Institut für Informatik
Lehrstuhl für Programmierung und Softwaretechnik

LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diploma Thesis

Neuroevolution for Robot Control

Test Framework and Experimental Evaluation

Michael Würtinger

Advisor:	Prof. Dr. Martin Wirsing
Supervisor:	Dr. Matthias Hölzl, Annabelle Klarl, Christian Kroiß
Hand in date:	15. December 2011

Ich versichere hiermit eidesstattlich, dass ich die vorliegende Arbeit selbstständig angefertigt, alle Zitate als solche kenntlich gemacht sowie alle benutzten Quellen und Hilfsmittel angegeben habe.

München, den 15. December 2011

. . . .

(Unterschrift des Kandidaten)

Zusammenfassung

Künstliche Neuronale Netze sind ein Ansatz der Künstlichen Intelligenz. Sie sind frei modelliert nach dem menschlichen Gehirn und wurden als erstes von Frank Rosenblatt im Jahre 1962 eingeführt. Vier Jahre früher stellte Friedberg genetische Algorithmen vor, welche die biologische Evolution nachahmen um Computer Programme zu optimieren. Nachdem das Forschungsgebiet lange in Vergessenheit geraten war, wurden künstliche neuronale Netze 1986 wiederentdeckt und konnten sich auf breiter Front durchsetzen. Heutzutage finden sie in breitgefächerten Gebieten Anwendung, sowohl in der Forschung, als auch in kommerziellen Produkten. Aber selbst nach vielen Jahrzehnten der Forschung ist es immer noch eine Herausforderung Neuronale Netze auf alltägliche Probleme anzuwenden. Die Kombination aus neuronalen Netzen und genetischen Algorithmen wird als Neuroevolution bezeichnet und findet erfolgreich Anwendung bei vielen Problemstellungen. Diese Diplomarbeit konzentriert sich auf die Steuerung von Robotern durch Neuroevolution.

Während die meisten Publikationen nur die Algorithmen veröffentlichen, welche die erwünschten Ergebnisse liefern, vergleicht diese Arbeit systematisch verschiedene Ansätze für die Steuerung von autonomen Robotern. Um dies zu erreichen wird eine Aufgabe und eine passende Umgebung entworfen. Für jedes Experiment wird eine Population von Robotern in die Umgebung gesetzt. Jeder Roboter hat das Ziel so lange wie möglich zu überleben und dabei so viele Ressourcen wie möglich zu sammeln. Der Hauptalgorithmus, der in dieser Arbeit evaluiert wird, basiert auf Neuroevolution. Um die Leistungsfähigkeit zu bestimmen werden zwei zusätzliche Algorithmen entworfen. Der erste Algorithmus steuert den Roboter zufällig durch die Umgebung und soll so eine unter Schranke für die mögliche Leistung aufzeigen. Der zweite Algorithmus ist hand-optimiert und besonders gut auf die Aufgabe und die Umgebung abgestimmt. Es ist leicht ersichtlich, dass dieser Algorithmus nahe an das theoretische Optimum herankommt und er somit verwendet werden kann um die obere Schranke für die mögliche Leistung abstecken zu können. Um die Experimente effizient durchführen zu können wird das BRAIn Framework entworfen und implementiert. Die Algorithmen werden auf der marXbot Roboter Plattform getestet. Um die Umgebung und die Roboter zu simulieren wird der ARGoS Simulator eingesetzt.

Die Ergebnisse zeigen, dass es immer noch schwierig ist, künstliche neuronale Netze in der Robotik anzuwenden. Allerdings zeigt diese Arbeit auch einige Hauptvorteile dieses Konzepts. Nicht zuletzt wird ein Überblick über die Vor- und Nachteilen von neuronalen Netzen gegeben und mit anderen Ansätzen verglichen.

Abstract

Artificial neural networks are an approach towards creating artificial intelligence, loosely modelled after the human brain and first introduced by Frank Rosenblatt in 1962. Four years earlier Friedberg introduced genetic algorithms, which mimic biological evolution to optimize computer programs. Artificial neural networks were rediscovered in 1986 and later became widely accepted. Nowadays they are successfully applied in a wide variety of commercial and scientific fields. But even after many decades of research, applying neural networks to real world problems remains a challenge. This thesis is about the combination of artificial neural networks and genetic algorithms, which is called neuroevolution. This concept has been applied to many real world problems, but this thesis focuses on the control of robots.

Whereas most research papers only present the algorithms producing the desired results, this thesis systematically compares different approaches for the control of autonomous robots. To accomplish this, a foraging task along with an appropriate environment is designed. For each experiment, a population of robots is placed within the environment. Each robot has the goal to survive as long as possible and to collect as many resources as possible. The main algorithm, which is evaluated in this thesis is based on neuroevolution. To assess the performance, two additional robot controllers were designed. The first controller steers the robots randomly through the environment without any goal-oriented behavior and is supposed to delimit the lower bound for the achievable performance. The second algorithm is hand crafted and especially tuned for the defined task. It is apparent that this algorithm performs close to the theoretical optimum and therefore can be used to estimate the upper bound for the achievable performance. In order to conduct these experiments in an efficient manner, the BRAIn framework was designed and implemented. The algorithms were tested on the marXbot robotic platform. The environment, as well as the robots were simulated using the ARGoS simulator.

The results show that it is still difficult and tedious to apply artificial neural networks to robotics. However, the thesis also demonstrates some key benefits of this concept. The thesis is concluded with an overview of the advantages and disadvantages of neural networks and a comparison with other approaches.

It is not my aim to surprise or shock you - but the simplest way I can summarize is to say that there are now in the world machines that think, that learn and that create. Moreover, their ability to do these things is going to increase rapidly until - in a visible future - the range of problems they can handle will be coextensive with the range to which the human mind has been applied. — **Herbert Simon, 1957**

Acknowledgements

I would like to thank Prof. Dr. Martin Wirsing for offering me the opportunity to conduct research in the field of robotics and evolutionary algorithms. Special thanks go to Dr. Matthias Hölzl, Annabelle Klarl and Christian Kroiß for supervising my research and for supporting me in lengthy meetings and through countless emails.

Furthermore I want to thank Carlo Pinciroli from the Université libre de Bruxelles for developing the excellent ARGoS simulator, which I used intensively during my research, as well as for supporting me whenever I had trouble with the software.

Last but not least my gratitude goes to the Leibniz-Rechenzentrum for supplying me with thousands of hours of computing time on their SuperMUC Petascale System.

Contents

Chapter 1

Introduction

After the establishment of artificial intelligence in 1956 the research was coined by enthusiasm and great expectations [RN03]. Among the broad variety of developed algorithms were artificial neural networks (introduced by Frank Rosenblatt in 1962) and genetic algorithms (introduced by Friedberg in 1958). Just one decade after the formation of this research field, researchers realized that the developed algorithms were unable to fulfill the expectations when applied to real life problems. This setback lead to a depression and it wasn't before 1980 that the first solutions became commercially available. Although abandoned in the late 1970s, neural networks were rediscovered in 1986 and are now successfully applied in a wide variety of fields. But even after many decades of research and a great amount of successful applications, it still remains challenging to use ANNs in new projects. The sheer number of different network types, learning algorithms and pre- and post-processing algorithms makes it difficult to get started.

Robotics is the research field of the intelligent connection between perception and action [SK08]. The word *robot* was first introduced in the Czech play *Rossum's Universal Robots* in 1920. It is derived from the Slav *robota* and means subordinate labour. As mentioned before, the field of artificial intelligence dates back to the middle of the twentieth century. Around the same time the first actual robots were created. Early robots were entirely controlled by humans and had only few sensors. After the invention of integrated circuits, researchers had the tools which were necessary to make robots more intelligent. One of the earliest applications of this new generation of robots was in industrial manufacturing. Industrial robots are an example for so called manipulators, because they are capable of manipulating their environment through actuators, but are stationary [RN03]. Mobile robots, on the other hand, are not necessarily capable of manipulating their environment, but are not bound to a certain location. Unmanned land vehicles (ULV), for example, are capable of moving around using legs, wheels or similar devices. Popular ULV examples include autonomous cars or military robots. Similarly, unmanned air vehicles (UAV) and autonomous underwater vehicles (AUV) are capable to move through air or water.

All of these types of robots have in common, that they are hard to program. The more actuators and sensors a robot has, the harder it gets. Many scenarios expect robots to adapt to new environments or even to learn entirely new behavior. A car manufacturing robot, for example, might have to adapt to new car models from time to time. For many real world applications, it is sufficient to program the required behaviors manually, but often this is impossible, since environments might simply change too often or might even be unknown to the engineers, programming the robot.

1.1 Motivation

To solve these problems, the machine learning research field, a branch of artificial intelligence research, has been established [RN03]. This research area deals with machines, which are capable of learning new behavior and adapting to environmental changes [Alp04]. This research is not limited to robotics, but is applied to a wide range of areas. A search engine, for example, might learn what a user typically searches for and use this information to present more relevant results. An online shop might be capable of learning a user's consuming behavior to offer more appropriate products or to present targeted advertising. However, one of the most interesting applications of this technology is robotics, because is is often hard or impossible to implement the desired behavior.

Many different learning algorithms exist for controlling robots. A particularly interesting approach are artificial neural networks (ANN), which utilize a simplified model of the mechanisms used by the human brain [Hay08]. What makes this approach so interesting is that the human brain is capable of performing a vast variety of different tasks. It is capable of processing image and audio signals, extracting the relevant details of massive amounts of data and above all to learn new behavior. It seems tempting to exploit these mechanisms for robots and other applications.

What makes ANNs powerful is the capability to learn and adapt [Hay08]. To achieve this behavior, learning algorithms are required. One particularly interesting algorithm is neuroevolution, which is a combination of artificial evolution and neural networks [MD89]. This type of algorithm automatically evolves neural networks for a particular task and environment. The advantage of this is that one only has to define the desired behavior in an abstract way and the algorithm optimizes the ANN as much as possible to fulfill the requirements. However, it still is challenging to apply neuroevolution, or ANNs in general, for robotics and the achieved performance is often lower than expected. As ANNs in general and neuroevolution algorithms in particular have many advantages and disadvantages, compared to alternatives, it is interesting to compare them to other approaches in a systematic manner.

To compare different algorithms, one needs a carefully selected scenario. In the course of this thesis, a foraging task, along with an appropriate environment is designed. Foraging is defined as "wander[ing] in search of food or provisions" [for]. As this is a fundamental task of most animals, it is often used in artificial intelligence experiments.

1.2 Objectives

The goal of this thesis is to compare the performance of a randomly operating algorithm (random walk), a neuroevolution algorithm and a hand crafted special purpose algorithm for different configurations.

To accomplish this, BRAIn (BRAIn Robot Algorithm Insight), a scalable and flexible framework, which simplifies experiment execution and analysis, is designed and implemented. The framework's architecture provides maximum flexibility. It is independent of concrete simulator implementations or experiments. To improve statistical significance, the framework supports multi experiment execution and result aggregation.

The algorithms are tested using the marXbot robotic platform and the ARGoS simulator. To compare the algorithms, an environment and a task is designed and implemented for ARGoS. The environment contains several resources and a base. Multiple robots, which have to collect energy from the resources and drop it of in the base, are placed in the environment. The goal for the robots is to transfer as much energy as possible from the resources to the base. The environment provides flexible configuration options to support many different scenarios. The task is tuned in a way that it is both simple enough for various types of algorithms, as well as challenging enough to encourage complex algorithm behavior.

To be able to compare the algorithms a fitness evaluation methodology is designed, which provides a simple way of ranking different algorithms, according to their behavior during experiments. The results of the experiments are presented and analyzed. The different algorithms are compared and their performance is discussed, focusing on the advantages and disadvantages of the approaches.

1.3 Outline

The required theoretical and conceptual foundations are introduced in Chap. 2. It starts by giving a general overview of intelligent agents and their environments, continues by introducing different learning approaches and presents a broad overview about artificial neural networks and genetic algorithms.

Chap. 3 presents the robotic platform and the simulation software used for the experiments in this thesis.

After establishing the theoretical foundations, Chap. 4 presents the BRAIn framework, which was developed to conduct the above mentioned experiments of this thesis. After introducing the requirements for the software, the architecture and some key implementation details are outlined.

Chap. 5 contains a detailed description of the foraging task and the environment. It also describes all of the five experiments conducted using the BRAIn framework, the marXbot platform and the ARGoS simulator. The chapter introduces three different algorithm types, which are then applied to a common task to explore their performance characteristics. The results of all of the experiments are also presented in this chapter.

At last, Chap. 6 compares and discusses the collected experiment results. The chapter gives an overview of the performance characteristics, which can be drawn from the results. The chapter is concluded by giving an outlook to possible future extensions of this work.

Chapter 2

Foundations and Related Work

The purpose of this chapter is to introduce all the concepts needed to understand this thesis. Circumstantial background information is provided where appropriate. The chapter starts by giving a general overview about intelligent agents and learning approaches before explaining the structure and functionality of the human brain, thus smoothing the way for the introduction of artificial neural networks. Next, genetic algorithms and their application to neural networks are explained. The last part of this chapter covers controlling robots with neuroevolution.

2.1 Agents and their Environments

This section, which is based on Chap. 2 of *Artificial Intelligence: A Modern Approach* [RN03], introduces the general concept of intelligent agents, how they interact with the environment and what environments typically look like.

2.1.1 Intelligent Agents

An agent is an entity which is capable of perceiving its environment using sensors as well as acting on its environment using actuators. This principle is presented in Fig. 2.1. A human is an example for an agent, which uses eyes, ears and further sensory organs to perceive the environment and feet, hands and other actuators to act appropriately. A typical robot has at least cameras and distance sensors and acts using electric motors. Agents are not necessarily tangible objects, but can also exist entirely in software. An Internet search engine, for example, perceives its environment through user inputs and acts by returning search results.

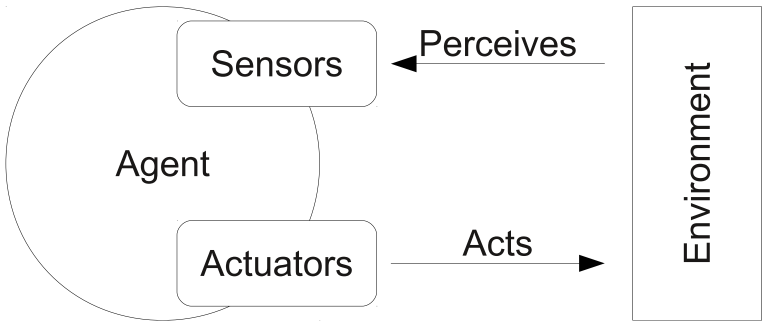


Figure 2.1 – Schematics of an intelligent agent (adopted from [RN03], p. 33).

Agents usually can perceive their own actions, but not necessarily the effects of these actions on the environment. The entire history of what an agent perceived is called the *percept sequence*. An agent can be mathematically described by an *agent function*, which determines the agent's action based on its entire percept sequence. Therefore, the behavior of an agent can be fully described by defining the action for every possible percept sequence. As these sequences can have arbitrary length, it is virtually impossible to fully describe an agent function this way. A concrete implementation of an agent function is called an *agent program*.

When designing an agent, the goal is to get the right behavior. But which agent functions are good and which are bad? *Performance measures* are usually applied to determine the answer to this question. The performance measures depend on the type of task the agent is supposed to fulfill. If designing an agent capable of driving a car, for example, one might base the performance measures on fuel consumption, journey time and the degree of complying with the road traffic act. Generally speaking, an agent does the right thing if it is successful at accomplishing the task it is supposed to do.

2.1.2 Environments

The environments agents perceive and act on can be very diverse. They might be as simple as a board game or as complex as a busy road with lots of other cars, pedestrians, traffic signs and lights. To better understand environments, they can be classified according to several properties, which are now introduced. These properties are used in Chap. 5.1 to classify the environment designed for this thesis.

If an agent can perceive the entire state of the environment at any point in time it is called *fully observable*, otherwise it is called *partially observable*. An environment might not be fully observable, because the agent's sensors are not capable of measuring all the relevant inputs or because the sensor readings are noisy. Often it is also the case that aspects of the environment are hidden and thus cannot be observed. A fully observable environment simplifies the task of the agent because it does not have to explore and it does not have to keep track of any previously encountered facts.

An environment is called *deterministic*, if its state only depends on its previous state and the actions of the agent. If this is not the case, the environment is called *stochastic*. Driving a car is an example for a stochastic environment, because engine failures might occur without premature warnings and tires might blow from time to time. If an environment is deterministic, with the exception of the actions of other agents, it is called a *strategic* environment.

If an environment is subdivided into independent tasks, consisting of a set of inputs and a single action, it is called *episodic*. In episodic environments, the actions of the agent are only based on the inputs of the current episode, but not on any previous episodes. An example for an episodic environment is a classification agent, the task of which is to determine whether parts on an assembly line are good or not. The opposite of episodic is *sequential*. Driving a car is a typical example for a sequential environment.

Environments can either be *static* or *dynamic*. The state of a static environment does not change if the agent is not acting on it. This has the advantage that the agent does not have to keep track of the environment while idling and that it can take arbitrarily long to make a decision. However, most environments are dynamic, which means that while the agent is deciding what to do next, it is interpreted as doing nothing, which might have a negative effect on the agent's performance. Driving a car is also an example for a dynamic environment. If the car in front of the agent's car brakes, the agent's car crashes, if it does not decide to act accordingly in a timely manner.

If an environment has only a finite amount of states, it is called a *discrete* environment, otherwise it is called a *continuous* environment. An example for a discrete environment is a simple board game, the state of which solely consists of the positions of the gaming pieces on the board. Driving a car, on the other hand, clearly is not a discrete environment, because the possible positions and velocities of other cars on the road are infinite. Strictly speaking, any form of simulated environment is discrete, because computers cannot represent continuous values. However, as the amount of possible states in a simulated 3D environment are virtually endless, these environments are also classified as *continuous*.

If an agent is acting on an environment together with other agents, it is called a *multiagent* environment. If the agent is alone in its environment it is called *single agent*. However, it sometimes is not clear whether other entities should be treated as agents or simply as stochastically behaving objects. If an agent is driving a car on a busy road, are the other vehicles agents or not? The answer of this question depends on whether the other entities influence the agent's performance measures or not. In the driving example this is typically not the case because the performance of the agent (arriving on time, obeying traffic rules, optimizing fuel consumption) in general does not depend on the behavior of other vehicles. If the performance measure is extended to include "avoid collisions", however, it depends on the behavior of other drivers, which are therefore referred to as agents. An environment, where a key aspect of the performance measures (avoiding collisions) leads to an improved performance for *all* of the agents, is called a *cooperative* multiagent environment. In a car race, however, it is the other way around, because the performance (winning the race) of one agent increases, the performance of other agents decreases. Such an environment is called a *competitive* multiagent environment.

2.2 Learning Approaches

To make intelligent agents powerful, they have to be able to adapt to environmental changes. To allow this, it is important to utilize appropriate learning algorithms. These algorithms are not only used to allow robots to adapt to their environments, but also to learn entirely new behaviors, that might be too complex to be specified manually. Just like humans, artificial agents can learn in various different ways [Hay08]. Therefore, learning algorithms can be divided into different classes, which are introduced in the following sections.

2.2.1 Learning with a Teacher

In this type of learning algorithms, which is also known as *supervised learning*, the agent is accompanied by a *teaching instance* [Hay08]. This instance has built-in knowledge about the desired functionality. The learning process is iterative. In each iteration, an example input vector is presented to the agent and the teacher. The agent then calculates an output according to its current knowledge and the teacher produces the desired output. The difference between the agent's output and the desired output, which is referred to as error signal, is then fed to the agent. By using this signal, the agent can gradually improve its behavior. Eventually, the agent learns to emulate the teacher, which is then no longer necessary. The set of example input values, along with their desired output values, is called the *training set* of the learning process [RN03].

2.2.2 Learning without a Teacher

In many cases learning with a teacher is impossible, because no preliminary knowledge about the target function is available. If this is the case, a separate class of algorithms can be used, which is known as *learning without a teacher* [Hay08]. This group of algorithms can be subdivided into *reinforcement learning* and *unsupervised learning.*

Reinforcement Learning

The algorithmic class of *reinforcement learning* is the most general of all learning algorithms and can be applied to many real world problems [RN03]. Reinforcement learning is based on a cost function, which calculates the cost for a sequence of actions [Hay08]. This cost function acts as a rewarding system. If the agent behaves correctly, the cost goes down, if it behaves in an undesirable way, the cost goes up. It is important that the cost function considers a whole sequence of actions instead of a single one, because expensive actions might lead to reduced cost in the future. The challenging part of this type of algorithm is the correct mapping between a calculated cost and individual actions. It is difficult to identify the actions, which are responsible for high cost and the actions, which are responsible for low cost. Another challenging part of this algorithm is the definition of the cost function itself. This function has to be chosen carefully, as it directly determines the success or failure of the application.

Unsupervised Learning

The process of discovering patterns in the input data is called unsupervised learning [RN03]. No desired output values are supplied to this type of algorithm and the agent is unable to observe its environment. Therefore, the only possibility to learn is to look at the data fed to the agent and to determine, whether it is possible to divide it into some sort of classes. It is important to note, that an agent, which is trained entirely unsupervised is unable to determine how to act, because it is unable to perceive the consequences of its outputs and has no information about what is desirable and what is not.

2.3 Artificial Neural Networks

This section, which is based on the introduction, as well as Chap. 1 and 4 of *Neural Networks and Learning Machines* [Hay08], introduces the basic concepts behind artificial neural networks. The idea to use neural networks for computations was first introduced by McCulloch and Pitts in 1943. Research in this field has been motivated by the fact that the human brain - and other nervous systems - process information in an entirely different way than Von-Neumann machines do. This section introduces the concepts behind neural networks and their computational capabilities.

2.3.1 The Human Brain

The human brain consists of an enormous amount of nervous cells. These cells, which are called neurons, were first discovered by Ramón y Cajál in 1911. A neuron basically

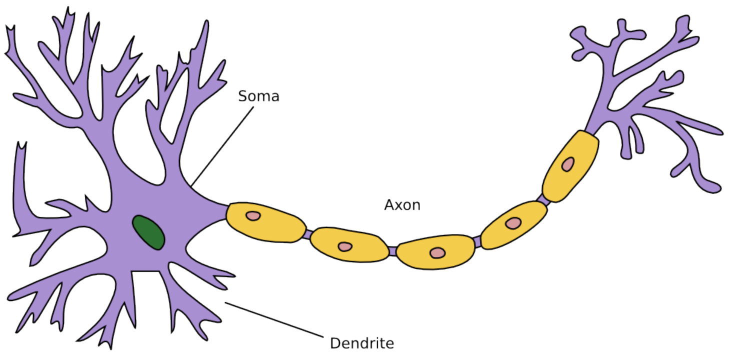


Figure 2.2 – Schematics of a biological neuron (©Quasar Jarosz, License: CC-BY-SA-3.0).

consists of four parts, which are illustrated in Fig. 2.2. The *soma* (also called cell body) is the central component of the neuron and connects all the other parts. The body serves as the origin for the so called *dendrites* and the *axon*, a long and thin part of the cell which forks into a great amount of *synaptic terminals* at its end. Both ends of the cell are tree like and can contain thousands of branches. While the dendrites collect the cell's inputs and transmit it to the body, the axon and its synaptic terminals are responsible for broadcasting information to other cells. Therefore, the cell is divided into a receiving and a transmitting end. Neurons can exchange information if they are connected through a so called synapse, which is formed by one cell's axon and another cell's dendrite. A typical neuron (the *pyramidal cell*) can receive data from about ten thousand other neurons and can send data to thousands of receivers.

There are different types of synapses, but the most common one is the chemical synapse, which converts the electrical signal of one cell's axon into a chemical transmitter substance, which is then received by the dendrite of another cell and converted back into an electrical signal [Hay08, SK90]. Instead of communicating with steady potentials, the output of neurons is typically encoded as a series of voltage spikes. This can be explained using electrical engineering. Axons can be seen as a long cable having high resistance and high capacity. If a potential were applied to one end of the axon it would quickly drop while propagating through the line until it would be undetectable. Short spikes, however, are an elegant way to circumvent this problem. It was discovered that inter-neuron communication actually is based on precise timing between the spikes sent by a neuron [Maa97]. This concept was adopted into so called spiking neural networks, a special type of artificial neural networks which is more closely modelled after biological neural networks.

Neurons operate in the time frame of milliseconds, which translates to a clock rate in the kilo Hertz range [Hay08, Mor98]. State of the art micro processors, on the other hand, operate with up to 4GHz. Therefore, neurons are approximately six magnitudes slower than current silicon circuits. What makes the brain so powerful is massive parallelization. There are approximately 10^{10} neurons in the human cortex, which are connected to each other with about $6 \cdot 10^{13}$ synapses. The overall axon length in a twenty year old man's brain was found to be 176,000 km [LJYB03].

2.3.2 Modeling a Neuron

As researchers realized how powerful neural networks like our brain can be, they tried to learn more about their unique architecture, which enabled them to do complex tasks, like image processing, much more efficient than any man made computer. The goal was and still is to exploit some of the concepts to make computers more powerful. This is especially eligible for tasks, which are hard to accomplish with conventional algorithms.

To emulate biological neural networks in software or on dedicated silicon hardware one first needs a suitable model of all of its elements. As mentioned in the previous section, a neural network's most important components are neurons and synapses, which is why most models concentrate on these parts. The number of components is not only reduced, but the components themselves are also simplified greatly.

Many models exist, each having different computational complexity and capabilities [Maa97]. One of the simplest models consists of a list of connections, each of which has its own weight, an adder which sums up all of the weighted inputs collected by the connections and an activation function (see Fig. 2.3). The purpose of this function is to limit the output of the artificial neuron to certain boundaries, thus ensuring that other neurons can process the values correctly. The model can be summarized using Eq. 2.1, 2.2, 2.3 where y_k is the output, x_{kj} are the inputs, w_{kj} are the connection weights, b_k is the bias and φ is the activation function of neuron k.

$$u_k = \sum_{j=1}^{m} w_{kj} x_{kj} \tag{2.1}$$

$$v_k = u_k + b_k \tag{2.2}$$

$$y_k = \varphi(v_k) \tag{2.3}$$

In this model, the neuron first applies a weight factor w_{kj} (which can be positive or negative) to each of its inputs x_{kj}. The resulting values and the neuron's bias are then summed up to form the input v_k for the activation function, which produces an output value y_k within a certain range. All three equations can be combined into Eq. 2.4:

$$y_k = \varphi\left(\sum_{j=1}^{m} w_{kj} x_{kj} + b_k\right) \tag{2.4}$$

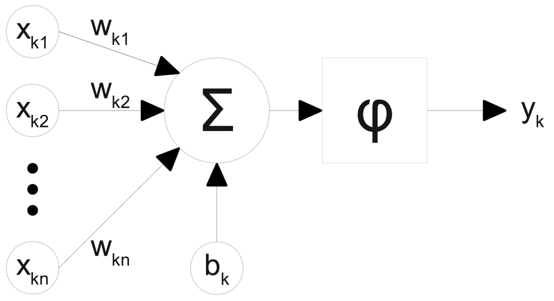


Figure 2.3 – A simple model of a neuron consisting of various inputs, weighted connections, an adder, an activation function, a bias and one output (adopted from [Hay08], p. 41).

It is worth pointing out that neurons, using this simple model, are stateless. This limits the computational power, because it is not possible to produce outputs based on the development of input signals over a certain period of time. As mentioned in Sec. 2.1.1, agents usually act according to the entire percept sequence. However, this simple model allows agents to act only according to the current sensor readings. Recurrent neural networks are an example for a more advanced type of neural network, which is stateful and therefore more powerful [WZ89].

The argument of the activation function v_k is also referred to as the induced local field or the activation potential. Applying bias b_k is effectively an affine transformation to the output u_k. To simplify the model even further, the bias is often removed and replaced by an additional input x_{k0} with a fixed value of $+1$ and an additional weight w_{k0}. According to the following equation, w_{k0} successfully replaces the bias b_k.

$$b_k = x_{k0} w_{k0} = +1 \cdot w_{k0} = w_{k0} \tag{2.5}$$

As mentioned previously, the main task of the activation function is to limit the neuron's output to a specific range. A broad variety of possible activation functions exists in literature. The two most widely used are the threshold and the sigmoid function.

The threshold function, which is defined in Eq. 2.6, is simple, easy to implement and fast to compute. The output of a neuron using this activation function can either be 1 or 0 (see Fig. 2.4(a)), depending on whether the input value is above or below the threshold. Neural networks using threshold functions were first introduced by McCulloch and Pitts in 1943.

$$\varphi\left(v_k\right) = \left\{ \begin{array}{ccc} 1 & : & v_k \geq 0 \\ 0 & : & v_k < 0 \end{array} \right. \tag{2.6}$$

In most cases, the threshold is 0, but other thresholds can be used as well. However, a threshold of 0 is ubiquitous, as neural networks are using a bias value. If extending Eq. 2.6 with Eq. 2.2, one gets Eq. 2.7, which can be transformed into Eq. 2.8. Therefore, the bias value successfully replaces the variable threshold thus simplifying the equation.

$$\varphi\left(u_k + b_k\right) = \left\{ \begin{array}{ccc} 1 & : & u_k + b_k \geq 0 \\ 0 & : & u_k + b_k < 0 \end{array} \right. \tag{2.7}$$

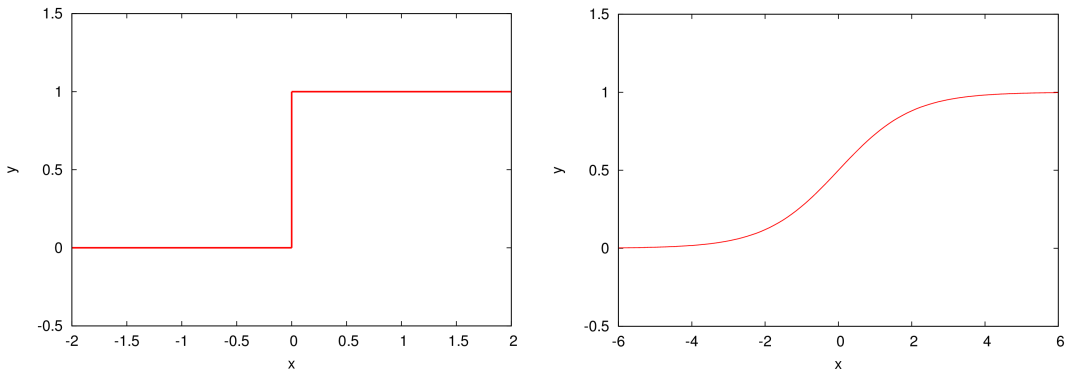
$$\varphi\left(u_k + b_k\right) = \left\{ \begin{array}{ccc} 1 & : & u_k \geq -b_k \\ 0 & : & u_k < -b_k \end{array} \right. \tag{2.8}$$

The most widely used activation functions are the so called sigmoid functions, which are a class of strictly increasing functions with an "S" shaped plot (see Fig. 2.4(b)). A popular example for a sigmoid function is the logistics equation, which is defined in Eq. 2.9. The results of this equation lie within $[0; 1]$ [Hay08, MMMR96]:

$$P(t) = \frac{1}{1 + e^{-t}} \tag{2.9}$$

Usually an additional constant s is added, which controls the steepness of the function. Doing this forms the most commonly used activation function [Hay08, MMMR96]:

$$\varphi\left(v_k\right) = \frac{1}{1 + e^{-st}} \tag{2.10}$$



(a) Threshold function. (b) Sigmoid function.

