DAT3 REPORT

Applying Machine Learning to Robocode

Morten Gade
Michael Knudsen
Rasmus Aslak Kjær
Thomas Christensen
Christian Planck Larsen
Michael David Pedersen
Jens Kristian Søgaard Andersen

Department of Computer Science
Aalborg University

16th of December 2003

The Faculty of Engineering and Science
Aalborg University

Department of Computer Science

Title:

Applying Machine Learning to Robocode

Project Period:
DAT3,
3rd of September -
16th of December 2003

Project Group:
E2-216

Group Members:
Morten Gade
Michael Knudsen
Rasmus Aslak Kjær
Thomas Christensen
Christian Planck Larsen
Michael David Pedersen
Jens Kristian Søgaard Andersen

Supervisor:
Søren Holbech Nielsen

Number of copies: 9

Report – pages: 139

Appendix – pages: 18

Total number of pages: 157

Abstract

This report documents the development of the Robocode robot, Aalbot, which uses the machine learning methods reinforcement learning, genetic programming and neural networks. An analysis reveals the nature of the Robocode game, the previous work done with machine learning in Robocode and agent architectures suitable for Robocode robots.

A modular hybrid agent architecture is designed specifically for the characteristics of Robocode. A discussion of the use of machine learning and problem solving in a general context leads to an argued choice of the machine learning methods used in Aalbot. Detailed designs of each of the modules incorporating the machine learning methods are then presented and evaluated.

Aalbot is finally tested and found to be competitive against traditional robots which do not employ machine learning.

Preface

The authors of this DAT3 project are seven 5th semester students at the Department of Computer Science, Aalborg University, Denmark.

Source code for the Robocode robot developed in this project, Aalbot, can be found on the accompanying CD-ROM. Refer to the README file in the root directory of this CD for instructions on how to experience Aalbot in live action.

Throughout the report general concepts and methods in a source are cited by referring entire chapters, while self contained equations and definitions are referred to by page number. Implementation level terms such as class names and robot names will be typeset in the `Typewriter` font.

Morten Gade	*Michael David Pedersen*
Rasmus Aslak Kjær	*Christian Planck Larsen*
Michael Knudsen	*Jens Kristian Søgaard Andersen*
Thomas Christensen	

Contents

Introduction

Let the battle begin! Yet another intensive fight is about to take place as our robotic battle tank is dumped onto the battlefield together with a number of opponents. Common to them all is their definitive goal: Outmaneuver all opponents and ruthlessly put an end to their existence.

Our robot immediately scans its surroundings and spots an enemy robot heading straight toward it. The enemy fires its gun, but our robot manages to dodge the bullet just in time. To avoid the enemy our robot engages in a circular movement, which brings it behind the enemy. Now it is pay-back time. While repeatedly firing the robot moves with maximum speed in the direction of the enemy and finally rams him from behind causing his annihilation. . .

The above was a glimpse into the world of Robocode – a so called *programming game*, where programmers use their skills in Java to write a program defining the behaviour of a robotic battle tank. Once a programmer feels that his robot is fit for fight, he can submit it to various competitions where it will battle against other programmed robots.

The preliminary goal of this project is to:

> Design and implement a robot, Aalbot, using machine learning that will score well in combat against other robots. The robot will differ from the majority of other robots in that it should be able to *learn* from experience, allowing it to gradually improve its skills over time. Moreover, the robot should posses adaptability such that it is capable of acting and reacting according to the properties of its environment.

To create such a robot, a number techniques from the area of machine learning (ML) will have to be applied. The focus of this project will thus be on investigating, applying and evaluating different ML techniques suitable for the control of a robotic battle tank within the framework of Robocode.

The report is separated into three parts, namely analysis, design and finally evaluation. To provide an overview of the report, the three parts are briefly introduced below.

1. **Analysis.** The first part of the analysis will introduce and explain the details of the Robocode framework. Specific strategies that a robot can

adopt in a battle to improve its chances of winning are analysed. Following this is a brief study of some existing robots, which have applied ML techniques with success. This will provide valuable information that will help direct the design of our robot. In the last part of the analysis a robot is viewed in a more general setting as an autonomous agent, and three general agent architectures are considered. The preliminary goal will guide the analysis and lead to an elaboration of the success criteria for this project.

2. **Design.** An argued selection of suitable machine learning methods is made based on the knowledge gained in the analysis. The selected methods are considered in relation to Robocode with emphasis on how the battle strategies recommended from the analysis and the chosen agent architecture can be combined with or supported by these methods. The design part results in a complete specification of Aalbot.

3. **Evaluation.** How and to which degree the design has been realized is documented in this last part of the report. The created robot will be tested in various settings to see if it manages to fulfill the success criteria defined in the analysis. The results of these tests are evaluated and discussed.

Part I

Analysis

Chapter 1

Robocode in Essence

Robocode is a framework by IBM AlphaWorks for creating and running programmable, competing robots in Java. Since the initial release in 2001, it has evolved a great deal and today a big community exists in which robot design ideas and source codes are shared. Tournaments and ongoing leagues are arranged and web sites with rankings of robots are continuously updated. In this chapter the gameplay and rules of the Robocode game are described. Furthermore different game strategies will be introduced and analysed.

This chapter is primarily based on [AN01a] and [AN01b].

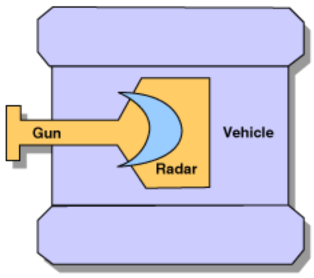


Figure 1.1: An illustration of a Robocode robot.[Li02]

1.1 The Gameplay

In Robocode a number of robots battle to the death in an arena. A battle consists of a number of rounds, the winner of the battle is the robot with the most points. A round ends when all robots but one are dead – the remaining robot is the survivor of that round. The scoring scheme of Robocode is introduced later. The robots start with an energy level of 100 and die when the energy level falls below zero. Robots fight each other by shooting with a gun mounted on a turret. The robot is equipped with an radar, which can scan for other robots up to a distance of 1200 pixels away. The radar returns information

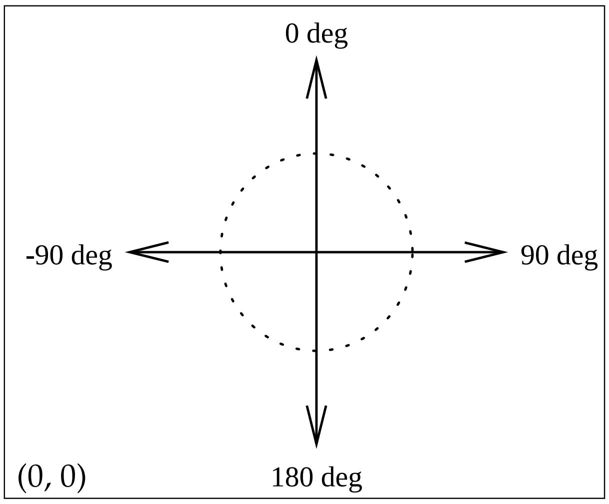


Figure 1.2: The coordinate system used in Robocode. Note the unusual orientation of the axes.

about the scanned robot namely distance, heading, velocity, name and energy. The radar, gun turret and robot body can rotate 360 degrees independent of each other, though the gun turret can be locked to follow the vehicle rotations, and the radar can be locked to follow the gun turret and vehicle rotations. The gun can fire with variable power and it heats up when it fires. A robot has to wait for the gun to cool down before another shot can be fired, and the bullet travels faster when fired with low power. If two robots collide, they both lose 0.6 energy points. If a robot is heading toward an opponent upon colliding with it, it is by Robocode considered *ramming* and rewarded with 1.2 score points. All battles take place in a rectangular, wall-surrounded arena of variable size. It has a default size of 800×600 pixels – robots have a size of 36×45 pixels. The robot, shown in Figure 1.1, can move around in the arena with constant acceleration and variable velocity (forward and backwards) whilst turning, shooting and operating the radar simultaneously. A positive velocity is a forward motion and a negative velocity is backwards, speed is simply the absolute value of the velocity. The maximum speed of a robot is 8 pixels/turn.

Robocode has a physics model which limits the possible actions of the robots, e.g. the robot cannot move at full speed while turning. Firing the gun causes a robot to lose the same amount of energy as the shooting power. The lost energy is regained by factor of 3 if the bullet hits a target – see Table 1.1 for some examples. Robocode is turn based, and in each turn every robot is given a limited amount of time to choose which actions to perform. When all robots have completed their turn, their actions are performed sequentially. If a robot exceeds its time limit for a turn, it misses the entire turn. If it exceeds its time limit 30 times in a round, it is disqualified from the round by Robocode. This means that a robot's skills have to be a compromise between computational complexity and time needed to react sensibly to events. A robot which

Firing power	**Damage inflicted**	**Energy regained**	**Bullet velocity (pixels/turn)**
0.1	0.4	0.3	19.7
1.0	4.0	3.0	17
2.0	10.0	6.0	14
3.0	16.0	9.0	11

Table 1.1: Possible damage, energy gained and bullet velocity given some example firing powers.[AN01a]

is overly complex will thus probably lose, because it will spend all its time deciding reactions and possibly get shot (and killed) in the mean time, and a really simple robot will act without much consideration, resulting in suboptimal actions.

Robocode uses a scoring system to determine the winner of a match. The winner is not necessarily the robot surviving most rounds because Robocode also gives points for e.g. attacking. The rules are as follows[AN01a]:

- Each live robot is given 50 points each time a robot dies.
- The last robot alive is given an additional reward of 10 points for each dead robot.
- Each point of damage inflicted is rewarded by 1 point.
- Killing an enemy is rewarded additional 20% of total damage inflicted on that robot.
- Ramming an opponent is rewarded with 2 points per damage inflicted by the ramming.
- Killing an opponent by ramming it is rewarded additional 30% of the total damage inflicted on that robot.

The final winner is the robot with most points in total after all rounds have been fought.

The processing loop used by the Robocode engine to execute each round from [AN01a] is given below:

- All robots execute their code until taking action
- Time is updated
- All bullets move and check for collisions
- All robots move (heading, acceleration, velocity, distance, in that order)
- All robots perform scans (and collect team messages)

- The battlefield draws

Robocode is based on an event-driven system, where every robot has an event queue of its own. Events occur e.g. when the robot is hit by a bullet or it scans another robot. Robots are created by extending the `Robot` or `AdvancedRobot` class. Robots based on the `AdvancedRobot` Java class are asynchronous, which means that it is possible to perform actions simultaneously by using non-blocking methods. `AdvancedRobot` receives the following events:

- **BulletHitEvent**. Received when one of *the robot's* bullets hits another robot. It gives information about the opponent's name and remaining energy along with a Bullet object, which holds the bullet's heading, owner[1], power, velocity, victim[2] and a status flag which tells if the bullet is still active.

- **BulletHitBulletEvent**. Received when one of *the robot's* bullets hits another bullet and is destroyed. It can return the two bullet objects.

- **BulletMissedEvent**. Received when one of *the robot's* bullets hits a wall and gives the bullet object.

- **HitByBulletEvent**. Received upon being hit by a bullet and gives the name of the shooter along with the bullet's heading, power and velocity.

- **HitRobotEvent**. Received upon colliding with another robot and gives the other robot's name, energy, bearing and whether the robot was heading toward the other robot.

- **HitWallEvent**. Received when *the robot* collides with a wall. It gives the robot's angle to the wall.

- **ScannedRobotEvent**. Received when the radar scans a robot. It gives the other robot's name, energy, bearing, distance, heading and velocity.

- **WinEvent**. Received when *the robot* wins a round and holds no information.

- **DeathEvent**. Received when *the robot* dies and holds no information.

- **SkippedTurnEvent**. Received when *the robot* has not decided on an action within the given time limit of the turn. It holds no information.

- **RobotDeathEvent.** Received when another robot dies. It holds the name of the killed robot.

[1]The robot that fired the bullet.
[2]The robot that was hit by the bullet, if any.

1.2 Robocode Strategies

In battles, strategies are guidelines of how to act in a given situation. Obviously, a set of strategies would be useful in Robocode, but first a definition of a strategy is needed:

Definition: A systematic plan of action to reach predefined goals.[FS97, Chapter 9]

This very general definition allows the use of partial strategies which describe only a subset of the possible actions to take. Strategies will be described divided into four categories:

- **Percepting.** How the robot controls its radar.
- **Planning**. Deciding long-term actions and goals.
- **Moving.** Controlling the motion of the robot.
- **Targeting and Shooting.** When, how powerful and in which direction to shoot.

A robot might do well in 1-on-1 battles, but for melee[3] battles other strategies are needed. One can imagine that a good strategy in melee combat is to try to stay out of the way and attract no attention. Then, after the other robots have killed each other, one could easier kill the last robot because it probably has sustained a lot of damage. This strategy would not work equally well in 1-on-1 battles, because the opponent has full energy. Another argument against this strategy is the scoring system used in Robocode. A battle is not won solely based on being the last man standing. The amount of damage inflicted and opponents killed also count.

In the following, various strategies for controlling a robot are presented along with any apparent pros and cons. This is based on [AN01b], [McC02], [Mar02], [Owe02] and [Robb].

1.2.1 Percepting

The *basic* way to scan for opponents is to simply rotate the radar 360 degrees continuously and remember the positions of all other opponents. However, since robots are rarely spread evenly over the arena due to fighting, this is obviously not an optimal strategy. Time is wasted scanning empty parts of the arena.

[3]Melee means a battle with more than two robots in the gaming area.

Another scanning strategy, *alternating direction scanning strategy,* is one in which the robot scans in one direction, and once all other robots are found, the scanning direction is reversed. The idea is that robots tend to fight in small areas and this can be used to reduce the time spent on searching for opponents.

Especially in 1-on-1 battles a *locked scanning strategy* where the radar remains fixed on a single target is very efficient. This can be done by moving the radar a little back and forth every turn – perhaps selecting scanning direction based on the desired target's heading – in order to not lose track of the target. This would probably not be a good strategy for melee battles because it can often be a good idea to change target for a while, e.g. if an opponent low on energy is nearby.

Since there are no events which represent that another robot has fired a bullet, the radar can be used to guess about this. If an opponent has lost a certain amount of energy between two scans, it *could* mean that the opponent has fired a shot. The shot power can also be estimated based on the amount of energy lost. Of course, shooting is not the only reason for losing energy, so this is only a way to make an *educated guess*.

The Robocode `AdvancedRobot` has the following methods for controlling percepting actuators:

- **scan**. Scans the arena using the radar at its current setting. This is only rarely needed to be called explicitly, as Robocode calls this for each robot at each turn.
- **setTurnRadarLeft** and **turnRadarLeft**. Turns the radar left.
- **setTurnRadarRight** and **turnRadarRight**. Turns the radar right.
- **setAdjustRadarForRobotTurn**. Enable or disable the locking of the radar to the base of the robot.
- **setAdjustRadarForGunTurn**. Enable or disable the locking of the radar to the gun turret.

In Robocode the **set*** methods are the non-blocking methods which allow several actions to be performed simultaneously. These methods simply add the actions to a queue, the actions in the queue are executed by using the **execute** method.

1.2.2 Planning

Since the primary goal in Robocode is to win by attacking and killing opponents, a robot must decide which opponent to attack, if it wants to attack at all. This is most relevant in melee battles, since in 1-on-1 battles this problem is reduced to determining whether to attack or avoid the opponent.

A basic approach is simply to attack the target with the lowest energy or try to hide if all opponents have higher energy. However, this target may be in the other end of the arena, thus it may prove to be a poor strategy, because the robot may have to move quite far in order to be sure of hitting the target. While moving, another opponent may kill the currently selected target. Also, another robot may become even weaker, causing the robot to select a new target and begin moving toward it. This may repeat until the robot is killed and it might never fire a single bullet.

Another approach would be to attack the *closest* target. However, the closest target may indeed be the *best performing* robot in the battle, which might not be the best target to select.

A third approach would be choosing a compromise between target distance and target energy. However, finding the right ratio is non-trivial and depends much on the situation in which the decision must be made.

Robocode provides no methods to aid in planning in any of the robot classes, contrary to scanning.

1.2.3 Moving

Obviously, staying in the same place is a really bad idea because a sitting duck is easy to hit.

Moving around the arena in *straight lines* is also a poor strategy because a robot will have few problems predicting an opponent's route, thus it will be easy to target future positions. Robots moving this way are likely to take many hits from its opponents.

Circular movement is better than moving in straight lines, but not much. It is not particularly hard to compute the positions on a circle given a velocity, so this strategy is not likely to work much better against advanced robots either.

Random movement can be used to confuse robots that try to predict the moving patterns of other robots. It is not beneficial to achieve complete random movement, because this makes it hard to follow other robots and the risk of ramming the walls becomes higher.

Anti-gravity is a general method which exists in many variants. It is implemented in many robots[Robb] and seems to be a simple and elegant solution. It is used to push the robot away from dangerous areas and attract it to safe or otherwise advantageous areas. This is very versatile because it can be used in every situation where movement is required, and it is possible to decide areas to avoid dynamically during battles through algorithms.

Bullet dodging[Mar02] is a useful ability for a robot to possess in order to be successful. Apart from not being directly able to detect the firing of bullets, the radar cannot determine which way an opponent's gun turret is pointing. Thus, the path of the guessed bullet cannot be determined. However, since the firing power influences the loss of energy, the speed of a bullet can be determined. The idea is to draw a circle symbolising a wave front with centre at the firing opponent's position at the time of the firing, and then increase the radius with the speed of the bullet. This way, the robot can ignore the guessed bullet until the bullet *can* be close to hitting it and at this point in time make a sudden, random move.

It is quite easy to prevent a robot from hitting walls, but this can have big impact on its abilities to dodge bullets etc. *Factored wall avoidance*[McC02] is an algorithm designed to keep robots from hitting the arena walls without getting trapped in a corner or moving away from the desired direction or area.

In 1-on-1 battles, ramming could be a good supplement to shooting for a robot that has more energy than its opponent, because the opponent will be killed first. In melee battles this is a less efficient strategy, because the robot is not guaranteed to remain in the lead. In both 1-on-1 and melee battles using ramming can cause the attacking robot to be killed.

The Robocode `AdvancedRobot` has the following methods for controlling movement actuators:

- **setAhead** and **ahead**. Move a given distance ahead.
- **setBack** and **back**. Move a given distance back.
- **setMaxTurnRate**. Limit the rate at which the robot turns.
- **setMaxVelocity**. Limit the speed at which the robot moves.
- **setStop** and **stop**. Stop any movement and remember what the robot was doing.
- **setResume** and **resume**. Resume the movement stopped with **setStop** or **stop**.
- **setTurnLeft** and **turnLeft**. Turn the robot left.
- **setTurnRight** and **turnRight**. Turn the robot right.

1.2.4 Targeting and Shooting

The task of targeting is to calculate the angle to rotate the gun turret so that it points in the correct direction to hit an enemy robot. The major complication is that it is not sufficient to merely point the gun turret in the immediate direction of that enemy. The bullet must travel a distance to reach the target. During the flight time of the bullet, the enemy could have moved away from its original position. A further complication is the fact that bullets have different velocities according to the shot power.

A *simple* strategy is to scan for opponents and immediately shoot in an opponent's direction with full power once one is found. This obviously does not work very well with moving robots.

Fire away is another simple strategy in which a robot shoots in all directions but only with a low power. The idea is that at least some of the shots will hit an opponent.

Move-in move-out is a general strategy where a robot finds a target, quickly moves close to it and shoots with maximum power. Afterwards, it quickly moves to a safe distance. This would probably work quite well in 1-on-1 battles, since big parts of the arena will be empty. It may not, however, be trivial to run away from an opponent. This strategy is not always possible to use in melee combats, because there may be nowhere to hide. This caveat is common to all evasive strategies.

A more advanced strategy might be to try to predict the target's future positions from the heading and velocity returned from the radar scan. If the opponent moves in straight lines, it is fairly easy to determine how to shoot. *Pattern matching* can recognise robots who exhibit a particular behaviour such as following the arena walls, hiding in corners or moving in circles. Pattern matching can also be used to deduce the current strategy used by opponents which can aid in choosing which strategies to use. This also enables a robot to set shooting power based on complex factors such as confidence in that the shot will hit.

The Robocode `AdvancedRobot` has the following methods for controlling shooting actuators:

- **fire** and **fireBullet**. Shoots a bullet if the gun is cold. **fireBullet** returns a Bullet object for the fired bullet.
- **setTurnGunLeft** and **turnGunLeft**. Turns the gun left.
- **setTurnGunRight** and **turnGunRight**. Turns the gun right.
- **setAdjustGunForRobotTurn**. Enable or disable the locking of the gun to the base of the robot.

1.3 Summary

In this chapter Robocode has been introduced as a game platform, relevant details of the game rules have been discussed, and an overview of a number of strategies and methods has been given. This overview has shown that most strategies have both advantages and disadvantages, and typically these disadvantages depend highly on the game situation. The chapter has also shown that within Robocode there are problems for which it is hard to specify an algorithm or method which defines the desired solution.

Chapter 2

Previous Work

Although Robocode was originally intended as a framework for learning Java, a lot of skilled developers have shown interest in the system resulting in a range of advanced robots. Through the study of existing robots, this chapter unveils advanced methods which have proved effective and provide inspiration for the design of future robots.

Since the focus of the project is on machine learning, this chapter will start by defining the class of methods covered by the term “machine learning”. Three existing methods, previously used in Robocode robots and complying to this definition, are identified and discussed: Neural network targeting, symbolic pattern matching and genetic programming.

2.1 Definition of Machine Learning

ML is intuitively associated with systems which are not restricted by their original programming and automatically evolve over time. This intuition is captured in the definition from [FOL]:

Definition: Machine learning is the ability of a machine to improve its performance based on previous results.

To further formalise the terms “improve performance” and “previous results”, the definition of learning from [Mit97, p. 2] is adopted:

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

In the context of Robocode, the task will generally be to *compete against other robots*, and the performance measure P could be the *percentage of games won* or *score achieved*.

Consider as an example a robot that predicts the future position of its enemy by employing the pattern matching methods to be discussed in the next section. In this case, the task, T_1, is to hit the enemy, the performance measure, P_1, is the percentage of successful hits, and the experience, E_1, will be defined by previous enemy movement patterns. If the robot successfully improves P_1 based on this experience, it is considered to learn.

On the other hand, consider a robot which always assumes that its enemy moves linearly and tries to predict future positions based on a linear target prediction mechanism. Because this robot does not base its actions on any experience, E, it does not learn anything with respect to T_1 and P_1. Similar arguments apply to many other targeting and movement techniques, which consequently are not considered here.

2.2 Pattern Matching

As the examples in the previous section suggest, robots which employ pattern matching for targeting are covered by the definition of learning. Two interesting approaches are mentioned in [Robb], namely *neural network targeting* and *symbolic pattern matching*, where two example implementations include the robots `ScruchiPu` and `NanoLauLectrik`, respectively.

These are chosen for investigation because they perform relatively well, e.g. they have both achieved a top 3 position in the Robocode Outpost league [Robb, Sections ScruchiPu and NanoLauLectrik]. Furthermore they are some of the first robots to apply the techniques in focus, and are therefore used as the basis for other robots employing similar techniques. The following sections build upon the brief descriptions in [Robb, Sections NeuralTargeting and SymbolicPatternMatcher] as well as the `NanoLauLectrik` source code available from [Roba].

From a high level perspective, neural network targeting and symbolic pattern matching both strive to approximate the following function:

$$f: \quad X^n \times T \rightarrow X \tag{2.1}$$

X is the set of all possible samples, each of which represents the enemy's state (e.g. position) at a certain time, and T is the set of possible time instances. Given a sequence, $s = x_1, x_2, \cdots, x_n$, of recently observed enemy states and a future time, t, the expected future enemy state at this time is given by $f(s, t) = x_t$.

2.2.1 Neural Network Targeting

`ScruchiPu` bases its approximation of Equation 2.1 on a neural network; refer to Figure 8.1 for a general introduction to this topic. In this particular solution,

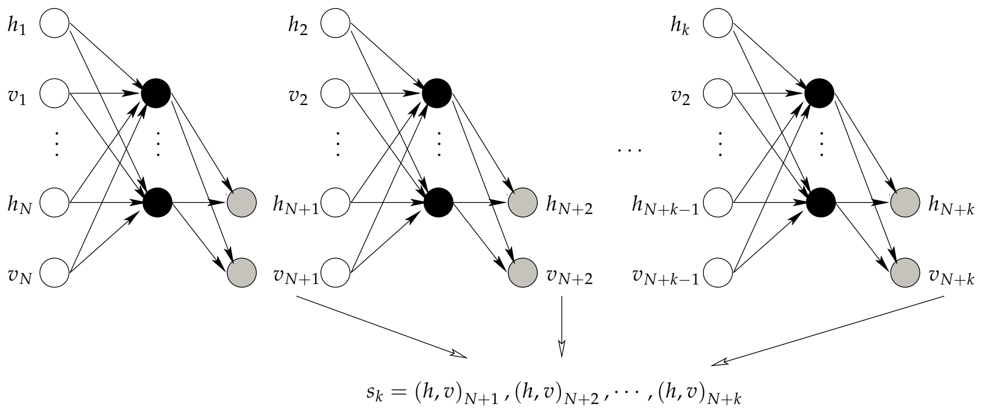


Figure 2.1: Construction of s_k using multiple network feed-forward cycles. The gray neurons represent the output layer, and are used as the last two inputs in the succeeding cycle.

the sequence s of observed enemy positions is composed of pairs (h, v) of enemy heading and velocity: $s = (h,v)_1, (h,v)_2, \cdots, (h,v)_n$.

The neural network is designed with one hidden layer, and $2 \cdot n$ input neurons are used to represent the sequence s. Two output neurons represent the expected $(h,v)_{n+1}$ pair.

Observe that the described network does not take any time parameter into account. This means that each of the samples in s must be sampled with the same delay, Δt, and that the output always represents the expected velocity and heading at time Δt after the sampling of the input sequence. In order to derive the enemy's position at at an arbitrary point in time, t, a sequence $s_k = (h,v)_{N+1}, \cdots (h,v)_{N+k}$ must be constructed by performing $k = t/\Delta t$ network feed-forward cycles. For instance the $(h,v)_{N+2}$ pair is produced by inputting $(h,v)_2 \cdots (h,v)_{N+1}$ to the network; see Figure 11.3 for the general approach.

Finally the actual future Cartesian coordinates of the enemy must be constructed from s_k. This will require the assumption of linear movement between each sample, but obviously the error induced by this assumption approaches 0 as Δt approaches 0. As an example, consider deriving from s_k the expected increase in Cartesian coordinates $(\Delta x, \Delta y)$ between samples $(h,v)_i$ and $(h,v)_{i+1}$, given that the velocity is measured in pixels/second and the delay, Δt is measured in seconds:

$$\begin{aligned} \Delta x &= v_i \cdot \Delta t \cdot sin\left(h_i\right) \\ \Delta y &= v_i \cdot \Delta t \cdot cos\left(h_i\right) \end{aligned}$$

Note that the heading is measured positive clockwise from the y-axis, hence this arrangement of the trigonometric functions. Following this approach, the enemy's expected path can easily be constructed from its initial position by vector addition.

Further details on network topology, activation functions, training algorithms and methods (online/offline) are not given by the authors of `ScruchiPu`, except that a low momentum and low learning rate are mentioned to give the best results.

2.2.2 Symbolic Pattern Matching

In symbolic pattern matching the enemy robot's movement patterns are explicitly sampled and preserved in a list referred to as the *history* $h = s_1, s_2 \cdots s_M$, the size of which increases linearly with time. This is in contrast to neural network targeting discussed above, where previous experience is implicitly encoded in a constant number of network weights (although a recent history is also maintained).

When anticipating future locations of an enemy robot, the following actions are generally taken:

1. Find a sequence in h matching the recent n enemy samples, and let p mark the end position of the match. If no match is found (or only the trivial match at the end of the history is found) then abort. Relating to Equation 2.1, the recent n samples correspond to sequence s on which the prediction should be based.

2. Replay the history from position p for a duration corresponding to the time it takes an emerging bullet to reach the target robot. Again, this time interval corresponds to t given in Equation 2.1. The history sample at the end of replay will be the anticipated enemy position after time t.

The questions of how to actually represent the history and perform the matching are of vital importance, and the following considers these issues in relation to the `NanoLauLectrik` robot.

`NanoLauLectrik` maintains two separate histories for the same robot: One for matching, h_m, and one for reconstructing, h_c. Matching is based exclusively on velocity patterns, so each sample in h_m simply represents the enemy's velocity ($[-8; 8]$) at the sample time. A velocity history may appear insufficient at first, as it does not explicitly capture changes in direction (except for the sign which captures forward and backward movement). However a change in direction is often reflected in the velocity, because a robot cannot turn while moving at full speed. As a technical detail, the velocity is represented by a symbol (character) and not a real number; this allows the matching to be based on Java's regular expression engine, hence the name *symbolic* pattern matching.

When a matching position p in h_m is found, the enemy's expected position after further t samples can be constructed by replaying sample p through $p + t$

in the second history, h_c. To allow this reconstruction, samples in h_c could for instance contain the enemy's horizontal and vertical velocity[1].

2.2.3 Nondeterminism

Common to both pattern matching methods described above is the assumption of deterministic movement. This assumption holds for many simple robots, e.g those included in the Robocode `sample` package. But whenever a robot exhibits nondeterministic movement (e.g by randomly choosing its direction) the movement function will not surrender itself to approximation, and the pattern matching fails utterly. Some robots which employ pattern matching therefore disengage the matching when they discover that the frequency of failed predictions exceeds a given threshold. Note however that neural networks can handle some degree of nondeterminism, as they are more robust to noise than symbolic pattern matching, which requires an exact history match.

2.3 Genetic Programming

In the context of Robocode, genetic programming involves automatic generation of robot controlling programs. Appendix C gives an introduction to the area of genetic programming.

In the terms of *learning,* set forth in Section 2.1, the goal (or task), T, could be to evolve a robot that achieves the highest performance, P, measured by the score assigned to the robot by the Robocode system after a battle. Experience, E, could be defined by the success rates (i.e. scores) and genome strings of robots from earlier generations.

An attempt at applying genetic programming to Robocode robots is documented in [Eis03]. This section outlines that approach and presents the main results from the paper.

2.3.1 Design Outline

A small language, TableRex, is used for evolving robots and is interpreted at runtime to control the Robocode robots. In the example TableRex program shown in Table 2.1, each row defines an action constituted by a function and two inputs (parameters) and one output. For instance row 6 evaluates the boolean expression "value at line 4 less than 90" and stores the result in the last column of the row (1 for true). A total of 16 functions are available, providing basic arithmetic and logic operations as well as a few operations convenient in the context of Robocode – most importantly a function to control the robot's

[1]`NanoLauLectrik` actually stores only the (accumulated) vertical velocity in h_c, but by Pythagoras's theorem the horizontal velocity can be derived from the vertical velocity and the absolute velocity stored in h_m.

Function	Input 1	Input 2	Output
1. Random	ignore	ignore	0,87
2. Divide	Const_1	Const_2	0,5
3. Greater Than	Line 1	Line 2	1
4. Normalize Angle	Enemy bearing	ignore	-50
5. Absolute Value	Line 4	ignore	50
6. Less Than	Line 4	Const_90	1
7. And	Line 6	Line 3	1
8. Multiply	Const_10	Const_10	100
9. Less Than	Enemy distance	Line 8	0
10. And	Line 9	Line 7	0
11. Multiply	Line 10	Line 4	0
12 Output	Turn gun left	Line 11	0

Table 2.1: An example TableRex program from [Eis03, Figure 1]. Note that even though the fourth column actually represents function results; it can also be used as an input – its value would be the result from the previous run of the entire program.

actuators (as shown in the last row of the table). Inputs may reference game parameters such as *enemy distance* in line 9.

