E S T I M A T O R

exuberant search to identify maximum adaptive determination of robots

a project by

E2-214

Marc Nyboe Kammersgaard
Jesper Kristensen
Arne Mejlholm
Jasper Melsvik
Rasmus Bang Nielsen
Ole Pedersen
Jacob Volstrup Pedersen

Department of Computer Science
Aalborg University

Title:
ESTIMATOR - Exuberant Search To Identify Maximum Adaptive Termination Of Robots

Semester:
Dat3
2. September - 17. December, 2004

Group:
E2-214, 2004

Members:
Marc Nyboe Kammersgaard
Jesper Kristensen
Arne Mejlholm
Jasper Melsvik
Rasmus Bang Nielsen
Ole Pedersen
Jacob Volstrup Pedersen

Supervisor:
Uffe Kjærulff

Copies: 9

Report – pages: 147

Total pages: 162

Synopsis:

This report describes the development of an adaptive expert system for the Robocode platform called ESTIMATOR. ESTIMATOR is built to use machine learning, namely the techniques Genetic Algorithms, Artificial Neural Networks, Reinforcement Learning plus Bayesian Networks and Decision Graphs.

The report first offers a detailed look at the Robocode platform to emphasize certain properties about the game. Then a general introduction to agent systems and architectures is offered in order to build our own architecture for ESTIMATOR.

The report then documents the developed architecture and how it utilizes the different machine learning techniques through a number of modules that handle certain responsibilities.

Last but not least, it evaluates the ESTIMATOR system and the different modules through a number of tests against non-adaptive robots. We found that all the agents work with the architecture and all modules (except one) learns through experience.

Preface

This report is written by the seven members of the DAT3 group E2-214 at Aalborg University during the fall semester of 2004. It takes the reader through the steps of analysis, design and evaluation made during the project period. The project course followed this semester is Decision Support Systems (DSS).

Along with the report is delivered a website which contains the source code of the developed robot ESTIMATOR. The Robocode framework installer, multiple versions of ESTIMATOR, which are ready to run, and this report as a PDF. The URL for this website is `http://www.cs.aau.dk/~mejlholm/dat3/index.html`.

Throughout the report a number of conventions are used: Whenever talking about implementional details such as modules, object etc. will be typeset using the `Typewriter` font.

Furthermore literature sources will be enclosed in brackets as follows, [Author abbreviation, eventual page]. An example: [Mit97, p. 2].

Marc Nyboe Kammersgaard

Jesper Kristensen

Arne Mejlholm

Jasper Melsvik

Rasmus Bang Nielsen

Ole Pedersen

Jacob Volstrup Pedersen

Contents

Part I

Analysis

Introduction

In this Part of the report we will outline the aspects about Robocode necessary to understand how Robocode works. The facts presented in Chapter 1 are based on [alpa], [alpd] and our own experiences with Robocode.

By knowing the details of Robocode we should be better equipped to build an expert system for the Robocode framework.

In Chapter 2 we will continue discussing how a Robocode robot can be split up into different areas of responsibility as a multi agent system. The reason for doing this is that Robocode is a large environment with many tasks to be performed. By delegating areas of responsibilities we should be able to reduce the complexity of the expert system.

To design an expert system it is necessary to understand what an agent is and to have some common terminology established. This will be presented in Chapter 3. Furthermore we will discuss means for agents to communicate in a multi agent system.

To have a sound basis for choosing or building an agent architecture, Chapter 4 will introduce different architectures and evaluate these in the settings of Robocode.

Finally in Chapter 5, we will summarize this Part, define the problem at hand and discuss approaches to build the expert system.

1 Physics of Robocode

In this Chapter we will give a short introduction to what Robocode is. This introduction will be a look at what the elements of the framework is and how they work. The reason for the analysis of Robocode is to form the basis for writing robots, that can be used in Robocode. We start with the Robocode concept.

1.1 What is Robocode?

The slogan of Robocode outlines the goal of the application:

> "Build the best. Destroy the rest. In Robocode, you'll program a robotic battle-tank in Java^{tm} for a fight to the finish."[alpa]

In Robocode you program small robots to battle against each other in order to determine which robot is the best. The framework has grown rather popular and as of the time of this report, there are a large number of robots available throughout the Internet. There are forums, where Robocode robot developers gather to exchange techniques for building robots. You can use these techniques to build a robot that you feel will be the best and then enter it in one of the tournaments hosted by the Robocode community.

Robocode is, however, not limited to battles where robots fight one on one, it to supports battles between teams. The objectives in these two battle modes are slightly different:

- Everybody vs. everybody where the objective is to be the last robot standing
- Teams vs. teams where the objective is to be the last team standing. Meaning that all the opposing team members are dead.

1.2 Anatomy of a Robocode Robot

A robot is visualized as a tank in the game and all robots have a fixed size of 40 times 40 pixels. It consists of 3 elements as pictured in Figure 1.1.

- A vehicle
- A gun
- A radar

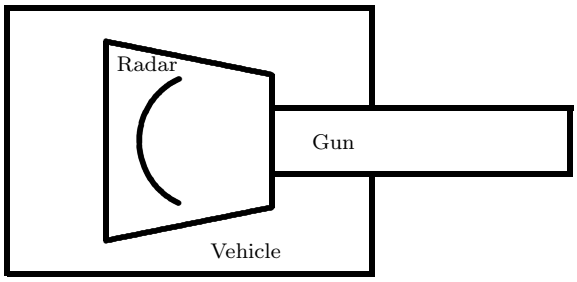


Figure 1.1: A Robocode robot

The three elements are all placed on top of each other, with the radar at the top, the gun in the middle and the vehicle at the bottom. This means that when rotating the vehicle, the two top elements rotate as well. This also applies to when rotating the gun, the radar rotate as well. It is, however, possible to turn this behaviour off on a robot basis.

1.2.1 The Vehicle

The vehicle part of the robot is what makes it capable of moving around the battlefield. When the robot moves around on the battlefield, it accelerates and decelerates, thus allowing for different but still limited velocities. The engine controls the acceleration and deceleration of the robots. The speed of rotation for a robot, depends on the velocity of the robot. All the values for the acceleration, speed and rotation can be found in Table 1.1.

1.2.2 The Gun

The gun can rotate to the right and to the left, and can fire in the direction in which it is pointing. Rotation of the gun is relative to the guns point of view and the rotation speed is limited to 20 degrees in one tick (see Section 1.3 for description of tick).

The gun can fire bullets with different kinds of firepower. The firepower of a bullet determines how much damage the bullet causes. When hitting a target the damage is subtracted from the energy (See Section 1.5 for description of energy) of the robot, which was hit. After a robot has shot, its gun has to cool down. The robot can only fire if the heat of the gun is 0. The physics of the gun and the bullets can be found in Table 1.1.

From the fired bullet it is possible to retrieve information about, which robot fired the bullet and the direction the bullet was heading.

The robot which fired the bullet can use the bullet object to check if the bullet is still active, which robot was hit by this bullet and the current position of the bullet on the battlefield.

1.2.3 The Radar

All robots, except those of type `Droid` (see Section 1.7 for description of `Droid`), is equipped with a radar, which can be rotated (left or right). The rotation speed of the radar is limited (see Table 1.1). The radar is used for detecting where other robots is on the battlefield. Because the radar can rotate 45 degrees per tick the robot can get information from 45 degrees of the battlefield per tick.

Vehicle	**Robocode value**
Accelerating rate	1 pixel per tick.
Deacceleration rate	2 pixels per tick.
Max velocity	8 pixels per tick.
Speed of rotation	$(10 - 0.75 \cdot abs(\text{velocity}))$ degrees per tick.
Freedom of rotation	360 degrees (no restriction).
Gun	**Robocode value**
Speed of rotation	20 degrees per tick + the rotation of the vehicle.
Freedom of rotation	360 degrees (no restriction).
Heating rate	1 + firepower/5 units.
Cooling rate	0.1 unit per tick.
Bullet	**Robocode value**
Velocity	$20 - 3 \cdot$ firepower.
Damage caused	$4 \cdot$ firepower (if firepower < 1).
Damage caused	$4 \cdot$ firepower $+ 2 \cdot$ (firepower - 1) (if firepower > 1).
Power returned on hit	$3 \cdot$ firepower.
Radar	**Robocode value**
Sight	1 degree.
Length	1200 pixels.
Speed of rotation	45 degrees per tick + the rotation of the vehicle.
Freedom of rotation	360 degrees (no restriction).
Collision	**Robocode value**
Robot hits robot	0.6 point damage to each.
Robot hits wall	$abs(\text{velocity}) \cdot 0.5 - 1$ (only for AdvancedRobots and never below 0.)

Table 1.1: Physics in Robocode

1.3 The Perception of Time

Time in a Robocode match is closely tied to the graphical frame rate of the game. The time is measured in "ticks", which corresponds to the frames displayed on the screen. Each robot on the battlefield gets one turn per tick [alpd].

This means that the time in Robocode is discrete. The length of a turn is roughly 15 millisecond, depending on the system running the game.

It is possible to skip a turn, but it is not recommendable, since the robot then will not be able to perform anything in the tick that is skipped, which could give the upper hand to the enemy. A robot will skip a turn if it processes for longer than the turn lasts. If a robot skips two turns in a row it will loose events (see Section 1.4.2 for description of events.), and if it skips 30 turns in a row it will be removed from the round and receive no score.

1.4 Battlefield Environment

This Section and the following Sections will describe the Robocode environment and the possibilities for gathering information within this environment.

The environment in Robocode is built up around the battlefield which is a rectangular space constrained by the notion of walls. The field can be of any given size, but during a battle the size is fixed measured in pixels. Robots cannot escape the field of battle during a battle unless they suffer death or are removed by the engine from the battlefield.

1.4.1 Movements on the Battlefield

When fighting in Robocode the robot has a wide range of possible movements it can perform during the battle. These movements are listed in Table 1.2. When a `move` command has been issued, the robot will execute it and move the amount of pixels specified in the command or until it hits a wall.

Direction	**Robocode command**
Forward	`ahead(double distance)`
Backward	`back(doudle distance)`
Turn vehicle left	`turnLeft(double degrees)`
Turn vehicle right	`turnRight(double degrees)`
Turn gun left	`turnGunLeft(double degrees)`
Turn gun right	`turnGunRight(double degrees)`
Turn radar left	`turnRadarLeft(double degrees)`
Turn radar right	`turnRadarRight(double degrees)`

Table 1.2: Movements in battle

When moving the radar or the gun this is done in degrees relative to the way the gun or radar is pointing when the commands are executed. For more details on the gun see Section 1.2.2 and for information about the radar see Section 1.2.3.

1.4.2 Intelligence Gathering

There are a wide range of different sets of information to collect from the battlefield in Robocode. The Robocode engine is an event based system e.g. when a robot is hit by another robot there will be raised a `HitRobotEvent`, which the robot can react on. In Table 1.3 the different events are listed to give an overview of the possibilities.

Event handle	Description of event
`BulletHitBulletEvent`	One of the fired bullets hit another bullet in the air.
`BulletHitEvent`	A fired bullet has hit a robot.
`BulletMissedEvent`	Bullet has missed the target.
`DeathEvent`	The robot dies.
`HitByBulletEvent`	The robot have been hit by a bullet.
`HitRobotEvent`	The robot collided with another robot.
`HitWallEvent`	The robot has collided with a wall.
`MessageEvent`	The robot received a message from a Teammate.
`RobotDeathEvent`	A robot dies.
`ScannedRobotEvent`	A robot is scanned with the radar.
`SkippedTrunEvent`	A turn is skipped.
`WinEvent`	The robot(s) wins a round.
`CustomEvent`	User specified condition is true.

Table 1.3: Events in Robocode

When an event occurs in the Robocode environment, an object of the current event is created. With this object the robot can get information about the event e.g. with `HitRobotEvent` information about bearing, energy of enemy, name of enemy etc, can be collected. The quantity of information a robot can collect from events can be seen in the Robocode API [alpb].

1.5 Energy Concept

When a round starts each robot has a specific amount of points, which can be viewed as its energy. When a robot fires the gun, energy from the robot decrease and a bullet is created with an amount of energy according to the decrease in energy of the robot. The energy will be returned times 3 if the bullet hits another robot. When a robot collide with another robot each robot looses energy. Colliding with a wall the robot is also punished. The calculation of the damage done when hitting a robot or a wall can be seen in Table 1.1. When the energy of a robot reaches below zero the robot will be destroyed, however if the robots energy is low and the robot fires the gun with such power, that the energy of the robot would reach zero or below, the robot will be disabled. In this state a robot only need to be hit by a single bullet or another robot to be destroyed. However even if a robot has been disabled it can still be enabled again if one or more of its earlier fired bullets hits a target (another robot), because hitting a target will be rewarded with an amount of energy.

1.6 Engine Processing Loop

The processing loop of Robocode specifies how the engine calculates tasks to perform. The order in which the Robocode processing loop runs is as described in [alpc].

1. All robots execute their code until calling a blocking action, `excute` or the

robots processing time is up.

2. Time is updated (currentTick + 1)
3. All bullets move and check for collision
4. All robots move
5. All robots perform scans (and collect team messages)
6. Draw the GUI for the battlefield

Every robot in Robocode is its own thread, which means that robots perform calculations in parallel until they call a blocking action, `execute` or their time is up. If a robot calls the action `ahead()`, it will stop doing calculations and wait for the processing loop to run trough item 2 to 6 and return to item 1, before doing further calculations.

1.7 Classes of Robots

The Robocode engine supports four variants of robots: `Robot`, `AdvancedRobot`, `TeamRobot`, and `Droid`.

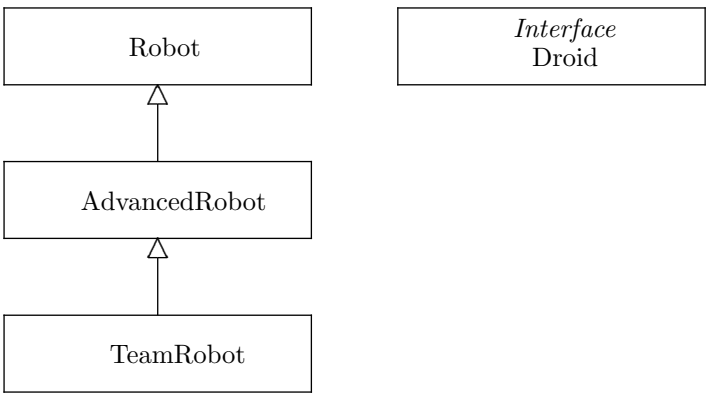


Figure 1.2: The relation between classes in Robocode

Robot: The standard `Robot` class, which starts with 100 points of energy. All actions performed by this robot are blocking (blocking and non-blocking actions are discussed in Section 1.8).

AdvancedRobot: Like the standard `Robot` class, but for each blocking action there is a corresponding non-blocking action.

TeamRobot: Is like AdvancedRobot, but extended with the capability to communicate with its teammates through message transmission. One of the team members will randomly be chosen as a leader by the game engine and it will be rewarded with 100 extra energy points.

Droid: A droid is a robot without a radar, and 20 extra points of energy. Its primarily intended to be used in team play.

An illustration of the classes and interfaces can be seen in Figure 1.2.

1.8 Blocking and Nonblocking Actions

The `Robot` class implements only blocking actions. Blocking means that the robot cannot do anything until a given action has been performed. Using blocking actions are good if you want your robot to behave like a program that is executed sequentially.

```
1 public void run() {
2   turnLeft(90);
3   turnGunRight(90);
4 }
```

Listing 1.1: Example of blocking in Robocode

Both of the actions in the Listing above are blocking actions. Meaning that `turnLeft(90)` will first be executed and after that `turnGunRight(90)` will be executed.

The AdvancedRobot class extends the Robot class and implements both blocking and nonblocking actions. Nonblocking actions do not block the robot, but instead puts the actions in a pending list that await simultaneous execution. With nonblocking actions it is possible to move, turn the gun and shoot at the same time, which is not the case with blocking actions.

To tell the engine to execute the actions in the pending list a call to `execute` will have to be made. This will execute any pending action or continue execution of any action that is in progress [alpb].

```
1 public void run() {
2   setTurnLeft(90);
3   setTurnGunRight(90);
4   execute();
5 }
```

Listing 1.2: Example of nonblocking in Robocode

The two first actions in the Listing above are both nonblocking, this means that first the `setTurnLeft(90)` will be placed on a pending list, next the `setTurnGunRight(90)` will be placed on the list. When the `execute()` action is called both of the elements on the list will be carried out simultaneous.

1.9 Robocode Score System

Robocode uses a score system as described in [alpc] to determine who was the overall winner. The score system will be described in this Section.

- **Survival Score:** Each robot that is still alive scores 50 points every time another robot dies.
- **Last Survivor Bonus:** The last robot alive scores 10 additional points for each robot that died before it.
- **Bullet Damage:** Robots score 1 point for each point of damage they do to enemies.
- **Bullet Damage Bonus:** When a robot kills an enemy, it scores an additional 20% of all the damage it did to that enemy that round.
- **Ram Damage:** Robots score 2 points for each point of damage they cause by ramming enemies.
- **Ram Damage Bonus:** When a robot kills an enemy by ramming, it scores an additional 30% of all the damage it did to that enemy.

The total score for a robot is all scores added up for the whole match. A Robot wins if its total score is the highest, when all rounds has been played in a match. A team will win if the total score for all teammates added up is the highest of all teams on the battlefield.

1.10 Agent Communication in Robocode

In this Section we will describe the remedies that allow agents to communicate. All details can be found in the Robocode API [alpb].

The agents use simple message passing to communicate as known from distributed systems. For a robot to get these properties they have to inherit from the `TeamRobot` class. A `TeamRobot` may send messages to teammates and likewise receive messages. However robots, which have implemented the interface `Droid`, do not have the ability to send messages to teammates, they may only receive. A message may either be sent to a specific teammate or broadcasted to all teammates. In Robocode message passing is obtained by sending serializable objects, which can contain any kind of objects in Java. To send a message to all teammates the method `broadcastMessage(Serializable message)` will have to be used.

If a robot wishes to send informations to a specific teammate on the battlefield, it can use the method `sendMessage(String name, Serializable message)`. The argument `String name` is the name of the specific robot, which should receive the message.

To receive a message from teammates on the battlefield, the robots should implement the `onMessageReceived` method, which will be called every time a message is received. From the `MessageEvent` object it is possible to get the name of the sender and the message which have been transmitted.

What should be transmitted is all up to the individual robots, however the teammates should know how to handle the received objects. Informations, which a robot could want to transmit, could be some stimuli the robot has received, or information the robot has about the environment. A leader robot could also use these transmissions to tell other robots how to behave.

Summary

In this Chapter we have introduced the Robocode platform. This introduction included a view at what a Robocode robot is and what actions it may perform. We also looked at the battlefield, the concept of energy and how different classes of robots have differnet sets of actions. Furthermore we looked at how time units in Robocode are percieved and how the engine processes actions and events. Finally we considered how robots may communicate in Robocode.

All in all we saw that Robocode is a large and complex environment, where you can play around with almost anything you desire.

2 Division of Robots

The robots in Robocode can be divided into several modules, each having their own areas of responsibility. Each of these modules can be implemented by different techniques and should be independent in the way they act, but still be able to cooperate with the other modules. In the following sections we will try to point out these tasks.

2.1 Vehicle Module

One part of the task "controlling a Robocode robot" is to control the vehicle on the battlefield. When the vehicle moves (ahead, backward, turn, stop, accelerate and decelerate) it moves in a movement pattern e.g. linear, curved, circular, and oscillation movement. The vehicle can be controlled with different purposes in mind and one could be to stay alive by moving in a pattern, which makes the robot hard to hit. By moving the robot in unpredictable movement patterns it becomes harder to hit (e.g. by moving in continuous changing directions and speeds or by dodging the bullets when the robot is shot at).

Another purpose could be to score points by ramming into the enemies, or to avoid being rammed (which will result in energy loss and points to the enemy). A goal could also be to avoid hitting the walls as hitting the walls results in energy loss.

The `Vehicle` module will need to use information gathered by the `Radar` module, which scans the other robots positions and energy. By knowing the positions of other robots and their amount of energy we can decide whether to avoid or ram them. When the energy for a robot on the battlefield changes, which is observable through the radar, this can be used to guess what the other robot is doing e.g. the energy will drop when a robot fires a bullet. If we know when another robot fires a bullet we can use this information to perhaps avoid the bullet.

A module for moving can also use information about the other robots movement pattern when trying to avoid or ram them.

2.2 Gun Module

To win a battle it is useful to master the shooting part of a robot. The module has two responsibilities:

- **Predicting movement:** The robot must be able to predict where an enemy will move based on the enemy's previous movements. To accomplish this it should be able to track the movement of the enemy and use this information to predict the location of an enemy at a given time.

- **Deciding wether to shoot:** The robot will be able to decide whether it wants to shoot at the enemy or not. We remember that shooting costs units of energy and it is only returned if the bullet hits a target. So when the robot is not certain that it will hit, it should not put a high amount of energy into the shot or perhaps not shoot at all. Shooting at an enemy far away reduces the chance of hitting where as "close combat" shooting makes the chance of hitting the target higher. Therefore a decision based on different variables should be reached e.g. how far away is the enemy, does it move, does the robot itself move. By these different variables the robot should have a reasonable chance to decide to shoot or not.

2.3 Radar Module

The robots radar needs to be tuned to provide the rest of the robots modules with information about the environment. There are different radar turning tactics which best suite different types of robots, which are:

- **Simple:** Rotate the radar 360^o as fast as possible.

- **Sweep:** Only sweep the part of the battlefield, where it is needed e.g. if the robot is in a corner then it only needs to sweep in one direction or range of the battlefield.

- **Communication sweep:** The robot gets radar information from other robots. This information can be used to decide where to sweep. A team of robots can then in cooperation cover a wide range of the battlefield by scanning different areas of the battlefield and transmit the information to other teammates through the `Communication` module.

- **Tracking:** Keeps tracking an enemy robot by locking the radar to it. This technique can be efficient in combination with communication sweep to spy on an enemy.

2.4 Communication Module

When several robots work together as a team they need to have the ability to communicate or else they lack the ability to share information between them.

Because the robots should work together as a team, it is obvious that each of the robots should be able to communicate with each of the other robots on the team.

In this way the robots can carry out a given task with help from the teammates, or try to help other teammates with their tasks or goals, e.g. by sending information scanned by the `Radar` to the other robots and retrieve information from the other robots.

Summary

In this Chapter we discussed how to split the robot into four different areas of responsibility. These were: `Vehicle`, `Gun`, `Radar` and `Communication`. Splitting them up makes sense because you get a clear defintion of responsibilities and independence. The differnet part are, however, still dependent on each others performance in order to perform well, as the `Vehicle` and `Gun` modules cannot do much without a `Radar` module that finds the enemy.

3 Agent Concept

The purpose of this Chapter is to briefly introduce terms used to describe agents. We do this partly to introduce the reader to agents and partly to give our definitions of the terms, as there is virtually no consensus in the litterature. Furthermore we will discuss means for agents to communicate in multi agent systems.

3.1 Definition of the Agent Concept

There are many attemps to define an agent in the litterature. One definition is:

Definition 1. *We consider an agent to be an entitiy, such as a robot, with goals, actions, and domain knowledge, situated in an environment. The way it acts is called its "behaviour" [PM00, p. 2].*

Another is:

Definition 2. *An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives [Woo02, p. 15].*

From Definition 1 and 2, we see that an agent is some computing entity, which tries to achieve its goals given an environment. The latter may be the physical world or a simulated environment. Further more an agent system consists of a number of agents as defined by Definition 3.

Definition 3. *An agent system is a configuration of an environment and agents placed in the environment.*

In Robocode, however, we define a robot as an agent system or a set of agents, each one keeping track of some sensory input.

Furthermore a team is a collection of robots, where each robot has a set of actions, such as shooting and moving. The robot is informed about dynamic changes in the environment through the events discussed in Section 1.4.2 on page 14. By a dynamic environment we mean an environment, where conditions changes and the agents have the ability to change it.

3.2 Agent Systems

There are several ways to classify an environment. How accessible is it? Meaning can the agent obtain complete, accurate, up-to-date information about the environment's state? Is the environment deterministic or non-deterministic? Meaning if the agent performs an action, there will be uncertainty about what the resulting effect will be. Environments can be static or dynamic. In a static environment all changes of the environment is under the control of the agent. In dynamic environments changes is beyond the control of the agent. Finally an environment can be discrete or continuous. A discrete environment is one where there are a fixed, finite number of actions and perceptions.

The Robocode environment is inaccessible since it is not possible for a robot to obtain a complete, accurate, up-to-date information about the environment. Primarily due to the limits of the basic robot sensors, but also the fact that other robots make up part of the environment. Since there are other robots involved, which can do whatever they like within the boundaries of the game, it also makes the environment non-deterministic from the view of a single robot. The environment is also dynamic since changes in the environment can be done by other robots and therefore control is out of the hands of a single robot. Finally the environment is discrete (its a simulation on a computer), but the amount of states of the environment are so large that it might as well have been continuous.

3.2.1 Homogeneous and Heterogeneous Systems

In a homogeneous system, all agents can be perceived as being instances of the same agent. Agents will have the same sensors to obtain percepts, their difference in behavior (if any), will be the result of the difference in what they observe in the environment.

A heterogeneous system, is a system that consists of an environment with multiple different agents. Here agents are not replica of the same agent, and therefore agents could be capable of perceiving different inputs, acting out different actions, trying to achieve different (perhaps even conflicting) objectives.

Robocode is a heterogeneous system, since it is possible to utilize robots with different sensors. E.g. `Droid` do not have a radar, and the other classes of robots do. Robocode can also be a homogeneous system if a robot consisting a one agent is competing against itself.

3.2.2 Multi Agent Systems

A multi agent system is, as the name implies, a system where multiple agents operate at the same time.

An obvious use of multi agent systems is to solve a problem by dividing it into numerous subproblems and distributing the subproblems to different agents. This also has the advantage that the subproblems can be solved in parallel. These capabilities makes it easy to create the system by modules, which can easily be extended by injecting new modules, where each module is represented by an agent.

One might ask how multi agent systems are different from distributed or concurrent systems? One important aspect is the fact, that agents are autonomous and capable of making decisions on their own, and they are primarily concerned with their own interests. This means that the synchronization and coordination structures in multi agent systems are not hardwired into the system and agents do not share a common goal, as is typical for components in distributed or concurrent systems [Woo02].

3.3 Agent Cooperation

In order for agents to cooperate they need to have some protocol of communicating. Humans communicate through speech and physical gestures, such as waving the hand. In Robocode the communication between robots are limited by the engine to messages passing and events.

When looking at agent cooperation it automatically leads to looking at agent communication and information sharing. When two agents have to communicate with each other they have to know what information they can send and receive to be able to understand one another.

3.3.1 Commitment/De-commitment Protocol

With the commitment/de-commitment protocol, agents may make commitments to each other for a finite amount of time. This commitment involves some kind of agreement e.g. pursuing the same goal. This approach may optimize agent systems, because for a given time slot, a certain agent may disregards certain responsibilities. For this system to work, agents have to *trust* one another to make commitments [PM00].

3.3.2 Blackboard Communication

When looking at different communication protocols for agents, one of the first that comes to mind is the Blackboard protocol. The idea here is to use a blackboard, where the different agents can read and write information to and from. There are some complications with a blackboard e.g. when should its information be wiped from the board due to the fact that a board have a finite amount of space, mutual exclusion, etc.

3.4 Cooperation Protocols in Robocode

Now that we have described some of the different communication protocols and cooperation techniques, it is time to analyse these in relation to what the Robocode engine offers of possibilities for cooperation.

As we see it, communication in Robocode must be made by broadcasting messages to all members on a team or by sending individual messages to each other. Message passing is done through objects, to save the overhead of parsing strings, character by character.

If commitment/de-commitment should be implemented in Robocode, a larger scheme must be designed in order to make the robots able to trust each other. One example might be the case where one robot is scanning in a given direction and it wishes to have another robot scanned in the opposite direction. This design may very well prove too large a task to be fruitful enough.

A Blackboard protocol to store information about the enemies could easily be implemented in Robocode, although there are some issue about mutual exclusion and when to mark information as outdated.

3.4.1 Information Content

Another important subject when looking at communication in Robocode is what information to communicate to each other; known as information content. The information that can be collected in the Robocode environment is mostly gathered through events, as described in Section 1.4.2 on page 14.

The information content to be transmitted will be based on the current state of the environment, so the information content question is not simple. One solution could be to just transmit the current received object event to the other teammates and they could then extract the need information. There is in fact a large communication overhead, because a lot of unnecessary data will be exchanged between the robots.

Another solution could be to classify the different information and only send the absolutely necessary information between the team members. The communication module could take care of the classification in relation to the given state of the robot.

An example is when a `ScannedRobotEvent` occurs; this event has got 9 different methods to access different information regarding the scanned robot[1]. If only a limited amount of the available information is needed, these could be selected and transmitted, to reduce the amount (and overhead) of transmitting information. Such a limitation could be to only have interest in the information received from the following methods:

- **Enemy Name:** `getName()`
- **Position:** Can be calculated from `getBearing()` and `getDistance()`.
- **Heading:** `getHeading()`
- **Energy:** `getEnergy()`
- **Velocity:** `getVelocity()`

The classification of information has reduced the number of information from nine to five and thereby removed some of the transmission overhead. But the real compression could be reached, if the communication module during battle determines which information to transmit.

[1] Actually the event has got 19 methods, but 10 of these are deprecated methods, only delivering equal information as the non-deprecated methods, by internally calling those non-deprecated methods, making the deprecated methods superflous

Summary

In this Chapter we defined what an agent and agent system is. We discussed terms for classifying an enviroment as static or dynamic, deterministic or nondeterministic and discrete or continuous. We also had a brief look at multi agent systems and ways to cooperate and communicate within these.

4 Agent Architectures

In this Chapter we will describe some of the different existing agent architectures. This should help us choose an architecture or aid us in designing our own later on in the design phase. Note that this survey will by no means be exhaustive as a complete survey of agent architectures is beyond the scope of this report. This Chapter serves only as a means of inspiration for the design phase. The description of the architectures in this Chapter are based on [Woo02].

4.1 Reactive Agents

Reactive agents have no internal states and on input the agent simply chooses a pre-set action given a certain configuration of inputs. The advantage of using reactive agents is the low cost in computational power. The development of reactive agents can be quite cumbersome, as the developers have to make sure that the agent can handle every possible situation. Subsumption as a reactive agent architecture (discussed below) proposed by Brooks in 1985 is a quite good scheme for making reactive agents, which exhibits intelligent behavior without maintaining a map of the environment.

4.1.1 Subsumption

The main idea behind subsumption is that you can create an agent that makes sound choices without Artificial Intelligence (AI). This is achieved by designing a number of decision making modules arranged in a hierarchy, thus the name subsumption.

We now elaborate on this architecture by an example. We wish to create a Robocode agent which has five responsibilities: Targeting, Communication, Driving, Obstacle Avoidance and Ramming. We organize these in the hierarchy shown in Figure 4.1.

The principle is: The bottom layer, Ramming, evaluates it is stimuli and makes a decision, either to perform a specific action or not. It then passes control, following the arrows, onto the layer on top of it, layer number 2, which does exactly the same.

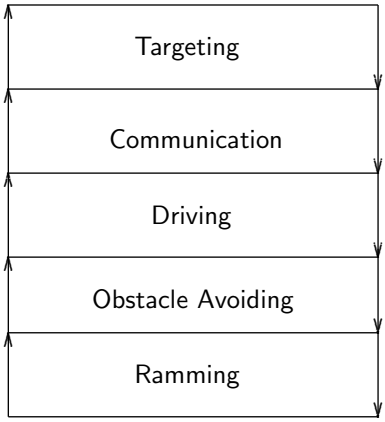


Figure 4.1: Example of a reactive agent built on the Subsumption architecture.

This way the control passes all the way to the top, layer number n, making all layers propose an action. Control then passes down again to the layer beneath the top one, $n-1$, with a message containing the action the layer chose. This layer may then choose to pass the message from layer number n along or pass its own choice of action. Arranged like this, it is up to the lowest layer to make the final decision about what to do, while it keeps the advice of the other layers in mind.

So in a situation where the agent showed in Figure 4.1 is close to an enemy, it would prioritize ramming the enemy over avoidance.

4.2 Deliberative Agents

Deliberative agents are knowledge based, meaning that they keep information about their environment as internal states. They use this knowledge to reason about what is the best action given a specific state. The result of this is a plan. An agent may have a number of plans for a given state and between these it chooses one, called the intention.

One commonly adapted strategy is for a deliberative agent to model other agents beliefs, trying to predict their actions. This can, however, lead to problems like: when do the agents stop modeling other agents? (example from [PM00] *what agent A thinks agent B thinks agent A thinks...*) This technic is mostly relevant when regarding homogeneous agents, because for a given agent to model another agent of different type is often quite hard to do unless it knows every detail about the other agent.

4.2.1 BDI: Belief, Desire and Intention or PRS: Procedural Reasoning System

In this architecture the agent does not reason about plans, it comes with a number of plans specified by the programmer. A plan has the following components:

- Goal: postcondition, meaning the environment after the plan has been carried out.

- Context: precondition, the environment, represented in its beliefs before the plan is carried out.

- Body: a sequence of actions to carry out.

This may seem straightforward at first, but in the body there may be listed other plans, making it a recursive data-structure. This may be used to model, that the agent must carry out certain subgoals in order to finish a goal. In Robocode, this situation might be the fact that a robot, in order to achieve the goal of killing an enemy, it must achieve the goal of getting within a certain shooting distance of the enemy.

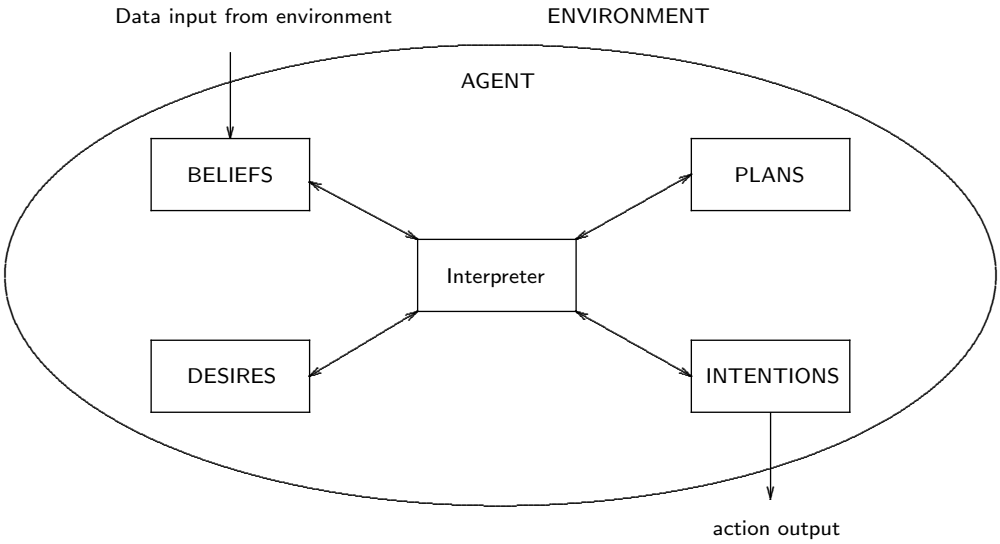


Figure 4.2: The BDI or PRS architecture.

The engine/interpreter, shown in Figure 4.2, of the agent has an intention stack upon which goals, which are about to be carried out are placed. The agent runs through its plan library, checking which plans have the same postcondition as the one on top of the stack. Some of these do not have the necessary preconditions and are rejected. The rest become the agents options, from which the agent chooses one; the new intention.

4.3 The Limits of Reactive and Deliberative Agents

The application of these architectures does, however, come at a prize. While the reactive architecture does faster computations than the deliberate architecture, it does not need a model of the environment, it is very hard to imagine a scheme where the reactive agent can improve over time. There are no means for it to learn and it does not employ global tasks or goals.

The deliberative agent architecture opens more possibilities, but in fact these are often overwhelming and it is hard to construct an efficient world model. There should be no superfluous information and there should not be too many data sets to consider, so that the agents performance is degraded. The data sets must still

have enough information, so that the model is accurate enough for the agent to be effective. Furthermore there is incidents where it is just better to react immediately.

This has lead to the idea that hybrid agents may be constructed to utilize the best features of each architecture.

4.4 Hybrid Agents

As we saw in the previous Section, purely reactive and deliberative agents have their limitations. Additionally, you might encounter tasks where it would be profitable to combine the two. E.g. maneuvering the robot in Robocode might be split into two parts: one that calculates the best route to a specific location on the map and another that avoids obstacles, teammates and enemies. As the agent is divided into two parts, they may run in parallel thereby making the robot able to react while calculating goals.

An attempt to make a hybrid architecture of layers might be as shown in Figure 4.3. Each layer takes perceptual input and produces an action output. The problem with this approach is that a function for selecting amongst these suggested actions is needed. This function must know each possible configuration which amounts to m^n possibilities, where m is the number of possible actions produced by one layer and n is the number of layers. This is unwanted complexity, so either one must come up with a scheme that limits this amount of possibilities or use the advantages learned from the Subsumption architecture.

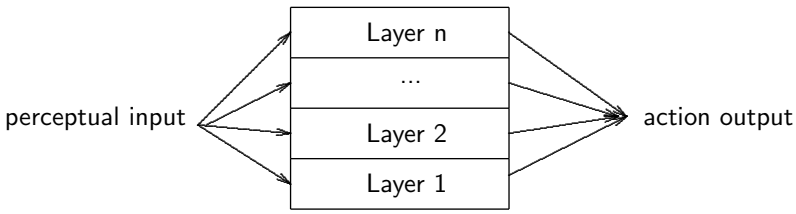


Figure 4.3: Agent with horizontal layering

Because of this we will have a look at two other architectures, the hybrid architectures TouringMachines and InterRRaP, which uses horizontal and vertical layering respectively.

4.4.1 TouringMachines

This architecture is, as shown in Figure 4.4, primarily built from three suggestion layers who produce suggestions, for what actions the agent should perform. The reactive layer is as we know it from the Subsumption architecture, and it produces a number of reactive actions from the given situation. The planning layer produces plans as we know them from PRS/BDI. The modeling layer handles the belief part of the agent, replicating the world inside the agent. It also predicts conflicts between the other two layers. The layer in the bottom of the figure, the Control Subsystem, is responsible for determining and exercising which layer should have control of the agent. It may suppress certain signals and inputs between the layers. The perception

subsystem is responsible for turning the sensor input into something the rest of the system is able to understand and the action subsystem is responsible for turning produced output into actions.