Figure 2.4 – Two very common activation functions: The threshold function (a) and the sigmoid function (b).

In many cases it is desirable to have activation functions with an output range of $[-1;+1]$. The corresponding version of the threshold function is

$$\varphi\left(v_k\right) = \begin{cases} +1 & : \quad v_k > 0 \\ 0 & : \quad v_k = 0 \\ -1 & : \quad v_k < 0 \end{cases} \tag{2.11}$$

and the corresponding sigmoid function is:

$$\varphi\left(v_k\right) = \tanh(v_k) \tag{2.12}$$

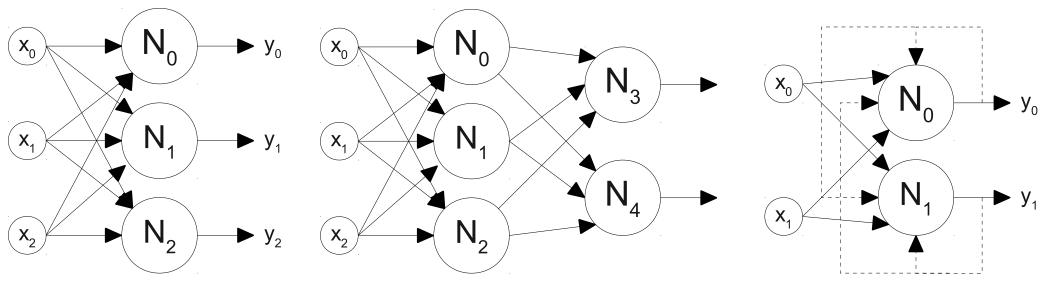
2.3.3 Neural Network Architectures

The architecture of biological neural networks differs greatly. Primitive invertebrates like insects seem to have completely hard wired nervous systems, where the purpose of each neuron, as well as its connections to other neurons, are completely predetermined in the animal's genome [Mor98]. The nervous system of humans, on the other hand, consists of such a vast amount of components that it is simply impossible to encode its entire structure in the genome, which is in the order of 10^9 bits long. It appears that most of the human brain, instead, consists of regular structures without predetermined purpose, which are later allocated and modified as new skills are learned.

When designing artificial neural networks, structure plays an important role [Hay08]. Most of the networks used are static in a way that only connection weights, but not the overall network graph is allowed to change. The following types of networks are commonly used.

Stand-alone Neuron

Thinking about the enormous size of most biological neural structures, it might seem that one neuron on its own cannot accomplish much. Frank Rosenblatt, however, dedicated a great amount of his work to this type of neural "network" which he called *Perceptron*. The theoretical background as well as the perceptron's capabilities are discussed in Sec. 2.3.4.



(a) single (b) multi (c) recurrent

Figure 2.5 – A single layer (a), a multi layer (b) and a recurrent (c) neural network (adopted from [Hay08], p. 51ff).

Feedforward Networks

If several neurons are combined to a neural network the units are usually arranged in the form of layers. One of the layers is usually the dedicated input layer, the sole purpose of which is to collect information from the outside world. This input layer is not a real neural layer as it does not perform any computations. It merely forwards its current state to all connected neurons of the next layer.

In most layered networks neurons within the same layer are not connected at all, but all neurons within one layer are connected to all (or some) neurons of the next layer. If this is the case and if all connections are oriented in the same way, thus allowing information to flow only in one direction (towards the output layer), the network is called a *feedforward network*. If each neuron is connected to all neurons in its consecutive layer the network is called *fully connected*, otherwise it's called *partially connected*.

If a feedforward network has only one input layer as well as one layer of real neurons, it is called a *single layer feedforward network*. This originates from the fact the the input layer does not perform any computations and is therefore usually not considered when counting the layers. If the network contains more than one layer of real neurons, it is called a *multilayer feedforward network*. Figures 2.5(a) and 2.5(b) show both network types. Layers, which are neither input nor output layer are referred to as *hidden layers* because they are not visible from either side. By adding hidden layers the network can become more powerful.

Recurrent Networks

While the distinguishing feature of feedforward networks is that information flows only in one direction (towards the output layer), recurrent networks are allowed to contain so called feedback loops, which transfer calculated results in the opposite direction. The concept of a neuron feeding its output directly back into its own input is called *self feedback loop*. Unlike feedforward networks, recurrent networks can have a state, which makes them applicable to a much wider area of problems, but also much harder to understand.

2.3.4 Computational and Learning Capabilities of Neural Networks

Before being able to apply neural networks to any problems, it is important to understand their computational and learning capabilities. This section introduces several learning algorithms and explores the capabilities of single neurons and multilayer feed-forward networks.

Rosenblatt's Perceptron

Applying learning algorithms to neural networks was first introduced by Frank Rosenblatt in 1958, using the so called *Perceptron* [Ros58, Hay08]. Rosenblatt's model is based on a single neuron using the McCulloch Pitts model (i.e. having a threshold activation function). Several inputs are multiplied with their corresponding weights, then summed up together with the bias and fed into the threshold activation function, producing either -1 or $+1$ (see Eq. 2.4 and 2.11). While the threshold function in Eq. 2.11 can also produce a result of 0, the activation function is often altered in a way that the 0 case is assigned to the positive or negative case, thus reducing the number of possible outputs to two.

The Perceptron is capable of classifying the applied inputs $x_1, \ldots, x_m$ into one of two classes C_1 and C_2, which are indicated by either $+1$ or -1. In the Perceptron's most elementary form, the decision regions of these two classes are separated by a hyperplane, defined by:

$$\sum_{i=1}^{m} w_i x_i + b = 0 \tag{2.13}$$

A hyperplane is the generalization of a plane for an arbitrary amount of dimensions [Cur84], p. 73. A hyperplane of an m-dimensional space is a $m-1$ dimensional subset. An $m-1$ dimensional hyperplane divides an m-dimensional space into two half-spaces [DT03], p. 8.

For the case of $m = 2$, the hyperplane becomes 1-dimensional and therefore becomes a straight line, separating both classes as indicated in Fig. 2.6. All the points below the line are part of class C_1 and everything above the line is part of C_2.

The perceptron is capable of reliably classifying any set of input data as long as it is linearly separable, which means that there has to exist a hyperplane which successfully divides the input sets into the classes C_1 and C_2.

To allow the perceptron to work correctly its weight vector $\vec{w} = b, w_1, \ldots, w_m$ has to be set to the right values. This can be achieved automatically by presenting training data to the iteration based *perceptron convergence algorithm*. Given that the data is linearly separable, an upper bound can be provided for the number of steps necessary

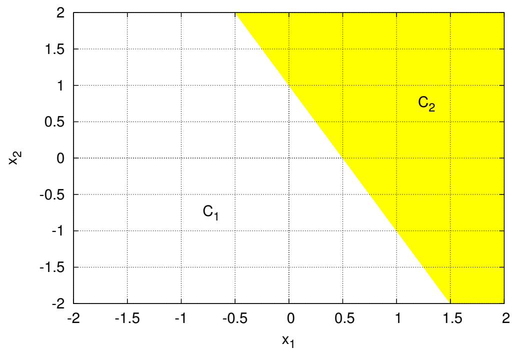


Figure 2.6 – Classification of two dimensional input data.

for the algorithm to converge. While explaining the algorithm the following conventions are used [Hay08]:

- Input vector of step n: $\vec{x}(n) = [+1, x_1(n), \ldots, x_m(n)]^T$
- Weight vector of step n: $\vec{w}(n) = [b, w_1(n), \ldots, w_m(n)]$
- Actual response of the perceptron in step n: $y(n)$
- Desired response in step n: $d(n)$
- Learning rate: $0 < \alpha < 1$

The algorithm consists of three simple steps, which are executed until a convergence criterion is reached. Each iteration of the algorithm uses one pair of sample input value and desired result. The entire algorithm is presented in Fig. 2.7. The first step is to initialize the perceptron's weights $\vec{w}(0)$. In step two, the response $y(n)$ for the first training data is calculated. In step three, this response is compared to the desired response $d(n)$ and used along with the learning rate α to change the weight vector $\vec{w}$.

```
1. Initialize $\vec{w}(0) = \vec{0}$.
2. Calculate the perceptron's actual response $y(n)$
3. Adapt the weights: $\vec{w}(n+1) = \vec{w}(n) + \alpha[(d(n) - y(n)]\vec{x}(n)$
4. Continue with step 2.
```

Figure 2.7 – The perceptron convergence algorithm.

By looking at the difference between the computed result and the expected result, the error $d(n) - y(n)$ can be calculated. Each of the weights is then adjusted by the product of its current input $\vec{x}$, the learning rate α and the error $d(n) - y(n)$. If the result is larger than expected the algorithm decreases the weights, if it is smaller than expected, the algorithm increases the weights [RN03]. The learning rate α determines by how much the weights are adjusted in each iteration. This value has to be selected with care, as too small values might result in a very slow learning rate, whereas too large values lead to divergence, meaning that the algorithm does not work at all.

The functionality is now demonstrated using the `AND` function as an example. Tab. 2.1 shows the results of the `AND` function for all possible input values. A single perceptron with two inputs and a bias, using the threshold activation function, is used to learn the `AND` function. Tab. 2.2 shows the weights, the desired result, the actual result and the computed error of the perceptron for each learning step. The bias is represented as $b = w_0$. The training data is fed to the algorithm in order, meaning

$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{x_1 \wedge x_2}$
0	0	0
0	1	0
1	0	0
1	1	1

Table 2.1 – Tabular representation of the AND function.

step	w_0	w_1	w_2	d(n)	y(n)	d(n) − y(n)
0	0.00	0.00	0.00	0.00	1.00	-1.00
1	-0.10	0.00	0.00	0.00	0.00	0.00
2	-0.10	0.00	0.00	0.00	0.00	0.00
3	-0.10	0.00	0.00	1.00	0.00	1.00
4	0.00	0.10	0.10	0.00	1.00	-1.00
5	-0.10	0.10	0.10	0.00	1.00	-1.00
6	-0.20	0.10	0.00	0.00	0.00	0.00
7	-0.20	0.10	0.00	1.00	0.00	1.00
8	-0.10	0.20	0.10	0.00	0.00	0.00
9	-0.10	0.20	0.10	0.00	1.00	-1.00
10	-0.20	0.20	0.00	0.00	1.00	-1.00
11	-0.30	0.10	0.00	1.00	0.00	1.00
12	-0.20	0.20	0.10	0.00	0.00	0.00
13	-0.20	0.20	0.10	0.00	0.00	0.00
14	-0.20	0.20	0.10	0.00	0.00	0.00

Table 2.2 – The progress of the perceptron learning algorithm for the AND example.

that step 0 uses $(0, 0)$, step 1 uses $(0, 1)$ and so on. The table was produced by a short Ruby script, the source of which can be found in App. C.

Multilayer Neural Networks

After explaining the capabilities of a single neuron, or single layer neural networks in general, the question remains, how additional layers change the computational capabilities.

Before explaining multi-layer networks, a quick example of a function, that cannot be learned by a single neuron is appropriate. Tab. 2.3 shows the so called *exclusive or* (XOR) function, which is usually denoted as $x_1 \oplus x_2$ [BSMM01]. The function has two parameters, which can either be 0 or 1 and one output, which also can be either 0 or 1. As is clearly visible in the table, the output is 1, if exactly one of the inputs is 1. In all other cases the output is 0.

x_1	x_2	$x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

Table 2.3 – Tabular representation of the XOR function.

As the perceptron is capable of learning every linearly separable classification, it now has to be determined, whether the XOR function fulfills this criterion. The two possible outputs can be interpreted as the two possible classes. Fig. 2.8 shows the four sample values along with their class in an x_1, x_2 chart. It is apparent that it is impossible to draw a line, which separates the two classes. This means that the XOR function is not linearly separable and is therefore a very simplistic example for a function, which cannot be learned by the perceptron.

To solve the XOR problem using a neural network, at least three neurons are re-

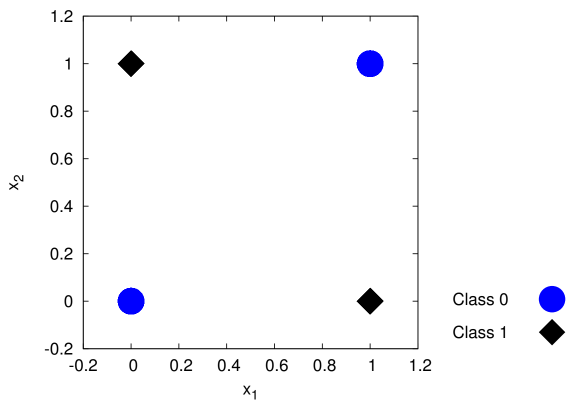


Figure 2.8 – Graphical representation of the XOR function.

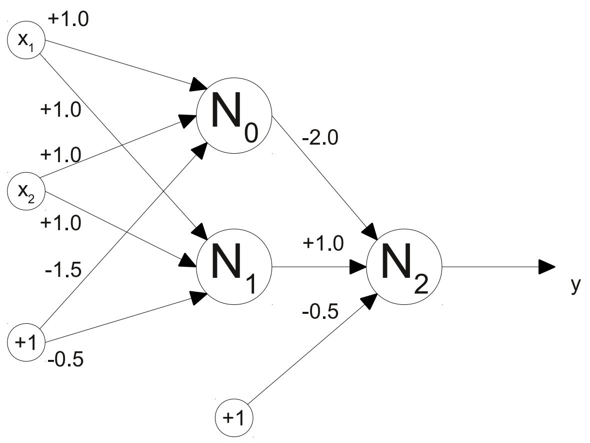


Figure 2.9 – Neural network emulating the XOR function (adopted from [Hay08], p. 173)

quired, leading to a feed forward network with one hidden layer. Figure 2.9 shows a possible solution. Neurons are represented as large circles named N_0, N_1 and N_2. Input nodes are represented by smaller circles. Note that bias values are also supplied by input nodes, which are marked by +1 labels. The threshold function is used in this example.

The ANN output is now calculated manually for the example values $x_1 = 0, x_2 = 1$, to show that this network successfully emulates the XOR function. Calculating the results for the remaining three possible inputs is accomplished in a similar way. The first thing to calculate are the results of neurons N_0 and N_1:

$$N_0 = sgn(-1.5 + 1 \cdot x_1 + 1 \cdot x_2) = sgn(-1.5 + 1 \cdot 0 + 1 \cdot 1) = sgn(-0.5) = -1 \quad (2.14)$$

$$N_1 = sgn(-0.5 + 1 \cdot x_1 + 1 \cdot x_2) = sgn(-0.5 + 1 \cdot 0 + 1 \cdot 1) = sgn(+0.5) = +1 \quad (2.15)$$

Now the result of neuron N_2, which is the overall result, can be calculated. According to Tab. 2.3 the calculation produced the desired result.

$$N_2 = sgn(-0.5 - 2 \cdot N_0 + 1 \cdot N_1) = sgn(-0.5 - 2 \cdot (-1) + 1 \cdot 1) = sgn(+2.5) = +1 \quad (2.16)$$

As this example shows, multilayer neural networks are more powerful than their single layer pendants. The hidden neurons actually function as feature detectors by

$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{N_0}$	$\mathbf{N_1}$
0	0	1	-1
0	1	1	1
1	0	1	1
1	1	-1	1

Table 2.4 – The results of the XOR ANN's hidden layer.

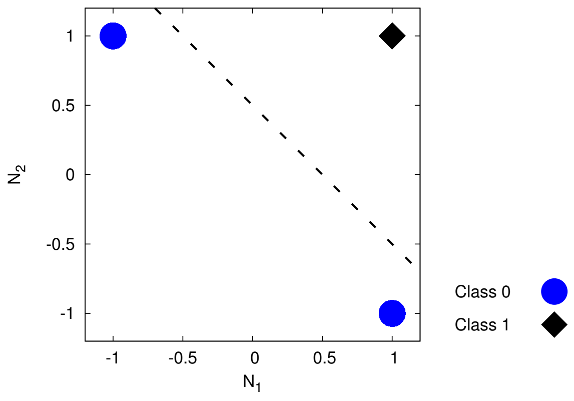


Figure 2.10 – The results of the XOR ANN's hidden layer.

performing a non-linear transformation on the input data into the so called feature space. By looking at this feature space instead of at the raw input data, the classification job of the output neurons is simplified greatly.

It turns out that feedforward networks with one hidden layer are capable of representing any possible continuous function with arbitrary accuracy given that the hidden layer is sufficiently large [RN03]. However, the number of required hidden units grows exponentially with the number of inputs. The number of hidden neurons h required to represent all possible Boolean functions with i inputs can be calculated using the following equation:

$$h = \frac{2^i}{i} \tag{2.17}$$

The outputs of the hidden layer are now analyzed to give a better understanding of the capabilities of feedforward networks. Tab. 2.4 shows the results of neurons N_0 and N_1 for all possible inputs. The results are visualized in Fig. 2.10. Apparently the hidden layer transforms the input in a way that it is now linearly separable.

A neural network is mostly worthless without an adequate learning algorithm. Training the perceptron is simple, because there is only one parameter that needs to be adjusted for each input value. In multilayer networks, however, this task becomes a lot more complicated, due to an issue called the *credit assignment problem.* Learning is based on the idea of evaluating the result and assigning positive or negative credit to the individual units of the network. Assigning credit to hidden units, however, is a non trivial task. Looking back at the XOR example, what weight would have to be changed if the result turned out to be wrong for a certain input?

The answer to this question lies in the so called *back propagation algorithm*, which is presented now. The main problem of having a hidden layer is that it is not clear, which values are to be expected [RN03]. This is in contrast to the output layer, where training data clearly shows the expected results, which can be compared to the computed results, thus allowing to calculate the error. It turns out that this error can be *back-propagated* from the output to the hidden layer. The following equations specify the back propagation algorithm. Eq. 2.18 defines the current error of the network, which is the difference of the desired output $\vec{d}$ and the computed output $\vec{y}$. By multiplying the j-th component of the error with the partial derivative of the activation function φ, one gets the modified error as defined in Eq. 2.19. Each weight vector component w_{kj} is now simply incremented by the modified error M_j, multiplied with the learning rate α and the computed output y_k.

$$\vec{E} = \vec{d} - \vec{y} \tag{2.18}$$

$$M_j = E_j \cdot \frac{\partial \varphi\left(x_j\right)}{\partial x_j} \tag{2.19}$$

$$w_{kj} \leftarrow w_{kj} + \alpha \cdot y_k \cdot M_j \tag{2.20}$$

Search Space

In the beginning of this section it was mentioned, that the free parameters of an ANN with fixed structure are its connection weights. If the number of connections is denoted by N, the network's parameters can be formulated as an N dimensional vector $\vec{w}$. Assuming that an optimal solution for a problem to which an ANN is applied exists, it would therefore be a point in an N dimensional space. If no further constraints exist, the connection weights all lie within $\mathbb{R}$, which leads to an overall search space of $\mathbb{R}^N$.

As fully connected feedforward networks with one hidden layer are the most common type of networks, this case will now be discussed thoroughly. Assuming that an ANN has i input, h hidden and o output neurons. To be able to represent any Boolean function with i inputs, the number of required hidden neurons has to be as specified in Eq. 2.17. Assuming further that the simplification according to Eq. 2.5 is applied, the total number of connections N in the network can be calculated with Eq. 2.21.

$$N = (i+1) \cdot h + (h+1) \cdot o = (i+1) \cdot \frac{2^i}{i} + (\frac{2^i}{i} + 1) \cdot o \tag{2.21}$$

By applying some approximations, this equation can be simplified and the appropriate O-notation can be deduced as indicated in Eq. 2.22.

$$N \approx 2^i + \frac{2^i}{i} = O(e^i) \tag{2.22}$$

This implies, that if the number of output neurons o is fixed, the number of connections N in the network only depends on the number of inputs i. According to Eq. 2.22, the correlation between i and N is exponential. This means, that the search space grows exponentially with the number of inputs and therefore the amount of time required to learn a specific behavior grows exponentially with the number of input neurons. Thus, it is desirable to minimize the amount of required inputs for proper functionality of the network. However, this task proves to be tedious and often the only way to determine a good set of input variables is trial and error. In most applications not the full amount of hidden neurons is required but a substantially smaller amount might suffice. In a recently published paper about neuroevolution, for example, the number of input neurons was selected to be 6 [WFK11]. According to Eq. 2.17 the amount of hidden neurons required for full Boolean computational functionality would be $\frac{32}{3} \approx 10.7$. However, in this paper a neural network with only 3 hidden neurons proved to be sufficient for the task.

2.3.5 Real World Applications: What are ANNs typically used for?

The purpose of this section is to give an overview about what artificial neural networks are used for in real life. The examples presented in this section also show that ANNs are usually not the sole technology used to solve a problem, but are integrated with many other algorithms, which pre- or post-process data. Unfortunately it is hard to find information about the usage of ANNs in actual products. However, the research papers presented in the following paragraphs should give a good overview about what is possible.

Medical Applications

To prevent birth asphyxia it is required to monitor the heart rate of the fetus during labour [GPMR11]. Usually this data is collected and analyzed by hand, which is tedious and error prone. Recently a team from Oxford University successfully trained a committee of six neural networks to analyze the heart rate time series data. Despite being in a very early stage, the system might one day be able to assist doctors interpreting the data more rapidly and to circumvent false diagnoses.

Over ten percent of all women in developed countries might be affected by breast cancer during their life, which makes it the most common cause of death for women in these parts of the world [can, JMW$^+$05]. To fight cancer, early diagnosis is vital. A team from Orissa, India successfully utilized linear wavelet neural networks to improve breast cancer detection and classification success rates [SMDD11].

Vision

Back in 2008 BMW and Opel introduced cars with integrated traffic sign recognition [Gru09]. The purpose of the system is to prevent the driver from missing an important sign. This is achieved by permanently displaying the most important categories, like the last speed limit sign, in the vehicle's head up display. The technology can also be connected to warning systems, that notify the driver when significantly exceeding the current speed limit. Recognizing traffic signs is a complicated problem, which is solved using multistage algorithms, often containing neural networks [PdY].

Financial

Ever since the establishment of stock exchanges people tried to forecast stock prices to maximize profits. Neural networks are often combined with various other technologies, like feature selection algorithms and genetic programming, to achieve a high prediction accuracy [Hsu11]. Researchers from Taiwan applied an even more complex chain of algorithms to this problem, using genetic programming, the artificial fish swarm algorithm and gray model neural networks, to predict Taiwan stocks [HCP11]. Another research group also focuses on the Taiwan market, trying to predict the Taiwan Stock Index, using a probabilistic neural network, trained with historic data, showing that it significantly outperforms competing approaches [CLD03].

Another possible use of ANNs in the financial sector is the prediction of possible acquisition targets, using a broad variety of well known factors [CWY99]. A portfolio, containing stock selected by this algorithm, significantly outperformed the market.

Control

Standard direct current (DC) motors use brushes and a mechanical commutator to supply an alternating current to the moving rotor. Brushless DC motors, on the other hand, have a permanent magnet rotor and stationary electrical magnets. This, however, leads to the need for a control circuit, which emulates the mechanical commutator. Chinese researchers successfully applied a single neuron, along with other components, to control such a motor and achieved high performance [XWX11].

Researchers from the University of Chicago successfully used a neural network to control energy usage in a modern home, using grid, as well as battery power, while optimizing electricity cost [HL11].

In database systems, a huge number of transactions has to be executed in parallel [KE06]. When using optimistic concurrency control, transactions might get canceled if a conflict occurs. As canceling transactions means losing a lot of already executed work, one of the goals is to minimize conflicts and therefore maximize performance. Researchers from Tehran, Iran, managed to use a neural network, based on adaptive resonance theory, to improve performance at various transaction rates [SRA11].

Analysis

As gear faults are one of the main causes of machine unavailability, it is important to detect these defects as early as possible. Scientists developed a way of using, so called, fuzzy lattice neuro-computing, to analyze frequencies along with other components to successfully detect the current state of a gear [LlZsM+11].

Other mechanical engineering applications include the identification of cracks in curvilinear beams, which can be useful to detect otherwise hard to observe failures in complex machines [SGP11].

Optimization

When operating an online shop, one of the most important aspects is usability. Neural networks, in combination with genetic algorithms and other techniques, can be used to automatically optimize an e-commerce site [SMR11].

In material engineering, it was discovered that ANNs provide good performance when analyzing and optimizing the compressive strength of concrete [YKID11].

One of the greatest challenges in civil engineering is the analysis of the behavior of structures during an earth quake [AJA+11]. Using artificial neural networks, genetic programming and simulated annealing, a model was developed to estimate the base shear of steel structures. Base shear is the maximum lateral force, appearing at the base of a building during an earthquake and depends on various factors.

2.4 Genetic Algorithms

This section establishes a basic understanding, of both biological and artificial evolution. The first part explains the concepts of biological evolution, natural selection and survival of the fittest. After that, artificial implementations of these ideas are introduced and last but not least the application of these algorithms to artificial neural networks is presented.

2.4.1 Biological Evolution

The theory of biological evolution dates back to Charles Darwin, was first published in his groundbreaking book *On the Origin of Species* and is the well accepted theory about the past and future development of life on earth [Gre09]. According to Jerry A. Coyne "[evolution] shows how everything from frogs to fleas got here via a few easily grasped biological processes" [Coy06]. The following paragraphs explain the concepts behind these processes.

Basic Concepts

Evolution can only function properly, if *all* of the underlying concepts are in place. Leaving only one of them out might lead to completely different results.

Overproduction and the Struggle for Existence The first concept is the power of biological reproduction [Gre09]. Assume one individual is capable of producing 10 offspring. This means, that the second generation, consisting of 10 individuals, is capable of producing 100 offspring. The third generation, which now already consists of 100 individuals, is capable of producing 1000 offspring, and so on. Therefore, the growth is exponential. This means, the total number of produced individuals is $N = r^n$ where n is the number of reproduction cycles or generations and r is the number of individuals, a single individual can produce.

The potential of this simple rule is enormous. Assuming, one has a single Escherichia coli bacterium and assuming that cell division happens twice per hour, the overall bacteria population would exceed the earth's mass in less than a week [Gre09]. This concept also applies for other species and is called overproduction. But, as populations on our planet appear to be rather stable, there has to be some other concept which neutralizes this massive reproduction.

Most of the offspring never gets the chance to reproduce, because they simply do not survive long enough [Gre09]. This is mostly caused by the limited amount of available resources, as well as by predators. The massive discrepancy between the number of created offspring and the number of individuals, surviving long enough to produce offspring themselves, creates a so called "struggle for existence".

Variation and Inheritance Darwin discovered, that the individuals of a species are not equal, but have different traits [Dar]. He also realized, that individuals related to each other are more similar to each other than unrelated individuals. Darwin knew, that these concepts are critical for natural selection to happen. However, Darwin was not capable of understanding the background of variation and inheritance.

Nowadays, it is well understood that both observations can be explained through genetics [Gre09]. Inheritance functions by recombining strands of DNA from both parents. This process is not perfect and therefore introduces defects into the offspring from time to time. These defects are known as mutations and happen completely randomly. Odds are, mutations will have negative effect for the offspring. But there is also a small chance, that it might turn out to lead to a competitive advantage. Most of the time, the mutations have no measurable effect at all. Mutations, along with the recombination of both parents' DNAs, are the effects responsible for new variation to appear.

The Natural Selection Process Variation leads to the fact, that individuals of the same species all have different traits, which make some of them slightly better equipped for their environment, which in turn might give them a slightly better chance of survival [Gre09]. Individuals, capable of surviving longer, might in turn on average be capable of producing more offspring. Variations might occur at random, however, the selection process that determines, which of these variations is passed on to the next generation, is not random at all.

Darwinian Fitness The economist Herbert Spencer summarized the natural selection process in the well known term "survival of the fittest" [Gre09]. This term was later adopted by Darwin in one of his articles. However, the word "fitness" in this context does not refer to physical condition or strength, as is assumed by many people. What Darwin actually meant is the reproductive success compared to alternatives.

Another key aspect, which is often ignored, is that fitness is mostly referring to an individual's reproduction capabilities [Gre09]. While it is obvious, that individuals have to survive for a certain amount of time to produce offspring, it might happen that evolution actually leads to shorter lives of individuals, while improving reproduction e.g. by increasing fecundity.

Combined Concepts Lead to Evolution

None of the concepts introduced so far is capable of leading to evolution on its own, but all of them combined do [Gre09]. Producing massive amounts of offspring alone has no effect. Neither have differences between the individuals of a species as long as they cannot be passed on to future generations. Variation alone also does not work, if it does not lead to different reproduction rates. Only if all of Darwin's concepts hold, natural selection can happen.

The selection process alone is incapable of producing new properties [Gre09]. It turns out that a two step process is required to achieve this goal. First new variations are generated by recombining existing genomes and by erroneous manipulations of those, which are known as mutations. The second step is to determine, which of those randomly created traits are passed on to future generations.

2.4.2 Simulating Evolution through Genetic Algorithms

Many artificial intelligence researchers share the opinion, that it is hard to encode all the rules, which make up intelligent behavior by hand [Mit95]. Therefore, they are looking for ways to provide simple rules, from which complex and intelligent behavior emerges on its own. One promising attempt to realize this are genetic algorithms.

The concept of Genetic Algorithms (GAs) was first introduced in the 1960s by John Holland as an attempt to understand the ability of adaptation as it appears in nature [Mit95]. Holland's goal was to build computer software, that could adapt to new requirements and environments. The GA introduced by Holland was an abstraction from natural evolution, featuring multiple generations of a population and a method to create a generation from its predecessor. This method uses all the concepts known from natural evolution, like selection, recombination and mutation.

	Binary Genome	**Decimal Genome**
Chromosome 1	10100100 00010011	164, 019
Chromosome 2	00110000 11101010	048, 234
Result for Integer Symbols	10100100 11101010	164, 234
Result for Binary Symbols	10110000 11101010	176, 234

Table 2.5 – Possible crossover results for integer and binary symbols.

A Basic Genetic Algorithm

Many different genetic algorithms were developed in the past decades, each of them optimized for a specific purpose, but all of them share some common basics, which are introduced now.

A genetic algorithm simulates multiple generations of populations of individuals, the first of which is usually generated in a random fashion [Mit95]. The algorithm determines the fitness of each individual in each generation and then decides, which of the individuals is allowed to reproduce. Usually pairs of individuals are chosen and their genomes are combined. Combining the genome from the parents can happen in different ways. One of the most widely used algorithms is called crossover. This algorithm randomly selects a position within the genome. Everything up to this position is copied from the first parent and everything after this position is copied from the second parent. Other algorithms randomly decide for each position within the genome, whether it is copied from the first or the second parent. The result of this recombination phase is then usually randomly mutated to create offspring, which forms the next generation.