Three TableRex design criteria are mentioned:

1. Support of an efficient interpreter, giving fast evolution of programs.
2. Programs should be encodable as a fixed-length genome which enables easy cross-over.
3. Output of ordered sequences of actuator control commands should be possible as robots typically require multiple sequential actions to work satisfactory.

TableRex programs are incorporated into Robocode robots using a subsumption architecture, as further discussed in Chapter 3. This means that a TableRex program is associated with each event that the robot should respond to, and that the programs are arranged into hierarchies of importance. Further details of the architecture are not given by the author.

2.3.2 Results

The evolved robots have been tested against the sample robots as well as the non-adapting `SquigBot` from the showcase category [Roba] in which robots have been rated as exceptional. A total of four test settings were considered by adjusting two variables: *Number of adversaries* (one or multiple) and *starting position* (fixed or random initial map positions).

In the simplest setup, *one adversary and fixed starting position,* robots were evolved which convincingly beat `SquigBot` as well as the sample robots. But when using *multiple starting positions,* the success rate against `SquigBot` dropped to 50% and longer evolution was required. The increased difficulty in this configuration can be explained by the need to compensate for random "bad" initial placements (e.g. a corner of the battlefield).

When using *multiple adversaries and fixed starting positions,* robots could no longer evolve to avoid the adversary's firing pattern, and only robots from the sample collection could be beaten. The final setup, *multiple adversaries and random starting positions* never succeeded in evolving any competitive robots because of unacceptably long training delays (estimated 130 hours on a 1 GHz Pentium 3). Hence the problem of limited processing resources should be kept in mind.

As a general problem, robots were rarely evolved to take advantage of their gun. Several possible explanations for this are given, the most feasible being that early generation robots that attempted to fire were removed from the population because they typically missed their target and thus lost energy fast. This issue could be a subject for improvement, e.g using neural networks for targeting.

2.4 Considerations

Based on the definition of learning, two methods for targeting and one method for general control of robots have been discussed. 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 difficult 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.

Chapter 3

Agent Architectures

In this chapter, a robot is considered in the more general perspective as an agent and several architectures for agents are explored. This is done in order to enhance modularity so that the general problem of building a robot can be split into smaller, more manageable problems. In addition, architectures as a concept are employed so that standard terminology can be used. This makes it possible to draw on previous experience with agents, and makes it easier to communicate the structure of the robot.

First agents are defined, followed by an analysis of three existing classes of architectures — leading to a summary of the pros and cons of each of the classes.

3.1 Definitions

The concept of agents has become an important subject in computer science and is usually divided into three major categories: Theory, Architecture and Languages. The theory deals with what one could call *the essence of agency*, or in specific terms: A definition of an autonomous agent.

The pursuit of defining autonomous agents has given rise to many discussions throughout the scientific community, and it could seem beneficial to display the central aspects of these discussions here as they influence the choice of architecture for Aalbot. But for the purpose of this report, and to maintain the focus of autonomous agents in computer games, we simply adopt the following definition, presented in [SA96]:

Definition: An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to affect what it senses in the future.

The definition serves the purpose of being both generic and precise, while leaving the choice of architecture open. To grasp the precise meaning of this definition, the following keywords are elaborated:

- **Autonomous.** An agent is considered autonomous when able to operate without direct intervention from humans, and it has some control over its actions and internal state.
- **System.** The collection of software or hardware entities that the agent is comprised of.
- **Environment.** The boundary that the agent operates within, and the only thing that the agent is able to sense and manipulate, e.g. the Internet or the Robocode environment. The agent itself is a part of the environment.
- **Senses.** Input to the agent. Various agents *sense* in various manners, e.g. through sensors attached to the agent such as the radar on a Robocode robot.
- **Acts.** The means for the agent to affect its environment.
- **Agenda.** The goal of an agent. Can be comprised of wishes, desires or some form of a plan. Some agendas are implicitly defined by the mere structure of the agent.

Agent architectures can be viewed as software or hardware engineering models. They deal with the design of systems that satisfy a definition of an autonomous agent. As the keen reader observes, different agent definitions yield different architectures.

The research on languages regarding autonomous agents strives to define programming languages that specifically support the programming of agents. This is done by providing primitives that allow a programmer to express agents as software systems using the principles defined by agent theorists. The subject of autonomous agent languages will not be revisited in this report.

The analysis will be conducted separately on two traditional classes of agent architectures, namely *deliberative* and *reactive* architectures. Finally a short description of a set of recent hybrid architectures, combining the two traditional architectures, will be presented.

3.2 Deliberative Architectures

The deliberative agent architectures consider an agent as a particular type of knowledge based system, as knowledge gathered throughout the lifetime of an agent is used to pursue its agenda. The architectures in this class maintains a symbolic world model explicitly represented within the agent. The symbolic world model differs from the definition of *environment* in the previous section as the agent needs not be an explicit part of it, nor does the world model have to encompass the whole environment. Nevertheless, one could argue that an agent is always implicitly a part of such a world model since all observations are made by the agent itself, and thus the model becomes relative to the agent.

Deliberative agents make decisions by reasoning about, and symbolic manipulation of, the aforementioned world model as depicted in Figure 3.1. This architecture is often referred to as *symbolic AI,* as the symbolic world model makes way for a great variety of intelligent decision-making.

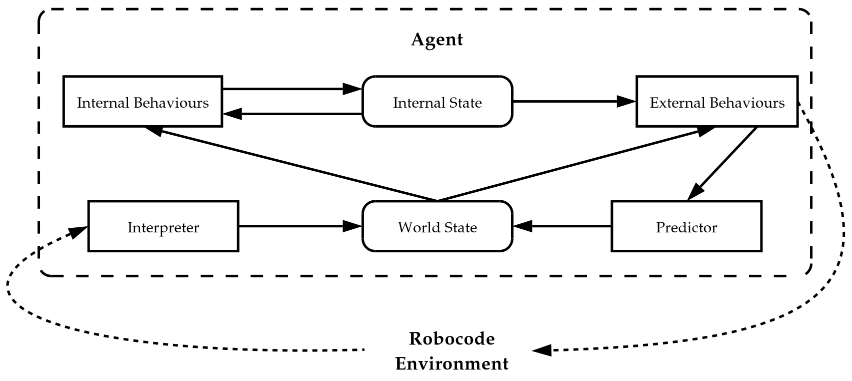


Figure 3.1: A deliberative architecture for a Robocode robot adopted from the Robocup CMUnited99 Team Member Agent Architecture [Bea00]. The agent maintains a model of the environment in the world state component, which is used to choose the action of the robot.

In order to grasp the symbolic AI paradigm consider a fly as an autonomous agent using a deliberative architecture as controller. The fly has an internal representation of the world it resides in, for now be it a kitchen, with every object in the kitchen mapped along with the distance to the walls. The fly also needs to be aware of its own position in the kitchen. The fly has sensors fitted onto it, i.e. eyes and some sort of scent scanner, in order to maintain this internal model of the world.

If the fly senses a delicious eatable substance in the kitchen and at approximately the same time sees a threat, e.g. a man with a newspaper, the fly has to perform logical reasoning to determine its future behaviour. This could be to evaluate the distance to the food and the threat relative to itself, and perhaps even consider if it could grab some of the food on the getaway from the newspaper.

Finally the fly could use some sort of memory to decide if the eatable substance is really nutritious enough to risk its life for, or if the person holding the newspaper is a poor hitter. All of these considerations have to be performed within a time boundary for the fly to survive, at the same time maintaining its internal representation of the kitchen and its objects.

Some unresolved issues regarding deliberative agents remain. Especially the task of creating an accurate symbolic world model of the environment in which the agent resides has proven difficult. The difficulties have revolved around keeping this model simple enough for efficient knowledge extraction, while still being adequate for an agent to make intelligent decisions within some time boundary[Bea00].

3.3 Reactive Architectures

The problems surrounding the deliberative architectures, as explained in the previous section, have led to the development of alternative architectures. These alternatives can roughly be divided into purely reactive architectures and hybrid architectures (i.e. between deliberative and reactive architectures [WJ95]). The reactive architectures are characterised by the lack of a world state model. In 1985 Rodney Brooks, an ardent critic of the symbolic AI paradigm, presented a reactive architecture, the so called subsumption architecture. This architecture has many promising features, yet many modifications and improvements have materialised since it was first introduced.

The subsumption architecture rests on two key ideas originating from Brooks.

- The real intelligence is situated within the world itself, not in disembodied systems.
- Intelligent behaviour arises as a result of an agent's interaction with its environment. It is not an innate, isolated property.

These two ideas outline the idea that intelligent behaviour can be achieved without maintaining an explicit model of the environment of an agent and without the abstract reasoning as presented in the symbolic AI paradigm.

The subsumption architecture is a vertically layered architecture, with each layer transforming input to output, as depicted in Figure 3.2. The input to an agent built upon this architecture is typically produced by sensors, and the output most often activates an actuator. Upper layers have precedence over lower layers and *subsumes* the output of lower layers, thus the name *subsumption* architecture.

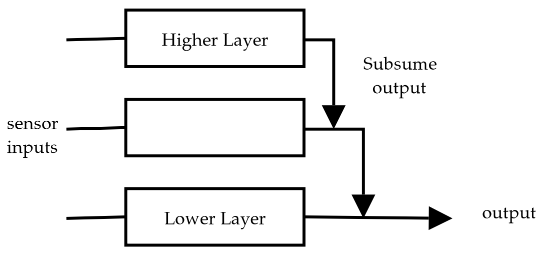


Figure 3.2: The layered approach. Higher positioned layers subsumes the output of lower layers.

To exemplify the subsumption architecture consider again the fly. This fly now has two layers, a danger management layer and a food management layer. These two layers react to sensor input from the fly, the exact specification of the sensors is not relevant for this discussion. When the danger management layer receives input the fly is more inclined to move fast and away from its current location. When the fly receives input from its food management layer the fly is inclined to move towards the stimulant. In case a fly receives strong inputs in both layers at approximately the same time the fly has to perform intelligently to survive (or to get food one might say). A vital property for the fly to survive is that the danger management layer is able to subsume the output of other layers. If we arrange for the food management layer to be *below* the danger management layer, the possibility of the fly escaping instead of being smashed while eating certainly exist. And we might say that the fly made an intelligent choice by escaping.

As Brooks has demonstrated [WJ95], adding more layers to specific agents will allow for intelligent behaviour, without suffering from the downside of maintaining a symbolic model of the world of the agent. Still the architecture suffers from a couple of drawbacks: The subsumption architecture does not allow for any shared memory or other communication between the layers, making it impossible for the layers to cooperate in order to achieve a common goal. To illustrate this, consider Figure 3.3 where the two layers' considerations are depicted.

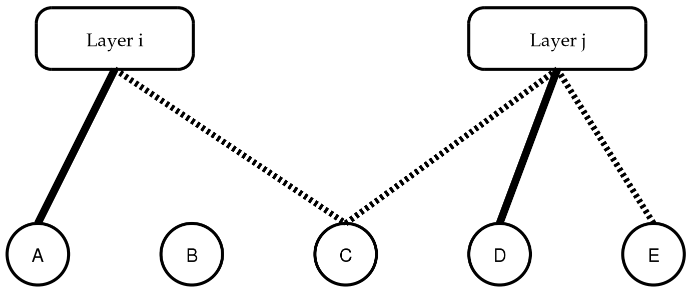


Figure 3.3: The subsumption architecture does not allow for two layers to agree on a best common output.

The layers determine their output, marked with bold lines. The dashed lines indicate acceptable but not optimal output for each layer, the layers do not agree on the best option. The output *C* would be acceptable by both layers, but it is not chosen. This is due to the upper layers subsuming the entire output of lower layers disabling any two layers from cooperating and outputting a best common result.

Critics further argue that reactive architectures can only perform successfully under specific circumstances, such as [WJ95]:

- Sensor stimuli for unambiguously determining the behaviour of an agent is always sufficiently present.
- No *global task* has to be reasoned about by the agent, e.g. no global strategy.

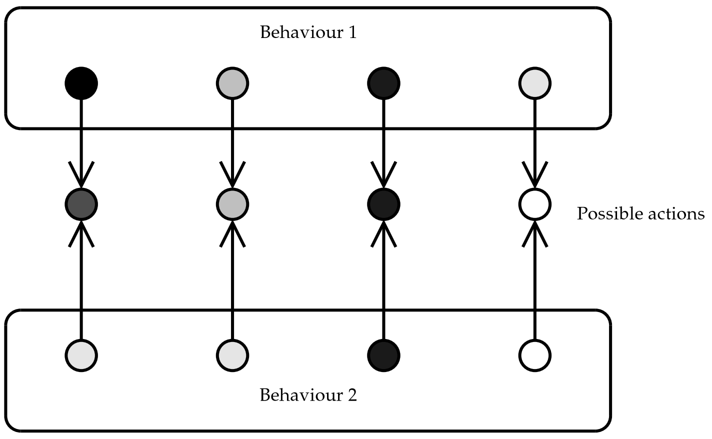


Figure 3.4: Fine-grained subsumption. Two behaviours weight possible actions. The circles represent the weighting of possible actions. Black is a maximum weight, white is the minimal weight; grey-scales represent intermediate values. The circles within the behaviours represent the weighting of the actions from the behaviour. The circles outside the behaviour is the compromise of two behaviours. The selected action is simply the one with highest weight, in this case the third from the left. [Ros89]

- The goals and desires of an agent can be implicitly defined by a ranking order.

Interesting alterations have been proposed to alleviate these undesirable properties of the subsumption architecture, the most promising one being fine grained subsumption.

The fine grained subsumption, proposed in [Ros89], aims to resolve the lack of communication possibilities between layers in the subsumption architecture. This is achieved by fragmenting the layers into smaller decision-making units. Each unit has the task of transforming input values to intermediate output values, thus the connections of a decision-making unit define its ultimate role in the system. Each layer represent a certain behaviour of the agent Between each pair of neighbouring layers is a set of intermediate units through which the two layers communicate to find an optimal solution. The fine grained approach allows for distinguishing between behaviours and commands, resulting in behaviours with access to the internal state of other behaviours, and commands with the basic limitations adhering to the initial subsumption architecture. On Figure 3.4 the cooperation of two behaviours using fine-grained subsumption is shown. This architecture enables an agent to pursue a global strategy, thus removing one necessary circumstance, namely that of no *global task* to be performed by the agent.

3.4 Hybrid Architectures

In the field of research on agent architectures it has been argued that neither a pure deliberative nor a pure subsumption architecture is sufficient to produce highly intelligent agents[WJ95, p. 23]. Instead a combination of these two classical architectures with other alternative architectures has been proposed. The approach of hybrid architectures seems viable, as the limitations of the two other classes of architectures can be alleviated. E.g. some minimal model of the environment could enhance the performance of reactive agents. The research on hybrid architectures is only very recent and thus no deep theory has yet materialised, resulting in ad hoc approaches to highly specialised applications [WJ95].

Chapter 4

Analysis Results

Based on the three previous chapters, the preliminary goal from the introduction of this report will be elaborated and narrowed into a concrete goal and success criteria for Aalbot.

The concrete goal of this project is to:

> Design and implement a robot using various machine learning methods that in 800×600 pixel large arenas score well in combat against 3 opponents. The measurable success criteria will be the score achieved in combats against the robot `Walls` from the sample robot set, and `Squigbot` and `Peryton` from the show-case section of the Robocode repository. Aalbot must not be trained specifically for battling these 3 robots. It must be able to achieve at least 2nd place in a battle comprised of 10 rounds.

Based on the analysis of Robocode, it is chosen to design Aalbot to be used in melee battles with a fixed number of opponents. Melee battles are chosen because many interesting problems arise in melee battles, which are not present in 1-on-1 battles. By choosing melee battles, Aalbot will also have to do well in 1-on-1 battles, because otherwise it cannot be the last surviving robot.

It is also chosen to use a fixed arena size of 800×600 pixels – partly because this is the default in Robocode but also because the greatest distance in such an arena is $\sqrt{800^2 \times 600^2} = 1000$ pixels which is less than the radar scanning distance limit. Choosing a fixed arena size removes a further complication for the machine learning methods, as Aalbot does not have to adapt to different arena sizes. Battles of 4 robots in total are chosen because this seems to leave enough space for the robots to move around without leaving vast areas of the arena empty, as well of ensuring that Aalbot does not have to adapt to battles with a variable number of starting robots.

To ensure that Aalbot is able to adapt to enemies and uses general strategies not tailored to just one type of opponent, it is essential that it is not trained with the robots used for evaluating the results. The three opponents are chosen to be the `Wall` robot from the sample package and `Squigbot` and `Peryton` from the show-case section of the Robocode repository. The latter two robots are considered to be “exceptional”, and it can therefore be assumed that if the robot beats them, it must be good.

Based on these decisions, it is anticipated that Aalbot will be a highly efficient robot which will adapt to its opponents because of the machine learning involved. In Part III, the results achieved with Aalbot will be evaluated.

Part II

Design

Chapter 5

Aalbot Architecture

In Chapter 3 it was shown that although there exists different agent architectures, the deliberative and reactive architectures have known limitations. The pure deliberative architectures have some unresolved issues regarding symbolic representation of the agent's environment. Although the pure reactive architectures were developed to avoid some of the problems with the deliberative architectures, they also have some problems: the reactive subsumption architecture does not allow for any shared memory or other form of cooperation between the layers. Thus making it impossible for the layers to cooperate and choose the most optimal action for all the layers. Fine grained subsumption attempts to alleviate this by dividing the layers into smaller units that are able to communicate between layers, making it possible for the agent to take the best overall action. As mentioned in Chapter 3 hybrid architecture is an architecture that is neither pure reactive nor pure deliberative, but instead something in between the two; taking advantage of both extremes.

This leads to the design of the hybrid architecture to be used in Aalbot, which is intended to suit the Robocode framework. In the following, the considerations behind this are elaborated.

5.1 Structure

The designed architecture, shown in Figure 5.1, is based on three components: a world state component, a subsumption layers component and an arbitrator component. The architecture has two types of flow: data flow and wish flow. The data flow carries sensor input to the layers, data from the world state component to the layers and actions from the arbitrator to the actuators. The wish-flow is a special kind of flow between each layer in the architecture, from the top layer to the arbitrator and from the layers to the world state component. The layers create a *wish list* for the arbitrator to convert into actions. This wish list is carried by the wish flow.

There are two types of wishes, *actuation wishes* and *abstract wishes*. Actuation wishes are simple requests that map directly to actuator activations, i.e. they

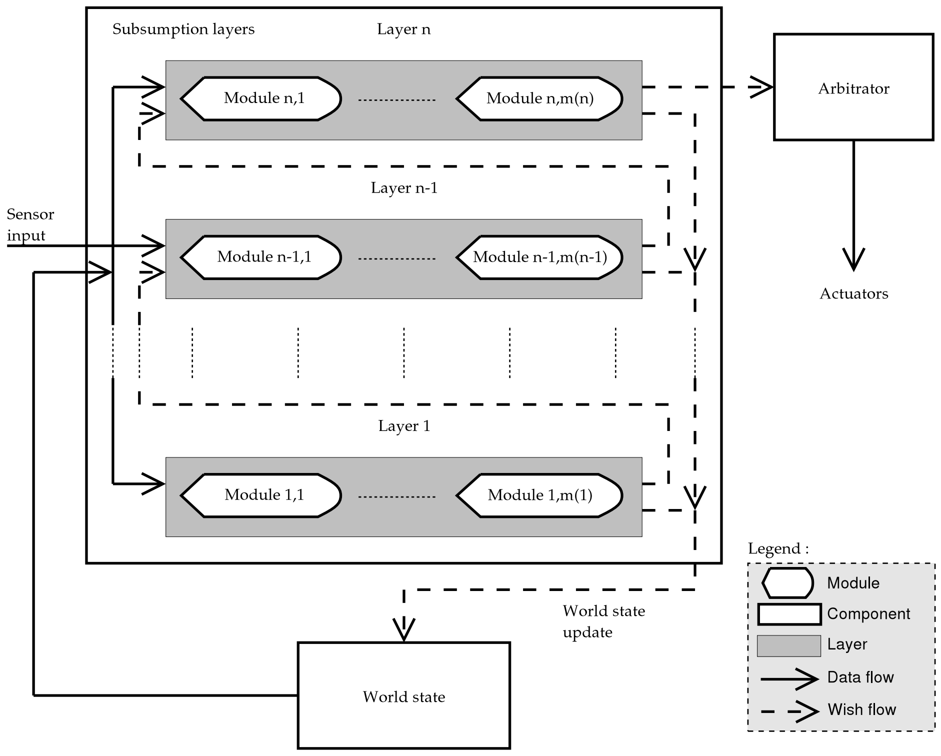


Figure 5.1: The overall structure of the chosen architecture. For each layer x, $m(x)$ is the number of modules in that layer.

are *fulfilled* by the arbitrator, e.g. "move forward 30 pixels" or "turn gun 30 degrees clockwise". Abstract wishes are a higher level of requests, that do not map directly into simple actuator activations, i.e. "move to (x, y) when convenient" or "add enemy e to world state map of enemies". To *fulfill* a wish is simply to remove the wish from the wish list and to perform some action in an attempt to make the wish come true. For the arbitrator to fulfill an actuation wish is simply to translate the wish into an actuator activation and issue it. Abstract wishes can be fulfilled by modules and the world state component. Abstract wishes do not necessarily result in actuator activations, but are instead included in some model used by the fulfilling module or component. An example of this could be an abstract wish on the form "attack enemy e". A fulfilling module could then perform the actual job of aiming and firing at enemy e, e.g. by issuing actuation wishes "turn gun 10 degrees counter-clockwise" and "fire gun with power 3".

5.2 World State Component

The world state component is a shared memory for the layers. At the beginning of each turn the world state data are sent to Layer 1. The layer is able to alter the world state by sending an abstract wish with the desired change to the

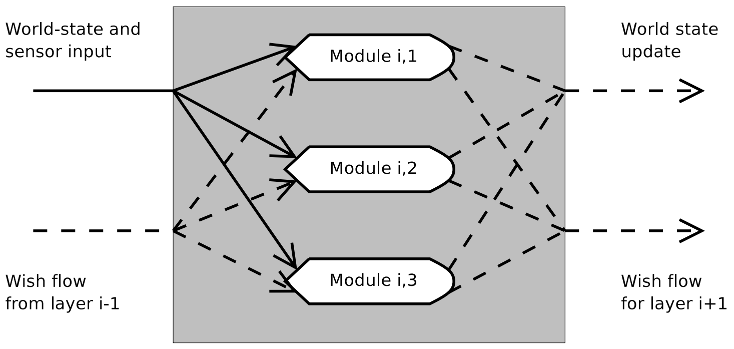


Figure 5.2: A closer look at a layer showing the flow between modules inside layer i, for all $1 < i < n$. Layer 1 does not receive any wish flow, as it starts with an empty wish list. Layer n does not send a wish list to the next layer, instead it sends it to the arbitrator component.

world state component. The world state component then fulfills the abstract wish by updating the world state according to the wish. The world state data are then sent to Layer 2, which has the same ability as Layer 1. This continues until layer n is finished. I.e all layers are initiated in each turn.

5.3 Subsumption Layers Component

The n subsumption layers build up a wish list for the arbitrator to manifest into actuator activations. The first layer starts with an empty wish list. Each layer is able to alter the wish list it receives as input, by adding new wishes and removing wishes from underlying layers. Abstract wishes are meant to be fulfilled by modules in overlying layers, i.e. removed from the wish list and taken into consideration when the fulfilling module makes its wishes. Thus when the wish list reaches the arbitrator it should not contain any abstract wishes, if it does then these are simply ignored.

Each layer consists of a arbitrary number of modules. A module has access to the world state data, sensor input sent to the layer and the wish list received from the underlying layer. The operation of each layer is simply the execution of each module in the layer. When every module in a layer has finished executing, the layer sends the wish list to the next layer and a world state update to the world state component. This is shown in Figure 5.2. An important point is that the architecture allows for parallel execution of the modules in each layer, as the modules can execute independently.

5.4 Arbitrator Component

The arbitrator component receives its input from the top layer. The duty of this component is to resolve possible conflicts in the wish list, and to transform the wish list into actuator activations. The arbitrator has to be smart, i.e. it has to know how to combine several wishes into one consistent action. If the arbitrator is not able to come to a compromise, it can prioritise wishes based on which layer they originated from.

5.5 Design Considerations

The architecture is designed to fit the specific characteristics of the Robocode game. Recall from Chapter 3 that in Robocode a robot is given the same amount of time each turn, and thus the only key element is to issue the actions before the end of the turn. Thus in order to maximise the potential of a robot, it is best to spend an amount of time very close to the maximum at each turn before issuing any action, so that the robot has the most time to ponder its current situation. This is reflected through the design of the architecture, where each turn activates all modules in all cases and therefore a turn is estimated to spend about the same amount of time. In other words, there are no "quick" or specific "slow" decisions, where the robot is significantly faster or slower at deciding its actions by preventing some modules from running. This is a clear difference between Aalbot architecture and other hybrid architectures, where "shortcuts" are often used to enable *reflex-like* decisions by the robot [WJ95, p. 23-25], which is clearly not an issue considering the turn based structure in Robocode.

The Aalbot architecture is designed to avoid the common problems with pure deliberative and reactive architectures. The architecture contains a shared memory, the world state component, as the deliberative architectures — however modules need not use this component, making them reactive. The world state is updated by the modules in the subsumption layers, which makes it versatile. Each module receives the direct, uninterpreted sensor input at each turn, as in the reactive architectures.

Chapter 6

Component Design

In Chapter 5 several components for the Aalbot robot were introduced. In this chapter the design of the world state component and arbitrator component is presented, as used in the implementation of Aalbot. The subsumption layers component is composed of a number of modules. To avoid clutter these are introduced later, in Chapter 7. The design of the world state component and arbitrator component is introduced to give the necessary overview needed to understand how Aalbot works.

6.1 World State Component

The world state component is the shared memory for the modules in the subsumption layers component. There are two ways for modules to communicate, one way is to use wishes that can be fulfilled by overlying modules, the other way is through the world state component. The modules maintain a model of Aalbot's environment in the world state component, which will be explained in the following.

6.1.1 Table of Enemy Information

Aalbot needs to keep information of the enemy robots in the arena. These data are kept in a table of enemy information. The table is indexed by a unique robot identifier. For each robot, e, on the battlefield the following fields exist in the table:

- **Position.** The (x, y) coordinates of e at the time e was scanned.
- **Energy.** The energy of e at time of scan.
- **Heading.** The direction that e was facing at time of scan.
- **Velocity.** The velocity of e at time of scan.

- **Time stamp.** The time at which the information was last updated, i.e. time of last radar scan of e.

6.1.2 Fading Memory Maps

The world state component uses the concept fading memory map, introduced in this section. Aalbot uses two kinds of these fading memory maps. A more specific description is given later of the two fading memory maps. A fading memory map consists of a map of points, which can be placed with pixel precision corresponding to the Robocode battlefield. The points can be used to hold any kind of information, for instance gravity points for the movement module or enemy positions. Each point is tagged with some data and a time stamp describing the time at which the point was placed on the map. This stamp is used to degrade validity of the information contained within the point.

The rationale behind the fading memory is that it reflects the fact that the world state changes over time. Information gathered about the world is likely to get obsolete after a certain amount of time. The more recent an observation is, the more it can be trusted to be precise. It also helps to keep the model of the robot simple and the space cost for each turn from becoming a monotonically increasing function over time. The worlds state component maintains the fading memory maps.

Fading Memory Anti-gravity Movement Map

The fading memory anti-gravity movement map is a map used by the movement module described in Subsection 7.4.1. Its functionality in the world state component is limited; it is only used to store gravity points.

Fading Memory Target Map

The target map, T, consists of a set of slots, where each slot represents a non-overlapping area of the arena. Each slot holds a set of pairs (r, p) where r is a unique robot identifier, and $p \in [0; 1]$ represents the probability that the robot identified by r is in the area represented by the slot. The sum of probabilities for each robot r' over all slots where there exists a pair (r', p) is 1 if the robot r' is in the arena or 0 if it is not. This is referred to as the summation property in the following.

Once a robot r is scanned by the radar in an area represented by slot, $S \in T$, that slot has any previous (r, p) pair replaced with $(r, 1)$. And for each slot $W \in T$, where $W \neq S$, any (r, p) pair is removed from W, if present. This is the situation where the position of robot r is known with absolute precision, and there is only one slot with a probability pair (r, p). Figure 6.1 a) shows this situation.

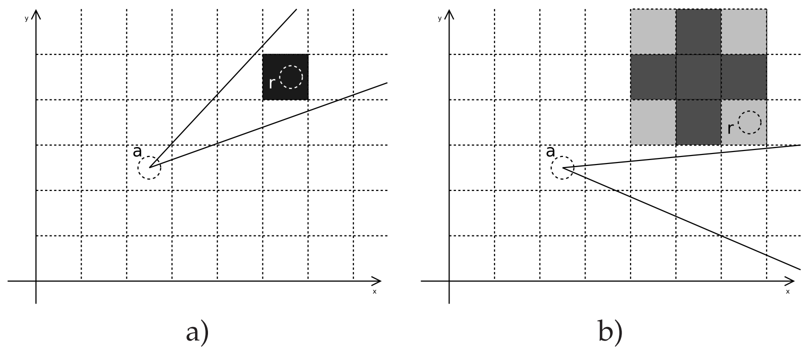


Figure 6.1: In a), Aalbot, identified by *a* scans a robot *r* and updates its fading memory target map. Each square between dotted lines represent a slot, white squares indicate zero probability of a robot in this area, black squares indicate a probability of 1 that this slot holds a robot. Grey squares represent values between 0 and 1. The radar cone of Aalbot is shown as the lines originating from *a*. In b), the Aalbot and robot *r* are depicted in another situation. Aalbot has moved its radar so it does not scan *r* in this turn. However Aalbot updates its target map to reflect the new possible positions of *r*. In the mean time *r* has also moved from one slot to another.

For each turn when a robot *r* is not scanned by the radar, the target map *T* is updated to reflect the new probabilities of the whereabouts of robot *r*. This is done in such a manner that the summation property holds for *r*. This is shown in Figure 6.1 b). The updating of *T* is done using v_r and h_r which are velocity and heading of the robot *r* retrieved from the table of enemy information. The probability of *r* is moved in the direction h_r by v_r.

When an area is scanned and it is found to not contain a certain robot *r*, for each slot *S* covered by the radar cone the slot is updated to remove any probability that *r* is in this area, simply by removing any pair (r, p) from *S* if the slot contains a pair (r, p). All other slots must be updated, to distribute the removed probability from this slot into the other slots. This is shown in Figure 6.2.

The above description leaves out several details. Recall from Section 5.1 that the world state component does not receive any sensor input and thus cannot decide when robots have been scanned or not. Instead this is done by the sensor interpretation module which is placed in Layer 1 as described in 7. The sensor interpretation module receives sensor input when robot *r* has been scanned and sends an abstract wish destined for the fading memory target map with this information. The fading memory target map then performs the necessary updating to reflect the new probabilities for robot *r*. For each robot *i* not scanned in that turn, the fading memory target map updates the probabilities for *i* accordingly.

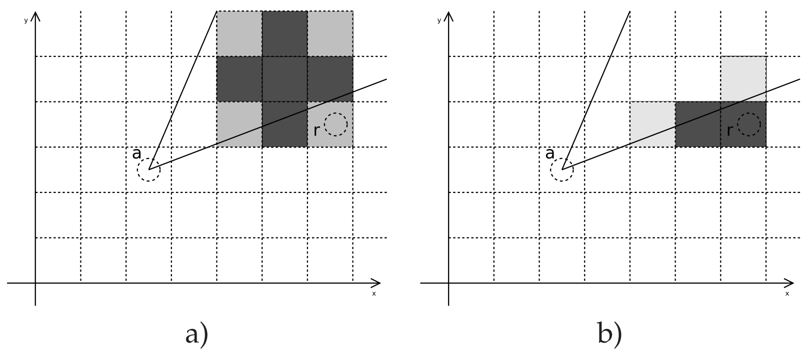


Figure 6.2: This figure shows Aalbot, identified by *a*, and robot *r*. In a) Aalbot moves its radar and scans an area where it is highly probable that *r* is positioned However *r* has moved to an area not scanned in this turn. The updated target map is shown in b).