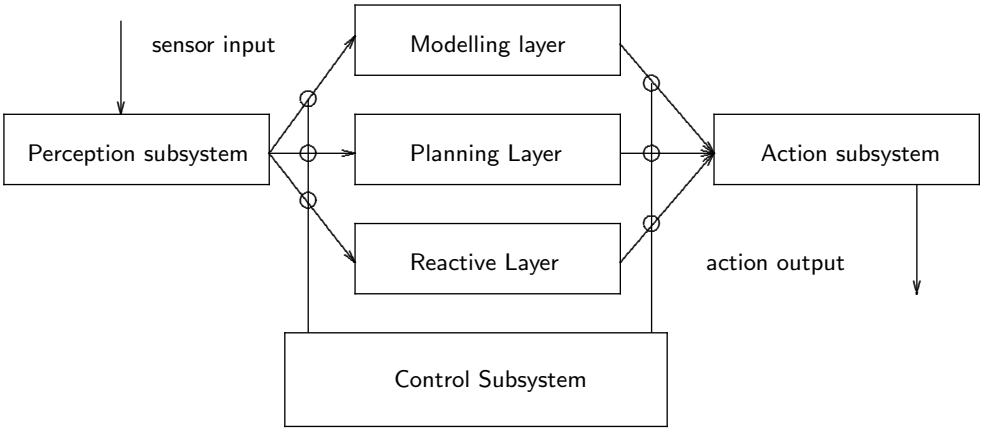


Figure 4.4: TouringMachines architecture

4.4.2 InteRRaP

As can be seen from Figure 4.5 this two pass[1] agent architecture is split into three layers: behavior, plan and cooperation each having their own knowledge base.

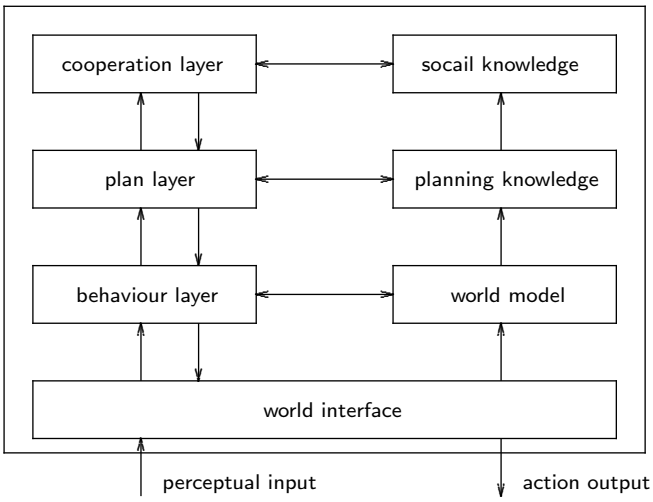


Figure 4.5: The InteRRaP architecture.

The first is a reactive layer that handles reactive behavior, the second layer handles planning, e.g. maintaining the intention stack we mentioned in the BDI architecture, and the last layer handles social[2] planning.

The way the layers cooperate is non-trivial. First the behavioral layer receives sensorial input and if it can choose a reactive action based on these input, the

[1] Two pass, meaning that control flow passes through each layer two times.
[2] Regarding cooperation between agents, agreement etc.

agents performs the given action. If it is not competent to choose an action it passes control to the next layer which exhibits the same behavior. This is called Buttom-up activation and the opposite Top-down activation is sometimes also utilized in this architecture.

Summary

In this Chapter we were familiarized with reactive, deliberative and hybrid agents. We weighed reactive agents against deliberative agents and concluded that hybrid agents in general are the best solutions, as they can use the best of both types of agents. We then saw the problems connected with using horizontal layering and noted that this will be a problem with the TouringMachines architecture, if not solved by other means. InteRRaP on the other hand was able to use the advantages of the Subsumption architecture.

5 Conclusion and Problem Definition

In this Chapter we will discuss approaches to design robots and explain why machine learning can be applied. This leads up to our problem definition, which states what we wish to accomplish during this project period. After this definition we will delimit the problem by stating what we do not wish to accomplish.

5.1 Results So Far

In Chapter 1 we started out by having an in-depth look at what Robocode is and the possibilities it offers. What we found was that it is a large environment with many different variables and inputs as well as actions to perform.

In Chapter 2 we then considered the choice of splitting up the robot in different areas of responsibility. We saw that the responsibilities in the robot at least could be split into four different areas: `Vehicle`, `Gun`, `Radar` and `Communication`.

These responsibilities may be implemented by a number of agents as introduced in Chapter 3, where we also considered what agent systems are and how agents may communicate within multi agent systems.

If we are to divide the robot into different agents with their own responsibilities a well defined architecture is needed. Therefore we introduced a number of different architectures in Chapter 4 and concluded that a hybrid architecture is the best solution, if the robot must be able to both do reactions and reasoning.

Based on this we will now be able to discuss different approaches to building a Robocode robot upon.

5.2 Expert System: Hardcoded vs. Machine Learning

Loosely phrased, our goal for this project is to design and implement a Robocode robot, which performs well against other Robocode robots. As this robot should be able to make decisions about what actions to take, it can be regarded as an expert system, as follows from Definition 4.

Definition 4. *An expert system is a computer program that simulates the judgement and behavior of a human or an organization that has expert knowledge and experience in a particular field [Tec].*

The development of such an expert system can in general be quite cumbersome. If we could get a hold of an expert within the Robocode field who knows the optimal reaction to every input at all times, it could perhaps be a task which could be overcome, but we are not so lucky to have an expert hence we have to consider another approach.

Then we consider the various Internet pages dedicated to Robocode, where many Robocode robot developers have ideas about how to build the best robot. To determine which of these approaches or even which combination of approaches is the optimal, is however a long and tedious task, because we have to code the approaches ourselves and compare them with each other.

Another approach could be to have the system *learn* the optimal behavior. We define learning as follows:

Definition 5. *A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E [Mit97, p. 2].*

In respect to a Robocode robot, the class of tasks T could be viewed as the task of getting a high score after a match. Performance may be measured in many ways, but one measure that always holds true is the score after each match.

The task of learning may be divided further into two classes: *supervised* and *unsupervised* learning, which we define as follows:

Definition 6. *Supervised learning is a learning process in which changes in the system are due to the intervention of any external teacher. The teacher typically provides output targets.*

Definition 7. *Unsupervised learning is a learning process in which changes in the system are not due to the intervention of any external teacher. Instead the system parameters are changed using only information from inputs and are constrained by prespecified internal rules.*

To use supervised learning we would still have to know the optimal action in every situation, as we should tell or show the robot what to do each time. At some point the robot has learned enough from us to know what to do in a given situation. Thus this task is quite hard to accomplish, as we do not know the optimal action at all times.

With this in mind we have a look at unsupervised learning, where we construct a set of rules that reward the robot when doing good actions. This way the robot plays a high number of matches and when it performs well, it is rewarded. A good score from a match should also give a good reward.

We conclude that unsupervised learning is the best approach for us when making a Robocode robot. The approach itself does however not guarantee a good result, as there are a number of pitfalls. The actual learning modules of the robot must be well designed and precisely defined. There is not necessarily any guarantee that a

module will converge to the optimal solution, as the performance of some approaches to unsupervised machine learning tend to converge to some local maximum instead of the global maximum. It should also be noted that, as stated before, the Robocode environment is large and holds much information. In order to make the robot effective we also need to reduce the amount of information, which the robot must process.

A small twist to our task is that we must build a team of robots, not just one. This opens up a world of opportunities to explore and take advantage of machine learning between multiple robots. If one robot has to bear in mind what the other robots are doing, it adds rather much complexity to the environment and the robot, so we have decided to disregard the option of exploring machine learning between multiple robots. The social activities of the robots on the team will be limited to not obstruct each other and to share their information about the environment.

5.3 Problem Definition

All these considerations lead us to the actual phrasing of the problem at hand:

> The goal of this project is to build an expert system to control a robot for the Robocode platform. It shall use a number of supervised and unsupervised machine learning techniques to achieve its goals. The performance of this robot must improve over time and it shall be able to adapt to changes in the environment. The robot must be split into different modules, which handle delegated responsibilities. The tree main responsibilities are the vehicle, the gun and the radar. Furthermore the architecture of the robots must support replacing an agent using a certain type of machine learning with another module using another type of machine learning. The robot should perform as well on a team as alone when acting.

5.4 Delimitation of the Problem

As stated above we will use a number of supervised and unsupervised machine learning techniques to build the robot. As said earlier supervised learning is harder to do, because we have to gather data and classify it for the robot to be able to learn from it. As a result we have disregarded this type of learning, except from one case where supervised learning actually turns out to be quite easy to use. This is when trying to estimate where the opponent will be in a number of time units.

The machine learning techniques used in this project will be: Artificial Neural Networks, Reinforcement Learning, Genetic Algorithms and Bayesian Networks and Decision Graphs, which will all be introduced in Chapter 8.

Using these techniques we will design and implement a number of modules for the robot. These modules will handle certain responsibilities, divided into the three groups: `Vehicle`, `Gun` and `Radar`.

These three groups will be reflected in the architecture, each one being a deliberative module. Constructed this way, we will be able to exchange e.g. a vehicle module using Reinforcement Learning with a vehicle module using Genetic Algorithms.

This will all be described in the next part.

Part II

Design

Introduction

This Part of the report documents the actual design of the ESTIMATOR robot.

In Chapter 6 we will follow up on the problem definition from Chapter 5 and rate how we will prioritize our time during the design.

We will introduce our robot architecture in Chapter 7 in relation to Chapter 4 where we introduced the chosen architecture for the robot.

In Chapter 8 we will then introduce the machine learning techniques that we have chosen to design the robots modules with. The chosen techniques are Bayesian Network and Decision Graphs, Reinforcement Learning, Artificial Neural Networks and Genetic Algorithms. The techniques will only be briefly introduced as an extensive explanation of all of them is beyond the scope of this report.

The descriptions of the actual modules in Chapter 9 will be quite detailed and will often relate to the introduction from Chapter 8. The modules will be divided into three groups: Vehicle, Gun and Radar and will be presented in that order. The Vehicle and Radar modules are quite straightforward to design as their complexity is quite low. The Gun module however is more complex and requires a number of sub modules, as it must do prediction of where the enemy will move and assess about how much energy it should spend shooting at the enemy. Thus the description of this module takes up a larger part of this Chapter.

Finally in Chapter 10 we summarize and conclude about what we have achieved in the design.

6 Criteria Assessments

This Chapter will present the design criteria of our project. These are divided into groups: team, robot and modules, each corresponding to an aspect of Robocode. Additionally we will outline some general criteria, which should cover the overall design and implementation of our project.

In the following Sections we will outline the criteria and explain their meaning related to the project.

6.1 Team

Criterion	Very important	Important	Less important	Not important
Cooperatability			X	
Learnability			X	
Informationsharing			X	
Scalability				X

Table 6.1: Design Criteria for a Team

- **Cooperatibility:** This criterion describes how well the robots on the team can co-operate with each other to obtain a common goal. We have rated this criterion as *less important* as we want to focus on the implementation of the different techniques and models for machine learning, thus making the communicative part of the robot partly irrelevant.

- **Learnability:** Learnability is a measure of how well the team is able to share information learned by a single robot on team basis and to improve sequences of actions according to shared experiences. We have rated this criterion as *less important* as we do not focus on team learning, but on a single agent learning from experiences.

- **Informationsharing:** The informationsharing criterion covers how well the robots should be able to communicate. This may be all the way from small messages to huge amounts of collected data from the field. We have rated this criterion as *less important* as we see it as a useful feature but not a crucial nor important feature..

- **Scalability:** The ability to increase or decrease the amount of robots on the team can be important to a team. We have rated this criterion as *not important* because it is not necessary for us to have a variable team size, where you can add and remove robots from the team if wanted.

6.2 Single Robot

Criterion	Very important	Important	Less important	Not important
Modular		X		
Independent		X		

Table 6.2: Design Criteria for a Single Robot

- **Modular:** By modular we mean how well we separates the functionality inside the robot and how easy it will be to replace a decision making module with another decision making module in the robot, thus making it relatively easy to test more than one technology. We have rated this criterion as *important* because we deem it is important to have the ability to change the internals of a module in the robot and still have the rest of the robot work as before.

- **Independent:** This criterion covers how well the robot is able to operate in the field without any teammates. We have rated this criterion as *important* because we want the robots to act on their own, thus being independent of the other teammates.

6.3 Modules

Criterion	Very important	Important	Less important	Not important
Learnability	X			
Independent		X		
Reusability				X
Testability		X		
Integratable		X		

Table 6.3: Design Criteria for Modules in the Robot

- **Learnability:** Learnability is a measure of how well the module is able to learn from experiences throughout the game and thereby improve the sequence of actions it can perform. We have rated this criterion as *very important* as the goal of our project is to create a team of robots who, on the individual level, are able to improve themselves by applying machine learning.

- **Independent:** Because we want to be able to test more than one technology for the different parts of the robot, we rate it *important* that these modules shall be independent of one another, meaning that no specific module shall be required for other modules to be fully functional. This way it will be possible to replace a module with another module without changing any code in other modules.

- **Reusability:** The reason for having reusability in mind when designing the modules is to be able to use each module in different robots and maybe later in robots for other projects than this. We rate this criterion *not important* because we will only focus on making the modules useful for this project.

- **Uniformity:** We will probably end up having lots of different modules for the robots, and they should all be able to perform the same actions, thereby making modules transparent to the robot and the other modules. To ensure that each module reacts similarly, this criterion is rated *important*.

- **Integratable:** By this criteria we mean how easy it is to integrate a given module in a given robot. We have rated this criterion as *important* as we want every module to fit into every robot.

6.4 General

Criterion	Very important	Important	Less important	Not important
Testability		X		
Correctness		X		
Learnability	X			
Efficiency			X	
Standardability				X
Tournamentability			X	

Table 6.4: General Design Criteria

- **Testability:** Since we will focus on machine learning in this project it is *important* for us, that we are able to test whether our modules increase their performance during experience cf. Definition 5 on page 35.

- **Correctness:** Correctness describes how well the resulting product complies with the described specifications. Should the final result deviate from the described specifications, it would mean that the system does not work as planned.

This could lead to problems which will be hard to trace. The result would also be more complex than necessarily to reach and understand for an outstanding person. We have rated this criterion as *important* because we want to follow our own design and thereby making it easier to catch and correct errors.

- **Learnability:** This criteria describes the balance between hardcoding a robot to master different technics and knowing when to use them, or to let the robot learn different policies by themselves. This criterion also influences how the final product will behave. By letting the robot learn by themselves it will be harder for other robots to recognize their patterns and they can probably come up with a much better pattern than it is possible to hardcode, thus making this criterion *very important*.

- **Efficiency:** Efficiency can be described as how well the robot is able to defeat the other robots in the battlefield. As we deem it more important to learn and use different machine learning techniques, we have rated this criterion *less important*.

- **Standardability:** This criteria describes how much effort we will put into making a robot, which will work with the original version of Robocode. The original Robocode engine has its own sandbox[1] where a lot of restriction have been made. These restrictions have to be fulfilled if a robot should run in the original Robocode engine. However by changing the original code of the Robocode engine, the sandbox can be removed and thereby allow robots to use external applications, thus giving more freedom when designing robots. A robot designed for a modified version of the Robocode engine will not be able to participate in an official Robocode match. As we want to experiment with different types of machine learning and do not want to participate in some Robocode tournament outside Aalborg University, we have rated this criterion as *not important*.

- **Tournamentability:** The effort we will put into developing a robot, which can be set up for a tournament match between the four Dat3 groups working with DSS (Decision Support Systems) and Robocode. Though this is a fun and motivating factor, we do not want to let this tournament give us any boundaries, thus we have rated this criterion as *less important*.

Summary

In this Chapter we presented our design criterion, which was a way to express how we will prioritize our time during the design phase. We rated Learnability as the single most important criteria, while sharing of information and a flexible and modular design was only rated important. It is also important for us to be able to test the design, so Testability and Correctness were also rated important.

[1] Restricted environment where the framework control method calls.

7 Architecture

In this Chapter we will describe the architecture which we have designed for the robots. We will do this first at a general level while emphasizing our design choices. The actual modules will be described in the following chapters.

7.1 Choice of Architecture

Choosing an architecture for the robots requires some consideration about the behavior we wish for the robot to have and the environment it is situated in. The environment in Robocode is as we have mentioned a dynamic environment.

Subsumption could, as we showed briefly in Section 4.1 on page 28, be used to build an architecture for our Robocode robots. The disadvantages of using this architecture is that it does not allow planning as well as internal states. The latter is necessary for us to apply the different kinds of machine learning to the robot.

BDI, as described in Section 4.2.1 on page 29, on the other hand does not have the abilities to react in certain situations where we deem that a given reaction is the best.

These facts imply that we should choose one of the hybrid architectures: InteRRaP or TouringMachines. We have chosen to base our architecture on the TouringMachines architecture, which we described in section 4.4.1 on page 31. This was done partly because we wish to have the ability to do parallel processing in the modules which should make decisions about which actions to perform. This is not allowed within the InteRRaP architecture, which yields sequential processing.

The architecture does however have some modifications, which are not reflected by the original model: There is no `ControlLayer` or `ModelLayer`, instead we have added a module to handle events called `EventDispatcher` as shown on Figure 7.1. The reason for leaving out the `ControlLayer` is that the deliberative layers are to run in parallel, thus making the `ControlLayer` unnecessary. The only controlling necessary is for the `ActionManager` to control priority of the deliberative modules and the reactive layer, where the latter always gets the highest priority. Furthermore as we let the deliberative layers keep their own model of the world, there is no need for a `ModelLayer`.

The `EventDispatcher` module receives events from the Robocode engine through the robot and delivers them to the modules which has subscribed to them. This

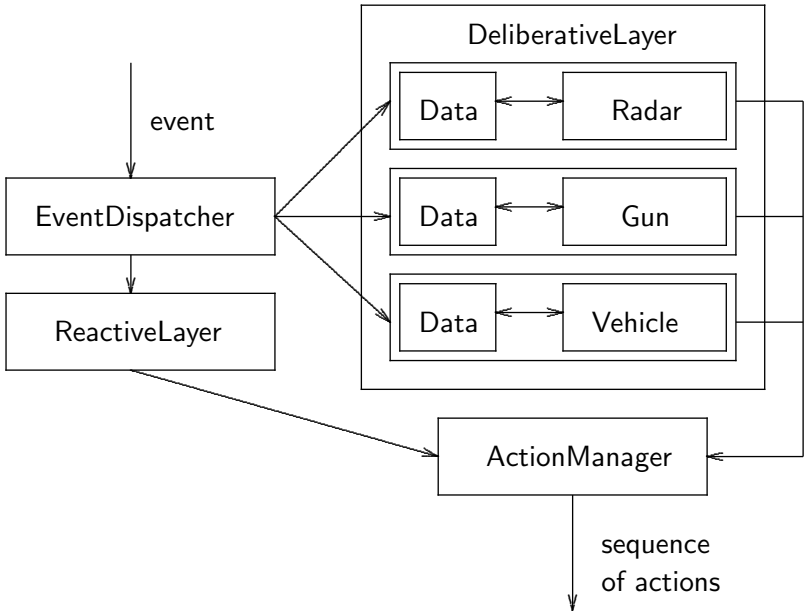


Figure 7.1: The architecture we have designed for ESTIMATOR

design has been made to decrease the amount of events passed to every module, as we were concerned that broadcasting all events to all modules might be an unnecessary overhead within the robot. Another reason is that the actual modules of the robot is not highly coupled to the Robocode engine, only to the `EventDispatcher`, thus yielding a more modular design.

The `ReactiveLayer` is designed to make two checks every round. The first is to check whether it is too close to the wall and the second is to check wether a teammate is in its direct line of fire. If either of these checks are positive, then it may pass a message on to the `ActionManager` that it should not allow the `Gun` module to fire or the `Vehicle` module to move in a certain direction. The functionality represented in the `ReactiveLayer` should be limited to check that actions always are relevant and have no exceptions. We believe that these two checks does fulfill this property.

The `Data` components contains the necessary information needed by a given deliberative module. We had two choices of design; either allowing each module to having its own data component or have one large shared component. The first would be a better choice in the case where every module use different data to base their choices on. But in the case where every module uses the same information, it would just cause redundant data. We have chosen to let each module have its own data component, because they need different data and need to save different things dependent of the technology used within the module. The price of this is that each module must administer its own data storage, possibly leading to redundant information. The advantage of this approach is that the modules are somewhat more independent compared to the solution having only one shared data component.

7.1.1 Handling the Actions Produced by the Deliberative Modules

As we showed in Section 4.4 on page 31 it may pose to be a problem having horizontally aligned layers. The reason is that the number of possible actions which has to be taken into consideration grows exponentially. Our architecture however solves this problem for us, none of the layers can produce the same actions. A `Gun` module may only produce actions that relates to controlling the gun, not for instance choose to move the radar. The available action sorted by each deliberative layer is listed in Table 7.1.

Module	Action
`Vehicle`	setAhead() setBack() setTurnRight() setTurnLeft() setMaxVelocity()
`Gun`	setTurnGunRight() setTurnGunLeft() setFire() setFirePower()
`Radar`	setTurnRadarRight() setTurnRadarLeft()

Table 7.1: Available actions assigned to each module

The `ReactiveLayer` does however, clash with the deliberative modules, as it may produce any of the actions listed above and it may also choose to halt the robot with the `setStop()` method, which stops all actions. The solution to this problem is to let the `ActionManager` contain a buffer, which contains the last actions produced by the deliberative layers. This way, when the `ReactiveLayer` blocks the agent, the `ActionManger` may execute the actions produced by the `ReactiveLayer` produces and resume those of the buffered actions which does not conflict. This way the `ReactiveLayer` always gets prioritized over the deliberative modules, exactly as we want.

7.1.2 Message Passing

We have chosen to let the different modules and objects between each other in order to communicate. This means that the `EventDispatcher` will pass on objects as it receives them through the `Communication` module, without removing unnecessary information. This does however provide for a reasonable optimization, especially if the deliberative modules use much of the same information and if there is much unneeded information within a given `Event` object.

We could instead have chosen to send the bare minimum of information to have the fastest communication, but it would come at the price of parsing messages within the modules. This would lead to more complexity within the modules and a higher coupling between the `EventDispatcher` and the deliberative modules.

Additionally it may be noted that, as it is references being passed between the

robots, the overhead of processing a given object and creating a new instance just to reduce the size of an object, is only relevant when different robots are communicating with each other. This is because robots may not send references between each other, they have to communicate through serialized objects[1]. Serialized objects is an internal in Java making it possible to parse objects directly to a stream, e.g. to a file or as in Robocode where it is a stream between two robots.

7.1.3 Perception of time

The robot perceives time in correspondence with the Robocode engine, which means that the robot is built around "ticks". Each robot has an instance of the `EventDispatcher`, `ReactiveLayer` and `ActionManager` objects. Each of the deliberative modules percepts time as it is best for them, as one module may not be interested in making decisions each tick, while another may just want this. The `ReactiveLayer` and the `ActionManager` on the other hand is evaluated each tick, as they need to process sensorial input and the output of different modules respectively.

7.1.4 Team Architecture

The communication between robots within our team is designed as pictured in Figure 7.2. The team is constituted by five homogeneous robots, which share information by transmitting event objects to each other through the communication module. When this module receives an event object from another robot, it is passed on to the `EventDispatcher` and thereby to the modules subscribed to these events.

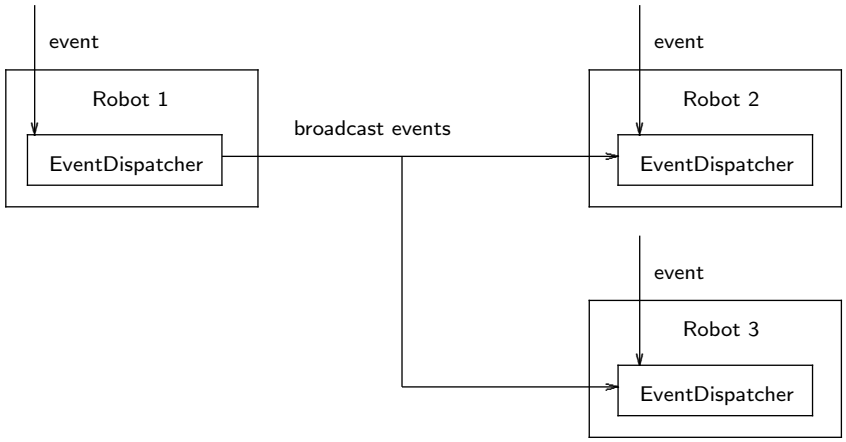


Figure 7.2: Sharing of information between ESTIMATOR robots

7.2 Components

For the sake of clarity we here offer the class diagram of ESTIMATOR in Figure 7.3. When looking at this class diagram we note that `Estimator` is the only class which

[1] Basically this means that when sending an object from one robot to another the receiver robot will receive a cloned copy of the original object.

inherits from `TeamRobot` (as shown in Figure 1.2 on page 16), as this is the only class that needs to be directly in touch with the Robocode engine and receive the generic Robocode events. `Estimator` will pass these events on to the `EventDispatcher` and thereby transmitting them to the subscribers. Additionally we see that all modules inherit from the `Subscriber` class, which has a reference to the `EventDispatcher`, thus allowing the modules to access events. The reason why the modules in the gun inherit from the `Component` class is that they too need to get informations through the `EventDispatcher`; this is only possible if they do inherit from the `Subscriber` class, through the `Component` class. `Module` and `Component` are split into two classes because a `Module` may be exchanged within the robot, while `Component` may not.

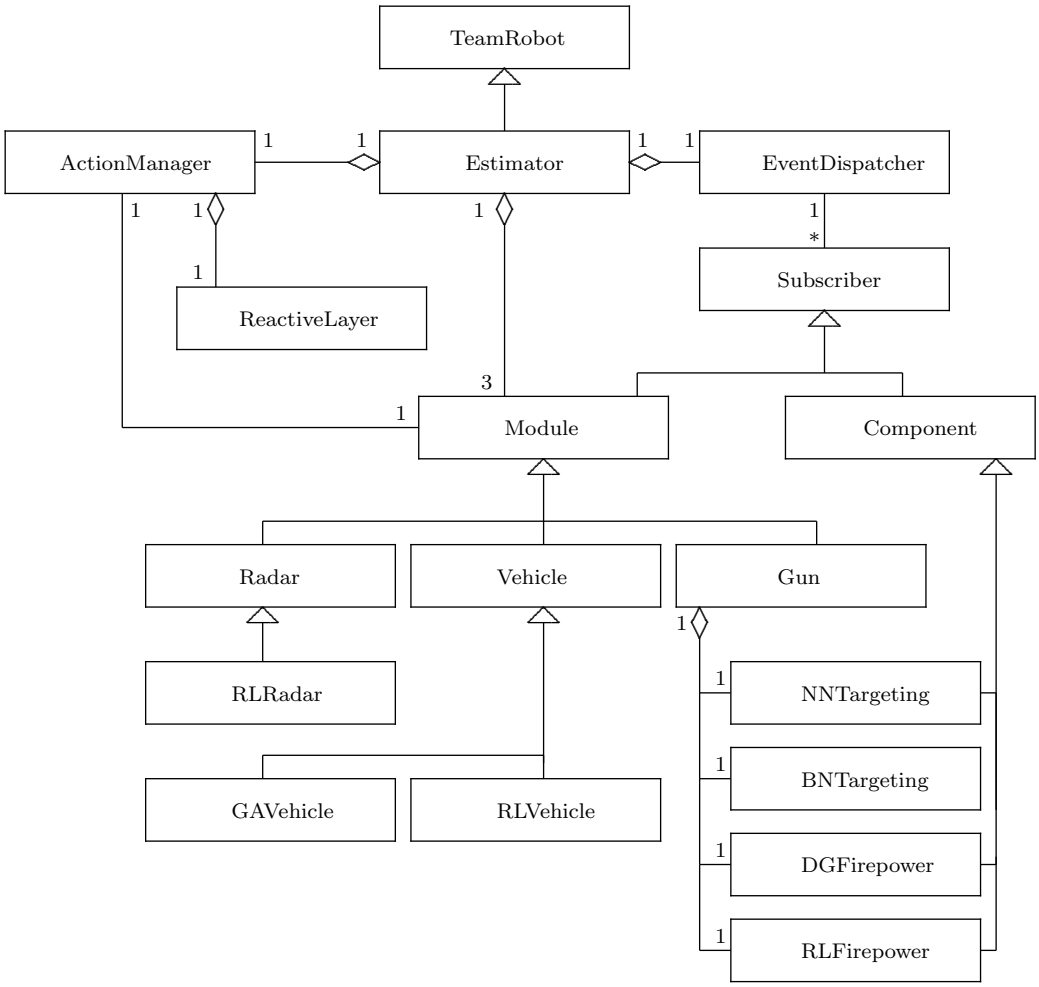


Figure 7.3: Class diagram for ESTIMATOR

Summary

In this Chapter we presented the architecture for the ESTIMATOR robot. It is based on the previously described TouringMachines architecture, but altered to fit our specifications. We explained how the horizontal layering will not be a problem

to us and explained how the robot percepts time. Furthermore we explained how the robots do the little cooperation that they are designed to do.

8 Machine Learning Technics

In this Chapter we will introduce five different approaches to machine learning. First we will introduce Bayesian Networks and Decision Graphs, then we will move on to Reinforcement Learning, Artificial Neural Networks and Genetic Algorithms. The reason for introducing these types of machine learning techniques is to give the reader a foundation for understanding our design, besides that we will often be referring to this Chapter when we introduce our actual design.

8.1 Bayesian Networks and Decision Graphs

The description of Bayesian Networks (BN) and Decision Graphs (DG) in this Chapter is based on [Jen02].

The idea behind BN and DG is to model the relations between cause and effect with the goal of doing reasoning and make decisions under uncertainty.

An example usage of BN is within the health-care sector, where doctors can use BN to diagnose illnesses of patients, by asking the network, what is the most likely illness of the patient given, that the patient have a given set of observable symptoms? Another example where BN are used is in spam filtering of emails.

A BN is a directed acyclic graph of chance nodes, where each chance node, in the graph represent a variable from a given domain, with a finite set of mutually exclusive states. Each chance node in the graph is associated with a probability table, containing the probabilities of the chance node given its parents.

A DG is a BN extended with decision nodes and utility nodes. Decision nodes represents sets of decisions one can make and utility nodes specifies the expected utility of making the different decisions.

We can calculate the expected utility of a decision, $D = d$, with

$$EU(D|e) = \sum_{i=1}^{n} \sum_{X_i} U_i(X_i) P(X_i|D, e), \tag{8.1}$$

where e is the evidence on the chance nodes, $U_1, \ldots, U_n$ is a set of utility functions

over domains, $X_1, \ldots, X_n$. An optimal decision is the decision maximizing Equation 8.1.

Autonomous agents using BNs or DGs can adapt to changes in the environment by updating the probabilities in the chance nodes' probability tables using fractional updating.

If a probability distribution table for a chance node, A, given $B = b_i$ and $C = c_j$, is expressed with fractions like

$$P(A|b_i, c_j) = \left(x_1 = \frac{n_1}{s}, x_2 = \frac{n_2}{s}, \ldots, x_m = \frac{n_m}{s}\right), \tag{8.2}$$

where the sample size $s = n_1 + n_2 + \ldots + n_m$, then when we observe a case like $P(b_i, c_j|e) = z$ and with $P(A|b_i, c_j, e) = (y_1, y_2, \ldots, y_m)$ we can update the probability x_k with

$$x_k = \frac{n_k + z \cdot y_k}{s + z}. \tag{8.3}$$

The idea is that we adjust the probability of the state variable A, by counting the times we have observed the state x_k of A n_k times out of s times, and adding a count $z \cdot y_k$, where y_k is the chance that A is in state x_k, and z is the probability for observing the state of the parents of A. If we have observed the state of the parents of A, then $z = 1$.

Using Equation 8.3 we will continue to update the sample size which can make a system less adaptive to changes in the environment. To avoid this we can use fading.

In fading we can fix the sample size, s^*, and use a fading factor, q_k, as this

$$q_k = \frac{s^* - y_k}{s^*} \tag{8.4}$$

Then when updating the probability for x_k we use

$$x_k = \frac{n_k \cdot q_k + y_k \cdot z}{s \cdot q_k + z} \tag{8.5}$$

In Robocode the decisions a robot has to make could be modeled in a DG, for instance the question of when, where and how fast to move in order to avoid taking damage. The DG would be given a state of the battlefield (enemy positions, movement patterns, etc.) What is the best place on the battlefield to be, in order not to take damage? A BN could be used to give the probability that we will score more points than an enemy, if we try to ram him or move closer to him for better aiming. Another example could be targeting and shooting bullets. When should the gun be fired and where should it point in order to obtain the best probability of a hit?

In relation to team communication a DG could be constructed, which could answer the question of what a single robot should do, given that the other team members execute a specific strategy. This would require that team strategies would be developed, which the robots could choose from.

An advantage of the BN and the DG approach is the ability to make us of existing expert knowledge, which can be built into the network and thereby save learning time. Another advantage is the BNs can be used as a compact way of representing data e.g. information about the battlefield.

A disadvantage is that even if we do have the expert knowledge to construct a network, it is not certain that the network can describe the *optimal* solution, as the model will not be smarter than its creators.

8.2 Reinforcement Learning

This description of RL (Reinforcement Learning) is based upon the description in [Mit97] and [SB98].

As with other machine learning techniques, RL is made for autonomous agents. Given the actions the agent performs in the environment, it will receive real-valued rewards, which is used by the agent to evaluate these actions.

An application of RL is autonomous mobile robots, which moves around on a floor; avoids driving into obstacles [IK97].

The task for a robot using RL is to learn a policy, stating the best action to take in each possible state of the environment. One way to learn this policy is known as *Q learning*. In Q learning, the reward of applying an action in a state can be immediate or delayed, and the policy can be learned without preceding knowledge about the environment. The main idea of Q learning is to build a table of the possible states and actions. For each state-action pair there is a specification of the *utility value* of applying the certain action in that state.

In Q learning the time must be discrete, which fits Robocode since the time is Robocode is discriminated into time ticks.

RL could for example be used in Robocode for controlling the vehicle module of the robot according to the movement of the other players on the field.

A feature of RL is, that the robot learns its own policy, which both can be an advantage and a disadvantage. The disadvantage is that our pre-knowledge of Robocode about good and bad actions can not be pre-specified into the system; the robot has to learn it through the rewards. The advantage is that the robot can learn to control itself, in the way it finds best. This could even be a policy we (as developers) have not considered.

Even though our knowledge cannot be pre-specified into the system, some of the knowledge can still be made available for the robot as actions the agent can select. In this way we can e.g. combine BN with RL by letting a RL agent choose actions recommended by a BN.

A disadvantage with the Q learning is that since it needs a table of all state-action pairs, the number of states in the system grows exponentially with the amount of information relevant for specifying the state. If we as an example represent a Robocode game with a battlefield of 2000×2000 pixels and 10 different robots each placed on one of the pixels on the battlefield, the number of different states will be

$$(2000 \times 2000)^{10} \approx 1 \cdot 10^{66}.$$

This representation of a Robocode game we could add huge amounts of information like the heading, energy, name, velocity and movement pattern of the robots.

For each state we have a set of actions - say 10. Even for the representation only of the robots placements on the battlefield, this will make 10^{67} different state-action

pairs, that the agent needs to learn a utility value for. This way of presenting the problem will be too complex to ever converge to a useful policy.

So for RL to be useful we need to make a rough and simplified specification of the states of the battlefield.

Reinforcement Learning Theory

We will now give some general considerations about how to use RL in Robocode. These considerations will be taken into account when we design our RL modules.

When we in Robocode are in a given state and select a given action, we do not know for sure how the enemy is moving and therefore also do not know what state we end up in. This means that we have to look at Robocode as a *nondeterministic environment*. Since the outcomes of the actions are nondeterministic there are no single reward assigned to a state action pair. We can only talk about an *expected reward* for an state action pair.

We have the evaluation function $Q^*(s,a)$ for a nondeterministic environment from [Mit97, p. 381].

$$Q^*(s,a) = E[r(s,a)] + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q^*(s',a'). \tag{8.6}$$

$Q^*(s,a)$ is the sum of the expected reward received immediately if we apply the action a in the state s, $E[r(s,a)]$ and the value (discounted by γ) of following the optimal policy afterwards. The value of the optimal policy is the maximum Q-value we can get by selecting the action a' if we end up in state s' multiplied with the probability for ending up in state s'. γ is a constant, $0 \leq \gamma < 1$, determining how much we favor an immediate reward compared to a delayed reward. If $\gamma = 0$ then only the immediate reward is of importance.

Since we do not know $P(s'|s,a)$ and $E[r(s,a)]$ we do not know Q^*. Instead we can learn an approximation, Q, of Q^*. Every time we observe the state,s, and select an action a we can collect a reward r and observe the new state s'. By the use of this data we can update Q with the equation

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[r + \gamma \max a' Q(s',a')]. \tag{8.7}$$

We can use the following algorithm to update Q adapted from [SB98], and slightly modify it for use in Robocode.

```
1 Initialize Q(s,a) with an arbitrary value.
2 Repeat (for each Robocode match)
3   observe s.
4   Repeat (for each time interval)
5     Choose a from s using policy derived from Q.
6     Execute action a, observe r and s'.
7     Q(s,a) ← (1 − α)Q(s,a) + α[r + γ max_a' Q(s',a')].
8     s ← s'.
9   Until die or win
```

Listing 8.1: Algorithm for learning the Q function

α is a factor, $0 \leq \alpha < 1$, determining how much we weight new experiences in proportion to old experiences. If α is small new experiences matters less than if α is greater.

One way of determining α is

$$\alpha_n = \frac{1}{1 + visits_n(s, a)}, \tag{8.8}$$

where $visits_n(s, a)$ is the number of times we have chosen this state-action pair. When $visits_n(s, a) = 0$ we will update Q to the experienced value and when $visits_n(s, a) \to \infty$ then $\alpha \to 0$, which correspond to not updating Q at all in Equation 8.7. Using the algorithm shown in Listing 8.1 and Equation 8.8 Q will converge to Q^* if all state-action pairs are visited infinitely often ([Mit97] and [WD92]).

