathrm{enemy}} + 3l^{\mathrm{Qbot}}))))))$$

Note that there are 4 different strategies available for the robot in the gun and body strategy type categories (Qbot only uses one radar strategy). The total required number of states to be stored then becomes: $432 \times 4 \times 4 =$ 6,912. Still well below the calculated maximum of 50,000.

5.2 Testing

The first test is whether the state values are according to what we would expect from the state's situation or not. The second test is to see if the robot improves over time and what the best settings are for the earlier mentioned temperature and step-size variables. Testing was done using RoboCode version 1.7.1.2. In this version a minor glitch was found during testing. The calculated bullet and ram damage bonus were incorrect by a minor percentage. The percentages used by the `Scoring` system of `Qbot` were adjusted to compensate. The glitch was reported and is being fixed in version 1.7.1.3. It should not influence test results.

State Value Function Tested

`Qbot` is set up against `sample.Walls`, `sample.Fire` and `RaikoMX 0.32`[1] to get a diversely trained state value function. Note that `Qbot` does not yet learn, it is set to use only circular targeting for the gun and random movement for the body. When using these strategies, `Qbot` generally wins from the sample robots and loses from the more advanced one. Each opponent robot is set up against Qbot for 10,000 rounds. Every round the two robots are placed at random on the battlefield. Apart from the number of rounds in the battle, every other custom setting is left to the default value.

[1] A robot by Jamougha. When tinkering around with RaikoMX 0.32 it showed that it could easily beat a simple strategy deploying Qbot.

The values of the states are being learned/updated using the TD(0) update equation with a step size of 0.1, see page 31.

The testing is done by looking at the parameter value in each state and correlate this with what the value function produces when inputting the state. First I evaluate the relation between energy parameters and state values. Then the relation of the state values and distance parameter is evaluated. Lastly the general location's relation with the state value is elaborated. It should be noted that the true value of the state depends on the combination of all these factors. But looking at each of the parameter as a single factor it can generally be determined whether the state value's abide to intuitions.

The expected relation of the energy levels with the state value is quite simple. It is expected that the difference between the energy levels of the enemy and Qbot has a correlation with the state value. If the enemy robot has a higher energy level than Qbot, the state value should be lower then when Qbot has the superior energy level. The graph in figure 5.2 on page 37 shows the real relation, as derived from the test data. The graph shows the difference between the energy levels.

Note that the energy segments were fourfold: $lowest(0)$, $low(1)$, $high(2)$ and $highest(3)$. The value on the horizontal axis equals $energy^{Qbot} - energy^{enemy}$. Minus 3 stands for the worse possible value, with Qbot almost disabled at 10% energy left and the enemy still standing strong with more than 45% energy. Positive 3 is the best situation, with a reverse of previous told energy relation. As one can see, the

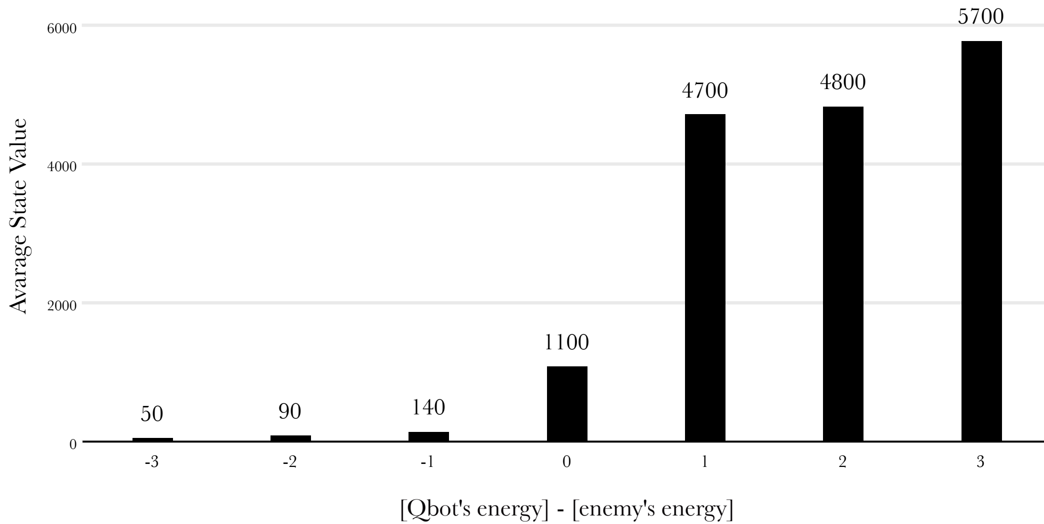

Figure 5.2: Energy ratio plot against avarage state value

relation is the same as expected. It should be noted that there are fewer states in the test data representing the negative spectrum than there were states for in the positive spectrum. Making the calculated average less precise on the negative side.

This is due to the way energy is uneven segmented, having a higher resolution at lower energy levels comes with its price.

Still, there is an obvious decrease in state value as the energy levels are less advantageous for Qbot. In so far energy level parameters are concerned, TD(0) learns the expected values for these states.

When looking at the distance between the robots, it is not directly clear what the expected relation should be. Does Qbot generally win if the competitors are close or when they are far apart? The state value is expected to be highly dependent on what type of robot Qbot faces. One could say that when the enemy is far away, the chance that it dodges a bullet from Qbot increases. And that it is therefore less advantageous for Qbot. But this goes for Qbot as well; it has a better chance to dodge an enemy bullet when the enemy fires from afar. The graph in figure 5.3 shows the relation as derived from the test data. As one can see, there does not appear to

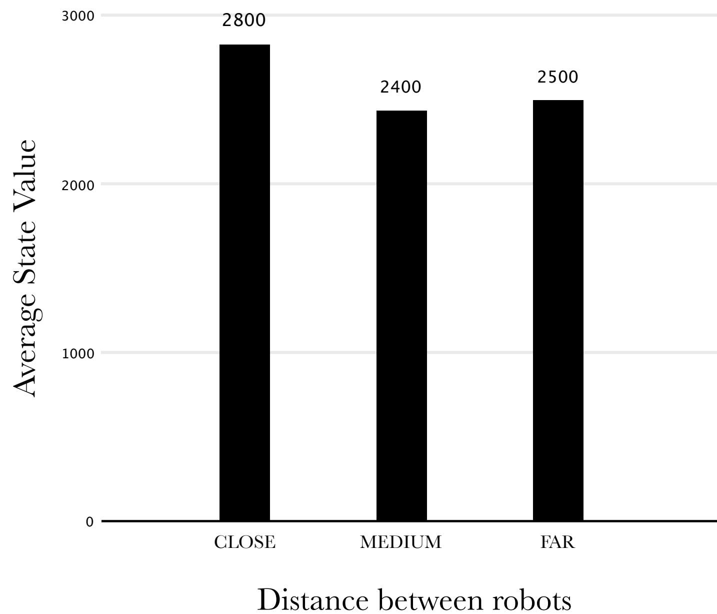

Figure 5.3: Distance plot against average state value

be a single distance that stands out as the best nor is one that is obvious the worse. It is believed that this is due to the state value's high dependency on the strategy used by the enemy robots. Due to this dependency it cannot be said yet whether the distance parameter segmentation has been a bad choice for representing a situation on the battlefield or not.

