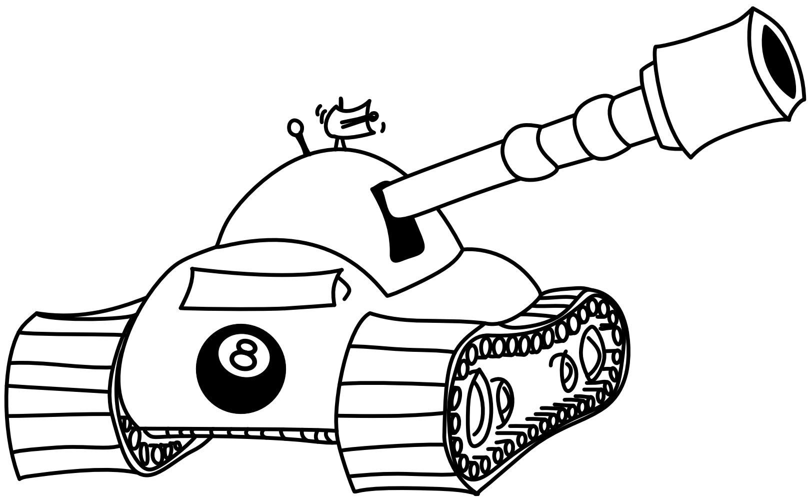
Robocode

Development of a Robocode team.



E1-209

Jens Frøkjær
Palle B. Hansen
Martin L. Kristiansen
Ivan V. S. Larsen
Dan Malthesen
Tom Oddershede
René Suurland

December 2004

The University of Aalborg
Department of Computer Science

Aalborg Universitet
Department of Computer Science, Frederik Bajers Vej 7E, DK 9220 Aalborg Øst

Title:

Robocode – Development of a Robocode team

Project period:
Dat3, Sep. 2th - Dec 17th, 2004

Project group:
E1-209

Group members:
Jens Frøkjær
Palle B. Hansen
Martin L. Kristiansen
Ivan V. S. Larsen
Dan Malthesen
Tom Oddershede
René Suurland

Supervisor:
Jens Dalgaard Nielsen

Copies: 9

Report page count: 112

Abstract:

This report describes the steps taken to develop a Robocode team which uses bayesian network, genetic algorithm and neural network for evolving in Robocode.
The main goal of the project is to produce a working Robocode team. The "Decision support systems and machine learning" course is used for this project.
The report comprises three main parts: I) Analysis, II) Design, III) Implementation, Test, and Conclusion.
The analysis describes different technologies needed for building a robot and an analysis of the Robocode environment.
The Design specifies a general framework of the implementation as well as the design of the different machine learning systems.
The last part consists of the three chapters: Implementation, Test, and Conclusion. The Implementation chapter describes the important parts of the implementation process. The Test chapter outlines the test of the machine learning in order to see if it actually learns anything.

Preface

This report is the result of a DAT3 semester at the university of Aalborg. The project uses the course “Decision support systems and machine learning” as PE-course. This project is study-oriented with no commercial or economic interests involved, thus the main object of the project is to develop a functional program.

Reference resources are marked with [*number*]. The corresponding *number* can be found in the Bibliography in the back of the report.

The web-site “www.cs.aau.dk/∼fr0/” contains: Source code, JavaDoc, a copy of the report, and a copy of the compiled robots.
We would like to thank our supervisor, Jens Dalgaard Nielsen, for assistance during the project period.

Aalborg, December 2004

Jens Frøkjær *Palle B. Hansen*

Martin L. Kristiansen *Ivan V. S. Larsen*

Dan Malthesen *Tom Oddershede*

Rene Suurland

Contents

I

Analysis

1 Robocode

It seems agents are everywhere today. Agent implementations are used in a lot of systems that need to somehow make *intelligent* decisions, for example biological systems, industrial robots, and aircrafts. An agent is called autonomous when it is able to map received input to output in the form of an action or a specific diagnosis without being controlled by an external authority[9]. The use of autonomous agents is indispensable in many situations. Consider a robot that is used for exploring the surface of the planet Mars. Such a robot would benefit from having some kind of decision support system, because every time it encounters an obstacle it would be terribly inefficient had the robot have to waste precious time messaging Earth asking for instructions. This would be prevented if the robot had a system that could decide what to do based upon a number of input variables. For this project, machine learning and decision support mechanisms will be applied in the development of autonomous agents for a simulated battlefield, called Robocode.

1.1 Introduction to Robocode

Robocode is an environment in which virtual robots, developed in Java, can battle against each other. The robots simulate tanks in a battlearena, and in order to find other robots they are equipped with radars. A robot can move forwards and backwards at different speeds and turn left and right. The radar and turret can be turned left or right independently of each other and the rest of the tank. And finally, the gun can be fired.

When setting up a battle, it is possible to watch the battle played out on the screen, or just letting the computer simulate the battle without showing the graphics. The latter will complete the battles faster, because the battle does not have to be rendered on the screen.

When an enemy robot is spotted with the radar, an event is generated, and the appropriate action can be taken by our robot. It is possible to get information about the robot being spotted, such as velocity, heading, remaining energy, name, the angle between the heading of your own robot and the robot being spotted, and the distance to that robot. During the game these pieces of information will form the basis for the actions to be taken by our robot. For example when spotting an enemy robot, the gun could simply be told to fire. But in which direction is the turret currently pointing? Obviously, the gun has to be pointing in the direction of the robot you want to fire at. But with knowledge about the enemy robot's heading and speed, there are further considerations that can be taken into account when computing the direction in which the gun should be fired, because you have to compensate for the fact that the enemy is on the move. Such considerations will optimize the chances of hitting the target.

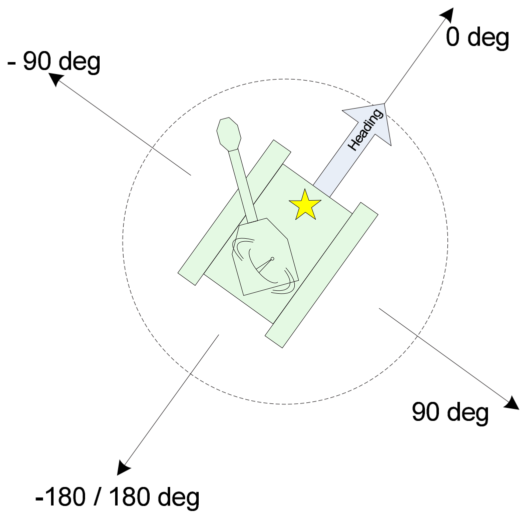
Robots in Robocode can battle against each other in teams. By communicating with each other, they can exchange information about where they have spotted opponent robots etc. And based upon a chosen strategy, a robot might choose to run away from opponents or perhaps letting your team gather round an opponent robot, and try to take it out.

The purpose of this chapter is to describe the environment in which the robots of Robocode fight. In order to understand this world it is necessary to understand the laws of physics that control it.

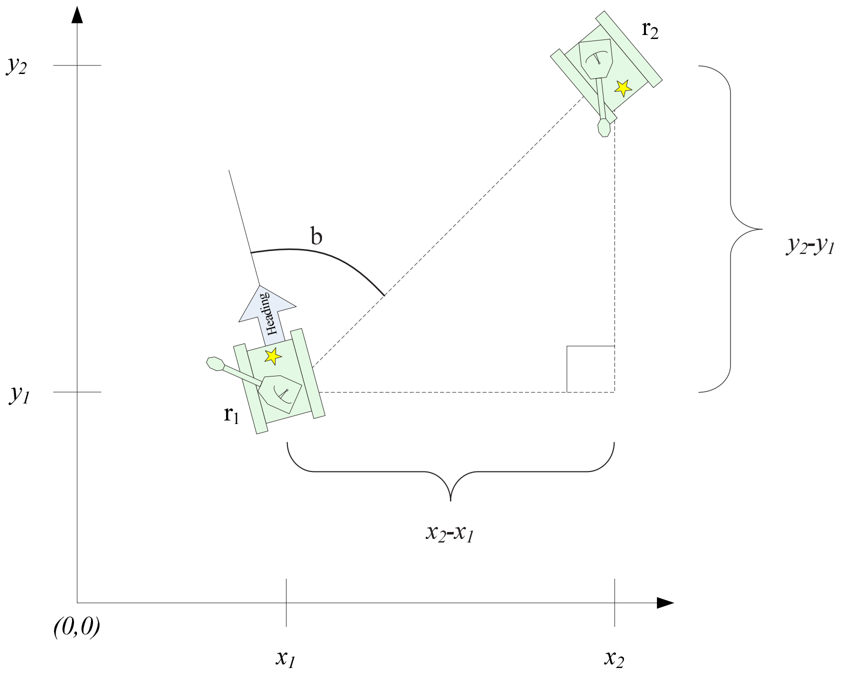
1.2 The Battlefield

The battlefield is where all the fights take place. It is a two-dimensional plane surrounded by walls. A position on the battlefield is a pair of coordinates (x, y). The origin $(0, 0)$ of the battlefield is placed in the left bottom of the battlefield, see Figure 1.1(b). This means that a coordinate (x, y) on the battlefield will always be a pair of positive real numbers.

The following methods from the Robocode API can be used for getting information about the size of the battlefield and the position of robots. These methods can be found on the `Robot` class:



(a) Shows the concept of bearing



(b) Shows the bearing $b = 60°$ between two robots, r_1 and r_2

Figure 1.1: Bearing

- `getBattleFieldHeight()`
- `getBattleFieldWidth()`
- `getX()`
- `getY()`

1.3 Heading and Bearing

In order for the robots to know the position and direction of enemies, they must have some data to calculate these from. This is where the heading and bearing are vital. The following sections will describe how heading and bearing are implemented in Robocode.

Heading

When a robot is moving, it has a heading. This heading can be acquired for robots in Robocode using the `getHeading()` method on an instance of the `Robot` class. We can acquire the heading of our own robot as well as other robots being scanned by the radar.

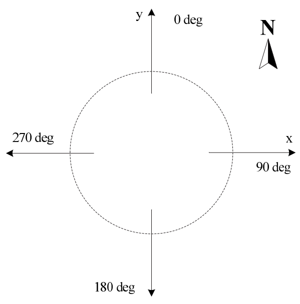
The heading in Robocode is measured clockwise from 0° to 360°. An important thing to note is that a heading of zero degrees is achieved when the robot is facing north on the battlefield, see Figure 1.2(a). Usually, in mathematics, an angle of zero degrees means that the robot is facing east on the battlefield, see Figure 1.2(b). This means that using normal trigonometry functions like $\sin(x)$, $\cos(x)$, and $\tan(x)$ will not yield the expected result.

If one wants to convert from conventional degrees n into Robocode degrees r, Equation (1.1) can be used.

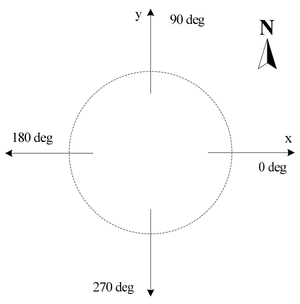
$$r = \begin{cases} 90 - n & \text{if } n < 90 \\ 450 - n & \text{if } n \geq 90 \end{cases} \tag{1.1}$$

Bearing

Bearing is a direction in the interval $-180°$ to $180°$ relative to your current heading. A bearing from 0° to $-180°$ will be on the left side of the robot and a bearing from 0° to 180° will be the right side of the robot, see Figure 1.1.



(a) Robocode



(b) Conventional mathematics

Figure 1.2: Shows the difference between the orientation of angles as they are used in Robocode and conventional mathematics.

1.4 The Environment

This section will explain the most important parts of the Robocode environment. It is important to understand the environment in which the robots act, in order to implement a working robot.

Let us start by explaining how Robocode handles time and distance.

Time is measured in *ticks*. At the start of a game the tick count is set to 0 and all robots are told to execute their commands. After a set time interval all robots are halted and the tick count is incremented. If a robot has not executed all its commands when this happens, the commands are simply discarded. This then repeats itself until the game ends. One tick = one frame = 1 turn. When a new round begins the tick counter is reset.

Distance is measured in pixels. The smallest possible battlefield in Robocode is 400×400 pixels and the largest is 5000×5000 pixels.

1.4.1 Main Loop

To control the game, Robocode has a built in loop, called the Battle Manager, see Figure 1.3. The Battle Manager basically works as a program that uses the different robots as plug-ins. Every robot has its own thread, which has its own event queue. Events are generated whenever something happens over which we have no direct control. This could be when our robot collides with a wall, an enemy is scanned with the radar, etc. These events are put in the

```
• All robots execute their event-queue.
• The time is increased by one.
• All bullets are moved.
• Collisions are checked.
• All robots are moved in the following order:
    – Heading
    – Acceleration
    – Velocity
    – Distance
• The robots perform scans and all team communication is done.
• The battlefield is updated.
```

Figure 1.3: Pseudo code of the battle manager

event queue by the Battle Manager. The events are then carried out by the robot itself. This separation of the game and the robots has been made to prevent poorly programmed robots from taking the game down by entering a dead lock or causing some other fatal error to occur.

1.4.2 Power and Energy

The concept of power and energy is central to the Robocode game. A robot is given an amount of energy at the start of every game, and a robot can then lose and gain energy during the game. Energy is gained if your bullets hit other robots and energy is lost if a bullet hits your robot. A typical robot has energy 100 at the start of a game.

Power is the amount of energy put into a bullet when fired. This means that a robot will lose energy when firing a bullet. The more powerful the shot, the more energy will be lost. There is a one-to-one relationship between power and energy, meaning that a shot given power 0.5 will subtract 0.5 from the energy of the robot. Equation 1.2 describes this relationship.

$$\Delta e_{shot} = -p \tag{1.2}$$

Power p is a defined as a range

$$p = [0.1; 3.0] \tag{1.3}$$

Equation 1.4 describes the change in energy Δe_{gain} of a robot, if one of its bullets hits another robot. p is the power given to the bullet.

$$\Delta e_{gain} = 3 \cdot p \tag{1.4}$$

The amount of energy our robot currently has can be acquired by calling the `getEnergy()` method on the `Robot` class.

Equation 1.5 describes the change in energy Δe_{lost} of a robot, when it is hit by a bullet with power p.

$$\Delta e_{lost} = \begin{cases} -4 \cdot p & \text{if } p \leq 1 \\ -4 \cdot p - 2 \cdot (p - 1) & \text{otherwise} \end{cases} \tag{1.5}$$

The amount of energy a robot loses when driving into another robot is 0.6.

Equation 1.6 describes the speed of a bullet s as a function of the power p given to the bullet. $s(p)$ is measured in $\frac{\text{pixels}}{\text{tick}}$.

$$s(p) = 20 - 3 \cdot p \tag{1.6}$$

Equation 1.7 describes the change in energy Δe_{lost} of a robot when driving into a wall with velocity v.

$$\Delta e_{lost} = -|v| \cdot 0.5 + 1 \tag{1.7}$$

Equation 1.8 describes the amount of heat generated when firing a bullet with power p. The gun will not be able to fire again until it has cooled down (gunheat = 0) from firing the last bullet.

$$\text{gunheat} = 1 + (p \cdot 5) \tag{1.8}$$

Equation 1.9 describes the time before a gun is able to fire again.

$$\text{ticksToFire} = \frac{\text{gunheat}}{0.1} \tag{1.9}$$

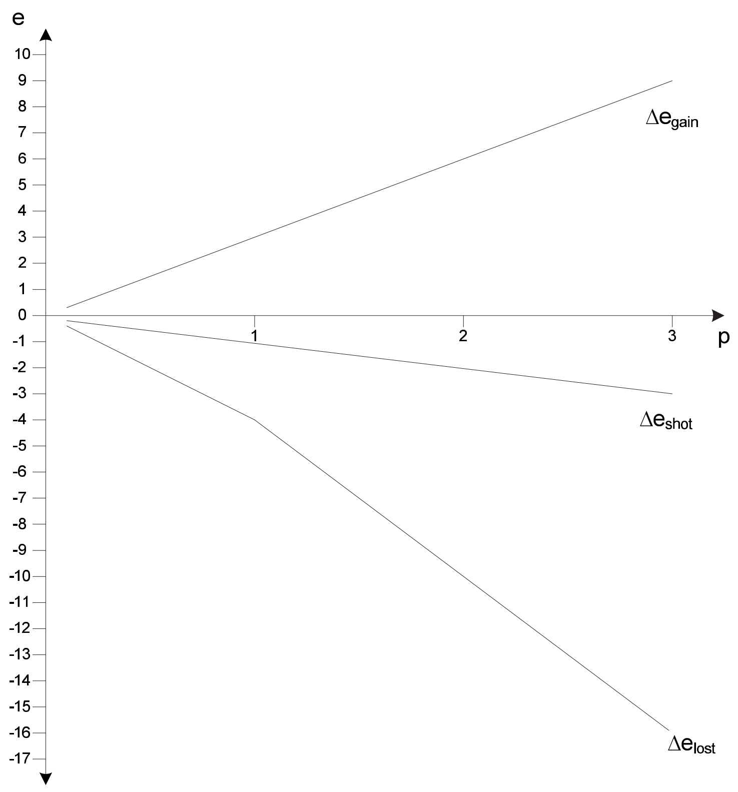


Figure 1.4: Shows Equation 1.2, 1.4, and 1.5

1.5 Robocode Robot

The robots in Robocode consist of three parts: a radar, a gun, and a vehicle. A representation of a robot can be seen in Figure 1.6. The three parts of a robot are able to move independently of each other. Since the radar is mounted on the gun, which again is mounted onto the vehicle, the three parts can influence each other. An example taken from the Robocode FAQ[6] explains this better:

> "If you are turning the gun left, and turning the radar right, the radar will only turn (45 - 20) = 25 degrees to the right. On the other hand, if they are both turning right, it will turn (45 + 20) = 65 degrees to the right. And of course, since the gun mounted on the body..."

The maximum rate of rotation of the gun is $\frac{20^\circ}{\text{tick}}$, which is added to the current rate of rotation of the robot. The maximum rate of rotation of the radar is

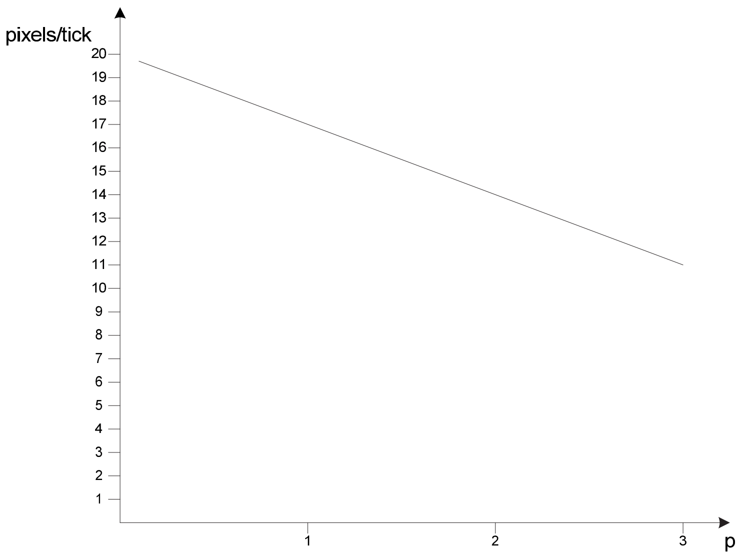


Figure 1.5: The graph shows the speed of a bullet s as a function of the power p given to the bullet, Equation 1.6

$\frac{45°}{\text{tick}}$, which again is added to the current rotation rate of the gun.

In Robocode it is faster for a robot to slow down than it is for it to speed up. The acceleration of a robot is $1\frac{\text{pixels}}{\text{tick}^2}$ and deceleration is $2\frac{\text{pixels}}{\text{tick}^2}$.

The velocity of a robot can be acquired by using the `getVelocity()` method on an instance of the `Robot` class. The velocity of a robot is a range from -8 to $8\ \frac{\text{pixels}}{\text{tick}}$.

The rate at which it is possible to turn a vehicle depends on the velocity of this vehicle. Equation 1.10 describes how hard the robot is able to turn at velocity v. The turning rate $t(v)$ is measured in $\frac{\text{degrees}}{\text{tick}}$.

$$t(v) = 10 - \frac{3}{4} * |v| \tag{1.10}$$

A Robocode robot can inherit from three different robot classes: `Robot`, `AdvancedRobot`, and `TeamRobot`, see Figure 1.7. The difference between the first two is that the `AdvancedRobot` class allows non-blocking calls, custom events, and the possibility to write to a file. With support for non-blocking calls, you can tell your robot to drive forward, turn the radar, and turn the turret–all at once. With the simpler `Robot` class you would have to wait for one event to end before starting another. The final class, `TeamRobot`, makes it possible for robots to communicate with each other and thereby make team-based decisions.