The genome of an individual is usually represented as a string or an array of symbols, which are called alleles [RN03, Mit95]. The alphabet these symbols are chosen from influences how the algorithm behaves. For example if one wants to encode an 8-bit integer, one could either interpret the entire integer as a symbol with 256 different states, or one could interpret the individual bits as symbols with only two different states (0 and 1). The information contained in the genome is the same for both cases, but when it comes to crossover the solutions provide different results. As the crossover operator works on symbol basis, it will always divide the genome of the first solution at integer boundaries whereas the genome in the second solution might be divided in the middle of an integer, leading to more or less random results.

Both algorithms are illustrated in Tab. 2.5 for a 16-bit genome, representing two 8-bit integers. Using 8-bit integer symbols, the only possible crossover point is between the first and second symbol. Using bit symbols, the crossover can happen at any bit position. Note how the resulting integers are chosen from existing integers for case one, but a new integer is generated for case two. It is important to choose the symbol alphabet depending on whether or not this behavior is acceptable or even desirable.

The following algorithm summarizes the concepts introduced so far. It is based on the algorithm introduced in [RN03].

1. Create the first generation of N individuals with random genomes.
2. Evaluate each individual and calculate its fitness.
3. Repeat the following steps N times to create the next generation.
   (a) Mating: Randomly pick two individuals. The probability for being chosen should be directly proportional to the individual's fitness. It is usually possible that an individual gets chosen multiple times or not at all.

(b) Crossover: The genomes of the selected individuals are combined through crossover by using a randomly selected crossover point.

(c) Mutation: Randomly select some of the symbols of the produced genome and set them to random values.

4. Replace the current population with the newly created individuals and start again with step (2).

Many variations of this algorithm exist. It is, for example, very common to discard a certain amount of individuals at the lower end of the fitness scale before starting the reproduction [RN03]. This procedure, which is called *culling*, is inspired by the fact that, in nature, many individuals die before they get the chance to reproduce. It can be shown that GAs converge faster when using culling [BBG95].

Neuroevolution - ANNs and GAs Combined

It turns out that artificial neural networks and genetic algorithms are a perfect match. The combination of both worlds is called neuroevolution and was established around 1988 [MD89]. GAs are an interesting alternative (as well as an addition) to the more commonly used learning algorithms introduced in Sec. 2.2 and can be applied to neural networks in many ways. Typical properties of neural networks, which can be evolved, include the connection weights, the network structure or the parameters of a separate learning algorithm.

Evolving Connection Weights The idea of training neural networks using genetic algorithms was first introduced by Montana and Davis in 1989 [MD89]. They encoded the weights of a fixed topology neural network as a list of real numbers, which formed the genome. Fig. 2.11 shows an example for such a mapping. By applying a genetic algorithm to a population of individuals (i.e. neural networks), the weights can be progressively adjusted until reaching a satisfying solution for the specified problem. Montana and Davis also showed that neuroevolution can easily outperform back propagation algorithms.

Evolving Network Topology Designing a neural network architecture for a given problem can be a complex task, which often involves a lot of guessing, experience and experimentation [Mit96, RN03]. Often, the network topology decides, whether a

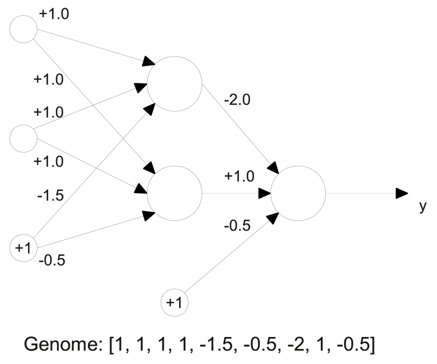
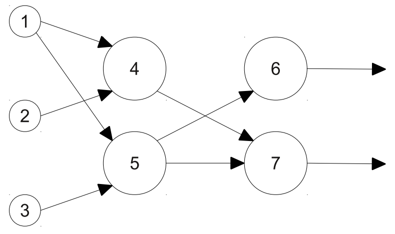


Figure 2.11 – Mapping the connection weights of a neural network to the genome.



(a) Neural Network

to unit:	1	2	3	4	5	6	7
from unit: 1	0	0	0	1	1	0	0
2	0	0	0	1	0	0	0
3	0	0	0	0	1	0	0
4	0	0	0	0	0	0	1
5	0	0	0	0	0	1	1
6	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0

(b) Connection Matrix

Figure 2.12 – A binary matrix is used to encode the connections of a neural network to be used in genetic algorithms. The genome is obtained by serializing the matrix: [0001100||0001000||0000100||0000001||0000011||0000000||0000000].

particular ANN application succeeds or fails. It would reduce the developer's workload if there was an algorithm, which could automate this tedious job. As it turns out genetic algorithms can also be applied for this task.

There are two possible ways to encode the network structure in the genomes [Mit96]. The first one is *direct encoding*, which uses a fixed number of N neurons. All possible connections are represented as a binary N by N matrix, each element of which represents a possible connection. If an element is 1, the corresponding connection is present. If it is 0, the corresponding connection is left out. This matrix can then be serialized to form a binary string genome, which is used in a genetic algorithm as usual (see Fig. 2.12 for an example). Note that the connection matrix is capable of representing recurrent neural networks. If a pure feedforward network is preferred, the bottom left half of the matrix can simply be ignored. The connection weights of the resulting structures can be trained by a separate learning algorithm, which might again be a GA or a conventional algorithm like back propagation.

Direct encoding is simple, but fails to encode complex structures as the length of the genome increases quadratically with the number of neurons in a network [Mit96]. Additionally, the method is unable to encode repeated or nested structures in a network. Due to these shortcomings Kitano invented the *grammatic encoding*, which uses a graph generation grammar to define network architectures. For more information about this type of encoding consider reading [Kit90].

2.5 Controlling Robots with Neuroevolution

Neuroevolution can be applied in a wide variety of fields. For example, Faustino John Gomez successfully used neuroevolution to control a finless rocket, which is a particularly hard job, as finless rockets are unstable and would tumble without active control [Gom03]. Another popular usage of neuroevolution is the control of robots. Stanley et al. developed a real time neuroevolution algorithm, which can be used to control agents in the NERO video game [SBM05]. The agents in this game constantly improve their behavior and are able to adapt to the strategies of other players.

To control a robot using neuroevolution, it is required to connect the sensors and actuators of the robot to the neural network. This can be accomplished in various different ways. One of the simplest possibilities is to connect each of the sensors, to one or more input neurons, and each of the actuators to one or more output neu-

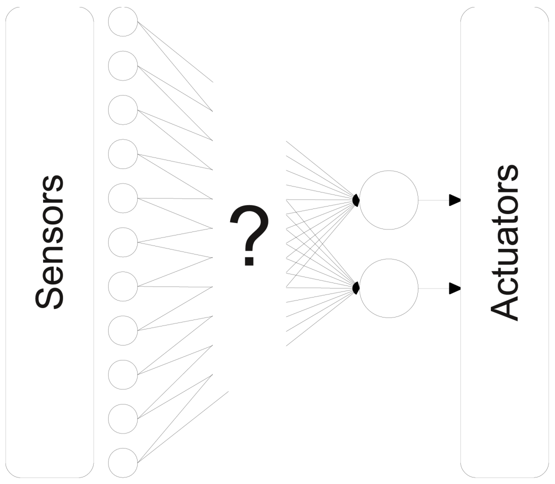


Figure 2.13 – Controlling robots with neuroevolution.

rons, as illustrated in Fig. 2.13. This approach has been widely tested. For example, Waibel et al. used this approach to control robots, which are supposed to collect food items [WFK11].

Many problems would require ANNs of massive proportions or are simply easier to implement using other types of algorithms. Therefore, it often makes sense to add some data pre- and/or post-processing to the neural network. ANN-based face-detection, for instance, requires several pre- and post-processing steps [Row99]. Another example is the above mentioned NERO video game. The neural network of an agent outputs abstract commands, like *walk forward/back*, *turn left/right*, *fire weapon*, using three output neurons [SBM05]. These commands are then executed accordingly by additional logic. The input of this neural network is also pre-processed, using complex algorithms. Each robot has a so called enemy radar, which determines where enemies are located around the robot. This information is aggregated and then fed into five input neurons. Five additional input neurons are fed with information about the distance of objects around the robot. Two inputs are fed with information about whether an enemy is in the line of fire and one input is active, only if an enemy is directly in front of the robot. This configuration, which is very successful within the NERO video game suggests how important it is, to carefully select and pre-process input data.

This chapter introduced all the conceptual foundations, especially neural networks and genetic algorithms, required to understand the next chapters. The following chapter presents the marXbot robotic platform and the ARGoS simulator, both fundamental tools for the experiments of this thesis.

Chapter 3

Robotic Platform and Simulator

This chapter introduces the two most essential tools used for the experiments in Chap. 5: The *marXbot*, which is a multi-purpose robot, and *ARGoS*, which is used to simulate robot instances and their environment.

3.1 The marXbot Robotic Platform

Some of the tools used for this thesis have been developed in the *Swarmanoid* project[1]. The researchers of this project employ three different types of robots with various different capabilities.

The *hand-bot* is capable of climbing vertical surfaces as well as manipulating objects. The *eye-bot* has eight rotors, which allow it to fly like a helicopter. It is also capable of attaching itself to the ceiling, thus providing a bird's eye view. Last but not least the *foot-bot*, which is based on the marXbot robotic platform, is capable of driving and docking with other robots.

The marXbot platform was chosen as the robotic platform for all experiments in this thesis and is therefore explained thoroughly. Fig. 3.1 shows a rendering of the robot, which was generated using the ARGoS simulator (see Sec. 3.2) and POV-Ray[2]. The robot has a diameter of 17cm, a height of 29cm and a mass of 1.8kg [RMV11].

3.1.1 Sensors

The marXbot has a great amount of sensors that make it very versatile. First of all the robot has two cameras, both using a three megapixel sensor [RMV11]. One of the cameras is looking straight up towards a hyperbolic mirror, which allows the robot to observe 360° of its environment. The camera and its mirror are clearly visible in the upper half of Fig. 3.1. The second camera is optional and is either looking forward or also up, but without a mirror. A long range rotating infrared distance sensor is capable of detecting objects in a range of 4cm to 150cm. Additionally, 24 static infrared proximity sensors can be used to observe nearby obstacles. A three dimensional accelerometer and a three axis gyroscope allow the robot to determine its current orientation and estimate its current position in space. Eight infrared and four visual ground sensors can be used to determine various properties of the surface, the robot is currently on. Last but not least, an RFID reader can be used to detect and read RFID tags in the environment.

[1]http://www.swarmanoid.org/, accessed 17. Dec. 2011

[2]http://www.povray.org/, accessed 17. Dec. 2011

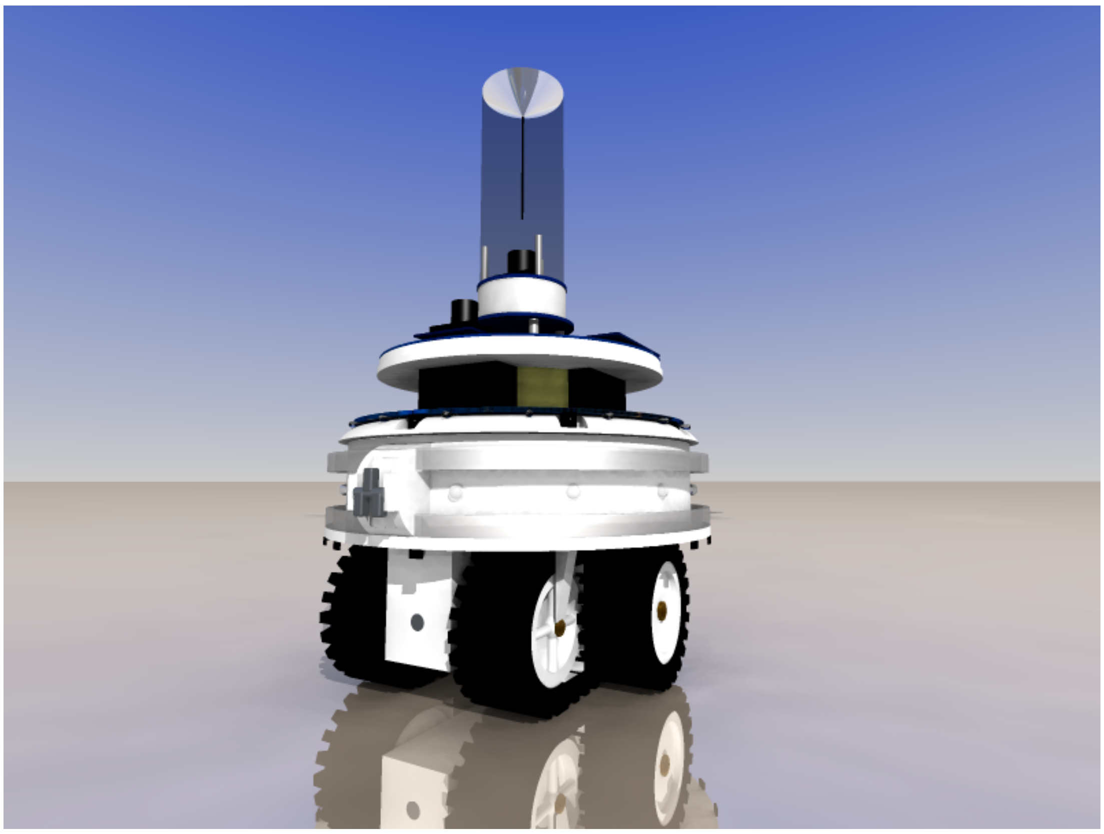


Figure 3.1 – The marXbot rendered using the ARGoS simulator and POV-Ray.

3.1.2 Actuators

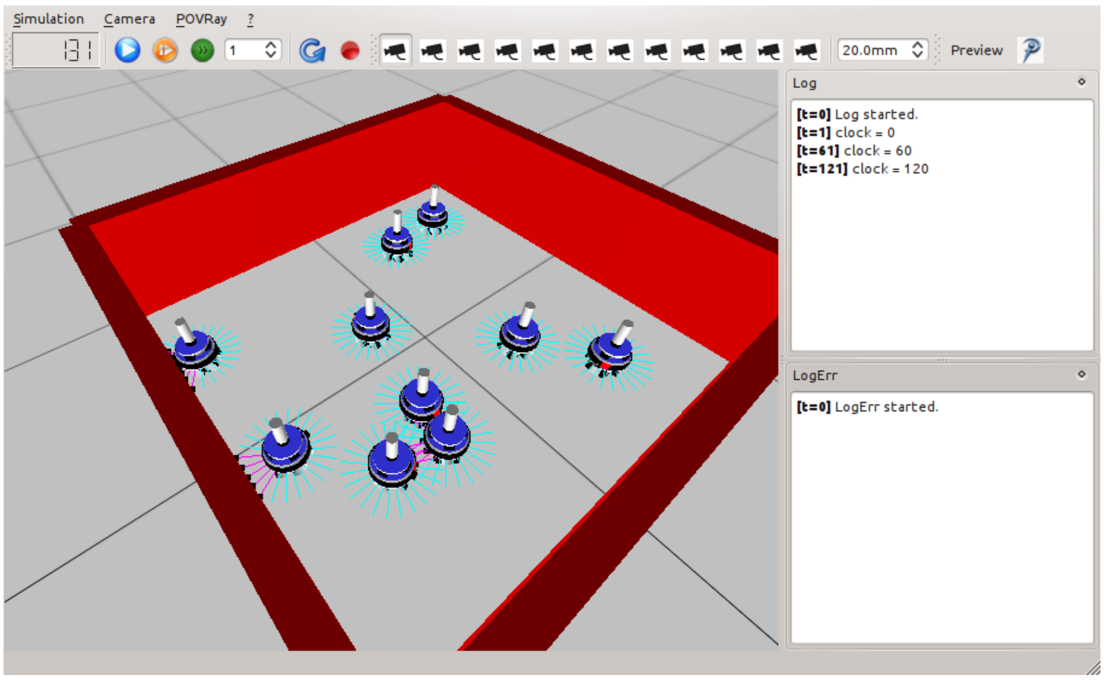
The robot has two so called treels, which are combinations of tracks and wheels [RMV11]. The rendering in Fig. 3.1 shows what this construction looks like. Both treels can be controlled individually which allows the robot to move forward, backward and to turn like a tank. A rotatable attachment device consisting of two small fingers can be used to connect mechanically to other robots or objects with a compatible spline. Lastly, the robot has a powerful 3.5W RGB beacon LED on its top and 12 RGB LEDs arranged around its body. The LEDs can be used to communicate with other robots or to simplify locating robots visually.

3.1.3 Controller

A complete marXbot consists of ten microcontrollers, which are interconnected through a CAN and an I2C bus [RMV11]. Each of the microcontrollers is responsible for controlling one or more of the sensors or actuators. An embedded computer is responsible for coordinating all of the microcontrollers and for executing logic and behavioral algorithms. This computer is based on an ARM 11 CPU with 533MHz and 128MiB of RAM. It can communicate with the outside world over TCP using either WiFi or USB.

3.1.4 Programming Model

The microcontrollers use a high level, event based, scripting language called *ASEBA*, which was developed to simplify marXbot programming [RMV11]. Each of the microcontrollers runs a small virtual machine, which is responsible for executing ASEBA



```
[t=0] Log started.
[t=1] clock = 0
[t=61] clock = 60
[t=121] clock = 120
```

```
[t=0] LogErr started.
```

Figure 3.2 – The ARGoS Qt4/OpenGL user interface.

programs. The embedded computer runs a Linux based operating system, which controls the hardware and simplifies programming by providing a familiar environment without having to deal with low level tasks like memory management or networking.

3.2 The ARGoS Simulator

The ARGoS (Autonomous Robots Go Swarming) simulator was developed within the Swarmanoid project [PTO+11]. It is written in C++ and features a modular architecture with a focus on extensibility and scalability. A diverse set of robots is supported and it scales easily up to several thousand robots within the same simulation. All entities like robots, sensors, actuators and physics engines are realized as separate modules, which can easily be replaced to provide the optimum solution for all types of experiments. By providing different implementations of the same entity type, the user can select the best trade-off between realism and performance. Scalability is obtained by a multi-threaded architecture focusing on high CPU utilization. One of ARGoS' unique features is the ability to use multiple physics engines within the same simulation, assigning each of them a part of the available space and allowing objects to seamlessly transition between the engines. This allows, for instance, to assign flying robots to a 3D and wheel based robots to a 2D physics engine, which improves performance without limiting capabilities. ARGoS supports multiple visualization engines, which, again, concentrate on performance or realism. Fig. 3.2 shows a screenshot of the Qt4 and OpenGL based user interface. It is also possible to turn off visualizations all together for unattended batch runs.

Robots are programmed using a special controller interface, which is the same for simulated and real robots [PTO+11]. Therefore, it is possible to write controllers for the simulator and cross compile them for real robots, once they are mature enough. On real robots, the controllers are executed on the built-in computer. The robot's microcontrollers cannot be programmed using this model, but have to be programmed using the ASEBA scripting language. However, for most usecases, there is no need to change any of the built in ASEBA programs.

3.2.1 Configuration

Experiments can be configured using a flexible XML format, which allows to control global options like multi-threading, physics engines, visualizations, as well as the experiment itself [PTO+11]. The experiment is specified by setting up the space and the robots. The space consists of a rectangle of arbitrary size, on which static and dynamic blocks, as well as robots and other entities can be placed. The configuration file is also the place where custom modules can be specified and implementations can be chosen. A reference manual for the configuration file format can be found in App. A.

3.2.2 Programming Model

The following sections describe the programming model, which is used to extend ARGoS. The architecture distinguishes between robot controllers, which govern a robot's behavior, loop functions, which can be used to execute arbitrary code within the simulation and Qt/OpenGL user function, which can be used to extend the rendered scene.

Robot Controllers

Robots are governed by so called controllers, which are defined by extending the `CCI_Controller` class. This class offers several lifecycle methods, which are called on various occasions. The class also offers methods to access robot sensors and actuators. The following listing shows an overview of the interface. Modifiers like `const`, `public`, `virtual` or `= 0`, constructors, some utility methods and default values have been omitted for the sake of brevity.

```
 1 class CCI_Controller : public CBaseConfigurableResource, public CMemento
     {
 2   void Init(TConfigurationNode& t_node);
 3   void ControlStep();
 4   void Reset() {}
 5   void Destroy();
 6
 7   CCI_Robot& GetRobot() {
 8     return *m_pcRobot;
 9   }
10
11   bool IsControllerFinished() const {
12     return false;
13   }
14 }
```

The lifecycle methods `Init()`, `ControlStep()`, `Reset()` and `Destroy()` of lines 2-5 are used to specify custom controller behavior. The `Init()` method is called exactly once, when the robot is initialized. A reference to the `<parameters>` XML node within the tag, specifying this controller, is passed to the method. This reference can be used to access custom configuration options. The `Init()` method is used to initialize the controller and to request references to sensors and actuators. The `ControlStep()` method is executed for each simulation step. It therefore contains all of the controller's logic. A typical `ControlStep()` method implementation consists of querying sensor data, performing calculations and adjusting actuator settings. The `Reset()` method is called whenever ARGoS decides to reset the simulation. This is usually only the case if the user clicks on the reset button. This method should be used to restore the state, which existed immediately after the `Init()` method has been called. The `Destroy()` method is called when ARGoS terminates. This method should be used to free all of the allocated resources and to persist collected information.

The `GetRobot()` method can be used to get a reference to the robot entity, this controller is assigned to. This reference can be used, for example, to query the current location of the robot. The `IsControllerFinished()` method can be overwritten to inform ARGoS, whether or not the controller is still active. If all the controllers return `false` in this method, ARGoS is able to terminate the simulation prematurely. Otherwise, the simulation would run for the entire predefined duration. Obviously, these methods only make sense within the simulator and cannot be used in real robots.

Controllers are registered using the `REGISTER_CONTROLLER` macro. The following listing shows an example:

```
  REGISTER_CONTROLLER(CSomeController, "some-controller")
```

The controller can be specified within the ARGoS XML configuration using the following using syntax (see App. A for a detailed configuration reference):

```
<controllers>
  <some-controller id="foac" library="libsome_controller.so">
    <actuators>
      <!-- List of actuators -->
    </actuators>
    <sensors>
      <!-- List of sensors -->
    </sensors>
    <parameters param1="value1" param2="value2" ... />
  </some-controller>
</controllers>
```

The tag name (`<some-controller>` in this case) has to match the string specified in the `REGISTER_CONTROLLER` macro. The `library` parameter contains the path to the shared object file containing the controller. The `parameters` tag can be used to pass custom parameters to the controller.

Loop Functions

Loop functions are a concept, which allows the programmer to run arbitrary code during the simulation and to control many aspects of the simulator. Loop functions are implemented by extending the `CLoopFunctions()` class provided by ARGoS. This class contains several virtual methods, allowing to execute code for various occasions during the simulation. The following listing gives an (incomplete) overview over the `CLoopFunctions` interface.

```
class CLoopFunctions : public CBaseConfigurableResource {
  void Init(TConfigurationNode& t_tree) {}
  void Reset() {}
  void Destroy() {}
  void PrePhysicsEngineStep();
  void PostPhysicsEngineStep();
  bool IsExperimentFinished();
  CColor GetFloorColor(CVector2& c_position_on_plane);
};
```

Modifiers like `const`, `public`, `virtual` or `= 0`, constructors, some utility methods and default values have been omitted for the sake of brevity. The `Init()` method is called just once during the initialization phase of ARGoS. It allows the module to perform all the necessary initialization tasks. The parameter passed to this method is a reference to the `<loop_functions>` XML element defining the module within the experiment configuration. The reference can be used to read custom configuration parameters and sub nodes from the configuration file.

`Reset()` is called, when the simulation is reset and `Destroy()` is called immediately before ARGoS terminates. The `IsExperimentFinished()` method allows the loop functions to terminate the simulation by returning `true`.

The methods `PrePhysicsEngineStep()` and `PostPhysicsEngineStep()` can be used to execute custom code in each simulation step (either before or after the physics engine invocation).

Last but not least, `GetFloorColor()` can be used to specify a custom floor pattern. This method is invoked for every pixel on the floor, which is then drawn in the specified color. The floor color is perceivable for the robots using the ground sensors.

Loop functions are specified in the experiment configuration using the following tag:

```
  <loop_functions library="libsomeloopfunctions.so" label="
     some_loop_functions" />
```

The library parameter has to point to the shared object file containing the loop functions and the label has to specify the loop functions within the library. The label is registered within the C++ code by using the `REGISTER_LOOP_FUNCTIONS` macro:

```
  REGISTER_LOOP_FUNCTIONS(CSomeLoopFunctions, "some_loop_functions")
```

Qt/OpenGL User Functions

While loop functions are used to change the behavior of simulations, Qt/OpenGL user functions can be used to customize the rendered scene. These two interfaces are kept separate, because ARGoS is designed to function entirely headless, i.e. without Qt and OpenGL. The user functions interface offers several methods to draw 3D primitives. The following listing gives an overview about the user functions interface. Modifiers like `const`, `public`, `virtual` or `= 0`, constructors, some utility methods and default values have been omitted for the sake of brevity.

```
class CQTOpenGLUserFunctions {
  void Draw(CCylinderEntity& c_entity) {}
  void Draw(CFootBotEntity& c_entity) {}
  void Draw(CEyeBotEntity& c_entity) {}
  void Draw(CEPuckEntity& c_entity) {}
  void DrawOverlay(QPainter& c_painter) {}

  void DrawTriangle(CVector3& c_center_offset, CColor& c_color, bool
      b_fill, CQuaternion& c_orientation, Real f_base, Real f_height);
  void DrawCircle(Real f_radius, CVector3& c_center_offset, CColor&
     c_color, bool b_fill, CQuaternion& c_orientation, GLuint
     un_vertices);
  void DrawCylinder(Real f_radius, Real f_height, CVector3&
     c_center_offset, CColor& c_color, CQuaternion& c_orientation,
     GLuint un_vertices);
  void DrawSegment(CVector3& c_end_point, CVector3& c_start_point,
     CColor& c_segment_color, bool b_draw_end_point, bool
     b_draw_start_point, CColor& c_end_point_color, CColor&
     c_start_point_color);
12   void DrawPoligon(const std::vector<CVector3>& vec_points, CColor&
         c_color);
13   void DrawPoint(const CVector3& c_position, CColor& c_color, Real
         f_point_diameter);
14 }
```

Lines 2-6 contain methods, which are invoked by ARGoS, whenever it wants to draw one of the entities. For example, the method in line 3 is executed when ARGoS is drawing the marXbot model. By overriding these methods it is possible to add 3D elements, like gauges, to the scene. Lines 8-13 contain the actual drawing commands, which are an abstraction of the OpenGL interface. These methods can be used in the overwritten methods to add custom elements, like cylinders, to the scene.

User functions have to be registered using the `REGISTER_QTOPENGL_USER_FUNCTIONS` macro. The following listing shows an example:

```
  REGISTER_QTOPENGL_USER_FUNCTIONS(CSomeUserFunctions, "
      some_user_functions")
```

Qt/OpenGL user functions are specified in the ARGoS configuration using the following tag within the `<qtopengl_render>` tag. The `library` parameter specifies the location of the shared object file containing the user functions. The label has to be identical to the string specified in the `REGISTER_QTOPENGL_USER_FUNCTIONS`.

```
  <user_functions library="libsomeuserfunctions.so" label="
      some_user_functions" />
```

Chapter 4

BRAIn - BRAIn Robot Algorithm Insight

In Chap. 5, several experiments are introduced, where robots are placed in an environment, which contains several food items and a base. The goal for the robots is to collect as much energy from the food items as possible and to drop it off in the base. To be able to execute these experiments in an automated fashion, a framework had to be developed first. This framework is named *BRAIn*, which is a recursive acronym[1] for *BRAIn Robot Algorithm Insight*. This chapter first describes all the requirements for a software to successfully fulfill the needs. After that, an architecture is introduced, which enables the software to conform with all the requirements. One section deals with ARGoS specific features and a separate section deals with implementation details. The following four sections contain an in depth description of how to use and extend BRAIn. The chapter is concluded with a list of features, which have not been realized yet, but would be worthwhile implementing.

4.1 Requirements

This section describes all the requirements, which were regarded during the design of BRAIn. This list contains entries of the initial brainstorming phase as well as requirements added later on. Early versions of BRAIn were already used to conduct experiments. The experiences earned during these early applications were then used to update and extend the list of requirements.

4.1.1 General Software Requirements

The first group of requirements apply to most types of software, but are explicitly mentioned here to emphasize their importance.

1. **Correctness.** The software has to be *reliable* and *correct*, because a software producing wrong results is worthless and might lead to wrong assumptions, if errors are not detected on time.

2. **Reliability.** Reliability is also a key issue, because the software is supposed to run on large super computers or clusters for several days or even weeks. An

[1]http://www.wordspy.com/words/recursiveacronym.asp, accessed 28. Nov. 2011

unreliable software would waste computing time and would require a lot of manual interference, which is not desirable.

3. **Extensibility.** Adding a new type of simulator should be easy and straight forward and should not require any complex changes of existing code. Above all, adding new types of experiments should be as simple as possible, while providing the flexibility to add experiments not conceived during the design of the software.

4. **Scalability.** As BRAIn is supposed to run on various different computer architectures, ranging from one to hundreds of processors, scalability is important. Additionally it is vital that the software is capable of utilizing as many processors as possible to speed up experiment execution.

5. **Portability.** The software needs to be portable, because the computers executing the experiments might be significantly different from the computers on which the software was developed. In the case of this thesis the software was developed on an *Ubuntu 11.04* based dual core machine, but the experiments were executed on the *SuperMUC*[2], which is based on Xeon processors and has 40 cores per partition.

6. **Usability.** The software should be easy to use despite its flexibility to allow as many people as possible to conduct their own experiments.

4.1.2 Domain Specific Requirements

After dealing with the more general requirements for BRAIn, this paragraph deals with the domain specific requirements, mostly extracted from real life experience with development versions of this software.

7. **Running Arbitrary Experiments.** BRAIn has to be capable of running arbitrary experiments. The framework's architecture must not limit the types of experiments that it can support. This is especially important because a wide variety of different experiment types exists, all of which are executed slightly differently.

8. **Traceability.** The software has to be capable of persisting all the information required to re-run any part of an experiment at a later point in time. This implies that BRAIn behaves entirely deterministically.

9. **Support For Other Simulators.** Right now, ARGoS is the only simulator supported by BRAIn, as all the experiments in this thesis have been conducted with it. However, it is important to support any other type of simulator as well, which implies that BRAIn must not make any assumptions considering simulator architecture or interface.