The expected state value relation, as far as the general location is concerned, is that how closer the robot is to the wall, the less advantageous its position and therefore state should be. Being in the corner would be the worse situation, since being there increases the chance of a wall collision and decreases the dodge changes of enemy fire

. The graph in figure 5.4 shows the location of the enemy robot and that of Qbot in a state, plot against the state value. The enemy's location graph shows that the value

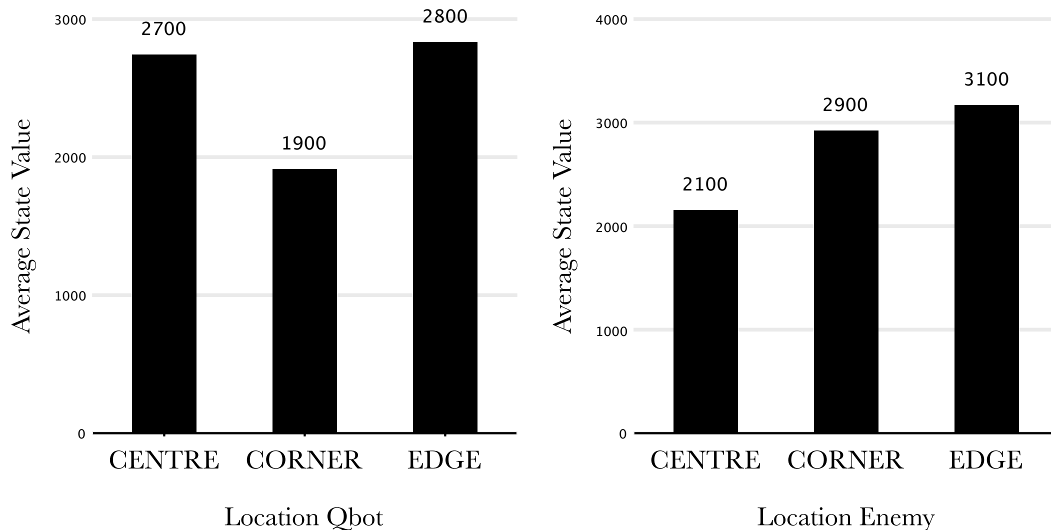

Figure 5.4: Location plot against average state value

function learns to expect a higher reward when the enemy is in a corner then when it is at the centre or near an edge, as was expected. Qbot's location shows that it is not a very good situation when being in a corner, due to the nature of the random movement (oscillation perpendicular to the enemy). Why the value function shows that it is almost equal advantageous to be in the centre as it is to be near an edge is not clear.

When looking at the enemy's location it can be said that segmenting the general location is successful.

This concludes preliminary testing. The state values are generally what I expected them to be, it now remains to be seen if `Qbot` has the capability to learn from these values and actually improve over time.

Variables α And τ Tested

The real update equation is that of one-step Q-Learning. See equation 4.3 on page 31. The effectiveness of learning by way of an update equation and using a Boltzmann-distribution action selection greatly depends on the step-size used for the equation and the temperature used for the distribution (α and τ). Please remember that the step-size's value range was $0 < \alpha \geq 1$ and the temperature's value range was $0 < \tau$.

The step-size regulates how much weight the update equation hangs on newly found state(-action) values. If set to 1 it replaces the old value completely and turns into a look-ahead-one-step-at-a-time machine. The temperature regulates the exploration of the robot, if set hot the controller explores extensively to find the coldest refreshing beverage. If set close absolute zero point, the robot thinks there

is a second ice age coming and stacks greedily on all survival supplies. In respect to action selection, it will always select the action with the highest known state-action value.

A variety of values for one variable cannot be tested correctly without setting the other a constant value. Because it is generally the case that the value of an effective temperature variable differs greatly between applications – sometimes with a multitude of 100 – and a step-size of 0.1 seems to almost always work, I choose to start varying the temperature first.

Testing was done by setting up a range of battles between `Qbot` and two selected robots from the samples, `sample.SittingDuck` and `sample.Fire`. The first does exactly what its name entails. It sits around doing pretty much nothing. The second one moves for or backwards when hit by a bullet, turns its gun and radar and fires when it scans an enemy. SittingDuck is used as a control, to see if the Qbot's score improves at all. Fire is used to see if it still does so against a somewhat more dangerous opponent.

At start the two competing robots are placed on a 800×800 battlefield on fixed locations. By using fixed location the random factor of starting location, that might influence learning, is taken out of the testing. The first robot is placed on co-ordinates $(150, 150)$ the second one on co-ordinates $(650, 650)$. Both are facing the corner they are set in. Each battle lasts 10 rounds and a total of 500 battles have been fought each trial. All customisable battle settings are default. With a step-size of 0.1, trials with a temperature of 20, 40 and 400 are run. The graph in figure 5.5 shows the results. As should be quite clear, the trail with the lowest temperature scored best. A temperature τ of 20 seems to perform better than a temperature of 40 or 400. To test if further lowering the temperature had any benefits I ran another set of trials with temperatures set to 10, 20 and 30. The results can be seen in figure 5.6 on page 41. Using a temperature τ of 30 seems to lower the score, while using 10 does not seem to be an enormous improvement when comparing it to the scoring gained by using $\tau = 20$. It is therefore concluded that 20 is the best temperature for when fighting against `sample.Fire` and `sample.SittingDuck` and this is generalised to all opponents.

Now that an optimal temperature has been established under the assumption that setting the step-size α to 0.1 is efficient, we should look at the step-size variation itself. Setting the temperature to a somewhat exploring value of 40 I have run trials with the step-size values of 0.1, 0.2, 0.5 and 1. In figure 5.7 on page 42 the results are displayed. In the trials with sample.Fire, using step-size 0.1 quickly establishes the highest score. In the trials with sample.SittingDuck it takes a while but ultimately it shows higher adaptation compared to the rest of the step-sizes. As one might have noticed, the step-size of 1 results in a strange scoring average. After performing some more tests I found that setting the step-size to 1 produces some random stabilisation on an initial score. At what value it would stabilise was depended on the randomly chosen actions in early matches. See the graph on page 57 for example trials. Using a step-size of 1 results in an unreliable scoring without any noticeable learning.