Another idea could be to let α be a constant. The system will then continue to update Q with the same fraction each time. In this way we can make the system adapt to changes in the environment. In our Robocode modules we will let α *be a constant*, because then our robots will then adapt to handle changes in the environment even if it has already updated the $Q(s, a)$ value lots of times ($visits_n(s, a) \gg 1$). But when α is constant it is not given that Q will converge to Q^* [WD92].

If all state-action pairs are to be visited infinitely often, we cannot just take the action, a, with the highest approximated Q value every time we enter state s. This is because our approximation may be inexact. If we always in a given state s, just take the action a with the highest expected Q value, then we will never explore the Q value for selecting the other actions.

To avoid this problem we can use the $\varepsilon-$greedy (or near-greedy) algorithm. The *greedy* algorithm will always choose $\max_a Q(s, a)$, but the $\varepsilon-$*greedy* algorithm selects $\max_a Q(s, a)$ with the probability $1 - \varepsilon$ where ε is $0 < \varepsilon \leq 1$ and takes a random action with the probability ε. If ε is small we will favor the action with the highest approximated Q value, and we will choose less actions with a lower approximated Q value, but the system will only slowly reach a good approximation for Q ([Man] and [SB98]).

Another way of selecting actions can be to select the action a_i with a probability corresponding to the approximated Q value. This can be done by

$$P(a_i|s) = \frac{k^{Q(s,a_i)}}{\sum_j k^{Q(s,a_j)}} \tag{8.9}$$

where k is a constant, $k > 0$, that determines how much we favor actions with a higher approximated Q value [Mit97]. If several actions have nearly the same Q value for a given state, s, this method will select all the actions almost equally often, whereas ε-greedy will favor the best action a for the state s no matter if some other actions have a Q value for s nearly as great as a.

8.3 Artificial Neural Networks

The description of the Artificial Neural Network (ANN) in this Section is based on [Mit97] and [Hea].

The development of ANN has been inspired by nature and the fact that most biological learning systems consists of a complex web of interconnected neurons (e.g. the brain of higher life forms). A neural network provide a general and practical method for doing supervised learning real-valued, discrete-valued and vector-valued functions from data sets. The neural networks are well-suited for learning problems, where the training data is noisy and complex sensor data (e.g. input from microphones).

There are several examples of successful usage of neural networks, e.g. it has been used to steer an autonomous vehicle at normal speed on public highways [Ins]. An ANN have also been used for handwriting, speech and face recognition and for learning robot control strategies.

An ANN is constructed with neurons and these are composed into layers, which are interconnected by synapsis. When the network is running it is in these synapsis that all the calculations are performed, they represent the networks memory. The synapse uses weights to manipulate the patterns, parsed to the network, as they transport them from one layer to the next. When the network is being trained to recognize a pattern it is these weights, that are being adjusted in relation to the learning algorithm used. There are many different synapses configurations, which interconnect the layers differently. We have chosen to use the full-synapse which interconnect all neurons in one layer with all neuron in the next layer as it can be seen in Figure 8.1.

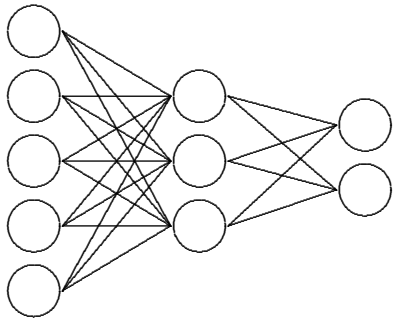


Figure 8.1: Simple Artificial Neural Network with five input gates, three hidden neurons and two output gate interconnected by full-synapsis.

The full-synapse uses the Feed-forward method to transport the patterns between the layers. Therefor the input layer can only send information *forward* into the network and not receive any information backward. The same apply for the other layers in the network, the information stream is only going forward hence the name feed-forward.

A commonly used learning algorithm for an ANN is the back-propagation algorithm. It is appropriate where learning problems have the following characteristics:

- Instances are represented by many attribute-value pairs.
- The target function is discrete-valued, real-valued or a vector of several real or discrete-valued attributes.
- The training examples contains the “correct” answer.

- The training examples may contain errors.
- Long training times are acceptable.
- Fast evaluation of the learned target function may be required.
- The ability of humans to understand the learned target function is not important.

With back-propagation the network is trained by giving it samples of data and the outcome of the network is then compared with the known result. Here the back-propagation algorithm is used to minimize the error of the outcome from the network compared to the known results. This is done by adjusting the weights through all the layers back to the input gates hence the name back-propagation.

The back-propagation algorithm is as follows (from [Mit97, p. 98])

1 `Backpropagate(`*training_examples*`,` η`,` α`,` n_{in}`,` n_{out}`,` n_{hidden}`)`
2 `Each training example is a pair on the form` $<\vec{x},\vec{t}>$`, where` $\vec{x}$ `is the vector of network input values and` $\vec{t}$ `is the target network output values.`
3 η `is the learning rate (`α `is the momentum).` n_{in} `is the number of network inputs,` n_{hidden} `is the number of units in the hidden layer and` n_{out} `is the number of output units.`
4 `The input from unit` i `into unit` j `(without the weight` w_{ji}`) is denoted` x_{ji}`, and the weight from unit` i `to unit` j `is denoted` w_{ji}`.`
5 ▶ `Create a feed−forward network with` n_{in} `inputs,` n_{hidden} `hidden units, and` n_{out} `output units.`
6 ▶ `Initialize all network weights to small random numbers (e.g. between` -0.05 `and` 0.05`)`
7 ▶ `Until the terminating condition is met, Do`
8 ▶ `For each` $<\vec{x},\vec{t}>$ `in` *training_examples*`, Do`
9 `Propagate the input forward through the network:`
10 1. `Input the instance` $\vec{x}$ `to the network and compute the output` o_u `(e.g. using sigmoid) of every unit` u `in the network.`
11 `Sigmoid is` $\sigma(net) = 1/(1+exp(-net))$`, where` $net = \sum_{i=0}^{n} w_i x_i$
12 `Propagate the error backward through the network:`
13 2. `For each network output unit,` k`, calculate its error term` δ_k
14 $\delta_k \leftarrow o_k(1-o_k)(t_k-o_k)$
15 3. `For each hidden unit,` h`, calculate its error term` δ_h
16 $\delta_h \leftarrow o_h(1-o_h)\sum_{k\in outputs} w_{kh}\delta_k$
17 4. `Update each network weight` w_{ji}
18 $w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$
19 `where`
20 $\Delta w_{ji} = \eta\delta_j x_{ji} + \alpha\Delta w'_{ji}$`, where`

```
21      Δw'_ji is Δw_ji from the last iteration.
```

Listing 8.2: Back-propagation algorithm

When using the Back-propagation algorithm one should be aware that, the error is not guaranteed to reach a global minimum but could converge to a local minimum. But even with this problem Back-propagation has shown to be very useful in practice and rarely ends up in a local minimum as this can be avoid by adjusting the momentum.

A neural network could be use in Robocode to recognize enemy movement patterns. If the movement pattern of an enemy is known, it can be taken into account, when shooting at an enemy or when trying to avoiding or ramming the enemy.

Artificial Neural Networks are not suitable to select between different control strategies for a Robocode robot, since the decisions decided by an ANN needs to be categorized into right and wrong and one strategy is rarely definitively right or wrong. E.g. if we had a database with a lot of different battlefield settings, then we need to specify the correct move for each setting and there are not necessarily a single right move for each setting.

8.4 Genetic Algorithms

The description of GA (Genetic Algorithms) in this Section is based upon [Mit97].

The idea of GA is to let hypotheses adapt through evolution, using some guidelines for evolution (e.g. Darwinian evolution). The search for the best hypothesis is thus an approach for machine learning.

GA have been used several times for learning behavior of robots ([Mit97] and [Flo]). GA have also been used to learn the parameters for setting up Artificial Neural Networks [Mit97]. Another use of GA are in games, where opponent (e.g. monsters) behavior adapts to the individual players playing style [Gam].

In GA hypotheses are often represented by bit-strings. A hypotheses can be a representation of environment values and actions. "If-then" rules can also easily be represented by bit strings. Here one part of the bit string will describe the preconditions of a "if-then" rule. An example of preconditions could be the distance to an opponent and an incoming event such as `HitByBulletEvent`. The other part of the bit string will represent the postcondition. An example of postconditions could be the action to perform (e.g. drive ahead).

When GA is implemented the following prototypical genetic algorithm from [Mit97] can be used.

```
1 GA(Fitness_threshold, p, r, m)
2   Fitness_threshold: A threshold specifying the termination
       criterion.
3   p: The number of hypotheses to be included in the
       population.
4   r: The fraction of the population to be replaced by
       Crossover at each step.
5   m: The mutation rate.
6
```

```
7   Initialize population: P ←Generate p hypotheses at random.
8   Evaluate: For each h in P, compute Fitness(h).
9   While [max_h Fitness(h)] ≤ Fitness_threshold do
10    Create a new generation, P_s:
11      1. Select: Probabilistically select (1 − r) · p hypothesis
            from P to carry over to P_s.
12      2. Crossover: Probabilistically select (r · p)/2 pairs of
           hypotheses from P. For each pair, (h_1, h_2), produce
           two offspring by applying the Crossover operator.
           Add all offspring to P_s.
13      3. Mutation: Choose m percent of the members of P_s
           with uniform probability. For each, invert one
           randomly selected bit in its representation.
14      4. Update: P ← P_s.
15      5. Evaluate: For each h in P, compute Fitness(h).
16  Return the hypothesis from P that has the highest fitness.
17
18 Fitness(hypothesis)
19  Return the fitness of the hypothesis.
```

Listing 8.3: A prototypical Genetic Algorithm

The process of evolution in GA is as follows. The fitness of each hypothesis in the population is calculated. Then we select a subset population which we carry over to the new generation without modifications. Thereafter we fill up the rest of the new population with hypotheses on which we do crossover before we add them to the new population. At last we perform mutation on some randomly picked hypothesis in the new population. The process stops when an hypothesis has been found to comply with the fitness-threshold.

Different strategies can be used when probabilistically selecting hypotheses. Examples of selection algorithms are: Fitness proportional selection, Tournament selection and Rank selection [Mit97].

In fitness proportional selection the hypothesis h_i is selected with the probability calculated by Equation 8.10.

$$P(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^{p} Fitness(h_j)}. \tag{8.10}$$

In the above Equation the hypothesis h_i is selected with a probability proportional to the fitness of h_i corresponding to the sum of the fitness of all the hypotheses in the population. This strategy tends to only select the most fit hypotheses.

In tournament selection two hypotheses are randomly paired, and the most fit of the two hypotheses is chosen with probability q and the less fit is chosen with probability $1 - q$.

In rank selection all hypotheses are sorted according to their fitness. The probability of selecting a hypothesis h_i is then proportional to its position in the sorted list. Tournament selection and rank selection tends to yield more diverse populations, which can prevent crowding [Mit97, p. 259]. Crowding is when similar hypotheses

that are more fit than the others, take over a large fraction of the population. This may slow down the evolution.

Another way to prevent crowding is called "fitness sharing". In fitness sharing we lower the probability for selecting a single hypothesis if there are other similar hypotheses in the population.

Crossover and mutation is used to generate new hypotheses for new populations through some defined methods which we will describe now.

Crossover takes two hypotheses and a crossover mask as input and generates an offspring; a new hypothesis. The input hypotheses are used as parents to create the offspring according to the crossover mask which determines how the crossover method will execute the offspring. The offspring will be created where each bit is chosen from either the first or the second parent regaring the crossover mask where the given bit in the offspring is from the first parent if the associated bit in the crossover mask is set to 1 (`true`) and from the second parent if the associated bit it is set to 0 (`false`).

The created crossover masks can result in three different types of crossover:

- **Single point crossover:** The crossover mask is either `true` or `false` from the start up till some point and from this point to the end it will be the opposite value, hence the name single point crossover. An example of such mask could be `11110000`.

- **Two-point crossover:** The crossover mask is `true` or `false` from start to a given point, from this point on until some other point the values will be the opposite, and then from this second point until the end, the values will be as the starting values. An example of such mask could be `11000111`.

- **Uniform crossover:** The crossover mask is randomly `true` or `false` thus giving an offspring where each bit is chosen from either the first parent or the second parent regarding to the crossover mask. An example of such mask could be `10110010`.

We believe that uniform crossover will explore highly different hypotheses faster than single point crossover and two point crossover because it in theory could take all the best parts from .

Mutation shifts one random bit in a hypothesis, thus helping to explore new hypotheses in different scopes of the hypotheses space.

In Robocode, robots could learn their control behavior through GA e.g. the behavior could be represented by a bit string stating the actions of the robot according to the state of the environment and the incoming events.

A part of the bit string of a robot can then indicate, which parts of the environment attribute states are important and another part indicates, which actions to take.

When designing the bit strings of a robot, it will be easier to reach a good hypothesis (a good behavior) if the bit string is kept short. This is because it is easier to find a good configuration over the bits in a short bit string, than it is in a long string. This is due to the reduction of the hypothesis space. GA could also be

used for developing a collision/avoidance tactic e.g. when to avoid other robots and when to ram them.

One disadvantage with GA is that one cannot be sure to find the optimal hypothesis through evolution as it may reach a local maximum. However the crossover and mutation operators generates new hypotheses in a way, where different hypotheses are explored more than when backtracking is used in ANN, so the risk of reaching a local maximum is smaller, but still present.

An advantage with GA is that we do not need to know the optimal behavior of the robot to make a robot which can learn to act in Robocode. We only need a way to determine the fitness for the different robot behaviors - this could be by the accumulated score in a battle.

Summary

This Chapter served as a brief introduction to the machine learning techniques we are going to design our deliberative modules with. These were: Genetic Algorithms, Reinforcement Learning, Artificial Neural Networks and Bayesian Networks and Decision Graphs. A basis for understand the fundamentals of these were given and a number of algorithms and equations have been provided for further reference.

8 Modules

In this Chapter we will present the design of the modules which constitutes the ESTIMATOR robot. As a robot can be divided into three major modules: `Vehicle`, `Gun` and `Radar`, we will discuss them in this order.

9.1 Vehicle Module

The purpose of the vehicle module is to manage all aspects of the robots movement. This means coordinating with the vehicle to perform different movement strategies based on different techniques.

9.1.1 Genetic Algorithms

- Task T: Decide the next movement for the vehicle.
- Performance measure P: The total score for a robot using GA-vehicle, with a fixed gun and radar playing against a fixed set of opponents.
- Training experience E: Playing against a fixed set of opponents.

The idea of using GA in the `Vehicle` module is to let it decide movement tasks depending on some events and states of the environment. Hopefully the way the robot will react to this given environment will improve over time; throughout evolution. This evolution should filter out the worst hypotheses in the population, which do not react in an attractive behavior and leave the best for further evolution.

The fitness for the hypothesis can be the final score for the robot after it has played 20 matches with a fixed (non machine learning) gun module and a fixed (non machine learning) radar module against a fixed (non machine learning) set of opponents. The reason why the fitness is the *final score* after 20 matches is that we want to determine the performance of *the movement policy* as a whole, not measure whether a specific move in a round was a good or a bad move. If the robot moves ahead 200 pixels, how can we tell if this was a good move? The fitness over *20 matches* is that the outcome of a single battle will not give a exact measurement of the vehicle performance, since other factors (e.g. the random selected start position) is of importance of the result. While computing the average fitness over 20 matches we smoothen out the impact of the random selected start position.

The reason why the other robot modules and the opponent should be *non machine learning* is that we want to measure the performance of the different hypotheses only and sort out performance changes according to machine learning in the other modules and the opponents.

As mentioned in Section 8.4 on page 60 it is important to keep the bit sequences as short as possible to reach good hypotheses in a short amount of time, thus leading us to minimize the amount of data to be saved in the string.

We want this vehicle module to react on the following events: `HitByBulletEvent`, `HitRobotEvent`, `ScannedRobotEvent` and our own event `TickEvent`[1]. Each time one of these events occur the hypothesis should specify what action to make. These inputs have been chosen by us in consideration of what we think as being the most significant events to look at.

If the module receives a `HitByBulletEvent` we will allow the module to take two different actions depending on the energy level of the robot, because we believe that different actions can be appropriate if the robot is nearly dead versus if the robot has much energy left. Therefore we dedicate two different actions for moving on `HitByBullet`; one if the energy is 20 units or below and one if the energy is sharp above 20 units.

We want the robot to move differently depending on the position on the battlefield, so we dedicate 3 different action-specifications to this purpose; one if the robot is in a corner, one if it is along the long side of the battlefield and one if the robot is on the center of the battlefield. These actions will be executed each time module receives an `TickEvent`. Figure 9.1 on the following page illustrates our division of the battlefield. Although we are playing on a battlefield of $2000 \cdot 2000$ pixels, we do not care about the outermost part of the battlefield because we will never enter this area due to the `ReactiveLayer`.

When the module receives a `HitRobotEvent` we also want the robot to be able to execute an action. To keep the hypotheses short we will only allow one action to be specified no matter if the robot is hit at the front or back or if the robot is hit by an enemy or a teammate. But since it is of importance to distinguish between these four cases we will interpret the bit string different dependent of the case. The different interpretations will be explained right after we have explained the structure of an action specification.

Finally when the module receives a `ScannedRobotEvent` we want to let the robot choose between four different actions depending on the angle to the enemy. One action if the enemy is in front of us, one if the enemy is behind us and two if the enemy is to the left or right of us. Since the situations where the enemy is to the left or right of us are very similar, we will only specify one action for these situations and then flip the bit for which direction to turn (the fourth bit in the according action sequence, which is explained below). We will not specify any action to the case where we scan a teammate.

Each action specification is represented by 7 bits:

- **Distance:** The first 1 + 2 bits specify the distance the robot should move. The first bit is to represent either forward (0) or backwards (1) direction and

[1] Triggered once for every tick in Robocode.

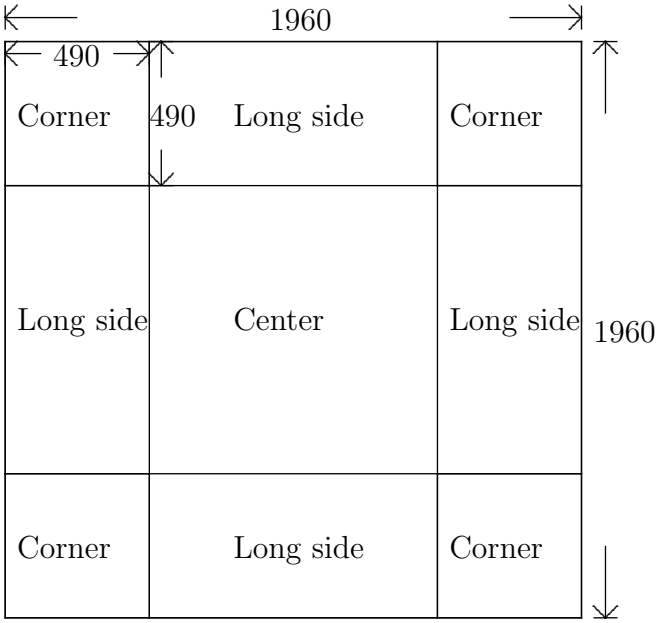


Figure 9.1: Our division of the battlefield into 3 different types of areas.

the last 2 bits is to represent the values 0-3 to use as the distance. When used the value will be `value · 20` to get distances from 0 to 60 pixels.

- **Rotate:** Represented by 1 + 2 bits. The first bit is to either turn left (0) or right (1) and the last 2 bits is to represent how many degrees to turn. The value will be `value · 30` to rotating from 0 to 90 degrees.

- **Speed:** Represented by 1 bit to represent the values 4 pixels/tick (0) and 8 pixels/tick (1).

Once more we have restrained the possible actions to what we think is an absolute minimum to specify useful actions for the movement.

The bit string "110 001 1" will result in the following actions

- **Distance:** 1 10 ⇒ `setBack(2 · 20)`

- **Rotate:** 0 01 ⇒ `setTurnLeft(1 · 30)`

- **Speed:** 1 ⇒ `setVelocity(8)`

The total composition of a hypothesis is as shown in Table 9.1 on the next page

When the robot is hit by another robot we will let bit 36-42 specify the action. If we are hit by an enemy and the enemy is hitting us at our front the actions is interpreted as usual. But if he is hitting us from behind we will reverse the direction bit (the first bit in the according action sequence). So if an hypothesis is specifying that we will ram into the enemy if we are hit at the front then we will also ram backwards into the enemy if we are hit from behind because of the flipped bit.

Bits	Trigger
1-7	`HitByBullet` and robot energy ≤ 20
8-14	`HitByBullet` and robot energy > 20
15-21	`TickEvent` and robot position in a corner
22-28	`TickEvent` and robot position along a long side
29-35	`TickEvent` and robot position at the center
36-42	`HitRobotEvent`
43-49	`ScannedRobotEvent` and $\lvert$angle to enemy$\rvert \leq 22,5^o$
50-56	`ScannedRobotEvent` and $22,5^o < \lvert$angle to enemy$\rvert \leq 67,5^o$
57-63	`ScannedRobotEvent` and $\lvert$angle to enemy$\rvert > 67,5^o$

Table 9.1: The composition of a hypothesis. A hypothesis is composed by 9 action specification of each 7 bits. Each action specification triggered by an event and (except for `HitRobotEvent`) a state in the Robocode game (e.g. the position of the robot on the battlefield).

If we are hit by a teammate we will also use bit 36 to 42 to specify the action, but now we will reverse bit 36 when we are hit from the *front*. In this way will an hypothesis specifying to ram into an enemy and to drive away from the teammates.

One could think that it would be smarter to divide the possible inputs and actions into even more states to let it be more flexible, but as the bit-sequence is quite long as it is we have chosen to keep it this way.

When we evolve our population of hypotheses we will use a slightly modified version of the algorithm shown in Listing 9.1. We will not specify a fitness threshold which will terminate the evolution. Instead we will use a `while(true)` loop and after each evolution, we will print out the best hypothesis, if it has an higher fitness than the other hypotheses we have had earlier in the evolution process. In this way we will continue the search for better hypotheses and for every generation in the evolutionary process we will know the fittest hypotheses in these generations.

We will use an population size of 100 hypotheses. A higher size will result in a highly probability for finding a high fit hypothesis in few evolution steps, but it will take more time to evaluate the fitness of an population if the population is big.

For each generation we will pick out 10 % of the hypotheses using tournament selection to pass on without crossover and 90 % of the hypotheses in a new generation will be formed by uniform crossover. We will use tournament selection to avoid crowding and we will apply mutation to 5 % of the hypotheses in a new generation. According to [Mit97, page 256] a typical mutation rate is 0.1 %. We differ a lot from the typical mutation rate, because we want to speed up the evolution so we can test a lot of different hypotheses in few generations, even though this will probably make the search more stray. For the same reason we will use uniform crossover and we will flip 2 of the bits on the hypotheses chosen for mutation, though it typically is only one bit which is flipped when applying mutation, but we want to spread our search. According to the considerations just discussed our Genetic Algorithm is as shown in Listing 9.1.

```
1 GA(p = 100, r = 0.9, m = 0.05, s = 2)
```

```
2   p: The number of hypotheses to be included in the
       population.
3   r: The fraction of the population to be replaced by
       Crossover at each step.
4   m: The mutation rate.
5   s: The amount of bits to be flipped in a hypothesis for
       mutation.
6
7   Initialize population: P ←Generate p hypotheses at random.
8   h_best ←a random hypothesis from P.
9   Loop forever
10    Evaluate: For each h in P, compute Fitness(h).
11    If max_h(Fitness(h)) > Fitness(h_best)
12      Print out max_h(Fitness(h))
13      h_best ← max_h(Fitness(h))
14    Create a new generation, P_s:
15      1. Select: Tournament select (1 − r) · p hypothesis from P
           to carry over to P_s.
16      2. Crossover: Tournament select (r · p)/2 pairs of
           hypothesis from P. For each pair, (h_1, h_2), produce
           two offspring by applying uniform Crossover
           operator. Add all offspring to P_s.
17      3. Mutation: Choose m percent of the members of P_s
           with uniform probability. For each, invert s
           randomly selected bits in its representation.
18      4. Update: P ← P_s.
19
20  Fitness(hypothesis)
21    Return the score of the hypothesis after 20 Robocode
         matches.
```

Listing 9.1: A prototypical Genetic Algorithm

9.1.2 Reinforcement Learning

We will now describe how we have chosen to use RL for controlling the movement of the vehicle.

- Task T: Decide the movement actions for the vehicle.
- Performance measure P: Measure if the robot learn to move in patterns which minimize the damage received from enemies.
- Training experience E: Playing with a fixed non machine learning radar and gun against different non machine learning robots with different movement patterns.

The task for the RL module is to choose the next action to execute for the vehicle, this should be decided in correspondence with the state of the environment.

An optimal solution for choosing an action to execute, would be to tell the robot exactly which action is the right to use given a certain state of the environment. But because we do not know the right action given a certain state, the robot should have a way to decide what to do based on earlier experience. For modeling this we have chosen to make a RL module.

We have chosen to illustrate our model of the RL module using graphs, the graph for the model will be future described when the different variables and actions used in the graph has been described.

To minimize the amount of state configurations, not every variable in Robocode will be considered in this model. To minimize the states we have decided only to consider the variables, that we think are most important for the vehicle module. The exact values which we have chosen is only based on our intuition, these could be changed during the test phase if we discover other values which would be better.

- **Own energy**: The energy of the robot. This could be relevant if the robot should have the ability to learn not to drive into (ramming) other robots, if its own energy, is so low it will kill itself.

  Values = {Critical (Energy $0 - 10$), Not critical (Energy > 10)}

- **Opponent energy**: The energy of the last seen closest enemy robot. Necessary if the robot should learn to destroy other robots with low energy by ramming them.

  Values = {Slow (Velocity $1 - 4$), Fast (Velocity 5-8)}

- **Distance to opponent**: The distance to the last seen closest enemy robot. Necessary if our robot should learn to avoid colliding other robots or learn to ram other robot, it needs to know the distance to the enemy robot.

  Values = {Near (Distance $0 <$ 100 pixels), Far away (Distance $100 \leq 500$ pixels), Farther (Distance $>$ 500 pixels)}

- **Angle relative to opponent**: The angle to the last seen closest enemy robot, relative to the front of the vehicle. If the robot should learn to avoid or ram other robots, it will be necessary to predict which direction the robot should point.

  Values = {In front (-45^o to 45^o), To the left (-45^o to -135^o), To the Right (45^o to 135^o), Behind us (below -135^o or above 135^o)}

- **Action Running**: Tell if the vehicle is already executing an action. Because more than one action can be executed at the same time, it could be possible to learn only to execute new actions, when no other actions are running.

  Values = {Yes, No}

Based on these inputs the model should be able to choose the best action, for a given configuration of all the described inputs. Not all the actions from Robocode is relevant for the vehicle, so only a few of the actions will be available for this module. As above not every value of the possible actions will be represent because of the complexity it would add to the model. The actions relevant for the vehicle are.

- `setAhead`: Gives the option to move forward. Can be used for avoiding or ramming enemy robots. For avoiding bullets or avoid being rammed by enemies small parameter should be enough. For moving around on the battlefield larger values will more useful.

  setAhead input values = {20, 100)}

- `setBack`: Could be modeled by given negative input to `setAhead` but for readability we have chosen to use `setBack`

  setBack input values = {20, 100}

- `setTurnLeft`: Gives the robot the option to rotate, so that all of the battlefield can be used. As with `setAhead` and `setBack` we have chosen to have a small value and a larger value as parameter for rotating the robot. We have chosen not to have the large value close to 180, because turning the robot a lot takes time and makes the robot vulnerable to attacks. If the robot want to rotate more than our large value, it should learn to combine two or more actions of `setTurnLeft` right after each other.

  setTurnLeft input values = {15, 60}

- `setTurnRight`: Could be specified with use of `setRotateLeft` with negative input, but again to ease the readability `setRotateRight` will be used in our module.

  setTurnRight input values = {15, 60}

- `setMaxVelocity`: This action gives the robot the opportunity to change its velocity, this could be used to confuse enemy robots, in such a way that it will be harder to predict the robots movement. However by choosing to low values for the speed of the robot, will cause the robot to be an easy target, because the lack of movement.

  setMaxVelocity input values = {4, 8}

- `Stand still`: This action is not an action from Robocode, but a lack of action which will give the robot the possibility to do nothing. This action could be used if the robot wants to wait on earlier tasks to complete, before activating new actions.

The model as described now could be built as the graph illustrated in Figure 9.2 on the facing page.

The graph could be viewed as an decision graph, but in this graph there should always be evidence on all the nodes except on the `state` and `utility` node, but with evidence on all the other nodes the state of the `state` node will also be known.

The action table on Figure 9.2 is the decisions the robot can choose. The utility node of Figure 9.2 indicates a table for the approximated Q-values.

The double arrows on Figure 9.2, illustrate that the figure repeats itself. Figure 9.3 on the next page shows a small piece of what this figure would have looked like, without the double arrows.

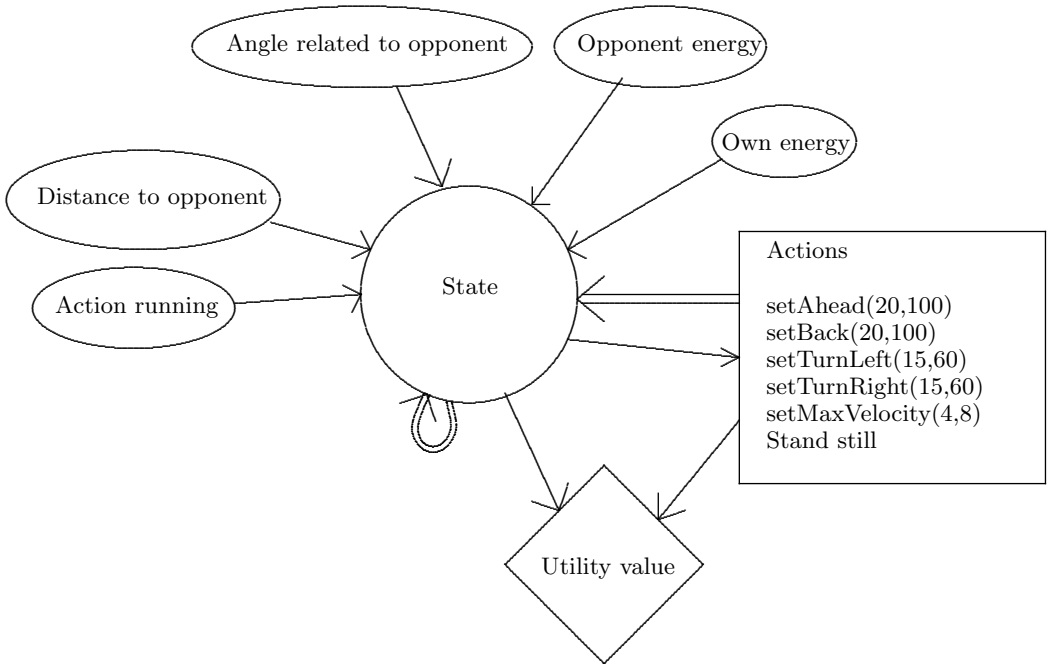


Figure 9.2: Graph for the Reinforcement Learning movement module.

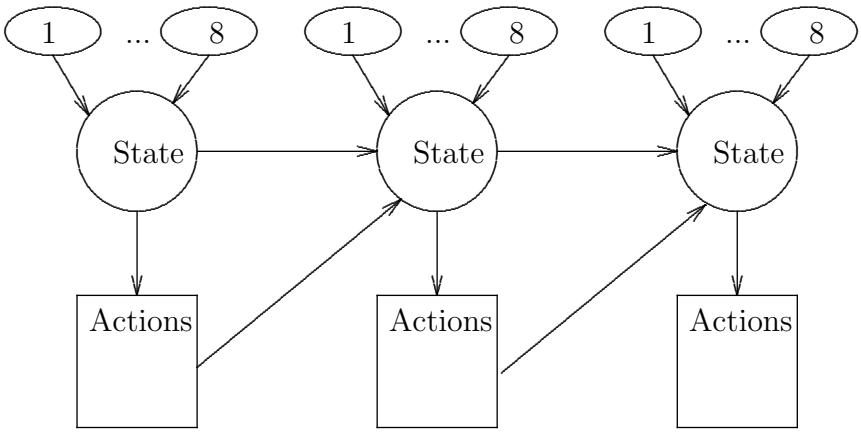


Figure 9.3: Graph which shows the double arrows from Figure 9.2 expanded into three steps.

For learning the Q function we can use algorithm shown in Listing 8.1 on page 54. The time interval for this module will be every 8th. Robocode time ticks. (8 because it is the time it takes the radar to rotate 360 degrees, which therefor should result in evidence of some kind) We will use Equation 8.9 on page 55 to select actions, because some actions can be equally good and if we switch between these actions our robot will move more unpredictably to the enemies.

α can be 0.01 and $\gamma = 0.5$ in the Q function and $k = e^1$ in Equation 8.9.

To calculate the number of state-action pairs, all possible states must be multiplied with each other. This result multiplied with the possible action gives the number of state-action pairs. For the described input and actions the number of state-action pairs will be

$$2 \cdot 2 \cdot 3 \cdot 4 \cdot 2 = 96$$

Multiplied with all the possible action which can be chosen

$$96 \cdot 11 = 1056 \text{ Entries.}$$

This means that the RL module should update a Q-table with 1056 entries. The Q-table should be updated as shown in Equation 8.7 on page 54. If all state-action pairs are visited equally often and if a round in Robocode consist of (say) 100 time ticks, then this system will require 10 rounds to update all Q-table entries once. Since not all states occur equally often and we most of the time only select the action with the highest approximated Q-value, then this system will need more than 10 rounds to update all states once, and even more rounds for the Q-table to converge to Q^*.

Depending on the reward function we create for the robot, this module should learn different tactics. For instance giving the robot a bonus every time it collide with another robot should result in a robot which use ramming as a technique for killing other robots. By giving the robot a negative reward every time an enemy robot gets to near, it would properly results in a robot which will keep an almost constant distance from the enemies and thereby avoid being rammed.

By giving reward for both of these scenario would cause a conflict in the Q-table, because by avoiding enemy robots it will be impossible to ram the enemies.

Therefore we will try different rewards function for the vehicle module. The reward functions we will try is shown in the next five Sections.

9.1.2.1 Reward Function for Ramming

We wanted to see if it was possible to have the robot learn to ram other robots on the battlefield, therefor this reward function has been created. To make our robot learn to ram opponents we have chosen to reward it with a positive reward whenever it rams an enemy robot. By not getting rewards for anything else than this, the reward function should force our robot to seek ramming rewards.

$$r_1(s,a) = \begin{cases} \textbf{2}\cdot\textbf{Damage done to an enemy} \text{ if ramming an enemy robot} \\ \textbf{0.30}\cdot\textbf{Total damage done to enemy} \text{ if killing an enemy by ramming} \\ \textbf{0} \text{ if none of above is fulfilled} \end{cases}$$

This reward function have been made with the intention of having the robot learn to kill other robots by ramming them. The values in the reward function are the same values as the Robocode engine use to calculate the final score of every match. The graph of the scores during a match should converge to a positive value.

9.1.2.2 Reward Function for Avoiding Enemy Robots

In contrast to the former reward function, we also wanted to see if it was possible to learn a robot to keep distance to enemy robots, and by doing so maybe live longer. To make the robot learn to keep away from opponent, we have chosen to feed it negative rewards whenever it is too close to enemies. If this should not be enough, the robot will also be rewarded with a negative reward whenever an enemy robot rams our robot. This should learn our robot to keep away from being rammed and situations that could lead to our robot being rammed.

$$r_2(s,a) = \begin{cases} \textbf{-10} \text{ if distance to enemy robot} < 50 \text{ pixels} \\ \textbf{-1} \text{ if distance to enemy robot} < 200 \text{ pixels} \\ \textbf{-0.30}\cdot\textbf{Total damage done to enemy} \text{ if killing an enemy by ramming} \\ \textbf{-Damage} \text{ if hit by enemy} \\ \textbf{2} \text{ if none of above is fulfilled} \end{cases}$$

This reward function should result in a robot which avoids being rammed by enemy robots. The learning rate of this reward function could be measured the same way as the former method.

9.1.2.3 Reward Function for Bullet Avoidance

If our robot could learn to dodge bullets it would mean longer survival and thereby give the robot a better chance at winning a battle. Even though the Robocode API does not have any information about when enemies are shooting bullets, we have tried to make a reward function which gives negative rewards when hit by a bullet and positive rewards when not. Hopefully this will result in a robot which learn to dodge bullets or just avoid situation where it would be an easy target.

$$r_3(s,a) = \begin{cases} \textbf{-Damage} \text{ if hit by bullet} \\ \textbf{0} \text{ if none of above is fulfilled} \end{cases}$$

The reward function specified should give the robot the ability to learn to avoid bullets. However because the reward function have no positive values to reward with, the graph of the scores should converge to 0.

9.1.2.4 Reward Function Based on Robocode Rewards, Avoiding Enemy and Bullets

To maximize the point received by the Robocode engine after each battle, we have made a reward function which uses some of the same rewards as the Robocode engine. Beside these Robocode rewards we have choosen some extra rewards, so

that our robot should last longer on the battlefield, thereby give the gun module a better chance of gaining more Robocode rewards.

$$r_4(s,a) = \begin{cases} \textbf{-10} \text{if distance to enemy robot} < 50 \text{ pixels} \\ \textbf{-1} \text{if distance to enemy robot} < 200 \text{ pixels} \\ \textbf{-2}\cdot\textbf{Damage done to enemy} \text{ if ramming an enemy robot} \\ \textbf{-0.30}\cdot\textbf{Total damage done to enemy} \text{ if killing an enemy by ramming} \\ \textbf{50} \text{ if an enemy robot dies} \\ \textbf{10}\cdot\textbf{(Number of enemies-1)} \text{ if we win the battle} \\ \textbf{-Damage}\cdot\textbf{5} \text{ if hit by bullet (times 5 just to fit with the other values)} \\ \textbf{2} \text{ if none of above is fulfilled} \end{cases}$$

This reward function is a combination of the "Avoiding Enemy-function", the "Avoiding Bullets-function" and the reward given by the Robocode engine. This function also returns a reward when an enemy dies (same amount of points as the Robocode engine gives) and likewise a reward is returned when winning a battle. This reward function should give the robot the possibility to keep a distance from both enemy robots and bullets, while the gun module on the robot can try to kill opponents. Plotting the results from this reward function should result in a graph which converge to some positive value.