10. **Flexible Experiment Configuration.** To simplify usage as much as possible, BRAIn has to support specifying experiments in configuration files. This concept allows to change parts of an experiment, without recompiling any parts of the framework or of corresponding experiment modules. The configuration file format has to be flexible enough to allow the specification of a great amount of experiments in a systematic manner.

[2]http://www.lrz.de/services/compute/supermuc/systemdescription/, accessed 17. Dec. 2011

11. **Large Scale Experiment Execution.** One of the key aspects about BRAIn is to enable the execution of large amounts of experiments, without manual intervention. This allows users to specify a large amount of slightly different experiment configurations, that can be set off simultaneously and do not require any user interaction until their completion. By doing this it can easily be determined how specific experiment parameters influence the results.

12. **Flexible Simulator Communication.** Experiment execution is based on passing information to the simulator, launching the simulator and collecting the results. Two of these three steps involve transferring data between BRAIn and the simulator. The type of information transferred depends on the type of experiment as well as on the simulator. Some simulators might need configuration files, while others expect their configuration on the command line. What makes this task especially challenging is that many experiments require to pass information directly to the robots, running inside the simulator. Therefore, it is important to provide a flexible way of communication between BRAIn and the selected simulation software.

13. **Parallelization.** Some types of experiments take up very long time. For example one of the experiments presented in Chap. 5 took over two weeks to complete. If these experiments were executed sequentially, one would have to wait months or years for the results. The solution is to execute as much as possible in parallel. BRAIn has to support this paradigm and it is the framework's responsibility to schedule computation tasks as efficiently as possible.

14. **Suspend and Resume.** To ensure that already partially computed experiments are not lost due to system failures, the software has to support suspend and resume. This implies that the execution of an experiment can be resumed even if it is terminated in an uncontrolled way, i.e. due to a power failure or system reboot. The loss of data has to be kept to a minimum under all circumstances. Without this feature it could happen that after days or even weeks of executing an experiment, a simple handling error could force the user to restart the experiment from the beginning.

15. **Reporting.** One of BRAIn's key aspects is to provide insight into algorithms, applied to robots. To accomplish this, the software has to collect and report detailed statistics. As the type and amount of data to collect largely depends on the experiment, these mechanisms have to be easily extensible. To allow the user to monitor the experiment progress and to gain an overview about the development of already executed parts, it is important to display human readable experiment overviews.

16. **Logging.** As experiments and the framework itself can be complex, it is appropriate to offer detailed logging information to the user. Everybody who has to handle logs on a regular basis, knows that these files can get confusing very quickly. Consequently, BRAIn should support fine grained log output filtering.

17. **Epoch Based Simulation.** Many experiments (especially evolution based experiments) are epoch based, which means that they are composed of several simulator runs, which might have slightly different configurations. These types of

experiments commonly contain several hundred epochs, which have to be executed sequentially. BRAIn has to support this type of experiments with all arising consequences, like statistics processing and configuration generation.

18. **Epoch Variations.** To improve the quality of measured results, it is sometimes required to run each experiment epoch several times with slightly different parameters. These runs are called variations and have to be supported by the framework. The statistics information collected during each of these runs has to be aggregated into one consistent epoch data set.

4.2 BRAIn's Architecture

Now that all the requirements are defined, the architecture, which enables the code to fulfill each of them, can be presented. The following sections first describe BRAIn's overall architecture and then present the classes and services, which form the execution model. A special section deals with ARGoS specific architectural details. BRAIn's design follows the state-of-the-art software design methodologies and was essentially influenced by *Effective Java* [Blo08], *Java Concurrency in Practice* [GPB+09] and *Dependency Injection* [Pra09].

4.2.1 Module Overview

Just like any other complex software, BRAIn is divided into several modules, as this serves the *low coupling, high cohesion* principle, besides improving overall readability (requirement 1) and understandability of the code. Fig. 4.1 provides an overview of BRAIn's architecture, including the experiment code. It is important to understand that while the dependencies in this drawing exist during compile time, additional dependencies might appear at runtime. The following paragraphs describe all of the modules and their responsibilities.

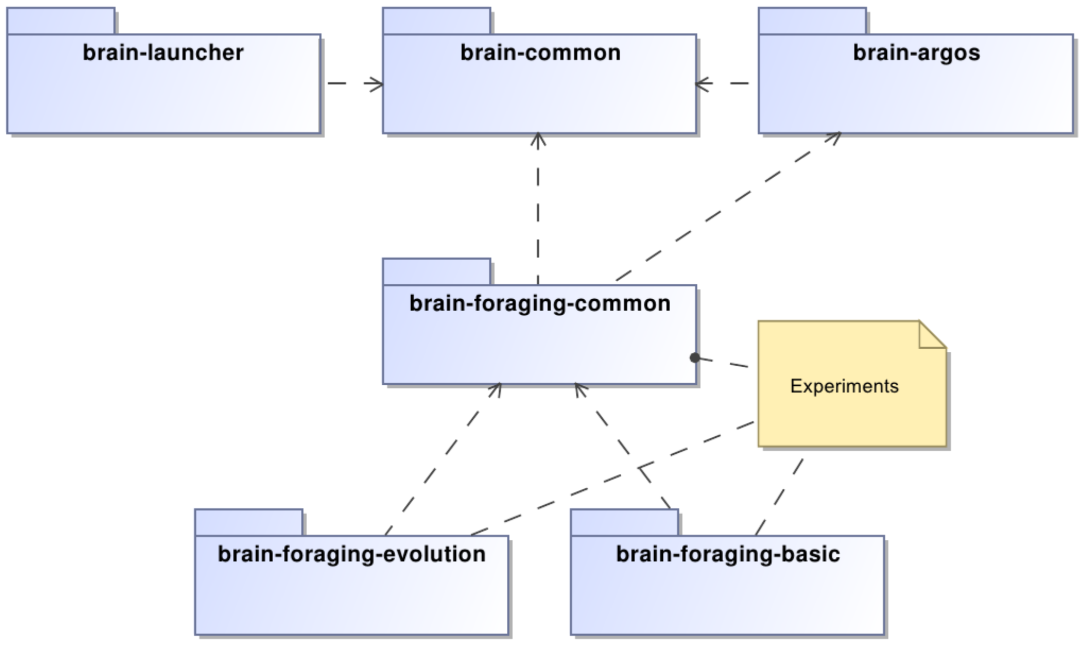


Figure 4.1 – Core and experiment modules of the BRAIn framework.

Only the top row of components (`brain-launcher`, `brain-common` and `brain--argos`) actually belongs to the BRAIn infrastructure. The remaining modules contain code specific for the experiments conducted in Chap. 5, as indicated in the drawing.

It is apparent, that the `brain-common` module is the only module without any dependencies to other components. The reason for this is that `brain-common` is the core module of the framework, containing the data model and all the services required for experiment execution. The code in this module is independent of a specific simulator, as well as independent from any user interface implementation.

The module `brain-launcher` contains all the command line interface specific code. The responsibility of these classes is to parse the command line as well as the configuration files, to wire everything together, to load all the required modules, if any, and to kick off the execution.

The last remaining core module is `brain-argos`. As its name suggests, it contains all the ARGoS specific code, which includes a service to launch ARGoS with specified parameters, as well as utility classes, required for passing information to ARGoS. It also contains the base classes for all experiments, utilizing the ARGoS simulator. It is worth pointing out, that ARGoS utilizes a pseudo random number generator, the seed of which can explicitly be configured in BRAIn based experiments. Bundling all of the ARGoS specific code into a separate module keeps dependencies to this code to a minimum and therefore reduces the barrier for adding other simulators in the future (requirement 9).

As mentioned earlier, all the remaining modules contain experiment specific code. The `brain-foraging-common` module provides functionality similar among all the experiments. The focus lies on experiment set up and the calculation of statistics.

The module `brain-foraging-evolution` is responsible for executing evolution based foraging experiments, while `brain-foraging-basic` is capable of executing standard experiments, consisting of just one epoch, without much logic.

4.2.2 Execution Model

Before being able to understand BRAIn's core classes, one must first understand its experiment execution model, which is illustrated in Fig. 4.2 and which is now explained, starting from the outside.

The first thing BRAIn does, when starting execution is to create a so called `Batch`, which is basically a group of experiments. The class `DefaultBatch` is the standard and, at the moment, the only implementation of `Batch`. It simply contains a list of experiments, that are executed in sequence or in parallel, according to the selected level of parallelism. It would, however, make perfect sense to write special `Batch` implementations, which execute experiments according to certain specifications. Although this case is not supported right now, it could be easily implemented.

No matter what `Batch` implementation one chooses, in the end, each instance of them consists of a list of `Experiments`, which have to be executed. As one of BRAIn's key requirements is scalability (requirement 4), these experiments are usually executed in parallel by a special `Executor` service, which is explained in more detail in the following section. By default, the framework starts running all of the experiments in parallel, but one can limit the number of concurrent experiments with a configuration parameter, which is explained later. The reason why executing all experiments simultaneously usually does not pose any problems, is that experiments are not doing much work, except some administrative tasks. The actual work is done two levels down by so called `Variations`, which are explained later.

As mentioned earlier, each experiment consists of so called `Epochs` (requirement 17). Unlike other parts of the execution model, epochs of the same experiment have to be executed strictly in order, because each of them depends on the results of its predecessor. As the first Epoch obviously does not have any predecessor it has to be dealt with separately in code. As epochs of the same experiment cannot be run in parallel, they are the constraining factor for overall execution time.

The deepest hierarchy level is occupied by so called `Variations`, which represent minor variations of the `Epoch` parameters (requirement 18). As briefly mentioned earlier, these are the instances, which do the actual work. This means that this is the point where ARGoS, or any other simulator, is invoked. To avoid system overload, the number of variations, that are allowed to run in parallel, is limited by a special configuration parameter, which is discussed later. As variations are the only part of the system doing actual work, it is also the only part of the system, which must not run with an arbitrary amount of threads. To ensure that, again, a special `Executor` service is used. All of the epochs running in parallel are submitting their variations to the same `Executor` instance, which limits concurrency. This is also the reason why parallelism is not a problem for experiments as mentioned earlier.

BRAIn's execution model is especially designed for large scale experiment execution (requirement 11) and therefore allows to run great amounts of experiments in parallel without manual intervention.

4.2.3 Core Classes

Now that the execution model of BRAIn has been introduced, the more elementary building blocks can be explained. Fig. 4.3 provides an overview of the classes, reflecting

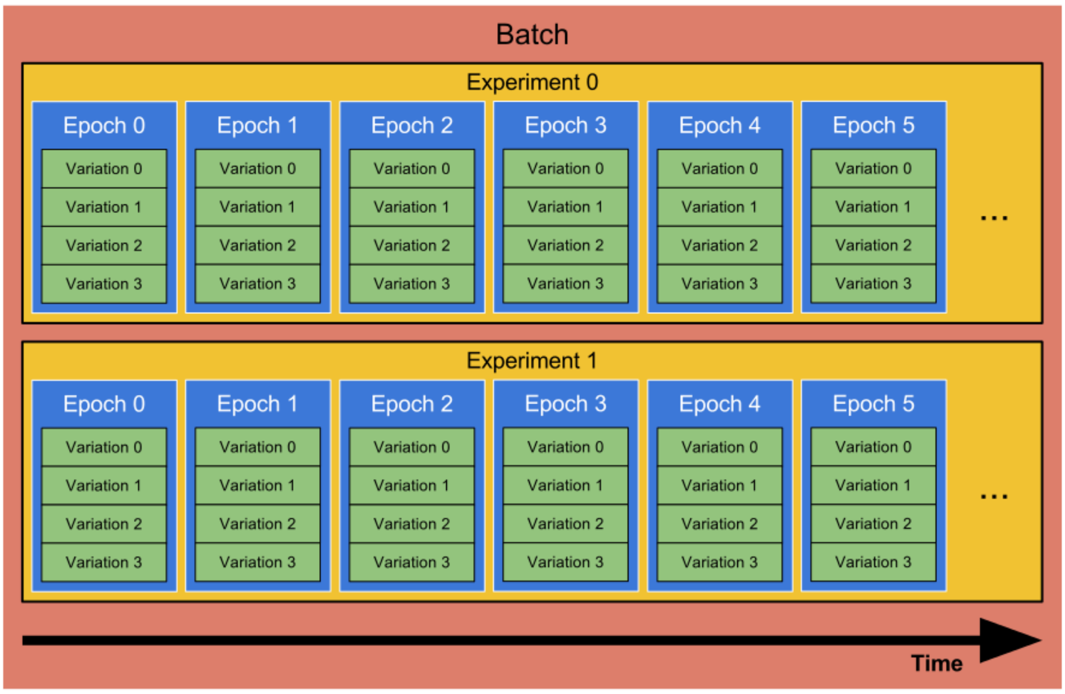


Figure 4.2 – BRAIn's execution model.

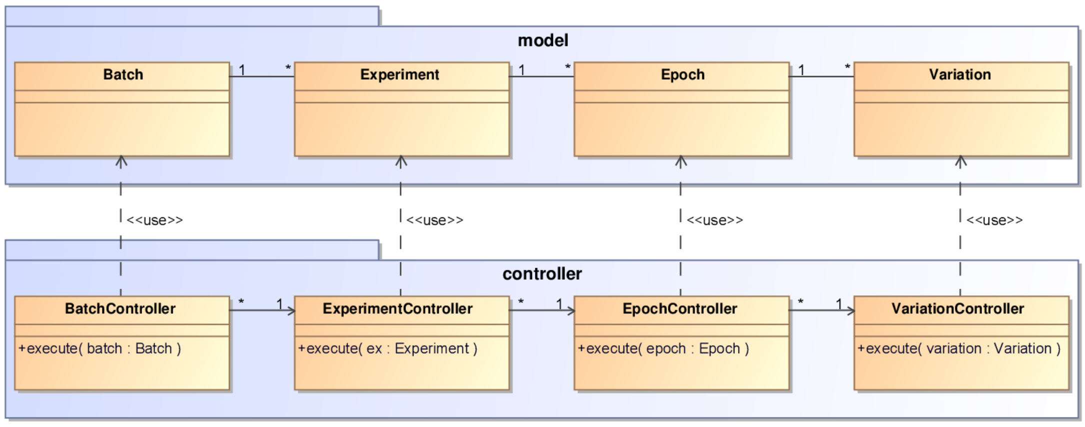


Figure 4.3 – The central architecture of BRAIn.

the execution model just explained. One key aspect in the design of BRAIn is the separation of the model and the logic, which is clearly visible in the UML diagram. The top row of classes represents BRAIn's core model. Theses classes hold all the information as well as the structure. Just like it was explained in the previous section, a `Batch` forms the base object for each BRAIn invocation. This object contains a list of `Experiment`s, which in turn contain a list of `Epoch`s, which in turn contain a list of `Variation`s.

All of these classes are designed to have minimal state. Minimizing mutability eliminates a huge share of typical software reliability problems as indicated in "Effective Java" [Blo08]. According to this book, a class should be immutable, unless there is a very good reason to make it mutable. As the `Batch`, `Experiment`, `Epoch`, `Variation` object tree cannot be constructed entirely during initialization, at least some of the classes have to be mutable. It should be noted, that the classes mentioned in the diagram are in fact immutable, but this might not be the case for their sub-classes, due to reasons just explained.

The second row in the diagram contains the so called controllers. As mentioned earlier, BRAIn uses separation of model and logic. While the model classes have just been explained, the controller classes, which contain the logic, are explained now. It's apparent that each model class has a corresponding controller class. A special property of controllers is that they are completely stateless and therefore immutable, as demanded by *Effective Java* [Blo08]. As "immutable objects are always thread-safe" [GPB+09], only little effort is required, to fulfill BRAIn's high concurrency demands (requirement 4). Being stateless, the controller classes should actually be seen more like services, an idea suggested in [Pra09]. Each controller offers the service to execute one particular `Batch`, `Experiment`, `Epoch` or `Variation` type and each controller utilizes the services of other classes. All of these classes are designed to be extended to provide maximum flexibility for the experiment execution (requirement 7).

BRAIn uses a couple of other services, most of which are living in the `brain-common` module. The `ResourceUtil` service is used to read, write or copy files, to create directories or to read Java resources. `Random` serves as the central source of random numbers. It uses a pseudo random number generator internally, the seed of which can be configured to produce deterministic results.

BRAIn's execution model is reflected in its working directory structure, the details of which are explained in Sec. 4.2.5. The service, responsible for managing this file system structure, is called `Path`. This class provides methods, which return the correct file system location for batches, experiments and so on. It also provides access to many commonly used files.

In the previous section it was mentioned, that concurrency in BRAIn is actually controlled by a special service. The corresponding interface is called `Executor` and there are two implementations, the first of which is for single threaded execution. By providing a separate class for this, the code stays clean and simple and it is possible to decide at runtime to turn off multithreading altogether. The second implementation obviously is multithreaded. It utilizes a thread pool to execute all submitted jobs. The size of this pool determines the details of the timing behavior. Two independently configurable instances of this service are created at runtime, one for experiments and one for variations. The former is usually unconstrained in pool size, while the latter should be constrained to the amount of available CPU cores. This architecture reflects the details mentioned in the execution model earlier and leads to high parallelization (requirement 13).

4.2.4 ARGoS Specific Features

As ARGoS is the only supported simulator by now, BRAIn has several components targeting this software, all of which are located in the `brain-argos` module. The following sections describe the most important features of this module.

Argos Configuration Substitution

BRAIn supports ARGoS configuration templates, which are simple text files, which contain most of the experiment configuration, as usual. However, the templates can also contain special predefined or custom variables, which are then substituted by their specific value, just before ARGoS is executed.

The classes responsible for this process are `ArgosConfigSubstitution` and `Argos-Launcher`. The former class, which is described in the following section, defines the mapping from variable names to values and the latter class performs the string substitution before launching ARGoS.

Variable names have to start with a $ sign to distinguish them from other strings. The `ArgosConfigSubstitution` contains several predefined variables:

`$randomSeed` The pseudo random number generator seed, assigned to this ARGoS instantiation.

`$modulesBaseDir` The directory containing all the ARGoS modules. All module paths should be specified relative to this folder.

`$epochDir` The directory containing the current epoch (see Sec. 4.2.5).

`$epochVariationDir` The directory, which contains the current epoch variation (see Sec. 4.2.5).

`$numBots` The number of robots to use in the experiment.

Launching ARGoS

The `ArgosLauncher` interface and its default implementation `ArgosLauncherImpl` allow to start ARGoS with a specific configuration. The class takes care of all the necessary steps, like creating the ARGoS configuration file from the template, executing the process in the right directory, redirecting the output to the logging system and checking the exit code.

The `ArgosLauncher` interface contains only one method, which allows to start the simulator. The method is blocking, which means that it does not return until ARGoS terminates successfully or unsuccessfully. If the exit code is not 0, the method throws an exception.

```java
public interface ArgosLauncher {
  void run(String workingDir, String argosConfigTemplatePath,
      ArgosConfigSubstitution argosConfigSubstitution);
}
```

The `workingDir` parameter specifies the directory, where ARGoS is executed. The `argosConfigTemplatePath` argument specifies the ARGoS configuration template file to use and the `argosConfigSubstitution` specifies the configuration substitution map introduced in the previous section.

Controller BRAIn Communication

To implement complex experiments and to provide maximum flexibility, BRAIn supports two separate communication channels to pass information from BRAIn to ARGoS and to the robot controllers and vice versa. To avoid inter process communication, the channels are based on files, which are written by BRAIn and read by ARGoS and/or by the controllers or the other way around. Google Protocol Buffers (see Sec. 4.2.6) are used to specify the file format and to serialize and deserialize data. Fig 4.4 illustrates the communication architecture, along with the components typically participating during an experiment.

The first communication channels is the ARGoS experiment configuration file, which is indicated in the upper part of Fig. 4.4. A template of this file has to be provided for each experiment within the BRAIn batch configuration. The template can contain predefined or custom variables, which are replaced according to the specific experiment,

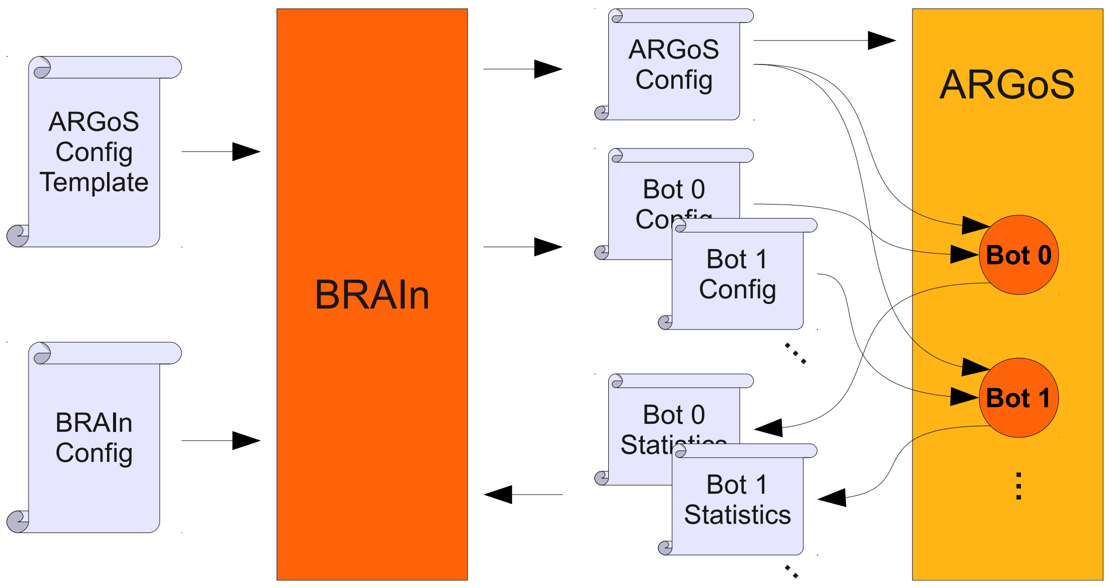


Figure 4.4 – BRAIn to ARGoS / ARGoS to BRAIn communication.

epoch and variation. Therefore, this file can be used to pass arbitrary information to ARGoS or to one of its modules, including robot controllers.

The second channel, which is indicated in the lower part of Fig. 4.4, is designated for BRAIn-to-controller and controller-to-BRAIn communication. As indicated in the figure, this channel is bidirectional and made up of two files per robot controller. Within each experiment variation, BRAIn creates one of these files for each controller. After the experiment is finished, each controller writes statistics information into its own statistics file. The entire set of statistics file can then be read and processed by BRAIn.

As the entire communication is based on files, all of the experiments are automatically documented, which leads to a high level of traceability (requirement 8). As all of the information is stored on disk, experiments can be easily suspended and resumed (requirement 14).

4.2.5 The Directory Structure

The execution model introduced in Sec. 4.2.2 is reflected in the directory structure, which is created while BRAIn executes a batch of experiments. Fig. 4.5 shows an example directory structure, which is explained in the following paragraphs.

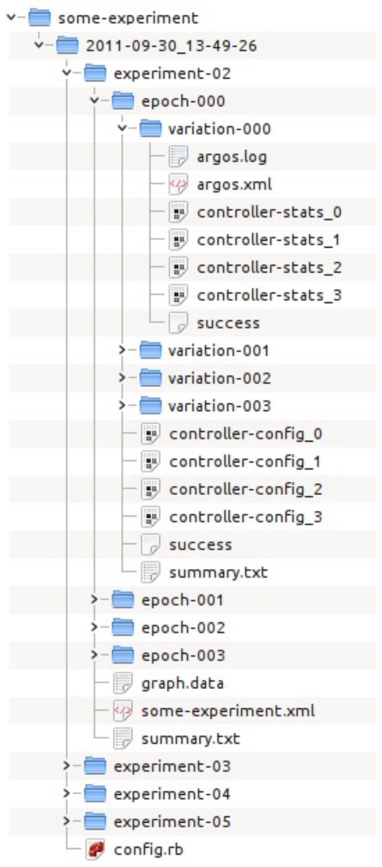
The root directory (`some-experiment` in the example) for a batch has to be specified in the BRAIn batch configuration using the `working_dir` parameter. As the same batch configuration often is executed multiple times, a sub directory (called the batch directory) is created using a timestamp (`2011-09-30_13-49-26`). To be able to resume batch execution after a planned or unplanned cancellation, the batch configuration is copied into this directory. The configuration is always named `config.rb`, no matter what the original file name is.

Within this directory, a sub-folder is created for each experiment. This folder is named `experiment-<id>`, where `id` is the experiment identifier, which is an arbitrary integer. The assigned ARGoS configuration template file is copied into each experiment directory, which again is necessary to be able to resume batch execution.

During execution BRAIn creates two statistics files for each experiment. The content of these files is specific to the experiment of Chap. 5, but can be easily adapted, to support other experiments. The file `summary.txt` is human readable and contains experiment progress information, as well as base energy, relative survival time and fitness trend information. The `graph.data` file contains character separated values, intended to be either plotted or analyzed further. The file contains one line per epoch, containing the average and maximum fitness, the average and maximum relative survival time and the average and maximum collected energy.

Each epoch has its own sub directory in the associated experiment directory, which contains all epoch specific files. As the epoch defines the configuration of the robot controllers, this is the place for the controller configuration files, which form the second communication channel. Just like it is the case for experiments, the epoch directories contain human readable statistics in the `summary.txt` file. These files contain the collected energy, relative survival time and calculated fitness for each robot in a ranked list (i.e. the fittest controller is at the top) as well as average values. The `success` file marks the epoch as successfully completed. It is an empty file, serving as a flag, that is required for telling BRAIn which epochs have been successfully completed, when resuming a batch (requirement 14).

The `summary.txt` and `graph.data` files are the result of the reporting feature of BRAIn, as demanded in requirement 15.



```
some-experiment
  2011-09-30_13-49-26
    experiment-02
      epoch-000
        variation-000
          argos.log
          argos.xml
          controller-stats_0
          controller-stats_1
          controller-stats_2
          controller-stats_3
          success
        variation-001
        variation-002
        variation-003
        controller-config_0
        controller-config_1
        controller-config_2
        controller-config_3
        success
        summary.txt
      epoch-001
      epoch-002
      epoch-003
      graph.data
      some-experiment.xml
      summary.txt
    experiment-03
    experiment-04
    experiment-05
    config.rb
```

Figure 4.5 – The file system structure created during a typical BRAIn run.

Each epoch variation has its own sub directory in the corresponding epoch folder. This directory contains a `success` file, as well, to tell BRAIn, which of the variations have been successfully completed. The complete ARGoS configuration file is also present in this directory as it is specific for each epoch variation. While ARGoS is running, its standard and error output is redirected to the `argos.log` file. Each controller can place information collected during the experiment in its own file (`controller-stats-<id>`). These files constitute the second communication channel.

4.2.6 Implementation

This section describes several key implementation aspects of BRAIn and the reasons, which lead to the choices while developing.

Java SE 6

As correctness (requirement 1), reliability (requirement 2), portability (requirement 5) and implementation speed were key requirements during the project, Java SE 6 and the Java programming language were chosen [NK05]. After almost two decades of development Java is now one of the most popular and most reliable software platforms. Just-in-time compilation and adaptive optimization make the *HotSpot Virtual Machine* (also known as *Java Virtual Machine*, JVM) one of the fastest on the market, thus virtually closing the performance gap to native code. As BRAIn does not perform any complex computations, performance is not an issue anyway. However, the Java ecosystem was definitely the crucial factor in the selection of Java. Tools like Eclipse and Maven and libraries like Guice and log4j significantly reduce the programming burden and lead to improved code quality.

Maven

Maven (see [mav]) is one of the most popular build systems for Java based projects. It uses convention over configuration to define exactly how projects have to be structured, which significantly reduces the amount of work required to set up and maintain projects. Maven's dependency management capabilities enable simple third party library usage by specifying its group and artifact ID in the project configuration. These capabilities can also be used to divide a project into several sub projects or modules, thus improving the extensibility of the software (requirement 3).

JRuby

The JVM supports many different languages, among them the Ruby scripting language. The *JRuby* (see [jrub]) project is a re-implementation of the popular Ruby language, which compiles scripts into JVM byte code, thus allowing to run Ruby scripts just like Java programs. The advantage of this approach is that Ruby code can use Java code and vice versa. This allows, for example, to define a class in Java and then instantiate or even extend it in Ruby.

This powerful system was chosen as the foundation for BRAIn configuration files. The choice of a scripting language for the configuration files has the advantage that experiments can be specified in a flexible way (requirement 10). One can use loops and conditions within the configuration or even query data from external sources, thus improving the usability of the software (requirement 6).

Using a general purpose language like Ruby for configuration files is an example for *internal domain specific languages* (DSL) as defined by Martin Fowler [Fow].

JUnit and Mockito

To guarantee correctness (requirement 1), BRAIn was developed with a high emphasis on unit testing. To test the software in an efficient and fully automated way, the *JUnit* framework (see [jun]) is used. This framework allows to test small units of a software (i.e. classes) with test cases, which report any faulty behavior. JUnit is one of the most famous frameworks in this area and was chosen because of its low implementation overhead and good integration into Maven. The *Mockito* framework (see [moc]) allows to replace (i.e. mock) the dependencies of a class for testing purposes, thus ensuring that only the class under test and not its dependencies are tested. Mockito was chosen because it is easy to use and extremely flexible. The framework uses an internal DSL based on Java for configuration.

Guice

Google's *Guice* (see [gui]) is a novel and light weight dependency injection library, which allows to write massive projects in a modular way, thus improving extensibility (requirement 3). Dependency injection eliminates the need for compile time dependencies between classes, thus allowing effective unit testing by plugging in mock objects for all of the dependencies of the class under test. Guice is very light weight and causes almost no runtime overhead.

Argument	Description
`-c FILE`	Specifies a configuration file.
`-r DIRECTORY`	Resumes a previously started batch.
`-s`	Enforces single-threaded execution.

Table 4.1 – BRAIn command line options.

log4j and slf4j

The *log4j* (see [log]) and *slf4j* (see [slf]) libraries are a popular choice for flexible and configurable logging in Java applications. The latter library provides a logging interface, for which multiple different implementations can be plugged in, thus allowing maximum flexibility. One of these implementations is *log4j*. The big advantage of the latter is its configurability, as each class has its own logger, which is usually named after the fully qualified name of the class. A configuration file can define multiple appenders, which redirect the log output to a destination, like a file, standard out or a service like an SMTP server. The exact format of each log message can be configured and the minimum log level can be set for each logger independently. This allows to get the maximum possible details, when necessary, without flooding log output with unimportant messages, thus fulfilling requirement 16.

Google Protocol Buffers

The *Google Protocol Buffers* (see [pro]) provide a way to encode data structures in a very lightweight format. The project was initially developed at Google to improve internal data center communication, but it can just as well be used to write structured data to a file and read it later on. One of the key advantages is that the data format can be specified in a domain specific language, which is then compiled into the required target languages. This allows to specify data formats once and then use them from every programming language imaginable, which improves extensibility (requirement 3) and correctness (requirement 1) and allows a flexible communication between BRAIn and the simulator (requirement 12). The following listing shows an example protocol definition:

```
message Person {
  required int32 id = 1;
  required string name = 2;
  optional string email = 3;
}
```

Protocol Buffers are used for the BRAIn / controller communication channel implementation, which is based on various files as mentioned earlier.