6.2 Arbitrator Component

The task of the arbitrator component is to perform the actuator activations corresponding to the actuation wishes in the wish list from the subsumption layers. While doing this the arbitrator component has to be aware of possible conflicts in the actuation wishes. To simplify the task of identifying conflicting wishes, the actuation wishes are separated into 5 different types:

- **Robot Motion.** Forward and backwards movement of the robot.
- **Robot Turning.** Turning the robot clockwise and counter-clockwise.
- **Gun Turning.** Turning the gun clockwise and counter-clockwise.
- **Gun Firing.** Firing the gun with a given power.
- **Radar Turning.** Turning the radar clockwise and counter-clockwise.

Conflicts can only occur between wishes of the same type and the arbitrator solves some conflicts through priority, i.e. the layer number from which the wish originated. Thus a basic solution can be found by simply sorting the wishes by type and sorting wishes of the same type by their priority. The wishes with the maximal priority of each type are then the only ones left to be considered. From this reduced set it is trivial to decide the wish to be executed if there only exists one wish of this type. It is however not straightforward to decide which wish to select if there exists more than one wish of a certain type (e.g. TargetWish), after the initial reduction. In fact since the wishes originated from the same layer, there is no way for the arbitrator component to distinguish between them. To solve this dilemma the arbitrator simply selects the wish that was generated first.

Chapter 7

Module Design

In Chapter 5 an architecture serving as the basic framework for Aalbot was defined. In this chapter the behaviour controlling modules used in Aalbot's subsumption layers are defined, and the methods to be used for the implementation of the individual modules are chosen.

The necessary basis for choosing methods is given in the first section, where problem models and problem solving methods are discussed in a general setting. The structure of the following sections reflects the grouping of modules into four categories, namely:

- Percepting
- Planning
- Moving
- Targeting and Shooting

These four types of modules collectively allow for a robot to be built with a complete behaviour capable of participating in a Robocode match, as identified in Section 1.2. The chapter concludes with a choice of priority and placement of the selected modules into the subsumption layers.

7.1 Necessity of Machine Learning

Faced with the problem of programming a robot to play and win the game of Robocode it is interesting to look at several possible methods to achieve this goal. As seen in Section 1.2 and Chapter 2 existing attempts at solving the problem involve both traditional "hand-coded" methods as well as machine learning. In this chapter the key difference between hand-coded strategies and machine learning is discussed. Furthermore some guiding rules for selecting the best method for solving a given problem are given.

7.1.1 Problem Models

Solving a real-world problem like calculating the strength of wires in the construction of a bridge cannot be directly solved by a computer, as that would require an omniscient computer. Instead the computer must work on a *model* of the problem to be solved. This model can be simple and very general, or it can be highly complex and tailored for one specific application – but still it is not the actual bridge the computer interacts with, its calculations are based on an idealised model of the bridge. A good model should ensure that results obtained using the model are still applicable in the real world in the majority of cases. Rare cases may exist where the assumptions of the model do not hold; these cases should ideally be few.

A model of a problem is typically formalised as the calculation of a certain mathematical function. For example, if the problem at hand is to determine whether or not a person is eligible for a loan, then the formalised model of the problem is a function that given the person's age, yearly income, seniority, etc. calculates a truth value: True meaning that the person is eligible, and false meaning that the person is not eligible.

Many different such functions, each called a hypothesis, can be considered a model. Some might yield wrong results in all cases, others might be approximately correct and again some hypotheses can be completely correct. Problem solving boils down to finding a correct hypothesis.

7.1.2 Properties of a Good Model

An important observation is, that the choice of domain and codomain for the hypothesis to be found is a part of the model of the problem. Using the same example as before, imagine that the person's yearly income was not part of the input to the hypothesis function; then it could be impossible to find any function that will output correct answers simply because insufficient data is available. On the other hand, having too many possibly irrelevant inputs would overly complicate the model and make it harder to distinguish the useful data from the useless.

The other aspect of problem solving is modelling the structure of solutions. The hypothesis functions that are possible solutions can be defined by for example a single real number, a vector of binary values, or very generally, as a computer program written in some programming language. The chosen structure defines the hypothesis space[Mit97, chapter 2], which is the set of all possible hypotheses for the model. A simple hypothesis space is to be preferred, as this minimises the number of wrong hypotheses. Compare it to finding a needle in a haystack: The smaller the haystack is, the easier it is to find the needle. On the other hand, if the hypothesis space is too narrowly defined, it might not be possible to express a solution to the problem within the hypothesis space; no correct hypothesis lies within the hypothesis space. This means that a simpler hypothesis space should be preferred only as long as the best hypothesis in the simple hypothesis space is at least as good as the best hypothesis in the complex hypothesis space.

This phenomenon is captured in "Occam's Razor"[Mit97, p. 65], which can be formulated as:

> "Prefer the simplest hypothesis that fits the data."

That is, in striving to build a good model of the problem setting, it is important to find the simplest model that still provides enough information about the real problem to be able to deduce the correct answers in the majority of cases. The correctness of this principle has been debated for centuries and still remains unresolved[Mit97, p. 65-66]. As such it is important to determine its applicability to a certain problem setting before using it.

To define what is meant by a *simple* hypothesis or a *simple* model, [Piv00] defines different types of *simplicity*. The relevant types are summarised below, and can be used to analyse the results of implemented machine learning algorithms:

- **Structural Simplicity**. A simple model is one with a simple mathematical structure. E.g. one where the dimensionality of the hypothesis space is low. For example in neural networks, the simpler structure is the one with fewer weights. This is preferred as too many weights increases the risk of overfitting.
- **Descriptive Simplicity.** A simple model can be described concisely. In genetic programming this could indicate that smaller individuals are inherently better than larger individuals (given that both individuals have the same fitness value).
- **Computational Simplicity.** The simplest model is the one that requires the least amount of computational effort to predict phenomena. For example comparing two similar performing individuals evolved using genetic programming, the one requiring the least amount of execution time would be preferred.
- **Predictive Conservatism.** The model that claims the fewest correlations between events is the simplest one.

To justify Occam's Razor, [Piv00] gives a pragmatic and an epistemological argument. The pragmatic being, that simpler models are better because they make prediction tasks more straightforward, they make it easier to communicate the model and interpretations of the model, they involve less error-prone computations, etc. The epistemological argument says that a simple model is more likely to be true, by virtue of its simplicity.

The applicability of those arguments to the area of machine learning is not entirely without problems. For example a neural network is inherently hard to understand for a human observer — getting from the weights in the network to reasoning about the actual problem is often impossible. It can seem like black magic, but yet neural networks works in some cases. The model is neither easy to communicate or interpret, however it has computational simplicity. Similarly, genetic programming uses individuals with a relatively complex mathematical structure, namely a computer program. On the other hand, computer programs can be described quite concisely and in most cases interpreted quite easily. Given a computer program that solves a specific problem, it is often possible to reason about how it works.

7.1.3 Problem Solving Methods

Given a proper model of the problem setting, a correct or approximately correct hypothesis can be found using several methods. The main approaches considered in this project are analytical problem solving as used in the "hand-coded" strategies, and computer-assisted problem solving.

Analytical Problem Solving

Through careful examination of the problem and its surroundings, perhaps using experiments or logical deductions from already known facts it can be possible to solve a problem completely using analytical tools. This often requires a lot of a priori knowledge about the problem setting and structure. An example of this type of problem solving could be the problem of finding the distance between the two points, (x_1, y_1, z_1) and (x_2, y_2, z_2) in a Euclidean three-dimensional space.

A person with a mathematical background can easily solve this problem, and determine that the solution must be of the form:

$$f : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}$$

And that the correct solution is:

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

A person working without this a priori knowledge about geometry would probably have a tougher time finding the correct solution to the problem.

However, analytical problem solving has many advantages. Due to the fact that solutions are based on already established knowledge, the analytical approach implies that it is possible to reason about *why* a solution is correct, and in some cases solutions can be proven correct. It is often also possible to determine which assumptions the solution is based on, and thus clarify if there are any extreme cases in which the model does not hold.

Computer-Assisted Problem Solving

Computers can assist in problem solving by searching for the correct solution amongst a number of possible hypotheses in a hypothesis space. Normally this is done by having the programmer define a hypothesis space and a algorithm for searching through all or a part of this space. In some cases, the hypothesis space is not entirely specified in advance by the programmer; instead this is also learned or optimised by the problem solving algorithm. This can for example be a method where the structure of neural networks are developed using genetic programming, as in [Mit02, p. 70-76].

In addition, it must be specified how the computer can determine whether a given hypotheses is "good or bad". This specification is normally referred to as a *fitness function* or an *evaluation function,* which is structured as follows:

$$f \,:\, H \to \mathbb{R} \ , \ \text{where } H \text{ is the hypothesis space} \tag{7.1}$$

A hypothesis, h, is then preferred to another hypothesis, j, if $f(h) > f(j)$. In some cases, the fitness function is defined so that the hypothesis h is preferred to the hypothesis j if $f(h) < f(j)$.

The need for a fitness function restricts the applicability of this type of computer-assisted problem solving to settings where it is possible to relatively easily determine whether a given hypothesis is a good one, without being able to deduct the correct solution from this knowledge. The constructed fitness function ought to be computationally cheap.

Often in machine learning methods a set of training data is collected, i.e. a list with an input for the hypothesis function and the corresponding correct output value. The fitness of a particular hypothesis is then judged by testing it with each training example and estimating how closely the output resembles the known correct output. In game settings it is often possible to determine the fitness of an hypothesis by using the hypothesis to play the game against another computer-controlled opponent and noting how often it wins.

Below several different algorithms for searching through a hypothesis space are briefly described and their advantages and disadvantages are summarised.

Exhaustive Problem Solving

Given that the hypothesis space and the form of the solution function is known in advance, it is in some cases possible to perform an exhaustive or brute-force search.

An example of an exhaustive search could be determining whether or not a piece of sensitive equipment has become too hot. Given a temperature as an integer value, the problem is to determine whether this is "too hot" or "okay". The form of the solution is then:

$$g : \{0, 1, \ldots, 99\} \rightarrow \{\text{toohot}, \text{okay}\}$$

It is known in advance that there are two temperature intervals, one which is too hot, and one which is okay. Also the temperature sensor can only report integer values in the interval $[0; 99]$ degrees. Therefore the simplest representation of the sought function is an integer threshold value. The hypothesis space is then finite and sufficiently small to perform an exhaustive search; simply try all possible hypotheses in the hypothesis space and determine the fitness value for each. This gives 100 different values and the best function is then the one with the optimal fitness value, i.e. the highest or lowest fitness value depending on the definition of the fitness function.

The advantage of the exhaustive search is the guarantee to find the model corresponding to the global extremum of the fitness function. The disadvantage is that in praxis it is only possible to try all hypotheses for small hypothesis spaces, which is normally not the case. With too many possible hypotheses, the time spent trying them all one after one will be prohibitive. With an infinite hypothesis space, the exhaustive search will never complete.

Neural Networks

Another method for searching through a hypothesis space is the use of neural networks trained using the back-propagation algorithm. In the case of neural networks, the hypothesis space is defined by the structure of the network. A single hypothesis consists of the values of all the weights in the network. As noted in Appendix A, the backpropagation algorithm searches the hypothesis space in a hill climbing (or rather descending) manner by following the gradient.

According to [Mit97, p. 85], neural networks trained using back-propagation are advantageous in settings where:

- Training set instances are represented by many attribute-value pairs.
- The hypothesis function output may be a vector of several real- or discrete-valued attributes.
- The training examples may contain errors.
- Longer training times are acceptable.
- Fast evaluation of the learned target function may be required.
- The ability of humans to understand the learned target function is not important.

Contrary to the exhaustive search, it is possible for neural networks to get "trapped" in a local minimum during training.

Reinforcement Learning

In reinforcement learning the sought hypothesis is a Q table which is used to determine an optimal policy, as described in Appendix B. Thus the hypothesis space is the set of all such tables. As noted in [Mit97, p. 367], reinforcement learning is appropriate to problems where an autonomous agent, which senses and acts in its environment, must learn to choose optimal actions to achieve its goal. This description is clearly very appealing in the Robocode scenario, where Aalbot is the agent and the environment is the battle field and the enemies.

In order to apply reinforcement learning, it must be possible to define an appropriate and limited number of states in which Aalbot can be. Rewards which acknowledge or disacknowledge actions taken by Aalbot must also be possible to define, and these should be delayed in time.

Genetic Programming

In genetic programming the hypothesis space is defined by all possible programs which can be represented using the chosen syntax. The search proceeds by evaluating an entire population of hypotheses, which is typically spread throughout the hypothesis space. This approach can be considered a compromise between exhaustive search and the search performed by neural networks, as the latter only maintains a single hypothesis at a time.

In theory, the set of solutions solvable by genetic programming is a superset of the solutions which can be solved by any other machine learning method [Ben98, p. 22]. The intuition is that every machine learning method is implemented as a computer program, which hence is contained in the hypothesis space of a genetic programming search. Therefore genetic programming can in principle be chosen for modules in Aalbot in which neither neural networks nor reinforcement learning are suited. However for genetic programming to be applicable in praxis and in the time frame of this project, it must be possible to chose a well defined fitness function and a delimited syntax for the language to be evolved.

7.2 Percepting

Perception is the functionality of the robot that transform immediate sensor inputs into manageable information. It uses the relatively simple sensor inputs to update more advanced data structures continuously. Examples of valuable information that could be collected through perception are the locations of enemies, possible locations of bullets in flight, the movement patterns of enemies, etc.

The sensor interpretation module is responsible for updating the world state, whereas the radar control module is responsible for rotating the radar in an useful fashion.

7.2.1 Radar Control

As described, the problem of determining an optimal scanning strategy is a complex problem. Section 1.2 describes a few hand-coded scanning strategies used in typical robots. Both the basic scanning strategy and the alternate direction scanning strategy have pros and cons, and it is not easy to determine which of the two would be best in Aalbot, thus we have chosen to use machine learning to develop the radar control module.

It is intuitively a good idea to let the radar control module take advantage of the fading memory target map, to optimise its scanning strategy. Many other types of information are of possible use to the radar control module, for example movement histories of enemies, estimates about the gun heat of enemies, the energy level of enemies, the position of Aalbot itself, knowledge about which enemy is the currently selected target, etc. It is not easy to determine the usefulness of each type of information in advance by analytical methods. In addition the encodings of these inputs for a neural network or for reinforcement learning would be large and difficult to handle. Therefore genetic programming is chosen for the radar control module.

7.2.2 Sensor Interpretation

The sensor interpretation module uses the sensor input in each turn to update the world state component. This includes the enemy information table and the fading memory target map. This is done by transforming the events where Aalbot scans an enemy robot into abstract wishes to update the world state component.

Due to the straightforward functionality of this module, this module is chosen to be handcoded.

7.3 Planning

A *planning* robot is capable of making deliberate choices internally that affect its future actions. That is, given the internal world state model, it decides on specific behavioural patterns to exhibit in the following turns. This can be relatively simple choices, like choosing which robot to target for an attack, or it can be more advanced choices that involve choosing an overall strategy for the whole game.

The concept of planning is relaxed a bit here to accommodate the nature of the Rocobode game. Specifically it is hard to let the robot do projection, that is, having it understand which effects on the environment each possible action will have. This is both because of the sheer number of possible actions, as well as the fact that the actions of enemies are not easily predictable, nor is the game fully deterministic. Because of this only a target selection module is employed.

7.3.1 Target Selection

Choosing the right target is crucial for success in Robocode. It is easier to kill some robots rather than others – some may be very advanced robots, some may be dumb robots, and some robots have large amounts of energy, others are near death.

A number of parameters describing partial world state can be used to determine the enemy that it would be most fruitful to target. Choosing the most fruitful targets throughout the game means that the choices are optimal for winning the game overall. The opposite, namely that winning the game means that the most fruitful targets have been chosen, is not necessarily true. It is possible to select suboptimal targets and still win the game by luck. The parameters to be considered could be:

- Number of enemies
- Enemy distance
- Enemy position
- Enemy energy level
- Enemy velocity
- Enemy heading
- Probability that we can aim correctly at each enemy

The information in the last item, the probability that we can aim correctly at a specific enemy, is obtained from the targeting module explained later in this chapter.

The analysis shows that several analytical methods for target selection have been tested, including selecting the nearest enemy, selecting the weakest enemy - or alternatively selecting the enemy based on a ratio between distance and energy difference. No single method is generally accepted to be the optimal method. Therefore machine learning will be applied to this problem.

As a single robot takes up several pixels on the gaming area, it is possible to segment the above mentioned position inputs into manageable categories. Similarly the energy levels and probabilities can be segmented into quite few segments. Therefore the number of possible permutations of these input parameters are greatly reduced: this means that reinforcement learning can potentially be used to solve the problem, as the number of states would be manageable.

In addition it is possible to define rewards that can easily be justified – such as hitting a target, destroying a target and winning the game.

These two assumptions together leads to the choice of reinforcement learning for developing a target selection policy.

7.4 Moving

Proper robot movements are vital in ensuring survival and success. Contrary to classical movement problems described in the literature, such as the block stacking problem[Mit97, p. 263-265] or path finding, the robot cannot easily think ahead to determine how its movements will affect the game due to the other robots. Therefore it is neither important nor feasible for a robot to construct movement plans covering many turns into the future as this future is very uncertain. Instead the movement modules should focus on reactive "real-time" decisions.

The modules in this section all rely on well established, non-machine learning methods. However a movement module using genetic programming will also be designed in order to investigate if this machine learning method can produce equally good results.

7.4.1 Fading Memory Anti-gravity Movement

Anti-gravity, identified in Subsection 1.2.3, is a technique for deciding robot movement. It has been used successfully in many high-end robot variants, and therefore this method is also adopted for Aalbot. That is, movement will be hand-coded using the analytical approach described below.

Inspired by Newtonian physics, the robot maintains a map of the gaming area with a number of attractive or repelling gravity points. Each gravity point has a specific location on the gaming area, and radiates force of a specified strength. The impact of a force on the robot diminishes by the square of the distance from the origin of the force to the robot. Deciding the movement direction of the robot is simply to sum up the contributions from all the forces as vectors, and moving the robot in the direction indicated by the resulting vector.[Owe02]

For use in Aalbot the anti-gravity technique is expanded with "fading memory", i.e. the force of a gravity point diminishes over time.

Mathematically, a single gravity point, GP, can be described as follows:

GP is a 4-tuple, (x, y, s, t), where (x, y) is a position on the gaming area,
$s \in [-1; 1]$ is the strength of the force
and $t \in \mathbb{N}$ is the time the point was placed

The strength of a specific gravity point, (x_i, y_i, s_i, t_i), observed at the robot at time t_n is then:

$$\begin{aligned} s_{i_{observed}} &= \frac{s_i}{(\Delta t + 1)\left(\sqrt{(x - x_{robot})^2 + (y - y_{robot})^2}\right)^2} \\ &= \frac{s_i}{(\Delta t + 1)\left((x - x_{robot})^2 + (y - y_{robot})^2\right)} \end{aligned}$$

where x_{robot} and y_{robot} denotes the position of Aalbot and $\Delta t = t_n - t_i$ is the age of the gravity point. This specifies that the absolute strength of a gravity point diminishes with the square of the distance between the gravity point and the robot times the age of the gravity point. A negative strength means an attractive force, whereas a positive strength yields a repelling force.

The force on the robot given by a single gravity point is then expressed as a vector as follows:

$$F_i = \begin{pmatrix} s_{i_{observed}} \cdot \sin\phi \\ s_{i_{observed}} \cdot \cos\phi \end{pmatrix}$$

where ϕ is the angle between the robot and the gravity point relative to the y-axis. Note that the order of sin/cos is the reverse of standard mathematical coordinate systems due to the orientation of the coordinate system in Robocode.

Every gravity point in the system affect the movements of Aalbot. The total force on Aalbot is:

$$F_{total} = \sum_{i=1}^{n} F_i$$

Aalbot can then be instructed to move according to the F_{total} vector.

Other modules can send abstract wishes to the anti-gravity module asking for it to add or delete gravity points to the map.

7.4.2 Wall Avoidance

Part of the Robocode rule set is that the robot is penalised for ramming into a wall. This makes it attractive to build in automatic wall avoidance, i.e. taking evasive actions if the module detects the robot being near a wall. This is done by asking the anti-gravity module to place static repelling gravity points along the walls at each pixel.

The location of the walls is known in advance, so this module will be handcoded.

7.4.3 Randomisation

To avoid that Aalbot becomes a deterministic robot, the choice of adding a randomisation module is made. The module is invoked when it receives an abstract wish from another module; it will then place a very strong gravity point to ensure a sudden movement in a random direction. Due to the simplicity of this module, it will be hand-coded.

7.4.4 Bullet Dodging

As identified in Subsection 1.2.3, an effective bullet dodging method has been analytically defined. Therefore it is chosen to hand-code this module instead of using machine learning.

The module must be able to employ a heuristics to determine whether or not opponents have fired a bullet or not. A drop in energy below a threshold of 3 could mean that the opponent fired a bullet, but this is not necessarily the case. The module must somehow "guess" whether or not this is the case.

Once this module detects the firing of a bullet, it notes the position from where the bullet was fired. Since there is no way to detect the bullet's direction, a bullet is represented by a *virtual wave front* (a circle). In every turn, the circle radius is updated to be equal to the expected distance travelled by the bullet. When the circle is close to the robot's current position, a virtual wish for randomised movement is added to the wish list.

7.5 Targeting and Shooting

Important to most attacks is the successful use of the gun on the robot. In particular it is important to be able to aim correctly so that the chance of hitting the enemy is optimised.

This module should output for each enemy the angle to rotate the gun turret, the shooting power as well as a probability stating how likely it is that Aalbot will be able to hit this enemy. This last piece of information is required by the target selection module.

The analysis discovered that a form of machine learning, pattern matching, had successfully been employed to handle targeting. No analytical approaches with as good results in the general case as the pattern matching method was identified. Therefore machine learning has been chosen for the development of a targeting module.

Prediction of future enemy positions must be based on a history, where possible parameters include:

- The location of Aalbot

- The location of the enemies
- The heading and velocities of enemies

These inputs are few and simple to encode as real numbers. Also because the module must calculate angles and shooting strength for all enemies every round, it is important that this is relatively, computationally cheap. Finally the mentioned input parameters are not necessarily recorded with the same time interval, so noise may be present.

Neural networks work well for problems with such characteristics[Mit97, p. 85]. In addition, the analysis described how neural networks had earlier been successfully applied to this area, therefore the choice of using neural networks for this module is made.

7.6 Module Priorities

Nine modules have been identified for placement in Aalbot's subsumption layers, and machine learning has been chosen for three of these. Neural networks are used for targeting, reinforcement learning is used for target selection and genetic programming is used for radar control.

Figure 7.1 illustrates the placement and priority of the chosen modules; higher layers are given higher priority, and the top layer is the interface to the arbitrator component. In cases where modules are independent, they have been placed in the same layer. The sensor interpreter and target map modules are responsible for updating the world state module, and must therefore be executed in the first layer, prior to the remaining modules. Target selection depends on the wish list from the targeting module, hence the relative placement of these two modules in the second and third layers. Modules for movement, radar control and bullet dodging follow in the fourth and fifth layers – these may depend on the selected target. The randomisation module is given highest priority in the sixth module.

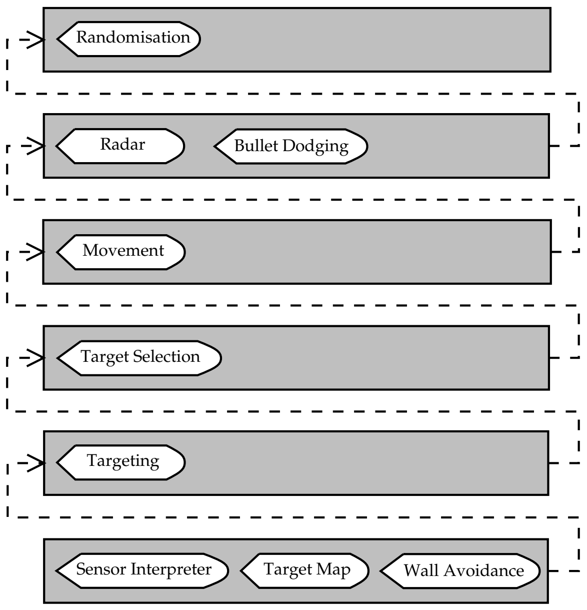


Figure 7.1: The placement of modules in Aalbot's subsumption layers.

Chapter 8

Neural Network Module

The targeting module has been selected as a candidate for neural network control, which is the topic of this chapter. Focus will be on training a network to target a single robot, since in the case of melee battles multiple networks can be employed, one for each enemy. The target selection module would then be responsible for selecting the appropriate enemy to be pursued. In terms of the Aalbot framework, the output of the neural network module will be an abstract wish containing, for each enemy, the angle in which to turn the turret and the probability of a successful hit. The wish is the fulfilled by the target selection module, which chooses one of the given angles in which to fire the gun.

Neural network targeting design is divided into three steps. First a suitable target function is chosen in a top-down manner by considering which Robocode game parameters are relevant and how these can be encoded as network inputs and outputs. Secondly a data preprocessing scheme is derived based on the chosen target function, and finally network training is discussed. The reader is referred to Appendix A for an introduction to neural networks.

One overall design criteria will be guiding throughout the chapter, namely the *support of online training.* Relying solely on offline training would require extensive training time, because a network would have to be trained for all conceivable types of enemy robot movements. On the other hand, the requirement of online training will place some limitations on network and preprocessing complexity in order to deliver real time results. Therefore the criteria for *computational efficiency* naturally follows from the online training criteria. The extent to which online training is realistic, given the processing restrictions of Robocode, is hard to predict, but the possibility should nevertheless be pursued because of the substantial gain in robot adaptability.

8.1 Target Function Representation

Neural networks are applied to function approximation tasks, so the first step in designing a neural network for targeting is to define the domain and range of the function to be approximated. In other words the issues of *what to compute* and *which inputs are required* to achieve good results must be dealt with.

Because the ultimate task of the targeting module is to decide on the possible angles in which to move the gun turret before shooting, the neural network should ideally be able answer the following question:

Given all relevant game parameters, in which direction should the turret point in order to hit the enemy, and with which power should the gun be fired?

Here the relevant game parameters amount to the parameters which influence the movement of the enemy in question as well as Aalbot's position. Let a game *configuration* be defined as the bullet speed, the positions of all robots, their heading and their energy at a given time. In the most difficult case, the enemy's movement at time t will depend on t such configurations, so the number grows linearly in time. Additionally the number of variables in a configuration gives rise to a large amount of possible configurations.

Hence the task is to select a limited number of parameters which are likely to have the largest impact on enemy movement, a process sometimes referred to as *feature selection[Chr95, p. 2].* The next subsection deals with this problem with offset in the analysis, and is followed by a discussion on how to encode network outputs. The section concludes with a summary of the possible target functions to be tested empirically.

8.1.1 Input Parameter Selection

As a first step in reducing the number of game parameters to be considered, assume that the movement of a robot depends primarily on the robot's own movement history. This is certainly the case for many simple robots, e.g. a robot which always moves around the battlefield while sticking to the edges (take the `Walls` robot of the `sample` package as an example). But even if the movement of a target robot depends on other robots, parts of this dependency will be encoded implicitly in the target robot's movement history – consider as an example a robot which always moves away from its enemies. Hence the assumption seems reasonable.

Recall from Subsection 2.2.1 on previous work that the representation of a target robot's history can be based on a fixed size sequence of heading/velocity pairs:

$$s = (h, v)_1, (h, v)_2, \cdots, (h, v)_n$$

Is this history representation suitable, and is the history sufficient for good prediction of future positions?

In answering the first question one problem is apparent: prediction cannot take into account the target robot's position in the battle field. While this simplifies the model, many robots will move differently in the centre of the battle field than at the corners. For instance a robot might be prevented from repeating its movement pattern when in danger of colliding with the battle field walls. One solution is to augment the history with the target robot's initial position in Cartesian coordinates (x, y). Alternatively the heading/velocity representation can be altogether replaced with (x, y) pairs.

Another potential problem with the heading/velocity history representation is that no time parameter is taken into account. The absence of time is reasonable if there are strict upper and lower bounds on sampling delays, i.e. the time between successive radar sweeps of the target robot. In one-to-one battles this is often the case, whereas in melee battles the radar must cover more than one enemy. Furthermore in the Aalbot architecture several modules might be competing for radar control, and compromises have to be made. Consequently the time parameter should not be left out entirely; instead it will be taken into account during the preprocessing phase, which is to be discussed in the next section.

So far only a suitable history has been considered. The need for further input parameters will depend on the desired network output. In Subsection 2.2.1 the network output was simply the heading/velocity following the last element in the input history (after a fixed time period). Several feed forward cycles were then required, and some clever postprocessing was needed to predict the actual angle in which to point the turret. Alternatively the network could directly output the angle in which to point the turret given appropriate input data. Figure 8.1 illustrates the information required, namely the target robot's movement history, Aalbot's position and the velocity with which the bullet is to be fired (recall that bullet velocity depends on the power with which the bullet was fired). Hence bullet velocity must be determined in advance, and for simplicity this is assumed constant. The collection of training data for the angle prediction needs to be tailored for this approach to work, an issue to be discussed further in Section 8.3.

These added parameters will place increased demand on the amount of training data, and additional time and processing resources may be required to harvest a single training example. On the other hand, only one feed forward cycle will be needed for prediction, thus eliminating the inevitable accumulation of errors caused by multiple feed forward cycles[Chr95, p. 303]. Furthermore when receiving the turret angle directly as network output, processing resources are not needed to reconstruct the enemy's expected path.

8.1.2 Network Output Encoding

Several possibilities for network output values were discussed in the previous section, but the issue of how to encode these values remains unresolved. All options share the characteristic that they require quantitative valued outputs, such as an angle or a pair of (x, y) coordinates. Approximation of quantitative valued functions is in the literature referred to as *regression* [Chr95, p. 5].

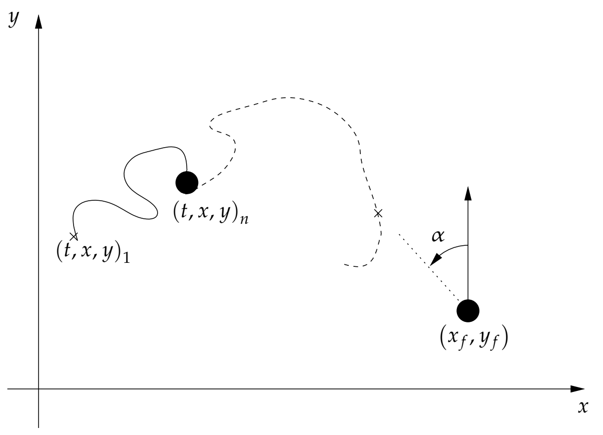


Figure 8.1: Illustration of the required input parameters when using the neural network to predict the angle in which to point the turret. The filled circles represent the enemy robot (left) and Aalbot (right). The solid curve is the observed history of the enemy, while the dotted curve is the expected future path of the enemy. α is the network output, indicating the angle that the turret should point relative to the y axis in order to hit the enemy on its future path. Note that time is not explicitly depicted.

The obvious approach would be to simply use these quantitative values directly as target outputs when training the network. However this makes it difficult to asses how certain the network is of its output: while the outputs are within reasonable bounds (e.g. an x coordinate should be within the width of the battle field), there is no reason to doubt the result. It would then be impossible to contribute to target selection based on network certainty.