9.2 Gun Module

The purpose of the `Gun` module is to determine if the gun is to fire and how much energy to put into the bullet. Additionally it also has to predict where the enemy will be at a given time. This yield two different submodules: Targeting and Energy Choice, which we will now describe.

9.2.1 Bullet Energy Choice with Decision Graphs

In this section, it will be describe how we can decide the amount of energy to put into the bullet using Decision Graphs.

- Task, T: Determine amount of energy to put into a bullet.
- Performance measure, P: The total utility of all the bullets fired in a round.
- Experience, E: Experience is gathered by scanning an enemy and shooting a bullet, and recording if the bullet hit of missed.

When we want to fire on an enemy, there is a trade-off between the amount of energy we will fire with and the amount of energy we are willing to loose. The more energy located in a striking bullet, the more damage we cause on the enemy and the higher our score in the final ranking will be. If the bullet fired misses, then all the energy we have put in the bullet will be lost. If we put a lot of energy in a bullet it will move slowly against the target, in proportion to the speed the bullet will obtain if we fire with low energy. The speed of the bullet influences on the chance of hitting the enemy, the faster the bullet the better the chance of hitting the enemy.

Besides the speed of the bullet there are a lot of other factors which influences our chance of hitting an enemy target. We assume that the direction of the enemy, related to our position, also has influences on our chance of hitting him. If the enemy robot is heading straight against us, it will probably be easier to hit him, than if he is moving sideways to our position or away from us. Furthermore it will be easier to hit an enemy target if our gun is already pointing in the right direction so we do not have to wait for the gun to rotate 180° before it is ready to fire.

Additionally there will probably be a lot of other factors influencing our chance of hitting the enemy target, factors we have not, or can not, take into consideration at this point. All these factors can be present at the same time with varying strengths. If we assume that these factors are all independent then the combined probabilities for hitting the enemy can be modeled with a chance node, as illustrated in Figure 9.4.

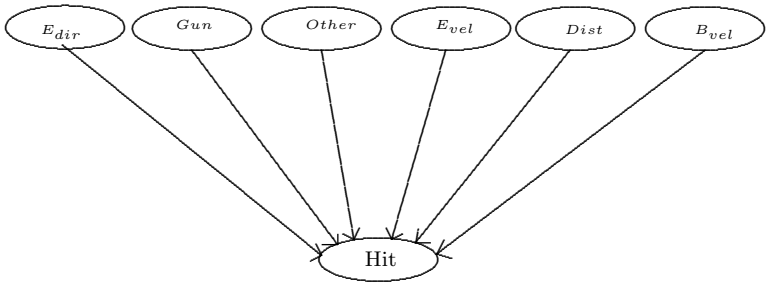


Figure 9.4: Bayesian Network for the probability of hitting an enemy.

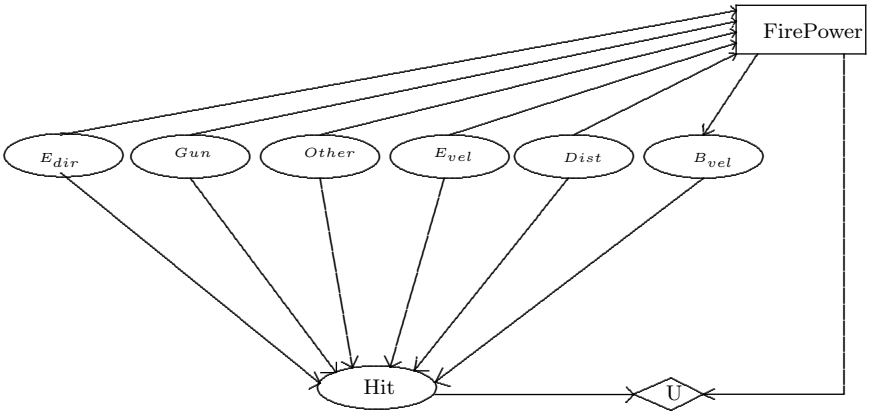


Figure 9.5: Decision Graph for selecting firepower.

We can extend this Bayesian Network to an influence diagram by adding a decision node `FirePower`, which determines the bullets velocity and a utility node, which is the expected utility of firing a bullet given the chance for hitting the enemy and the decided amount of energy in the bullet. See Figure 9.5. If we hit the target the utility is given by Equation 9.1 and if we miss the target the utility will be given by Equation 9.3. See Table 1.1 for the values of damage caused and the point bonus. In total, if we hit the utility U will be:

$$U = \text{DamageCaused(FP)} + \text{PointReturned(FP)} - \text{EnergySpend(FP)} \tag{9.1}$$

$$\Downarrow$$

$$U = \begin{cases} 6 \cdot \text{FP if FP} \leq 1 \\ 8 \cdot \text{FP} - 2 \text{ if FP} > 1 \end{cases}, \tag{9.2}$$

where FP is the bullet energy.

And if we miss

$$U = \text{EnergySpend(FP)} \tag{9.3}$$

$$\Downarrow$$

$$U = -\text{FP}. \tag{9.4}$$

If in the Decision Graph the "E_{dir}" node has 3 states, "*Gun*" has 3 states, "*Other*" has 1 state (always "on"), "E_{vel}" has 3 states, "*Dist*" has 5 states, "B_{vel}" has 7 states and "*Hit*" has 2 states, this makes in total $3 \cdot 3 \cdot 1 \cdot 3 \cdot 5 \cdot 7 \cdot 2 = 1890$ probabilities in the Table for the chance node "*Hit*".

If we want this model to have a chance of adapting to the environment during one or a few rounds, 1890 is too many probabilities to work with. If we for example have a round where we have 100 decisions, then based on this model we will need over 19 rounds to update every probability *only once*. And several updates are needed to get the model to adapt to the environment.

If we extend the model to the one illustrated in Figure 9.6 we can reduce the number of probabilities to be specified. In this model we have introduced inhibitors. Now the chance of hitting is the product of the chances for hitting due to each inhibitor. Now we only have $3+3+1+3+5+7 = 22$ probabilities to specify - and a plain or-gate.

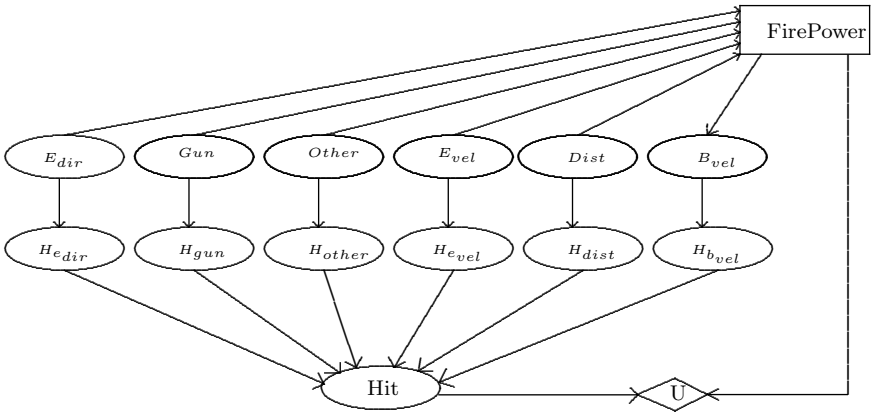


Figure 9.6: Decision Graph for selecting firepower. We have introduced inhibitors to reduce the size of the Decision Graph.

When deciding which bullet energy to shoot with, we will choose the bullet that maximizes the expected utility.

$$EU(\text{Firepower}|e) = U(\text{Hit}, \text{Firepower})P(\text{Hit}|\text{Firepower}, e), \tag{9.5}$$

where

$$e = \{Dist = d_a, E_{vel} = e_b, E_{dir} = d_c, Gun = g_d, Other = On\}.$$

We can ask a domain expert, which in our case this would be us self, to estimate how much each inhibitor affects the hitting probability. But this estimate will probably be uncertain and if only general for only a set of all Robocode matches. If we updates this model using fractional updating, like described in Section 8.1, after having seen the outcome of each fired bullet, we can then get this model to adapt to the opponent during one round or a couple of rounds. The estimated probabilities can be seen in Appendix D on page 160.

When firing and observing the outcome of a bullet we have the evidence e, which gives us:

$$\begin{aligned} e = \{&Dist = d_a, E_{vel} = e_b, E_{dir} = d_c, Gun = g_d, Other = On, \\ &B_{vel} = b_e, Hit = h_f\}. \end{aligned}$$

For each inhibitor, I_i, we have a distribution $(x_1 = \frac{n_1}{s}, x_2 = \frac{n_2}{s}, \ldots, x_m = \frac{n_m}{s})$, where $s = n_1 + n_2 + \ldots + n_m$.

After observing the outcome of a bullet we can now calculate the distribution $P(I_i|e) = (y_1, y_2, \ldots, y_m)$ and update the distribution for I_i with Equation 8.3 on page 52 where $z = 1$, because we know for sure the state of the inhibitor, when we observe the outcome of the fired bullet:

$$x_k = \frac{n_k + y_k}{s + 1}, \tag{9.6}$$

for the k'th entries in the distribution. In this way we update the distribution for each inhibitor after each shot.

If our robot always selects the bullet with the energy, which will give us the highest expected utility then the robot will be rational in this field. We expect the influence for each inhibitor to vary in relation to our opponents. If we just update our distributions as described earlier and continue to increase the sample size, then an agent with a high sample size will only adapt slowly to an environment which can change from match to match.

To avoid this we can use fading as described in Section 8.1, and fix the sample size to e.g. $s^* = 50$. In this way we can fade out some of the old experiences each time we update the probability distribution for an inhibitor. If we increase the sample size the system will be more resistant to changes in the environment and if we lower the sample size the system will be more adaptive. But if we set the sample size too low, then we will trust too much on the results of the latest fired bullet, which can lead to uncertainty if the results of the latest fired bullet was abnormal.

The model in Figure 9.6 can be made even smaller by using divorcing on the parents to the or-gate, which is the node `Hit`. If all the six binary inhibitors are parents to the or-gate, then the probability table to the or-gate contains $2^{(6+1)} = 128$ probabilities (all 0 and 1s). If we use divorcing like illustrated in Figure 9.7 then we can reduce the number of probabilities needed to model the or-gate to 5 tables with each $2^{(2+1)} = 8$ probabilities giving 40 probabilities in total.

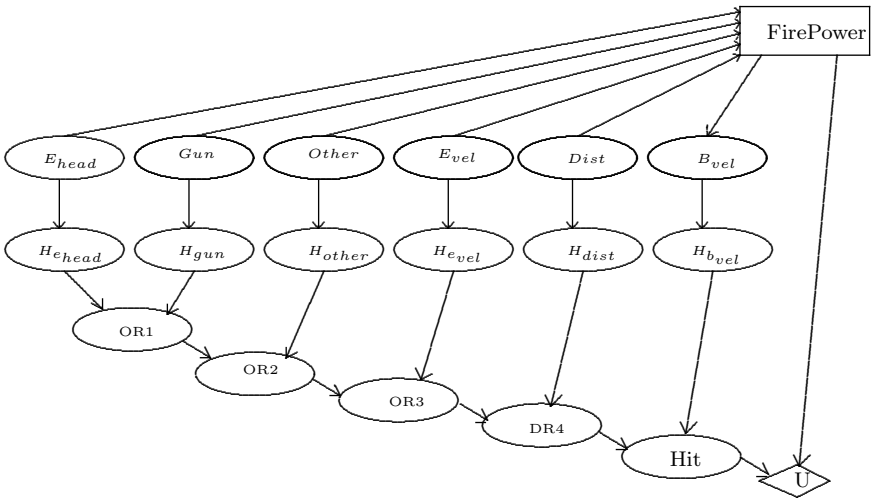


Figure 9.7: Decision Graph for selecting firepower. We have introduced divorcing to reduce the total size of the or-gate.

9.2.1.1 The Node States in the Decision Graph

We have to classify the different states the node can be in, when working with the Decision Graph and to aid in analyzing these states, we have to look at a situation where we will use the graph.

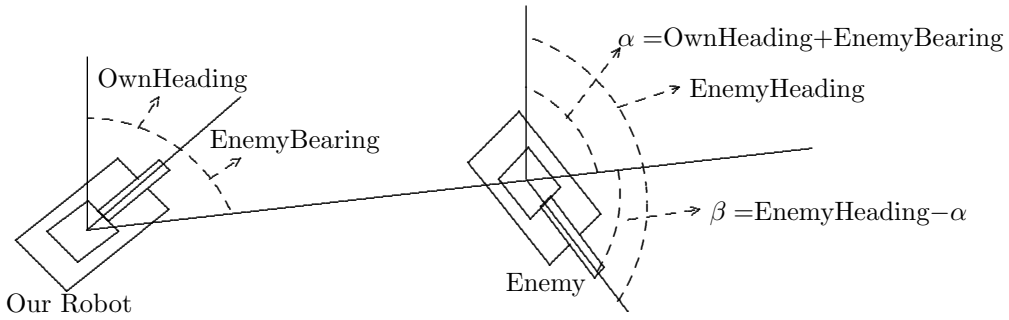


Figure 9.8: The direction of the enemy in relation to our position.

The angles in Figure 9.8 is calculated in two steps. First the angle α is calculated, which is the angle between the line intersecting the two tanks and the direction facing up the screen. Then the direction β is calculated, which is the enemy heading minus α. The enemy heading can be extracted from a `ScannedRobotEvent`.

- **Enemy direction**: This chance node is called "E_{dir}". It denotes the direction β, which is the enemy moving in respect to our position.

  Values = {Against us ($135° < \beta \leq 225°$),
  Sideways to us ($45° < \beta \leq 135°$ or $225° < \beta \leq 315°$),
  Away from us ($-45° < \beta \leq 45°$)}

- **Gun ready**: This chance node is called "*Gun*". It denotes the guns pointing direction relative to the direction of the enemy.

Values = {Almost ready ($|v| \leq 20^o$),
Turn a little ($20^o < |v| \leq 80$),
Turn a lot ($80^o < |v| \leq 180^o$)}

The value range for "almost ready" is the range about, which the gun can point at the enemy within one tick (remember that the gun rotates with max 20 degrees per tick). For the value within the "Turn a little"-state the gun can be ready within maximum 4 ticks, and in "Turn a lot" will it take above 4 ticks.

- **Distances to enemy**: This chance node is called "Dist". It represent the distance in pixels to the enemy.

Values = {$0 \leq d \leq 100$
$100 < d \leq 300$
$300 < d \leq 600$
$600 < d \leq 1200$
$1200 < d$}

- **Enemy velocity**: This chance node is called "E_{vel}. It denotes the enemies velocity in pixels per tick.

Values = {$0 \leq v < 2$
$2 \leq v < 5$
$5 \leq v < 8$}

- **Other factors**: This chance node is called "Other" and it denotes the background factors influencing our chance of hitting. It has only one state, because the background factors is always on.

Values = {On}

- **Fire power**: This decision node is called "FirePower" and it is listing the possible choices of energy to fire the bullet with.

Values = {0.0, 0.1,
0.5, 1.0,
1.5, 2.0,
2.5, 3.0}

The value 0.0 indicates do not shoot. The value 0.1 denotes the minimum bullet energy in Robocode. The value 3.0 denotes the maximum bullet energy in Robocode, and the rest of the values is chosen to be an even distribution over the bullet energy range.

- **Bullet velocity**: This chance node represents the velocity of the fired bullet in pixels per tick. Each possible velocity corresponds to a energy choice in "FirePower".

Values = {N/A, 19.7,
18.5, 17.0,
15.5, 14.0,
12.5, 11.0}

- **Hit**: This chance node is an or-gate. It returns "miss" if one or more of the inhibitors is in the state "miss", otherwise is returns "hit".

  Values = {hit,
  miss}

9.2.2 Bullet Energy Choice with Reinforcement Learning

- Task, T Determine amount of energy to put into a bullet.
- Performance measure, P The cumulative reward after a round.
- Experience, E Experience is gathered by scanning an enemy and shooting a bullet.

If we assume that the sequence of our energy choices is of importance, we can make a Reinforcement Learning system that can be trained to learn the utility of the bullets energy choices.

Let the states be defined by four factors:

- **Distance:** The distance to the enemy, five states.
- **Velocity:** The velocity of the enemy, three states.
- **Direction:** The direction in which the enemy is moving, three states.
- **Gun:** Our guns direction in proportion to the direction to the enemy, three states.

Let the interval of each factor be as described in Section 9.2.1.1, and let the actions be the choices of bullet energy, which has eight states, then we have 135 different states and eight actions. Then we can define the reward function as the utility function described in the Equations 9.2 and 9.4 and assign the reward at the same time as the bullet hits something (the enemy, the wall or something else).

Since Robocode is a non-deterministic environment as we explained in Section 8.2 on page 54, when we fire at an enemy it will not be given if we hit or miss him.

In Q learning the time must be discrete. The ticks in Robocode cannot be used as the discrete time for Q learning because the division of time we need has to be in a way so that we can make exactly one decision about how much energy to put in the bullet for each module time tick. We do not need to select one bullets energy value per tick, thus we cannot fire frequently because of the guns cooling rate. So we define the module time to be the time which increases by one each time a decision about a bullet energy is needed.

The reward given for selecting the bullets energy will be determined when the bullet hits something. This means that the reward for the bullet energy choice will be delayed. The reward will be given to the module to the next time the module is activated after the bullet have hit something.

The following pseudo-code can be used to approximate Q to Q^* in this module:

```
1  Initialize Q(s,a) arbitrarily
2  Repeat (for each Robocode match)
3    Wait for first activation of the energy choice module
4      Initialize observe s
5      Choose a from s using policy derived from Q
6      Execute a
7    Repeat (Wait for next activation of the energy choice
        module)
8      Observe r, s'
9      Calculate α
10     Q(s,a) ← (1 − α)Q(s,a) + α[r + γ max a'Q(s',a')]
11     s ← s'
12     Choose a from s using policy derived from Q
13     Execute a
14   Until die or win
15   Observe r, s
16   Q(s,a) ← (1 − α)Q(s,a) + α[r]
```

Listing 9.2: Learning the Q function

This pseudo-code is slightly modified from the one in Listing 8.1 because we cannot immediately observe the reward. In this algorithm we wait to determine the reward, r, and observe the state, s', until the module time increases by one (that is we wait for the next activation of the module), and when we have the r and s' we can update $Q(s, a)$. When $Q(s, a)$ is updated we can continue as in Listing 8.1 with choosing an action a.

In this module we will use Equation 8.9 to select the bullets energy (the action), because some bullet energy choices can be equally good, and if we switch between these choices, our bullets will move more unpredictable to the enemies.

α can be 0.01, and $\gamma = 0.1$ in the Q function and $k = e^1$ in Equation 8.9.

9.2.3 Artificial Neural Network Targeting

In this Section we will describe a targeting strategy, that uses an ANN to predict where to shoot. This means predicting enemy movement and turning the gun in the direction of the enemy.

- Task T: Predict enemy movement.
- Performance measure P: Comparison of how many of the prediction was correct in relation to the actual position of the enemy.
- Training experience E: Log a robots movement pattern and train an ANN to learn this pattern.

The task of predicting enemy movement, can be quite extensive and since extensiveness usually leads to complexity it is probably a good idea to limit the predictions. Since robots in Robocode acts autonomously through out the time span of the game, predictions that stretch too far into the future will become less and less dependable. We will therefore limit the predictions to only span a short time into the future.

Since we have no way of knowing what an enemy robot is about to do, we can only give a qualified guess (a prediction) of its position, by using its last known positions. This is information we can gather directly by observing the enemy robot using the radar.

9.2.3.1 ANN Input and Output Encoding

We have chosen to use a *feed-forward* network using *back-propagation* to update the network. The combination of these two techniques is the most widely used of all the ANN techniques [Hea].

To construct an ANN, we need to specify the number of input gates and what the input is. We also need to specify output gates, and what the output means.

We have chosen to encode the input gates with information about the coordinates, (x, y), of the scanned robot at time t, which we get from the data module (see Figure 7.1 on page 46). The time t is collected together with the (x, y) coordinates from this module. We have decided to use 34 input gates, which are grouped together in 10 triples of gates and the last four are single gates. The 10 triples are encoded with coordinates and times obtained from scan-events (x, y, t). The last four single input gates $t_{in,1}, \ldots, t_{in,4}$ are used to tell the network at which time, the output locations should be predicted. In other words we want to have a way of saying that to 10, 15, 23 and 26 ticks in the future we want the location of the enemy robot. This is controlled by the four last gates. This will come in handy when we are to train the network.

We have decided to make the network predict the next four locations. We have made this decision based upon the idea that if the robot cannot get a clean shot at the first predicted location it will be able to start shooting after the next location. The 4 output pairs are encoded with x and y coordinates.

9.2.3.2 Training

The ANN is a two layered network with one hidden layer as illustrated in Figure 9.9.

Training of the network will be done by playing against a single robot, and logging the enemy locations to different times. These information can be collected by the `ScannedRobotEvent`s. The information about the enemy locations and the time for the first 10 scans is given in the 10 triples of gates and the time for the next 4 scans is given to the gates $t_{in,1} \ldots t_{in,4}$. The predicted locations are compared to locations of the 4 scans to $t_{in,1} \ldots t_{in,4}$, and the weights in the ANN is adjusted using back-propagation. Then the ANN is trained with the time-locations for scan number 2 to 11 as input together with the time for scan number 12 to 15 as validation data. Then 3 to 12 and 13 to 16 and so on. The algorithm is like

```
1 Form the training examples
2   Receive a log of m (x,y,t) scans.
3   For i = 1 to m - 14
4     Extract (x_i, y_i, t_i) from the first i to i + 10 scans and
          extract t_i for scan i + 11 to i + 14.
5     Form a vector, x⃗, of the input values.
6     Extract (x_j, y_j) from scan number i + 11 to i + 14.
```

```
7     Form a vector, $\vec{t}$ of the output values.
8   Form a collection, training_examples of $<\vec{x},\vec{t}>$ and use it when
      calling ``Backpropagate(training_examples, $\eta$, $n_{in}=34$,
    $n_{out}=8$, $n_{hidden}$)''
```

Listing 9.3: Algorithm for Training of ANN

It usually requires many iterations over training examples for the back-propagation algorithm to adjust the weights. So training cannot take place *during* the Robocode matches, but must take place prior to matches. When using the ANN during a match we need 10 events and 4 times in the future where we want the coordinates to be predicted.

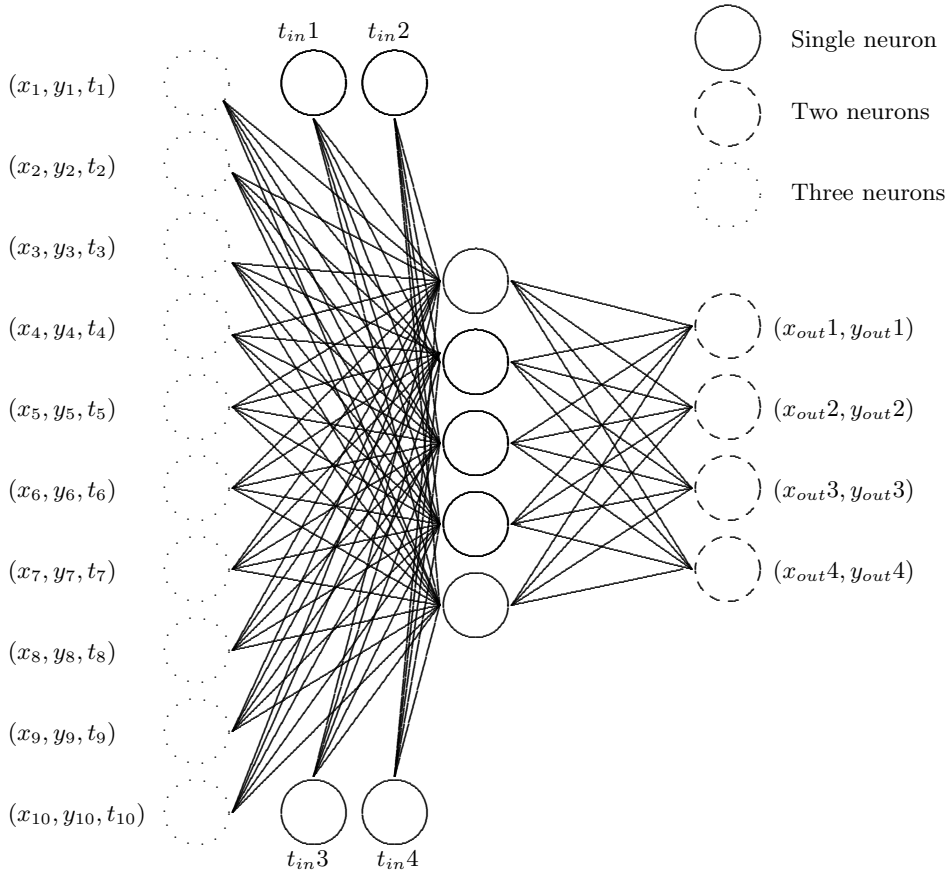


Figure 9.9: Artificial Neural Network for targeting with 10 triples of input gates, 4 single input gates and 4 pairs of output gates.

A challenge while training the ANN is estimating the number of neurons in the hidden layer. The more hidden neurons the more weights we have to update during training. The method of trial and error, is a way to find the best number of neurons which is needed for the network to be able to describe the optimal solution.

To determine the number of hidden neurons needed to get the best classification in the ANN we can construct several ANNs with different numbers of hidden neurons, N. Then train each network, and use a validation set to determine the classification

score, S, (percent of correct classified cases) of each network. Thereafter create a (N, S)-plot as shown on Figure 9.10 and select the number of hidden neurons as the number with the highest classification score, S.

If the number of hidden neurons are to low, then will the ANN not be able the learn anything, and if the number of hidden neurons are to high, then there are a risk, that the ANN begins to memorize the trainings data, which will result in good classification of the trainings data but less good classification of the validation data.

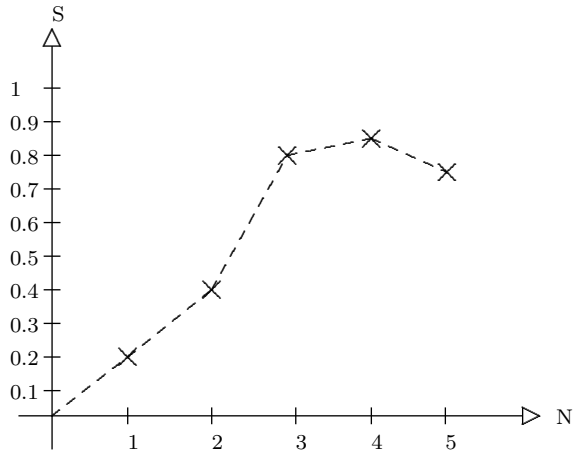


Figure 9.10: Plot to determine the number of hidden neurons needed for optimal classification strength. The plot shows an example of the classification scores for different numbers of hidden neurons.

There have also been proposed different rules-of-thumb, which may be used to determine the number of hidden neurons [Hea]. These can be used to make a decision about how may neurons to start out with before beginning the training of the network.

- The number of hidden neurons should be in the range between the size of the input layer and the size of the output layer.
- The number of hidden neurons should be $\frac{2}{3}$ of the size of the input layer, plus the size of the output layer.
- The number of hidden neurons should be less than twice the size of input layer.

9.2.4 Bayesian Network Targeting

As an alternative to the ANN targeting, we will now design a Bayesian Network intended to perform an equivalent task of predicting the enemies moving patterns.

- Task T: Predict the most likely position of the enemy at a given time.
- Performance measure P: Comparison of how accurate the prediction was in relation to the actual position of the enemy.
- Training experience E: Playing against different robots with different movement patterns.

The idea is to divide the battlefield into a finite number of x and y zones, where each zone is an interval of pixels. Then make two Bayesian Networks, one that can predict the most likely position of the enemy robot in the x-zone, and an equivalent one for the y-zone predicting y-zone position. The situation is illustrated on Figure 9.11.

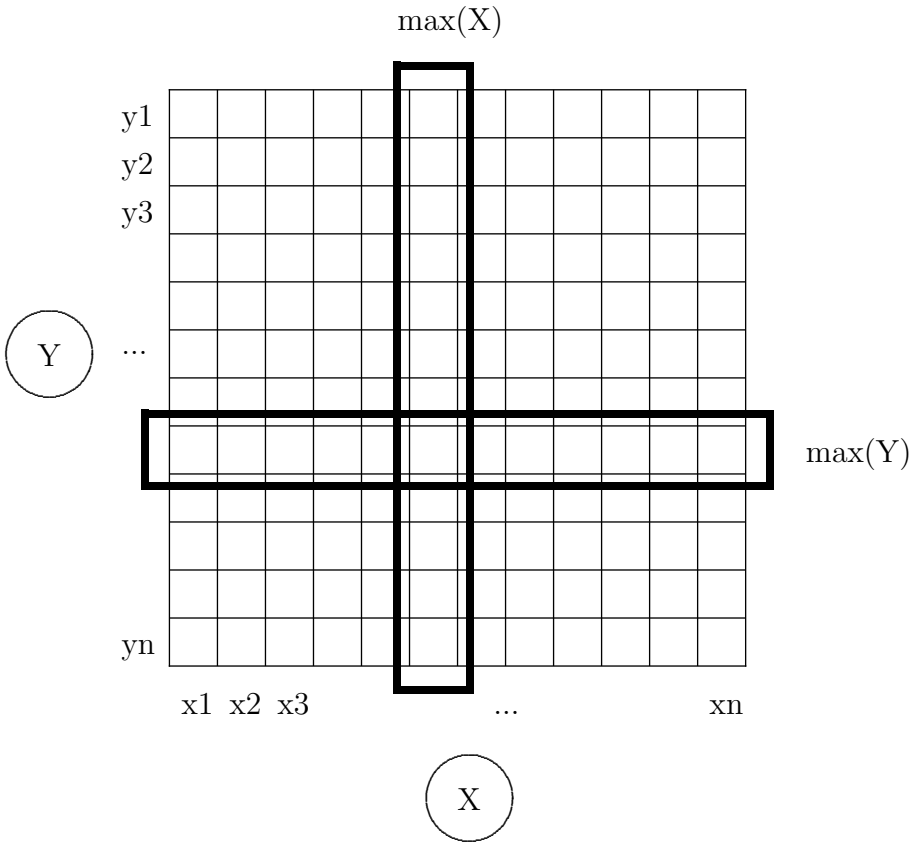


Figure 9.11: Division of the battlefield.

To retrieve the prediction of the enemy position from the network, we simply find the state in X and Y, that has the highest probability. This will then be the target zone in which the enemy is likely to be. It is the square shown on Figure 9.11 where the two outlined rectangles intersect.

Having the target zone, we can choose a suitable algorithm to determine where to shoot inside this target zone. A simple option could be to fire towards the center of the zone. To increase accuracy of the prediction, the number of zones could be increased, but it comes at the cost of a larger probability table, which again increases the cost of propagation through the network. We therefore have to find a ratio of prediction accuracy in relation to propagation time. The method of trial and error could be applied here.

The states of the nodes in the network for predicting x-zone position is $X = \{x_1, x_2, \cdots, x_n\}$ and for the y-zone prediction $Y = \{y_1, y_2, \cdots, y_n\}$. The two networks are shown on Figure 9.12.

Figure 9.12: The two networks for predicting x and y zone position. 1..n indicates that there are n temporal links, each going one step further into the future than the previous.

Ideally the causal relations of the network will be that the position of the robot will depend on all previous positions of the robot, making it a network, that has a very large number of nodes, where each node is causally connected to future nodes, making it a network, which does not obey the Markov property (knowing the present makes the future independent of the past). This has the effect, that the state table which the network describe will be large. If there are n nodes, and each node has m states, equivalent to dividing the battlefield into m x-zones. The number of entries in the table will be m^n, and remember we have two networks that need to be calculated in order to get a prediction.

Due to the size of the table, we choose to simplify the model. Instead of dividing the entire battlefield, we only divide a small area of the battlefield surrounding the robot, and we only talk about enemy positioning in relation to ourselves. A zone is now an interval of pixels relative to the robots position.

The new scenario is shown on Figure 9.13. This reduces the number of states to five per node, if we choose to divide as shown on Figure 9.13.

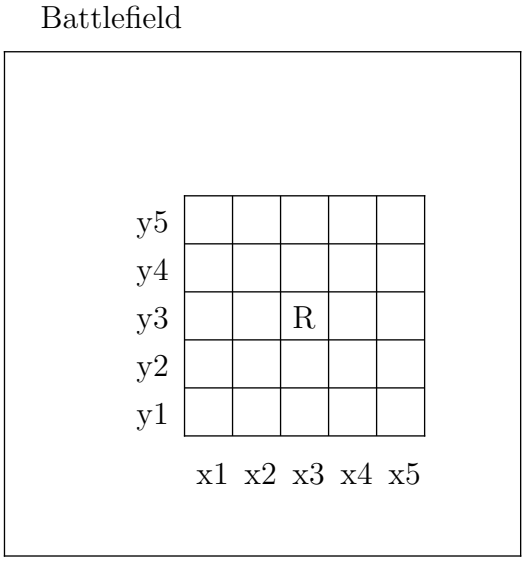


Figure 9.13: The simplified division of the battlefield. R is the robots position on the battlefield.

Even though we have reduced m, we still need to reduce n. Since we now are looking at only parts of the battlefield, the enemy can escape from our grid, meaning,

that it would not make sense to predict far out into the future, because the enemy probably would not be within the grid anymore. Therefore we can reduce the number of nodes. Lets say we only want 6 nodes, making 3 nodes for observation and 3 nodes for prediction. This will make $5^6 = 15625$ entries in the table. We can reduce this further by reducing the number of temporal links, meaning that we reduce the number of nodes, that influence the future. For instance we can pretend that only the last three positions have an effect on the new position. This would reduce the number of entries in the table to m^4, so if we have 5 states in each node the size of the table would be $5^4 = 625$. The reduced network is shown on Figure 9.14.

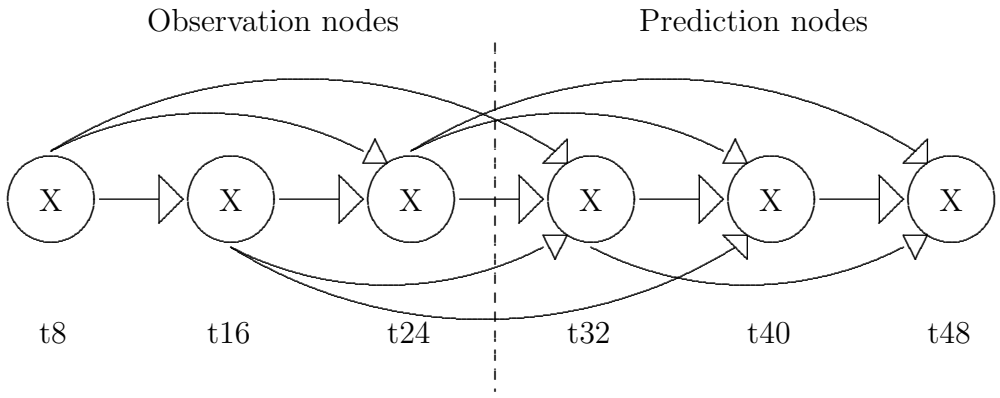


Figure 9.14: The reduced Bayesian network.

The $t8, t16, \cdots, t48$ represent the time (in Robocode ticks) to which the nodes are valid. The nodes are intervalled once every 8th tick. The reason for this is the fact that the radar can turn 360 degrees in 8 ticks, which gives a good probability for getting evidence on a node.

Now we have specified the structure of the network, we have to consider how we are to train the two networks.

9.2.4.1 Learning Probabilities

To learn we need to know how accurate our guess was. This means that we need to observe the actual position of the enemy and compare it to the guess, before we can update.

When the game begins we will not know where the enemy is, so an even distribution is created. If we have m states then each state gets the value $\frac{1}{m}$. The idea is now that we get radar events like `ScannedRobotEvent` then we can put evidence on the observation nodes and thereby getting an estimate of future enemy positions. We can divide the updating into two sets; updating of observation nodes and updating of prediction nodes. Ideally we would not need to update observation nodes, since we can require that we have evidence on all of them, however since we only target on a small bit of the battlefield there is the possibility that the enemy will move out of target range, and we might be short on evidence on an observation node. Therefore we can update these nodes to get a better default guess in the absence of evidence.

We can do updates of variables using fractional updating with fading as described in Section 8.3 on page 52. To avoid overestimation of the sample size we use fading.

The idea is that we multiply every update with a fading factor, this makes older observations assert less influence over time, which is a feature that fits well into the dynamic scheme of our targeting model, remembering that we only target a small area of the battlefield around ourselves.

9.2.5 Targeting Using Predictions

The predictions made by either the ANN or the Bayesian Network in the gun module gives us predictions about the enemies positions to some time t. We will now explain how these predictions can be used to aim at the enemies.

9.2.5.1 Point Targeting

The BN is giving us a square in which the enemy is most likely to be in at three different times in the near future.

When firing against the enemy we will fire against the center of the square in which the enemy most probably will be at time t. This square should also be the one that our robots bullet has the biggest probability of begin in so it can hit the target before it has moved out of the square. We want to aim against the first coming square because we believe that predictions about enemies positions will become more and more inaccurate, the further into the future they predict.

To aim at the center of a square we can use this Equation:

$$v_{toPoint} = 90^o - \arctan(dy/dx), \tag{9.7}$$

where dx and dy is the displacement to the center of the square in relation to our position. This Equation gives us the angle that our gun is supposed to have to point at the center of the square. To adjust our gun to actually point in this direction we turn the gun by

$$v_{pointTargeting} = v_{toPoint} - \text{OurGunHeading}, \tag{9.8}$$

where `OurGunHeading` is the angle in which our gun points in relation to the direction up the screen. This angle is available thought Robocode.

9.2.5.2 Select Square

A bullet with a velocity, B_{vel}, can reach the center of a square $d = \sqrt{dx^2 + dy^2}$ pixels away within the time t if $B_{vel} \cdot t \geq d$ where dx and dy are the displacement to the center for the square from our robot's position. We want to shot against the first predicted square if the bullet can reach that square within 8 ticks else we shot against the next predicted square but only if the bullet can reach that within 16 ticks, otherwise we shot against the next predicted square, if and only if the bullet can reach that square... and so on. If the bullet cannot reach any of the squares before the time for the last predicted square, then we do not shot at all.