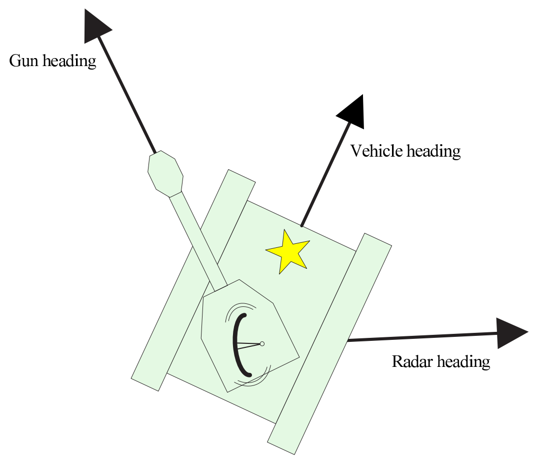


Figure 1.6: A Robocode robot

1.6 Teamplay

In Robocode a team is a collection of robots working together to defeat the enemy. Initially robots are not able to communicate with each other, and because of this it is virtually impossible to build a cooperating team of robots. Robots must extend the `TeamRobot` class in order to provide inter-robot communication, as this class contains methods for sending and receiving messages between robots. It is possible to send messages as either a broadcast or a direct message. A message sent as a broadcast message reaches all robots in a team, while direct messages reach only a single teammate. The `TeamRobot` class also provides robots with the possibility to check whether another robot is a teammate or not. Furthermore, a team in Robocode has a leader. The first robot added to the team is automatically made leader of the team. The leader gets a bonus in energy of 100 points, resulting in a total of 200 energy points. Robocode also provides the possibility of using droids, which are essentially normal robots with 20 more energy points and no radar. Droids will of course only work in a team-based game, where they rely on the robots with radar for information about what is happening on the battlefield. Droids have the advantage of 20 extra energy points, but they will be in serious trouble if the teammates with radars get killed.

This chapter outlined the rules and mechanics of the Robocode game. Based on these rules it is now necessary to study autonomous agent design. The Robocode framework allows for a various range of control structures, so a design must be decided for the different controls and a general behavior

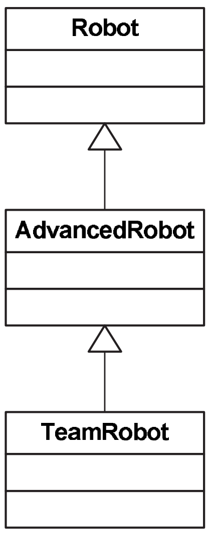


Figure 1.7: Robot classes in Robocode

pattern must be decided.

The primary goal for this research aims to construct a team of robots that implements various machine learning and models for reasoning techniques. This is looked into in the following chapter.

2 Agents

This chapter will examine some of the more popular agent architectures so that an informed decision can be made to which systems would be useful when implementing a robot for Robocode.

2.1 Reactive Agents

This section is mainly based on The Design of Intelligent Agents by Jörg P. Müller[9].

Reactive agents react upon inputs obtained from sensors. These agents are considered to be the fastest kind of agent to execute since they must only react based on tactics that are known beforehand, which only have to be compared with the stimuli. These decisions are made at run-time.

A reactive agent works by executing a rule matching the current input. This rule could be run-away-and-hide, attack, or search-for-targets. Since the decision is not based on elaborated reasoning or planning, the calculations are not computational complex and therefore faster to execute. The focus of reactive architectures is to make agents with robust behavior instead of optimal behavior, meaning that an agent using the reactive architecture will continue functioning even if you suddenly place it in a totally different environment. This is because the agent will react based on the input it receives from its sensors. It does not care about how the surrounding world looks, and therefore it will not have to construct and maintain a model of this world.

The most frequently used reactive architecture for implementing agents that operate alone is the subsumption architecture [12]. A robot implemented using the subsumption architecture is organized in a vertical layered structure,

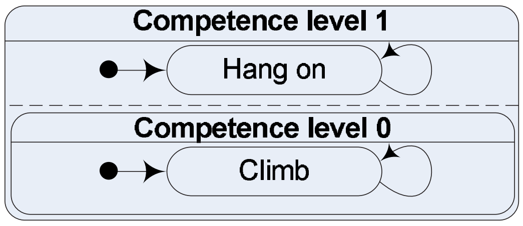


Figure 2.1: The reactive architecture is divided into layers.

where the topmost layers have precedence over lower layers and the upper level *subsumes* the output of lower layers, hence the name *subsumption architecture*. These agents are flexible, meaning that they are able to work in a changing environment and they have the capability to have multiple goals. The lower layers are used for implement basic behavior, such as drive around and avoid hitting the walls, whereas the higher levels are used for fulfilling the goals of the robot, such as trying to hit other robots and avoid getting hit.

In order to demonstrate this model consider the example concerning a mountain climber. This model has two layers, a “climb” layer and a “hang on” layer, which is shown in Figure 2.1. These layers react upon inputs from sensors which are obtained while climbing the mountain. Now, if the “climb” layer receives input, for example that the mountain climber sees a new rock where he can place his hands in order to keep climbing, he will keep climbing. If the “hang on” layer receives input, for example that the mountain climber looses grip with his feet, then the mountain climber tends to hold on to the rock he already has a grip on. If the two layers both receive heavy input, the agent will prioritize some actions over others. So if the mountain climber looses grip with his feet and sees a new rock at the same time, he will tend to hold on with his hands instead of keep climbing as the “hang on” layer is at a higher level than the “climb” layer (the “hang on” layer *subsumes* the output of the “climb” layer).

One of the advantages with the subsumption architecture is that it is fast because it just reacts upon the input it receives from its sensors. This is very useful when implementing systems that have a limited amount of calculation time, like for instance Robocode.

One of the major drawbacks with a pure reactive architecture is that there is no way to make the layers communicate with each other. For example consider again the example with the mountain climber. Maybe the optimal solution in case the mountain climber looses grip with his feet would be to hold on to another rock because it offers a better grip. But as the "hold on" layer subsumes the "climb" layer, it does not look at which solutions the layers below have suggested. This is clearly a disadvantage since the layers below might help making a decision, as the lower levels can strengthen the belief of the decision of the "hold on" layer.

Another drawback with a pure subsumption architecture is that there is no obvious way to implement communication between robots that are cooperating as a team. The problem is that there is no natural placement for a communication layer. Consider that the communication is placed as the top layer. It will subsume output from all other layers, and therefore it will never do anything "intelligent" other than communicating with other robots. If on the other hand the communication layer is placed on a lower level, higher levels will subsume the communication with other robots.

2.2 Deliberative Agents

A deliberative agent will at all times, using an appropriate symbolic representation, maintain an overview of the world in which it operates. Typical considerations when implementing this agent architecture is that all computation concerning the surrounding world must be done in time for it to be useful. This means that the implementer of such an agent should consider how this, sometimes very complicated, world should be symbolically represented, and how calculations on this representation should be done. Environments that are very dynamical, as in fast-changing, require calculations to be done quickly, as actions need to be performed before they expire. Static environments are more forgiving in this aspect, as you, in theory, have all the time in the world to perform calculations. This also means that there is a trade-off between calculation time and the amount of detail in the world model.

This section is mainly based on Deliberation by Derek Bridge [3]. Deliberative agents are said to "think ahead". This means that deliberate agents will consider a sequence of actions prior to execution and thereby utilize the gathered effect of multiple actions (consider, for instance, the way players think ahead in a game of chess).

But how do deliberative agents consider a whole sequence of actions in order to think ahead? A simulation is done where three things are required and iteratively reconsidered:

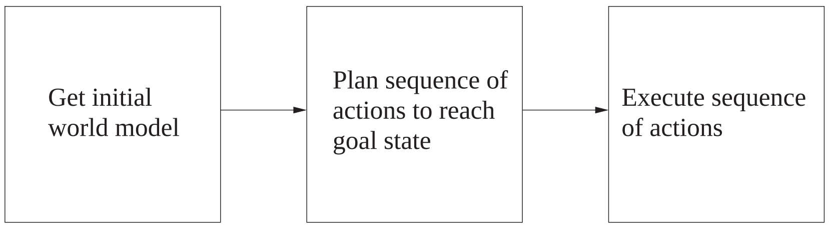

Get initial world model
Plan sequence of actions to reach goal state
Execute sequence of actions

Figure 2.2: Simulation with a static world model

1. A model of the initial world. The effects of the chosen actions will be simulated on this model.
2. A set of actions provided by the agent.
3. A *goal condition* that specifies how the world should look, in order for the agent to achieve its goal.

There are two ways of doing this simulation. If the world in which the agent operates is static, then we only need to consider a sequence of actions to take once. After inspecting the world, as it looks initially, the agent will plan a sequence of actions, that will achieve its goal condition. An example of this would be a mountain climber, trying to plan an optimal route for climbing a mountain. This type of simulation is represented in Figure 2.2.

If the agent is operating in a dynamic world, meaning a world that is constantly changing, then we need to go about things a little differently. Initially we do just as before, calculating a sequence of actions that will, if sequentially executed, achieve the agent's goal. What is different, is the fact that after this calculation, we only execute the first action. This is because the world is constantly changing, and we do not know if the actions that were applicable before are applicable any more. After executing the first action we go back and start over. This makes for an agent that will adapt its behavior according to the world as it changes over time. This simulation is represented in Figure 2.3.

However, it might not always be desirable for an agent to consider every single step to take for it to achieve some goal, because this consideration must be done at *every* turn when in a dynamical world model, and it might easily be too time consuming. In a setting such as Robocode this inefficiency is a crucial factor in your ability to stay alive. This approach is similar to gradient descent, which also takes small steps towards a minimum explained in Section 3.3.3. It might be useful if the agent only considered a smaller

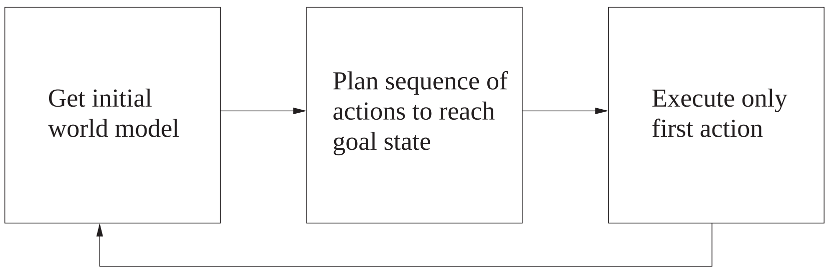

Get initial world model
Plan sequence of actions to reach goal state
Execute only first action

Figure 2.3: Simulation with a dynamic world model

sequence of steps to take. This way an agent would be able to operate more efficiently, while still maintaining the ability to choose sensible actions to perform.

The question of how far an agent should look ahead in order to make these choices is determined by the context in which it operates. The more dynamic the world, the more unpredictable it will be. A world that is very fast-moving will not require an agent to look very far ahead, since choices made several turns ago quickly loose their validity. A world that is relatively static will give an agent the opportunity to look further ahead. It can be argued that the further an agent looks ahead, the more accurate or correct its choices will be, but in a fast-moving world these choices might still be very hard to make correctly.

Architectures following the deliberative architecture has become known as *Belief, Desire, Intention* architectures. The last part of this section is mainly based on The Design of Intelligent Agents by Jörg P. Müller[9]. The Belief, Desire, Intentions (BDI) architecture is a field that has become very popular in the research of agent design. An agent designed using the BDI architecture works by utilizing a set of mental categories. These categories are *belief*, *desire*, and *intentions*.

Belief is a category describing expectations on how the world will look after a set of actions is performed. This way the agent can determine which actions to perform in order to achieve some effect.

Desire is the mechanism that ensures that the agent is able to associate a preference with different states of the world. This means that an agent can have a high desire to achieve some arbitrary state of world. It is important to note that the agent does not have to believe that its desires are achievable.

Intentions are necessary since agents are not able to pursue all desires (goals) at once. An agent must select a single desire (or set of desires) to pursue – these are the intentions of the agent.

More practical approaches have added the notions of *goals* and *plans*.

Goals is the subset of desires that the agent believes to be achievable. Goals were introduced to the BDI architecture because of the weak definition of *desires*.

Plans are collections of intentions. This means that it is possible to combine intentions into larger plans. An agent's intentions are defined by the plans it has currently committed to.

2.3 Reactive vs. Deliberative Agents

Deliberative agents are most often more advanced than their reactive counterparts. Whereas reactive agents function by simply looking through a set of predefined behaviors, deliberative agents work more informed by modelling the world around them, allowing them to predict consequences of actions determined by a set of behaviors. Looking at reactive agents, we see that they function by searching through a set of actions looking for an action that fits the agents current situation exactly. This in effect means that reactive agents are unable to make decisions that utilize the gathered effect of multiple actions.

An advantage of deliberative agents is the fact that a lot of the effort invested in the implementation of the agent is moved from design-time to run-time. Reactive agents require a large set of actions in order to ensure that every possible situation it might find itself in will cause the right action to be executed. These actions must be cleverly thought out and provided to the agent, either by designing them by hand or by evolution or learning. Deliberative agents do not require such an extensive set of actions to be supplied by the designer (or by evolution or learning). This is because when dealing with deliberative agents, it partly becomes the responsibility of the agent to choose the right action for any given situation.

In conclusion a pure implementation of the reactive architecture would not be optimal for this Robocode project, as we are implementing agents that will be part of a team. Teams are problematic to design using a purely reactive architecture. This, however, does not mean that the reactive architecture is completely omitted from this project.

2.4 Hybrid Agents

The approach of making the implementation using a purely deliberative or reactive agent has been discussed and argued that both architectures suffer from different shortcomings when making an intelligent robot[9]. Because of this a combination of the two architectures is proposed. This combination does not suffer from the drawbacks that an implementation based on a pure architecture does.

Whereas reactive agents cannot be implemented containing a goal-directed behavior, the deliberative agents are based on general reasoning and therefore much less reactive. The idea is to organize the functionality of the agent types into layers that are hierarchically organized. This has the advantage that the agent's reactivity is preserved while it still maintains a model of the world. Another advantage with the hybrid architecture is that it is possible to add team communication to a reactive model, which is not possible in a pure reactive model.

Therefore a hybrid architecture is chosen for the our Robocode robots. This way the reactivity is kept while still maintaining a model of the world, in order to let the robots share information about enemies and bullets.

2.5 Adaptive Agents

Adaptive agents are, as the name implies, able to adapt to their environment. They utilize learning algorithms that allow the agent to react to situations according to previous experiences. In other words, an adaptive agent will get smarter along the way. This will enable us to develop agents that are very effective in their environment.

There are several tasks in the agent that can be trained by a "network". One could consider movement as a task to be learned, as is aiming and shooting. According to Thomas J. Loredo and David F. Cherno, a Bayesian network could be constructed for reactive movement, this is well documented in the article "Bayesian Adaptive Exploration"[13]. The framework for such a net could be constructed by using the previous movement combined with the terrain information along with information about possible collisions with other agents. The same approach could be taken with shooting and aiming.

2.6 Team Models

When implementing agents that have to cooperate, several different approaches can be taken. Questions that need to be asked are: Is some sort of central (or distributed) coordination needed or is it possible for autonomous agents to decide by themselves what to do?

When working with robots that extend the `TeamRobot` class it is possible for the robots to send messages to each other. A message arriving at the receiving robot is exactly identical to when it was sent. This is in contrast to some real world examples, for example radio-transmitted messages which are affected when sent through the air.

The only sensor from which a robot in Robocode can get information about the environment, is the radar. But since the radar only has a limited range and field of view, it is impossible for a robot to know everything about the environment all at once. Therefore maintaining a world model with information gathered by all robots is an advantage.

Basically, two different methods can be used in Robocode when battling in teams. One robot could decide what strategy to use and how to execute it, and then order the other robots around. The other possibility is that each robot could decide by itself what to do. No matter which of the two approaches is chosen, the more information known about the environment, the better they will perform. On the other hand, each robot is only given a limited time slice in each round, and therefore the information exchange has to be limited. If a robot discovers something, it can choose to tell the others about it. But if the other robots will discover it themselves within a certain amount of time, the information need not be exchanged. Therefore deciding what information to put in the world model, and thereby telling the other teammates, is important.

If one robot is chosen to decide what the other robots should do, this robot will look at the world using the accumulated vision of its own and all the teammate's radars. It will figure out some plan to execute and then tell the other robots what to do. But if one robot is chosen to lead the others, there will be a larger timespan from information is collected until actions, based on that information, are carried out. This is because messages sent in round x do not arrive until round $x + 1$. Actions planned from these messages are then computed, and new messages containing these plans are sent out. They will arrive in round $x + 2$ where they will be carried out. That time can be halved if there is no centralized coordination. When each robot has received its messages in round $x+1$, they can compute and execute their plans in that same round. To illustrate the point, we use the knowledge from Section 1.5, that the maximum velocity for a robot is 8 $\frac{\text{pixels}}{\text{tick}}$. This means that a robot can move as much as 16 pixels in 2 ticks. Obviously, more accurate plans can be computed using information that is as new as possible, and centralized coordination provides more delay than distributed does.

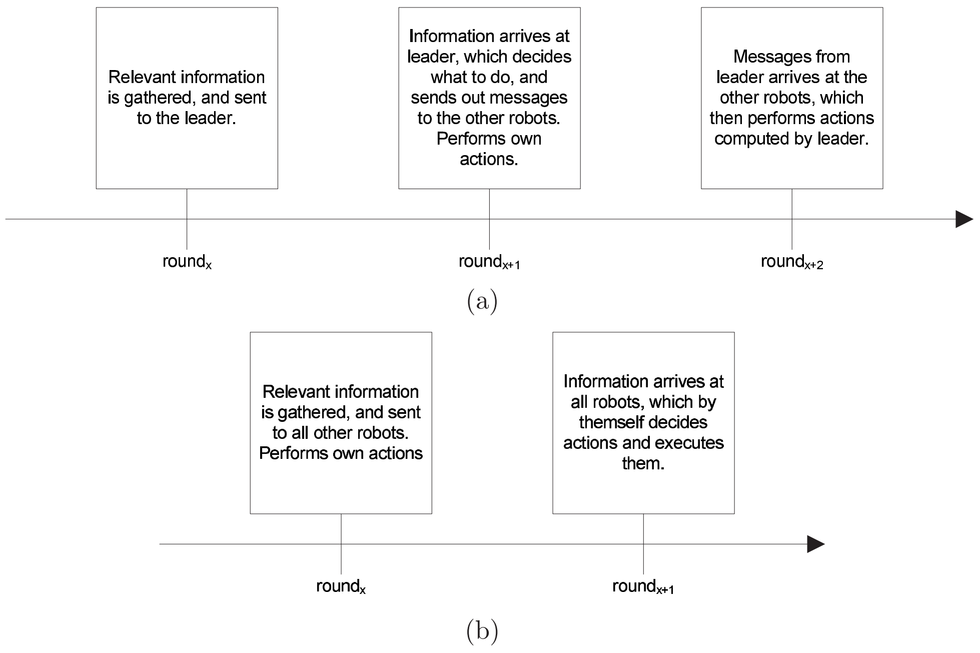

Relevant information is gathered, and sent to the leader.
Information arrives at leader, which decides what to do, and sends out messages to the other robots. Performs own actions.
Messages from leader arrives at the other robots, which then performs actions computed by leader.
round_x
round_{x+1}
round_{x+2}
(a)
Relevant information is gathered, and sent to all other robots. Performs own actions
Information arrives at all robots, which by themself decides actions and executes them.
round_x
round_{x+1}
(b)

Figure 2.4: Showing time span with or without leader.

If a model with centralized coordination is chosen, the first step is to decide at which level the leader should hand out orders. It could tell each robot what to do at the lowest level, e.g. "*go forward* 100*, turn x degrees right, fire with power p*". Or the orders could be more general, like "*defend a certain area, follow a robot, attack*". If a model based on general orders is chosen, the leader could assign different roles to the robots on the team. Roles could be e.g. attacker, information collector, or protector. If coordination is done by letting the leader tell the teammates exactly what to do (lowest level), the leader will be able to carry out a strategy exactly as planned, as the leader has total control over all the robots on the team. There is however one major disadvantage with a centralized control mechanism. If the leader is killed, there will be no one to give the orders, and thereby rendering the other robots useless, because they will not know what to do.

Without central coordination, roles could also be used. Robots could look at the other robots on the team and figure out what roles they have taken on, and then decide which role to play themselves. When using this approach, the team communication is increased because the robots must communicate collected information to all teammates since they all must maintain a model of the world on their own. This approach has the advantage that there is no "weak link," meaning that if one of the robots die, the others will just carry on, functioning as they did before. Each robot has their own world model and is able to decide what to do based on this model. Another advantage when having no centralized coordination is that a robot is able to react promptly upon unexpected obstacles, because it does not have to communicate with the leader to get instructions on what to do.