Instead categorical target values could be chosen, where one output neuron generally represents one *category*. In the case of Robocode, an x coordinate holding values in the range $[1 \ldots 800]$ could be encoded categorically by 800 output neurons. If for instance the value of x is 5, neuron five should output the value 1, while the remaining 799 neurons should output the value 0.

In the general case consider a computed output vector, $\overrightarrow{o}$, of dimension N. The value to select will be the vector index, n, of which the element o_n has the highest value. The error function (A.2) defined in Appendix A can then be used to express the certainty of output $\overrightarrow{o}$ in reference to a target, $\overrightarrow{t}$, of which the n'th component is 1 and the remaining components are 0. Smaller errors would then indicate higher certainty.

However there is a problem with this approach, namely that the error is difficult to relate to certainty – how low should the error be for Aalbot to rely on the network result? Ideally each output value should be interpreted as the *probability* that the associated category holds, i.e. the output values should sum to 1. This property can be achieved by using the *softmax*[Chr95, p. 238] activation function in output layer neurons:

$$\sigma\left(y_k\right) = \frac{exp\left(y_k\right)}{\sum_i exp\left(y_i\right)}$$

where y_k is the activation function input for output neuron k.

Clearly, using 800 output neurons to represent one x value is out of the question because of the efficiency criteria. However this number can be reduced by letting each output represent a range of e.g. 10 pixels, reducing the number of output neurons with a factor of 10. Relative to the dimensions of a robot at approximately 36×45 pixels, this figure is reasonable. Whether the categorical output encoding should be pursued depends heavily on the outputs to be represented and the number of feed forward cycles required for a single prediction. If multiple feed forward cycles are required, the error accumulated by letting each neuron cover a range of values is probably unacceptable. On the other hand, if the network uses a single feed forward cycle to output an angle in which to point the turret, categorical encoding is feasible.

8.1.3 Summary

A definitive choice of game parameters to include as network input and how to encode the target output is difficult to make. The preceding subsections have argued for different target function representations which are likely to provide good results. However one cannot reason about whether the computational requirements to approximate these functions will allow for sufficiently fast response times in the context of Robocode. The only way to resolve this issue is therefore by empirical testing.

The following list summarises the target functions, o, to be considered in a test. Here $\overrightarrow{s}$ will denote the function domain vector (network inputs), while $o\left(\overrightarrow{s}\right)$ will denote the function range (network outputs). The time parameter is included, but will be dealt with in the preprocessing phase.

1. Predicting one-step-ahead velocity/heading pairs based on a history given by time, heading and velocity:
   $\overrightarrow{s} = (t,h,v)_1, (t,h,v)_2, \cdots, (t,h,v)_n, t_{n+1}$
   $o\left(\overrightarrow{s}\right) = (h,v)_{n+1}$
   The last time parameter, t_{n+1}, refers to the time of the output heading/velocity pair $(h,v)_{n+1}$.

2. Predicting one-step-ahead (x,y) position based on time and coordinate history:
   $\overrightarrow{s} = (t,x,y)_1, (t,x,y)_2, \cdots, (t,x,y)_n, t_{n+1}$
   $o\left(\overrightarrow{s}\right) = (x,y)_{n+1}$

3. Predicting the angle in which to point the turret in order to hit the enemy on its future path. Input is based on a history given by time and Cartesian coordinates, as well as the position of Aalbot, $\left(x_f, y_f\right)$. The

output angle is encoded categorically into m outputs:
$\overrightarrow{s} = (x_f, y_f), (t, x, y)_1, (t, x, y)_2, \cdots, (t, x, y)_n$
$o(\overrightarrow{s}) = v_1, v_2, \cdots, v_m$

8.2 Preprocessing

Having identified possible target functions, domain/range data could in principle be used directly as network input/output. However this is not necessarily desirable, as a systematic processing of training data can significantly enhance the effectiveness of the neural network. In general the process of applying a neural network to a regression or classification task can be divided into three distinct phases: *preprocessing*, *network processing* and *postprocessing*. In the case of the Robocode movement prediction task, three transformations will be applied to training data in the preprocessing phase: missing time compensation, translation and scaling. These will be the main foci of this section.

The postprocessing phase simply involves applying the inverse transformation of the preprocessing phase, and hence will not be discussed further.

The transformations to be discussed are independent of how the movement history is represented; whether inputs are given by (x, y) coordinates or heading/velocity pairs does not matter. Similarly no distinction between input and output need be made. But because transformations on coordinates in the plane are most intuitive, data on the following form will be used as a running example in this section:

$$\overrightarrow{s} = (t, x, y)_1, (t, x, y)_2, \cdots, (t, x, y)_n \tag{8.1}$$

8.2.1 Time Normalisation

Ideally samples should be recorded with equal time delay, but as mentioned in Subsection 8.1.1, this cannot be assumed. Since robustness to noisy data is one of the strengths of neural networks, one option is to simply discard the time parameter. Another option is to preserve the time parameter, including it as input to the network. This would mean that for each sample, i, three inputs with the values t_i, x_i and y_i would be given.

Preserving the time dimension will however require both an increased number of training examples and an increased number of network weights because of the *curse of dimensionality*[Chr95, p. 7-9]. This principle states that the size of the hypothesis space increases exponentially in the number of dimensions added to the training data.

In support of the online training criteria, a solution should therefore be sought where time is not explicitly included in the training data, but where the data is

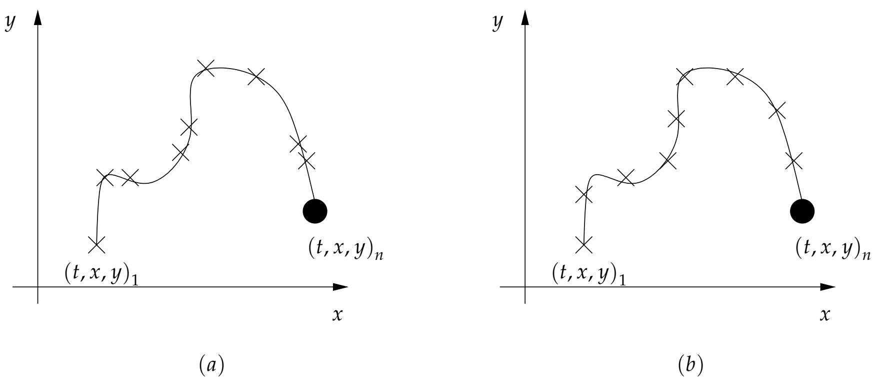


Figure 8.2: Training data before (a) and after (b) time normalisation in the case of an (x, y) based history. The solid lines represent actual movement paths, while the crosses represent sampling points. In (a) the time interval between sampling points varies, while in (b) it is constant. In both cases a constant speed is assumed.

processed to compensate for the varying delay between samples. The situation is depicted in Figure 8.2, where data is altered such that the time between each sample coordinate is constant. This transformation will be referred to as *time normalisation.*

The illustration is simplified by assuming constant speed of the enemy robot. However the speed is important to capture in the transformation because it varies over time – for instance a robot is forced to decrease its speed while turning. In such cases the distance travelled in between two samples is not necessarily constant after normalising time.

Details on how to calculate the time normalised coordinates will now be given. Using the notion in Equation 8.1, the total increase in time during sampling of history $\overrightarrow{s}$ is $t_n - t_1$, and the average time between samples is therefore:

$$\bar{\Delta t} = \frac{t_n - t_1}{n}$$

Let $\Delta t_i = t_i - t'_{i-1}$, $\Delta x_i = x_i - x'_{i-1}$and $\Delta y_i = y_i - y'_{i-1}$ where the prime indicates values after normalisation and $t'_1 = t_1$, $x'_1 = x_1$, $y'_1 = y$. Every coordinate $(x, y)_i$, $i \in \{2, \dots, n-1\}$ can now be replaced with coordinate $(x', y')_i$ as follows:

$$\begin{aligned} x'_i &= x'_{i-1} + \Delta x_i \cdot \tfrac{\bar{\Delta t}}{\Delta t_i} \\ y'_i &= y'_{i-1} + \Delta y_i \cdot \tfrac{\bar{\Delta t}}{\Delta t_i} \end{aligned}$$

Figure 8.3 gives the intuition in why these formulae work and serves as an example of how to apply them. Time normalisation thus results in a constant time between samples, and the time parameter can be removed entirely from

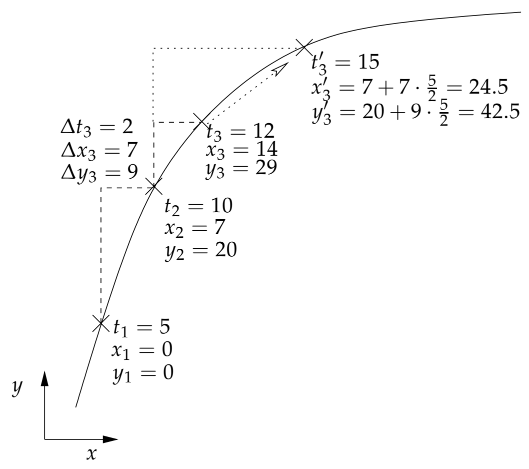


Figure 8.3: Example of how a coordinate is modified when normalising time. Each cross marks a sample $(t, x, y)_1$, $(t, x, y)_2$ and $(t, x, y)_3$. The average delay between samples is assumed to be 5, so $(t, x, y)_3$must be shifted up and to the right in order to capture the implicit constant delay, thus resulting in $(t', x', y')_3$.

the data set. Although the explanation of time normalisation has been grounded in (x, y) based histories, the method applies equally well to heading/velocity based histories.

8.2.2 Translation and Scaling

This subsection considers how translation and scaling of network data can be applied and why these transformations are desirable. Data is again assumed to consist of (x, y) coordinates, but with time removed by normalisation. A distinction between the preprocessing of data column vectors and row vectors has to be made, so let the training data be represented as a matrix:

$$M = \begin{bmatrix} x_1^1 & y_1^1 & x_2^1 & y_2^1 & \cdots & x_n^1 & y_n^1 \\ x_1^2 & y_1^2 & x_2^2 & y_2^2 & \cdots & x_n^2 & y_n^2 \\ & & & & \ddots & & \\ x_1^m & y_1^m & x_2^m & y_2^m & \cdots & x_n^m & y_n^m \end{bmatrix}$$

Each row in M corresponds to a single training example, where no distinction between input and output is made. A *variable* x_i is then defined to represent a value from the column vector $x_i^1, \cdots, x_i^m$, and y_i correspondingly represents values from the column vector $y_i^1, \cdots, y_i^m$. The scaling transformation to be introduced shortly operates on column vectors (and thus considers variables), while the translation transformation operates on row vectors (individual training examples).

In a more general setting, scaling and translation is often combined to perform *normalisation*[Chr95, p. 298]. Since this is the inspiration of the approach adopted in Aalbot, normalisation is briefly explained in the following. The n'th value of variable x_i, x_i^n, is normalised by subtracting the mean $\overline{x_i}$ of x_i and dividing with the standard deviation σ_i over x_i (i.e. all values in the column represented by x_i):

$$\tilde{x}_i^n = \frac{x_i^n - \overline{x_i}}{\sigma_i}$$

After applying normalisation, a variable has zero mean and a standard deviation of one, hence assuring that all input variables in the training data have the same order of magnitude.

Since normalisation is a linear transformation, one might argue that the network could deal with this itself if necessary – so why bother with this during preprocessing? There are two answers to why scaling is appropriate. First, the use of a particular activation function such as sigmoid requires that outputs are scaled to the range of the function prior to training. Second, consider the case where values of the input variable x_i is an order of magnitude 1000 times that of the values of y_j, and assume that weights are initialised to random values in the interval $[-1;1]$, which will be the case for the Aalbot network. There is no particular reason to believe that x_i should have more influence on network output than y_j, so it is likely that a lot of training cycles will be spend adjusting the weights from x_i to very small values. This problem is remedied by scaling [Chr95, p. 299].

Normalisation is not directly applicable in the case of Robocode. Because online training must be supported, the number of training examples grows over time, and the standard deviation would change with each added example. Repeating the preprocessing of the entire training set every time a new training example is added would be in breach with the efficiency demands.

Scaling is instead performed by dividing every value of variable x_i by the maximum possible value of the variable, which in this particular example corresponds to the width of the Robocode battle field (and equivalently for the y_i variables). Every variable will then have values in the same range $[0;1]$, which was the primary goal of applying scaling in the first place.

Translation involves adding a constant term to each value of the input/output data. More specifically the value x_1 is subtracted from each of the values $x_1 \cdots x_n$ and the value y_1 is subtracted from each of the values $y_1 \cdots y_n$. This translation has the effect that the history always starts at $(0,0)$, while the relative position between coordinates is preserved. Translated coordinates may thus contain negative values.

However the motivation for representing histories by (x, y) coordinates was that absolute battle field positions are of significance, and this information is discarded in translation. Hence translation on a per example basis is better suited for training data consisting of heading/velocity pairs: the heading and

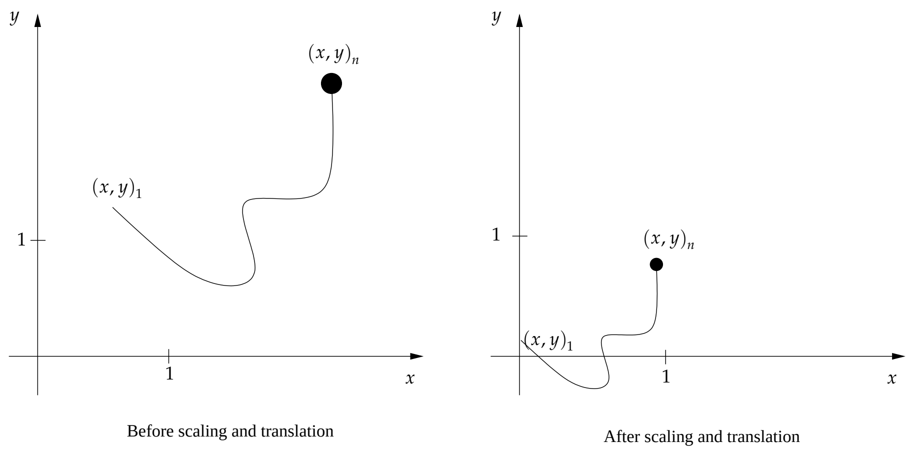


Figure 8.4: The solid curve traces the robot movement history before and after scaling and translation.

velocity of the first sample in the history is unlikely to have any impact on future values, so both heading and velocity can be translated. The number of required training examples is likely to be reduced if this kind of translation is applied, because the network does not have to generalise beyond examples which only differ by translation.

Figure 8.4 concludes this subsection by illustrating the result of applying translation and scaling.

8.3 Training

The previous two sections have focused on the choice of target function and how network input data can be preprocessed. In this section the chapter is completed with a brief discussion on which network topology and training algorithm to choose, as well as issues relating to data collection.

8.3.1 Training Algorithm

The backpropagation algorithm, based on gradient descent, is one of the most common algorithms for training neural networks [Mit97, p. 83]. Implementation is simple compared to other algorithms, and the results are generally good.

From an efficiency point of view, the basic version of backpropagation unfortunately performs rather bad[Mit97, p. 264-265]. One reason is that a suitable value for the learning rate, η, is difficult to choose, and that this value is constant during training – toward the end of training, a very small learning rate would be desirable, while at the beginning, the learning rate should be large

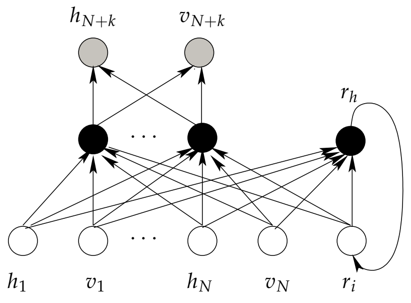


Figure 8.5: Single-step ahead neural network organised with a feed back cycle.

in order to allow for fast training[1]. Another reason for bad performance is the manner in which the search is guided along the gradient, which (informally speaking) follows a zig-zag pattern toward a local minimum. A number of alternative, more efficient training algorithms are described in [Chr95, Chapter 7].

In support of the online training design criteria, an efficient training algorithm should be chosen. However the increased complexity of advanced training algorithms requires larger effort for implementation, which does not fit the time frame of this project. Instead backpropagation is chosen, as this allows for use of existing programming libraries which have been thoroughly tested and are known to work well.

Two training parameters are directly related to the backpropagation algorithm, namely the learning rate, η, and the momentum, α [Mit97, p. 100]. In [Mit97, p. 115] the value 0.3 was used for both parameters, so values of this magnitude can be assumed to be appropriate for the Aalbot neural network. However tweaking η and α can result in better convergence during training, so more suitable values should be determined empirically.

8.3.2 Network Topology

The standard backpropagation algorithm operates on an acyclic network, i.e. there are no feed back cycles. Recall that the two first target function representations in Subsection 8.1.3 only predict target positions one step ahead in time, and hence multiple feed forward cycles need to be performed. This process was discussed in the analysis and depicted on Figure 11.3. However one problem is apparent, namely the accumulation of errors in each cycle. To overcome this, the concept of recurrent networks is investigated based on [Mit97, pages 119-121].

A recurrent network contains one or more feed back cycles. Figure 8.5 shows how a recurrent version of the network discussed in the analysis can be reconstructed by adding a feed back cycle from one additional hidden neuron, r_h, to

[1]In the specific scenario of Aalbot, a constant learning rate is still desirable because training is performed online and new examples are added incrementally.

one additional input unit, r_i. When the k'th feed forward cycle is performed, r_h will serve as a memory for the previous $k-1$ cycles. By inputting this memory into r_i, the k'th output will thus be influenced by the computations of all $k-1$ feed forward cycles and not just the output from the last computation. Hence a potential error in one feed forward cycle will not have as profound an impact on the final result when a recurrent connection is present.

A variation of backpropagation can be used for training a recurrent network, but the results are generally not very good: training is more difficult and networks do not generalise as reliably as standard acyclic networks[Mit97, p. 121]. Therefore this solution will not be pursued, although the method has some appeal.

A network with three layers (one hidden) will be used, as this has the potential of approximating any bounded continuous function (see Chapter 8); there is no reason to believe that this property does not hold for the sought target function. The number of input units and hidden units is very difficult to reason about, and hence these parameters will be left for empirical testing.

8.3.3 Data Collection

Possible approaches to the collection and management of training data is briefly discussed in the following. Data collection refers to the process of composing training examples for the target function representations which have been discussed earlier in this chapter. Data management in Aalbot concerns the selection of training examples from the total set of collected training data, which differs from approaches taken where offline training is employed.

Consider again the three target function representations chosen in Subsection 8.1.3. Collecting data for the first two is straight forward. A sequence of (t, x, y) or time/heading/velocity triples is recorded, and if the data is used for training, the last triple is used as the target output value.

Collecting training examples for the third case requires more consideration though. Given the enemy movement history, the position of Aalbot and an angle between the two, an angle in which to point the turret should be chosen. Using the concept of *virtual bullets* is one possible way of collecting training data on this form.

A virtual bullet is a bullet that is not actually fired, but its movement across the battle field is simulated. Simulation can be performed by calculating the bullet's position at each clock tick, and testing if it is within a certain radius of an enemy robot; if so, the bullet has hit. The simulation stops when the virtual bullet leaves the battle field boundaries. With regular intervals, the robot can fire a spread of e.g. 40 virtual bullets in all directions. The angle of the first bullet to hit the enemy (if any) can then be used as the target output value in a training example.

Simulating bullets in this manner is not optimal though, because a finite number of bullets must be maintained and the resolution (i.e. distance between bullets in a spread) decreases with the square root of the bullet's distance from its firing position. Another more analytical simulation can be performed by only maintaining a single, direction-less bullet, b, for every time Aalbot has an opportunity to fire, which simply holds the Cartesian position $(x, y)_b$ and time, t_b, at which the bullet was "fired". The speed, $v_{required}$, with which b should have been fired in order to hit an enemy, e, at its present position, $(x, y)_e$, and at time, t_e, can then be calculated:

$$v_{required} = \frac{\sqrt{(x_e - x_b)^2 + (y_e - y_b)^2}}{t_e - t_b}$$

Recall that a constant bullet velocity, v, was chosen when training the network based on virtual bullets; if $|v_{required} - v|$ is reasonable close to 0, the bullet could potentially reach target e with the given speed. The angle at which bullet b should have been fired can then easily be calculated from $(x, y)_b$ and $(x, y)_e$.

When training a neural network, the problem of overfitting must be addressed. In offline training, a static set of training examples are used repeatedly to train the network, and there is a risk that the network loses its ability to generalise. That is, it might not be capable of classifying examples which are not in the training set. In order to overcome this, a validation set is typically extracted from the training set, and with regular intervals during training, the network's error on the validation set is assessed. A sudden increase in validation set error is an indication of overfitting, and hence the training should be stopped[Mit97, p. 110-111].

Aalbot's targeting module is trained online, so a definitive validation set cannot be selected before training commences. Although training examples are added continuously, the risk of overfitting is still present if the earliest added examples are used over and over. In order to overcome this, only the N newest examples are used every time a training cycle is performed. A higher degree of adaptiveness is then achieved as well, because new examples indicating a change in enemy movement pattern will have greater impact on weight updates. On the other hand, the network will be more susceptible to noisy data.

8.4 Summary

The design of a neural network for targeting has been discussed in the preceding sections, and the main results are now summarised. In particular the choice of methods and parameters to test empirically will be stated.

Three target function representations were considered and preprocessing schemes appropriate for these were discussed:

1. Predicting one-step-ahead velocity/heading pairs based on a history given by time, heading and velocity. Time normalisation, scaling and translation are applied during preprocessing. Network outputs are encoded

quantitatively, i.e. one neuron representing predicted velocity and one neuron representing predicted heading. The tanh activation function is used in all network neurons, because outputs may be negative as a result of translating data.

2. Predicting one-step-ahead (x, y) enemy position based on time and coordinate history. Time normalisation and scaling is applied during preprocessing, but not translation; this would contradict the motivation of using Cartesian coordinates, namely that the enemy's absolute position on the battlefield matters. Network outputs are be encoded quantitatively, i.e. one neuron representing predicted x coordinate and one neuron representing predicted y coordinate. The sigmoid activation function is used in all network neurons.

3. Predicting the angle in which to point the turret in order to hit the enemy on its future path. Input is based on a history given by Cartesian coordinates, as well as the position of Aalbot, (x_f, y_f). The output angle is encoded categorically, and the softmax activation function is used in output layer neurons in order to interpret outputs as probabilities. The sigmoid activation function is used in the remaining neurons. All three preprocessing schemes should be applied, namely time normalisation, scaling and translation. The concept of virtual bullets is used for data collection.

In all cases network weights will be initialised to random values in the interval $[-1 \ldots 1]$, which is appropriate because network inputs are scaled to the same interval. An acyclic network structure with three layers (one hidden) has been chosen.

Four general parameters remain to be found in tests: the learning rate, η, the momentum, α, the number of input units and the number of hidden units.

Chapter 9

Reinforcement Learning Module

It was decided in Section 7.3 to apply reinforcement learning to the target selection module in Aalbot. This chapter documents the design of such a target selection module. Each section consists of an explanation of the ideas used alongside reasoning as to why, they are expected to perform successfully in a reinforcement learning context. The chapter builds on the basics of reinforcement learning introduced in Appendix B.

9.1 Optimisation Task

The task in reinforcement learning is to optimise the interaction with the environment, in this case Robocode, over time. The interaction between this module and the environment is given by a policy, π. Hence the optimisation task is to learn an optimal policy, $\pi^\star_{\text{Aalbot}}$.

Initially a discussion of the *goals* for a target selection module in Robocode will be presented, these goals will direct the choice of rewards to be used in learning $\pi^\star_{\text{Aalbot}}$.

9.2 Goals

One goal considered, but rejected, for the target selection module is: *"Select a target such that Aalbot wins the current round"*. Using this goal for the target selection module, a reward can be given whenever Aalbot wins the round.

This goal rests on the assumption that, in case Aalbot has won, it did so because its target selection during the round was optimal. But that does not necessarily hold. The possibility of Aalbot choosing a sub-optimal target, but still winning the round, e.g. due to very skillful targeting, movement or radar control, does exist. Because winning the round relies on all other modules of Aalbot, this choice of goal obfuscates the actual process of target selection, and complicates the assessment of the behaviour of this module.

If the only goal of the target selection module is to win the round, the first update of the Q-table can only be performed after a round has been completed. The time factor, and the above mentioned obfuscated target selection process suggests that a better solution should be pursued.

An alternative goal that has been chosen for the target selection module of Aalbot is: *"A robot, targeted by the target selection module, is killed"*. Using this goal for the target selection module, a reward can be given whenever the currently selected target is destroyed.

Consecutive achievements of this goal will ultimately lead Aalbot to win the round. Therefore, this goal can, in loose terms, be considered as a more fine grained version of the above goal, *"Select a target such that Aalbot wins the current round"*.

The fact that multiple robots engage on the same battlefield in Robocode, affects the clarity of this goal. E.g. the currently selected target may be killed by another robot. This can be considered noise in the training of the target selecting module, as the target selection module receives a reward even though it did not kill the target. The learning algorithm is expected to be able to handle such noise.

9.3 Rewards

The means to assess the policy have been chosen in the form of a goal. Still it remains to define the means to learn the policy, namely the rewards. This subsection defines the available rewards during the training of this module.

Initially, we consider an example of a flawed reward scheme. If one were to develop an entire Robocode robot by using reinforcement learning, one apparent goal to pursue could be *"Win the round"*. In trying to win the round, it seems like a good thing to ram other robots, as this produces bonus points, hence the naive designer chooses to reward ramming as well as winning the round. However, ramming causes damage to both of the involved robots. The robot, simply pursuing the highest accumulative reward, could then learn to pursue ramming as a subgoal without actually achieving the ultimate goal, namely to win the round. This example illustrate the fact that rewards should communicate to the robot *what* it should learn, and not *how* it should learn it.

Adhering to these considerations, and the goal selected in Subsection 9.2, rewards in the target selection module are only granted upon destruction of the currently selected target. The reward will be a fixed value r.

9.4 Data Representation

9.4.1 Input Data

The functionality of the target selection module is to choose a target among the remaining enemy robots. In order to choose such a target wisely, this module relies on two types of information: *Sensor input*, which gives information about the environment, and *internal information*, which consists of values calculated by the other modules of the robots. The chosen set of inputs defines the state space of Aalbot, which is the set of states that Aalbot can be in.

The *sensor inputs* to be considered could be the position of enemy robots (represented by Cartesian coordinates (x, y)), the energy of enemy robots, their heading and velocity, etc. Sadly, it is not possible to include every possible bit of information available to the module in the state space of Aalbot for reasons which will be explained in the following. In Subsection 9.5 the choice of sensor inputs is made.

The *internal information* relevant for this module is especially the information that comes from the targeting module. For each enemy, the targeting module calculates both an angle, which defines the direction in which to shoot, as well a probability that states how likely it is for the targeting module to be able to hit that particular enemy. This could easily turn out to be an important factor, so the target selection uses the probabilities $p_1, ..., p_n$ for hitting each of the other robots,$R_1, ..., R_n$ as part of the current state of Aalbot..

9.4.2 Output Data

As the architecture prescribes, the output of this module is a list of wishes. Those wishes can be abstract wishes to be fulfilled by other modules, or actuator wishes that are directly converted into Robocode actions. The target selection module is chosen to be the deciding module regarding aiming and shooting, therefore the module outputs a list of actuator wishes.

The output from this module is an actuator wish to turn the turret of the robot a specific angle in order to hit the selected target. The module also outputs a wish to actually fire the gun with a specified power. The angle and shooting power is part of the input from the targeting module, designed in Chapter 8, thus limiting the responsibility of this module to merely selecting a target enemy between viable alternatives.

9.5 States

At first glimpse it seems desirable to calculate Aalbot's current state simply by collating all incoming sensor inputs and the different values that p can maintain. The number of possible states for Aalbot could then be calculated in the following way by using the specified Robocode inputs[1], and the calculated probability p:

$$\begin{array}{rcc} n & = & \text{number of enemies} \\ pos & = & \text{position}\,\{0,...,x_{\max}\} \times \{0,...,y_{\max}\} \\ e & = & \text{energy level}\,[0;e_{\max}] \\ v & = & \text{velocity}\,[0;v_{\max}] \\ h & = & \text{heading}\,[0;h_{\max}] \\ p & = & \text{probability}\,[0;1] \end{array}$$

$$\text{Total number of configurations:}$$
$$((x_{\max}+1)\cdot(y_{\max}+1)\cdot\mid[0;e_{\max}]\mid\cdot\mid[0;v_{\max}]\mid\cdot\mid[0;h_{\max}]\mid\cdot\mid[0;1]\mid)^n$$

The main problem with the above is that the set of states becomes infinite since for example $p \in [0;1]$ is an infinite set. This is a problem because the convergence of Q-learning relies on each state-action pair being visited infinitely often[Mit97, p. 382], understood in the sense that over time each action in all states must be executed with nonzero frequency as the length of the action sequence approaches infinity. This is impossible if the number of states is infinite.

Therefore it is necessary to make some simplifying assumptions about the environment and behaviour of Aalbot. Sensor inputs must be chosen with caution, and then the job is to minimise the number of states, so that $\pi^{\star}_{Aalbot}$ can be learned.

9.5.1 Segmentation

The key technique employed is *segmentation*. This means to find a mapping between a large, possible infinite, set to a small set of *bins*. This way an infinite set such as the set of possible probabilities can be mapped to a set of three bins, {unlikely,likely,evident}, using a function like the following:

$$p_{\text{bin}}(p) = \begin{cases} \text{unlikely for } p < 0.4 \\ \text{likely for } 0.4 < p < 0.9 \\ \text{evident for } 0.9 < p \end{cases}$$

The function p_{bin} could have been defined in many other ways, so it is important to choose a good definition. What exactly constitutes a good definition depends on the type of input and what it will be used for. In some cases a high resolution (i.e. a large number of bins) is required, in other cases a low resolution (i.e. small number of bins) could be sufficient. To make a compromise between wanting a high resolution while still wanting to keep the number of

[1]These inputs are not the selected , but serves merely as an example of why not to use all possible sensor inputs.

bins low, it is possible to define the mapping function so that the number of bins for one part of the original interval is higher than the number of bins for another part of the interval.

The chosen mappings for the probability, energy level and distance are described in the following subsections.

9.5.2 Probability from the Targeting Module

As seen in the previous section, $p \in [0;1]$ has to be segmented into a finite, small set of bins in order to be used with reinforcement learning. By mapping each possible value of p into a value $p_{\text{bin}} \in \mathbb{N}$ in a small range, a useful outcome can be produced. By using the equation $p_{\text{bin}} = \lceil p * 10 \rceil$ the range becomes $\{1, 2, ..., 10\}$.

9.5.3 Enemy Energy Level

Aalbot receives a reward whenever its current target is killed, therefore it is desirable that the energy levels of enemies are considered when determining the current state. If the energy level is not segmented, it is a value $e \in [0;200]$. If all possible values had to be considered the total number of states would be infinite. There are two different approaches to alleviate this that seems valid:

The energy level of the enemy robot could be segmented in the same way as the probability calculation. The positive effect is that the number of possible values this binned parameter can take is 10 and not 200. On the other hand, this means that it cannot be known which target has the least amount of energy. It can only be known within which range it resides.

Another approach is to calculate the enemy's energy relative to Aalbot's energy, e_{Aalbot}. Then the bins could be defined as follows:

$$e_{\text{bin}}\left(e_{\text{enemy}}, e_{\text{Aalbot}}\right) = \begin{cases} 0 & \text{for } e_{\text{enemy}} < \frac{e_{\text{Aalbot}}}{2} \\ 1 & \text{for } \frac{e_{\text{Aalbot}}}{2} < e_{\text{enemy}} < e_{\text{Aalbot}} \\ 2 & \text{for } e_{\text{Aalbot}} < e_{\text{enemy}} < 2 \cdot e_{\text{Aalbot}} \\ 3 & \text{for } 2 \cdot e_{\text{Aalbot}} < e_{\text{enemy}} \end{cases}$$

By using this approach the number of bins is even smaller (4), but the major disadvantage is that it becomes impossible to point out the weakest enemy, since more than one enemy could have less than half of Aalbot's energy. Furthermore this mapping say nothing about how close to zero the other robots energy levels are.