4.3 Using BRAIn to Run Experiments

This section describes how to use BRAIn to run experiments and how to extend BRAIn to support custom experiments and simulators. To start using BRAIn, the system first has to be prepared according to App. B.2. The BRAIn command line options are summarized in Tab. 4.1.

4.3.1 Invoking BRAIn From the Command Line

After building BRAIn, using the included Maven configuration files, according to App. B.2 several `.jar` files are created. The build process is configured to create a special JAR file, which contains the BRAIn launcher, along with all of its required dependencies (`brain-launcher-1.0-SNAPSHOT-jar-with-dependencies.jar`). Therefore, starting BRAIn is as easy as executing the following command:

```
$ java -jar brain-launcher-1.0-SNAPSHOT-jar-with-dependencies.jar
```

Executing a Batch Configuration

However, executing the framework without any parameters simply prints an error message to the screen. By adding the `-c` parameter, followed by the path to a batch configuration file, BRAIn starts executing the specified experiments:

```
$ java -jar brain-launcher-1.0-SNAPSHOT-jar-with-dependencies.jar -c
    batch.rb
```

Resuming a Previously Started Batch

The details of the ARGoS batch configuration files are explained in the next section. If a batch has already been started, but was aborted, for example due to service interruption, BRAIn can resume the experiments, reusing all the successfully completed simulator invocations. This can be accomplished, by adding the `-r` flag, followed by the path to the batch directory. The complete path is made up of the BRAIn working directory, followed by the batch name and followed by a timestamp.

```
$ java -jar brain-launcher-1.0-SNAPSHOT-jar-with-dependencies.jar -r
    some-batch/2011-10-31
```

Enforcing Single Threaded Execution

As mentioned earlier, one of BRAIn's essential features is parallelism, which is implemented by using multiple threads. As multi-threading can make debugging a lot more complicated, BRAIn supports the command line switch `-s`, which disables this feature and enforces single threaded execution. This flag overrides the settings of the batch configuration.

```
$ java -jar brain-launcher-1.0-SNAPSHOT-jar-with-dependencies.jar -s -r
    some-batch/2011-10-31
```

Debugging BRAIn

When developing new experiments, it is often necessary to use a debugger to step through the code, use break points and to analyze internal data structures during runtime. Fortunately, the JVM includes excellent debugging support, which can be enabled at the command line using several flags:

```
$ java --Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=y,address
    =8000 jar brain-launcher-1.0-SNAPSHOT-jar-with-dependencies.jar -s -
    r some-batch/2011-10-31
```

Not all of the flags are required, but this combination should support most usecases. The `address=8000` option specifies the TCP port, which is used to listen for the debugger. This port, therefore, has to be specified in the debugging application (eg. Eclipse). The `suspend=y` flag ensures that the application is not executed until a debugger is connected, which is useful for debugging initialization code. If it is preferred to start the application immediately this flag can be omitted. While debugging, it is recommended to enforce single threaded execution by appending the `-s` flag.

4.3.2 Using Predefined Functionality

This section describes how to run experiments by writing batch configuration files. The first part gives an overview about the syntax and semantics of these configuration files, before giving an overview of all the built in options. The second part shows the power of the JRuby based configuration solution, using several examples.

Running a Predefined Experiment Class

As mentioned previously, the experiments to execute are specified in a batch configuration file. The configuration is entirely based on JRuby, which means that the files are actually Ruby scripts, executed during BRAIn initialization. To learn more about the Ruby programming language consider reading [Fit07]. This section explains the syntax as well as all options, usable in this file. The following listing shows an example batch configuration with only one experiment:

```ruby
working_dir     "#{Dir.pwd}/results"
module_base_dir "#{Dir.pwd}/argos-modules"
batch_name      'test-batch'

num_experiment_workers   0
num_variation_workers   10

require 'brain-foraging-evolution-1.0-SNAPSHOT.jar'
require 'brain-foraging-common-1.0-SNAPSHOT.jar'
require 'brain-argos-1.0-SNAPSHOT.jar'

add_module Java::de.lmu.ifi.pst.ascends.brain.argos.ArgosModule.new
add_module Java::de.lmu.ifi.pst.ascends.brain.experiments.foraging.evolution.common.EvolutionModule.new

import Java::de.lmu.ifi.pst.ascends.brain.experiments.foraging.evolution.ForagingEvolutionExperiment
import Java::de.lmu.ifi.pst.ascends.brain.experiments.foraging.evolution.common.CrossoverType

add_experiment ForagingEvolutionExperiment.new(0, 17, "argos.xml", 300, 10, CrossoverType::UNIFORM_8, 0.05, 2, 2, 7, {}, [42])
```

The contents of this simple configuration file are explained line by line. In the first line the *working directory* is set, using the `working_dir` command. The working directory is the place, where BRAIn creates a directory for each batch type, which in turn contains the batch directory that is named using the current timestamp. This line makes use of the built-in Ruby class `Dir`, which allows to access the current working directory, using the `pwd` method.

The second line defines the path to a directory, where ARGoS looks for loadable modules, using the `module_base_dir` command. Again, this line utilizes the `Dir` class to access the current working directory.

Lines 5 and 6 configure the `Executor` services. Two of these services exist in BRAIn. One of them is responsible for executing experiments. The maximum number of simultaneously executed experiments can be set using the `num_experiment_workers` parameter. Usually it is not necessary to limit the number of experiments executed in parallel, because the number of simultaneous simulator invocations is limited by the second `Executor` service. Specifying 0 workers means that the number of worker threads is unlimited. The second executor is responsible for executing epoch variations and can be configured using the `num_variation_workers` command. As epoch variations are the lowest level in the execution model, this parameter directly controls the number of simulator instances running in parallel. Therefore, it is required to set this parameter to a sane value. Usually, the number of available CPU cores is a good choice. If one of the parameters mentioned in this paragraph is not specified, BRAIn sets it to 0, which leads to unlimited parallelism.

To make use of the classes defined within the BRAIn framework, the according Java modules first have to be included using the `require` command as in lines 8-11. The argument of this command has to be the full path to the `.jar` file to include. In this case, however, it is assumed that all of the `.jar` files are in the same directory, which reduces the path to just the file name.

Each of the modules, included using the `require` command, has to be initialized once by calling `add_module`. The argument of this command consists of three parts. The first part is the `Java::` prefix, which tells JRuby that a Java class is referenced, instead of a Ruby class. The second part is the fully qualified Java class name of the module to load. Last but not least, the module has to be instantiated by appending the `new` method. These lines actually instantiate a Java class, pass the resulting object to the `add_module` method, which then invokes a method withing BRAIn, thus loading the module.

In lines 15-16, two Java classes are imported using the `import` statement, which is JRuby specific and does not appear in the Ruby programming language. The semantics of this statement is similar to the identically named Java statement. The purpose is to make the specified class available, without having to use the fully qualified class name every time. Again, the `Java::` prefix is used to indicate that this is not a Ruby, but a Java class.

Finally, in line 18, an experiment is defined, by instantiating the experiment class (`ForagingEvolutionExperiment` in this case). The result has to be passed to the `add_experiment` method, to add the experiment to the batch. The parameters of the experiment depend on the specific experiment class constructor.

Advanced Experiment Configuration

As batch configuration files are essentially Ruby scripts, they can be used to define a huge set of experiments with only a few lines of code. The following listing shows an excerpt from a more advanced configuration file:

```ruby
def calculate_num_connections(num_hidden)
  return (10 + 1) * num_hidden + (num_hidden + 1) * 2;
end

experiment_id=0
(2..5).each do |num_hidden|
  CrossoverType.values.each do |crossover_type|
    [0.1, 0.01, 0.001].each do |mutation_probability|
 9       [-1, -2, -3].each do |power_exp|
10         standbyPower = 10**power_exp
11         drivingPower = 10**(power_exp-1)
12         genome_length = calculate_num_connections(num_hidden)
13         argosParams = {'$numHiddenNeurons' => num_hidden, '$standbyPower
               ' => standbyPower, '$drivingPower' => drivingPower}
14         ex = ForagingEvolutionExperiment.new(experiment_id, 17, "argos.
               xml", 300, 10, crossover_type, mutation_probability, 2, 2,
               genome_length, argosParams, [1, 2, 3, 4])
15         print "#{ex}\n"
16         add_experiment ex
17         experiment_id+=1
18       end
19     end
20   end
21 end
```

The omitted lines are similar to lines 1-16 from the first example. This configuration creates 108 experiments with various different parameters using Ruby control statements. In line 5 a variable is defined, which is incremented in the inner most loop (line 17) to create unique IDs for the experiments.

The outer most loop (line 6) iterates over different numbers of hidden neurons used for the experiment. The `2..5` statement creates a range, containing the numbers 2, 3, 4, and 5. The `each do |num_hidden|` iterates over all the numbers in the range, assigning one of them to the `num_hidden` variable during each iteration.

When conducting neuroevolution experiments, BRAIn has to know the exact length of the genome, which is equal to the number of connections in the neural network. This number can be calculated using Eq. 2.21. If the number of input and output neurons is fixed, this parameter only depends on the number of hidden neurons. As this parameter changes due to the loop in line 6, the script needs a way to calculate the correct number of connections and therefore the correct genome length. Thanks to the usage of a real programming language it is easy to define utility methods. In this example, the method in lines 1-3 performs the task of calculating the number of connections and therefore the genome length. The method is used in line 12 to calculate the experiment argument.

The second loop (line 7) iterates over all possible cross over types of the genetic algorithm. The third loop (line 8) iterates over three possible mutation probabilities for the genetic algorithm. The details of these parameters are explained in Sec. 5.2.2.

The inner most loop (line 9) controls the power consumption of the robots in the environment. The loop iterates over three different exponents, which are then used in lines 10 and 11 to compute different standby and driving power parameters for the robots in the experiments. The details of these parameters are explained in Sec. 5.1.1.

Using Ruby control statements, the example script manages to create all possible combinations of four different parameters and to instantiate an experiment for each of them, with only few lines of code. For debugging purposes it is also possible to print each experiment configuration using the Ruby `print` method. This statement (line 15) calls the objects' `toString()` method to produce human readable output.

All of the commands built into the BRAIn configuration system are listed in Tab. 4.3. For non BRAIn specific statements consider reading *Learning Ruby* [Fit07], the Ruby reference documentation [rub] and the JRuby documentation [jrua].

Statement	**Type**	**Required**	**Description**
`working_dir`	String	yes	Sets the BRAIn working directory path. The batch directories are created within this directory.
`module_base_dir`	String	yes	The directory where ARGoS will look for loadable modules.
`batch_name`	String	yes	The name of the batch defined in this configuration file.
`num_experiment_workers`	int	no	The number of experiment worker threads. If 0 (the default) the number of workers is unlimited.
`num_variation_workers`	int	no	The number of epoch variation worker threads. If 0 (the default) the number is set to the number of available CPU cores.
`add_experiment`	Experiment	no	Adds an experiment to the batch.
`add_module`	Module	no	Loads a module into BRAIn.

Table 4.3 – BRAIn batch configuration referencce.

4.3.3 Extending BRAIn

While the previous sections dealt with invoking BRAIn and using pre-defined experiments, this section explains how to extend the framework and how to write custom experiment classes.

Adding a New Project

As mentioned in Sec. 4.2.1, BRAIn is divided into several modules or sub-projects. The reason for that is to keep the source code modular and understandable. Before being able to define new experiment classes, it is recommended to create a new sub-project. It is also possible, but not recommended to add the classes to an existing module.

The following instructions assume that the new experiment project is created within the BRAIn source tree. This is not required, but simplifies things a little bit. To add a new experiment project, go to the `brain-experiments` sub-folder within the BRAIn source tree and execute the following command. The `-DartifactId` parameter specifies the name of the project and can be replaced by an arbitrary string. However, it is recommended to stick to the `brain-` prefix.

```
mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false -Dversion="1.0-SNAPSHOT" -DgroupId=de.lmu.ifi.pst.ascends.brain.experiments -DartifactId=brain-myexperiment -DpackageName=de.lmu.ifi.pst.ascends.brain.experiments.myexperiment
```

After the command finishes, it will have created a new directory named like the specified `artifactId` (`brain-myexperiment` in this example). To include the new project into BRAIn's build process it has to be added to the list of modules in the `pom.xml` file of the `brain-experiments` directory. If Maven has been invoked correctly, this should have happened automatically. You can verify this by opening the file and looking for the `<modules>` section, which should look something like this:

```
1 <modules>
2   <module>brain-foraging-common</module>
3   <module>brain-foraging-basic</module>
4   <module>brain-foraging-evolution</module>
5   <module>brain-foraging-neat</module>
6   <module>brain-myexperiment</module>
7 </modules>
```

Verify that the new project was added to the list (line 6 in this case). Now also open the `brain-myexperiment/pom.xml` file in the newly created project folder and look for the `<parent>` section.

```
<parent>
  <artifactId>brain-experiments</artifactId>
  <groupId>de.lmu.ifi.pst.ascends.brain</groupId>
  <version>1.0-SNAPSHOT</version>
</parent>
```

If it looks like in this example, the project was successfully added to BRAIn's build system. To build BRAIn and the new project, go up one level to BRAIn's root source folder and execute the following command:

```
$ mvn install
```

After Maven finishes building the project, it prints a summary to the screen. Verify that the new project is part of that list. To use the new project in the Eclipse IDE follow the instructions in App. B.2.

Maven creates an example class, called `App` and an example test, called `AppTest` in the newly created project. Both classes can be safely removed.

Defining Custom Experiment Class

Now that a new project has been defined, the actual classes representing the experiments can be written. As introduced in Sec. 4.2.3 and in Fig. 4.3, there are eight core classes, which define the behavior of an experiment. It is possible to extend each of these classes to introduce new behavior. However, in many cases it is sufficient to extend only few classes and utilize existing ones for the missing parts.

The first pair of classes (`Batch` and `BatchController`) usually does not need to be changed, as a batch of experiments is always executed in the same way. Using a custom `Batch` or `BatchController` is unsupported right now, as the `Launcher` always uses `DefaultBatch` and `BatchControllerDefault`.

The remaining three pairs of classes (`Experiment`, `Epoch`, `EpochVariation` and their associated controllers) are usually extended to define new experiments. However, it is also possible to extend one of their sub-classes or to extend only some of them and reuse others.

Defining Modules

As mentioned in Sec. 4.2.6, BRAIn uses Google Guice for dependency injection. Guice is configured, using modules, which are Java classes, extending the `AbstractModule` class from the `com.google.inject` package. The modules are specified using a domain specific language, built on top of Java. For further details about dependency injection and Guice module definition refer to the Guice homepage[3].

4.4 Further Improvements

BRAIn has reached a mature state and can be used for various types of experiments in many environments. Nevertheless, there are many features, which didn't make it into the code base yet. The following is a list feature ideas, which might be worth implementing in the future.

4.4.1 User Configuration File

It would be useful to add a user specific BRAIn configuration file, located in the user's home directory (`~/.brain.rb`). This would allow users to specify common parameters like the number of execution threads locally on their machine, instead of changing the experiment configurations all the time. The system should be implemented in a way that configuration parameters can be overwritten in the experiment configuration, if necessary.

[3]http://code.google.com/p/google-guice/, accessed 18. Dec. 2011

4.4.2 Remote Simulator Invocation

As many experiment batches consist of thousands of individual ARGoS calls, computing resources become the limiting factor in many cases. To overcome this issue and to facilitate cluster architectures, a remote execution service should be implemented. This could be realized as an additional *ArgosLauncher*, which takes the remote host as a parameter. The remote execution itself could be realized via SSH. Assuming that a shared file system exists, this would solve all of the communication requirements. To utilize the available computing resources as efficiently as possible, an additional service would have to keep track of existing resources, their properties as well as the assigned and the queued computing jobs.

4.4.3 Multiple Simulator Instances Within a Variation

Sometimes it would be interesting, whether an experiment would end differently if the robots were unable to influence each other. Therefore, it would make sense to add an additional flag to the epoch variation controllers, which would lead to each robot getting simulated in its own ARGoS instance. This approach could be designed in an even more flexible way if the user were able to decide, how many robots should be assigned to the same simulator instance. For example, one could have an experiment with 20 robots and two simulator instances, assigning 10 robots to each of them.

4.4.4 Reduced Number of Runtime Files

Last but not least, BRAIn produces a huge amount of files during a typical execution. The number of files quickly becomes so large, that backing up the data with archiving tools, like *tar*, takes a very long time. It would make sense to think about ways to reduce the number of files created during execution to overcome these problems.

Chapter 5

Using BRAIn for Foraging Experiments

This section describes the experiments, conducted and analyzed using the BRAIn framework and the ARGoS simulator. The first experiment uses a random-walk approach. The second experiment applies a genetic algorithm to optimize the random-walk controller. The third experiment uses a neuroevolution based controller. The fourth experiment uses a special purpose controller, which is designed especially for the task and the last experiment applies the genetic algorithm to optimize this controller. While the first two experiments are designed to provide a lower bound, the last two experiments are designed to provide an upper bound for the achievable performance. To conduct these experiments an environment and a task has to be designed, which is explained in the first part of this chapter. The second part describes the five experiments and presents their results. As the special purpose controller is especially tuned for this task, it is expected to perform better than all of the other algorithms. Therefore this controller is named *Uber Controller*.

5.1 The Environment

Designing a challenging, yet simple enough task to test different algorithms is anything but easy. Inspired by the papers *A Quantitative Test of Hamilton's Rule for the Evolution of Altruism* [WFK11] and *Genetic Team Composition and Level of Selection in the Evolution of Cooperation* [WKF09] an environment was designed, that is easy enough to survive but features properties that encourage the development of complex strategies, to outperform competing robots. Like in most experiments used in the literature, the environment consists of a rectangular arena, containing robots, a base and food items. The environment is illustrated in Fig. 5.1.

5.1.1 The Foraging Task

As mentioned in Sec. 1.1, foraging is the task of searching for food or provisions [for]. This type of task is often used for artificial intelligence experiments, as it is vital for all animals to collect food.

Each robot i in the simulation has an energy level $E^{(i)}(t)$, which is initialized with a certain value E_0 and cannot exceed E_{max}. Every robot has a standby energy consumption P_s and a driving energy consumption P_d. The former applies all the time, even if the robot is at rest. The latter is multiplied with the current power settings $d_l^{(i)}(t)$ and

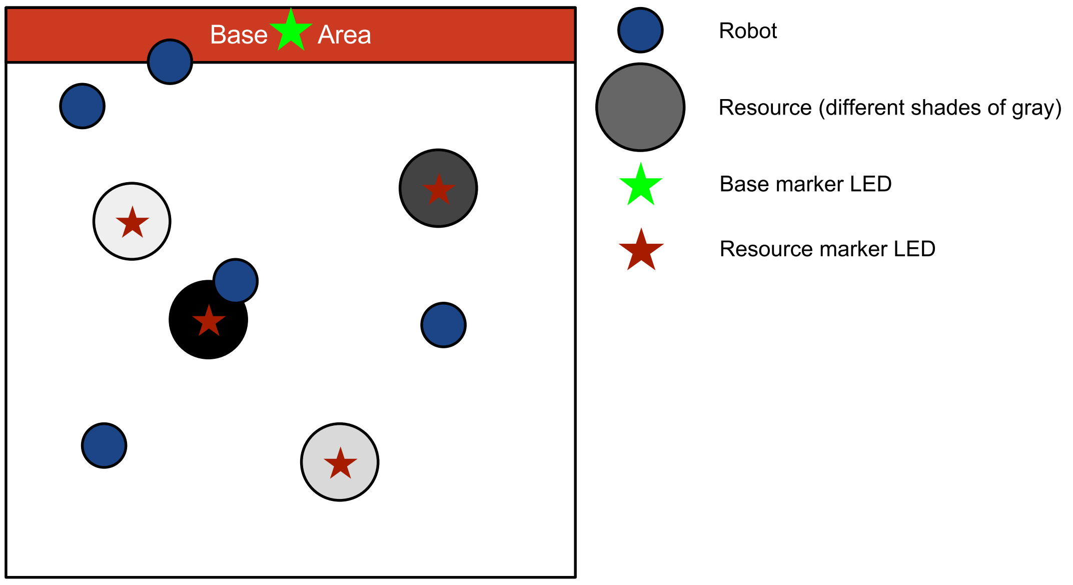


Figure 5.1 – The foraging environment with bots, resources and a base area.

$d_r^{(i)}(t)$ of the left and right treel. Therefore, the overall energy consumption at time t can be calculated according to Eq. 5.1. For each simulation step with the duration Δt, the robot's energy is updated according to Eq. 5.2. The moment $E^{(i)}(t)$ reaches zero, the robot dies and remains dead until the end of the experiment.

$$P^{(i)}(t) = P_s + \left(\left| d_l^{(i)}(t) \right| + \left| d_r^{(i)}(t) \right| \right) \cdot P_d \tag{5.1}$$

$$\Delta E^{(i)}(t) = -\Delta t \cdot P^{(i)}(t) \tag{5.2}$$

A rectangular area along one of the walls of the arena forms the so called base (often also referred to as nest). The base spans the entire width of the arena and is about one robot diameter broad. To allow robots to perceive the location of the base, a green LED is placed at the center of the wall, adjacent to the base area. Technically, each robot has its own base, but all the bases are at the exact same location. Each robot base has an energy level $B^{(i)}(t)$, which is initialized with zero. While the robot is in its base area, energy is transferred from its internal energy storage to the base with the energy drop off rate $d > 0$, according to Eq. 5.3 and 5.4. Therefore, the robot can control the amount of energy, transferred to the base, by the amount of time it spends in the designated area.

$$\Delta B^{(i)}(t) = \Delta t \cdot d \tag{5.3}$$

$$\Delta E^{(i)}(t) = -\Delta t \cdot d \tag{5.4}$$

Several food items (also referred to as resources) are distributed randomly across the arena. Each food item j has circular shape and, unlike in most other experiments, is drawn as a gray spot on the white floor by the `EnvironmentGenerator` (see Sec. 5.1.3), instead of being a three dimensional object in the simulation. Each resource has an initial value of $V_0^{(j)}$ and a current value of $V^{(j)}(t)$. The brightness $C^{(j)}(t)$ of resource j is defined by Eq. 5.5.

$$C^{(j)}(t) = 1 - \frac{V^{(j)}(t)}{V_0^{(j)}} \tag{5.5}$$

A full resource ($V^{(j)}(t) = V_0^{(j)}$) is rendered as a black spot (i.e. $C^{(j)}(t) = 0$), whereas an empty resource ($V^{(j)}(t) = 0$) is rendered white (i.e. $C^{(j)}(t) = 1$) and therefore cannot be distinguished from the white floor. When a robot i is within the area of a resource j, energy is transferred from the resource to the robot according to Eq. 5.6, 5.7 and 5.8, where h is the global harvesting rate and $H^{(j)}(t)$ is the current harvesting rate for resource j.

$$H^{(j)}(t) = h \cdot \frac{V^{(j)}(t-1)}{V_0^{(j)}} \tag{5.6}$$

$$\Delta E^{(i)}(t) = \Delta t \cdot H^{(j)}(t) \tag{5.7}$$

$$\Delta V^{(j)}(t) = -\Delta t \cdot H^{(j)}(t) \tag{5.8}$$

This means that the current harvesting rate $H^{(j)}(t)$ between resource and robot decreases with the amount of energy already harvested ($V_0^{(j)} - V^{(j)}(t)$) from that resource. Therefore, the harvesting rate $H^{(j)}(t)$ declines exponentially, but never reaches zero. This behavior is illustrated in Fig. 5.2 for a resource j with $V_0^{(j)} = 5000J$, $h = 10\frac{J}{s}$, $\Delta t = 0.1s$.

In the center of each resource is a red LED, which allows the robots to locate the items using their omnidirectional camera. However, the LEDs brightness is constant and equal among all of the resources, thus circumventing remote resource value estimation. This way robots are forced to drive to a resource, before they can measure its value, using their ground brightness sensors.

As introduced in Sec. 2.1.2, the foraging task can be classified according to several properties. The environment is partially observable, because the robots cannot perceive the values of the resources unless standing on top of them. A robot also cannot perceive another robot's current energy or base energy level. It is strategic, because it is deterministic, except for the behavior of other robots. The experiments are sequential, because the robots can perceive the environment and act accordingly for a long period of time. As the state of the environment changes, even if the robot decides not to act for a certain amount of time, it can be classified as a dynamic environment. Each robot and each resource has a two-dimensional position in space and robots additionally have an orientation. Furthermore, each resource, base and robot has an energy level. Assuming that an experiment consists of four resources and ten robots and assuming that 32-bit floating point numbers are used to represent the state, the total state space adds up to $32 \cdot (4 \cdot (2+1) + 10 \cdot (2+1+2)) = 1984$ bits or $1.8 \cdot 10^{597}$ possible states. Therefore, the environment can be considered as continuous. As several agents compete for a limited amount of resources, this is a competitive multiagent environment. The environment properties are summarized in Tab. 5.1.

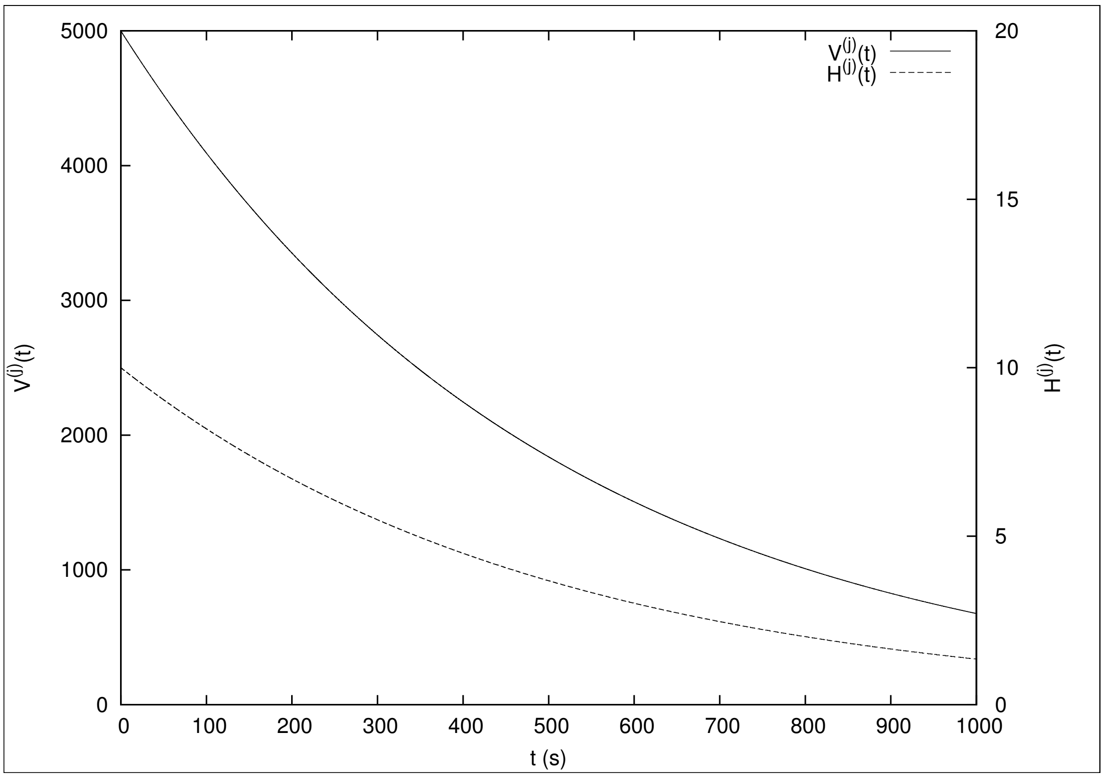


Figure 5.2 – Harvesting simulation for a resource with $V_0^{(j)} = 5000$, a global harvesting rate of $h = 10$ and simulation interval of $\Delta t = 0.1$.

Property	Foraging Task Classification
Observability	Partially
Determinism	Strategic
Episodic	Sequential
Static	Dynamic
Discrete	Continuous
Agents	Competitive Multiagent

Table 5.1 – Properties of the foraging task environment.

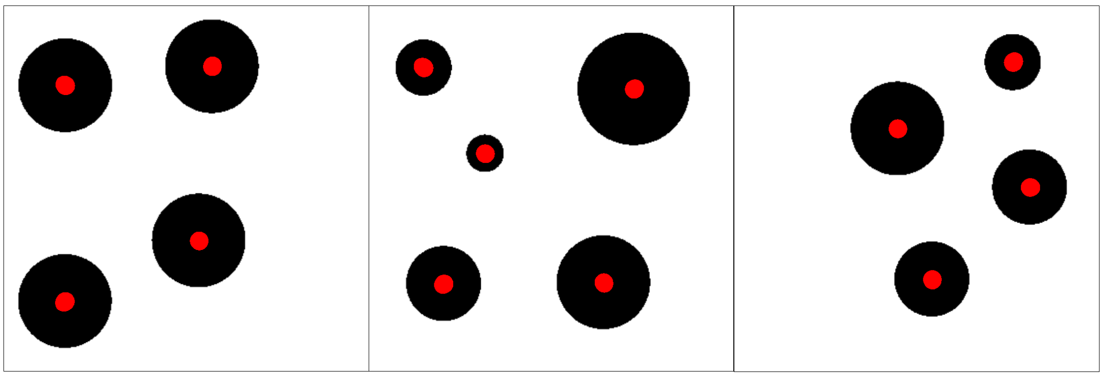


Figure 5.3 – Three environments generated with the `EnvironmentGenerator`.

5.1.2 The Fitness Function

To determine a robot's performance, a fitness function has to be defined. As the goal is to collect as much energy from the resources as possible, the amount of energy $B^{(i)}$ transferred to the base is intuitively part of the fitness function. A higher survival time should also be honored in the fitness function, as it usually leads to a higher chance of reproduction in nature. Combining these two values leads to Eq. 5.9, where $0 < t_s^{(i)} \leq 1$ is the relative survival time of robot i. If a robot survives throughout the entire simulation, the relative survival time is 1. The base energy is divided by 1 joule (J), to get a unit less fitness value. The variable t_{max} indicates the end of the simulation.

$$F^{(i)} = \frac{B^{(i)}(t_{max})}{1J} \cdot t_s^{(i)} \tag{5.9}$$

5.1.3 The `EnvironmentGenerator`

The `EnvironmentGenerator` is responsible for distributing, drawing and updating the resources, as well as for placing the marker LEDs. To support the simulation of various environments, the plugin can be configured in the ARGoS experiment configuration file. The logic is implemented using the ARGoS *loop functions* concept (see Sec. 3.2.2).

Fig. 5.3 shows three example environments, generated with the plugin for different configurations. The black circular shapes represent the resources. The resource radius $r^{(j)}$ can be configured independently for each resource, but does not affect its value. All of the resources are black, because they still contain the initial amount of energy, i.e. they haven't been harvested yet. The red circular object at the center of each resource is the marker LED, which allows robot to locate the resources.