Trying runs with a step-size lower than 0.1 resulted in slower learning and the

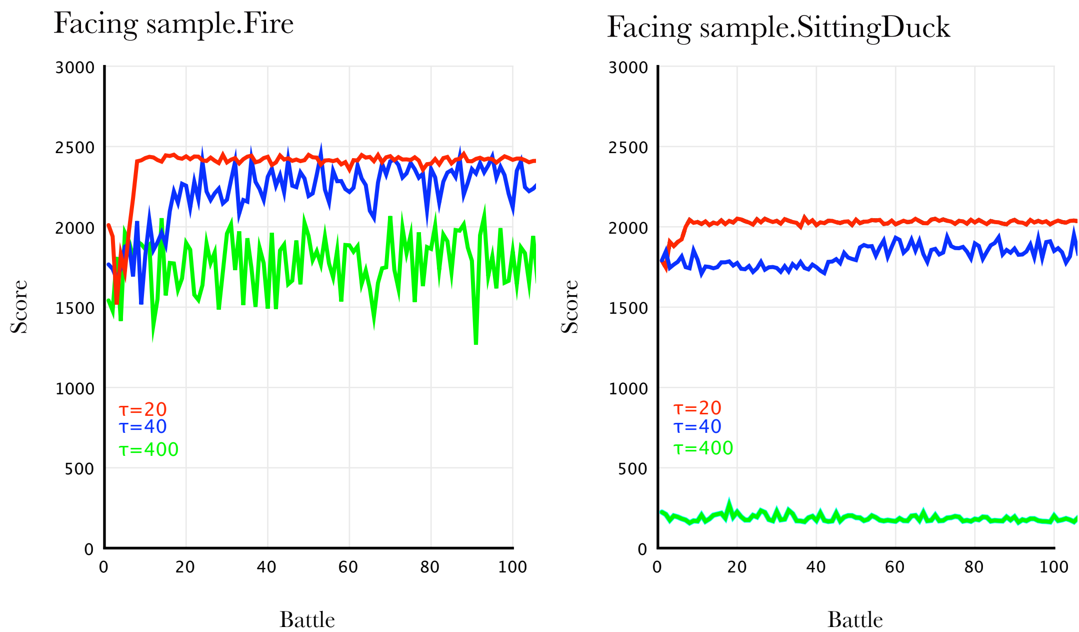

Figure 5.5: Trials of varying temperature with scoring of Qbot using Q-Learning with a step-size α of 0.1

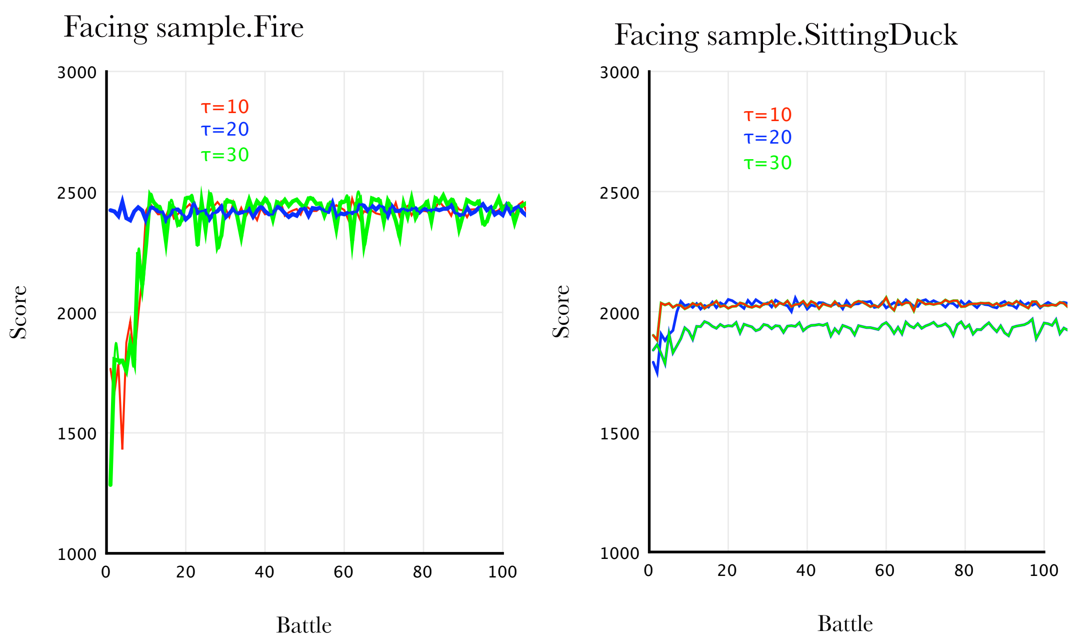

Figure 5.6: Trials of varying temperature with scoring of Qbot using Q-Learning with a step-size α of 0.1