This concludes that some vital information is lost no matter if the first or the second approach is chosen. Therefore a compromise between the number of states and loss of information has to be made. The mapping to be used maps the energy levels into 11 bins using the following function:

$$e_{\text{bin}}\left(e_{\text{enemy}}\right) = \begin{cases} 0 \text{ for } e_{\text{enemy}} \leq 5 \\ 1 \text{ for } 5 < e_{\text{enemy}} \leq 10 \\ 2 \text{ for } 10 < e_{\text{enemy}} \leq 20 \\ \vdots \\ 10 \text{ for } 90 < e_{\text{enemy}} \leq 200 \end{cases}$$

The bins are chosen so that Aalbot has more fine grained information when the energy of the enemy is low. With higher energy levels, Aalbot uses a lower resolution.

9.5.4 Distance to the Enemy

Another aspect that Aalbot must consider in order to learn $\pi^{\star}_{\text{Aalbot}}$ is the distance to its opponents. If two potential targets is within the same range of energy, Aalbot must have a way to prefer one of them. Once again the sensor input is segmented into smaller bins. The bin corresponding to a distance is found using the Euclidean distance, d, within the range $0 \leq d \leq 1000$. The range assumes that the battlefield is 800×600 pixels large. The mapping function is:

$$d_{\text{bin}}(d) = \left\lceil \frac{d}{100} \right\rceil$$

This means that there are ten different possible bins.

9.5.5 After-states

The state of Aalbot thus consists of, for each enemy: The probability of aiming correctly, the energy level of the enemy and the distance to the enemy. The total number of possible states using the previously described mappings, then amounts to $(10 \times 11 \times 10)^n = 1.331.000.000$ for the number of enemies $n = 3$.

This number of states is sadly huge, and not manageable in a reasonable amount of time within the Robocode limitations. Therefore yet another technique is employed to reduce the size of the state space, namely so called *after-states.*

For any given scenario in Robocode, Aalbot knows the probability, energy level and distance of all enemies. It can then select a target, and after selecting this target the current state will then be comprised of the probability, energy level and distance of *the currently selected target*. That is, we introduce an *after-state.* The after-state can always be found from the initial information available to Aalbot and its chosen action: Aalbot has perfect information about the immediate effect of choosing an action. That effect is simply that the current state then will be comprised of the values for that target. This reasoning is similar to the Tic-Tac-Toe example in [Sut98, section 6.8].

Using this new definition of the state space, the total number of possible states then amounts to $10 \times 11 \times 10 = 1100$ states. This is manageable within the limitations set by the Robocode framework.

9.6 Actions

The architecture in Chapter 5 prescribes that outputs of modules are represented as wishes. In Subsection 9.4.2 it is documented that the target selection module outputs actuator wishes.

The available actions in the target selection module of Aalbot is a one-to-one correspondence with the wishes it produces. The maximum number of enemies in the battlefield is assumed to be five, and thus the set of possible actions are:

$$\{\text{target enemy } n \mid n \in \{1, 2, 3\}\}$$

Each of these actions can be transformed directly into a valid actuator wish by replacing n with the angle to turn the turret in order to hit the corresponding robot. The angle is selected from a list of possible targets, where each action corresponds to an index in the target list.

9.6.1 Action Selection

The target selection module of Aalbot adapts the probabilistic approach from [Mit97] for selecting actions. The probabilities for selecting each action are calculated as:

$$P\left(a_i|s\right) = \frac{k^{Q(s,a_i)}}{\sum_j k^{Q\left(s,a_j\right)}} \tag{9.1}$$

where $k > 0$ is a constant, or possibly a variable, indicating how Q-values should be weighted in the probability calculations, and $P\left(a_i|s\right)$ is the probability of selecting action a_i from state s . The target selection module of Aalbot selects actions according to the probabilities given by this equation. To illustrate the role of k in the action selection, the following can be stated:

- As $k \rightarrow 0$, the probability of selecting the action with the *lowest* Q-value increases.
- If $k = 1$ every action is equally probable to be selected.
- As $k \rightarrow \infty$, the probability of selecting the action with the *highest* Q-value increases.

Letting k increase over time will eventually favour exploitation over exploration. In that way, the target selection module will respect its current policy, π , more strictly as π is optimised. On the other hand, keeping k constant and low allows for a higher level of adaptability as the target selection module is frequently challenging the policy, by exploring new (state,action) pairs.

In the training of this module two approaches will be tested with regards to k :

1. let k increase over training epochs.
2. let k be constant.

To define what an epoch is, consider how many actions must be selected to actually try all actions from all states. Given Equation 9.1, the required number of action selections amounts to on average:

$$|S| \times |A| \times \frac{1}{P_{\forall a}}$$

where S is the set of states, A is the set of actions and $P_{\forall a}$ is the probability of selecting every action, a , at least once, using $|A|$ action selections. Initially, when no updates to the policy has been made, a clean Q-table is used and $P_{\forall a} = \frac{|A|!}{|A|^{|A|}}$. As the policy is updated consecutively, this probability becomes significantly smaller. However, it is difficult to reason about the exact size of this probability during training. An epoch is, hence, defined to be the least known number of updates to the policy, needed to try all actions from each state, namely $|S| \times |A| \times \frac{1}{\frac{|A|!}{|A|^{|A|}}} = 14.850$.

The exact values of k , in both approaches, are to be tested empirically.

9.7 Transition Function

The reinforcement learning method presented in Appendix B considers the case were the learning environment is deterministic. However, the transition function, $\delta(s, a)$, designed for this module is non-deterministic. This means that choosing a particular action in a given state does not always results in the same succeeding state. This is due to the fact that the other robots change the world state all the time. When it is Aalbot's turn to evaluate again, the distances and energy levels may have been changed because some of the robots hit each other, or they moved in a new direction than before, etc. In other words the value of $\delta(s, a)$ can change even though the values of s and a are the same.

As a result of this, the Q learning method for non-deterministic environments has been adapted from [Mit97, p. 381].

9.8 Summary

To conclude this chapter, the choices made throughout the design of the target selection module are presented here.

The goal of the target selection module is *"A robot, targeted by the target selection module, is killed"*. Fixed sized rewards are given only when the currently selected target is in fact killed.

Sensor inputs to this module are chosen to be the distance and the energy level of enemy robots. The only internal information that will be used is the output from the targeting module: For each enemy, the probability of hitting it, the angle to turn the turret and the power to shoot with.

The output from this module is a wish describing the angle which the turret should be turned in able to hit the chosen target and a wish to fire the gun with a specific strength.

A state is comprised of the probability of hitting the currently selected target, the energy level of the currently selected target and the distance to the currently selected target. Total number of states amounts to $10 \times 11 \times 10 = 1100$.

Three actions has been chosen. Each action designating a wish to select one of the three enemies as the currently selected target.

A probabilistic approach has been chosen for action selection. An explore vs. exploit variable k will be subject to experiments, in terms of defining this k as a constant, or by letting it increase over epochs of training.

Chapter 10

Genetic Programming Modules

In Chapter 7 it was chosen to apply genetic programming to the movement and radar controlling modules of Aalbot. This chapter first documents a customised approach to genetic programming taken to accommodate the Robocode framework. Subsequently, genetic programming is applied to the two modules in question. The chapter builds on the terminology and principles of genetic programming introduced in Appendix C.

10.1 Customised Approach

This section gives the overall design of a genetic programming framework created by the project group to support genetic evolution of modules in Aalbot.

More accurately, the following areas are to be elaborated upon:

- How an individual in the population is represented.
- How the initial population is generated.
- How the fitness of an individual is measured.
- Which termination criterion is used.
- And finally, which of the genetic operations presented in Appendix C are used, and which alternatives are considered.

In addition some of these areas might require exogenous parameters, such as the population size, ratios for determining how large parts of the population a certain genetic operation should be applied to between generations etc.

10.1.1 Representation of Individuals

As known, genetic programming evolves a population of individuals over a number of generations. In regards to Aalbot, an individual is defined more precisely as a program constituting a particular module, which is chosen to be developed using genetic programming.

The *representation* of the programs to be evolved is chosen as in most other examples of genetic programming. According to the common literature [Koz92, p. 71] on the field, using parenthesised Lisp-like expressions to represent the programs is a widely used practice. Lisp expressions have the all-important benefit of being direct representations of their own parse trees, which makes the genetic operations manipulating parse trees directly applicable without generating syntactically invalid programs.

The various structures and operators of genetic programming are implemented in Java, and therefore the main representation of an expression is chosen to be a tree structure built of Java objects, resembling the parse tree of the expression. It is not necessary to manually write expressions, which makes the construction of a parser superfluous, and hence a parse tree is a sufficient representation of an expression. This parse tree can easily be serialised in Java and saved to a file that can be read by a robot for interpretation. An auxiliary print function is useful though, to convert the parse tree into a Lisp-like program string, suitable for visual inspection of e.g. the best-so-far individual.

A small language is designed to fit the application domain, and can be considered a variant of Lisp. An interpreter for the language is constructed to handle single expressions using an eval/apply cycle as described in [ASJ96, section 4.1]. The basic idea of this eval/apply cycle is to evaluate a function, by first evaluating its arguments recursively, and secondly applying the function to the values obtained from this evaluation.

A program written in this language is basically one large expression, which is possibly composed of many sub-expressions. An expression can refer to functions, special forms and terminals, including integer constants, world state data and sensor input received from the environment of Aalbot. Functions and terminals must evaluate to real values, thus ensuring the property of closure between the function and terminal set.

The syntax of the language is expressed in the BNF grammar below.

$$
\begin{array}{lcl}
\langle Exp\rangle & ::= & \langle Function\rangle \\
 & | & \langle Special\ form\rangle \\
 & | & \langle Terminal\rangle \\
\\
\langle Function\rangle & ::= & (+\ \langle Exp\rangle\ \langle Exp\rangle\ ) \\
 & | & (-\ \langle Exp\rangle\ \langle Exp\rangle\ ) \\
 & | & (*\ \langle Exp\rangle\ \langle Exp\rangle\ ) \\
 & | & (/\ \langle Exp\rangle\ \langle Exp\rangle\ ) \\
 & | & (\,\texttt{random}\ \langle Exp\rangle\ ) \\
\\
\langle Special\ form\rangle & ::= & (\,\texttt{sequence}\ \langle Exp\rangle\ \langle Exp\rangle\ ) \\
 & | & (\,\texttt{iflessthan}\ \langle Exp\rangle\ \langle Exp\rangle\ \langle Exp\rangle\ \langle Exp\rangle\ ) \\
\\
\langle Terminal\rangle & ::= & 0 \mid 1 \mid 3 \mid 5 \mid 10 \mid 20 \mid 30 \mid 50 \mid 180 \mid 360 \\
 & | & \text{World state data} \\
 & | & \text{Sensor input}
\end{array}
$$

The first four functions are the usual arithmetic operators with the minor exception that the division operator is changed so that division by zero yields zero as result. The random function generates a random integer in the interval ranging from 0 to the indicated limit.

Note that the last two clauses of ⟨*Terminal*⟩ are described informally here, as they vary depending on the module. Hence these terminals will be specified more accurately later. In addition, multiple Robocode functions are added to the language, but as with the terminals these are also specific to each module.

Two *special forms*, which are expressions that require a special evaluation order, are included in the language. The `sequence` expression is evaluated sequentially by evaluating the first expression before evaluating the second expression. The value of the entire expression is given by the result of evaluating the last expression.

The semantics of the `iflessthan` special form is as follows. The first two expressions are evaluated sequentially, and if the value of the first expression is less than the value of the second expression then the third expression is evaluated, else the fourth. The value of the whole expression is thus the value of either the third or fourth expression.

No further special forms or module independent functions are included in the language, as functions such as *if-greater-than* or similar would not make the language more expressive.

More advanced features of Lisp, such as environments, loops, lists, lambda functions etc. have been left out to limit the search space of possible expressions to be considered by the genetic programming process.

Choosing the explained Lisp-like representation of individuals ensures that the implementation of genetic operators are simplified. Still a relatively high

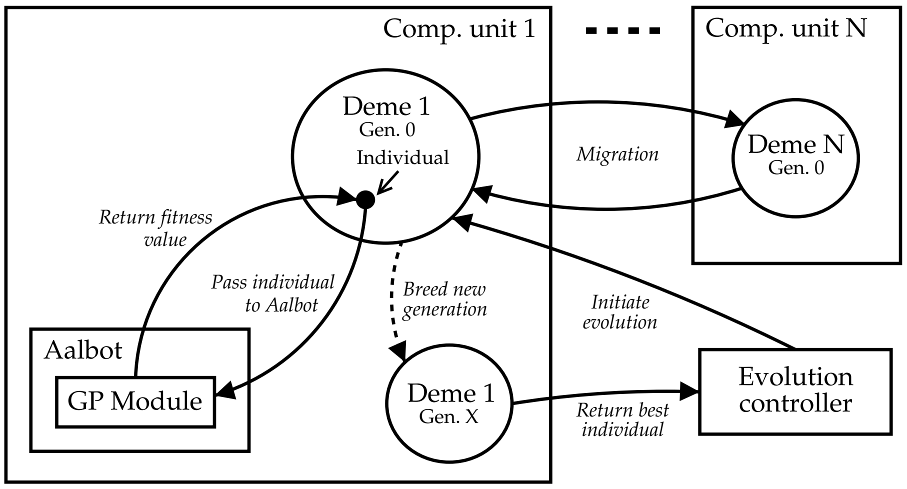


Figure 10.1: Overview of the distributed evolutionary framework employed in this project.

level language is preserved, making it easy to understand the meaning of a program from its parenthesised string form. The interpreter is significantly easier to implement than writing a compiler to convert parse trees into byte code. The price paid is a slight decrease in execution speed.

10.1.2 Distributed Evolution

The amount of computational resources required by the process of genetic evolution is without doubt very large, mainly because the fitness of each individual has to be measured between every generation. For this reason, a framework is designed to allow distributed evolution of the genetic programming modules in Aalbot. The approach taken is to divide the population of individuals into *demes* (small local populations), which are to evolve separately on different computational units. Individuals can *migrate*, i.e. move from one deme to another, which minimises the possibility of a deme being dominated by individuals with undesirable characteristics.[Mit97, p. 268]

The distributed evolution framework is presented in Figure 10.1. *N* computational units each evolve a deme of individuals and are controlled by the *evolution controller*, which has the responsibility of initiating and stopping the entire evolutionary process. On the left-hand side of the figure, the details of the first evolutionary steps on a single computational unit are illustrated. The evolution controller initiates the evolution of a deme by sending the computational unit a message containing relevant parameters such as deme size, the function and terminal set to be used, the fitness function etc. The initial generation 0 is then generated randomly to contain the required number of individuals, each of which is tree structure of Java objects representing an expression, as described in the previous section.

As shown in the figure, an individual (represented by the dot) is passed to the genetic programming module of Aalbot. The individual is then *interpreted*, while Aalbot takes part in a simulated battle against a number of opponents. Once the battle is over, some fitness value is calculated depending on how well Aalbot performed in the battle. This value is returned and stored with the corresponding individual for later retrieval. When the fitness values of all individuals have been calculated, a new generation is bred by applying the genetic operations, which also implies that some individuals migrate to other demes in the system. The migration operator is implemented by serialising the object representing the individual and sending it to the destination deme.

At any time, the evolution controller can request the best-so-far individual from a deme, and it can then decide whether to stop or continue the evolution depending on the termination criterion used.

10.1.3 Generation of Initial Demes

The initial demes are generated randomly, thus beginning the evolution with broad and diverse populations of individuals. Two methods described in [Koz92, p. 92] for generating a random parse tree representing an individual are used. Both methods employ a minimum and maximum allowed depth of the tree, which are initially chosen to be respectively 2 and 6 as suggested in [Koz92, p. 93]. By choosing a minimum depth parameter of 2, the occurrence of trivial and useless individuals such as a tree only consisting of a single terminal is avoided.

The so called *grow* method grows trees of depths between the minimum and maximum specified depth. It randomly selects a function from the function set and designates it as the root node of the tree. For each of the root function's arguments a function is again chosen randomly from the function set and inserted appropriately in the tree. From this point on an element from the union of the terminal and function set is chosen to further grow the tree until all leaf nodes are terminals or the maximum depth is reached, in which case the final selection of leaf nodes is of course restricted to the terminal set.

The other method for generating a random tree is the *full* method, which works similar to the grow method except that it always produces trees with the maximum allowed depth on all branches.

According to [Koz92, p. 93] creating a diverse initial population (or deme) having a wide variety of sizes and shapes can be done using a "ramped half-and-half" method. The method prescribes, that a depth parameter ranging between 2 and the maximum specified depth should be used. As the max. depth parameter is 6, 20% of the population should be created with a depth of 2, 20% with a depth of 3 and so on. For each value of depth, half of the individuals should be created using the grow method and the other half using the full method.

The total number of individuals to be randomly generated for the initial population in a non-distributed system, i.e. the population size M, is typically chosen to be around 500 as proposed in [Koz92, p. 116]. As the evolutionary framework designed for Aalbot is distributed, it is chosen to generate initial demes of 100 individuals. It is impossible to do any sensible reasoning on whether 100 is an optimal choice, and it will thus be tested whether generating e.g. only 50 individuals per deme will produce the same results.

10.1.4 Fitness Measure

Between generations the fitness of each program in a deme must be measured. A module consisting of some expression taken from a population must be incorporated into a pre-made robot, where all other modules are functioning.

The fitness of a robot is chosen to be the number points scored in a Robocode battle of 10 rounds, plus an amount of bonus points if the robot is the last survivor. The fitness of a robot should ideally be influenced by some specific measure of how well the module developed by genetic programming performs and not just the overall performance of the robot. For example, the radar module's performance could be measured by taking into account the average time between an enemy is scanned by the radar.

3 enemies with different skills and behaviours are selected to battle against the robot, which fitness needs to be measured. The enemies are selected in a way, such that the module performs well in the general case, and not only against specific types of opponents. Obeying this requirement, there is no need to evaluate the fitness of a robot over a number of fitness cases, which would slow down the process unnecessarily.

10.1.5 Termination Criterion

No strict termination criterion will be specified, but instead a loosely defined, yet acceptable time bound determines when to stop breeding a deme of individuals. The method of registering the best-so-far individual found from generation to generation will be used, though the usual specification of the maximum number of generations G is left out. Thus it is possible to simply run the process for as long as possible within the restriction of the deadline of this project. Stopping the process at any state is possible, since the best-so-far individual from each generation is always stored.

10.1.6 Genetic Operations

This subsection presents the genetic operations used to generate new demes of individuals. The individuals are chosen from the deme using fitness-proportionate selection favouring individuals with higher fitness as described in Appendix C.

It is chosen to apply the mutation operator, which works simply by choosing a random node in the individual and replacing the sub-tree starting at this node with a new randomly generated tree. Many different types of cross-over operations exists — for example 2-point cross-over, context preserving cross-over or variants where an individual is recombined with itself (i.e. subtrees are swapped within the same individual). However, it is chosen to initially restrict the operations to the simple 1-point cross-over as described in Appendix C.

The reproduction operator will be used to select individuals exhibiting a high fitness from a deme and copy these directly into the following generation. This operator will be used relatively few times compared to the mutation and crossover operations, as it imposes a risk of premature convergence. The migration operator is comparable to the reproduction operator, except that it copies individuals between demes residing on different computational units.

The operators are given the following probabilities of being chosen in the process of breeding a new generation.

- Crossover: 60%
- Mutation: 30%
- Reproduction: 5%
- Migration: 5%

In other words, the resulting generation created with operations having these probabilities should contain a number of individuals, of which roughly 60% were created using crossover, 30% using mutation etc.

10.2 Movement Module

The movement module is responsible for dictating a movement strategy for Aalbot leading it to victory. The module has to be highly aware of the properties of the environment, and will thus depend on the information received from the world state component and the sensor inputs. The purpose of the module is to test whether a machine learning technique can develop a movement controlling program, that performs better than the anti-gravity movement technique, which was described in Subsection 10.1.1.

As prescribed by the method of genetic programming, the first step is to identify the terminals, that have relevance to the problem. The terminals are generally speaking sensor inputs, world state variables or wishes from lower layers. Following is an expansion of the grammar introduced in Subsection 10.1.1, with the terminals assumed to be significant in the construction of a movement controlling program.

$$
\begin{array}{lcl}
\langle \textit{Terminal} \rangle & ::= & \texttt{enemyCount} \\
 & | & \texttt{enemyN\_x} \\
 & | & \texttt{enemyN\_y} \\
 & | & \texttt{enemyN\_heading} \\
 & | & \texttt{enemyN\_distance} \\
 & | & \texttt{aalbotHeading} \\
 & | & \texttt{aalbotHealth} \\
 & | & \texttt{currentTime}
\end{array}
$$

Note, that each of the enemies is given a unique number, denoted by `N` in the above grammar. To exemplify, the terminal `enemy3_distance` holds the distance from Aalbot to the enemy identified by the number 3. Furthermore, `enemyCount` gives the number of enemies currently on the battlefield, `enemyN_x` and `enemyN_y` gives the x and y coordinates of enemy number `N`, respectively.

As the next step, the functions controlling the movement of the robot is specified. They are all named after the corresponding methods in the Robocode API, and hence their functionality should be known from Chapter 1.

$$
\begin{array}{lcl}
\langle \textit{Function} \rangle & ::= & (\,\texttt{ahead}\ \langle \textit{Exp} \rangle\ ) \\
 & | & (\,\texttt{back}\ \langle \textit{Exp} \rangle\ ) \\
 & | & (\,\texttt{turnRight}\ \langle \textit{Exp} \rangle\ ) \\
 & | & (\,\texttt{turnLeft}\ \langle \textit{Exp} \rangle\ ) \\
 & | & (\,\texttt{stop}\,) \\
 & | & (\,\texttt{resume}\,) \\
 & | & (\,\texttt{execute}\,)
\end{array}
$$

When an expression is interpreted each of the above functions should produce a wish representing the desired movement and add it to the layer's ordered wish list.

10.3 Radar Module

In Subsection 1.2.1 a number of different strategies for controlling the radar were presented. Exactly which strategy to employ depends on many factors, i.e. the number of enemies, their positions etc. The main purpose of the radar module is to keep the fading memory map, containing information on the positions of the enemies, as updated as possible. The hope is to find a good strategy that will reduce the time spend between scanning each of the enemies. Genetic programming will be applied to the problem of finding the best overall radar controlling program, as no satisfying control protocol has been identified.

Following is a list of the terminals which are assumed to be relevant for the radar module:

$$
\begin{array}{rcl}
\langle \textit{Terminal} \rangle & ::= & \texttt{enemyCount} \\
 & | & \texttt{enemiesSpotted} \\
 & | & \texttt{enemyN\_x} \\
 & | & \texttt{enemyN\_y} \\
 & | & \texttt{enemyN\_heading} \\
 & | & \texttt{enemyN\_distance} \\
 & | & \texttt{enemyN\_lastSeen}
\end{array}
$$

The following functions controls the radar, and are hence relevant to the radar module:

$$
\begin{array}{rcl}
\langle \textit{Function} \rangle & ::= & (\,\texttt{turnRadarLeft}\ \langle \textit{Exp} \rangle\ ) \\
 & | & (\,\texttt{turnRadarRight}\ \langle \textit{Exp} \rangle\ ) \\
 & | & (\,\texttt{execute}\,)
\end{array}
$$

10.4 Summary

This chapter has presented the customised approach taken to genetic programming. Terminals and functions have been identified for the movement and radar module, which are to be developed using genetic programming. Many choices regarding parameters, fitness measure, generation of initial demes etc. have been made based on both experience reported by others and on rational assumptions. Several tests will hence reveal if the choices made were good, and will provide the opportunity of reconsidering and changing the parameters to achieve better performance.

The following list summarises the parameters of the genetic programming framework, that can be altered in a test situation:

- More advanced features such as loops, lists, lambda functions could be added to the language.
- The number of individuals in a deme.
- The terminal and function set depending on the module. Some elements are perhaps irrelevant, while others might be missing.
- The genetic operations used for breeding new generations, and the probabilities given to each of them.

Part III

Evaluation

Chapter 11

Neural Network Evaluation

In the following, an evaluation of the neural network module used for targeting is given. This chapter starts by presenting the overall results achieved from informal tests of each of the three methods listed in Section 8.4. One of these methods is then chosen as subject for thorough testing, and the test environment and performance measure are discussed. The performance measure section includes a discussion on how the network error evolves over time.

The following list of parameters have been selected for empirical testing – the numbers indicate the values used during initial tests:

- Learning rate, $\eta = 0.3$
- Momentum, $\alpha = 0.3$
- History size, e.g. number of (x, y) pairs is 5
- 5 hidden neurons

The purpose of the tests is to get insight into the workings of the neural network, but also to determine the values of each parameter that lead to the best targeting performance. However, it is impossible to perform an exhaustive test of all possible parameter combinations. Therefore it is assumed that an optimal value for one parameter will remain optimal independently of the remaining parameter values. That is, only one parameter will be varied in each test. The default values indicated in the list above will be used for parameters which are held constant.

Each parameter test will be discussed in the respective sections below, and the chapter will conclude with general comments on issues regarding the targeting module.

11.1 Choice of Method

Because of time restrictions only one of the three overall approaches to neural network targeting has been chosen for thorough testing. The choice is based on initial informal tests through which the method likely to perform the best is chosen.

The third approach predicts the turret turn angle in a single feed forward cycle based on an enemy (x, y) history and the position of Aalbot. No immediate success was achieved with this approach: Although the network error converges to a reasonable low value, the network fails when predicting angles based on unseen inputs. That is, it does not generalise. There are a number of possible reasons for this:

- The network is overfitting on its training data, which might indicate that there are too many weights in the network or that too many training cycles are applied at each of the robot's turns.
- Insufficient training data are collected. Recall that training data are collected by means of virtual bullets, and that a fixed bullet speed is assumed. Possible virtual bullet hits are only calculated when a robot position is sampled, and a robot may have moved a large distance (typically more than 100 pixels) between successive samples. Even though a virtual bullet could have hit the enemy in between these samples, it may not be able to hit within a radius, r, of the sampled position. In this case the virtual bullet will be discarded, and no training example will be recorded.
  In the initial tests only approximately 50 training examples were collected in the first round; considering the complexity of the prediction task, this number is too low. One solution might be to experiment further with the radius threshold r.

Because of these problems, further training using the third approach will not be prioritised. This introduces a problem, as only the third approach allows for a direct measure of network certainty, which is required in the target selection module. Instead of measuring network certainty based on categorical network output encoding, the certainty of hitting a target is therefore calculated based on the latest recorded error of the corresponding neural network. Each certainty, $p \in [0; 1]$, is calculated so that the sum of certainties for all targets is 1. Although this does not give an definite certainty of a network output, it is a reasonable alternative to discarding the certainty factor all together.

The first and second approaches are based on one-step-ahead prediction of heading/velocity pairs and (x, y) pairs, respectively, and they construct the expected enemy movement path from a list of these pairs. In contrast to the third approach discussed above, both of these have given reasonable results during the initial tests. These tests also imply that the (x, y) based approach is superior to the heading/velocity approach, especially for enemies which move in the same area of the battle field; however it also generalises readily when facing enemies which move all over the battle field. Therefore the second approach based on a (x, y) history is chosen as a basis for further testing, which is the subject of the rest of this chapter.

11.2 Test Environment

Ultimately the neural network targeting module must perform well together with all other modules in Aalbot and should assist in achieving the goals set forth in Chapter 4. Testing the targeting module together with other modules will, however, complicate the interpretation of the neural network performance. Therefore the module will be tested in a test robot with little other functionality than targeting: the test robot immediately moves to one of the arena walls, after which it does nothing but targeting and shooting.

Only one enemy will be present and this enemy must not be chosen from the robots used for the final evaluation of Aalbot, as this would bias the network parameters to work especially well for these. Instead the `sample.SpinBot` robot, which moves in small circles, is used, but with a few modifications:

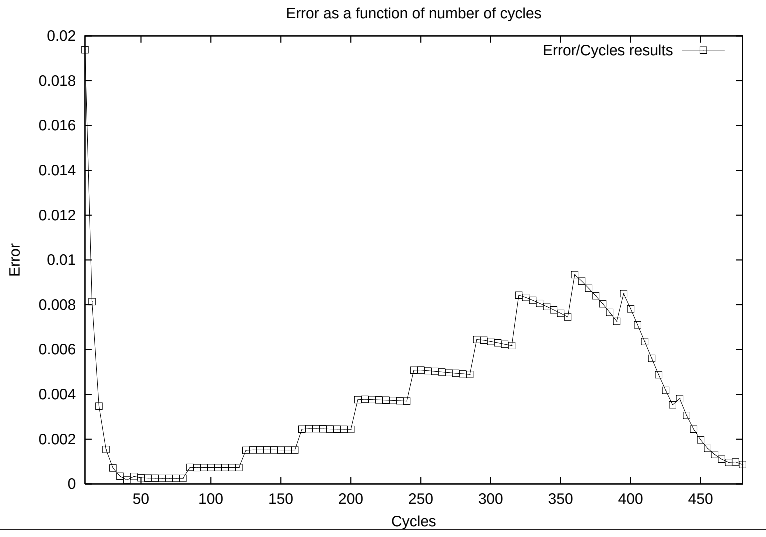
- Shooting functionality is removed, as the test robot would otherwise be destroyed too fast (because it does not move).
- `SpinBot` moves in very small circles, so the radius of the circle is increased in the the modified version. The radius is chosen in order to ensure that the enemy does not repeatedly move over the exact same circle, but inevitably hits a wall and is forced to change trajectory. This is important when evaluating the (x, y) based history approach, because the network's ability to generalise over unseen coordinate sets would otherwise not be reliably tested.

It is assumed that network parameters found to work well against this modified version of `SpinBot` will also work well against the robots used in the final evaluation of Aalbot.

11.3 Performance Measure

A performance measure must be chosen in order to evaluate how different network parameters influence the ability of the targeting module to hit an enemy. In off-line training tasks a validation set (or possibly a test set) often serves this purpose. Since no validation set is available, one might simply evaluate the network's performance based on the lowest network error over the training set during the course of training.

To investigate this idea, consider the two graphs in Figure 11.1. The top graph shows how the error evolves during the initial training cycles. The key to explaining this behaviour is that training data is added dynamically to the training set, and at the first training cycles only one training example is available. Therefore the error rapidly decreases from the initial error determined by the random initial network weights. After approximately 80 iterations new training data is added, and consequently the network error grows because weights



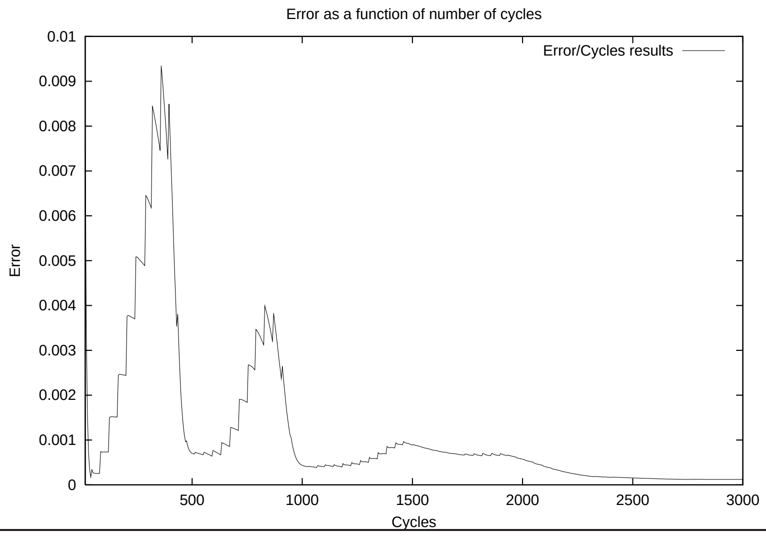


Figure 11.1: Network error over time. The top graph shows the error in the cycles 10 to 480, while the bottom graph shows the error over a larger interval. With the settings used in this test, approximately 6500 training cycles are completed within the first round in Robocode.

have not yet adapted to the new examples. The error then slowly decreases until new training data is added. This behaviour repeats until cycle 360 is reached: Here the impact of added training data diminishes, because the network is starting to generalise over the training data.