9.2.5.3 Linear Targeting

We now consider how to aim at an enemy using the predictions from our Artificial Neural Network. We first consider the simple case where we aim at an enemy moving in a straight line. This situation is illustrated in Figure 9.15. When shooting we assume that the target is moving in a straight line from the time of shooting the bullet, until it hits the target. So the aiming procedure can be reduced to shooting against a specific point laying in front of the target.

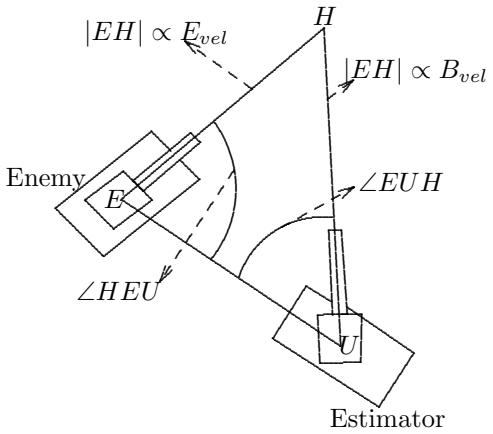


Figure 9.15: Linear targeting example

As it can be seen in Figure 9.15 a triangle can be made with the enemy in the first corner, E, moving along the sides $|EH|$. Our robot is in the other corner, U, firing along the side $|UH|$ opposite to the corner of the enemy. The last corner of the triangle, H, is the point where the bullet will hit the enemy. The length of the sides $|UH|$ and $|EH|$ is proportional to the velocity of the bullet B_{vel} and the enemy E_{vel}, so we can use these velocities instead of the length of $|UH|$ and $|EH|$. The velocities is available through Robocode, so we do not need to calculate the distances $|UH|$ and $|EH|$.

To turn the gun such that it is pointing along $|UH|$ we need to calculate the angle $\angle EUH$. This angle is:

$$\begin{aligned} \angle EUH &= sin^{-1}\left(\frac{sin(\angle HEU)}{B_{vel}} \cdot E_{vel}\right) \text{ where,} \\ \angle HEU &= 180^o - (EnemyHeading - OwnHeading + EnemyBearing) \end{aligned} \tag{9.9}$$

which are all values available in the Robocode engine.

When $\angle EUH$ is known, the targeting is reduced to letting the gun point at the enemy (see Equation 9.8), and then adjust the gun by turning it an extra $\angle EUH$ degrees.

$$v_{linearTargeting} = v_{pointTargeting} + \angle EUH. \tag{9.10}$$

9.2.5.4 Choosing which Line Segment to Aim at.

When we have obtained two or more predictions about the enemies position, we assume that the target is moving in a straight line between the predictions and then

using linear targeting to aim at the enemy on one of these lines. We will fire against the *line segment*, which is in the very near future of where the bullet can reach before the target has done moving in this line segment.

Figure 9.16 illustrates a situation used in the following description of the aiming tactic. We want to shoot at the enemy while it is moving along $|AB|$, if the bullet can reach him before he starts to move along $|BC|$. This is if $B_{vel} \cdot (t_B - t_A) \geq |BU|$. Otherwise we want to aim at the enemy while he is moving along $|BC|$ if the bullet can reach him before he starts to move along $|CD|$ - that is if $B_{vel} \cdot (t_C - t_A) \geq |CU|$ etc.

If the bullet cannot reach the enemy before the enemy has passed by the last predicted point we will not fire against him at all. That is if $B_{vel} \cdot (t_Z - t_A \geq |ZA|$, where Z is the last predicted point.

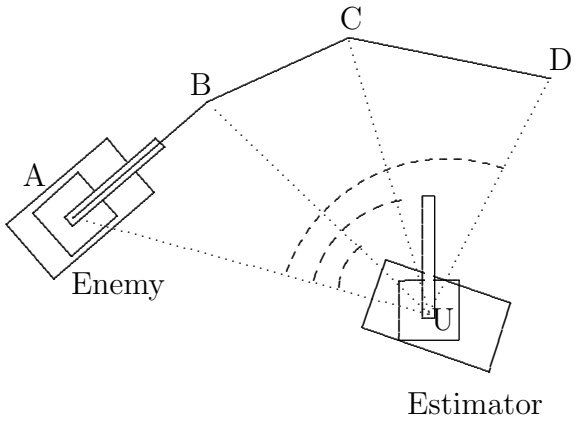


Figure 9.16: Depiction of more than one prediction and how to target in this situation

9.2.5.5 Trace Back Movement

When we have decided which line segment we will aim at we can reuse the Equations from linear targeting. See Figure 9.17 in the following explanation. We pretend that the enemy is coming from a new point (x'_0, y'_0) (maybe) different from the real point (x_0, y_0) that the enemy was at the time t_0, moving forward and passing thought point (x_n, y_n) to the time t_n, and continuing forward through the point (x_{n+1}, y_{n+1}) to the time t_{n+1}. We can then reuse Equation 9.9 to aim at the enemy. We just set the points $E = (x'_0, y'_0)$, $H = (x_{n+1}, y_{n+1})$ and U is still our position.

To calculate the point (x'_0, y'_0) we use the Equation

$$x'_0 = x_n - (t_n - t_0) \cdot v_{mean,x} \tag{9.11}$$

$$y'_0 = y_n - (t_n - t_0) \cdot v_{mean,y} \text{ , where} \tag{9.12}$$

$$v_{mean,x} = \frac{x_{n+1} - x_n}{t_{n+1} - t_n} \text{ and}$$

$$v_{mean,y} = \frac{y_{n+1} - y_n}{t_{n+1} - t_n}.$$

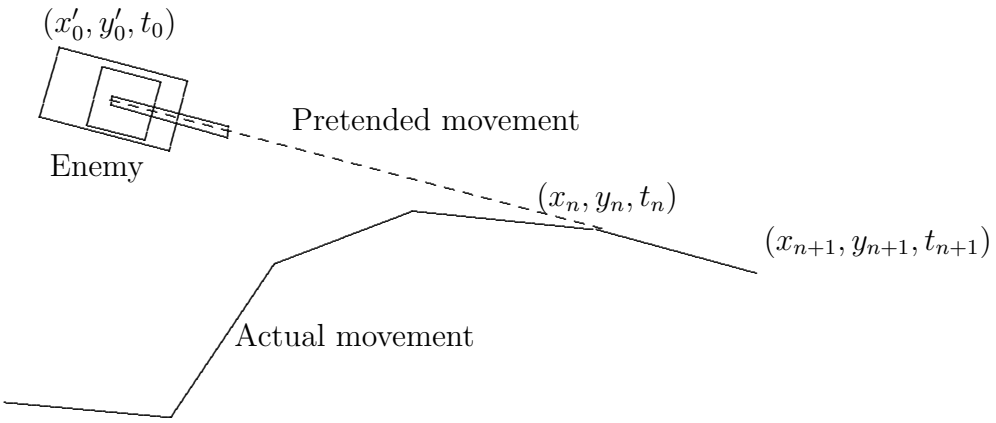


Figure 9.17: Trace Back Movement

9.3 Radar Module

The Radar modules has the task of detecting enemies and communicating it to the `EventDispatcher`, which will send it to the modules which have made a subscription of the `ScannedRobotEvent`.

9.3.1 Reinforcement Learning

This Section will describe a RL strategy which will track an enemy until it dies.

- Task, T: Track an enemy robots movement by scanning it.
- Performance measure, P: Number of scans of the robot per tick.
- Experience, E: Playing against an enemy robot and every time it scans the enemy robot the Q-table is updated.

The strategy goal is to scan an enemy as much as possible. If it does not scan an enemy, then it should extend the scanning area next time it scans. An illustration of how the robot should extend its scanning area can be seen in Figure 9.18.

In this module we can let the time interval be equal to the time ticks in Robocode.

The following variables is used as states for this module. The exact values in the text have all been chosen based on our intuition.

- **Last seen distance**: The distance from the robot to the enemy.

  Values = $\{0-100,\ 100-300,\ 300-600$, 600-1200 and >1200$\}$

- **Last seen enemy direction**: Last calculated direction of the enemy. See Figure 9.8 on page 76 for an illustration of this.

$$\text{Values} = \{(0^o \text{ to } 45^o), (45^o \text{ to } 90^o), (90^o \text{ to } 135^o), (135^o \text{ to } 180^o),$$
$$(0^o \text{ to } -45^o), (-45^o \text{ to } -90^o), (-90^o \text{ to } -135^o), (-135^o \text{ to } -180^o)\}$$

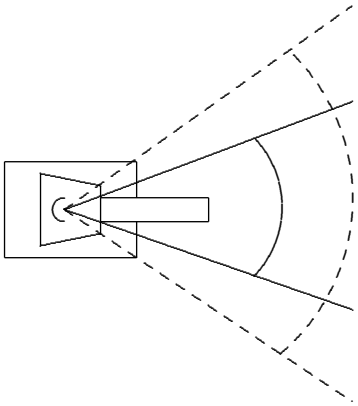


Figure 9.18: Shows how a robot could extend its radar scanning area. The dotted lines are the extended sight.

- **Angle relative to opponent**: The radar angle to the last seen enemy robot, in relation to the front of the vehicle. See Figure 9.19 for an illustration of the angle relative to an enemy robot.

  Values = $\{(0^o \text{ to } 45^o), (45^o \text{ to } 90^o), (90^o \text{ to } 135^o), (135^o \text{ to } 180^o), (0^o \text{ to } -45^o), (-45^o \text{ to } -90^o), (-90^o \text{ to } -135^o)$ and $(-135^o \text{ to } -180^o)\}$

- **Seen enemy**: If an enemy robot has been spotted in this state.

  Values = {true, false}

And the actions for this module is

- **Actions**: The values are chosen to fit into the time interval, since the radar can scan 45^o in one time tick the values are:

  Values = {setTurnRadarRight(45), setTurnRadarLeft(45)}

A graph of the RL radar module as described can be seen at Figure 9.20. With all these different configurations we can construct 160 different states.

$$(\textbf{distance}) \cdot (\textbf{direction}) \cdot (\textbf{Angle}) \cdot (\textbf{Seen enemy}) \Downarrow$$

$$5 \cdot 8 \cdot 8 \cdot 2 = 640$$

Considering the different actions which can be chosen the 640 different states becomes 1280 state-action pairs in the Q table.

$$640 \cdot \textbf{Actions}(2) = 1280 \text{ Table entries}$$

The reward function, $r(s, a)$, we have chosen for the radar module is as follows:

$$r(s,a) = \begin{cases} \textbf{0} \text{ if no enemy is scanned} \\ \textbf{1} \text{ if scanned an enemy robot} \\ \textbf{2} \text{ if scanning the same enemy robot two ticks in a row} \end{cases} \tag{9.13}$$

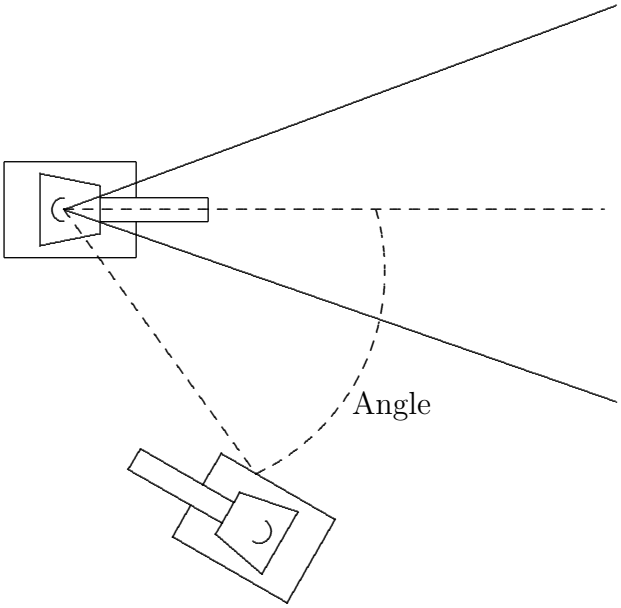


Figure 9.19: Shows an example of the angle relative to last seen enemy robot

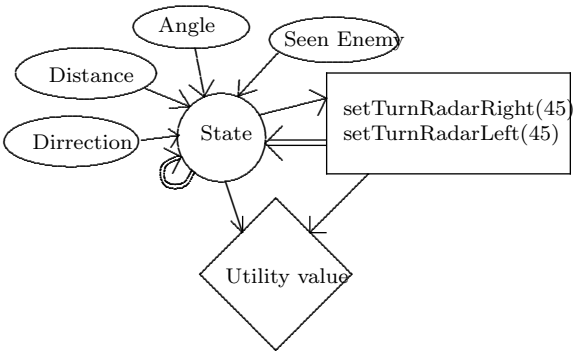


Figure 9.20: Shows a graph of the RL radar module

This RL module can be trained with Algorithm 8.1. α and γ can be found by trial-and-error. The values can e.g. be $\alpha = 0.1$ and $\gamma = 0.5$.

In this module we will use the ε-greedy policy to select actions, because this policy will favor the best action most. In contrast to the other modules using RL we cannot confuse the enemies by moving the radar more unpredictable, which was our argument for using Equation 8.9 in the modules for moving and selecting firepower (see Section 9.2.2).

During training we will set ε higher than during an important match, because during training we want to explore the result of different actions more than during an important match, where we just want the best radar scanning policy. In training we can e.g. set $\varepsilon = 0.1$ and during a match we can e.g. set $\varepsilon = 0.01$.

Summary

In this rather large Chapter we introduced the design of the deliberative modules that constitute the ESTIMATOR robot. There were two `Vehicle` modules, which used Genetic Algorithms and Reinforcement Learning. There were four different submodules for the `Gun` module, which was split into two responsibilities: Prediction and bullet energy choice. Prediction was designed using Artificial Neural Networks and Bayesian Networks, while bullet energy choice was designed using Reinforcement Learning and Decision Graphs. We also explained how you can use predictions to target with. Finally we presented the Reinforcement Learning module for the `Radar` module.

10 Design Conclusion

In this Part of the report we started in Chapter 6, by showing how we would prioritize the design. The most important parts were that we were not going to focus on cooperation between agents, but primarily on machine learning and the skill of adapting to the environment.

In Chapter 7 we presented our designed architecture, which should allow us to change the internals of the robot as we wish. We also showed how the robot is split into different areas of responsibility, so that there is no need to sort the actions produced by the deliberative modules.

Chapter 8 offered the basis for understanding the different machine learning techniques used to design the modules in Chapter 9.

Chapter 9 described the different designs for the modules constituting the ESTIMATOR robot. The `Vehicle` module had the responsibility of moving the vehicle part of the robot around the battlefield, dodging bullets and ramming opponents. For this module we designed two modules: One that used Genetic Algorithms to evolve robots and one that used Reinforcement Learning to choose the next movement.

The `Gun` module however was a bit more complicated, so this was split up into two different submodules: One that chooses when to shoot and how much energy to put into the bullet, as well as a submodule that predicted where the enemy would be at a given time. The first submodule used Bayesian Networks and Decision Graphs to decide when to shoot, while the second submodule use Reinforcement Learning to decide the same thing.

The prediction part of the `Gun` module was also split up into two different submodules. The first used Artificial Neural Networks to calculate where an enemy would be at a given time. The other submodule used Bayesian Networks to do the same thing.

Finally the `Radar` module used Reinforcement Learning to keep the radar fixed at one opponent at a time.

All in all we have designed an expert system as a Robocode robot, that is able to choose which actions to perform and adapt to its environment. Its architecture allows us to change the different machine learning modules without any problems. Whether the modules are well designed and actually able to defeat opponents, will remain uncertain until we have tested and evaluated them, which we will do in the next Part.

Part III

Evaluation

Introduction

This Part of the report documents the evaluation of our implemented modules, for the ESTIMATOR robot.

Before the actual evaluation we will provide an overview of the implementational status in Chapter 11 as some of the designed modules are not implemented.

In Chapter 12 we will start the evaulation by explaining the basis for our tests and the associated criteria on which we will evaluate these tests. After this the actual evaluation of our modules will take place, where we will first explain how the test for every module is performed. Furthermore we will examine how the testing environment will be organized, what output we would expect from each tests, the actual results, and the graphs that visualizes these outputs. At the end of every module test we will end up with a conclusion evaluating the results from the test and propose suggestions for improvements, where we find that improvements could be needed.

Finally in Chapter 13 we will document an overall test of ESTIMATOR. This will be done by combining the different modules and test how well they operate together. ESTIMATOR will be lined up in a match against, what we deem to be a suitable opponent.

11 Implementation Overview

This Chapter will describe the current status of the implementation in our project, the modules which have been implemented and which modules have not been implemented and why the modules have not been implemented.

The following modules have been fully implemented:

- **RL vehicle**: This module uses no external tools.
- **GA vehicle**: This module uses no external tools.
- **RL radar**: This module uses no external tools.
- **DG energy**: This module uses Hugin (see Section 11.1 on the following page).
- **ANN targeting**: This module is implemented without any external tools and with Joone (see Section 11.2 on the next page).

We have not implemented the BN targeting module, the RL energy choice module nor the communication module. None of the implemented modules have been designed to communicate with other robots, so the implementation of the communication module has been prioritized low, leading to no implementation at all. Even though we do not need it in our current implementation the communication module is still important in the overall design, thus giving us the ability to extend modules with the ability to share information between robots and use the other robots to get extra sensory input. The communication feature would probably improve the chance of winning in Robocode, but this feature is not vital for the ability to learn using machine learning which is our focus and therefore it is not needed in our implementation.

The RL energy choice module and the BN targeting module has not been implemented do to lack of time and with the BN targeting some problems with the representation of the network occurred.

11.1 Hugin

Hugin is a tool which offers a GUI based interface and a Java Application Programming Interface (API) to its C engine, allowing for easy building and training of bayesian network and decision graphs. We use the API to program and adapt the energy decision graph and inspect it using the Hugin GUI. More information on Hugin can be found at [A/S].

11.2 Joone

Joone (Java Object Oriented Neural Engine) is an open source project providing a Neural Network framework to create, train and test Artificial Neural Networks. It provides an engine and a GUI editor to perform various common task related to Artificial Neural Networks, but we have only used the engine in our project. More information on Joone can be found at [Joo].

Our intentions was to implement the ANN to predict movements using Joone, but as we implemented our design with Joone we encountered some implementation problems. As a part of the debug process we implemented a complete solution handcoded without Joone using the backpropagation algorithm from Listing 8.2 on page 57.

Summary

In this Chapter an overview of the status of our implementation of the designed modules was presented. We also briefly introduced the two external libraries, that we have used in some of the modules.

12 Learning Capabilities

In this Chapter we will test the implemented modules thoroughly to find out whether the chosen methods are suitable for the field in which they have been used. We perform a test of each module and by the end of each test we will conclude upon these results and suggest improvements.

The success criteria for these tests will be if the modules are able to improve their performance through experience.

12.1 Vehicle using Genetic Algorithms

This Section describes how we have tested our vehicle module implemented using GA, hereafter denoted as `GAVehicle`.

The purpose of this test is to see whether the module improves through generations.

12.1.1 Test Method

When testing our `GAVehicle` we wanted to let the robot act in an environment exactly like the environment it was designed for; a battlefield of $2000 \cdot 2000$ pixels, a member on a team consisting of five robots and with an opponent team of five robots. The opponent team was the **GimpTeam**, downloaded from the list of teams on [rob]. A reason for testing as a team is to get results at a much higher rate than if we were only testing with a single robot. All teammates are part of the same generation and thereby all feeding the population with scores to use for the fitness function, thus the traversed number of generations is increased.

We needed a lot of finalized rounds between the two teams so we could calculate the fitnesses for the hypotheses in our generations. Therefore we just let our two teams fight through many rounds without looking at the final score, because we only wanted the score from for each round to increase throughout the generations.

Whenever a new population was created we saved the combined fitness for the hypotheses, hereafter mentioned as the united fitness, this is the way we compare

how well the different generations behave.

We have implemented Tournament Selection, this method is implemented as discussed in Section 8.4 on page 58 giving 40% of the hypotheses to the new population.

Using this test method we would expect the average progress of the united fitness to grow for the first generations and then slowly level off when it moves towards a maximum for the domain space.

We will compare the result with a maximum at 38.000 which is calculated as a team of 5 robots never getting hit and always hitting the opponent team. This theoretical maximum is given by:

$$\begin{aligned} \text{Bonus for enemy dead (ed)} &= 50 \\ \text{Bonus for win (w)} &= 10 \\ \text{Energy of team (e)} &= 600 \\ \text{Team size (s)} &= 5 \\ \text{Size of population (p)} &= 100 \end{aligned}$$

$$\begin{aligned} \text{Maximum score} &= \left((ed \cdot s) + w + \frac{e}{s}\right) \cdot p \\ &= \left((50 \cdot 5) + 10 + \frac{600}{5}\right) \cdot 100 \\ &= 38,000 \end{aligned}$$

We cannot expect that the `GAVehicle` will ever be good at ramming, since there are no bit sequences which will result in a policy where the robot always will ram the enemies, thus we calculates with a maximum at $38,000$ which is the maximum score when we neglect the score from the `HitRobotEvent`.

During the test we will use a radar which are constantly rotating in the same direction, and a gun using point targeting[1].

12.1.2 Results

As shown on Figure 12.1 on the facing page the `GAVehicle` keeps a steady learning rate after 35 generations. Until the 35th generation the generation fitness is increasing.

12.1.3 Interpretation of Results

After 35 generations we have a population, which is better and more adapted to the environment than the initial population. This is not surprising as the first was created totally random. Our population seems to stagnate about 29.300. This can be because we end up in a local maximum instead of heading forward to the total maximum. But cf. the theory of GA it is not likely that GA will reach a local maximum, due to crossover and mutation. If the fitness evaluation is stagnating,

[1] Point targeting is to fire in the direction straight against the enemy.

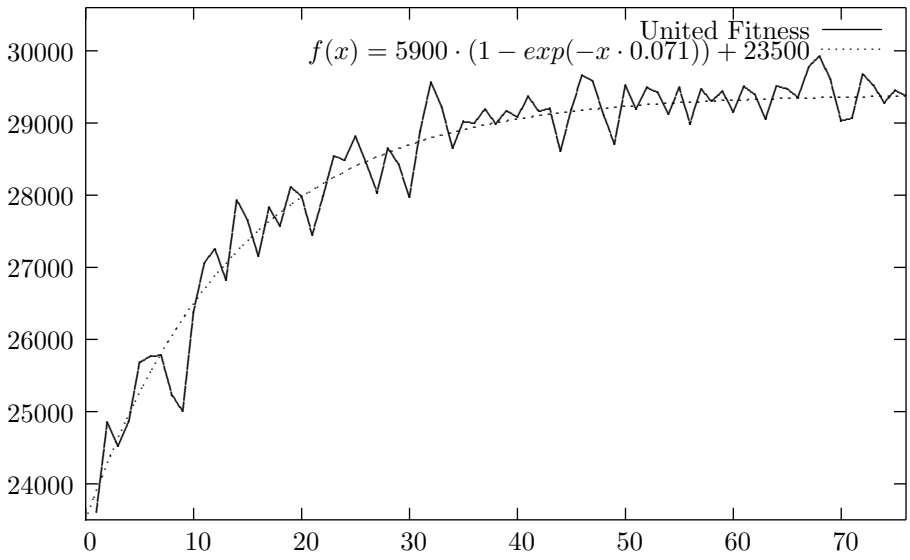


Figure 12.1: GA Vehicle. The graph shows the evaluation of the united fitness for the generations.

mutation can produce new hypotheses which can reactivate the evaluation. Instead it can be the case that 29, 400 is close to the actual maximum fitness for our hypotheses, e.g. if our design of the bit sequence or the action composition is suboptimal then a generation with only the most fit hypotheses will never reach or get near 38, 000 in fitness.

One way to find out if we have reached a local maximum or if 29, 400 is the maximum for our design, is to let the evaluation continue in more generations, and see if mutation can create a population with a higher fitness. This takes a long time to evaluate the fitness of a generation since each hypotheses need to be tested in 20 rounds. So with 100 hypotheses and 20 rounds needed for each hypothesis and 5 hypotheses tested per round, one generation requires $\frac{100 \cdot 20}{5} = 400$ Robocode rounds.

When looking at the module and comparing the first and later generations, our assessment of the movement policies for the robots is that the vehicle sophisticates its movements e.g. by minimizing how often it gets hit by bullets, by beginning to drive in circles when it is being hit by bullets. This often prevents the robot from being hit by a cascade of bullets and looks kind of clever compared to the starting generation. Though this might seem well it is only our judgement, when watching the module "in action".

12.1.4 Conclusion

When the algorithm has been running for some time it will consist of a well performing generation which will continue to sophisticate over time. Though it performs well against **GimpTeam** it is not given that it will perform well against a team with very different combat strategies and because of the long evolving time to get a well performing `GAVehicle` it will be hard to get a generation which will perform well for all kinds of enemies, thus making this vehicle module a bad decision for

combat against very different robots.

12.2 Vehicle Movement - Reinforcement Learning

This Section will describe how the `Vehicle` module implemented using Reinforcement Learning, hereafter denoted as `RLVehicle`, has been tested and what the outcome of the test was. The purpose of this tests is to determine if the `RLVehicle` module has improved its performance over time, during battles.

12.2.1 Test Method

As described in Section 9.1.2 on page 66 different reward functions have been made for the `RLVehicle` module. To make sure we are only testing the RL `Vehicle` module, we have kept different factors constant during most of the tests. The factors we have kept constant during the test can all be seen below.

- The radar module have been rotating constantly during all test.
- Reward function which did not depend on a gun module have all been tested with a gun which did not fire.
- Reward function which were depending on a gun module firing at opponent, have all been using a non machine learning gun using point targeting.
- Reward function which do not receive negative rewards when being hit by opponent bullets have been tested with opponents which did not fire at enemies.

The reason why every reward function has not been tested with the exact same conditions is because some reward functions are very simple, and because they do not rely on other modules, it was possible to get better tests without involvement from other modules.

For instance by testing the `RLVehicle` module, which uses the Ram-reward function (see Section 9.1 on page 70) with a gun which fires at the opponents, would only disturb the learning of ramming opponent because the gun would probably most of the time kill the opponent and thereby not give the `RLVehicle` module a chance to learn to ram opponents.

To speed up the learning rate for `RLVehicle`, enemy robots have been designed specially for every test. For instance to test the `RLVehicle` module with the "Enemy Avoidance-function" (see Figure 9.1 on page 71) an enemy robot, which tries to ram its opponent has been used. By doing this some of the measuring-noise have been filtered out. For instance with the `RLVehicle` module using the "Enemy Avoidance-function", this will minimize the outcome where the two robots never meet, because they have been placed too far from each other. If the robots had just kept away from each other `RLVehicle` would incorrectly think it had done great during the battle because it would not have received any negative rewards, even though it really was the opponent which had done bad (for our test case) during the battle, because it did not manage to be near or ram our robot.

12.2.2 Ramming-Reward Function

As described in the design part, this module has been designed with a reward function with the purpose of learning to ram enemy robots.

Besides this `RlVehicle` module this test was performed with a gun which did not shoot and a radar that just rotates constantly. The opponent for the test was `SittingDuck`:

SittingDuck (no shooting): This robot do not move at all, it just stand still at the location in which the Robocode engine places it.

12.2.2.1 Results

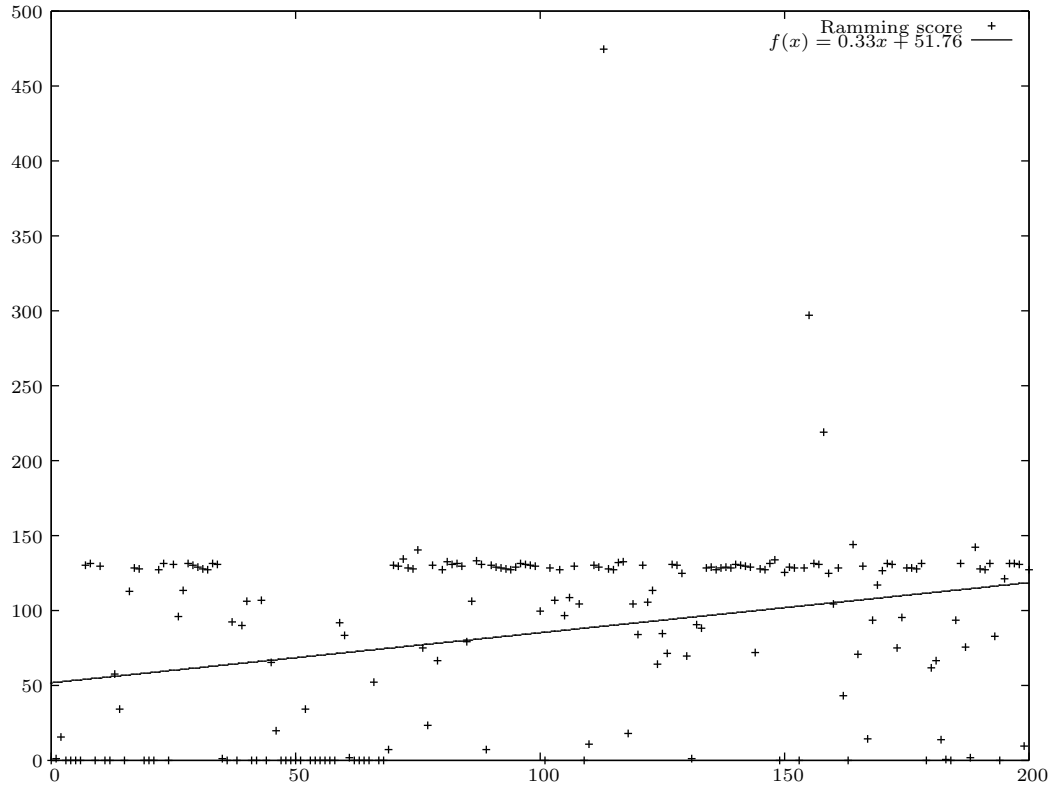


Figure 12.2: Plot of 200 rounds (x-axis) and "Ramming-score" (y-axis) $f(x)$ is a linear regression of the ramming score.)

The "Ramming-score" is the score received each round, calculated with use of the Ramming-reward function (see Figure 9.1 on page 70).

12.2.2.2 Interpretation of Results

The reason why Figure 12.2 only shows 200 rounds is that at this point the graph converges to a constant value of approximately 130, which can be seen in Figure 12.3 on the following page and Figure 12.4 on page 105.

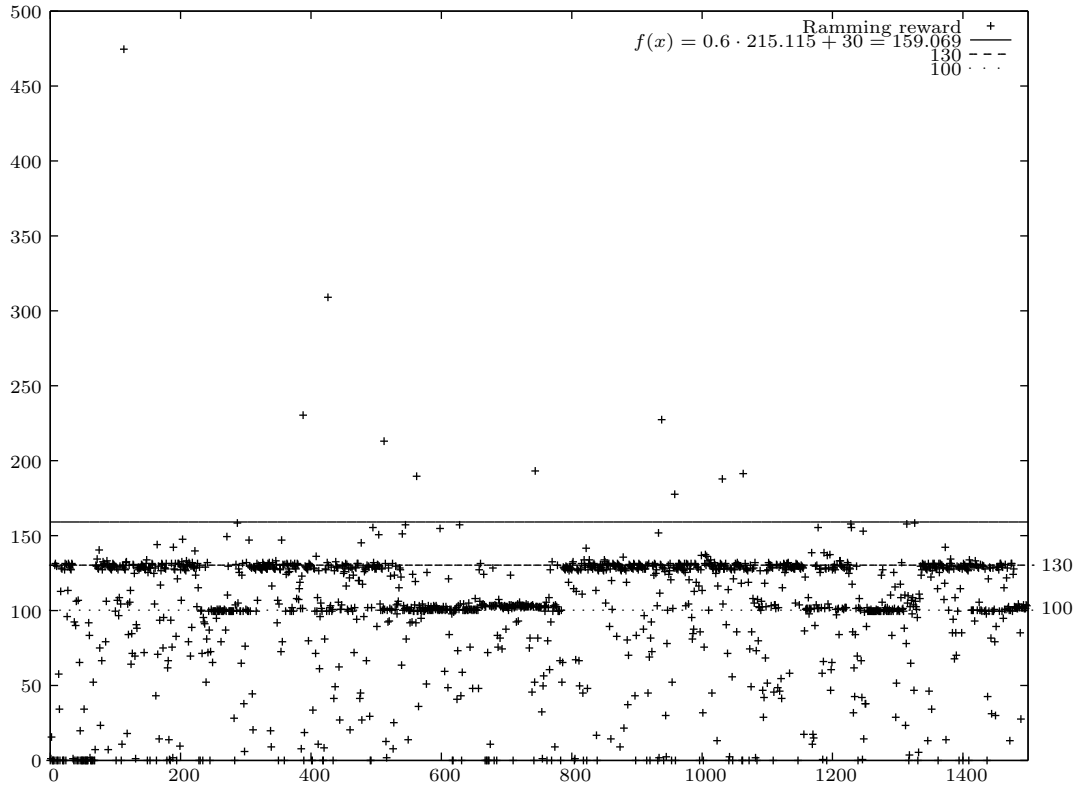


Figure 12.3: Plot of 1500 rounds (x-axis) and "Ramming-score" (y-axis) $f(x)$ is the practical point of converting.

On Figure 12.3 $f(x)$ is the practical point of converting, based on all rounds. It have been calculated with use of the reward function for "Ramming":

$$\begin{aligned} \text{Energy loss by ramming (el)} &= 0.6 \\ \text{Number of average rounds (ar)} &= 1720.92 \\ \text{Bonus reward (br)} &= 30 \\ \text{Frequency of ticks to act on (t)} &= 8 \end{aligned} \tag{12.1}$$

$$\begin{aligned} \text{el} \cdot \frac{\text{ar}}{t} + \text{br} &= \text{Practical point of convergence} \\ 0.6 \cdot \frac{1720.92}{8} + 30 &= 159.069 \end{aligned}$$

The reason why it is not the theoretical maximum, is that every now and then a round which is 2-3 times longer than an average round occurs. Normally the Robocode engine starts to punish all robots on the battlefield if they do not execute any actions. If `RLVehicle` have managed to do some action right before the timeout, the Robocode timeout will be reset. If this keeps on happening the round will be longer than if Robocode had punished the robots on battlefield.

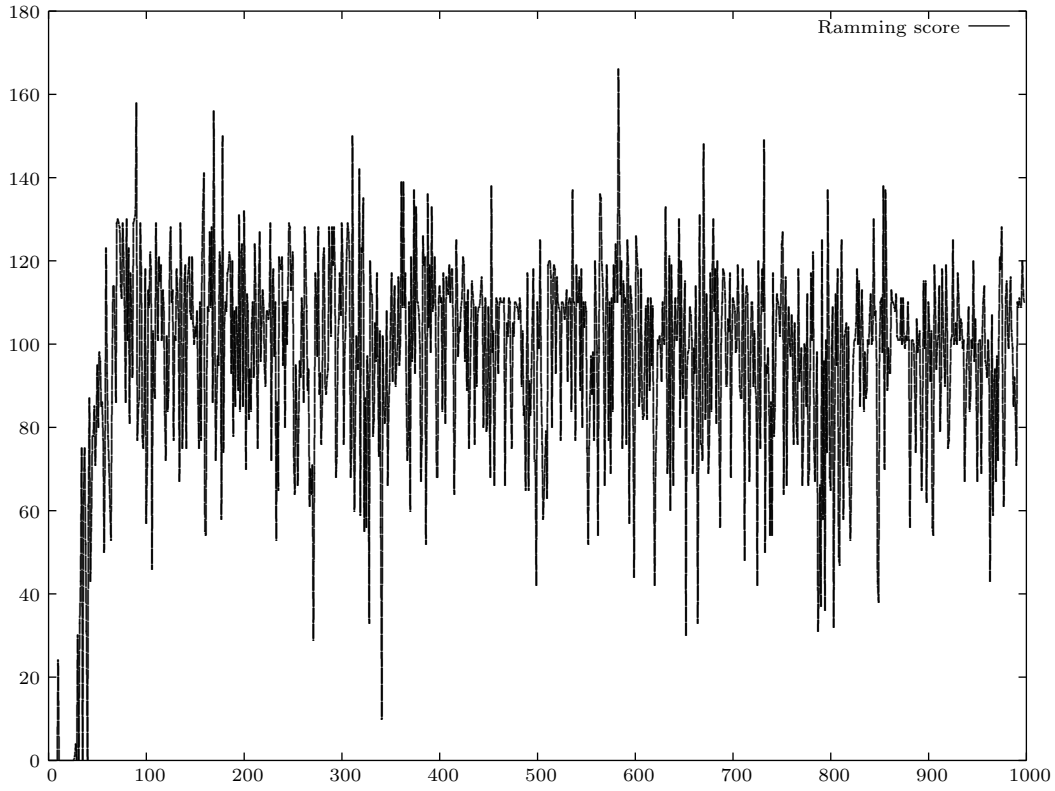


Figure 12.4: Plot of 1000 rounds (x-axis) and average score of 10 `RLVehicle` using "Ramming-score" (y-axis)

Figure 12.4 shows the average score of 10 times 1000 matches against SittingDuck and it shows that the tendency from Figure 12.2 on page 103 and Figure 12.3 on the preceding page is not just lucky coincidence. By doing this some of the measuring noises disappear. However these kind of tests takes much longer time than the normal tests, while it is a little harder to see the tendency on single robot plots, it is still possible, leading us to using only a single robot plots through the rest of the test.

By looking at Figure 12.2 on page 103 it is clear that `RLVehicle` have improved its performance during the 200 rounds. By looking at the first 70 rounds almost every result have a value of 0 and during the rest of the rounds results with a value of 0 still occur, but they occur more and more rarely, which implies that `RLVehicle` has improved its performance.

The reason why the learning happens so sudden must be due to the fact the when `RLVehicle` hits the opponent the first time by random it receives an immediate reward, which results in the first high Q-table value therefore, this action will also be the next chosen action (because of the exploration `RLVehicle` is forced to do it is not given that it will choose this action) and by choosing the same action again it will receive an immediate reward again. By continuing this behavior `RLVehicle` will, when the round is over, have a very high Q-table value for the specific state-action pair, and therefore trying to turn back to this state during the next rounds.

As Figure 12.3 on page 104 shows, `RLVehicle` has two different values which repeat frequently, namely the values around 130 and 100. The reason why these

values repeats so often is because they are, in most of the rounds, the maximum amount of point the robot can receive. The reason why there is two values is because, `RLVehicle` gets a bonus of 30 points for killing its opponent. Because the Robocode engine send out `RobotHitEvent` even though the robots on the battlefield have not yet touched each other, but only standing close to each other. Because of this `RLVehicle` some times learn just to stand beside the opponent and receive rewards, instead of ramming the opponent, but when the timeout in Robocode is reached, both of the robots slowly looses energy and at some point one of the robots will die. Because it was the Robocode engine and not `RLVehicle` which killed the opponent, `RLVehicle` will not receive the final reward of 30 points, which results in the value around 100.