There are several ways to choose which target to attack. One way is simply to choose the nearest target and attack it. This has the disadvantage that all robots on the team could end up choosing unique enemies to attack and as a consequence get killed because the robots "forgot" to utilize team behavior. Perhaps the battle could have been won if some of the robots had chosen the same target and attacked the enemy as a team would. So the question is how the robots should decide which enemy to attack. One way is to base the decision upon previous experiences from situations that look like the one that the team is currently in. When a robot chooses a target it tells the other robots which target it has chosen. With this information the other robots will be able to decide whether or not they should attack the same robot also. An example: If a fellow robot chooses an enemy with low energy that is a certain amount of pixels away from you, you must "look up" if a similar situation has been experienced before. If so, how was the outcome of this previously experienced situation? If the outcome was good you should do as you did then.

Another approach is making the deciding robot ask the team if anyone can assist in attacking an enemy. If someone answers, the robot will choose which robots it wants to get help from.

Having considered several agent architectures, the choice for this project fell on a hybrid architecture. Now that we are familiar with the most general agent designs, the time has come to study which machine learning techniques will be useful when implementing a Robocode robot. It is important to realize what properties are important in the different kinds of machine learning techniques and how to use them efficiently in our agent implementation.

3 Models for Reasoning and Machine Learning

This chapter will examine some different models for reasoning and machine learning technologies. Each of these technologies could be utilized when implementing the tasks a Robocode robot must perform. This chapter will try to outline the strengths and weaknesses of each of these technologies as well as look at which type of tasks each of them is suitable for.

3.1 Representation of Uncertainties

This section is based on lecture notes from lecture 1 of the course *Decision Support Systems and Machine Learning* [10] and Chapter 1 from *Bayesian Network and Decision Graphs* [5].

Nothing is certain in decision support systems, so the first thing we need to do is formalize a probability:

Definition 1 *A probability $x \in [0; 1]$, where x represents how likely it is that some event will occur. A probability of 0 means that the event will never occur, whereas a probability of 1 means that the event will always occur.*

Furthermore we need to have formalized definitions of several other notions.

Definition 2 *If events a and b can never occur at the same time, we call them mutually exclusive.*

Definition 3 *We write the probability of an event a, as $P(a)$.*

Definition 4 *We write the probability of either event a* or *b occurring, as $P(a \vee b)$.*

Definition 5 *We write the probability of both event a* and *b occurring, as $P(a \wedge b)$ or $P(a, b)$.*

If event a and b are mutually exclusive $P(a \vee b) = P(a) + P(b)$.

Definition 6 *The probability of a given b is written as $P(a|b)$.*

The definition above expresses our belief in a after b has been observed.

Definition 7 *A is a variable with states $\{a_1, \ldots, a_n\}$, then $P(A) = (x_1, \ldots, x_n)$ where $x_i = P(a_i)$. a_j must always be mutually exclusive of a_k for $a_j, a_k \in \{a_1, \ldots, a_n\}$ and $a_j \neq a_k$, furthermore $\sum_{i=1}^{n} x_i = 1$*

Definition 8 *Given two variables A and B where A has i states and B has j states, $P(A|B)$ is a $i \times j$ matrix.*

	b_1	b_2	b_3
a_1	0.2	0.3	0.5
a_2	0.8	0.7	0.5

Figure 3.1: Example of $P(A|B)$

Figure 3.1 shows $P(A|B)$ where A has the states a_1, a_2 and B has b_1, b_2, b_3. Notice that the columns sum to 1.

Definition 9 *We write the joint probability of two variables A and B as $P(A, B)$. If A has i states and B has j states, $P(A, B)$ is a $i \times j$ matrix.*

It is possible to construct the joint probability $P(A, B)$ from the conditional probability $P(A|B)$ and $P(B)$ using Equation 3.1.

$$P(A|B)P(B) = P(A, B) \tag{3.1}$$

	b_1	b_2	b_3
a_1	0.06	0.06	0.25
a_2	0.24	0.14	0.25

Figure 3.2: Example of $P(A, B)$

Equation 3.1 is known as the *fundamental rule*. Figure 3.2 is the result of applying the fundamental rule to Figure 3.1 if $P(B) = (0.3, 0.2, 0.5)$.

A mathematical rule that explains how beliefs should be changed based on new evidence is shown in Equation 3.2.

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)} \tag{3.2}$$

Equation 3.2 is known as *Bayes rule*. It is easy to construct a proof for Bayes rule: $P(B|A)P(A) = P(A, B) = P(A|B)P(B)$.

From a probability table $P(A, B)$ the probability of $P(A)$ can be calculated by using Equation 3.3

$$P(A) = \sum_{B} P(A, B) \tag{3.3}$$

Equation 3.3 is called *marginalization*. We say that we marginalize B out of $P(A, B)$. If A has i states and B has j states, $P(A) = \sum_B P(A, B)$ should be read as $P(a_n) = \sum_{k=1}^{j} P(a_n, b_k)$ for each $n \in [1; i]$.

$$\boxed{P(A) = (0.06 + 0.06 + 0.25, 0.24 + 0.14 + 0.25) = (0.37, 0.63)}$$

Figure 3.3: B marginalized out of $P(A, B)$ from Figure 3.2

Definition 10 *We say that we have evidence on a variable if we know in which state the variable is.*

If we have evidence on B in Figure 3.1 we could write $P(A|B = b_1) = (0.2, 0.8)$.

3.2 Bayesian Networks

The following section is based on Chapter 1-4 in *Bayesian Network and Decision Graphs* [5].

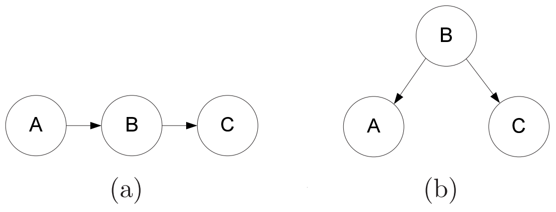


Figure 3.4: Two boring Bayesian networks

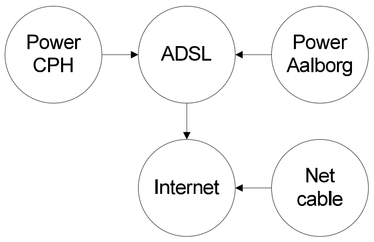


Figure 3.5: Simple Bayesian network for an internet connection

3.2.1 Causal Network

A causal network is an acyclic oriented graph where vertices are variables and edges are causal relations. Causal relations could be considered as causes–hence the name. Figure 3.4(a) should be read as *A is the cause of B and B is the cause of C*, and Figure 3.4(b), *B is the cause of A and C*.

In the example in Figure 3.5 we see that if the internet connection is down, this can be either the network cable being unplugged or the ADSL being down–possibly both. The ADSL could be down because of either a power failure in Copenhagen or Aalborg. Of course there can be a large number of other reasons for this misshapen, but in the simple example in Figure 3.5 these reasons are not modelled.

In a Bayesian network each vertex is a variable and each variable's probability is specified as the condition given its parents. In Figure 3.4(a) we need to specify $P(A)$, $P(B|A)$, and $P(C|B)$, and in Figure 3.4(b) $P(B)$, $P(A|B)$, and $P(C|B)$. If all variables in Figure 3.5 are binary, they will each have two states, $\{\text{OK}, \text{FAILED}\}$. The following tables needs to be specified:

- $P(\text{Power CPH})$ (2 parameters)
- $P(\text{Power Aalborg})$ (2 parameters)
- $P(\text{ADSL up}|\text{Power CPH}, \text{Power Aalborg})$ (8 parameters)

$P(\text{Power CPH}) = (0.999, 0.001)$

$P(\text{Power Aalborg}) = (0.997, 0.003)$

$P(\text{ADSL}|\text{Power CPH}, \text{Power Aalborg}) =$

Aalborg \ CPH	OK	FAILED
OK	$(0.99, 0.01)$	$(0, 1)$
FAILED	$(0, 1)$	$(0, 1)$

$P(\text{Netcable plugged}) = (0.98, 0.02)$

$P(\text{Internet}|\text{ADSL}, \text{Netcable}) =$

Netcable \ ADSL	OK	FAILED
OK	$(0.99, 0.01)$	$(0, 1)$
FAILED	$(0, 1)$	$(0, 1)$

Figure 3.6: Example of probabilities for Figure 3.5

- $P(\text{Netcable plugged})$ (2 parameters)
- $P(\text{Internet}|\text{ADSL up}, \text{Netcable plugged})$. (8 parameters)

which is a total of 22 parameters. It could be argued that it is only necessary to specify $P(\text{Internet}|\text{Power CPH}, \text{Power Aalborg}, \text{ADSL up}, \text{Netcable plugged})$, which is correct, but it would result in a total of 32 parameters. A Bayesian network can be seen as a compact representation of the full joint probability, because when applying the Fundamental rule to the network one can always get the full probability table. The reason for the smaller number of parameters is that there is an assumption of independence. It could be that there, for example, was no connection between Power Copenhagen and Power Aalborg. This assumption is probably wrong, so the assumption is that the connection is very small.

3.2.2 Fractional Updating

Bayesian networks are based on static statistic information, this statistical data could have been implemented into the network by an expert, who knows the exact probability values of the data. But if there is no expert, a Bayesian network lack the ability to dynamically learn these data. One method for achieving this ability is known as fractional updating. For example the probabilities in Figure 3.7 could be learned from some data set. Figure 3.8 is an example of such a data set for Figure 3.7.

The idea is that for each element in the data set, we update the probabilities of $P(A)$, $P(C)$, and $P(B|A, C)$. If we have a data element $d = (a_1, b_1, c_2)$ we would update the probabilities of $P(A) = (\frac{n_{a_1}+1}{s_a+1}, \frac{n_{a_2}}{s_a+1})$ and $P(C) =$

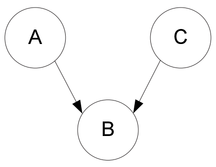


Figure 3.7: A Bayesian network, $A = \{a_1, a_2\}, B = \{b_1, b_2\}$ and $C = \{c_1, c_2\}$

A	B	C
a_1	b_1	c_2
a_2	b_1	c_1
a_2	b_1	c_2
a_1	b_1	c_1
a_1	b_1	c_2
a_2	b_2	c_2
a_1	b_1	c_1

A	B	C
a_2	b_1	c_2
a_1	b_1	c_2
a_2	b_1	c_1
a_1	b_2	c_2
a_2	b_1	c_2
a_2	b_1	c_2
a_2	b_1	c_1

A	B	C
a_2	b_1	c_2
a_2	b_1	c_1
a_2	b_1	c_2
a_1	b_1	c_2
a_2	b_1	c_2
a_1	b_2	c_1
a_1	b_1	c_1

Figure 3.8: Example of data set for Figure 3.7

$(\frac{n_{c_1}}{s_c+1}, \frac{n_{c_2}+1}{s_c+1})$. Where s_a and s_c are the number of updates that have been done to $P(A)$ and $P(C)$ respectively, and n_{a_1} is the previous number of seen (a_1, b_x, c_y) and n_{a_2} is the number of seen (a_2, b_x, c_y), the same for $n_{c_{\{1,2\}}}$, s_a must always be equal to $n_{a_1} + n_{a_2}$ and s_c always $n_{c_1} + n_{c_2}$.

$P(B|A, C)$ still needs to be updated. $P(B|a_1, c_2) = (\frac{n_1+1}{s+1}, \frac{n_2}{s+1})$, where s is the number of updates that has been done to $P(B|a_1, c_2)$ and n_1 is the number of previously seen data elements with the form (a_1, b_1, c_2) and n_2 the number of data elements on the form (a_1, b_2, c_2). Note that only $\frac{2}{8}$ of the parameters of $P(B|A, C)$ is updated.

3.2.3 Fading Fractional Updating

The problem with the model just described, is that if the system runs for a long time and there is a sudden change in the environment, then it would literally take forever to get the previously learned data filtered out of the system. In other words, it would take a very long time to get even a fair approximation of the new environment. One way to overcome this problem is to introduce a fading factor $q \in [0; 1]$. In the example from the previous section we instead update $P(A) = (\frac{qn_{a_1}+1}{qs_a+1}, \frac{qn_{a_2}}{qs_a+1})$. The same goes for $P(C)$ and $P(B|A, C)$. In this scheme all influence from the past will fade away exponentially.

$$s^* = \frac{1}{1-q} \tag{3.4}$$

s^* from Equation 3.4 is known as the *effective sample size*, and is the value that s will converge to.

Bayesian networks are well suited for problem domains where causal relations can easily be established between discrete variables. Furthermore it is also possible to use the Fractional updating to implement some sort of learning mechanism.

3.3 Neural Networks

This section will deal with the idea behind using neural networks as decision support systems, how to build such a net with the use of perceptrons, and how to update multi-layered neural networks using the backpropagation algorithm. The section is based on *Tom M. Mitchell's Machine Learning*[8].

3.3.1 Motivation Behind Artificial Neural Networks

The idea behind the neural network comes from studying the human brain. According to the research field of neuron biology, the human brain consists of approximately 10^{11} neurons, each connected to an average of 10^4 other neurons. These dense neural networks are capable of very complex and fast calculations/decisions. If we consider the fastest known neuron switching, which is approximately 10^{-3} seconds, and then consider the switching speed of a computer, which is around 10^{-10} seconds, our conclusion could be that with a computer we can construct neural networks capable of making decisions extremely fast.

We all know that computers still have a long way to go before we can even begin to compare their "intelligence" to that of the human brain. Even though computers can operate at speeds the human mind can only dream of, the human brain is said to operate in a highly parallel manner that today's computers are nowhere near capable of. But still, the motivation behind the neural network theory remains clear: To capture the process of learning and decision making found in the most complex machine known to man–the human brain–and use this to create a simplified model that resembles how this process works.

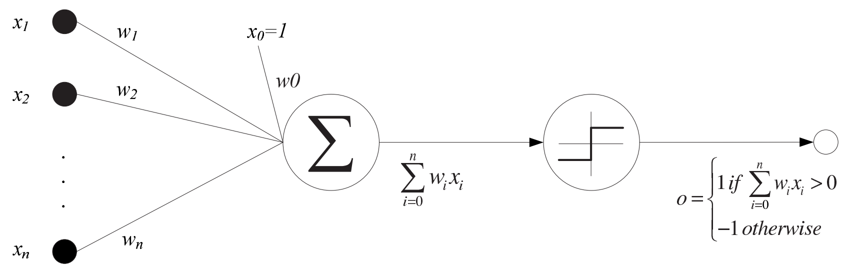

x_1
x_2
x_n
w_1
w_2
w_n
$x_0=1$
$w0$
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

Figure 3.9: A basic perceptron

3.3.2 Perceptron

One common type of building block for neural networks is the perceptron, see Figure 3.9. The perceptron is basically a function that takes in a vector of real-valued inputs, then maps this vector to an output of either 1 or −1. It does this by calculating a linear combination of these inputs, in order to evaluate whether the result surpasses some threshold. The output is 1 if the result is greater than the threshold and −1 if its lower, see Figure 3.9.

To give a precise definition of the output function o, we look at the input entries of the vector, $x_1, x_2, \ldots, x_n$, where n corresponds to the number of observations or perceptions in our network.

o is calculated like this

$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n > 0 \\ -1 & \text{otherwise} \end{cases} \tag{3.5}$$

where w_i is a real-valued constant weight. This weight determines the contribution of the specific perception x_i to the final sum of weighted perceptions. Note that $-w_0$ is the threshold the sum of the weighted vector entries $w_1x_1 + w_2x_2 + \cdots + w_nx_n$ must overcome for the function o to produce a 1. If we modify this by introducing $x_0 = 1$ we can write the inequality from Equation 3.5 as

$$\sum_{i=0}^{n} w_i x_i > 0 \tag{3.6}$$

or

$$\vec{w} \cdot \vec{x} > 0 \tag{3.7}$$

We can use this to simplify Equation 3.5 by denoting a new function *sgn*

$$sgn(y) = \begin{cases} 1 & \text{if } y > 0 \\ -1 & \text{otherwise} \end{cases} \tag{3.8}$$

3.3.3 Training Perceptrons

A perceptron produces an output based on the input vector and the vector of weights. The goal for the perceptron is to produce ±1 as output, but for this to take place we need to have the weights that will produce the desired output. Since the weight vector is not known in advance, the perceptron will have to "learn" the correct values. One approach is to initialize the weights to random values, and then adjust the values by updating the weights according to a set of training examples. This means updating the weights whenever an example is encountered that does not produce the correct output. To update the individual weights associated with input x_i, we use the perceptron training rule:

$$w_i \leftarrow w_i + \Delta w_i \tag{3.9}$$

where

$$\Delta w_i = \eta(t - o)x_i \tag{3.10}$$

in which t is the target output, o is the output generated by the perceptron, and η is the learning rate.

The learning rate η is used for controlling how much is learned in each step, meaning how much the weights are updated in each iteration. Note that if the perceptron classifies the input correctly, Δw_i evaluates to zero because $(t-o)$ will yield zero. The perceptron training rule have been proven to converge to the correct function given enough training examples and iterations, assuming the training examples are linearly separable[8].

Unfortunately the perceptron training rule runs into a problem when trying to classify training examples that are not linearly separable, see Figure 3.10 and Figure 3.11.

To overcome this obstacle we introduce the delta rule that uses gradient descent to approximate the target concept. Gradient descent makes it possible to guarantee that the delta rule converges toward some best-fit approximation.

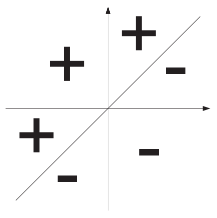


Figure 3.10: A set of training examples classified correctly by the perceptron training rule (+ denotes positive training examples, and - denotes negative training examples).

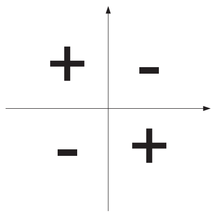


Figure 3.11: A set of training examples that are not linearly separable, which means it cannot be classified by a straight line.

3.3.4 Gradient descent

To find a "global minimum" one needs a direction to reduce the error margin in an n-dimensional plane. This direction is decided by a gradient vector. The gradient vector points towards the direction where the plane rises up the most, given a location in the n-dimensional plane. The gradient vector can be found by calculating the derivative of the error margin E with regards to the vector of weight components $\vec{w}$.

$$E(\vec{w}) = [\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, .., \frac{\partial E}{\partial w_k}] \tag{3.11}$$

The result of Equation 3.11 is a vector containing information about what direction the error grows the fastest. The thing we are interested in is the direction in which the error margin becomes smaller, this method of obtaining the minimum error is known as gradient descent. The gradient descent is the negated gradient vector $\Delta E(\vec{w})$. Using this rule for training each component in the weight vector, assign the new value to the ith weight by using Equation 3.12.

$$w_i \leftarrow w_i + \Delta w_i \tag{3.12}$$

where

$$\Delta w_i = -n \cdot \frac{\partial E}{\partial w_i} \tag{3.13}$$

Equation 3.13 changes w_i, minimizing the error and moves towards a global or local minimum. The positive number n is the learning rate, which decides the size of the leap the update takes towards a global minimum.

There are two different ways to implement gradient descent: stochastic or true gradient descent. The major differences are:

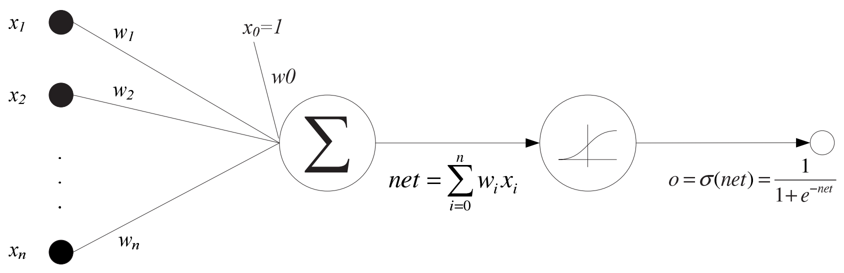
- True Gradient Descent (batch learning): This technique sums up the error, which is the gradient vector, for the whole training set and updates the weight vectors. Using this method is somewhat slower than Stochastic Gradient Descent.
- Stochastic Gradient Descent (on-line learning): This technique updates the weight vector after every input to the net. This results in the gradient vector being updated every round and will possibly encounter a number of small variations every round which might have been removed when using true gradient descent. This property makes a Stochastic learning technique more capable of handling irregularities.