The bottom graph of Figure 11.1 shows the network error from the same test case, but over a larger number of training cycles. The network error peaks at cycles 360, 800 and 1450, after which the error stabilises at a low value. The sudden growth in network error at these positions are likely to be caused by `SpinBot` changing its trajectory after hitting a wall, which means that the network experiences input data from another area of the battle field. However the height of these peaks decreases rapidly with the number of training cycles, suggesting that the network is capable of generalising over specific enemy locations.

To get the intuition of the error's magnitude, recall that x and y coordinates are scaled to the range $[0; 1]$. The maximum value of an x coordinate is 800, hence the error in pixels is typically between 1 and 10. Although this error is accumulated over multiple network feed forwards, it is still reasonable considering the dimensions of a robot.

The above investigation of the network error over time provides interesting insight on how the network works, but is of little use when measuring targeting performance of the network. The lowest recorded error would probably occur in the initial cycles, and this does not provide any indication of overall performance. Instead a more direct performance measure will be used, namely the bullet damage points rewarded to the test robot by the Robocode system over 5 rounds. This measure is not ideal either, because it is not a direct measure of how many bullets hit their target – bullet damage is also affected by bullet power. However this measure will still give a good overall picture of network performance. To put the bullet damage numbers presented in the following sections into perspective, note that the targeting module scores in the order of 60 bullet damage points on an untrained network.

11.4 Learning Rate

The learning rate, η, influences how much network weights are adjusted at each iteration of the backpropagation algorithm. Figure 11.2 shows how this parameter affects the targeting ability of the test robot.

As expected, very high learning rates degrade the performance of the targeting module. The reason is that network weights are adjusted such that the network error oscillates around a local minimum. However there are also high values of η where the targeting performs reasonably well, e.g. $\eta = 15.5$ where a score of over 200 is achieved. The large performance difference between e.g. $\eta = 17.5$ and $\eta = 16.5$ can be explained by the illustration in Figure 11.3. Case R corresponds to $\eta = 16.5$ where the error oscillates between R_1 and R_2 and never gets closer to the local minimum. Case S corresponds to $\eta = 17$ where

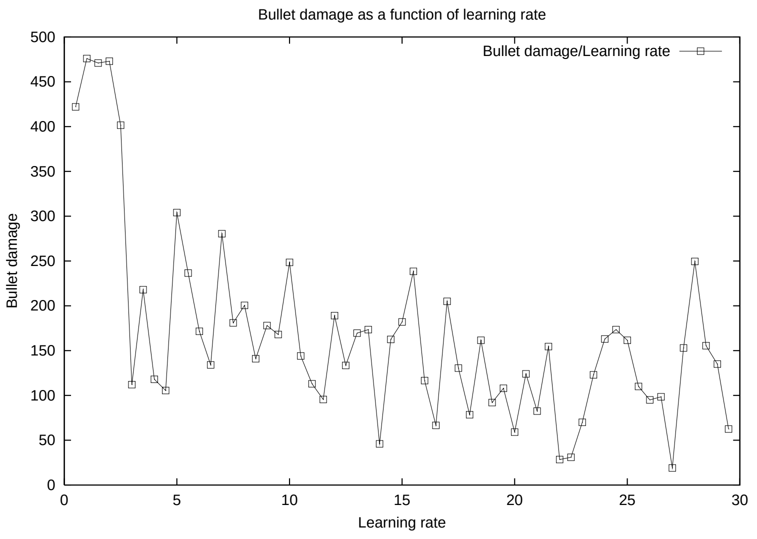


Figure 11.2: Bullet damage as a function of the network learning rate.

the initial value of w makes it possible to reach a local minimum, S_1, with the step size dictated by η.

The graph indicates that a learning rate should be found in the interval $[0;2]$, so a test with η-values in this range has been run. Very low values result in slow convergence, and the targeting performs bad; the optimal value to be used in Aalbot has thus turned out to be $\eta = 0.3$.

11.5 Momentum

Momentum can have the effect of guiding the gradient search through a bad local minimum, but it can also have the opposite effect of leading the search through a global minimum. It is therefore difficult to reason about the value of the momentum parameter, α. Figure 11.4 shows how the momentum parameter affects the targeting performance of the neural network. Apparently $\alpha = 0$ gives the best results, meaning that no momentum is used at all. Hence no momentum is used in Aalbot.

One reason for the poor results at high momentum values could be that network adaptiveness is decreased, as new training examples have less effect on weight updates. Furthermore, at momentum values over 1, the weight updates are performed in much larger steps – these steps might be too large for the network error to converge at a local minimum. A decrease in learning rate might compensate for this but has not been attempted in the test.

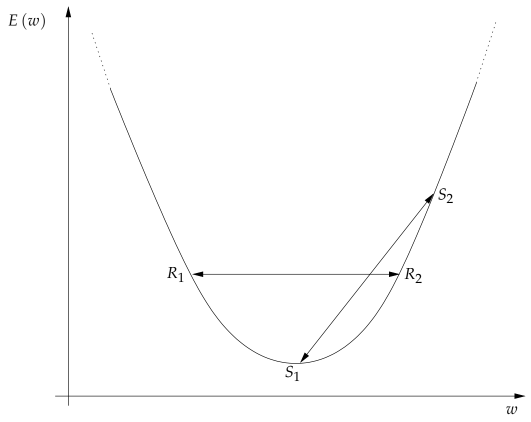


Figure 11.3: Illustration of the network error's position on the error surface in the hypothetical case of one network weight, w. Two training cases, R and S, are represented. R_1 and R_2 are errors from training case R, while S_1 and S_2 are errors from training case S. Lines connect errors from successive training cycles in the same training case, and their length imply the size of the training rate, η. Even though η is almost identical in the two cases, the network converges to a lower error for case S.

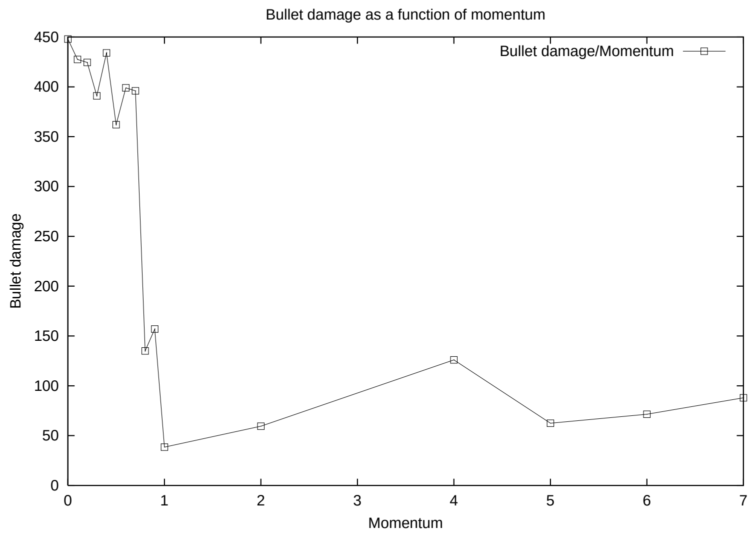


Figure 11.4: Bullet damage as a function of network momentum.

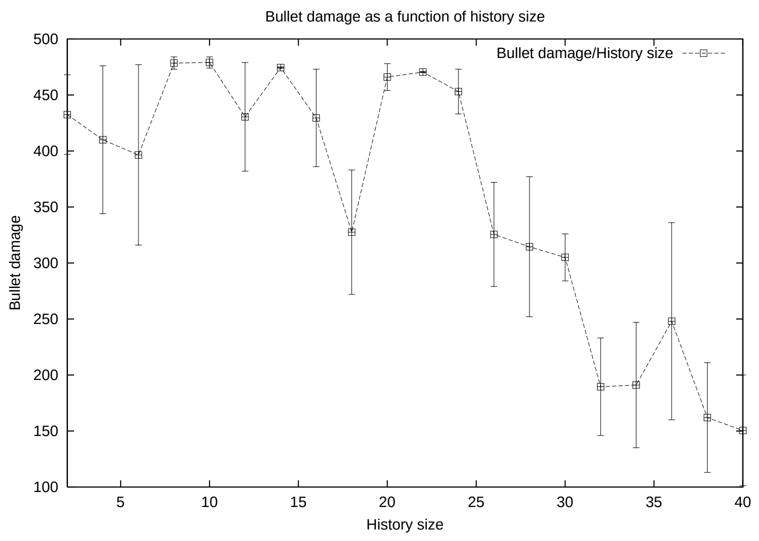


Figure 11.5: Bullet damage as a function of the history size. Vertical lines show the standard deviation over two runs of the same test case.

11.6 History Size

The history size, N, refers to the number of (x, y) pairs used for prediction of a future position, so the number of network inputs is $2 \cdot N$. Figure 11.5 shows how the test robot's performance varies with N: At history sizes larger than 14, the performance starts to degrade. The reason for this is two-fold. First preprocessing is more difficult at larger history sizes, because training data is discarded if the time between successive samples varies too much. In these cases time normalisation does not work well. The second reason follows from the curse of dimensionality – the higher the dimension of the search space, the more training data and training time is required in order to perform well.

The history size is probably the parameter which depends most on the robot whose movement path is to be predicted. To give the intuition behind this statement, consider a human predicting the future positions of a robot, r. If r always moves in circles, two (x, y) pairs would be enough for predicting the next position of this robot. On the other hand, if r follows a sine curve, a longer history would be required to reliably predict a future position. This explains why a history size of 2 is sufficient for predicting the modified `SpinBot`, which moves in large circles most of the time.

A history size of 8 is chosen because this works well in the test case, the deviation is low and can also be expected to give good results for robots with more complex movement patterns than `SpinBot`.

11.7 Number of Hidden Neurons

Surprisingly targeting performance has shown to be insensitive to the number of hidden neurons in the network for values between 4 and 200 – values less than 4 only give slightly worse performance. Therefore no graph is shown for this test case, and 4 hidden neurons are chosen for Aalbot. This results in the simplest network which still performs well (approximately 450 bullet score points).

One might expect a network with many hidden neurons to overfit easily, but this is clearly not the case in this test. The reason might be that the training set changes dynamically, so the network has no opportunity to learn all training examples before they are removed from the training set to make room for new examples. The test robot naturally gets slower as the number of hidden neurons increases, but still it does not miss any turns, even when training with the maximum number of 200 hidden neurons used in this test.

11.8 Conclusion

Results from neural network training have been discussed in the preceding sections, which has yielded insight into the inner workings of the neural network. Parameter values which are likely to result in the optimal targeting performance have been chosen, namely a learning rate $\eta = 0.3$, momentum $\alpha = 0.0$, history size of 8, and 4 hidden neurons.

The optimised test robot has a typical bullet damage score against the modified `SpinBot` of almost 500. This is 7- 8 times better than the score achieved using random targeting, which is satisfactory. A visual inspection of a battle also reveals that the test robot performs well after having trained through the first round.

Chapter 12

Reinforcement Learning Evaluation

This chapter documents the results of using reinforcement learning in the target selection module of Aalbot. First the test environment is discussed and a choice of testing parameters is made, especially regarding the selection of actions. Following this, the actual evaluation is carried out based on two approaches. The first approach examines how the maximum cumulated reward progresses over time, which establishes the correctness of the implemented Q-algorithm. The second approach goes into more detail on how the learned policy chooses actions and seeks an optimal value of k through empirical tests.

In order to test the performance of the target selection module used in Aalbot, the neural network based targeting module must perform reasonably well. Reasonably well, in this case, means that the module is able to produce a qualified guess on the angle to turn the turret, the power to shoot with and the probability of hitting for at least two of the targets on the battlefield. Thus such a behaviour from the targeting module is assumed throughout the performance tests.

12.1 Test Environment

Different approaches for selecting an action from the Q-table, or more specifically varying a constant k, were proposed in Chapter 9. Due to the time restrictions in this project, only one approach is tested. If the constant k from equation 9.1 on page 77 is chosen to increase over time many training epochs have to be carried out before the action selection relaxes its tendency to explore. Testing with an increasing k is skipped, and hence the action selection method is limited to a constant value of k during each training case.

In order to choose a reasonable value of k, some considerations about the consequences of different k values have to be made. In order to do so, an arbitrary entry in a Q-table, $(4.7, 8.6, 6.4)$, has been considered, associating actions with expected cumulative rewards. As an example the entry 4.7 is interpreted as the expected cumulative reward when choosing action 1, denoted a_1 in the table below, from the state s. Each action corresponds to selecting a target in Robocode. The consequences of varying k are illustrated in the following table of probabilities, and the reader is referred to equation 9.1 on page 77 for information on how to calculate the table values:

$\mathbf{p(a_i \mid s)}$	**weighted**	$\mathbf{k = 0.5}$	$\mathbf{k = 1.0}$	$\mathbf{k = 1.1}$	$\mathbf{k = 1.2}$	$\mathbf{k = 1.5}$
$\mathbf{Q_{a_1} = 4.7}$	0.239	0.727	0.333	0.276	0.227	0.127
$\mathbf{Q_{a_2} = 8.6}$	0.436	0.049	0.333	0.400	0.463	0.619
$\mathbf{Q_{a_3} = 6.4}$	0.325	0.224	0.333	0.324	0.310	0.254

The column *weighted* is the weighted probability of choosing the different actions if k was removed from the equation. The table indicates that values of k in the interval $]1.0; 2.0]$ are viable choices. A value of $k = 1.1$ ensures a certain degree of exploration, and values of $k = 1.5$ and upwards tend to suppress exploration in favour of exploitation. The target selection module in Aalbot uses three different values during performance tests of k, namely $k = \{1.1, 1.2, 1.5\}$.

The target selection module is trained in melee battles against 3 other robots. The 3 training opponents are chosen to be `sample.Crazy`, `sample.SpinBot` and `sample.MyFirstRobot`, because these robots do not perform exceptionally well. Data are collected during training to facilitate the analysis of the target selection module's performance. Training against more advanced robots could be expected to lengthen the training process, as no rewards will ever be given if Aalbot is the first robot to be destroyed. However, training against more advanced robots would accordingly prepare Aalbot for the final tests against the robots selected in Chapter 4.

12.2 Performance Measure

Two different performance measures are used to test the target selection module of Aalbot:

- The cumulative reward.
- The action selection pattern.

The first measure will give an indication of whether or not the implementation of reinforcement learning used in Aalbot is correct. Maximising the cumulative reward is the sole purpose of reinforcement learning, and the applied Q-learning algorithm should ensure this. The test itself is conducted by storing the cumulative reward obtained during each round of Robocode.

The second performance measure is more expressive with regards to the acquired policy of the target selection module. Aalbot selects many targets during a Robocode battle. Each time a target is selected data are collected about that target – relative to the targets Aalbot did *not* choose. That is, for each target Aalbot selects, it stores three pieces of information;

- **Probability.** The target chosen had highest, lowest or medium probability to get hit.
- **Energy.** The target chosen had highest, lowest or medium energy.
- **Distance.** The target chosen had highest, lowest or medium distance to Aalbot.

This information, gathered over successive rounds, indicates what sort of selection scheme the target selection module has learned. In other words, it illustrates the learned policy.

Note that, due to the discrete nature of the data and the applied segmentation in Subsection 9.5, it is not always possible to determine which of two targets has e.g. the highest energy. In such cases both targets are considered to have highest amount of energy.

The latter performance measure strives to put the target selection module and reinforcement learning into perspective by applying the bullet damage score, from Robocode, as a measure of performance. The results will guide the choice of k to be used in the final test setup against the robots selected in Chapter 4.

12.3 The Cumulative Reward

The cumulated reward over time obtained by the target selection module has been collected over 500 rounds. These cumulated rewards express how many points the target selection module collects as it moves through different states in the Q-table.

The cumulated rewards of course varies with the number of selected actions during a round, which exactly amounts to the number of turns in a round of Robocode. Hence, longer battles can be expected to result in greater cumulated rewards since more Q-table entries are visited. However, as Figure 12.1 illustrates these fluctuations in round time do not affect the graph considerably.

From Figure 12.1, it is clear that the Q-values of the states visited by the target selection module, given by the choice of target, increase over each round of Robocode.

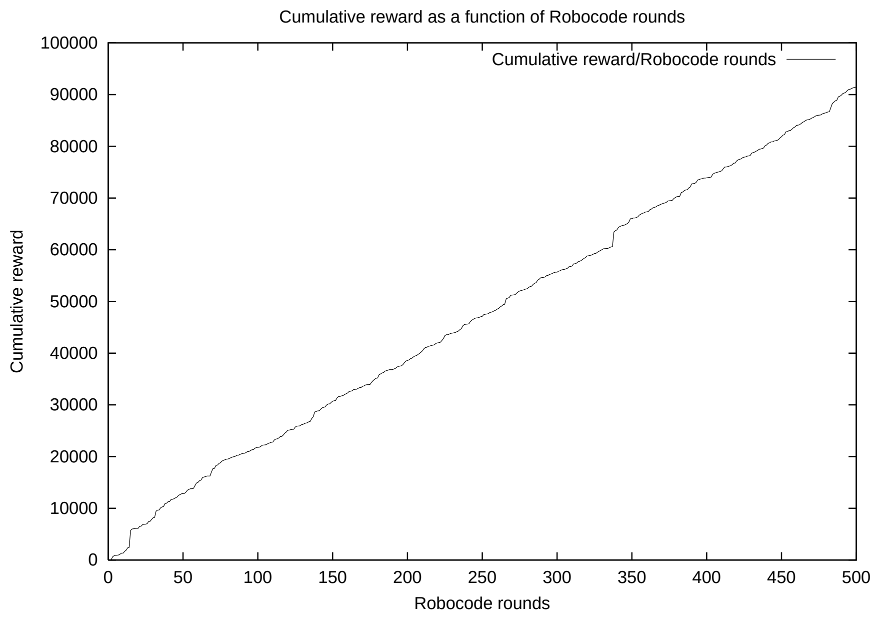


Figure 12.1: The cumulated rewards over Robocode rounds.

12.4 The Action Selection Pattern

This performance measure is used to examine the policy that the target selection module has learned using different compromises between exploration and exploitation. This section presents the results of the individual performance measurements obtained using the values, $k = \{1.1, 1.2, 1.5\}$, as described in Section 12.1. This measure allows for comparing the learned policy with an intuitively devised optimal policy outlined by three primitive rules.

1. Choose a target with a lower energy level above targets with higher energy level.
2. Choose a target closer to Aalbot over those farther away.
3. Choose a target that is more probable to be hit over any other targets.

The test were conducted by, for each value of k, collecting data from 20 rounds starting with a Q-table of all 0.0 entries. These initial rounds of testing were then followed by 100 rounds without any data collection, but of course, still optimising the target selection policy. Finally, Aalbot fought another 20 rounds this time collecting data again. The collected data were used to produce the graphs in Figure 12.2, Figure 12.3 and Figure 12.4.

It must be mentioned that selecting the medium values of probability, energy and distance happens seldomly, since the targeting module rarely produces more than two possible targets, and a third target is needed in order to pick this value.

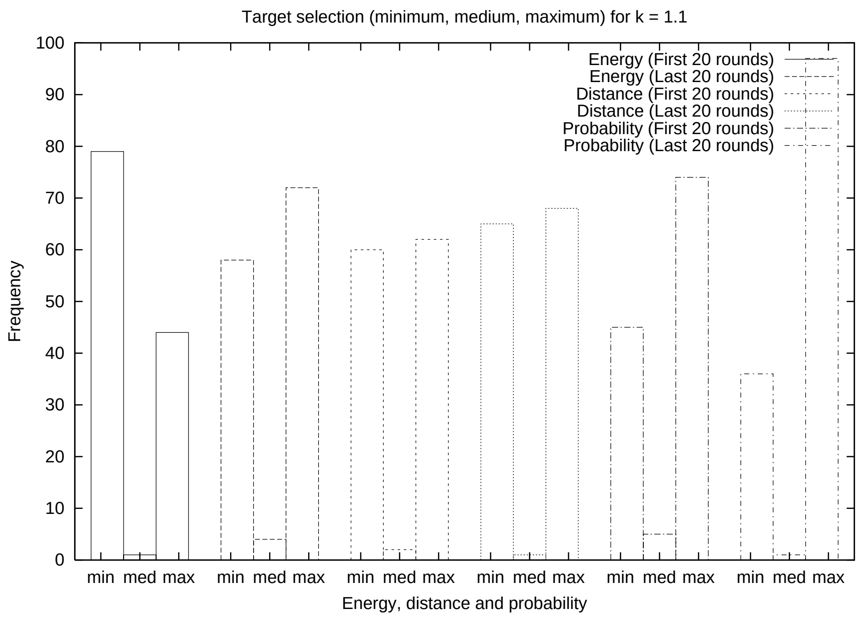


Figure 12.2: Starting with an empty Q-table and a value of $k = 1.1$, this figure illustrates the policy seen over the first and last 20 rounds of measuring performance.

As illustrated in Figure 12.2, the acquired policy seen over the first 20 rounds, using a value of $k = 1.1$, is not able to decide whether to prefer targets close to or far from Aalbot. However it tends to choose targets that have either a low energy level or a high probability of being hit.

The acquired policy seen over the last 20 rounds seems equally likely to choose targets with the highest and the lowest distance. But contrary to the initial rounds, the probability is weighted very highly in this policy, and is actually the most important factor. Surprisingly the policy no longer favours targets with the lowest energy level over highest. This could easily be caused by the high exploration probability, which means that Aalbot often tries new actions and thereby a pattern cannot be seen.

Increasing k to a value of 1.2, the policy seems to choose its targets more or less randomly over the first 20 rounds. No factor stands out as very important, but targets close to Aalbot are chosen a bit more frequently than others.

Seen over the last 20 rounds, the policy appears more adapted. Targets with the lowest energy are definitely preferred and, surprisingly, so are the targets that Aalbot are least probable to hit, according to the targeting module. This could be seen as a natural consequence of the definition of when to reward Aalbot. Recall from Chapter 12 that Aalbot is given a reward if the robot currently targeted by Aalbot had died. This does not say anything about that Aalbot must be able to hit the target, but just select it as a target. This is of course a disadvantage, since Aalbot may miss its target and not get so many

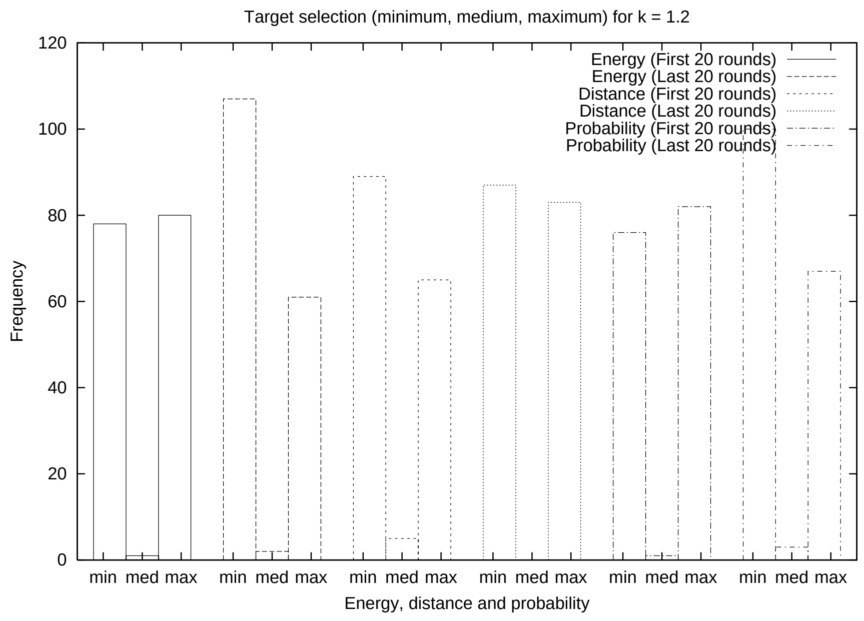


Figure 12.3: Starting with an empty Q-table and a value of $k = 1.2$, this figure illustrates the policy seen over the first and last 20 rounds of measuring performance. Notice that energy becomes the most significant parameter.

Robocode points as it could by selecting a closer target.

After a further increase of k to a value of 1.5 the policy seems already adapted during the first 20 rounds. As illustrated in Figure 12.4 targets with the lowest energy are clearly preferred over those with higher energy levels. An obvious tendency can also be detected with regards to the distance parameter, where the closest targets are clearly preferred. The targets with the lowest probabilities of being hit by Aalbot are, once again, favoured over targets with the highest probabilities.

The tendency to choose the closest target above the farthest has been decreased a bit when observed over the last 20 rounds. However, the other tendencies have become even stronger. This corresponds well to the high value of k, exploiting already known Q-values instead of exploring new (state, action) pairs.

12.5 Conclusion

Results from the target selection module have been discussed and evaluated in the previous sections. The discussion leads to a conclusion of how well reinforcement learning is applied in Aalbot. It has been shown that the constant k has as large impact on how the actions are selected. With values of $k \geq 1.5$ the energy parameter becomes very significant. It could be expected that if more

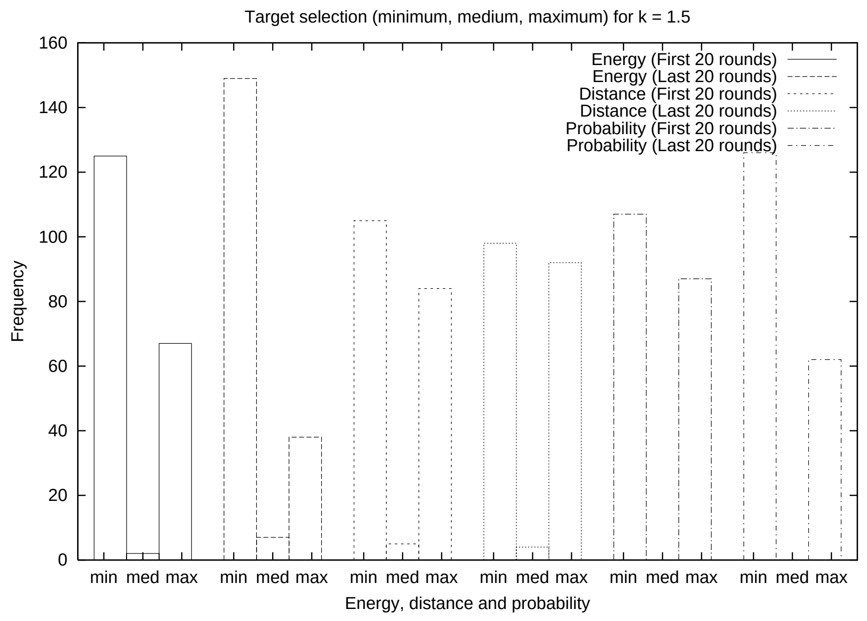


Figure 12.4: Starting with an empty Q-table and a value of $k = 1.5$, this figure illustrates the policy seen over the first and last 20 rounds of measuring performance. Notice that energy becomes the preferred parameter, and the other parameters become increasingly insignificant.

training rounds were run, the results of testing with a value of $k = 1.1$ could be interesting since this seems to be a good compromise between exploration and exploitation.

Chapter 13

Genetic Programming Evaluation

In this chapter the evolutions of the two genetic programming modules, movement module and radar control module, are evaluated. First the default training parameters and environment are specified. Secondly the results gained in the tests of the two modules are documented. Finally the possible explanations for those results are discussed.

13.1 Training Parameters

No exact rules exist for determining in advance which set of parameters such as mutation rate, deme size, etc. is the best for a specific task. Therefore the following parameters can be seen as a guess of at least reasonable values.

A set of default parameters were decided in Chapter 10 before the tests began. Some tests vary one of more of these parameters to determine their influence on the results. The default parameters used are:

- Number of demes: 1
- Deme size: 100
- Mutation rate: 30%
- Reproduction rate: 5%
- Crossover rate: 65%

Due to the time constaints on the project, the distributed version of the software was never fully developed and tested, and therefore the evolution runs have been limited to one deme. This means that the migration operator was not used.

The operators used are:

- **Mutation.** Substitutes a randomly chosen subtree with a new which is generated by the grow-tree method with minimum tree depth of 1 and maximum tree depth of 5.
- **Crossover.** One-point cross over.
- **Reproduction.** Individuals are copied unmodied to the succeeding generation.
- **Selection.** Fitness-proportional selection.

The initial population is generated using the ramped half-and-half methods with minimum depth 2 and maximum depth 6.

The fitness of each individual is calculated by running a single Robocode round with Aalbot against the enemies `sample.SpinBot`,`sample.MyFirstRobot` and `sample.Crazy`. The resulting score is the fitness value, though an extra fitness bonus of 250 is given when Aalbot is the last survivor. The reason for not running 10 rounds in a battle, as suggested in Subsection 10.1.4, is the time constraints of the project.

13.2 Test Environment

To parallelise the evolution process, four computers were used to run several evolutions at the same time. The evaluation of a single individual took about 30-80 seconds on a dual 2.4 GHz Xeon computer. This is due to the fact that in order to evaluate the fitness, a Robocode battle must be run.

The information collected for each generation is:

- The program code (an expression) of the best individual.
- Fitness of the best individual in the generation.
- Average fitness of individuals.
- Size (number of nodes) of the best individual in the generation.
- Average size of individuals.

These were chosen to enable visual inspection on the generated program code, so that the functionality of the genetic evolution programming can be tested. Also it is necessary to have the program code of the best individual in order to test it against the evaluation set of robots for the final evaluation as set forth in the analysis, see Chapter 4. The fitness of the best individual and average fitness are collected, as it is then possible to see if there is a trend in the fitness; i.e. is the population converging to an optimal solution. The size of the best individual and the average size are collected to make it possible to evaluate the results of the genetic programming using Occam's razor: Are the best individuals also simple?

13.3 Movement Module Results

Several different evolutions of the movement module were tried. In general they were tested with Aalbot using the following modules:

- Sensor Interpration
- Scanner Task
- Neural Network Targetting
- Reinforcement Learning Target Selection

The scanner task is a hand-coded module that simply rotates the radar 360 degrees forever. This was necessary, as the real radar control module is also built using genetic programming, and it was found infeasible to evolve both at the same time.

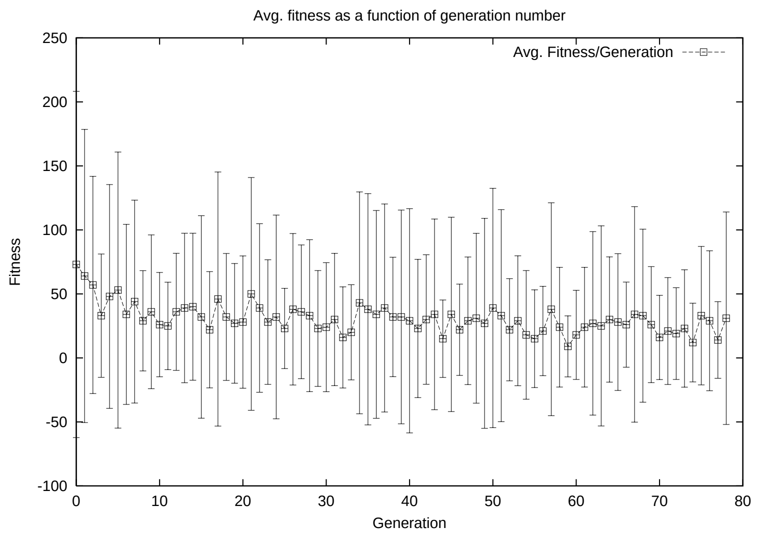
The anti-gravity module was not used in the tests, as the purpose of the genetic programming of the movement module is too see how well it performs compared to the hand-coded anti-gravity module.