The reason why this value lies beneath 100 must be because when a battle starts, the robots is placed randomly on the battlefield and therefore `RLVehicle` first has to locate the opponent and drive the distance between them to ram him, before getting an reward. Another reason that `RLVehicle`s score lies beneath $f(x)$ is also that `RLVehicle` once in a while do exploration, which distracts `RLVehicle` from receiving rewards, because it already were in an optimal position.

Even though `RLVehicle` clearly learns during the battles some values still remain around 0-50 points and some of these scores seems to occur when `RLVehicle` gets to close to a wall. When this happens the `ReactiveLayer` takes over and stops the action the `RLVehicle` module is trying to execute. This, however, seems to confuse `RLVehicle`, which after being saved by the `ReactiveLayer`, either tries to ram the wall again or just gets off course for a while. This error could indicate an small implementation or design flaw with the `ReactiveLayer`, which purpose is to save `RLVehicle` from the wall and bring it back to the former state, however the possibility that the `ReactiveLayer` takes to many turns, and thereby brings the `RLVehicle` module back to an old state which may have changed, leading `RLVehicle` off course.

12.2.3 Avoidance-Reward Function

This module has been designed with a reward function which should give the robot the possibility to learn to avoid enemy robots on the battlefield (See Figure 9.1 on page 71).

Beside the `RLVehicle` module this test were performed with a gun which do not shoot, and a radar just rotating constantly. The opponent for the test were `RamBot` (which does not shoot).

`RamBot:` This opponent is at all time trying to kill opponent robots by ramming them.

12.2.3.1 Results

The "Enemy Avoidance-score" is the score received in each round, calculated with use of the Enemy "Avoidance-score-function", see Figure 9.1 on page 71.

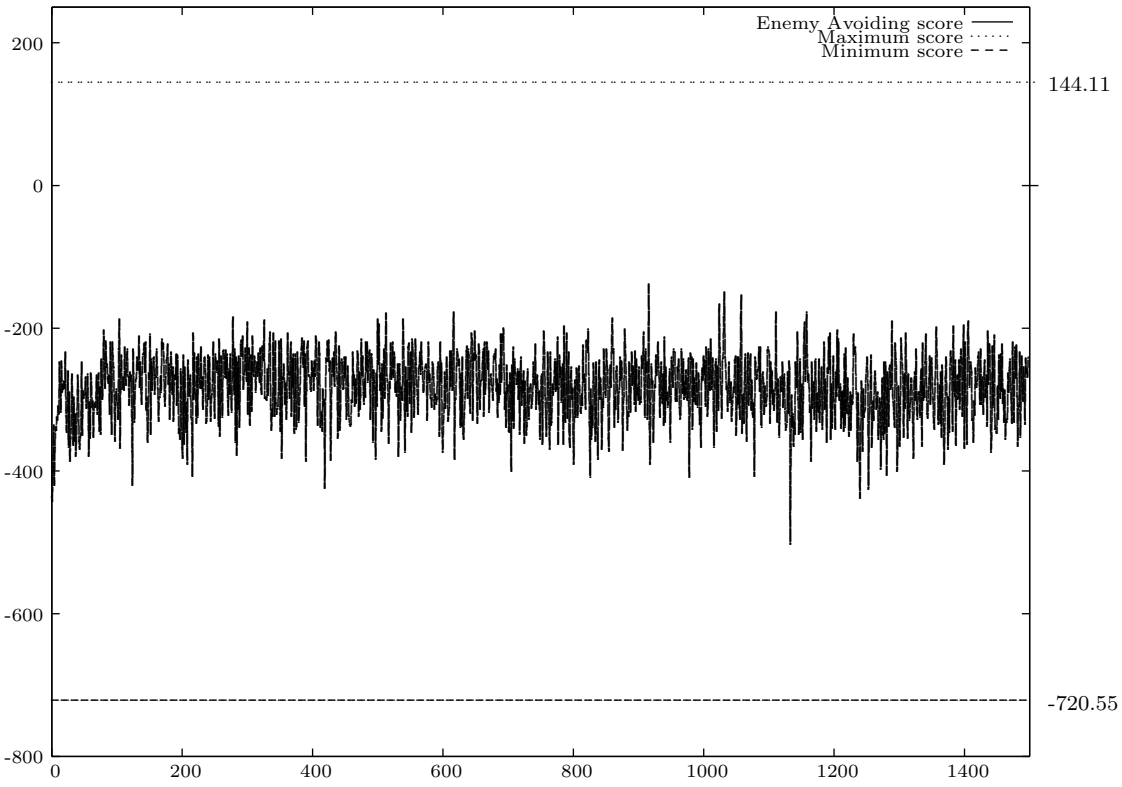


Figure 12.5: Plot of 1500 rounds (x-axis) and "Enemy Avoiding-score" score (y-axis) for the test against `RamBot`.

12.2.3.2 Interpretation of Results

The reason Figure 12.6 on the following page only shows 150 round is due to the fact that after 150 rounds the graph seems to start looping in almost the same pattern.

On Figure 12.5 the practical maximum possible value and the lowest possible score `RLVehicle` reachable is marked. These values have been calculated based on the average length of a round for all the matches.

The maximum and minimum have been calculated with use of the function for "Enemy Avoidance-rewards"

$$\begin{aligned} \text{Average length of round (lr)} &= 72,055 \\ \text{Distance more than 200 pixels (mp)} &= 2 \\ \text{Distance less than 50 pixels (lp)} &= -10 \end{aligned}$$

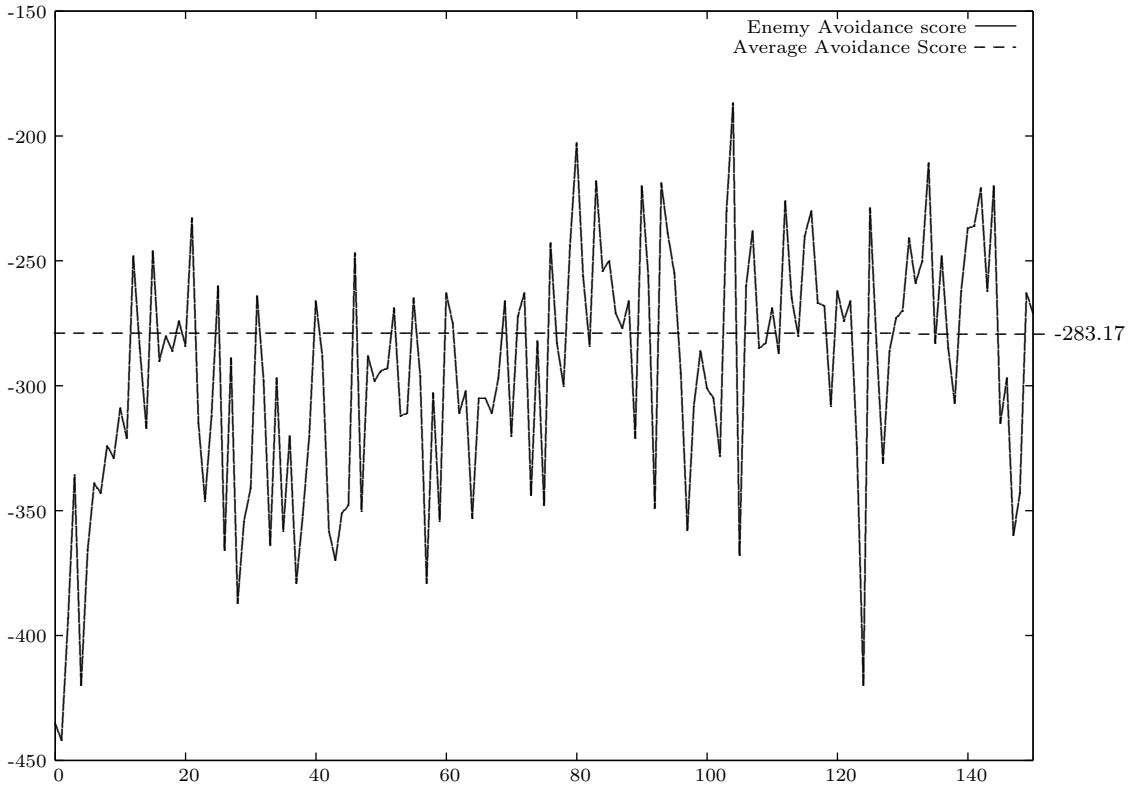


Figure 12.6: Plot of 200 rounds (x-axis) and "Enemy Avoiding-score" score (y-axis) for the test against `RamBot`.

$$\begin{aligned} \text{Maximum} &= lr \cdot mp \\ &= 72,055 \cdot 2 \\ &= 144.11 \\ \text{Minimum} &= lr \cdot lp \\ &= 72,055 \cdot -10 \\ &= -720.55 \end{aligned}$$

To reach the maximum reward `RLVehicle` would have to keep a distance of more than 200 pixels from its opponent at all time during a match. To reach a score close to the minimum limit `RLVehicle` would have to keep a distance of less than 50 pixels from its opponent during a match.

The first 150 rounds it shows that `RLVehicle` have a starting value laying around -440 which during the first 100 rounds reach the value of -280, which is close to the average value for the whole plot. However, when this value has been reached it do not converge to -283.17 very clearly, the reason for this and why the score do not have a higher value can be discovered by looking at the two robots during a match on the battlefield. When watching the two robots it seems clearly that `RLVehicle` tries to drive away, when `RamBot` gets too close. However at some point in the chase

RLVehicle will hit a wall and be caught by RamBot and killed. The reason RLVehicle do not keep away from the walls is a design flaw, which we first thought off when we first time observed the two opponents battling. The reason is that RLVehicle do not know anything about walls, because they do not exist in RLVehicles possible state configuration. The reason we did not designed this into the module was that we thought that the ReactiveLayer would save RLVehicle from ramming walls, which it does, however the ReactiveLayer only works as an invisible wall which RLVehicle can not drive through. And because RLVehicle do not know that the ReactiveLayer have taken control, it will not change state, and resulting that the next action choosen will be the same as last time, which makes RLVehicle keep trying to penetrate the invisible wall until it changes state because the opponents has moved, or the opponent has rammed RLVehicle enough times to unlearn what it else had learned to do in this specific state.

12.2.4 Bullet Avoidance-Reward Function

This module has been made with use of the Bullet Avoidance-function, and should give the robots the ability to dodge enemy bullets (See Figure 9.1 on page 71),

Besides the RLVehicle module this test were performed with a gun which do not shoot, and a radar just rotating constantly. The opponent for the test were Crazy (shooting).

Crazy: This opponent is moving in a deterministic pattern resembling a flower with four leafs (See Figure 12.7). When an enemy is in front of Crazy it will fire a bullet.

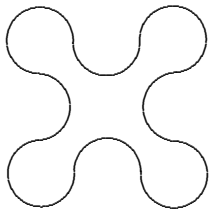


Figure 12.7: The movement pattern of Crazy

12.2.4.1 Results

The "Bullet Avoidance-score" is the score received in each round, calculated with use of the "Bullets Avoidance-score" function, see Figure 9.1 on page 71.

12.2.4.2 Interpretation of Results

Looking at Figure 12.8 on the next page it shows that RLVehicle do not learn to dodge bullets. According to the regression line RLVehicle have actually performed worse over the 1500 battles. Because the slope of $f(x)$ is so close to zero, this test shows that dodge bullets is not something RLVehicle can learn with the current states

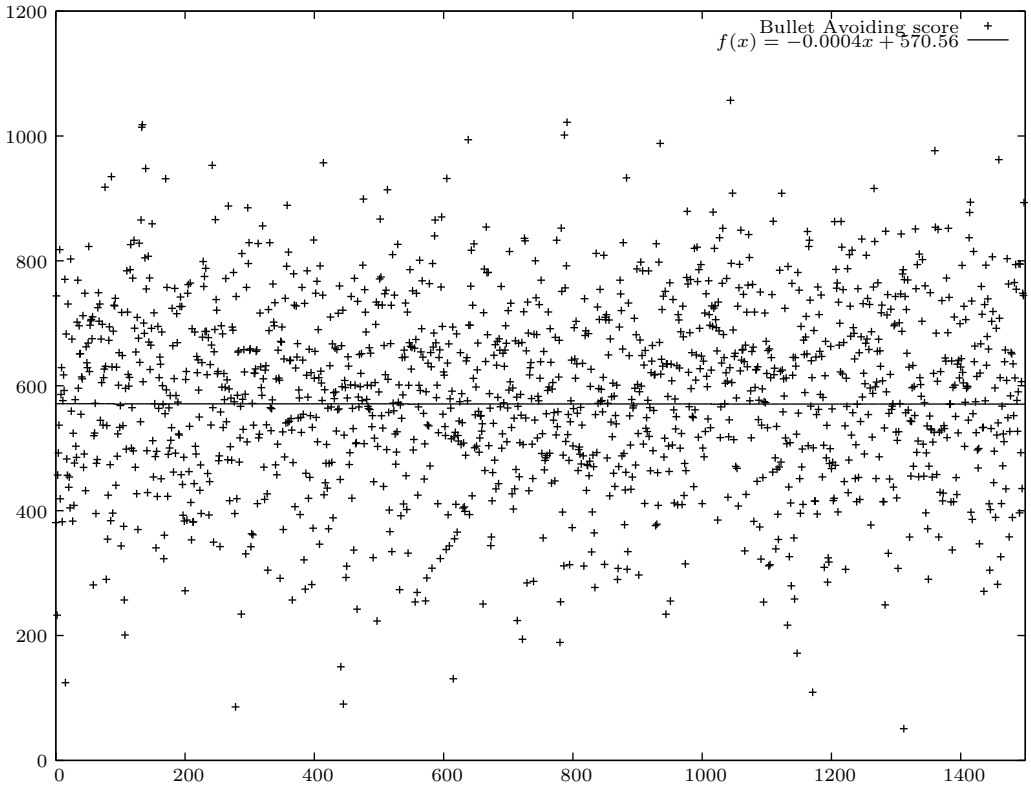


Figure 12.8: Plot of 1500 rounds (x-axis) and "Bullets Avoidance-score" (y-axis) for the test against `RamBot`. $f(x)$ is a linear regression of the bullet avoidance.

The reason `RLVehicle` do not learn anything must be that it have no clear state which specify when and where bullets have been fired.[2]

12.2.5 Robocode Reward Function

This module is a combination of the reward function for Avoiding Enemys on the battlefield, avoiding bullets on the battlefield and the rewards given by the Robocode engine. (See Figure 9.1 on page 72)

Beside the `RLVehicle` module, this test were performed with a gun which shoot, and a radar just rotating constantly. This module were tested against five Crazy opponents.

12.2.5.1 Results

The "Robocode Reward-score" is the score received each round, calculated with use of the "Robocode Reward-function", see Figure 9.1 on page 72.

[2]Such information do not exist in the Robocode API

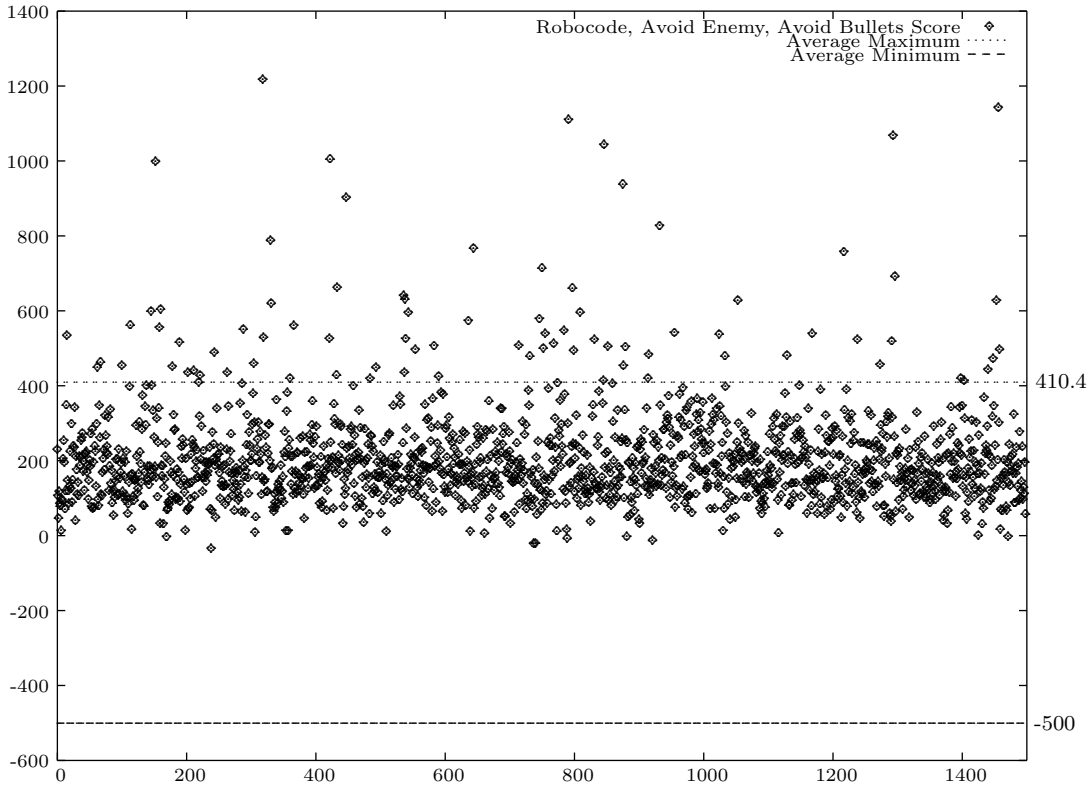


Figure 12.9: Plot of 1500 rounds (x-axis) and "Robocode Reward-score" (y-axis) for the test against 5 Crazy opponents.

12.2.5.2 Interpretation of results

On Figure 12.9 the average maximum score and the average minimum score `RLVehicle` are marked. These values have been calculated based on the average round length for all the battles.

The maximum and minimum have been calculated with use of the function for "Enemy Avoidance-rewards"

$$\begin{aligned} \text{Average rounds length (arl)} &= 155.20 \\ \text{Enemy death reward (edr)} &= 10 \\ \text{Winning bonus (wb)} &= 50 \\ \text{health (h)} &= 100 \\ \text{Energy lost by being hit (el)} &= 4 \\ \text{Reward for being hit by a bullet (rbh)} &= 5 \\ \text{Maximum round length (ml)} &= 651 \text{ Ticks (in round 318)} \end{aligned} \tag{12.2}$$

$$\begin{aligned} \text{Maximum} &= ml \cdot 2 + (5 \cdot edr) + wb = 1402 \\ \text{Average Maximum} &= arl \cdot 2 + (5 \cdot edr) + wb = 410,4 \\ \text{Minimum} &= ml \cdot (-10) + \frac{h}{el} \cdot (-4 \cdot rbh) = -7010 \\ \text{Average Minimum} &= \frac{100}{4} \cdot -4 \cdot 5 = -500 \end{aligned}$$

This graph (Figure 12.9 on the page before shows that `RLVehicle` during the 1500 rounds has not improved, which can be concluded because the value at round 0 is almost the same as the value around 1500. Basically this means that `RLVehicle` have not improved itself during these battles. The reason that `RLVehicle` have not improved itself, is probably the same reason that the results from the "Avoid Enemy-function" test did not manage to improve dramatically. Because this test contains all the same rewards as the "Avoid Enemy-function" the same design flaw is contained in this test as well.

12.2.6 The Best Reward Function

To test which of the four different reward functions is best suited for use in a Robocode match, we have made a test to specify the best reward function. The test will be performed as follows

1. All modules with different reward functions will be equipped with a constantly rotating radar, and a point targeting gun.
2. All modules will have a learning period of 3000 rounds, against `Crazy` (shooting)
3. First test: all modules will be tested against `Crazy` (shooting)
4. Second test: all modules will be tested against `Fire` (shooting)
5. The module which recives the highest total Robocode score, will be the module best fit for Robocode.

Fire (shooting): This opponent keeps the same position until it is hit by a bullet, then it drives 100 pixels forward and after that 100 pixels backward. At all times `Fire` is trying to shoot the opponent on the battlefield.

Modules	Total Score
Ramming module	9649
Avoid Enemy module	4962
Avoid Bullets module	5932
Robocode (Avoid module)	6472

Table 12.1: Test against Crazy.

Based on these tests we can conclude that the ramming funtion seems to be the best suited module for fighting in Robocode.

Modules	Total Score
Ramming module	15753
Avoid Enemy module	11224
Avoid Bullets module	8898
Robocode (Avoid module)	7808

Table 12.2: Test against Fire.

12.2.7 Conclusion

Based on these tests we can conclude that the `RLVehicle` module designed have the ability to improve its performance during battles. But only a single test case have proved to be close to the maximum possible reward; when the ramming reward function is used.

The reason why the other tests did not work as well as the first test case, it could be that we have not tested `RLVehicle` long enough, perhaps it did not reach a converge value on the graphs, but only looks that way. It is probably more likely that the states which we have chosen to represent the world is not good enough, or the action we have chosen for `RLVehicle` could have been better. Another error which we first thought of during the test phase is that even though the `ReactiveLayer` protects the vehicle from ramming into walls it is still an obstacle for the vehicle.

The overall conclusion for this test must be that it is possible to use RL for the `RLVehicle` module, however choosing the right states and action can be difficult, and it is not certain if RL eventual would be a worthy opponent for a hand coded robot using known good tactics.

12.2.8 Improvement Suggestions

A good improvement for the `RLVehicle` module would be to add an extra state, which should give some indication of how close to a wall it is at all time and in which direction the wall is in regards to the robot.

12.3 Bullet Energy Decision

This Section describes the test of the submodules in the gun, concerned with the decision of bullet energy using DG.

The purpose of the test is to determine if the submodules adapts to the environment, meaning that it will give a better advice on how much energy to put into the bullet.

12.3.1 Test Method

The energy decision submodules is part of the gun, meaning that it to some extent is dependent on the module and other submodules. Since we are not interested in testing the entire gun module we will have to make some simplifications.

- The targeting submodules of the gun module is the same during all tests.

- The gun module will on `ScannedRobotEvent` ask the decision graph and shoot immediately according to the answer from the network.

Under the assumption that the targeting part of the gun performs equally during all tests, we can assume that its contribution to the test results will be a constant.

Remembering how we calculate utility (from Section 9.2.1 on page 72), we have that

$$EU(FP|e) = U(H, FP) \cdot P(H|FP, e) \tag{12.3}$$

where $EU(FP|e)$ is the expected utility for a certain firepower, FP, decision given evidence e. $U(H, FP)$ is the utility of a hit, H, with the firepower FP. $P(H|FP, e)$ is the probability of the hit, H, given the firepower, FP, and the evidence e.

We define the *round utility* for a round to be the sum of the expected utility of all the bullets fired in a round $\sum_{\text{All the bullets fired in a round}} EU(FP, e)$, and the *overall utility* to be the sum of all round utilities in a match.

If we record the utility and number of the round and plot a (*round utility*, *round number*)-graph we would expect a plot similar to the plot shown on Figure 12.10(a) on the next page if we during the match adapt the probabilities for the inhibitor tables. For comparison the upper bound for the round utility is shown on Figure 12.10(b) on the facing page. The equation that describes this upper bound for the round utility is given by

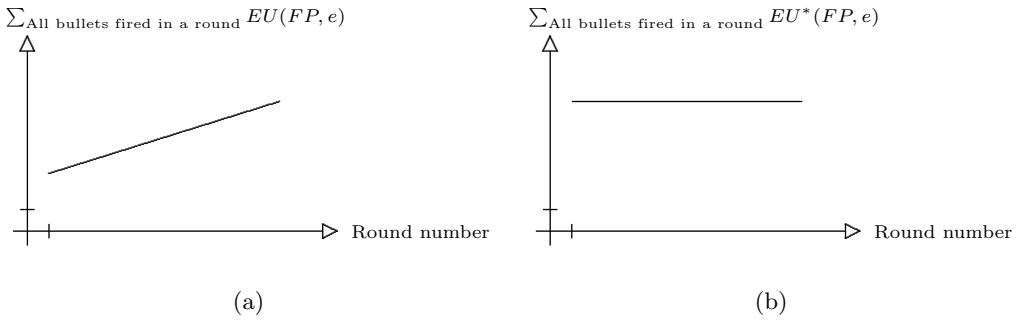
$$\sum_{\text{All bullets fired in a round}} EU^*(FP|e) = U(H, FP) \cdot P^*(H|FP, e) \tag{12.4}$$

where $EU^*(FP|e)$ is the optimal expected utility for the firepower, FP, given evidence e. $P^*(H|FP, e)$ is the actual probability for the hit, H, given the firepower, FP, and the evidence, e, where the actual probability is a probability taken from the perfect adapted probability table. This upper bound will occur if all the bullets fired always hit their target and the energy choice is always the maximum of three. This, however, does not mean that we with certainty can achieve the optimal probability table for P, hence is dependent on the targeting skill of the gun, which can be non-optimal. If the probability tables for the inhibitors are perfectly adapted, then EU^* will be the actual expected utility and we can maximize our utility for our bullets by following EU^*. If the probability tables are inexact, then EU^* will not state the correct utility (EU) and by following an inexact advice from EU we will not get the maximum utility of our bullets.

The upper bound for the expected utility can be calculated as:

$$\sum_{\text{All bullets fired in a round}} EU^* = (8 \cdot FP - 2) \cdot NB \tag{12.5}$$

where NB is the number of bullets fired and FP is the firepower. The number of bullets fired in order to destroy an enemy can be calculated as $NB = \frac{EH}{\text{DC}}$, where EH is the health of the enemy, DC is damage caused by a firepower of FP, which



(a) (b)

Figure 12.10: Concept graph for the plotting of the tests.

is calculated as $4 \cdot FP + 2 \cdot (FP - 1)$. The upper bound for the expected utility will in our case be playing against the robot:

$$EU^*_{crazy} = 8 \cdot 3 - 2 \cdot \frac{100}{4 \cdot 3 + 2 \cdot (3 - 1)} = 154 \tag{12.6}$$

Remember we cannot obtain EU^*_{crazy} value of 154, with the module that we test, since the gun used in these module is far from perfect. The gun we use is point targeting as described in Section 12.1.1 on page 100.

The tests will be performed using the following template:

1. X rounds, $X \gg 1$, will be played with the standard distribution S against an opponent O using no adaptation and the round utility for each round will be recorded, as will the match hit rate.
2. Y rounds will be played against opponent O using adaptation.
3. X rounds will be played with the adapted distribution, A, against opponent O using no adaptation, and the overall utility for each round will be recorded, as will the hit rate for the match.

Using this test template we will expect the overall utility of the adapted match (step 3) to increase in comparison to the unadapted match (step 1).

The reason for having step 1 and 3 is to measure the round utility exact as a mean over many rounds.

The opponents used for testing is described below:

Crazy (no shooting) : This opponent is moving in a deterministic pattern resembling a flower with four leafs (See Figure 12.7 on page 109). For this test it has been modified to disable shooting.

The module setup of ESTIMATOR is: A vehicle movement like Crazy. A Radar that just circles round and round, meaning that we get a `ScannedRobotEvent` at approximately every 8th tick. For the gun module we are using point targeting. The point targeting module will ask the decision module every time it scans an enemy robot how much energy should be put into the bullet and shoot immediately following the reply, if the reply is larger than zero.

12.3.2 Results

The results of the tests are displayed in this table 12.3. Two test have been performed in adherence to the template.

O	X	Y	Overall utility for S	Overall utility for A
Crazy (fading 0.25)	100	200	1,113.4	1,062.0
Crazy (fading 0.98)	1,000	1,000	13,028.1	12,799.8

Mean utility per match m_1 (standard)	Mean utility per match m_3 (adapted)
11.13	10.62
13.03	12.79

Table 12.3: Test against Crazy.

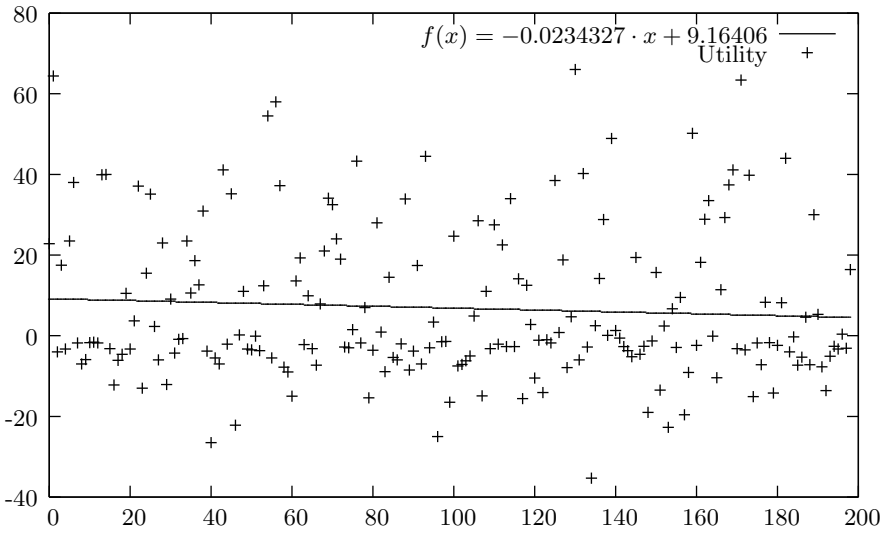


Figure 12.11: The adaptation plot of the $X = 100$ test.

12.3.3 Interpretation of Results

Looking at Table 12.3 it is clear that the overall utility is dropping in both tests, which is exactly the opposite of what we expected. What might be the reason for this in consistence with our expectations?

If we plot the graph from the adaptation of the decision graph (see Figure 12.11 and Figure 12.12 on the facing page) for each test. We note that the tendency line for the $X = 100$ test is decreasing and for the $X = 1000$ test it is growing although it is with such a small rate, that it could be described to uncertainty and therefore it is more or less constant. This is also in conflict with what we would expect; we would like the tendency line to climb in these plots indicating the presence of learning. [3].

We could now calculate the standard deviation of the mean for the steps 1 (s_1) and 3 (s_3) for the two test $X = 100$ and $X = 1000$ (see Table 12.4 on the next page).

[3] We are only learning in step 2

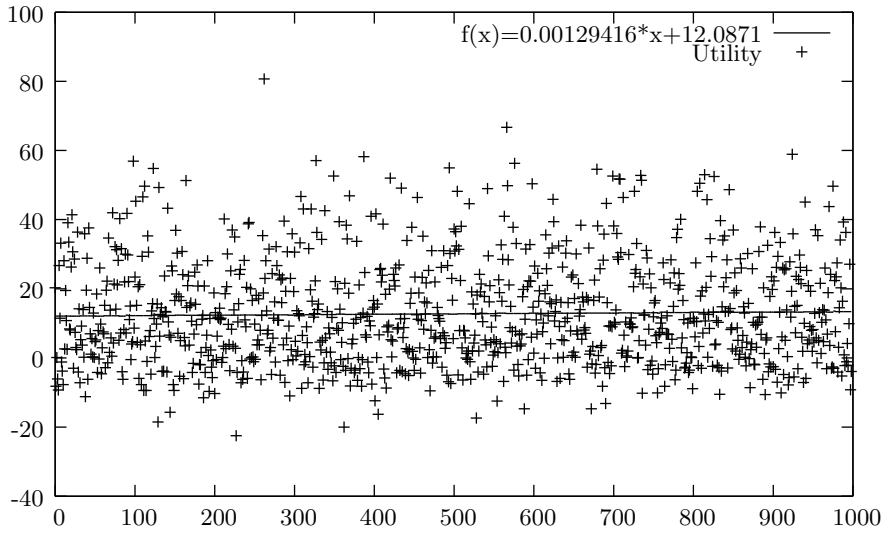


Figure 12.12: The adaptation plot of the $X = 1000$ test.

The standard deviation s for a number n of measurements x with the mean $\bar{x}$ is given by

$$s = \sqrt{\frac{(x - \bar{x})^2}{n - 1}} \cdot \frac{1}{\sqrt{n}}. \tag{12.7}$$

We can use this to get an indication of the uncertainty of the measurements.

X	s_1	s_3
100	1.548	1.347
1000	0.4788	0.4728

Table 12.4: Standard deviation for the two tests $X = 100$ and $X = 1000$.

Now if we look at the mean utility per match for the standard and adapted decision graph in relation to the standard deviation for both tests.

$$m_1 \pm s_1 = 11.13 \pm 1.548 \tag{12.8}$$

$$m_3 \pm s_3 = 10.62 \pm 1.347 \tag{12.9}$$

We note that the two comparisons in Equation 12.8 and 12.9 overlap, meaning that the uncertainty of the measurements overrules any conclusion about changes about the mean utility.

If we look at the $X = 1000$ test we get a smaller standard deviation on the mean.

$$m_1 \pm s_1 = 13.03 \pm 0.4788 \tag{12.10}$$

$$m_3 \pm s_3 = 12.79 \pm 0.4728 \tag{12.11}$$

We note that the two comparisons in Equation 12.10 and 12.11 overlap just like for the $X = 100$ test. This means that the uncertainty still overrules any conclusion

about changes in the mean utility. The Figures in Appendix C on page 156 displays the spreading of the measurements.

12.3.4 Conclusion

We can conclude that the change in the mean utility, is smaller than the change we can measure trough out 1000 rounds due to adaption. This leads to a problem where we cannot conclude whether the module is actually learning something. If the module is learning anything, then the influence of the learning on the mean utility is less than the standard deviation we get for the mean utility in 1000 rounds.

Although we might not be learning, the advice of the decision graph seems reasonably if we inspect it by hand in Hugin, meaning that if we give it some evidence we get a decision, that is consistent with what we would expect by logic reasoning. This would indicate that it is not the decision graph that is failing but rather the adaptation. There could be a number of reasons for this behavior; it could be the fading factor which have been to large or an error in the designed module.

12.3.5 Improvement Suggestions

During the testing of this module an improvement was thought of but never implemented. The idea is to expand the network to handle an evidence node, describing the quality of the gun targeting. This would make the module capable of adapting to a targeting module using machine learning technique, which might vary in its accuracy to target enemies. This extra node would be redundant if a static targeting component were to be used, since the decision graph would simply adapt to the gun with out it as well.

Another idea is to let the network start with a uniform distribution of the inhibitor tables, and let it learn from scratch. This however defies the idea of using existing expert knowledge. If an uniform distribution is entered into the inhibitor tables, the network will always no matter what evidence is entered advice the bullet energy of zero.

Our DG states that we hit, if we hit due to every single inhibitor. The chance for hitting in every single inhibitor with uniform distribution (no matter of the evidence) is 50%. So the total chance for hitting due to all of the six inhibitors is $0.5^6 = 1,56\%$. Because the chance for hitting is so low the advice for the DG is not to shot at all. This leads to no adaptation since shooting is a requirement for generation of cases for adaption. This could be solved in a manner like reinforcement learning using ε-greedy and try choosing bullet energies that defies the advice from the decision graph in order to get experiences with the influence of the inhibitors.

12.4 Artificial Neural Networks

In the following Sections the performance of the ANN in both Joone and our own implementation will be documented with focus on how precise the predictions are and the usefulness of these in a Robocode match.

12.4.1 Neural Network Built with Joone

As mentioned we have attempted to make an implementation of the ANN using the Joone engine. We have divided the implementation into three different small programs to ease the training and development of the network.

- **Data Collector:** A robot that collects data to train the ANN.
- **ANN Trainer and Validate:** Used to train and validate the network to prevent over-fitting of the network with data collected by the `DataCollector`.
- **ANN Precision:** This program is used to verify the predictions as illustrated in Figure 12.15 on page 121 and it also calculates the precision of the network.

When performing the tests we have used a two layer feed-forward network with 5 hidden neurons using the back-propagation algorithm where the learning rate was set to 0.8 and our momentum was 0.2.

12.4.1.1 Test Methods

We have tested the network against `SpinBot`, which is shipped with the Robocode engine. We have chosen `SpinBot` because it moves in the same circular pattern all the time and it should be a recognizable pattern for the ANN.

For collecting training-, validation- and precision verification data to use in the test of the network we have made a `DataCollector` robot, which only functionality is to scan the battlefield for enemies and save the scanned positions to data files. We only use the *collector* against one robot at the time, because we only need positions for one robot at the time as input to our network. The data collected is then used outside the Robocode environment to train and test the performances of the network.

To control the training of the network, we plotted the training error after every epoch (training cycle) and the validation error every 10th epochs. This is done by using 1,000 different patterns for training and another 1,000 for validation of the network. Figure 12.13 on the next page illustrate such a training session.

In Figure 12.13 we have stopped the training after 6,000 epochs. The training has been terminated at this point, because the precision of the network was not getting more precise relative to the training time. The changes in the error was 0.01% and less for every 1,000 epochs above 6,000 epochs and the training time was the same as under 6,000. Since the RMSE (Root Mean Square Error) on the validation set does not increase, there is no indication of over-fitting.