The derivation of gradient descent can be seen in Section 4.4.3.2 in The Design of Intelligent Agents[8] and is not something we will comment on here. However, the use of gradient descent introduces another problem to be considered. The perceptron, see Figure 3.9, uses a discontinuous threshold which makes it unsuitable for the gradient descent, since this discontinuity means that it is not differentiable and gradient descent uses differentiation to calculate Δw_i as seen in Equation 3.13. A way around this problem is to replace the discontinuous threshold with a new one. Figure 3.12 shows the sigmoid unit, which is a common modification to the perceptron, that uses a continuous function to compute the output. The reason for choosing the sigmoid function is that its derivative is easily calculated see Equation 3.14.

$$\frac{\partial E}{\partial w_i} = o(1-o)(-x_i) \tag{3.14}$$

o is the output function seen in Figure 3.12, note that the derivative is used in the error term δ_k in Equation 3.16.

Now we have looked at how to train single perceptrons, but a single perceptron can only represent linear decision surfaces. Multi-layered neural



x_1
x_2
x_n
w_1
w_2
w_n
$x_0=1$
$w0$
$\sum$
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1+e^{-net}}$

Figure 3.12: A figure of the sigmoid unit

networks constructed out of many perceptrons or sigmoid units are capable of expressing highly non-linear decisions[8]. We will now introduce a way to train such multi-layered networks.

3.3.5 The Backpropagation Algorithm

The Backpropagation Net was first introduced by G.E. Hinton, E. Rumelhart, and R.J. Williams in 1986 [4].

We intend to use a feedforward neural network as our targeting mechanism for the robot. A feedforward network is a network that is fed an input at the top level and then propagates it forward through the network. We have chosen to look at feedforward networks as they are widely utilized for pattern recognition, dynamic modelling, data mining, function approximation, and forecasting [14][7]. In order to be able to train the feedforward network we examine the backpropagation algorithm.

The backpropagation algorithm is the most common way to learn weights for a multi-layered network, and also a very powerful one [4]. The backpropagation algorithm uses gradient descent from Section 3.3.3 to minimize the squared error. Because we are looking at a multi-layered network with multiple output units, we redefine the error E to sum over all the outputs, in contrary to the single output from before:

$$E(\vec{w}) = \frac{1}{2} \sum_{d \in D} \sum_{k \in \text{outputs}} (t_{kd} - o_{kd})^2 \tag{3.15}$$

where outputs is the set of outputs in the network, t_{kd} is the target value associated with the kth output unit and training example d, and o_{kd} is the output value associated with the kth output unit and training example d. The backpropagation algorithm is faced with the task of searching every possible weight value for each unit in the network.

One of the large differences between the single-layered network and the multi-layered network, is that in contrast to having a single parabolic plane with one global minimum, the multi-layered network consists of many dimensional planes, which can each create a local minimum. This means that in the case of a multi-layered network, the gradient descent training rule described in Section 3.3.3 is not guaranteed to converge to the global minimum, but might fall into some local minimum. In Section 3.3.6 we discuss this a bit further and we present a modification to the backpropagation algorithm called momentum.

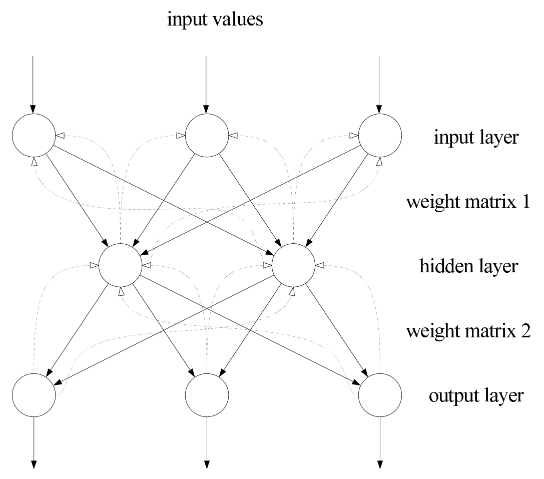


Figure 3.13: A visualization of the backpropagation algorithm. Note how every unit in the input layer affects the hidden layer, which in turn affects the output layer. The dashed lines shows how the weight updating propagates up through the network.

Observe that the backpropagation algorithm, see Figure 3.3.5, begins by constructing a feedforward network with a given number of hidden units and output units, and then it initializes all the weights to small random values. Now that it has a fixed network structure to work with, the main loop of the algorithm runs for a certain amount of time. This time can be any number set by the one running the algorithm, or it could be based on whether the algorithm converges to some value.

Inside the main loop we have the inner loop that iterates over all the training examples one by one, updating the weights according to the target vector $\vec{t}$ by calculating the output given the input vector $\vec{x}$. The inner loop has three parts to calculate before a fourth and final step resolves the new weights and updates each weight accordingly. The first calculation step is to find the values of the output units, then we backpropagate the error through backpropagationAlgorithm($training\_examples, \eta, n_{in}, n_{out}, n_{hidden}$)

traning_examples is the set of training examples, where each example is a pair of vectors $(\vec{x}, \vec{t})$, where $\vec{x}$ is the input values and $\vec{t}$ is the target values for the given example. η is the learning rate, meaning how fast changes to the weights will propagate through the network. The learning rate is usually set to some small number, e.g. 0.01, because this prevents an erroneous training example to influence the weights too much. n_{in}, n_{out}, and n_{hidden} are the number of input units, the number of output units, and the number of hidden units, respectively.

- Create a feedforward network with n_{in} inputs, n_{out} output and n_{hidden} hidden units.
- Randomly choose the initial small-valued weights.
- While the termination condition is not yet met, for each training pair $< \vec{x}, \vec{t} >$ in *training_examples* do

  1. Apply the inputs $\vec{x}$ to the network and calculate the output for every neuron from the input layer, through the hidden layer(s), to the output layer
  2. Propagate the errors backward through the network by first calculating the error term δ_k for each output unit

     $$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \tag{3.16}$$

     where o_k is the output of the kth output unit.
  3. For each hidden unit in the network calculate the error term δ_h

     $$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k$$

     where o_h is the output of the hth hidden unit, and w_{kh} corresponds to the weight from unit$_h$ to unit$_k$
  4. Now update each weight in the network

     $$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

     where w_{ji} is the weight from unit$_i$ to unit$_j$, and

     $$\Delta w_{ji} = \eta \delta_j x_{ji} \tag{3.17}$$

Figure 3.14: The backpropagation algorithm. The figure is taken from *Machine Learning*[8] page 98

the network by starting with the lowest level of the network, calculating the error corresponding to the output units. By propagating up to the next level, and calculating the error corresponding to the hidden units, there is enough information to finally calculate the new weights. Figure 3.13 shows an overview of backpropagation algorithm.

3.3.6 Convergence and Momentum

As mentioned in the previous section, backpropagation in multi-layered networks have the unfortunate complication that it is no longer guaranteed to converge to the global minimum, see Figure 3.15.

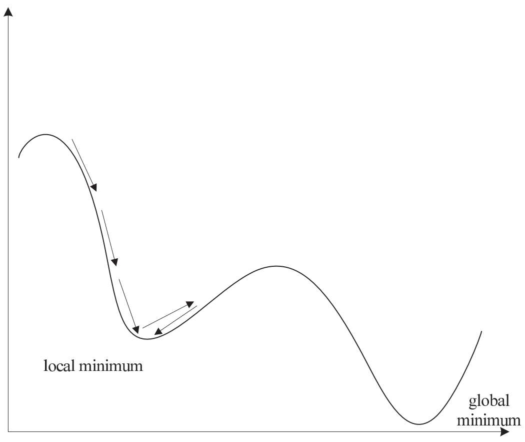


Figure 3.15: The backpropagation ends up in a local minimum

One way to try and overcome this in the backpropagation algorithm is to add momentum to the weight update rule in Equation 3.17. The weight update rule is changed to depend partially on the previous iteration, so the nth iteration is calculated by:

$$\Delta w_{ji}(n) = \eta d_j x_{ji} + \alpha \Delta w_{ji}(n-1) \tag{3.18}$$

where $\Delta w_{ji}(n)$ is the weight update in the nth iteration, and α denotes the momentum which should be a value in the interval $[0; 1[$.

The effect of the added momentum is to accelerate down steep slopes as seen in Figure 3.16. This has a tendency to keep the gradient descent going through local minima and ending up in the global minimum.

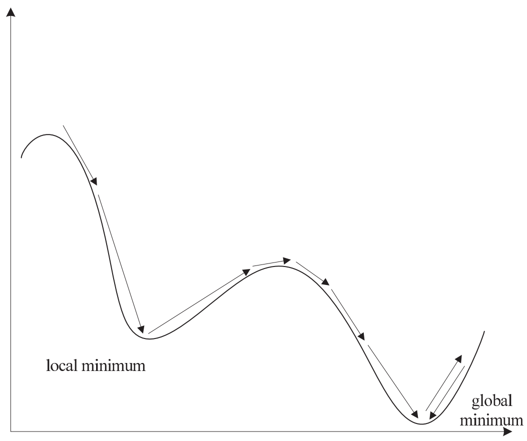


Figure 3.16: The modified backpropagation ends in a global minimum

3.4 Genetic Algorithms

This section is based on the ninth chapter on Genetic algorithms and Genetic programming from *Machine Learning*[8] and *Introduction to Genetic Algorithms*[11].

Evolutionary computing was introduced in the 1960s by Rechenberg in his work *Evolution Strategies*. Genetic algorithms were first described by John Holland and his students in *Adaption in Natural and Artificial Systems*. This research went public in 1975. In 1992 ideas from genetic algorithms were transformed into genetic programming by John Koza.[11]

First of all, evolution is known from biology as natures way of making individuals adapt to an environment. In terms of Robocode, the environment will change every time a robot is faced with a new opponent. This means that a robot needs to be able to adapt to its environment, and genetic algorithms allow robots to do this. A solution found by a genetic algorithm is not necessarily the best possible solution, but it will always be better than the ones found previously. The found solution has a fitness value that is better than the assigned *fitness threshold*. Fitness and fitness threshold will be explained later in Section 3.4.1.

Genetic algorithms offer a way to search for a solution in an environment where it is not obvious how different factors effect the solution. As a result of how genetic algorithms are implemented they are easy to parallelize, which again means that increasing CPU power would make the algorithms more efficient. Genetic algorithms have been applied to solving problems like:

- Data analysis and designing neural networks

- Robot trajectory
- Strategy planning
- Evolving LISP programs (genetic programming)
- TSP and sequence scheduling
- Functions for creating images

In other words, genetic algorithms provide a general strategy for finding optimized solutions to a problem.

3.4.1 The General Algorithm

Finding the solution to a problem is often a question of finding the minimum or maximum value of a function. The co-domain of the function is called the search space or pool of hypotheses. The search space of genetic algorithms is a pool of all possible sets of individuals. A population is a subset of the search space. These individuals within the population are string representations of the domain. The strings could contain any possible combination of characters that could describe this domain.

The best example of such a string must be DNA. DNA is a string of the letters G, T, A, and C. These four letters together describe every living organism. To keep it simple, the strings discussed throughout this section will be binary strings.

Like in the real world, evolution is performed by a genetic algorithm based on the principle that is *survival of the fittest.* Survival of the fittest is ensured by the fitness function described in Section 3.4.2. The evolution is driven by the genetic operators *crossover* and *mutation* described in Section 3.4.3. The crossover operators are used for recombining the individuals inside a population. Small changes are added to the individuals with the mutation operator.

There exists different implementations of genetic algorithms, but they all have more or less the same structure. This basic structure is described below. A genetic algorithm usually takes the following input:

Fitness is a function. This function is used for ranking the individuals in a population.

Fitness_threshold The algorithm terminates when an individual in a given population gets a fitness value that is greater than the *Fitness_threshold*. A possible problem could be that the fitness threshold is never reached, thereby creating an infinite loop. If the fitness values does not seem to change much in a number of generations one could argue that we have reached a maximum. By keeping track of this change, such a loop could be avoided.

p is the size of the population that the algorithm is going to maintain during its lifetime. The size of a population equals the number of individuals contained in it.

r is the fraction of individuals that are modified using crossover in every iteration of the main loop. How the size of r influences the behavior of the algorithm is described in 3.4.3.

m is a rate that tells how often mutations should be added to the individuals that go into the next population. How the size of m influences the behavior of the algorithm is described in Section 3.4.3.

Now that we know what parameters the algorithm depends on, we can start describing the algorithm itself. Before the algorithm enters its main loop for the first time, a random population of individuals is generated. The fitness value for each of these individual is calculated. After these steps the algorithm enters the main loop. The main loop runs until one individual gets a *Fitness* that is greater then the *Fitness_threshold* variable. The main loop is the while statement in Figure 3.17. The following operations are repeated at every run of the main loop.

1. A fitness value of every individual in the population is calculated.
2. Once the fitness values has been calculated, individuals are selected according to these values with the selection function. Equation 3.19 is such a selection function. This is described in Section 3.4.2.
3. Once the individuals for the next population have been selected, crossover operations are applied to them according to r.
4. Finally the mutation is applied to the new individuals with respect to m.

The following expression is used for calculating the probability that an individual is going to be a part of the next population. The probability is

```
GA(Fitness,Fitness_threshold,$p$,$r$,$m$)

  • Initialize population : $P \leftarrow$ Generate $p$ hypotheses at random.

  • For each $h$ in $P$ , compute Fitness(h).

  • While(max $Fitness(h) <$ Fitness_threshold)

      1. Select: Probabilistically select $(1 - r)p$ members of $P$ to add to
         $P_s$. The probability $Pr(h_i)$ of selecting hypothesis $h_i$ from $P$ is
         given by Equation 3.19.
      2. Crossover: Probabilistically select $\frac{r \cdot p}{2}$ pairs of hypotheses from $P$,
         according to Equation 3.19. For each pair $(h_1, h_2)$, produce two
         offspring by applying the Crossover operator. Add all offspring to
         $P_s$.
      3. Mutate: Choose m percent of the members of $P_s$ with uniform
         probability. For each, invert on randomly selected bit in its rep-
         resentation.
      4. Update $P \leftarrow P_s$
      5. For each $h$ in $P$ , compute Fitness(h).

  • Return the hypothesis with the highest fitness from $P$.
```

Figure 3.17: The general genetic algorithm. The figure is taken from *Machine Learning*[8] page 251

calculated as the fitness of an individual divided by the sum of the fitness values of all individuals.

$$Pr(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^{p} Fitness(h_j)} \tag{3.19}$$

The function above is a generalized expression that covers the most frequently used selection functions. A more detailed description of various selection functions and their drawbacks and advantages can be found in Section 3.4.4. The generalized function states that a high fitness value means that the individual has a high probability of getting into the next population in some form. Now the general algorithm has been described, but the functions it utilizes still remain. The first function to be described is the fitness function.

3.4.2 The Fitness Function

The fitness function is used for selecting the individuals that are going to become parents of the next population. The fitness function calculates a value for every individual in a population. The individuals with the highest values are typically the ones that become parents of the next population. In other words, the fitness function is a tool for ranking the individuals in a population.

Exactly how the fitness function calculates the fitness of an individual depends on the domain. If the domain is a game of chess, the value could be the number of moves needed to win the game. If the domain is a game of Robocode, the value could be the score gained during the game or the number of battles won.

3.4.3 Genetic Operators

The copying of bits is done by idealized versions of the operators that are used in genetics by mother nature. The most commonly used operators are mutations and crossovers.

Crossovers and mutations are made for different reasons. The idea of performing the crossovers is to create new individuals based on properties from the already existing individuals, by combining properties from two individuals, and thereby hopefully enhancing the fitness value. Mutations are made to prevent a search ending up in a local extreme of a function. Evolution speed in a genetic algorithm is controlled by the crossover and mutation rate. This rate is a probability. If the crossover rate is 1 then all individuals in the next population would be the result of a crossover. If the rate is 0 all individuals would just be passed onto the next population without changes. The crossover rate is the parameter r given to the algorithm. A mutation rate of 1 means that mutations are applied to all individuals in a population.

Crossover

Performing a crossover means that strings exchange substrings among each other. The crossover operator uses a crossover mask to perform the crossover between the parents strings. The difference between the crossover operators is basically how the bits gets selected by the crossover mask.

Single-point Crossover

This is the simplest way to perform a crossover between two strings. A random number n is picked, which is greater than zero and smaller than the length of the parent string. This number is used for dividing the crossover mask into two parts. The first part is marked using 1s and the last with 0s. The string 1111100000, where n is 5, will be used as crossover mask in the following example. The underlined parts form the first offspring and the overlined parts form the second offspring.

Crossover	1111100000	Crossover	1111100000
Parent1	$\underline{11001}$ $\overline{10011}$	Parent2	$\overline{00011}$ $\underline{10001}$
Offspring1	$\underline{11001}$ $\underline{10001}$	Offspring2	$\overline{00011}$ $\overline{10011}$

Two-point Crossover

This operator works by randomly choosing two number n_0 and n_1. These two numbers are used for making the crossover mask. n_0 and n_1 are both shorter than the parent string and $n_0 \neq n_1$, or else the two-point crossover will become a single point crossover. n_0 and n_1 divides the crossover mask into 3 parts: the characters up to n_0, the characters between n_0 and n_1, and the characters after n_1. The string 1110001111, where n_0 is 3 and n_1 is 6, is used in the following example. The underlined parts form the first offspring and the overlined parts form the second offspring.

Crossover	111 000 1111	Crossover	111 000 1111
Parent1	$\underline{110}$ $\overline{011}$ $\underline{1001}$	Parent2	$\overline{000}$ $\underline{111}$ $\overline{0111}$
Offspring1	$\underline{110}$ $\underline{111}$ $\underline{1001}$	Offspring2	$\overline{000}$ $\overline{011}$ $\overline{0111}$

Uniform Crossover

For every bit in the parent string, a random decision is made whether the bit should be part of the crossover mask or not. The string 1100010001, where the bits at position 1,2,6, and 10 were chosen to be in the crossover mask, will be used in the following example. The underlined parts form the first offspring and the overlined parts form the second offspring.

Crossover	1100010001	Crossover	1100010001
Parent1	$\underline{11}\overline{001}\underline{1}\overline{100}\underline{1}$	Parent2	$\overline{00}\underline{011}\overline{1}\underline{011}\overline{1}$
Offspring1	$\underline{1101110111}$	Offspring2	$\overline{0000111001}$

Elitism

The idea of Elitism is that every population may contain good individuals which should not be lost, due to the use of the genetic operators. Thus a small fraction (1% to 5%) of a population is passed on to the next generation without applying crossover.

Mutation

As mentioned earlier, mutations are performed to prevent that the search ends up in a local extreme.

The mutation operator is used for making small changes to the new individual. A random position in the parent string is selected and then inverted.

$$000\underline{0}001111 \longrightarrow 000\underline{1}001111$$

3.4.4 Selection and Ranking Individuals Functions

The learning method used by genetic algorithms is different from the one used by backpropagation, which is based on gradient descent. The gradient descent moves smoothly from one hypothesis to the next, meaning that the hypothesis n is very similar to hypothesis $n + 1$.

Genetic algorithms performs a randomized beam search that ends up in finding an individual who's fitness value is greater than the fitness threshold for the algorithm. Because of this random behavior, genetic algorithms are not likely to end up in a local extreme. A genetic algorithm produces hypotheses that vary in similarity because of the genetic operators, meaning that hypothesis n is very dissimilar from hypothesis $n + 1$. The algorithm needs a guide that helps it find the right individual, a naive way to do this is by using the roulette wheel selection described below.

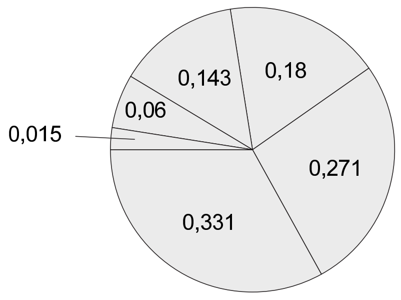
Roulette Wheel Selection The idea behind the Roulette Wheel Selection is to make a roulette wheel, where the size of the fields represent the probability that an individual gets selected. Such a wheel can be seen in Figure 3.18(a).

- The sum of all fitness values is found
- The probability of every fitness value is found

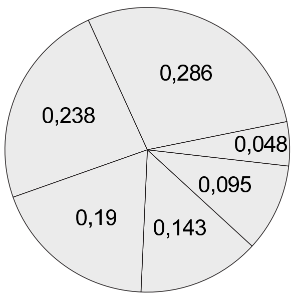
Fitness value	2200	1800	1200	950	400	100
Selection probability	0.394	0.323	0.215	0.17	0.0718	0.017

The fitness values in this example are scores of different robots (or individuals). The wheel can be found in Figure 3.18(a).

When selecting an individual, the wheel is spun. The greater the piece of the wheel an individual covers, the greater the chance this individual has of being picked.



(a) The wheel based on the probabilities of the individuals.



(b) The wheel after ranking has been performed.