13.3.1 Default Parameters

First, a test was made with the default parameters mentioned above. After 25 generations no clear improvements were seen in the average fitness or best fitness of individuals. Still, because of the relatively large deme size (100) the test progressed very slowly. Therefore a different test was started with the same parameters but instead with a deme size of 50. This test was run for 80 generations giving the results shown in figures 13.1 and 13.2. A third test was started similar to the second test and run for 30 generations. The results of this third test was similar to the second test, which makes it more likely that the results from the second test are not just the result of an unfortunate initial population or similar bad fortune.

The average size graph in Figure 13.1 shows, that the average fitness of the population does not grow as expected[1]. Neither are the best individuals in later generations any better than the best individuals from the first generations. The standard deviation bars in the average size graph shows, that the fitness of the individuals are spread out in a large interval, and even this does not get better as generations pass.

Comparing the two graphs in Figure 13.1 shows that even though the average fitness is quite low, the fitness of the best individuals are much higher. This means that some individuals are "super-individuals" that exhibit a much better performance than others. As these consistenly show up in even the initial



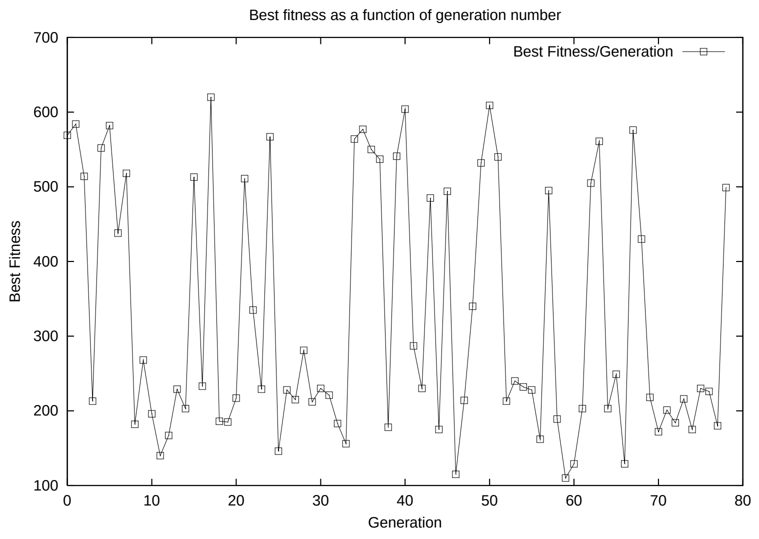


Figure 13.1: The average and best fitness of individuals in the test with default parameters.

generations, it can be theorised that it is quite easy to obtain a high fitness value in matches against the chosen sample robots.

Looking through the best individuals in each generation a pattern leaps to the eye. Most of them are really variations over the same general strategy, namely to move in circles or spirals. Some of the individuals are cluttered by extra superflous nodes that have no effect on the actuator, but stripping those away the pattern is seen again.

For example the best individual from generation 13 was this:

```
(TurnRight
  (Plus
    (Back (Div
            (SpinBotDist)
            (MyHeading)))
    (EnemyCount)))
```

This means that the individual should drive backforwards for a distance of the distance to the SpinBot robot divided by Aalbot's heading. And that it should turn right an angle of that previous amount plus the number of enemies left in the game. Essentially this is a circular motion.

The best individual from generation 34 was also the best individual in generations 35 and 36. The program code for this individual was:

```
(TurnLeft (Back (Time)))
```

That is, backing a distance of the current time, while turning left using that value as the angle. Again a circular motion, this time with the radius being enlarged proportional to number of ticks.

The top graph in Figure 13.2 shows a somewhat linear increase in the average size of individuals as generations pass. Individuals in generation 79 are on average more than twice the size of individuals in the first generations.

This happens even though the average fitness of the individuals neigher increases or decreases significantly. The increase in node count happens naturally through the evolutionary process, as the mutation operator inserts new possibly larger subtrees in individuals, and the cross-over operator can potentially substitute a very large part of one individual with a very small part of the other individual — altogether making larger individuals.

The fact that the average fitness does not change throughout the evolution, the increased size of individuals means that there must be useless nodes. This means nodes that are used either in a way that adds nothing to the behaviour of the robots, or nodes that are never executed. An individual with nodes that adds nothing to the behaviour of the robot is exemplified in the following individual from generation 23:

[1]The implementation of the evolution framework has been tested with sample problems, where the average fitness converged as expected.

```
(Plus
  (Plus
    (MyFirstRobotX)
    (Time))
  (Div
    (Minus
      (MyFirstRobotY)
      (SpinBotX))
    (CrazyY)))
```

This individual merely uses arithmetic functions on the believed coordinates of enemy robots as well as the current time. No actuator functions are used, so the individual is useless in that it exhibits the behaviour of just standing still.

Correspondingly an example of an individual with nodes that can never be executed would be:

```
(IfLessThan
  (SpinBotX)
  (Plus
    (SpinBotX)
    2)
  (Ahead 10)
  (Back 10))
```

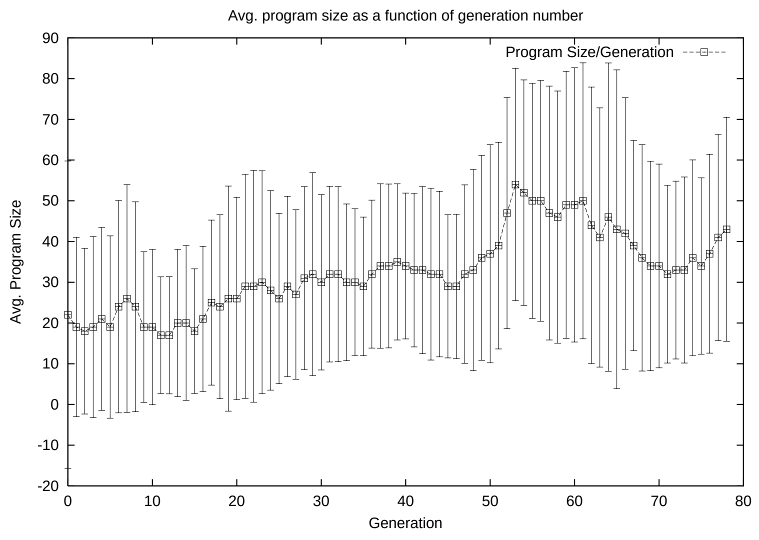
13.3.2 Size Penalised Fitness Function

In attempt to alleviate the problem with many useless or unused nodes, a test was run with with a modified fitness function. The default parameters were used as earlier, the deme size was 100 and the fitness function was modifed to be:

$$f_{\text{SizePenalised}}(h) = f(h) - \left\lfloor \frac{\text{nodeCount}(h)}{2} \right\rfloor$$

That is the new fitness function is simply the fitness value used earlier minus half of the number of nodes in the individual. Smaller individuals that perform as well as a larger individual thus gets a better score. This was done in attempt to see if forcing programs to be small (simple) would influence the evolution so that the fitness values got better according to Occams razor, as described in Subsection 7.1.2.

The results from this test are shown in figures 13.3 and 13.4. As before no real improvements can be seen in the average fitness over the 40 generations. Interestingly, the fitness values of the best individual from this test is superior



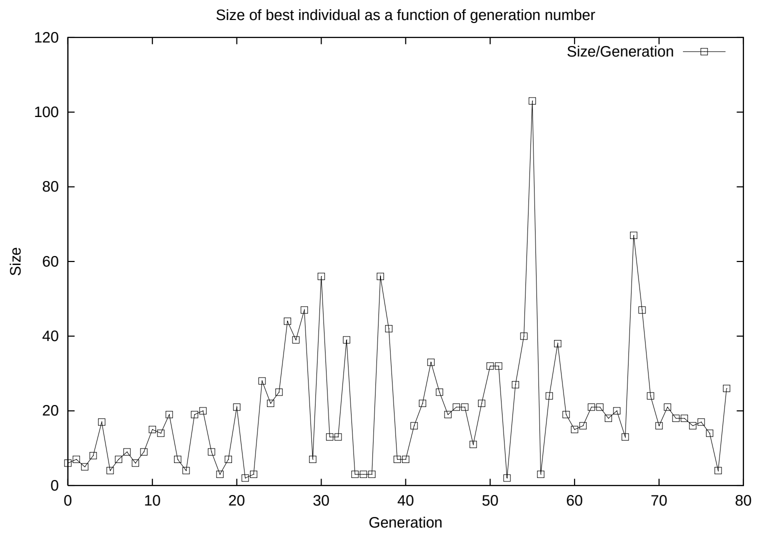
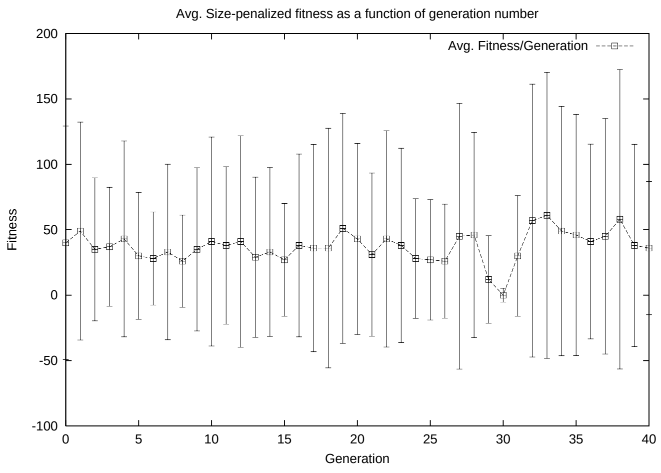


Figure 13.2: The average size and size of the best individuals in the test with default parameters.



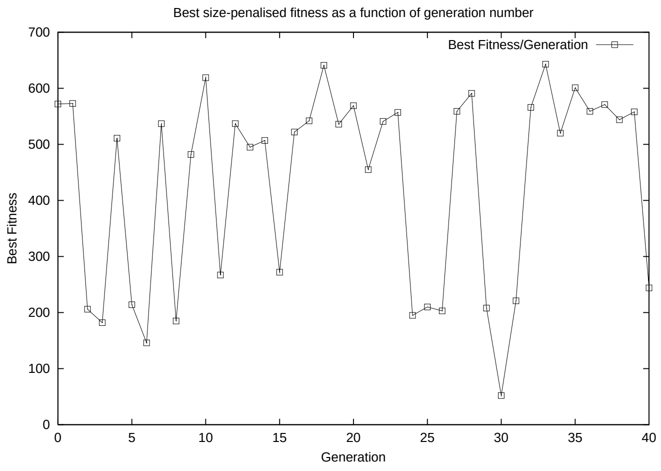
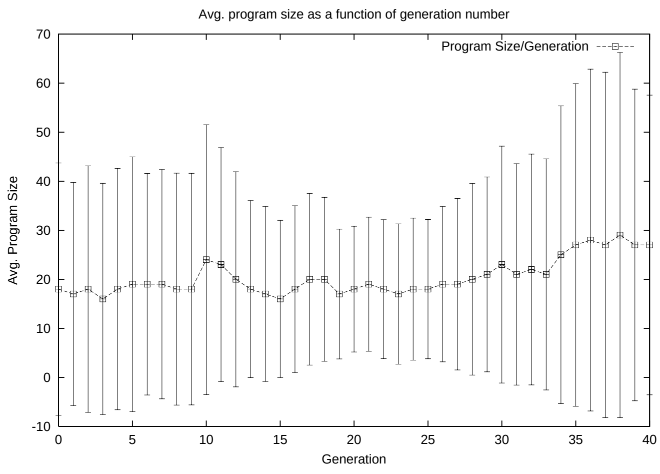


Figure 13.3: The average and best size-penalised fitness of individuals in the test with a custom fitness function.

to the best fitness in the earlier tests. The best fitness value occurs at generation 33 with a size-penalised fitness of 643. As this particular individual has 65 nodes, its Robocode score was 675. In comparison, the best fitness value observed in the two previous tests was 633. It is not significant enough to conclude though, that this could not be purely a chance due to for example a particularly fortunate initial population.

Comparing the graphs of average program sizes in figures 13.2 and 13.4 it can be seen that the average program sizes are in general lower over time with the size-penalised fitness function than with the regular fitness function. After 40 generations the average program size was 27 with the size-penalised fitness function, and 34 with the regular fitness function. Even though this is not a solid basis for strong conclusions, it seems that the numbers indicate that a size-penalised fitness function might be beneficial in keeping the individual size low while still maintaining a high fitness value.



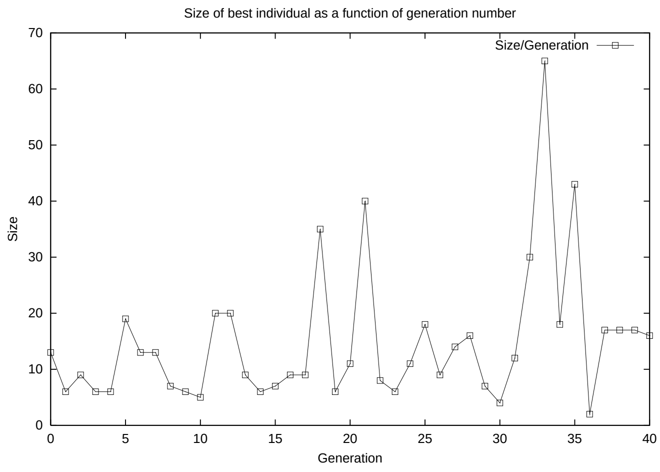


Figure 13.4: The average size and size of the best individuals in the test with a size-penalising fitness function.

13.3.3 Modified Parameters

In an attempt to investigate how the exogenous parameters influence the evolution, a test was performed with these parameters:

- Deme size: 25
- Mutation: 60%
- Reproduction: 10%
- Crossover: 30%

This more extensive use of the mutation operator could be hoped to produce a more diverse population that would ultimatively lead to an increased average fitness over time.

The results of this test is shown in Figure 13.5. Despite of the new parameter settings, the end result is the same – the average fitness seemingly stays at the same level over time. Also, no better best individuals were produced.

13.3.4 Modified Robot

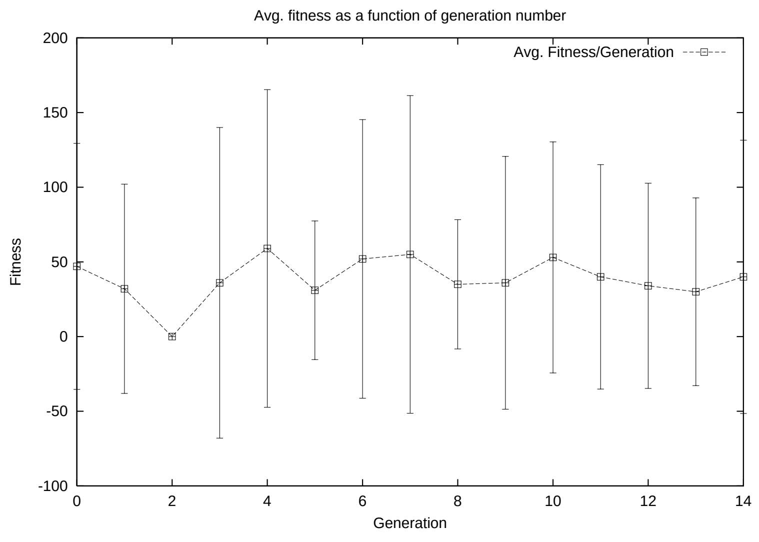
Due to a suspicion that the other modules in the robot were dominating the movement module, so that the efficiency of the movement module was of no real importance to the overall performance of the robot, a new test was devised. In this test the neural network targeting, reinforcement learning target selection and anti-gravity modules were disabled. Instead a module was made from the sources of a robot known as `SnippetBot`, which is a tutorial robot. This module simply shoots every time the radar scans another robot, without any special target selection or aiming methods.

This made the evaluation of a Robocode round faster, so the deme size was set to a medium value of 50. The results from this test can be seen in Figure 13.6.

Again, the average fitness of the individuals did not consistently improve over time. Though, worth noticing is the fact that the average fitness in this case is higher than in the previous test. Roughly speaking an average fitness of 150 compared to 50. The fitnesses of the best individuals were also distinct as they were larger than in the previous tests. The best fitness value was 704, compared to the best from the earlier test with fitness 675.

Examining the program code of the best individuals for each generation, one particular individuals leaps to the eye:

```
(Ahead
  (Random
    (TurnRight
      (Div
        (EnemyCount)
        (MyFirstRobotY)))))
```



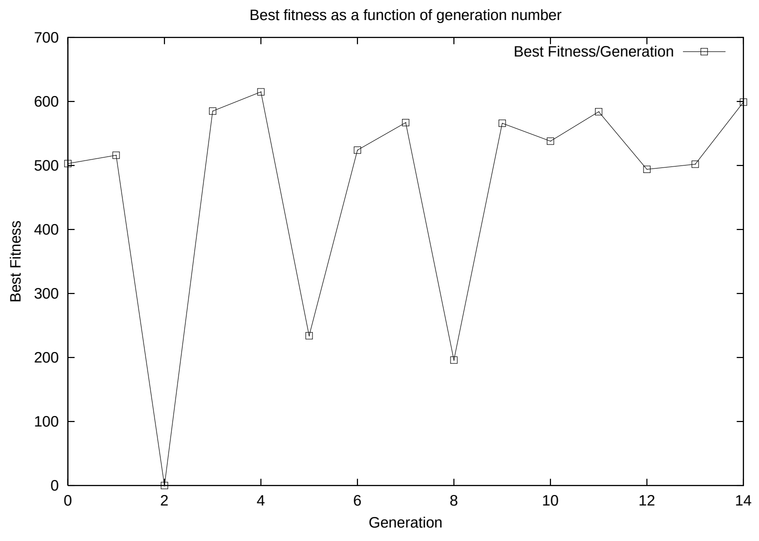
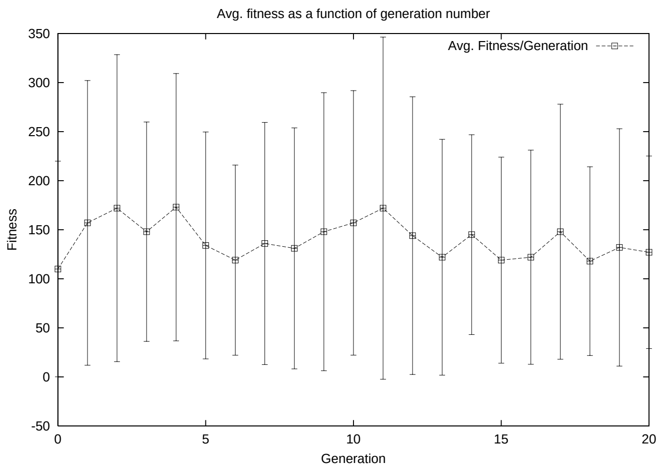


Figure 13.5: The average and best fitness of individuals in the test with modified parameters.



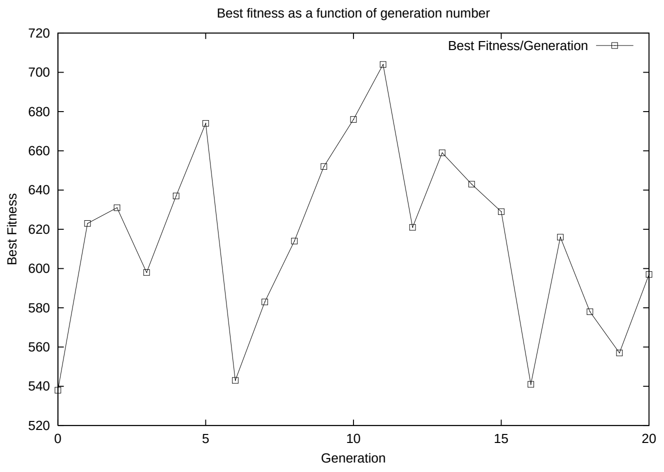
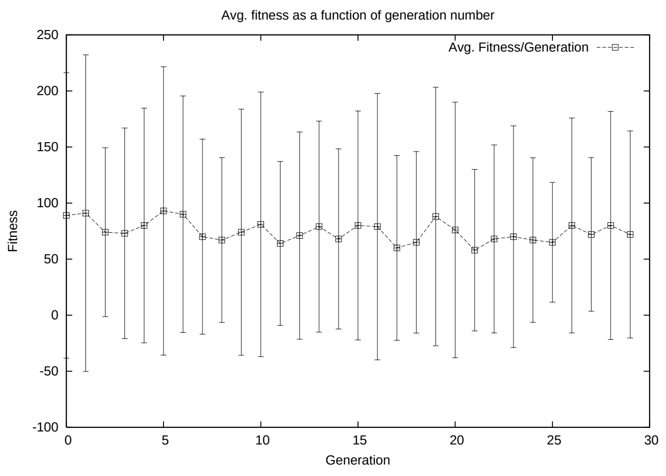


Figure 13.6: The average and best fitness of individuals in the test with no other machine learning modules.

This particular individual had the best fitness for four consecutive generations, namely generations 7, 8, 9 and 10. The program code shows the same idea as the previously examined program code, namely that it moves the robot in circular motions. The special behaviour here is that the radius of the circle is chosen at random every turn. It seems natural that this type of behaviour is hard for other robots to predict.

13.4 Radar Control Module Results

Because of the limited time available, the performance of the genetic programming radar module was only measured over a single test case with the parameters set to their default values, see Section 13.1.



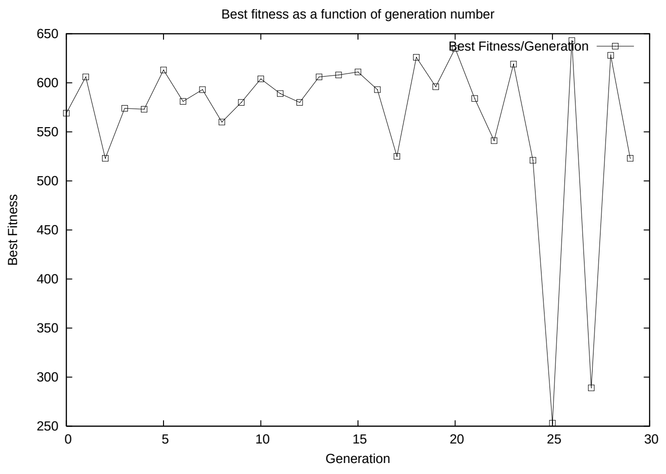


Figure 13.7: The average and best fitness of individuals in the radar control module test.

As can be seen in Figure 13.7, the fitness of the best individuals remains on a high level of approximately 500-600 through the entire evolution. Already in the initial, randomly generated, deme of individuals, an individual with a high fitness value is found. The best individual seen over all generations was found in generation 26 having a fitness value of 643. All the best individuals found exhibit a very simple radar controlling strategy, which is illustrated by the best individual of generation 5:

```
(TurnRadarLeft
  (Sequence
    (CrazyHeading)
    (SpinBotLastSeen)))
```

All this program does is simply to turn the radar left by some amount each turn, which corresponds to the basic radar controlling strategy described in Subsection 1.2.1. This strategy simply rotates the radar 360 degrees continuously through the entire game. The above program represents the general trend among the best programs found by genetic programming process, and as such terminals as e.g. the above `CrazyHeading` and `SpinBotLastSeen` only provide numerical arguments to either the `TurnRadarLeft` or `-Right` functions. The intentions of including these terminals representing information on the enemies is thus not employed to devise some complex and environment dependent radar controlling strategy as intended.

13.5 Conclusion

Generally for both the evolution of the movement module and the radar module was, that a steady increase in the average fitness of individuals never occured as was hoped for. This does not necessarily mean that the individuals were particular bad, just that it was not possible for the genetic programming process to make them better over time. This could happen for multiple independent reasons; these will be analyzed below.

One reason for this could be that a better hypothesis than the evolved individuals does not exist in the defined hypothesis space. That is, perhaps other types of terminals or functions were necessary to evolve an optimal movement or radar controlling strategy, or perhaps it was really necessary to include loops or function environments. During work with the movement module, other possible terminals were identified. For example it would probably be an advantage to have a terminal that states whether or not the robot is in a collision with an enemy.

On the other hand, the set of functions and terminals for the radar controlling module seems plentiful enough to build a good module. This could mean that the sheer amount of possible terminals and functions slowed down the evolution, as the evolutionary process had too many options to choose from. That could explain why many individuals were filled with useless expressions using the arithmetic functions for no purpose. Removing those functions from the language might have made the evolutionary process improve the individuals over fewer generations. Similarly it might have happened with the full language if the process had been continued for a longer period of time.

Another view on the results could be that these results indicate that the optimal movement strategy and radar control module has been found within the limitations of the hypothesis space. The workings of the other modules in the robot might have made Aalbot so efficient in battles against the chosen opponents that the movement module could never improve much on that. On the other hand, the other modules could have been so inefficient in battles, that their erraneous behaviour simply made it impossible for Aalbot to improve its fitness from better movement. A robot that moves well but targets very poorly will probably never succeed. The test run with the modified robot examined this, and the fitness was actually improved using other targeting modules than the original. However a steady increase in fitness did not occur here either.

A potential pitfall is the fact that the default parameters were focused on cross-over as the most important operator. The nature of the chosen language might mean, that cross-over operations very rarely produce meaningful offspring. Thereby the cross-over of two individuals with high fitness might in almost any case produce offspring with low fitness. This could be explained by the fact that the individuals with high fitness all seem to be very simple in structure with few nodes. Cross-overs on individuals with few nodes does not make as much sense as on larger individuals. This was explored using the modified parameters test, where the focus was put on the mutation operator primarily. This also failed to deliver steady increases in fitness values.

An area of interest was the structure and size of individuals. From Subsection 7.1.2 it was determined that it might be fruitful to focus on smaller individuals, as they should provide a more general solution to a problem without as many dependencies on specifics. Therefore the idea from Occam's Razor was sought encapsulated in a custom fitness function that penalised the fitness of larger individuals. The intent was to evolve smaller individuals that hopefully had fewer useless or non-used nodes, which could potentially improve the off-spring of cross-over operations. This was tested in the custom fitness function test, but sadly did not live fully up to its promising idea. The results did indicate that smaller individuals were produced on average, and that these individuals on average performed as well as the their larger counterparts evolved in other tests. In addition there was a slight indication that the best performing individuals it evolved were better than the ones from other tests. However, further tests are necessary to conclude anything as many random factors play a part in the evolutions, and as the results only cover few generations.

Chapter 14

Conclusion

The report is concluded in this chapter by an elaboration of the experiences and results gained by applying machine learning with the purpose of making self-improving autonomous robots for the Robocode framework.

14.1 Measurable Success Criteria

The analysis was concluded with a measurable success criteria for Aalbot, namely that it should score well in a combat against a chosen set of robots. The criteria in its entirety can be found in Chapter 4.

After completion of the implementation and training of Aalbot, it was tested whether or not it fulfilled the criteria. The results from one of the battles are shown below:

Robot	Total Score	Bullet Damage
Peryton	3166	1390
Aalbot	1800	850
SquigBot	1070	454
sample.Walls	720	399

This battle was run several times, and the above is representative for all the scores achieved. It can thus be concluded that Aalbot succeeded in fulfilling the stated goal.

14.2 Future Work

The work with Robocode was intriguing, but time was limited for this project. Therefore a lot of issues remain unsolved or unexplored. Suggestions for future work on a Robocode robot based on machine learning would be to explore some of the following topics:

- **Module structure.** The results of some of the machine learning methods suffered under the fact that the task of controlling the behaviour of the robot was analysed to great depths, and split into many small modules. It would be interesting to develop a full Robocode robot using just a single machine learning method such as genetic programming to control the entire behaviour of the robot, and thus obsolete the need for an extensive analysis of the game.
- **Genetic programming.** The evolution of the genetic programming modules was hindered by the fact that the implementation only supported evolution on a single computer. The enhancement of the system to allow parallelised evolution using demes on several computers could make it possible to improve the results achieved.
- **Fading Memory Maps.** The usage and complexity of the fading memory maps used in the implementation could easily be enhanced. Especially the modules in the robot could be changed to allow a stronger connection with the fading memory maps. Also the anti-gravity movement maps could be enhanced with the possibility of specifying a maximum effect radius for each gravity point to allow easier use by machine learning methods.

14.3 Recommendations

As recommendations, our general experiences in working with Robocode and machine learning will be summarised.

We found that the application of neural networks to targeting yielded reasonable results, although further explorations concerning the approaches not prioritised in this report could be pursued. The choice of neural networks is particular interesting as it allows for the creation of an adaptive robot.

Contrarily the use of genetic programming in Robocode did not yield the results expected from studies of the theory behind. Considerable computation time and many experiments are required in order to attain results comparable to traditional hand-coded robots. This is after all what makes it interesting to work with machine learning methods.

The use of reinforcement learning is promising, as it might be possible to obtain an adaptable robot by redefining the state space to be much smaller. However, the non-determinism involved in Robocode makes it complicated and the lack of good explanations of reinforcement algorithms handling non-determinism in the literature makes it worse.

Our general experience with Robocode is that it makes for an exciting way of getting results with machine learning that can be tested in practice. However, the lack of exact documentation about the Robocode internals, that Robocode is closed-source and the fact that it posed many limitations regarding thread usage, file usage, etc. made it a troublesome venture.

Bibliography

[AN01a] Mathew A. Nelson, 2001. `http://robocode.alphaworks.ibm.com/help/`.

[AN01b] Mathew A. Nelson, 2001. `http://robocode.alphaworks.ibm.com/docs/robocode/`.

[ASJ96] Harold Abelson, Gerald Jay Sussman, and Sussman Julie. *Structure and Interpretation of Computer Programs*. MIT Press, 2nd edition, 1996.

[Bea00] Zhang B. and Chen X. et al. Agent architecture: A survey on robocup-99 simulator teams. *Proceedings of the 3rd World Congress on Intelligent Control and Automation*, 2000. `http://wrighteagle.org/sim/paper/3705.pdf`.

[Ben98] Wolfgang et al Benzhaf. *Genetic Programming – An Introduction*. Morgan Kaufmann Publishers, Inc., 1998.

[Chr95] Bishop M. Christopher. *Neural Networks for Pattern Recognition*. Oxford University Press Inc., 1995.

[Eis03] Jacob Eisenstein. Evolving robot tank controllers. 2003. `http://www.ai.mit.edu/people/jacobe/research/robocode/genetic_tanks.pdf`.

[FOL] Free on-line dictionary of computing. `http://foldoc.doc.ic.ac.uk/`.

[FS97] Joy Frecthling and Laure Sharp. *User-Friendly Handbook for Mixed Method Evaluations*. Directorate for Education and Human Resources, NSF, 1997. `http://www.ehr.nsf.gov/EHR/REC/pubs/NSF97-153/START.HTM`.

[Koz92] John R. Koza. *Genetic Programming - On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[Li02] Sing Li. Rock 'em, sock 'em robocode!, 2002. `http://www-106.ibm.com/developerworks/java/library/j-robocode/`.

[Mar02] Rae Marsh. Secrets from the robocode masters: Dodge bullets, 2002. `http://www-106.ibm.com/developerworks/library/j-dodge/`.

[McC02] David McCoy. Secrets from the robocode masters: Factored wall avoidance, 2002. `http://www-106.ibm.com/developerworks/library/j-fwa/`.

[Mit97] Tom M. Mitchell. *Machine Learning*. MIT Press and The McGraw-Hill Companies, Inc, 1997.

[Mit02] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 2002.

[Owe02] Alisdair Owens. Secrets from the robocode masters: Anti-gravity movement, 2002. `http://www-106.ibm.com/developerworks/library/j-antigrav/`.

[Piv00] Marcus Pivato. *On Occam's Razor*. University of Toronto, Department of Mathematics, 2000. `http://citeseer.nj.nec.com/pivato00occams.html`.