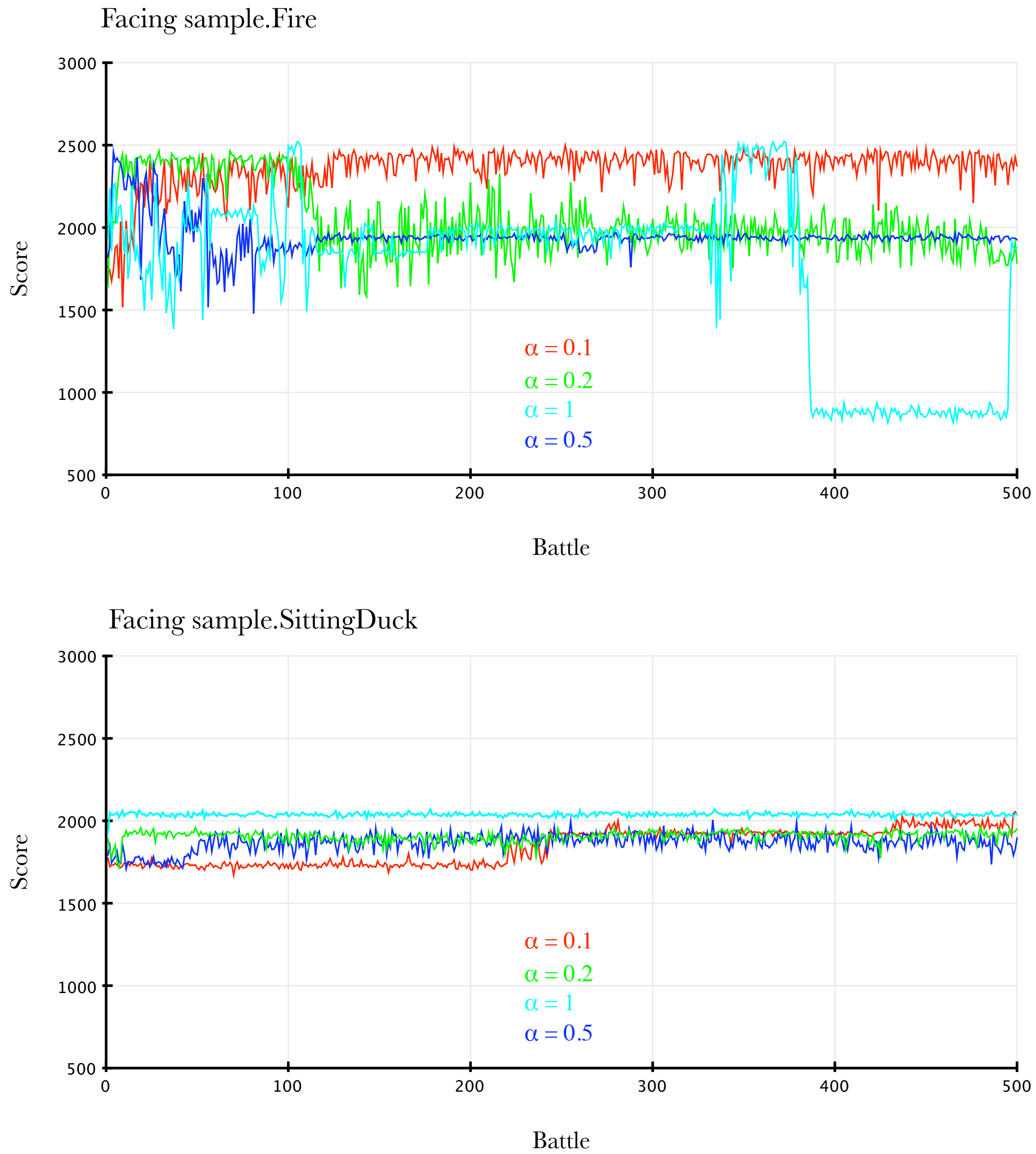

Figure 5.7: Trials of varying step-size with scoring of Qbot using Q-Learning with a temperature τ of 40

same score at the end of the trial. It is therefore concluded that a step-size of 0.1 is best. For reference; I added some test graphs for step-size 0.01 and some other test trails, i.e. against more advanced robots, in appendix D on page 56.

5.3 Conclusion

After a short survey of reinforcement learning and RoboCode, a robot was designed using this knowledge. By implementing Q-Learning and using higher-level states and actions the robot has shown lot be effective at early stages of the game as to able adapt itself rapidly to the enemy's strategy. A step-size α of 0.1 and a temperature τ of 20 resulted in the best learning against the simpler robots. Exhaustive training against more sophisticated robots was omitted due to time restraints. It is expected however, that with the right set of strategies and update equation, a robot can be trained to beat the highest ranking robots. The results of a small test trial against a former #1 top ranking robot can bee seen in appendix D on page 57. The score of the design implementing robot `Qbot` slowly improves as the learning progresses. Designing a robot that satisfies the goal, to show improvement against in an one on one fight, is a success.

As a concluding remark I would like to point out that, even though hard coded behaviour seems to work really well within RoboCode, there is always some incomplete knowledge or prediction power needed to control a robot tank in an intelligent fashion. Many forms of machine learning exist that have been designed and tested on prediction and incomplete knowledge problems. There is some still a lot of unexplored possibilities for machine learning within the RoboCode world.

`Build the best - destroy the rest!`

Discussion and Further Work

Several discussion points arise in retrospect to the design. The different targeting strategies are actually all variations of the same targeting strategy. Head-On targeting can be transformed into GuessFactor targeting using a guess factor of 0. Circular targeting is almost the same as using a guess factor of 1. Learning what targeting to use is actually learning what guess factor works best. Since its value is normalised between -1 and 1, I believe that an artificial neural network might be able in optimising the guess factor then high level the implemented one-step Q-Learning.

By using high-level strategies, `Qbot` seemed severely crippled when facing more advanced robots. The level on which the actions are chosen might need to be lowered. By allowing the controller to move to a certain location or shoot at a certain area, it might be possible to allow for more adaptive power.

Using an off-policy temporal-difference update equation, as used by one-step Q-Learning, might not be the best way to let the robot controller update its estimation of the value function. By keeping the temperature constant, instead of letting it

decrease over time, the robot controller might be learning sub-optimal behaviour. By gradually dropping the temperature or by implementing an on-policy update equation the learning might be improved. Also, the current used update equation is very simple, there exist far more complicated and presumed better ways to estimate the value functions.

Part III

Appendices

Appendix A

The Rules Of The Game

Here follows a short introduction to the rules and physics that govern Robocode. Most of this information has been gathered from the online robowiki[1], from the documentation on the application programming interface (A.P.I.)[2] or from an online article by Li (2002).

Game Rules

The robot that wins a battle is the one that has the highest score at the end of a certain number of rounds. (default set at 10) Each robot starts at a random location facing a random direction with a set number of Energy points, namely 100. At expense of a small bit of Energy it can fire a bullet. At hitting another robot the said bullet restores Energy. A successful hit by a bullet restores more then the cost of firing it. Robots can move around, but hitting the wall or other robots will damage robot. Scoring is done at the end of each round and is a combination of the following factors:

Survival Score - Each robot that is still alive scores 50 points every time another robot dies.

Last Survivor Bonus - The last robot alive scores 10 additional points for each robot that died before it.

Bullet Damage - Robots score 1 point for each point of damage they do to enemies.

Bullet Damage Bonus - When a robot kills an enemy, it scores an additional 20% of all the damage it did to that enemy.

Ram Damage - Robots score 2 points for each point of damage they cause by ramming enemies.

[1] `http://robowiki.net/`
[2] `http://robocode.sourceforge.net/docs/robocode/`

Ram Damage Bonus - When a robot kills an enemy by ramming, it scores an additional 30% of all the damage it did to that enemy.

Robot and Battlefield Anatomy

The battlefield whereon the robots battle is free of any obstacles less the enemies. It can best be described as a two-dimensional bounded plane. The size can be customised with the minimum at 400 pixels and the maximum limit at 5000 pixels. The default settings are 800 x 600 pixels. Location is determined by Cartesian co-ordinates with the origin being $(0, 0)$ at the lower left corner of the battlefield. The co-ordinates can be fractional, i.e. a robot's location can be between two whole pixels co-ordinates.

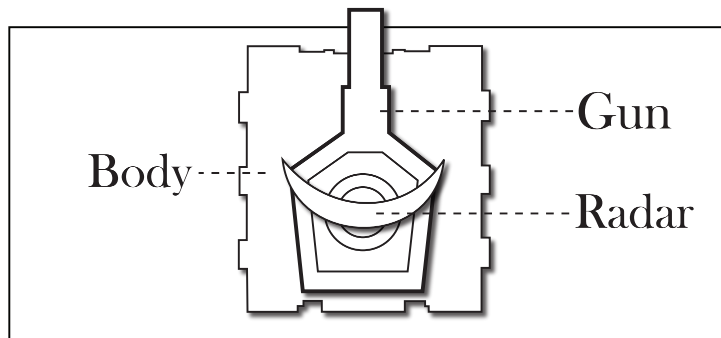

Figure A.1: The anatomy of a robot

Game Physics

The robots itself consists of three parts (as shown in Figure A.1 on 47) each resembles a set of actions:

The radar can detect other robots and can be set to turn around. It is placed on top of the gun.