Figure 3.18:

When using wheel selection a problem is known to occur when some individuals are more fit than the rest of the population. These individuals will quickly become a large fraction of the population. This will slow down the search process because of the reduced heterogeneity in the population. To prevent this problem, people have come up with different ways of selecting individuals of the population, one of which is the rank selection.

Rank Selection This method is used for preventing an individual with a high fitness from dominating the next population. This ranking of the individuals guarantees a more uniform selection of individuals, see Figure 3.18(b). A more uniform selection guarantees a greater spreading among the individuals, which again results in a better performing algorithm, in the form of reducing the chance of the algorithm getting stuck in a local extreme.

1. First the individuals are ranked according to their fitness.
2. Next the individuals are given a value from 1 to n, where n is the size of the population.

Fitness value	2200	1800	1200	950	400	100
Ranking values	6	5	4	3	2	1
Selection probability	0.285	0.238	0.190	0.142	0.095	0.04

With ranking selection the genetic algorithm will provide a robust way to drive the evolution forward.

Now that we have covered the Robocode game and several agent architectures and machine learning technologies, it is time to start designing our robot. The knowledge of what different machine learning technologies can provide, enables us to design a robust, yet flexible and adaptive robot design. The functionalities of each component of our robot must be considered and defined. Before doing this we will briefly establish a goal for the project in the following chapter.

4 Project Goals

After familiarizing ourselves with Robocode we have learned the following:

Execution time. The processing time a robot has per tick is limited, and therefore we have to limit our computations

Geometry. Terms like heading, bearing, velocity, etc. have been analyzed and will come in handy when designing and implementing a robot

Team communication. The ability to make a robot communicate with its teammates makes a collection of several robots capable of cooperating in order to pursue a common goal, and hopefully achieve a synergy effect

Having analyzed different agent architectures, the hybrid architecture was found most suitable for this project due to its way of maintaining its reactivity while still maintaining a world model, which makes the robots capable of sharing information about the world in which they operate. A world model enables us to save information about the movement of the enemies. The movement of an enemy robot is likely to follow a pattern of some kind, which is what neural networks has proven to recognize quite well, hence a neural network would be suitable for use in the aiming system.

Several types of machine learning technologies have been examined in order to reveal their strengths and weaknesses, giving us a basis for selecting what technologies to use for which tasks. The genetic algorithm was found to be well suited for optimizing weighting of parameters and therefore quite suitable for implementing the robots movement subsystem. Furthermore, a Bayesian network was chosen for target selection as each situation in which target must be selected must be compared to previously experienced situations.

Different types of team coordination have been discussed. Because of the world in which our robots are going to operate is quite dynamic, we chose to make use of a decentralized coordination, because if a teammate gets killed the other robots will still be able to continue just as before. Another reason for choosing the decentralized coordination model is the CPU time restriction. It could be problematic if a single robot should calculate actions for the whole team within the modestly sized time slice it is assigned.

> The overall goal of the project is to develop an autonomous robot for Robocode, which is able to learn from previous experiences as well as adapt to the environment and different opponents.
> The robot developed should be able to operate in a team, making decisions based upon information gathered by the whole team, without having a central coordination mechanism.

In Part II of this project we will design the various components of our robot, examining the subsystems of the robot and how these should interact. We will also discuss and design the incorporation of several machine learning technologies. In Part III we will explain the implementation of our robot and afterwards present test results and discuss whether or not our goals have been fulfilled. Furthermore, tests for finding the best configurations for the different components will be carried out.

II

Design

5 Component Design

This chapter describes the design of the code of the robot. Here the different modules and their tasks will be explained, and how the machine learning should be put into these modules. Furthermore this chapter will describe how this module design is mapped into a Java framework.

5.1 Module Design

This section describes the design of the modules used for the implementation of the whole team of robots. Figure 5.1 shows an outline of the whole design, which will be elaborated on in the sections below. These sections explain the basic behavior of each module in different situations, as well as which technologies will be used for implementing them.

Our model has three overall layers, namely *General Strategy*, *Strategic Modules*, and *Interaction Modules.* All of these layers are represented in Figure 5.1.

- The General Strategy layer is represented as two branches, *Offensive* and *Defensive.*
- The Strategic Modules layer is represented as three rows, *Retreat*, *Attack*, and *Acquire Target.*

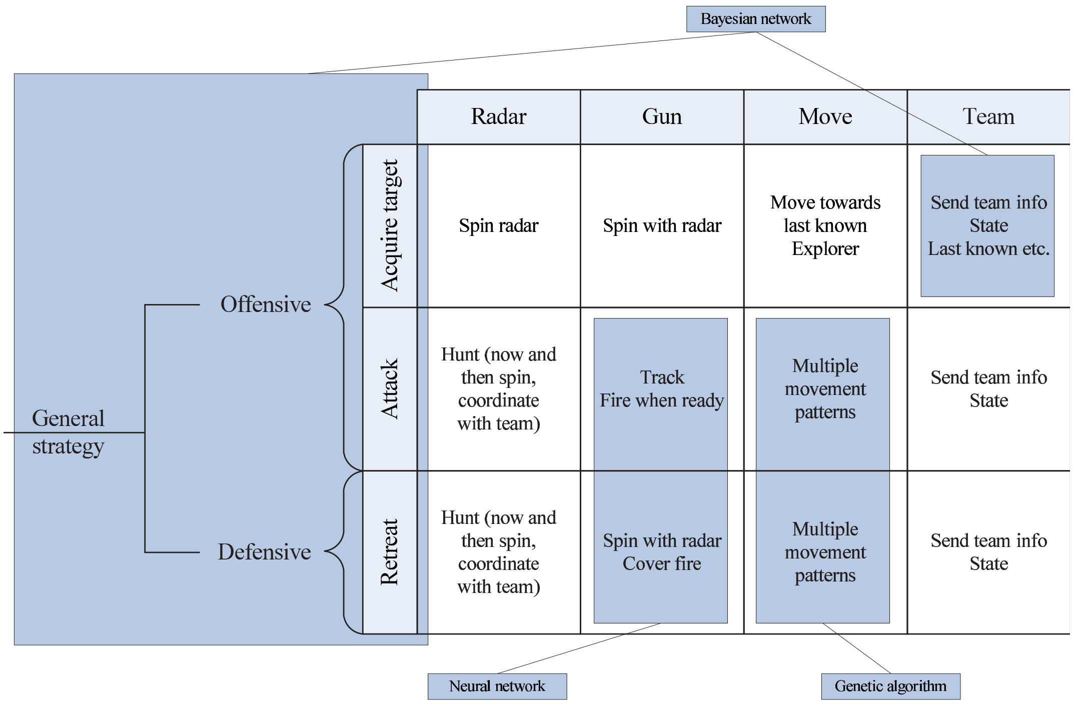

Bayesian network
Radar
Gun
Move
Team
General strategy
Offensive
Defensive
Acquire target
Attack
Retreat
Spin radar
Spin with radar
Move towards last known Explorer
Send team info State Last known etc.
Hunt (now and then spin, coordinate with team)
Track Fire when ready
Multiple movement patterns
Send team info State
Hunt (now and then spin, coordinate with team)
Spin with radar Cover fire
Multiple movement patterns
Send team info State
Neural network
Genetic algorithm

Figure 5.1: This figure shows an outline of the whole module design

- The Interaction Modules layer is represented as the four columns labelled *Radar*, *Gun*, *Move*, and *Team*.

5.1.1 General Strategy Layer

This layer is in charge of choosing an offensive or defensive strategy as the general strategy. The module asks each of the Strategic modules for their opinion on whether passing control to them will be profitable or not. These opinions are expressed as numerical values that determine which module to choose. How this value is calculated is explained in Section 5.1.3. When each of the Strategic modules has returned a value, they are compared to each other and when the optimal solution is found, the control of the robot is given to the module that offers this solution. For this task a Bayesian network is chosen in order to make an informed decision based on the opinions returned by the Strategic modules.

5.1.2 Strategic Modules Layer

This layer contains modules that are in charge of the tactical strategies. They are the ones that take care of the communication with the Interaction modules. As mentioned earlier, the Strategic modules layer contacts each of the Interaction modules which in turn give indications of how their current situation looks. These indications reflect whether each of the components is in a potentially good or bad situation. Each of these indications are summed and returned to the general strategy module as described above. When a strategic module is chosen, it will be in charge of by turn giving the control to each interaction module.

Acquire target

When the robot is in no immediate danger and it has no target to chase, this module is activated. When activated the robot attempts to find an enemy robot to attack.

Attack

When the robot is in no immediate danger and it has found a target to attack or if the possible gain of attacking an enemy is greater than the possible loss of energy, this module is activated. This means that the enemy robot found by the Acquire target module is pursued and attacked.

Retreat

If the robot is in immediate danger and there is no better solution than fleeing, this module is activated. When activated the robot attempts to find a route away from the current danger.

5.1.3 Interaction Modules Layer

This layer contains modules that are in charge of substrategies. A robot consists of a radar, a gun, and a vehicle. Therefore a module controlling each of these components is constructed. Furthermore, because a team is being implemented, a module handling the team communication is also constructed. When the Strategic module by turn gives control of the robot to each of these modules, they find an action to execute in the given situation.

Radar Module

Determining a radar scanning strategy is important in order to find the position of enemy robots. For this task either a hand-coded or an adaptive strategy could be chosen. There are both pros and cons for choosing either strategy. An advantage with the adaptive approach is that the robot in time will learn the movement patterns of the enemy robots and therefore will be able to predict their whereabouts, causing the robot to be able to scan for enemies more precisely than a hand-coded strategy would. However maintaining such an algorithm is rather expensive, and as the robot has a limited amount of time for choosing a move, this could pose a problem. Because the radar has limited functionality, and a predefined strategy is easy to make, a hand-coded model was chosen for this task.

Acquire target
A simple manner for acquiring a target would be to just spin the radar until an enemy robot is eventually scanned. If an enemy is within a radius of 1200 pixels it will be found within eight ticks, which is the time it takes for the radar to spin a full 360° rotation.

Attack
When in attack mode, a possible strategy for the radar is to let the radar follow the enemy that is to be attacked. This way the robot is always aware of where the enemy is. However this poses a problem if another enemy robot has sneaked up behind the robot, because this is not registered by the radar. To solve this problem, the radar is programmed to spin every now and then in order to find out if there are other enemies in the vicinity of the robot. However, if all of the robots are in attack mode, it would be profitable to make them communicate with each other so that they do not all scan the battlefield at the same time. This will help us maintaining a world model that is as up-to-date as possible.

Retreat
When fleeing from an enemy a sensible strategy could be to keep tracking the enemy we are fleeing from. This way we are always aware of where the enemy is, and we know if the enemy has changed direction or stopped the pursue. This however poses the same problem as when attacking, where another enemy could sneak up behind us and attack. Therefore the behavior of the radar should be the same as when attacking, namely to spin the radar once in a while.

Gun module

It is essential that the robot's gun is able to adapt to the situation that it is in. Both a hand-coded and an adaptive algorithm could be chosen for implementing the gun module. It is however hard to use a hand-coded approach to implement an algorithm that is able to predict how the enemy will move in the future. Instead we rely on a neural network to make this prediction for us.

Acquire target
This part of the module is hand-coded. When acquiring a target the radar simply spins while searching for enemies. We have made the radar and the gun spin together because, as described in Section 1.5, if the gun spins at the same time as the radar and in the same direction, the radar will be able to scan a full rotation in six ticks. Had the gun not been moving, the radar would take eight ticks to spin 360°. Since we want the radar to spin as fast as possible, we have chosen to spin both the radar and the gun simultaneously when acquiring a target.

Attack
When attacking an enemy the gun should fire at the enemy. The strategy of the gun is to point towards the enemy at all times and fire when it is able to hit that enemy. A neural network handles this task because it is able to recognize the movement patterns of the enemy, and will be more likely to hit the enemy than a hand-coded algorithm would.

Retreat
Even when fleeing from an enemy the robot should still fire at this enemy, hoping to drain some energy from it or maybe even kill it. The gun is controlled by a neural network, just as before.

Move module

Moving around the battlefield is essential in order to survive in a game of Robocode. If a robot is not constantly moving, it will be easy for enemy robots to hit it. Driving in a straight line will also be easy to predict for even very simple robots. In order to keep a robot as unpredictable as possible, some kind of randomization must be used. However, other robots using intelligent firing algorithms will find even this randomization easy to predict. In order to make the driving route as unpredictable as possible, part of this module is implemented using a genetic algorithm produces movement patterns optimized for avoiding bullets and walls.

Acquire target
When looking for a potential target, it is a good strategy to keep moving around the battlefield and scan for enemies. But the question is where the robot should drive to. There are two possibilities: If an enemy robot was scanned a while ago, it is quite possible that it is not far away from our current position and we should just start driving to where we spotted this enemy. If no robots were ever scanned or no robots were present where we have previously seen some, our robot should just explore the battlefield in a random direction which is unreachable by the radar at the moment.

Attack
In order to avoid getting hit, a genetic algorithm was chosen for the task of attacking. A genetic algorithm enables the vehicle to adjust its route in order to optimize its movement strategy according to the enemy's strategy. In order to make the movement pattern of the robot even more unpredictable it is possible to let the robot choose between multiple movement patterns. The robot will then at random choose which movement pattern to use.

Retreat
When fleeing from an enemy, it is essential that the robot does not flee in a straight line, which would make the robot an easy target. Therefore the same strategy as when attacking is chosen.

Team module

Because we are implementing a team of robots, there must be some kind of a communication between the robots. Of course it is possible to implement a team without any kind of communication between the team members, but this will not be very effective. It is better to make a decision based on facts obtained by all robots rather than facts obtained by only one robot, because gathering information from all robots provides a more detailed and up-to-date world model. The Team module maintains the world model so when a robot makes changes to its own world model the change is broadcast to the other teammates in order for them to have the most up-to-date world model.

The world model contains the following information:

- The position, heading, velocity, energy, state and possible target of teammates
- The position, heading, velocity and energy of enemies from the last scan

- The last 100 recorded positions of enemies

Acquire target
When acquiring a target, the robot "writes" to the world model that it is currently acquiring a target. Furthermore, when an enemy is scanned it is also written to the world model.

Because multiple enemies could be scanned in a single rotation of the radar and a teammate might meanwhile have scanned additional robots, a choice must be made of which target to select. This decision could be based on several facts, for example a robot could choose the enemy with the lowest amount of energy, or it could choose one that the teammates are attacking, or one who is attacking a teammate etc. The responsibility of choosing the right target lies with a Bayesian network that makes its decision based on information acquired from the robots own radar and the world model.

Attack
When in attack mode the robot writes to the world model that it is currently in attack mode and also which robot it is attacking. This way the other robots are able to take this information into consideration when choosing a target to attack.

Retreat
When in retreat mode a robot will write to the world that it is currently in retreat mode.

5.2 Framework

As described in Section 5.1, the design is layered and split into modules. This design can more or less be mapped directly onto a class diagram of the Java classes used for the implementation of the robots. Figure 5.2 shows how the classes interact with each other. The main class is the `OurRobot` class which inherits from `TeamRobot` as described in Section 1.6. The `OurRobot` class is responsible for communicating with the strategy classes. The class contains four methods, one for deciding what to do and three for giving control to each of the strategy classes.

- `decideWhatToDo()`
- `doAttack()`
- `doAcquire()`

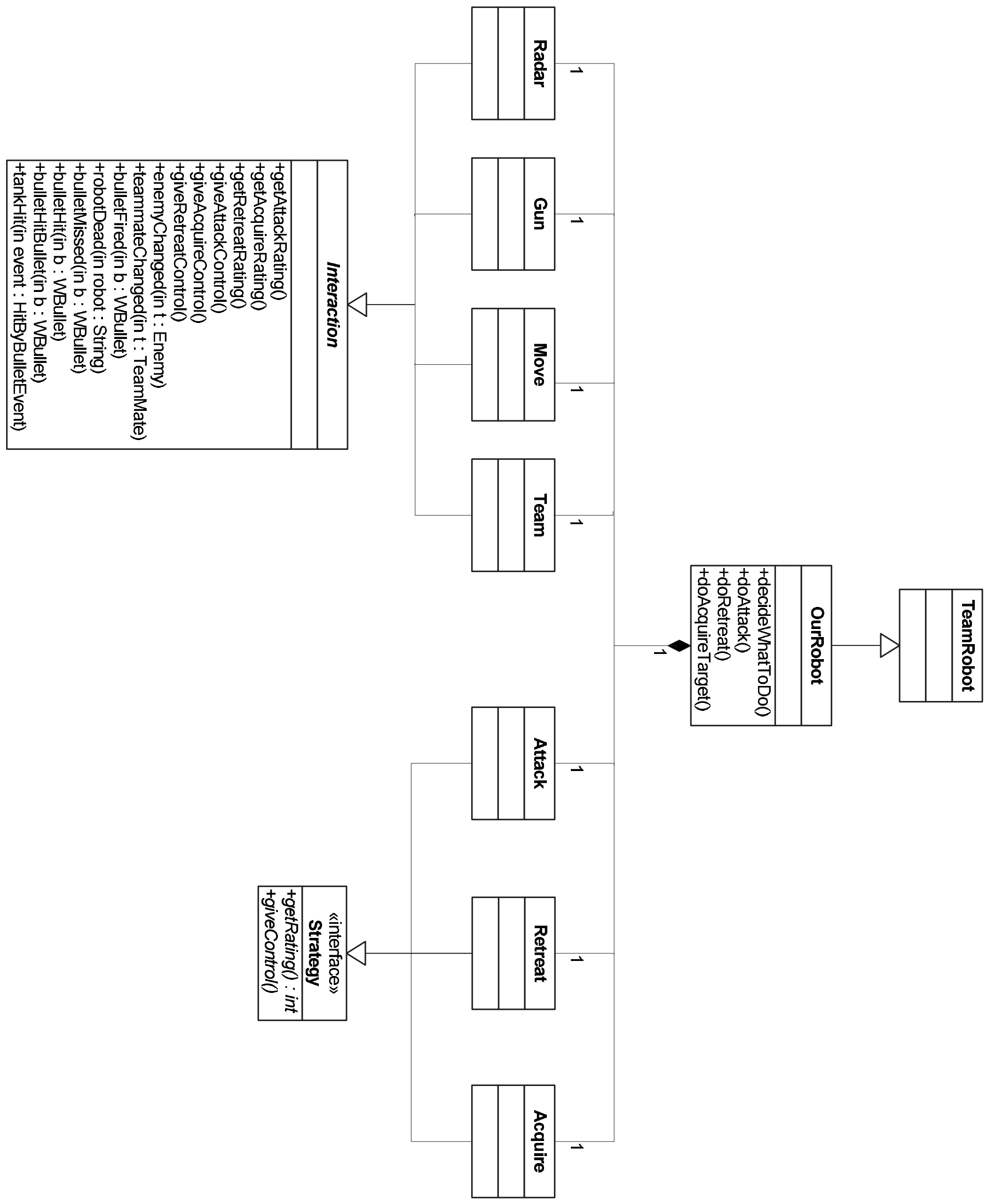


Figure 5.2: This figure shows the overall framework of the code

- `doRetreat()`

All three strategy classes have identical interfaces, as all of them should contain exactly the same methods.

Another abstract class is defined for the interaction classes which must be extended by each module. The following methods is contained within each module:

- `getAttackRating()`
- `getAcquireRating()`
- `getRetreatRating()`
- `giveAttackControl()`
- `giveAcquireControl()`
- `giveRetreatControl()`

The methods `getAttackRating`, `getAcquireRating`, and `getRetreatRating` are used for getting an estimate of how profitable it will be to hand over control to the attack, acquire, or retreat strategy modules, respectively. This rating is an integer which is weighted as shown in Figure 5.1. The methods `giveAttackControl`, `giveAcquireControl`, and `giveRetreatControl` are used for giving the aforementioned strategy modules control of the robot for one tick.

Because the `OurRobot` class extends the `TeamRobot` class the robot will be able to perform non-blocking calls in order to let the robot perform multiple actions in a single tick. When an interaction module is controlling the robot, non-blocking calls are used–for example `setFire(2)`. When a strategy is chosen, all interaction modules make their move using non-blocking calls, and when every module has made their move, the `OurRobot` object calls the `execute()` method and all the moves are executed simultaneously.

In Robocode it is preferred that every robot is put into a package in order for the Robocode environment to distinguish which classes belong to which robot. Therefore a package `robopack` is defined that contains all the robot code.

Now we have divided the robot design into modules, each with a specific task. By doing this, each module in the robot will have a well defined assignment. The following chapter will discuss how the various types of machine learning is applied to the tasks that our robot must perform.

6 Machine Learning Design

This chapter will provide you with descriptions of how the parts of our robot that utilize machine learning techniques are designed, as well as the thoughts put into these designs.