The following listing shows an example configuration of the plugin:

```
<loop_functions library="libenvironment_generator.so"
  label="environment_generator" gap="0.5">
  <resources>
    <resource radius="0.5" value="5000" position="1,1.5" />
    <resource radius="0.5" value="5000" position="-1,-1"/>
    <resource radius="0.5" value="5000" />
    <resource radius="0.5" value="5000" />
  </resources>
</loop_functions>
```

Parameter	Required	Description
`gap`	yes	Minimum distance in meters between two resources.
`position`	no	Position of a resource.
`radius`	yes	Radius in meters of a resource.
`value`	yes	Energy in joule initially stored in the resource.

Table 5.2 – The `EnvironmentGenerator` parameters.

All distances and coordinates are specified in meters and the coordinate origin is in the center of the arena. The `EnvironmentGenerator` supports resources with and without specified `position`. If the position attribute of a `<resource>` tag is missing, the resource is placed at a random location. The `gap` parameter specifies the minimum distance between two resources. This parameter is ignored when the position of a resource is explicitly defined in the configuration. The `radius` parameter of a resource determines its size. The `value` specifies the total amount of energy (in joule), stored in the resource. Tab. 5.2 summarizes the configuration options.

5.1.4 Experiment Configuration

As mentioned in Sec. 5.1.1, the environment has several parameters, which are summarized in Tab. 5.3. The level of difficulty of the foraging task fundamentally depends on these parameters. During several manually executed experiments, the parameters have been tuned to a reasonable level. The results are presented in the second column of the table. For all the experiments, the `EnvironmentGenerator` is configured with four resources, each of which having a radius of $r = 0.5m$ and a value of $500J \cdot N$, where N is the number of robots. Therefore, the total energy in the arena is $4 \cdot 500J \cdot N = 2000J \cdot N$, which is $2000J$ per robot. The resources are distributed randomly. To keep the rate of collisions between the robots low, the number of robots in the arena has been chosen to be $N = 10$. The total simulation time has been selected to be $t_{max} = 1000s$, which is a trade off between simulation execution duration and quality of results.

Parameter	Default	Description
N	10	Number of robots in the arena.
E_h	$100J$	Initial robot energy level.
E_{max}	$200J$	Maximum robot energy level.
P_s	$0.2\frac{J}{s}$	Robot standby power consumption.
P_d	$0.05\frac{J}{s}$	Robot driving power consumption.
d	$10\frac{J}{s}$	Energy drop off rate while inside the base.
V_0	$5000J$	The initial energy for all resources.
r	$0.5m$	The radius of all resources.
h	$10\frac{J}{s}$	The global foraging rate.
t_{max}	$1000s$	The total simulation duration.

Table 5.3 – The foraging task parameters and their selected default values.

5.2 The Experiments

The following sections describe all of the experiments conducted using BRAIn and ARGoS for simulating the environment, introduced in the previous section. Each experiment is designed to analyze the performance of a particular algorithm or a group of algorithms. The first algorithm tests the performance achievable by robots that are moving randomly in the arena. The intention behind this experiment is to get a lower bound for the performance. To prove the concept behind the genetic algorithm, the second experiment uses this concept to tune the parameters of the random walk controller. The following experiments evaluate the performance of neuroevolution based algorithms. The last two experiments use a special purpose controller, which is designed specifically for the foraging task. The purpose of these experiments is to show the maximum achievable performance.

Although any other wheel based robot with basic sensors would suffice, the marXbot (see Sec. 3.1) is used for all experiments, because ARGoS (see Sec. 3.2) supports it out of the box. Each robot has its own controller instance, which is written in C++. However, the modules cannot be transferred to real robots without modification, because they contain the foraging logic, which has to access internal ARGoS features like the simulation ticks, required to record the relative survival time. Besides, the filesystem has to be accessed to communicate with BRAIn, as presented in Sec. 4.2.4. The controller can query readings from all available sensors and can choose actuator settings like wheel speeds.

5.2.1 Experiment 1 - Random Walk

The random walk experiment is supposed to give a lower bound for the achievable fitness. The robots in this experiment keep driving straight until they encounter an obstacle, which can either be another robot or one of the arena perimeter walls. In this case, the robot starts turning away from the obstacle, until it can go straight again. The overall movement is therefore random, depending on the initial position and orientation and on the behavior of other robots. However, the robots occasionally hit the base or one of the resource areas by accident, giving them a moderate chance of succeeding.

Controller Architecture

The controller is essentially based on the `footbot_obstacle_avoidance_controller` from the *argos2-examples*[1] package. The robots start driving straight forward until they encounter an obstacle, which can be either one of the perimeter walls or another robot. Once an obstacle is recognized, using the proximity sensors, the controller calculates the angle between its driving direction and the obstacle. The robot then starts to turn by stopping one of the treels and driving the other treel forward with the specified speed v. The turn direction (i.e. which of the treels is stopped) is determined by the sign of the previously calculated collision angle. A negative angle means that the collision happened on the right hand side of the robot, leading to a left turn and vice versa.

The algorithm has three parameters, which influence its performance. The first parameter is the *treel velocity* v, which has to be in the range of $]0; 20]$. If the robot drives faster, it can potentially collect more resources within a specified amount of time, but it also has a higher power consumption and it increases the chance of getting stuck due to an unavoidable collision.

[1]http://iridia.ulb.ac.be/argos/download.php, accessed 18. Dec. 2011

The second parameter α determines the angular range, for which objects reported by the proximity sensor are taken into account. The third parameter δ specifies the minimum proximity value, necessary for collisions to be recognized. For the random walk experiment, the following default values are in use: $\alpha = 7.5°$, $\delta = 0.1$ and $v = 5$.

The following listing shows the C++ implementation of this algorithm:

```
 1 const CCI_FootBotProximitySensor::TReadings& tProxReads =
       GetProximitySensor()->GetReadings();
 2
 3 CVector2 cAccumulator;
 4 for (size_t i = 0; i < tProxReads.size(); ++i) {
 5   cAccumulator += CVector2(tProxReads[i].Value, tProxReads[i].Angle);
 6 }
 7 cAccumulator /= tProxReads.size();
 8
 9 CRadians cAngle = cAccumulator.Angle();
10 if (m_cGoStraightAngleRange.WithinMinBoundIncludedMaxBoundIncluded(
       cAngle) && cAccumulator.Length() < m_fDelta) {
11   SetLinearVelocity(m_fWheelVelocity, m_fWheelVelocity);
12 } else {
13   if (cAngle.GetValue() > 0.0f) {
14     SetLinearVelocity(m_fWheelVelocity, 0.0f);
15   } else {
16     SetLinearVelocity(0.0f, m_fWheelVelocity);
17   }
18 }
```

In line 1, the controller retrieves the readings from the robot's 24 proximity sensors, which are equally spaced around the robot's chassis. The readings contain the angle of the sensor and the measured value, which gets higher when an obstacle gets closer. Lines 3 - 7 aggregate all the readings into one collision vector. In line 10, the angle, as well as the length of the vector, are checked. If the angle is within $[-\alpha; \alpha]$ (represented by `m_cGoStraightAngleRange`) and the length does not exceed δ, the robot can continue to go straight. Otherwise, the comparison in line 13 determines, whether to turn left or right.

Experiment Results

To achieve a high degree of statistical significance, the experiment was conducted for four different pseudo random number generator seeds. As the starting positions, as well as the initial orientations of the robots and the positions of the resources are determined using a peuso random number generator, this results in four entirely different starting configurations. The aggregated results of all four runs are summarized in Tab. 5.4 where F is the fitness, B is the base energy and t_s is the relative survival time.

Parameter	**Average**	**Minimum**	**Maximum**
F	116.71	10.33	206.65
B	194.20J	52.50J	269.00J
t_s	0.56	0.09	0.80

Table 5.4 – The results of the random walk foraging experiment.

5.2.2 Experiment 2 - Genetic Random Walk

As mentioned in Sec. 5.2.1, the random walk controller has three parameters (α, δ and v) that influence its performance. To test the genetic algorithm implemented for the BRAIn framework, the controller from the previous section is extended to accept its parameters as the genome vector $\vec{g}$. As is explained in the next section, each element of $\vec{g}$ is in the range $[0;1]$. The α parameter has to be in the range $[0°;180°]$, therefore its value is calculated according to Eq. 5.10, where g_0 refers to the first element of the genome. The δ value has to be in the range $[0;1]$ and therefore can be directly obtained from the genome (Eq. 5.11). The treel velocity parameter v has to be in the range $[0;20]$ and therefore is calculated according to Eq. 5.12. As the genome only has a length of 3 and 10 robots are in use, the mutation probability is selected to be 0.1, which means that, on average, $3 \cdot 10 \cdot 0.1 = 3$ mutations appear within each epoch.

$$\alpha = 180° \cdot g_0 \tag{5.10}$$

$$\delta = g_1 \tag{5.11}$$

$$v = 20 \cdot g_2 \tag{5.12}$$

The Genetic Algorithm

The genetic algorithm was designed using the concepts, introduced in Sec. 2.4. The algorithm is based on epochs (or generations), each of which contains a population of individuals. The number of individuals can be configured and is constant throughout the experiment. Each individual has a genome, which is represented as a `float[]` array, the length of which is constant for all the individuals and has to be explicitly set for each experiment. Each array entry is in the range $[0;1]$ and the array needs to have exactly the length expected by the controller. The genetic algorithm writes the genome for each individual into a separate file using Google Protocol Buffers (see Sec. 4.2.6).

The first generation is created using random genomes for the individuals. After each epoch, all the robot controllers write statistics information into a statistics file (again using Google Protocol Buffers). The GA then reads all the files and ranks the individuals according to their fitness. The C fittest individuals are then cloned for the next generation (*cloning range*) and the c unfittest individuals are culled (*culling range*). Pairs of individuals are selected from the remaining list using a triangular random distribution. This distribution is constructed in a way that the chance of an individual for being selected for reproduction is higher if it is fitter. As the process is random, an individual might be selected multiple times or not at all. For each pair, the genome is recombined into a new genome. Three different recombination operators are supported:

`SINGLE_POINT` A position within the genome is randomly determined. Every gene up to the position is copied from the first individual and the rest from the second individual.

`UNIFORM` For each genome index it is randomly determined whether it is copied from the first or the second individual. Chances are equal for both individuals.

`UNIFORM_8` The same as *UNIFORM*, except that chances that the superior individual is selected are 80% and therefore chances that the inferior individual is selected are 20%.

After producing the offspring genome, the mutation operator is applied. The chances for a genome element of getting mutated can be defined in the experiment configuration and are usually in the range of $[0.0001; 0.1]$. What values make sense depends on the genome length. If the genome has 100 elements, a mutation probability of 0.01 means that on average, one of the elements of each individual is mutated. However, if the genome only has a length of 10 the same probability means that, on average, only one out of ten individuals is subject to mutation. If an element of a genome gets selected by this algorithm, it is set to a random value in the range $[0; 1]$.

During preliminary experiment runs it was determined that the `UNIFORM` recombination type yields the best results. Therefore, this parameter is used for all of the GA experiments in this section.

Experiment Results

The results of the experiment can be seen in Fig. 5.4. A steep rise of the average fitness and base energy is clearly visible in the first few epochs, which are enlarged in Fig. 5.5. However, the survival time remains relatively constant over time. The fluctuations visible in the charts can be explained by mutations, which in most cases have a negative effect for the fitness of an individual. Tab. 5.5 shows the aggregated results for the last 50 epochs of the simulation, where F ist the achieved fitness, B is the collected base energy and t_s is the relative survival time. The top performing individual i_{top} appeared in epoch 73, collected $B^{(i_{top})}(t_{max}) = 508.75J$ of energy, survived for $t_s^{i_{top}} = 1.00$ and achieved a fitness of $F^{(i_{top})} = 508.75$. The genome of this individual was $[0.893395602703094, 0.0354135632514954, 0.32399970293045]$, which, according to Eq. 5.10, 5.11 and 5.12, translates to the following configuration (rounded to 5 significant digits): $\alpha = 160.81°$, $\delta = 0.035414$ and $v = 6.4800$. These values dramatically deviate from the configuration used in experiment 1 (Sec. 5.2.1). Especially the high α value seems surprising compared to the $\alpha = 7.5°$ used in the manual configuration of experiment 1. The δ value also deviates with a factor of more than 10. The velocity v, on the other hand is only slightly higher than in experiment 1. Tab. 5.6 shows an overview of the manually selected values of experiment 1 and the algorithmically selected parameters for the winning robot of this experiment.

Parameter	Average	Minimum	Maximum
F	101.16	19.70	206.89
B	182.32	65.17	280.24
t_s	0.51	0.22	0.81

Table 5.5 – Aggregated results for the last 50 epochs of the random walk foraging experiment.

Parameter	Experiment 1	Experiment 2
$\alpha/1°$	7.5	160.81
δ	0.5	0.035414
v	5.0	6.4800

Table 5.6 – Comparison of the manually selected parameters of experiment 1 and the algorithmically selected parameters of experiment 2.

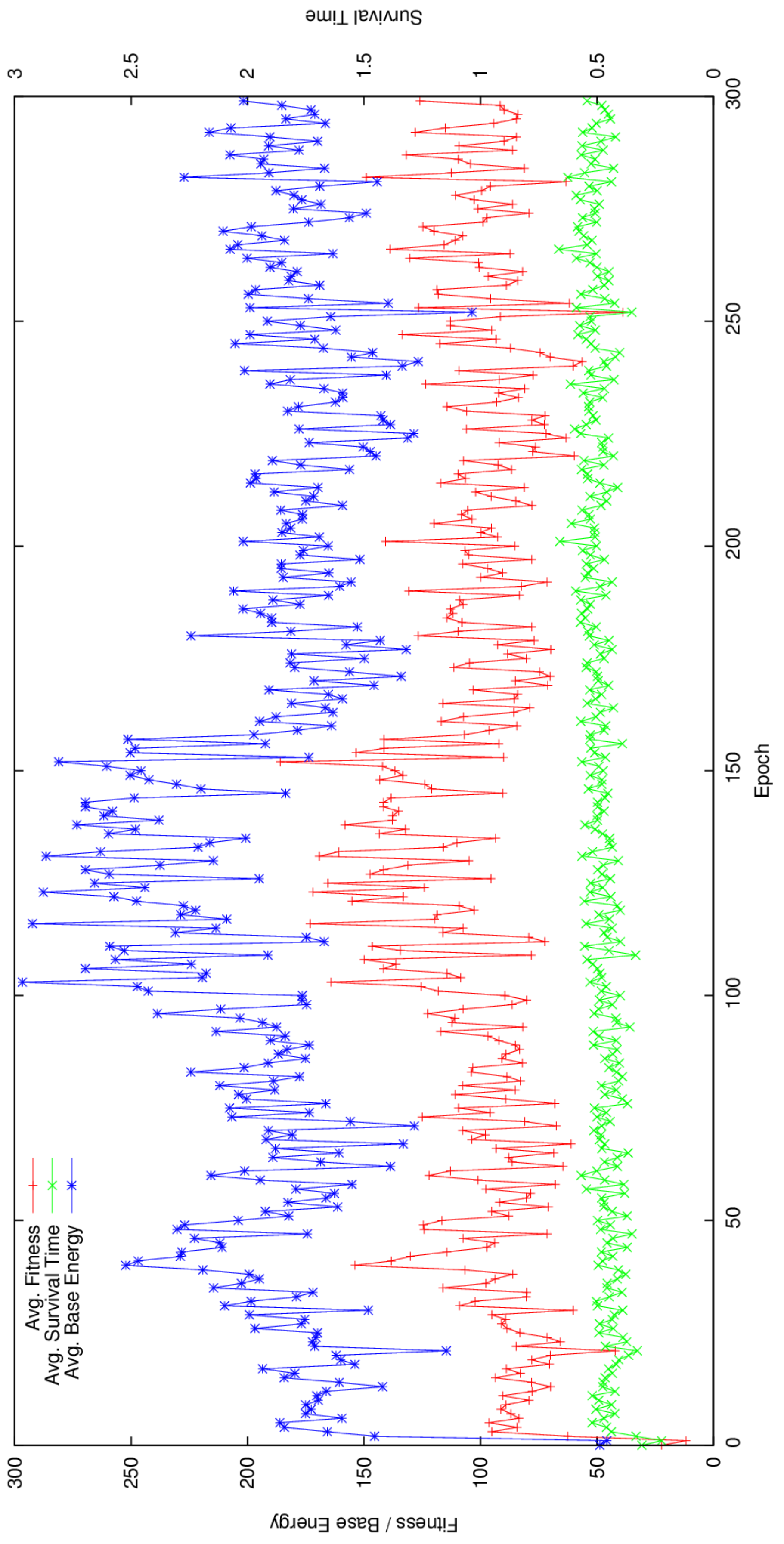


Figure 5.4 – The results for all epochs of experiment 2.

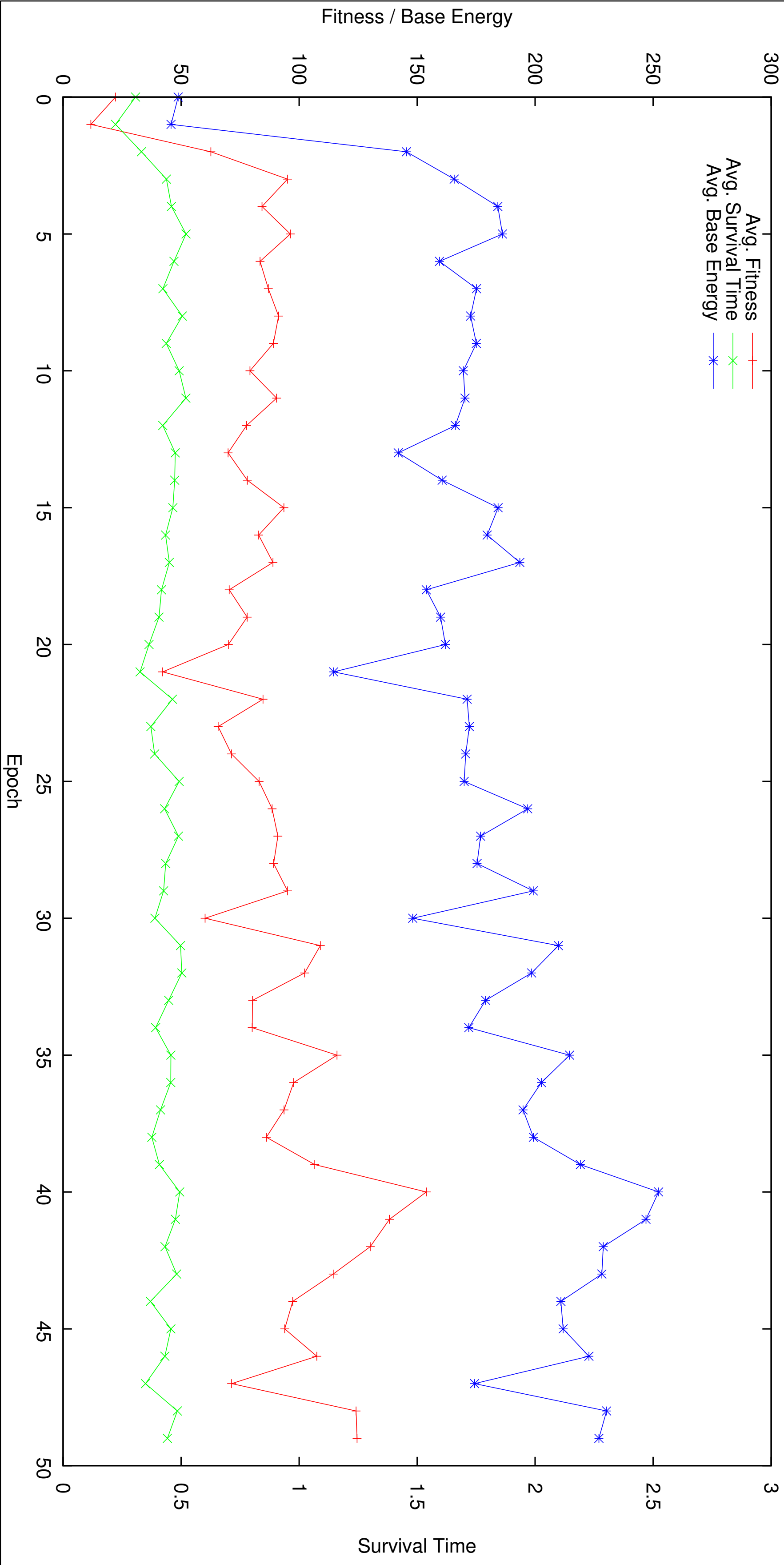


Figure 5.5 – The results for the first 50 epochs of experiment 2.

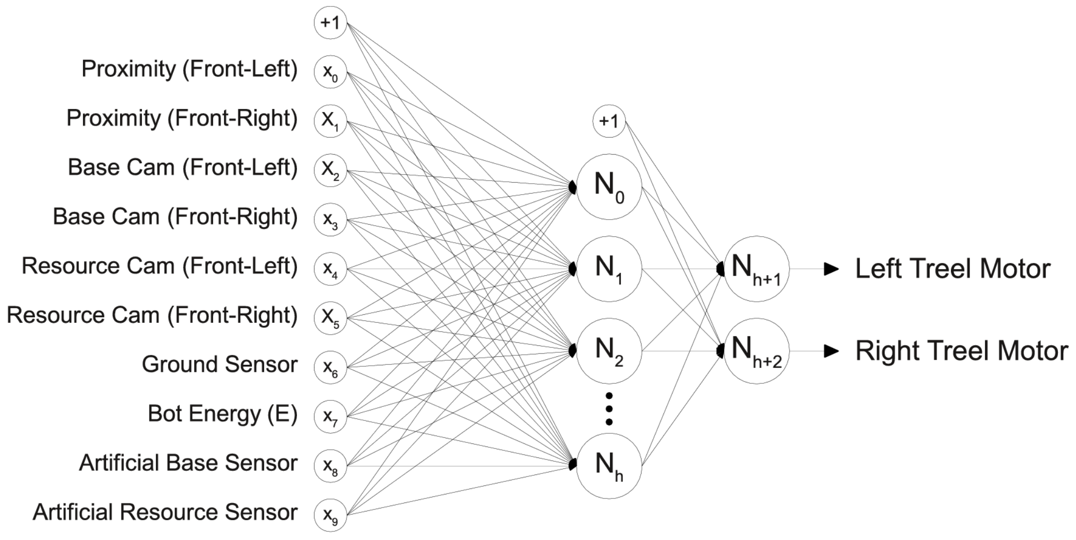


Figure 5.6 – The FannBot architecture.

5.2.3 Experiment 3 - Neuroevolution

In this experiment the controller is based on a three layer feedforward neural network, the weights of which are evolved using a genetic algorithm. This algorithm is referred to as neuroevolution, as introduced in Sec. 2.4.2. The controller is called *FannBot* to emphasize the fact that it is based on the *fann* (fast artificial neural network) library[2].

Controller Architecture

The neural network has 8 or 10 inputs (depending on the selected sensors) and two outputs, which directly control the speed of the robot's left and right treel. Between input and output layer is one hidden layer, the size of which can be selected in the ARGoS configuration file. Fig. 5.6 shows an overview of the neural network controller.

One of the most challenging tasks, when designing a neural network based controller, is the selection of required sensors and input preprocessing algorithms. Often, the number of available sensor readings is simply too large to be fed directly to the network. According to Eq. 2.22 it is required to limit the size of the input layer to keep the search space of the genetic algorithm small. The following paragraphs describe, which inputs have been selected and how their readings are fed to the network.

As the robot has to avoid collisions with obstacles, the controller needs access to the proximity sensors. The marXbot has 24 of these sensors which are arranged around the robot's chassis (see Sec. 3.1). As 24 input values is way too much for a simple neural network, the six sensors facing front-left and the six sensors facing front-right have been selected, thus reducing the number of inputs to 12. These two groups of inputs are then aggregated into two values (one for the front-left and one for the front-right sensor group). The values are -1 if no obstacles are within the sensing distance and progressively increase when an obstacle comes closer until they reach the maximum possible value of 1. The following listing shows how the calculation is performed. The `tProxReads` variable contains all of the 24 proximity sensor readings and the `arrfAnnInput` variable holds all of the neural network inputs.

[2]http://leenissen.dk/fann/wp/, accessed 18. Dec. 2011

```
Real arrfProximityInput[4] = { 0, 0, 0, 0 };
for (int i = 0; i < 24; i++) {
  arrfProximityInput[i / 6] += tProxReads[i].Value;
}
arrfAnnInput[0] = 2.0 * arrfProximityInput[0] / 6.0 - 1.0;
arrfAnnInput[1] = 2.0 * arrfProximityInput[3] / 6.0 - 1.0;
```

To allow the robot to perceive the value of a resource, it has to access the ground sensors. The marXbot has four of these sensors, which measure the brightness of the surface underneath the robot. As the sensors are arranged closely together, the readings only differ on the edges of resources. Therefore, it is appropriate to aggregate all four of the sensor readings into one, by calculating the average value and normalizing it to $[-1; 1]$. This is accomplished using Eq. 5.13 where $s_0, \ldots, s_3$ are the sensor readings and x is the value, which is fed to the ANN. The four sensor readings are summed up and then divided by 4 to get the average value. The result is multiplied by 2 and 1 is subtracted to match the required input range of $[-1; 1]$.

$$x = \frac{2}{4} \sum_{i=0}^{3} s_i - 1 \tag{5.13}$$

One of the most important values for the survival of the robot is its current energy level as the robot dies if this value reaches zero. Feeding this indication to the neural network is simple as it is a single scalar value. The energy level only has to be normalized to fit into the allowed input range of $[-1; 1]$. This is accomplished using Eq. 5.14.

$$x = 2\frac{E^{(i)}(t)}{E_{max}} - 1 \tag{5.14}$$

The most challenging sensor is the omnidirectional camera. Many different possibilities exist to feed camera images into a neural network. One possibility is to directly connect one of the ANN inputs to each of the camera pixels. As this requires a network of massive proportions, the resolution of the image is often reduced, before feeding it to the neural network. Using this technique Dom A. Pomerleau successfully trained a set of neural networks to drive a car, using a supervised learning approach [Pom89]. However, this approach has the disadvantage of requiring a huge amount of input neurons, which in turn dramatically increases the search space of the genetic algorithm. As the interface for the omnidirectional camera implemented in the ARGoS simulator only supports high level access to the camera readings, this approach can be ruled out. The camera interface already performs a lot of preprocessing, returning only recognized color blobs along with their distance and their angle. This means that the distance and angle of a resource or the base LED are directly returned. As each resource is marked with red and the base is marked with a green LED, these objects are easy to distinguish.

However, several possibilities remain to feed the obtained locations into the neural network. One possibility is to connect one input neuron to the angular and one to the distance value for each of the recognized objects. As the number of visible objects might change due to obstruction, there has to be a distinctive input value configuration, which allows the ANN to determine this condition. Therefore, an additional input neuron is required per object, which is -1 if no valid readings are available and $+1$ if the readings can be used (or the other way around). This increases the number of required inputs to three per visible object. As all of the experiments are configured using four resources

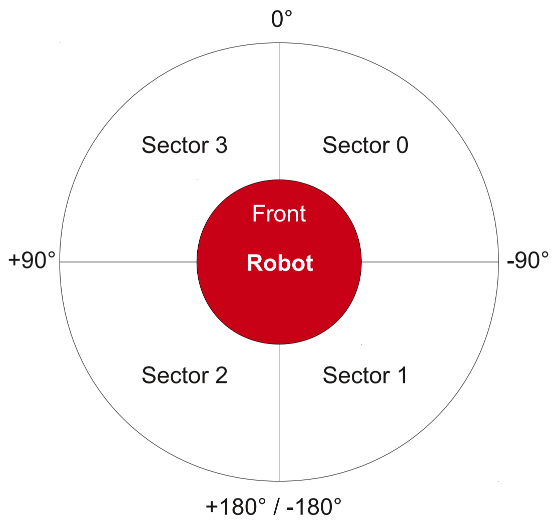


Figure 5.7 – The omnidirectional camera color blob readings are divided into four different sectors.

and one base marker LED the total number of required input neurons for visual data processing would be 15.

As this number is high, compared to other neural networks, a different approach was selected. In the neuroevolution foraging experiment of [WFK11] a very simplistic approach is chosen to allow the robots to perceive their environment using infrared distance sensors and a camera. The robots are equipped with four distance sensors, one of which is placed higher than the remaining three sensors. As robots are taller than the food items in the arena, the third sensor allows the ANN to distinguish between these object classes. The pixels of the camera are divided into a left and a right group leading to only two additional neural network inputs. These inputs allow the robots to recognize the base area, which is marked by a white wall. Inspired by these techniques, the color blob readings of marXbot's omnidirectional camera are divided into four sectors, as visualized in Fig. 5.7. The algorithm counts the number of color blobs for each object class (base or resource) and for each sector. The final counts are then divided by the total number of blobs encountered for each class, as defined in Eq. 5.15. The total number of visible blobs for an object class is denoted by n, the number of visible blobs for sector i are denoted as n_i and the resulting value for sector i is assigned to s_i.

$$s_i = \frac{n_i}{n} \tag{5.15}$$

The results for sectors 0 and 3 are then fed to the neural network, resulting in a total of two inputs for the resources and two for the base. The results for sectors 1 and 2 are ignored, because they relate to objects behind the robot. The limitation to objects in front of the robot takes into account that most biological individuals can only observe the environment in front of them.

The controller has two optional input values, which are neither based on internal state, nor on sensor readings. These values are obtained by accessing internal ARGoS and `EnvironmentGenerator` state. The values are optional because they can only be used within the simulator, but not in real robots. The first of these values is a simulated base sensor. If the robot is within the base area, this sensor emits 1, otherwise it

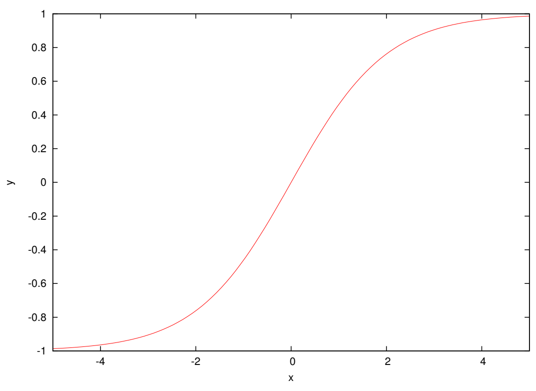


Figure 5.8 – Visualization of the activation function used in the neuroevolution experiment.

emits -1. The second sensor determines, whether the robot is currently on a resource. The sensor also emits 1 if the condition is fulfilled and -1 otherwise. Whether or not these two sensors are used can be selected using a switch in the ARGoS configuration file. The experiments have been conducted for both cases to be able to analyze the performance impact.