The gun can fire bullets (with a variety of firepower) at the direction it's currently pointing and can be set to change it's heading.

The body is whereupon the gun is mounted, it can be set to turn but also to accelerate forward or backwards. It enables movement over the battlefield.

Every part can be set to change its heading, but the speed is dependent on the part it is mounted on, e.g. if only the body is set to turn clockwise, the gun and radar

also turn clockwise. This can be disabled though. Though this makes the robot automatically rotate the gun and radar to compensate for body movement. This might not be desirable as it takes away some control of the robot.

Units of time are represented as “ticks” in Robocode. Each of the robots receives one turn every tick the round progresses. In this turn the robots determine which actions to perform, these are then simulated by the battle-simulation.

Robot Movement

Acceleration(a) Robots accelerate at the rate of 1 $\frac{pixel/turn}{turn}$ and decelerate at 2 $\frac{pixel/turn}{turn}$. It should be noted that the Robocode battlesimulation determines what exact acceleration the robots has. The controller influence this directly.

Velocity(v) $v = at$. ($t = turns$) Velocity has a maximum of 8 $pixels/turn$. But it can be negative, which means the body is moving backwards. ([−8.8])

When driving the heading of the robot is the direction in which it is moving and is best described as a full circle ranging from 0° to 360°. (As depict in Figure A.2)

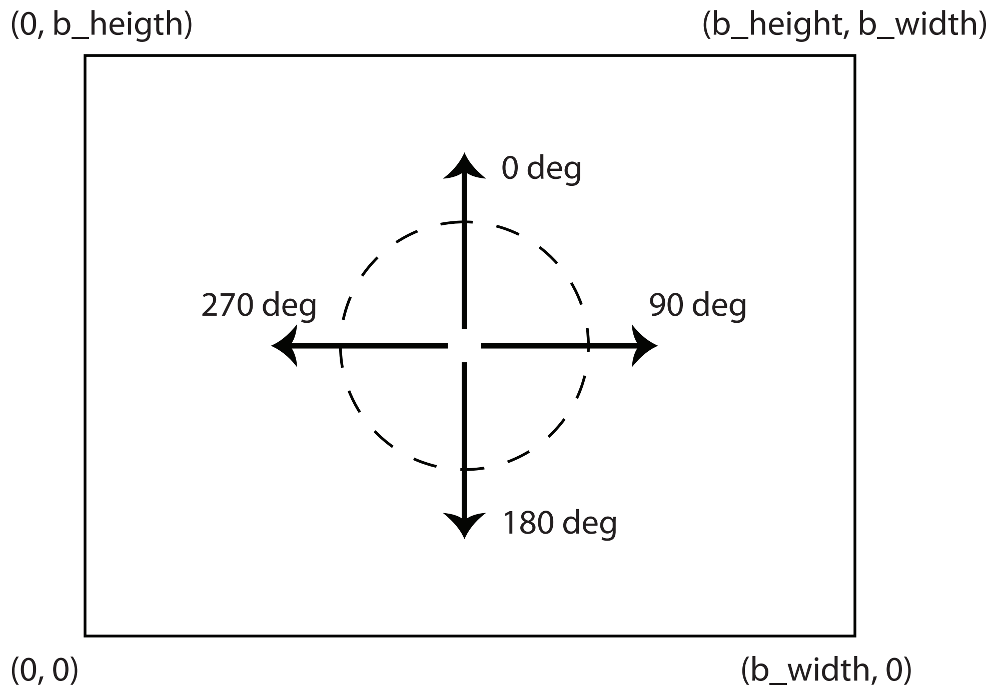

Figure A.2: Headings on the battlefield

Robot Rotation

The robot turn 360° in a single tick. Instead there is a maximum on how many degrees a part can move every turn.

The Radar 45 $degree/turn$

The Gun 20 $degree/turn$

The Body 's maximum turn rate is dependent on its velocity:

$$(10 - 0.75 * |velocity|)\ degree/turn \tag{A.1}$$

Robot bullets

Bullets can be fired with dynamic firepower, whereby $0.1 < firepower < 3.0$. The firepower has a direct influence on the speed and damage of the bullet. Setting the $firepower$ correctly determines wether a bullet is a successful hit or a failure miss. The resulting Energy after firing a bullet is; $Energy \leftarrow (Energy - firepower)$

Damage the target receives is:

$$4 * firepower + arg_{max}(0, 2 * (firepower - 1)) \tag{A.2}$$

Velocity the bullet has:

$$20 - 3 * firepower \tag{A.3}$$

Gun Heat prevents you from firing, drops 0.1 each tick by default, but can be set to any value. It is generated as:

$$\frac{1 + firepower}{5} \tag{A.4}$$

Energy returned to the robot which fired the bullet is:

$$3 * firepower \tag{A.5}$$

Robot collisions

When a robot runs into a wall or another robot, it receives damage.

With a robot - each robot takes 0.6 damage

With a wall - the damage is $arg_{max}(0, (|velocity| * 0.5 - 1))$

Side Notes

Heading And Bearing

If the robot faces the exact north of the battlefield, it has a heading of $0°$. In mathematics the convention stands that a heading of $0°$ corresponds to the east of a union circle. This means that using normal trigonometry functions, e.g. $\sin(x)$, $\cos(x)$ and $\tan(x)$, do not yield the expected result.

If one wants to convert from conventional degrees n into Robocode degrees r, the following equation can be used.

$$r = \begin{cases} 90 - n & \text{if } n < 90 \\ 450 - n & \text{if } n \geq 90 \end{cases}$$

Body Size

Even though the visual simulation might suggest otherwise, the body is for all purposes processed as a 36x36 pixel square. The square does neither rotate nor tilt. Not even when the robot turns around. This also means that the effective size of the battlefield is 36 pixels shorter on all sides, e.g. a 800x800 sized battlefield has an effective size of 764x764.

Appendix B

Event List

The RoboCode battle simulator provides a set of events and methods to supply the robots with information to base their actions on. In this appendix an extensive list is presented.

Events

Table B.1: The events, including the information the event provides

Name	**Event**	**Information**
BulletHitEvent	Your bullet impacts on robot	Target's name and remaining energy
BulletHitBulletEvent	If two bullets collide	Bullet^table B.2 objects
BulletMissedEvent	A Bullet impacts a wall	Bullet^table B.2 object
DeathEvent	Your robot dies	—
HitByBulletEvent	A bullet impacts you	Bullet^table B.2 object
HitRobotEvent	A robot collision	Name and energy or robot, fault
HitWallEvent	A collision with the wall	Bearing to wall
RobotDeathEvent	An enemy robot dies	Name
ScannedRobotEvent	A radar detects enemy	Distance, energy, velocity, heading, bearing
StatusEvent	start of every turn	RobotStatus^table B.2 object all relative information of the robot itself
WinEvent	the robot wins	—

Non-Event Sources

Table B.2: Sources of input that are not events

Object	**Information**
Bullet object	velocity, firepower, name of robot, heading, co-ordinates bullets
RobotStatus object	co-ordinates, energy, velocity, heading et cetera

Appendix C

Advanced Strategies

GuessFactor Targeting

Thought up by Paul Evans and first implemented in the robot `SandboxLump` this targeting strategy is now the most commonly used by nearly all top ranking robots on the RobotRumble one versus one main ranking list (on 06-21-2009).