6.1 Target Selection

As mentioned in Section 5.1.3, a Bayesian network is chosen for the implementation of the target selection. Implementation of the Bayesian network used for target selection will be explained in Section 7.1. A Bayesian network has several advantages in the context of target selection. First of all, a Bayesian network is a compact representation of the joint probability, in which it is easy to look up probabilities based on evidence. Secondly, it is easy to model a Bayesian network that handles target selection. This is done by constructing variables for each input that influences the target selection. Each of these states must be mutually exclusive and complete, see Definition 7 in Section 3.1. By giving evidence to these variables the Bayesian network will come up with a rating r, indicating how profitable it will be to choose some target. This rating is defined to be

$$r = \text{probability}_{\text{kill}} \tag{6.1}$$

This way, the rate for each known enemy in the world model is calculated, and the one with the highest rate is chosen as the target.

6.1.1 Variables

Now the question is which input data should be used for input variables. One could argue that every single known piece of data should be modelled as a variable in order to get the best result. This, however, could pose a problem, as the probability tables would be rather comprehensive and demanding as far as calculation goes. Therefore a choice must be made, choosing the most important inputs to be variables. All information about the enemies are available from the world model, which holds information collected by the team. Some of the information available is chosen to be left out of the Bayesian network, because it is not that important when choosing a target. Some of the data left out is the velocity and heading of the robots, as the gain of including this knowledge is approximated to be less than the overhead of having to do the extra calculations. The following data has been chosen to be enough to base a decision upon.

The energy of our robot. It is important to take into account how much energy the robot has. This could prevent a robot from choosing to target an enemy with significant more energy than it has itself.

The energy of the enemy. As mentioned above, it is important to know the difference between the amount of energy the enemy has and the amount that our robot has. The reason for this is that when choosing between two targets, it could be more profitable to choose the one with lesser energy if our own energy is low.

Distance to enemy. When choosing a target, it is important to take into account the distance to the enemy. This could prevent a robot from choosing a target that is far away when there is an equally good target that is closer to our robot. This will also prevent the robot from putting itself in danger by driving across the entire battlefield in order to attack an enemy.

The enemy's distance to its nearest teammate. If our robot is by itself and is choosing between two targets, it could be profitable to choose an enemy that is also by itself. Therefore it is important to take into consideration how far the enemy is from one of its own teammates.

How many teammates have chosen a given target. Sometimes it could be a good idea for teammates to attack the same enemy. However, it could also be a disadvantage if all the robots choose to attack the same target at the same time. Therefore it is an advantage to know how many teammates that have already chosen a certain enemy as a target.

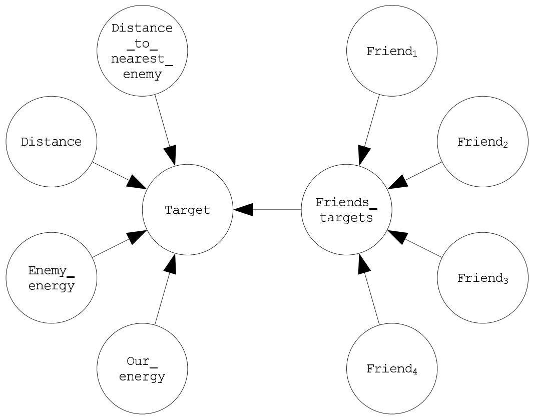


Figure 6.1: The design of the Bayesian network

Win or lose. When a robot dies it must be specified whether or not we win using the current configuration.

Now the question is how to decide if the chosen target was a good or bad choice in order to update the Bayesian network. How the network is updated is elaborated on later in this section. If the chosen target dies before our robot, then the choice was probably a good one. Similarly, if our robot dies before the target, then the choice must have been a bad one.

The design of the Bayesian network is outlined in Figure 6.1. It consists of ten variables, and every time the Bayesian network is used there will be evidence on eight of them. In order to get the rating r mentioned earlier, one only needs to read the `Target` variable. In the following the symbols T, FT, D, DE, E, OE, F are short for the variables: `Target`, `Friends_targets`, `Distance`, `Distance_to_nearest_enemy`, `Enemy_energy`, `Our_energy`, `Friend`$_n$.

Definition 11 *The probability of winning is* $P(T = win|FT, D, DE, E, OE)$

The variable FT (`Friends_targets`) is defined as

Definition 12 *Friends' target is* $P(FT|F_1, F_2, F_3, F_4)$

6.1.2 States

One of the problems with Bayesian networks is that there must be a state for every possible input. This, however, is not possible. For example, distance is measured using a `double` and a state for every number must be constructed. This would make the number of states huge and thereby the probability table would be enormous. The solution to this problem is to split the input range into intervals and construct a state for each of these ranges. This is also known as discretization of continuous variables.

But how do we create these ranges in such a way that the network will still be able to make good decisions. For the energy variables, reasonable ranges could be:

- 0-20
- 21-50
- 51-120
- 121-∞

This way a robot will be able distinguish between "normal" robots having one of three different levels of energy. Furthermore it will be able to see if one of the enemies in sight is the leader, since it will have an energy amount of more than 120. This is of course only true at the start of a game.

For the distance variables, reasonable ranges could be

- 0-100
- 101-300
- 301-700
- 701-1200
- 1201-∞

This will make the robots able to distinguish whether other robots are very close, close, just barely within radar range and outside radar range. It is important to construct one of the ranges with respect to the radar range, because it is a good idea to choose a target that the robot is able to scan.

The only state left that needs specifying is the $\texttt{Friend}_\texttt{n}$ variable. This must contain the states, in which a teammate can be. Intuitively, a teammate can be in the following states:

- No target

- Dead
- Same target
- Another target

From this information the robot will be able to determine whether the target it has selected is the target of a friend or not.

6.1.3 Updating the Network

The only thing that needs to be done now, is to update the Bayesian network. This is done on-line when running the Robocode program. The output variable is `Target` , which specifies the probability of the robot winning. When choosing a target, information about the enemy and the state of the robot itself is saved, and when either the enemy or the robot dies, the Bayesian network is updated accordingly. The approach used when updating the network is fractional updating, see Section 3.2.2. So every time a positive example is experienced, the probability table for the given configuration is incremented.

6.2 Overall Strategy

For the task of selecting an overall strategy, another Bayesian network was chosen. The network, which can be seen in Figure 6.2, consists of four variables. The output variable is called `overall_strategy` and has three states, $\texttt{acquire\_target}_{\texttt{out}}$, $\texttt{retreat}_{\texttt{out}}$, and $\texttt{attack}_{\texttt{out}}$. There are three input variables as well, `Acquire_target`, `Retreat`, and `Attack`. All of these have two states, `rating` and `1-rating`.

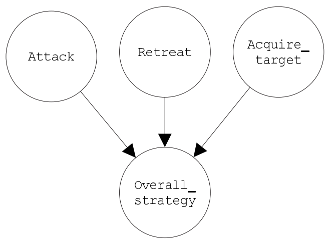


Figure 6.2: Design of the Bayesian network.

The `probability` of e.g. `retreat` is the sum of the return values of the `getRetreatRating()` method on each of the four interaction modules. To make sure they map onto $[0;1]$, the Sigmoid function is used as seen in Equation 6.2

$$P(\texttt{retreat} = \texttt{rating}) = \mathrm{sigmoid}(\texttt{retreat.getRating()}) \tag{6.2}$$

Another question is how to update the network. This should be done by saving all decisions made by the network during the round and afterwards updating the network positively, using fractional updating, if we win, otherwise negatively.

The following abbreviations are used in the section below: OS is `Overall_strategy`, AT is `Attack`, R is `Retreat` and AQ is `Acquire_target`. `1-r` is short for `1-rating` and r for `rating`

When a positive result has been found all the decisions made should be updated in the following way, e.g. for $\texttt{retreat}_{\texttt{out}}$ $P(OS = \texttt{retreat}_{\texttt{out}} | AT = \texttt{1-r}, R = r, AQ = \texttt{1-r}) = (n_1 + 1, n_2, n_3)$ where n_1 is the values from the last update.

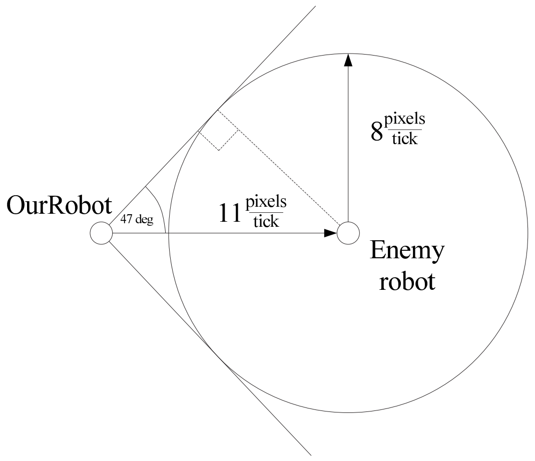
A negative result should be updated as $P(OS = \texttt{retreat}_{\texttt{out}} | AT = \texttt{1-r}, R = r, AQ = \texttt{1-r}) = (n_1, n_2 + \frac{1}{2}, n_3 + \frac{1}{2})$

$P(OS|AT = r, R = r, AQ = r)$, $P(OS|AT = r, R = r, AQ = \texttt{1-r})$, $P(OS|AT = r, R = \texttt{1-r}, AQ = r)$, $P(OS|AT = \texttt{1-r}, R = r, AQ = r)$ and $P(OS|AT = \texttt{1-r}, R = \texttt{1-r}, AQ = \texttt{1-r})$ should all be set to $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$

6.3 Aiming System

In order to design an aiming support system for the robot, we took a look at the most common techniques used in other robot designs. This seemed to be a difficult task, as many of the robot designs claimed it was difficult to implement a well working support system for aiming. Many suggested a neural network for this part of the design.

This preliminary research ended with us focusing on neural networks, and how to utilize a neural network for aiming. As discussed in Section 3.3.1, neural networks are based on the human brain's ability to quickly recognize patterns and translate these into actions. Just consider how fast a child learns to calculate the trajectory of a thrown ball, and how to compensate for various factors like wind etc. The primary reason for choosing a neural network, was to capture this ability to create an adaptive guiding system for aiming that was able to recognize the movement patterns of the opponent robots and thereby predict in which direction to shoot.



OurRobot
47 deg
$11 \frac{\text{pixels}}{\text{tick}}$
$8 \frac{\text{pixels}}{\text{tick}}$
Enemy robot

Figure 6.3: The intuition behind the maximum span of the correction angle.

6.3.1 Input and Output Representation

To decide what the input to the neural network should be, the first thing to do was to look at what the output should be. This way, it could be determined what influenced the output, thus making it possible to construct a suitable network representation.

Output

The target selection, described in Section 6.1, is done by a Bayesian network. The objective of the neural network is to supply, or rather predict, an angle based on previous observations. This output angle of the network is added to the already calculated angle that the gun needs to turn, in order for the gun turret to point straight at the target. The output can be seen as the correction needed to be applied to the angle, if the target is not stationary. When shooting at a stationary target, the neural network would not need to correct the angle, and should output a value of zero.

There are a few observations to be taken into account. First we calculate the spanning angle of the correction needed, as seen in Figure 6.3. This is done because it is not possible for the target to actually move more than $8 \frac{\text{pixels}}{\text{tick}}$, and the bullets fly with a velocity of at least $11 \frac{\text{pixels}}{\text{tick}}$. This gives us a spanning angle of approximately 94 degrees, see Equation 6.3, and since the target cannot move outside this targeting cone before it is either passed or hit by the bullet, we can narrow down the output of the neural network from 360 degrees, to span over 94 degrees in the interval $[-47, 47]$.

First we calculate the angle using the rules that apply for right-angled triangles:

$$A = \sin^{-1}(\frac{8}{11}) \approx 47^\circ \tag{6.3}$$

This gives us an angle of approximately 47° that we multiply by two to get the mirrored angle for the complete span. We now set the output of the net to be a value in the interval $[-47; 47]$.

For a robot to shoot, it needs to decide on where to shoot, and with what power. Now that the neural network is able to calculate the angle correction, and together with the relative bearing of the gun, we know how much to turn the gun turret, but not what power to shoot the gun with. One idea is to incorporate this power into the neural network, and let the net learn by itself what the appropriate power value should be in a given situation. It was however decided to leave this out of the design of neural network in order to simplify the representation and give us a precise model for anticipating the correction angle of the shot. It was assumed this smaller representation would give a better result.

Another idea was to use some form of decision support system to learn the appropriate power to fire the gun with, but instead it was decided that a static function should be used for calculating the power of a shot. Equation 6.4 represents this static function.

$$pow(dist) = \begin{cases} 3.0 & \text{if } dist < 500 \\ 1.0 & \text{if } dist < 1200 \\ 0.0 & \text{otherwise} \end{cases} \tag{6.4}$$

where *dist* denotes the distance to the target.

Input

Based on what the output is, we can examine the various inputs to be taken into account. Below is a list the basic things that can influence the correction angle for the shot.

- Enemy heading
- Enemy velocity
- Gun bearing

- Distance

Enemy heading is the basic information that gives us a relative heading for the target at hand. For the neural network to successfully calculate the correction angle, the aiming support system needs to know whether the target is moving to the left or to the right, relative to when the gun is pointing straight towards the target. Consider a target that is driving straight ahead heading right, which would give us a relative heading of 90 degrees. The aiming support system should ideally learn to shoot a bit to the right of the target, in order to hit the robot.

Enemy velocity gives us the information of how fast the target is moving. Now that the aiming support system knows the direction in which the target is moving, it needs to know how fast the target is moving, in order to successfully calculate the correction angle.

Gun bearing gives us the relative angle of where the gun turret is currently pointing. If the gun turret is $30°$ off from where it needs to be pointing, it will have to turn $-30°$ before it can actually shoot. This gives the support system a chance to adjust for the time it takes the gun turret to turn from its current angle to where it needs to be. This will buy us time since it takes a few time units to achieve the correct position of the gun turret.

Distance is the raw distance to a target. With the target heading, velocity, and distance, the support system has enough information to make a qualified guess of where the target will be when the bullet reaches its proximity. There is one thing to be noted though. The bullet speed will have an impact on the correction angle, but this information is actually hidden inside the distance input, because the speed is a function of the distance. So given a distance, the support system will know exactly how fast the bullet flies, and will thereby be able to predict the right correction angle.

Model

We can use the input and output of the neural network to create a representation of the neural network associated with the aiming support system. In Figure 6.4 a simplified model of the neural network can be seen.

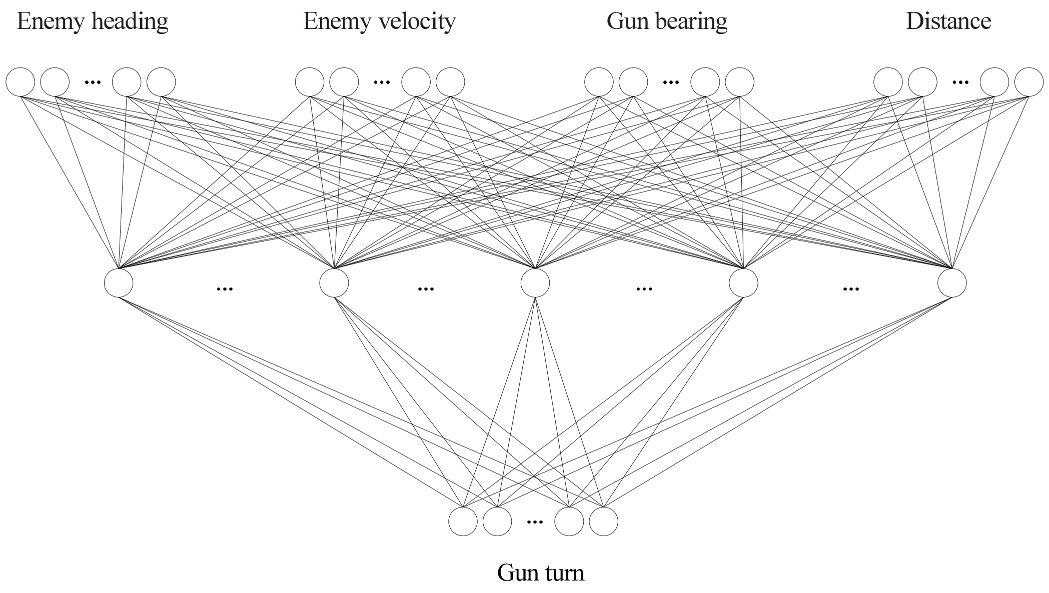


Figure 6.4: A simplified overview model of the neural network.

6.3.2 Training the Network

To train the network, the backpropagation algorithm was chosen. The algorithm is described in Section 3.3.5, and it is as far as we know one of the most commonly used algorithms for training neural networks. Further reasons for choosing the backpropagation algorithm are that it is easy to implement, and that it is usually described as a good approach that generally performs well according to Jörg P. Müller[8]. One of the drawbacks with the algorithm is that it is relatively slow, but the alternatives seemed to be either just as slow or even slower. That being said, faster alternatives are available, such as the Simplex algorithm [2]. But this was ruled out due to uncertainties regarding whether or not the Simplex algorithm would converge to the right value in our situation. Another drawback is the problem of ending up in a local minimum, as described in Section 3.3.5 and 3.3.6. This problem could be solved by using Simulated Annealing [1] algorithm, which guarantees to end up in a global minimum. However, the Simulated Annealing algorithm is very slow [1], so the final decision was to accept the drawbacks of the backpropagation algorithm, and use this as a training module. This choice was also based on the fact that it is such a well documented and widely known algorithm.

Momentum

The last thing to be considered during the design of the aiming support system, was the adding of a momentum as described in Section 3.3.6. Using momentum should give us less chance of ending in a local minimum. It is meant as an optimization to the backpropagation algorithm, and should serve as further assurance that the network actually learns the right things.

6.4 Move Patterns

It was decided to use a genetic algorithm for making decisions of where to head for in order to control the motion of the robot. These decisions should be based on the following:

- The robots own position, velocity, and heading
- The position of the opponents (their x and y coordinates), velocity, heading, and bearing
- The position of the teammates, velocity, and heading
- The position of all robots and bullets, as well as their heading and velocity
- The size of the battlefield

When the information in the list above have been used to form a basis for the next move, something as simple as a pair of coordinates will be output. This set of coordinates can be translated into an angle and a distance, which can be used directly in Robocode. The general structure of a genetic algorithm can be seen in Figure 3.4.1. Because of this knowledge, we come to the conclusion that the design criteria for our algorithm are:

- The algorithm must support on-line learning
- The algorithm must be able to perform on inputs of different lengths
- A fixed structure format to hold the vector functions

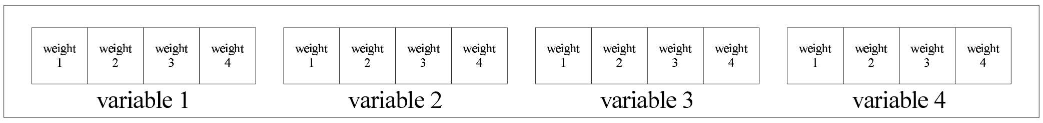


Figure 6.5: The internal distribution of the vector function

Because the algorithm must be capable of performing on-line learning, it must be able to produce a new population without restarting the Robocode environment. A data structure containing an array of values can be used to store each individual in a population, and will be able to be modified on runtime. Vectors are known from mathematics to be a good way to describe motion, and can be easily stored in arrays.

Vector functions are normally considered as a good description of motion in a plane, and they also provide a pair of coordinates as output, just as we need. Therefore these coordinates were chosen to be the target of the evolution performed by the genetic algorithm.

Now we will describe how to implement the following things:

- How to guarantee a fixed structure in which to hold the vector functions.
- The genetic operators (mutation and crossover)
- A way to collect the fitness values of the vector functions used.
- A Fitness_threshold of the algorithm

A fixed structure could be guaranteed if the algorithm had the following extra inputs: the number of variables and the order n of the polynomial used for approximating a value of a given variable. If the number of variables and the order of the polynomial are the same for all the vector functions, the vector functions could be stored as an array containing only the weights, as shown in Figure 6.5. Since it would be nice to be able to express weights as floats, we have chosen to represent the weights with a `double` in Java. This also cope well with the fact that all movement in Robocode is measured in doubles.

The crossover operator could, if the vector functions are kept, be made on the variable level or at the weight level. The weights or variables could be copied to two new arrays as described in Section 3.4.3.

The mutation operator could be implemented by picking a random weight in the array, and then changing it to something else.

The fitness value could just be a number stored together with the arrays. Since it is not easy to tell if a given move is a bad or good one, it would be nice to have a fitness value that would represent an overall performance of the vector function. The following things are events in Robocode, and should provide some information about how well the move layer works:

1. The robot collides with another robot
2. The robot collides with a wall
3. The robot gets hit by a bullet
4. A fired bullet leaves the battlefield
5. A fired bullet hits an enemy robot

The first four items should be given negative numbers. The first three items are given negative numbers because it involves a loss of energy. The fourth item is given a negative number because the gun layer missed a target. The reason for this is that the move layer should always position the vehicle so that gun layer has the best chance of hitting a target. If an enemy target is hit, we reward the move layer for positioning the vehicle so the gun was able to hit the target. All of this combined should make it possible to make an overall fitness value for a given vector function.