As mentioned above, the neural network of the controller is implemented using the *fann* library, which offers a wide variety of settings. The `FANN_SIGMOID_SYMMETRIC` activation function, which is defined as Eq. 5.16 is used for the experiments. The steepness is selected to be $s = 0.5$, which is the default value of fann. Fig. 5.8 shows a visualization of the activation function used in the experiments.

$$y = \tanh(s \cdot x) = \frac{2}{1 + \exp(-2 \cdot s \cdot x)} - 1 \tag{5.16}$$

The robots genome is mapped to the connection weights of the ANN in a layer-by-layer and neuron-by-neuron approach. The first element of the genome is mapped to the connection of the first input neuron (the bias neuron) to the first neuron of the second layer (N_0). The second element is mapped to the connection between the bias neuron and N_1. This scheme continues until all of the connections of the bias neuron are covered. Next, the connections of $x_0, x_1, \ldots, x_9$ are mapped, until the first layer of connections is complete. The second layer is mapped in a similar fashion. As an input of $+1$ to a neuron should lead to an output close to $+1$, the weights need to be in the range $[-4; 4]$. It is clearly visible in Fig. 5.8, that -4 results in an output close to -1 and $+4$ results in an output close to $+1$. As all elements of the genome are in the range $[0; 1]$, it has to be expanded, to fit the range $[-4; 4]$, which is accomplished using Eq. 5.17.

$$w_i = 8 \cdot g_i - 4 \tag{5.17}$$

The total number of connections N can be calculated using Eq. 5.18, where i is the number of inputs, h is the number of hidden and o is the number of output neurons. Due to bias neurons in each of the layers except the output layer, $+1$ has to be added to those layers.

$$N = (i + 1) \cdot h + (h + 1) \cdot o \tag{5.18}$$

Experiment Results

The experiment was executed for 300 epochs, once with artificial base and resource sensors and once without, as well as for different number of hidden neurons (h). The entire batch of experiments took about 8000 processor hours and about 100 wall clock hours to complete on the SuperMUC. Fig. 5.9 and 5.10 show the average fitness results for each epoch and for each configuration of the experiments. To give a better overview about the overall performance of each configuration, Tab. 5.7 contains the aggregated values for the last 50 generations of each configuration. The first column specifies, whether the artificial base and resource sensors are in use. The second column contains the number of hidden neurons h. The third column contains the average achieved fitness $\bar{F}$. The fourth column contains the average collected base energy $\bar{B}$ and the last column contains the average relative survival time $\bar{t_s}$.

The top performing robot, without artificial base and resource sensors, appeared in epoch 253, used $h = 7$ hidden neurons, achieved a fitness of 485.37, collected 537.00J of energy and survived for 90% of the simulation time ($t_s = 0.90$). The top performing robot with artificial base and resource sensors appeared in epoch 250, used $h = 10$ hidden neurons, achieved a fitness of 236.27, collected 266.75J of energy and survived for 89% of the simulation time ($t_s = 0.89$). The genomes of these top performing individuals can be found in App. D.

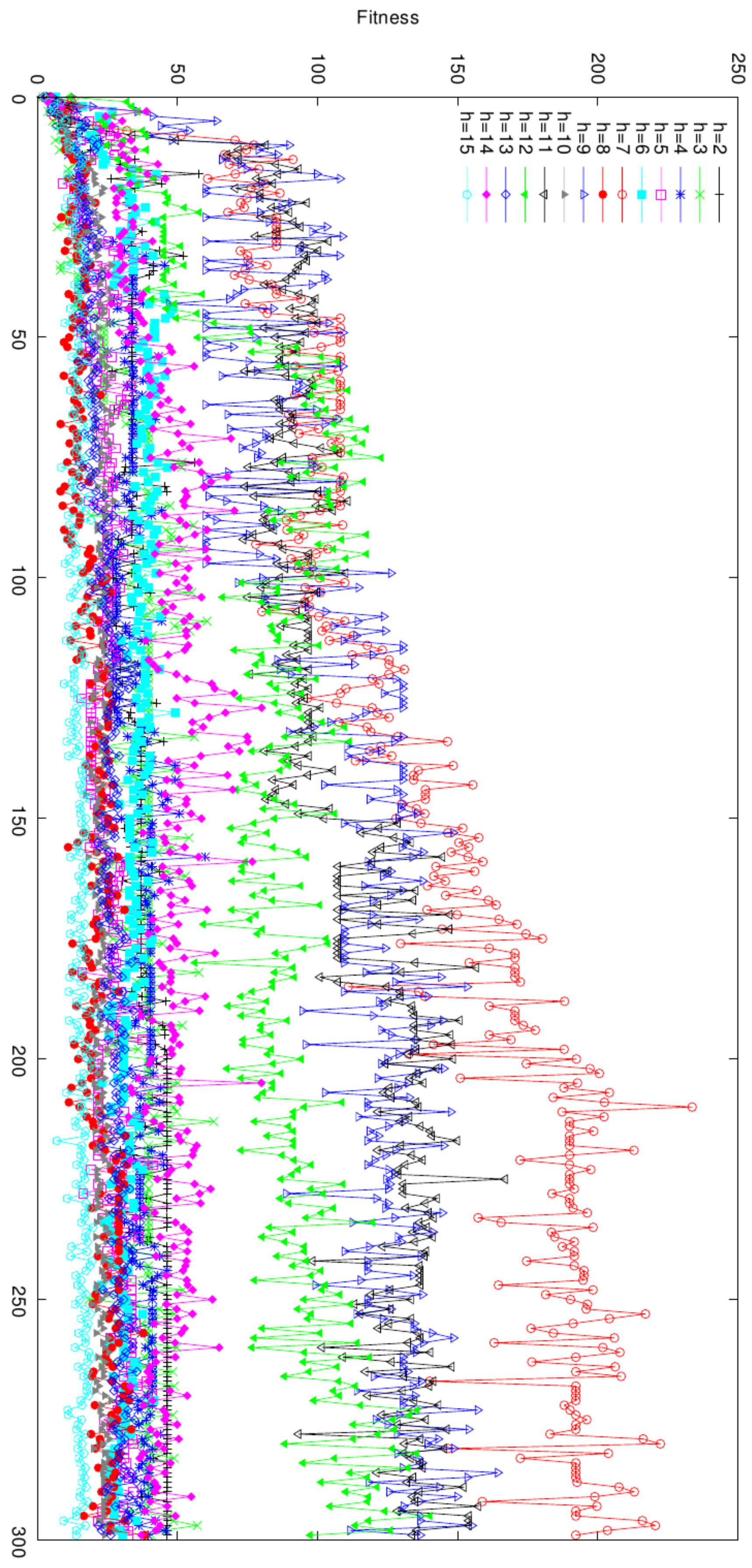


Figure 5.9 – Average fitness results for all epochs of experiment 3 without artificial resource and base sensor.

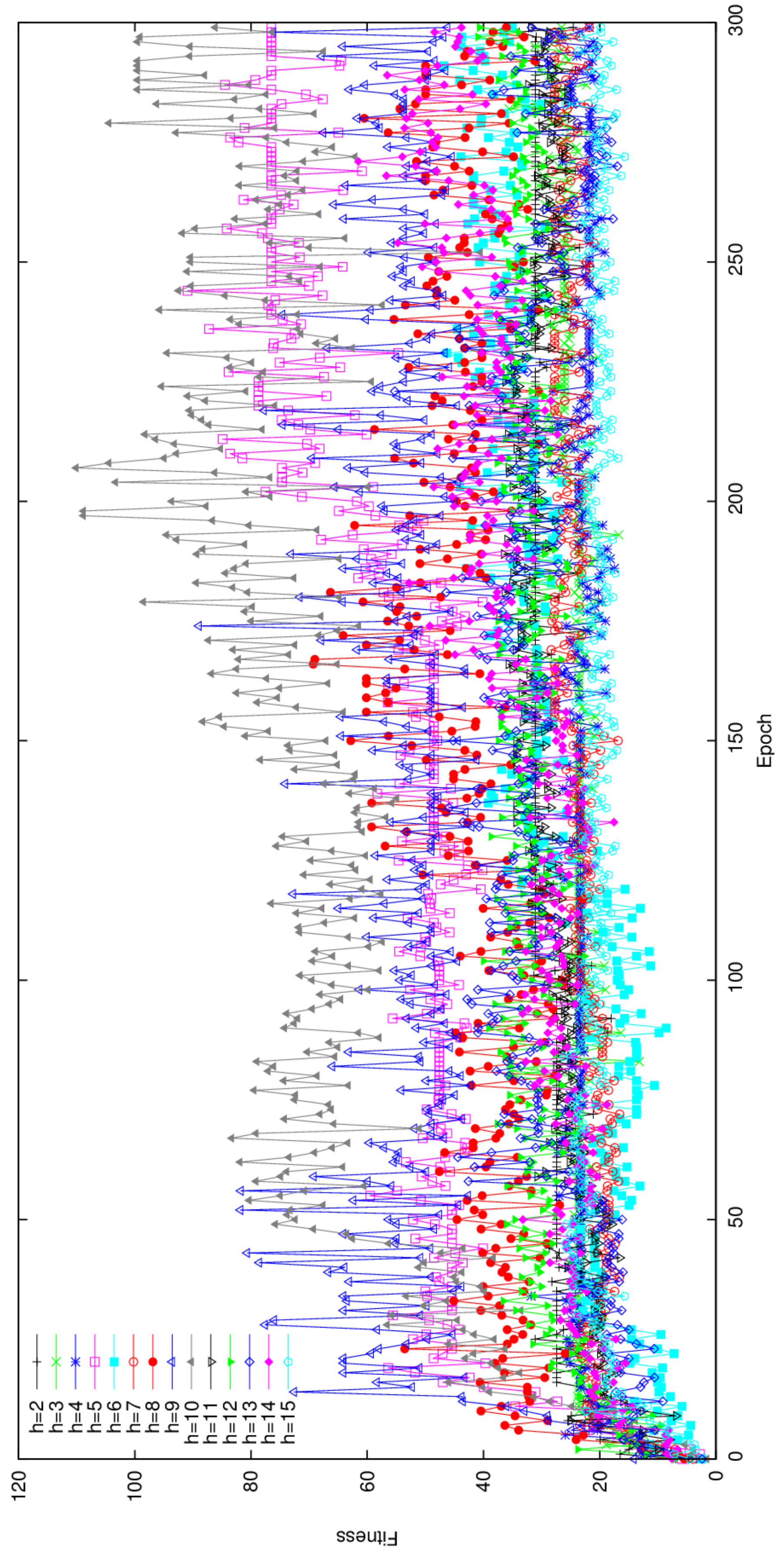


Figure 5.10 – Average fitness results for all epochs of experiment 3 with artificial resource and base sensor.

Additional sensors	**h**	$\bar{\mathbf{F}}$	$\bar{\mathbf{B}}$	$\bar{\mathbf{t_s}}$
no	2	46.15	88.78J	0.55
no	3	40.24	53.77J	0.69
no	4	38.67	131.08J	0.29
no	5	30.94	89.20J	0.34
no	6	29.94	133.67J	0.22
no	7	193.40	322.15J	0.59
no	8	26.23	74.68J	0.36
no	9	133.82	216.77J	0.60
no	10	23.41	96.57J	0.25
no	11	132.69	245.45J	0.53
no	12	109.64	198.87J	0.54
no	13	30.17	99.43J	0.31
no	14	44.99	105.75J	0.41
no	15	15.76	70.12J	0.21
yes	2	29.97	113.97J	0.27
yes	3	26.72	79.14J	0.35
yes	4	22.29	120.94J	0.19
yes	5	75.43	150.19J	0.51
yes	6	38.25	111.20J	0.35
yes	7	25.64	109.26J	0.24
yes	8	42.95	131.88J	0.32
yes	9	54.43	178.46J	0.29
yes	10	81.09	158.32J	0.50
yes	11	28.48	114.23J	0.25
yes	12	33.42	112.25J	0.31
yes	13	25.66	59.23J	0.44
yes	14	46.50	109.89J	0.43
yes	15	20.07	108.59J	0.19

Table 5.7 – The aggregated results for the last 50 epochs of experiment 3.

5.2.4 Experiment 4 - The UberController

The Uber Controller experiment is designed to show an upper bound for the achievable performance in the foraging experiment. It uses a special purpose controller based on a state machine, which is designed specifically for the experiment environment and the foraging task. The controller is named Uber Controller, because it is supposed to outperform all of the other controllers.

Controller Architecture

The controller uses a simple algorithm to collect as much energy as possible and to survive until the end of the simulation. After the simulation is initialized, the robot starts exploring its environment (`ExploringPhase`). First, all of the resources are detected. By using the omnidirectional camera and the robot's current location, it calculates the locations of all the resources and stores them in a special data structure. Next, the robot starts driving to the closest resource (`DrivingToResource`). Once the resource is reached, it measures its value, using its ground sensors. This process is repeated, until all of the resources have been visited. All of the results are stored into the data structure mentioned earlier. Once the exploring phase is complete, the robot switches to the `ForagingPhase`. First it determines, which of the resources is the most economical one. This is accomplished by estimating the energy, required for driving from the resource to the base and back and by taking into account the current harvesting rate of that resource. Once the resource has been selected, the robot starts harvesting. As soon as a different resource becomes more economical, the robot switches to that resource and starts harvesting it. Every time the robot drives over a resource, it updates its data structure, to have the most up-to-date values available for future decisions. When the robot reaches the base, it stops and waits until a certain amount of energy has been dropped of. The controller state model is presented in Fig. 5.11.

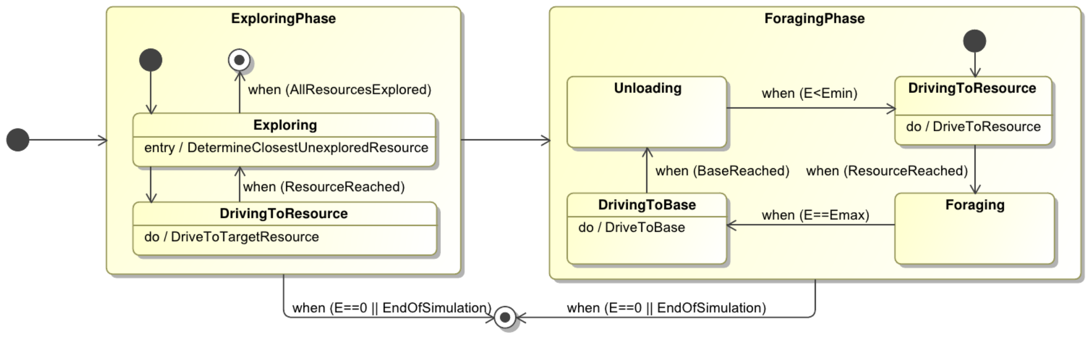


Figure 5.11 – The state machine of the *UberController.*

It would be possible for the robot to determine its current location by using the input from the omnidirectional camera. However, as this process is highly complex and beyond the scope of this thesis, the robot determines its current location directly from the internal ARGoS state.

To determine the adjusted relative value $R^{(j)}$ of a resource j, the controller takes into account the distance $b^{(j)}$ between resource and base, the arena length l_a, the driving velocity v, the journey time t_j required for driving from one end of the arena to the other and the distance cost factor c. The entire calculation used for this estimation is presented in Eq. 5.19.

$$R^{(j)} = \frac{V^{(j)}(t)}{V_0^{(j)}} - c \cdot \frac{b^{(j)}}{l_a} \cdot v \cdot t_j \tag{5.19}$$

As the robot needs to avoid collisions, it has a built in collision avoidance algorithm, based on the algorithm, used in the random walk controller. This algorithm has a higher priority than the state machine and takes over control whenever it determines that a collision is imminent. As soon as the collision has been successfully prevented, the algorithm turns back control to the state machine.

The controller has several parameters, which have to be adjusted carefully to achieve optimal performance. Tab. 5.9 gives an overview about all of the parameters and their selected values. The default values have been determined using knowledge about the experiment environment.

Experiment Results

The experiment was conducted four times, using different random seeds and using the default values of Tab. 5.9. The results are presented in Tab. 5.10. The first line contains

Parameter	Default	Description
E_{min}	0.75 E_{max}	The controller stops unloading energy in the base when this minimum energy level is reached.
c	0.1	The distance cost factor.
l_a	4m	The arena length.
V_{min}	0.1	The minimum relative resource value. The robot stops using a resource if its relative value $V^{(j)}$ falls below V_{min}.
α	7.5°	The go straight angle used for collision avoidance.
δ	0.1	The maximum collision vector length.
v	10	The relative wheel velocity.

Table 5.9 – The uber controller parameters and their default values.

the average, minimum and maximum fitness (F). The second line contains the average, minimum and maximum collected energy (B in joules) and the last line contains the average, minimum and maximum relative survival time (t_s).

Parameter	Average	Minimum	Maximum
F	531.28	190.19	826.75
B/J	656.65	55.74	826.75
t_s	0.78	0.44	1.00

Table 5.10 – The results of the uber controller experiment.

5.2.5 Experiment 5 - Genetically Optimizing the Uber Controller

As mentioned in the previous section, the `UberController` has several parameters, which determine its performance. Although the values have been selected carefully, it is interesting to see whether a genetic algorithm would end up with different values.

Experiment Details

Just like in Sec. 5.2.2, the parameters of the `UberController` are now specified using a genome. The genetic algorithm used to optimize the parameters is equal to the algorithm used in previous experiments. As the a_l parameter is fixed, the genome has a total length of 6. The mutation probability is selected to be 0.05, which leads to an average of $10 \cdot 6 \cdot 0.05 = 3$ mutations per epoch, which is equal to the probability, used in experiments 2 and 3.

Experiment Results

The results of this experiment are presented in Fig. 5.12. A steep rise of the average, as well as the maximum of the fitness (F) and the collected energy (B) can be identified for the first few epochs. The relative survival time (t_s) does not show this behavior, it even drops within the first few epochs and only slowly increases later on. The maximum survival time converges towards 100%, with several deviations in between. These deviations, as well as the deviations for fitness and base energy, are caused by mutations

Parameter	Experiment 4	Experiment 5
$\mathbf{E_{min}}$	$0.75 \cdot E_{max}$	$0.420 \cdot E_{max}$
$\mathbf{c}$	0.1	0.00194
$\mathbf{V_{min}}$	0.1	0.00229
α	7.5°	2.78°
δ	0.1	0.0270
$\mathbf{v}$	10	15.8

Table 5.11 – The parameters of the top performing controller of experiment 5 compared to the manually selected parameters of experiment 4.

which, in most cases, have a negative effect on performance. On average, the robots achieved a fitness (F) of 719.05, a base energy (B) of 830.46J and a relative survival time (t_s) of 84% within the last 50 epochs of the experiment. The top performing genome achieved a fitness of 1577.25, a base energy of 1577.25J and a relative survival time of 100%. This genome appeared in epoch 114. This top performin genome is presented in the following listing:

```
[0.419540345668793, 0.387048065662384, 0.00229340791702271,
    0.884248495101929, 0.0269693732261658, 0.79144412279129,
    0.330124855041504]
```

These values translate to the controller parameters presented in Tab. 5.11, along with the values of experiment 4.

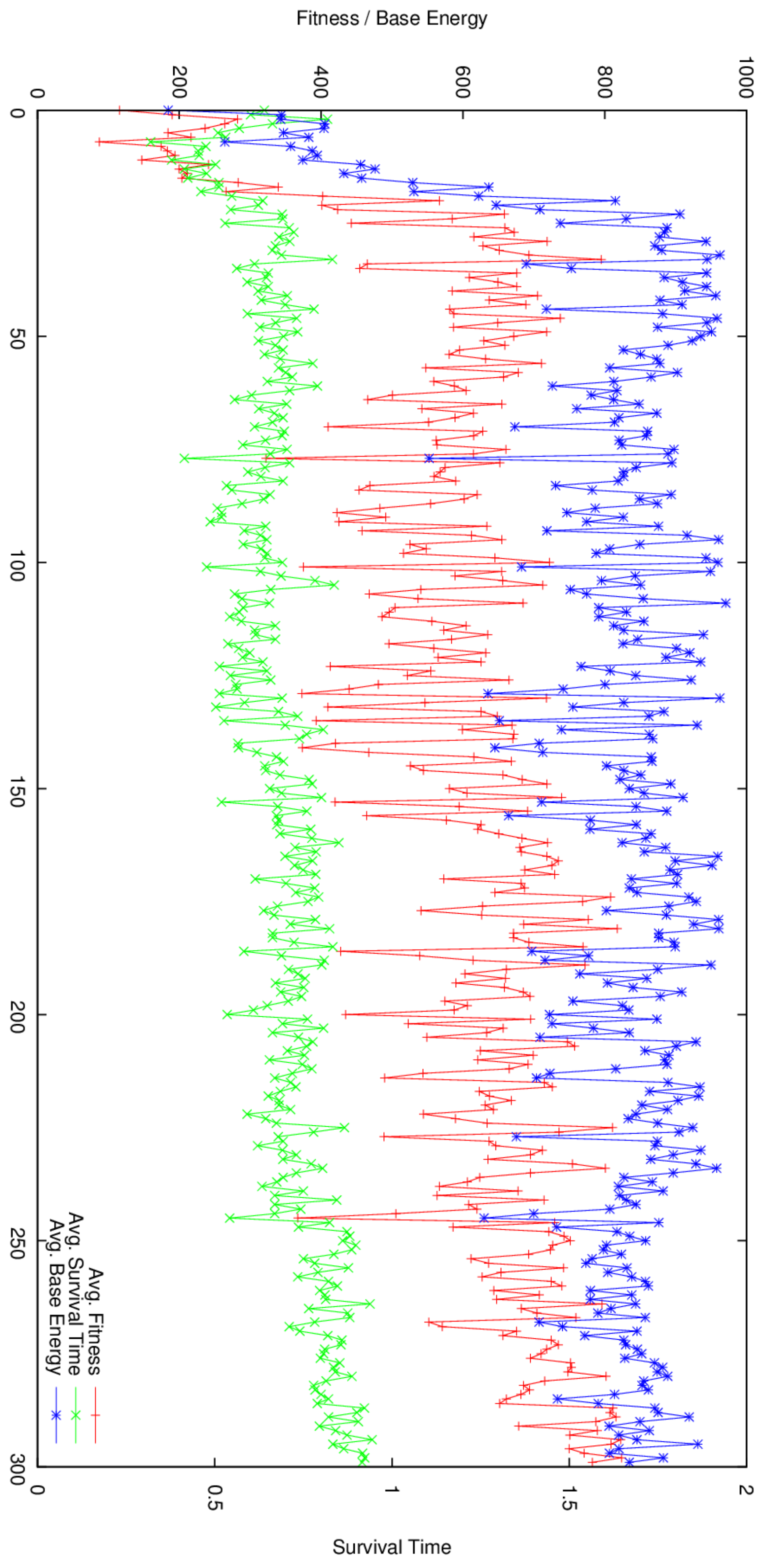


Figure 5.12 – The results of the genetic uber controller experiment.

Chapter 6

Interpretation of Results and Conclusion

The objectives of this chapter are to compare the results of the previous chapter with each other and to discuss the results. After that, a conclusion is drawn and recommendations are presented. The last part of this chapter gives an outlook for potential future work.

6.1 Algorithm Performance Comparison

The results of the individual experiments of the previous chapter are now compared with each other. As the fitness F characterizes the overall performance of a controller, only this value is taken into account. Comparing the collected energy B and the relative survival time t_s would be interesting, but would not add any value to the results. To create a meaningful comparison, the average ($\bar{F}$) as well as the maximum ($F^{(i_{top})}$) fitness are taken into account. For non-genetic-algorithm experiments, these values are simply the average and the maximum of all measured fitness values. Genetic algorithms lead to better performance over time. Therefore, the last 50 epochs of experiments, using this technology, are taken into account. To derive meaningful results, the GAs should have reached a performance plateau before epoch 250, which can be confirmed using Fig. 5.4, 5.9, 5.10 and 5.12. The average values of experiments using GAs contain all the performance values of all individuals of the epochs 250 through 299. The maximum fitness of these experiments is obtained using the top performing individual within all of the epochs. As experiment 3 encompasses many different configurations, the results are divided into configurations with and without artificial base and resource sensor. The top performing configurations (i.e. number of hidden neurons h) are explicitly mentioned. The results are presented in Tab. 6.1.

As mentioned in Sec. 5.2, the intention behind experiments 1 and 2 is to get a lower bound for the achievable fitness. The intention of experiments 4 and 5, on the other hand, is to get an upper bound for the achievable fitness. The values presented in Tab. 6.1 show, that these goals are fulfilled. On average, the `UberController` performed 455% better than the controller of experiment 1. The result is similar for the genetically enhanced version of these controllers, where, on average, the `UberController` performed 711% better. If comparing the maximum fitness, which was achieved by one of the individuals, during the evolution, the `UberController` performed 562% better than the `RandomWalk` controller.

Ex.	**Algorithm**	$\bar{\mathbf{F}}$	$\mathbf{F}^{(\mathbf{i}_{\mathbf{top}})}$
1	Random Walk	116.71	206.65
2	Random Walk GA	101.16	280.77
3	Neuroevolution w/o add. sensors	64.00	485.37
3	$h = 7$	193.40	485.37
3	Neuroevolution w/ add. sensors	39.35	236.27
3	$h = 10$	81.09	236.27
4	UberController	531.28	826.75
5	UberController GA	719.05	1577.25

Table 6.1 – Comparison of the algorithm performance of all experiments.

Experiments 2 and 5 successfully proof that the genetic algorithm works. The maximum performance of the genetically optimized version of the `RandomWalk` controller is 136% of the maximum performance of its original version. However, the average performance is only 87% of its unoptimized pendant. This unexpected result shows one of the drawbacks of genetic algorithms. The results cannot always be predicted and are sometimes surprising. In this special case, the high mutation probabilities of 0.1 (experiment 2) and 0.05 (experiment 5), which lead to 3 mutations per epoch, is most likely responsible for the low average results. As mutations have, almost always, negative performance impact on the robots, they lower the average performance. The genetically improved `UberController`, on average, achieved 135% of the performance of its unoptimized pendant. The maximum performance is even 191% of the maximum performance of the unoptimized version.

The `FannBot`, which is evaluated in experiment 3, on average, achieved a rather poor performance. The average performance of the `FannBot` is even lower than the average performance of the `RandomWalk` controller. The version without artificial base and resource sensors, on average, achieved only 55% of the performance of the controller in experiment 1. The `FannBot` with artificial base and resource sensors achieved even lower performance, reaching only 34% of the reference performance of experiment 1. However, experiment 3 evaluated many different configurations of the same controller. As presented in Sec. 5.2.3, the top performing configuration used no artificial base and resource sensors and used $h = 7$ hidden neurons. This configuration on average achieved a fitness of 193.40, which is 166% of the reference performance. The top performing genome with this configuration achieved a fitness of 485.37 which is 235% of the reference performance. Therefore, the neuroevolution controller can do significantly better as the `RandomWalk` controller, if the conditions are optimal. Surprisingly, the controllers with artificial base and resource sensors performed significantly worse, than the controller without these sensors.

6.2 Conclusion

It takes a lot of computing time to get acceptable results from genetic algorithms. As mentioned in Sec. 5.2.3, experiment 3 took about 8000 processor hours and about 100 wall clock hours to complete. As this kind of computational power often is not available, this is definitely something to keep in mind when choosing an algorithm. Overall it is hard and costly to use artificial neural networks, as many configurations need to be evaluated until an acceptable solution is found. Above all, it is hard to understand why certain configurations perform better than others. As feed-forward artificial neural networks do not have any internal state there computational capabilities are very limited, which might render them inferior to competing approaches.

However, neuroevolution is a completely generic approach, that can be applied without any detailed knowledge about the environment. Additionally, this class of algorithm can adapt to any (unforeseen) changes in the environment. When designing special purpose controllers, detailed information about the problem and the environment is required to achieve acceptable results. Furthermore, it is hard or even impossible to design these controllers in a way that makes them adaptable to any unforeseen changes within the environment.

Thus, neuroevolution is the algorithm of choice if knowledge about a problem is very limited or if the problem might change in an unpredictable way in the future. If, however, the environment and the problem are well understood, writing an algorithm from scratch is the best choice. As experiment 5 proves it is always worth a try to optimize the parameters of a special purpose controller using a genetic algorithm. Tab. 6.3 shows an overview of advantages and disadvantages of the two approaches.

Approach	**Advantages**	**Disadvantages**
Neuroevolution	• No detailed knowledge about the problem necessary. • Agents adapt to environmental changes.	• Unpredictable and abstruse results. • Computation intensive simulations.
Special purpose	• Usually higher performance. • Results are predictable and comprehensible.	• Detailed problem knowledge required. • Usually unable to adapt to changes in the environment.

Table 6.3 – Advantages and disadvantages of neuroevolution and special purpose controllers.

6.3 BRAIn Review

Thousands of experiment epochs have been executed in the course of this thesis. The BRAIn framework allowed to accomplish this in a fully automatic manner for local test runs, as well as for large scale simulations on the SuperMUC petascale system. By utilizing BRAIn's modular architecture, it was possible to implement all of the experiments in a simple and reusable way. The flexible configuration system allowed to specify many different experiments within the same batch configuration and to execute all of them in parallel, utilizing all of the available cores on SuperMUC. The data collection and reporting features built into BRAIn helped to analyze and document tens of thousands of collected measurements. A task, that would have been tedious without a proper framework.

6.4 Outlook

The algorithms presented in this thesis are only a small subset of the available algorithms. One particularly interesting algorithm, which is not covered in this thesis is *NeuroEvolution of Augmenting Topologies* (NEAT), which was developed by Stanley et al. at the University of Texas [SM02]. Unlike most neuroevolution algorithms, including the one used in this thesis, NEAT optimizes both the connection weights and the structure of an artificial neural network. The algorithm starts with only one layer of neurons, that are connected to all inputs, and then gradually adds new neurons and connections. This approach has the advantage, that no topology has to be designed in advance. Additionally, NEAT allows recurrent connections, eliminating one of the biggest restrictions of the algorithm used in this thesis. Therefore, analyzing NEAT, and other neuroevolution algorithms, using the methodology of this thesis is a potential topic of future research in this area.

The previous sections show that it is still very challenging to use artificial neural networks in a satisfying way. To avoid these drawbacks, future research should systematically analyze the characteristics of the neural networks used in this thesis. The goal is to provide easy guidelines for designing neural networks or to conceive an algorithm, which selects the best topology in a more efficient way.

Last but not least, future research in this area could encompass extending the BRAIn framework by the functionality presented in Sec. 4.4 or by other features, required to support further experiments or simulators.