It works as follows: when a target is scanned it has a maximum distance it can travel before a bullet could possibly reach it. This can then be reformulated to an angle from the firing robot to the target. This is called the maximum escape angle (MEA) and defined by the robowiki as:

> "Maximum Excape Angle (MEA) is the largest angle offset from zero (i.e., Head-On Targeting) that could possibly hit an enemy bot, given the game physics of RoboCode."

Once the MEA is calculated the robot can produce a guess factor (GF) that us unique for an enemy robot. That is, a normalisation over this angle. With a GF of 1.0 being the largest angle the target can reach if it drives at maximum velocity and a GF of -1.0 if it suddenly reverses its direction. (See figure C.1 on 54) The key is finding what GF is best fired upon to give the highest chance of hitting the enemy. As Matthew Reeder (known as the user 'kawagi') mentions in his short tutorial on GuessFactor Targeting[1]:

> "The philosophy and assumption we make for GuessFactor Targeting is that our enemy is moving randomly in some way or another [...] The direction we need to shoot is the sum of several random decisions made by our enemy. One nice thing about sums of random values is that they tend to show statistical trends. The trick to GuessFactor Targeting is to find out which direction we should shoot each time we fire, and in the future, we fire in the direction that was correct the most often."

[1] See: http://robowiki.net/w/index.php?title=GuessFactor_Targeting_Tutorial

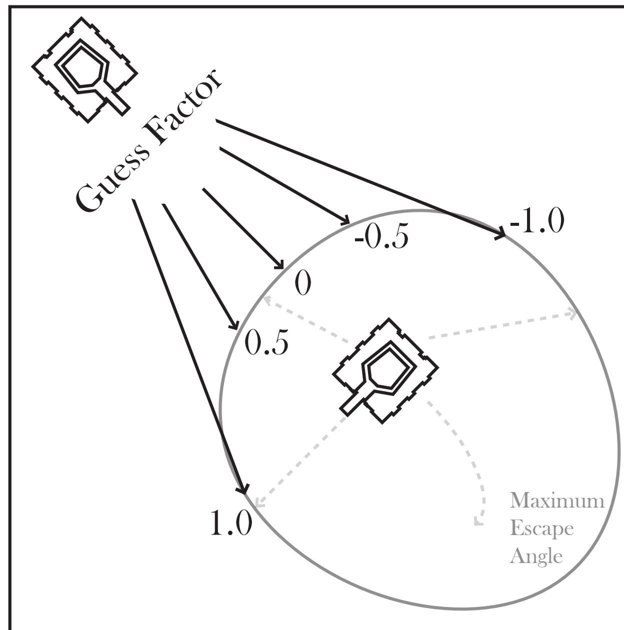

Figure C.1: The GuessFactor

"The trick..." is usually by means of statistical approach or feedback via virtual bullets (ghost bullets fired by your own robot) and keeping track of the results. The difference between most high ranking robots is in the way they estimate the GF, and how they use segmentation on the GF range to be influenced by statistical data.

In the design of the robot a simple form of GuessFactor Targeting will be implemented, using the online tutorial mentioned earlier. It will use a virtual bullet, in the form of a computational cheap wave to find the bearing it should have fired at. It will very much be alike to `FloodMini` version 1.4 by Kawigi.

Wave Surfing Movement

One property enemy bullets have in RoboCode is that they are invisible for all robot. This makes dodging them difficult to say the least. The Wave Surfing movement strategy is a strategy that copes with the fact that the robot cannot detect these bullets. By initiating a wave if an enemy fires a bullet it predict when the bullet will close and posses a threat. The robot cannot directly detect the firing of a bullet, but by monitoring the energy drops of the enemy it can estimated correctly when a bullet is being fired.

A wave consists of a source location (of the firing robot), a velocity (generally based on firing robot's bullet power), the time the wave was created, and the bearing to the target at fire time. You can imagine the wave as a circle that radiates out from

the point from which it was fired. Every turn the waves are checked to see if they have passed some boundary (usually a set distance from the robot) and then reacted upon. (See figure C.2 on 55) Movement strategies based on waves are currently used

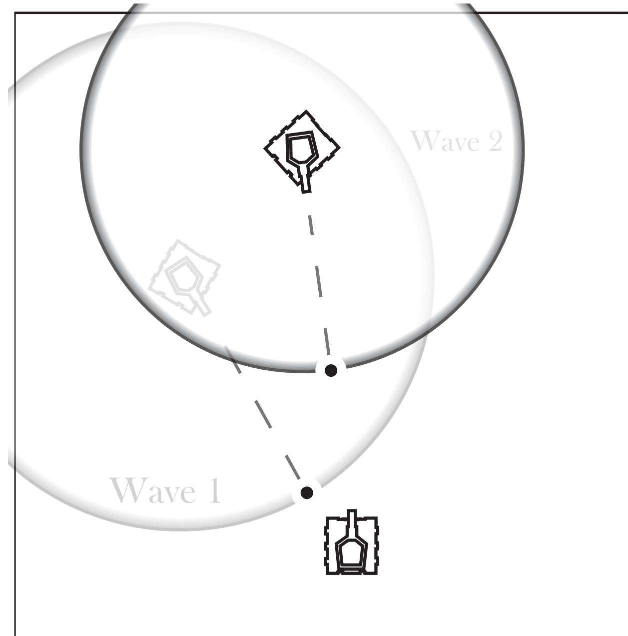

Figure C.2: An example of waves

by the number 1st, 2nd and 3rd ranking robots on the RobotRumble one versus one main ranking list (on 06-21-2009). The implementations differ mainly in what they do with the wave information. *True surfer style* robots re-evaluates forward and backward movement while *GoTo style* robots try to find out what part of the battlefield is the least dangerous and moves towards that point.

The designed implementation will be based on the *true surfer style* robots. Especially `RaikuMX` version 0.38 by Jamougha.

Appendix D

Test Results

In this appendix some more of the test graphs, that are part of the testing phase, are shown. Due to the number and their cluttering power, they are placed here. The first graphs are of some extra testing against some more advanced robots. `sample.Walls` seems a very simple robot at start, but can be quite a pain to defeat using simple strategies. Qbot easily beats it. `abc.Shadow v3.83` was a long standing #1 in the RoboRumble General 1v1. Qbot shows some improvement over time, but will probably never be able to defeat Shadow due to the limitations of the strategies.