The fitness threshold should represent a goal for when a sufficiently good vector function has been produced. Since there is just one positive event that can occur, a high fitness value must indicate that a good vector function has been found.

We now have an algorithm that is capable of taking a different number of variables, a format for which it is easy to implement genetic operators, and a way to collect the fitness values for the vector functions.

This chapter has finalized the analysis and design of our robot, as well as outlined the machine learning techniques needed to implement decision support systems in our robot. The following chapter contains detailed descriptions of our implementation.

III

Implementation, Test and Conclusion

7 Implementation

In this chapter we will provide in-depth descriptions of how the highlights of various components of our Robocode robot has been implemented.

7.1 Target Selection

We have decided to implement the target selection Bayesian network only, since the process of implementing the Bayesian network for selecting the overall strategy would be very similar to implementing the target selection. This was also due to the fact that it was decided to put more effort into other areas of the implementation. This section will continue using the abbreviations from Section 6.1.

When implementing the Bayesian network it was decided to remove the `Friends_targets` variable since we only need the *input variables* and the single *output variable*, so that the Bayesian network looks like the one in Figure 7.1(a). This makes the network a bit larger. In `Target` alone there needs to be specified $|D| \times |DE| \times |E| \times |OE| \times |F_1| \times |F_2| \times |F_3| \times |F_4| \times 2 = 204800$ probabilities. This, although possible, is not considered a good solution, simply because the chance of seeing the same configuration twice in a game is very little, and therefore there needs to be a tremendous amount of off-line learning to get a good reference.

7.1.1 Size Does Matter

The `Friends_targets` variable is reintroduced and $\texttt{Friend}_{\texttt{\{1-4\}}}$ is removed, so that the `Friends_targets` variable now represents all of the other friends,

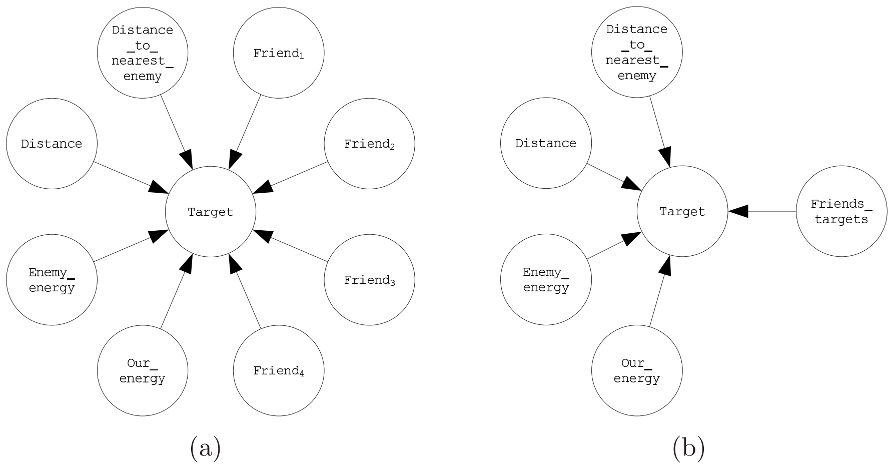


Figure 7.1: Steps taken towards minimizing the Bayesian network size

see Figure 7.1(b). This would require that there be specified $4 \times 4 \times 4 \times 4$ probabilities in this variable. It was also decided that it was not necessary to know if your friends were dead or had no target, but only whether they had the same target or not. Furthermore we decided to reduce the number of states to three, or more precisely the states `0`, `1`, and $\geq$ `2`, meaning that 0 teammates has selected the target in question, 1 teammate has the target in question, and 2 *or more* teammates has selected the target, respectively. This has reduced the total number of probabilities that needs to be specified to 2400, and reduced the size to $\frac{2400}{204800} \approx \frac{1}{85}$ of the original.

7.1.2 Representation

The variable `Target` has two states, one that represents winning the battle against the currently selected target and another that represents loosing the battle. We have evidence on all other variables, so there is no need to store their probabilities. Because the `Target` variable has two states and since the network will use fractional updating, it is sufficient to store only three integers: the *number of times we have seen a given configuration (s)*, the *number of times we have won (n_w)* in this configuration, and the *number of times we have lost (n_l)*. The probability of winning is $\frac{n_w}{s}$ and the probability of loosing is $\frac{n_l}{s}$. From Definition 7 we know that $\frac{n_w}{s} + \frac{n_l}{s} = 1 \Leftrightarrow \frac{n_w + n_l}{s} = 1 \Leftrightarrow n_w + n_l = s$, meaning that it is not necessary to store s, hence there only needs to be stored two integers for each configuration. The probability of winning is $\frac{n_w}{s} = \frac{n_w}{n_w + n_l}$.

```
private static int[][][][][][] weights = new int[4][4][5][5][3][2];

public double getScore(a1, a2, a3, a4, a5) {
    return weights[a1][a2][a3][a4][a5][0]/(weights[a1][a2][a3][a4][
        a5][0]+weights[a1][a2][a3][a4][a5][1]);
} public void updateScore(a1,a2,a3,a4,a5,whowon) {
    weights[a1][a2][a3][a4][a5][whowon]++;
}
```

Listing 7.1: Pseudo code for part of the Bayesian network implementation

7.1.3 Data Storage

The probabilities could be stored in a 6-dimensional array as seen in Listing 7.1, and then simply incrementing one of the values will give us the new probability table. Also getting the probability of an event occurring is only two memory lookups and a few simple arithmetical operations, making this representation very fast and small in size.

7.1.4 Initialization

When creating this Bayesian network, it is obvious that if we initialize all the weights to 0, there would be a `division by zero` error on a lot of `getScore()` method calls. It was then decided to initialize every entry in the array to 1, making the probability of a given configuration 0.5 initially. This also has the advantage that nothing will ever have a probability of 0.

7.2 Aiming System

We designed the neural network as a feedforward net, using the backpropagation algorithm. Implementation of the neural network is a straightforward process, as the algorithms are very easy to implement in Java. After implementing the neural network, we had to decide which parameters to feed the network, and which output it should yield. The aim function is assigned a target, and is then in charge of deciding how much the gun shall be turned before firing, in order to hit. The neural network will be fed a binary string. We decided that the neural network should have the following parameters as input:

Enemy relative heading. This is a number between 0 and 360. In order to represent these numbers we need nine bits.

Enemy velocity. This is a value between -8 and 8. We need to represent 17 different values, and so we need five bits. We do not want the neural network to deal with negative numbers, so before feeding this variable to the neural network, we add 8 to it.

Our gun bearing. The number we need to input is in the same format as the enemy relative heading, so we need nine bits for this value.

Distance to enemy. Since the largest possible battlefield in Robocode has a size of 5000 by 5000, the largest possible distance to an enemy is

$$a^2 + b^2 = c^2$$

$$\sqrt{5000^2 + 5000^2} = c \approx 7071$$

for which we need 13 bits.

This gives us a total input of 36 bits.

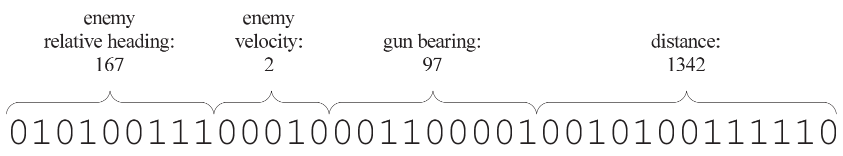


Figure 7.2: An example of a bit-representation input to our neural network.

As described in the design of our neural network in Section 6.3.1, we want the network to output an angle, telling us how much the gun shall be turned, anticipating the target's movement, in order to hit the enemy robot. For this we need to represent up to 94°, which is seven bits, since it is only the correction to an angle and not the exact angle.

One training example is a bit-representation of the input followed by a bit-representation of the target output.

```
010100111000100110101110010100111110 1001110
```

The last seven bits represent the optimal output from the network given the 36-bit input. We have two ways of training the network with such an example: off-line and on-line learning.

7.2.1 Off-line Learning

As a result of the restriction in how much CPU time a robot can use in each tick, it could be a necessity to teach the robot outside the game. To use the off-line learning, we play some games in Robocode and collect relevant data, which is then written to a file. After we have collected the data, we construct a neural network with random weights or weight from an previous saved weight file. The backpropagation algorithm is used on the training examples. The number of times the data is passed through the network is dependent on how accurate we want the network to be. If we use too large a number of hidden neurons we get the risk of overfitting the network, so that the network will only recognize those exact examples and nothing else. When all training examples have been passed through the backpropagation algorithm, the weights are saved to disk. This file of weights can then be used when battling in Robocode.

7.2.2 On-line Learning

Alternatively, by using on-line learning, we can battle against other robots and be able to learn to anticipate movement patterns of the enemy while competing. In principle, on-line learning is almost the same as off-line learning, the difference being that when learning on-line, we learn while playing Robocode. Instead of writing the input data to a file, the network receives it as input immediately, and updates the weights while playing. On-line learning is done for each shot fired by the gun like this:

1. Input x to the neural network and calculate the output
2. Shoot, corrected by the output angle
3. Save the data used for shooting
4. Wait to receive either a hit or a miss event from the bullet fired
5. Construct a training example from the saved data and the hit/miss event
6. Train the neural network with the training example using the backpropagation algorithm

In step five we construct a training example using the saved data and the information from the received event, plus the information collected since the bullet was fired. Using all of this data we will know the exact movement of the target and we can construct an algorithm to calculate the optimal shot for the circumstances the robot was in at the time the bullet was fired. When calculating this we get both the input and the output for the training example, and when we have this information about the optimal shot we can update the weights in the network accordingly.

Figure 7.3 shows how the algorithm calculates the optimal shot for updating the neural network. The circles represent recorded positions of the enemy target and the straight lines represent how far the bullet was able to travel towards the enemy's position. The filled circle is the first instance where the bullet was able to actually hit the target. This shot becomes the optimal solution as the target value for the training example.

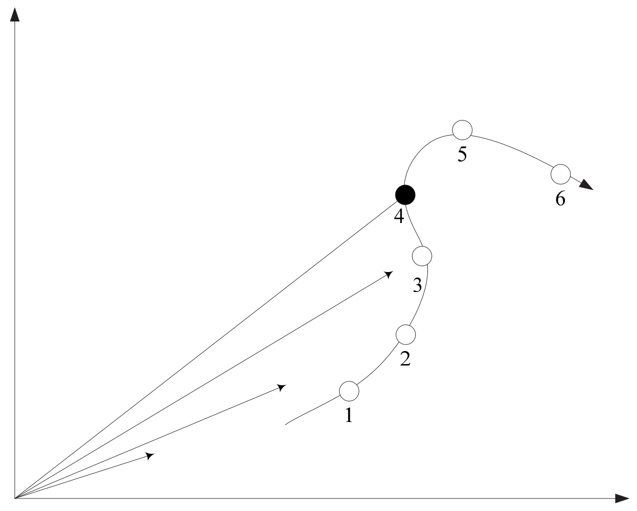


Figure 7.3: Calculation of the optimal shot

We only run the training example through the backpropagation algorithm once because of the time restriction that applies. So after each shot the weights will be updated and thereby making the robot's aiming support system better for each bullet the robot fires.

7.3 Move Pattern

This section will explain how the genetic algorithm and its utilities are implemented to fit the Robocode environment. The algorithm performs on-line learning for the following reasons: The fitness values of the single vector functions are collected, and the genetic operators are applied during the execution of Robocode. In other words new populations are created during execution.

The structure

It was chosen to implement an algorithm that produces vector functions. The vector functions could then be called in every tick to calculate a new pair of coordinates to head for. The algorithm has the following parameters:

- Size of the population
- Mutation rate
- Elitism rate, which tells how much of a population will be moved unchanged from one population to the next
- The number of variables. For example $f(x, y, z)$ has three variables, x, y, and z.
- The order of the polynomial that should be used to approximate a variable input.
- The size of the weights that are used in the equations.

The algorithm is implemented using two classes, `GA` and `MovePattern`.

The `GA` class, which has a constructor that takes the parameters listed above, holds the main loop of the algorithm and acts as a container for all the `MovePattern` classes that are currently in use by the algorithm.

The `MovePattern` class holds a movement pattern, which is basically a vector function with added methods for mutation and crossover, plus a method for calculating the output of the vector function. The vector function part is stored in two arrays of doubles, one array for calculating the x coordinate and another array for calculating the y coordinate. In Figure 6.5 one can see how these arrays are constructed. The calculate method is used in every tick to calculate a new set of coordinates to head for. The doubles stored within the arrays are used as coefficients in the equations within the vector function. An example of one of these equations can be seen in Equation 7.1, where the w_n's are the coefficients and x is the input.

$$w_1 \cdot x^3 + w_2 \cdot x^2 + w_3 \cdot x + w_4 \tag{7.1}$$

The calculate method evaluates every equation within the vector function and returns the total summed result. This procedure is done for both of the arrays.

The `MovePattern` class also contains its own fitness value. How this value is collected and used by the algorithm will be explained later in this section.

Crossover. For every crossover operation that is performed, a crossover mask is made. The crossover mask is an array of 0s and 1s with length equal to the number of variables we are performing the crossover on. The crossover type used by the algorithm is called uniform crossover and its principal operation can be found in Section 3.4.3. It was chosen to make the crossover on the variable level. The idea behind this is that when a good set of weights are found there is no reason for ruining these, which would have been the case if the crossover had been performed on the weights level, making the crossover produce almost random data, which is the job of the mutation operator.

Mutation. This operator is simple. A random number between 0 and the length of the array, in which the movement patterns are stored, is picked and then changed into another number that is provided along with the interval as input. This is done for both arrays.

The Algorithm

Now we know how the different parts of the algorithm are implemented, but it has not yet been clarified how the different parts work together. When Robocode is started, and there exists no file with data collected earlier, a number of random movement patterns are created. The number of movement patterns created depends on the input given to the algorithm. If there exists a data file, the movement patterns from the file are loaded. Every movement pattern is given a number of rounds to collect a fitness value. When all move patterns have received a fitness value and have completed their number of rounds, the following steps are performed

1. The movement patterns are ranked according to their fitness value
2. Elitism is performed
3. Crossovers are performed
4. Mutations are applied

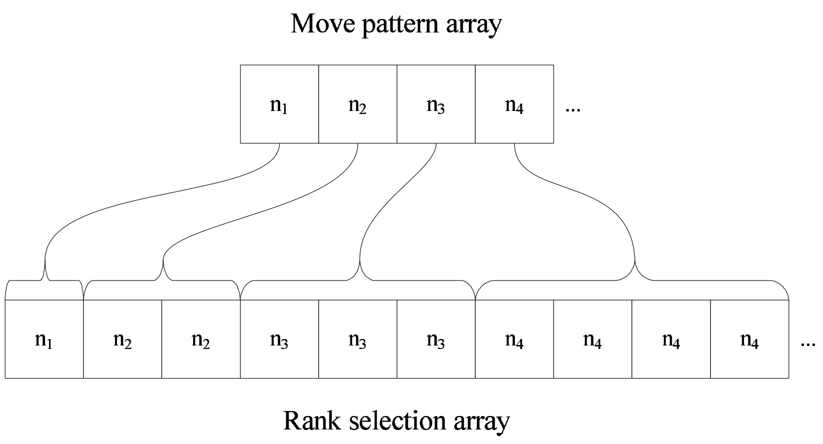


Figure 7.4: The rank selection array

5. The weights are stored to disk
6. The algorithm starts over again

The selection function. We chose to use rank selection. The selection function is used when performing crossovers in the algorithm, in order to decide what patterns should be crossed with each other. Every time a crossover is performed, two random numbers between zero and the number of used movement patterns are picked. These numbers are used as indexes in the rank selection array. The rank selection array is made the following way:

- The movement patterns are ranked in increasing order
- The size of the rank selection array is found using Equation 7.2, where s denotes the size and n is the number of individuals in the population
- The rank selection array is built, see Figure 7.4

$$s = \frac{n+1}{2} \cdot n \tag{7.2}$$

The fitness value. The collection of fitness values of the individual movement patterns are implemented as follows. When a movement pattern gets its turn it starts with zero as its associated fitness value. The following list shows what increases or reduces the fitness value.

- If the robot collides with another robot, the fitness value is reduced by one
- If the robot collides with a wall, the fitness value gets reduced by three
- If the robot gets hit by a bullet, the fitness value is reduced by one
- If a fired bullet leaves the battlefield without hitting an enemy robot, the fitness value gets reduced by one half
- If a fired bullet hits an enemy robot, the fitness is increased by three

The fitness threshold. Finding the correct fitness threshold should be part of performing tests, in order to see how well the algorithm performs. A fitness threshold of 30 seems like a reasonable value to start with. A value of 30 could mean a robot firing 10 times where all the 10 bullets hit an enemy robot.

Now that we have implemented the robot, we have to perform tests to see if the robot is able to adapt to enemies during a game. The following chapter will cover these tests.

8 Test

This chapter will describe the design and implementation of the testing of the machine learning systems used in our robot.

8.1 Target Selection

Target selection is the task of selecting which of the opponents to attack, based on different criteria. An example of a criteria could be distance, because it is most likely not a great idea to select targets that are 3,000 pixels away.

8.1.1 Design

A test of the target selection should show that the Bayesian network becomes better at selecting targets over time. To test this we could run a large number of rounds–no less than 20,000–and not change any of the game settings or enemy robots nor use any kind of machine learning in these rounds. The rounds should be played against a team that does not use machine learning techniques, such as `sampleteam.MyFirstTeam`, which is a simple team provided by the standard Robocode package. By using a standard team we are ensured that any improvement of our team will be apparent. Had we battled against a team that utilizes machine learning techniques, improvements of our team might not be visible due to the other team improving faster than ours.

The reason for the test having to be based on at least 20,000 rounds, is that the Bayesian network is only updated a few times in each round, and since there are 2,400 different configurations, we have to run a large number of rounds in order to see a clear result.

The test is successful if there is an improvement in the number of rounds won.

8.1.2 Implementation and Conclusion

Due to the limited amount of resources within this project, we have chosen not to implement this test.

8.2 Aiming System

The aiming system is a vital part of the implementation of our robot. Without it one would probably never win as much as a single game. The overall performance of a robot depends heavily on the ability to hit targets in a game, therefore a test of the aiming system is essential.

8.2.1 Design

Intuitively, a test of the aiming algorithm could easily be carried out by logging how often the robot hits a target compared to the number of bullets fired. The test must be carried out on a 2000×2000 pixel map with standard rules against the standard robot `sample.Crazy`, which is a robot that seems to move in random patterns, but actually follows a simple mathematical equation. One could argue that choosing this single target is not sufficient, but it will definitely show us if the network has the ability to learn how to hit a moving target.

The movement layer in the `OurRobot` class is modified, so that the distance to a target is approximately the same at all times. The recommended distance is 400 pixels but this can be changed if another distance characterizes the robot better. The reason for this modification of the movement layer is to ensure that it is actually the neural network that is learning, and not the genetic algorithm used to control the movement.

An initial test should measure the ability of our robot to learn. The test should run over a series of rounds and the hit accuracy of each round should be logged to see if there is an improvement.

Another series of tests should be performed on a number of configurations, hidden neurons, and learning rates, in order to find the best possible combination. These tests should be performed using the same initial weights, because random weights could lead to inconsistencies in the results of the tests. Furthermore, each configuration should be tested over a number of rounds in order to let the network adapt to the configuration in question. The number of rounds is chosen to be 300. The tests should conclude on which configuration is the most profitable.

8.2.2 Implementation

The first test is implemented using two integers–one that is incremented each time a bullet is fired (s), and one that is incremented each time a bullet hits (h). The result is the ratio $\frac{h}{s}$, which is calculated at the end of each round. The variables s and h are reset to zero at the beginning of each round. The result is plotted on a graph in Figure 8.1.

Movement is done by using a special movement module only used for testing purposes. This movement module ensures an average distance to the target of 400 pixels. The robot starts firing when a target is scanned, even if the distance to that target is 1200 pixels. But since this will probably happen in all rounds, it will not have an effect on the overall result.

The second test is implemented like the first one, where different configurations are tested. The average of all rounds is plotted on a graph in Figure 8.2.

8.2.3 Conclusion

Looking at Figure 8.1 it is clear that the neural network is able to learn to predict the movement of enemy robots. There is a substantial improvement, especially in the early learning phase.