[Roba] Robocode repository. `http://www.robocoderepository.com/`.

[Robb] Robo wiki. `http://robowiki.dyndns.org/`.

[Ros89] J. Kenneth Rosenblatt. A fine-grained alternative to the subsumption architecture for mobile robot control. *Proc of the IEEE Int. Conf. on Neural Networks*, 1989. `http://citeseer.nj.nec.com/rosenblatt89finegrained.html`.

[SA96] Franklin S. and Graesser A. Is it an agent, or just a program?: A taxonomy for autonomous agents. *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag*, 1996. `http://www.msci.memphis.edu/~franklin/AgentProg.html`.

[Sut98] Richard S Sutton. Reinforcement learning: An introduction. 1998. `http://www-anw.cs.umass.edu/~rich/book/the-book.html`.

[WJ95] Michael J. Wooldridge and Nicholas R. Jennings. Intelligent Agents: Theory and Practice. *Knowledge Engineering Review*, 10(2):115–152, June 1995.

Appendix A

Introduction to Neural Networks

The material presented in Chapter 2 and Chapter 8 concerns neural targeting and assumes a basic understanding of neural networks. Based on [Mit97, Chapter 4] a short introduction to this topic is now given, beginning with a discussion of neurons and their arrangement into networks. Subsequently an overview of network training is given with emphasis on the backpropagation algorithm and its derivation.

A.1 Architecture

Artificial neural network are inspired by biological learning systems, which are based on interconnected neurons in the brain. Artificial neural networks (hence forth referred to as neural networks) are typically applied to function approximation problems, and neurons are typically arranged into a number of layers. Figure A.1 shows an example of a two-layer neural network (a) and a detailed sketch of a single neuron (b). Neurons are depicted as circles, network inputs are depicted as unfilled circles and lines represent connections between neurons.

The layers are referred to as the *input layer*, *hidden layer* and *output layer*, from left to right respectively. Although three layers are present, the first layer only serves as inputs to the network and contains no actual neurons, so the network is only considered two-layered. Neural networks of two layers can represent any bounded continuous function, and with three layers any function can be approximated[Mit97, Section 4.6.2]. Hence networks with more than three layers are rarely used in praxis.

In Figure A.1 the layers are *fully connected:* each neuron receives input from every neuron in the preceding layer. All inputs have an associated weight, denoted w_{kj} for output layer neurons, as shown in the (b)-figure. Informally the weight specifies the contribution of the input to the neuron output, and implicitly to the final output of the network. Since all edges are directed from left to right, the network is *feed forward*; a property assumed by the backpropagation algorithm to be discussed later.

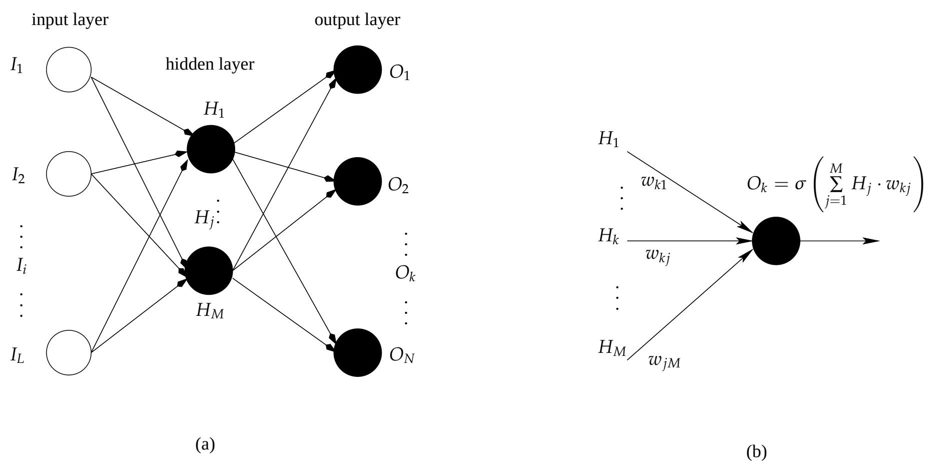


Figure A.1: Example of a fully connected, feed forward neural network (a) and a single neuron in the output layer (b).<

As implied by the figure, the following notation will be used:

- There are L input neurons, each representing a value, I_i.
- There are M hidden neurons, each calculating a function value, H_j.
- There are N output neurons, each calculating a function value, O_k.
- Weights are denoted by w_{ji} and w_{kj} for input layer and hidden layer neurons, respectively. E.g. w_{kj} denotes the weight to the neuron computing O_k from the neuron computing H_j.

Neurons will be referred to by the function values they compute, e.g. I_i refers to a neuron as well as a function value. When referring to the function computed by e.g. I_i, the notation $I_i\left(\right)$ will be used. Vectors will refer to all values in a layer, e.g. $\overrightarrow{H}$ is the vector of all outputs from the hidden layer.

Network computations proceed by inputting a real valued number to each of the L neurons in the input layer, and the result is given by the values of the N output neurons (to be formalized shortly). Hence the network can represents the following type of functions:

$$t : \mathbb{R}^L \rightarrow \mathbb{R}^N. \tag{A.1}$$

A.2 Computational Units

Computations are divided between each neuron of the network, so consider as an example the sketch of the single output layer neuron depicted in Figure A.1. It receives as input a vector, $\overrightarrow{H}$, which is weighed by a corresponding weight vector, $\overrightarrow{w_k}$. The neuron computes a function value, O_k, as follows:

$$O_k\left(\overrightarrow{w_k}, \overrightarrow{H}\right) = \sigma\left(\overrightarrow{w_k} \cdot \overrightarrow{H}\right) = \sigma\left(\sum_{j=1}^{M} H_j \cdot w_{kj}\right),$$

where the activation function, σ, defines an important characteristic of the neuron. Without an activation function, the network would be biased to represent linear functions only, which might not represent the target hypothesis sufficiently well. Furthermore the activation function can limit the output interval to a desired range.

σ is required to be differentiable, which will become apparent when network training is discussed in the next section. In praxis, the sigmoid activation function, $\sigma_{sig}\left(y\right) = \frac{1}{1+e^{y}}$, is often chosen and will consequently be used in the next section. This monotonically increasing function limits the range to $]0;1[$; in general, monotonicity is desirable because large absolute neuron inputs will always contribute to a higher neuron output, independently of other inputs, while smaller neuron inputs will give correspondingly smaller outputs. This is in contrast to e.g. $\sigma\left(y\right) = y^2$. When an output in the interval $]-1;1[$is required, the tanh activation function, $\sigma_{tanh}\left(y\right) = \frac{e^{y}-e^{-y}}{e^{y}+e^{-y}}$, can be used [Chr95, p. 127].

A.3 Training

A hypothesis represented by a neural network is defined by the network's structure (number of layers and neurons in each layer) as well as its weights and activation function. In this section the problem of determining the optimal weights based on a fixed network structure and activation function is discussed. This corresponds to a search through an incomplete hypothesis space, since the target hypothesis might not be representable by a network with e.g very few hidden neurons. Determining the optimal structure is often an empirical matter, although other approaches such as genetic algorithms have been applied [Mit02, p. 65-76]. In the next subsection network error representation is discussed followed by an outline of error minimization using backpropagation.

A.3.1 Error Representation

Neural network training is based on a training set, D, of pairs, $\left\langle \overrightarrow{x}, t\left(\overrightarrow{x}\right)\right\rangle$, where t is the target function that the network should be trained to represent; in other words, D is a set of inputs, $\overrightarrow{x}$, with associated outputs, $t\left(\overrightarrow{x}\right)$, known to be in conformity with t. Before training, network weights are typically initialized to small random values, and the network will consequently do a very bad job at approximating t. The training task now amounts to searching a hypothesis space consisting of all possible network weights, where the search is guided by minimizing the following error function:

$$E_d\left(\overrightarrow{w}\right) = \frac{1}{2} \cdot \sum_{k=1}^{M} \left(t_k - O_k\right)^2 . \tag{A.2}$$

Here E_d is the error on example, $d = \left\langle \overrightarrow{x}, t\left(\overrightarrow{x}\right)\right\rangle$, t_k is the kth components of $t\left(\overrightarrow{x}\right)$, and, as before, O_k is the kth output of the network when given $\overrightarrow{x}$ as input and with $\overrightarrow{w}$ as weights.

Intuitively E_d corresponds to the Euclidean distance between the target and actual output vectors, which seems sensible. This particular representation is convenient when minimizing the error function, which will become apparent in the following subsection. Note also that (A.2) is the error *per training example* as opposed to the accumulated error over all examples in D. This is the basis of the *stochastic* version of the backpropagation algorithm to be discussed next.

A.3.2 Backpropagation

Backpropagation is a common algorithm for searching the space of network weights in the endeavor to minimize E_d. It does so according to the gradient descent rule. The gradient, ∇E_d, is a vector defining the steepest slope in positive direction in a point on the error surface defined by E_d. Gradient descent exploits this by altering weights in all layers in the *negative* direction of the gradient. In the case of a hidden layer weight, w_{ji}, the following weight update is added:

$$\Delta w_{ji} = -\eta \cdot \frac{\partial E_d}{\partial w_{ji}}, \tag{A.3}$$

where the *training rate* η determines in how large steps the weight updates should be performed. The key idea in calculating this partial derivative is to express (A.2) as a composite function and applying the chain rule. In the following the derivation of the backpropagation rule is outlined for a network with two layers, as the one depicted in Figure A.1.

Let the weighed sum of all inputs to the output layer neuron O_k be denoted by $net_k = \sum_{j=1}^{M} w_{kj} \cdot H_j$. When considering neurons in the **output layer**, the actual output of O_k depends directly upon net_k, and the partial derivative is as follows:

$$\begin{aligned} \frac{\partial E_d}{\partial w_{kj}} &= \frac{\partial E_d}{\partial O_k} \cdot \frac{\partial O_k}{\partial net_k} \cdot \frac{\partial net_k}{\partial w_{kj}} \\ &= -\left(t_k - O_k\right) \cdot O_k\left(1 - O_k\right) \cdot H_j \\ &= -\delta_k \cdot H_j \\ &\quad \text{, where } \delta_k = \left(t_k - O_k\right) \cdot O_k\left(1 - O_k\right) . \end{aligned} \tag{A.4}$$

The first and third terms follow from the rule of sum differentiation, the second from the derivative of the sigmoid function. Note how the exact choice of E_d

in (A.2) results in a simple term containing both target and actual output of the network. Finally the error term, δ_k, is introduced for later convenience.

Next consider deriving the partial derivative of E_d with respect to weights to a neuron H_j in a **hidden layer**, and let net_j be defined analogous to net_k. The situation is complicated by the fact that output layer neurons do not depend directly on net_j in the hidden layer. Rather they depend on net_k in the output layer, which again depends on the outputs H_j from the hidden layer, which then depend on net_j. This reasoning leads to the partial derivatives of E_d for weights w_{ji} in the hidden layer:

$$\begin{aligned}
\frac{\partial E_d}{\partial w_{ji}} &= \sum_{k=1}^{N} \frac{\partial E_d}{\partial O_k} \cdot \frac{\partial O_k}{\partial net_k} \cdot \frac{\partial net_k}{\partial H_j} \cdot \frac{\partial H_j}{\partial net_j} \cdot \frac{\partial net_j}{\partial w_{ji}} \\
&= \sum_{k=1}^{N} -\left(t_k - O_k\right) \cdot O_k \left(1 - O_k\right) \cdot w_{kj} \cdot H_j \left(1 - H_j\right) \cdot I_i \\
&= \sum_{k=1}^{N} -\delta_k \cdot w_{kj} \cdot H_j \left(1 - H_j\right) \cdot I_i \\
&= -\delta_j \cdot I_i \\
&\quad \text{, where } \delta_j = \sum_{k=1}^{N} -\delta_k \cdot w_{kj} \cdot H_j \left(1 - H_j\right).
\end{aligned} \tag{A.5}$$

Here the error term δ_j is analogous to δ_k, which conveniently can be reused from output-layer computations and hence need not be recomputed. Note that a sum is applied because a weight in the hidden layer influences all output layer neurons.

It should be apparent that this approach can be generalized to networks with more than one hidden layer: For each additional layer, new error terms, δ_s, are introduced by computing a longer chain of dependencies. Each δ_s is then propagated back through the layers, hence the name *backpropagation*.

A.3.3 The Algorithm

The backpropagation weight update for each training example can now be summarized in the following natural language algorithm:

1. For each training example $\left\langle \overrightarrow{x}, t\left(\overrightarrow{x}\right)\right\rangle$:
   (a) Compute the actual network output $\overrightarrow{O}$ on input $\overrightarrow{x}$.
   (b) For each output layer neuron O_k:
      i. Compute the output layer error term δ_k as shown in (A.4).
      ii. Apply weight updates to each weight by adding $\Delta w_{kj} = \eta \cdot \delta_k \cdot H_j$ to w_{kj}.
   (c) For each hidden neuron H_j:

   i. Compute the hidden layer error term δ_j as shown in (A.5).
   ii. Apply weight updates to each weight by adding $\Delta w_{ji} = \eta \cdot \delta_j \cdot I_i$ to w_{kj}.

This procedure is typically repeated thousands of times until a stopping criteria is met, e.g. when the error is below a given threshold over all examples in the *validation* or *test set [Mit97, p. 110-111]*. In the case of Aalbot the training data is not static, and training is performed *online*; this is discussed further in Chapter 8.

Appendix B

Introduction to Reinforcement Learning

This appendix serves as an introduction to reinforcement learning and the basic concepts and notation related to it. An introduction to central keywords is given and mathematical definitions are presented. Following this the idea behind the Q learning algorithm is presented. This chapter is based upon [Mit97] and [Sut98].

In reinforcement learning the environment consists of different states. These states are chosen by the programmer and should yield a logical way to encapsulate a larger problem, e.g. moving a robot from one location to another in a fixed sized environment. This problem can be encapsulated by dividing the environment into different states. The task at hand is now to learn which states the robot should go through in order to reach its goal. This problem can be reduced to measuring how good a given state is for the robot, and then finally choose the chain of *good states* that leads to the goal. For the robot to learn how good a state is *rewards* are given.

The robot chooses an *action* given its *state*, possibly affecting the environment such that the robot perceives a change of state. The task is now: for any given state, choose the action that maximises the accumulated reward over time. This solves the problem of getting from one state to another.

B.1 Reinforcement Learning Notation

To solve the task given in the previous section the following notation will prove necessary.

A control policy, π, is a mapping from states to actions:

$$\pi : S \rightarrow A,$$

where S is the set of all states and A is the set of all actions, and $\pi(s) = a$ if a is the action selected in state s.

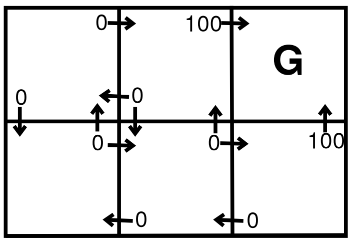


Figure B.1: An example of an environment with six states. The arrows between states denote a valid action. Rewards, given by the R_w function, are associated with each valid action in each state. G marks the goal state.

For the robot to learn the value of each (state, action) pair a reward function, R_w , is introduced. The reward function in the environment of the robot is illustrated in Figure B.1. The reward function maps from a given state s and an action a to a real value.

$$R_w \,:\, S \times A \rightarrow \mathbb{R}.$$

In order to calculate the resulting state, from applying action a in state s, a transition function, δ, is introduced. It is on the following form:

$$\delta \,:\, S \times A \rightarrow S\,,$$

where $\delta(s, a)$ is the resulting state from taking action a in state s.

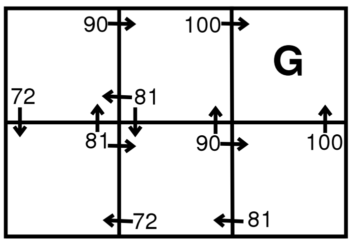
Recall that we want to learn a policy which produces the greatest accumulated reward over time. The accumulated reward $V^{\pi}(s_t)$ is achieved by following an arbitrary policy, π, from an arbitrary state. And is defined as follows:

$$V^{\pi}(s_t) = \sum_{i=0}^{\infty} \gamma^i R_w(s_{t+i}, \pi(s_{t+i})). \tag{B.1}$$

Here the sequence of rewards, $R_{w_{t+i}}$, is generated by starting in state s_t and recursively using the policy π to select an action (e.g. $a_t = \pi(s_t)$, $a_{t+1} = \pi(s_{t+1})$) and determining the next state $s_{j+1} = \delta(s_j, \pi(s_j))\,, \forall j \geq t$. The constant γ ($0 \leq \gamma \leq 1$) is used to weight the value of future rewards relative to immediate ones. Note that if $\gamma = 0$, future rewards are not considered. Contrary, as γ approaches 1, future rewards are weighted more heavily. (B.1) is often referred to as the *discounted cumulative reward,* because of the discounting of future rewards relative to immediate ones.

With a function describing the greatest accumulative reward, it is now possible to define an *optimal control policy*, $\pi^{\star}$, as:

$$\pi^{\star}(s_t) = \underset{\pi}{\text{argmax}} \; [R_w(s,a) + \gamma V^{\star}(\delta(s,a))]\;, \forall s_t \in S. \tag{B.2}$$



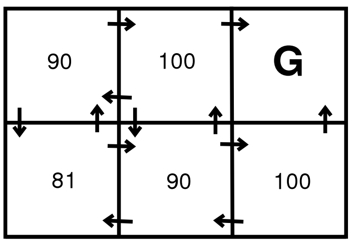


Figure B.2: Left: $Q(s, a)$ values for an optimal control policy for the robot. Right: $V^\star(s)$ values of such an optimal control strategy.

The *argmax* operator, used in (B.2), maximises the expression with regards to the argument, π . The value function of such an optimal policy, $\pi^\star$, will be denoted $V^\star(s)$. It maps from an initial state, s, to the maximum accumulative reward achievable by following $\pi^\star(s)$.

(B.2) implies that to learn the optimal policy, $\pi^\star$, it is only required to learn $V^\star$. The solution, however, has a limitation: Learning $V^\star$ requires perfect knowledge of both the transition function, $\delta(s, a)$, and the reward function, $R_w(s, a)$. This limitation renders the definition of $\pi^\star$ useless in problem domains where δ and R_w are not perfectly known to the robot. Fortunately, reinforcement learning provides a remedy, namely Q learning.

B.2 Q-learning

Not having perfect knowledge of the δ and R_w functions corresponds to the lack of training data in the form of $\langle s, \pi(s)\rangle$ pairs used in e.g. neural networks. The strength of Q learning is the capability to learn an optimal policy, $\pi^\star$ even under these circumstances. This builds on a function $Q : S \times A \to \mathbb{R}$.

$$Q(s, a) = R_w(s, a) + \gamma V^\star(\delta(s, a)) \tag{B.3}$$

Notice that $Q(s, a)$ is exactly the quantity that is maximised in (B.2). Because of that, the optimal policy can be formulated as:

$$\pi^\star(s) = \underset{a}{\text{argmax}}\ Q(s, a)$$

At a first glance this rewrite might seem useless, but it serves the purpose of showing that the optimal policy can be learned, merely by learning the Q function. Figure B.2 illustrates both the Q function and the optimal value function that ideally should be learned by the robot.

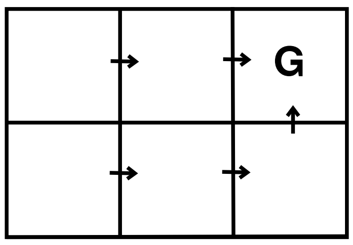


Figure B.3: An optimal control policy for the robot.

B.3 Algorithm for Learning Q

Learning the Q function can be accomplished by iterative approximation. This can be seen from the close relationship between the optimal value function, $V^{\star}(s)$, and the function $Q(s,a)$:

$$V^{\star}(s) = \max_{a'} Q(s,a')$$

where the *max* operator returns the maximum value of $Q(s,a')$ by varying a'. This relationship allows for rewriting equation B.3 to

$$Q(s,a) = R_w(s,a) + \gamma \max_{a'} Q(\delta(s,a), a') \tag{B.4}$$

Using equation B.4 it is possible to learn an optimal policy by iteratively performing an action, a, observing the resulting reward, $r = R_w(s,a)$, and the new state, $s' = \delta(s,a)$. At any given time the learner has an estimate, $\hat{Q}(s,a)$, of the actual Q function. At each step of the iterative approximation the current estimate is updated for the previous (state,action) pair as follows:

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a').$$

This approach can be considered as successively sampling the δ and R_w functions one step ahead from the current state s. Theorem 13.1 in [Mit97] states that performing the iterative approximation, as described in Algorithm 1, ensures convergence towards an optimal control policy.

Algorithm 1 The algorithm for learning Q

```
For each $s, a$ initialise the table entry $\hat{Q}(s,a)$ to zero
Observe the current state $s$
Do forever:
  • Select an action $a$ and execute it
  • Receive the immediate reward $r$
  • Observe the new state $s'$
  • Update the table entry for $\hat{Q}(s,a)$as follows:

        $\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$

  • update $s \leftarrow s'$
```

B.4 Action Selection

The problem of learning an optimal control policy seems related to the problem of finding a reasonable approximations to a target function in other machine learning paradigms (e.g. neural networks). However an interesting aspect has to be taken into consideration when speaking of Q learning, namely how to perform action selection.

Action selection is the process of choosing an action, a' , given the Q values for all actions $a_j \in A$ in a state s. Different approaches have been devised for selecting actions, as described in [Mit97, 379] and [Sut98]. Two of these are presented here.

(B.5) illustrates an action selection scheme often referred to as *greedy* in the literature. It simply selects the action with the highest Q value.

$$a' = \underset{a}{\operatorname{argmax}}\ Q(s,a) \tag{B.5}$$

Another approach for selecting actions are the calculation of weighted probabilities of the Q values in the given state. (B.6) shows how such probabilities are calculated. This approach introduces two interesting subtleties to Q learning, namely *exploration* and *exploitation*.

$$P(a_i|s) = \frac{Q(s,a_i)}{\sum_j Q(s,a_j)} \tag{B.6}$$

Using only the greedy action selection from (B.5) might lead the learners estimate, $\hat{Q}(s,a)$, to converge to a local optimum. This is because the choice of initial actions has a huge impact on the future accumulated reward. This action selection scheme maximises the cumulative reward over time, by always exploiting the current estimate, $\hat{Q}$, hence the name *exploitation*.

Using the probabilistic approach from (B.6) ensures that even though an action in a given state, s , has been rewarded, the other actions in s will have a chance of being selected in the future. This can help the current estimate, $\hat{Q}$, to overcome a local optimum and continue the search for a global optimum.

Appendix C

Introduction to Genetic Programming

This appendix provides an introduction to the theory of genetic programming. First, the background of genetic programming is introduced followed by a general description of how to apply genetic programming to problem solving.

C.1 Background

Genetic programming can be considered as a new field in computer science. It was first presented by John R. Koza at a conference in 1989, but relies heavily on the concept of *genetic algorithms*, which was invented by John Holland in the 1960s.

Both genetic algorithms and genetic programming are inspired by the biological theories of genome evolution. Genetic algorithms most often work on simple bit strings (genetic strings) each representing a possible solution to some problem. A *population* of bit strings is evolved over several *generations* by measuring each string's capability of solving the task and only allowing the best *individuals* to survive and reproduce themselves into the next generation. This idea originates from the Darwinian principle: "Survival of the fittest".

In genetic programming a population of *computer programs* is evolved over generations, and after a number of generations a program which solves (or approximately solves) a specific problem will occur among the programs in the current population. This holds only under the assumption that a solution in fact exists and that the evolutionary process is given enough time to find it.

Genetic programming removes the constraints of genetic algorithms with respect to the limited amount of representational power that lies in a relatively short bit string. As the problems get more complex, a more powerful representation in form of a computer program becomes the natural choice. Contrary to a simple bit string, a computer program can easily utilise the concepts of procedures, functions, iteration and recursion, that might be necessary to solve some types of problems. Representing such concepts in a simple bit string is of course possible, though very tedious. A general problem, or obstacle one might say, is that when working with machine learning techniques it is often necessary to specify the size and shape of the solutions. This is not the case with genetic programming, where size and shape of the program resulting from generations of evolution is not known in advance. [Koz92, p. 63]

Genetic programming is considered to be a domain-independent technique for solving a broad range of problems and has been successfully applied in many cases. For example, good results have been achieved within the fields of cellular automata, space satellite control, molecular biology, design of electrical circuits and robot control. [Mit02, p. vii]

C.2 Applying Genetic Programming

When applying genetic programming to some problem, the following five preparatory steps must be taken:

1. First, the set of *terminals* in the problem domain must be determined. Terminals can be thought of as variables and constants or as the inputs available to the computer programs in the population.

2. A set of *functions* related to the problem also needs to be specified. In addition, certain basic and domain-independent functions such as arithmetic functions, conditionals etc. must also be included in the function set. The functions together with the terminals can be considered as the building blocks from which a program in the population is constructed. With this in mind, the idea of genetic programming can be restated as a search within the space of all possible combinations of functions and terminals for the program that solves the problem. The property of closure of functions must exist on the sets of functions and terminals, as any program constructed should be syntactically legal. That is, a function must be able to take any other function or terminal as an argument. Programs consisting of the chosen functions and terminals needs to be represented in some programming language. A functional language would seem to be the natural choice, and in fact many applications of genetic programming uses the LISP programming language.

3. Some performance measure of evolved programs must be specified, because as mentioned above, only the best programs (or individuals) in a population are allowed to reproduce into the following generation. The measured performance of an individual is called the *fitness* of the individual. The fitness of a program could for example be a numerical measure of the difference between the program's output and the desired output. In this case, evolution of the programs should continue until a

program with a fitness of zero or close to zero is found. In problem domains involving games, the fitness of a program could be equal to the points scored, implying that the evolution should breed programs with high fitness values. When dealing with problems that exhibit randomness in some manner, a number of fitness cases can be used to measure the fitness of a program under different circumstances. A more general and representative value of a program's fitness can be obtained in this way.

4. Several parameters need to be specified before applying genetic programming. According to [Koz92, p. 115-116], the two main parameters are the population size, M, and the maximum number of generations, G. However, one can question the importance of the last mentioned parameter G, as it would seem more natural to limit the process by some specified time bound or a satisfactory fitness value. Moreover, there exist a lot of other parameters influencing how to perform the evolutionary steps between generations. These parameters have minor significance and can only influence the time spend on the evolutionary process by some relatively small amount. Hence they will not be described in further detail.

5. A termination criterion should be stated as the point at which the evolutionary process has produced a satisfactory result. The termination criterion could more specifically be the point at which one of the programs in the population is found to have some adequate fitness. Another common used method is to register the fittest individual found from generation to generation, and then returning the *best-so-far* individual after G generations.

C.2.1 Genetic Operations

In genetic programming, the first generation of programs is randomly composed using the available functions and terminals. These initial programs are not likely to exhibit a worthy fitness with respect to the problem to be solved, but still some of the programs will have better fitness than others. Genetic operations are then applied to these programs, which are selected based on their fitness. More precisely, the genetic operations work on the parse trees of the programs, and hence programs will be represented as parse trees in the following.

The first of two primary genetic operations is the *reproduction* operation. It works by selecting an individual from the population based on its fitness, and then copying the individual directly, i.e. without alterations, into the next generation. The individual is most often selected using fitness-proportionate selection, which means that an individual I_j is reproduced with the probability:

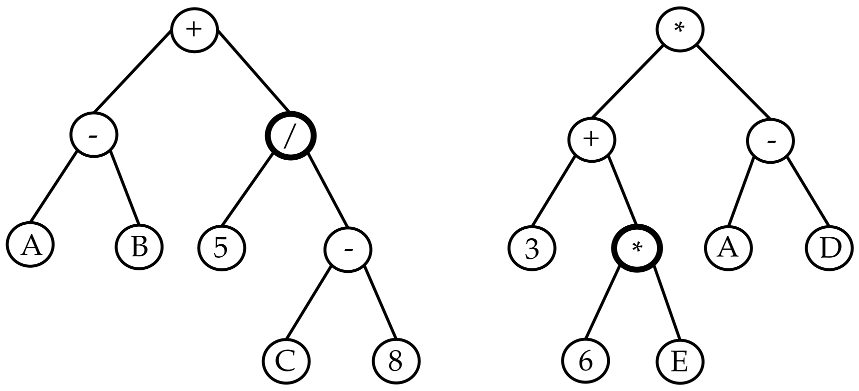


Figure C.1: The two parental programs selected for crossover represented by their parse trees and with randomly chosen crossover points shown in bold.

$$P\left(I_j\right) = \frac{f\left(I_j\right)}{\sum\limits_{i=1}^{M} f\left(I_i\right)}$$

where $f\left(I_j\right)$ is the fitness of individual I_j in the population at some generation, and M is (as mentioned earlier) the number of individuals in the population.

The other of the two primary genetic operations is the *crossover* operation. It works by selecting two parental individuals based on fitness and then produces two offspring individuals to be placed in the next generation. To illustrate, assume that the two simple arithmetic programs whose parse trees are shown in Figure C.1 are chosen to take part in a crossover operation.

A crossover point in each of the programs is selected randomly, indicated by the bold nodes in the figure. The subtrees originating at the crossover points are then swapped producing the two offspring programs shown in Figure C.2, where the newly swapped-in subtrees are bolded for clarity.

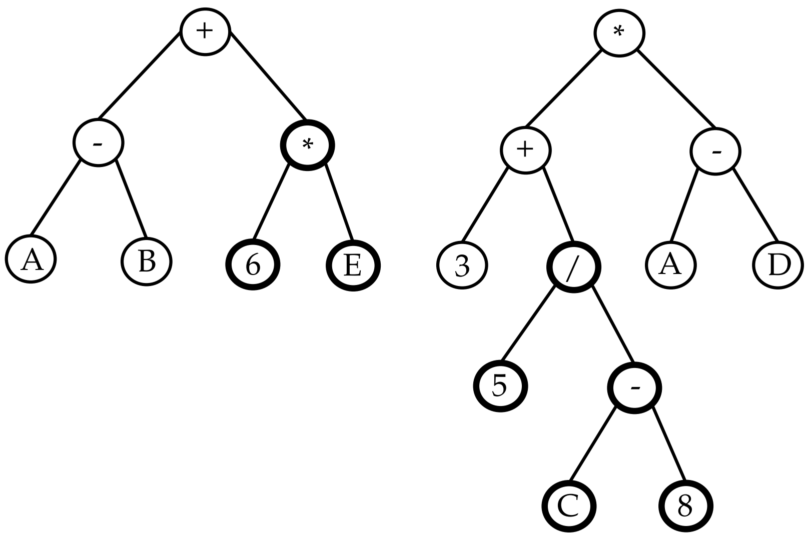


Figure C.2: The two offspring programs as a result of the crossover operation.

Reproduction and crossover are the two most used genetic operations in the evolutionary process, even though a few other operations exist. Worth mentioning is the *mutation* operation, where a program is selected from the population, a random mutation point of its parse tree is chosen and a new randomly generated tree is inserted at this point. Similarly there is a *permutation* operation, that randomly chooses a point of the selected program, and then permutes the arguments of the function at the chosen point. [Koz92, p. 105-106]

C.2.2 The Algorithm

To summarise, a natural language algorithm describing the process of genetic programming is given below: [Koz92, p. 77]

1. Randomly generate an initial population of *M* programs composed from the set of functions and terminals.
2. Repeat the following steps until the termination criterion is satisfied.
   (a) Run each of the programs in the population and measure its fitness, possibly over a number of fitness cases.
   (b) Apply the genetic operations (reproduction, crossover etc.) to programs selected from the population based on their fitness. Exactly which operation to be used could be selected probabilistically, where each operation in question has a fixed probability. (The crossover

operation typically has the highest probability among the operations.) Apply the genetic operations until M programs has been produced.

(c) Then create a new population containing the newly produced programs and increment the variable counting the generation number.

3. Return the program satisfying the termination criterion. This could e.g. be the best-so-far individual after G generations.