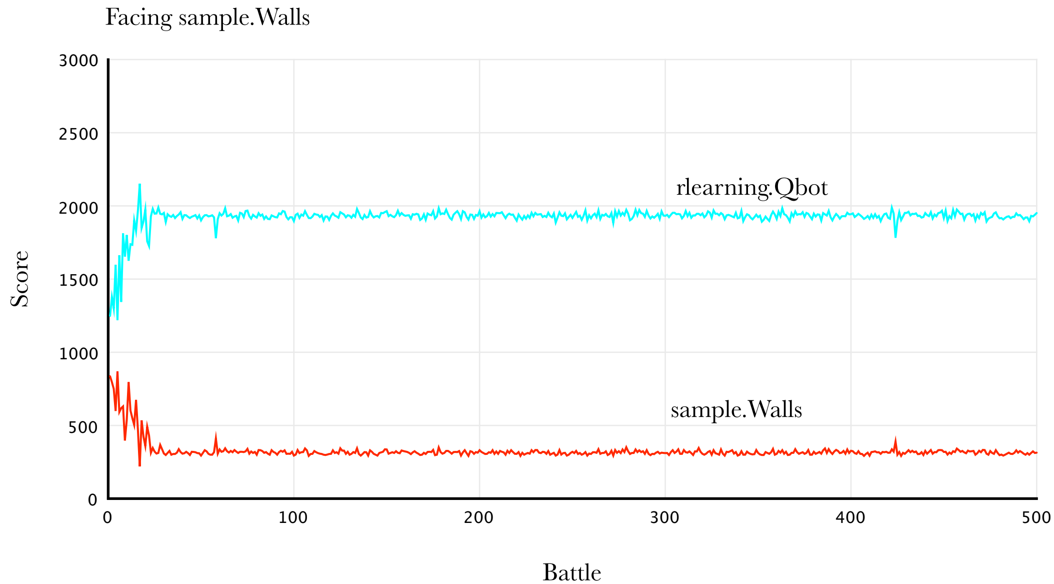

Figure D.1: Result of setting Qbot up against some more advanced robots. Using α = 0.1 and τ = 20

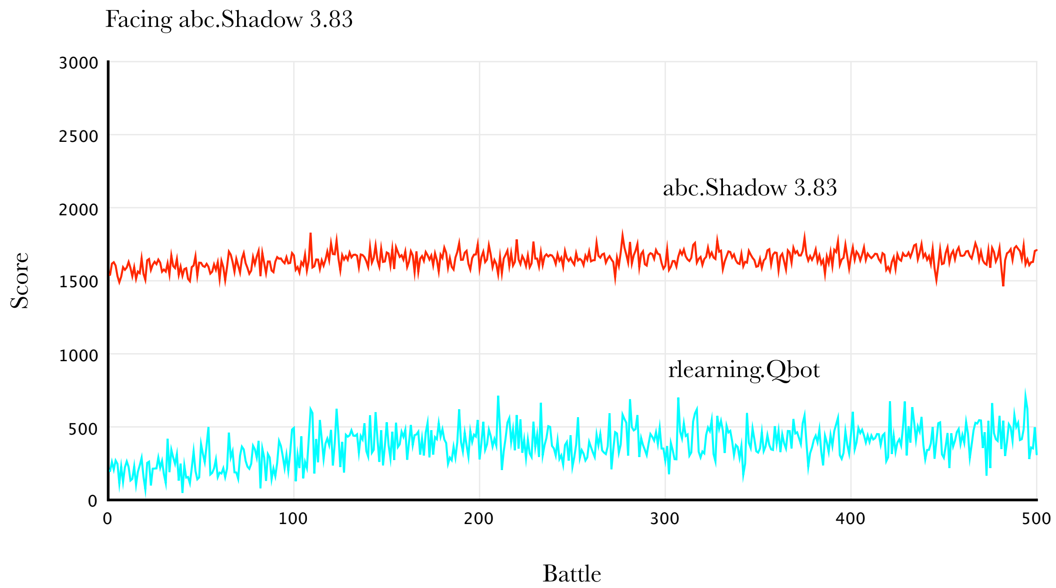

Figure D.2: Result of setting Qbot up against some more advanced robots. Using $\alpha = 0.1$ and $\tau = 20$

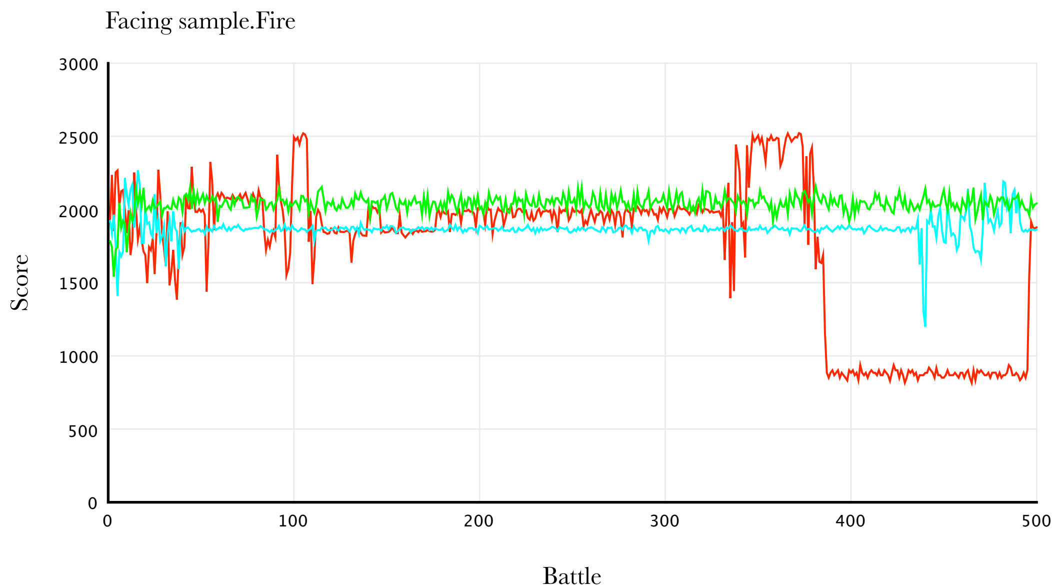

Figure D.3: Trials of Qbot versus sample.Fire. Using $\alpha = 1$ and $\tau = 40$