The second test, whose results are plotted in the graph in Figure 8.2, shows that one should be careful when choosing a configuration. Depending on which environment the robot is operating in, different configurations will provide optimal performance. In general, many hidden neurons and small learning rates are slower converging than a small amount of hidden neurons and high learning rates are.

The test has been carried out using 300 rounds for each configuration. If the number of rounds was increased, the result could be significantly different.

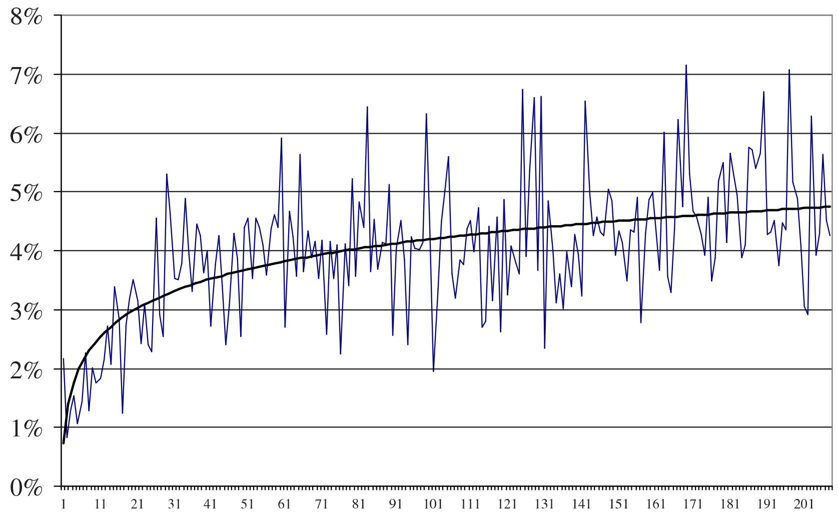


Figure 8.1: The hit accuracy from the test results and a trendline

Generally, configurations with high learning rates and a small hidden neuron count are faster to adapt to new environments, whereas configurations with a high hidden neuron count and small learning rates are more precise in a homogeneous environment.

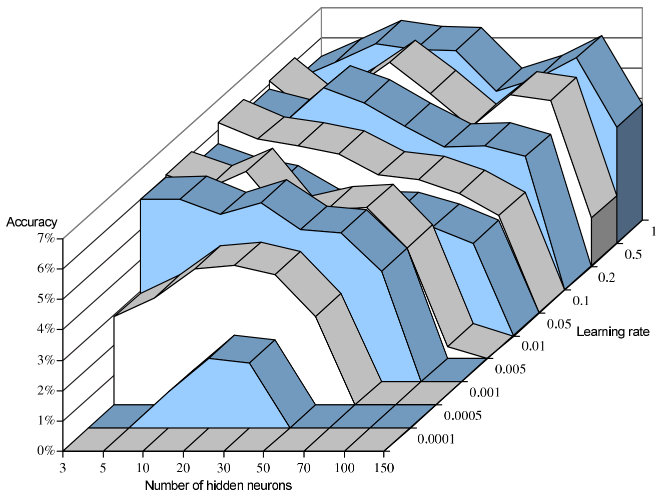


Figure 8.2: The hit accuracy from the test results using different configurations

In conclusion to the first test, it is obvious that our robot has the ability to learn, meaning that the neural network is working. Possible improvements could be not to fire over wide distances, since these shots have a low probability of hitting the target. It is also plausible that inputs to the neural network could be extended with further information, leading to higher accuracy. The second test shows that depending on how many rounds are played in a game, different configurations could be chosen. For a game consisting of 300 rounds, Figure 8.2 shows that the best configuration is a hidden neuron count of 20 and a learning rate of 0.5, or possibly a configuration with 3 hidden neurons and a learning rate of 0.1, which has approximately the same accuracy as the other configuration, but a lower neuron count performs better because of faster calculations.

8.3 Movement Pattern

The movement algorithm is essential for surviving in the Robocode scenery and can be the direct cause for a significant amount of energy loss, if e.g. two robots collide, a robot hits the wall, or a robot fails to steer free of a bullets path.

8.3.1 Design

In order to test the genetic algorithm a population must be bred for a large amount of generations. This is because evolution of the genetic algorithm takes a lot of generations to evolve into a useful solution.

We need to develop a test where the robot involved utilizes no machine learning, except for the genetic part of our implementation. To do this we create a version of the `OurRobot` class that features only the genetic algorithm, and not the Bayesian network nor the neural network. We do this to purify our test results, meaning that the two other machine learning parts of our implementation do not affect the data gathered by this test. Robocode is set up to use a battlefield of size 1000x1000 throughout the test. We also constrain the test to involve two robots only, where one is the `sample.Crazy` robot and the other is the modified version of `OurRobot`.

All of these precautions should ensure that we construct a static environment where only the inputs we want to test will be changed for each subtest. Below we list the inputs tested during the testing of the genetic algorithm.

Test Inputs

As described in Section 7.3, the algorithm takes a number of parameters which include:

- Population size
- Mutation rate
- Elitism/crossover rate

These parameters together with

- The life span of an individual and
- The number of generations

will make up the input parameters we would like to include in the test. We have decided to assign static values to the rest of the parameters mentioned in Section 7.3, so these will not affect the test in any way. Note that in the test we only list the elitism rate because the crossover rate is directly linked by this equation.

$$\text{crossover\_rate} = (1 - \text{elitism\_rate}).$$

The mutation method used is the single-point mutation. The crossover uses the uniform crossover method. For selection, the rank selection method was chosen. They are all described in Section 7.3.

Execution of The Test

Test execution will be divided into two main parts. The first part will consist of two subcategories, where we test the mutation rate and elitism/crossover rate. The second part will focus on both population size and long term evolution, where we increase the life span of the individual robot and observe the effect when increasing the time span of the evolution to consist of 200,000 consecutive games.

Output and Success

Each subtest will consist of a *setup* that explains what inputs were chosen, a *result* in the form of a graph of the collected data, and finally a *conclusion* that will comment on the result of the subtest. We have chosen to collect the sum of all fitness values for each generation as data output of the test. The fitness value of an individual, described in Section 7.3, represents how well the move layer performs for the given individual. By choosing the total fitness value of a generation as the output value, we are ensured that we get the general fitness improvement of a generation, whether the value is positive or negative.

A successful test will hopefully show us a graph that converges to some maximum.

8.3.2 Test Output

This section will describe the various test outputs.

Mutation Rate

The primary setup for testing the effect of changing the mutation rate is as follows:

- Population size: 25
- Mutation rate: *tested*
- Elitism rate: 0.1
- Lifespan: 20 rounds
- Generations: 25

We chose eight different values for the mutation rate ranging from 0.01 to 0.5. According to *Introduction to Genetic Algorithms*[11], a reasonable mutation rate would be 0.01, so we chose this as the initial value and then gradually increased the mutation rate up to 0.5 in order to observe the effect.

The graph in Figure 8.3 shows us that during all subtests of adjusting the mutation rate, we start out with a very low fitness value. Around the 6th to 9th generation the graph starts to become stagnant, which means the evolution rate starts to slow down. At the 13th generation we still observe big fluctuations, but for every new generation the fluctuations lessen. At the end we see that the set of graphs are pretty close together. Figure 8.4 shows a close-up of the 6th to 9th generation, and Figure 8.5 shows a close-up of the 13th to 20th generation.

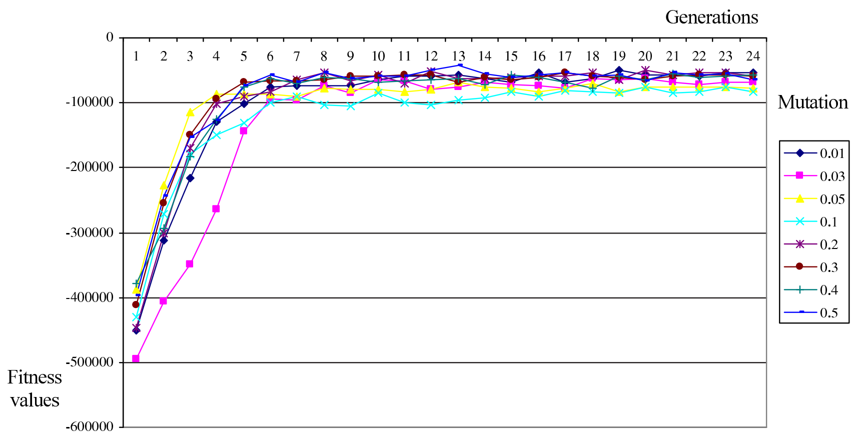


Figure 8.3: The test output for mutation rate

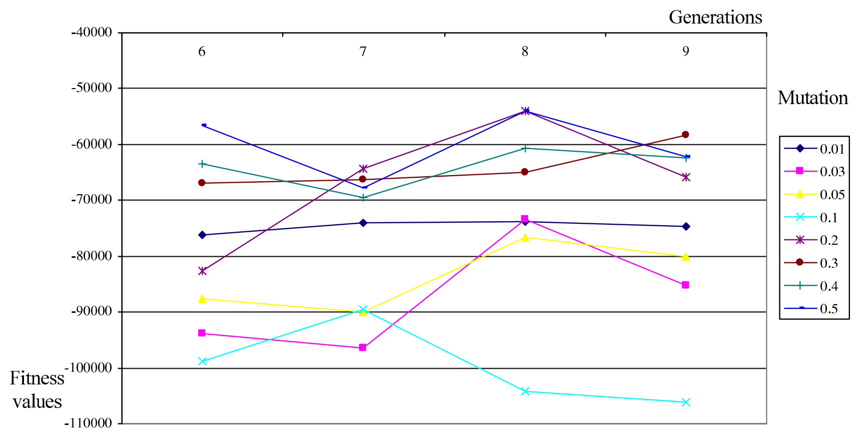


Figure 8.4: Close up of the 6th to 9th generation

Elitism Rate

The primary setup for testing the effect of changing the elitism rate is as follows:

- Population size: 25
- Mutation rate: 0.03
- Elitism rate: *tested*
- Lifespan: 20 rounds
- Generations: 25

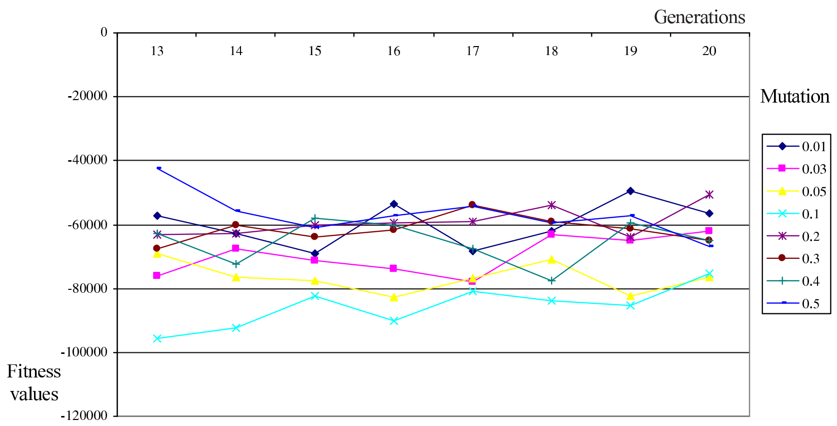


Figure 8.5: Close up of the 13th to 20th generation

We chose to use the exact same values for testing the elitism rate as we did for the mutation rate, so it ranges from 0.01 to 0.5.

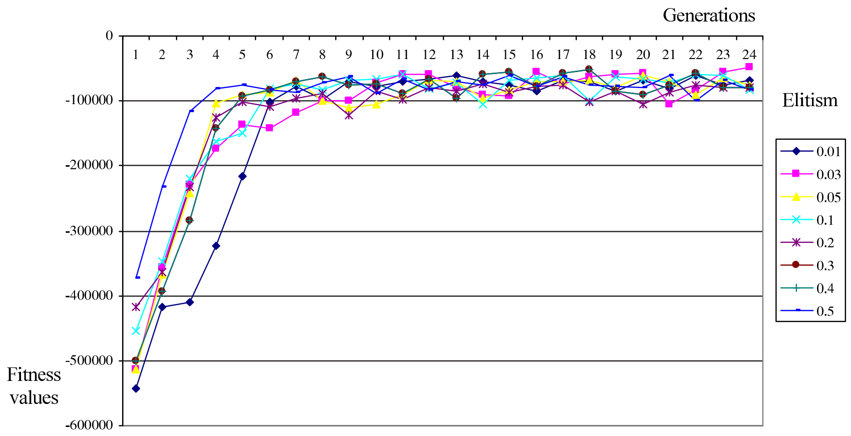


Figure 8.6: The test output for elitism rate

Figure 8.6 Shows the output when testing with respect to elitism rate. As we saw when testing with respect to mutation rate, the graph seems to start with low values and rising fast, and then stagnating around the 6th generation.

Long Term Testing

The primary setup for the long term test is as follows:

- Population size: 100

- Mutation rate: 0.1
- Elitism rate: 0.1
- Lifespan: 20 rounds
- Generations: 100

The long term test was decided to have a population size of 100 individuals, each having a lifespan of 20 rounds. This corresponds to 2,000 rounds per generation, and running for 100 generations would give a total of 200,000 rounds. The test took about twelve hours to complete.

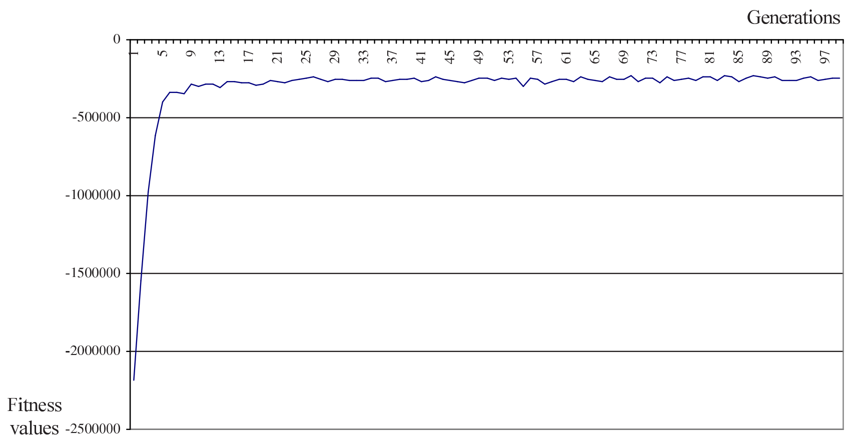


Figure 8.7: The output of the long term test

The graph in Figure 8.7 seems to be very similar to the other graphs we have seen so far, but looking a bit closer reveals that given a larger population our evolution rate seem to be slowing down around the 20th generation instead of the 6th to 9th generation. But otherwise the graph follows the same pattern and almost stagnates with little fluctuation.

8.3.3 Test Conclusion

We can see that in every test our fitness values starts off very low and rises quickly during the first generations, and then after a number of generations starts to converge to some maximum fitness value. We interpret this as proof that our robot learns to cope with the environment we have placed it in, moving more intelligently by avoiding walls, avoiding bullets, avoiding collisions with other robots, and placing itself in better positions for hitting targets.

We can see that the various tests with respect to mutation rate shows the same evolution with little variation. An explanation could be found in the implementation of the mutation function. In this function the mutation is done by changing a single cell in the array that holds the vector functions. Figure 8.8 shows a representation of the vector function array, in which we can see that the different cells have different meaning. For instance, changing the cell labelled x_1^3 would have a much higher impact on the movement pattern than changing the cell labelled w_1 would, and this holds true for the fitness value as well.

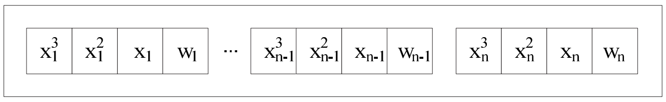


Figure 8.8: Representation of the problem with mutation

If the mutation is done on the lower three cells (x_i^2, x_i, and w_i) after each generation, the mutation would not have much effect on the fitness output. We assume this is the reason for the small variation in the tests concerning mutation rate.

After a certain point, around where the graph in Figure 8.6 starts to stagnate, the individuals on which crossovers are performed start to resemble the elite individuals. This is because of the huge evolution rate we have observed in the beginning of the test, causing us to quickly converge to where all individuals look very similar. At this point there is really no benefit in elitism, since the individuals moved directly to next generation are so similar to the ones on which crossovers have been performed.

For the tests concerning mutation and elitism, we see that when we hit a fitness value of around $-2{,}800$ for an individual, no matter how many additional generations are used, the fitness will just oscillate around this particular value. We assume that this is the optimal fitness value for an individual the fitness function can achieve with a population of 25. The long term test in Figure 8.7, which showed the same tendency as the mutation and elitism tests, reached an average fitness value of about $-2{,}400$ when the evolution rate stagnated. But as seen in the graph in Figure 8.7, a larger population means it takes longer to converge towards the value. So our conclusion is that given a larger population we will have a better result, but at the expense of time. Another conclusion is that the mutation and elitism rate seem to be of no consequence in a small population.

9 Conclusion

This chapter concludes this report by discussing what further development our robot would go through had we had more time to spend on the implementation, as well as summarizing what we have achieved during this project.

9.1 Further Development

Looking back at this project there are many things that could be improved on. First of all, because the robot developed was thought as a *proof of concept* implementation, there is still a need for further testing and adjustments of the variables in the various machine learning systems. For example, further tests and adjustment of the number of hidden neurons and learning rates of the neural network could be performed on additional battles against other enemies.

We are especially proud of the aiming algorithm that learns much faster than we had expected. This being said, we expect that the performance could be even better if the robot went through further development. Especially the input to the network could be extended in order to get even higher precision.

Other inputs for the neural network might be considered. An example is previous moves done by the enemy. Tests could be performed to see if such a modification would improvement on the hit accuracy.

As discussed in Chapter 7, the retreat layer as well as the Bayesian network for choosing an overall strategy has not been implemented. These are of course subjects for further development.

9.2 Conclusion

We have managed to implement three different machine learning technologies resulting in a team of autonomous robots for the Robocode environment. These three technologies include a neural network for the aiming system, a Bayesian network for the target selection, and a genetic algorithm for controlling movement of the robots. We have chosen to implement the robots using a hybrid agent architecture, meaning that they make use of the best of both reactive as well as deliberative systems.

On-line learning has been implemented in order for the various components of the robots to be able to adapt to the environment in which they operate and the enemies they face. We have used the test chapter to show that our robot has the ability to learn during a battle.

Team behavior has been implemented using decentralized coordination, which means that each of the robots must maintain a world model of their own, and therefore they do not depend on having to protect a leader.

As we have shown throughout the project, we have developed an understanding of using decision support system on a somewhat real-life-like application. We feel that we have gained enough knowledge or at least interest to make use of these technologies in projects to come as well as in other contexts.

Bibliography

[1] Daniela M. Bailer-Jones. Modelling data: Analogies in neural networks, simulated aannealing and genetic algorithms. `http://www.mpia-hd.mpg.de/homes/calj/mbr01.pdf`, 2004.

[2] David Barrett. Learning algorithms for self evolving robot controllers. `http://www.rdg.ac.uk/scarp/resources/siu00drb.pdf`, 2004.

[3] Derek Bridge. Deliberation. `http://www.cs.ucc.ie/~dgb/courses/ai/notes/notes13.pdf`, 2003.

[4] Jochen Froehlich. Neural networks with java. `http://rfhs8012.fh-regensburg.de/~saj39122/jfroehl/diplom/e-12-text.html#TypesOfNeuralNets`, 2004.

[5] Finn V. Jensen. *Bayesian Network and Decision Graphs*. Springer, 2001.

[6] Christian Krause. Robocode faq. `http://robocode.alphaworks.ibm.com/help/robocode.faq.txt`, 2004.

[7] P. Mehra and B. W. Wah. *Artificial Neural Networks, concept and theory*. Society Press, 1992.

[8] Tom M. Mitchell. *Machine Learning*. McGRAW-HILL INTERNATIONAL EDITIONS, 1997.

[9] Jörg P. Müller. *The Design of Intelligent Agents - A Layered approach*. Springer, 1996.

[10] Thomas D. Nielsen. Decision support systems and machine learning. `http://www.cs.auc.dk/~raistlin/Teaching/BSS04/`, 2004.

[11] Marek Obitko. Introduction to genetic algorithms. `http://cs.felk.cvut.cz/~xobitko/ga/`, 1998.

[12] Anet Potgieter. Agent architectures. `http://www.cs.uct.ac.za/courses/CS400W/AI/Resources/Agent%20Architectures.PDF`, 2004.

[13] David F. Cherno Thomas J. Loredo. Bayesian adaptive exploration. `http://astrosun.tn.cornell.edu/staff/loredo/bayes/bae.pdf`, 2004.

[14] J. M. Zurada. *Introduction to Artificial Neural System.* St. Paul, 1992.

All URLs are valid as of December 17th.