Appendix A

ARGoS Configuration Reference

This chapter is intended to give a brief and by no means complete overview of ARGoS configuration files. The knowledge was obtained by experimentation and by reading the ARGoS source code, as no complete manual is available yet. The following listing shows a typical configuration file.

```xml
<?xml version="1.0" ?>
<argos-configuration>
  <framework>
    <system threads="2" />
    <experiment length="100000" ticks_per_second="10" random_seed="42"
       />
    <profiling file="some/path" format="human_readable" truncate_file="
       true" />
  </framework>

  <controllers>
    <my_controller id="foac" library="path/to/my-controller.so">
      <actuators>
        <footbot_wheels implementation="default" />
        <footbot_leds implementation="default" />
        <footbot_beacon implementation="default" />
      </actuators>
      <sensors>
        <footbot_proximity implementation="rot_z_only" show_rays="true"
           calibrate="true" />
        <footbot_motor_ground implementation="rot_z_only" calibrate="
           true" />
        <footbot_omnidirectional_camera implementation="rot_z_only"
           aperture="89" show_rays="true" />
      </sensors>
      <parameters param1="23" param2="42"/>
    </my_controller>
  </controllers>

  <loop_functions library="path/to/my-loop-function.so" label="my-loop-
     function" param1="foo" param2="bar" />

  <arena size="4, 4, 1">
    <floor id="floor" source="loop_functions" pixels_per_meter="100" />

    <light id="light0" position="0,0,0" orientation="0,0,0" color="
       yellow" intensity="1.0" />

    <box id="wall_north" position="0,2,0.25" orientation="0,0,0" size="
```

```
        4,0.1,0.5" movable="false" />
    <box id="wall_south" position="0,-2,0.25" orientation="0,0,0" size="
        4,0.1,0.5" movable="false" />
    <box id="wall_east" position="2,0,0.25" orientation="0,0,0" size="
        0.1,4,0.5" movable="false" />
    <box id="wall_west" position="-2,0,0.25" orientation="0,0,0" size="
        0.1,4,0.5" movable="false" />

    <distribute>
      <position method="uniform" min="-4,-4,0" max="4,4,0" />
      <orientation method="gaussian" mean="0,0,0" std_dev="360,0,0" />
      <entity quantity="10" max_trials="100">
        <foot-bot id="fb" controller="foac" />
      </entity>
    </distribute>
  </arena>

  <physics_engines>
    <dynamics2d id="dyn2d" />
  </physics_engines>

  <arena_physics>
    <engine id="dyn2d">
      <entity id="fb_[[:digit:]]*" />
      <entity id="wall_north" />
      <entity id="wall_south" />
      <entity id="wall_east" />
      <entity id="wall_west" />
    </engine>
  </arena_physics>

  <visualization>
    <qtopengl_render splash="false">
      <camera>
        <placement idx="0" position="0,0,5" look_at="0,0,0"
            lens_focal_length="20" />
        <placement idx="1" position="-3,0,2" look_at="0,0,0"
            lens_focal_length="20" />
      </camera>
      <user_functions library="path/to/my-qt-user-function.so" />
    </qtopengl_render>
  </visualization>
</argos-configuration>
```

The entire file is XML based, but no schema is defined. This provides the possibility to add tags and parameters at any location as long as the XML syntax is not violated. This concept increases flexibility, as custom configuration nodes can be added for user defined modules, without having to worry about schema definitions.

The entire file is enclosed by `<argos-configuration>` tags, which contain all of the system and module specific configuration tags. The following table describes most of the possible configuration options in a systematic manner. However, some tags are too complex to describe in this tabular format and therefore are described later on in more detail. The first column of the table contains the name of the configuration elements, which can be tags and parameters. The element hierarchy is reflected by indentation. The type of an element is specified in the second column. The type *tag* means, that the element specified in this line is an XML tag. All the other types mean that the element is a parameter in side of a tag. The remaining types should be familiar from *C++*.

Parameter	Type	Required	Default	Description
framework	tag	yes	-	Contains global configuration options.
system	tag	no	-	Only used for multithreading configuration.
threads	int	no	0	The number of slave threads.
experiment	tag	yes	-	Various experiment parameters.
length	int	no	0	The experiment length in seconds. If 0, the experiment runs forever.
ticks_per_second	int	yes	-	Number of simulation steps per simulated second.
random_seed	int	yes	-	Used to initialize ARGoS' pseudo random number generator. If unspecified, the seed is deduced from the current time.
profiling	tag	no	-	Can be used to collect runtime performance statistics.
file	string	yes	-	The file to write the performance data to.
format	enum	yes	-	`table` or `human_readable`.
truncate_file	bool	no	true	True means that the specified file is cleared, before writing any new data to it.
controllers	tag	yes	-	Contains the robot controller configurations.
loop_functions	tag	yes	-	Specifies the loop functions to use during the simulation.
library	string	yes	-	The shared object file, containing the loop functions.
label	string	yes	-	The name of the loop function, as defined in the shared object file.
arena	tag	yes	-	Defines the arena for the simulation.
size	int[3]	yes	-	The width, length and height of the arena.
hashing	enum	no	`tr1`	Configures space hashing. Possible values are `off`, `tr1` and `native`.
distribute	tag	no	-	Can be used to place objects ramdomly in the arena.
position	tag	yes	-	Specifies where and how to distribute entities.
orientation	tag	yes	-	Specifies how distributed objects are oriented.
entity	tag	yes	-	Specifies the entities to place.

quantity	int	yes	-	Number of entities to created.
max_trials	int	yes	-	Number of attempts to place the items.
base_num	int	no	0	The id of the first entity.
physics_engines	tag	no	-	Contains a list of physics engines.
arena_physics	tag	yes	-	Specifies which entity belongs to which physics engine.
visualization	tag	yes	-	Contains a list of visiualizations.

Specifying Controllers

The `<controllers>` tag can contain multiple child tags, each specifying a controller. Beware that the name of the child tag has to match the controller name, specified in the controller sourcecode. A controller tag contains exactly three sub tags. The `<actuators>` tag has to list all actuators, which are used by the controller. The `<sensors>` tag contains all the sensors, required for the controller to work. Last but not least, the `<parameters>` tag can contain arbitrary configuration options, required by the controller. It is possible to specify properties as well as child tags.

For a list of supported sensors and actuators refer to the `simulator/sensors` and `simulator/actuators` folders within the ARGoS source code. The `*.cpp` files of the sensor and actuator implementations, each contain a brief documentation explaining usage and configuration options at their bottoms.

Adding Entities

Arbitrary static and dynamic entities can be added in the `<arena>` tag. To add an entity, simply specify its specific tag, along with its required and optional parameters, where applicable. In the example configuration file of the appendix, for instance, a floor, a light and multiple box entities are added to the arena. Each entity needs to have a unique ID, which is an arbitrary string.

For a list of supported entities and their parameters, refer to the `simulator/-space/entities` sub-folder of the ARGoS source archive. Each entity `*.cpp` file contains a brief documentation about its usage, along with its unique name, just like it is the case for sensors and actuators.

Distributing Entities

You can either specify the position and orientation of entities manually, as it is the case for the `light` and `box` entities in the example, or you can ask ARGoS to distribute them automatically. In the example configuration, the robots are placed using a distribution. The `<position>` and `<orientation>` specify how to determine these properties. The following distribution types are available.

uniform Uses a uniform distribution and expects a `min` and a `max` parameter specifying the boundaries of a cuboid, which is filled uniformly with the specified entities.

gaussian Uses a gaussian distribution with the parameters `mean` and `std_dev` (standard deviation).

grid Distributes the entities on a 3D grid and expects a *center* (a position in space), a *distances* (the x, y and z distances of the grid) and a *layout* (specifies the number of layers in x, y and z direction) parameter.

The `<entity>` tag specifies the entities to place. Its `quantity` parameter tells ARGoS how many entities of the specified type to add to the arena. ARGoS tries placing the entities randomly, but aborts as soon as two or more entities collide. To avoid infinite loops, the `max_trials` parameter specifies when to abort the distribution process.

The tags inside `<entity>` each specify a type of entity, which has to be known to ARGoS. As mentioned earlier, all of the entities can be found in `simulator/space/-entities`. Refer to the entity documentation within its source file for a description of optional and required parameters.

Using Visualizations

ARGoS supports multiple visualization engines, which are defined in the `simulator/-visualizations` source folder. Just like for any other type of modular component, a brief documentation, as well as the unique name, can be found within the module's source file. To enable a specific visualization one has to add a child tag with the visualization's unique name to the `<visualization>` tag.

Probably the most prominent visualization for day to day usage is the `qtopengl_-render`. It is possible to specify up to ten cameras, which can be selected conveniently from within the Qt4 user interface.

Cameras can be added by specifying `<placement>` tags within the `<camera>` child tag. Each `<placement>` tag expects an `idx` parameter, which assigns the camera to a button in the user interface. The `position` and `look_at` parameters specify the location and orientation of the camera. User function (see Sec. 3.2.2) can be specified by adding a `<user_functions>` tag, which expects a `library` parameter, that points to the shared object file, containing the user function and a `label` parameter, identifying the user function within the library.

To run batch experiments, visualizations can be disabled entirely by removing the content of the `<visualiation>` tag. Beware that the `<visualization>` tag has to be present, though.

Appendix B

Working with the Source Code

This chapter explains how to use the source code, which was developed for this thesis. The first part deals with the directory layout, while the second part deals with the steps, which are necessary to get started.

B.1 Directory Layout

To get started quickly, it is important to understand the directory layout of the source code. The following is a shallow overview of the `src` folder contents.

- `argos-modules`
- `brain`
- `distribution`
- `distribution.tar.gz`
- `experiments`
- `make-distribution.sh`
- `stuff`

The two most important source directories are `argos-modules` and `brain`. The former directory contains all of the ARGoS extensions, like the `environment_generator` and all of the robot controllers. The latter directory obviously contains the BRAIn framework. The sub folders of which are reflecting its architecture, detailed in Sec. 4. The `experiments` folder contains all the BRAIn experiment configuration files, along with their ARGoS configuration templates. Last but not least, the `stuff` directory contains external projects like the ARGoS simulator and example code.

To build the entire project, one can use the `make-distribution.sh` script, which creates a folder, called `distribution`. The script also bundles this directory into the `distribution.tar.gz` file. The folder and the archive contain everything necessary to run experiments.

B.2 Preparing the System

All of the code (including ARGoS) has been developed and tested under Ubuntu 10.10 and 11.04 but it should work equally well with other POSIX based operating systems, provided that all the required libraries are available.

On Ubuntu, the following command has to be executed, to install all of the requirements. This has been tested with Ubuntu 11.04:

```
$ sudo apt-get install subversion build-essential maven2 \
  sun-java6-jdk libgoogle-perftools-dev protobuf-compiler \
  cmake pkg-config libgsl0-dev freeglut3 libaudio2 \
  libflac8 libgl1-mesa-glx libglu1-mesa libogg0 libpulse0 \
  libqt4-dbus libqt4-opengl libqt4-xml libqtcore4 \
  libqtgui4 libsdl1.2debian-pulseaudio libsndfile1 \
  libvorbis0a libvorbisenc2 libx11-xcb1 libxcb-atom1 \
  libxdamage1 libxfixes3 libxmu6 libxxf86vm1 libjpeg62 \
  libmng1 libtiff4 fontconfig liblcms1 libprotobuf-dev \
  libfann-dev libcppunit-dev
```

To use Google Protocol Buffers with Maven, the appropriate Maven plugin has to be installed. Unfortunately, the plugin does not seem to be actively maintained and does not even work without patching it[1]. To install it, the source code has to be checked out from the repository:

```
$ svn checkout http://protobuf.googlecode.com/svn/branches/maven-plugin
```

Now open the file `AbstractProtocMojo.java`, and look for the following line, which should be around line 213:

```
$ protoDirectories.add(uncompressedCopy);
```

Once located, change this line to:

```
$ protoDirectories.add(uncompressedCopy.getParentFile());
```

Now the plugin can be installed using the following command:

```
$ cd maven-plugin/tools/maven-plugin
$ mvn install
```

The system is now ready to build the project source code. Now, ARGoS can be built and installed:

```
$ cd da/src/stuff/argos2
$ ./build.sh
$ ./make_distribution.sh debian
$ sudo apt-get install argos2-xxx.deb
```

Last but not least, the project source code can be built by going back to the `src` directory and executing:

```
$ ./make-distribution.sh
```

This script not only takes care of building the BRAIn and ARGoS modules, but also creates a distributable archive, ready to be deployed on a remote machine to run experiments.

[1]http://code.google.com/p/protobuf/issues/detail?id=138

Using Eclipse

Maven comes with excellent Eclipse support. To import all of BRAIn's projects into Eclipse, one first has to create the project files, required by the IDE. This can be accomplished by switching to the BRAIn root source folder and executing the following command:

```
$ mvn eclipse:eclipse
```

After the command returns, it will have created `.project` and `.classpath` files for each of the sub-projects. Now go to Eclipse and select `File` → `Import` from the menu. Select `Existing projects into Workspace` from the `General` section and click `Next` in the appearing dialog. In the next page, click on the `Browse` button next to the `Select root directory` field and select the root source folder. All of the projects should now appear in the `Projects` section of the dialog. If everything looks right, click on `Finish` to import the projects into Eclipse.

Every time one of the `pom.xml` files is modified, the `mvn eclipse:eclipse` command has to be invoked again to update the project files and all of the projects have to be refreshed in Eclipse.

The Eclipse project files can be removed by invoking the following command:

```
$ mvn eclipse:clean
```

Appendix C

Perceptron Learning Script

```
#!/usr/bin/ruby

def phi(v)
  if(v >= 0)
    return 1
  else
    return 0
  end
end

$w = [0, 0, 0]

def perceptron(x1, x2)
  phi($w[0] + $w[1] * x1 + $w[2] * x2)
end

$output = ""
$latex_output = ""

def learn(training_data, title, iterations, alpha)
  $output += "\n=== " + title + " ===\n\n"
  $latex_output += "\n=== " + title + " ===\n\n"

  iterations.times do |n|
    data = training_data[n % training_data.length]

    x1 = data[0]
    x2 = data[1]
    d = data[2]

    y = perceptron(x1, x2)
    e = d - y

    $output += sprintf("n=%2d, w=[%5.2f, %5.2f, %5.2f]" +
      ", d=%5.2f, y=%5.2f, e=%5.2f\n", n, $w[0],
      $w[1], $w[2], d, y, e)

    $latex_output += sprintf("%2d & %5.2f & %5.2f & " +
      "%5.2f & %5.2f & %5.2f & %5.2f \\\\\n", n,
      $w[0], $w[1], $w[2], d, y, e)

    $w[0] += alpha * e
    $w[1] += alpha * e * x1
    $w[2] += alpha * e * x2
  end
end

learn([[0, 0, 0], [0, 1, 0], [1, 0, 0], [1, 1, 1]], "AND", 30, 0.1)
learn([[0, 0, 0], [0, 1, 0], [1, 0, 0], [1, 1, 1]], "AND", 30, 5)
learn([[0, 0, 0], [0, 1, 1], [1, 0, 1], [1, 1, 1]], "OR ", 30, 0.1)
learn([[0, 0, 0], [0, 1, 1], [1, 0, 1], [1, 1, 1]], "OR ", 30, 5)

print $output
print $latex_output
```

Appendix D

Top Performing Genomes of Experiment 3

The top performing genome of experiment 3 without artificial base and resource sensors. This data can be found on the attached DVD in the file `experiment-10/epoch-253/-controller-config_8`.

```
[0.189427077770233, 0.00303781032562256, 0.17195051908493,
    0.124108552932739, 0.902129530906677, 0.575103044509888,
    0.959746181964874, 0.371288180351257, 0.925398886203766,
    0.874302327632904, 0.918968260288239, 0.469708621501923,
    0.910488247871399, 0.997938692569733, 0.570859909057617,
    0.523582935333252, 0.778995335102081, 0.861459016799927,
    0.404265224933624, 0.970952332019806, 0.407971262931824,
    0.905794262886047, 0.715595841407776, 0.230603158473969,
    0.149160921573639, 0.435250818729401, 0.110326707363129,
    0.254726767539978, 0.731291055679321, 0.460285186767578,
    0.638931572437286, 0.892984807491302, 0.178224623203278,
    0.189116060733795, 0.399775922298431, 0.365486621856689,
    0.103869140148163, 0.629932284355164, 0.0492684841156006,
    0.714975595474243, 0.904214859008789, 0.0658162832260132,
    0.237806379795074, 0.480188846588135, 0.869837284088135,
    0.429013729095459, 0.967480659484863, 0.446723639965057,
    0.457476556301117, 0.228854417800903, 0.0640694499015808,
    0.766560316085815, 0.813131093978882, 0.676378130912781,
    0.347673237323761, 0.232722043991089, 0.00941330194473267,
    0.965183258056641, 0.307389736175537, 0.15833204984664
9,
    0.0936659574508667, 0.303705155849457, 0.396176755428314,
    0.375425815582275, 0.38825798034668, 0.522453248500824,
    0.73470801115036, 0.115208268165588, 0.439407587051392,
    0.0629608035087585, 0.34333735704422, 0.775610387325287,
    0.0498694181442261, 0.996167242527008, 0.99142187833786,
    0.080797553062439, 0.436462938785553, 0.0720661282539368,
    0.787326633930206]
```

The top performing genome of experiment 3 with artificial base and resource sensors. This data can be found on the attached DVD in the file `experiment-17/epoch-250/-controller-config_2`.

```
[0.101841270923615, 0.542972326278687, 0.953332781791687,
    0.768657684326172, 0.386671602725983, 0.423865079879761,
    0.701108753681183, 0.347370147705078, 0.00282984972000122,
    0.963719666004181, 0.376665771007538, 0.979808509349823,
    0.322535574436188, 0.671220064163208, 0.0763792991638184,
    0.407272517681122, 0.814533948898315, 0.620485842227936,
0.325440406799316, 0.455562174320221, 0.520989537239075,
0.49634861946106, 0.518668055534363, 0.404020965099335,
0.564167618751526, 0.783891439437866, 0.236178815364838,
0.0904685258865356, 0.509209156036377, 0.623986005783081,
0.980750977993011, 0.647472441196442, 0.024944007396698,
0.420155882835388, 0.715914189815521, 0.609794080257416,
0.545187413692474, 0.227712571620941, 0.261496543884277,
0.412327587604523, 0.384870767593384, 0.627764701843262,
0.506571650505066, 0.703603386878967, 0.829880952835083,
0.20378977060318, 0.106975078582764, 0.558040380477905,
0.0035395622253418, 0.608073711395264, 0.265367805957794,
0.233837068080902, 0.405840754508972, 0.252925038337708,
0.232855200767517, 0.359926342964172, 0.698240101337433,
0.0953984260559082, 0.968190670013428, 0.376338422298431,
0.162436187267303, 0.62673819065094, 0.807673990726471,
0.0350344181060791, 0.187889099121094, 0.798649251461029,
0.75320440530777, 0.133968472480774, 0.013517439365387,
0.993913650512695, 0.231245577335358, 0.14457768201828,
0.200252115726471, 0.766600430011749, 0.192850410938263,
0.499713182449341, 0.701053500175476, 0.289142489433289,
0.172838151454926, 0.090063750743866, 0.686260104179382,
0.436583995819092, 0.166717767715454, 0.95188307762146,
0.0888900756835938, 0.709761261940002, 0.926010966300964,
0.755458235740662, 0.776119112968445, 0.0595329999923706,
0.142618715763092, 0.490822672843933, 0.528886675834656,
0.322580635547638, 0.869046747684479, 0.344156265258789,
0.281339466571808, 0.200292110443115, 0.577095806598663,
0.286913812160492, 0.20842570066452, 0.210157811641693,
0.510974049568176, 0.748934984207153, 0.249893188476562,
0.511870682239532, 0.594643473625183, 0.547506749629974,
0.928706228733063, 0.691949188709259, 0.201952993869781,
0.397345244884491, 0.335498631000519, 0.842258453369141,
0.596347868442535, 0.541809439659119, 0.508456468582153,
0.663657784461975, 0.445010781288147, 0.487370729446411,
0.613987147808075, 0.751830160617828, 0.88357138633728,
0.278889894485474, 0.0838443636894226, 0.060584545135498,
0.224907279014587, 0.479591846466064, 0.765725433826447,
0.94118469953537, 0.61225038766861, 0.856002509593964]
```

List of Figures

List of Tables

Contents of the attached DVD

The attached DVD contains the following:

- This diploma thesis as PDF,
- The BRAIn sourcecode,
- The sourcecode of all the experiments,
- The sourcecode of various tools used for this thesis,
- The complete simulation results.

Bibliography

[AJA^{+}11] Pejman Aminian, Mohamad Javid, Abazar Asghari, Amir Gandomi, and Milad Esmaeili. A robust predictive model for base shear of steel frame structures using a hybrid genetic programming and simulated annealing method. *Neural Computing & Applications*, 20:1321–1332, 2011. 10.1007/s00521-011-0689-0.

[Alp04] Ethem Alpaydin. *Introduction to Machine Learning.* MIT Press, 2004.

[BBG95] Eric B. Baum, Dan Boneh, and Charles Garrett. On genetic algorithms. In *COLT*, pages 230–239, 1995.

[Blo08] Joshua Bloch. *Effective Java.* Addison-Wesley, 2008.

[BSMM01] Bronstein, Semendjajew, Musiol, and Mühlig. *Taschenbuch der Mathematik.* Verlag Harri Deutsch, 2001.

[can] Seer stat fact sheets: Breast. `http://seer.cancer.gov/statfacts/html/breast.html`, accessed 28. Nov. 2011.

[CLD03] An-Sing Chen, Mark T. Leung, and Hazem Daouk. Application of neural networks to an emerging financial market: forecasting and trading the taiwan stock index. *Computers and Operations Research*, 30:901–923, May 2003.

[Coy06] Jerry A. Coyne. Selling darwin. *Nature*, 442:983–984, 2006.

[Cur84] Charles W. Curtis. *Linear Algebra - An Introductory Approach.* Springer, 1984.

[CWY99] John J. Cheh, Randy S. Weinberg, and Ken C. Yook. An Application Of An Artificial Neural Network Investment System To Predict Takeover Targets. *Journal Of Applied Business Research*, 15, 1999.

[Dar] Charles Darwin. *On the origin of species by means of natural selection, or the preservation of favoured races in the struggle for life.*, volume 1859.

[DT03] George B. Dantzig and Mukund N. Thapa. *Linear Programming: Theory and extensions.* Springer, 2003.

[Fit07] Michael Fitzgerald. *Learning Ruby.* O'REILLY, 2007.

[for] Definition of foraging by the free online dictionary. `http://www.thefreedictionary.com/foraging`, accessed 08 Dec. 2011.

[Fow] Martin Fowler. DomainSpecificLanguage. `http://martinfowler.com/bliki/DomainSpecificLanguage.html`, accessed 08. Dec. 2011.

[Gom03] Faustino J. Gomez. Phd thesis: Robust non-linear control through neuroevolution. Technical Report AI-TR-03-303, Department of Computer Sciences, University of Texas at Austin, August 2003.

[GPB$^+$09] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency In Practice.* Add, 2009.

[GPMR11] Antoniya Georgieva, Stephen J. Payne, Mary Moulden, and Christopher W. G. Redman. Artificial neural networks applied to fetal monitoring in labour. *Neural Computing & Applications*, 2011.

[Gre09] T. Gregory. Understanding natural selection: Essential concepts and common misconceptions. *Evolution: Education and Outreach*, 2:156–175, 2009. 10.1007/s12052-009-0128-1.

[Gru09] Stefan Grundhoff. Verkehrszeichenerkennung - Das magische Auge. `http://www.focus.de/auto/ratgeber/sicherheit/assistenzsysteme/verkehrszeichenerkennung-das-magische-auge_aid_346187.html`, accessed 17. Dec. 2011, January 2009.

[gui] google-guice. `http://code.google.com/p/google-guice/`, accessed 08. Dec. 2011.

[Hay08] Simon Haykin. *Neural Networks and Learning Machines.* 2008.

[HCP11] Chien-Jen Huang, Peng-Wen Chen, and Wen-Tsao Pan. Using multi-stage data mining technique to build forecast model for taiwan stocks. *Neural Computing & Applications*, 2011.

[HL11] Ting Huang and Derong Liu. A self-learning scheme for residential energy system control and management. *Neural Computing & Applications*, 2011.

[Hsu11] Chih-Ming Hsu. A hybrid procedure with feature selection for resolving stock/futures price forecasting problems. *Neural Computing & Applications*, 2011.

[JMW$^+$05] Ahmedin Jemal, Taylor Murray, Elizabeth Ward, Alicia Samuels, Ram C. Tiwari, Asma Ghafoor, Eric J. Feuer, and Michael J. Thun. Cancer statistics, 2005. *CA: A Cancer Journal for Clinicians*, 55(1):10–30, 2005.

[jrua] Documentation - JRuby.org. `http://jruby.org/documentation`, accessed 04. Dec 2011.

[jrub] JRuby.org. `http://jruby.org/`, accessed 08. Dec. 2011.

[jun] Junit.org. `http://www.junit.org/`, accessed 08. Dec. 2011.

[KE06] A. Kemper and A. Eickler. *Datenbanksysteme.* Oldenbourg, 2006.

[Kit90] Hiroaki Kitano. Designing Neural Networks Using Genetic Algorithms with Graph Generation System. *Complex Systems Journal*, 4:461–476, 1990.

[LJYB03] Marner L., Nyengaard JR., Tang Y., and Pakkenberg B. Marked loss of myelinated nerve fibers in the human brain with age. *J Comp Neurol*, 2003.

[LlZsM+11] Bing Li, Pei lin Zhang, Shuang shan Mi, Peng yuan Liu, and Dong sheng Liu. Applying the fuzzy lattice neurocomputing (fln) classifier model to gear fault diagnosis. *Neural Computing & Applications*, 2011.

[log] Apache Logging Services Project - Apache log4j. `http://logging.apache.org/log4j/`, accessed 08. Dec. 2011.

[Maa97] Wolfgang Maass. Networks of Spiking Neurons: The Third Generation of Neural Network Models. *Neural Networks*, 10:1659–1671, 1997.

[mav] Apache maven. `http://maven.apache.org/`, accessed 08. Dec. 2011.

[MD89] David J Montana and Lawrence Davis. Training feedforward neural networks using genetic algorithms. *Proceedings of the eleMachine Learningventh international joint conference on artificial Intelligence*, pages 762–767, 1989.

[Mit95] Melanie Mitchell. Genetic algorithms : An overview. *Complexity*, 1:31–39, 1995.

[Mit96] Melanie Mitchell. *An Introduction To Genetic Algorithms.* MIT Press, 1996.

[MMMR96] Anil Menon, Kishan Mehrotra, Chilukuri K. Mohan, and Sanjay Ranka. Characterization of a class of sigmoid functions with applications to neural networks. *Neural Networks*, 6, 1996.

[moc] mockito - simpler & better mocking. `http://code.google.com/p/mockito/`, accessed 08. Dec. 2011.

[Mor98] Hans Moravec. When will computer hardware match the human brain? *Journal of Evolution and Technology*, 1, 1998.

[NK05] Patrick Niemeyer and Jonathan Knudsen. *Learning Java.* O'REILLY, 3rd edition, 2005.

[PdY] Hossein Pazhoumand-dar and Mahdi Yaghoobi. A new approach in road sign recognition based on fast fractal coding. *Neural Computing & Applications*, pages 1–11. 10.1007/s00521-011-0718-z.

[Pom89] Dean A. Pomerleau. ALVINN: an autonomous land vehicle in a neural network. In *Advances in neural information processing systems 1*, pages 305–313. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1989.

[Pra09] Dhanji R. Prasanna. *Dependency Injection.* Manning, 2009.

[pro] `http://code.google.com/p/protobuf/`, accessed 08. Dec. 2011.

[PTO$^+$11] Carlo Pinciroli, Vito Trianni, Rehan O'Grady, Giovanni Pini, Arne Brutschy, Manuele Brambilla, Nithin Mathews, Eliseo Ferrante, Gianni Di Caro, Frederick Ducatelle, Timothy Stirling, Álvaro Gutiérrez, Luca Maria Gambardella, and Marco Dorigo. ARGoS: a modular, multi-engine simulator for heterogeneous swarm robotics. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2011)*, pages 5027–5034. IEEE Computer Society Press, Los Alamitos, CA, September 2011.

[RMV11] Philippe Rétornaz, Stéphane Magnenat, and Florian Vaussard. MarXbot User Manual, Version 1.1. `http://mobots.epfl.ch/data/robots/marxbot-user-manual.pdf`, accessed 28. Nov. 2011, April 2011.

[RN03] S J Russell and P Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2003.

[Ros58] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65, 1958.

[Row99] Henry A. Rowley. *Neural Network-Based Face Detection*. PhD thesis, School of Computer Science, Computer Science Department, CarnegieMellon University, 1999.

[rub] Ruby-doc.org: Documenting the ruby language. `http://www.ruby-doc.org/`, accessed 04. Dec. 2011.

[SBM05] Kenneth O. Stanley, Bobby D. Bryant, and Risto Miikkulainen. Realtime Neuroevolution in the NERO Video Game. *IEEE Transactions on Evolutionary Computation*, pages 653–668, 2005.

[SGP11] R. A. Saeed, A. N. Galybin, and V. Popov. Crack identification in curvilinear beams by using ann and anfis based on natural frequencies and frequency response functions. *Neural Computing & Applications*, 2011.

[SK90] G.M. Shepherd and Koch. Introduction to synaptic circuits. *The Synaptic Organization of the Brain*, pages 3–31, 1990.

[SK08] Bruno Siciliano and Oussama Khatib. *Springer handbook of robotics*. Springer, 2008.

[slf] Simple Logging Facade for Java (SLF4J). `http://www.slf4j.org/`, accessed 08. Dec. 2011.

[SM02] Kenneth O. Stanley and Risto Miikkulainen. Competitive coevolution through evolutionary complexification. *Journal of Artificial Intelligence Research*, 21:63–100, 2002.

[SMDD11] M. R. Senapati, A. K. Mohanty, S. Dash, and P. K. Dash. Local linear wavelet neural network for breast cancer recognition. *Neural Computing & Applications*, 2011.

[SMR11] Babak Sohrabi, Payam Mahmoudian, and Iman Raeesi. A framework for improving e-commerce websites usability using a hybrid genetic algorithm and neural network system. *Neural Computing & Applications*, 2011.

[SRA11] Mansour Sheikhan, Mohsen Rohani, and Saeed Ahmadluei. A neural-based concurrency control algorithm for database systems. *Neural Computing & Applications*, 2011.

[WFK11] Markus Waibel, Dario Floreano, and Laurent Keller. A Quantitative Test of Hamilton's Rule for the Evolution of Altruism. *PLoS Biol*, 9(5):e1000615, 05 2011.

[WKF09] Markus Waibel, Laurent Keller, and Dario Floreano. Genetic Team Composition and Level of Selection in the Evolution of Cooperation. *IEEE Transactions on Evolutionary Computation*, 13(3):648–660, 2009.

[WZ89] Ronald J. Williams and David Zipser. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Computation*, 1:270–280, 1989.

[XWX11] Jie Xiu, Shiyu Wang, and Yan Xiu. Fuzzy adaptive single neuron nn control of brushless dc motor. *Neural Computing & Applications*, 2011.

[YKID11] Hasbi Yaprak, Abdülkadir Karacı, and İlhami Demir. Prediction of the effect of varying cure conditions and w/c ratio on the compressive strength of concrete using artificial neural networks. *Neural Computing & Applications*, 2011.