12.4.1.2 Results

When the network has been trained we test its performance by plotting the actual positions of the enemy robot and the predicted positions of the robot. See Figure 12.15 on page 121. For clarity we have included the two smaller Figures 12.14(a) and 12.14(b), which represent the positions and predictions in their own graphs. These plots are made with another data set, which is collected in yet a match against `SpinBot`. The reason for doing this is to simulate a real match, in which the network should be able to predicted the enemies intention and where it

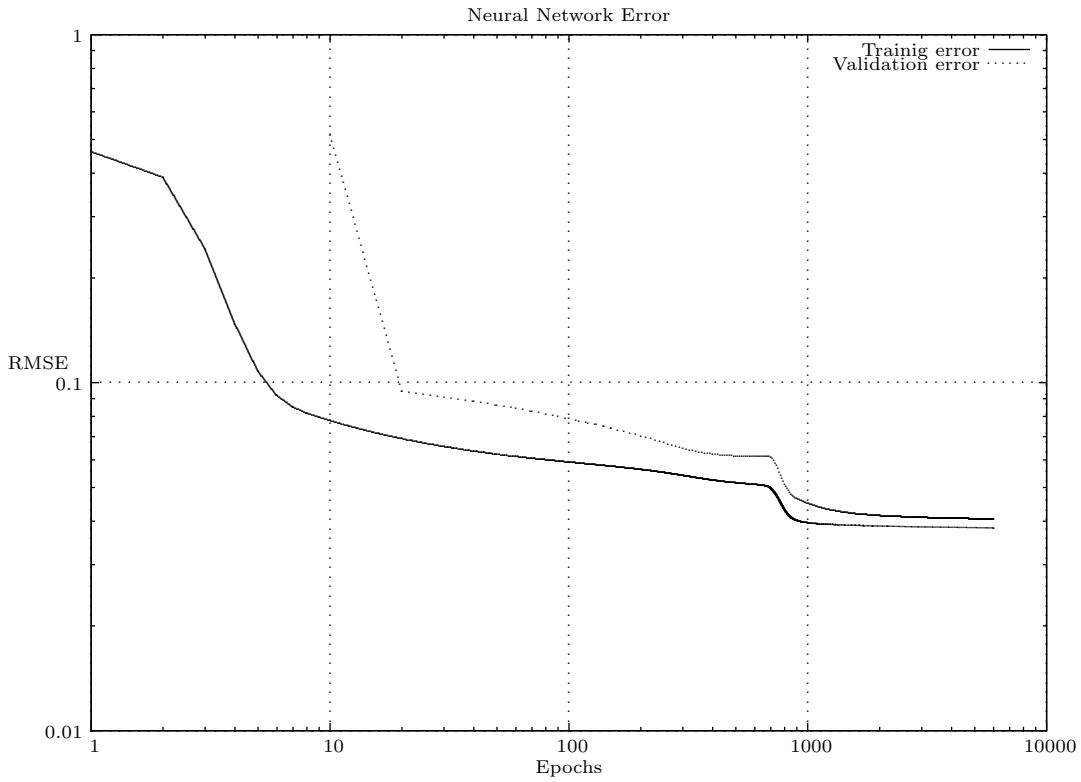


Figure 12.13: The training and validation errors when training against `SpinBot` using 1,000 distinct patterns in the training and the validation set over 6,000 epochs.

has no knowledge of the data and not has be trained. This could also be done by using the validation set, since the network is not trained on these data either.

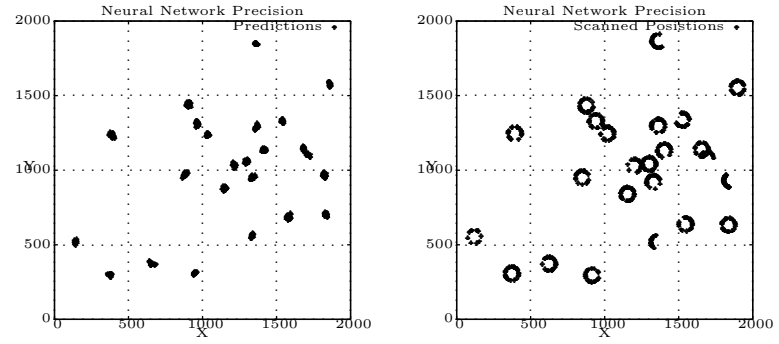
The data used to plot Figure 12.15 can also be used to calculate the performance of the network by looking at how well the predictions fits the positions. We have chosen to look at the network in form of the predictions of the four different sets of coordinates from the eight output gates.

Since we know that a robot is 40×40 pixels in size we have defined a correct guess to be a guess where both the x and the y coordinate are no more than ± 20 pixels away from the center of the robot. The overall performance of the ANN is measured by the percents of correct guesses.

Using the data used in Figure 12.15 the overall performance of the network is:

Number of correct guesses	: 65
Number of wrong guesses	: 495
Correct guess rate, total for all predictions	: 11.6%
Correct guess rate, for the 1. prediction	: 12.8%
Correct guess rate, for the 2. prediction	: 10.0%
Correct guess rate, for the 3. prediction	: 10.0%
Correct guess rate, for the 4. prediction	: 13.6%

If we look at the precision of the x- and y-coordinates separately we will get the



(a) (b)

Figure 12.14: (a) is predictions made by the network and (b) is positions collected form `SpinBot`

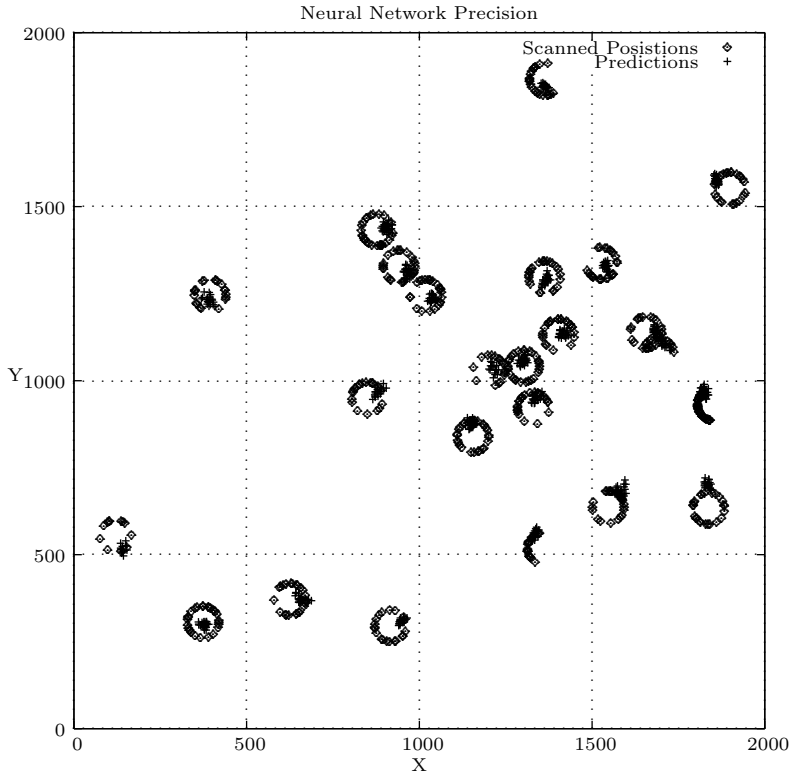


Figure 12.15: Illustration of 560 scanned positions of `SpinBot` and 560 predictions made by the network based on information form 1410 scans.

following results for the x-coordinates:

Number of correct guesses x	: 227
Number of wrong guesses x	: 333
Correct guess rate x, total for all predictions	: 40.5%
Correct guess rate, for the 1. prediction	: 42.9%
Correct guess rate, for the 2. prediction	: 37.9%
Correct guess rate, for the 3. prediction	: 39.3%
Correct guess rate, for the 4. prediction	: 42.1%

and the following results for the x-coordinates:

Number of correct guesses y	: 179
Number of wrong guesses y	: 381
Correct guess rate y, total for all predictions	: 32.0%
Correct guess rate, for the 1. prediction	: 31.4%
Correct guess rate, for the 2. prediction	: 35.0%
Correct guess rate, for the 3. prediction	: 30.0%
Correct guess rate, for the 4. prediction	: 31.4%

12.4.1.3 Conclusion

When looking on the overall performances of the ANN built and trained using the Joone engine, the results are not overwhelmingly good. In the examples used in this test the network will make a clean prediction in 65 out of 560 cases, which gives us a precision of 11.6%. But if we look at the x- and y-coordinates separately then the precision is 40.5% for the x-axis and 32.0% for the y-axis. In some situations it may be sufficiently only to look at one of the axis, e.g in the situation where we are standing head-on with an enemy in the direction of the x-axis shooting along the y-axis we only need precise predictions on the x-coordinates and some uncertainty on the y-coordinate does not matter. The precision would be 40.5% and the change of hitting the enemy is fairly good.

12.4.2 Neural Network Coded by Hand

We have tested the ANN, which we have coded by hand in the same way as the ANN built using Joone.

We have setup the ANN coded by hand with the same parameters as the ANN implemented in Joone.

12.4.2.1 Results

The performance is showed in the following list.

Number of correct guesses	: 99
Number of wrong guesses	: 461
Correct guess rate, total for all predictions	: 17.7%
Correct guess rate, for the 1. prediction	: 25.0%
Correct guess rate, for the 2. prediction	: 17.9%
Correct guess rate, for the 3. prediction	: 15.0%
Correct guess rate, for the 4. prediction	: 12.9%

If we look at the precision of the x- and y-coordinates separately we will get the following results for the x-coordinates:

Number of correct guesses x	: 291
Number of wrong guesses x	: 269
Correct guess rate x, total for all predictions	: 52.0%
Correct guess rate, for the 1. prediction	: 64.3%
Correct guess rate, for the 2. prediction	: 47.9%
Correct guess rate, for the 3. prediction	: 49.3%
Correct guess rate, for the 4. prediction	: 46.4%

and the following results for the x-coordinates:

Number of correct guesses y	: 192
Number of wrong guesses y	: 368
Correct guess rate y, total for all predictions	: 34.3%
Correct guess rate, for the 1. prediction	: 35.0%
Correct guess rate, for the 2. prediction	: 39.3%
Correct guess rate, for the 3. prediction	: 32.9%
Correct guess rate, for the 4. prediction	: 30.0%

12.4.2.2 Interpretation of Results

The overall performance of our networks is 53% better that the network implemented in Joone. This result is calculated as follows:

$$\frac{17.7 - 11.6}{11.6} \cdot 100 \quad = \quad 53\% \tag{12.12}$$

We do not have any explanation for this fact, because we have used the same setup of the network[4]. The only difference in the two networks should be the initial random values of the weights and a difference in the way the log from our data collector robot is split up into training data and test data. The data collector saves its data each time it has made 14 scans of the enemy. Each time it has saved its data, the data is appended as a new line to a file. In Joone we use the first half of the lines in the file as training data and the second half of the lines in the file as validation data. In our own ANN we use every second line as training data and validation data respectively. This should give a more diverse training set, which may explain the difference in the performance. It could be tested by training the ANN in Joone with exactly the same division of the log file into training and validation data.

As discussed in Section 9.2.3 on page 79 we expected the ANN to be better at predicting the enemy positions in near future than enemy positions far into the future. From the above list we can conclude that this expectation holds true for our own ANN because the guess rate is higher for the first predictions than for the last predictions. Though it does not seem to be the case for the network implemented using Joone.

[4]Same number of hidden neurons, learning rate and momentum

The training and validation errors when training the network is shown in Figure 12.16. The network is like the network in Joone trained on a data set from `SpinBot` using 1,000 patterns in the training set and 1,000 patterns in the validation set. The evaluation is shown for the first 1,000 iterations over the training set. Figure 12.16 shows that the squared error is decreasing with the first 1,000 iterations over the training set. Figure 12.17 shows the same as Figure 12.16 but with 100,000 iterations. Because the squared error for the validation set is not increasing we can conclude that there are no over-fitting during this training session within the first 100,000 iterations.

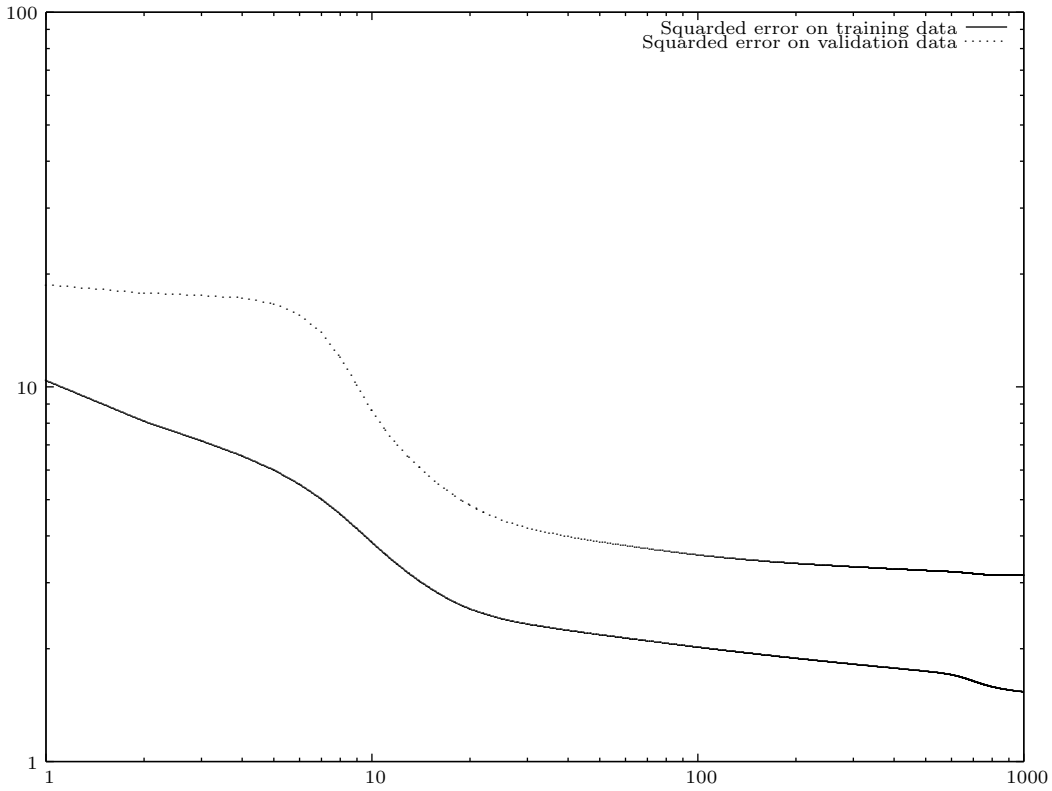


Figure 12.16: The squared error for the training and validation sets during the first 1,000 iterations over the training set.

Figure 12.18 shows how many of the 4,208 positions in the validation set is classified correct (both the x and the y coordinate is off by most 20 pixels). It peeks at 28,000 iterations with 745 correct classifications and at 80,000 iterations it reaches its max with 750 correct classifications.

As with the Joone network we have made a plot of the predictions and the actual positions of `SpinBot`. The predictions shown are the predictions of a net trained with 80,000 training iterations. See Figure 12.20 on page 127, 12.19(a) on page 127 and 12.19(b) on page 127.

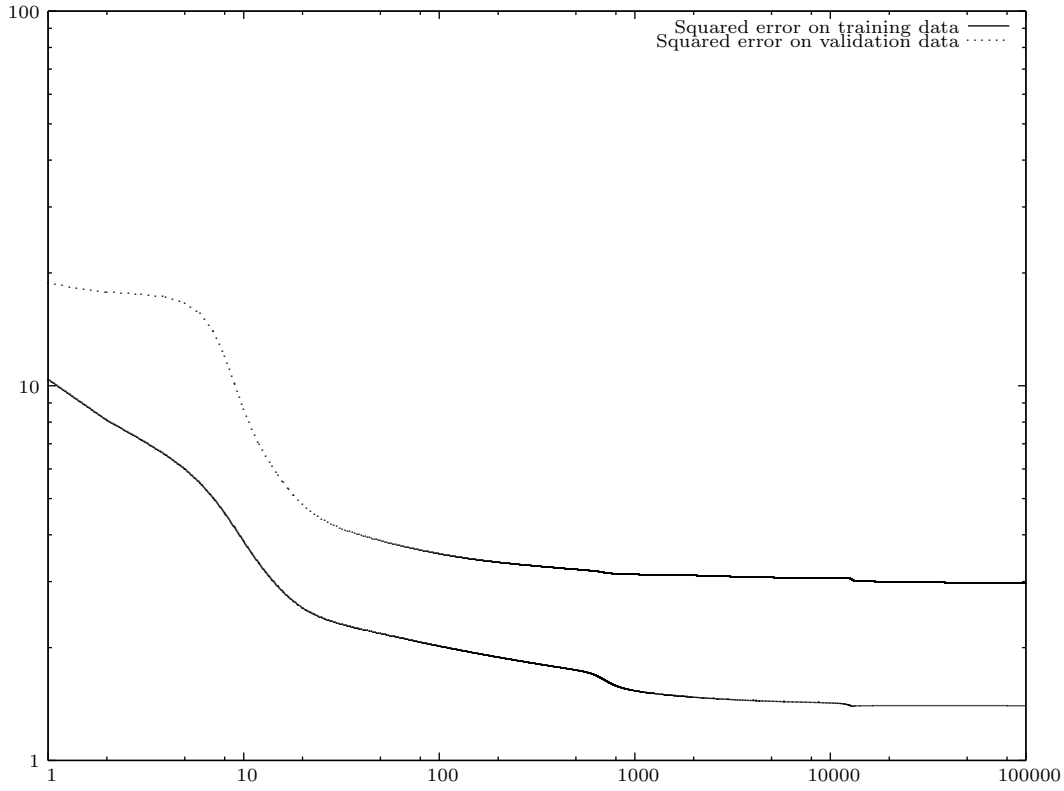


Figure 12.17: The squared error for the training and validation sets during the first 100,000 iterations over the training set. (logarithmic scale)

12.4.2.3 Conclusion

We can conclude that the two ANNs are able to predict the movement of `SpinBot` within ±20 pixels with a classification rate at 11.6% for the network using Joone and 17.7% for the network we have coded by hand. When plotting `SpinBots` positions and the predictions made by our two networks we can see, that even thought only 11.6% to 17.7% of the predictions are within the margin of ±20 pixels, all the predictions are reasonably close to the actual positions.

12.4.2.4 Suggestions for Improvement

After the test phase we tried to tweak the network performance by changing number of hidden neurons, the learning rate and the momentum. The best configuration we found for the network coded by hand, was 9 hidden neurons, learning rate at 0.7 and momentum at 0.2. After this tweak we got a performance as shown below, after 4,000 iterations.

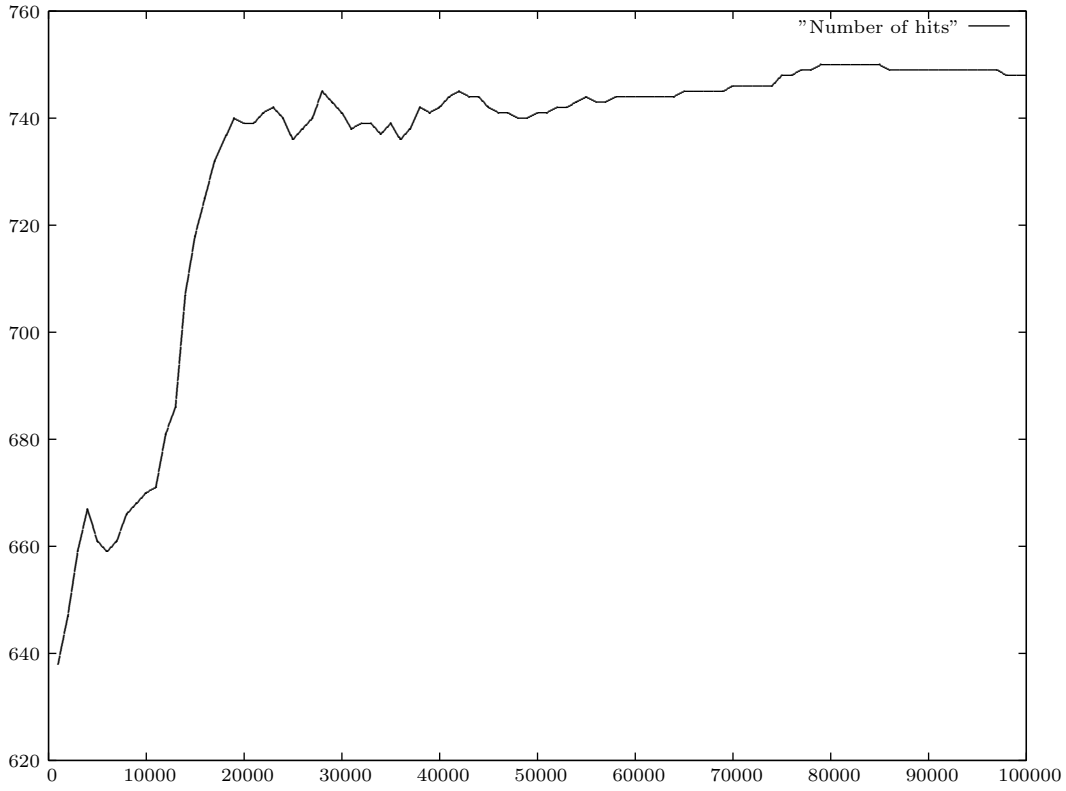


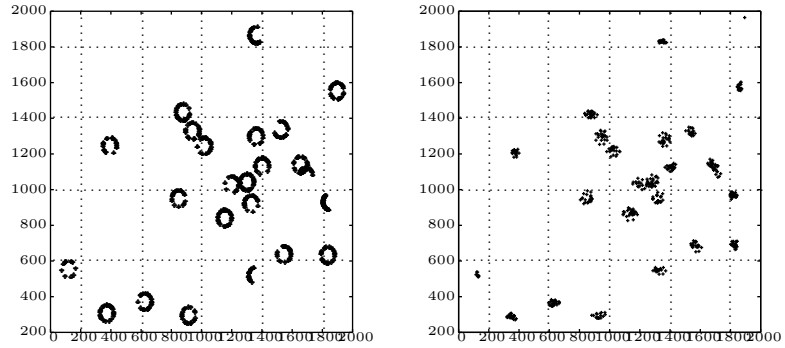
Figure 12.18: The number of correct predictions during the 100,000 iterations as shown in Figure 12.17.

Number of correct guesses	: 158
Number of wrong guesses	: 402
Correct guess rate, total for all predictions	: 28.2%
Correct guess rate, for the 1. prediction	: 52.1%
Correct guess rate, for the 2. prediction	: 32.1%
Correct guess rate, for the 3. prediction	: 17.1%
Correct guess rate, for the 4. prediction	: 11.4%

If we look at the precision of the x- and y-coordinates separately we will get the following results for the x-coordinates:

Number of correct guesses x	: 301
Number of wrong guesses x	: 259
Correct guess rate x, total for all predictions	: 53.8%
Correct guess rate, for the 1. prediction	: 72.9%
Correct guess rate, for the 2. prediction	: 52.9%
Correct guess rate, for the 3. prediction	: 47.1%
Correct guess rate, for the 4. prediction	: 42.1%

and the following results for the x-coordinates:



(a) (b)

Figure 12.19: (a) is predictions made by our own network and (b) is positions collected form `SpinBot`

.

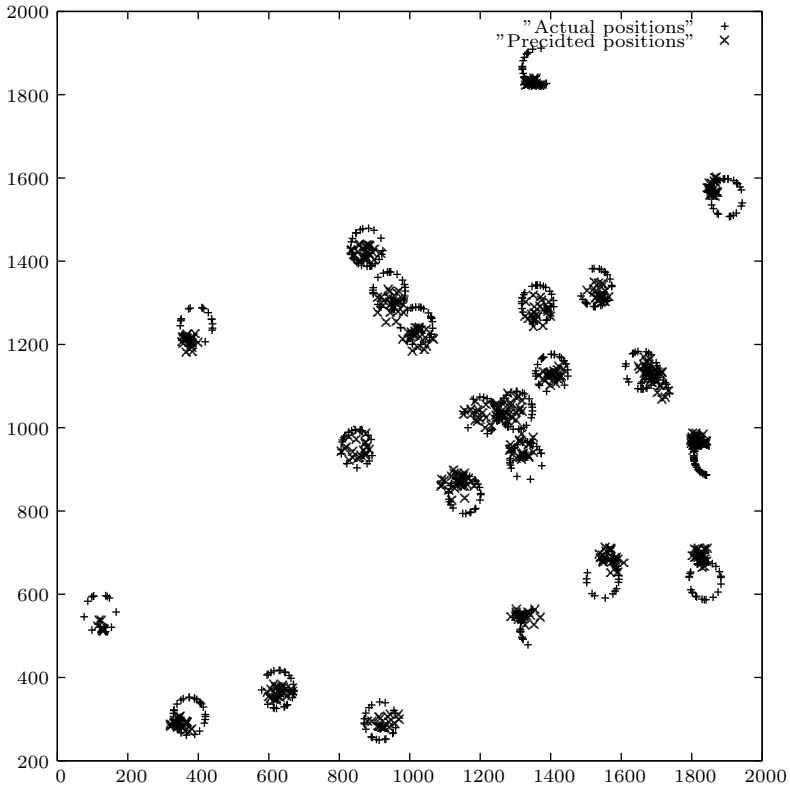


Figure 12.20: Illustration of 560 scanned positions of `SpinBot` and 560 predictions made by the network based on information form 1,410 scans.

Number of correct guesses y	: 274
Number of wrong guesses y	: 286
Correct guess rate y, total for all predictions	: 48.9%
Correct guess rate, for the 1. prediction	: 64.3%
Correct guess rate, for the 2. prediction	: 55.7%
Correct guess rate, for the 3. prediction	: 40.0%
Correct guess rate, for the 4. prediction	: 35.7%

We have also found a better performance for the neural network made with Joone. We have tweaked the network to get a hit precision of 17.14% overall, 43.03% for the x-axis and 39.46%, which gives us a % better precision. All parameters are the same as before 6,000 epochs and validated over 1000 patterns, we have only changed the number of hidden neurons form 5 to 17. The results are as follows:

Number of correct guesses	: 96
Number of wrong guesses	: 464
Correct guess rate, total for all predictions	: 17.14%
Correct guess rate, for the 1. prediction	: 5.00%
Correct guess rate, for the 2. prediction	: 3.75%
Correct guess rate, for the 3. prediction	: 4.29%
Correct guess rate, for the 4. prediction	: 4.11%

If we look at the precision of the x- and y-coordinates separately we will get the following results for the x-coordinates:

Number of correct guesses x	: 241
Number of wrong guesses x	: 319
Correct guess rate x, total for all predictions	: 43.03%
Correct guess rate, for the 1. prediction	: 12.32%
Correct guess rate, for the 2. prediction	: 10.36%
Correct guess rate, for the 3. prediction	: 10.00%
Correct guess rate, for the 4. prediction	: 10.36%

and the following results for the x-coordinates:

Number of correct guesses y	: 221
Number of wrong guesses y	: 339
Correct guess rate y, total for all predictions	: 39.46%
Correct guess rate, for the 1. prediction	: 10.89%
Correct guess rate, for the 2. prediction	: 10.54%
Correct guess rate, for the 3. prediction	: 9.46%
Correct guess rate, for the 4. prediction	: 8.57%

12.5 Radar - Reinforcement Learning

Here we will describe how we have tested the learning capabilities of our `RLRadar` module, and how fast it reaches its maximum learning capability.

12.5.1 Test methods

We have devised several different tests to see how well our `RLRadar` module performs compared to a module which only rotates the radar to the right (dummy radar module). All the tests were started on a Q-table initialize with zeros and each test spawns a thousand rounds. We have also performed all of the test with the dummy radar module, instead of `RLRadar` module, this has been done to have something to compare our `RLRadar` module to. The tests have been conducted against two robots, `Crazy` and `Walls`. Crazy's movement pattern makes Crazy's movement specially hard to forecast. The Walls robot moves in a pattern where it try to crawl along the walls. Neither of the robots fire.

Here we have listed the tree different tests we have performed:

1 **Non moving vehicle vs. Crazy**: Our robot was equipped with non moving Vehicle, non shooting gun and `RLradar` and the enemy was Crazy.

2 **Crazy vehicle vs. Crazy**: Our robot was moving identical with the movement of Crazy, non shooting gun and `RLradar` and the enemy was Crazy. We look at this test as the hardest situation to manage, because both the target and our robot is moving.

3 **Non moving vehicle vs. Walls**: Our robot was equipped with non moving vehicle, non shooting gun and `RLradar` and the enemy was Walls.

12.5.2 Results

The data from test 1 is illustrated in Figure B.1 and Figure B.2 in the appendix. The data from test 2 is illustrated in Figure 12.21 The data from test 3 is illustrated in Figure B.4 and Figure B.5 in the appendix.

12.5.3 Interpretation of Results

From Figure 12.21 on the next page we can see that it does most of its learning in the first hundred rounds and after these the learning curve flattens. This is due to the fact that after the first hundred rounds the module has a good approximation for the optimal scanning policy because it have visit most of the states of the Q-table several times.

If both our robot and the enemy are moving, then we have more diversified visits in the Q-table during a round because the distance to the enemy varies more. This give the smooth growing in scan rate. In contrast if we stand still we are not visiting the same amount of different Q-table entries during a round and as a result the graphs in the Appendix shows a greater spreading of our "average scan event per tick for each round".

The theoretical maximum for the number of scans per tick is 1, since we can at most scan the enemy once per tick. When we are moving we converge about 0.5 scans per tick, and the maximum average per round is about 0.7. When we are not moving we sometimes reach the max at 1 scan per tick in average in a whole round. This is impressive, since we actual scans the enemy as much as theoretical possibly. But in average for all the rounds we are not near 1 scan per tick.

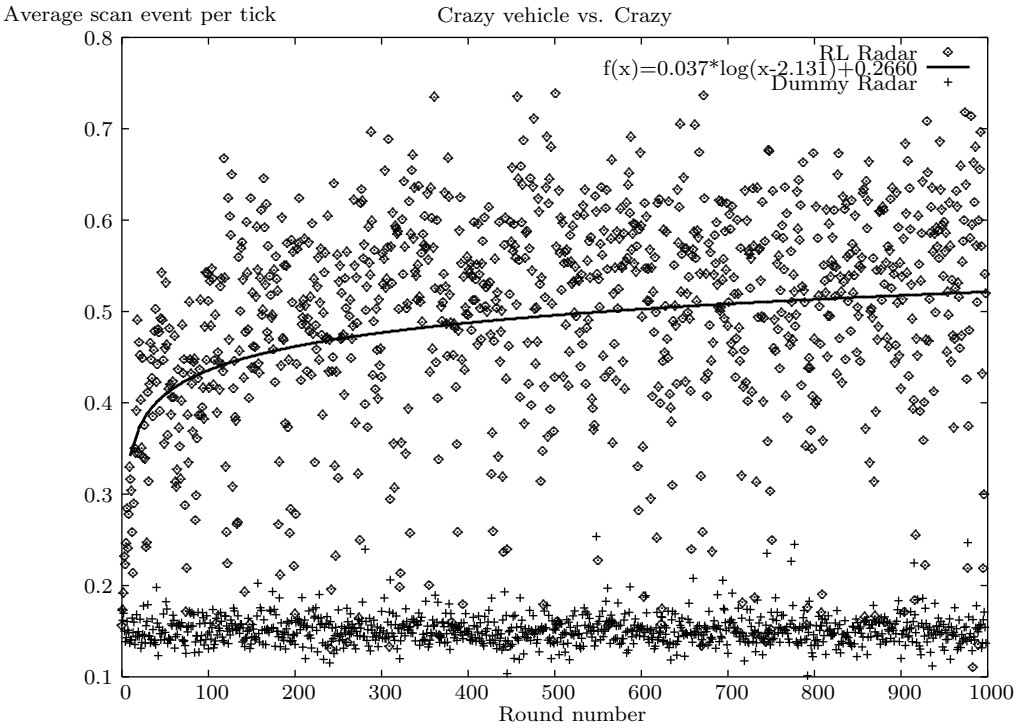


Figure 12.21: Crazy moving vehicle vs. Crazy non fire

In the graphs we have also plotted the performance of a radar just turning around. This scans the enemy once per rotation, this gives 0.125 scan per tick in average, because it takes 8 ticks per rotation. In relation to this dummy radar our `RLradar` is performing well. If we converge to 0.5 scans per tick, then we perform 4 times better than the dummy radar.

12.5.4 Conclusion

All of the graphs of our `RLRadar` module, shows that our `RLRadar` module becomes better at scanning a robot often in about 100 rounds. Under the hardest conditions we get an average scan per tick at 0.5 which is half of the theoretical maximum, but still 4 times better than a dummy radar module which is just rotating the radar the same way at all time.

12.5.5 Improvement Suggestions

In the discussion in Section 8.2 at page 55 we choose α to be a constant to make our system able to adapt to changes in the environment, this choice was made despise the proof that Q will converge to Q^*. Maybe we can improve the radar module if we use this equation when calculating α?

We have tested this radar module again with α given by equation 8.8. The test result can be seen at Figure B.3. The Figure B.3 should be compared with Figurefig:grafNonMovingCrazy, which is the same test with $\alpha = 0.1$.

On the graph with α constant we can see that there are more rounds with $0.9-1.0$

scans per tick than when α is given by equation 8.8 within the first 1,000 rounds. When we are using the equation we only get a single round with more than 0.9 scans. This can be because Q converge slower when using equation 8.8. If we have played more than 1,000 rounds maybe we have got a totally better scan policy, than with $\alpha = 0.1$, but at least we can conclude that we learn faster when $\alpha = 0.1$.

Summary

In this Chapter we presented the tests of our learning modules. The overall result was that most of the modules were able to improve performance when given experience. Only the Bullet Energy Decision submodule of the Gun module, was not able to do any improvement or adaption. We saw that GA Vechicle was able to find a population with higher fitness than the one it started with. RL vehicle was able to learn to ram its enemies. ANN targeting learned to predict the movement of predictable pattern and last but not least the RL radar module learned to scan its enemies better than the brute force approach.

13 Test of the Robot Team

In this Chapter we will perform a test of the ESTIMATOR team containing the best machine learning modules. We do this in order to see if the complete robot team is actually getting better by using machine learning modules that have been trained.

13.1 Test Method

The ESTIMATOR team will consists of five members all containing the same modules, playing against the `SpinBot` team, which consists of five `SpinBot`'s that have been modified not to shoot at each other.

13.2 Results

In the first test the modules of ESTIMATOR are: `RLVehicle`, `RLRadar` and `AimGun` containing the components `Energy Decision Graph` and `Neural Networks Targeting`.

Robot name	Total score	Survival	Last Survivor Bonus	Bullet Damage
ESTIMATOR	155	150	0	5
Spinbot	21501	12350	2350	5667

Bonus	$RamDamage \cdot 2$	Bonus	Survival 1sts	Survival 2nds
0	0	0	0	10
1133	0	0	10	0

Table 13.1: Test of ESTIMATOR team (`RLVehicle`, `RLRadar` and `AimGun`) vs. `SpinBot` team using untrained modules. (Table is split to fit the page.)

In the second test the modules of ESTIMATOR are: `GAVehicle`, `DummyRadar` and `AimGun` containing the components `Energy Decision Graph` and `Neural Networks Targeting`.

Robot name	Total score	Survival	Last Survivor Bonus	Bullet Damage
ESTIAMTOR	269	250	0	5
Spinbot	22022	12250	2450	5613

Bonus	*RamDamage* · 2	Bonus	Survival 1sts	Survival 2nds
0	14	0	0	10
1134	514	59	10	0

Table 13.2: Test of ESTIMATOR team (`RLVehicle`, `RLRadar` and `AimGun`) vs. `Spinbot` team using trained modules. (Table is split to fit the page.)

Robot name	Total score	Survival	Last Survivor Bonus	Bullet Damage
ESTIMATOR	817	450	0	36
Spinbot	20707	12050	2300	4939

Bonus	*RamDamage* · 2	Bonus	Survival 1sts	Survival 2nds
0	300	30	0	10
964	380	73	10	0

Table 13.3: Test of ESTIMATOR team (`GAVehicle`, `DummyRadar` and `AimGun`) vs. `SpinBot` team using untrained modules. (Table is split to fit the page.)

13.3 Interpretation of Results

From Table 13.3 and Table 13.4 we can see that our team is performing better with the trained modules, since the "Total score" in Table 13.4 is higher than the "Total score" in Table 13.3.

We note that the test using the `GAVehicle` and `DummyRadar` module is performing better than the `RLVehicle` and `RLRadar`. This is what we would expect, since the ANN is trained using a dummy radar, and are thereby dependent on receiving a `ScannedRobotEvent` approximatly every 8th tick for the predictions to be generated correctly. The fact that the gun works better with the `DummyRadar` is shown in the Table 13.4, where the bullet damage is larger than in Table 13.2.

13.4 Conclusion

We can see that our robot team's score is improving with the use of trained machine learning modules compared to untrained modules. However we also note that we are

Robot name	Total score	Survival	Last Survivor Bonus	Bullet Damage
ESTIMATOR	2428	1650	0	313
Spinbot	18544	10800	1750	4619

Bonus	*RamDamage* · 2	Bonus	Survival 1sts	Survival 2nds
20	428	15	0	10
855	424	95	10	0

Table 13.4: Test of ESTIMATOR team (`GAVehicle`, `DummyRadar` and `AimGun`) vs. `SpinBot` team using trained modules.(Table is split to fit the page.)

not even close to beating the simple team of `SpinBot`s, which leads to the conclusion that there still is room for improvements for the ESTIMATOR team.

14 Evaluation Conclusion

This Part of the report is concerned with evaluating the designed system. The evaluation was done on a per module level and general level.

Chapter 11 gave a quick overview of how much of the design we actually have implemented. The technologies which we have implemented includes the fundamentals of the architecture, the two `Vehicle` modules, the ANN `Gun` submodule and DG `Gun` submodule as well as the `RLRadar` module. This means that we have been able to test a working robot.

The actual test of the modules were described in Chapter 12 where we saw that the vehicle using GA showed improvement over time, which was the result we were looking for. The improvement was however a lengthy task, where we had to evolve for at least 35 generations before the module converged. 35 generations take about 400 round a piece, therefore it takes around 16,000 rounds before the module is fully evolved for combat. As this is against one team, it is not given that the evolved module will perform well against another team. Thus for the module to be a part of a real Robocode robot which should compete against hardcoded enemies, it would be of little use. Do keep in mind that this is just a way to put the module into a larger perspective, as the goal of the module is actually fulfilled, it is able to learn and improve through time and evolution.

The test of the `RLVehicle` module showed similar results. The module was able to learn to ram the enemy and to avoid the enemy but when it came to avoid the bullets from enemies, we saw no improvement. It seemed that the two successful reward functions converged within 200 rounds, which actually is a useful result, at least compared to the 16,000 rounds played by the `GAVehicle` module.

ANN prediction proved to be able to recognize patterns and predict reasonable future locations of the enemy. The best result of the two implementations, roughly 28%, should not be considered bad, because prediction is quite hard. It is able to predict where on the battlefield of 2000x2000 pixels a robot of 40x40 pixels will move next.

The DG energy choice module was not able to improve over time, so it was not able to adapt at all. However, when we inspect it, it seem to be able to make sound

choices, but unfortunately these are just hardcoded into the DG, hence making the DG energy choice module unsuccessful for us.

The `RLRadar` module is the best module according to the tests. After a few rounds it started outperforming the module it was compared to and still after this point it kept improving until it stabilized roughly at an average scan of 0,5 after 200 rounds.

As we showed in Chapter 13, the robot as a whole was able to improve its performance during the duration of this test. However, it was not able to beat `SpinBot` team at all, but this is not the main goal for us. This latter evaluation performed in this Chapter is just further comparison.