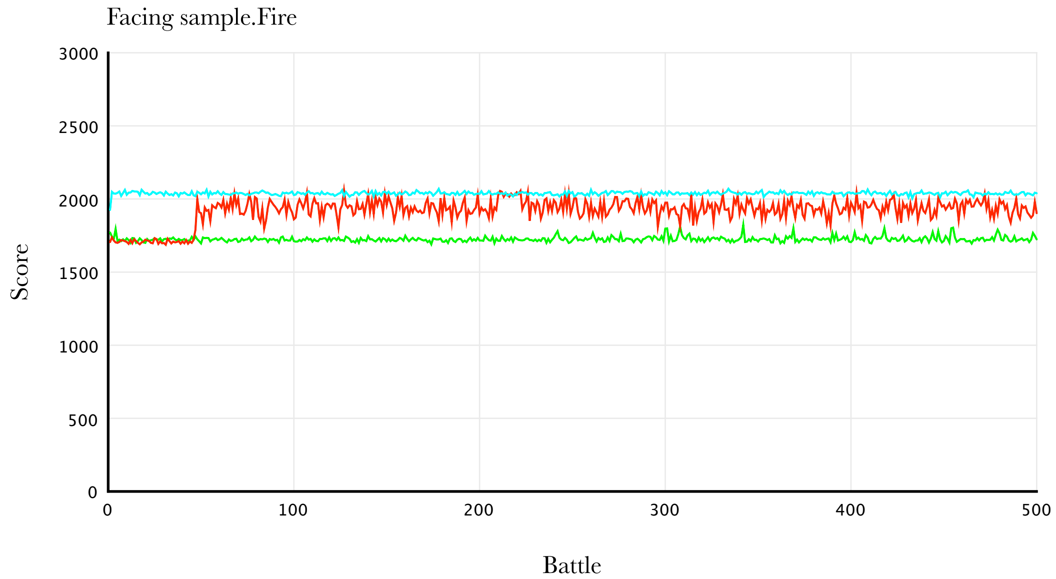

Figure D.4: Trials of Qbot versus sample.SittingDuck. Using $\alpha = 1$ and $\tau = 40$

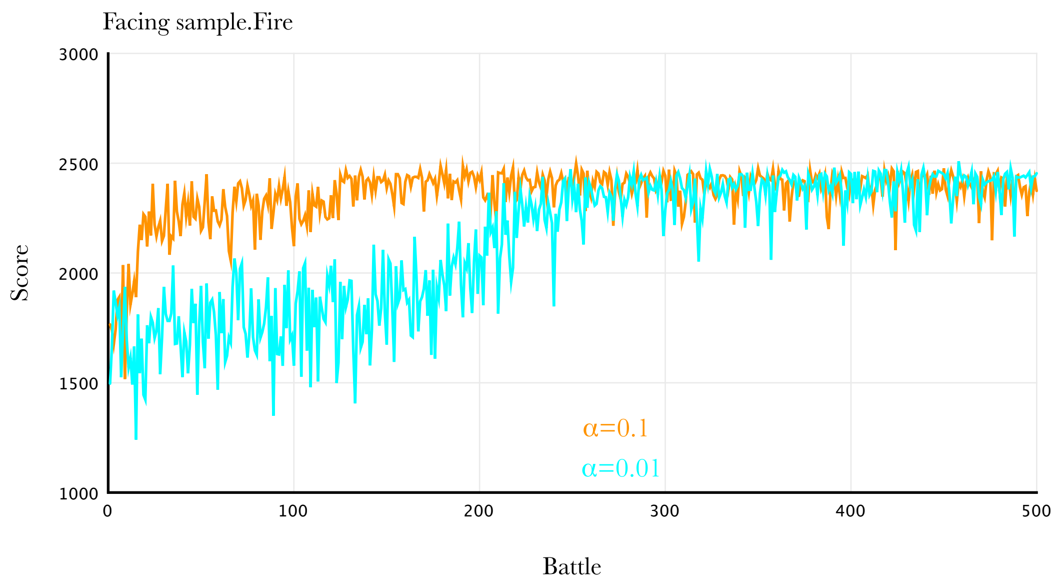

Figure D.5: Trials of Qbot versus sample.Fire, in order to see if a step-size of 0.01 makes any difference. Using $\tau = 40$.

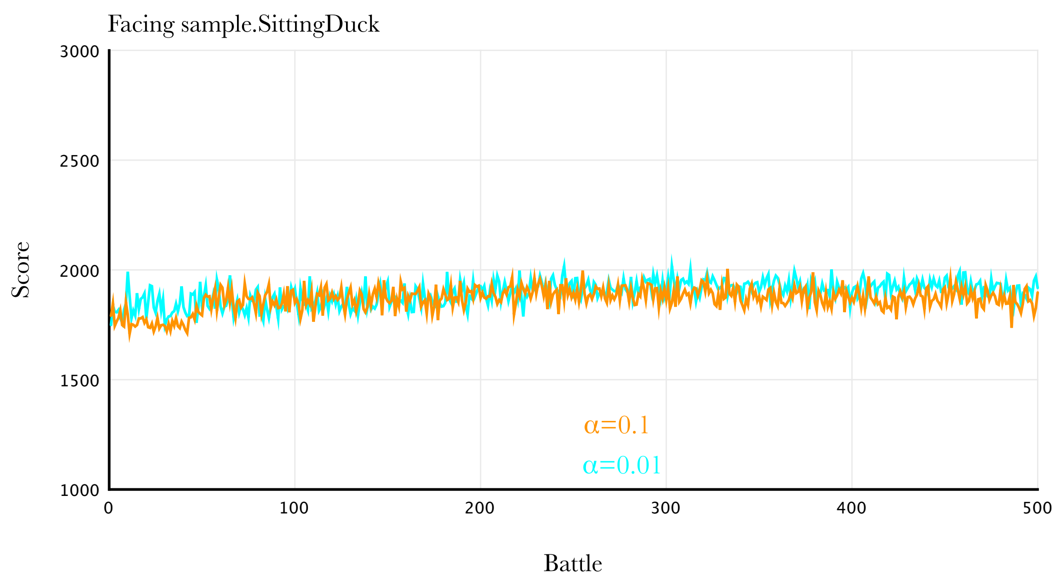

Figure D.6: Trials of Qbot versus sample.SittingDuck, in order to see if a step-size of 0.01 made any difference. Using $\tau = 40$

Bibliography

J. Eisenstein. Evolving robocode tank fighters. *CSAIL Technical Reports, Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory*, 2003.

J. Frøjhær, M. L. Kristiansen, P. B. Hansen, I. V. S. Larsen, D. Malthesen, T. Oddershede, and R. Suurland. Robocode, development of a robocode team. Technical report, Department of Computer Science, Aalborg University, 2004.

M. Gade, M. Knudsen, R. A. Kjær, T. Christensen, C. P. Larsen, M. D. Pedersen, and J. K. S. Andersen. Appying machine learning to robocode. Technical report, Department of Computer Science, Aalborg University, 2003.

L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4, 1996.

S. Li. Rock 'em sock 'em robocode! learning java programming. *IBM developerWorks*, 2002. URL http://www.ibm.com/developerworks/java/library/j-robocode/.

S. J. Russell and P. Norvig. *Artificial Intelligence - A Modern Approach*. Prentice Hall Series in Artificial Intelligence. Pearson Education, 2003.

Y. Shichel, E. Ziserman, and M. Sipper. Gp-robocode: Using genetic programming to evolve robocode players. *Department of Computer Science, Ben Gurion University, Israel*, 2005.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning. The MIT Press, 1999.

C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, 1989.