All in all we have made an expert system capable of improving through time. Whether it is able to perform adaptation within a reasonable amount of time has not yet been settled. However, it is not able to make a significant difference in the Robocode world.

Part IV

Conclusion

15 Conclusion

In this Chapter we will summarize and conclude upon the results achieved during this project.

We have analyzed the Robocode platform by designing a robot for this platform in mind. Furthermore we have also analyzed different agent architectures with the objective of creating an expert system in the shape of a robot in Robocode,thus we have found it obvious to design a robot in Robocode as a multi agent system. After analyzing different multi agent architectures we have decided to design our robot as a modified version of the TouringMachines architecture.

Before we designed our agents for our multi agent architecture, we presented a selection of the design criteria, which outlined our priorities for the design. We will return to these criteria later in this conclusion, but before that we will first conclude upon our design and tests.

We designed five agents, which constitutes our multi agent system. Two of these were supposed to be in control of the vehicle part of the robot. There were also two agents to control the gun, where one of them were to predict the enemies behavioral pattern and the other one were to decide how much energy to use when shooting at the enemies. Additionally an agent was designed to control the radar.

All of the agents were designed to use different techniques of machine learning. We have worked with Bayesian Networks, Decision Graphs, Genetic Algorithms, Reinforcement Learning and last but not least Artificial Neural Networks.

After having tested all the agents together in *one* robot, we have been able to conclude that our multi agent architecture is able to handle all the agents. Our architecture also supports exchanging the agents for the different parts of the robot as it becomes needed. The only flaw to point out in our architecture is the implementation of the reactive layer, which were supposed to keep our robot from driving into the walls. It does accomplish this, but it is not able to move the robot away from the walls, so a deadlocked situation may occur. In this deadlock the reactive layer keeps the robot from driving towards the walls and the agent controlling the vehicle does not know that the robot is near the walls and therefore still wishes to move ahead.

This situation could be avoided by changing the implementation of the reactive layer, so it makes sure that the robot is moved away from the walls.

After having tested all the agents separately we are able to conclude that all

(except one) are able to improve their performance through experience.

Our Genetic Algorithm vehicle module is able to evolve a population of hypotheses, where each entity express a policy about how to move. This evolution stagnates after approximately 35 generations, where we reach a united fitness at about 29,000, which is quite far from our theoretical fitness for a generation, which is 38,000. This boundary is calculated from our fitness function. The reason why we do not reach our theoretical maximum can be that our design of the hypotheses is not able to express a policy that is able to reach the 38,000. Besides that the opposition from the opponents prevent our hypotheses from reaching maximal fitness. As 400 matches are played per generation in our GA module, the module uses 14,000 matches to reach the best hypotheses that this module may produce.

Our Reinforcement Learning module is able to produce a policy, that can make our robot score points, by ramming the opponents. This policy is learned after 200 matches. The module is also able to learn to avoid being rammed, but this is not reflected so clearly through our test, because of the conflict between the reactive layer and the deliberative layer. In this conflict our robot drives away from the enemy, but ends up at the wall, where it is stopped by the reactive layer and the enemy without problems may ram us.

Using our design of the Genetic Algorithms and Reinforcement Learning modules for the vehicle; the Reinforcement Learning module uses only 200 rounds to learn a useful policy, while the Genetic Algorithm module uses 14,000 rounds to find a good policy.

One of our gun modules, the one that is in charge of aiming at the enemies by predicting patterns in their movement, is able to predict the location of a robot moving in small circles 28% of the times with a precision of 40×40 pixels, on a battlefield that is 2000×2000 pixels. The other 72% of the predictions are also located reasonably close to the actual position of the enemy.

Our module for choosing the energy to shoot with, proposes a bid which seems logical. So at this point the module works fine, but the idea was the module should be able to learn the probability of us hitting the enemies with our bullets. This was supposed to happen through an adjustment of the probabilities that into the system, which determines the probability of hitting the enemy, given a sequence of inhibiting factors. We are however not able to make this adjustment happen. We know the theory of how the updates are supposed to be calculated, but we cannot make Hugin do the updates. This problem could probably be solved by doing the update of the probabilities without the use of Hugin. If this should not work, we could suspect that it is the model that is missing vital information.

Our radar module is able to, through approximately a 100 rounds, learn to scan an enemy once every two ticks on average. This is in spite of it being on a vehicle that moves the whole robot around the battlefield without telling the radar about it and even though the enemy that is being scanned is moving in an advanced pattern. This module has a smaller representation of the world in its Q-table than the Reinforcement Learning module of the vehicle, which means that this module may learn the desired policy faster.

All in all we have made a variety of agents that basically are able to improve their performance, which was the main goal of this project.

Having presented the achievements during this project period, we will now finish

the conclusion by going through the criteria to conclude upon how well the final product complies with the criterion. We will only comment on the criteria, which we have rated very important and important as we have not put any effort into the others.

Single Robot: We have created a modular architecture that makes it possible to replace the `Vehicle`, `Gun` and `Radar` components of the robot and it has been shown to work well under the implementation. Furthermore replacing one module, does not affect the others, meaning that they are not dependent on each other. So about the criterion of modularity, which we deemed important, we conclude to be achieved.

We can also conclude that the criterion of independence, also rated important, has been fulfilled in that the robot does not depend on any information coming from teammates.

Modules: We conclude that the criteria of learnability on the module level, which was rated very important, has been accomplished with mixed success. As we saw in the previous Part, all the implemented modules were able to learn except for one.

We conclude that the important rated criteria independency only has been partially fulfilled, since most of the modules are dependent on another module to perform reasonably well. For instance the `Gun` modules are very dependent of the `Radar` module scanning enough enemies. They can, however, still function without, but the results of the modules will then not be good, as in the learning will probably not commence.

As for the uniformity of the modules, which was deemed important. Uniformity was whether the modules return actions in the same format. We conclude that it was a success. The success of this criterion is mainly due to the fact that we split the robot up into the three *independent* areas of responsibility: `Vehicle`, `Gun` and `Radar`.

About the important rated criterion of integrateability, we conclude that it has been fulfilled in that we only have a single robot, and the way the modules fit into it, is due to the fact that the interface is well defined.

General: as we managed to test for learnability and adaption for all the modules. However one of our tests showed that there where no adaption in the Energy Decision submodule.

Correctness as a criterion was rated important. We believe that this criterion has been partially fulfilled as we follow the design in almost every detail. One exception is that the `ReactiveLayer` currently is implemented within the `ActionManager` class.

We conclude that learnability which was rated very important, is fulfilled. The reason is that we have practically not hardcoded anything in the robot except from the `ReactiveLayer` and the part of the `Gun` module that makes use of the predictions made by the prediction submodule.

As most of our important criteria have been fulfilled, we can conclude that all in all the design was successful. Thus we complete the report. □

Part V

Bibliography

Bibliography

[alpa] IBM alphaWorks. Robocode. http://robocode.alphaworks.ibm.com/home/home.html. Seen on 17-9-2004.

[alpb] IBM alphaWorks. Robocode api. http://robocode.alphaworks.ibm.com/docs/robocode/index.html. Seen on 17-9-2004.

[alpc] IBM alphaWorks. Robocode help. http://robocode.alphaworks.ibm.com/help. Seen on 23-9-2004.

[alpd] IBM alphaWorks. Robocode physics. http://robocode.alphaworks.ibm.com/help/physics/physics.html. Seen on 23-9-2004.

[A/S] Hugin Expert A/S. Hugin expert. http://www.hugin.com/. Seen on 11-12-2004.

[Flo] Dario Floreano. Robot learning group. http://diwww.epfl.ch/lami/learning/learning.html. Seen on 11-10-2004.

[Gam] Alife Games. Artificial life games as free software bSerene. http://alifegames.sourceforge.net/bSerene/index.html. Seen on 12-10-2004.

[Hea] Jeff Heaton. Java neural network. http://www.jeffheaton.com/ai/index.shtml. Seen on 14-11-2004.

[IK97] Kevin Warwick Ian D. Kelly, David A. Keating. Mutual learning by autonomous mobile robots. Technical report, Department of Cybernetics, University of Reading, 1997.

[Ins] Carnegie Mellon University Robotics Institute. Autonomous land vehicle in a neural network (alvinn). http://www.ri.cmu.edu/projects/project_160.html. Seen on 11-10-2004.

[Jen02] Finn Verner Jensen. *Bayesian Networks and Decision Graphs*. Springer, 2002.

[Joo] Joone - java object oriented neural engine. http://www.jooneworld.com/. Seen on 11-12-2004.

[Man] Yishay Mansour. Reinforcement learning. http://www.cs.tau.ac.il/~evend/Workshop/sadna-games.ppt. Seen on 14-11-2004.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw-Hill International Editions, 1997.

[PM00] Peter, Stone and Manuela, Veloso. Mulitagent systems: A survey from a machine learning perspective. *Autonomous Robotics*, 8(3), July 2000.

[rob] Robocode repository. http://robocode.turbochicken.com/. Seen on 12-12-2004.

[SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning*. The MIT Press, 1998.

[Tec] TechTarget. Expert system. http://whatis.techtarget.com/definition/0,,sid9_gci212087,00.html. Seen on 07-12-2004.

[WD92] Christopher J.C.H. Watkins and Peter Dayan. *Q-Learning*. Machine Learning, 8, 279-292, 1992.

[Woo02] Michael J. Wooldridge. *An Introduction to MultiAgent Systems*. Number 0-471-49691-X. John Wiley & Sons Ltd., 2002.

Part VI

Appendix

A Competition Rules

Robocode setup

- Battlefield size: 2000x2000
- Radar range: 1200
- Number of players per team: 5
- Gun cooling rate: 0.1 (Default)
- Inactivity time: 450 (Default)

Tournament

- A team fights. One team against another team ("Head to head").
- All teams fight 1000 matches against each other team.
- Each project group can participate with one team.
- There are no restrictions on how one group can compose their team.
- After battles, groups should share their current robot.
- Experience gathered during one battle in the tournament may not be used in another battle in the tournament.
- When a robot have been lined up for a official match, official matches will take place every Friday, by the the teams which have a robot ready to compete, the robot will be appointed to be a "open source" robot, this means that, every team which took part of the competition can have a copy of all the other robots, participated in the match. By doing this it will be possible for all teams to train their robot against the other teams. This could result in more exiting battles, and more important, give the possibility for the different teams to train their AI to be better.
- It is not allowed to cheat during a competition. Cheat in this regard could be to exploit security holes or bugs in the Robocode engine.

B Reinforcement Learning Radar graphs

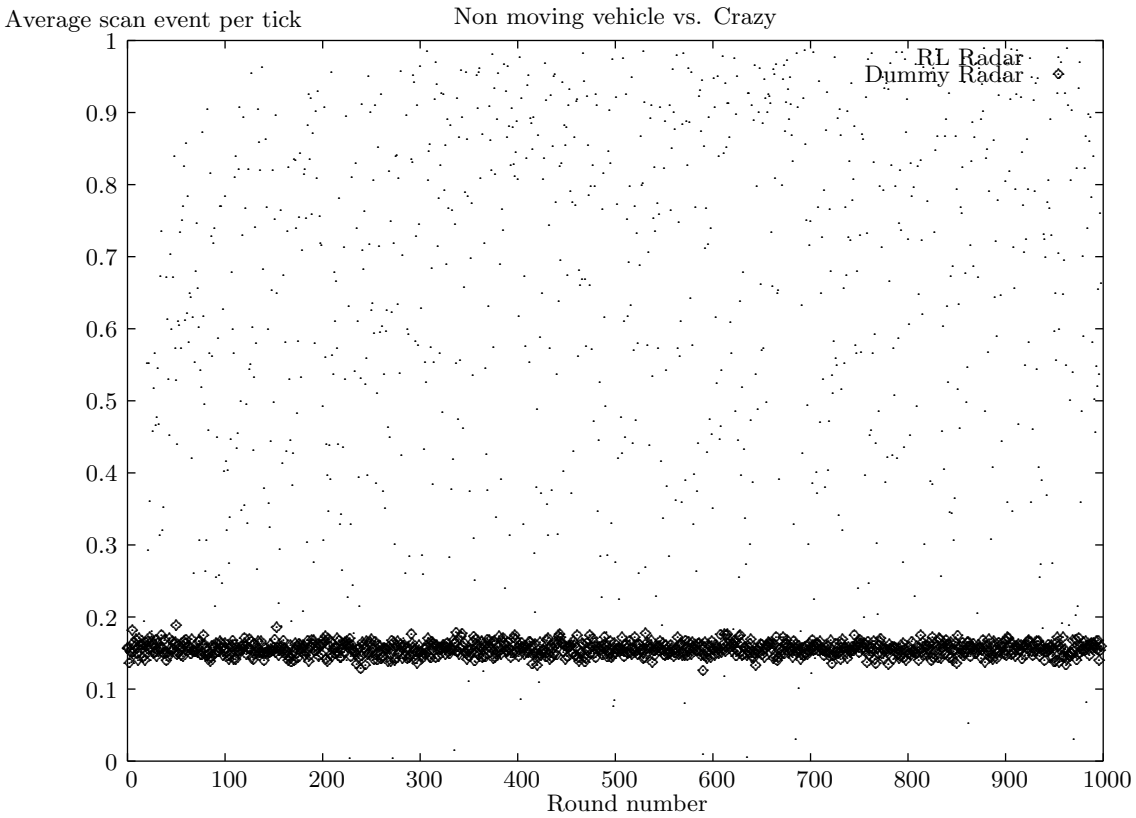


Figure B.1: Shows **Non moving vehicle vs. Crazy non fire**

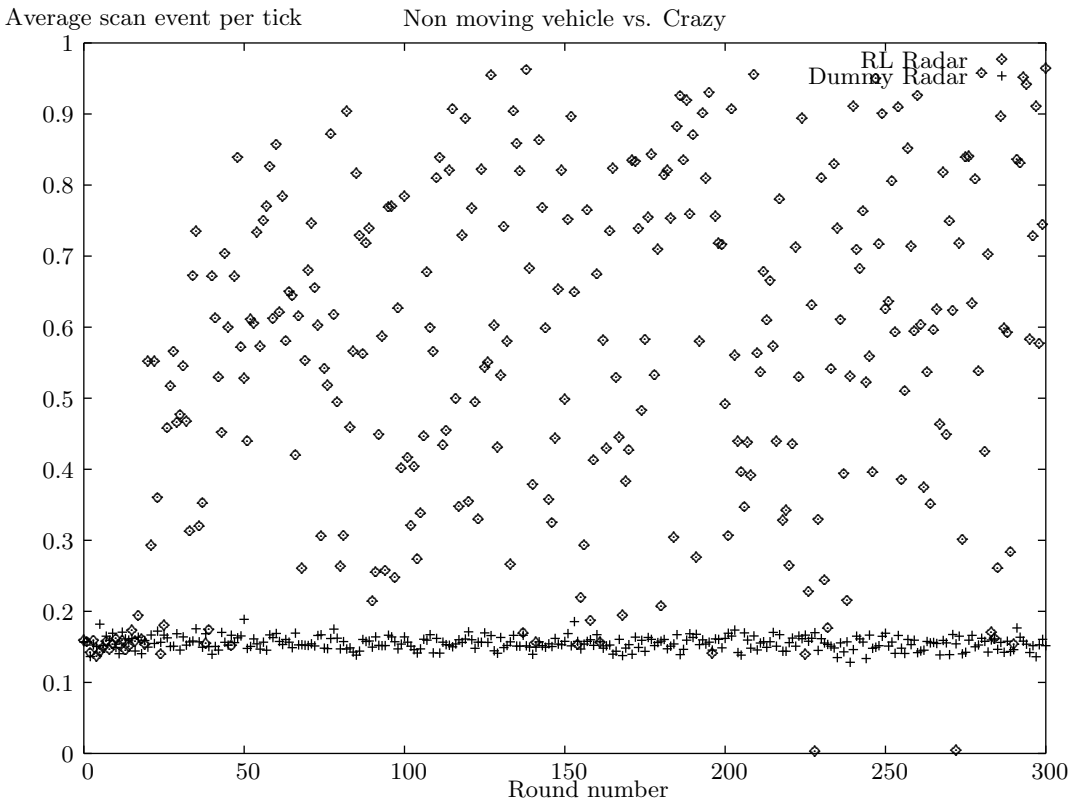


Figure B.2: Shows the first 200 battles of **Non moving vehicle vs. Crazy non fire**

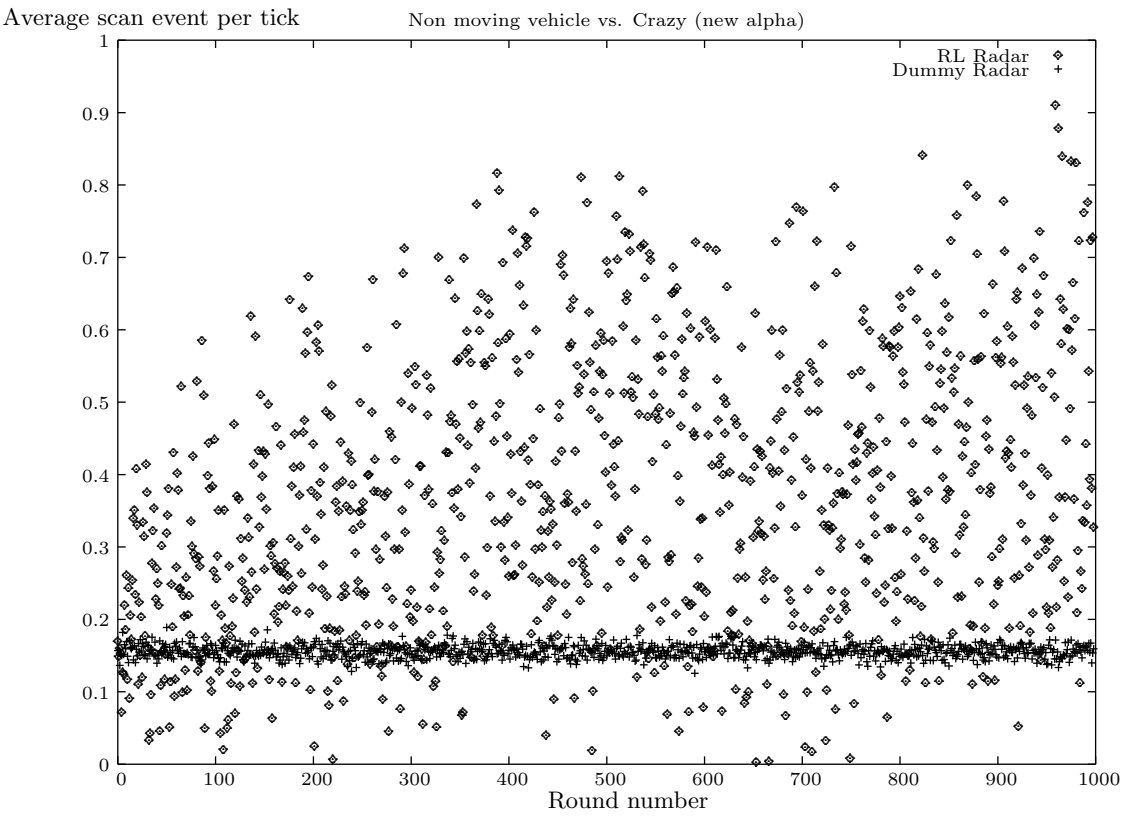


Figure B.3: Shows **Non moving vehicle vs. Crazy non fire** with the Alpha calculated using equation 8.8.

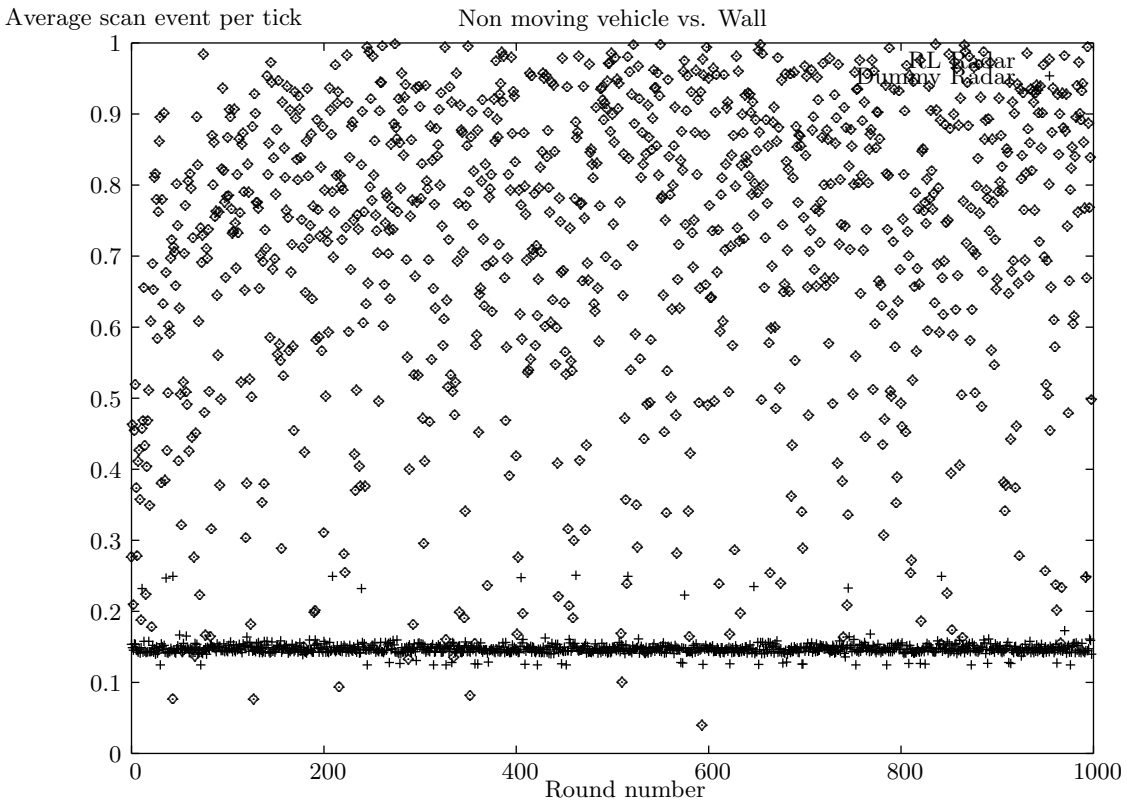


Figure B.4: Shows **Non moving vehicle vs. Wall crawling movement non fire**

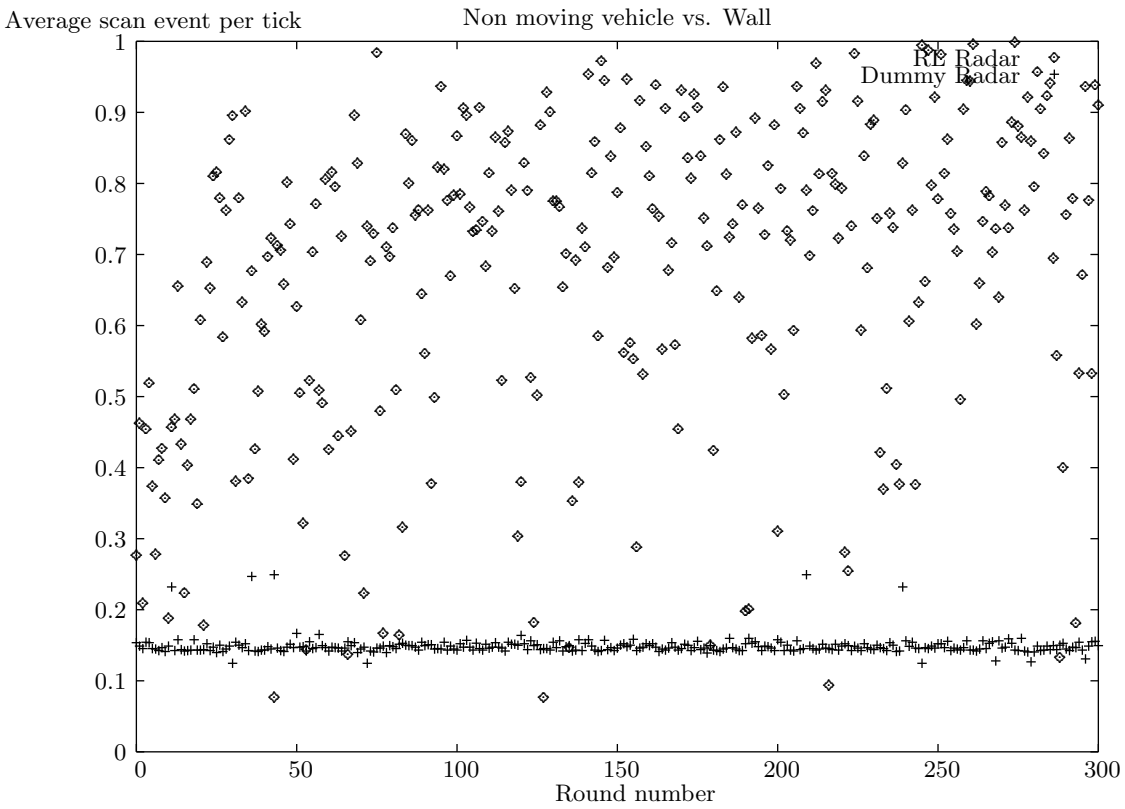


Figure B.5: Shows the first 200 battles of **Non moving vehicle vs. Wall crawling movement non fire**

C Decision Graph Test Plots

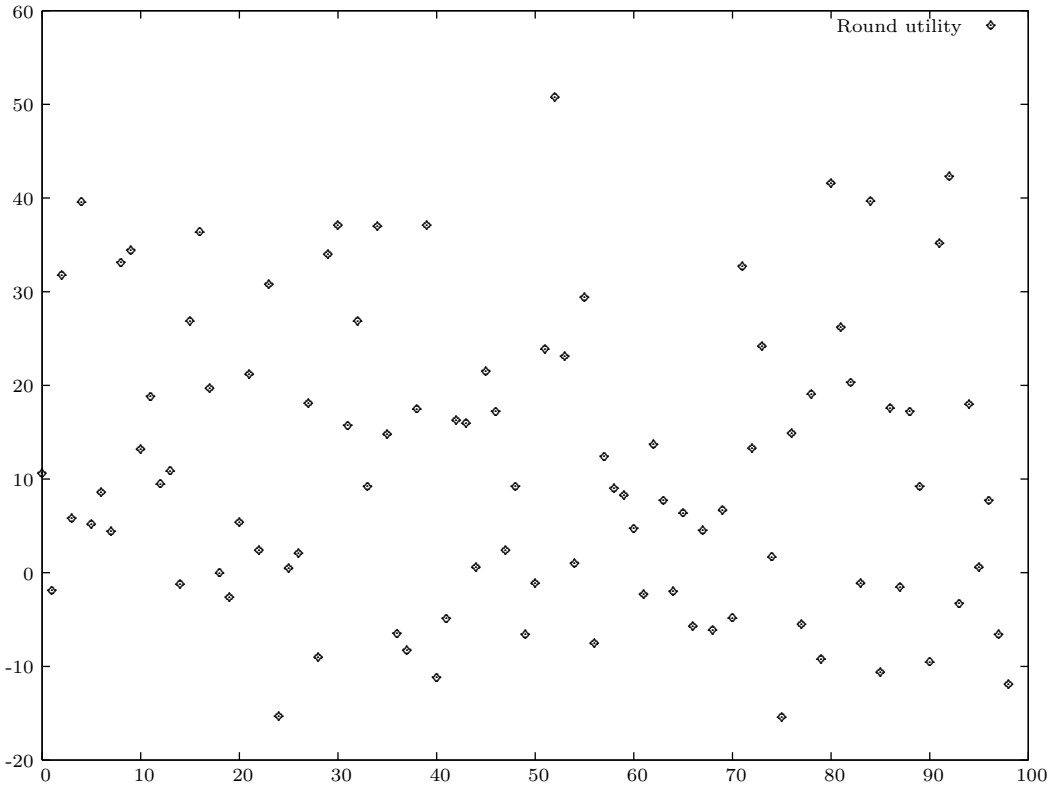


Figure C.1: Plot displaying the spreading of the measurements of the test where $X = 100$ (step 1).

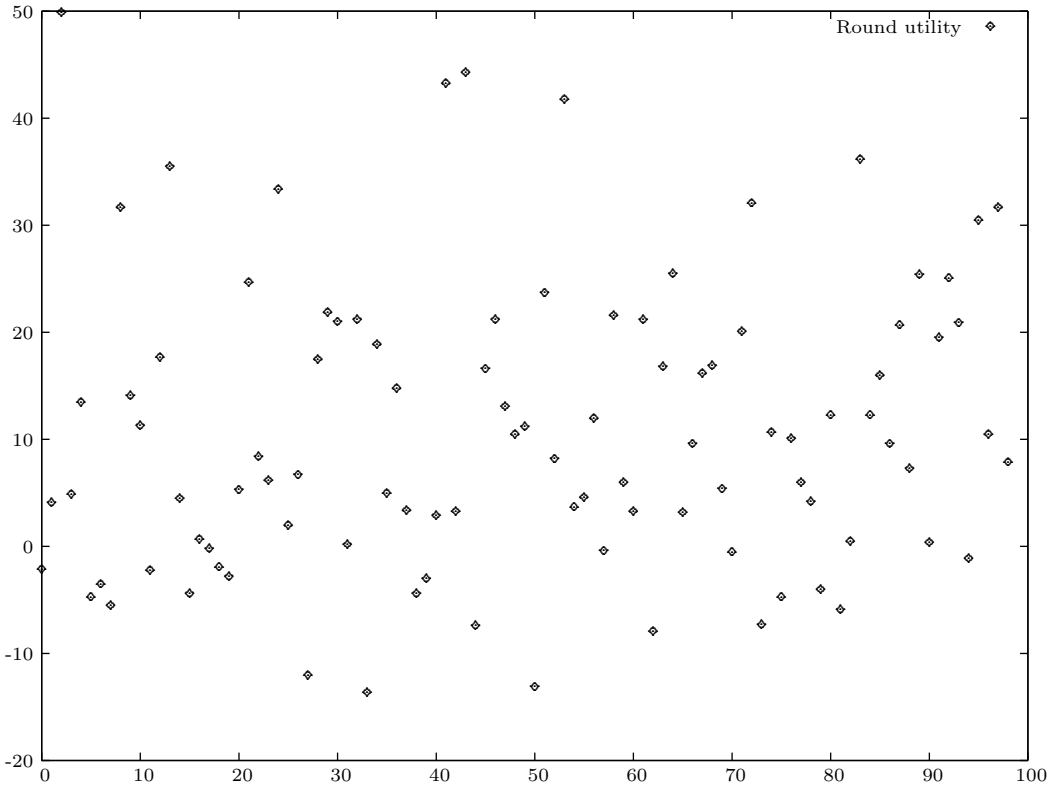


Figure C.2: Plot displaying the spreading of the measurements of the test where $X = 100$ (step 3).

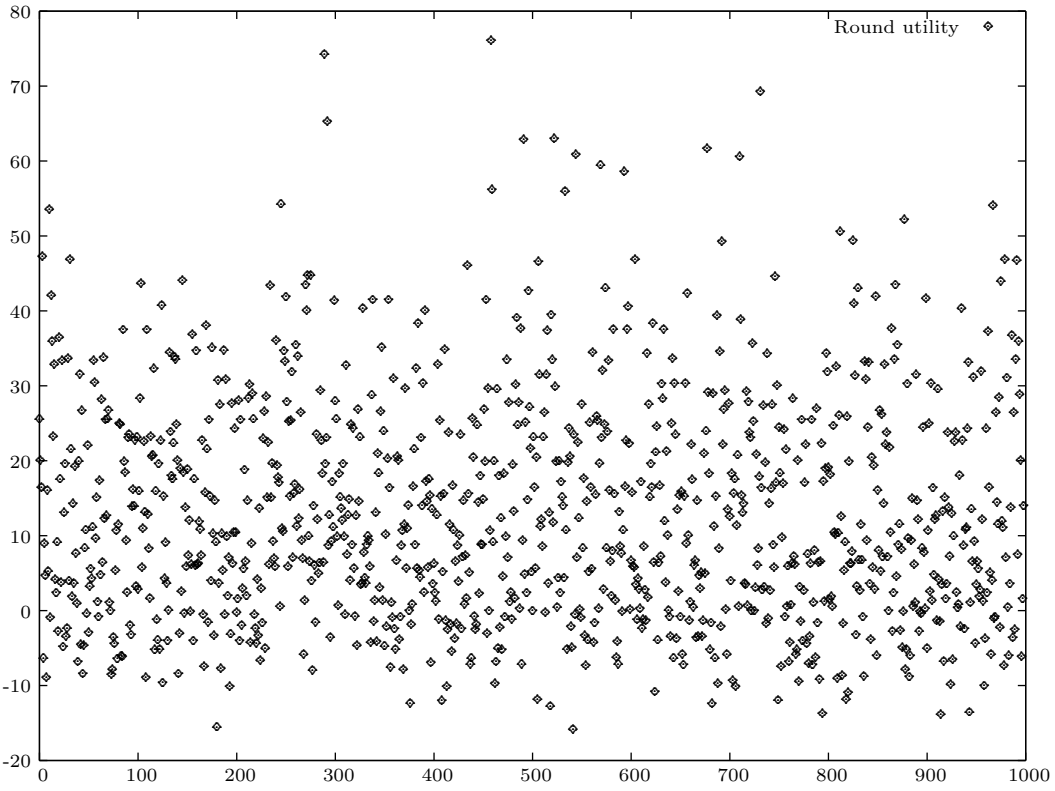


Figure C.3: Plot displaying the spreading of the measurements of the test where $X = 1000$ (step 1).

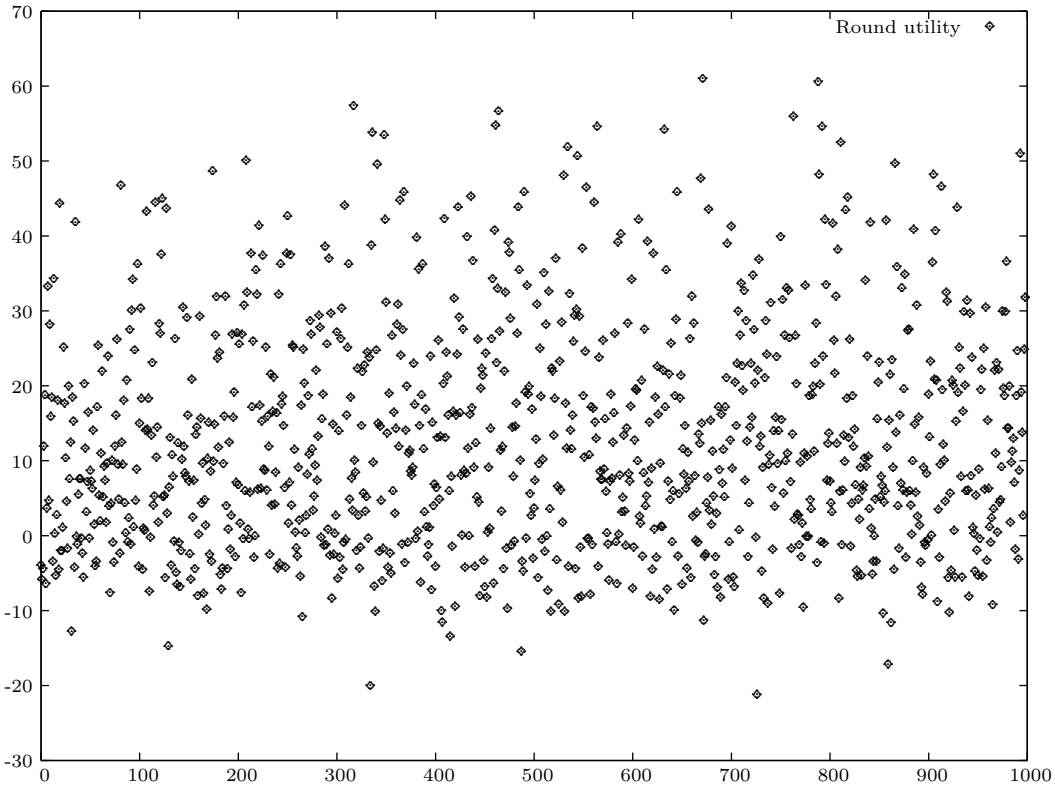


Figure C.4: Plot displaying the spreading of the measurements of the test where $X = 1000$ (step 3).

D Inhibitor Probability Tables

This appendix shows the initial distribution of the probabilities of the inhibitors of the decision graphs for bullet energy decision module. To keep the tables short, we introduce the following abbreviations:

Abbreviation	Meaning
ED	Enemy direction
EDI	Enemy direction inhibitor
G	Gun
GI	Gun inhibitor
DTE	Distance to enemy
DTEI	Distance to enemy inhibitor
EV	Enemy velocity
EVI	Enemy velocity inhibitor
O	Other
OI	Other inhibitor
BV	Bullet velocity
BVI	Bullet velocity inhibitor

Table D.1: Abbreviations and their meaning

The inhibitor tables:

P(EDI|ED)	Towards	Sideways	Away
Hit	0.9	0.6	0.8
Miss	0.1	0.4	0.2

Table D.2: Initial probability distribution of the enemy direction inhibitor.

P(GI|G)	Almost ready	Turn a little	Turn a lot
Hit	0.95	0.9	0.7
Miss	0.05	0.1	0.3

Table D.3: Initial probability distribution of the gun inhibitor.

P(DTEI|DTE)	[0:100[	[100:300[	[300:600[	[600:1200[	[1200:∞]
Hit	0.95	0.8	0.5	0.1	0.05
Miss	0.05	0.2	0.5	0.9	0.95

Table D.4: Initial probability distribution of the distance to enemy inhibitor.

P(EVI|EV)	[0:2]	[3:5]	[6:8]
Hit	0.95	0.6	0.4
Miss	0.05	0.4	0.6

Table D.5: Initial probability distribution of the inhibitor Enemy Velocity.

P(OI|O)	Always on
Hit	0.8
Miss	0.2

Table D.6: Initial probability distribution of the inhibitor Other

P(BVI|BV)	N/A	19.7	18.5	17.0	15.5	14.0	12.5	11.0
Hit	0	0.8	0.7	0.6	0.5	0.4	0.35	0.3
Miss	1	0.2	0.3	0.4	0.5	0.6	0.65	0.7

Table D.7: Initial probability distribution of the Bullet Velocity inhibitor.